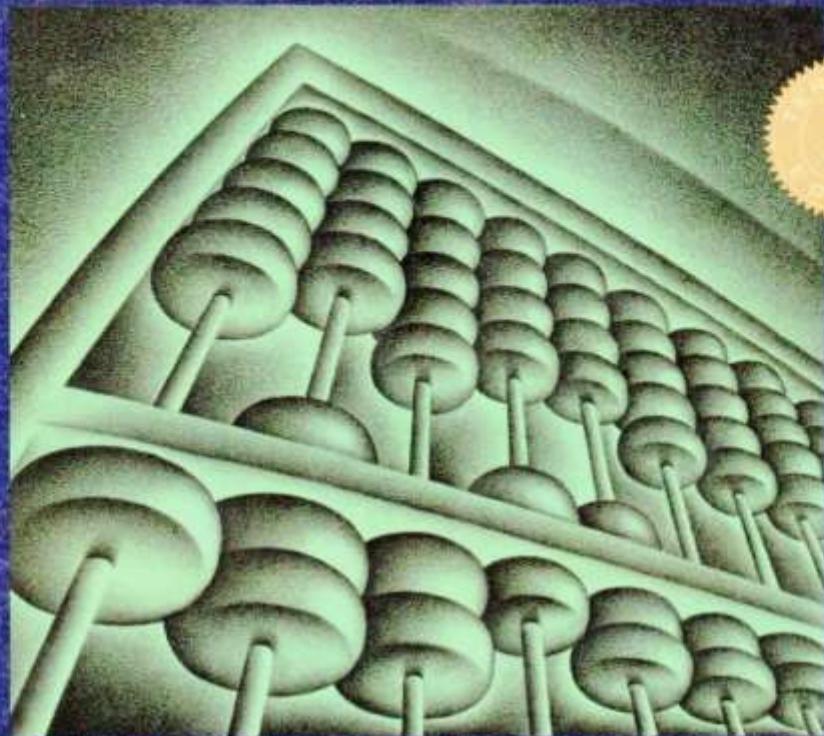




ORGANIZAÇÃO E PROJETO DE COMPUTADORES

A INTERFACE HARDWARE/SOFTWARE



3

DAVID A. PATTERSON
JOHN L. HENNESSY



Material do
Professor
na WEB

[www.campus.com.br/
professores](http://www.campus.com.br/professores)

DAVID A. PATTERSON foi o primeiro de sua família a se formar numa faculdade (1960 A.B UCLA) e gostou tanto que não parou até obter o Ph.D (1976 UCLA). Depois de 4 anos desenvolvendo um computador do tamanho de um wafer na Hughes Aircraft, juntou-se à U.C. Berkeley em 1977. Passou o ano de 1979 na DEC, trabalhando no minicomputador VAX. Ele e seus colegas mais tarde desenvolveram o Reduced Instruction Set Computer (RISC). Juntando esforços com os projetos 801 da IBM e o MIPS de Stanford, RISC tornou-se bastante divulgado. Em 1984, a Sun Microsystems o recrutou para iniciar a arquitetura MIPS. Em 1987, David Patterson e seus colegas tentaram construir sistemas de armazenamento confiáveis a partir dos novos discos para PC. Isso levou ao popular Redundant Array of Inexpensive Disks (RAID). Passou o ano de 1989 trabalhando no supercomputador CM-5. Mais tarde tentaram construir um supercomputador usando computadores desktop padrão e switches. O projeto resultante, Network of Workstations (NOW), levou à tecnologia de cluster utilizada por muitos iniciantes. Ele agora está trabalhando no projeto Recovery Oriented Computing (ROC). No passado, trabalhou como Presidente da Divisão CS de Berkeley. Atualmente está trabalhando no comitê consultivo de TI para o Presidente dos Estados Unidos e foi eleito Presidente da ACM.

Tudo isso resultou em 150 artigos, 5 livros e diversas horas, algumas compartilhadas com seus amigos, incluindo a eleição para a National Academy of Engineering e vários prêmios da Universidade da Califórnia: Outstanding Alumnus Award (UCLA Computer Science Department), McHntyre Award for Excellence in Teaching (Berkeley Computer Science) e o Distinguished Teaching Award (Berkeley). Pela ACM, onde foi colaborador, recebeu o SIGMOD Test of Time Award e o Karlstrom Outstanding Educator Award. Também é colaborador do IEEE, onde recebeu o Johnson Information Storage Award, o Undergraduate Teaching Award, a Mulligan Education Medal e a von Neumann Medal.

ORGANIZAÇÃO E PROJETO DE COMPUTADORES

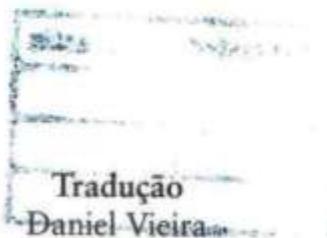




Preencha a **ficha de cadastro** no final deste livro
e receba gratuitamente informações
sobre os lançamentos e as promoções da
Editora Campus/Elsevier.

Consulte também nosso catálogo
completo e últimos lançamentos em
www.campus.com.br

ORGANIZAÇÃO E PROJETO DE COMPUTADORES



Tradução
Daniel Vieira

*Presidente da Multinet Informática
Programador e tradutor especializado
em Informática*

Revisão Técnica
Mario João Jr.

*Professor do NCE/UFRJ
e Doutorando do IM-NCE/UFRJ*

DAVID A. PATTERSON
JOHN L. HENNESSY

4ª Tiragem



Do original;
Computer Organization Design
Tradução autorizada do idioma inglês da edição publicada por Morgan Kaufmann Publishers
Copyright © 2005 by Elsevier Inc.

© 2005, Elsevier Editora Ltda.

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998.
Nenhuma parte deste livro, sem autorização prévia por escrito da editora,
poderá ser reproduzida ou transmitida sejam quais forem os meios empregados:
eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

Editoração Eletrônica: Estúdio Castellani

Revisão Gráfica: Marco Antonio Correa

Projeto Gráfico

Elsevier Editora Ltda.

A Qualidade da Informação.

Rua Sete de Setembro, 111 – 16º andar

20050-006 Rio de Janeiro RJ Brasil

Telefone: (21) 3970-9300 FAX: (21) 2507-1991

E-mail: info@elsevier.com.br

Escritório São Paulo:

Rua Quintana, 753/8º andar

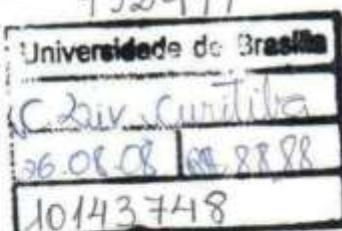
04569-011 Brooklin São Paulo SP

Tel.: (11) 5105-8555

ISBN 13: 978-85-352-1521-2

ISBN 10: 85-352-1521-2

Edição original: ISBN 1-55860-604-1



004.2

P317C

=690

3.ed

Nota: Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses, solicitamos a comunicação à nossa Central de Atendimento, para que possamos esclarecer ou encaminhar a questão.

Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso desta publicação.

Central de atendimento

Tel: 0800-265340

Rua Sete de Setembro, 111, 16º andar – Centro – Rio de Janeiro

e-mail: info@elsevier.com.br

site: www.campus.com.br

CIP-Brasil. Catalogação-na-fonte.
Sindicato Nacional dos Editores de Livros, RJ

P344o

Patterson, David A.

Organização e projeto de computadores : a interface hardware/software / David A. Patterson e John L. Hennessy ; tradução de Daniel Vieira. – Rio de Janeiro : Elsevier, 2005 – 4ª Reimpressão.

Il.

Tradução de: Computer organization and design, 3rd ed

Apêndice

Inclui bibliografia

ISBN 85-352-1521-2

1. Organização de computador. 2. Engenharia de computador. 3. Interfaces (Computadores). I. Hennessy, John L. II. Título.

05-0263.

CDD – 004.22

CDU – 004.2

Guia de instrução rápida

CONJUNTO DE INSTRUÇÕES BÁSICO

NOME	MNEMÔNICO	FORMATO	OPERAÇÃO (EM VERILOG)	OPCODE/FUNCT (HEXA)
Add	add	R	$R[rd] = R[rs] + R[rt]$	(1) 0/20 _{hexa}
Add Immediate	addi	I	$R[rt] = R[rs] + ImSinExt$	(1)(2) 8 _{hexa}
Add Imm. Unsigned	addiu	I	$R[rt] = R[rs] + ImSinExt$	(2) 9 _{hexa}
Add Unsigned	addu	R	$R[rd] = R[rs] + R[rt]$	0/21 _{hexa}
And	and	R	$R[rd] = R[rs] \& R[rt]$	0/24 _{hexa}
And Immediate	andi	I	$R[rt] = R[rs] \& ImZeroExt$	(3) C _{hexa}
Branch On Equal	beq	I	$\text{If}(R[rs]==R[rt]) PC=PC+4+\text{EndBranch}$	(4) 4 _{hexa}
Branch On Not Equal	bne	I	$\text{If}(R[rs]\neq R[rt]) PC=PC+4+\text{EndBranch}$	(4) 5 _{hexa}
Jump	j	J	$PC=\text{EndJump}$	(5) 2 _{hexa}
Jump And Link	jal	J	$R[31]=PC+4; PC=\text{EndJump}$	(5) 3 _{hexa}
Jump Register	jr	R	$PC=R[rs]$	0/08 _{hexa}
Load Byte Unsigned	lbu	I	$R[rt]=(24'b0,M[R[rs]]+ImSinExt)[7:0]$	(2) 0/24 _{hexa}
Load Halfword Unsigned	lhu	I	$R[rt]=(16'b0,M[R[rs]]+ImSinExt)[15:0]$	(2) 0/25 _{hexa}
Load Upper Imm.	lui	I	$R[rt]=(Imm, 16'b0)$	fh _{hexa}
Load Word	lw	I	$R[rt]=M[R[rs]]+ImSinExt$	(2) 0/23 _{hexa}
Nor	nor	R	$R[rd] = \neg(R[rs])\ R[rt]$	0/27 _{hexa}
Or	or	R	$R[rd] = R[rs] R[rt]$	0/25 _{hexa}
Or Immediate	ori	I	$R[rt]=R[rs] ImZeroExt$	(3) d _{hexa}
Set Less Than	slt	R	$R[rd] = (R[rs] < R[rt]) ? 1 : 0$	0/28 _{hexa}
Set Less Than Imm.	slli	I	$R[rt] = (R[rs] < ImSinExt) ? 1 : 0$	(2) a _{hexa}
Set Less Than Imm. Unsigned	situ	I	$R[rt] = (R[rs] < ImSinExt) ? 1 : 0$	(2)(6) b _{hexa}
Set Less Than Unsigned	situ	R	$R[rd] = (R[rs] < R[rt]) ? 1 : 0$	(6) 0/2b _{hexa}
Shift Left Logical	sll	R	$R[rd] = R[rs] << shamt$	0/00 _{hexa}
Shift Right Logical	srl	R	$R[rd] = R[rs] >> shamt$	0/02 _{hexa}
Store Byte	sb	I	$M[R[rs]]+ImSinExt)[7:0] = R[rt][7:0](2)$	28 _{hexa}
Store Halfword	sh	I	$M[R[rs]]+ImSinExt)[15:0] = R[rt][15:0]$	(2) 29 _{hexa}
Store Word	sw	I	$M[R[rs]]+ImSinExt] = R[rt]$	(2) 2b _{hexa}
Subtract	sub	R	$R[rd] = R[rs] - R[rt]$	(1) 0/22 _{hexa}
Subtract Unsigned	subu	R	$R[rd] = R[rs] - R[rt]$	0/23 _{hexa}

- (1) Pode causar exceção de overflow
(2) ImSinExt = {16(immediato[15]), imediato}
(3) ImZeroExt = {16(1b'0)}, imediato
(4) EndBranch = {14(immediato[15]), imediato, 2'b0}
(5) JumpAddr = {PC[31:28], address, 2'b0}
(6) Operandos considerados números sem sinal (vs. compil. 2)

FORMATOS DE INSTRUÇÃO BÁSICOS

R	opcode	rs	rt	Rd	shamt	funct	
	31	26 25	21 20	16 15	11 10	6 5	0
I	opcode	rs	rt		immediato		
	31	26 25	21 20	16 15		0	
J	opcode	Endereço					
	31	26 25				0	

CONJUNTO DE INSTRUÇÕES BÁSICAS ARITMÉTICAS

NOME	MNEMÔNICO	FORMATO	OPERAÇÃO	OPCODE/FMT/FT/FUNCT (HEXA)
Branch On FP True	beq	R	$\text{if}(FPcond)PC=PC+4+\text{EndBranch}$ (4)	11/8/1-
Branch On FP False	bcf	R	$\text{if}(\neg FPcond)PC=PC+4+\text{EndBranch}$ (4)	11/8/0-
Divide	div	R	$Lo=R[rs]/R[rt]; Hi=R[rs]\%R[rt]$	0/-/-/1a
Divide Unsigned	divu	R	$Lo=R[rs]/R[rt]; Hi=R[rs]\%R[rt]$ (6)	0/-/-/1b
FP Add Single	add.s	FR	$Ffd = F[fs] + F[ft]$	11/10/-/0
FP Add Double	add.d	FR	$[Ffd, F[fd+1]] = [F[fs], F[fs+1]] + [F[rt], F[rt+1]]$	11/11/-/0
FP Compare Single	c.x,s*	FR	$\text{FPcond} = (F[fs] op F[ft]) ? 1 : 0$	11/10/-/y
FP Compare Double	c.x,d*	FR	$\text{FPcond} = ([F[fs], F[fs+1]] op [F[rt], F[rt+1]]) ? 1 : 0$	11/11/-/y
FP Divide Single	div.s	FR	$F[fd] = F[fs]/F[ft]$	11/10/-/3
FP Divide Double	div.d	FR	$[F[fd], F[fd+1]] = [F[fs], F[fs+1]] / [F[rt], F[rt+1]]$	11/11/-/3
FP Multiply Single	mul.s	FR	$F[fd] = F[fs] * F[ft]$	11/10/-/2
FP Multiply Double	mul.d	FR	$[F[fd], F[fd+1]] = [F[fs], F[fs+1]] * [F[rt], F[rt+1]]$	11/11/-/2
FP Subtract Single	sub.s	FR	$F[fd] = F[fs]-F[ft]$	11/10/-/1
FP Subtract Double	sub.d	FR	$[F[fd], F[fd+1]] = [F[fs], F[fs+1]] - [F[rt], F[rt+1]]$	11/11/-/1
Load FP Single	lwc1	I	$R[rt]=M[R[rs]]+ImSinExt$	31/-/-
Load FP Double	ldc1	I	$R[rt]=M[R[rs]]+ImSinExt+4$	35/-/-
Move From HI	mfhi	R	$R[rd] = HI$	0/-/-/10
Move From LO	mflo	R	$R[rd] = LO$	0/-/-/12
Move From Control	mfco	R	$R[rd] = CR[rs]$	16/0/-/0
Multiply	mult	R	$(HI, LO) = R[rs] * R[rt]$	0/-/-/18
Multiply Unsigned	multu	R	$(HI, LO) = R[rs] * R[rt]$ (6)	0/-/-/19
Store FP Single	swc1	I	$M[R[rs]]+ImSinExt] = R[rt]$ (2)	39/-/-
Store FP Double	sdc1	I	$M[R[rs]]+ImSinExt+4] = R[rt]$ (2)	3d/-/-

FORMATOS DAS INSTRUÇÕES DE PONTO FLUTUANTE

FR	opcode	fmt	r	fs	fd	funct	
	31	26 25	21 20	16 15	11 10	6 5	0
FI	opcode	fmt	r		immediato		
	31	26 25	21 20	16 15			0

CONJUNTO DE PSEUDO-INSTRUÇÕES

NOME	MNEMÔNICO	OPERAÇÃO
Branch Less Than	blt	$\text{If}(R[rs]<R[rt]) PC = \text{Label}$
Branch Greater Than	bgt	$\text{If}(R[rs]>R[rt]) PC = \text{Label}$
Branch Less Than or Equal	ble	$\text{If}(R[rs]\leq R[rt]) PC = \text{Label}$
Branch Greater Than or Equal	bge	$\text{If}(R[rs]\geq R[rt]) PC = \text{Label}$
Load Immediate	li	$R[rd] = \text{immediato}$
Move	move	$R[rd] = R[rs]$

NOME DO REGISTRADOR, NÚMERO, USO E CONVENÇÃO DE CHAMADA

NOME	NÚMERO	USO	PRESERVADO ENTRE CHAMADAS?
\$zero	0	O valor constante 0	N.A.
\$at	1	Temporário do montador	Não
\$v0-\$v1	2-3	Valores para resultados de função e avaliação de expressão	Não
\$a0-\$a3	4-7	Argumentos	Não
\$t0-\$t7	8-15	Temporários	Não
\$s0-\$s7	16-23	Temporários salvos	Sim
\$t8-\$t9	24-25	Temporários	Não
\$k0-\$k1	26-27	Reservado para kernel do SO	Não
\$gp	28	Ponteiro global	Sim
\$sp	29	Stack Pointer	Sim
\$fp	30	Frame Pointer	Sim
\$ra	31	Endereço de retorno	Sim

OPCODES, CONVERSÃO DE BASE, SÍMBOLOS ASCII

OPCODE	(1) MIPS MIPS (31:26)	(2) MIPS FUNCT (5:0)	BINÁRIO	DECIMAL	HEXA	ASCII	DECIMAL	HEXA	ASCII
(1)	s11	add.f	00 0000 0	0	NUL	64	40	0	
		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	slv	sqrt.f	00 0100 4	4	EOT	68	44	D	
bne		abs.f	00 0101 5	5	ENQ	69	45	E	
blez	sriv	mov.f	00 0110 6	6	ACK	70	46	F	
bgtz	srav	neg.f	00 0111 7	7	BEL	71	47	G	
addi	jr		00 1000 8	8	BS	72	48	H	
addiu	jalr		00 1001 9	9	HT	73	49	I	
alti	mvov		00 1010 10	a	LF	74	4a	J	
stiu	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.w.f	00 1110 14	e	SO	78	4e	N	
lui	sync	floor.w.f	00 1111 15	f	SI	79	4f	O	
	mfhi		01 0000 16	10	DLE	80	50	P	
(2)	mtfi		01 0001 17	11	DC1	81	51	Q	
mflo	movzf.f		01 0010 18	12	DC2	82	52	R	
mtlo	movnf.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	
	multu		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	1c	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	Espaco	96	60	'	
lh	addu	cvt.d.f	10 0001 33	21	!	97	61	a	
lw	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	
ihu	or		10 0101 37	25	%	101	65	e	
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	
swl	Sit		10 1010 42	2a	*	106	6a	j	
sw	Situ		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	
t1	tge	c.f.f	11 0000 48	30	0	112	70	P	
hwc1	tgeu	c.un.f	11 0001 49	31	1	113	71	q	
hwc2	tl	c.eq.f	11 0010 50	32	2	114	72	r	
pref	ttu	c.ues.f	11 0011 51	33	3	115	73	s	
	teq	c.lt.f	11 0100 52	34	4	116	74	t	
ldc1		c.ltif.f	11 0101 53	35	5	117	75	u	
ldc2	tne	c.le.f	11 0110 54	36	6	118	76	v	
		c.ule.f	11 0111 55	37	7	119	77	w	
sc		c.sf.f	11 1000 56	38	8	120	78	X	
swc1		c.ngle.f	11 1001 57	39	9	121	79	y	
swc2		c.seq.f	11 1010 58	3a	:	122	7a	z	
		c.nglf.f	11 1011 59	3b	:	123	7b	{	
		c.lt.f	11 1100 60	3c	<	124	7c		
sdc1		c.ngf.f	11 1101 61	3d	=	125	7d]	
sdc2		c.le.f	11 1110 62	3e	>	126	7e	-	
		c.ngt.f	11 1111 63	3f	?	127	7f	DEL	

(1) opcode[31:26]==0

(2) opcode[31:26] == 17₁₀ dec (11_{hex}); if fmt(25:21)==16₁₀ (10_{hex}) f==s (single); if fmt(25:21)==17₁₀ dec (11_{hex}) f=d (double)

PADRÃO IEEE 754 DE PONTO FLUTUANTE

$(-1)^{\text{Expoente}} \times (1 + \text{Fração}) \times 2^{\text{Expoente Bias}}$
onde Bias Precisão simples = 127.
Bias Precisão dupla = 1023.

Símbolos IEEE 754

Exponente	Fração	Objeto
0	0	±0
0	≠0	± Denorm
1 a MAX - 1	qualquer coisa	± Núm. Pt. Flut.
MAX	0	± ∞
MAX	≠0	NaN

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

Formatos IEEE de precisão simples e dupla:

S	Exponente	Fração	0
1	31 30	23 22	0

ALOCAÇÃO DE MEMÓRIA



ALINHAMENTO DE DADOS

Word dupla	Word	Word					
Half word	Half word	Half word					
Byte Byte	Byte Byte	Byte Byte					
0	1	2	3	4	5	6	7

Valor de três bits menos significativos do endereço de byte (Big Endian)

REGISTRADORES DE CONTROLE DE EXCEÇÃO: CAUSA E STATUS

B	D	Máscara de interrupção	Interrupções pendentes
31	15	8	6
		Código de exceção	U M E L I E

BD = Branch Delay, UM = User Mode, EL = Exception Level, IE = Interrupt Enable

CÓDIGOS DE EXCEÇÃO

NÚMERO	NOME	CAUSA DA EXCEÇÃO	NÚMERO	NOME	CAUSA DA EXCEÇÃO
0	Int	Interface (hardware)	9	Bp	Exceção de breakpoint
4	AdEL	Address Error Exception (load ou busca de instrução)	10	Rl	Exceção de instrução reservada
5	AdES	Address Error Exception (store)	11	CpU	Co-processador não implementado
6	IBE	Erro de barramento na busca de instrução	12	Ov	Exceção de overflow aritmético
7	DBE	Erro de barramento no Load ou Store	13	Tr	Trap
8	Sys	Exceção de Syscall	15	FPE	Exceção de ponto flutuante

PREFIXOS DE TAMANHO (10^X para Disco, Comunicação; 2^X para Memória)

TAMANHO	PREFIXO	TAMANHO	PREFIXO	TAMANHO	PREFIXO
10^{32} , 2^{30}	Kilo-	10^{36} , 2^{30}	Peta-	10^3	milli-
10^{24} , 2^{20}	Mega-	10^{38} , 2^{30}	Exa-	10^6	micro-
10^{12} , 2^{10}	Giga-	10^{40} , 2^{30}	Zetta-	10^9	atto-
10^{-12} , 2^{-10}	Tera-	10^{42} , 2^{30}	Yotta-	10^{12}	zepto-

O símbolo para cada prefixo é apenas sua primeira letra, exceto que é usado para micro.

Sumário

CAPÍTULOS

1 Abstrações e Tecnologias Computacionais 2

- 1.1 Introdução 3
- 1.2 Por baixo do seu programa 9
- 1.3 Sob as tampas 11
- 1.4 Vida real: fabricando chips de Pentium 4 22
- 1.5 Faláncias e armadilhas 25
- 1.6 Comentários finais 27
- 1.7 Perspectiva histórica e leitura adicional 28
- 1.8 Exercícios 28

COMPUTADORES NO MUNDO REAL

Tecnologia da informação para os 4 bilhões sem TI 32

2 Instruções: A Linguagem de Máquina 34

- 2.1 Introdução 36
- 2.2 Operações do hardware do computador 37
- 2.3 Operandos do hardware do computador 39
- 2.4 Representando instruções no computador 44
- 2.5 Operações lógicas 50
- 2.6 Instruções para tomada de decisões 52
- 2.7 Suporte a procedimentos no hardware do computador 58
- 2.8 Comunicando-se com as pessoas 66
- 2.9 Endereçamento no MIPS para operandos imediatos e endereços de 32 bits 70
- 2.10 Traduzindo e iniciando um programa 78
- 2.11 Como os compiladores otimizam 85
- 2.12 Como os compiladores funcionam: uma introdução 89
- 2.13 Um exemplo de ordenação em C para juntar tudo isso 89
- 2.14 Implementando uma linguagem orientada a objetos 96
- 2.15 Arrays *versus* ponteiros 96
- 2.16 Vida real: instruções do IA-32 99
- 2.17 Faláncias e armadilhas 106
- 2.18 Comentários finais 107
- 2.19 Perspectiva histórica e leitura adicional 109
- 2.20 Exercícios 110

COMPUTADORES NO MUNDO REAL

Ajudando a salvar nosso meio ambiente com dados 116

3 Aritmética Computacional 118

- 3.1 Introdução 120
- 3.2 Números com sinal e sem sinal 120
- 3.3 Adição e subtração 128
- 3.4 Multiplicação 132
- 3.5 Divisão 137
- 3.6 Ponto flutuante 142
- 3.7 Vida real: ponto flutuante no IA-32 164
- 3.8 Falácia e armadilhas 167
- 3.9 Comentários finais 170
- 3.10 Perspectiva histórica e leitura adicional 173
- 3.11 Exercícios 174

C O M P U T A D O R E S N O M U N D O R E A L**Reconstruindo o mundo antigo 178****4 Avaliando e Compreendendo o Desempenho 180**

- 4.1 Introdução 182
- 4.2 Desempenho da CPU e seus fatores 187
- 4.3 Avaliando desempenho 192
- 4.4 Vida real: dois benchmarks SPEC e o desempenho dos recentes processadores Intel 196
- 4.5 Falácia e armadilhas 201
- 4.6 Comentários finais 204
- 4.7 Perspectiva histórica e leitura adicional 205
- 4.8 Exercícios 205

C O M P U T A D O R E S N O M U N D O R E A L**Movendo pessoas com mais rapidez e segurança 210****5 O Processador: Caminho de Dados e Controle 212**

- 5.1 Introdução 214
- 5.2 Convenções lógicas de projeto 217
- 5.3 Construindo um caminho de dados 220
- 5.4 Um esquema de implementação simples 225
- 5.5 Uma implementação multiciclos 239
- 5.6 Exceções 257
- 5.7 Microprogramação: simplificando o projeto de controle 261
- 5.8 Uma introdução ao projeto digital usando uma linguagem de projeto de hardware 261
- 5.9 Vida real: a organização das recentes implementações Pentium 261
- 5.10 Falácia e armadilhas 264
- 5.11 Comentários finais 265
- 5.12 Perspectiva histórica e leitura adicional 266
- 5.13 Exercícios 266

C O M P U T A D O R E S N O M U N D O R E A L**Capacitando o deficiente 274**

6 Melhorando o Desempenho com Pipelining 276

- 6.1 Visão geral de pipelining 278
- 6.2 Um caminho de dados usando pipeline 289
- 6.3 Controle de um pipeline 300
- 6.4 Hazards de dados e forwarding 303
- 6.5 Hazards de dados e stalls 311
- 6.6 Hazards de desvio 313
- 6.7 Usando uma linguagem de descrição de hardware para descrever e modelar um pipeline 322
- 6.8 Exceções 322
- 6.9 Pipelining avançado: extraíndo mais desempenho 326
- 6.10 Vida real: o pipeline do Pentium 4 338
- 6.11 Falácia e armadilhas 340
- 6.12 Comentários finais 341
- 6.13 Perspectiva histórica e leitura adicional 343
- 6.14 Exercícios 343

COMPUTADORES NO MUNDO REAL**Comunicação em massa sem gatekeepers 350****7 Grande e Rápida: Explorando a Hierarquia de Memória 352**

- 7.1 Introdução 354
- 7.2 Princípios básicos de cache 358
- 7.3 Medindo e melhorando o desempenho da cache 371
- 7.4 Memória virtual 385
- 7.5 Uma estrutura comum para hierarquias de memória 406
- 7.6 Vida real: as hierarquias de memória do Pentium P4 e do AMD Opteron 412
- 7.7 Falácia e armadilhas 415
- 7.8 Comentários finais 417
- 7.9 Perspectiva histórica e leitura adicional 419
- 7.10 Exercícios 419

COMPUTADORES NO MUNDO REAL**Salvando os tesouros artísticos do mundo 424****8 Armazenamento, Redes e Outros Periféricos 426**

- 8.1 Introdução 428
- 8.2 Armazenamento em disco e confiabilidade 430
- 8.3 Redes 439
- 8.4 Barramentos e outras conexões entre processadores, memória e dispositivos de E/S 439
- 8.5 Interface dos dispositivos de E/S com processador, memória e sistema operacional 445
- 8.6 Medidas de desempenho de E/S: exemplos de sistemas de disco e de arquivos 452
- 8.7 Projetando um sistema de E/S 455
- 8.8 Vida real: uma câmera digital 457

- 8.9 Falácia e armadilhas 460
- 8.10 Comentários finais 463
- 8.11 Perspectiva histórica e leitura adicional 463
- 8.12 Exercícios 464

C O M P U T A D O R E S N O M U N D O R E A L
Salvando vidas com diagnósticos melhores 472

■ **Índice 474**

(9) Multiprocessadores e Clusters 9-2

- 9.1 Introdução 9-4
- 9.2 Programando multiprocessadores 9-7
- 9.3 Multiprocessadores conectados por um único barramento 9-9
- 9.4 Multiprocessadores conectados por uma rede 9-17
- 9.5 Clusters 9-21
- 9.6 Topologias de rede 9-22
- 9.7 Multiprocessadores no interior de um chip e multithreading 9-25
- 9.8 Vida real: o cluster de PCs do Google 9-28
- 9.9 Falácia e armadilha 9-32
- 9.10 Comentários finais 9-34
- 9.11 Perspectiva histórica e leitura adicional 9-38
- 9.12 Exercícios 9-44

APÊNDICES

(A) Montadores, Link-editores e o Simulador SPIM A-1

- A.1 Introdução A-2
- A.2 Montadores A-7
- A.3 Link-editores A-13
- A.4 Carga A-14
- A.5 Uso da memória A-14
- A.6 Convenção para chamadas de procedimento A-16
- A.7 Exceções e interrupções A-24
- A.8 Entrada e saída A-28
- A.9 SPIM A-29
- A.10 Assembly do MIPS R2000 A-33
- A.11 Comentários finais A-60
- A.12 Exercícios A-61

(B) Fundamentos do Projeto Lógico B-1

- B.1 Introdução B-2
- B.2 Portas, tabelas verdade e equações lógicas B-3
- B.3 Lógica combinacional B-6
- B.4 Usando uma linguagem de descrição de hardware B-15

B.5	Construindo uma unidade lógica e aritmética	B-20
B.6	Adição mais rápida: Carry Lookahead	B-29
B.7	Clocks	B-37
B.8	Elementos de memória: flip-flops, latches e registradores	B-39
B.9	Elementos de memória: SRAMs e DRAMs	B-45
B.10	Máquinas de estados finitos	B-52
B.11	Metodologias de temporização	B-56
B.12	Dispositivos programáveis em campo	B-60
B.13	Comentários finais	B-61
B.14	Exercícios	B-62

(C)**Mapeando o Controle no Hardware C-1**

C.1	Introdução	C-2
C.2	Implementando unidades de controle combinacionais	C-3
C.3	Implementando controle com máquinas de estados finitos	C-6
C.4	Implementando a função de próximo estado com um seqüenciador	C-16
C.5	Traduzindo um microprograma para hardware	C-20
C.6	Comentários finais	C-24
C.7	Exercícios	C-24

(D)**Uma Visão Geral das Arquiteturas RISCs para Computadores Desktop, Servidores e Embutidos D-1**

D.1	Introdução	D-2
D.2	Modos de endereçamento e formatos de instrução	D-4
D.3	Instruções: o subconjunto do núcleo MIPS	D-7
D.4	Instruções: extensões para multimídia dos RISCs desktop/servidores	D-13
D.5	Instruções: extensões para processamento de sinais digitais dos RISCs embutidos	D-15
D.6	Instruções: extensões comuns ao núcleo MIPS	D-15
D.7	Instruções específicas do MIPS64	D-19
D.8	Instruções específicas do Alpha	D-21
D.9	Instruções específicas do SPARC v.9	D-23
D.10	Instruções específicas do PowerPC	D-25
D.11	Instruções específicas do PA-RISC 2.0	D-26
D.12	Instruções específicas do ARM	D-29
D.13	Instruções específicas do Thumb	D-30
D.14	Instruções específicas do SuperH	D-30
D.15	Instruções específicas do M32R	D-31
D.16	Instruções específicas do MIPS16	D-32
D.17	Comentários finais	D-34
D.18	Agradecimentos	D-35
D.19	Referências	D-36

■ Glossário G-1

Prefácio

O que podemos experimentar de mais belo é o mistério. Ele é a fonte de toda a arte e ciência verdadeira.

Albert Einstein, *Como vejo o mundo*, 1930

Sobre este livro

Acreditamos que o aprendizado na Ciência da Computação e na Engenharia deve refletir o estado atual da área, além de introduzir os princípios que estão moldando a computação. Também achamos que os leitores em cada especialidade da computação precisam apreciar os paradigmas organizacionais que determinam as capacidades, o desempenho e, por fim, o sucesso dos sistemas computacionais.

A tecnologia computacional moderna exige que os profissionais de cada especialidade da computação entendam tanto o hardware quanto o software. A interação entre hardware e software em diversos níveis também oferece uma estrutura para se entender os fundamentos da computação. Não importa se seu interesse principal é hardware ou software, Ciência da Computação ou Engenharia Elétrica, as idéias centrais na organização e projeto de computadores são as mesmas. Assim, nossa ênfase neste livro é mostrar o relacionamento entre hardware e software e apresentar os conceitos que são a base para os computadores atuais.

Este livro é útil para aqueles com pouca experiência em linguagem assembly ou projeto lógico, que precisam entender a organização básica do computador, e também para leitores com base em linguagem assembly e/ou projeto lógico, que queiram aprender a projetar um computador ou entender como um sistema funciona e por que ele funciona de determinada forma.

Sobre o outro livro

Alguns leitores podem estar familiarizados com *Arquitetura de Computadores: Uma abordagem quantitativa*, conhecido popularmente como Hennessy e Patterson. (Este livro, por sua vez, é chamado Patterson e Hennessy.) Nossa motivação ao escrever aquele livro foi descrever os princípios da arquitetura de computadores usando fundamentos sólidos de engenharia e compromissos quantitativos de custo/benefício. Usamos uma técnica que combinava exemplos e medições, baseada em sistemas comerciais, para criar experiências de projeto realísticas. Nosso objetivo foi demonstrar que arquitetura de computadores poderia ser aprendida por meio de metodologias quantitativas, em vez de por uma técnica descritiva. Ele é voltado para profissionais de computação sérios, que desejam um conhecimento detalhado dos computadores.

A maioria dos leitores deste livro não planeja se tornar arquitetos de computador. Contudo, o desempenho dos sistemas de software futuros será drasticamente afetado pela forma como os projetistas entendem as técnicas de hardware básicas em funcionamento em um sistema. Assim, aqueles que escrevem compiladores, projetistas de sistema operacional, programadores de banco de dados e a maioria dos outros engenheiros de software precisam de um fundamento sólido sobre os princípios apresentados neste livro. De modo semelhante, os projetistas de hardware precisam entender claramente os efeitos de seu trabalho sobre as aplicações de software.

Assim, sabíamos que este livro tinha de ser muito mais do que um subconjunto do material contido em *Arquitetura de Computadores*, e o material foi bastante revisado para corresponder a esse público-alvo diferente. Ficamos tão satisfeitos com o resultado, que as edições seguintes do *Arquitetura de Computadores* foram revisadas para remover a maior parte do material introdutório; logo, há muito menos repetição hoje do que nas primeiras edições dos dois livros.

Mudanças para a terceira edição

Tivemos seis objetivos principais para esta terceira edição de *Organização e Projeto de Computadores*: fazer com que o livro funcione da mesma forma para leitores com foco no software e com foco no hardware; melhorar a pedagogia em geral; melhorar a compreensão a respeito do desempenho dos programas; atualizar o conteúdo técnico para refletir mudanças ocorridas na área desde a publicação da segunda edição em 1998; unir mais as idéias do livro com o mundo real *fora* da área de computação; e reduzir o tamanho do livro.

Primeiro, a tabela a seguir mostra os caminhos de hardware e software no decorrer do material. Os Capítulos 1, 4 e 7 aparecem nos dois caminhos, não importa qual seja a experiência ou o foco. Os Capítulos 2 e 3 provavelmente repassam o material para os que possuem formação em hardware, mas são uma leitura essencial para os que têm foco no software, especialmente os leitores interessados em aprender mais sobre compiladores e linguagens de programação orientadas a objeto. As primeiras seções dos Capítulos 5 e 6 oferecem introduções para os que possuem foco no software. Entretanto, aqueles que têm um foco no hardware poderão achar que esses capítulos apresentam material básico. Dependendo da sua formação, pode ser interessante ler primeiro o Apêndice B, sobre projeto lógico, e as seções sobre microprogramação e como usar as linguagens de descrição do hardware para especificar o controle. O Capítulo 8, sobre entrada/saída, é a chave para os leitores com foco no software e deve ser lido, se houver tempo, pelos demais leitores. O último capítulo, sobre multiprocessadores e clusters, novamente é uma questão de tempo para o leitor. Até mesmo as seções sobre história mostram esse foco balanceado; elas incluem histórias curtas de linguagens de programação, compiladores, software numérico, sistemas operacionais, protocolos de rede e bancos de dados.

O próximo objetivo foi melhorar a exposição das idéias no livro, com base nas dificuldades mencionadas pelos leitores da segunda edição. Acrescentamos cinco novos elementos ao livro. Para o livro funcionar melhor como uma referência, colocamos definições dos novos termos nas margens, em sua primeira ocorrência. Esperamos que isso ajude os leitores a encontrar as seções quando quiserem consultar o material já lido. Outra mudança foi a inserção das seções "Verifique você mesmo", acrescentadas para ajudar os leitores a verificar sua compreensão do material na primeira vez em que passa por ele. Uma terceira mudança é que acrescentamos exercícios extras na seção "Aprofundando o aprendizado". Quarto, acrescentamos as respostas das seções "Verifique você mesmo" e dos exercícios de "Aprofundando o aprendizado", para ajudar os leitores a verificar se entenderam o material, comparando suas respostas com as do livro.

Em terceiro lugar, os computadores são tão complexos hoje que entender o desempenho de um programa envolve entender muito sobre os princípios básicos e a organização de determinado computador. Nossa objetivo é que os leitores deste livro sejam capazes de entender o desempenho de seus programas e como melhorá-lo. Para ajudar nesse objetivo, acrescentamos um novo elemento ao livro, chamado "Entendendo o desempenho dos programas", em vários capítulos. Essas seções, em geral, oferecem exemplos concretos de como as idéias apresentadas no capítulo afetam o desempenho dos programas reais.

Em quarto lugar, no intervalo desde a segunda edição deste livro, a Lei de Moore seguiu em frente, de modo que agora temos processadores com 200 milhões de transistores, chips de DRAM com um bilhão de transistores e velocidades de clock de vários gigahertz. Os exemplo da "Vida Real" foram atualizados para descrever esses chips. Esta edição também inclui AMD64/IA-32e, a versão com 64 bits de endereço da duradoura arquitetura 80x86, que parece ser a nêmesis do mais recente IA-64. Isso também reflete a transição de barramentos paralelos para redes e switches seriais. Os últimos capítulos descrevem o Google, que nasceu após a segunda edição, em termos da sua tecnologia de cluster e dos novos usos da pesquisa.

Em quinto lugar, embora muitos alunos de Ciência da Computação e Engenharia gostem da tecnologia da informação pela tecnologia em si, alguns possuem interesses mais altruístas. Esse último grupo costuma ter mais mulheres e minorias pouco representadas. Consequentemente, acrescentamos um novo elemento ao livro, "Computadores no mundo real", com duas páginas encontradas en-

Capítulo ou apêndice	Seções	Foco no software	Foco no Hardware
1. Abstrações e tecnologias computacionais	1.1 a 1.6 ■ 1.7 (História)	■	■
	2.1 a 2.11 ■ 2.12 (Compiladores)	■	■
2. Instruções: linguagem de máquina	2.13 (Ordenação em C) ■ 2.14 (Java)	■	■
	2.15 a 2.18 ■ 2.19 (História)	■	■
3. Aritmética computacional	3.1 a 3.9 ■ 3.10 (História)	■	■
D. Arquiteturas com conjuntos de instruções RISC	■ D.1 a D.19	■	
4. Avaliando e compreendendo o desempenho	4.1 a 4.6 ■ 4.7 (História)	■	■
B. Fundamentos de Projeto lógico	■ B.1 a B.13		■
	5.1 (Introdução)	■	■
5. O Processador: Caminho de dados e controle	5.2 a 5.6 ■ 5.7 (Microcódigo) ■ 5.8 (Verilog)		■
	5.9 a 5.11 ■ 5.12 (História)	■	■
C. Mapeando controle em hardware	■ C.1 a C.6		■
	6.1 (Introdução)	■	■
6. Melhorando o desempenho com pipelining	6.2 a 6.6 ■ 6.7 (Verilog)		■
	6.8 a 6.9 ■ 6.10 a 6.12		■
	■ 6.13 (História)	■	■
7. Grande e rápida: explorando a hierarquia de memória	7.1 a 7.8 ■ 7.9 (História)	■	■
	8.1 a 8.2 ■ 8.3 (Redes)	■	■
8. Armazenamento, redes e outros periféricos	8.4 a 8.10 ■ 8.11 (História)	■	■
	■ 9.1 a 9.10 ■ 9.11 (História)	■	
9. Multiprocessadores e clusters	■ A.1 a A.12	■	■
A. Montadores, link-editores, e o simulador SPIM	Entre os capítulos	■	■
Computadores na vida real		■	■

Leia cuidadosamente ■

Revise ou leia ■

Leia se tiver tempo ■

Leia por cultura ■

Referência ■

tre os capítulos. Nossa ponto de vista é de que a tecnologia da informação é mais valiosa para a humanidade do que a maioria dos tópicos que você poderia estudar – seja ele preservar nossa herança artística, ajudar o Terceiro Mundo, salvar nosso meio ambiente ou ainda mudar sistemas políticos – e, assim, demonstramos nossa visão com exemplos concretos de aplicações não-tradicionalas. Acreditamos que os leitores desses segmentos terão uma apreciação maior da cultura da computação, além da tecnologia inherentemente interessante, assim como aqueles que leem as seções de história ao final de cada capítulo.

Por fim, os livros são como pessoas: eles normalmente ficam maiores com o passar do tempo. Usando a tecnologia, conseguimos fazer tudo isso e ainda reduzir o livro em centenas de páginas. Como ilustramos na tabela, a parte principal do livro para leitores de hardware e software está em papel, mas as seções que alguns leitores valorizariam mais do que outras aparecem no CD que acompanha o livro. Essa tecnologia também permite que seus autores ofereçam histórias maiores e exercícios mais extensos sem que se preocupem em não aumentar o tamanho do livro. Quando incluímos o CD, pudemos então incluir muitos softwares gratuitos e tutoriais (em inglês), que muitos professores nos disseram que gostariam de usar em seus cursos. Essa mistura de publicação em papel e CD pesa cerca de 30% menos do que há seis anos – um objetivo muito perseguido, tanto para livros quanto para pessoas.

Comentários finais

Ao ler a seção de agradecimentos a seguir, verá que trabalhamos bastante para corrigir erros. Como um livro passa por muitas tiragens, temos a oportunidade de fazer ainda mais correções.

Este livro é verdadeiramente colaborativo, apesar de um de nós estar dirigindo uma grande universidade. Juntos, estudamos as idéias e o método de apresentação, depois escrevemos individualmente metade dos capítulos e atuamos como revisores para cada rascunho da outra metade. A quantidade de páginas sugere que mais uma vez escrevemos quase exatamente a mesma proporção. Assim, compartilhamos a mesma culpa pelo que você está para ler.

Agradecimentos da terceira edição

Gostaríamos novamente de expressar nossa gratidão a Jim Larus pela vontade em contribuir com sua experiência na programação em linguagem assembly, além de incentivar os leitores deste livro a usarem o simulador que ele desenvolveu e continua mantendo. Nosso revisor de exercícios, **Dan Sorin**, assumiu a tarefa hercúlea de acrescentar novos exercícios e suas respostas. **Peter Ashenden** também trabalhou bastante para coletar e organizar o CD que acompanha o livro.

Somos gratos aos muitos instrutores que responderam às pesquisas da editora, revisaram nossas propostas e participaram dos grupos para analisar e responder aos nossos planos para esta edição. Entre eles estão: Michael Anderson (University of Hartford), David Bader (University of New Mexico), Rusty Baldwin (Air Force Institute of Technology), John Barr (Ithaca College), Jack Briner (Charleston Southern University), Mats Brorsson (KTH, Suécia), Colin Brown (Franklin University), Lori Carter (Point Loma Nazarene University), John Casey (Northeastern University), Gene Chase (Messiah College), George Cheney (University of Massachusetts, Lowell), Daniel Citron (Jerusalem College of Technology, Israel), Albert Cohen (INRIA, França), Lloyd Dickman (PathScale), Jose Duato (Universidad Politecnica de Valencia, Espanha), Ben Dugan (University of Washington), Derek Eager (University of Saskatchewan, Canadá), Magnus Ekman (Chalmers University of Technology, Suécia), Ata Elahi (Southern Connecticut State University), Soundararajan Ezeikel (Indiana University of Pennsylvania), Ernest Ferguson (Northwest Missouri State University), Michael Fry (Lebanon Valley College, Pensilvânia), R. Gaede (University of Arkansas at Little Rock), Jean-Luc Gaudiot (University of California, Irvine), Thomas Gendreau (University of Wisconsin, La Crosse), George Georgiou (California State University, San Bernardino), Paul Gillard (Memorial University of Newfoundland, Canadá), Joe Grimes (California Polytechnic State University, SLO), Max Hailperin (Gustavus Adolphus College), Jayantha Herath (St. Cloud State University, Minnesota), Mark Hill (University of Wisconsin, Madison), Michael Hsaio (Virginia Tech), Richard Hughey (University of California, Santa Cruz), Tony Jebara (Columbia University), Elizabeth Johnson (Xavier University), Peter Kogge (University of Notre Dame), Morris Lancaster (BAH), Doug Lawrence (University of Montana), David Lilja (University of Minnesota), Nam Ling (Santa Clara University, Califórnia), Paul Lum (Agilent Technologies), Stephen Mann (University of Waterloo, Canadá), Diana Marculescu (Carnegie Mellon University), Margaret McMahon (U.S. Naval

Academy Computer Science), Uwe Meyer-Baese (Florida State University), Chris Milner (University of Virginia), Tom Pittman (Southwest Baptist University), Jalel Rejeb (San Jose State University, Califórnia), Bill Siever (University of Missouri, Rolla), Kevin Skadron (University of Virginia), Pam Smallwood (Regis University, Colorado), K. Stuart Smith (Rocky Mountain College), William J. Taffe (Plymouth State University), Michael E. Thomodakis (Texas A&M University), Ruppa K. Thulasiram (University of Manitoba, Canadá), Ye Tung (University of South Alabama), Steve VanderLeest (Calvin College), Neal R. Wagner (University of Texas at San Antonio) e Kent Wilken (University of California, Davis).

Também somos gratos aos que cuidadosamente leram nossos manuscritos de rascunho; alguns leram rascunhos sucessivos para ajudar a garantir que nenhum erro apareceria enquanto revisávamos. Entre eles estão: Krste Asanovic (Massachusetts Institute of Technology), Jean-Loup Baer (University of Washington), David Brooks (Harvard University), Doug Clark (Princeton University), Dan Connors (University of Colorado at Boulder), Matt Farrens (University of California, Davis), Manoj Franklin (University of Maryland College Park), John Greiner (Rice University), David Harris (Harvey Mudd College), Paul Hilfinger (University of California, Berkeley), Norm Jouppi (Hewlett-Packard), David Kaeli (Northeastern University), David Oppenheimer (University of California, Berkeley), Timothy Pinkston (University of Southern California), Mark Smotherman (Clemson University) e David Wood (University of Wisconsin, Madison).

Para nos ajudar a cumprir nosso objetivo de criar 70% de exercícios e soluções novas para esta edição, recrutamos vários alunos formados, recomendados por seus professores. Agradecemos sua criatividade e persistência: Michael Black (University of Maryland), Lei Chen (University of Rochester), Nirav Dave (Massachusetts Institute of Technology), Wael El Essawy (University of Rochester), Nikil Mehta (Brown University), Nicholas Nelson (University of Rochester), Aaron Smith (University of Texas, Austin) e Charlie Wang (Duke University).

Gostaríamos de agradecer especialmente a **Mark Smotherman** que fez uma revisão final cuidadosa, para descobrir problemas técnicos e de escrita, o que melhorou significativamente a qualidade desta edição.

Gostaríamos de agradecer a toda a família Morgan Kaufmann que concordou em publicar este livro novamente, sob a liderança capaz de **Denise Penrose**. Ela desenvolveu a visão do livro híbrido (papel e CD) e recrutou as muitas pessoas anteriormente relacionadas, que desempenharam papéis importantes no desenvolvimento do livro.

Simon Crump gerenciou o processo de produção do livro, e **Summer Block** coordenou a pesquisa com os usuários e suas respostas. Agradecemos também aos muitos fornecedores autônomos que contribuíram para este volume, especialmente **Nancy Logan** e à **Dartmouth Publishing, Inc.**, responsáveis pela editoração eletrônica.

As contribuições de quase 100 pessoas que mencionamos aqui tornaram esta terceira edição nosso melhor livro até agora. Divirta-se!

David A. Patterson

John L. Hennessy

ORGANIZAÇÃO E PROJETO DE COMPUTADORES



Abstrações e Tecnologias Computacionais

*A civilização avança estendendo
o número de operações importantes
que podem ser realizadas sem se
pensar nelas.*

Alfred North Whitehead

Uma Introdução à Matemática, 1911

1.1	Introdução	3
1.2	Por baixo do seu programa	9
1.3	Sob as tampas	11
1.4	Vida real: fabricando chips de Pentium 4	22
1.5	Falácia e armadilhas	25
1.6	Comentários finais	27
■ 1.7	Perspectiva histórica e leitura adicional	28
1.8	Exercícios	28

1.1

Introdução

Bem-vindo a este livro! Estamos felizes por ter a oportunidade de compartilhar o entusiasmo do mundo dos sistemas computacionais. Esse não é um campo árido e monótono, no qual o progresso é glacial e as novas idéias se atrofiam pelo esquecimento. Não! Os computadores são o produto da impressionante e vibrante indústria da tecnologia da informação, cujos aspectos são responsáveis por quase 10% do produto interno bruto dos Estados Unidos. Essa área incomum abraça a inovação com uma velocidade surpreendente. Desde 1985, surgiram inúmeros novos computadores que prometiam revolucionar a indústria da computação; essas revoluções sempre foram interrompidas quando alguém construía um computador ainda melhor.

Essa corrida para inovar levou a um progresso sem precedentes desde o início da computação eletrônica no final da década de 1940. Se o setor de transportes, por exemplo, tivesse tido o mesmo desenvolvimento da indústria da computação, hoje nós poderíamos viajar de costa a costa dos Estados Unidos em aproximadamente um segundo por apenas alguns centavos! Imagine por alguns instantes como esse progresso mudaria a sociedade – morar em Taiti e trabalhar em São Francisco, indo para Moscou no início da noite para assistir a uma apresentação do balé de Bolshoi. Não é difícil imaginar as implicações dessa mudança.

Os computadores levaram a humanidade a enfrentar uma terceira revolução, a revolução da informação, que assumiu seu lugar junto das revoluções industrial e agrícola. A multiplicação da força e do alcance intelectuais do ser humano afetou naturalmente muito nossas vidas cotidianas, além de ter mudado a maneira como conduzimos a busca de novos conhecimentos. Agora, existe um novo veio de investigação científica, com a ciência da computação unindo os cientistas teóricos e experimentais na exploração de novas fronteiras na astronomia, biologia, química, física etc.

A revolução dos computadores continua. Cada vez que o custo da computação melhora por um fator de 10, as oportunidades para os computadores se multiplicam. As aplicações que eram economicamente proibitivas de repente se tornam viáveis. As seguintes aplicações, no passado recente, eram “ficção científica para a computação”:

- *Caixas automáticas*: computadores instalados nas paredes dos bancos para distribuir e coletar dinheiro teria sido um conceito ridículo na década de 1950, quando o computador mais barato custava pelo menos US\$500 mil e tinha o tamanho de um carro.

- *Computação em automóveis*: até os microprocessadores melhorarem significativamente de preço e desempenho no início dos anos 80, o controle dos carros por computadores era considerado um absurdo. Hoje, os computadores reduzem a poluição e melhoram a eficiência do combustível, usando controles no motor, e aumentam a segurança por meio da prevenção de derrapagens perigosas e pela ativação de *air-bags* para proteger os passageiros em caso de colisão.
- *Computadores laptop*: quem sonharia que os avanços dos sistemas computacionais levariam aos computadores laptop, permitindo que estudantes levassem computadores para restaurantes e aviões?
- *Projeto do genoma humano*: o custo do equipamento computacional para mapear e analisar as sequências do DNA humano é de centenas de milhares de dólares. É improvável que alguém teria considerado esse projeto se os custos computacionais fossem 10 a 100 vezes mais altos, como há dez ou 20 anos.
- *World Wide Web*: ainda não existente na época da primeira edição deste livro, a *World Wide Web* transformou nossa sociedade. Entre suas finalidades, estão a distribuição de notícias, o envio de flores, a compra por catálogos on-line, a realização de passeios eletrônicos para ajudar a escolher locais de férias, o encontro de pessoas que compartilham os interesses mais obscuros, e até mesmo assuntos mundanos como encontrar as notas de aula dos autores dos seus livros-textos.

Claramente, os avanços dessa tecnologia hoje afetam quase todos os aspectos da nossa sociedade. Os avanços de hardware permitiram que os programadores criassem softwares maravilhosamente úteis e explicassem por que os computadores são onipresentes. As aplicações do futuro que ainda são consideradas ficção científica são a sociedade sem dinheiro, as estradas inteligentes automatizadas e a computação genuinamente universal: ninguém carregando computadores porque eles estão em todo lugar.

Classes de aplicações de computador e suas características

Embora um conjunto comum de tecnologias de hardware (discutidas nas Seções 1.3 e 1.4) seja usado em computadores variando dos dispositivos domésticos inteligentes e telefones celulares aos maiores supercomputadores, essas diferentes aplicações possuem diferentes necessidades de projeto e empregam os fundamentos das tecnologias de hardware de diversas maneiras. Genericamente falando, os computadores são usados em três diferentes classes de aplicações.

Os **computadores desktop** são possivelmente a forma mais conhecida de computação e caracterizam-se pelo computador pessoal, que a maioria dos leitores deste livro provavelmente já usou extensivamente. Os computadores desktop enfatizam o bom desempenho a um único usuário por um baixo custo e normalmente são usados para executar software independente, também chamado de software shrink-wrap. Computadores desktop são um dos maiores mercados para os computadores, e a evolução de muitas tecnologias de computação é motivada por essa classe de computação, que só tem cerca de 30 anos!

Os **servidores** são a forma moderna do que, antes, eram os mainframes, minicomputadores e supercomputadores, e, em geral, são acessados apenas por meio de uma rede. Os servidores são projetados para suportar grandes cargas de trabalho, que podem consistir em uma única aplicação complexa, normalmente científica ou de engenharia, ou manipular muitas tarefas pequenas, como ocorreria no caso de um grande servidor Web. Essas aplicações muitas vezes são baseadas em software de outra origem (como um banco de dados ou sistema de simulação), mas freqüentemente são modificadas ou personalizadas para uma função específica. Os servidores são construídos a partir da mesma tecnologia básica dos computadores desktop, mas fornecem uma maior capacidade de expansão tanto da capacidade de processamento quanto de entrada/saída. Como veremos no Capítulo 4, o desempenho de um servidor pode ser medido de várias maneiras diferentes, dependendo da aplicação de interesse. Em geral, os servidores também dão grande ênfase à estabilidade, já que uma falha normalmente é mais prejudicial do que seria em um computador desktop de um único usuário.

computador desktop

Um computador projetado para uso por uma única pessoa, normalmente incorporando um monitor gráfico, um teclado e um mouse.

servidor

Um computador usado para executar grandes programas para múltiplos usuários quase sempre de maneira simultânea e normalmente acessado apenas por meio de uma rede.

Os servidores abrangem a faixa mais ampla em termos de custo e capacidade. Na sua forma mais simples, um servidor pode ser menor do que uma máquina desktop sem monitor ou teclado e com um custo de mil dólares. Esses servidores de baixa capacidade normalmente são usados para armazenamento de arquivos, pequenas aplicações comerciais ou serviço Web simples. No outro extremo, estão os **supercomputadores**, que, atualmente, consistem em centenas ou milhares de processadores e, em geral, de gigabytes a terabytes de memória e de terabytes a petabytes de armazenamento, e custam desde milhões até centenas de milhões de dólares. Os supercomputadores normalmente são usados para cálculos científicos e de engenharia de alta capacidade, como previsão do tempo, exploração de petróleo, determinação da estrutura da proteína e outros problemas de grande porte. Embora esses supercomputadores representem o topo da capacidade de computação, eles são uma fração relativamente pequena dos servidores e do mercado de computadores geral em termos de receita total.

Os **computadores embutidos** são a maior classe de computadores e abrangem a faixa mais ampla de aplicações e desempenho. Os computadores embutidos incluem os microprocessadores encontrados em sua máquina de lavar e em seu carro, os computadores em um telefone celular ou PDA, os computadores em um videogame ou televisão digital, e as redes de processadores que controlam um avião moderno ou um navio de carga. Os sistemas de computação embutidos são projetados para executar uma aplicação ou um conjunto de aplicações relacionadas como um único sistema; portanto, apesar do grande número de computadores embutidos, a maioria dos usuários nunca vê realmente que está usando um computador!

As aplicações embutidas normalmente possuem necessidades específicas que combinam um desempenho mínimo com limitações rígidas em relação a custo ou potência. Por exemplo, considere um telefone celular: o processador só precisa ser tão rápido quanto o necessário para manipular sua função limitada; além disso, minimizar custo e potência é o objetivo mais importante. Apesar do seu baixo custo, os computadores embutidos freqüentemente possuem a menor tolerância a falhas, já que os resultados podem variar desde um simples incômodo, quando sua nova TV falha, até a completa devastação que poderia ocorrer quando o computador em um avião ou em um carro falha. Nas aplicações embutidas orientadas para o consumidor, como um eletrodoméstico digital, a estabilidade é obtida principalmente por meio da simplicidade – a ênfase está em realizar uma função o mais perfeitamente possível. Nos grandes sistemas embutidos, freqüentemente são empregadas as mesmas técnicas de redundância utilizadas nos servidores. Embora este livro se concentre nos computadores de uso geral, a maioria dos conceitos se aplica diretamente – ou com ligeiras modificações – aos computadores embutidos. Em várias partes, mencionaremos alguns dos aspectos exclusivos dos computadores embutidos.

A Figura 1.1 mostra que, durante os últimos anos, o crescimento do número de computadores embutidos foi muito mais rápido (taxa de crescimento anual composta de 40%) do que o crescimento entre os computadores desktop e servidores (9% anualmente). Observe que os computadores embutidos incluem telefones celulares, videogames, TVs digitais, PDAs e uma variedade de aparelhos desse tipo. Veja que esses dados não incluem dispositivos de controle embutidos de baixa capacidade que usam processadores de 8 e 16 bits.

Detalhamento: os detalhamentos são seções curtas usadas em todo o texto para fornecer mais detalhes sobre um determinado assunto, que pode ser de interesse. Os leitores que não possuem um interesse específico no tema podem pular essas seções, já que o material subsequente nunca dependerá do conteúdo desta seção.

Muitos processadores embutidos são projetados usando *núcleos de processador*, uma versão de um processador escrita em uma linguagem de descrição de hardware como Verilog ou VHDL. O núcleo permite que um projetista integre outro hardware específico para uma aplicação com o núcleo do processador para a fabricação de um único chip. A disponibilidade de ferramentas de síntese que podem gerar um chip de uma especificação Verilog, juntamente com a capacidade dos chips de silício modernos, tem tornado esses processadores de finalidade especial altamente atraentes. Como o núcleo pode ser sintetizado para diferentes linhas de fabricação de semicondutor, o uso de um núcleo também oferece flexibilidade na escolha de um fabricante. Nos últimos anos, o uso dos núcleos cresceu muito rápido. Por exemplo, em 1998, apenas 31% dos processadores embutidos eram núcleos, enquanto em 2002, 56% desses processadores eram constituídos de núcleos. Além disso, esse crescimento foi liderado principalmente pelos núcleos: a taxa composta de crescimento anual foi de 63%!

supercomputador Uma classe de computadores com desempenho e custo mais altos; eles são configurados como servidores e normalmente custam milhões de dólares.

terabyte Originalmente, 1.099.511.627.776 (2^{40}) bytes, embora alguns sistemas de comunicações e de armazenamento secundário o tenham redefinido como significando 1.000.000.000.000 (10^{12}) bytes.

computador embutido Um computador dentro de outro dispositivo, usado para executar uma aplicação predeterminada ou uma coleção de software.

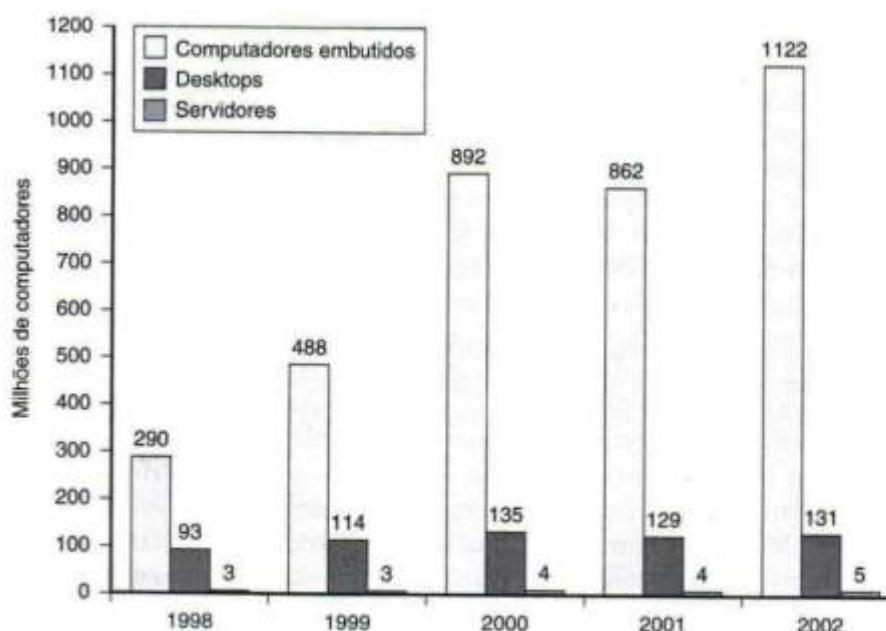


FIGURA 1.1 Número de processadores diferentes vendidos entre 1998 e 2002. Esses números são obtidos de maneira um tanto diferente e, portanto, é necessário algum cuidado na interpretação dos resultados. Por exemplo, os totais para os desktops e servidores consideram sistemas computacionais completos, pois uma parte deles inclui processadores múltiplos; o número de processadores vendidos é um pouco maior, mas provavelmente em apenas 10 a 20% no total (já que os servidores, que podem ter em média mais de um processador por sistema, são apenas cerca de 3% das vendas de desktops, que são predominantemente sistemas monoprocessados). Os totais para computadores embutidos consideram realmente processadores, muitos dos quais não são sequer visíveis e, em alguns casos, pode haver vários processadores por dispositivo.

A Figura 1.2 mostra as principais arquiteturas vendidas nesses mercados com totais para cada arquitetura, por meio dos três tipos de produtos (embutidos, desktop e servidor). Apenas os processadores de 32 e 64 bits estão incluídos, embora os processadores de 32 bits sejam a grande maioria das arquiteturas.

O que você pode aprender neste livro

Os bons programadores sempre se preocuparam com o desempenho de seus programas porque gerar resultados rapidamente para o usuário é uma condição essencial na criação bem-sucedida de software. Nas décadas de 1960 e 1970, uma grande limitação no desempenho dos computadores era o tamanho da memória do computador. Assim, os programadores freqüentemente seguiam um princípio simples: minimizar o espaço ocupado na memória para tornar os programas mais rápidos. Na última década, os avanços em arquitetura de computadores e nas tecnologias de fabricação de memórias reduziram drasticamente a importância do tamanho da memória na maioria das aplicações, com exceção dos sistemas embutidos.

Agora, os programadores interessados em desempenho precisam entender os problemas que substituíram o modelo de memória simples dos anos 60: a natureza hierárquica das memórias e a natureza paralela dos processadores. Os programadores que desejam construir versões competitivas de compiladores, sistemas operacionais, bancos de dados e mesmo aplicações precisarão, portanto, aumentar seu conhecimento em organização de computadores.

Sentimo-nos honrados com a oportunidade de explicar o que existe dentro da máquina revolucionária, decifrando o software por trás do seu programa e o hardware sob a tampa do seu computador. Ao concluir este livro, acreditamos que você será capaz de responder às seguintes perguntas:

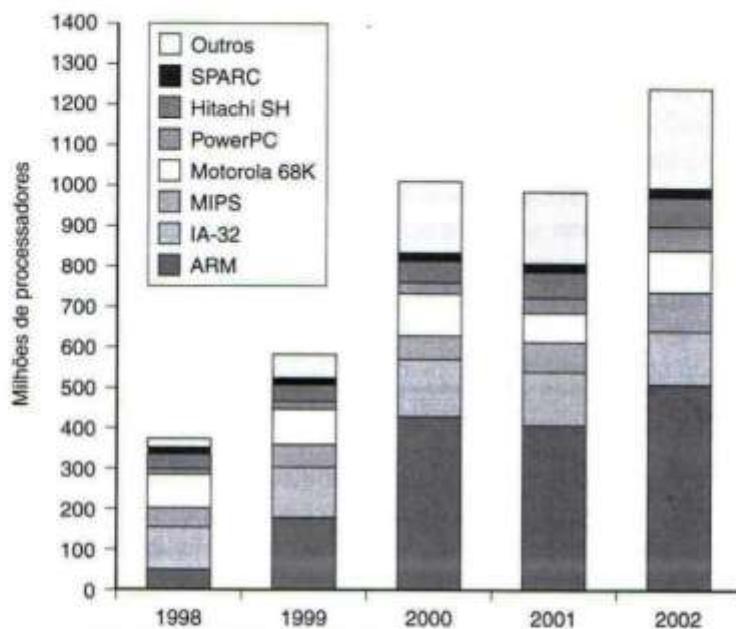


FIGURA 1.2 Vendas de microprocessadores entre 1998 e 2002 por arquitetura combinando todos os usos. A categoria “outros” se refere aos processadores específicos à aplicação ou às arquiteturas personalizadas. No caso do ARM, aproximadamente 80% das vendas são para telefones celulares, sendo que um núcleo do ARM é usado em conjunto com lógica específica da aplicação em um chip.

- Como os programas escritos em uma linguagem de alto nível, como C ou Java, são traduzidos para a linguagem de máquina e como o hardware executa os programas resultantes? Compreender esses conceitos forma o alicerce para entender os aspectos do hardware e software que afetam o desempenho dos programas.
- O que é a interface entre o software e o hardware, e como o software instrui o hardware a realizar as funções necessárias? Esses conceitos são vitais para entender como escrever muitos tipos de software.
- O que determina o desempenho de um programa e como um programador pode melhorar o desempenho? Como veremos, isso depende do programa original, da tradução desse programa para a linguagem do computador e da eficiência do hardware em executar o programa.
- Que técnicas podem ser usadas pelos projetistas de hardware para melhorar o desempenho? Este livro apresentará os conceitos básicos do projeto de computador moderno. O leitor interessado encontrará muito mais material sobre esse assunto em nosso livro avançado, *Arquitetura de Computadores: Uma abordagem quantitativa*.

Sem entender as respostas a essas perguntas, melhorar o desempenho do seu programa em um computador moderno ou avaliar que recursos podem tornar um computador melhor do que outro para uma determinada aplicação será um complicado processo de tentativa e erro, em vez de um procedimento científico conduzido por consciência e análise.

Este primeiro capítulo prepara a fundação para o restante do livro. Ele introduz as idéias e definições básicas, coloca os principais componentes de software e hardware em perspectiva e apresenta os circuitos integrados, a tecnologia que estimula a revolução dos computadores. Neste capítulo e em capítulos seguintes, você provavelmente verá muitas palavras novas ou palavras que já pode ter ouvido, mas não sabe ao certo o que significam. Não entre em pânico! Sim, há muita terminologia especial usada para descrever os computadores modernos, mas ela realmente ajuda, uma vez que nos permite descrever precisamente uma função ou capacidade. Além disso, os projetistas de computa-

acrônimo Uma palavra construída tomando-se as letras iniciais das palavras. Por exemplo: RAM é um acrônimo para Random Access Memory (memória de acesso aleatório) e CPU é um acrônimo para Central Processing Unit (unidade central de processamento).

dor (inclusive estes autores) *adoram* usar **acrônimos**, que são fáceis de entender quando se sabe o que as letras significam! Para ajudá-lo a lembrar e localizar termos, nós incluímos uma definição destacada de cada termo novo na primeira vez que aparece no texto. Após um pequeno período trabalhando com a terminologia, você estará fluente e seus amigos ficarão impressionados quando você usar corretamente palavras como BIOS, DIMM, CPU, cache, DRAM, ATA, PCI e muitas outras.*

Para enfatizar como os sistemas de software e hardware usados para executar um programa irão afetar o desempenho, usamos uma seção especial, “Entendendo o desempenho dos programas”, em todo o livro, sendo que a primeira aparece a seguir. Esses elementos resumem importantes conceitos quanto ao desempenho do programa.

Entendendo o desempenho dos programas

O desempenho de um programa depende de uma combinação entre a eficácia dos algoritmos usados no programa, os sistemas de software usados para criar e traduzir o programa para instruções de máquina e da eficácia do computador em executar essas instruções, que podem incluir operações de E/S (entrada/saída). A tabela a seguir descreve como o hardware e o software afetam o desempenho.

Componente de hardware ou software	Como este componente afeta o desempenho	Onde este assunto é abordado?
Algoritmo	Determina o número de instruções do código-fonte e o número de operações de E/S realizadas	Outros livros!
Linguagem de programação, compilador e arquitetura	Determina o número de instruções de máquina para cada instrução em nível de fonte	Capítulos 2 e 3
Processador e sistema de memória	Determina a velocidade em que as instruções podem ser executadas	Capítulos 5, 6 e 7
Sistema de E/S (hardware e sistema operacional)	Determina a velocidade em que as operações de E/S podem ser executadas	Capítulo 8

Verifique você mesmo

As Seções “Verifique você mesmo” se destinam a ajudar os leitores a avaliar se compreenderam os principais conceitos introduzidos em um capítulo e se entenderam as implicações desses conceitos. Algumas questões “Verifique você mesmo” possuem respostas simples; outras são para discussão em grupo. As respostas às questões específicas podem ser encontradas no final do capítulo. As questões “Verifique você mesmo” aparecem apenas no final de uma seção, o que faz com que fique mais fácil pulá-las se você estiver certo de que entendeu o assunto.

1. A Seção 1.1 mostrou que o número de processadores embutidos vendidos a cada ano supera, e muito, o número de processadores para desktops. Você pode confirmar ou negar isso com base em sua própria experiência? Tente contar o número de processadores embutidos na sua casa. Como esse número se compara ao número de computadores desktop em sua casa?
2. Como mencionado anteriormente, tanto o software quanto o hardware afetam o desempenho de um programa. Você pode pensar em exemplos em que cada um dos fatores a seguir é o responsável pelo gargalo no desempenho?
 - O algoritmo escolhido
 - A linguagem de programação ou compilador
 - O sistema operacional
 - O processador
 - O sistema de E/S e os dispositivos

1.2

Por baixo do seu programa

Uma aplicação típica, como um processador de textos ou um grande sistema de banco de dados, pode consistir em centenas de milhares a milhões de linhas de código e se basear em bibliotecas de software sofisticadas que implementam funções complexas no suporte da aplicação. Como veremos, o hardware em um computador só pode executar instruções de baixo nível extremamente simples. Ir de uma aplicação complexa até as instruções simples envolve várias camadas de software que interpretam ou traduzem operações de alto nível nas instruções simples do computador.

Essas camadas de software são organizadas de maneira hierárquica, na qual as aplicações são o anel mais externo e uma variedade de **software de sistemas** situando-se entre o hardware e as aplicações, como mostra a Figura 1.3.

Existem muitos tipos de software de sistemas, mas dois tipos são fundamentais em todos os sistemas computacionais modernos: um sistema operacional e um compilador. Um **sistema operacional** fornece a interface entre o programa de usuário e o hardware e disponibiliza diversos serviços e funções de supervisão. Entre as funções mais importantes estão:

- manipular as operações básicas de entrada e saída
- alojar armazenamento e memória
- possibilitar e controlar o compartilhamento do computador entre as diversas aplicações que o utilizam simultaneamente

Exemplos de sistemas operacionais em uso hoje são Windows, Linux e MacOS.

Os **compiladores** realizam outra função fundamental: a tradução de um programa escrito em uma linguagem de alto nível, como C ou Java, em instruções que o hardware possa executar. Devido à sofisticação das linguagens de programação modernas e às instruções simples executadas pelo hardware, a tradução de um programa de linguagem de alto nível para instruções de hardware é complexa. Voltaremos ao assunto e veremos um breve resumo do processo no Capítulo 2.

De uma linguagem de alto nível para a linguagem do hardware

Para poder falar com uma máquina eletrônica, você precisa enviar sinais elétricos. Os sinais mais fáceis de ser entendidos pelas máquinas são *ligado* e *desligado*; portanto, o alfabeto da máquina se re-

Em Paris, eles simplesmente olhavam perdidos quando eu falava em francês; eu nunca consegui fazer aqueles idiotas entenderem sua própria língua.

Mark Twain,
The Innocents Abroad, 1869

software de sistemas
Software que fornece serviços que normalmente são úteis, inclusive sistemas operacionais, compiladores e montadores.

sistema operacional
Programa de supervisão que gerencia os recursos de um computador para o benefício dos programas executados nessa máquina.

compilador Um programa que traduz as instruções de linguagem de alto nível para instruções de linguagem assembly.

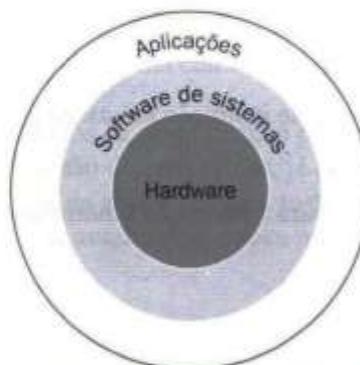


FIGURA 1.3 Uma visão simplificada do hardware e software como camadas hierárquicas, mostradas como círculos concêntricos, onde o hardware está no centro e as aplicações aparecem externamente.

Nas aplicações complexas, muitas vezes existem múltiplas camadas de software de aplicação. Por exemplo, um sistema de banco de dados pode ser executado sobre o software de sistemas hospedando uma aplicação, que, por sua vez, roda sobre o banco de dados.

dígito binário Também chamado bit. Um dos dois números na base 2 (0 ou 1) que são os componentes da informação.

montador (assembler) Um programa que traduz uma versão simbólica de instruções para a versão binária.

linguagens de programação de alto nível Uma linguagem, como C, Fortran ou Java, composta de palavras e notação algébrica, que pode ser traduzida por um compilador para a linguagem assembly.

sume a apenas duas letras. Assim como as 23 letras do alfabeto português não limitam o quanto pode ser escrito, as duas letras do alfabeto do computador não limitam o que os computadores podem fazer. Os dois símbolos para essas duas letras são os números 0 e 1, e normalmente pensamos na linguagem de máquina como números na base 2, ou *números binários*. Chamamos cada “letra” de um **dígito binário** ou **bit**. Os computadores são escravos dos nossos comandos, chamados de instruções. As instruções, que são apenas grupos de bits que o computador entende, podem ser imaginadas como números. Por exemplo, os bits

1000110010100000

dizem ao computador para somar dois números. O Capítulo 3 explica por que usamos números para instruções e dados; não queremos roubar o brilho desse capítulo, mas usar números para instruções e dados é um dos conceitos básicos da computação.

Os primeiros programadores se comunicavam com os computadores em números binários, mas isso era tão maçante que rapidamente inventaram novas notações mais parecidas com a maneira como os humanos pensam. No início, essas notações eram traduzidas para binário manualmente, mas esse processo ainda era cansativo. Usando a própria máquina para ajudar a programá-la, os pioneiros inventaram programas para traduzir da notação simbólica para binário. O primeiro desses programas foi chamado de **montador (assembler)**. Esse programa traduz uma versão simbólica de uma instrução para uma versão binária. Por exemplo, o programador escreveria

add A, B

e o montador traduziria essa notação como

1000110010100000

Essa instrução diz ao computador para somar dois números, A e B. O nome criado para essa linguagem simbólica, ainda em uso hoje, é **linguagem assembly**.

Embora sendo um fantástico avanço, a linguagem assembly ainda está longe da notação que um cientista poderia desejar usar para simular fluxos de fluidos ou que um contador poderia usar para calcular seus saldos de contas. A linguagem assembly requer que o programador escreva uma linha para cada instrução que a máquina seguirá, obrigando o programador a pensar como a máquina.

A descoberta de que um programa poderia ser escrito para traduzir uma linguagem mais poderosa em instruções de computador foi um dos mais importantes avanços nos primeiros dias da computação. Os programadores atuais devem sua produtividade – e sua sanidade mental – à criação de **linguagens de programação de alto nível** e de compiladores que traduzem os programas escritos nessas linguagens em instruções.

Um compilador permite que um programador escreva esta expressão em linguagem de alto nível:

A + B

O compilador compilaria isso na seguinte instrução em assembly:

add A, B

O montador traduziria essa instrução para a instrução binária que diz ao computador para somar os dois números, A e B:

1000110010100000

A Figura 1.4 mostra as relações entre esses programas e linguagens.

As linguagens de programação de alto nível oferecem vários benefícios importantes. Primeiro, elas permitem que o programador pense em uma linguagem mais natural, usando palavras em inglês e notação algébrica, resultando em programas que se parecem muito mais com texto do que com tabelas de símbolos enigmáticos (veja a Figura 1.4). Além disso, elas permitem que linguagens sejam projetadas de acordo com o uso pretendido. É por isso que a Fortran foi projetada para computação

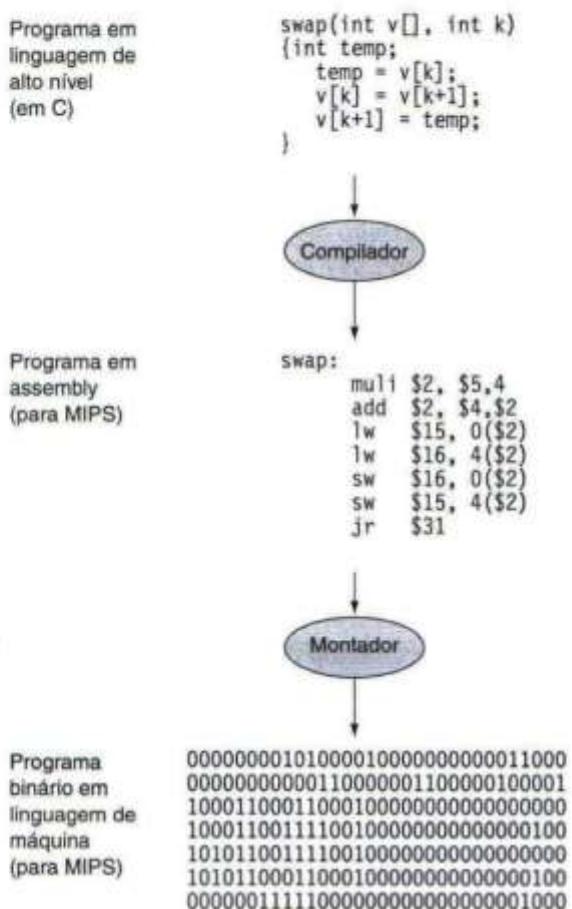


FIGURA 1.4 Programa em C compilado para assembly e, depois, montado em linguagem de máquina.

Embora a tradução de linguagem de alto nível para a linguagem de máquina seja mostrada em duas etapas, alguns compiladores produzem linguagem de máquina diretamente. Essas linguagens e esse programa são analisados em mais detalhes no Capítulo 2.

cientifica, Cobol para processamento de dados comerciais, Lisp para manipulação de símbolos e assim por diante.

A segunda vantagem das linguagens de programação é a maior produtividade do programador. Uma das poucas áreas em que existe consenso no desenvolvimento de software é que é necessário menos tempo para desenvolver programas quando são escritos em linguagens que exigem menos linhas para expressar uma idéia. A concisão é uma clara vantagem das linguagens de alto nível em relação à linguagem assembly.

A última vantagem é que as linguagens de programação permitem que os programas sejam independentes do computador no qual elas são desenvolvidas, já que os compiladores e montadores podem traduzir programas de linguagem de alto nível para instruções binárias de qualquer máquina. Essas três vantagens são tão fortes que, atualmente, pouca programação é realizada em assembly.

1.3 Sob as tampas

Agora que olhamos por trás do seu programa para descobrir como ele funciona, vamos abrir a tampa do computador para aprender sobre o hardware dentro dele. O hardware de qualquer computador realiza as mesmas funções básicas: inserir, processar e armazenar dados, além de gerar saída dos dados.

dos. A forma como essas funções são realizadas é o principal tema deste livro, e os capítulos subsequentes lidam com as diferentes partes dessas quatro tarefas. Quando tratamos de um aspecto importante neste livro, tão importante que esperamos que você se lembre dele para sempre, nós o enfatizamos identificando-o como um item “Colocando em perspectiva”. Temos aproximadamente 12 itens desses no livro, sendo que o primeiro descreve os cinco componentes de um computador que realizam as tarefas de inserir, processar, armazenar e gerar saída dos dados.

Colocando em perspectiva

Os cinco componentes de um computador são entrada, saída, memória, caminho de dados e controle, sendo que os dois últimos, às vezes, são combinados e chamados de processador. A Figura 1.5 mostra a organização padrão de um computador. Essa organização é independente da tecnologia de hardware: você pode classificar cada parte de cada computador, antigos ou atuais, em uma dessas cinco categorias. Para ajudar a manter tudo isso em perspectiva, os cinco componentes de um computador são mostrados na primeira página dos capítulos seguintes, com a parte relativa ao capítulo destacada.

dispositivo de entrada
Um mecanismo por meio do qual o computador é alimentado com informações, como o teclado e o mouse.

dispositivo de saída
Um mecanismo que transmite o resultado de uma computação para o usuário ou para outro computador.

A Figura 1.6 mostra um computador desktop típico com teclado, mouse, monitor e um gabinete contendo ainda mais hardware. O que não está visível na fotografia é a rede que conecta o computador a outros computadores. Essa fotografia revela dois dos principais componentes dos computadores: os **dispositivos de entrada**, como o teclado e o mouse, e os **dispositivos de saída**, como o monitor. Como o nome sugere, a entrada alimenta o computador, e a saída é o resultado da computação enviada para o usuário. Alguns dispositivos, como redes e discos, fornecem tanto entrada quanto saída para o computador.

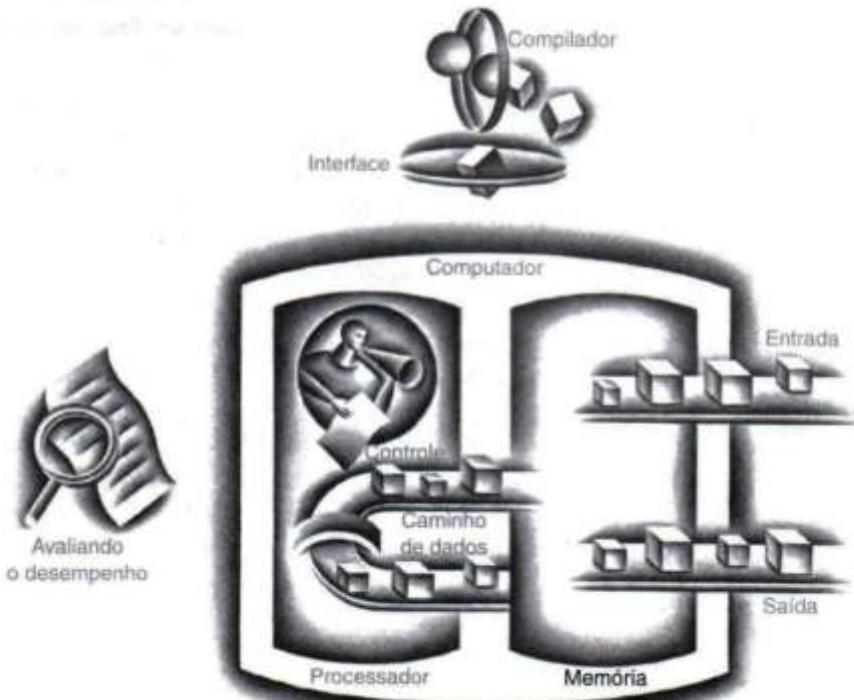


FIGURA 1.5 A organização de um computador, mostrando os cinco componentes clássicos. O processador obtém instruções e dados da memória. A entrada escreve dados na memória, e a saída lê os dados da memória. O controle envia os sinais que determinam as operações do caminho de dados, da memória, da entrada e da saída.

O Capítulo 8 descreve dispositivos de entrada e saída (E/S) em mais detalhes, mas vamos dar um passeio introdutório pelo hardware do computador, começando com os dispositivos de E/S externos.

Anatomia do mouse

Embora muitos usuários agora aceitem o mouse sem questionar, a idéia de um dispositivo apontador como um mouse foi mostrada pela primeira vez por Engelbart usando um protótipo em 1967. A Alto, que foi a inspiração para todas as estações de trabalho, inclusive para o Macintosh, incluiu um mouse como seu dispositivo apontador em 1973. Na década de 1990, todos os computadores desktop tinham esse dispositivo, e as novas interfaces gráficas com o usuário e os mouses se tornaram regra.

O mouse original era eletromecânico e usava uma grande esfera que, quando rolada sobre uma superfície, fazia com que um contador x e um y fossem incrementados. A quantidade do aumento de cada contador informava a distância e a direção em que o mouse tinha sido movido.

O mouse eletromecânico está sendo substituído pelo novo mouse ótico. O mouse ótico é, na verdade, um processador ótico em miniatura incluindo um LED para fornecer iluminação, uma minúscula câmera em preto-e-branco e um processador ótico simples. O LED ilumina a superfície abaixo do mouse; a câmera tira 1.500 fotografias em cada segundo sob a iluminação. Os quadros sucessivos são enviados para um processador ótico simples que compara as imagens e determina se e quanto o mouse foi movido. A substituição do mouse eletromecânico pelo mouse electroótico é uma ilustração de um fenômeno comum, no qual os custos cada vez menores e a segurança cada vez maior fazem uma solução eletrônica substituir a tecnologia eletromecânica mais antiga.

Tive a idéia de criar o mouse enquanto ouvia uma palestra de uma conferência. O orador era tão chato que comecei a me distrair e conceber a idéia.

Doug Engelbart



FIGURA 1.6 Um computador desktop. O monitor de cristal líquido (LCD) é o principal dispositivo de saída, e o teclado e o mouse são os principais dispositivos de entrada. O gabinete contém o processador e os dispositivos de E/S adicionais. Esse sistema é um Dell Optiplex GX260.

Pela tela do computador, aterrisssei um avião no pátio de uma transportadora. observei uma partícula nuclear colidir com uma fonte potencial, voei em um foguete quase na velocidade da luz e vi um computador revelar seus sistemas mais íntimos.

Ivan Sutherland, o "pai" da computação gráfica, citado em "Computer Software for Graphics", *Scientific American*, 1984

monitor de tubo de raios catódicos (CRT) Um monitor, como um aparelho de televisão, que exibe uma imagem usando um feixe de elétrons que varre a tela.
pixel O menor elemento da imagem. A tela é composta de centenas de milhares a milhões de pixels, organizados em uma matriz.

monitor de tela plana, monitor de cristal líquido Uma tecnologia de vídeo usando uma fina camada de polímeros líquidos que podem ser usados para transmitir ou bloquear a luz conforme uma corrente seja ou não aplicada.

monitor de matriz ativa Um monitor de cristal líquido usando um transistor para controlar a transmissão da luz em cada pixel individual.

Diante do espelho

Talvez o dispositivo de E/S mais fascinante seja o monitor gráfico. Baseado na tecnologia da televisão, um monitor de **tubo de raios catódicos (CRT)** varre uma imagem uma linha de cada vez, 30 a 75 vezes por segundo. Nessa *taxa de atualização*, as pessoas não percebem uma oscilação na tela.

A imagem é composta de uma matriz de elementos de imagem, ou **pixels**, que podem ser representados como uma matriz de bits, chamada *mapa de bits*, ou bitmap. Dependendo do tamanho da tela e da resolução, o tamanho da matriz de vídeo variava de 512×340 a 1920×1280 pixels em 2003. O monitor mais simples possui 1 bit por pixel, permitindo que ele seja preto ou branco. Para monitores que suportam 256 tons diferentes de preto e branco, algumas vezes chamados de monitores em *tons de cinza*, ou grayscale, são necessários 8 bits por pixel. Um monitor colorido pode usar 8 bits para cada uma das três cores primárias (vermelho, verde e azul), totalizando 24 bits por pixel, permitindo que milhões de cores diferentes sejam exibidas.

Todos os computadores laptop e todas as calculadoras, PDAs e celulares, além de muitos computadores desktop, usam **monitores de cristal líquido (LCDs)** em vez de CRTs para poder usar uma tela fina e de baixo consumo. A principal diferença é que o pixel do LCD não é a fonte da luz, mas controla a transmissão da luz. Um LCD típico contém moléculas em forma de bastão em um líquido que forma uma hélice retorcida que desvia a luz que entra no monitor, vinda de uma fonte de luz atrás do mesmo ou, menos frequentemente, vinda da luz refletida. Os bastões se tornam retos quando uma corrente é aplicada e não desviam mais a luz; como o material de cristal líquido está entre duas telas polarizadas a 90 graus, a luz não pode atravessar a menos que seja desviada. Hoje, a maioria dos monitores de cristal líquido usa uma **matriz ativa** que possui um minúsculo transistor em cada pixel para controlar precisamente a corrente e produzir imagens mais nitidas. Como em um CRT, uma máscara vermelha-verde-azul associada a cada pixel determina a intensidade dos três componentes de cor na imagem final; em um LCD de matriz ativa, existem três transistores em cada pixel.

Seja qual for o monitor, o suporte de hardware do computador para a utilização de gráficos consiste principalmente em um *buffer de atualização de varredura*, ou *buffer de quadros*, para armazenar o mapa de bits. A imagem a ser representada na tela é armazenada no buffer de quadros, e o padrão de bits de cada pixel é lido para o monitor gráfico a uma certa taxa de atualização. A Figura 1.7 mostra um buffer de quadros com 4 bits por pixel.

O objetivo do mapa de bits é representar fielmente o que está na tela. As dificuldades dos sistemas gráficos surgem porque o olho humano é muito bom em detectar mesmo as mais sutis mudanças na tela. Por exemplo, quando o vídeo está sendo atualizado, o olho pode detectar a inconsistência entre a parte da tela que foi alterada e a que não foi.

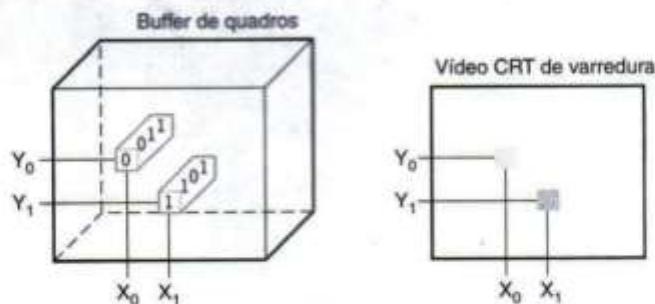


FIGURA 1.7 Cada coordenada no buffer de quadros à esquerda determina o tom da coordenada correspondente para o monitor CRT de varredura à direita. O pixel (X_0, Y_0) contém o padrão de bits 0011, que, na tela, é um tom de cinza mais claro do que o padrão de bits 1101 no pixel (X_1, Y_1) .

Abrindo o gabinete

Se abrirmos o gabinete de um computador, veremos uma interessante placa de plástico verde, coberta com dezenas de pequenos retângulos verdes ou pretos. A Figura 1.8 mostra o conteúdo do computador desktop da Figura 1.6. Essa **placa-mãe** é mostrada verticalmente à esquerda com a fonte de alimentação. Três unidades de disco – uma unidade de DVD, uma unidade Zip e um disco rígido – aparecem à direita.

Os pequenos retângulos na placa-mãe contêm os dispositivos que impulsionam nossa tecnologia avançada, os **circuitos integrados** ou **chips**. A placa é composta de três partes: a parte que se conecta aos dispositivos de E/S mencionados anteriormente, a memória e o processador. Os dispositivos de E/S são conectados por meio das duas placas grandes encaixadas perpendicularmente à placa-mãe próximos ao centro no lado direito.

A **memória** é onde os programas são mantidos quando estão sendo executados; ela também contém os dados necessários aos programas em execução. Na Figura 1.8, a memória é encontrada nas duas pequenas placas conectadas perpendicularmente próximas ao centro da placa-mãe. Cada pequena placa de memória contém oito circuitos integrados.

O **processador** é a parte ativa da placa, que segue rigorosamente as instruções de um programa. Ele soma e testa números, sinaliza dispositivos de E/S para serem ativados e assim por diante. O processador é o quadrado grande ao lado da ventoinha e abaixo da fonte no alto à esquerda da Figura 1.8. Ocionalmente, as pessoas chamam o processador de *CPU*, que significa o termo pomposo **unidade central de processamento**.

Descendo ainda mais para dentro do hardware, a Figura 1.9 revela detalhes do processador da Figura 1.8. O processador contém dois componentes principais: o caminho de dados e o controle, cor-

placa-mãe Uma placa de plástico contendo pacotes de circuitos integrados ou chips, incluindo processador, cache, memória e conectores para dispositivos de E/S, como redes e discos.

círculo integrado também chamado **chip**.

Um dispositivo que combina de dezenas a milhões de transistores.

memória A área de armazenamento temporária em que os programas são mantidos quando estão sendo executados e que contém os dados necessários para os programas em execução.

unidade central de processamento (CPU)

Também chamada de processador. A parte ativa do computador, que contém o caminho de dados e o controle e que soma e testa números e sinaliza dispositivos de E/S para que sejam ativados etc.

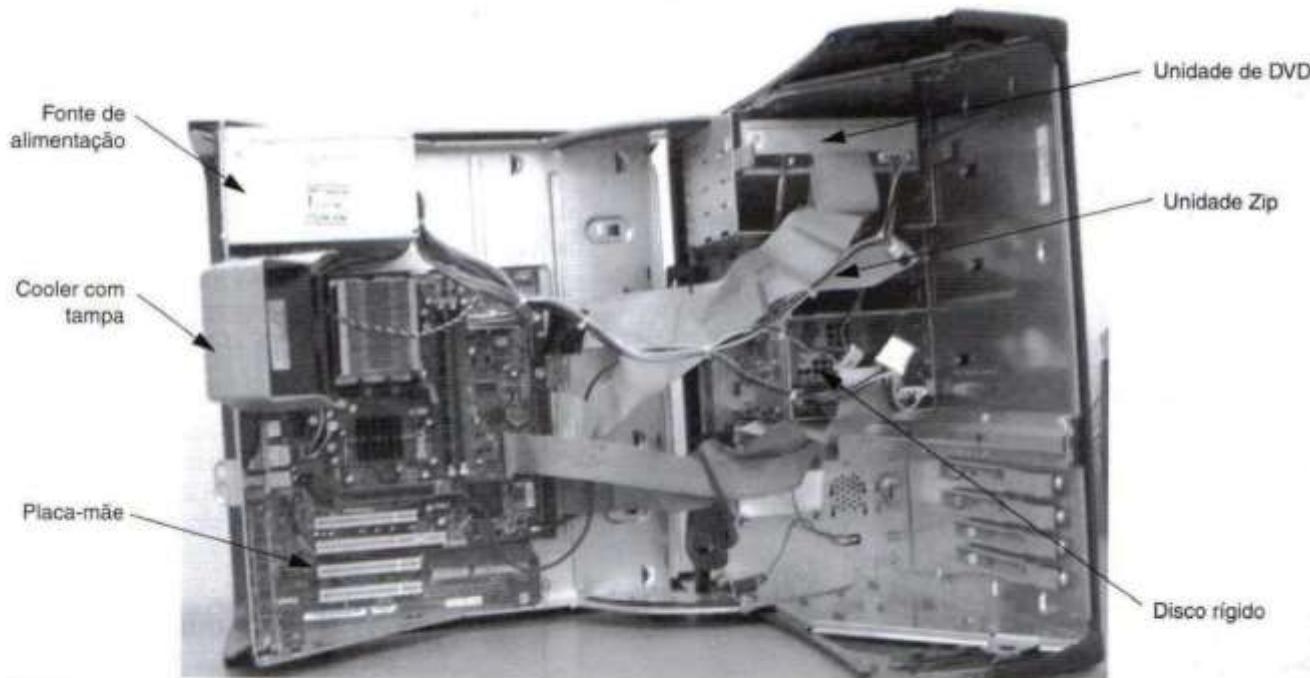


FIGURA 1.8 Dentro do computador pessoal da Figura 1.6. Essa embalagem algumas vezes é chamada de dobradiça pela maneira como se abre, com dobradiças na lateral. Para ver o que há dentro, vamos começar no lado superior esquerdo. A caixa de metal brilhante no lado superior esquerdo é a fonte de alimentação. Logo abaixo, à esquerda, está o cooler, com sua tampa puxada para trás. À direita e abaixo do cooler, há uma placa de circuito impresso chamada *placa-mãe* em um PC, que contém a maioria dos circuitos eletrônicos do computador, a Figura 1.10 é uma visão em close dessa placa. O processador é o grande retângulo elevado logo à direita do cooler. No lado direito, podemos ver as baias destinadas a abrigar os tipos de unidades de disco. A baia superior contém uma unidade de DVD, a baia do meio, uma unidade Zip e a baia inferior contém uma unidade de disco rígido.

caminho de dados

O componente do processador que realiza operações aritméticas.

controle O componente do processador que comanda o caminho de dados, a memória e os dispositivos de E/S de acordo com as instruções do programa.

RAM dinâmica (DRAM)

Memória construída como um circuito integrado para fornecer acesso aleatório a qualquer local.

memória cache Uma memória pequena e rápida que age como um buffer para uma memória maior e mais lenta.

abstração Um modelo que revela detalhes de nível inferior dos sistemas computacionais temporariamente invisíveis, a fim de facilitar o projeto de sistemas sofisticados.

respondendo respectivamente aos músculos e ao cérebro do processador. O **caminho de dados** realiza as operações aritméticas, e o **controle** diz ao caminho de dados, à memória e aos dispositivos de E/S o que fazer de acordo com as instruções do programa. O Capítulo 5 explica o caminho de dados e o controle para uma implementação simples, e o Capítulo 6 descreve as alterações necessárias para um projeto de desempenho mais alto.

Descer até as profundezas de qualquer componente de hardware revela os interiores da máquina. A memória na Figura 1.10 é construída de chips DRAM. DRAM significa **RAM dinâmica** (Dynamic Random Access Memory). Várias DRAMs são usadas em conjunto para conter as instruções e os dados de um programa. Ao contrário das memórias de acesso seqüencial, como as fitas magnéticas, a parte *RAM* do termo DRAM significa que os acessos à memória levam o mesmo tempo independente da parte da memória lida. Dentro do processador, existe outro tipo de memória – a memória cache. A **memória cache** consiste em uma memória pequena e rápida que age como um buffer para a memória DRAM. (A definição não-técnica de *cache* é um lugar seguro para esconder as coisas.) A cache é construída usando uma tecnologia de memória diferente, a SRAM (RAM estática – Static Random Access Memory). A SRAM é mais rápida, mas menos densa e, portanto, mais cara do que a DRAM.

Você pode ter observado um conceito comum nas descrições de software e de hardware: penetrar nas profundezas do hardware ou software revela mais informações, ou seja, detalhes de nível mais baixo estão ocultos para oferecer um modelo mais simples aos níveis mais altos. O uso dessas camadas, ou **abstrações**, é uma técnica importante para projetar sistemas computacionais extremamente sofisticados.

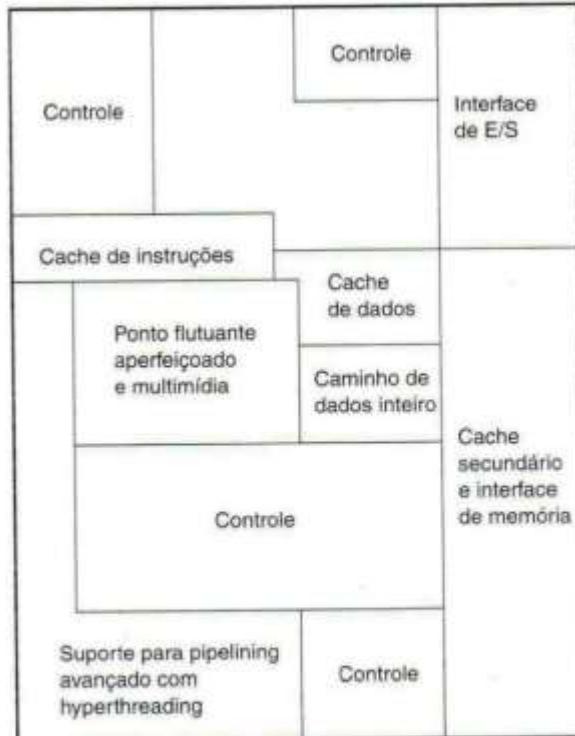
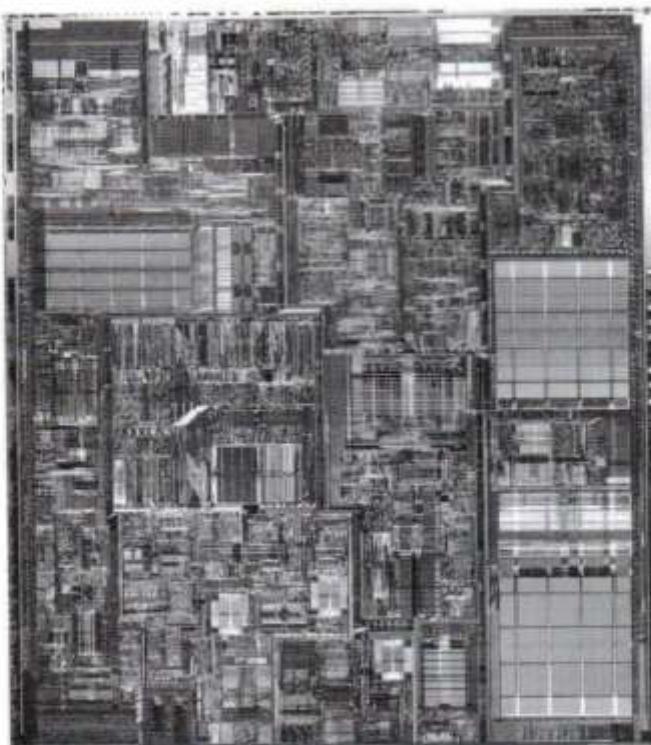


FIGURA 1.9 Dentro do chip de processador usado na placa mostrada na Figura 1.8. O lado esquerdo é uma microfotografia do chip de processador Pentium 4, e o lado direito mostra os blocos principais do processador.

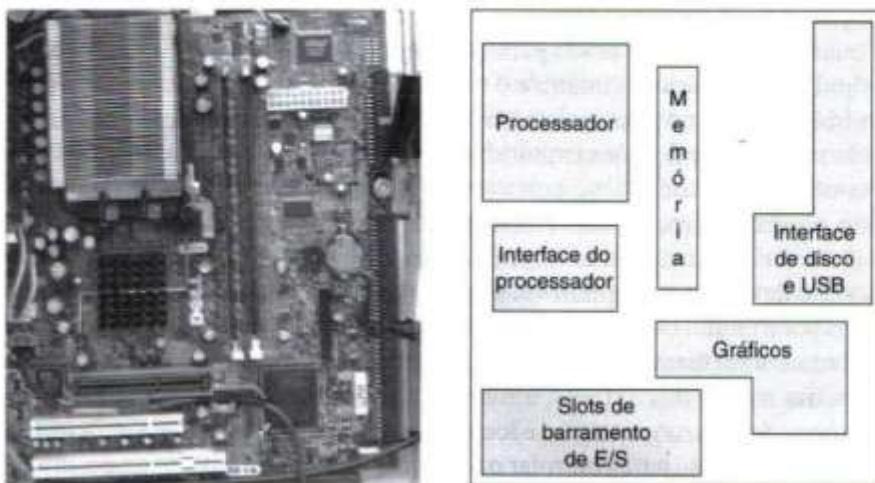


FIGURA 1.10 Close da placa-mãe do PC. Esta placa usa o processador Intel Pentium 4, que está localizado no quadrante superior esquerdo da placa. Ele é coberto por uma série de aletas de metal, semelhante a um radiador de automóvel. Essa estrutura é o *dissipador de calor*, usado para ajudar a resfriar o chip. A memória principal está contida em uma ou mais pequenas placas, perpendiculares à placa-mãe, próximas ao centro. Os chips de DRAM são montados nessas placas (chamadas **DIMMs – Dual Inline Memory Modules**) e, então, ligados aos conectores. Uma grande parte do restante da placa constitui-se de conectores para dispositivos de E/S externos: áudio/MIDI e paralela/serial na lateral direita, dois slots PCI próximos à parte inferior e um conector ATA usado para conectar discos rígidos.

Uma das abstrações mais importantes é a interface entre o hardware e o software de nível mais baixo. Devido à sua importância, ela recebe um nome especial: a **arquitetura do conjunto de instruções**, ou simplesmente **arquitetura**, de uma máquina. A arquitetura do conjunto de instruções inclui tudo que os programadores precisam saber para fazer um programa em linguagem de máquina binária funcionar corretamente, incluindo instruções, dispositivos de E/S etc. Em geral, o sistema operacional encapsulará os detalhes da E/S, da alocação de memória e de outras funções de baixo nível do sistema, para que os programadores das aplicações não precisem se preocupar com esses detalhes. A combinação do conjunto de instruções básico e a interface do sistema operacional fornecida para os programadores das aplicações é chamada de **interface binária de aplicação (ABI)**.

Uma arquitetura do conjunto de instruções permite aos projetistas de computador falarem sobre funções independentes do hardware que as realiza. Por exemplo, podemos falar sobre as funções de um relógio digital (marcar as horas, exibir as horas, definir o alarme) sem falar sobre o hardware do relógio (o cristal de quartzo, os visores de LEDs, os botões plásticos). Os projetistas de computador distinguem entre a arquitetura e uma **implementação** da arquitetura da mesma maneira: uma implementação é o hardware que obedece à abstração da arquitetura. Esses conceitos nos levam a outra seção “Colocando em perspectiva”.

Um lugar seguro para os dados

Até agora, vimos como os dados são inseridos, processados e exibidos. Entretanto, se houvesse uma interrupção no fornecimento de energia, tudo seria perdido porque a **memória** dentro do computador é **volátil** – ou seja, quando perde energia, ela se esquece. Por outro lado, uma fita cassete em um aparelho de som não se esquece da música gravada quando você desliga o aparelho, porque a fita é magnética e, portanto, uma tecnologia de **memória não-volátil**.

DIMM (dual inline memory module)

Uma pequena placa que contém chips DRAM em ambos os lados. Os SIMMs possuem DRAMs em apenas um lado. Tanto os DIMMs quanto os SIMMs devem ser conectados nos slots de memória, normalmente em uma placa-mãe.

arquitetura do conjunto de instruções Também chamada simplesmente de arquitetura. Uma interface abstrata entre o hardware e o software de nível mais baixo de uma máquina que abrange todas as informações necessárias para escrever um programa em linguagem de máquina que será corretamente executado, incluindo instruções, registradores, acesso à memória, E/S e assim por diante.

Interface binária de aplicação (ABI)

A parte voltada ao usuário do conjunto de instruções mais as interfaces do sistema operacional usadas pelos programadores das aplicações. Define um padrão para a portabilidade binária entre computadores.

implementação

Hardware que obedece à abstração de uma arquitetura.

memória

A área de armazenamento onde os programas são mantidos quando estão sendo executados e que contém os dados necessários pelos programas em execução.

memória volátil

Armazenamento, como a DRAM, que conserva os dados apenas enquanto estiver recebendo energia.

memória não-volátil Uma forma de memória que conserva os dados mesmo na ausência de energia e que é usada para armazenar programas entre execuções. O disco magnético é não-volátil, e a DRAM é volátil.

memória principal A memória volátil usada para armazenar os programas enquanto estão sendo executados; normalmente consistindo na DRAM nos computadores atuais.

memória secundária Memória não-volátil usada para armazenar programas e dados entre execuções; normalmente consistindo em discos magnéticos nos computadores atuais.

disco magnético (também chamado de disco rígido) Uma forma de memória secundária não-volátil composta por discos giratórios cobertos com um material de gravação magnético.

megabyte Tradicionalmente, $1,048,576 (2^{20})$ bytes, embora alguns sistemas de comunicações e de armazenamento secundário o tenham definido como $1.000.000 (10^6)$.

Para distinguir entre a memória usada para armazenar os programas enquanto estão sendo executados e essa memória não-volátil usada para armazenar programas entre as execuções, o termo **memória principal** é usado para o primeiro e o termo **memória secundária** é usado para o último. As DRAMs dominam a memória principal desde 1975; e os **discos magnéticos** dominam a memória secundária desde 1965. Em aplicações embutidas, a memória FLASH, uma memória de semicondutor não-volátil, também é utilizada.

Atualmente, o armazenamento não-volátil principal usado em todos os computadores desktop e servidores é o disco rígido magnético. Como mostra a Figura 1.11, um disco rígido magnético consiste em uma série de discos, que giram em torno de um eixo a velocidades que variam entre 5.400 e 15.000 rotações por minuto. Os discos de metal são cobertos por um material de gravação magnético em ambos os lados, semelhante ao material encontrado em uma fita cassete ou de vídeo. Para ler e gravar informações em um disco rígido, um *braço móvel* com um pequena bobina eletromagnética, chamada de *cabeça de leitura/gravação*, é localizada logo acima de cada superfície. A unidade inteira é selada permanentemente para controlar o ambiente dentro da unidade, o que, por sua vez, permite que as cabeças do disco estejam muito mais próximas da superfície da unidade.

Os diâmetros dos discos rígidos atuais variam por um fator de mais de 3, de menos de 2,5cm até 9cm, e têm sido reduzidos ao longo dos anos para se adequarem a novos produtos; servidores, estações de trabalho, computadores pessoais, laptops, palmtops e câmeras digitais têm inspirado novos tamanhos de disco. Tradicionalmente, os discos maiores possuem melhor desempenho, os discos menores possuem o menor custo, e o melhor custo por megabyte normalmente é de um disco com um tamanho intermediário. Embora a maioria dos discos rígidos apareça dentro dos computadores (como na Figura 1.8), os discos rígidos também podem ser conectados usando-se interfaces externas, como Firewire ou USB.

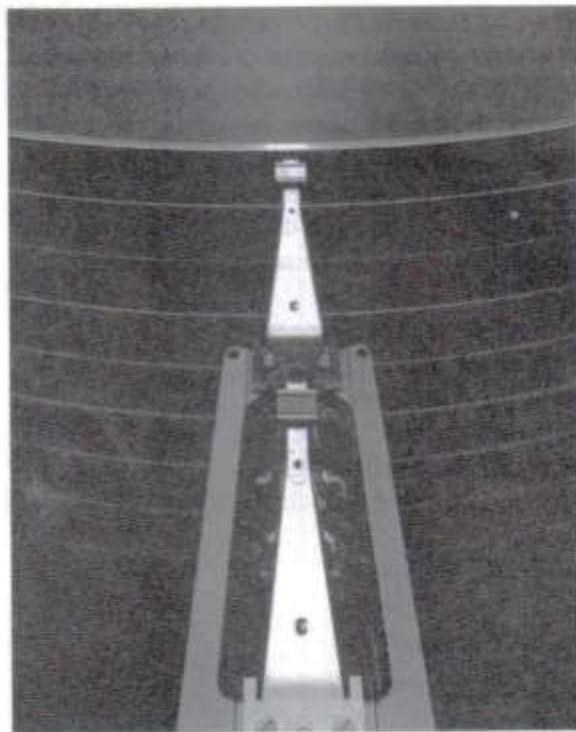


FIGURA 1.11 Uma unidade de disco rígido mostrando 10 discos e as cabeças de leitura/gravação.

O uso de componentes mecânicos significa que os tempos de acesso para os discos magnéticos são muito maiores do que para as DRAMs: os discos em geral levam de 5 a 15 milissegundos, enquanto as DRAMs levam de 40 a 80 nanosegundos – tornando as DRAMs cerca de 100.000 vezes mais rápidas. Entretanto, os discos possuem custos muito mais baixos do que a DRAM para a mesma capacidade de armazenamento, pois os custos de produção para uma determinada quantidade de armazenamento em disco são menores do que para a mesma quantidade de circuito integrado. Em 2004, o custo por megabyte de disco era aproximadamente 100 vezes menor do que o custo da DRAM.

Assim, existem três principais diferenças entre os discos magnéticos e a memória principal: os discos não são voláteis porque são magnéticos; possuem um tempo de acesso maior porque são dispositivos mecânicos; e são mais baratos por megabyte porque possuem capacidade de armazenamento muito alta a um custo razoável.

Tanto o hardware quanto o software são constituídos por camadas hierárquicas, sendo que cada camada inferior oculta detalhes para o nível acima. Esse princípio de *abstração* é a maneira como projetistas de hardware e de software lidam com a complexidade dos sistemas computacionais. Uma importante interface entre os níveis de abstração é a *arquitetura do conjunto de instruções* – a interface entre o hardware e o software de baixo nível. Essa interface abstrata permite que muitas *implementações* de custo e desempenho variáveis executem software idêntico.

Embora os discos rígidos não sejam removíveis, há várias tecnologias de armazenamento em uso que incluem as seguintes:

- Os discos ópticos, incluindo os compact disks (CDs) e digital video disks (DVDs), constituem a forma mais comum de armazenamento removível.
- A fita magnética fornece apenas acesso serial lento e tem sido usada para realização de backups de disco – uma função que, hoje, normalmente está sendo substituída por discos rígidos duplicados.
- As placas de memória removíveis baseadas em FLASH geralmente são conectadas por uma conexão USB (Universal Serial Bus) e muitas vezes são usadas para transferir arquivos.
- As unidades de disco flexível e as unidades Zip são uma versão da tecnologia de disco magnético com discos flexíveis removíveis. Os **discos flexíveis** (ou disquetes) foram originalmente o meio de armazenamento dos computadores pessoais, mas hoje praticamente estão desaparecendo.

A tecnologia de disco ótico funciona de uma maneira completamente diferente da tecnologia de disco magnético. Em um CD, os dados são gravados em espiral, com os bits individuais gravados queimando-se pequenas cavidades – de aproximadamente 1 micron (10^{-6} metros) de diâmetro – na superfície do disco. O disco é lido emitindo-se um laser na superfície do CD e determinando-se, pelo exame da luz refletida, se existe uma cavidade ou uma superfície plana (reflexiva). Os DVDs usam o mesmo método de emissão de um feixe de laser em direção a uma série de cavidades e superfícies planas. Além disso, há diversas camadas em que o feixe de laser pode ser focalizado, e o tamanho de cada cavidade é muito menor, o que, em conjunto, representa um significativo aumento na capacidade.

Os gravadores de CD e DVD nos computadores pessoais usam um laser para criar as cavidades na camada de gravação na superfície do CD ou DVD. Esse processo de gravação é relativamente lento, levando de dezenas de minutos (para um CD inteiro) a cerca de uma hora (para um DVD inteiro). Portanto, para grandes quantidades, é usada uma técnica diferente, chamada *impressão*, que custa apenas alguns centavos por CD ou DVD.

Colocando em perspectiva

disco flexível Uma forma portátil de armazenamento secundário composta por um disco giratório coberto com um material de gravação magnético.

Os CDs e DVDs regraváveis usam uma superfície de gravação diferente que possui um material reflexivo cristalino; as cavidades não-reflexivas são formadas de maneira semelhante ao CD ou DVD de gravação única. Para apagar o CD ou DVD, a superfície é aquecida e resfriada lentamente, permitindo um processo de recocimento para restaurar a camada de gravação da superfície à sua estrutura cristalina original. Esses discos regraváveis são mais caros do que os discos de gravação única; para os discos somente de leitura – usados para distribuir música, software ou filmes – os custos do disco e da gravação são muito menores.

Comunicação com outros computadores

Explicamos como podemos realizar entrada, processamento, exibição e armazenamento de dados, mas ainda falta um item que é encontrado nos computadores modernos: as redes de computadores. Exatamente como o processador mostrado na Figura 1.5 está conectado à memória e aos dispositivos de E/S, as redes conectam computadores inteiros, permitindo que os usuários estendam a capacidade de computação incluindo a comunicação. As redes se tornaram tão comuns que, hoje, constituem o backbone (espinha dorsal) dos sistemas de computação atuais; uma máquina nova sem uma interface de rede opcional seria ridicularizada. Os computadores em rede possuem diversas vantagens importantes:

- **Comunicação:** as informações são trocadas entre computadores em altas velocidades.
- **Compartilhamento de recursos:** em vez de cada máquina ter seus próprios dispositivos de E/S, os dispositivos podem ser compartilhados pelos computadores que compõem a rede.
- **Acesso remoto:** conectando computadores por meio de longas distâncias, os usuários não precisam estar perto do computador que estão usando.

As redes variam em tamanho e desempenho, com o custo da comunicação aumentando de acordo com a velocidade de comunicação e a distância em que as informações viajam. Talvez o tipo de rede mais comum seja a *Ethernet*. Sua extensão é limitada em aproximadamente um quilômetro, e a versão mais popular em 2004 levava cerca de um décimo de segundo para enviar 1 milhão de bytes de dados. Sua extensão e velocidade tornam a Ethernet útil para conectar computadores no mesmo andar de um prédio; portanto, esse é um exemplo do que é chamado genericamente de **rede local (LAN)**. As redes locais são interconectadas com switches que também podem fornecer serviços de roteamento e segurança. As **redes remotas (WAN)** atravessam continentes e são a base da Internet, que suporta a World Wide Web. Elas costumam ser baseadas em cabos de fibra ótica e são alugadas de empresas de telecomunicações.

As redes mudaram a cara da computação nos últimos 25 anos, por se tornarem muito mais comuns e por aumentar dramaticamente o desempenho. Na década de 1970, poucas pessoas tinham acesso a e-mail. A Internet e a Web não existiam, e a remessa física de fitas magnéticas era o meio principal de transferir grandes quantidades de dados entre dois locais. Nessa década, as redes locais eram quase inexistentes, e as poucas redes remotas existentes tinham capacidade limitada e acesso restrito.

À medida que a tecnologia de redes avançou, ela se tornou muito mais barata e obteve uma capacidade de transmissão muito mais alta. Por exemplo, a primeira tecnologia de rede local a ser padronizada, desenvolvida há cerca de 25 anos, foi uma versão da Ethernet que tinha uma capacidade máxima (também chamada de largura de banda) de 10 milhões de bits por segundo, normalmente compartilhada por dezenas, se não centenas, de computadores. Hoje, a tecnologia de rede local oferece uma capacidade de transmissão de 100 milhões de bits por segundo até um gigabit por segundo, em geral compartilhada por, no máximo, alguns computadores. Além disso, a tecnologia de 10 gigabits está em desenvolvimento! A tecnologia de comunicação ótica permitiu um crescimento semelhante na capacidade das redes remotas de centenas de kilobits até gigabits, e de centenas de computadores conectados a uma rede mundial até milhões de computadores conectados. Essa combinação do dramático aumento no emprego das redes e os aumentos em sua capacidade tornaram a tecnologia de redes o ponto nevrálgico para a revolução da informação nos últimos 25 anos.

rede local (LAN) Uma rede projetada para transportar dados dentro de uma área geograficamente restrita, em geral, dentro de um mesmo prédio.

rede remota (WAN) Uma rede estendida por centenas de quilômetros, que pode atravessar continentes.

Recentemente, outra inovação na tecnologia de redes está reformulando a maneira como os computadores se comunicam. A tecnologia sem fio se tornou amplamente utilizada, e a maioria dos laptops hoje incorpora essa tecnologia. A capacidade de criar um rádio com a mesma tecnologia de semicondutor de baixo custo (CMOS) usada para memória e microprocessadores permitiu uma significativa melhoria no preço, levando a uma explosão no consumo. As tecnologias sem fio disponíveis atualmente, conhecidas pelo padrão IEEE 802.11, permitem velocidades de transmissão de 1 a quase 100 milhões de bits por segundo. A tecnologia sem fio é um pouco diferente das redes baseadas em fios, já que todos os usuários em uma área próxima compartilham as ondas aéreas.

1. A DRAM e o armazenamento de disco diferem significativamente. Descreva a principal diferença quanto a cada um dos seguintes aspectos: volatilidade, tempo de acesso e custo.

Verifique você mesmo

Tecnologias para construção de processadores e memórias

Os processadores e a memória melhoraram em uma velocidade espantosa porque os projetistas de computadores, durante muito tempo, abraçaram o que havia de mais moderno na tecnologia eletrônica para tentar vencer a corrida para projetar um computador melhor. A Figura 1.12 mostra as tecnologias usadas ao longo do tempo, com uma estimativa do desempenho relativo por custo unitário para cada tecnologia. Esta seção explora a tecnologia que impulsionou a indústria da computação desde 1975 e continuará a impulsioná-la no futuro próximo. Como essa tecnologia esboça o que os computadores serão capazes de fazer e a velocidade com que irão evoluir, acreditamos que todos os profissionais de computação devem estar familiarizados com os fundamentos dos circuitos integrados.

Ano	Tecnologia usada nos computadores	Desempenho relativo/custo unitário
1951	Válvula	1
1965	Transistor	35
1975	Círculo integrado	900
1995	Círculo VLSI (Very Large Scale Integrated)	2.400.000
2005	Círculo ULSI (Ultra Large Scale Integrated)	6.200.000.000

FIGURA 1.12 Desempenho relativo por custo unitário das tecnologias usadas nos computadores ao longo do tempo. Fonte: Computer Museum, Boston, com o ano de 2005 estimado pelos autores.

Um transistor é simplesmente uma chave liga/desliga controlada por eletricidade. O *círculo integrado (IC)* combinou dezenas a centenas de transistores em um único chip. Para descrever o incrível aumento no número de transistores de centenas para milhões, o adjetivo *escala muito grande* é acrescentado ao termo, criando a abreviação *VLSI* (de **Very Large Scale Integrated**).

Essa taxa de integração crescente tem se mantido notavelmente estável. A Figura 1.13 mostra o crescimento na capacidade da DRAM desde 1977. Durante 20 anos, a indústria quadruplicou consistentemente a capacidade a cada três anos, resultando em um aumento de mais de 16.000 vezes! Esse aumento no número de transistores para um círculo integrado é popularmente conhecido como a Lei de Moore, que diz que a capacidade em transistores dobra a cada 18 a 24 meses. A Lei de Moore resultou de uma previsão desse crescimento na capacidade do círculo integrado feita por Gordon Moore, um dos fundadores da Intel durante a década de 1960.

Sustentar essa taxa de progresso por quase 40 anos exigiu incríveis inovações nas técnicas de fabricação. Na Seção 1.4, discutimos como os circuitos integrados são fabricados.

válvula Um componente eletrônico, predecessor do transistor, que consiste em um tubo de vidro oco de aproximadamente 5 a 10 centímetros de comprimento do qual o máximo de ar foi removido e que usa um feixe de elétrons para transferir dados.

transistor Uma chave liga/desliga controlada por um sinal elétrico.

círculo VLSI (Very Large Scale Integrated) Um dispositivo com centenas de milhares a milhões de transistores.

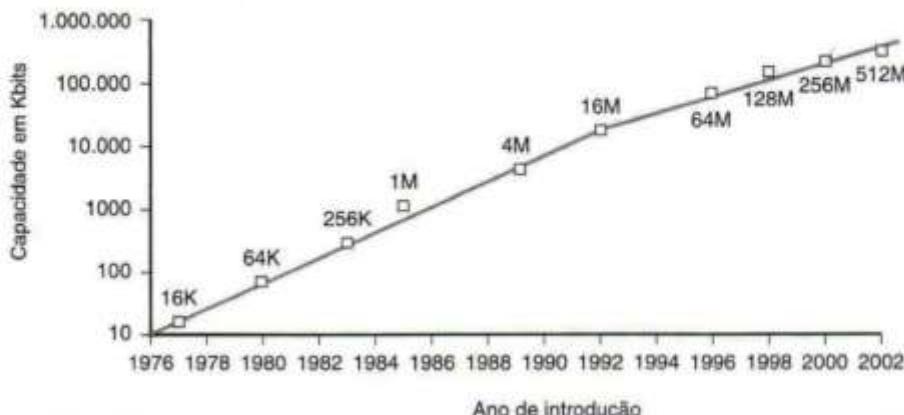


FIGURA 1.13 Crescimento da capacidade por chip de DRAM ao longo do tempo. O eixo y é medido em Kbits, onde $K = 1024 (2^{10})$. A indústria de DRAM quadruplicou a capacidade a cada quase 3 anos, um aumento de 60% por ano, durante 20 anos. Essa estimativa de “quatro vezes a cada três anos” foi conhecida como a *regra do crescimento da DRAM*. Nos últimos anos, essa taxa diminuiu um pouco e está próximo do dobro a cada dois anos.

Eu acreditava que os computadores seriam uma idéia universalmente aplicável, assim como os livros. Só não imaginava que se desenvolveriam tão rapidamente, pois não pensei que fôssemos capazes de colocar tantas peças em um chip quanto finalmente colocamos. O transistor apareceu inesperadamente. Tudo aconteceu muito mais rápido do que esperávamos.

J. Presper Eckert, co-inventor do Eniac, falando em 1991

silício Um elemento natural que é um semicondutor.

semicondutor Uma substância que não conduz bem a eletricidade.

lingote de cristal de silício Uma barra composta de um cristal de silício que possui entre 15 e 30cm de diâmetro e cerca de 30 a 60cm de comprimento.

wafer Uma fatia de um lingote de silício de não mais de 0,25cm de espessura, usada para criar chips.

1.4

Vida real: fabricando chips de Pentium 4

Cada capítulo possui uma seção intitulada “Vida Real”, que associa os conceitos no livro com um computador que você pode usar em seu dia-a-dia. Essas seções abordam a tecnologia na qual se baseia o IBM PC, o Apple Macintosh, um servidor comum ou um computador embutido. Nesta primeira “Vida Real”, veremos como os circuitos integrados são fabricados, tendo o Pentium 4 como exemplo.

Vamos começar do inicio. A fabricação de um chip começa com o **silício**, uma substância encontrada na areia. Como não é um bom condutor de eletricidade, é chamado de **semicondutor**. Com um processo químico especial, é possível acrescentar ao silício materiais que permitem que minúsculas áreas se transformem em um entre três dispositivos:

- Excelentes condutores de eletricidade (usando fios microscópicos de cobre ou alumínio)
- Excelentes isolantes de eletricidade (como cobertura plástica ou vidro)
- Áreas que podem conduzir *ou* isolar sob condições especiais (como uma chave)

Os transistores se encaixam na última categoria. Um circuito VLSI, então, simplesmente consiste em bilhões de combinações de condutores, isolantes e chaves, fabricados em um único e pequeno pacote.

O processo de fabricação dos circuitos integrados é decisivo para o custo dos chips e, consequentemente, fundamental para os projetistas de computador. A Figura 1.14 mostra esse processo. O processo inicia com um **lingote de cristal de silício**, que se parece com uma salsicha gigante. Hoje, os lingotes possuem de 20 a 30cm de diâmetro e cerca de 30 a 60cm de comprimento. Um lingote é finalmente fatiado em **wafers** de não mais que 0,25cm de espessura. Esses wafers passam por uma série de etapas de processamento, durante as quais são depositados padrões de elementos químicos em cada lâmina, criando os transistores, os condutores e os isolantes discutidos anteriormente. Os circuitos integrados de hoje contêm apenas uma camada de transistores, mas podem ter de dois a oito níveis de condutor de metal, separados por camadas de isolantes.

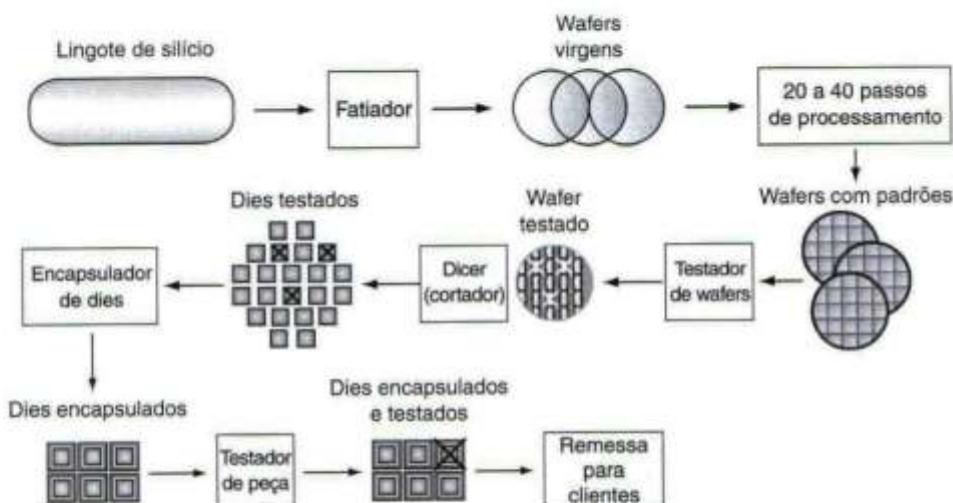


FIGURA 1.14 Processo de fabricação de um chip. Após ser fatiado de um lingote de silício, os wafers virgens passam por 20 a 40 passos para criar wafers com padrões (veja a Figura 1.15). Esses wafers com padrões são testados com um testador de wafers e é criado um mapa das partes boas. Depois, os wafers são divididos em dies (moldes) (veja a Figura 1.9). Nessa figura, um wafer produziu 20 dies, dos quais 17 passaram no teste. (X significa que o die está ruim.) O aproveitamento de dies bons nesse caso foi de 17/20, ou 85%. Esses dies bons são soldados a encapsulamentos e testados outra vez antes de serem remetidos para os clientes. Um die encapsulado ruim foi encontrado nesse teste final.

Uma única imperfeição microscópica no wafer propriamente dito ou em uma das dezenas de passos da aplicação dos padrões pode resultar na falha dessa área do wafer. Esses **defeitos**, como são chamados, tornam praticamente impossível fabricar um wafer perfeito. Para lidar com a imperfeição, várias estratégias têm sido usadas, mas a mais simples é colocar muitos componentes independentes em um único wafer. O wafer com os padrões é, então, cortado em seções individuais desses componentes, chamados **dies**, mais informalmente conhecidos como **chips**. A Figura 1.15 é uma fotografia de um wafer com microprocessadores Pentium 4 antes de serem cortados; anteriormente, a Figura 1.9 mostrou um die individual do Pentium 4 e seus principais componentes.

Cortar os wafers em seções permite descartar apenas aqueles dies que possuem falhas, em vez do wafer inteiro. Esse conceito é quantificado pelo **aproveitamento** de um processo, definido como a porcentagem de dies bons do número total de dies em um wafer.

O custo de um circuito integrado sobe rapidamente conforme aumenta o tamanho do die, devido ao aproveitamento mais baixo e ao menor número de dies que podem caber em um wafer. Para reduzir o custo, um die grande normalmente é “encolhido” usando um processo da próxima geração, que incorpora tamanhos menores de transistores e de fios. Isso melhora o aproveitamento e o número de dies por wafer.

Tendo dies bons, eles são conectados aos pinos de entrada/saída de um encapsulamento usando um processo chamado *soldagem*. Essas peças encapsuladas são testadas uma última vez, já que podem ocorrer erros no encapsulamento, e são remetidas para os clientes.

Outra limitação de projeto cada vez mais importante é o consumo. O consumo é um problema por duas razões. Primeiro, a corrente precisa ser trazida para o chip e distribuída por toda sua área; os microprocessadores modernos usam centenas de pinos apenas para alimentação e aterramento! Da mesma forma, múltiplos níveis de interconexões são usados unicamente para distribuição de corrente e aterramento para as partes do chip. Segundo, a energia é dissipada como calor e precisa ser removida. Um Intel Pentium 4 a 3.06GHz produz 82 watts, que precisam ser removidos de um chip cuja área de superfície é de apenas 1cm²! A Figura 1.16 mostra um Pentium 4 a 3.06GHz, montado sobre seu dissipador de calor, que, por sua vez, se encontra imediatamente ao lado do cooler no gabinete mostrado na Figura 1.8.

defeito Uma imperfeição microscópica em um wafer ou nos passos da aplicação dos padrões que pode resultar na falha do die que contém esse defeito.

dies As seções retangulares individuais cortadas de um wafer, mas informalmente conhecidos como chips.

aproveitamento A porcentagem de dies bons do número total de dies em um wafer.



Figura 1.15 Um wafer de 200mm de diâmetro com processadores Intel Pentium 4. O número de dies de Pentium por wafer em 100% de aproveitamento é 165. A Figura 1.9 é uma microfotografia de um desses dies de Pentium 4. A área do die é de 250mm^2 e contém cerca de 55 milhões de transistores. Esse die usa uma tecnologia de 0,18 micron, o que significa que os menores transistores possuem um tamanho de aproximadamente 0,18 micron, embora normalmente sejam um pouco menores do que o tamanho real catalogado, que se refere ao tamanho dos transistores como “desenhados” versus o tamanho final fabricado. As várias dezenas de chips parcialmente arredondados nas bordas do wafer são inúteis; são incluídas porque é mais fácil criar as máscaras usadas para imprimir os padrões desejados ao silício.

O que determina a potência consumida por um circuito integrado? Ignorando os fatores de tecnologia e circuito, a potência é proporcional ao produto do número de transistores pela frequência com que são chaveados. Portanto, em geral, velocidades de clock mais altas ou números de transistores mais altos ocasionam maior consumo de energia. Por exemplo, o Intel Itanium 2 possui quatro vezes o número de transistores do Intel Pentium 4; embora sua velocidade de clock seja apenas a metade do Pentium 4, o Itanium produz 130 watts comparados com os 82 watts produzidos pelo Pentium 4. Como veremos em capítulos posteriores, tanto o desempenho quanto o consumo de energia podem variar significativamente.

Detalhamento: em CMOS (Complementary Metal Oxide Semiconductor), a tecnologia dominante para os circuitos integrados, a principal fonte de dissipação de potência, é a chamada “potência dinâmica” – ou seja, a energia consumida durante o chaveamento dos transistores. A tecnologia CMOS, ao contrário das tecnologias anteriores, não consome energia diretamente quando está ociosa – daí o uso de baixas velocidades de clock para permitir que um processador “durma” e poupe energia. A dissipação de potência dinâmica depende da carga capacitiva de cada transistor, da voltagem aplicada e da frequência com que o transistor é chaveado:

$$\text{Potência} = \text{Carga capacitiva} \times \text{Voltagem}^2 \times \text{Frequência de chaveamento}$$

A potência pode ser reduzida diminuindo a voltagem, o que geralmente ocorre com uma nova geração de tecnologia; em 20 anos, as voltagens caíram de 5V para 1,5V, reduzindo drasticamente a potência. A carga capacitativa por transistor é uma função do número de transistores conectados a uma saída (chamada *fan-out*) e da tecnologia, que determina a capacidade dos fios e dos transistores.

Embora a potência dinâmica seja a principal fonte de dissipação de potência no CMOS, a dissipação de potência estática ocorre devido ao vazamento de corrente existente mesmo quando um transistor está desligado. Em 2004, o vazamento talvez fosse responsável por 20 a 30% do consumo de energia. Assim, aumentar o número de transistores aumenta a dissipação de potência, mesmo que os transistores estejam sempre desligados. Diversas técnicas de projeto e inovações tecnológicas têm sido implementadas para controlar o vazamento.

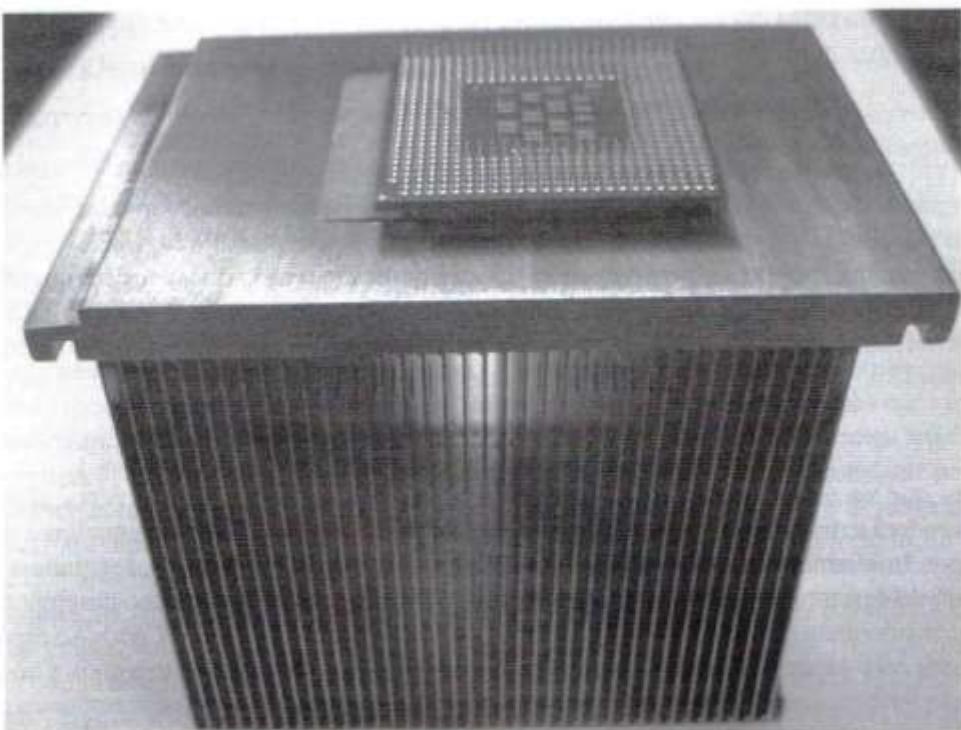


FIGURA 1.16 Um Intel Pentium 4 a 3.06GHz, montado sobre seu dissipador de calor, projetado para remover os 82 watts gerados dentro do die.

Um fator fundamental para determinar o custo de um circuito integrado é o volume de produção. Quais das seguintes afirmativas são razões para um chip fabricado com alto volume de produção custar menos?

Verifique você mesmo

1. Com altos volumes de produção, o processo de fabricação pode ser transformado em um projeto particular, aumentando o aproveitamento.
2. É menos trabalhoso projetar uma peça com alto volume de produção do que uma com baixo volume de produção.
3. Como as máscaras usadas para fabricar o chip são caras, o custo por chip é menor para volumes de produção mais altos.
4. Os custos de desenvolvimento de engenharia são altos e quase sempre independem do volume de produção; portanto, o custo de desenvolvimento por die é menor com peças de alto volume de produção.
5. Peças de alto volume de produção normalmente possuem dies menores do que as peças de baixo volume de produção e, portanto, têm um aproveitamento mais alto por wafer.

1.5

Falácia e armadilhas

A finalidade de uma seção de falácia e armadilhas, que será incluída em cada capítulo, é explicar alguns conceitos errôneos comuns que você pode encontrar. Chamamos esses equívocos de *falácia*. Quando estivermos discutindo uma falácia, tentaremos fornecer um contra-exemplo. Também discutiremos *armadilha*, ou erros facilmente cometidos. Em geral, as armadilhas são generalizações

A ciência deve começar com os mitos e com a análise dos mitos.

Sir Karl Popper,
The Philosophy of Science, 1957

de princípios verdadeiros em um contexto restrito. O propósito dessas seções é ajudar a evitar esses erros nas máquinas que você pode projetar ou usar.

Falácia: os computadores têm sido construídos da mesma maneira por muito tempo, e esse modelo antiquado de computação está perdendo a força.

Para um modelo antiquado de computação, ele sem dúvida está melhorando rapidamente. A Figura 1.17 indica o melhor desempenho por ano das estações de trabalho entre 1987 e 2003. (O Capítulo 4 explica a maneira certa de medir o desempenho.) O gráfico mostra uma linha que indica uma melhoria de 1,54 por ano, ou o dobro do desempenho aproximadamente a cada 18 meses. Ao contrário da afirmação anterior, os computadores estão melhorando seu desempenho mais rapidamente hoje do que em qualquer momento de sua história, com uma melhoria de mais de mil vezes entre 1987 e 2003!

Armadilha: ignorar o inevitável progresso do hardware ao planejar uma nova máquina.

Suponha que você planeje introduzir uma máquina no mercado em três anos. Você alega que a máquina venderá extraordinariamente porque ela é três vezes mais rápida do que qualquer uma disponível hoje. Infelizmente, é provável que sua máquina não venda muito, pois, seguindo a taxa de crescimento do desempenho médio, a indústria produzirá máquinas com o mesmo desempenho. Por exemplo, considerando uma taxa de crescimento anual de 50% no desempenho, pode-se esperar que uma máquina com desempenho x hoje tenha um desempenho $1,5^3x = 3,4x$ em três anos. Ou seja, sua máquina não teria qualquer vantagem de desempenho! Muitos projetos dentro das empresas de computador são cancelados porque ignoram essa regra ou porque o projeto é concluído com atraso e o desempenho da máquina obsoleta está abaixo na média da indústria. Esse fenômeno pode ocorrer em qualquer setor, mas as rápidas melhorias no custo/desempenho o tornam um importante problema na indústria da computação.

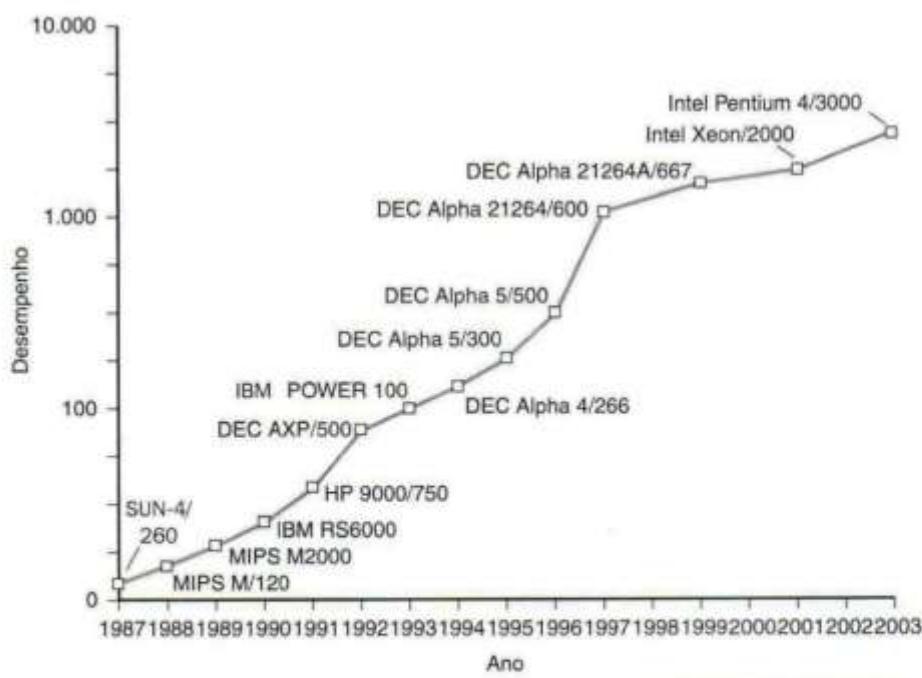


FIGURA 1.17 Aumento no desempenho das estações de trabalho entre 1987 e 2003. Aqui, o desempenho é fornecido aproximadamente como o número de vezes que o computador é mais rápido do que o VAX-11/780, que é um padrão comumente usado. O índice de melhoria de desempenho está entre 1,5 e 1,6 vezes por ano. Esses números de desempenho baseiam-se na execução do SPECint (consulte o Capítulo 2) e escalado ao longo do tempo para resolver as alterações do *benchmark*. Para os processadores assinalados com x/y ao lado do nome, x é o modelo e y é a velocidade em megahertz.



1.6

Comentários finais

Embora seja difícil prever exatamente o nível de custo/desempenho que os computadores terão no futuro, é seguro dizer que serão muito melhores do que são hoje. Para participar desses avanços, os projetistas e programadores de computador precisam entender uma grande variedade de questões.

Os projetistas de hardware e de software constroem sistemas computacionais em camadas hierárquicas, sendo que cada camada inferior oculta seus detalhes do nível acima. Esse princípio de abstração é fundamental para compreender os sistemas computacionais atuais, mas isso não significa que os projetistas podem se limitar a conhecer uma única tecnologia. Talvez o exemplo mais importante de abstração seja a interface entre hardware e software de baixo nível, chamada *arquitetura do conjunto de instruções*. Manter a arquitetura do conjunto de instruções como uma constante permite que muitas implementações dessa arquitetura – provavelmente variando em custo e desempenho – executem software idêntico. Como o lado negativo, a arquitetura pode impedir a introdução de inovações que exijam a mudança da interface.

As tecnologias vitais para os processadores modernos são os compiladores e o silício. Claramente, para participar, você precisa entender algumas das características de ambos. De igual importância para uma compreensão da tecnologia de circuito integrado é o conhecimento das taxas de mudança tecnológica esperadas. Enquanto o silício impulsiona o rápido avanço do hardware, novas idéias na organização dos computadores melhoraram seu custo/desempenho. Duas das principais idéias são a exploração do paralelismo no processador, normalmente por meio de pipelining, e a exploração da localidade dos acessos a uma hierarquia de memória, em geral por meio de caches.

Mapa para este livro

Na base dessas abstrações, estão os cinco componentes clássicos de um computador: caminho de dados, controle, memória, entrada e saída (veja novamente a Figura 1.5). Esses cinco componentes também servem de estrutura para os demais capítulos deste livro:

- *Caminho de dados*: Capítulos 3, 5 e 6
- *Controle*: Capítulos 5 e 6
- *Memória*: Capítulo 7
- *Entrada*: Capítulo 8
- *Saída*: Capítulo 8

O Capítulo 6 descreve como o pipelining no processador explora o paralelismo, e o Capítulo 7 descreve como a hierarquia de memória explora a localidade. Os demais capítulos fornecem a introdução e a conclusão para esse assunto. O Capítulo 2 descreve os conjuntos de instruções – a interface entre os compiladores e a máquina – e destaca o papel dos compiladores e das linguagens de programação ao usar os recursos do conjunto de instruções. O Capítulo 3 descreve como os computadores realizam operações aritméticas e manipulam dados aritméticos. O Capítulo 4 aborda o desempenho e, portanto, descreve como avaliar o computador como um todo. O Capítulo 9 descreve os multiprocessadores e é incluído no CD. O Apêndice B, também no CD, discute o projeto lógico.

Enquanto o
Eniac é equipado
com 18.000
válvulas
e pesa 30
toneladas, os
computadores no
futuro poderão
ter 1.000
válvulas e talvez
pesar apenas 1,5
tonelada.

Popular
Mechanics, março
de 1949

Um campo ativo da ciência é como um imenso formigueiro; a pessoa quase desaparece na massa de mentes afundando umas sob as outras, carregando informações de um lugar para outro, passando-as adiante na velocidade da luz.

Lewis Thomas,
"Natural Science",
em *The Lives of a Cell*, 1974

1.7

Perspectiva histórica e leitura adicional

Para cada capítulo, há uma seção dedicada à perspectiva histórica que pode ser encontrada no CD que acompanha este livro. Podemos traçar o desenvolvimento de uma idéia por meio de uma série de máquinas ou descrever alguns projetos importantes; e fornecemos referências, caso você esteja interessado em pesquisar mais a fundo.

A perspectiva histórica para este capítulo fornece uma base para algumas das principais idéias apresentadas neste capítulo de abertura. Sua finalidade é apresentar a história humana por trás dos avanços tecnológicos e colocar as realizações dentro de seu contexto histórico. Entendendo o passado, você pode compreender melhor as forças que formarão a computação no futuro. Cada seção de perspectiva histórica no CD termina com sugestões para leitura adicional. O restante desta ■ Seção 1.7 está no CD.

1.8

Exercícios

As avaliações do tempo relativo à solução dos exercícios são mostradas entre colchetes após cada número de exercício. Em média, um exercício avaliado em [10] levará o dobro do tempo de um avaliado em [5]. As seções do texto que devem ser lidas antes de resolver um exercício serão indicadas entre sinais de maior e menor; por exemplo, <§1.3> significa que você deve ler a Seção 1.3, "Sob as tampas", para ajudar a resolver esse exercício. Se a solução de um exercício depender de outros, eles serão indicados entre chaves; por exemplo, {Ex. 1.1} significa que você deve responder ao Exercício 1.1 antes de tentar resolver este exercício.

A partir do Capítulo 2, você também encontrará exercícios "Aprofundando o aprendizado". Esses exercícios incluem problemas adicionais elaborados para que o leitor interessado adquira mais prática no tratamento de um assunto. Os exercícios "Aprofundando o aprendizado" foram retirados principalmente de edições anteriores deste livro, bem como de exercícios desenvolvidos por outros instrutores. As Seções "Aprofundando o aprendizado" aparecem no CD associadas ao capítulo específico.

Exercícios de 1.1 a 1.28 Encontre a palavra ou frase da seguinte lista que melhor corresponde à descrição nas questões a seguir. Use os números à esquerda das palavras na resposta. Cada resposta deve ser usada apenas uma vez.

1	abstração	15	sistema embutido
2	montador	16	instrução
3	bit	17	arquitetura de conjunto de instruções
4	cache	18	rede local (LAN)
5	CPU (unidade central de processamento)	19	memória
6	chip	20	sistema operacional
7	compilador	21	semicondutor
8	família de computadores	22	servidor
9	controle	23	supercomputador
10	caminho de dados	24	transistor
11	computador desktop ou computador pessoal (PC)	25	VLSI (Very Large Scale Integrated Circuit)
12	DVD (Digital Video Disk)	26	wafer
13	defeito	27	rede remota (WAN)
14	DRAM (Dynamic Random Access Memory – RAM dinâmica)	28	aproveitamento

1.1 [2] Parte ativa do computador, que segue estritamente as instruções dos programas. Soma e testa números, controla outros componentes etc.

1.2 [2] Método para o projeto do hardware ou software. O sistema consiste em camadas hierárquicas, com cada camada inferior ocultando detalhes do nível acima.

1.3 [2] Dígito binário.

- 1.4** [2] Coleção de implementações da mesma arquitetura de conjunto de instruções. Normalmente são criados(as) pela mesma empresa e variam em preço e desempenho.
- 1.5** [2] Componente do computador onde residem todos os programas em execução e os dados associados.
- 1.6** [2] Componente do processador que realiza operações aritméticas.
- 1.7** [2] Componente do processador que diz ao caminho de dados, à memória e aos dispositivos de E/S o que fazer de acordo com as instruções do programa.
- 1.8** [2] Computador projetado para uso por uma pessoa, normalmente incorporando um monitor gráfico, um teclado e um mouse.
- 1.9** [2] Computador dentro de outro dispositivo, usado para executar uma aplicação predeterminada ou um grupo de softwares.
- 1.10** [2] Computador usado para executar grandes programas para múltiplos usuários normalmente de forma simultânea e em geral acessado apenas por meio da rede.
- 1.11** [2] Rede de computadores que conecta um grupo de computadores por um cabo de transmissão comum ou um link sem fio dentro de uma pequena área geográfica (por exemplo, dentro do mesmo andar de um prédio).
- 1.12** [2] Redes de computadores que conectam computadores por meio de grandes distâncias, a espinha dorsal (backbone) da Internet.
- 1.13** [2] Máquina de alto desempenho, custando mais de US\$1 milhão.
- 1.14** [2] Circuito integrado comumente usado para construir a memória principal.
- 1.15** [2] Falha microscópica em um wafer.
- 1.16** [2] Apelido para um die ou circuito integrado.
- 1.17** [2] Chave liga/desliga controlada por eletricidade.
- 1.18** [2] Meio de armazenamento ótico com uma capacidade de armazenamento de mais de 4,7GB. Foi comercializado(a) inicialmente para entretenimento e, mais tarde, para usuários de computador.
- 1.19** [2] Porcentagem de dies bons do número total de dies no wafer.
- 1.20** [2] Programa que converte uma versão simbólica de uma instrução na versão binária.
- 1.21** [2] Programa que controla os recursos de um computador para o benefício dos programas executados nessa máquina.
- 1.22** [2] Programa que traduz de uma notação de alto nível para assembly.
- 1.23** [2] Tecnologia em que um único chip contém centenas de milhares a milhões de transistores.
- 1.24** [2] Um único comando de software para um processador.
- 1.25** [2] Memória pequena e rápida que age como um buffer para a memória principal.
- 1.26** [2] Interface específica que o hardware fornece ao software de baixo nível.
- 1.27** [2] Substância que não conduz bem a eletricidade mas é a base dos circuitos integrados.
- 1.28** [2] Disco fino fatiado de um lingote de cristal de silício, que, posteriormente, será dividido em dies.

Exercícios de 1.29 a 1.45 Usando as categorias na lista a seguir, classifique os seguintes exemplos. Use as letras à esquerda das palavras na resposta. Diferente dos exercícios anteriores, as respostas neste grupo podem ser usadas mais de uma vez.

a	aplicações	f	computador pessoal
b	linguagem de programação de alto nível	g	semicondutor
c	dispositivo de entrada	h	supercomputador
d	circuito integrado	i	software de sistemas
e	dispositivo de saída		

- 1.29** [1] Montador
- 1.30** [1] C++
- 1.31** [1] Monitor de cristal líquido (LCD)

1.32 [1] Compilador

1.33 [1] Cray-1

1.34 [1] DRAM

1.35 [1] IBM PC

1.36 [1] Java

1.37 [1] Scanner

1.38 [1] Macintosh

1.39 [1] Microprocessador

1.40 [1] Microsoft Word

1.41 [1] Mouse

1.42 [1] Sistema operacional

1.43 [1] Impressora

1.44 [1] Silício

1.45 [1] Planilha

1.46 [15] <§1.3> Em um disco magnético, os discos com os dados estão constantemente girando. Em média, deve ser necessária meia rotação para que os dados desejados no disco se posicionem abaixo da cabeça de leitura/gravação. Considerando que o disco esteja girando a 7.200 rotações por minuto (RPM), qual é o tempo médio para que os dados se posicionem abaixo da cabeça do disco? Qual é o tempo médio se o disco estiver girando a 10.000RPM?

1.47 [5] <§1.3> Uma unidade de DVD, entretanto, funciona no modo CLV (Constant Linear Velocity). A cabeça de leitura precisa interagir com os círculos concêntricos em uma velocidade linear constante, quer ela esteja acessando os dados a partir das áreas mais internas ou mais externas do disco. Isso é afetado variando a velocidade de rotação do disco, de 1.600RPM no centro a 570RPM na extremidade. Considerando que a unidade de DVD lê 1,35MB de dados do usuário por segundo, quantos bytes o círculo central pode armazenar? Quantos bytes o círculo externo pode armazenar?

1.48 [5] <§1.3> Se um computador realiza 30 requisições a rede por segundo e cada requisição possui em média 64KB, um link Ethernet de 100Mbit será suficiente?

1.49 [5] <§1.3> Que tipos de redes você usa regularmente? Que tipos de mídia elas usam? Quanta largura de banda fornecem?

1.50 [15] <§1.3> O atraso de ponta a ponta é uma importante medição para as redes. Ele é o tempo decorrido entre o momento em que a origem começa a enviar dados e o momento em que os dados são completamente entregues no destino. Considere dois hosts A e B, conectados por um único link de velocidade R bps (bits por segundo). Suponha que os dois hosts estejam separados por m metros e que a velocidade de propagação ao longo do link seja de 5m/s. O host A está enviando um arquivo do tamanho de L bits para o host B.

- Obtenha uma expressão para o atraso ponta a ponta em função de R , L , m e s .
- Suponha que haja um roteador entre A e B, e os dados de A precisem ser encaminhados para B pelo roteador. Se o processo de forwarding leva t segundos, então, qual é o atraso ponta a ponta?
- Suponha que o roteador esteja configurado para fornecer QoS (qualidade de serviço) para diferentes tipos de dados. Se os dados forem um fluxo de multimídia, como dados de uma videoconferência, eles serão encaminhados em uma demora menor do que $t/2$ segundos. Para outros tipos de dados, a demora é t segundos. Se um host A estiver enviando um fluxo de multimídia de tamanho $2L$, qual é o atraso ponta a ponta?

1.51 [15] <§§1.4, 1.5> Considere que você esteja em uma empresa que venderá um certo chip. Os custos fixos, incluindo pesquisa & desenvolvimento, fabricação e equipamentos, totalizam US\$500.000. O custo por wafer é de US\$6.000 e cada wafer pode ser dividido em 1.500 dies. O aproveitamento do wafer é de 50%. Finalmente, os dies são encapsulados e testados, com um custo de US\$10 por chip. O aproveitamento de teste é de 90%; apenas as matrizes que passam no teste serão enviadas para os clientes. Se o preço de varejo é 40% maior do que o custo, quantos chips no mínimo precisam ser vendidos para alcançar o ponto de equilíbrio?

1.52 [8] <§1.6> Neste exercício, você avaliará a diferença de desempenho entre duas arquiteturas de CPU, a CISC (Complex Instruction Set Computing) e a RISC (Reduced Instruction Set Computing). Genericamente falando, as CPUs CISC possuem instruções mais complexas do que as CPUs RISC e, portanto, precisam de menos instruções para realizar as mesmas tarefas. Entretanto, em geral uma instrução CISC, por ser mais complexa, leva mais tempo para ser completada do que uma instrução RISC. Considere que uma determinada tarefa precise de P instruções CISC e $2P$ instruções RISC, e que uma instrução CISC leva $8T$ ns (nanossegundos) para ser completada enquanto uma instrução RISC leva $2T$ ns. Com essas considerações, qual arquitetura apresenta o melhor desempenho?

1.53 [15] <§§1.3, 1.6> Considere cinco computadores interconectados para formar uma rede local. A velocidade de transferência de dados (largura de banda) máxima que o cabo de rede pode fornecer é de 10Mbps (megabits por segundo). Se utilizarmos um dispositivo de baixa capacidade (hub) para conectá-los, todos os computadores na rede compartilharão a largura de banda de 10Mbps. Se usarmos um dispositivo de alta capacidade (switch), então, qualquer par de computadores poderá se comunicar um com o outro sem afetar os outros computadores. Se você quiser baixar um arquivo de 10MB de um servidor remoto que esteja localizado fora de sua rede local, quanto tempo levará se usar um hub? Quanto tempo levará se usar um switch? Considere que os outros quatro computadores se comunicam apenas uns com os outros e cada um possui uma velocidade de transferência de dados constante de 2Mbps.

1.54 [8] <§1.6> Algumas vezes, a otimização pode melhorar significativamente o desempenho de um sistema computacional. Considere que uma CPU pode realizar uma operação de multiplicação em 10ns e uma operação de subtração em 1ns. Quanto tempo a CPU levará para calcular o resultado da equação $d = a \cdot b - a \cdot c$? Você poderia otimizar a equação de modo que ela levasse menos tempo?

1.55 [8] <§§1.1-1.5> Este livro aborda as abstrações para sistemas computacionais em muitos níveis de detalhe diferentes. Escolha outro sistema com o qual você esteja familiarizado e escreva um ou dois parágrafos descrevendo alguns dos muitos níveis diferentes de abstração inerentes a esse sistema. Algumas possibilidades são automóveis, casas, aviões, geometria, a economia, uma cidade e o governo. Não se esqueça de identificar as abstrações de alto e baixo nível.

1.56 [15] <§§1.1-1.5> Um amigo com pouca inclinação técnica pediu que você explicasse como os computadores funcionam. Escreva uma descrição detalhada de uma página para seu amigo.

1.57 [10] <§§1.1-1.5> Em que aspectos você carece de um claro entendimento de como os computadores funcionam? Existem níveis de abstração que lhe sejam particularmente estranhos? Existem níveis de abstração que lhe sejam familiares, mas que ainda deixam dúvidas? Escreva pelo menos um parágrafo tratando cada uma dessas questões.

1.58 [15] <§1.3> Neste exercício, você aprenderá mais sobre interfaces ou abstrações. Por exemplo, você pode fornecer uma abstração para um disco desta maneira:

Características de desempenho:

- Capacidade (quantos dados ele pode armazenar?)
- Largura de banda (a que velocidade os dados podem ser transferidos entre o computador e o disco?)
- Latência (quanto tempo é necessário para encontrar uma posição específica para o acesso?)

Funções que a interface oferece:

- Ler/gravar dados
- Procurar uma posição específica
- Relatório de status (o disco está pronto para ler/gravar etc.?)

Seguindo este padrão, forneça uma abstração para uma placa de rede.

§1.1: página 8, Questões para discussão: muitas respostas são aceitáveis.

§1.3: página 21, Memória em disco: não-volátil, tempo de acesso longo (milissegundos), e custo de US\$2 a US\$4 por GB. Memória usando semicondutores: volátil, tempo de acesso curto (nanossegundos) e custo de US\$200 a US\$400 por GB.

§1.4: página 25, 1, 3 e 4 são razões válidas.

**Respostas
das Seções
“Verifique
você mesmo”**

Computadores no mundo real

Tecnologia da informação para os 4 bilhões sem TI

Em todo este livro, você verá seções intituladas “Computadores no mundo real”. Essas seções descrevem usos incontestáveis para os computadores fora de suas funções típicas na automação de escritório e no processamento de dados. O objetivo dessas seções é ilustrar a diversidade dos usos para a tecnologia da informação.

Problema a ser resolvido: torne a tecnologia da informação disponível para o restante da humanidade, como fazendeiros em vilarejos rurais, além de um conjunto de caracteres multilingüe como o Unicode.

Solução: desenvolva um computador, um software e um sistema de comunicação para um vilarejo rural. Entretanto, não há eletricidade, telefone, suporte técnico e os moradores não falam inglês.

A Jhai Foundation aceitou esse desafio para cinco vilarejos no distrito de Hin Heup no Laos. Essa fundação Lao-American foi criada para elevar o padrão de vida do Laos rural desenvolvendo uma economia de exportação. Ele também construiu escolas, instalou reservatórios e come-

çou a criar uma cooperativa. Quando perguntados o que queriam em seguida, os moradores disseram que queriam ter acesso à Internet! Primeiro, queriam conhecer os preços antes de levarem suas colheitas para o mercado mais próximo, que ficava a 35 quilômetros. Também poderiam aprender sobre o mercado em geral para tomar melhores decisões sobre o que as colheitas se tornam e para aumentar seu poder de barganha na hora de vendê-las. Segundo, queriam usar a telefonia via Internet para falar com parentes no Laos e de fora.

O objetivo era “um computador e uma impressora rudimentares, montados a partir de componentes caseiros, que consomem menos de 20 watts em uso normal – menos de 70 watts quando a impressora está em operação – e que possa resistir à poeira, calor e imersão em água.”

O projeto Jhai PC resultante usa memória FLASH em vez de uma unidade de disco, eliminando, assim, as partes móveis do PC para torná-lo mais rude e fácil de manter. Em vez de usar um tubo de raios catódicos faminto de energia, ele usa um monitor de cristal líquido. Para reduzir os custos e o consumo de energia, ele usa



Um morador laociano que queria acesso à Internet.

um microprocessador 80486. A energia é fornecida por uma bateria de automóvel, que pode ser carregada pedalando uma bicicleta. Uma antiga impressora matricial completa o hardware, trazendo o custo para cerca de US\$400. O sistema operacional é o Linux, e as aplicações são contabilidade, e-mail e processamento de texto, que os estrangeiros estão adaptando ao idioma do Laos.

A solução de comunicação é adaptar a rede sem fio WiFi (IEEE 802.11b) (consulte o Capítulo 8). O plano é reforçar o sinal usando antenas maiores e, então, instalar estações repetidoras nos topo dos montes entre o vilarejo e a cidade. Essas repetidoras são alimentadas por energia solar. O sistema de telefonia local é ligado a elas na outra extremidade, completando a conexão com a Internet. Vinte e cinco voluntários no Vale do Sílico estão desenvolvendo essa rede Jhai PC.

Uma alternativa é o *simputer*, que significa "simple, inexpensive, multilingual computer –

computador simples, barato e multilingüe". Os cientistas da computação indianos projetaram esse PDA, que é semelhante ao Palm Pilot, para atender às necessidades dos habitantes dos países do Terceiro Mundo. A entrada é por meio de uma tela de toque e do reconhecimento de fala, de modo que as pessoas não precisam saber escrever para usá-lo. Ele usa três pilhas AAA, que duram de 3 a 4 horas. O custo é de US\$250 e não há uma solução especial para comunicação. Não se sabe ao certo se os habitantes dos países do Terceiro Mundo gastariam US\$250 em um PDA, onde até mesmo as baterias são supérfluas.

Para aprender mais, veja estas referências

- “Making the Web world-wide”, *The Economist*, 26 de setembro de 2002, www.jhai.org/economist
- The Jhai Foundation, www.jhai.org/
- “Computers for the Third World”, *Scientific American*, outubro de 2002



Habitante indiano usando o simputer.

2

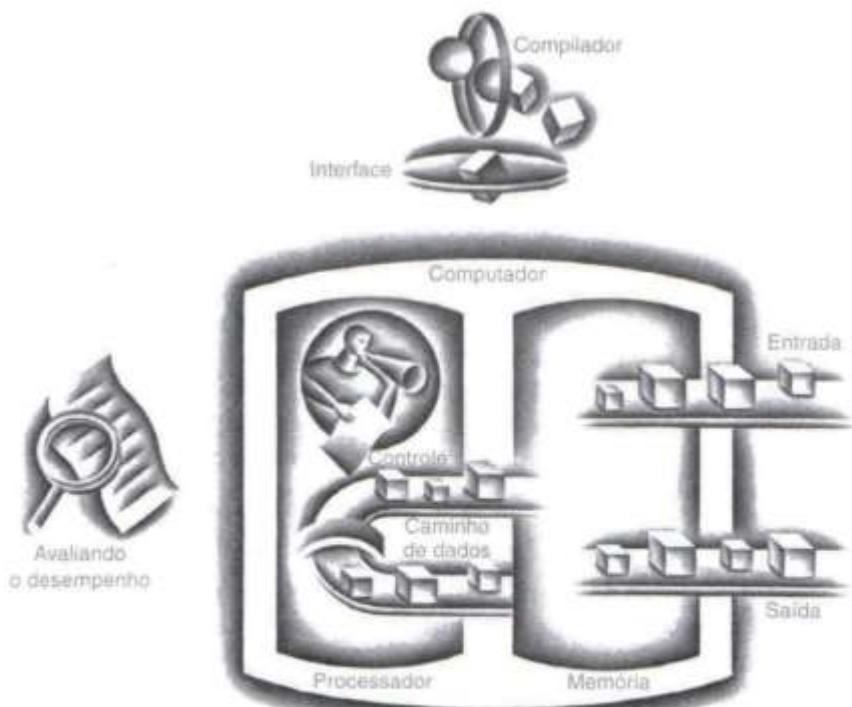
Instruções: A Linguagem de Máquina

*Eu falo espanhol com Deus,
italiano com as mulheres,
francês com os homens
e alemão com meu cavalo.*

Charles V, rei da França
1337-1380

2.1	Introdução	36
2.2	Operações do hardware do computador	37
2.3	Operandos do hardware do computador	39
2.4	Representando instruções no computador	44
2.5	Operações lógicas	50
2.6	Instruções para tomada de decisões	52
2.7	Suporte a procedimentos no hardware do computador	58
2.8	Comunicando-se com as pessoas	66
2.9	Endereçamento no MIPS para operandos imediatos e endereços de 32 bits	70
2.10	Traduzindo e iniciando um programa	78
2.11	Como os compiladores otimizam	85
■ 2.12	Como os compiladores funcionam: uma introdução	89
2.13	Um exemplo de ordenação em C para juntar tudo isso	89
■ 2.14	Implementando uma linguagem orientada a objetos	96
2.15	Arrays versus ponteiros	96
2.16	Vida real: instruções do IA-32	99
2.17	Falácia e armadilhas	106
2.18	Comentários finais	107
■ 2.19	Perspectiva histórica e leitura adicional	109
2.20	Exercícios	110

Os cinco componentes clássicos de um computador



conjunto de instruções O vocabulário dos comandos entendidos por uma determinada arquitetura.

2.1 Introdução

Para controlar o hardware de um computador, é preciso falar sua linguagem. As palavras da linguagem de um computador são chamadas *instruções*, e seu vocabulário é denominado **conjunto de instruções**. Neste capítulo, você verá o conjunto de instruções de um computador real, tanto na forma escrita pelos humanos quanto na forma lida pelo computador. Apresentamos as instruções em um padrão *top-down*. Começando com uma notação parecida com uma linguagem de programação restrita, nós a refinamos passo a passo, até que você veja a linguagem real de um computador. O Capítulo 3 continua nossa descida, expondo a representação dos números inteiros e de ponto flutuante e o hardware que os opera.

Você poderia pensar que as linguagens dos computadores fossem tão diversificadas quanto as dos humanos, mas, na realidade, as linguagens de computador são muito semelhantes, mais parecidas com dialetos regionais do que linguagens independentes. Logo, quando você aprender uma, será fácil entender as outras. Essa semelhança ocorre porque todos os computadores são construídos a partir de tecnologias de hardware baseadas em princípios básicos semelhantes e porque existem algumas operações básicas que todos os computadores precisam oferecer. Além do mais, os projetistas de computador possuem um objetivo comum: encontrar uma linguagem que facilite o projeto do hardware e do compilador enquanto maximiza o desempenho e minimiza o custo. Esse objetivo é antigo; a citação a seguir foi escrita antes que você pudesse comprar um computador e é tão verdadeira hoje quanto era em 1947:

É fácil ver, por métodos lógicos formais, que existem certos [conjuntos de instruções] que são adequados para controlar e causar a execução de qualquer sequência de operações... As considerações realmente decisivas, do ponto de vista atual, na seleção de um [conjunto de instruções], são mais de natureza prática: a simplicidade do equipamento exigido pelo [conjunto de instruções] e a clareza de sua aplicação para os problemas realmente importantes, junto com a velocidade com que tratam esses problemas.

Burks, Goldstine e von Neumann, 1947

A “simplicidade do equipamento” é uma consideração tão valiosa para os computadores da década iniciada no ano 2000 quanto foi para os da década de 1950. O objetivo deste capítulo é ensinar um conjunto de instruções que siga esse conselho, mostrando como ele é representado no hardware e o relacionamento entre as linguagens de programação de alto nível e essa linguagem mais primitiva. Nossos exemplos estão na linguagem de programação C; a Seção 2.14 mostra como esses exemplos mudariam para uma linguagem orientada a objetos, como Java.

Aprendendo como representar as instruções, você também descobrirá o segredo da computação: o **conceito de programa armazenado**. Além disso, exercitará suas habilidades com “linguagem estrangeira”, escrevendo programas na linguagem do computador e executando-os no simulador que acompanha livro. Você também verá o impacto das linguagens de programação e das otimizações do compilador sobre o desempenho. Concluímos com uma visão da evolução histórica dos conjuntos de instruções e uma visão geral dos outros dialetos do computador.

O conjunto de instruções escolhido vem do MIPS, que é um dos típicos conjuntos de instruções criados desde a década de 1980. Quase 100 milhões desses microprocessadores populares foram fabricados em 2002, e são encontrados em produtos da ATI Technologies, Broadcom, Cisco, NEC, Nintendo, Silicon Graphics, Sony, Texas Instruments e Toshiba, entre outros.

Revelamos o conjunto de instruções do MIPS aos poucos, mostrando o raciocínio juntamente com as estruturas do computador. Esse tutorial *top-down*, passo a passo, entrelaça os componentes com suas explicações, tornando o assembly mais fácil de digerir. Para ter em mente o quadro geral, cada seção termina com uma figura que resume o conjunto de instruções do MIPS revelado até aqui, destacando as partes apresentadas nessa seção.

conceito de programa armazenado A idéia de que as instruções e os dados de muitos tipos podem ser armazenados na memória como números, levando ao computador de programa armazenado.

2.2

Operações do hardware do computador

Todo computador precisa ser capaz de realizar aritmética. A notação em assembly do MIPS

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

instrui um computador a somar as duas variáveis b e c para colocar sua soma em a.

Essa notação é rígida no sentido de que cada instrução aritmética do MIPS realiza apenas uma operação e sempre precisa ter exatamente três variáveis. Por exemplo, suponha que queiramos colocar a soma das variáveis b, c, d e e na variável a. (Nesta seção, estamos sendo deliberadamente vagos com relação ao que é uma “variável”; na próxima seção, vamos explicar com detalhes.)

A seqüência de instruções a seguir soma as quatro variáveis:

```
add a, b, c      # A soma b + c é colocada em a.  
add a, a, d      # A soma b + c + d agora está em a.  
add a, a, e      # A soma b + c + d + e agora está em a.
```

Assim, são necessárias três instruções para realizar a soma de quatro variáveis.

As palavras à direita do símbolo # em cada uma dessas linhas são *comentários* para o leitor humano, e o computador as ignora. Observe que, diferente de outras linguagens de programação, cada linha dessa linguagem pode conter no máximo uma instrução. Outra diferença de C é que os comentários sempre terminam no final de uma linha.

O número natural de operandos para uma operação como a adição é três: os dois números sendo somados e um local para colocar a soma. Exigir que cada instrução tenha exatamente três operações, nem mais nem menos, está de acordo com a filosofia de manter o hardware simples: o hardware para um número variável de operandos é mais complicado do que o hardware para um número fixo. Essa situação ilustra o primeiro dos quatro princípios básicos de projeto do hardware:

Princípio de Projeto 1: simplicidade favorece a regularidade

Agora podemos mostrar, nos dois exemplos a seguir, o relacionamento dos programas escritos nas linguagens de programação de mais alto nível com os programas nessa notação mais primitiva.

COMPILANDO DUAS INSTRUÇÕES DE ATRIBUIÇÃO C NO MIPS

Este segmento de um programa em C contém as cinco variáveis a, b, c, d e e. Como o Java evoluiu a partir da linguagem C, este exemplo e os próximos funcionam para qualquer uma dessas linguagens de programação de alto nível:

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

A tradução de C para as instruções em linguagem assembly do MIPS é realizada pelo *compilador*. Mostre o código do MIPS produzido por um compilador.

Uma instrução MIPS opera com dois operandos de origem e coloca o resultado em um operando de destino. Logo, as duas instruções simples anteriores são compiladas diretamente nessas duas instruções em assembly do MIPS:

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

Certamente é preciso haver instruções para realizar as operações aritméticas fundamentais.

Burks, Goldstine e

von Neumann,

1947

EXEMPLO

RESPOSTA

COMPILANDO UMA ATRIBUIÇÃO C COMPLEXA NO MIPS**EXEMPLO**

Uma instrução um tanto complexa contém as cinco variáveis f, g, h, i e j:

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

O que um compilador C poderia produzir?

RESPOSTA

O compilador precisa desmembrar essa instrução em várias instruções assembly, pois somente uma operação é realizada por instrução MIPS. A primeira instrução MIPS calcula a soma de g e h. Temos de colocar o resultado em algum lugar, de modo que o compilador crie uma variável temporária, chamada t0:

```
add t0,g,h # variável temporária t0 contém g + h
```

Embora a próxima operação seja subtrair, precisamos calcular a soma de i e j antes de podermos subtrair. Assim, a segunda instrução coloca a soma de i e j em outra variável temporária criada pelo compilador, chamada t1:

```
add t1,i,j # variável temporária t1 contém i + j
```

Finalmente, a instrução de subtração subtrai a segunda soma da primeira e coloca a diferença na variável f, completando o código compilado:

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

A Figura 2.1 resume as partes do assembly do MIPS descritas nesta seção. Essas instruções são representações simbólicas daquilo que o processador MIPS realmente entende. Nas próximas seções, vamos evoluir essa representação simbólica para a linguagem real do MIPS, com cada etapa tornando a representação simbólica mais concreta.

Assembly do MIPS

Categoria	Instrução	Exemplo	Significado	Comentários
Aritmética	Add	add a,b,c	a = b + c	Sempre três operandos
	Subtract	sub a,b,c	a = b - c	Sempre três operandos

FIGURA 2.1 Arquitetura MIPS revelada até a Seção 2.2. Os operandos reais do computador serão revelados na próxima seção. As partes destacadas nesses resumos mostram as estruturas do assembly do MIPS introduzidas nesta seção; para esta primeira figura, tudo é novo.

Verifique você mesmo

Para determinada função, que linguagem de programação provavelmente utiliza mais linhas de código? Coloque as três representações a seguir em ordem.

1. Java
2. C
3. Assembly do MIPS

Detalhamento: para aumentar a portabilidade, Java foi idealizada originalmente contando com um interpretador de software. O conjunto de instruções desse interpretador é chamado *bytecode Java*, que é muito diferente do conjunto de instruções do MIPS. Para chegar a um desempenho próximo ao programa em C equivalente, os sistemas Java de hoje normalmente compilam os bytecodes Java para os conjuntos de instruções nativos, como MIPS. Como essa compilação em geral é feita muito mais tarde do que para programas C, esses compiladores Java normalmente são denominados compiladores JIT (*Just-In-Time* – na hora exata). A Seção 2.10 mostra como os JITs são usados mais tarde que os compiladores C no processo de inicialização, e a Seção 2.13 mostra as consequências no desempenho de compilar versus interpretar programas Java. Os exemplos em Java neste capítulo pulam a etapa do bytecode Java e mostram simplesmente o código MIPS produzido por um compilador.

2.3

Operandos do hardware do computador

Ao contrário dos programas nas linguagens de alto nível, os operandos das instruções aritméticas são restritos, precisam ser de um grupo limitado de locais especiais, embutidos diretamente no hardware, chamados *registradores*. Os registradores são os tijolos da construção do computador: os registradores são primitivas usadas no projeto do hardware que também são visíveis ao programador quando o computador é completado. O tamanho de um registrador na arquitetura MIPS é de 32 bits; os grupos de 32 bits ocorrem com tanta frequência que recebem o nome de **word** (palavra) na arquitetura MIPS.

Uma diferença importante entre as variáveis de uma linguagem de programação e os registradores é o número limitado de registradores, normalmente 32 nos computadores atuais. O MIPS possui 32 registradores. (Consulte a Seção 2.19 para ver a história do número de registradores.) Assim, continuando em nossa evolução *top-down*, passo a passo, da representação simbólica da linguagem MIPS, nesta seção, incluímos a restrição de que cada um dos três operandos das instruções aritméticas do MIPS precisa ser escolhido a partir de um dos 32 registradores de 32 bits.

O motivo para o limite dos 32 registradores pode ser encontrado no segundo dos quatro princípios de projeto básicos da tecnologia de hardware:

Princípio de Projeto 2: menor significa mais rápido.

Uma quantidade muito grande de registradores pode aumentar o tempo do ciclo do clock simplesmente porque os sinais eletrônicos levam mais tempo quando precisam atravessar uma distância maior.

Orientações como “menor significa mais rápido” não são absolutas; 31 registradores podem não ser mais rápidos do que 32. Mesmo assim, a verdade por trás dessas observações faz com que os projetistas de computador as levem a sério. Nesse caso, o projetista precisa equilibrar o desejo dos programas por mais registradores com o desejo do projetista de manter o ciclo de clock rápido. Outro motivo para não usar mais de 32 é o número de bits que seria necessário no formato da instrução, como demonstra a Seção 2.4.

Os Capítulos 5 e 6 mostram o papel central que os registradores desempenham na construção do hardware; como veremos neste capítulo, o uso eficaz dos registradores é fundamental para o desempenho do programa.

Embora pudéssemos simplesmente escrever instruções usando números para os registradores, de 0 a 31, a convenção do MIPS é usar nomes com um sinal de cifrão seguido por dois caracteres para representar um registrador. A Seção 2.7 explicará os motivos por trás desses nomes. Por enquanto, usaremos \$s0, \$s1,... para os registradores que correspondem às variáveis dos programas em C e Java, e \$t0, \$t1,... para os registradores temporários necessários para compilar o programa nas instruções MIPS.

word A unidade de acesso natural de um computador, normalmente um grupo de 32 bits; corresponde ao tamanho de um registrador na arquitetura MIPS.

COMPILANDO UMA ATRIBUIÇÃO EM C USANDO REGISTRADORES

É tarefa do compilador associar variáveis do programa aos registradores. Considere, por exemplo, a instrução de atribuição do nosso exemplo anterior:

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

As variáveis f, g, h, i e j são associadas aos registradores \$s0, \$s1, \$s2, \$s3 e \$s4, respectivamente. Qual é o código MIPS compilado?

O programa compilado é muito semelhante ao exemplo anterior, exceto que substituimos as variáveis pelos nomes dos registradores mencionados anteriormente, mais dois registradores temporários, \$t0 e \$t1, que correspondem às variáveis temporárias de antes:

```
add $t0,$s1,$s2 # registrador $t0 contém g + h
add $t1,$s3,$s4 # registrador $t1 contém i + j
sub $s0,$t0,$t1 # f recebe $t0 - $t1, que é (g + h)-(i + j)
```

EXEMPLO

RESPOSTA

Operandos em memória

As linguagens de programação possuem variáveis simples, que contêm elementos de dados isolados, como nesses exemplos, mas também possuem estruturas de dados mais complexas. Essas estruturas de dados complexas podem conter muito mais elementos de dados do que a quantidade de registradores em um computador. Logo, como um computador pode representar e acessar estruturas tão grandes?

Lembre-se dos cinco componentes de um computador, apresentados no Capítulo 1 e desenhados na abertura do Capítulo 2. O processador só pode manter uma pequena quantidade de dados nos registradores, mas a memória do computador contém milhões de elementos de dados. Logo, as estruturas de dados (arrays e estruturas) são mantidas na memória.

Conforme explicamos, as operações aritméticas só ocorrem com registradores nas instruções MIPS; assim, o MIPS precisa incluir instruções que transferem dados entre a memória e os registradores. Essas instruções são denominadas **instruções de transferência de dados**. Para acessar uma word na memória, a instrução precisa fornecer o **endereço** de memória. A memória é apenas uma sequência grande e unidimensional, com o endereço atuando como índice para esse array, começando de 0. Por exemplo, na Figura 2.2, o endereço do terceiro elemento de dados é 2, e o valor de Memória[2] é 10.

A instrução de transferência de dados que copia dados da memória para um registrador tradicionalmente é chamada de *load*. O formato da instrução load é o nome da operação seguido pelo registrador a ser carregado, depois uma constante e o registrador usado para acessar a memória. A soma da parte constante da instrução com o conteúdo do segundo registrador forma o endereço da memória. O nome MIPS real para essa instrução é *lw*, significando *load word* (carregar palavra).

instrução de transferência de dados Um comando que move dados entre a memória e os registradores.

endereço Um valor usado para delinear o local de um elemento de dados específico dentro de uma sequência da memória.

COMPILANDO UMA ATRIBUIÇÃO QUANDO UM OPERANDO ESTÁ NA MEMÓRIA

EXEMPLO

Vamos supor que A seja uma sequência de 100 words e que o compilador tenha associado as variáveis g e h aos registradores \$s1 e \$s2, como antes. Vamos supor também que o endereço inicial da sequência, ou *endereço base*, esteja em \$s3. Compile esta instrução de atribuição em C:

g = h + A[8];

RESPOSTA

Embora haja uma única operação nessa instrução de atribuição, um dos operandos está na memória, de modo que primeiro precisamos transferir A[8] para um registrador. O endereço desse elemento da sequência é a soma da base da sequência A, encontrada no registrador \$s3, com o número para selecionar o elemento 9. Os dados devem ser colocados em um registrador temporário, para uso na próxima instrução. Com base na Figura 2.2, a primeira instrução compilada é

lw \$t0,8(\$s3) # Registrador temporário \$t0 recebe A[8]

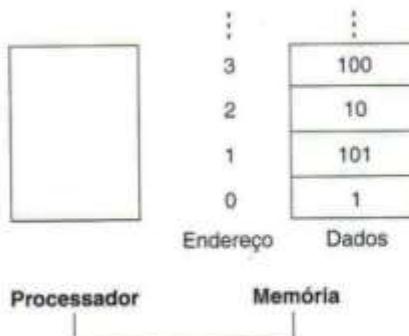


FIGURA 2.2 Endereços de memória e conteúdo da memória nesses locais. Essa é uma simplificação do endereçamento no MIPS; a Figura 2.3 mostra o endereçamento MIPS real para endereços de words sequenciais na memória.

(A seguir, faremos um pequeno ajuste nessa instrução, mas usaremos essa versão simplificada por enquanto.) A seguinte instrução pode operar sobre o valor em \$t0 (que é igual a A[8]), já que está em um registrador. A instrução precisa somar h (contido em \$s2) com A[8] (\$t0) e colocar a soma no registrador correspondente a g (associado a \$s1):

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

A constante na instrução de transferência de dados é chamada de *offset*, e o registrador acrescentado para formar o endereço é chamado de *registraror base*.

Interface hardware/software

Além de associar variáveis a registradores, o compilador aloca estruturas de dados, como arrays e estruturas, em locais na memória. O compilador pode, então, colocar o endereço inicial apropriado nas instruções de transferência de dados.

Como os bytes de 8 bits são úteis em muitos programas, a maioria das arquiteturas endereça bytes individuais. Portanto, o endereço de uma word combina os endereços dos 4 bytes dentro da word. Logo, os endereços seqüenciais das words diferem em 4 vezes. Por exemplo, a Figura 2.3 mostra os endereços MIPS reais para a Figura 2.2; o endereço em bytes da terceira word é 8.

No MIPS, words precisam começar em endereços que sejam múltiplos de 4. Esse requisito é denominado **restrição de alinhamento**, e muitas arquiteturas têm. (O Capítulo 5 explica por que o alinhamento ocasiona transferências de dados mais rápidas.)

Os computadores se dividem naqueles que utilizam o endereço do byte mais à esquerda, ou *big end*, como endereço da word e aqueles que utilizam o byte mais à direita, ou *little end*. O MIPS está no campo do *Big Endian*. (O Apêndice A mostra as duas opções para numerar os bytes de uma word.)

O endereçamento em bytes também afeta o índice do array. Para obter o endereço em bytes apropriado no código anterior, o offset a ser somado ao registrador base \$s3 precisa ser 4×8 , ou 32, de modo que o endereço de load selecione A[8], e não A[8/4]. (Veja a armadilha relacionada na Seção 2.17.)

restrição de alinhamento Um requisito de que os dados estejam alinhados na memória em limites naturais.

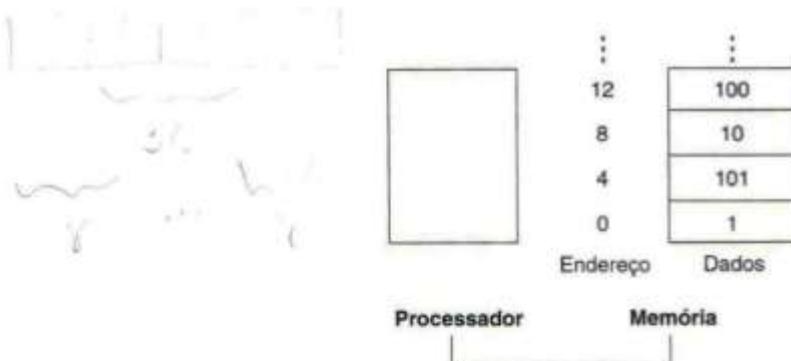


FIGURA 2.3 Endereços de memória reais do MIPS e conteúdo da memória para essas words. Os endereços alterados são destacados por comparação com a Figura 2.2. Como o MIPS endereça cada byte, os endereços em words são múltiplos de quatro: existem quatro bytes em uma word.

A instrução complementar ao load tradicionalmente é chamada de *store*; ela copia dados de um registrador para a memória. O formato de um store é semelhante ao de um load: o nome da operação, seguido pelo registrador a ser armazenado, depois o offset para selecionar o elemento do array e finalmente o registrador base. Mais uma vez, o endereço MIPS é especificado em parte por uma constante e em parte pelo conteúdo de um registrador. O nome real no MIPS é SW, significando *store word* (armazena palavra).

EXEMPLO**COMPILEANDO COM LOAD E STORE**

Suponha que a variável *h* esteja associada ao registrador \$s2, e o endereço base do array A esteja em \$s3. Qual é o código assembly do MIPS para a instrução de atribuição em C a seguir?

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

RESPOSTA

Embora haja uma única operação na instrução em C, agora dois dos operandos estão na memória, de modo que precisamos de ainda mais instruções MIPS. As duas primeiras instruções são iguais às do exemplo anterior, exceto que, dessa vez, usamos o offset apropriado para o endereçamento do byte na instrução load word para selecionar A[8], e a instrução add coloca a soma em \$t0:

```
lw    $t0,32($s3)  # Registrador temporário $t0 recebe A[8]
add  $t0,$s2,$t0  # Registrador temporário $t0 recebe h + A[8]
```

A instrução final armazena a soma em A[12], usando 48 como offset e o registrador \$s3 como registrador base.

```
sw    $t0,48($s3)  # Armazena h + A[8] de volta em A[12]
```

Constantes ou operandos imediatos

Muitas vezes, um programa usará uma constante em uma operação – por exemplo, ao incrementar um índice a fim de apontar para o próximo elemento de um array. Na verdade, mais da metade das instruções aritméticas do MIPS possuem uma constante como operando quando executam os *benchmarks* SPEC2000.

Interface hardware/software

Muitos programas possuem mais variáveis do que os computadores possuem registradores. Em consequência disso, o compilador tenta manter as variáveis mais utilizadas nos registradores e coloca as restantes na memória, usando loads e stores para mover variáveis entre os registradores e a memória. O processo de colocar variáveis menos utilizadas (ou aquelas que serão necessárias mais tarde) na memória é denominado *spilling registers*.

O princípio de hardware que relaciona tamanho e velocidade sugere que a memória deverá ser mais lenta do que os registradores, pois os registradores são menores. Isso realmente acontece; os acessos aos dados são mais rápidos se estiverem nos registradores, em vez de na memória.

Além do mais, os dados são mais úteis quando estão em um registrador. Uma instrução aritmética MIPS pode ler dois registradores, atuar sobre eles e escrever o resultado. Uma instrução de transferência de dados MIPS só lê um operando ou escreve um operando, sem atuar sobre ele.

Assim, os registradores MIPS levam menos tempo para serem acessados e possuem maior vazão do que a memória – uma combinação rara –, tornando os dados nos registradores mais rápidos de acessar e mais simples de usar. Para conseguir o melhor desempenho, os compiladores precisam usar os registradores de modo eficaz.

Usando apenas as instruções vistas até aqui, teríamos de ler uma constante da memória para utilizá-la. (As constantes teriam de ser colocadas na memória quando o programa fosse carregado.) Por exemplo, para somar a constante 4 ao registrador \$s3, poderíamos usar o código

```
lw    $t0, EndConstante4($s1)  # $t0 = constante 4
add  $s3,$s3,$t0                # $s3 = $s3 + $t0 ($t0 == 4)
```

supondo que EndConstante4 seja o endereço de memória da constante 4.

Operandos MIPS

Nome	Exemplo	Comentários
32 registradores	\$s0, \$s1, ..., \$t0, \$t1, ...	Locais rápidos para dados. No MIPS, os dados precisam estar em registradores para a realização de operações aritméticas.
2^{32} words na memória	Memória[0], Memória[4], ..., Memória[4294967292]	Acessadas apenas por instruções de transferência de dados no MIPS. O MIPS utiliza endereços em bytes, de modo que os endereços em words sequenciais diferem em 4 vezes. A memória contém estruturas de dados, arrays e splitted registers.

Linguagem assembly do MIPS

Categoría	Instrucción	Exemplo	Significado	Comentários
Aritmética	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Três operandos; dados nos registradores
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Três operandos; dados nos registradores
	add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	Usada para somar constantes
Transferência de dados	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memória}[\$s2 + 100]$	Dados da memória para o registrador
	store word	sw \$s1,100(\$s2)	$\text{Memória}[\$s2 + 100] = \$s1$	Dados do registrador para a memória

FIGURA 2.4 Arquitetura MIPS revelada até a Seção 2.3. As partes destacadas mostram as estruturas do assembly MIPS introduzidas na Seção 2.3.

Uma alternativa que evita a instrução load é oferecer versões das instruções aritméticas em que o operando seja uma constante. Essa instrução add rápida, com uma constante no lugar do operando, é chamada *add imediato*, ou addi. Para somar 4 ao registrador \$s3, simplesmente escrevemos

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

As instruções imediatas ilustram o terceiro princípio de projeto do hardware, mencionado inicialmente na Seção “Falácia e armadilhas”, do Capítulo 1:

Princípio de Projeto 3: agilize os casos mais comuns.

Os operandos constantes ocorrem com freqüência e, incluindo constantes dentro das instruções aritméticas, elas são muito mais rápidas do que se as constantes fossem lidas da memória.

A Figura 2.4 resume as partes da representação simbólica do conjunto de instruções MIPS descrito nesta seção. Load word e store word são as instruções que copiam words entre a memória e os registradores na arquitetura MIPS. Outros modelos de computadores utilizam instruções junto com load e store para transferir dados. Uma arquitetura com tais alternativas é a Intel IA-32, descrita na Seção 2.16.

Dada a importância dos registradores, qual é a taxa de aumento no número de registradores em um chip com o passar do tempo?

Verifique você mesmo

1. Muito rápida: eles aumentam tão rapidamente quanto a Lei de Moore, o que prevê o dobro do número de transistores em um chip a cada 18 meses.
2. Muito lenta: como os programas normalmente são distribuídos em linguagem de máquina, existe uma inércia na arquitetura do conjunto de instruções, e, por isso, o número de registradores aumenta apenas quando novos conjuntos de instruções se tornam viáveis.

Detalhamento: embora os registradores MIPS neste livro tenham 32 bits de largura, existe uma versão de 64 bits do conjunto de instruções MIPS, definido com 32 registradores de 64 bits. Para distingui-los, eles são chamados oficialmente de MIPS-32 e MIPS-64. Neste capítulo, usamos um subconjunto do MIPS-32. O Apêndice D mostra as diferenças entre MIPS-32 e MIPS-64.

O endereçamento formado pelo registrador base mais o offset do MIPS é uma combinação excelente para as estruturas e os arrays, pois o registrador pode apontar para o início da estrutura, e o offset pode selecionar o elemento desejado. Veremos esse exemplo na Seção 2.13.

O registrador nas instruções de transferência de dados foi criado originalmente para manter o índice do array com o offset utilizado para o endereço inicial do array. Assim, o registrador base também é chamado *registrador índice*. As memórias de hoje são muito maiores, e o modelo de software para alocação de dados é mais sofisticado, de modo que o endereço base do array normalmente é passado em um registrador, pois não caberá no offset, conforme veremos.

A Seção 2.4 explica que, como o MIPS admite constantes negativas, a subtração imediata não é necessária no MIPS.

2.4

Representando instruções no computador

Agora, estamos prontos para explicar a diferença entre o modo como os humanos instruem os computadores e como os computadores vêem as instruções. Primeiro, vamos rever rapidamente como um computador representa os números.

Nós, humanos, fomos ensinados a pensar na base 10, mas os números podem ser representados em qualquer base. Por exemplo, 123 base 10 = 1111011 base 2.

No hardware do computador, os números são mantidos como uma série de sinais eletrônicos altos e baixos e, por isso, são considerados números na base 2. (Assim como os números na base 10 são chamados números *decimais*, os números na base 2 são chamados números *binários*.) Um único dígito de um número binário, portanto, é o “átomo” da computação, pois toda a informação é composta por **dígitos binários**, ou *bits*. Esse bloco de montagem fundamental pode ter um dentre dois valores, que podem ser considerados por diversos pontos de vista: alto ou baixo, ligado ou desligado, verdadeiro ou falso, ou 1 ou 0.

As instruções também são mantidas no computador como uma série de sinais eletrônicos altos e baixos e podem ser representadas como números. Na verdade, cada parte da instrução pode ser considerada um número individual, e a colocação desses números lado a lado forma a instrução.

Como os registradores fazem parte de quase todas as instruções, é preciso haver uma convenção para mapear nomes de registrador em números. No assembly MIPS, os registradores \$s0 a \$s7 são mapeados nos registradores de 16 a 23, e os registradores \$t0 a \$t7 são mapeados nos registradores de 8 a 15. Logo, \$s0 significa o registrador 16, \$s1 significa o registrador 17, \$s2 significa o registrador 18, ..., \$t0 significa o registrador 8, \$t1 significa o registrador 9, e assim por diante. Nas próximas seções, descreveremos a convenção para o restante dos 32 registradores.

dígito binário Também chamado bit. Um dos dois números na base 2 (0 ou 1), que são os componentes básicos da informação.

TRADUZINDO UMA INSTRUÇÃO ASSEMBLY MIPS PARA UMA INSTRUÇÃO DE MÁQUINA

EXEMPLO

Realizaremos a próxima etapa no refinamento da linguagem do MIPS como um exemplo. Mostraremos a versão da linguagem real do MIPS para a instrução representada simbolicamente por

`add $t0,$s1,$s2`

primeiro como uma combinação dos números decimais e depois dos números binários.

RESPOSTA

A representação decimal é

0	17	18	8	0	32
---	----	----	---	---	----

Cada um desses segmentos de uma instrução é chamado de *campo*. O primeiro e o último campos (contendo 0 e 32, nesse caso) combinados dizem ao computador MIPS que essa instrução realiza

soma. O segundo campo indica o número do registrador que é o primeiro operando de origem da operação de soma ($17 = \$s1$), e o terceiro campo indica o outro operando de origem para a soma ($18 = \$s2$). O quarto campo contém o número do registrador que deverá receber a soma ($8 = \$t0$). O quinto campo não é utilizado nessa instrução, de modo que é definido como 0. Assim, a instrução soma o registrador $\$s1$ ao registrador $\$s2$ e coloca a soma no registrador $\$t0$.

Essa instrução também pode ser representada com campos em números binários, em vez de decimal:

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Para distinguir do assembly, chamamos a versão numérica das instruções de **linguagem de máquina**, e a seqüência dessas instruções é o *código de máquina*.

Esse *layout* da instrução é denominado **formato de instrução**. Como você pode ver pela contagem do número de bits, essa instrução MIPS ocupa exatamente 32 bits – o mesmo tamanho de uma word de dados. Levando em conta nosso princípio de projeto, em que a simplicidade favorece a regularidade, todas as instruções MIPS possuem 32 bits de largura.

Pode parecer que agora você estará lendo e escrevendo seqüências longas e cansativas de números binários. Evitamos esse tédio usando uma base maior do que a binária, que pode ser convertida com facilidade para binária. Como quase todos os tamanhos de dados no computador são múltiplos de 4, os números **hexadecimais** (base 16) são muito populares. Como a base 16 é uma potência de 2, podemos converter de modo trivial substituindo cada grupo de quatro dígitos binários por um único dígito hexadecimal e vice-versa. A Figura 2.5 converte hexadecimal para binário e vice-versa.

linguagem de máquina: Representação binária utilizada para a comunicação dentro de um sistema computacional.

formato de instrução: Uma forma de representação de uma instrução, composta de campos de números binários.

hexadecimal: Números na base 16.

Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal
0 _{hexa}	0000 _{bin}	4 _{hexa}	0100 _{bin}	8 _{hexa}	1000 _{bin}	C _{hexa}	1100 _{bin}
1 _{hexa}	0001 _{bin}	5 _{hexa}	0101 _{bin}	9 _{hexa}	1001 _{bin}	D _{hexa}	1101 _{bin}
2 _{hexa}	0010 _{bin}	6 _{hexa}	0110 _{bin}	A _{hexa}	1010 _{bin}	E _{hexa}	1110 _{bin}
3 _{hexa}	0011 _{bin}	7 _{hexa}	0111 _{bin}	B _{hexa}	1011 _{bin}	F _{hexa}	1111 _{bin}

FIGURA 2.5 A tabela de conversão hexadecimal-binário. Basta substituir um dígito hexadecimal pelos quatro dígitos binários correspondentes e vice-versa. Se o tamanho do número binário não for um múltiplo de 4, prossiga da direita para a esquerda.

Visto que freqüentemente lidamos com bases numéricas diferentes, para evitar confusão, vamos anexar em subscrito *dec* aos números decimais, *bin* aos números binários e *hex* aos números hexadecimais. (Se não houver um subscrito, a base padrão é 10.) A propósito, C e Java utilizam a notação *0xnnnn* para os números hexadecimais.

BINÁRIO PARA HEXADECIMAL E VICE-VERSA

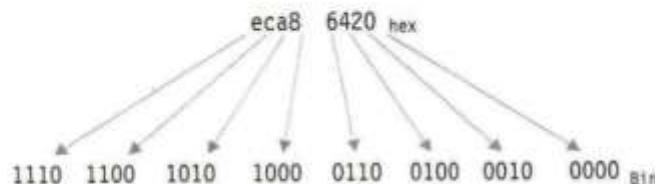
Converta os seguintes números hexadecimais e binários para a outra base:

EXEMPLO

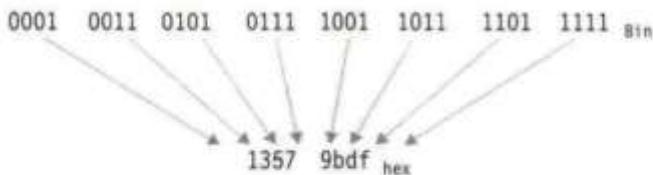
eca8 6420_{hex}
0001 0011 0101 0111 1001 1011 1101 1111_{bin}

Basta olhar na tabela em uma direção:

RESPOSTA



E depois na outra direção:



Campos do MIPS

Os campos do MIPS recebem nomes para facilitar a discussão sobre eles:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Aqui está o significado de cada nome dos campos nas instruções MIPS:

- *op*: operação básica da instrução, tradicionalmente chamado de **opcode**.
- *rs*: o registrador do primeiro operando de origem.
- *rt*: o registrador do segundo operando de origem.
- *rd*: o registrador do operando de destino. Ele recebe o resultado da operação.
- *shamt*: “Shift amount” (quantidade de deslocamento). (A Seção 2.5 explica as instruções de shift e esse termo; ele não será usado até lá, e, por isso, o campo contém zero.)
- *funct*: função. Esse campo seleciona a variante específica da operação no campo *op* e, às vezes, é chamado de *código de função*.

opcode O campo que indica a operação e o formato de uma instrução.

Existe um problema quando uma instrução precisa de campos maiores do que aqueles mostrados. Por exemplo, a instrução load word precisa especificar dois registradores e uma constante. Se o endereço tivesse de usar um dos campos de 5 bits no formato anterior, a constante dentro da instrução load word seria limitada a apenas 2^5 , ou 32. Essa constante é utilizada para selecionar elementos dos arrays ou estruturas de dados e normalmente precisa ser muito maior do que 32. Esse campo de 5 bits é muito pequeno para realizar algo útil.

Logo, temos um conflito entre o desejo de manter todas as instruções com o mesmo tamanho e o desejo de ter um formato de instrução único. Isso nos leva ao último princípio de projeto de hardware:

Princípio de Projeto 4: um bom projeto exige bons compromissos.

O compromisso escolhido pelos projetistas do MIPS é manter todas as instruções com o mesmo tamanho, exigindo assim diferentes tipos de formatos para diferentes tipos de instruções. Por exemplo, o formato anterior é chamado de *tipo-R* (de registrador) ou *formato R*. Um segundo tipo de formato de instrução é chamado *tipo I* (de imediato), ou *formato I*, e é utilizado pelas instruções imediatas e de transferência de dados. Os campos do formato I são

op	rs	rt	constante ou endereço
6 bits	5 bits	5 bits	16 bits

O endereço de 16 bits significa que uma instrução load word pode carregar qualquer word dentro de uma região de $\pm 2^{15}$, ou 32.768 bytes ($\pm 2^{13}$, ou 8.192 words) do endereço no registrador base *rs*. De modo semelhante, a soma imediata é limitada a constantes que não sejam maiores do que $\pm 2^{15}$. (O Capítulo 3 explica como representar números negativos.) Vemos que o uso de mais de 32 registradores seria difícil nesse formato, pois os campos *rs* e *rt* precisariam cada um de outro bit, tornando mais difícil encaixar tudo em uma word.

Vejamos a instrução load word da página 42:

```
lw $t0,32($s3) # Registrador temporário $t0 recebe A[8]
```

Aqui, 19 (para \$s3) é colocado no campo rs, 8 (para \$t0) é colocado no campo rt, e 32 é colocado no campo de endereço. Observe que o significado do campo rt mudou para essa instrução: em uma instrução load word, o campo rt especifica o registrador de *destino*, que recebe o resultado do load.

Embora o uso de vários formatos complique o hardware, podemos reduzir a complexidade mantendo os formatos semelhantes. Por exemplo, os três primeiros campos nos formatos de tipo R e tipo I possuem o mesmo tamanho e têm os mesmos nomes; o quarto campo no tipo I é igual ao tamanho dos três últimos campos do tipo R.

Caso você esteja curioso, os formatos são diferenciados pelos valores no primeiro campo: cada formato recebe um conjunto distinto de valores no primeiro campo (op), de modo que o hardware sabe se deve tratar a última metade da instrução como três campos (tipo R) ou como um único campo (tipo I). A Figura 2.6 mostra os números utilizados em cada campo para as instruções MIPS descritas na Seção 2.3.

Instrução	Formato	op	rs	rt	rd	Shamt	Funct	endereço
add	R	0	reg	reg	reg	0	32 _{dec}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34 _{dec}	n.a.
add immediate	I	8 _{dec}	reg	reg	n.a.	n.a.	n.a.	constante
lw (load word)	I	35 _{dec}	reg	reg	n.a.	n.a.	n.a.	endereço
sw (store word)	I	43 _{dec}	reg	reg	n.a.	n.a.	n.a.	endereço

FIGURA 2.6 Codificação de Instruções MIPS. Na tabela, “reg” significa um número de registrador entre 0 e 31, “endereço” significa um endereço de 16 bits, e “n.a.” (não se aplica) significa que esse campo não aparece nesse formato. Observe que as instruções add e sub têm o mesmo valor no campo op; o hardware usa o campo funct para decidir sobre a variante da operação: somar (32) ou subtrair (34).

TRADUZINDO DO ASSEMBLY MIPS PARA A LINGUAGEM DE MÁQUINA

Agora, já podemos usar um exemplo completo daquilo que o programador escreve até o que o computador executa. Se \$t1 possui a base do array A e \$s2 corresponde a h, então a instrução de atribuição

$A[300] = h + A[300];$

é compilada para

```
lw    $t0,1200($t1)  # Reg. temporário $t0 recebe A[300]
add  $t0,$s2,$t0      # Reg. temporário $t0 recebe h + A[300]
sw    $t0,1200($t1)  # Armazena h + A[300] de volta para A[300]
```

Qual o código em linguagem de máquina MIPS para essas três instruções?

Por conveniência, primeiro vamos representar as instruções em linguagem de máquina usando os números decimais. Pela Figura 2.6, podemos determinar as três instruções em linguagem de máquina:

op	rs	rt	rd	endereço/shamt	funct
35	9	8		1200	
0	18	8	8	0	32
43	9	8		1200	

A instrução lw é identificada por 35 (ver Figura 2.6) no primeiro campo (op). O registrador base 9 (\$t1) é especificado no segundo campo (rs) e o registrador de destino 8 (\$t0) é especificado no terceiro campo (rt). O offset para selecionar A[300] ($1200 = 300 \times 4$) aparece no campo final (endereço).

EXEMPLO

RESPOSTA

A instrução add, que vem em seguida, é especificada com 0 no primeiro campo (op) e 32 no último campo (funct). Os três registradores operandos (18, 8 e 8) aparecem no segundo, terceiro e quarto campos e correspondem a \$s2, \$t0 e \$t0.

A instrução sw é identificada com 43 no primeiro campo. O restante dessa última instrução é idêntico à instrução lw.

O equivalente binário ao formato decimal é o seguinte (1200 na base 10 é 0000 0100 1011 0000 na base 2):

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Observe a semelhança das representações binárias da primeira e última instruções. A única diferença está no terceiro bit a partir da esquerda.

A Figura 2.7 resume as partes do assembly do MIPS descritas nesta seção. Como veremos nos Capítulos 5 e 6, a semelhança das representações binárias de instruções relacionadas simplifica o projeto do hardware. Essas instruções são outro exemplo da regularidade da arquitetura MIPS.

Operandos MIPS

Nome	Exemplo	Comentários
32 registradores	\$s0, \$s1, ..., \$s7 \$t0, \$t1, ..., \$t7	Locais rápidos para dados. No MIPS, os dados precisam estar em registradores para a realização de operações aritméticas. Os registradores \$s0-\$s7 são mapeados para 16-23; \$t0-\$t7 são mapeados para 8-15.
2^{30} words na memória	Memória[0], Memória[4],..., Memória[4294967292]	Acessadas apenas por instruções de transferência de dados no MIPS. O MIPS utiliza endereços em bytes, de modo que os endereços em words seqüenciais diferem em 4 vezes. A memória contém estruturas de dados, arrays e spilled registers.

Assembly do MIPS

Categoria	Instrução	Exemplo	Significado	Comentários
Aritmética	add	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Três operandos; dados nos registradores
	subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	Três operandos; dados nos registradores
Transferência de dados	load word	lw \$s1,100(\$s2)	\$s1 = Memória[\$s2 + 100]	Dados da memória para o registrador
	store word	sw \$s1,100(\$s2)	Memória[\$s2 + 100] = \$s1	Dados do registrador para a memória

Assembly do MIPS

Nome	Formato	Exemplo						Comentários
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)
Tamanho do campo		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Todas as instruções MIPS possuem 32 bits
Formato R	R	op	rs	rt	rd	shamt	funct	Formato das instruções aritméticas
Formato I	I	op	rs	rt	Endereço			Formato das instruções para transferências de dados

FIGURA 2.7 Arquitetura MIPS revelada até a Seção 2.4. As partes destacadas mostram as estruturas da linguagem de máquina MIPS apresentadas na Seção 2.4. Os dois formatos de instrução MIPS até aqui são R e I. Os 16 primeiros bits são iguais: ambos contêm um campo op, indicando a operação básica; um campo rs, indicando um dos operandos origem; e um campo rt, que especifica o outro operando origem, exceto para load word, onde especifica o registrador destino. O formato R divide os 16 últimos bits em um campo rd, especificando o registrador destino; um campo shamt, explicado na Seção 2.5; e o campo funct, que especifica a operação específica das instruções no formato R. O formato I mantém os 16 bits finais como um único campo de endereço.

Por que o MIPS não possui uma instrução de subtração imediata (subtract immediate)?

**Verifique
você mesmo**

1. Constantes negativas aparecem com muito menos freqüência em C e Java e, por isso, não são o caso comum e não merecem suporte especial.
2. Como o campo imediato mantém constantes negativas e positivas, a soma imediata com um valor negativo é equivalente à subtração imediata com um número positivo, de modo que a subtração imediata é supérflua.

Os computadores de hoje são baseados em dois princípios fundamentais:

1. As instruções são representadas como números.
2. Os programas são armazenados na memória para serem lidos ou escritos, assim como os números.

Esses princípios levam ao conceito de *programa armazenado*; sua invenção permite que o gênio da computação saia de sua garrafa. A Figura 2.8 mostra o poder do conceito; especificamente, a memória pode conter o código-fonte de um editor de textos, o código de máquina compilado correspondente, o texto que o programa compilado está usando e até mesmo o compilador que gerou o código de máquina. Uma consequência de instruções em forma de números é que os programas normalmente são entregues como arquivos de números binários. A implicação comercial é que os computadores podem herdar softwares já prontos desde que sejam compatíveis com um conjunto de instruções existente. Essa “compatibilidade binária” normalmente alinha o setor em torno de uma quantidade muito pequena de arquiteturas de conjuntos de instruções.

**Colocando em
perspectiva**



FIGURA 2.8 O conceito de programa armazenado. Os programas armazenados permitem que um computador que realiza contabilidade se torne, em um piscar de olhos, um computador que ajuda um autor a escrever um livro. A troca acontece simplesmente carregando a memória com programas e dados e depois dizendo ao computador para iniciar a execução em determinado local na memória. Tratar as instruções da mesma maneira que os dados simplifica bastante tanto o hardware da memória quanto o software dos sistemas computacionais. Especificamente, a tecnologia de memória necessária para os dados também pode ser usada para programas, e programas como compiladores, por exemplo, podem traduzir o código escrito em uma notação muito mais conveniente para os humanos em código que o computador consiga entender.

Detalhamento: a representação de números decimais na base 2 oferece um modo fácil de representar inteiros positivos em words do computador. O Capítulo 3 explica como representar os números negativos, mas, por enquanto, acredite que uma word de 32 bits pode representar inteiros entre -2^{31} e $+2^{31} - 1$, ou $-2.147.483.648$ a $+2.147.483.647$, e que o campo constante de 16 bits na realidade mantém de -2^{15} a $+2^{15} - 1$, ou -32.768 a 32.767 . Esses inteiros são chamados números em complemento a dois. O Capítulo 3 mostra como codificariamos addi \$t0,\$t0,-1 ou lw \$t0,-4(\$s0), que exigem números negativos no campo de constante do formato imediato.

"Ao contrário",
continuou
Tweedledee,
"se foi assim,
poderia ser;
e se fosse assim,
seria; mas como
não é, então não
é. Isso é lógico."
Lewis Carroll,
*Alice no país das
maravilhas*, 1865

2.5

Operações lógicas

Embora os primeiros computadores se concentrassem em words completas, logo ficou claro que era útil atuar sobre campos de bits dentro de uma word ou até mesmo sobre bits individuais. Examinar os caracteres dentro de uma word, cada um dos quais armazenados como 8 bits, é um exemplo dessa operação. Instruções foram acrescentadas para simplificar, entre outras coisas, o empacotamento e o desempacotamento dos bits em words. Essas instruções são chamadas operações lógicas. A Figura 2.9 mostra as operações lógicas em C e Java.

Operações lógicas	Operadores C	Operadores Java	Instruções MIPS
Shift à esquerda	<<	<<	sll
Shift à direita	>>	>>>	srl
AND bit a bit	&	&	and, andi
OR bit a bit			or, ori
NOT bit a bit	-	-	nor

FIGURA 2.9 Operadores lógicos em C e Java e suas instruções MIPS correspondentes.

A primeira classe dessas operações é chamada de *shifts* (deslocamentos). Elas movem todos os bits de uma word para a esquerda ou direita, preenchendo os bits que ficaram vazios com 0s. Por exemplo, se o registrador \$s0 tivesse

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_{bin} = 9_{dec}$$

e fosse executada a instrução para deslocar 4 bits à esquerda, o novo valor se pareceria com:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{bin} = 144_{dec}$$

O complemento de um shift à esquerda é um shift à direita. Os nomes reais das duas instruções shift no MIPS são *shift left logical* (sll) e *shift right logical* (srl). A instrução a seguir realiza essa operação, supondo que o resultado deva ir para o registrador \$t2:

```
sll $t2,$s0,4 # reg $t2 = reg $s0 << 4 bits
```

Adiamos até agora a explicação do campo *shamt*, do formato R. O nome significa *shift amount* (quantidade de deslocamento) e é usado nas instruções de deslocamento. Logo, a versão em linguagem de máquina da instrução anterior é

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

A codificação de sll é 0 nos campos op e funct, rd contém \$t2, rt contém \$s0 e shamt contém 4. O campo rs não é utilizado e, por isso, é definido como 0.

O deslocamento lógico à esquerda oferece um benefício adicional. O deslocamento à esquerda de i bits gera o mesmo resultado que multiplicar por 2^i (o Capítulo 3 explica por quê). Por exemplo, a instrução \$11 anterior desloca de 4, o que gera o mesmo resultado que multiplicar por 2^4 , ou 16. O primeiro padrão de bits descrito anteriormente representa 9, e $9 \times 16 = 144$, o valor do segundo padrão de bits.

Outra operação útil que isola os campos é *AND*. AND é uma operação bit a bit que deixa um 1 no resultado somente se os dois bits dos operandos forem 1. Por exemplo, se o registrador \$t2 ainda tiver

0000 0000 0000 0000 1101 0000 0000_{bin}

e o registrador \$t1 tiver

0000 0000 0000 0000 0011 1100 0000 0000_{bin}

então, depois de executar a instrução MIPS

and \$t0,\$t1,\$t2 # reg \$t0 = reg \$t1 & reg \$t2

o valor do registrador \$t0 seria

0000 0000 0000 0000 1100 0000 0000_{bin}

Como você pode ver, o AND pode aplicar um padrão de bits a um conjunto de bits para forçar 0s onde houver um 0 no padrão de bits. Esse padrão de bits, em conjunto com o AND, tradicionalmente é chamado de *máscara*, pois a máscara “oculta” alguns bits.

Para colocar um valor em um desses 0s, existe o complemento do AND, chamado *OR*. Essa é uma operação bit a bit, que coloca 1 no resultado se *qualquer um* dos bits do operando for 1. Exemplificando, se os registradores \$t1 e \$t2 não tiverem sido alterados do exemplo anterior, o resultado da instrução MIPS

or \$t0,\$t1,\$t2 # reg \$t0 = reg \$t1 | reg \$t2

é este valor no registrador \$t0:

0000 0000 0000 0011 1101 0000 0000_{bin}

A última operação lógica é um contrário. O **NOT** apanha um operando e coloca um 1 no resultado se um bit do operando for 0, e vice-versa. Acompanhando o formato de dois operandos, os projetistas do MIPS decidiram incluir a instrução **NOR** (NOT OR) no lugar de NOT. Se um operando for zero, então ele é equivalente a NOT. Por exemplo, A NOR 0 = NOT (A OR 0) = NOT (A).

Se o registrador \$t1 não tiver mudado desde o exemplo anterior e o registrador \$t3 tiver o valor 0, o resultado da instrução MIPS

nor \$t0,\$t1,\$t3 # reg \$t0 = - (reg \$t1 | reg \$t3)

é este valor no registrador \$t0:

1111 1111 1111 1111 1100 0011 1111 1111_{bin}

A Figura 2.9 mostrou o relacionamento entre os operadores em C e Java e as instruções MIPS. As constantes são úteis nas operações lógicas AND e OR, assim como nas operações aritméticas, de modo que o MIPS também oferece as instruções *and imediato* (*andi*) e *or imediato* (*ori*). As constantes são raras para NOR, pois seu uso principal é inverter os bits de um único operando; assim, o hardware não possui uma versão imediata. A Figura 2.10, que resume as instruções MIPS vistas até aqui, destacando as instruções lógicas.

NOT Uma operação lógica bit a bit com um operando, que inverte os bits; ou seja, ela substitui cada 1 por um 0, e cada 0 por um 1.

NOR Uma operação lógica bit a bit com dois operandos, que calcula o NOT do OR dos dois operandos.

Operandos MIPS

Nome	Exemplo	Comentários
32 registradores	\$s0, \$s1, ..., \$s7 \$t0, \$t1, ..., \$t7	Locais rápidos para dados. No MIPS, os dados precisam estar em registradores para a realização de operações aritméticas. Os registradores \$s0-\$s7 são mapeados para 16-23; \$t0-\$t7 são mapeados para 8-15.
2 ³⁰ words na memória	Memória[0], Memória[4],..., Memória[4294967292]	Acessadas apenas por instruções de transferência de dados no MIPS. O MIPS utiliza endereços em bytes, de modo que os endereços em words seqüenciais diferem em 4 vezes. A memória contém estruturas de dados, arrays e spilled registers.

Assembly do MIPS

Categoría	Instrução	Exemplo	Significado	Comentários
Aritmética	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Três operandos
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Três operandos
	add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constante
Lógica	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Três operadores em registrador; AND bit a bit
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \$s3$	Três operadores em registrador; OR bit a bit
	nor	nor \$s1,\$s2,\$s3	$\$s1 = -(\$s2 \$s3)$	Três operadores em registrador; NOR bit a bit
	and immediate	andi \$s1,\$s2,100	$\$s1 = \$s2 \& 100$	AND bit a bit entre registrador com constante
	or immediate	ori \$s1,\$s2,100	$\$s1 = \$s2 100$	OR bit a bit entre registrador com constante
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Deslocamento à esquerda por constante
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Deslocamento à direita por constante
Transferência de dados	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memória}[\$s2 + 100]$	Word da memória para o registrador
	store word	sw \$s1,100(\$s2)	$\text{Memória}[\$s2 + 100] = \$s1$	Word do registrador para a memória

FIGURA 2.10 Arquitetura MIPS revelada até aqui. O destaque indica as partes introduzidas desde a Figura 2.7. O Guia de referência rápida, encontrado no inicio do livro, também lista a linguagem de máquina do MIPS.

2.6**Instruções para tomada de decisões**

A utilidade de um computador automático se encontra na possibilidade de usar determinada seqüência de instruções repetidamente, sendo que o número de vezes em que ela é repetida depende dos resultados do cálculo. Quando a iteração é concluída, uma seqüência diferente de [instruções] deve ser seguida, de modo que devemos, em quase todos os casos, oferecer dois fluxos de [instruções] em paralelo precedidos por uma instrução indicando qual rotina deve ser seguida. Essa escolha pode depender do sinal de um número (sendo que zero é considerado positivo para as finalidades da máquina). Consequentemente, apresentamos uma [instrução] (a [instrução] de transferência condicional) que, dependendo do sinal de determinado número, causa a execução de uma dentre duas rotinas.

Burks, Goldstine e von Neumann, 1947

O que distingue um computador de uma calculadora simples é a sua capacidade de tomar decisões. Com base nos dados de entrada e nos valores criados durante o cálculo, diferentes instruções são executadas. A tomada de decisão normalmente é representada nas linguagens de programação usando a instrução *if*, às vezes combinadas com instruções *go to* e rótulos (labels). O assembly do MIPS inclui duas instruções para tomada de decisões, semelhantes a uma instrução *if* com um *go to*. A primeira instrução é

`beq registrador1, registrador2, L1`

Essa instrução significa ir até a instrução rotulada por L1 se o valor no registrador1 for igual ao valor no registrador2. O mnemônico `beq` significa *branch if equal* (desviar se for igual). A segunda instrução é

```
bne registrador1, registrador2, L1
```

Elas significam ir até a instrução rotulada por L1 se o valor no registrador1 *não* for igual ao valor no registrador2. O mnemônico `bne` significa *branch if not equal* (desviar se não for igual). Essas duas instruções tradicionalmente são denominadas **desvios condicionais**.

desvio condicional Uma instrução que realiza uma comparação de dois valores e que leva em conta uma transferência de controle subsequente para um novo endereço no programa, com base no resultado da comparação.

COMPILEANDO IF-THEN-ELSE EM DESVIOS CONDICIONAIS

No segmento de código a seguir, f, g, h, i e j são variáveis. Se as cinco variáveis de f a j correspondem aos cinco registradores de \$s0 a \$s4, qual é o código MIPS compilado para esta instrução *if* em C?

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

A Figura 2.11 é um fluxograma de como deve ser o código MIPS. A primeira expressão compara a igualdade, de modo que poderíamos querer usar `beq`. De modo geral, o código será mais eficiente se testarmos a condição oposta ao desvio no lugar do código que realiza a parte *then* subsequente do *if* (o rótulo Else é definido a seguir):

```
bne $s3,$s4,Else      # vai para Else se i ≠ j
```

A próxima instrução de atribuição realiza uma única operação, e se todos os operandos estiverem em registradores, essa é apenas uma instrução:

```
add $s0,$s1,$s2      # f = g + h (ignorada se i ≠ j)
```

Agora, precisamos ir até o final da instrução *if*. Este exemplo apresenta outro tipo de desvio, normalmente chamado *desvio incondicional*. Essa instrução diz que o processador sempre deverá seguir o desvio. Para distinguir entre os desvios condicionais e incondicionais, o nome MIPS para esse tipo de instrução é *jump*, abreviado como *j* (o rótulo *Exit* é definido a seguir).

```
j Exit      # vai para Exit
```

A instrução de atribuição na parte *else* da instrução *if* pode novamente ser compilada para uma única instrução. Só precisamos anexar um rótulo *Else* a essa instrução. Também mostramos o rótulo *Exit* que está após essa instrução, mostrando o final do código compilado de *if-then-else*:

```
Else:sub $s0,$s1,$s2      # f = g - h (ignorada se i = j)
Exit:
```

EXEMPLO

RESPOSTA

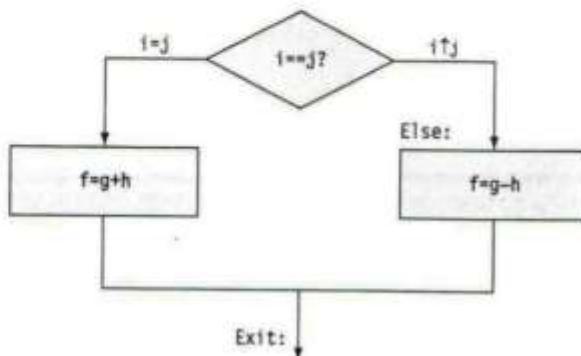


FIGURA 2.11 Ilustração das opções na Instrução If anterior. A caixa da esquerda corresponde à parte *then* da instrução *if*, e a caixa da direita corresponde à parte *else*.

Observe que o montador alivia o compilador e o programador assembly do trabalho de calcular endereços para os desvios, assim como ele os alivia do cálculo dos endereços de dados para loads e stores (ver Seção 2.10).

Interface hardware/software

Os compiladores constantemente criam desvios e rótulos onde eles não aparecem na linguagem de programação. Evitar o trabalho de escrever rótulos e desvios explícitos é um benefício da escrita em linguagens de programação de alto nível, e um dos motivos para a codificação ser mais rápida nesse nível.

COMPILANDO UM LOOP WHILE EM C

EXEMPLO

Aqui está um loop tradicional em C:

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

Suponha que *i* e *k* correspondam aos registradores \$s3 e \$s5 e a base do array save esteja em \$s6. Qual é o código assembly MIPS correspondente a esse segmento C?

RESPOSTA

O primeiro passo é ler *save[i]* para um registrador temporário. Antes que possamos ler *save[i]* para um registrador temporário, precisamos ter seu endereço. Antes que possamos somar *i* à base do array *save* para formar o endereço, temos de multiplicar *i* por 4, devido ao problema do endereçamento em bytes. Felizmente, podemos usar o deslocamento lógico à esquerda, pois o deslocamento à esquerda de 2 bits multiplica por 4 (ver Seção 2.5). Precisamos acrescentar o rótulo Loop para podermos desviar de volta a essa instrução no final do loop:

```
Loop: sll $t1,$s3,2 # Registrador temporário $t1 = 4 * i
```

Para obter o endereço de *save[i]*, temos de somar \$t1 e a base do array *save* em \$s6:

```
add $t1,$t1,$s6      # $t1 = endereço de save[i]
```

Agora, podemos usar esse endereço para ler *save[i]* para um registrador temporário:

```
lw $t0,0($t1)      # Registro temporário $t0 = save[i]
```

A próxima instrução realiza o teste do loop, terminando se *save[i]* ≠ *k*:

```
bne $t0,$s5, Exit    # vai para Exit se save[i] ≠ k
```

A próxima instrução soma 1 a *i*:

```
add $s3,$s3,1        # i = i + 1
```

O final do loop desvia de volta ao teste do *while* no início do loop. Simplesmente acrescentamos o rótulo *Exit* depois dele e terminamos:

```
j Loop            # vai para Loop
Exit:
```

(Veja, no Exercício 2.25, uma otimização para essa seqüência.)

Loops

Decisões são importantes tanto para escolher entre duas alternativas – como encontramos nas instruções *if* – quanto para repetir um cálculo – como nos loops. As mesmas instruções assembly são os blocos de montagem para os dois casos.

Interface hardware/software

Essas seqüências de instruções que terminam em um desvio são tão fundamentais para a compilação que recebem seu próprio termo: um **bloco básico** é uma seqüência de instruções sem desvios, exceto, possivelmente, no final, e sem destinos de desvio ou rótulos de desvio, exceto, possivelmente, no início. Uma das primeiras fases da compilação é desmembrar o programa em blocos básicos.

O teste de igualdade ou desigualdade provavelmente é o teste mais popular, mas às vezes é útil ver se uma variável é menor do que outra variável. Por exemplo, um loop *for* pode querer testar se a variável de índice é menor do que 0. Essas comparações são realizadas em assembly do MIPS com uma instrução que compara dois registradores e atribui 1 a um terceiro registrador se o primeiro for menor do que o segundo; caso contrário, é atribuído 0. A instrução MIPS é chamada *set on less than* (atribuir se menor que), ou *slt*. Por exemplo,

```
slt    $t0, $s3, $s4
```

significa que é atribuído 1 ao registrador \$t0 se o valor no registrador \$s3 for menor do que o valor no registrador \$s4; caso contrário, é atribuído 0 ao registrador \$t0.

Operadores constantes são populares nas comparações. Como o registrador \$zero sempre tem 0, já podemos comparar com 0. Para comparar com outros valores, existe uma versão imediata da instrução *slt*. Para testar se o registrador \$s2 é menor do que a constante 10, podemos simplesmente escrever

```
slti   $t0,$s2,10    # $t0 = 1 se $s2 < 10
```

Atentando para a advertência de von Neumann quanto à simplicidade do “equipamento”, a arquitetura do MIPS não inclui “desvio se menor que”, pois isso é muito complicado; ou ela esticaria o tempo do ciclo de clock ou exigiria ciclos de clock extras por instrução. Duas instruções mais rápidas são mais úteis.

bloco básico Uma seqüência de instruções sem desvios (exceto, possivelmente, no final) e sem destinos de desvio ou rótulos de desvio (exceto, possivelmente, no início). Uma das primeiras fases da compilação é desmembrar o programa em blocos básicos.

Interface hardware/software

Os compiladores MIPS utilizam as instruções *slt*, *slti*, *beq*, *bne* e o valor fixo 0 (sempre à disposição com a leitura do registrador \$zero) para criar todas as condições relativas: igual, diferente, menor que, menor ou igual, maior que, maior ou igual. (Como você poderia esperar, o registrador \$zero é mapeado para o registrador 0.)

Instrução Case/Switch

A maioria das linguagens de programação possui uma instrução *case* ou *switch*, para o programador poder selecionar uma dentre muitas alternativas, dependendo de um único valor. O modo mais simples de implementar *switch* é por meio de uma seqüência de testes condicionais, transformando a instrução *switch* em uma cadeia de instruções *if-then-else*.

tabela de endereços de desvio Também chamada tabela de desvios. Uma tabela de endereços de seqüências de instruções alternativas.

Às vezes, as alternativas podem ser codificadas de forma mais eficiente como uma tabela de endereços de seqüências de instruções alternativas, chamada **tabela de endereços de desvio**, e o programa só precisa indexar na tabela e depois desviar para a seqüência apropriada. A tabela de desvios é, então, apenas um array de words com endereços que correspondem aos rótulos no código.

Para apoiar tais situações, computadores como o MIPS incluem uma instrução *jump register* (jr), significando um desvio incondicional para o endereço especificado em um registrador. O programa carrega a entrada apropriada a partir da tabela de desvios para um registrador, e depois desvia para o endereço apropriado usando um jump register. Essa instrução é descrita na Seção 2.7.

Interface hardware/software

Embora haja muitas instruções para decisões e loops em linguagens de programação como C e Java, a instrução básica que as implementa no nível inferior é o desvio condicional.

A Figura 2.12 resume as partes do assembly MIPS descritas nesta seção, e a Figura 2.13 resume a linguagem de máquina MIPS correspondente. Essa etapa da evolução da linguagem do MIPS acrescentou desvios e jumps à nossa representação simbólica e fixa o valor 0, muito útil, permanentemente em um registrador.

Detalhamento: se você já ouviu falar em *delayed branches*, explicados no Capítulo 6, não se preocupe: o montador do MIPS os torna invisíveis ao programador assembly.

Operandos MIPS

Nome	Exemplo	Comentários
32 registradores	\$s0, \$s1, ..., \$s7 \$t0, \$t1, ..., \$t7	Locais rápidos para dados. No MIPS, os dados precisam estar em registradores para a realização de operações aritméticas. Os registradores \$s0-\$s7 são mapeados para 16-23; \$t0-\$t7 são mapeados para 8-15. O registrador MIPS \$zero sempre é igual a 0.
2 ³⁰ words na memória	Memória[0], Memória[4]..... Memória[4294967292]	Acessadas apenas por instruções de transferência de dados no MIPS. O MIPS utiliza endereços em bytes, de modo que os endereços em words seqüenciais diferem em 4 vezes. A memória contém estruturas de dados, arrays e spilled registers.

Assembly do MIPS

Categoria	Instrução	Exemplo	Significado	Comentários
Aritmética	Add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Três operandos; dados nos registradores
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Três operandos; dados nos registradores
Transferência de dados	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memória}[\$s2 + 100]$	Dados da memória para o registrador
	store word	sw \$s1,100(\$s2)	$\text{Memória}[\$s2 + 100] = \$s1$	Dados do registrador para a memória
Lógica	And	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Três operadores em registrador; AND bit a bit
	Or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \$s3$	Três operadores em registrador; OR bit a bit
	nor	nor \$s1,\$s2,\$s3	$\$s1 = -(\$s2 \$s3)$	Três operadores em registrador; NOR bit a bit
	and immediate	andi \$s1,\$s2,100	$\$s1 = \$s2 \& 100$	AND bit a bit entre registrador com constante
	or immediate	ori \$s1,\$s2,100	$\$s1 = \$s2 100$	OR bit a bit entre registrador com constante
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Deslocamento à esquerda por constante
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Deslocamento à direita por constante
Desvio condicional	branch on equal	beq \$s1,\$s2,L	if ($\$s1 == \$s2$) go to L	Testa igualdade e desvia
	branch on not equal	bne \$s1,\$s2,L	if ($\$s1 != \$s2$) go to L	Testa desigualdade e desvia
	set on less than	slt \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compara menor que; usado com beq, bne
	set on less than immediate	slt \$s1,\$s2,100	if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$	Compara menor que imediato; usado com beq, bne
Desvio incondicional	jump	jr L	go to L	Desvia para endereço de destino

FIGURA 2.12 Arquitetura MIPS revelada até a Seção 2.6. As partes destacadas mostram as estruturas MIPS introduzidas na Seção 2.6.

Linguagem de máquina do MIPS

Nome	Formato	Exemplo						Comentários
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
lw	I	35	18	17	100			lw \$s1,100(\$s2)
sw	I	43	18	17	100			sw \$s1,100(\$s2)
and	R	0	18	19	17	0	36	and \$s1,\$s2,\$s3
or	R	0	18	19	17	0	37	or \$s1,\$s2,\$s3
nor	R	0	18	19	17	0	39	nor \$s1,\$s2,\$s3
andi	I	12	18	17	100			andi \$s1,\$s2,100
ori	I	13	18	17	100			ori \$s1,\$s2,100
sll	R	0	0	18	17	10	0	sll \$s1,\$s2,10
srl	R	0	0	18	17	10	2	srl \$s1,\$s2,10
beq	I	4	17	18	25			beq \$s1,\$s2,100
bne	I	5	17	18	25			bne \$s1,\$s2,100
slt	R	0	18	19	17	0	42	slt \$s1,\$s2,\$s3
j	J	2	2500					j 10000 (ver Seção 2.9)
Tamanho do campo		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Todas as instruções MIPS de 32 bits
Formato R	R	op	rs	rt	rd	shamt	funct	Formato das instruções aritméticas
Formato I	I	op	rs	rt	endereço			Formato para transferências de dados e desvios

FIGURA 2.13 A linguagem de máquina do MIPS revelada até a Seção 2.6. As partes destacadas mostram as estruturas do MIPS introduzidas na Seção 2.6. O formato J, usado para instruções de jump, é explicado na Seção 2.9, que também explica os valores apropriados para os campos de endereço das instruções de desvio.

A linguagem C possui muitas instruções para decisões e loops, enquanto o MIPS possui poucas. Quais dos seguintes itens explicam ou não explicam esse desequilíbrio? Por quê?

Verifique você mesmo

1. Mais instruções de decisão tornam o código mais fácil de ler e entender.
2. Menos instruções de decisão simplificam a tarefa da camada inferior responsável pela execução.
3. Mais instruções de decisão significam menos linhas de código, o que geralmente reduz o tempo de codificação.
4. Mais instruções de decisão significam menos linhas de código, o que geralmente resulta na execução de menos operações.

Por que a linguagem C oferece dois conjuntos de operadores para AND (& e &&) e dois conjuntos de operadores para OR (| e ||), enquanto o MIPS não faz isso?

1. As operações lógicas AND e OR implementam & e |, enquanto os desvios condicionais implementam && e ||.
2. A afirmativa anterior é o contrário: && e || correspondem a operações lógicas, enquanto & e | são mapeados para desvios condicionais.
3. Elas são redundantes e significam a mesma coisa: && e || são simplesmente herdados da linguagem de programação B, a antecessora do C.

2.7

Supor te a procedimentos no hardware do computador

procedimento Uma sub-routine armazenada que realiza uma tarefa com base nos parâmetros com os quais ela é provida.

Um **procedimento** ou *função* é uma ferramenta que os programadores C ou Java utilizam para estruturar programas, tanto para torná-los mais fáceis de entender quanto para permitir que o código seja reutilizado. Os procedimentos permitem que o programador se concentre em apenas uma parte da tarefa de cada vez, com os parâmetros atuando como uma barreira entre o procedimento e o restante do programa e dos dados, permitindo que sejam passados valores e resultados de retorno. Descrevemos o equivalente em Java no final desta seção, mas a linguagem Java precisa de tudo de um computador que a linguagem C também necessita.

Você pode pensar em um procedimento como um espião que sai com um plano secreto, adquire recursos, realiza a tarefa, sobre seus rastros e depois retorna ao ponto de origem com o resultado desejado. Nada mais deverá ter sido perturbado depois que a missão terminar. Além do mais, um espião opera apenas sobre aquilo que ele “precisa saber”, de modo que não pode fazer suposições sobre seu patrão.

De modo semelhante, na execução de um procedimento, o programa precisa seguir estas seis etapas:

1. Colocar parâmetros em um lugar onde o procedimento possa acessá-los.
2. Transferir o controle para o procedimento.
3. Adquirir os recursos de armazenamento necessários para o procedimento.
4. Realizar a tarefa desejada.
5. Colocar o valor de retorno em um local onde o programa que o chamou possa acessá-lo.
6. Retornar o controle para o ponto de origem, pois um procedimento pode ser chamado de vários pontos em um programa.

Como já dissemos, os registradores são o local mais rápido para manter dados em um computador, de modo que queremos usá-los ao máximo possível. O software do MIPS utiliza a seguinte convenção na alocação de seus 32 registradores para a chamada de procedimento:

- \$a0-\$a3: quatro registradores de argumento, para passar parâmetros
- \$v0-\$v1: dois registradores de valor, para valores de retorno
- \$ra: um registrador de endereço de retorno, para retornar ao ponto de origem

Além de alocar esses registradores, o assembly do MIPS inclui uma instrução apenas para os procedimentos: ela desvia para um endereço e simultaneamente salva o endereço da instrução seguinte no registrador \$ra. A **instrução de jump-and-link** (*jal*) é escrita simplesmente como

jal EndereçoProcedimento

A parte do *link* no nome da instrução significa que um endereço ou link é formatado para apontar para o local de chamada, permitindo que o procedimento retorne ao endereço correto. Esse “link”, armazenado no registrador \$ra, é denominado **endereço de retorno**. O endereço de retorno é necessário porque o mesmo procedimento poderia ser chamado de várias partes do programa.

Implicita na idéia de programa armazenado é a necessidade de ter um registrador para manter o endereço da instrução atual sendo executada. Por motivos históricos, esse registrador quase sempre é denominado **contador de programa**, abreviado como *PC* (Program Counter) na arquitetura MIPS, embora um nome mais sensato teria sido *registraror de endereço de instrução*. A instrução *jal* salva o PC + 4 no registrador \$ra para o link com a instrução seguinte, a fim de preparar o retorno do procedimento.

instrução de jump-and-link Uma instrução que salta para um endereço e simultaneamente salva o endereço da instrução seguinte em um registrador (\$ra no MIPS).

endereço de retorno Um link para o local de chamada, permitindo que um procedimento retorne ao endereço correto; no MIPS, ele é armazenado no registrador \$ra.

contador de programa (PC) O registrador que contém o endereço da instrução que está sendo executada no programa.

Para apoiar tais situações, computadores como o MIPS utilizam uma instrução de *jump register* (*jr*), significando um desvio incondicional para o endereço especificado em um registrador:

```
jr $ra
```

A instrução de jump register pula para o endereço armazenado no registrador *\$ra* – que é exatamente o que queremos. Assim, o programa que chama, ou **caller**, coloca os valores de parâmetro em *\$a0-\$a3* e utiliza *jal X* para desviar para o procedimento *X* (às vezes denominado **callee**). O callee, então, realiza os cálculos, coloca os resultados em *\$v0-\$v1* e retorna o controle para o caller usando *jr \$ra*.

Usando mais registradores

Suponha que um compilador precise de mais registradores para um procedimento do que os quatro registradores para argumentos e os dois para valores de retorno. Como temos de cobrir nossos rastros após o término da nossa missão, quaisquer registradores necessários ao caller deverão ser restaurados aos valores que possuíam *antes* de o procedimento ser chamado. Essa situação é um exemplo em que são usados os spilled registers em memória, conforme mencionamos na Seção “Interface hardware/software” da página 42.

A estrutura de dados ideal para armazenar os spilled registers é uma **pilha** – uma fila do tipo “último a entrar, primeiro a sair”. Uma pilha precisa de um ponteiro para o endereço alocado mais recentemente na pilha, para mostrar onde o próximo procedimento deverá colocar os spilled registers ou onde os valores antigos dos registradores estão localizados. O **stack pointer** é ajustado em uma word para cada registrador salvo ou restaurado. As pilhas são tão populares que possuem seus próprios termos para transferir dados da pilha e para ela: colocar dados na pilha é denominado *push*, e remover dados da pilha é denominado *pop*.

O software do MIPS aloca outro registrador apenas para a pilha: o stack pointer (*\$sp*), usado para salvar os registradores necessários pelo procedimento chamado. Por motivos históricos, as pilhas “crescem” de endereços maiores para endereços menores. Essa convenção significa que você leva valores para a pilha subtraindo do valor do stack pointer. Somar ao stack pointer diminui essa pilha, removendo seus valores.

COMPILANDO UM PROCEDIMENTO EM C QUE NÃO CHAMA OUTRO PROCEDIMENTO

Vamos transformar o exemplo da página 51 em um procedimento em C:

```
int exemplo_folha (int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

Qual é o código assembly do MIPS compilado?

As variáveis de parâmetro *g*, *h*, *i* e *j* correspondem aos registradores de argumento *\$a0*, *\$a1*, *\$a2* e *\$a3*, e *f* corresponde a *\$s0*. O programa compilado começa com o rótulo do procedimento:

exemplo_folha:

O próximo passo é salvar os registradores usados pelo procedimento. A instrução de atribuição em C no corpo do procedimento é idêntica ao exemplo da página 38, que usa dois registradores temporá-

caller O programa que instiga um procedimento e oferece os valores de parâmetro necessários.

callee Um procedimento que executa uma série de instruções armazenadas com base nos parâmetros fornecidos pelo caller e depois retorna o controle para o caller novamente.

pilha (stack) Uma estrutura de dados utilizada para armazenar os spilled registers, organizada como uma fila do tipo “último a entrar, primeiro a sair”.

stack pointer Um valor indicando o endereço alocado mais recentemente em uma pilha, que mostra onde spilled registers devem ser armazenados ou onde os valores antigos dos registradores podem ser localizados.

EXEMPLO

RESPOSTA

rios. Assim, precisamos salvar três registradores: \$s0, \$t0 e \$t1. "Empilhamos" os valores antigos, criando espaço para três words na pilha e depois as armazenamos:

```
addi $sp,$sp,-12 # ajusta a pilha, criando espaço para 3 itens
sw $t1, 8($sp)   # salva reg. $t1 para usar depois
sw $t0, 4($sp)   # salva reg. $t0 para usar depois
sw $s0, 0($sp)   # salva reg. $s0 para usar depois
```

A Figura 2.14 mostra a pilha antes, durante e após a chamada do procedimento. As três instruções seguintes correspondem ao corpo do procedimento, que segue o exemplo da página 38:

```
add $t0,$a0,$a1 # reg. $t0 contém g + h
add $t1,$a2,$a3 # reg. $t1 contém i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, que é (g + h)-(i + j)
```

Para retornar o valor de f, nós o copiamos para um registrador de valor de retorno:

```
add $v0,$s0,$zero # retorna f ($v0 = $s0 + 0)
```

Antes de retornar, restauramos os três valores antigos dos registradores que salvamos, desempilhando-os:

```
lw $s0, 0($sp) # restaura reg. $s0 para o caller
lw $t0, 4($sp) # restaura reg. $t0 para o caller
lw $t1, 8($sp) # restaura reg. $t1 para o caller
addi $sp,$sp,12 # ajusta pilha para excluir 3 itens
```

O procedimento termina com um jump register usando o endereço de retorno:

```
jr $ra # desvia de volta à rotina que chamou
```

No exemplo anterior, usamos registradores temporários e consideramos que seus valores antigos precisam ser salvos e restaurados. Para evitar salvar e restaurar um registrador cujo valor nunca é utilizado, o que poderia acontecer com um registrador temporário, o software do MIPS separa 18 dos registradores em dois grupos:

- \$t0-\$t9: 10 registradores temporários que *não* são preservados pelo procedimento chamado em uma chamada de procedimento
- \$s0-\$s7: 8 registradores salvos que precisam ser preservados em uma chamada de procedimento (se forem usados, o procedimento chamado os salva e restaura)

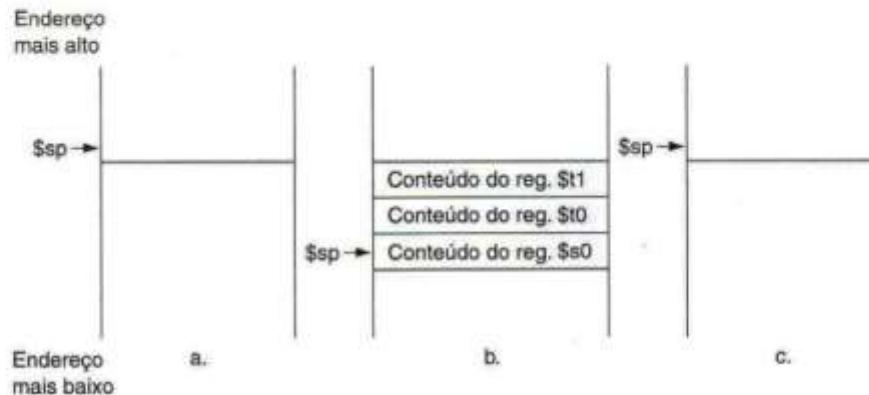


Figura 2.14 Os valores do stack pointer e a pilha (a) antes, (b) durante e (c) após a chamada do procedimento. O stack pointer sempre aponta para o "topo" da pilha, ou para a última word na pilha, neste desenho.

Essa convenção simples reduz o register spilling. No exemplo anterior, como o caller não espera que os registradores \$t0 e \$t1 sejam preservados durante uma chamada de procedimento, podemos descartar dois stores e dois loads do código. Ainda temos de salvar e restaurar \$s0, pois o procedimento chamado precisa considerar que o caller precisa de seu valor.

Procedimentos aninhados

Os procedimentos que não chamam outros são denominados procedimentos *folha*. A vida seria simples se todos os procedimentos fossem procedimentos folha, mas não são. Assim como um espião poderia empregar outros espiões como parte de uma missão, que, por sua vez, poderiam utilizar ainda mais espiões, os procedimentos também chamam outros procedimentos. Além do mais, os procedimentos recursivos ainda chamam “clones” de si mesmos. Assim como precisamos ter cuidado ao usar registradores nos procedimentos, também precisamos ter mais cuidado ao chamar procedimentos não folha.

Por exemplo, suponha que o programa principal chame o procedimento A com um argumento 3, colocando o valor 3 no registro \$a0 e depois usando ja1 A. Depois, suponha que o procedimento A chame o procedimento B por meio de ja1 B com um argumento 7, também colocado em \$a0. Como A ainda não terminou sua tarefa, existe um conflito com relação ao uso do registrador \$a0. De modo semelhante, existe um conflito em relação ao endereço de retorno no registrador \$ra, pois ele agora tem o endereço de retorno para B. A menos que tomemos medidas para evitar o problema, esse conflito eliminará a capacidade do procedimento A de retornar para o procedimento que o chamou.

Uma solução é empilhar todos os outros registradores que precisam ser preservados, assim como fizemos com os registradores salvos. O caller empilha quaisquer registradores de argumento (\$a0-\$a3) ou registradores temporários (\$t0-\$t9) que sejam necessários após a chamada. O callee empilha o registrador do endereço de retorno \$ra e quaisquer registradores salvos (\$s0-\$s7) usados por ele. O stack pointer \$sp é ajustado para levar em consideração a quantidade de registradores colocados na pilha. No retorno, os registradores são restaurados da memória e o stack pointer é reajustado.

COMPILANDO UM PROCEDIMENTO C RECURSIVO, MOSTRANDO A LIGAÇÃO DO PROCEDIMENTO ANINHADO

Vamos realizar um procedimento recursivo que calcula o fatorial:

EXEMPLO

```
int fact (int n)
{
    if (n < 1) return (1);
    else return (n * fact(n-1));
}
```

Qual é o código assembly do MIPS?

O parâmetro n corresponde ao registrador de argumento \$a0. O programa compilado começa com o rótulo do procedimento e depois salva dois registradores na pilha, o endereço de retorno e \$a0:

RESPOSTA

```
fact:
    addi $sp,$sp,-8    # ajusta pilha para 2 itens
    sw   $ra, 4($sp)  # salva o endereço de retorno
    sw   $a0, 0($sp)  # salva o argumento n
```

Na primeira vez que fact é chamado, sw salva um endereço do programa que chamou fact. As duas instruções seguintes testam se n é menor do que 1, indo para L1 se n = 1.

```

    slti $t0,$a0,1      # teste para n < 1
    beq $t0,$zero,L1    # se n >= 1, vai para L1

```

Se n for menor do que 1, `fact` retorna 1, colocando 1 em um registrador de valor: ele soma 1 a 0 e coloca essa soma em $\$v_0$. Depois, ele retira os dois valores salvos da pilha e desvia para o endereço de retorno:

```

    addi $v0,$zero,1    # retorna 1
    addi $sp,$sp,8      # retira 2 itens da pilha
    jr $ra              # retorna para depois de jal

```

Antes de retirar dois itens da pilha, poderíamos ter restaurado $\$a_0$ e $\$ra$. Como $\$a_0$ e $\$ra$ não mudam quando n é menor do que 1, pulamos essas instruções.

Se n não for menor do que 1, o argumento n é decrementado e depois `fact` é chamado novamente com o valor decrementado.

```

L1: addi $a0,$a0,-1   # n >= 1: argumento recebe (n - 1)
    jal fact            # chama fact com (n - 1)

```

A próxima instrução é onde `fact` retorna. Agora, o endereço de retorno antigo e o argumento antigo são restaurados, juntamente com o stack pointer:

```

lw $a0, 0($sp)    # retorna de jal: restaura argumento n
lw $ra, 4($sp)    # restaura o endereço de retorno
addi $sp, $sp,8    # ajusta stack pointer para retirar 2 itens

```

Em seguida, o registrador de valor $\$v_0$ recebe o produto do argumento antigo $\$a_0$ e o valor atual do registrador de valor. Consideraremos que exista uma instrução de multiplicação à disposição, embora isso não seja explicado antes do Capítulo 3:

```
mul $v0,$a0,$v0    # retorna n * fact (n - 1)
```

Finalmente, `fact` salta novamente para o endereço de retorno:

```
jr $ra              # retorna para o procedimento que chamou
```

Interface hardware/software

Uma variável em C é um local na memória, e sua interpretação depende tanto do seu *tipo* quanto da sua *classe de armazenamento*. A linguagem C possui duas classes de armazenamento: *automática* e *estática*. As variáveis automáticas são locais a um procedimento e são descartadas quando o procedimento termina. As variáveis estáticas permanecem durante entradas e saídas de procedimentos. As variáveis C declaradas fora de todos os procedimentos são consideradas estáticas, assim como quaisquer variáveis declaradas por meio da palavra reservada `static`. As outras são automáticas. Para simplificar o acesso aos dados estáticos, o software do MIPS reserva outro registrador, chamado **ponteiro global**, ou $\$gp$.

ponteiro global 0
registrador reservado
para apontar para
dados estáticos.

A Figura 2.15 resume o que é preservado em uma chamada de procedimento. Observe que vários esquemas preservam a pilha. A pilha acima de $\$sp$ é preservada simplesmente verificando se o procedimento chamado não escreve acima de $\$sp$; $\$sp$ é preservado pelo procedimento chamado somando-se exatamente o mesmo valor que foi subtraído dele, e os outros registradores são preservados por serem salvos na pilha (se forem usados) e restaurados de lá. Essas ações também garantem que o caller receberá, ao fazer um `load` da pilha, os mesmos dados que foram colocados lá com um `store`, pois o procedimento chamado promete preservar o $\$sp$ e não modificar a parte da pilha pertencente ao caller, ou seja, a área acima do $\$sp$ no momento da chamada.

Preservado	Não preservado
Registradores salvos: \$s0-\$s7	Registradores temporários: \$t0-\$t9
Registrador de stack pointer: \$sp	Registradores de argumento: \$a0-\$a3
Registrador de endereço de retorno: \$ra	Registradores de valores de retorno: \$v0-\$v1
Pilha acima do stack pointer	Pilha abaixo do stack pointer

FIGURA 2.15 O que é e o que não é preservado durante uma chamada de procedimento. Se o software contar com o registrador de frame pointer ou com o registrador de ponteiro global, discutidos nas próximas seções, eles também são preservados.

Alocando espaço para novos dados na pilha

A complexidade final é que a pilha também é utilizada para armazenar variáveis que são locais ao procedimento, que não cabem nos registradores, como arrays ou estruturas locais. O segmento da pilha que contém os registradores salvos e as variáveis locais de um procedimento é chamado **frame de procedimento, ou registro de ativação**. A Figura 2.16 mostra o estado da pilha antes, durante e após a chamada de um procedimento.

Alguns softwares MIPS utilizam o **frame pointer (\$fp)** para apontar para a primeira word do registro de ativação de um procedimento. O stack pointer poderia mudar durante o procedimento, e assim as referências a uma variável local na memória poderiam ter offsets diferentes, dependendo de onde estiverem no procedimento, o que torna o procedimento mais difícil de entender. Como alternativa, um frame pointer oferece um registrador base estável dentro de um procedimento para as referências locais à memória. Observe que um registro de ativação aparece na pilha independente de o frame pointer explícito ser utilizado. Evitamos o \$fp evitando mudanças no \$sp dentro de um procedimento: em nossos exemplos, a pilha é ajustada apenas na entrada e na saída do procedimento.

frame de procedimento Também chamado **registro de ativação**. O segmento da pilha contendo os registradores salvos e as variáveis locais de um procedimento.

frame pointer Um valor indicando o local dos registradores salvos e as variáveis locais para um determinado procedimento.

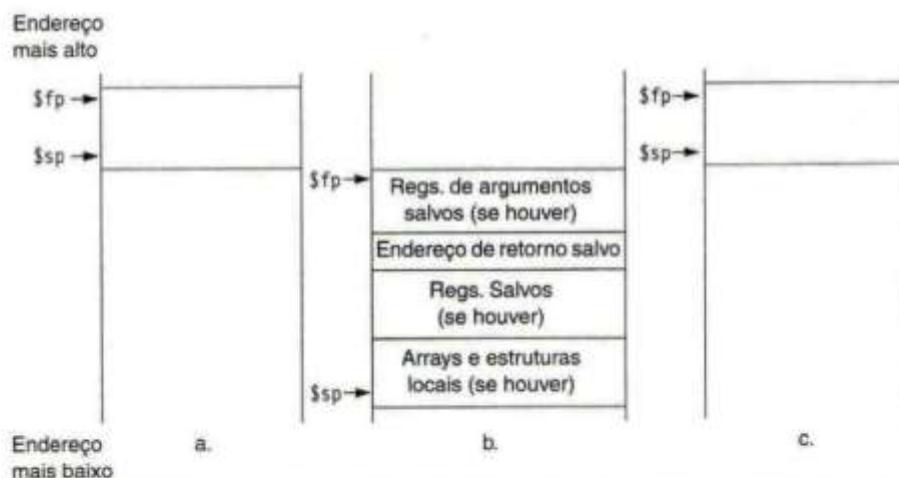


FIGURA 2.16 Ilustração da alocação de pilha (a) antes, (b) durante e (c) após a chamada de um procedimento. O frame pointer (\$fp) aponta para a primeira word do frame, normalmente um registrador de argumento salvo, e o stack pointer (\$sp) aponta para o topo da pilha. A pilha é ajustada para criar espaço para todos os registradores salvos e quaisquer variáveis locais residentes na memória. Como o stack pointer pode mudar durante a execução do programa, é mais fácil para os programadores referenciarem variáveis por meio do frame pointer estável, embora isso também pudesse ser feito por meio do stack pointer e um pouco de aritmética de endereços. Se não houver variáveis locais na pilha dentro de um procedimento, o compilador ganhará tempo não atribuindo um endereço ao frame pointer, e depois, restaurando-o. Quando um frame pointer é usado, ele é inicializado usando o endereço que está no \$sp em uma chamada, e o \$sp é restaurado usando o valor do \$fp.

Alocando espaço para novos dados no heap

Além das variáveis automáticas que são locais aos procedimentos, os programadores C precisam de espaço na memória para as variáveis estáticas e para estruturas de dados dinâmicas. A Figura 2.17 mostra a convenção do MIPS para a alocação de memória. A pilha começa na parte alta da memória e cresce para baixo. A primeira parte da extremidade baixa da memória é reservada, seguida pelo código de máquinas do MIPS, tradicionalmente denominado **segmento de texto**. Acima do código existe o **segmento de dados estáticos**, que é o local para constantes e outras variáveis estáticas. Embora os arrays costumem ter um tamanho fixo e, portanto, correspondem muito bem ao segmento de dados estático, estruturas de dados como listas encadeadas costumam crescer e diminuir durante suas vidas. O segmento para tais estruturas de dados é tradicionalmente chamado de *heap*, e fica posicionado logo a seguir na memória. Observe que essa alocação permite que a pilha e o heap cresçam um em direção ao outro, permitindo assim o uso eficiente da memória enquanto os dois segmentos aumentam e diminuem.

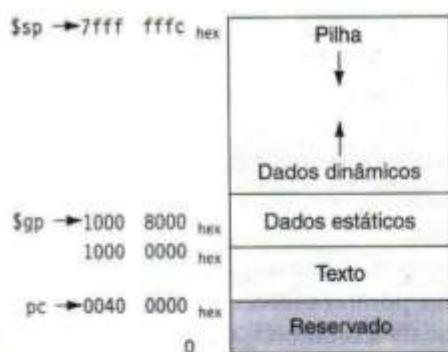


FIGURA 2.17 A alocação de memória do MIPS para programas e dados. Esses endereços são apenas uma convenção do software, e não fazem parte da arquitetura MIPS. De cima para baixo, o stack pointer é inicializado com 7fff ffff_{hexa} e cresce para baixo, em direção ao segmento de dados. Na outra extremidade, o código do programa ("texto") começa em 0040 0000_{hexa}. Os dados estáticos começam em 1000 0000_{hexa}. Os dados dinâmicos, alocados por malloc em C e por new em Java, vêm em seguida e crescem para cima em direção à pilha, em uma área chamada heap. O ponteiro global, \$gp, é definido como um endereço para facilitar o acesso aos dados. Ele é inicializado com 1000 8000_{hexa} para poder acessar de 1000 0000_{hexa} até 1000 ffff_{hexa} usando os offsets de 16 bits positivos e negativos a partir do \$gp (ver endereçamento com complemento a dois no Capítulo 3).

A linguagem C aloca e libera espaço no heap com funções explícitas. `malloc()` aloca espaço no heap e retorna um ponteiro para ela, e `free()` libera o espaço para o qual o ponteiro está apontando. A alocação da memória é controlada por programas em C, e essa é a fonte de muitos bugs comuns e difíceis de serem encontrados. Esquecer de liberar espaço ocasiona um “desperdício de memória”, que, por fim, pode vir a causar a falha do sistema operacional. Liberar espaço muito cedo ocasiona “ponteiros pendentes”, podendo fazer os ponteiros apontarem para áreas que o programa nunca desejou.

A Figura 2.18 resume as convenções de registrador para o assembly do MIPS. As Figuras 2.19 e 2.20 resumem as partes das instruções assembly do MIPS descritas até aqui e as instruções de máquina MIPS correspondentes.

Detalhamento: e se houver mais do que quatro parâmetros? A convenção do MIPS é colocar os parâmetros extras na pilha, logo acima do stack pointer. O procedimento, então, espera que os quatro primeiros parâmetros estejam nos registradores de \$a0 a \$a3, e que o restante esteja na memória, endereçável por meio do frame pointer.

Conforme dissemos na legenda da Figura 2.16, o frame pointer é conveniente porque todas as referências a variáveis na pilha dentro de um procedimento terão o mesmo offset. Contudo, o frame pointer não é necessário. O compilador C para MIPS sob licença GNU utiliza um frame pointer, mas não o compilador C da MIPS/Silicon Graphics; ele utiliza o registrador 30 como outro registrador de valor salvo (\$s8).

ja1, na realidade, salva o endereço da instrução que vem após ja1 no registrador \$ra, permitindo assim que um retorno de procedimento seja simplesmente jr \$ra.

Nome	Número do registrador	Uso	Preservado na chamada?
\$zero	0	o valor constante 0	n.a.
\$v0-\$v1	2-3	valores para resultados e avaliação de expressões	não
\$a0-\$a3	4-7	Argumentos	não
\$t0-\$t7	8-15	Temporários	não
\$s0-\$s7	16-23	Valores salvos	sim
\$t8-\$t9	24-25	Mais temporários	não
\$gp	28	Ponteiro global	sim
\$sp	29	stack pointer	sim
\$fp	30	frame pointer	sim
\$ra	31	Endereço de retorno	sim

FIGURA 2.18 Convenções de registradores MIPS. O registrador 1, chamado \$at, é reservado para o montador (ver Seção 2.10), e os registradores 26-27, chamados \$k0-\$k1, são reservados para o sistema operacional.

Operandos MIPS

Nome	Exemplo	Comentários
32 registradores	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra	Locais rápidos para dados. No MIPS, os dados precisam estar em registradores para a realização de operações aritméticas. O registrador MIPS \$zero sempre é igual a 0. \$gp (28) é o ponteiro global. \$sp (29) é o stack pointer. \$fp é o frame pointer, e \$ra (31) é o endereço de retorno.
2^{30} words na memória	Memória[0], Memória[4]...., Memória[4294967292]	Acessadas apenas por instruções de transferência de dados no MIPS. O MIPS utiliza endereços em bytes, de modo que os endereços em words seqüenciais diferem em 4 vezes. A memória contém estruturas de dados, arrays e spilled registers, como aqueles salvos nas chamadas de procedimento.

Assembly do MIPS

Categoria	Instrução	Exemplo	Significado	Comentários
Aritmética	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Três operandos; dados nos registradores
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Três operandos; dados nos registradores
Transferência de dados	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memória}[\$s2 + 100]$	Dados da memória para o registrador
	store word	sw \$s1,100(\$s2)	$\text{Memória}[\$s2 + 100] = \$s1$	Dados do registrador para a memória
Lógica	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Três operadores em registrador; AND bit a bit
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \$s3$	Três operadores em registrador; OR bit a bit
	nor	nor \$s1,\$s2,\$s3	$\$s1 = -(\$s2 \$s3)$	Três operadores em registrador; NOR bit a bit
	and immediate	andi \$s1,\$s2,100	$\$s1 = \$s2 \& 100$	AND bit a bit registrador com constante
	or immediate	ori \$s1,\$s2,100	$\$s1 = \$s2 100$	OR bit a bit registrador com constante
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Deslocamento à esquerda por constante
Desvio condicional	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Deslocamento à direita por constante
	branch on equal	beq \$s1,\$s2,L	if ($\$s1 == \$s2$) go to L	Testa igualdade e desvia
	branch on not equal	bne \$s1,\$s2,L	if ($\$s1 != \$s2$) go to L	Testa desigualdade e desvia
	set on less than	slt \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compara menor que; usado com beq, bne
Desvio Incondicional	set on less than immediate	slt \$s1,\$s2,100	if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$	Compara menor que imediato; usado com beq, bne
	jump	j L	go to L	Desvia para endereço de destino
	jump register	jr \$ra	go to \$ra	Para retorno de procedimento
	jump and link	jal L	$\$ra = PC + 4$. go to L	Para chamada de procedimento

FIGURA 2.19 Arquitetura MIPS revelada até a Seção 2.7. As partes destacadas mostram as estruturas do assembly do MIPS introduzidas na Seção 2.7. O formato J, usado para instruções de jump e jump-and-link, é explicado na Seção 2.9.

Linguagem de máquina do MIPS

Nome	Formato	Exemplo						Comentários
		0	18	19	17	0	32	
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
lw	I	35	18	17	100			lw \$s1,100(\$s2)
sw	I	43	18	17	100			sw \$s1,100(\$s2)
and	R	0	18	19	17	0	36	and \$s1,\$s2,\$s3
or	R	0	18	19	17	0	37	or \$s1,\$s2,\$s3
nor	R	0	18	19	17	0	39	nor \$s1,\$s2,\$s3
andi	I	12	18	17	100			andi \$s1,\$s2,100
ori	I	13	18	17	100			ori \$s1,\$s2,100
sll	R	0	0	18	17	10	0	sll \$s1,\$s2,10
srl	R	0	0	18	17	10	2	srl \$s1,\$s2,10
beq	I	4	17	18	25			beq \$s1,\$s2,100
bne	I	5	17	18	25			bne \$s1,\$s2,100
slt	R	0	18	19	17	0	42	slt \$s1,\$s2,\$s3
J	J	2	2500					J 10000 (ver Seção 2.9)
jr	R	0	31	0	0	0	8	jr \$ra
jal	J	3	2500					Jal 10000 (ver Seção 2.9)
Tamanho do campo		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Todas as instruções MIPS de 32 bits
Formato R	R	op	rs	rt	rd	shamt	funct	Formato das instruções aritméticas
Formato I	I	op	rs	rt	endereço			Formato para transferências de dados e desvios

FIGURA 2.20 A linguagem de máquina do MIPS revelada até a Seção 2.7. As partes destacadas mostram as estruturas em assembly do MIPS introduzidas na Seção 2.7. O formato J, usado para instruções de jump e jump-and-link, é explicado na Seção 2.9, que também explica por que colocar 25 no campo de endereço das instruções em linguagem de máquina beq e bne é equivalente a 100 em assembly.

Verifique você mesmo

Quais das seguintes afirmações sobre C e Java geralmente são verdadeiras?

- As chamadas de procedimento em C são mais rápidas do que a invocação de métodos em Java.
- Os programadores C gerenciam os dados explicitamente, enquanto isso é automático em Java.
- C ocasiona mais bugs relacionados com ponteiros e desperdício de memória do que Java.
- C passa parâmetros em registradores, enquanto Java os passa na pilha.

!f @ |=>
(wow open tab at
bar is great)

Quarta linha do poema de teclado "Hatless Atlas", 1991 (alguns dão nomes aos caracteres ASCII: "!" é "wow", "(" é open, ")" é bar, e assim por diante).

2.8

Comunicando-se com as pessoas

Os computadores foram inventados para devorar números, mas, assim que se tornaram comercialmente viáveis, eles foram usados para processar textos. A maioria dos computadores hoje utiliza bytes de 8 bits para representar caracteres, sendo que o ASCII (American Standard Code for Information Interchange) é a representação que quase todos seguem. A Figura 2.21 resume o código ASCII.

Diversas instruções podem extrair um byte de uma word, de modo que load word e store word são suficientes para transferir bytes e também words. Entretanto, devido à popularidade do texto em alguns programas, o MIPS oferece instruções para mover bytes. Load byte (lb) lê um byte da memó-

Código ASCII	Caráter										
32	espaço	48	0	64	@	80	P	96	~	112	P
33	!	49	1	65	A	81	Q	97	a	113	q
34	*	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	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	DEL

FIGURA 2.21 Representação dos caracteres no código ASCII. Observe que as letras maiúsculas e minúsculas diferem exatamente em 32; essa observação pode levar a atalhos na verificação ou mudança entre maiúsculas e minúsculas. Os valores não mostrados incluem caracteres de formatação. Por exemplo, 8 representa backspace, 9 representa o caracter de tabulação, e 13 um carriage return. Outro valor útil é 0 para null, o valor que a linguagem C utiliza para marcar o final de uma string.

ria, colocando-o nos 8 bits mais à direita de um registrador. Store byte (sb) separa o byte mais à direita de um registrador e o escreve na memória. Assim, copiamos um byte com a seqüência

```
1b $t0,0($sp)          # Lê byte da origem
sb $t0,0($gp)          # Escreve byte no destino
```

Os caracteres normalmente são combinados em strings, que possuem uma quantidade variável de caracteres. Existem três opções para representar uma string: (1) a primeira posição da string é reservada para indicar o tamanho de uma string, (2) uma variável acompanhante possui o tamanho da string (como em uma estrutura), ou (3) a última posição da string é ocupada por um caracter que serve para marcar o final da string. A linguagem C utiliza a terceira opção, terminando uma string com um byte cujo valor é 0 (denominado null, em ASCII). Assim, a string "Cal" é representada em C pelos 4 bytes a seguir, em forma de números decimais: 67, 97, 108, 0.

COMPILANDO UM PROCEDIMENTO DE CÓPIA DE STRING, PARA DEMONSTRAR O USO DE STRINGS EM C

O procedimento strcpy copia a string *y* para a string *x*, usando a convenção de término com byte nulo da linguagem C:

```
void strcpy (char x[ ], char y[ ])
{
    int i;

    i = 0;
    while ((x[i] = y[i]) != '\0') /* copia e testa byte */
        i += 1;
}
```

EXEMPLO

Qual é o código assembly correspondente no MIPS?

RESPOSTA

A seguir está o segmento básico em código assembly do MIPS. Considere que os endereços base para os arrays *x* e *y* são encontrados em \$a0 e \$a1, enquanto *i* está em \$s0. *strcpy* ajusta o stack pointer e depois salva o registrador de valores salvos \$s0 na pilha:

```
strcpy:
    addi $sp,$sp,-4 # ajusta pilha para mais 1 item
    sw   $s0, 0($sp) # salva $s0
```

Para inicializar *i* como 0, a próxima instrução define \$s0 como 0, somando 0 a 0 e colocando essa soma em \$s0:

```
add $s0,$zero,$zero # i = 0 + 0
```

Esse é o início do loop. O endereço de *y[i]* é formado inicialmente pela soma de *i* a *y[]*:

```
L1: add $t1,$s0,$a1 # endereço de y[i] em $t1
```

Observe que não temos de multiplicar *i* por 4, pois *y* é um array de bytes, e não de words, como nos exemplos anteriores.

Para carregar o caractere em *y[i]*, usamos load byte, que coloca o caractere em \$t2:

```
lb $t2, 0($t1) # $t2 = y[i]
```

Um cálculo de endereço semelhante coloca o endereço de *x[i]* em \$t3, e depois o caractere em \$t2 é armazenado nesse endereço.

```
add $t3,$s0,$a0 # endereço de x[i] em $t3
sb $t2, 0($t3) # x[i] = y[i]
```

Em seguida, saímos do loop se o caractere foi 0; ou seja, esse é o último caractere da string:

```
beq $t2,$zero,L2 # se y[i] == 0, vai para L2
```

Se não, incrementamos *i* e voltamos ao loop:

```
addi $s0, $s0,1 # i = i + 1
j   L1           # vai para L1
```

Se não voltamos, então esse foi o último caractere da string; restauramos \$s0 e o stack pointer, para depois retornar.

```
L2: lw   $s0, 0($sp) # y[i] == 0: fim da string;
      # restaura $s0 antigo
    addi $sp,$sp,4 # retira 1 word da pilha
    jr   $ra         # retorna
```

As cópias de string normalmente utilizam ponteiros no lugar de arrays em C, para evitar as operações com *i* no código anterior. Veja, na Seção 2.15, uma explicação sobre arrays e ponteiros.

Como o procedimento *strcpy* anterior é um procedimento folha, o compilador poderia alocar *i* a um registrador temporário e evitar as operações de salvar e restaurar \$s0. Em virtude disso, em vez de pensar nos registradores \$t como sendo apenas para temporários, podemos pensar neles como registradores que o procedimento chamado deve utilizar sempre que for conveniente. Quando um compilador encontra um procedimento folha, ele esgota todos os registradores temporários antes de usar registradores que precisa salvar.

Caracteres e strings em Java

Unicode é uma codificação universal dos alfabetos da maior parte das linguagens humanas. A Figura 2.22 é uma lista de alfabetos Unicode; existem tantos *alfabetos* em Unicode quanto *símbolos* úteis em ASCII. Para ser mais específico, Java utiliza Unicode para os caracteres. Como padrão, ela utiliza 16 bits para representar um caractere.

Latin	Malayalam	Tagbanwa	General Punctuation
Greek	Sinhala	Khmer	Spacing Modifier Letters
Cyrillic	Thai	Mongolian	Currency Symbols
Armenian	Lao	Limbu	Combining Diacritical Marks
Hebrew	Tibetan	Tai Le	Combining Marks for Symbols
Arabic	Myanmar	Kangxi Radicals	Superscripts and Subscripts
Syriac	Georgian	Hiragana	Number Forms
Thaana	Hangul Jamo	Katakana	Mathematical Operators
Devanagari	Ethiopic	Bopomofo	Mathematical Alphanumeric Symbols
Bengali	Cherokee	Kanbun	Braille Patterns
Gurmukhi	Unified Canadian Aboriginal Syllabic	Shavian	Optical Character Recognition
Gujarati	Ogham	Osmanya	Byzantine Musical Symbols
Oriya	Runic	Cypriot Syllabary	Musical Symbols
Tamil	Tagalog	Tai Xuan Jing Symbols	Arrows
Telugu	Hanunoo	Yijing Hexagram Symbols	Box Drawing
Kannada	Buhid	Aegean Numbers	Geometric Shapes

FIGURA 2.22 Exemplos de alfabetos em Unicode. O Unicode versão 4.0 possui mais de 160 “blocos”, que é o nome para uma coleção de símbolos. Cada bloco é um múltiplo de 16. Por exemplo, Greek começa em 0370_{hexa} e Cyrillic em 0400_{hexa}. As três primeiras colunas mostram 48 blocos que correspondem a linguagens humanas em ordem numérica aproximada no Unicode. A última coluna possui 16 blocos que são multilinguais e não estão em ordem. Uma codificação de 16 bits, chamada UTF-16, é o padrão. Uma codificação de tamanho variável, chamada UTF-8, mantém o subconjunto ASCII como 8 bits e utiliza 16-32 bits para os outros caracteres. UTF-32 utiliza 32 bits por caractere. Para saber mais sobre isso, consulte www.unicode.org.

O conjunto de instruções do MIPS possui instruções explícitas para carregar e armazenar quantidades de 16 bits, chamadas *halfwords*. Load half (lh) lê uma halfword da memória, colocando-a nos 16 bits mais à direita de um registrador. Store half (sh) separa a halfword correspondente aos 16 bits mais à direita de um registrador e a escreve na memória. Copiamos uma halfword com a seqüência

```
lh $t0,0($sp) # Lê halfword (16 bits) da origem
sh $t0,0($gp) # Escreve halfword (16 bits) no destino
```

As strings são uma classe padrão do Java, com suporte interno especial e métodos predefinidos para concatenação, comparação e conversão. Ao contrário da linguagem C, Java inclui uma word que indica o tamanho da string, semelhante aos arrays Java.

Detalhamento: o software do MIPS tenta manter a pilha alinhada em endereços de word, permitindo que o programa sempre use lw e sw (que precisam estar alinhados) para acessar a pilha. Essa convenção significa que uma variável char alocada na pilha ocupa 4 bytes, embora precise de menos. Contudo, uma variável string ou um array de bytes em C agrupará 4 bytes por word, e uma variável string ou array de shorts em Java agrupará 2 halfwords por word.

Verifique você mesmo

Quais das seguintes afirmações sobre caracteres e strings em C e Java são verdadeiras?

1. Uma string em C utiliza cerca da metade da memória da mesma string em Java.
2. Strings são apenas um nome informal para arrays de uma única dimensão de caracteres em C e Java.
3. C e Java utilizam null (0) para marcar o fim de uma string.
4. As operações sobre strings, como saber seu tamanho, são mais rápidas em C do que em Java.

2.9**Endereçamento no MIPS para operandos imediatos e endereços de 32 bits**

Embora manter todas as instruções MIPS com 32 bits simplifique o hardware, existem ocasiões em que seria conveniente ter uma constante de 32 bits ou endereço de 32 bits. Esta seção começa com a solução geral para constantes grandes, e depois apresenta as otimizações para endereços de instruções usados em desvios condicionais e jumps.

Operandos imediatos de 32 bits

Embora as constantes normalmente sejam curtas e caibam em um campo de 16 bits, às vezes elas são maiores. O conjunto de instruções MIPS inclui a instrução *load upper immediate* (*lui*) especificamente para atribuir os 16 bits mais altos de uma constante a um registrador, permitindo que uma instrução subsequente atribua os 16 bits mais baixos da constante. A Figura 2.23 mostra a operação de *lui*.

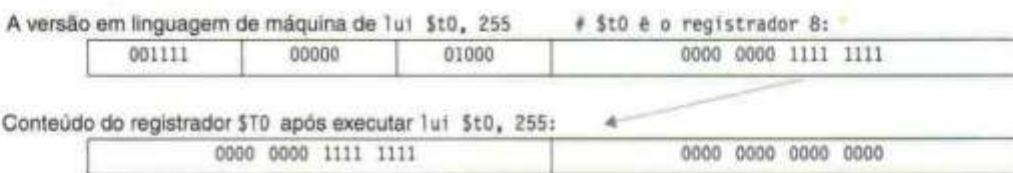


FIGURA 2.23 O efeito da instrução lui. A instrução *lui* transfere o valor do campo de constante imediata de 16 bits para os 16 bits mais à esquerda do registrador, preenchendo os 16 bits de menor ordem (direita) com 0s.

Interface hardware/software

Tanto o compilador quanto o montador precisam desmembrar constantes grandes em partes e depois remontá-las em um registrador. Como você poderia esperar, a restrição de tamanho do campo imediato pode ser um problema para endereços de memória em loads e stores, e também para constantes em instruções imediatas. Se esse trabalho recair para o montador, como acontece para o software do MIPS, então o montador precisa ter um registrador temporário disponível, onde criará valores longos. Esse é um motivo para o registrador *\$at*, que é reservado para o montador.

Logo, a representação simbólica da linguagem de máquina do MIPS não está mais limitada pelo hardware, mas a qualquer coisa que o criador de um montador decidir incluir (ver Seção 2.10). Vamos examinar o hardware de perto para explicar a arquitetura do computador, indicando quando usarmos a linguagem avançada do montador que não se encontra no processador.

CARREGANDO UMA CONSTANTE DE 32 BITS

Qual é o código assembly do MIPS para carregar esta constante de 32 bits no registrador \$s0?

```
0000 0000 0011 1101 0000 1001 0000 0000
```

EXEMPLO

Primeiro, carregariam os 16 bits mais altos, que é 61 em decimal, usando a instrução lui:

```
lui $s0, 61 # 61 decimal = 0000 0000 0011 1101 binário
```

O valor do registrador \$s0 depois disso é

```
0000 0000 0011 1101 0000 0000 0000 0000
```

RESPOSTA

O próximo passo é acrescentar os 16 bits inferiores, cujo valor decimal é 2304:

```
ori $s0, $s0, 2304 # 2304 decimal = 0000 1001 0000 0000
```

O valor final no registrador \$s0 é o valor desejado:

```
0000 0000 0011 1101 0000 1001 0000 0000
```

Detalhamento: a criação de constantes de 32 bits requer cuidado. A instrução addi copia o bit mais à esquerda do campo imediato de 16 bits da instrução para todos os 16 bits mais altos de uma word. O OR imediato lógico, da Seção 2.5, carrega Os nos 16 bits superiores e, portanto, é usada pelo montador em conjunto com lui para criar constantes de 32 bits.

Endereçamento em desvios condicionais e jumps

As instruções de jump no MIPS possuem o endereçamento mais simples possível. Elas utilizam o último formato de instrução do MIPS, chamado *tipo J*, que consiste em 6 bits para o campo de operação e o restante dos bits para o campo de endereço. Assim,

```
j 10000 # vai para a posição 10000
```

poderia ser gerada neste formato (normalmente, isso é um pouco mais complicado, como veremos no próximo exemplo):

2	10000	
6 bits	26 bits	

onde o valor do código da operação de jump é 2 e o endereço destino é 10000.

Ao contrário da instrução de jump, a instrução de desvio condicional precisa especificar dois operandos além do endereço de desvio. Assim,

```
bne $s0,$s1,Exit # vai para Exit se $s0 $s1
```

é gerada nesta instrução, deixando apenas 16 bits para o endereço de desvio:

5	16	17		Exit
6 bits	5 bits	5 bits		16 bits

Se os endereços do programa tivessem de caber nesse campo de 16 bits, nenhum programa poderia ser maior do que 2^{16} , que é muito pequeno para ser uma opção real nos dias atuais. Uma alternativa seria especificar um registrador que sempre seria somado ao endereço de desvio, de modo que uma instrução de desvio pudesse calcular o seguinte:

$$\text{Contador de programa} = \text{Registrador} + \text{Endereço de desvio}$$

Essa soma permite que o contador de programa tenha até 2^{32} bits e ainda possa usar desvios condicionais, solucionando o problema do tamanho do endereço de desvio. A questão, portanto, é: qual registrador?

A resposta vem da observação de como os desvios condicionais são usados. Os desvios condicionais são encontrados em loops e em instruções *if*, de modo que costumam desviar para uma instrução próxima. Por exemplo, cerca de metade de todos os desvios condicionais nos benchmarks SPEC2000 vão para locais a menos de 16 instruções. Como o contador de programa (PC) contém o endereço da instrução atual, podemos desviar dentro de $\pm 2^{15}$ words da instrução atual se usarmos o PC como o registrador a ser somado ao endereço. Quase todos os loops e instruções *if* são muito menores do que 2^{16} words, de modo que o PC é a opção ideal.

endereçamento relativo ao PC Um regime de endereçamento em que o endereço é a soma do contador de programa (PC) e uma constante na instrução.

Essa forma de endereçamento de desvio é denominada **endereçamento relativo ao PC**. Conforme veremos no Capítulo 5, é conveniente que o hardware incremente o PC desde cedo, para que aponte para a próxima instrução. Logo, o endereço MIPS, na realidade, é relativo ao endereço da instrução seguinte ($PC + 4$), em vez da instrução atual (PC).

Como na maioria dos computadores atuais, o MIPS utiliza o endereçamento relativo ao PC para todos os desvios condicionais, pois o destino dessas instruções provavelmente estará próximo do desvio. Por outro lado, instruções de jump-and-link chamam procedimentos que não têm motivo para estarem próximos à chamada e, por isso, normalmente utilizam outras formas de endereçamento. Logo, a arquitetura MIPS oferece endereços longos para chamadas de procedimento, usando o formato do tipo J para instruções de jump e jump-and-link.

Como todas as instruções MIPS possuem 4 bytes de extensão, o MIPS aumenta a distância do desvio fazendo com que o endereçamento relativo ao PC se refira ao número de *words* até a próxima instrução, no lugar do número de bytes. Assim, o campo de 16 bits pode se desviar para uma distância quatro vezes maior, interpretando o campo como um endereço relativo à word, em vez de um endereço relativo a byte. De modo semelhante, o campo de 26 bits nas instruções de jump também é um endereço de word, significando que representa um endereço de 28 bits.

Detalhamento: como o PC (contador de programa) utiliza 32 bits, 4 bits precisam vir de outro lugar. A instrução de jump do MIPS substitui apenas os 28 bits menos significativos do PC, deixando os 4 bits mais significativos inalterados. O loader e o link-editor (Seção 2.9) precisam ter cuidado para evitar colocar um programa entre um limite de endereços de 256MB (64 milhões de instruções); caso contrário, um jump precisa ser substituído por uma instrução de jump register precedida por outras instruções, a fim de carregar o endereço de 32 bits inteiro em um registrador.

MOSTRANDO O OFFSET DO DESVIO EM LINGUAGEM DE MÁQUINA

EXEMPLO

O loop *while* da página 54 foi compilado para este código em assembly do MIPS:

```

Loop: sll $t1,$s3,2    # Reg. temporário $t1 = 4 * i
      add $t1,$t1,$s6   # $t1 = endereço de save[i]
      lw $t0,0($t1)     # Reg. temporário $t0 = save[i]
      bne $t0,$s5, Exit  # Vai para Exit se save[i] != s5
      addi $s3,$s3,1     # i = i + 1
      j Loop            # vai para Loop
Exit:

```

Se consideramos que o loop inicia na posição 80000 da memória, qual é o código de máquina do MIPS para esse loop?

As instruções montadas e seus endereços se pareceriam com:

RESPOSTA

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	...					

Lembre-se de que as instruções do MIPS possuem endereços em bytes, de modo que os endereços das words seqüenciais diferem em 4, a quantidade de bytes em uma word. A instrução bne na quarta linha acrescenta 2 words ou 8 bytes ao endereço da instrução *seguinte* (80016), especificando o destino do desvio em relação à instrução seguinte ($8 + 80016$), e não em relação à instrução de desvio ($12 + 80012$) ou ao uso do endereço de destino completo (80024). A instrução de salto na última linha utiliza o endereço completo ($20000 \times 4 = 80000$), correspondente ao rótulo Loop.

Interface hardware/software

Quase todo desvio condicional é feito para um local nas proximidades, mas, ocasionalmente, ele se desvia para um ponto mais distante, longe do que pode ser representado nos 16 bits da instrução de desvio condicional. O montador vem ao auxílio como fez com endereços ou constantes grandes: ele insere um jump incondicional para o destino do desvio, e inverte a condição de modo que o desvio decida para onde fazer o jump.

DESVIANDO PARA UM LUGAR MAIS DISTANTE

Dado um desvio onde o registrador \$s0 é igual ao registrador \$s1,

EXEMPLO

```
beq    $s0,$s1, L1
```

substitua-o por um par de instruções que ofereça uma distância de desvio muito maior.

Estas instruções substituem o desvio condicional com endereço curto:

RESPOSTA

```
bne    $s0,$s1, L2
      j     L1
L2:
```

Resumo dos modos de endereçamento no MIPS

Múltiplas formas de endereçamento geralmente são denominadas **modos de endereçamento**. Os modos de endereçamento do MIPS são os seguintes:

modo de endereçamento Um dos diversos regimes de endereçamento delimitados por seu uso variado de operandos e/ou endereços.

1. **Endereçamento em registrador**, onde o operando é um registrador
2. **Endereçamento de base ou deslocamento**, onde o operando está no local de memória cujo endereço é a soma de um registrador e uma constante na instrução
3. **Endereçamento imediato**, onde o operando é uma constante dentro da própria instrução

4. **Endereçamento relativo ao PC**, onde o endereçamento é a soma do PC e uma constante na instrução
5. **Endereçamento pseudodireto**, onde o endereço de jump são os 26 bits da instrução concatenados com os bits mais altos do PC

Interface hardware/software

Embora tenhamos mostrado a arquitetura MIPS como tendo endereços de 32 bits, quase todos os microprocessadores (incluindo o MIPS) possuem extensões de endereço de 64 bits (ver o Apêndice D). Essas extensões foram a resposta às necessidades de software para programas maiores. O processo de extensão do conjunto de instruções permite que as arquiteturas se expandam de modo que o software possa prosseguir de forma compatível para a próxima geração da arquitetura.

Observe que uma única operação pode usar mais de um modo de endereçamento. Add, por exemplo, utiliza um endereçamento imediato (addi) e por registrador (add). A Figura 2.24 mostra como os operandos são identificados para cada modo de endereçamento.

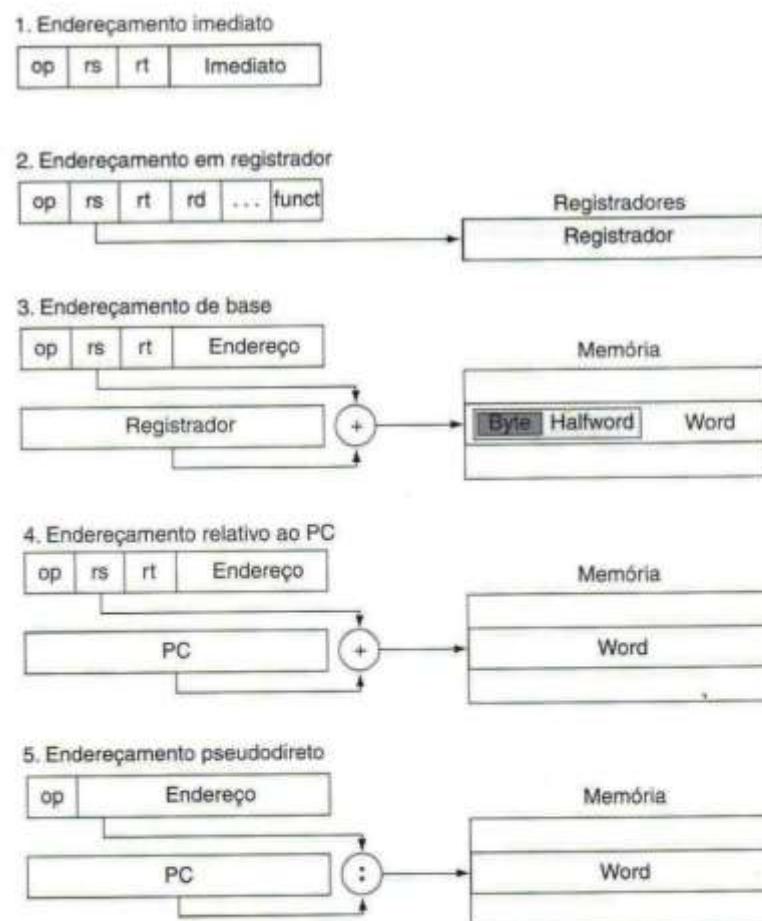


FIGURA 2.24 Ilustração dos cinco modos de endereçamento do MIPS. Os operandos estão sombreados na figura. O operando do modo 3 está na memória, enquanto o operando para o modo 2 é um registrador. Observe que as versões de load e store acessam bytes, halfwords ou words. Para o modo 1, o operando é formado pelos 16 bits da própria instrução. Os modos 4 e 5 endereçam as instruções na memória, com o modo 4 adicionando um endereço de 16 bits deslocado à esquerda em 2 bits no PC, e o modo 5 concatenando um endereço de 26 bits deslocado à esquerda em 2 bits com os 4 bits superiores do PC.

Decodificando a linguagem de máquina

Às vezes, você é forçado a usar engenharia reversa na linguagem de máquina para criar o código assembly original. Um exemplo é quando se examina um dump de memória. A Figura 2.25 mostra a codificação dos campos para a linguagem de máquina do MIPS. Essa figura ajuda na tradução manual entre o assembly e a linguagem de máquina.

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)	Formato R	Blitz/gez	jump	jump & link	Branch eq	branch ne	bnez	bgtz
1(001)	add immediate	addiu	set less than imm.	sltiu	andi	ori	xori	load upper imm
2(010)	TLB	F1Pt						
3(011)								
4(100)	load byte	Load half	lw	load word	lbu	lhu	lw	
5(101)	store byte	store half	sw	store word			sw	
6(110)	lwz0	lwz1						
7(111)	swz0	swz1						

op(31:26)=0100000 (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)	mfc00		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

op(31:26)=0000000 (formato R), 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	sraw
1(001)	jump reg.	jalr			syscall	break		
2(010)	mfhi	nthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	Add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set 1.t.	sltu				
6(110)								
7(111)								

FIGURA 2.25 Codificação de instruções do MIPS. Essa notação indica o valor de um campo por linha e por coluna. Por exemplo, a parte superior da figura mostra load word na linha número 4 (100_{bin} para os bits 31-29 da instrução) e a coluna número 3 (011_{bin} para os bits 28-26 da instrução), de modo que o valor correspondente do campo op (bits 31-26) é 100011_{bin}. Formato R na linha 0 e coluna 0 (op = 000000_{bin}) é definido na parte inferior da figura. Logo, subtract na linha 4 e coluna 2 da seção inferior significa que o campo funct (bits 5-0) da instrução é 100010_{bin} e o campo op (bits 31-26) é 000000_{bin}. O valor de F1Pt na linha 2, coluna 1, é definido na Figura 3.20, no Capítulo 3. Blitz/gez é o op-code para quatro instruções encontradas no Apêndice A: blitz, bgez, bltzal e bgezal. O Capítulo 2 descreve as instruções indicadas com nome completo usando cor, enquanto o Capítulo 3 descreve as instruções indicadas em mnemônicos usando cor. O Apêndice A abrange todas as instruções.

DECODIFICANDO A LINGUAGEM DE MÁQUINA

EXEMPLO

Qual é a instrução em assembly correspondente a esta instrução de máquina?

00af8020_{hexa}

RESPOSTA

O primeiro passo na conversão de hexadecimal para binário é encontrar os campos op:

(Bits: 31 28 26)	5 2 0
0000 0000 1010 1111 1000 0000 0010 0000	

Examinamos o campo op para determinar a operação. Consultando a Figura 2.25, quando os bits 31-29 são 000 e os bits 28-26 são 000, essa é uma instrução no Formato R. Vamos reformatar a instrução binária para campos no Formato R, listado na Figura 2.26:

op	Rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

A parte inferior da Figura 2.25 determina a operação de uma instrução no Formato R. Nesse caso, os bits 5-3 são 100 e os bits 2-0 são 000, o que significa que esse padrão binário representa uma instrução add.

Decodificamos as demais instruções examinando os valores de campo. Os valores decimais são 5 para o campo rs, 15 para rt, 16 para rd (shamt não é usado). A Figura 2.18 diz que esses números representam os registradores \$a1, \$t7 e \$s0. Agora, podemos mostrar a instrução assembly:

add \$s0.\$a1,\$t7

A Figura 2.26 mostra todos os formatos de instrução do MIPS. A Figura 2.27 mostra o assembly do MIPS revelado no Capítulo 2; a outra parte ainda oculta das instruções MIPS trata principalmente de aritmética, abordada no próximo capítulo.

Nome	Campos						Comentários
Tamanho do campo	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Todas as instruções do MIPS têm 32 bits
Formato R	op	rs	rt	rd	shamt	funct	Formato das instruções aritméticas
Formato I	op	rs	rt	endereço/imediato			Formato imediato para transferências e desvios
Formato J	op	endereço de destino					Formato das instruções de jump

FIGURA 2.26 Formatos das instruções do MIPS no Capítulo 2. As partes destacadas mostram os formatos de instrução introduzidos nesta seção.

Verifique você mesmo

Qual é o intervalo de endereços para desvios condicionais no MIPS ($K = 1024$)?

- Endereços entre 0 e 64K – 1
- Endereços entre 0 e 256K – 1
- Endereços desde cerca de 32K antes do desvio até cerca de 32K depois
- Endereços desde cerca de 128K antes do desvio até cerca de 128K depois

Operandos MIPS

Nome	Exemplo	Comentários
32 registradores	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Locais rápidos para dados. No MIPS, os dados precisam estar em registradores para a realização de operações aritméticas. O registrador MIPS \$zero sempre é igual a 0. O registrador \$at é reservado para o montador tratar de constantes grandes.
2 ³⁰ words na memória	Memória[0], Memória[4]...., Memória[4294967292]	Acessadas apenas por instruções de transferência de dados no MIPS. O MIPS utiliza endereços em bytes, de modo que os endereços em words seqüenciais diferem em 4 vezes. A memória contém estruturas de dados, arrays e spilled registers, como aqueles salvos nas chamadas de procedimento.

Assembly do MIPS

Categoría	Instrucción	Exemplo	Significado	Comentários
Aritmética	add	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Três operandos; dados nos registradores
	subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	Três operandos; dados nos registradores
	add immediate	addi \$s1,\$s2,100	\$s1 = \$s2 + 100	Usada para somar constantes
Transferência de dados	load word	lw \$s1,100(\$s2)	\$s1 = Memória[\$s2 + 100]	Dados da memória para o registrador
	store word	sw \$s1,100(\$s2)	Memória[\$s2 + 100] = \$s1	Dados do registrador para a memória
	load half	lh \$s1,100(\$s2)	\$s1 = Memória[\$s2 + 100]	Halfword da memória para registrador
	store half	sh \$s1,100(\$s2)	Memória[\$s2 + 100] = \$s1	Halfword de um registrador para memória
	load byte	lb \$s1,100(\$s2)	\$s1 = Memória[\$s2 + 100]	Byte da memória para registrador
	store byte	sb \$s1,100(\$s2)	Memória[\$s2 + 100] = \$s1	Byte de um registrador para memória
	load upper immed.	lui \$s1,100	\$s1 = 100 * 2 ¹⁶	Carrega constante nos 16 bits mais altos
Lógica	and	and \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	Três operadores em registrador; AND bit a bit
	or	or \$s1,\$s2,\$s3	\$s1 = \$s2 \$s3	Três operadores em registrador; OR bit a bit
	nor	nor \$s1,\$s2,\$s3	\$s1 = -(\$s2 \$s3)	Três operadores em registrador; NOR bit a bit
	and immediate	andi \$s1,\$s2,100	\$s1 = \$s2 & 100	AND bit a bit registrador com constante
	or immediate	ori \$s1,\$s2,100	\$s1 = \$s2 100	OR bit a bit registrador com constante
	shift left logical	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	Deslocamento à esquerda por constante
	shift right logical	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	Deslocamento à direita por constante
Desvio condicional	branch on equal	beq \$s1,\$s2,25	if (\$s1 == \$s2) go to PC + 4 + 100	Testa igualdade; desvio relativo ao PC
	branch on not equal	bne \$s1,\$s2,25	if (\$s1 != \$s2) go to PC + 4 + 100	Testa desigualdade; relativo ao PC
	set on less than	slt \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compara menor que; usado com beq, bne
	set less than immediate	slti \$s1,\$s2,100	if (\$s2 < 100) \$s1 = 1; else \$s1 = 0	Compara menor que constante
Desvio incondicional	jump	j 2500	go to 10000	Desvia para endereço de destino
	jump register	jr \$ra	go to \$ra	Para switch e retorno de procedimento
	jump and link	jal 2500	\$ra = PC + 4. go to 10000	Para chamada de procedimento

FIGURA 2.27 Assembly do MIPS revelado no Capítulo 2. As partes destacadas mostram o que foi introduzido nas Seções 2.8 e 2.9.

Qual é o intervalo de endereços para jump e jump-and-link no MIPS ($M = 1024K$)?

- Endereços entre 0 e $64M - 1$
- Endereços entre 0 e $256M - 1$
- Endereços desde cerca de $32M$ antes do desvio até cerca de $32M$ depois
- Endereços desde cerca de $128M$ antes do desvio até cerca de $128M$ depois
- Qualquer lugar dentro de um bloco de $64M$ endereços, onde o PC fornece os 6 bits mais altos
- Qualquer lugar dentro de um bloco de $256M$ endereços, onde o PC fornece os 4 bits mais altos

Qual é a instrução em assembly do MIPS correspondente à instrução de máquina com o valor 0000 0000_{hexa}?

1. j
2. Formato R
3. addi
4. sll
5. mfc0
6. Opcode indefinido: não existe uma instrução válida que corresponda a 0.

2.10

Traduzindo e iniciando um programa

Esta seção descreve as quatro etapas para a transformação de um programa em C, armazenado em um arquivo no disco, para um programa executando em um computador. A Figura 2.28 mostra a hierarquia de tradução. Alguns sistemas combinam essas etapas para reduzir o tempo de tradução, mas elas são as quatro fases lógicas pelas quais os programas passam. Esta seção segue essa hierarquia de tradução.

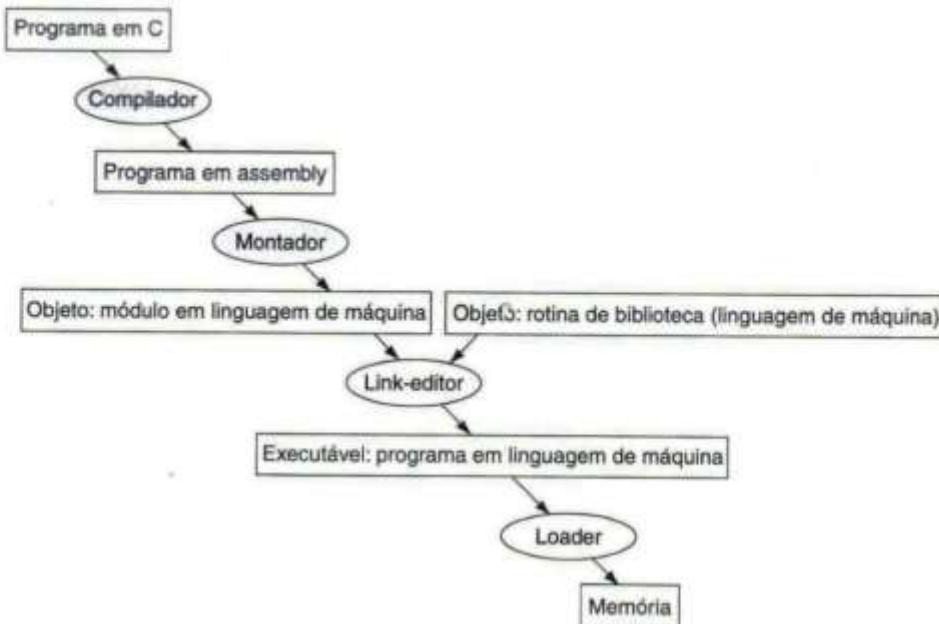


FIGURA 2.28 Uma hierarquia de tradução para a linguagem C. Um programa em linguagem de alto nível é inicialmente compilado para um programa em assembly, e depois montado em um módulo objeto em linguagem de máquina. O link-editor combina vários módulos com as rotinas de biblioteca para resolver todas as referências. O loader, então, coloca o código de máquina nos locais apropriados da memória, para ser executado pelo processador. Para agilizar o processo de tradução, algumas etapas são puladas ou combinadas. Alguns compiladores produzem módulos-objeto diretamente, e alguns sistemas utilizam loaders com link-editores, que realizam as duas últimas etapas. Para identificar o tipo de arquivo, o UNIX segue uma convenção de sufixo para os arquivos: os arquivos-fonte em C são chamados X.C, os arquivos em assembly são X.S, os arquivos-objeto são X.O, as rotinas de biblioteca link-editadas estaticamente são X.SO e os arquivos executáveis, como padrão, são chamados X.OUT. O MS-DOS utiliza os sufixos .C, .ASM, .OBJ, .LIB, .DLL e .EXE para indicar a mesma coisa.

Compilador

O compilador transforma o programa C em um *programa em assembly*, uma forma simbólica daquilo que a máquina entende. Os programas em linguagem de alto nível usam muito menos linhas de código do que o assembly, de modo que a produtividade do programador é muito mais alta.

Em 1975, muitos sistemas operacionais e montadores foram escritos em **assembly**, pois as memórias eram pequenas e os compiladores eram inefficientes. O aumento de 128.000 vezes na capacidade de memória em um único chip de DRAM reduz os problemas com tamanho do programa, e os compiladores otimizadores de hoje podem produzir programas em assembly quase tão bem quanto um especialista em assembly, e às vezes ainda melhores, para programas grandes.

assembly Uma linguagem simbólica, que pode ser traduzida para o formato binário.

Montador

Como dissemos na página 70, como o assembly é a interface com o software de nível superior, o montador também pode cuidar de variações comuns das instruções em linguagem de máquina como se fossem instruções propriamente ditas. O hardware não precisa implementar essas instruções; porém, seu aparecimento em assembly simplifica a tradução e a programação. Essas instruções são conhecidas como **pseudo-instruções**.

Como já dissemos, o hardware do MIPS garante que o registrador \$zero sempre tenha o valor 0. Ou seja, sempre que o registrador \$zero é utilizado, ele fornece um 0, e o programador não pode alterar o valor do registrador \$zero. O registrador \$zero é usado para criar a instrução move em assembly, que copia o conteúdo de um registrador para outro. Assim, o montador do MIPS aceita esta instrução, embora ela não se encontre na arquitetura do MIPS:

```
move $t0, $t1      # registrador $t0 recebe reg. $t1
```

O montador converte essa instrução em assembly para o equivalente em linguagem de máquina da seguinte instrução:

```
add $t0,$zero,$t1  # registrador $t0 recebe 0 + reg. $t1
```

O montador do MIPS também converte blt (branch on less than) para as duas instruções slt e bne mencionadas no exemplo da página 71. Outros exemplos são bgt, bge e ble. Ele também converte desvios a locais distantes para um desvio e um jump. Como já dissemos, o montador do MIPS permite que constantes de 32 bits sejam atribuídas a um registrador, apesar do limite de 16 bits das instruções imediatas.

Resumindo, as pseudoinstruções dão ao MIPS um conjunto mais rico de instruções assembly do que é implementado pelo hardware. O único custo é reservar um registrador, \$at, para ser usado pelo montador. Se você tiver de escrever programas em assembly, use pseudoinstruções para simplificar seu trabalho. Contudo, para entender a arquitetura do MIPS e ter certeza de que obterá o melhor desempenho, estude as instruções reais do MIPS, encontradas nas Figuras 2.25 e 2.27.

Os montadores também aceitarão números em diversas bases. Além de binário e decimal, eles normalmente aceitam uma base mais sucinta do que o binário, mas que seja convertida facilmente para um padrão de bits.

Esses recursos são convenientes, mas a tarefa principal de um montador é a montagem para código de máquina. O montador transforma o programa assembly em um *arquivo objeto*, que é uma combinação de instruções de **linguagem de máquina**, dados e informações necessárias para colocar instruções corretamente na memória.

Para produzir a versão binária de cada instrução no programa em assembly, o montador precisa determinar os endereços correspondentes a todos os rótulos. Os montadores registram os rótulos utilizados nos desvios e nas instruções de transferência de dados por meio de uma **tabela de símbolos**. Como você poderia esperar, a tabela contém pares de símbolo e endereço.

pseudo-instrução
Uma variação comum das instruções em assembly, normalmente tratada como se fosse uma instrução propriamente dita.

linguagem de máquina
Representação binária usada para a comunicação dentro de um sistema computacional.

tabela de símbolos
Uma tabela que combina nomes de rótulos aos endereços das words na memória ocupados pelas instruções.

O arquivo-objeto para os sistemas UNIX normalmente contém seis partes distintas:

- O *cabeçalho do arquivo objeto* descreve o tamanho e a posição das outras partes do arquivo objeto.
- O *segmento de texto* contém o código na linguagem de máquina.
- O *segmento de dados estáticos* contém os dados alocados por toda a vida do programa. (O UNIX permite que os programas usem *dados estáticos*, que são alocados para o programa inteiro, ou *dados dinâmicos*, que podem crescer ou diminuir conforme a necessidade do programa.)
- As *informações de relocação* identificam instruções e words de dados que dependem de endereços absolutos quando o programa é carregado na memória.
- A *tabela de símbolos* contém os rótulos restantes que não estão definidos, como referências externas.
- As *informações de depuração* contêm uma descrição resumida de como os módulos foram compilados, para o depurador poder associar as instruções de máquina aos arquivos-fonte em C e tornar as estruturas de dados legíveis.

A próxima subseção mostra como juntar rotinas que já foram montadas, como as rotinas de biblioteca.

Link-editor

O que apresentamos até aqui sugere que uma única mudança em uma linha de um procedimento exige a compilação e a montagem do programa inteiro. A tradução completa é um desperdício terrível de recursos computacionais. Essa repetição é um desperdício principalmente para rotinas de bibliotecas padrão, pois os programadores estariam compilando e montando rotinas que, por definição, quase nunca mudam. Uma alternativa é compilar e montar cada procedimento de forma independente, de modo que uma mudança em uma linha só exija a compilação e a montagem de um procedimento. Essa alternativa requer um novo programa de sistema, chamado **link-editor**, ou **linker**, que apinha todos os programas em linguagem de máquina montados independentes e os “remenda”.

Existem três etapas realizadas por um link-editor:

1. Colocar os módulos de código e dados simbolicamente na memória.
2. Determinar os endereços dos rótulos de dados e instruções.
3. Remendar as referências internas e externas.

O link-editor utiliza a informação de relocação e a tabela de símbolos em cada módulo objeto para resolver todos os rótulos indefinidos. Essas referências ocorrem em instruções de desvio e endereços de dados, de modo que a tarefa desse programa é muito semelhante à de um editor: ele encontra os endereços antigos e os substitui pelos novos. A edição é a origem do nome “link-editor”, ou linker para abreviar. O uso de um link-editor faz sentido porque é muito mais rápido remendar o código do que recompilá-lo e remontá-lo.

Se todas as referências externas forem resolvidas, o link-editor em seguida determina os locais da memória que cada módulo ocupará. Lembre-se de que a Figura 2.17 mostra a convenção do MIPS para alocação de programas e dados na memória. Como os arquivos foram montados isoladamente, o montador não poderia saber onde as instruções e os dados do módulo serão colocados em relação a outros módulos. Quando o link-editor coloca um módulo na memória, todas as referências *absolutas*, ou seja, endereços de memória que não são relativos a um registrador, precisam ser *relocadas* para refletir seu verdadeiro local.

O link-editor produz um **arquivo executável** que pode ser executado em um computador. Normalmente, esse arquivo possui o mesmo formato de um arquivo-objeto, exceto que não contém referências não resolvidas. É possível ter arquivos parcialmente link-editados, como rotinas de biblioteca, que ainda possuem endereços não resolvidos e, portanto, resultam em arquivos-objeto.

linker Também chamado **link-editor**, é um programa de sistema que combina programas em linguagem de máquina montados de maneira independente e traduz todos os rótulos indefinidos para um arquivo executável.

arquivo executável Um programa funcional no formato de um arquivo-objeto, que não contém referências não resolvidas, informações de relocação, tabela de símbolos ou informações de depuração.

LINK-EDIÇÃO DE ARQUIVOS-OBJETO

Link-edite os dois arquivos-objeto a seguir. Mostre os endereços atualizados das primeiras instruções do arquivo executável gerado. Mostramos as instruções em assembly só para tornar o exemplo compreensível; na realidade, as instruções seriam números.

Observe que, nos arquivos-objeto, destacamos os endereços e símbolos que precisam ser atualizados no processo de link-edição: as instruções que se referem a endereços dos procedimentos A e B e as instruções que se referem aos endereços das words de dados X e Y.

EXEMPLO

Cabeçalho do arquivo-objeto			
	Nome	Procedimento A	
	Tamanho do texto	100 _{hexa}	
	Tamanho dos dados	20 _{hexa}	
Segmento de texto	Endereço	Instrução	
	0	lw \$a0, 0(\$gp)	
	4	jal 0	
	
Segmento de dados	0	(X)	
	
Informações de relocação	Endereço	Tipo de instrução	Dependência
	0	lw	X
	4	jal	B
Tabela de símbolos	Rótulo	Endereço	
	X	-	
	B	-	

Cabeçalho do arquivo-objeto			
	Nome	Procedimento B	
	Tamanho do texto	200 _{hexa}	
	Tamanho dos dados	30 _{hexa}	
Segmento de texto	Endereço	Instrução	
	0	sw \$a1, 0(\$gp)	
	4	jal 0	
	
Segmento de dados	0	(Y)	
	
Informações de relocação	Endereço	Tipo de instrução	Dependência
	0	sw	Y
	4	jal	A
Tabela de símbolos	Rótulo	Endereço	
	Y	-	
	A	-	

O procedimento A precisa encontrar o endereço para a variável cujo rótulo é X para colocá-lo na instrução load e encontrar o endereço do procedimento B para colocá-lo na instrução jal. O procedimento B precisa do endereço da variável cujo rótulo é Y para a instrução store e o endereço do procedimento A para sua instrução jal.

Pela Figura 2.17, sabemos que o segmento de texto começa no endereço 40 0000_{hexa}, e o segmento de dados em 1000 0000_{hexa}. O texto do procedimento A é colocado no primeiro endereço, e seus dados no segundo. O cabeçalho do arquivo-objeto para o procedimento A diz que seu texto possui 100_{hexa} bytes, e seus dados possuem 20_{hexa} bytes, de modo que o endereço inicial para o texto do procedimento B é 40 0100_{hexa}, e seus dados começam em 1000 0020_{hexa}.

RESPOSTA

Cabeçalho do arquivo executável		
	Tamanho do texto	300 _{hexa}
	Tamanho dos dados	50 _{hexa}
Segmento de texto	Endereço	Instrução
	0040 0000 _{hexa}	lw \$a0, 8000 _{hexa} (\$gp)
	0040 0004 _{hexa}	jal 40 0100 _{hexa}
	0040 0100 _{hexa}	sw \$a1, 8020 _{hexa} (\$gp)
	0040 0104 _{hexa}	jal 40 0000 _{hexa}
	...	
Segmento de dados	Endereço	
	1000 0000 _{hexa}	(X)
	...	
	1000 0020 _{hexa}	(Y)
	...	

Agora, o link-editor atualiza os campos de endereço das instruções. Ele usa o campo de tipo de instrução para saber o formato do endereço a ser editado. Temos dois tipos aqui:

1. Os jal são fáceis porque utilizam o endereçamento pseudodireto. O jal no endereço 40 0004_{hexa} recebe 40 0100_{hexa} (o endereço do procedimento B) em seu campo de endereço, e o jal em 40 0104_{hexa} recebe 40 0000_{hexa} (o endereço do procedimento A) em seu campo de endereço.
2. Os endereços de load e store são mais difíceis pois são relativos a um registrador de base. Este exemplo utiliza o ponteiro global como registrador de base. A Figura 2.17 mostra que \$gp é inicializado com 1000 8000_{hexa}. Para obter o endereço 1000 0000_{hexa} (o endereço da word X), colocamos 8000_{hexa} no campo de endereço da instrução lw, no endereço 40 0000_{hexa}. O Capítulo 3 explica a aritmética em complemento a dois em 16 bits, motivo pelo qual 8000_{hexa} no campo de endereço gera 1000 0000_{hexa} como endereço. De modo semelhante, colocamos 8020_{hexa} no campo de endereço da instrução sw no endereço 40 0100_{hexa} para obter o endereço 1000 0020_{hexa} (o endereço da word Y).

Loader

Agora que o arquivo executável está no disco, o sistema operacional o lê para a memória e o inicia. Ele segue estas etapas nos sistemas UNIX:

1. Lê o cabeçalho do arquivo executável para determinar o tamanho dos segmentos de texto e de dados.
2. Cria um espaço de endereçamento grande o suficiente para o texto e os dados.
3. Copia as instruções e os dados do arquivo executável para a memória.
4. Copia os parâmetros (se houver) do programa principal para a pilha.
5. Inicializa os registradores da máquina e define o stack pointer para o primeiro local livre.
6. Desvia para uma rotina de partida, que copia os parâmetros para os registradores de argumento e chama a rotina principal do programa. Quando a rotina principal retorna, a rotina de partida termina o programa com uma chamada ao sistema exit.

As Seções A.3 e A.4 no Apêndice A descrevem os link-editores e os loaders com mais detalhes.

loader Um programa de sistema que coloca o programa-objeto na memória principal, para que esteja pronto para ser executado.

Dynamically Linked Libraries (DLLs)

A primeira parte desta seção descreve a técnica tradicional para a link-edição de bibliotecas antes de o programa ser executado. Embora essa técnica estática seja o modo mais rápido de chamar rotinas de biblioteca, ela possui algumas desvantagens:

- As rotinas de biblioteca se tornam parte do código executável. Se uma nova versão da biblioteca for lançada para reparar os erros ou dar suporte a novos dispositivos de hardware, o programa link-editado estaticamente continua usando a versão antiga.
- Ela carrega a biblioteca inteira, mesmo que apenas parte da biblioteca seja usada quando o programa for executado. A biblioteca pode ser grande em relação ao programa; por exemplo, a biblioteca padrão da linguagem C possui 2,5MB.

Essas desvantagens levaram às Dynamic Link Libraries (DLLs), nas quais as rotinas da biblioteca não são link-editadas e carregadas até que o programa seja executado. Tanto o programa quanto as rotinas da biblioteca mantêm informações extras sobre a localização dos procedimentos não locais e seus nomes. Na versão inicial das DLLs, o loader executava um link-editor dinâmico, usando as informações extras no arquivo para descobrir as bibliotecas apropriadas e atualizar todas as referências externas.

A desvantagem da versão inicial das DLLs era que ela ainda link-editava todas as rotinas da biblioteca que poderiam ser chamadas, quando apenas algumas são chamadas durante a execução do programa. Essa observação levou à versão da link-edição de procedimento tardio das DLLs, no qual cada rotina só é link-editada *depois* de chamada.

Como em muitos outros casos, esse truque conta com um certo nível de indireção. A Figura 2.29 mostra a técnica. Ela começa com as rotinas não locais chamando um conjunto de rotinas fictícias no final do programa, com uma entrada por rotina não local. Essas entradas fictícias contêm, cada uma, um jump indireto.

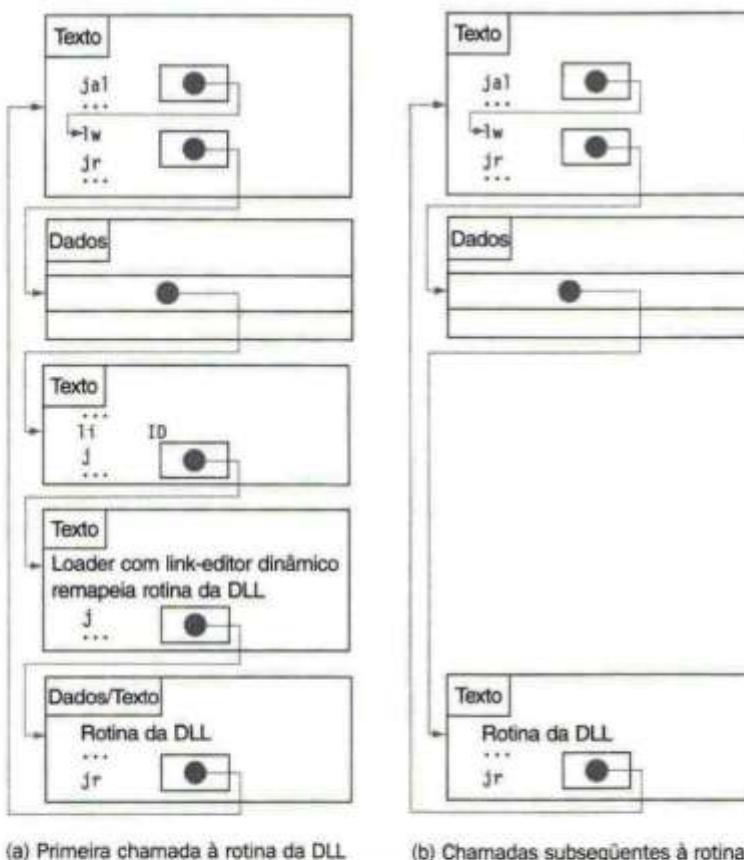


FIGURA 2.29 DLL por meio da link-edição de procedimento tardio. (a) Etapas para a primeira vez em que uma chamada é feita à rotina da DLL. (b) As etapas para encontrar a rotina, remapeá-la e link-editá-la são puladas em chamadas subseqüentes. Conforme veremos no Capítulo 7, o sistema operacional pode evitar copiar a rotina desejada remapeando-a por meio do gerenciamento de memória virtual.

Na primeira vez em que a rotina da biblioteca é chamada, o programa chama a entrada fictícia e segue o jump indireto. Ele aponta para o código que coloca um número em um registrador para identificar a rotina de biblioteca desejada e depois desvia para o loader com link-editor dinâmico. O loader com link-editor encontra a rotina desejada, remapeia essa rotina e altera o endereço do desvio indireto, para que aponte para essa rotina. Depois, ele desvia para ela. Quando a rotina termina, ele retorna ao local de chamada original. Depois disso, ele desvia diretamente para a rotina, sem os desvios extras.

Resumindo, as DLLs exigem espaço extra para as informações necessárias à link-edição dinâmica, mas não exigem que as bibliotecas inteiras sejam copiadas ou link-editadas. Elas realizam muito trabalho extra na primeira vez em que uma rotina é chamada, mas executam somente um desvio indireto depois disso. Observe que o retorno da biblioteca não realiza trabalho extra. O Microsoft Windows conta bastante com as DLLs dessa forma, e esse também é um modo normal de executar programas nos sistemas UNIX atuais.

Iniciando um programa Java

A discussão anterior captura o modelo tradicional de execução de um programa, no qual a ênfase está no tempo de execução rápido para um programa voltado para uma arquitetura específica, ou mesmo para uma implementação específica dessa arquitetura. Na verdade, é possível executar programas Java da mesma forma que programas C. No entanto, Java foi inventada com objetivos diferentes. Um deles era funcionar rapidamente e de forma segura em qualquer computador, mesmo que isso pudesse aumentar o tempo de execução.

A Figura 2.30 mostra as etapas típicas de tradução e execução para os programas em Java. Em vez de compilar para assembly de um computador de destino, Java é compilado primeiro para instruções fáceis de interpretar: o conjunto de instruções do **bytecode Java**. Esse conjunto de instruções foi criado para ser próximo da linguagem Java, de modo que essa etapa de compilação seja trivial. Praticamente nenhuma otimização é realizada. Assim como o compilador C, o compilador Java verifica os tipos dos dados e produz a operação apropriada a cada tipo. Os programas em Java são distribuídos na versão binária desses bytecodes.

Um interpretador Java, chamado **JVM (Java Virtual Machine)**, pode executar os bytecodes Java. Um interpretador é um programa que simula um conjunto de instruções. Por exemplo, o simulador do MIPS usado com este livro é um interpretador. Não é necessária uma etapa de montagem separada, pois ou a tradução é tão simples que o compilador preenche os endereços ou a JVM os encontra durante a execução.

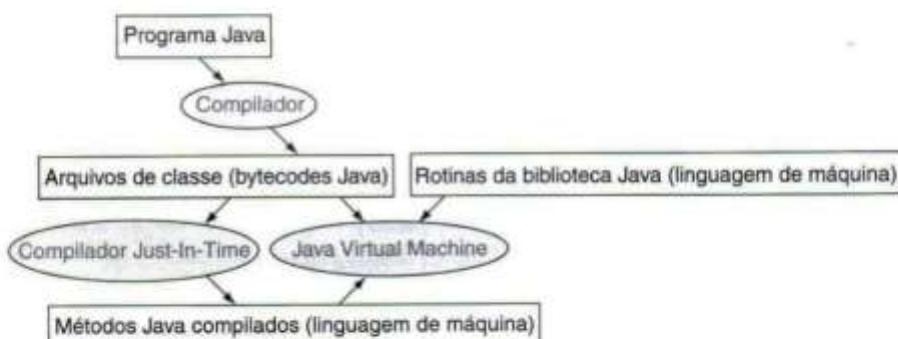


FIGURA 2.30 Uma hierarquia de tradução para Java. Um programa em Java primeiro é compilado para uma versão binária dos bytecodes Java, com todos os endereços definidos pelo compilador. O programa em Java agora está pronto para ser executado no interpretador, chamado **JVM (Java Virtual Machine – máquina virtual Java)**. A JVM link-edita os métodos desejados na biblioteca Java enquanto o programa está sendo executado. Para conseguir melhor desempenho, a JVM pode chamar o compilador JIT (Just-In-Time), que compila os métodos seletivamente para a linguagem nativa da máquina em que está executando.

bytecode Java

Instruções que formam programas em Java, escritas em um conjunto de instruções elaborado para ser interpretado.

JVM (Java Virtual Machine)

O programa que interpreta os bytecodes Java.

A vantagem da interpretação é a portabilidade. A disponibilidade das máquinas virtuais Java em software significou que muitos puderam escrever e executar programas Java pouco tempo depois que o Java foi anunciado. Hoje, as máquinas virtuais Java são encontradas em milhões de dispositivos, em tudo desde telefones celulares até navegadores da Internet.

A desvantagem da interpretação é o desempenho fraco. Os avanços incríveis no desempenho dos anos 80 e 90 do século passado tornaram a interpretação viável para muitas aplicações importantes, mas um fator de atraso de 10 vezes, em comparação com os programas C compilados tradicionalmente, tornou o Java pouco atraente para algumas aplicações.

Para preservar a portabilidade e melhorar a velocidade de execução, a fase seguinte do desenvolvimento do Java foram compiladores que traduziam *enquanto* o programa estava sendo executado. Esses compiladores **Just-In-Time (JIT)** normalmente traçam o perfil do programa em execução para descobrir onde estão os métodos “quentes”, e depois os compilam para o conjunto de instruções nativo em que a máquina virtual está executando. A parte compilada é salva para a próxima vez em que o programa for executado, de modo que possa ser executado mais rapidamente cada vez que for executado. Esse equilíbrio entre interpretação e compilação evolui com o tempo, de modo que os programas Java executados com frequência sofrem muito pouco com o trabalho extra da interpretação.

À medida que os computadores ficam mais rápidos, de modo que os compiladores possam fazer mais, e os pesquisadores inventam melhores meios de compilar Java durante a execução, a lacuna de desempenho entre Java e C ou C++ está se fechando. A Seção 2.14 contém muito mais detalhes sobre a implementação do Java, dos bytecodes Java, da JVM e dos compiladores JIT.

Compilador
Just-In-Time (JIT) O nome normalmente dado a um compilador que opera durante a execução, traduzindo os segmentos de código interpretados para o código nativo do computador.

Qual das vantagens de um interpretador em relação a um tradutor você acredita que tenha sido mais importante para os criadores do Java?

Verifique você mesmo

1. Facilidade de escrita de um interpretador
2. Melhores mensagens de erro
3. Código-objeto menor
4. Independência de máquina

2.11

Como os compiladores otimizam

Como o compilador afetará significativamente o desempenho de um computador, compreender a tecnologia do compilador hoje é fundamental para compreender o desempenho. A finalidade desta seção é oferecer uma idéia geral das otimizações que um compilador utiliza para conseguir mais desempenho. A próxima seção apresenta a anatomia interna de um compilador. Para começar, a Figura 2.31 mostra a estrutura dos compiladores recentes, e descrevemos as otimizações na ordem dos passos dessa estrutura.

Otimizações de alto nível

As otimizações de alto nível são transformações feitas em algo próximo da fonte.

A transformação de alto nível mais comum provavelmente é a utilização de *procedimentos inline*, que substitui uma chamada a uma função pelo corpo da função, substituindo os argumentos do caller pelos parâmetros do procedimento. Outras otimizações de alto nível envolvem transformações em loops que podem reduzir o trabalho adicional do loop, melhorar o acesso à memória e explorar o hardware com mais eficiência. Por exemplo, em loops que executam muitas iterações, como aqueles tradicionalmente controlados por uma instrução *for*, a otimização por **desdobramento de loop (loop unrolling)** normal-

desdobramento de loop (loop unrolling) Uma técnica para conseguir mais desempenho dos loops que acessam arrays, na qual são feitas várias cópias do corpo do loop e instruções de diferentes iterações são agrupadas.



FIGURA 2.31 A estrutura de um compilador otimizador moderno consiste em uma série de passos ou fases. Logicamente, cada passo pode ser imaginado como a execução até o fim antes do próximo passo. Na prática, alguns passos podem tratar de um procedimento por vez, basicamente intercalando com outro passo.

mente é mais útil. Desdobrar o loop significa replicar o corpo do loop várias vezes, executando-o transformado um número menor de vezes. O desdobramento do loop reduz o trabalho adicional de loop e oferece oportunidades para muitas outras otimizações. Outros tipos de transformações de alto nível incluem transformações de loop sofisticadas, como o intercâmbio de loops aninhados e bloqueio de loops para obter melhor comportamento da memória; veja alguns exemplos no Capítulo 7.

Otimizações locais e globais

Dentro do passo dedicado à otimização local e global, são realizadas três classes de otimizações:

1. A *otimização local* atua dentro de um único bloco básico. Um passo de otimização local normalmente é executado como um precursor e um sucessor da otimização global, para “limpar” o código antes e depois da otimização global.
2. A *otimização global* atua entre vários blocos básicos; veremos um exemplo disso em breve.
3. A *alocação de registradores* global aloca variáveis aos registradores para regiões do código. A alocação de registradores é fundamental para se obter um bom desempenho nos processadores modernos.

Várias otimizações são realizadas local e globalmente, incluindo a eliminação de subexpressões comuns, propagação de constante, propagação de cópia, eliminação de local de armazenamento morto e redução de força. Vejamos alguns exemplos simples dessas otimizações.

A *eliminação de subexpressões comuns* encontra várias ocorrências da mesma expressão e substitui a segunda por uma referência à primeira. Considere, por exemplo, um segmento de código para somar 4 a um elemento do array:

$$x[i] = x[i] + 4$$

O cálculo de endereço para $x[i]$ ocorre duas vezes e é idêntico, pois nem o endereço inicial de x nem o valor de i muda. Assim, o cálculo pode ser reutilizado. Vejamos o código intermediário para esse fragmento, pois ele permite a realização de várias outras otimizações. Aqui está o código intermediário não otimizado à esquerda, e o código com eliminação de subexpressões comuns à direita, substituindo o segundo cálculo do endereço pelo primeiro. Observe que a alocação de registradores ainda não ocorreu, de modo que o compilador está usando números de registradores virtuais, como R10 neste caso.

```
# x[i] + 4          # x[i] + 4
li R100,x          li R100,x
lw R101,i          lw R101,i
mult R102,R101,4   mult R102,R101,4
add R103,R100,R102 add R103,R100,R102
lw R104,0(R103)    lw R104,0(R103)
# valor de x[i] em R104 # valor de x[i] em R104
add R105,R104,4   add R105,R104,4
# x[i] =           # x[i] =
li R106,x          sw R105,0(R103)
lw R107,i          mult R108,R107,4 ~
add R109,R106,R107
sw R105,0(R109)
```

Se a mesma otimização fosse possível entre dois blocos básicos, esse seria um caso de *eliminação de subexpressões comuns global*. Vamos considerar algumas das outras otimizações:

- *Redução de força* substitui operações complexas por outras mais simples e pode ser aplicada a este segmento de código, substituindo o *mult* por um deslocamento à esquerda.
- *Propagação de constante* e sua correspondente *junção de constante* encontram constantes no código e as propagam, encolhendo os valores de constante sempre que possível.
- *Propagação de cópia* propaga valores que são cópias simples, eliminando a necessidade de ler valores e possivelmente habilitando outras otimizações, como a eliminação de subexpressões comuns.
- *Eliminação de local de armazenamento morto* encontra locais com valores que não são usados novamente e elimina o local; seu “primo” é a *eliminação de código morto*, que encontra código não utilizado – código que não pode afetar o resultado final do programa – e o elimina. Com o uso intenso de macros, modelos e técnicas semelhantes, criadas para reutilizar o código nas linguagens de alto nível, o código morto ocorre com uma freqüência surpreendente.

Entendendo o desempenho dos programas

Os programadores preocupados com o desempenho de loops críticos, especialmente em aplicações de tempo real ou embutidas, normalmente se vêem examinando o assembly produzido por um compilador e questionando por que o compilador deixou de realizar alguma otimização global ou de alocar uma variável a um registrador no decorrer de um loop. A resposta normalmente está no fato de o compilador ser conservador. A oportunidade para melhorar o código pode parecer óbvia ao programador, mas, nesse caso, o programador geralmente conhece algo que o compilador não sabe, como a ausência de dois ponteiros apontando para uma mesma posição, ou a ausência de efeitos colaterais causados pela chamada de uma função. Na verdade, o compilador pode ser capaz de realizar a transformação com um pouco de ajuda, o que eliminaria o comportamento de pior caso que ele precisa assumir. Essa idéia também ilustra uma observação importante: os programadores que utilizam ponteiros para tentar melhorar o desempenho no acesso a variáveis, especialmente ponteiros para valores na pilha que também possuem nomes, como variáveis ou como elementos de arrays, provavelmente inabilitam muitas otimizações do compilador. O resultado final é que o código de baixo nível para o ponteiro pode não executar melhor, ou executar ainda pior do que o código de alto nível, otimizado pelo compilador.

Os compiladores precisam ser *conservadores*. A primeira tarefa de um compilador é produzir código correto; sua segunda tarefa normalmente é produzir código rápido, embora outros fatores (como o tamanho do código) às vezes também possam ser importantes. Um código que seja rápido mas incorreto – por qualquer combinação possível de entradas – está simplesmente errado. Assim, quando dizemos que um compilador é “conservador”, estamos dizendo que ele realiza uma otimização somente se souber com toda a certeza que, não importando quais sejam as entradas, o código funcionará conforme o usuário o escreveu. Como a maioria dos compiladores traduz e optimiza uma função ou procedimento de cada vez, a maioria deles, especialmente nos níveis de optimização mais baixos, considera o pior sobre chamadas de função e sobre seus parâmetros.

Otimizações globais de código

Muitas optimizações de código globais possuem os mesmos objetivos daquelas usadas no caso local, incluindo a eliminação de subexpressões comuns, propagação de constante, propagação de cópia e eliminação de local de armazenamento morto e código morto.

Existem duas outras optimizações globais importantes: movimentação de código e eliminação de variável de indução. Ambas são optimizações de loop; ou seja, elas visam ao código dentro dos loops. A *movimentação de código* encontra o código que é invariante no loop: um trecho de código específico calcula o mesmo valor em cada iteração do loop e, portanto, pode ser calculado uma única vez, fora do loop. A *eliminação de variável de indução* é uma combinação de transformações que reduz o trabalho adicional na indexação de arrays, basicamente substituindo a indexação do array por acessos por meio de ponteiros. Em vez de examinar a eliminação de variável de indução em profundidade, indicamos a leitura da Seção 2.15, que compara o uso da indexação de array com os ponteiros; para a maioria dos loops, a transformação do código de array (mais óbvio) para o código de ponteiro pode ser realizada por um compilador optimizador moderno.

Resumo de optimização

A Figura 2.32 mostra exemplos de optimizações típicas, e a última coluna indica onde a optimização é realizada no compilador gcc. Às vezes, é difícil separar algumas das optimizações mais simples – optimizações dependentes de local e processador – das transformações feitas no gerador de código, e algumas optimizações são feitas várias vezes, especialmente optimizações locais, que podem ser realizadas antes e depois da optimização global, bem como durante a geração de código.

Interface hardware/software

Hoje em dia, basicamente toda a programação para aplicações de desktop e servidor é feita em linguagens de alto nível, assim como a maioria da programação para aplicações embutidas. Esse desenvolvimento significa que, como a maioria das instruções executadas vêm da saída de um compilador, um conjunto de instruções é basicamente um alvo do compilador. Com a Lei de Moore vêm a tentação de acrescentar operações sofisticadas em um conjunto de instruções. O desafio é que elas podem não combinar exatamente com o que o compilador precisa produzir, ou podem ser tão genéricas que não sejam velozes. Por exemplo, considere as instruções especiais de loop encontradas em alguns computadores. Suponha que, em vez de decrementar por um, o compilador quisesse incrementar por quatro, ou, em vez de desviar se for diferente de zero, o compilador quisesse desviar se o índice fosse menor ou igual ao limite. A instrução de loop pode ser incompatível. Quando encara tais objeções, o projetista do conjunto de instruções pode, então, generalizar a operação, acrescentando outro operando para especificar o incremento e talvez uma opção sobre qual condição de desvio deve ser utilizada. Nesse caso, o perigo é de que uma operação comum, digamos, incrementar por um, seja mais lenta do que uma seqüência de operações simples.

Nome da otimização	Explicação	Nível gcc
Alto nível	No nível de fonte ou próximo dele; independente de processador	
Integração de procedimento	Substitui chamada de procedimento pelo corpo do procedimento	O3
Local	Diretamente dentro do código	
Eliminação de subexpressões comuns	Substitui duas instâncias do mesmo cálculo por única cópia	O1
Propagação de constante	Substitui todas as instâncias de uma variável para a qual é atribuída uma constante pela própria constante	O1
Redução de tamanho da pilha	Reorganiza a árvore de expressões para minimizar os recursos necessários para a avaliação das mesmas	O1
Global	Atravessando fronteiras	
Eliminação de subexpressões comuns global	O mesmo que local, mas esta versão cruza fronteiras	O2
Propagação de cópia	Substitui todas as ocorrências de uma variável A para a qual tenha sido atribuído X (ou seja, A = X) por X	O2
Movimentação de código	Em um loop, remove código que calcula o mesmo valor a cada iteração do loop	O2
Eliminação de variável de indução	Simplifica/elimina os cálculos de endereçamento de array dentro dos loops	O2
Dependente do processador	Depende do conhecimento do processador	
Redução de força	Muitos exemplos; substitui multiplicação por deslocamentos à esquerda	O1
Escalonamento de pipeline	Reordena instruções para melhorar o desempenho do pipeline	O1
Otimização de offset do desvio	Escolhe o desvio de caminho mais curto para alcançar o alvo	O1

Figura 2.32 Principais tipos de otimizações e exemplos em cada classe. A terceira coluna mostra quando ocorrem em diferentes níveis de otimização no gcc. A organização Gnu chama os três níveis de otimização de médio (O1), total (O2) e total com integração de pequenos procedimentos (O3).

2.12

Como os compiladores funcionam: uma introdução

A finalidade desta seção é oferecer uma rápida introdução à função do compilador, que ajudará o leitor a entender como ele traduz um programa em linguagem de alto nível para instruções de máquina. Lembre-se de que o assunto de construção de compilador normalmente é ensinado em um curso de um ou dois semestres; nossa introdução necessariamente só abordará os fundamentos. O restante desta seção se encontra no CD.

2.13

Um exemplo de ordenação em C para juntar tudo isso

Um perigo de mostrar o código em assembly em pedaços é que você não terá idéia de como se parece um programa inteiro em assembly. Nesta seção, deduzimos o código do MIPS a partir de dois procedimentos escritos em C: um para trocar elementos do array e um para ordená-los.

O procedimento swap

Vamos começar com o código para o procedimento swap na Figura 2.33. Esse procedimento simplesmente troca os conteúdos de duas posições de memória. Ao traduzir de C para assembly manualmente, seguimos estas etapas gerais:

1. Alocar registradores a variáveis do programa.
2. Produzir código para o corpo do procedimento.
3. Preservar registradores durante a chamada do procedimento.

Esta seção descreve o procedimento swap nessas três partes, concluindo com a junção de todas as partes.

Alocação de registradores para swap

Como mencionamos na página 58, a convenção do MIPS sobre passagem de parâmetros é usar os registradores \$a0, \$a1, \$a2 e \$a3. Como swap tem apenas dois parâmetros, v e k, eles serão encontrados nos registradores \$a0 e \$a1. A única outra variável é temp, que associamos com o registrador \$t0, pois swap é um procedimento folha (ver página 61). Essa alocação de registradores corresponde às declarações de variável na primeira parte do procedimento swap da Figura 2.33.

```
void swap(int v[ ], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Figura 2.33 Um procedimento em C que troca o conteúdo de duas posições de memória. A próxima subseção utiliza esse procedimento em um exemplo de ordenação.

Código do corpo do procedimento swap

As linhas restantes do código em C do swap são

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Lembre-se de que o endereço de memória para o MIPS refere-se ao endereço em *bytes*, e, por isso, as words, na realidade, estão afastadas por 4 bytes. Logo, precisamos multiplicar o índice k por 4 antes de somá-lo ao endereço. *Esquecer que os endereços de words seqüenciais diferem em 4, em vez de 1, é um erro comum na programação em assembly.* Logo, o primeiro passo é obter o endereço de v[k] multiplicando k por 4:

```
sll    $t1,$a1,2      # registrador $t1 = k * 4
add    $t1,$a0,$t1    # registrador $t1 = v + (k * 4)
                    # registrador $t1 tem o endereço de v[k]
```

Agora, lemos v[k] para \$t1, e depois v[k+1] somando 4 a \$t1:

```
lw     $t0, 0($t1)    # registrador $t0 (temp) = v[k]
lw     $t2, 4($t1)    # registrador $t2 = v[k + 1]
                    # refere-se ao próximo elemento de v
```

Agora, armazenamos \$t0 e \$t2 nos endereços trocados:

```
sw     $t2, 0($t1)    # v[k] = registrador $t2
sw     $t0, 4($t1)    # v[k+1] = registrador $t0 (temp)
```

Até agora, alocamos registradores e escrevemos o código para realizar as operações do procedimento. O que está faltando é o código para preservar os registradores salvos usados dentro do swap. Como não estamos usando registradores salvos nesse procedimento folha, não há nada para preservar.

O procedimento swap completo

Agora, estamos prontos para a rotina inteira, que inclui o rótulo do procedimento e o jump de retorno. Para facilitar o acompanhamento, identificamos na Figura 2.34 cada bloco de código com sua finalidade no procedimento.

Corpo do procedimento			
swap:	\$t1	\$t1, \$a1, 2	# reg \$t1 = k * 4
	add	\$t1, \$a0, \$t1	# reg \$t1 = v + (k * 4)
			# reg \$t1 tem o endereço de v[k]
	lw	\$t0, 0(\$t1)	# reg \$t0 (temp) = v[k]
	lw	\$t2, 4(\$t1)	# reg \$t2 = v[k + 1]
			# refere-se ao próximo elemento de v
	sw	\$t2, 0(\$t1)	# v[k] = reg \$t2
	sw	\$t0, 4(\$t1)	# v[k+1] = reg \$t0 (temp)

Retorno do procedimento		
	jr	\$ra
		# retorna à rotina que chamou

FIGURA 2.34 Código assembly do MIPS do procedimento swap na Figura 2.33.

O procedimento sort

Para garantir que você apreciará o rigor da programação em assembly, vamos experimentar um segundo exemplo, maior. Nesse caso, montaremos uma rotina que chama o procedimento swap. Esse programa ordena um array de inteiros, usando ordenação por trocas, que é uma das ordenações mais simples, mas não a mais rápida. A Figura 2.35 mostra a versão em C do programa. Mais uma vez, apresentamos esse procedimento em várias etapas, concluindo com o procedimento 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 2.35 Um procedimento em C que realiza uma ordenação no array v.

Alocação de registradores para sort

Os dois parâmetros do procedimento sort, v e n, estão nos registradores de parâmetro \$a0 e \$a1, e alocamos o registrador \$s0 a i e o registrador \$s1 a j.

Código para o corpo do procedimento sort

O corpo do procedimento consiste em dois loops *for* aninhados e uma chamada a swap que inclui parâmetros. Vamos desembrulhar o código de fora para o meio.

O primeiro passo de tradução é o primeiro loop *for*:

```
for (i = 0; i < n; i += 1) {
```

Lembre-se de que a instrução *for* em C possui três partes: inicialização, teste de loop e incremento da iteração. É necessária apenas uma instrução para inicializar *i* como 0, a primeira parte da instrução *for*:

```
move $s0, $zero      # i = 0
```

(Lembre-se de que *move* é uma pseudo-instrução fornecida pelo montador para a conveniência do programador em assembly; ver página 79.) Também é necessária apenas uma instrução para incrementar *i*, a última parte da instrução *for*:

```
addi $s0, $s0, 1      # i += 1
```

O loop deverá terminar se $i < n$ não for verdadeiro ou, em outras palavras, deverá terminar se $i \geq n$. A instrução *set on less than* atribui 1 ao registrador *\$t0* se $\$s0 < \$a1$; caso contrário, ele é 0. Como queremos testar se $\$s0 \geq \$a1$, desviamos se o registrador *\$t0* for zero. Esse teste utiliza duas instruções:

```
forltst:slt $t0, $s0, $a1      # reg. $t0 = 0 se $s0 \geq \$a1 \geq (in)
beq $t0, $zero, exit1 # vai para exit1 se $s0 \geq \$a1 \geq (in)
```

O final do loop só desvia de volta para o teste do loop:

```
j forltst      # desvia para o teste do loop mais externo
exit1:
```

O código da estrutura do primeiro loop *for* é, então,

```
move $s0, $zero      # i = 0
forltst:slt $t0, $s0, $a1      # reg. $t0 = 0 se $s0 \geq \$a1 (in)
beq $t0, $zero, exit1 # vai para exit1 se $s0 \geq \$a1 (in)
...
(corpo do primeiro loop for)
...
addi $s0, $s0, 1      # i += 1
j forltst      # salta para teste do loop externo
exit1:
```

Voilà! O exercício 2.8 explora a escrita de código mais rápido para loops semelhantes.

O segundo loop *for* se parece com o seguinte em C:

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
```

A parte de inicialização desse loop novamente é uma instrução:

```
addi $s1, $s0, -1 # j = i - 1
```

O decremente de *j* no final do loop também tem uma instrução:

```
addi $s1, $s1, -1 # j -= 1
```

O teste do loop possui duas partes. Saímos do loop se a condição falhar, de modo que o primeiro teste precisa terminar o loop se falhar ($j < 0$):

```
for2tst:slti $t0, $s1, 0      # reg. $t0 = 1 se $s1 < 0 (j < 0)
bne $t0, $zero, exit2 # vai para exit2 se $s1 < 0 (j < 0)
```

Esse desvio pulará o segundo teste de condição. Se não pular, então $j \geq 0$.

O segundo teste termina se $v[j] > v[j + 1]$ não for verdadeiro, ou seja, termina se $v[j] \leq v[j + 1]$. Primeiro, criamos o endereço multiplicando *j* por 4 (pois precisamos de um endereço em bytes) e somamos ao endereço base de *v*:

```
sll $t1, $s1, 2      # reg. $t1 = j * 4
add $t2, $a0, $t1      # reg. $t2 = v + (j * 4)
```

Agora, lemos o conteúdo de $v[j]$:

```
lw    $t3, 0($t2)  # reg. $t3 = v[j]
```

Como sabemos que o segundo elemento é exatamente a word seguinte, somamos 4 ao endereço no registrador $\$t2$ para obter $v[j + 1]$:

```
lw    $t4, 4($t2)  # reg. $t4 = v[j + 1]
```

O teste de $v[j] \leq v[j + 1]$ é o mesmo que $v[j + 1] \geq v[j]$, de modo que as duas instruções do teste de saída são

```
slt  $t0, $t4, $t3      # reg. $t0 = 0 se $t4 \geq $t3  
beq  $t0, $zero, exit2  # desvia para exit2 se $t4 \geq $t3
```

O final do loop desvia de volta para o teste do loop interno:

```
j     for2tst  # desvia para o restante do loop interno
```

Combinando as partes, a estrutura do segundo loop *for* se parece com o seguinte:

```
addi  $s1, $s0, -1      # j = i - 1  
for2tst: sli $t0, $s1, 0  # reg. $t0 = 1 se $s1 < 0 (j < 0)  
bne   $t0, $zero, exit2  # vai para exit2 se $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 se $t4 \geq $t3  
beq   $t0, $zero, exit2  # desvia para exit2 se $t4 \geq $t3  
...  
(corpo do segundo loop for)  
...  
addi $s1, $s1, -1  # j = i - 1  
j     for2tst  # desvia para o teste do loop interno  
exit2:
```

A chamada de procedimento em sort

A próxima etapa é o corpo do segundo loop *for*:

```
swap(v,j);
```

Chamar *swap* é muito fácil:

```
jal    swap
```

Passando parâmetros em sort

O problema vem quando queremos passar parâmetros, porque o procedimento *sort* precisa dos valores nos registradores $\$a0$ e $\$a1$, enquanto o procedimento *swap* precisa que seus parâmetros sejam colocados nesses mesmos registradores. Uma solução é copiar os parâmetros para *sort* em outros registradores antes do procedimento, deixando os registradores $\$a0$ e $\$a1$ disponíveis para a chamada de *swap*. (Essa cópia é mais rápida do que salvar e restaurar na pilha.) Primeiro, copiamos $\$a0$ e $\$a1$ para $\$s2$ e $\$s3$ durante o procedimento:

```
move  $s2, $a0      # copia parâmetro $a0 para $s2  
move  $s3, $a1      # copia parâmetro $a1 para $s3
```

Depois, passamos os parâmetros para swap com estas duas instruções:

```
move    $a0, $s2      # primeiro parâmetro de swap é v
move    $a1, $s1      # segundo parâmetro de swap é j
```

Preservando registradores em sort

O único código restante é o salvamento e a restauração dos registradores. Com certeza, temos de salvar o endereço de retorno no registrador \$ra, pois sort é um procedimento que foi chamado por outro procedimento. O procedimento sort também utiliza os registradores salvos \$s0, \$s1, \$s2 e \$s3, de modo que precisam ser salvos. O prólogo do procedimento sort, portanto, é

```
addi   $sp,$sp,-20    # cria espaço na pilha para 5 registradores
sw     $ra,16($sp)   # salva $ra na pilha
sw     $s3,12($sp)   # salva $s3 na pilha
sw     $s2, 8($sp)    # salva $s2 na pilha
sw     $s1, 4($sp)    # salva $s1 na pilha
sw     $s0, 0($sp)    # salva $s0 na pilha
```

O final do procedimento simplesmente reverte todas essas instruções, depois acrescenta um jr para retornar.

O procedimento sort completo

Agora, juntamos todas as partes na Figura 2.36, tendo o cuidado de substituir as referências aos registradores \$a0 e \$a1 nos loops *for* por referências aos registradores \$s2 e \$s3. Novamente para tornar o código mais fácil de acompanhar, identificamos cada bloco de código com sua finalidade no procedimento. Neste capítulo, 9 linhas do procedimento sort em C tornaram-se 35 em assembly do MIPS.

Detalhamento: uma otimização que funciona com este exemplo é a utilização de procedimentos *inline*, mencionada na Seção 2.11. Em vez de passar argumentos em parâmetros e invocar o código com uma instrução *jal*, o compilador copiaria o código do corpo do procedimento *swap* onde a chamada a *swap* aparece no código. Essa otimização evitaria quatro instruções neste exemplo. A desvantagem da otimização que utiliza procedimentos *inline* é que o código compilado seria maior se o procedimento *inline* for chamado de vários locais. Essa expansão de código poderia ter um desempenho *inferior* se aumentasse a taxa de falhas na cache; ver Capítulo 7.

Os compiladores MIPS sempre reservam espaço na pilha para os argumentos, caso precisem ser armazenados, de modo que, na realidade, eles sempre decrementam o \$sp de 16, para dar espaço para todos os quatro registradores de argumento (16 bytes). Um motivo é que a linguagem C oferece *vararg* que permite que um ponteiro apanhe, digamos, o terceiro argumento de um procedimento. Quando, raramente, o compilador encontra *vararg*, ele copia os quatro registradores de argumento para os quatro locais reservados na pilha.

Entendendo o desempenho dos programas

A Figura 2.37 mostra o impacto da otimização do compilador sobre o desempenho do programa de ordenação, tempo de compilação, ciclos de clock, contagem de instruções e CPI. Observe que o código não otimizado tem o melhor CPI e a otimização O1 tem a menor contagem de instruções, mas O3 é a mais rápida, lembrando que o tempo é a única medida precisa do desempenho do programa.

Salvando registradores			
	sort: addi \$sp,\$sp,-20	# cria espaço na pilha para 5 registradores	
	sw \$ra, 16(\$sp)	# salva \$ra na pilha	
	sw \$s3, 12(\$sp)	# salva \$s3 na pilha	
	sw \$s2, 8(\$sp)	# salva \$s2 na pilha	
	sw \$s1, 4(\$sp)	# salva \$s1 na pilha	
	sw \$s0, 0(\$sp)	# salva \$s0 na pilha	
Corpo do procedimento			
Move parâmetros	move \$s2, \$a0 move \$s3, \$a1	# copia parâmetro \$a0 para \$s2 (salva \$a0) # copia parâmetro \$a1 para \$s3 (salva \$a1)	
Loop externo	move \$s0, \$zero forltst:slt \$t0, \$s0, \$s3 beq \$t0, \$zero, exit1	# i = 0 # reg. \$t0 = 0 se \$s0 ≥ \$s3 (i ≥ n) # vai para exit1 se \$s0 ≥ \$s3 (i ≥ n)	
Loop interno	addi \$s1, \$s0, -1 for2tst:slti \$t0, \$s1, 0 bne \$t0, \$zero, exit2 sll \$t1, \$s1, 2 add \$t2, \$s2, \$t1 lw \$t3, 0(\$t2) lw \$t4, 4(\$t2) slt \$t0, \$t4, \$t3 beq \$t0, \$zero, exit2	# j = i - 1 # reg. \$t0 = 1 se \$s1 < 0 (j < 0) # vai para exit2 se \$s1 < 0 (j < 0) # reg. \$t1 = j * 4 # reg. \$t2 = v + (j * 4) # reg. \$t3 = v[j] # reg. \$t4 = v[j + 1] # reg. \$t0 = 0 se \$t4 ≥ \$t3 # vai para exit2 se \$t4 ≥ \$t3	
Passa parâmetros e chama	move \$a0, \$s2 move \$a1, \$s1 (jal swap)	# lo. parâmetro de swap é v (antigo \$a0) # 2o. parâmetro de swap é j # código de swap mostrado na Figura 2.34	
Loop interno	addi \$s1, \$s1, -1 j for2tst	# j -= 1 # desvia para teste do loop interno	
Loop externo	Exit2: addi \$s0, \$s0, 1 j forltst	# i += 1 # desvia para teste do loop externo	
Restaurando registradores			
exit1: lw \$s0, 0(\$sp)	# restaura \$s0 da pilha		
lw \$s1, 4(\$sp)	# restaura \$s1 da pilha		
lw \$s2, 8(\$sp)	# restaura \$s2 da pilha		
lw \$s3, 12(\$sp)	# restaura \$s3 da pilha		
lw \$ra, 16(\$sp)	# restaura \$ra da pilha		
addi \$sp,\$sp, 20	# restaura stack pointer		
Retorno do procedimento			
jr \$ra	# retorna à rotina que chamou		

FIGURA 2.36 Versão em assembly do MIPS para o procedimento sort da Figura 2.35.

Otimização gcc	Desempenho relativo	Ciclos de clock (milhões)	Contador de instruções (milhões)	CPI
nenhuma	1,00	158.615	114.938	1,38
O1 (média)	2,37	66.990	37.470	1,79
O2 (completa)	2,38	66.521	39.993	1,66
O3 (integração de procedimentos)	2,41	65.747	44.993	1,46

FIGURA 2.37 Comparando desempenho, contagem de instruções e CPI usando otimizações do compilador para o Bubble Sort.

Os programas ordenaram 100.000 words com o array inicializado com valores aleatórios. Esses programas foram executados em um Pentium 4 com clock de 3,06GHz e um barramento de 533MHz com 2GB de memória SDRAM DDR PC2100. Ele usava o Linux versão 2.4.20.

A Figura 2.38 compara o impacto das linguagens de programação, compilação *versus* interpretação, e os algoritmos sobre o desempenho das ordenações. A quarta coluna mostra que o programa C otimizado é 8,3 vezes mais rápido do que o código Java interpretado para o Bubble Sort. O uso do compilador Java Just-In-Time torna o programa em Java 2,1 vezes *mais rápido* do que o programa em C não otimizado e dentro de um fator de 1,13 mais rápido do que o código C mais otimizado. (A próxima seção contém mais detalhes sobre interpretação *versus* compilação de Java e o código Java e MIPS para o Bubble Sort.) As razões não são tão próximas para o Quicksort na coluna 5, possivelmente porque é mais difícil amortizar o custo da compilação em runtime pelo tempo de execução mais curto. A última coluna demonstra o impacto de um algoritmo melhor, oferecendo um aumento de desempenho de três ordens de grandeza quando são ordenados 100.000 itens. Mesmo comparando o programa Java interpretado na coluna 5 com o programa C compilado com as melhores otimizações na coluna 4, o Quicksort vence o Bubble Sort por um fator de 50 (0,05 x 2468 ou 123 *versus* 2,41).

Linguagem	Método de execução	Otimização	Desempenho relativo ao Bubble Sort	Desempenho relativo ao Quicksort	Ganho do Quicksort versus Bubble Sort
C	compilador	nenhuma	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	interpretador	-	0,12	0,05	1050
	Compilador Just In Time	-	2,13	0,29	338

FIGURA 2.38 Desempenho de dois algoritmos de ordenação em C e Java usando interpretação e compiladores otimizados em relação à versão C não otimizada. A última coluna mostra a vantagem no desempenho do Quicksort em relação ao Bubble Sort para cada linguagem e opção de execução. Esses programas foram executados no mesmo sistema da Figura 2.37. A JVM é Sun versão 1.3.1, e o JIT é o Sun Hotspot versão 1.3.1.

2.14

Implementando uma linguagem orientada a objetos

linguagem orientada a objetos Uma linguagem de programação que é orientada em torno de objetos, em vez de ações, ou dados *versus* lógica.

Esta seção é para leitores interessados em ver como uma **linguagem orientada a objetos**, como Java, executa em uma arquitetura MIPS. Ela mostra os bytecodes Java utilizados para a interpretação e o código MIPS para a versão Java de alguns dos segmentos C nas seções anteriores, incluindo o Bubble Sort. O restante desta seção está no CD.

2.15

Arrays *versus* ponteiros

Um tópico desafiador para qualquer programador novo é entender os ponteiros. A comparação entre o código assembly que usa arrays e índices de array com o código assembly que usa ponteiros fornece esclarecimentos sobre ponteiros. Esta seção mostra as versões C e assembly do MIPS de dois procedimentos para zerar (clear) uma seqüência de words na memória: uma usando índices de array e uma usando ponteiros. A Figura 2.39 mostra os dois procedimentos em C.

A finalidade desta seção é mostrar como os ponteiros são mapeados em instruções MIPS, e não endossar um estilo de programação ultrapassado. Ao final da seção, veremos o impacto das otimizações do compilador moderno sobre esses dois procedimentos.

Versão de clear usando arrays

Vamos começar com a versão que usa arrays, `clear1`, focalizando o corpo do loop e ignorando o código de ligação do procedimento. Consideramos que os dois parâmetros `array` e `size` são encontrados nos registradores `$a0` e `$a1`, e que `i` é alocado ao registrador `$t0`.

```
clear1(int array[ ], int size)
{
    int i ;
    for (i = 0; i < size; i += 1)
        array[i] = 0;
}

clear2(int *array, int size)
{
    int *p;
    for (p = &array[0]; p < &array[size]; p = p + 1)
        *p = 0;
}
```

FIGURA 2.39 Dois procedimentos para definir um array com todos os valores iguais a zero. `clear1` usa índices, enquanto `clear2` usa ponteiros. O segundo procedimento precisa de alguma explicação para os que não estão acostumados com C. O endereço de uma variável é indicado por `&`, e a referência ao objeto apontando por um ponteiro é indicada por `*`. As declarações indicam que `array` e `p` são ponteiros para inteiros. A primeira parte do loop `for` em `clear2` atribui o endereço do primeiro elemento do `array` ao ponteiro `p`. A segunda parte do loop `for` testa se o ponteiro está apontando além do último elemento do `array`. Incrementar um ponteiro em um, na última parte do loop `for`, significa mover o ponteiro para o próximo objeto seqüencial do seu tamanho declarado. Como `p` é um ponteiro para inteiros, o compilador gerará instruções MIPS para incrementar `p` de quatro, o número de bytes de um inteiro MIPS. A atribuição no loop coloca 0 no objeto apontado por `p`.

A inicialização de `i`, a primeira parte do loop `for`, é simples:

```
move      $t0,$zero      # i = 0 (reg. $t0 = 0)
```

Para definir `array[i]` como 0, temos primeiro de obter seu endereço. Comece multiplicando `i` por 4, para obter o endereço em bytes:

```
loop1:s11      $t1,$t0,2      # $t1 = i * 4
```

Como o endereço inicial do array está em um registrador, temos de somá-lo ao índice para obter o endereço de `array[i]` usando uma instrução `add`:

```
add      $t2,$a0,$t1      # $t2 = endereço do array[i]
```

(Este exemplo é uma situação ideal para o endereçamento indexado.) Finalmente, podemos armazenar 0 nesse endereço:

```
sw      $zero, 0($t2)      # array[i] = 0
```

Essa instrução é o final do corpo do loop, de modo que o próximo passo é incrementar `i`:

```
addi     $t0,$t0,1      # i = i + 1
```

O teste do loop verifica se `i` é menor do que `size`:

```
slt  $t3,$t0,$a1      # $t3 = (i < size)
bne  $t3,$zero,loop1  # if (i < size) go to loop1
```

Agora, já vimos todas as partes do procedimento. Aqui está o código MIPS para zerar um array usando índices:

```

move    $t0,$zero          # i = 0
loop1: sll  $t1,$t0,2       # $t1 = i * 4
add     $t2,$a0,$t1         # $t2 = endereço de array[i]
sw      $zero, 0($t2)        # array[i] = 0
addi   $t0,$t0,1            # i = i + 1
slt    $t3,$t0,$a1           # $t3 = (i < size)
bne    $t3,$zero,loop1       # se (i < size) vai para loop1

```

(Esse código funciona desde que `size` seja maior que 0.)

Versão de clear usando ponteiros

O segundo procedimento que usa ponteiros aloca os dois parâmetros `array` e `size` aos registradores `$a0` e `$a1` e aloca `p` ao registrador `$t0`. O código para o segundo procedimento começa com a atribuição do ponteiro `p` ao endereço do primeiro elemento do array:

```
move    $t0,$a0          # p = endereço de array[0]
```

O código a seguir é o corpo do loop *for*, que simplesmente armazena 0 em `p`:

```
loop2:sw  $zero,0($t0)    # Memória[p] = 0
```

Essa instrução implementa o corpo do loop, de modo que o próximo código é o incremento da iteração, que muda `p` para que aponte para a próxima palavra:

```
addi   $t0,$t0,4          # p = p + 4
```

Incrementar um ponteiro em 1 significa mover o ponteiro para o próximo objeto seqüencial em C. Como `p` é um ponteiro para inteiros, cada um usando 4 bytes, o compilador incrementa `p` de 4.

O teste do loop vem em seguida. O primeiro passo é calcular o endereço do último elemento de array. Comece multiplicando `size` por 4 para obter seu endereço em bytes:

```

add   $t1,$a1,$a1          # $t1 = size * 2
add   $t1,$t1,$t1          # $t1 = size * 4

```

e depois acrescentamos o produto ao endereço inicial do array para obter o endereço da primeira word *após* o array:

```
add   $t2,$a0,$t1          # $t2 = endereço de array[size]
```

O teste do loop é simplesmente para ver se `p` é menor do que o último elemento de array:

```

slt   $t3,$t0,$t2          # $t3 = (p < &array[size])
bne   $t3,$zero,loop2       # se (p < &array[size]) vai para loop2

```

Com todas essas partes completadas, podemos mostrar uma versão do código para zerar um array usando ponteiros:

```

move   $t0,$a0          # p = endereço de array[0]
loop2:sw  $zero,0($t0)    # Memória[p] = 0
addi   $t0,$t0,4          # p = p + 4
add   $t1,$a1,$a1          # $t1 = size * 2
add   $t1,$t1,$t1          # $t1 = size * 4
add   $t2,$a0,$t1          # $t2 = endereço de array[size]
slt   $t3,$t0,$t2          # $t3 = (p < &array[size])
bne   $t3,$zero,loop2       # se (p < &array[size]) vai para loop2

```

Como no primeiro exemplo, esse código considera que `size` é maior do que 0.

Observe que esse programa calcula o endereço do final do array em cada iteração do loop, embora não mude. Uma versão mais rápida do código move esse cálculo para fora do loop:

```

move    $t0,$a0      # p = endereço de array[0]
sll     $t1,$a1,2    # $t1 = size * 4
add    $t2,$a0,$t1   # $t2 = endereço de array[size]
loop2:sw $zero,0($t0) # Memória[p] = 0
addi   $t0,$t0,4    # p = p + 4
slt    $t3,$t0,$t2   # $t3 = (p < &array[size])
bne   $t3,$zero,loop2 # se (p < &array[size]) vai para loop2

```

Comparando as duas versões de clear

A comparação das duas seqüências lado a lado ilustra a diferença entre os índices de array e ponteiros (as mudanças introduzidas pela versão de ponteiro estão destacadas):

<pre> move \$t0,\$zero # i = 0 loop1:sll \$t1,\$t0,2 # \$t1 = i * 4 add \$t2,\$a0,\$t1 # \$t2 = &array[i] sw \$zero,0(\$t2) # array[i] = 0 addi \$t0,\$t0,1 # i = i + 1 slt \$t3,\$t0,\$a1 # \$t3 = (i < size) bne \$t3,\$zero,loop1# se () vai para loop1 </pre>	<pre> move \$t0,\$a0 # p = & array[0] sll \$t1,\$a1,2 # \$t1 = size * 4 add \$t2,\$a0,\$t1 # \$t2 = &array[size] loop2: sw \$zero,0(\$t0) # Memória[p] = 0 addi \$t0,\$t0,4 # p = p + 4 slt \$t3,\$t0,\$t2 # \$t3=(p<&array[size]) bne \$t3,\$zero,loop2# se () vai para loop2 </pre>
--	--

A versão da esquerda precisa ter a “multiplicação” e a soma dentro do loop, porque `i` é incrementado e cada endereço precisa ser recalculado a partir do novo índice; a versão usando ponteiros para a memória à direita incrementa o ponteiro `p` diretamente. A versão usando ponteiros reduz as instruções executadas por iteração de 7 para 4. Essas otimizações manuais correspondem a otimizações do compilador chamadas redução de força (deslocamento em vez de multiplicação) e eliminação da variável de indução (eliminando cálculos de endereço de array dentro dos loops).

Detalhamento: o compilador C acrescentaria um teste para garantir que `size` seja maior do que 0. Uma maneira seria acrescentar um desvio, imediatamente antes da primeira instrução do loop, para a instrução `slt`.

Compreendendo o desempenho dos programas

As pessoas costumavam ser ensinadas a usar ponteiros em C para conseguir mais eficiência do que era possível com os arrays: “Use ponteiros, mesmo que você não consiga entender o código”. Os modernos compiladores com otimizações podem produzir um código muito bom para a versão usando arrays. A maioria dos programadores de hoje prefere que o compilador realize o trabalho pesado.

2.16 Vida real: instruções do IA-32

Os projetistas de conjuntos de instruções às vezes oferecem operações mais poderosas do que aquelas encontradas no MIPS. O objetivo geralmente é reduzir o número de instruções executadas por um programa. O perigo é que essa redução pode ocorrer ao custo da simplicidade, aumentando o tempo que um programa leva para executar, pois as instruções são mais lentas. Essa lentidão pode ser o resultado de um tempo de ciclo de clock mais lento ou a requisição de mais ciclos de clock do que uma seqüência mais simples (ver Seção 4.8).

*A beleza está
toda nos olhos
de quem vê.*

Margaret Wolfe
Hungerford, *Molly Bawn*, 1877

O caminho em direção à complexidade da operação é, portanto, repleto de perigos. Para evitar esses problemas, os projetistas passaram para instruções mais simples. A Seção 2.17 demonstra as armadilhas da complexidade.

O Intel IA-32

O MIPS foi a visão de um único grupo pequeno em 1985; as partes dessa arquitetura se encaixam muito bem, e a arquitetura inteira pode ser descrita de forma sucinta. Isso não acontece com o IA-32; ele é o produto de vários grupos independentes, que evoluíram a arquitetura por quase 20 anos, acrescentando novos recursos ao conjunto de instruções original, como alguém poderia acrescentar roupas em uma mala pronta. Aqui estão os marcos importantes do IA-32:

- **1978:** a arquitetura Intel 8086 foi anunciada como uma extensão compatível com o assembly para o então bem-sucedido Intel 8080, um microprocessador de 8 bits. O 8086 é uma arquitetura de 16 bits, com todos os registradores internos com 16 bits de largura. Ao contrário do MIPS, os registradores possuem usos dedicados, e, por isso, o 8086 não é considerado uma arquitetura com **registradores de uso geral**.
- **1980:** o coprocessador de ponto flutuante Intel 8087 foi anunciado. Essa arquitetura estende o 8086 com cerca de 60 instruções de ponto flutuante. Em vez de usar registradores, ele conta com uma pilha (ver Seção 2.19 e Seção 3.9).
- **1982:** o 80286 estendeu a arquitetura 8086, aumentando o espaço de endereçamento para 24 bits, criando um modelo de mapeamento e proteção de memória elaborado (ver Capítulo 7) e acrescentando algumas instruções para preencher o conjunto de instruções e manipular o modelo de proteção.
- **1985:** o 80386 estendeu a arquitetura 80286 para 32 bits. Além de uma arquitetura de 32 bits com registradores de 32 bits e um espaço de endereçamento de 32 bits, o 80386 acrescentou novos modos de endereçamento e operações adicionais. As instruções adicionais tornam o 80386 quase uma máquina de uso geral. O 80386 também acrescentou suporte para paginação além de endereçamento segmentado (ver Capítulo 7). Assim como o 80286, o 80386 possui um modo para executar programas do 8086 sem alteração.
- **1989-95:** os posteriores 80486 em 1989, Pentium em 1992 e Pentium Pro em 1995 visaram a um desempenho maior, com apenas quatro instruções acrescentadas ao conjunto de instruções visíveis ao usuário: três para ajudar com o multiprocessamento (Capítulo 9) e uma instrução move condicional.
- **1997:** depois que o Pentium e o Pentium Pro estavam sendo vendidos, a Intel anunciou que expandiria as arquiteturas Pentium e Pentium Pro com as MMX (Multi Media Extensions). Esse novo conjunto de 57 instruções utiliza a pilha de ponto flutuante para acelerar aplicações de multimídia e comunicações. As instruções MMX normalmente operam sobre vários elementos de dados curtos de uma só vez, na tradição das arquiteturas de única instrução e múltiplos dados (SIMD – Single Instruction, Multiple Data) (ver Capítulo 9). O Pentium II não introduziu novas instruções.
- **1999:** a Intel acrescentou outras 70 instruções, denominadas SSE (Streaming SIMD Extensions), como parte do Pentium III. As principais mudanças foram incluir oito registradores separados, dobrar sua largura para 128 bits e incluir um tipo de dados de ponto flutuante com precisão simples. Logo, quatro operações de ponto flutuante de 32 bits podem ser realizadas em paralelo. Para melhorar o desempenho da memória, as SSE incluiriam instruções de prefetch (pré-busca) da cache, mais instruções de armazenamento de streaming, que contornam as caches e escrevem diretamente na memória.

registradores de uso geral (GPR - General-Purpose Register) Um registrador que pode ser usado para endereços ou para dados, com praticamente qualquer instrução.

- **2001:** a Intel acrescentou ainda outras 144 instruções, dessa vez denominadas SSE2. O novo tipo de dados tem aritmética de precisão dupla, o que permite pares de operações de ponto flutuante de 64 bits em paralelo. Quase todas essas 144 instruções são versões de instruções MMX e SSE existentes que operam sobre 64 bits de dados em paralelo. Essa mudança não apenas habilita mais operações de multimídia, mas dá ao compilador um alvo diferente para operações de ponto flutuante do que a arquitetura de pilha única. Os compiladores podem decidir usar os oito registradores SSE como registradores de ponto flutuante, como aqueles encontrados em outros computadores. Essa mudança aumentou o desempenho de ponto flutuante no Pentium 4, o primeiro microprocessador a incluir instruções SSE2.
- **2003:** dessa vez, foi outra empresa, e não a Intel, que melhorou a arquitetura IA-32. A AMD anunciou um conjunto de extensões arquitetônicas para aumentar o espaço de endereçamento de 32 para 64 bits. Semelhante à transição do espaço de endereçamento de 16 para 32 bits em 1985, com o 80386, o AMD64 alarga todos os registradores para 64 bits. Ele também aumenta a quantidade de registradores para 16 e aumenta o número de registradores SSE de 128 bits para 16. A principal mudança no ISA vem da inclusão de um novo modo, chamado *modo longo*, que redefine a execução de todas as instruções IA-32 com endereços e dados de 64 bits. Para enfrentar a quantidade maior de registradores, ele acrescenta um novo prefixo às instruções. Dependendo de como você conta, o modo longo também acrescenta de 4 a 10 novas instruções e perde 27 antigas. O endereçamento de dados relativo ao PC é outra extensão. O AMD64 ainda possui um modo idêntico ao IA-32 (*modo legado*) e mais um modo que restringe os programas do usuário ao IA-32, mas permite que os sistemas operacionais utilizem o AMD64 (*modo de compatibilidade*). Esses modos permitem uma transição mais controlada para o endereçamento de 64 bits do que a arquitetura IA-64 da HP/Intel.
- **2004:** a Intel se rende e abraça o AMD64, trocando seu nome para Extended Memory 64 Technology (EM64T). A principal diferença é que a Intel acrescentou uma instrução de comparação e troca atômica de 128 bits, que provavelmente deveria ter sido incluída no AMD64. Ao mesmo tempo, a Intel anunciou outra geração de extensões de mídia. O SSE3 acrescenta 13 instruções para dar suporte à aritmética complexa, operações gráficas sobre arrays de estruturas, codificação de vídeo, conversão de ponto flutuante e sincronismo de threads (ver Capítulo 9). A AMD oferecerá o SSE3 nos próximos chips e quase certamente incluirá a instrução de troca atômica que estava faltando no ADM64, para manter a compatibilidade binária com a Intel.

Essa história ilustra o impacto das “algemas douradas” da compatibilidade com o IA-32, pois a base de software existente em cada etapa era muito importante para ser colocada em risco com mudanças arquitetônicas significativas.

Quaisquer que sejam as falhas artísticas do IA-32, lembre-se de que houve mais incidência dessa família arquitetônica nos desktops do que de qualquer outra arquitetura, aumentando em 100 milhões por ano. Apesar disso, esse ancestral diversificado levou a uma arquitetura difícil de explicar e impossível de amar.

Preste bem atenção ao que você está para ver! *Não* tente ler esta seção com o cuidado que precisaria para escrever programas IA-32; em vez disso, o objetivo é que você tenha alguma familiaridade com os pontos fortes e fracos da arquitetura para desktops mais populares do mundo.

Em vez de mostrar o conjunto de instruções inteiro de 16 e 32 bits, nesta seção, vamos nos concentrar no subconjunto de 32 bits originado com o 80386, pois essa parte da arquitetura é a usada. Começamos nossa explicação com os registradores e os modos de endereçamento, prosseguimos para as operações com inteiros e concluímos com um exame da codificação da instrução.

Registradores e modos de endereçamento de dados do IA-32

Os registradores do 80386 mostraram a evolução do conjunto de instruções (Figura 2.40). O 80386 estendeu todos os registradores de 16 bits (exceto os registradores de segmento) para 32 bits, inserindo

Nome	Uso
31	0
EAX	GPR 0
ECX	GPR 1
EDX	GPR 2
EBX	GPR 3
ESP	GPR 4
EBP	GPR 5
ESI	GPR 6
EDI	GPR 7
CS	Ponteiro do segmento de código
SS	Ponteiro do segmento de pilha (topo da pilha)
DS	Ponteiro do segmento de dados 0
ES	Ponteiro do segmento de dados 1
FS	Ponteiro do segmento de dados 2
GS	Ponteiro do segmento de dados 3
EIP	Ponteiro de instrução (PC)
EFLAGS	Códigos de condição

FIGURA 2.40 O conjunto de registradores do 80386. Começando com o 80386, os oito registradores iniciais foram estendidos para 32 bits e também poderiam ser usados como registradores de uso geral.

um *E* no início de seus nomes para indicar a versão de 32 bits. Vamos nos referir a eles genericamente como registradores de uso geral (ou GPRs – General-Purpose Registers). O 80386 contém apenas oito GPRs. Isso significa que os programas do MIPS podem usar quatro vezes isso.

As instruções aritméticas, lógicas e de transferência de dados são instruções de dois operandos que permitem as combinações mostradas na Figura 2.41. Existem duas diferenças importantes aqui. As instruções aritméticas e lógicas do IA-32 precisam ter um operando que atue como origem e destino; o MIPS admite registradores separados para origem e destino. Essa restrição coloca mais pressão sobre os registradores limitados, pois um registrador de origem precisa ser modificado. A segunda diferença importante é que um dos operandos pode estar na memória. Assim, praticamente qualquer instrução pode ter um operando na memória, ao contrário do MIPS e do PowerPC.

Tipo de operando de origem/destino	Segundo operando de origem
Registrador	Registrador
Registrador	Imediato
Registrador	Memória
Memória	Registrador
Memória	Imediato

FIGURA 2.41 Tipos de instrução para instruções aritméticas, lógicas e de transferência de dados. O IA-32 permite as combinações mostradas. A única restrição é a ausência de um modo memória-memória. Os imediatos podem ser de 8, 16 ou 32 bits de extensão; um registrador é qualquer um dos 14 principais registradores da Figura 2.40 (não EIP ou EFLAGS).

Os sete modos de endereçamento de memória, descritos com detalhes a seguir, oferecem dois tamanhos de endereços dentro da instrução. Esses chamados *deslocamentos* podem ser de 8 bits ou de 32 bits.

Embora um operando da memória possa usar qualquer modo de endereçamento, existem restrições em relação a quais registradores podem ser usados em um modo. A Figura 2.42 mostra os modos de endereçamento do IA-32 e quais GPRs não podem ser usados com esse modo, e mais, como você conseguiria o mesmo efeito usando instruções MIPS.

Operações com inteiros do IA-32

O 8086 oferece suporte para tipos de dados de 8 bits (*byte*) e 16 bits (*word*). O 80386 acrescenta endereços e dados de 32 bits (*double words*) ao IA-32. As distinções de tipo de dados se aplicam a operações com registrador e também a acessos à memória. Quase toda operação funciona sobre dados de 8 bits e sobre um tamanho de dados maior. Esse tamanho é determinado pelo modo e é de 16 bits ou de 32 bits.

Logicamente, alguns programas querem operar sobre dados de todos os três tamanhos, de modo que as arquiteturas 80386 oferecem um modo conveniente de especificar cada versão sem expandir muito o tamanho do código. Elas decidiram que os dados de 16 bits ou de 32 bits dominam a maioria dos programas, e, por isso, faz sentido ser capaz de definir um tamanho grande padrão. Esse tamanho de dados padrão é definido por um bit no registrador do segmento de código. Para redefinir o tamanho de dados padrão, um *prefixo* de 8 bits é anexado à instrução para dizer à máquina para usar o outro tamanho grande para essa instrução.

A solução do prefixo foi emprestada do 8086, o que permite que múltiplos prefixos modifiquem o comportamento da instrução. Os três prefixos originais redefinem o registrador de segmento padrão, bloqueiam o barramento para dar suporte a um semáforo (Capítulo 9) ou repetem a instrução seguinte até o registrador ECX chegar a 0. Esse último prefixo tinha por finalidade estar emparelhado com uma instrução mover byte para mover um número variável de bytes. O 80386 também acrescentou um prefixo para redefinir o tamanho de endereço padrão.

As operações com inteiros do IA-32 podem ser divididas em quatro classes principais:

1. Instruções para movimentação de dados, incluindo move, push e pop
2. Instruções aritméticas e lógicas, incluindo operações aritméticas de teste, inteiros e decimais

Modo	Descrição	Restrições de registrador	Equivalente MIPS
Registrador indireto	Endereço está em um registrador.	não ESP ou EBP	lw \$s0,0(\$s1)
Modo base com 8 ou 32 bits de deslocamento	Endereço é o conteúdo do registrador base mais deslocamento.	não ESP ou EBP	lw \$s0,100(\$s1) # deslocamento 16 bits
Base mais índice escalado	O endereço é Base + (2 ^{Escala} x Índice), onde Escala tem o valor 0, 1, 2 ou 3.	Base: qualquer GPR Índice: não ESP	mul \$t0,\$s2,4 add \$t0,\$t0,\$s1 lw \$s0,0(\$t0)
Base mais índice escalado com 8 ou 32 bits de deslocamento	O endereço é Base + (2 ^{Escala} x Índice) + deslocamento, onde Escala tem o valor 0, 1, 2 ou 3.	Base: qualquer GPR Índice: não ESP	mul \$t0,\$s2,4 add \$t0,\$t0,\$s1 lw \$s0,100(\$t0) # deslocamento 16 bits

FIGURA 2.42 Modos de endereçamento de 32 bits do IA-32 com restrições de registrador e o código MIPS equivalente. O modo de endereçamento Base mais Índice Escalado, que não aparece no MIPS ou no PowerPC, foi incluído para evitar as multiplicações por quatro (fator de escala 2) para transformar um índice de um registrador em um endereço em bytes (ver Figuras 2.34 e 2.36). Um fator de escala 1 é usado para dados de 16 bits, e um fator de escala 3 para dados de 64 bits. O fator de escala 0 significa que o endereço não é escalado. Se o deslocamento for maior do que 16 bits no segundo ou quarto modos, então o modo MIPS equivalente precisaria de mais duas instruções: um *lw* para ler os 16 bits mais altos do deslocamento e um *add* para somar a parte alta do endereço ao registrador base *\$s1*. (A Intel oferece dois nomes diferentes para o que é chamado modo de endereçamento com Base – com Base e Indexado –, mas eles são basicamente idênticos, e os combinamos aqui.)

3. Fluxo de controle, incluindo desvios condicionais, jumps incondicionais, chamadas e retornos
4. Instruções para manipulação de strings, incluindo movimento e comparação de strings

As duas primeiras categorias não precisam de comentários, exceto que as operações de instruções aritméticas e lógicas permitem que o destino seja um registrador ou um local da memória. A Figura 2.43 mostra algumas instruções IA-32 típicas e suas funções.

Instrução	Função
JE nome	se for igual(códigos de condição) {EIP=nome}; EIP-128 ≤ nome < EIP+128
JMP nome	EIP=nome
CALL nome	SP=SP-4; M[SP]=EIP+5; EIP=nome;
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	Define códigos de condição (flags) com EDX e 42
MOVSL	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

FIGURA 2.43 Algumas instruções IA-32 típicas e suas funções. Uma lista de operações freqüentes aparece na Figura 2.44. O CALL salva na pilha o EIP da próxima instrução. (EIP é o PC da Intel.)

Os desvios condicionais no IA-32 são baseados em *códigos de condição* ou *flags*. Os códigos de condição são definidos como um efeito colateral de uma operação; a maioria é usada para comparar o valor de um resultado com 0. Os desvios, então, testam os códigos de condição. O argumento para os códigos de condição é que eles ocorram como parte das operações normais e sejam mais rápidos de testar do que a comparação de registradores, como o MIPS faz para beq e bne. O argumento contra os códigos de condição é que a comparação com 0 aumenta o tempo da operação, pois utiliza hardware extra após a operação, e que freqüentemente o programador precisa usar instruções de comparação para testar um valor que não é o resultado de uma operação. Além do mais, os endereços de desvio relativos ao PC precisam ser especificados no número de bytes, visto que, ao contrário do MIPS, nem todas as instruções do 80386 possuem 4 bytes de extensão.

As instruções para manipulação de strings fazem parte da linhagem 8080 do IA-32, e não são comumente executadas na maioria dos programas. Elas freqüentemente são mais lentas do que as rotinas de software equivalentes (veja a Seção 2.17 neste capítulo).

A Figura 2.44 lista algumas das instruções do IA-32 com inteiros. Muitas das instruções estão disponíveis nos formatos byte e word.

Codificação de instruções do IA-32

Deixando o pior para o final, a codificação de instruções no 80386 é complexa, com muitos formatos de instrução diferentes. As instruções para o 80386 podem variar de 1 byte, quando não existem operandos, até 17 bytes.

A Figura 2.45 mostra o formato de instrução para várias instruções de exemplo na Figura 2.43. O byte de opcode normalmente contém um bit indicando se o operando é de 8 bits ou de 32 bits. Para algumas instruções, o opcode pode incluir o modo de endereçamento e o registrador; isso acontece em muitas instruções que possuem a forma “registrador = registrador op imediato”. Outras instruções utilizam um “pós-byte”, ou byte de opcode extra, rotulado “mod, reg, r/m”, que contém a informação sobre o modo de endereçamento. Esse pós-byte é usado para muitas das instruções que endereçam a memória. O modo “base mais índice escalado” utiliza um segundo pós-byte, rotulado com “sc, indice, base”.

Instrução	Significado
Controle	Desvios condicionais e incondicionais
JNZ, JZ	Desvia se condição para EIP + offset de 8 bits; JNE (para JNZ), JE (para JZ) são nomes alternativos
JMP	Jump incondicional – offset de 8 ou 16 bits
CALL	Chamada de sub-rotina – offset de 16 bits; o endereço de retorno é colocado na pilha
RET	Retira endereço de retorno da pilha e desvia para ele
LOOP	Desvio de loop – decrementa ECX; desvia para EIP + deslocamento de 8 bits se ECX ≠ 0
Transferência de dados	Move dados entre registradores ou entre registrador e memória
MOV	Move entre dois registradores ou entre registrador e memória
PUSH, POP	Coloca operando de origem na pilha; retira operando do topo da pilha para um registrador
LES	Lê ES e um dos GPRs da memória
Aritmética, lógica	Operações aritméticas e lógicas usando os registradores de dados e a memória
ADD, SUB	Adiciona origem ao destino; subtrai origem do destino; formato registrador-memória
CMP	Compara origem e destino; formato registrador-memória
SHL, SHR, RCR	Deslocamento à esquerda; deslocamento lógico à direita; giro à direita com código de condição de carry como preenchimento
CBW	Converte o byte nos 8 bits mais à direita de EAX para uma word de 16 bits mais à direita de EAX
TEST	AND lógico dos códigos de condição de origem e destino
INC, DEC	Incrementa destino, decrementa destino
OR, XOR	OR lógico; OR exclusivo; formato registrador-memória
String	Move entre operandos string; o tamanho é dado por um prefixo de repetição
MOVS	Copia da string origem para a string destino incrementando ESI e EDI; pode ser repetido
LODS	Lê um byte, uma word ou uma double word de uma string para o registrador EAX

FIGURA 2.44 Algumas operações típicas do IA-32. Muitas operações utilizam o formato registrador-memória, no qual a origem ou o destino podem ser a memória e o outro pode ser um registrador ou um operando imediato.

a. JE EIP + deslocamento

4	4	8
JE	Condição	Deslocamento

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	Pós-byte	r/m
				Deslocamento

d. PUSH ESI

5	3
PUSH	Regr

e. ADD EAX, #5765

4	1	32
ADD	Reg w	Immediato

f. TEST EDX, #42

7	1	8	32
TEST	w	Pós-byte	Immediato

FIGURA 2.45 Formatos típicos de instruções IA-32. A Figura 2.46 mostra a codificação do pós-byte. Muitas instruções contêm o campo de 1 bit w, que indica se a operação é de um byte ou doble word. O campo d em MOV é usado em instruções que podem mover de ou para a memória, e mostra a direção do movimento. A instrução ADD requer 32 bits para o campo imediato, visto que, no modo de 32 bits, os imediatos são de 8 bits ou de 32 bits. O campo imediato no TEST tem 32 bits de extensão, pois não existe um imediato de 8 bits para testar no modo de 32 bits. Em geral, as instruções podem variar de 1 a 17 bytes de extensão. O tamanho grande vem dos prefixos extras de 1 byte, tendo tanto um imediato de 4 bytes quanto um endereço de deslocamento de 4 bytes, usando um opcode de 2 bytes e usando o especificador do modo de índice escalado, que acrescenta outro byte.

A Figura 2.46 mostra a codificação dos dois especificadores de endereço pós-byte para os modos de 16 e 32 bits. Infelizmente, para entender quais registradores e quais modos de endereçamento estão disponíveis, você precisa ver a codificação de todos os modos de endereçamento e, às vezes, até mesmo a codificação das instruções.

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	end=BX+SI =EAX		mesmo	mesmo	mesmo	mesmo	mesmo
1	CL	CX	ECX	1	end=BX+DI =ECX		end. que	end. que	end. que	end. que	que
2	DL	DX	EDX	2	end=BP+SI =EDX		mod=0	mod=0	mod=0	mod=0	campo
3	BL	BX	EBX	3	end=BP+SI =EBX	+ desl8	+ desl8	+ desl16	+ desl32	reg	
4	AH	SP	ESP	4	End=SI =(sib)	SI+desl8	(sib)+desl8	SI+desl8	(sib)+desl32	"	
5	CH	BP	EBP	5	End=DI =desl32	DI+desl8	EBP+desl8	DI+desl16	EBP+desl32	"	
6	DH	SI	ESI	6	end=desl16 =ESI	BP+desl8	ESI+desl8	BP+desl16	ESI+desl32	"	
7	BH	DI	EDI	7	End=BX =EDI	BX+desl8	EDI+desl8	BX+desl16	EDI+desl32	"	

FIGURA 2.46 A codificação do primeiro especificador de endereço do IA-32, “mod, reg, r/m”. As quatro primeiras colunas mostram a codificação do campo reg de 3 bits, que depende do bit w do opcode e se a máquina está no modo de 16 bits (8086) ou no modo de 32 bits (80386). As demais colunas explicam os campos mod e r/m. O significado do campo r/m de 3 bits depende do valor do campo mod de 2 bits e do tamanho do endereço. Basicamente, os registradores utilizados no cálculo do endereço são listados na sexta e sétima colunas, sob mod = 0, com mod = 1 acrescentando um deslocamento de 8 bits e mod = 2 acrescentando um deslocamento de 16 ou 32 bits, dependendo do modo do endereço. As exceções são que r/m = 6 quando mod = 1 ou mod = 2 no modo de 16 bits seleciona BP mais o deslocamento; r/m = 5 quando mod = 1 ou mod = 2 no modo 16 bits seleciona EBP mais deslocamento; e r/m = 4 no modo de 32 bits quando mod = 3, onde (sib) significa o uso do modo de índice escalado, mostrado na Figura 2.42. Quando m = 3, o campo r/m indica um registrador, usando a mesma codificação que o campo reg combinado com o bit w.

Conclusão sobre o IA-32

A Intel tinha um microprocessador de 16 bits dois anos antes das arquiteturas mais elegantes de seus concorrentes, como o Motorola 68000, e essa largada na frente levou à seleção do 8086 como CPU para o IBM PC. Os engenheiros da Intel geralmente reconhecem que o IA-32 é mais difícil de ser montado do que máquinas como MIPS, mas o mercado muito maior significa que a Intel pode abrir mão de mais recursos para ajudar a contornar a complexidade adicional. O que o IA-32 perde no estilo é compensado na quantidade, tornando-o belo, do ponto de vista apropriado.

A graça salvadora é que os componentes arquitetônicos mais usados do IA-32 não são tão difíceis de implementar, como a Intel já demonstrou, melhorando rapidamente o desempenho dos programas com inteiros desde 1978. Para obter esse desempenho, os compiladores precisam evitar as partes da arquitetura difíceis de implementar com rapidez.

2.17 Falácias e armadilhas

Falácia: instruções mais poderosas significam maior desempenho.

Parte do poder do Intel IA-32 são os prefixos que podem modificar a execução da instrução seguinte. Um prefixo pode repetir a instrução seguinte até que um contador chegue a 0. Assim, para mover dados na memória, pode parecer que a seqüência de instruções natural seria usar move com o prefixo de repetição para realizar movimentações de memória para memória em 32 bits.

Um método alternativo, que usa as instruções padrão, encontradas em todos os computadores, é carregar os dados nos registradores e depois armazenar os registradores na memória. Essa segunda versão

desse programa, com o código replicado para reduzir o trabalho extra do loop, copia cerca de 1,5 vez mais rápido. Uma terceira versão, que usava os registradores de ponto flutuante maiores no lugar dos registradores inteiros do IA-32, copia cerca de 2,0 vezes mais rápido do que a instrução complexa.

Falácia: escreva em assembly para obter o maior desempenho.

Houve uma época em que os compiladores para as linguagens de programação produziam sequências de instrução ingênuas; a sofisticação cada vez maior dos compiladores significa que a lacuna entre o código compilado e o código produzido à mão está se fechando rapidamente. De fato, para competir com os compiladores atuais, o programador assembly precisa entender os conceitos dos Capítulos 6 e 7 (pipelining do processador e hierarquia de memória).

Essa batalha entre compiladores e codificadores assembly é uma situação em que os humanos estão perdendo terreno. Por exemplo, a linguagem C oferece ao programador uma chance de dar uma sugestão ao compilador sobre quais variáveis manter em registradores, em vez de passar para a memória. Quando os compiladores eram fracos na alocação de registradores, essas sugestões eram vitais para o desempenho. De fato, alguns livros-textos sobre C gastavam muito tempo dando exemplos com sugestões de como usar registradores com eficiência. Os compiladores C de hoje, em geral, ignoram essas sugestões, pois o compilador realiza um trabalho melhor na alocação do que o programador.

Mesmo se a escrita à mão resultasse em código mais rápido, os perigos de escrever em assembly são maior tempo gasto codificando e depurando, perda de portabilidade e dificuldade de manter esse código. Um dos poucos axiomas aceitos de modo generalizado na engenharia de software é que a codificação leva mais tempo se você escrever mais linhas, e claramente é preciso mais linhas para escrever um programa em assembly do que em C. Além do mais, uma vez codificado, o próximo perigo é que ele se torne um programa popular. Esses programas sempre vivem por mais tempo do que o esperado, significando que alguém terá de atualizar o código por vários anos e fazer com que funcione com novas versões dos sistemas operacionais e novos modelos de máquinas. A escrita em linguagem de alto nível no lugar do assembly não apenas permite que os compiladores futuros ajustem o código a máquinas futuras, mas também torna o software mais fácil de manter e permite que o programa execute em mais modelos de computadores.

Armadilha: esquecer que os endereços seqüenciais de word em máquinas com endereçamento em bytes não diferem em um.

Muitos programadores assembly têm lutado em cima de erros cometidos pela suposição de que o endereço da próxima word pode ser encontrado incrementando-se o endereço em um registrador por um, em vez do tamanho da word em bytes. Prevenir é melhor do que remediar!

Armadilha: usando um ponteiro para uma variável automática fora de seu procedimento de definição.

Um engano comum ao lidar com ponteiros é passar um resultado de um procedimento que inclui um ponteiro para um array que é local a esse procedimento. Segundo a disciplina de pilha da Figura 2.16, a memória que contém o array local será reutilizada assim que o procedimento retornar. Os ponteiros para variáveis automáticas podem levar ao caos.

2.18 Comentários finais

Os dois princípios do computador com *programa armazenado* são o uso de instruções que sejam indistintas de números e o uso de memória alterável para os programas. Esses princípios permitem que uma única máquina auxilie cientistas ambientais, consultores financeiros e autores de romance em suas especialidades. A seleção de um conjunto de instruções que a máquina possa entender exige um equilíbrio delicado entre a quantidade de instruções necessárias para executar um programa, a quan-

*Menos
significa mais.
Robert Browning,
Andrea del Sarto,
1855*

tidade de ciclos de clock necessários por uma instrução e a velocidade do clock. Quatro princípios de projeto orientam os autores de conjuntos de instruções para fazer esse equilíbrio delicado:

1. *Simplicidade favorece a regularidade.* A regularidade motiva muitos recursos do conjunto de instruções do MIPS: mantendo todas as instruções com um único tamanho, sempre exigindo três operandos de registrador nas instruções aritméticas e mantendo os campos de registrador no mesmo lugar em cada formato de instrução.
2. *Menor é mais rápido.* O desejo de velocidade é o motivo para que o MIPS tenha 32 registradores em vez de muito mais.
3. *Torne o caso comum veloz.* Alguns exemplos de tornar o caso comum do MIPS veloz são o endereçamento relativo ao PC para desvios condicionais e o endereçamento imediato para constantes como operandos.
4. *Um bom projeto exige bons compromissos.* Um exemplo do MIPS foi o compromisso entre providenciar endereços e constantes maiores nas instruções e manter todas as instruções com o mesmo tamanho.

Acima desse nível de máquina está o assembly, uma linguagem que os humanos podem ler. O monitor traduz isso para os números binários que as máquinas podem entender, e até mesmo “estende” o conjunto de instruções, criando instruções simbólicas que não estão no hardware. Por exemplo, constantes ou endereços que são muito grandes são divididos em partes com tamanho apropriado, variações comuns de instruções recebem seu próprio nome, e assim por diante. A Figura 2.47 lista as instruções MIPS que abordamos até aqui, tanto instruções reais quanto pseudo-instruções.

Instruções MIPS	Nome	Formato	PseudoMIPS	Nome	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 2.47 O conjunto de instruções do MIPS explicado até aqui, com as instruções MIPS reais à esquerda e as pseudo-instruções à direita. ■ O Apêndice A (Seção A.10) descreve a arquitetura MIPS completa. A Figura 2.27 mostra mais detalhes da arquitetura do MIPS revelada neste capítulo.

Essas instruções não nasceram iguais; a popularidade das poucas domina as muitas. Por exemplo, a Figura 2.48 mostra a popularidade de cada classe de instruções para o SPEC2000. A popularidade variada das instruções desempenha um papel importante nos capítulos sobre desempenho, caminho de dados, controle e pipelining.

Classe de instrução	Exemplos do MIPS	Correspondência com uma linguagem de alto nível	Frequência	
			Inteiro	Ponto flutuante
Aritmética	add, sub, addi	operações nas instruções de atribuição	24%	48%
Transferência de dados	lw, sw, lb, sb, lui	referências a estruturas de dados, como arrays	36%	39%
Lógica	and, or, nor, andi, ori, sll, srl	operações em instruções de atribuição	18%	4%
Desvio condicional	beq, bne,slt, slti	instruções if e loops	18%	6%
Jump	j, jr, jal	chamadas de procedimento, retornos e instruções case/switch	3%	0%

FIGURA 2.48 Classes de instruções MIPS, exemplos, correspondência com construções de linguagem de programação de alto nível e porcentagem média de instruções do MIPS executadas por categoria dos cinco programas de inteiros do SPEC2000 e cinco programas de ponto flutuante do SPEC2000. A Figura 3.26 mostra a porcentagem das instruções MIPS individuais executadas.

Cada categoria de instruções MIPS está associada a construções que aparecem nas linguagens de programação:

- As instruções aritméticas correspondem às operações encontradas nas instruções de atribuição.
- As instruções de transferência de dados provavelmente ocorrerão quando se lida com estruturas de dados, como arrays e estruturas.
- Os desvios condicionais são usados em instruções if e em loops.
- Os jumps incondicionais são usados em chamadas de procedimento e em retornos, e para instruções case/switch.

Depois que explicarmos a aritmética do computador no Capítulo 3, revelaremos mais da arquitetura do conjunto de instruções do MIPS.

2.19

Perspectiva histórica e leitura adicional

Esta seção analisa a história dos conjuntos de instruções com o tempo, e oferecemos uma pequena história das linguagens de programação e compiladores. As ISAs incluem arquiteturas de acumulador, arquiteturas de registrador de uso geral, arquiteturas de pilha e uma rápida história do IA-32. Também revemos os assuntos controvertidos de arquiteturas de computadores de linguagem de alto nível e arquiteturas de computadores com conjunto de instruções reduzido. A história das linguagens de programação inclui Fortran, Lisp, Algol, C, Cobol, Pascal, Simula, Smalltalk, C++ e Java, e a história dos compiladores inclui os principais marcos e os pioneiros que os alcançaram. O restante desta seção está no CD.

2.20 Exercícios

■ O Apêndice A descreve o simulador do MIPS, que é útil para estes exercícios. Embora o simulador aceite pseudo-instruções, tente não usar pseudo-instruções em qualquer exercício que pedir para produzir código do MIPS. Seu objetivo deverá ser aprender o conjunto de instruções MIPS real, e se você tiver de contar instruções, sua contagem deverá refletir as instruções reais executadas, e não as pseudo-instruções.

Existem alguns casos em que as pseudo-instruções precisam ser usadas (por exemplo, a instrução `la` quando um valor real não é conhecido durante a codificação em assembly). Em muitos casos, elas são muito convenientes e resultam em código mais legível (por exemplo, as instruções `li` e `move`). Se você decidir usar pseudo-instruções por esses motivos, por favor, acrescente uma sentença ou duas à sua solução, indicando quais pseudo-instruções usou e por quê.

2.1 [15] <§2.4> ■ **Aprofundando o aprendizado:** Formatos de instrução

2.2 [5] <§2.4> Que número binário este número hexadecimal representa: $7\text{fff}\text{ffff}_{\text{hex}}$? Que número decimal ele representa?

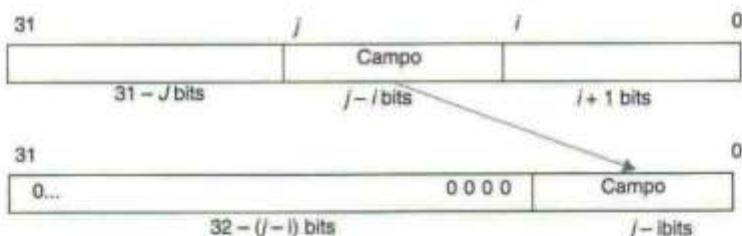
2.3 [5] <§2.4> Que número hexadecimal este número binário representa: $1100\ 1010\ 1111\ 1110\ 1111\ 1010\ 1100\ 1110_{\text{bin}}$?

2.4 [5] <§2.4> Por que o MIPS não tem uma instrução de subtração imediata?

2.5 [15] <§2.5> ■ **Aprofundando o aprendizado:** Código MIPS e operações lógicas

2.6 [15] <§2.5> Alguns computadores possuem instruções explícitas para extrair um campo qualquer de um registrador de 32 bits e colocá-lo nos bits menos significativos de um registrador. A figura a seguir mostra a operação desejada:

Encontre a seqüência mais curta de instruções MIPS que extraí um campo para os valores de constante $i = 5$ e $j = 22$ do registrador $\$t3$ e o coloque no registrador $\$t0$. (Dica: isso pode ser feito em duas instruções.)



2.7 [10] <§2.5> ■ **Aprofundando o aprendizado:** Operações lógicas no MIPS

2.8 [10] <§2.6> Construa um gráfico de fluxo de controle (como aquele mostrado na Figura 2.11) para a seguinte seção de código C ou Java:

```
for (i=0; i<x; i=i+1)
    y = y + i;
```

2.9 [10] <§2.6> ■ **Aprofundando o aprendizado:** Escrevendo código assembly

2.10 [25] <§2.7> Implemente o seguinte código C em MIPS, supondo que `set_array` seja a primeira função chamada:

```
int i;
void set_array(int num) {
    int array[10];
    for (i=0; i<10; i++) {
        array[i] = compare(num, i);
    }
}
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;
}
```

Não se esqueça de tratar o stack pointer e o frame pointer corretamente. A variável é alocada na pilha, e *i* corresponde a \$s0. Desenhe o status da pilha antes de chamar *set_array* e durante cada chamada de função. Indique os nomes dos registradores e variáveis armazenadas na pilha e marque o local de \$sp e \$fp.

2.11 [5] <§2.8> Iris e Julie são alunos de engenharia de computação aprendendo a respeito dos conjuntos de caracteres ASCII e Unicode. Ajude-os a escrever os nomes deles e o seu primeiro nome em ASCII (usando a notação decimal) e Unicode (usando a notação hexa e o conjunto de caracteres Latim Básico).

2.12 [10] <§2.8> Calcule os valores em decimal dos bytes que formam a representação ASCII terminada em nulo da seguinte string:

Um byte tem 8 bits

2.13 [30] <§§2.7, 2.8> ■ **Aprofundando o aprendizado:** Codificação MIPS e strings ASCII

2.14 [20] <§§2.7, 2.8> ■ **Aprofundando o aprendizado:** Codificação MIPS e strings ASCII

2.15 [20] <§§2.7, 2.8> {Ex. 2.14} ■ **Aprofundando o aprendizado:** Codificação MIPS e strings ASCII

2.16 [30] <§§2.7, 2.8> ■ **Aprofundando o aprendizado:** Codificação MIPS e strings ASCII

2.17 <§2.8> ■ **Aprofundando o aprendizado:** Comparando C/Java com MIPS

2.18 <§2.8> ■ **Aprofundando o aprendizado:** Traduzindo MIPS para C

2.19 <§2.8> ■ **Aprofundando o aprendizado:** Compreendendo o código MIPS

2.20 <§2.8> ■ **Aprofundando o aprendizado:** Compreendendo o código MIPS

2.21 [5] <§§2.3, 2.6, 2.9> Acrescente comentários ao código MIPS a seguir e descreva em uma sentença o que ele calcula. Suponha que \$a0 e \$a1 sejam usados para a entrada e ambos contenham inicialmente os inteiros *a* e *b*, respectivamente. Suponha que \$v0 seja usado para a saída.

```
loop:      add  $t0, $zero, $zero
           beq  $a1, $zero, finish
           add  $t0, $t0, $a0
           sub  $a1, $a1, 1
           j    loop
```

```

finish:      addi $t0, $t0, 100
             add  $v0, $t0, $zero

```

2.22 [12] <§§2.3, 2.6, 2.9> O seguinte fragmento de código processa dois arrays e produz um valor importante no registrador \$v0. Considere que cada array consista em 2.500 palavras indexadas de 0 a 2.499, que os endereços base dos arrays sejam armazenados em \$a0 e \$a1, respectivamente, e que seus tamanhos (2500) sejam armazenados em \$a2 e \$a3, respectivamente. Acrescente comentários ao código e descreva em uma sentença o que esse código faz. Especificamente, o que será retornado em \$v0?

```

sll      $a2, $a2, 2
sll      $a3, $a3, 2
add     $v0, $zero, $zero
add     $t0, $zero, $zero
outer:   add     $t4, $a0, $t0
          lw      $t4, 0($t4)
          add     $t1, $zero, $zero
inner:   add     $t3, $a1, $t1
          lw      $t3, 0($t3)
          bne    $t3, $t4, skip
          addi   $v0, $v0, 1
skip:    addi   $t1, $t1, 4
          bne    $t1, $a3, inner
          addi   $t0, $t0, 4
          bne    $t0, $a2, outer

```

2.23 [10] <§§2.3, 2.6, 2.9> Suponha que o código do Exercício 2.22 seja executado em uma máquina com um clock de 2GHz que exija a seguinte quantidade de ciclos para cada instrução:

Instrução	Ciclos
add, addi, sll	1
lw, bne	2

No pior caso, quantos segundos levará a execução desse código?

2.24 [5] <§2.9> Mostre a única instrução MIPS ou a seqüência mínima de instruções para esta instrução em C:

```
b = 25 | a;
```

Suponha que a corresponda ao registrador \$t0 e b corresponda ao registrador \$t1.

2.25 [10] <§2.9> ■ Aprofundando o aprendizado: Traduzindo de C para MIPS

2.26 [10] <§§ 2.3, 2.6, 2.9> O programa a seguir tenta copiar words do endereço no registrador \$a0 para o endereço no registrador \$a1, contando o número de words copiadas no registrador \$v0. O programa termina de copiar quando encontra uma word igual a 0. Você não precisa preservar o conteúdo dos registradores \$v1, \$a0 e \$a1. Essa word de término deverá ser copiada, mas não contada.

```

addi $v0, $zero, 0 # Inicializa contador
loop: lw  $v1, 0($a0)  # Lê próxima word da origem
      sw  $v1, 0($a1)  # Escreve no destino
      addi $a0, $a0, 4  # Avança ponteiro para próxima origem
      addi $a1, $a1, 4  # Avança ponteiro para próximo destino
      beq $v1, $zero, loop # Repete se a word copiada != zero

```

Existem vários bugs neste programa MIPS; conserte-os e mostre uma versão sem bugs. Como muitos dos exercícios neste capítulo, o modo mais fácil de escrever programas MIPS é usar o simulador descrito no ■ Apêndice A.

2.27 [10]<§§2.2, 2.3, 2.6, 2.9> ■ **Aprofundando o aprendizado:** Tradução reversa de MIPS para C

2.28 <§2.9> ■ **Aprofundando o aprendizado:** Traduzindo de C para MIPS

2.29 [25]<§2.10> Conforme discutimos na Seção 2.10, “Montador”, as pseudo-instruções não fazem parte do conjunto de instruções MIPS, mas normalmente aparecem nos programas MIPS. Para cada pseudo-instrução na tabela a seguir, produza uma seqüência mínima de instruções MIPS reais para realizar a mesma coisa. Você precisa usar \$at para algumas das seqüências. Na tabela, *big* refere-se a um número específico que exige 32 bits para representar e *small* a um número que possa caber em 16 bits.

Pseudo-instrução	O que ela realiza
move \$t1, \$t2	\$t1 = \$t2
clear \$t0	\$t0 = 0
beq \$t1, small, L	if (\$t1 == small) go to L
beq \$t2, big, L	if (\$t2 == big) go to L
li \$t1, small	\$t1 = small
li \$t2, big	\$t2 = big
ble \$t3, \$t5, L	if (\$t3 <= \$t5) go to L
bgt \$t4, \$t5, L	if (\$t4 > \$t5) go to L
bge \$t5, \$t3, L	if (\$t5 >= \$t3) go to L
addi \$t0, \$t2, big	\$t0 = \$t2 + big
lw \$t5, big(\$t2)	\$t5 = Memória[\$t2 + big]

2.30 [5]<§§2.9, 2.10> Dado seu conhecimento de endereçamento relativo ao PC, explique por que um montador poderia ter problemas para implementar diretamente a instrução de desvio na seguinte seqüência de código:

```
here:      beq $s0, $s2, there
          ...
there:     add $s0, $s0, $s0
```

Mostre como o montador poderia reescrever essa seqüência de código para solucionar esses problemas.

2.31 <§2.10> ■ **Aprofundando o aprendizado:** Pseudo-instruções MIPS

2.32 <§2.10> ■ **Aprofundando o aprendizado:** Link-editando código MIPS

2.33 <§2.10> ■ **Aprofundando o aprendizado:** Link-editando código MIPS

2.34 [20]<§2.11> Encontre um programa grande escrito em C (por exemplo, *gcc*, que pode ser obtido em <http://gcc.gnu.org>) e compile o programa duas vezes, uma com otimizações (*use -O3*) e outra sem. Compare o tempo de compilação e o tempo de execução do programa. Os resultados são o que você esperava?

2.35 [20]<§2.12> ■ **Aprofundando o aprendizado:** Melhorando os modos de endereçamento do MIPS

2.36 [10]<§2.12> ■ **Aprofundando o aprendizado:** Melhorando os modos de endereçamento do MIPS

2.37 [15]<§§2.6, 2.13> A tradução MIPS do segmento C (ou Java)

```

while  (save[i] == k)
    i += 1;

```

na Seção 2.6, “Compilando um Loop while em C”, utiliza um desvio condicional e um jump incondicional a cada passada pelo loop. Somente compiladores fracos produziriam um código com esse trabalho adicional do loop. Supondo que esse código esteja em Java (não C), reescreva o código assembly para que utilize no máximo um desvio ou jump a cada passada do loop. Além disso, acrescente código para realizar a verificação do Java para índice fora dos limites e garantir que esse código use no máximo um desvio ou jump a cada passada do loop. Quantas instruções são executadas antes e depois da otimização se a quantidade de iterações do loop for 10 e o valor de *i* nunca estiver fora dos limites?

2.38 [30] <§2.6, 2.13> Considere o seguinte fragmento de código Java:

```

for (i=0; i<=100; i=i+1)
    a[i] = b[i] + c;

```

Suponha que *a* e *b* sejam arrays de words e que o endereço base de *a* esteja em \$a0 e o endereço base de *b* esteja em \$a1. O registrador \$t0 está associado à variável *i* e o registrador \$s0 ao valor de *c*. Você também pode considerar que quaisquer constantes de endereço de que precise estejam disponíveis para serem lidas da memória. Escreva o código para o MIPS. Quantas instruções são executadas durante a execução desse código se não houver exceções de array fora dos limites? Quantas referências aos dados da memória serão feitas durante a execução?

2.39 [5] <§2.13> Escreva o código MIPS para o método Java compareTo (encontrado na Figura 2.35).

2.40 [15] <§2.17> Ao projetar sistemas de memória, é útil saber a freqüência das leituras de memória *versus* escritas, bem como a freqüência de acessos para instruções *versus* dados. Usando a informação sobre as classes de instruções do MIPS para o programa SPEC2000int, na Figura 2.48, encontre o seguinte:

- A porcentagem de todos os acessos à memória (tanto de dados quanto de instruções) que são para dados.
- A porcentagem de todos os acessos à memória (tanto de dados quanto de instruções) que são para leituras. Considere que dois terços das transferências de dados são loads.

2.41 [10] <§2.17> Realize os mesmos cálculos do Exercício 2.40, mas substitua o programa SPEC2000int por SPEC2000fp.

2.42 [15] <§2.17> Suponha que tenhamos feito as seguintes medições de CPI médio para instruções:

Instrução	CPI médio
Aritmética	1,0 ciclos de clock
Transferência de dados	1,4 ciclos de clock
Desvio condicional	1,7 ciclos de clock
Jump	1,2 ciclos de clock

Calcule o CPI efetivo para o MIPS. Calcule a média das freqüências de instruções para SPEC2000int e SPEC2000fp na Figura 2.48, a fim de obter a mistura de instruções.

2.43 [5] <§2.19> O conceito de programa armazenado, introduzido no final da década de 1940, provocou uma mudança significativa no modo como os computadores eram projetados e operados. Qual é um exemplo possível de uma máquina com programa não armazenado, e quais são os problemas com tal máquina? Como esses problemas podem ser contornados por uma máquina com programa armazenado?

Computadores no mundo real

Ajudando a salvar nosso meio ambiente com dados

Problema a ser resolvido: monitore plantas e animais do nosso ambiente para coletar informações que possam influenciar políticas ambientais.

Solução: computadores embutidos resistentes, alimentados por bateria, com sensores, comunicação sem fio e software apropriado.

A bióloga de Stanford Barbara Block estuda o atum bluefin. Uma questão política foi se o atum em um lado do Atlântico é diferente daquele no outro lado. Se for, então cada região poderia definir suas próprias cotas. Se não, então precisamos de cotas para todo o oceano.

Para responder a essa pergunta, ela começou a implantar dispositivos nos atuns que pudessem monitorar suas viagens. A cada dois minutos, uma etiqueta flutuante de arquivamento por satélite (PSAT) registra pressão da água, luz ambiente, temperatura, horário do dia e outras medições. Os dados são salvos em 1MB de

memória flash. O microprocessador embutido de 8 bits estima a profundidade a partir da pressão da água. Ele descobre a longitude usando os dados da intensidade de luz e o horário do dia. Ele determina o nascente, o poente e, portanto, a noite, e calcula o deslocamento de tempo entre a meia-noite local e a meia-noite em Greenwich, como um navegador usando um sextante ou cronômetro. A temperatura da água, mais tarde, é comparada com os registros do satélite para determinar a latitude. Barbara não conta com os pescadores para apanhar o atum e retornar as PSATs. Uma PSAT é presa a um peixe com um pino que se dissolve por meio de eletrólise depois de o computador ligar uma bateria. A etiqueta, então, flutua até a superfície e começa a transmitir dados para os satélites. A etiqueta flutuante pode transmitir por até duas semanas, enviando os dados diretamente para o laboratório de Barbara.



Barbara e alunos etiquetando um atum bluefin, que pode crescer até ficar com 900 quilos e 3 metros de extensão.



Uma etiqueta flutuante de arquivamento por satélite e sua eletrônica interna.

Barbara descobriu que o atum bluefin viaja mais de 16.000 quilômetros por ano; um atum etiquetado perto da Costa Leste dos Estados Unidos atravessará o Atlântico e se procriará no Golfo do México e no Mediterrâneo Oriental. Sua descoberta mudou as regulamentações, de modo que os atuns não são mais controlados separadamente no Atlântico Oriental e Ocidental. Agora, ela está desenvolvendo um censo da vida marinha no Pacífico, usando etiquetas menores, para animais menores, e etiquetas que transmitem toda vez que um peixe vem à superfície. Ela especula que o atum etiquetado poderia ser o “veículo” ideal para monitorar a mudança no oceano.

O biólogo de Berkeley Todd Dawson estuda a ecologia de uma árvore costeira, a *Sequoia sempervirens*, particularmente a interação da névoa marinha com as árvores. Durante anos, sua pesquisa envolveu a instalação de 50 quilos de equipamento e quilômetros de fios esticados até sensores. Esse trabalho normalmente é feito a mais de 80 metros acima do solo. Os dados só poderiam ser colhidos subindo até um registrador de dados do tamanho de uma impressora.

O cientista de computadores de Berkeley David Culler propôs uma nova técnica. Dawson agora está colocando, nas árvores, sensores em miniatura, com tecnologia sem fio, do tamanho

de embalagens de filme. Cada dispositivo tem menos de 3 polegadas cúbicas, pode transmitir até 40KBps e pode trabalhar durante meses com uma pilha comum. Como os dispositivos são pequenos e baratos, muitos podem ser colocados em uma árvore. Os dados são coletados com um laptop compatível, simplesmente caminhando-se até a base da árvore.

Dawson descobriu que a névoa no verão é responsável por 25% a 50% da água que as árvores recebem no ano inteiro. As árvores podem ainda estar bebendo água diretamente da névoa por meio de um relacionamento simbótico com os fungos que residem em suas folhas.

Dawson prevê que as redes de sensores sem fio mudarão o modo como as pessoas realizarão pesquisa ecológica.

Para saber mais, veja estas referências

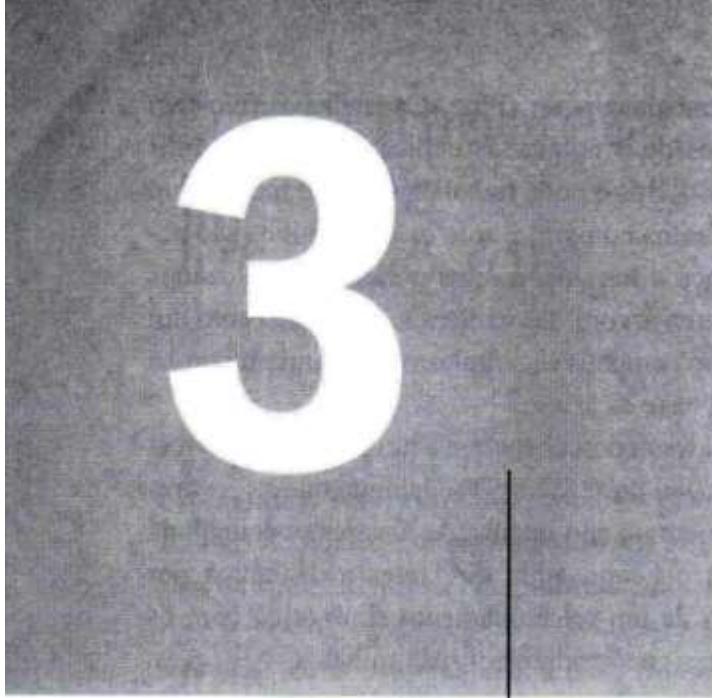
- Block *et al*, "Migratory movements, depth preferences, and thermal biology of atlantic bluefin tuna", *Science* 293: 1310-14, 2001
- "Redwoods", site do laboratório do Prof. Dawson
- "Redwood's drinking water from fog", *The Forestry Source*, novembro de 2002
- "Tagging of the Pacific Pelagics", www.toppcensus.org



Professor Dawson e um aluno subindo em uma sequóia para instalar monitores de névoa.



O dispositivo Mica com pilha. Ele possui o tamanho aproximado de uma embalagem de filme.



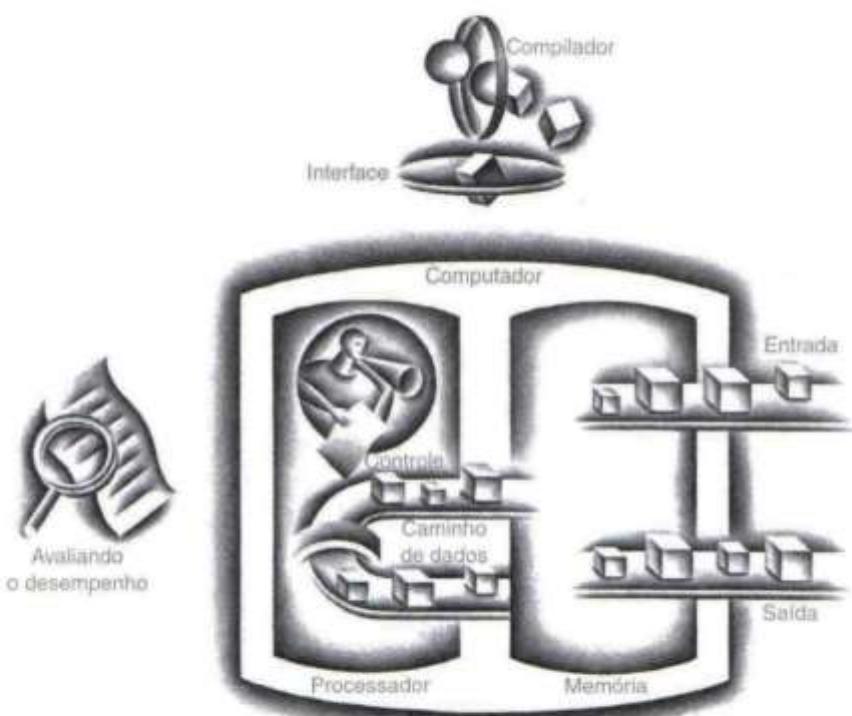
Aritmética Computacional

*A precisão numérica é a
própria alma da ciência.*

Sir D'Arcy Wentworth Thompson
On Growth and Form, 1917

3.1	Introdução	120
3.2	Números com sinal e sem sinal	120
3.3	Adição e subtração	128
3.4	Multiplicação	132
3.5	Divisão	137
3.6	Ponto flutuante	142
3.7	Vida real: ponto flutuante no IA-32	164
3.8	Falácia e armadilhas	167
3.9	Comentários finais	170
3.10	Perspectiva histórica e leitura adicional	173
3.11	Exercícios	174

Os cinco componentes clássicos de um computador



3.1 Introdução

As words do computador são compostas de bits; assim, podem ser representadas como números binários. Os números naturais 0, 1, 2 etc. possam ser representados em formato decimal ou binário, mas e quanto aos outros números? Por exemplo:

- Como são representados os números negativos?
- Qual é o maior número que pode ser representado em uma word do computador?
- O que acontece se uma operação cria um número maior do que poderia ser representado?
- E com relação a frações e números reais?

E por trás de todas essas perguntas existe um mistério: como o hardware realmente multiplica ou divide números?

O objetivo deste capítulo é desvendar esse mistério, incluindo a representação dos números, algoritmos aritméticos, hardware que acompanha esses algoritmos e as implicações de tudo isso para os conjuntos de instruções. Essas idéias podem ainda explicar truques que você já pode ter encontrado nos computadores. (Se você estiver acostumado com números binários com sinal, poderá pular a próxima seção e ir diretamente para a Seção 3.3)

3.2 Números com sinal e sem sinal

Os números podem ser representados em qualquer base; os humanos preferem a base 10 e, conforme examinamos no Capítulo 2, a base 2 é melhor para os computadores. Para evitar confusão, indicamos os números decimais com o subscrito *dec* e os números binários com *bin*. Em qualquer base numérica, o valor do *i*-ésimo dígito *d* é

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

onde *i* começa com 0 e aumenta da direita para a esquerda. Isso leva a um modo óbvio de numerar os bits na word: basta usar a potência da base para esse bit. Por exemplo,

1011_{bin}

representa

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

Logo, os bits são numerados com 0, 1, 2, 3, ... da *direita para a esquerda* em uma word. O desenho a seguir mostra a numeração dos bits dentro de uma word MIPS e o posicionamento do número 1011_{bin} :

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	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1		

(32 bits de largura)

Como as words são desenhadas vertical e horizontalmente, esquerda e direita podem não ser termos muito claros. Logo, o termo **bit menos significativo** é usado para se referir ao bit mais à direita (bit 0, no exemplo anterior) e **bit mais significativo** para o bit mais à esquerda (bit 31).

A word MIPS possui 32 bits de largura, de modo que podemos representar 2^{32} padrões diferentes de 32 bits. É natural deixar que essas representações representem os números de 0 a $2^{32} - 1$ ($4.294.967.295_{dec}$):

0000 0000 0000 0000 0000 0000 0000 _{bin}	= 0 _{dec}
0000 0000 0000 0000 0000 0000 0001 _{bin}	= 1 _{dec}
0000 0000 0000 0000 0000 0000 0010 _{bin}	= 2 _{dec}
...	...
1111 1111 1111 1111 1111 1111 1101 _{bin}	= 4.294.967.293 _{dec}
1111 1111 1111 1111 1111 1111 1110 _{bin}	= 4.294.967.294 _{dec}
1111 1111 1111 1111 1111 1111 1111 _{bin}	= 4.294.967.295 _{dec}

Ou seja, os números binários de 32 bits podem ser representados em termos do valor do bit vezes uma potência de 2 (aqui, x_i significa o i -ésimo bit de x):

$$(x31 \times 2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \dots + (x1 \times 2^1) + (x0 \times 2^0)$$

Interface hardware/software

A base 2 não é natural para os seres humanos; temos 10 dedos e, por isso, achamos a base 10 natural. Por que os computadores não usam decimal? Na verdade, o primeiro computador comercial *oferecia* aritmética decimal. O problema foi que o computador ainda usava sinais “ligado e desligado”, de modo que um dígito decimal simplesmente era representado por vários dígitos binários. A notação decimal provou ser tão ineficaz que os outros computadores revertem para binário, convertendo para a base 10 apenas para os eventos relativamente pouco freqüentes de entrada/saída.

ASCII VERSUS NÚMEROS BINÁRIOS

Poderíamos representar os números como seqüências de dígitos ASCII em vez de inteiros (ver Figura 2.21). Em quanto aumentaria o armazenamento se o número 1 bilhão fosse representado em ASCII, em vez de um inteiro de 32 bits?

EXEMPLO

Um bilhão significa 1.000.000.000, de modo que isso ocuparia 10 dígitos ASCII, cada um com 8 bits de largura. Assim, a expansão de armazenamento seria $(10 \times 8)/32$, ou 2,5. Além da expansão no armazenamento, é muito complicado para o hardware somar, subtrair, multiplicar e dividir tais números. Essas dificuldades explicam por que os profissionais de computação acreditam que o binário é natural e que o computador decimal ocasional é algo bizarro.

RESPOSTA

Lembre-se de que os padrões de bits binários que acabamos de mostrar simplesmente *representam* os números. Os números, na realidade, possuem uma quantidade infinita de dígitos, com quase todos sendo 0, exceto por alguns dos dígitos mais à direita. Só que, normalmente, não mostramos os 0s à esquerda.

O hardware pode ser projetado para somar, subtrair, multiplicar e dividir esses padrões de bits. Se o número que é o resultado correto de tais operações não puder ser representado por esses bits de hardware mais à direita, diz-se que houve um *overflow*. Fica a critério do sistema operacional e do programa determinar o que fazer quando isso ocorre.

bit menos significativo
O bit mais à direita em uma word MIPS.

bit mais significativo
O bit mais à esquerda em uma word MIPS.

Os programas de computador calculam números positivos e negativos, de modo que precisamos de uma representação que faça a distinção entre o positivo e o negativo. A solução mais óbvia é acrescentar um sinal separado, que convenientemente possa ser representado em um único bit; o nome dessa representação é *sinal e magnitude*.

Infelizmente, a representação com sinal e magnitude possui várias desvantagens. Primeiro, não é óbvio onde colocar o bit de sinal. À direita? À esquerda? Os primeiros computadores tentaram ambos. Segundo, os somadores de sinal e magnitude podem precisar de uma etapa extra para definir o sinal, pois não podemos saber, com antecedência, qual será o sinal correto. Finalmente, um bit de sinal separado significa que a representação com sinal e magnitude possui um zero positivo e um zero negativo, o que pode ocasionar problemas para os programadores desatentos. Como resultado desses problemas, a representação com sinal e magnitude logo foi abandonada.

Em busca de uma alternativa mais atraente, levantou-se a questão com relação a qual seria o resultado, para números sem sinal, se tentássemos subtrair um número grande de um número pequeno. A resposta é que ele tentaria pegar emprestado de uma seqüência de 0s à esquerda, de modo que o resultado seria uma seqüência de 1s à esquerda.

Como não havia uma alternativa melhor óbvia, a solução final foi escolher a representação que tornasse o hardware simples: 0s iniciais significam positivo, e 1s iniciais significam negativo. Essa convenção para representar os números binários com sinal é chamada representação por *complemento a dois*:

0000 0000 0000 0000 0000 0000 0000 _{bin}	= 0 _{dec}
0000 0000 0000 0000 0000 0000 0001 _{bin}	= 1 _{dec}
0000 0000 0000 0000 0000 0000 0010 _{bin}	= 2 _{dec}
...	...
0111 1111 1111 1111 1111 1111 1101 _{bin}	= 2.147.483.645 _{dec}
0111 1111 1111 1111 1111 1111 1110 _{bin}	= 2.147.483.646 _{dec}
0111 1111 1111 1111 1111 1111 1111 _{bin}	= 2.147.483.647 _{dec}
1000 0000 0000 0000 0000 0000 0000 _{bin}	= -2.147.483.648 _{dec}
1000 0000 0000 0000 0000 0000 0001 _{bin}	= -2.147.483.647 _{dec}
1000 0000 0000 0000 0000 0000 0010 _{bin}	= -2.147.483.646 _{dec}
...	...
1111 1111 1111 1111 1111 1111 1101 _{bin}	= -3 _{dec}
1111 1111 1111 1111 1111 1111 1110 _{bin}	= -2 _{dec}
1111 1111 1111 1111 1111 1111 1111 _{bin}	= -1 _{dec}

A metade positiva dos números, de 0 a 2.147.483.647_{dec} ($2^{31} - 1$), utiliza a mesma representação de antes. O padrão de bits seguinte (1000 ... 0000_{bin}) representa o número mais negativo -2.147.483.648_{dec} (-2^{31}). Ele é seguido por um conjunto decrescente de números negativos: -2.147.483.647_{dec} (1000 ... 0001_{bin}) até -1_{dec} (1111 ... 1111_{bin}).

A representação em complemento a dois possui um número negativo, -2.147.483.648_{dec}, que não possui um número positivo correspondente. Esse desequilíbrio era uma preocupação para o programador desatento, mas a representação com sinal e magnitude tinha problemas para o programador e para o projetista do hardware. Conseqüentemente, todo computador hoje em dia utiliza a representação de números binários por complemento a dois para os números com sinal.

A representação por complemento a dois tem a vantagem de que todos os números negativos possuem 1 no bit mais significativo. Conseqüentemente, o hardware só precisa testar esse bit para ver se um número é positivo ou negativo (com 0 considerado positivo). Esse bit normalmente é denominado *bit de sinal*. Reconhecendo o papel do bit de sinal, podemos representar números positivos e negativos de 32 bits em termos do valor do bit vezes uma potência de 2:

$$(x_{31} \times -2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \dots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

O bit de sinal é multiplicado por -2^{31} , e o restante dos bits são multiplicados pelas versões positivas de seus respectivos valores de base.

CONVERSÃO DE BINÁRIO PARA DECIMAL

Qual é o valor decimal deste número em complemento a dois com 32 bits?

1111 1111 1111 1111 1111 1111 1100_{bin}

Substituindo os valores dos bits do número na fórmula anterior:

$$\begin{aligned} & (1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \dots + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) \\ & = -2^{31} + 2^{30} + 2^{29} + \dots + 2^2 + 0 + 0 \\ & = -2.147.483.648_{dec} + 2.147.483.644_{dec} \\ & = -4_{dec} \end{aligned}$$

Logo, veremos um atalho para simplificar a conversão.

EXEMPLO

RESPOSTA

Assim como uma operação com números sem sinal pode ocasionar overflow na capacidade do hardware de representar o resultado, uma operação com números em complemento a dois também pode. O overflow ocorre quando o bit mais à esquerda da representação binária do hardware não é igual ao número infinito de dígitos à esquerda (o bit de sinal está incorreto): 0 à esquerda do padrão de bits quando o número é negativo ou 1 quando o número é positivo.

Interface hardware/software

Com sinal *versus* sem sinal aplica-se a loads e também à aritmética. A função de um load com sinal é copiar o sinal repetidamente para preencher o restante do registrador – chamado *extensão de sinal* –, mas sua finalidade é colocar a representação correta do número dentro desse registrador. Os loads sem sinal simplesmente preenchem com 0s à esquerda dos dados, pois o número representado pelo padrão de bits não tem sinal.

Ao ler uma word de 32 bits para um registrador de 32 bits, o ponto é discutível; loads com sinal e sem sinal são idênticos. O MIPS oferece dois tipos de loads de byte: *load byte* (lb) trata o byte como um número com sinal e, portanto, estende o sinal para preencher os 24 bits mais à esquerda do registrador, enquanto *load byte unsigned* (lbu) trabalha com inteiros sem sinal. Como os programas em C quase sempre usam bytes para representar caracteres em vez de considerar os bytes como inteiros com sinal muito curtos, lbu é usado de forma praticamente exclusiva para loads de byte. Por motivos semelhantes, *load half* (lh) trata a halfword como um número com sinal e, portanto, estende o sinal para preencher os 16 bits mais à esquerda do registrador, enquanto *load half-word unsigned* (lhu) trabalha com inteiros sem sinal.

Interface hardware/software

Diferente dos números discutidos anteriormente, os endereços de memória começam com 0 e continuam até o maior endereço. Em outras palavras, endereços negativos não fazem sentido. Assim, os programas desejam lidar às vezes com números que podem ser positivos ou negativos, e às vezes com números que só podem ser positivos. Algumas linguagens de programação refletem essa distinção. A linguagem C, por exemplo, chama os primeiros de *integers*, ou inteiros (declarados como int no programa), e os últimos de *unsigned integers*, ou inteiros sem sinal (*unsigned int*). Alguns guias de estilo C recomendam ainda declarar os primeiros como *signed int*, para deixar a distinção clara.

As instruções de comparação precisam lidar com essa dicotomia. Às vezes, um padrão de bits com 1 no bit mais significativo representa um número negativo e, naturalmente, é menor do que qualquer número positivo, que precisa ter um 0 no bit mais significativo. Com os inteiros sem sinal, por outro lado, 1 no bit mais significativo representa um número que é *maior* do que qualquer um que comece com 0. (Vamos tirar proveito desse significado dual do bit mais significativo para reduzir o custo da verificação de limites de array, dentro de algumas páginas.)

O MIPS oferece duas versões da comparação *set on less than*, para lidar com essas alternativas. *Set on less than* (*slt*) e *set on less than immediate* (*sli*) trabalham com inteiros com sinal. Os inteiros sem sinal são comparados por meio de *set on less than unsigned* (*sltu*) e *set on less than immediate unsigned* (*sliu*).

COMPARAÇÃO ENTRE NÚMEROS COM SINAL E SEM SINAL

EXEMPLO

Suponha que o registrador \$s0 tenha o número binário

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{bin}}$$

e que o registrador \$s1 tenha o número binário

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{bin}}$$

Quais são os valores dos registradores \$t0 e \$t1 após essas duas instruções?

```
slt  $t0, $s0, $s1 # comparação com sinal
sltu $t1, $s0, $s1 # comparação sem sinal
```

RESPOSTA

O valor no registrador \$s0 representa -1 se for um inteiro e $4.294.967.295_{\text{dec}}$ se for um inteiro sem sinal. O valor no registrador \$s1 representa 1 nos dois casos. Então, o registrador \$t0 possui o valor 1, pois $-1_{\text{dec}} < 1_{\text{dec}}$, e o registrador \$t1 possui o valor 0, visto que $4.294.967.295_{\text{dec}} > 1_{\text{dec}}$.

Antes de prosseguir para a soma e a multiplicação, vamos examinar alguns atalhos úteis quando trabalhamos com os números em complemento a dois.

O primeiro atalho é um modo rápido de negar um número binário no complemento a dois. Basta inverter cada 0 para 1 e cada 1 para 0, depois somar um ao resultado. Esse atalho é baseado na observação de que a soma de um número e sua representação invertida precisa ser $111 \dots 111_{\text{bin}}$, que representa -1. Como $x + x = -I$, portanto, $x + x + 1 = 0$, ou $x + 1 = -x$.

ATALHO PARA NEGAÇÃO

EXEMPLO

Negue 2_{dec} e depois verifique o resultado negando -2_{dec} .

RESPOSTA

$2_{\text{dec}} = 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{bin}}$
Negando esse número, invertendo os bits e somando um,

$$\begin{array}{r} 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{\text{bin}} \\ + \hspace{1cm} 1_{\text{bin}} \\ \hline = \hspace{0.5cm} 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{bin}} \\ = \hspace{0.5cm} -2_{\text{dec}} \end{array}$$

Na outra direção,

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{bin}$

primeiro é invertido e depois incrementado:

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{bin} \\
 + \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad 1_{bin} \\
 \hline
 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{bin} \\
 = 2_{dec}
 \end{array}$$

O segundo atalho nos diz como converter um número binário representado em n bits para um número representado com mais de n bits. Por exemplo, o campo imediato nas instruções load, store, branch, add e set on less than contém um número de 16 bits em complemento a dois, representando de -32.768_{dec} (-2^{15}) a 32.767_{dec} ($2^{15} - 1$). Para somar o campo imediato a um registrador de 32 bits, o computador precisa converter esse número de 16 bits para o seu equivalente em 32 bits. O atalho é pegar o bit mais significativo da menor quantidade – o bit de sinal – e replicá-lo para preencher os novos bits na quantidade maior. Os bits antigos são simplesmente copiados para a parte da direita da nova word. Esse atalho normalmente é chamado de *extensão de sinal*.

ATALHO PARA EXTENSÃO DE SINAL

Converta as versões binárias de 16 bits de 2_{dec} e -2_{dec} para números binários de 32 bits.

A versão binária de 16 bits do número 2 é

$$0000\ 0000\ 0000\ 0010_{bin} = 2_{dec}$$

Ele é convertido para um número de 32 bits criando-se 16 cópias do valor do bit mais significativo (0) e colocando-as na metade esquerda da word. A metade direita recebe o valor antigo:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{bin} = 2_{dec}$$

Vamos negar a versão de 16 bits de 2 usando o atalho anterior. Assim,

$$0000\ 0000\ 0000\ 0010_{bin}$$

torna-se

$$\begin{array}{r}
 1111\ 1111\ 1111\ 1101_{bin} \\
 + \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad 1_{bin} \\
 \hline
 = 1111\ 1111\ 1111\ 1110_{bin}
 \end{array}$$

Criar uma versão de 32 bits do número negativo significa copiar o bit de sinal 16 vezes e colocá-lo à esquerda:

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{bin} = -2_{dec}$$

EXEMPLO

RESPOSTA

Esse truque funciona porque os números positivos em complemento a dois realmente possuem uma quantidade infinita de 0s à esquerda, e os que são negativos em complemento a dois possuem uma quantidade infinita de 1s. O padrão binário que representa um número esconde os bits iniciais para caber na largura do hardware; a extensão do sinal simplesmente restaura alguns deles.

O terceiro atalho reduz o custo de verificar se $0 \leq x < y$, o que combina com a verificação de índice fora do limite para os arrays. A chave é que os inteiros negativos na notação em complemento a dois se parecem com números grandes na notação sem sinal; ou seja, o bit mais significativo é um bit de sinal na primeira notação, mas uma parte maior do número na segunda. Assim, uma comparação sem sinal de $x < y$ também verifica se x é negativo.

ATALHO PARA VERIFICAÇÃO DE LIMITES

EXEMPLO

Use este atalho para reduzir uma verificação de índice fora dos limites: desvie para IndiceForaDosLimites se $\$a1 \geq \$t2$ ou se $\$a1$ for negativo.

RESPOSTA

O código de verificação só utiliza `sltu` para realizar as duas verificações:

```
slt $t0,$a1,$t2 # Reg. temporário $t0=0 se k >= tamanho ou k<0  
beq $t0,$zero, IndiceForaDosLimites # se fora dos limites, vai para Error
```

Resumo

O ponto principal desta seção é que precisamos representar inteiros positivos e negativos dentro de uma word do computador, e embora existam prós e contras a qualquer opção, a escolha predominante desde 1965 tem sido o complemento a dois. A Figura 3.1 mostra os acréscimos ao assembly do MIPS revelados nesta seção. (A linguagem de máquina do MIPS também é ilustrada no Guia de referência rápida encontrado no início do livro.)

Verifique você mesmo

Que tipo de variável que pode conter $1.000.000.000_{dec}$ ocupa mais espaço na memória?

1. int em C
2. string em C
3. string em Java (que usa Unicode)

Detalhamento: o complemento a dois recebe esse nome devido à regra de que a soma sem sinal de um número de b bits e seu negativo é 2^b ; logo, o complemento ou a negação de um número em complemento a dois é $2^b - x$.

Uma terceira representação alternativa é chamada complemento a um. O negativo de um complemento a um é encontrado invertendo-se cada bit, de 0 para 1 e de 1 para 0, o que ajuda a explicar seu nome, pois o complemento de x é $2^b - x - 1$. Essa também foi uma tentativa de ser uma solução melhor do que a técnica de sinal e magnitude, e vários computadores científicos utilizaram a notação. Essa representação é semelhante ao complemento a dois, exceto que também possui dois 0s: $00\dots 00_{bin}$ é o 0 positivo, e $11\dots 11_{bin}$ é o 0 negativo. O maior número negativo $10\dots 00_{bin}$ representa $-2.147.483.647_{dec}$, e, por isso, os positivos e negativos são平衡ados. Os que aderiram ao complemento a um precisaram de uma etapa extra para subtrair um número, e, por isso, o complemento a dois domina hoje.

Uma notação final, que veremos quando discutirmos sobre ponto flutuante, é representar o valor mais negativo por $00\dots 00_{bin}$ e o valor mais positivo por $11\dots 11_{bin}$, com 0 normalmente tendo o valor $10\dots 00_{bin}$. Isso é chamado de **notação deslocada** (biased notation), pois desloca o número de modo que o número mais o deslocamento tenha uma representação não-negativa.

notação deslocada

(biased notation) Uma notação que representa o valor mais negativo por $00\dots 00_{bin}$ e o valor mais positivo por $11\dots 11_{bin}$, com 0 normalmente tendo o valor $10\dots 00_{bin}$, deslocando assim o número de modo que o número mais o deslocamento tenha uma representação não-negativa.

Operandos MIPS

Nome	Exemplo	Comentários
32 registradores	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Locais rápidos para dados. No MIPS, os dados precisam estar em registradores para a realização de operações aritméticas. O registrador MIPS \$zero sempre é igual a 0. O registrador \$at é reservado para o montador tratar de constantes grandes.
2^{30} words na memória	Memória[0], Memória[4]..... Memória[4294967292]	Acessadas apenas por instruções de transferência de dados no MIPS. O MIPS utiliza endereços em bytes, de modo que os endereços em words sequenciais diferem em 4 vezes. A memória contém estruturas de dados, arrays e spilled registers, como aqueles salvos nas chamadas de procedimento.

Assembly do MIPS

Categoría	Instrucción	Exemplo	Significado	Comentários
Aritmética	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Três operandos; dados nos registradores
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Três operandos; dados nos registradores
	add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	Usada para somar constantes
Transferência de dados	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memória}[\$s2 + 100]$	Dados da memória para o registrador
	store word	sw \$s1,100(\$s2)	$\text{Memória}[\$s2 + 100] = \$s1$	Dados do registrador para a memória
	load half unsigned	lhu \$s1,100(\$s2)	$\$s1 = \text{Memória}[\$s2 + 100]$	Halfword da memória para registrador
	store half	sh \$s1,100(\$s2)	$\text{Memória}[\$s2 + 100] = \$s1$	Halfword de um registrador para memória
	load byte unsigned	lbu \$s1,100(\$s2)	$\$s1 = \text{Memória}[\$s2 + 100]$	Byte da memória para registrador
	store byte	sb \$s1,100(\$s2)	$\text{Memória}[\$s2 + 100] = \$s1$	Byte de um registrador para memória
	load upper immed.	lui \$s1,100	$\$s1 = 100 * 2^{16}$	Carrega constante nos 16 bits mais altos
Lógica	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Três operadores em registrador; AND bit a bit
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \$s3$	Três operadores em registrador; OR bit a bit
	nor	nor \$s1,\$s2,\$s3	$\$s1 = -(\$s2 \$s3)$	Três operadores em registrador; NOR bit a bit
	and immediate	andi \$s1,\$s2,100	$\$s1 = \$s2 \& 100$	AND bit a bit registrador com constante
	or immediate	ori \$s1,\$s2,100	$\$s1 = \$s2 100$	OR bit a bit registrador com constante
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 << 10$	Deslocamento à esquerda por constante
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 >> 10$	Deslocamento à direita por constante
Desvio condicional	branch on equal	beq \$s1,\$s2,25	if ($\$s1 == \$s2$) go to PC + 4 + 100	Testa igualdade; desvio relativo ao PC
	branch on not equal	bne \$s1,\$s2,25	if ($\$s1 != \$s2$) go to PC + 4 + 100	Testa desigualdade; relativo ao PC
	set on less than	slt \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) \$s1 = 1; else \$s1 = 0	Compara menor que; usado com beq, bne
	set less than immediate	slti \$s1,\$s2,100	if ($\$s2 < 100$) \$s1 = 1; else \$s1 = 0	Compara menor que constante
	set less than unsigned	sltu \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) \$s1 = 1; else \$s1 = 0	Compara menor que; números sem sinal
	set less than immediate unsigned	sltiu \$s1,\$s2,100	if ($\$s2 < \$s3$) \$s1 = 1; else \$s1 = 0	Compara < constante; números sem sinal
Desvio incondicional	jump	j 2500	go to 10000	Desvia para endereço de destino
	jump register	jr \$ra	go to \$ra	Para switch e retorno de procedimento
	jump and link	jal 2500	$\$ra = PC + 4$; go to 10000	Para chamada de procedimento

FIGURA 3.1 Arquitetura do MIPS revelada até aqui. A cor indica as partes desta seção acrescentadas à arquitetura MIPS, revelada no Capítulo 2 (Figura 3.26). A linguagem de máquina do MIPS está listada no Guia de referência rápida no inicio do livro.

Subtração: o companheiro esquisito da adição

No. 10, Top Ten Courses for Athletes at a Football Factory, David Letterman e outros, Book of Top Ten Lists, 1990

Detalhamento: para os números decimais com sinal, usamos “-” para representar o negativo, já que não existem limites para o tamanho de um número decimal. Dado um tamanho de word fixo, as sequências de binárias e hexadecimais podem codificar o sinal, e, por isso, normalmente não usamos “+” ou “-” com a notação binária ou hexadecimal

3.3

Adição e subtração

A adição é exatamente o que você esperaria nos computadores. Dígitos são somados bit a bit, da direita para a esquerda, com carries (“vai-uns”), sendo passados para o próximo dígito à esquerda, como você faria manualmente. A subtração utiliza a adição: o operando apropriado é simplesmente negado antes de ser somado.

ADIÇÃO E SUBTRAÇÃO BINÁRIA

EXEMPLO

Vamos tentar somar 6_{dec} a 7_{dec} em binário e depois subtrair 6_{dec} de 7_{dec} em binário.

RESPOSTA

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{bin} = 7_{dec} \\
 +\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{bin} = 6_{dec} \\
 \hline
 =\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{bin} = 13_{dec}
 \end{array}$$

Os 4 bits à direita fazem toda a ação; a Figura 3.2 mostra as somas e os carries. Os carries aparecem entre parênteses, com as setas mostrando como são passados.

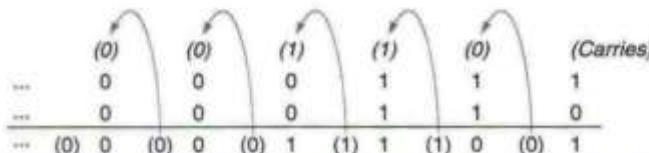


FIGURA 3.2 Adição binária, mostrando carries da direita para a esquerda. O bit mais à direita soma 1 a 0, resultando na soma desse bit sendo 1 e o carry desse bit sendo 0. Logo, a operação do segundo dígito à direita é $0 + 1 + 1$. Isso gera um 0 para esse bit de soma e um carry de 1. O terceiro bit é a soma de $1 + 1 + 1$, resultando em um carry de 1 e um bit de soma 1. O quarto bit é $1 + 0 + 0$, gerando uma soma 1 e nenhum carry.

A subtração de 6_{dec} de 7_{dec} pode ser feita diretamente:

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{bin} = 7_{dec} \\
 -\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{bin} = 6_{dec} \\
 \hline
 =\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{bin} = 1_{dec}
 \end{array}$$

ou por meio da soma, usando a representação de complemento a dois de -6 :

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{bin} = 7_{dec} \\
 +\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{bin} = -6_{dec} \\
 \hline
 =\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{bin} = 1_{dec}
 \end{array}$$

Já dissemos que o overflow ocorre quando o resultado de uma operação não pode ser representado com o hardware disponível, nesse caso, uma word de 32 bits. Quando pode ocorrer um overflow na adição? Quando se somam operandos com sinais diferentes, não poderá haver overflow. O motivo é que a soma não pode ser maior do que um dos operandos. Por exemplo, $-10 + 4 = -6$. Como os operandos cabem nos 32 bits, e a soma não é maior do que um operando, a soma também precisa caber nos 32 bits. Portanto, nenhum overflow pode ocorrer ao somar operandos positivos e negativos.

Existem restrições semelhantes à ocorrência do overflow durante a subtração, mas esse é apenas o princípio oposto: quando os sinais dos operandos são *iguais*, o overflow pode ocorrer. Para ver isso, lembre-se de que $x - y = x + (-y)$, pois subtraímos negando o segundo operando e depois somamos. Assim, quando subtraímos operandos do mesmo sinal, acabamos *somando* operandos de sinais *diferentes*. Pelo parágrafo anterior, sabemos que não pode ocorrer overflow também nesse caso.

Tendo examinado quando um overflow pode ocorrer na adição e na subtração, ainda não respondemos como detectar quando ele *ocorre*. O overflow ocorre quando se somam dois números positivos, e a soma é negativa, ou vice-versa. Logicamente, a soma ou a subtração de dois números de 32 bits pode gerar um resultado que precisa de 33 bits para ser totalmente expresso. A falta de um 33º bit significa que, quando o overflow ocorre, o bit de sinal está sendo definido com o *valor* do resultado, no lugar do sinal apropriado do resultado. Como precisamos apenas de um bit extra, somente o bit de sinal pode estar errado. Isso significa que um carry ocorreu no bit de sinal.

O overflow ocorre na subtração quando subtraímos um número negativo de um número positivo e obtemos um resultado negativo, ou quando subtraímos um número positivo de um número negativo e obtemos um resultado positivo. Isso significa que houve um empréstimo do bit de sinal. A Figura 3.3 mostra a combinação de operações, operandos e resultados que indicam um overflow.

Operação	Operando A	Operando B	Resultado indicando overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

FIGURA 3.3 Condições de overflow para adição e subtração.

Acabamos de ver como detectar o overflow para os números em complemento a dois em um computador. E com relação aos inteiros sem sinal? Os inteiros sem sinal normalmente são usados para endereços de memória onde os overflows são ignorados.

Logo, o projetista de computador precisa oferecer uma maneira de ignorar o overflow em alguns casos e reconhecê-lo em outros. A solução do MIPS é ter dois tipos de instruções aritméticas para reconhecer as duas escolhas:

- Adição (add), adição imediata (addi) e subtração (sub) causam exceções no overflow.
- Adição sem sinal (addu), adição imediata sem sinal (addiu) e subtração sem sinal (subu) *não* causam exceções no overflow.

Como a linguagem C ignora os overflows, os compiladores C do MIPS sempre gerarão as versões sem sinal das instruções aritméticas addu, addiu e subu, não importa o tipo das variáveis. No entanto, os compiladores Fortran do MIPS apanham as instruções aritméticas apropriadas, dependendo do tipo dos operandos.

Interface hardware/software

O projetista de computador precisa decidir como tratar overflows aritméticos. Embora algumas linguagens como C ignorem o overflow de inteiros, linguagens como Ada e Fortran exigem que o programa seja notificado. O programador ou o ambiente de programação precisa, então, decidir o que fazer quando ocorre o overflow.

exceção Também chamada interrupção. Um evento não-planejado que interrompe a execução do programa; usada para detectar overflow.

interrupção Uma exceção que vem de fora do processador. (Algumas arquiteturas utilizam o termo interrupção para todas as exceções.)

O MIPS detecta o overflow com uma exceção, também chamada de **interrupção** em muitos computadores. Uma exceção ou interrupção é basicamente uma chamada de procedimento não-planejada. O endereço da instrução que gerou o overflow é salvo em um registrador, e o computador desvia para um endereço predefinido, para invocar a rotina apropriada para essa exceção. O endereço interrompido é salvo de modo que, em algumas situações, o programa pode continuar após o código corretivo ser executado. (A Seção 5.6 abrange as exceções com mais detalhes; os Capítulos 7 e 8 descrevem outras situações em que ocorrem exceções e interrupções.)

O MIPS inclui um registrador, chamado *contador de programa de exceção* (EPC – Exception Program Counter), para conter o endereço da instrução que causou a exceção. A instrução *move from system control* (\$fc0) é usada para copiar o EPC para um registrador de uso geral, de modo que o software do MIPS tem a opção de retornar à instrução problemática por meio de uma instrução *jump register*.

Detalhamento: o MIPS pode interceptar um overflow, mas, diferente de muitos outros computadores, não existe desvio condicional para testar o overflow. Uma seqüência de instruções MIPS pode descobrir o overflow. Para a adição com sinal, a seqüência é a seguinte:

```

addu $t0, $t1, $t2 # $t0 = soma, mas não intercepta
xor $t3, $t1, $t2 # Verifica se sinais são diferentes
slt $t3, $t3, $zero # $t3 = 1 se os sinais são diferentes
bne $t3, $zero, Sem overflow # sinais de $t1, $t2 s,
portanto, sem overflow
xor $t3, $t0, $t1 # sinais =; sinal da soma também combina?
# $t3 negativo se soma diferente
slt $t3, $t3, $zero # $t3 = 1 se sinal da soma diferente
bne $t3, $zero, Overflow # Todos os três sinais ; vai para
overflow

```

Para adição sem sinal ($$t0 = $t1 + $t2$), o teste é

```

addu $t0, $t1, $t2 # $t0 = soma
nor $t3, $t1, $zero # $t3 = NOT $t1
# (compl. a dois - 1:  $2^{32} - $t1 - 1$ )  $\overset{\text{NOT } t_1}{\cancel{2^{32}}} \cancel{t_1} < t_2$ 
slt $t3, $t3, $t2 #  $(2^{32} - $t1 - 1) < $t2$   $\cancel{2^{32}} \cancel{t_1} > \cancel{2^{32}}$  ?
#  $\Rightarrow 2^{32} - 1 < $t1 + $t2$ 
bne $t3, $zero, Overflow # se  $(2^{32}-1 < $t1 + $t2)$  vai para
overflow

```

Resumo

O ponto principal desta seção é que, independente da representação, o tamanho de word finito dos computadores significa que as operações aritméticas podem criar resultados muito grandes para caber nesse tamanho de word fixo. É fácil detectar o overflow em números sem sinal, embora quase sempre sejam ignorados, pois os programas não querem detectar overflow para a aritmética de endereço, o uso mais comum dos números naturais. O complemento a dois apresenta um desafio maior, embora alguns sistemas de software exijam detecção de overflow, de modo que, hoje, todos os computadores tenham um meio de detectá-lo. A Figura 3.4 mostra as adições à arquitetura MIPS apresentadas nesta seção.

Assembly do MIPS

Categoría	Instrução	Exemplo	Significado	Comentários
Aritmética	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Três operandos; overflow detectado
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Três operandos; overflow detectado
	add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constante; overflow detectado
	add unsigned	addu \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Três operandos; overflow não detectado
	subtract unsigned	subu \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Três operandos; overflow não detectado
	add immediate unsigned	addiu \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constante; overflow não detectado
	move from coprocessor register	mfc0 \$s1,\$epc	$\$s1 = \epc	Usado para copiar PC de exceção para outros registradores especiais
Transferência de dados	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memória}[\$s2 + 100]$	Dados da memória para um registrador
	store word	sw \$s1,100(\$s2)	$\text{Memória}[\$s2 + 100] = \$s1$	Dados de um registrador para a memória
	load half unsigned	lhu \$s1,100(\$s2)	$\$s1 = \text{Memória}[\$s2 + 100]$	Halfword da memória para registrador
	store half	sh \$s1,100(\$s2)	$\text{Memória}[\$s2 + 100] = \$s1$	Halfword de um registrador para memória
	load byte unsigned	lbu \$s1,100(\$s2)	$\$s1 = \text{Memória}[\$s2 + 100]$	Byte da memória para registrador
	store byte	sb \$s1,100(\$s2)	$\text{Memória}[\$s2 + 100] = \$s1$	Byte de um registrador para memória
	load upper immediate	lui \$s1,100	$\$s1 = 100 * 2^{16}$	Carrega constante nos 16 bits mais altos
Lógica	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Três operadores em registrador; AND bit a bit
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \$s3$	Três operadores em registrador; OR bit a bit
	nor	nor \$s1,\$s2,\$s3	$\$s1 = -(\$s2 \$s3)$	Três operadores em registrador; NOR bit a bit
	and immediate	andi \$s1,\$s2,100	$\$s1 = \$s2 \& 100$	AND bit a bit registrador com constante
	or immediate	ori \$s1,\$s2,100	$\$s1 = \$s2 100$	OR bit a bit registrador com constante
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 << 10$	Deslocamento à esquerda por constante
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 >> 10$	Deslocamento à direita por constante
Desvio condicional	branch on equal	beq \$s1,\$s2,25	If ($\$s1 == \$s2$) go to PC + 4 + 100	Testa igualdade; desvio relativo ao PC
	branch on not equal	bne \$s1,\$s2,25	If ($\$s1 != \$s2$) go to PC + 4 + 100	Testa desigualdade; relativo ao PC
	set on less than	slt \$s1,\$s2,\$s3	If ($\$s2 < \$s3$) \$s1 = 1; else \$s1 = 0	Compara menor que; usado com beq, bne
	set less than immediate	slti \$s1,\$s2,100	If ($\$s2 < 100$) \$s1 = 1; else \$s1 = 0	Compara menor que constante
	set less than unsigned	sltu \$s1,\$s2,\$s3	If ($\$s2 < \$s3$) \$s1 = 1; else \$s1 = 0	Compara menor que; números sem sinal
	set less than immediate unsigned	sltiu \$s1,\$s2,100	If ($\$s2 < 100$) \$s1 = 1; else \$s1 = 0	Compara < constante; números sem sinal
	jump	j 2500	go to 10000	Desvia para endereço de destino
Desvio incondicional	jump register	jr \$ra	go to \$ra	Para switch e retorno de procedimento
	jump and link	jal 2500	\$ra = PC + 4; go to 10000	Para chamada de procedimento

FIGURA 3.4 Arquitetura do MIPS revelada até aqui. Para economizar espaço na tabela, ela não inclui os registradores e a memória, encontrados na Figura 3.1. A cor indica as partes reveladas desde a Figura 3.1. A linguagem de máquina do MIPS também está no Guia de referência rápida.

Algumas linguagens de programação permitem a aritmética de inteiros em complemento a dois com variáveis declaradas com um byte e meio. Que instruções do MIPS seriam usadas?

Verifique você mesmo

1. Leitura com lbu, lhu; aritmética com add, sub, mult, div; depois, armazenamento usando sb, sh.
2. Leitura com lb, lh; aritmética com add, sub, mult, div; depois, armazenamento usando sb, sh.
3. Leitura com lb, lh; aritmética com add, sub, mult, div, usando and para mascarar o resultado com 8 ou 16 bits após cada operação; depois, armazenamento usando sb, sh.

Detalhamento: no texto anterior, dissemos que você copia o EPC para um registrador por meio de `mfcc0` e depois retorna ao código interrompido por meio de jump register. Isso leva a uma pergunta interessante: como você primeiro precisa transferir o EPC para um registrador a fim de usar com jump register, como jump register pode retornar ao código interrompido e restaurar os valores originais de todos os registradores? Você restaura os registradores antigos primeiro, destruindo assim seu endereço de retorno do EPC, que colocou em um registrador para uso em jump register, ou restaura todos os registradores, menos aquele com o endereço de retorno, para que possa desviar – significando que uma exceção resultaria em alterar esse único registrador a qualquer momento durante a execução do programa! Nenhuma dessas opções é satisfatória.

Para auxiliar o hardware nesse dilema, os programadores MIPS concordaram em reservar os registradores `$k0` e `$k1` para o sistema operacional; esses registradores não são restaurados nas exceções. Assim como os compiladores MIPS evitam o uso do registrador `$at`, de modo que o montador possa utilizá-lo como um registrador temporário (ver a Seção “Interface Hardware/Software”, no Capítulo 2), os compiladores também se abstêm do uso dos registradores `$k0` e `$k1`, para que fiquem disponíveis para o sistema operacional. As rotinas de exceção colocam o endereço de retorno em um desses registradores e depois usam o jump register para armazenar o endereço da instrução.

Multiplicação é
vexação,
Divisão também
é ruim;
A regra de três
me intriga,
E a prática me
deixa louco.
Anônimo,
manuscrito de
Elizabeth, 1570

3.4

Multiplicação

Agora que completamos a explicação de adição e subtração, estamos prontos para montar a operação mais vexatória da multiplicação.

Entretanto, primeiro, vamos rever a multiplicação de números decimais à mão para nos lembrar das etapas e dos nomes dos operandos. Por motivos que logo se tornarão claros, limitamos esse exemplo decimal ao uso apenas dos dígitos 0 e 1. Multiplicando 1000_{dec} por 1001_{dec} :

$$\begin{array}{r}
 \text{Multiplicando} & 1000_{dec} \\
 \text{Multiplicador} & \times \quad 1001_{dec} \\
 \hline
 & 1000 \\
 & 0000 \\
 & 0000 \\
 & 1000 \\
 \hline
 \text{Produto} & 1001000_{dec}
 \end{array}$$

O primeiro operando é chamado *multiplicando* e o segundo é o *multiplicador*. O resultado final é chamado *produto*. Como você pode se lembrar, o algoritmo aprendido na escola é pegar os dígitos do multiplicador um a um, da direita para a esquerda, calculando a multiplicação do multiplicando pelo único dígito do multiplicador e deslocando o produto intermediário um dígito para a esquerda dos produtos intermediários anteriores.

A primeira observação é que o número de dígitos no produto é muito maior do que o número no multiplicando ou no multiplicador. De fato, se ignorarmos os bits de sinal, o tamanho da multiplicação de um multiplicando de n bits por um multiplicador de m bits é um produto que possui $n + m$ bits de largura. Ou seja, $n + m$ bits são necessários para representar todos os produtos possíveis. Logo, como na adição, a multiplicação precisa lidar com o overflow, pois constantemente desejamos um produto de 32 bits como resultado da multiplicação de dois números de 32 bits.

Neste exemplo, restringimos os dígitos decimais a 0 e 1. Com somente duas opções, cada etapa da multiplicação é simples:

1. Basta colocar uma cópia do multiplicando ($1 \times$ multiplicando) no lugar apropriado se o dígito do multiplicador for 1, ou
2. Colocar 0 ($0 \times$ multiplicando) no lugar apropriado se o dígito for 0.

Embora o exemplo decimal anterior utilize apenas 0 e 1, a multiplicação de números binários sempre usa 0 e 1 e, por isso, sempre oferece apenas essas duas opções.

Agora que já revisamos os fundamentos tradicionais da multiplicação, a próxima etapa é mostrar o hardware de multiplicação altamente otimizado. Quebramos essa tradição na crença de que você entenderá melhor vendo a evolução do hardware e do algoritmo de multiplicação no decorrer das diversas gerações. Por enquanto, vamos supor que estamos multiplicando apenas números positivos.

Versão seqüencial do algoritmo e hardware de multiplicação

Esse projeto imita o algoritmo que aprendemos na escola; o hardware aparece na Figura 3.5. Desenhamos o hardware de modo que os dados fluam de cima para baixo, para que fique mais semelhante à técnica do lápis e papel.

Vamos supor que o multiplicador esteja no registrador Multiplicador de 32 bits e que o registrador Produto de 64 bits está inicializado como 0. Pelo exemplo de lápis e papel, visto anteriormente, fica claro que precisaremos mover o multiplicando para a esquerda um dígito a cada passo, pois pode ser somado aos produtos intermediários. Durante 32 etapas, um multiplicando de 32 bits moveria 32 bits para a esquerda. Logo, precisamos de um registrador Multiplicando de 64 bits, inicializado com o multiplicando de 32 bits na metade direita e 0 na metade esquerda. Esse registrador, em seguida, é deslocado 1 bit para a esquerda a cada etapa, para alinhar o multiplicando com a soma sendo acumulada no registrador Produto de 64 bits.

A Figura 3.6 mostra as três etapas clássicas necessárias para cada bit. O bit menos significativo do multiplicador (Multiplicador0) determina se o multiplicando é somado ao registrador Produto. O deslocamento à esquerda na etapa 2 tem o efeito de mover os operandos intermediários para a esquerda, assim como na multiplicação manual. O deslocamento à direita na etapa 3 nos indica o próximo bit do multiplicador a ser examinado na iteração seguinte. Essas três etapas são repetidas 32 vezes, para obter o produto. Se cada etapa usasse um ciclo de clock, esse algoritmo exigiria quase 100 ciclos de clock para multiplicar dois números de 32 bits. A importância relativa de operações aritméticas, como a multiplicação, varia com o programa, mas a soma e a subtração podem ser de 5 a 100 vezes mais populares do que a multiplicação. Como consequência, a multiplicação pode demorar vários ciclos de clock sem afetar o desempenho de forma significativa. Mesmo assim, a lei de Amdahl (ver Capítulo 4) nos lembra que até mesmo uma freqüência moderada para uma operação lenta pode limitar o desempenho.

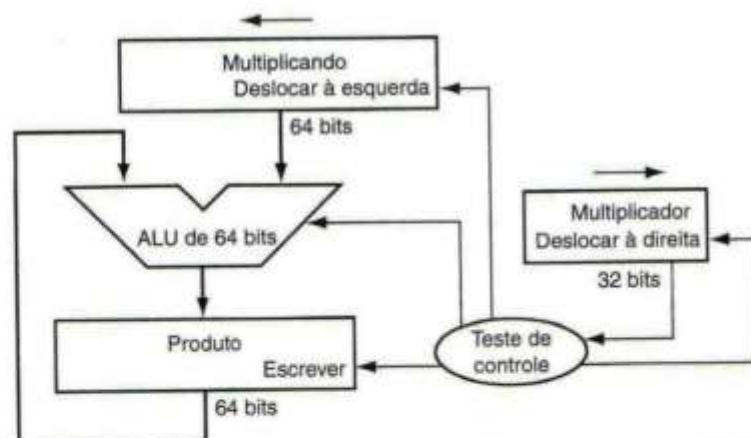


FIGURA 3.5 Primeira versão do hardware de multiplicação. O registrador do Multiplicando, a ALU, e o registrador do Produto possuem 64 bits de largura, apenas com o registrador do Multiplicador contendo 32 bits. O multiplicando com 32 bits começa na metade direita do registrador do Multiplicando e é deslocado à esquerda 1 bit em cada etapa. O multiplicador é deslocado na direção oposta em cada etapa. O algoritmo começa com o produto inicializado com 0. O controle decide quando deslocar os registradores Multiplicando e Multiplicador e quando escrever novos valores no registrador do Produto.

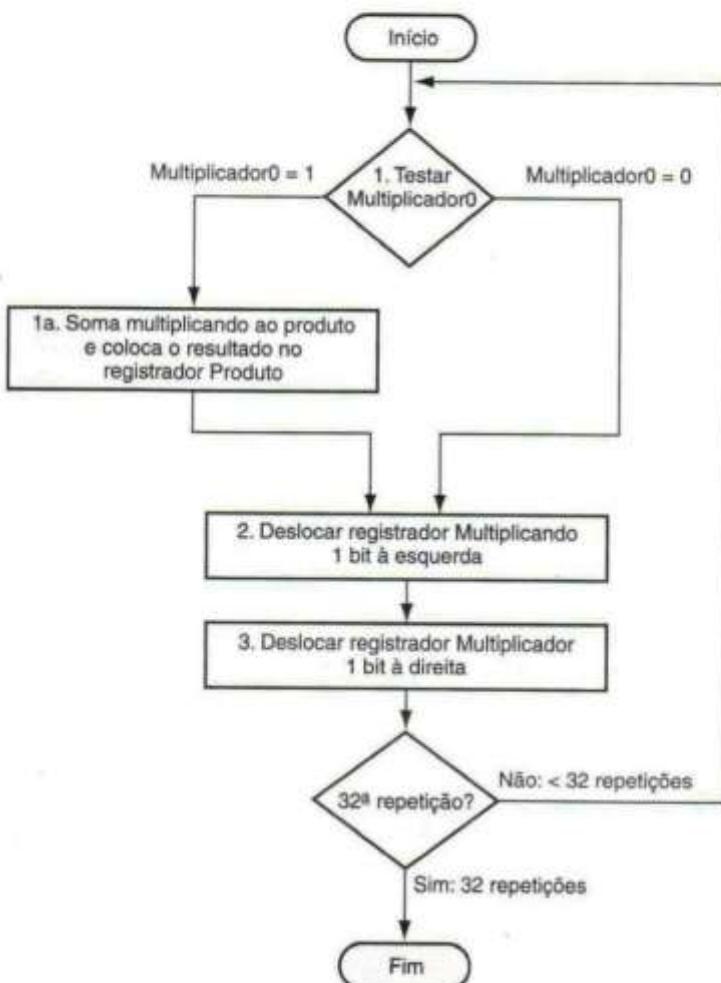


FIGURA 3.6 O primeiro algoritmo de multiplicação, usando o hardware mostrado na Figura 3.5. Se o bit menos significativo do multiplicador for 1, some o multiplicando ao produto. Caso contrário, vá para a etapa seguinte. Desloque o multiplicando para a esquerda e o multiplicador para a direita nas duas etapas seguintes. Essas três etapas são repetidas 32 vezes.

Esse algoritmo e o hardware são facilmente refinados para usar 1 ciclo de clock por etapa. O aumento de velocidade vem da realização das operações em paralelo: o multiplicador e o multiplicando são deslocados enquanto o multiplicando é somado ao produto se o bit do multiplicador for 1. O hardware simplesmente precisa garantir que testará o bit da direita do multiplicador e receberá a versão previamente deslocada do multiplicando. O hardware normalmente é otimizado ainda mais para dividir a largura do somador e dos registradores ao meio, observando onde existem partes não utilizadas dos registradores e somadores. A Figura 3.7 mostra o hardware revisado.

Interface hardware/software

Substituir a aritmética por deslocamentos também pode ocorrer quando se multiplica por constantes. Alguns compiladores substituem multiplicações por constantes curtas com uma série de deslocamentos e adições. Como deslocar um bit à esquerda representa um número duas vezes maior na base 2, o deslocamento de bits para a esquerda tem o mesmo efeito de multiplicar por uma potência de 2. Como dissemos no Capítulo 2, quase todo compilador realizará a otimização por redução de força substituindo uma multiplicação por uma potência de 2 por um deslocamento à esquerda.

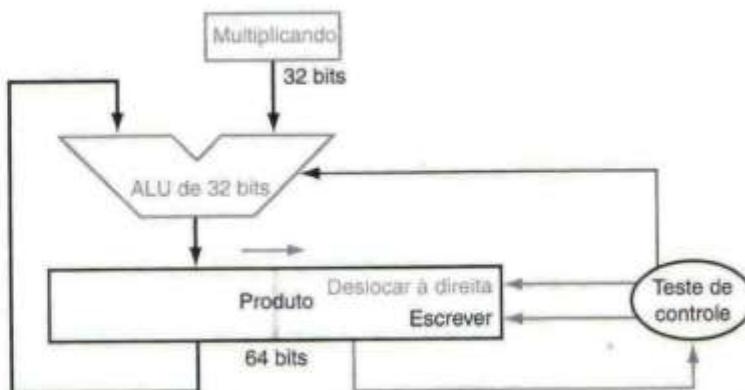


FIGURA 3.7 Versão refinada do hardware de multiplicação. Compare com a primeira versão na Figura 3.5. O registrador Multiplicando, a ALU, e o registrador Multiplicador possuem 32 bits de extensão, com somente o registrador Produto restando nos 64 bits. Agora, o produto é deslocado para a direita. O registrador Multiplicador separado também desapareceu. O multiplicador é colocado na metade direita do registrador Produto. Essas mudanças estão destacadas.

UM ALGORITMO DE MULTIPLICAÇÃO

Usando números de 4 bits para economizar espaço, multiplique $2_{dec} \times 3_{dec}$, ou $0010_{bin} \times 0011_{bin}$.

A Figura 3.8 mostra o valor de cada registrador para cada uma das etapas rotuladas de acordo com a Figura 3.6, com o valor final de $0000\ 0110_{bin}$ ou 6_{dec} . A cor é usada para indicar os valores de registrador que mudam nessa etapa, e o bit circulado é aquele examinado para determinar a operação da próxima etapa.

Multiplicação com sinal

Até aqui, tratamos de números positivos. O modo mais fácil de entender como tratar dos números com sinal é primeiro converter o multiplicador e o multiplicando para números positivos e depois lembrar dos sinais originais. Os algoritmos deverão, então, ser executados por 31 iterações, deixando os sinais fora do cálculo. Conforme aprendemos na escola, o produto só será negativo se os sinais originais forem diferentes.

EXEMPLO

RESPOSTA

Iteração	Etapa	Multiplicador	Multiplicando	Produto
0	Valores iniciais	0011	0000 0010	0000 0000
1	1a: 1 => Prod = Prod + Multiplicando	0011	0000 0010	0000 0010
	2: Shift left Multiplicando	0011	0000 0100	0000 0010
	3: Shift right Multiplicador	0001	0000 0100	0000 0010
2	1a: 1 => Prod = Prod + Multiplicando	0001	0000 0100	0000 0110
	2: Shift left Multiplicando	0001	0000 1000	0000 0110
	3: Shift right Multiplicador	0000	0000 1000	0000 0110
3	1: 0 => sem operação	0000	0000 1000	0000 0110
	2: Shift left Multiplicando	0000	0001 0000	0000 0110
	3: Shift right Multiplicador	0000	0001 0000	0000 0110
4	1: 0 => sem operação	0000	0001 0000	0000 0110
	2: Shift left Multiplicando	0000	0010 0000	0000 0110
	3: Shift right Multiplicador	0000	0010 0000	0000 0110

FIGURA 3.8 Exemplo de multiplicação usando o algoritmo da Figura 3.6. O bit examinado para determinar a próxima etapa está circulado.

Acontece que o último algoritmo funcionará para números com sinais se nos lembarmos de que os números com que estamos lidando possuem dígitos infinitos, e que só os estamos representando com 32 bits. Logo, as etapas de deslocamento precisariam estender o sinal do produto para números com sinal. Quando o algoritmo terminar, a word menos significativa terá o produto de 32 bits.

Multiplicação mais rápida

A Lei de Moore ofereceu tantos recursos que os projetistas de hardware agora podem construir um hardware de multiplicação muito mais rápido. Não importa se o multiplicando deve ser somado ou não, isso é conhecido no início da multiplicação analisando cada um dos 32 bits do multiplicador. Multiplicações mais rápidas são possíveis basicamente fornecendo um somador de 32 bits para cada bit do multiplicador: uma entrada é o AND do multiplicando pelo bit do multiplicador e a outra é a saída de um somador anterior. A Figura 3.9 mostra como seriam conectados.

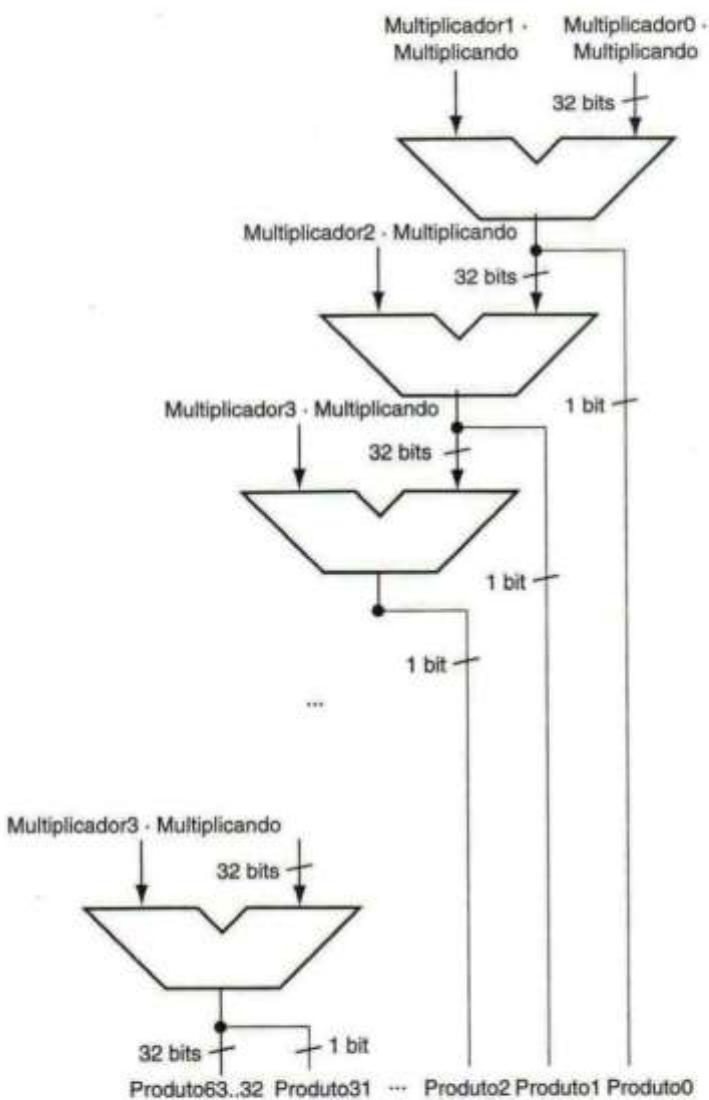


FIGURA 3.9 Hardware da multiplicação rápida. Em vez de usar um único somador de 32 bits 32 vezes, esse hardware “desenrola o loop” para usar 32 somadores. Cada somador produz uma soma de 32 bits e um bit de carry. O bit menos significativo é o bit do produto, e o carry com os 31 bits mais significativos da soma são passados adiante para o próximo somador.

Por que esse hardware é muito mais rápido? O multiplicador seqüencial paga o overhead de um clock para cada bit do produto. Esse multiplicador que usa um array de somadores, não. Um segundo motivo é que essa grande coleção de somadores serve para muitas otimizações, para gerar ainda mais melhorias. Um exemplo é o uso de *somadores para salvar carry*, para somar uma coluna muito grande de números; ver Exercícios 3.23 e 3.45. Um terceiro motivo é que é fácil usar um pipeline nesse projeto para que possam ser realizadas muitas multiplicações simultaneamente (ver Capítulo 6).

Multiplicação no MIPS

O MIPS oferece um par separado de registradores de 32 bits, para conter o produto de 64 bits, chamados *Hi* e *Lo*. Para produzir um produto com ou sem o devido sinal, o MIPS possui duas instruções: *multiply* (*mult*) e *multiply unsigned* (*multu*). Para apanhar o produto de 32 bits inteiro, o programador usa *move from lo* (*mflo*). O montador MIPS gera uma pseudo-instrução para multiplicar, que especifica três registradores de uso geral, gerando instruções *mflo* e *mfhi* para colocar o produto nos registradores.

Resumo

A multiplicação é feita pelo hardware simples de deslocamento e adição, derivado do método de lápis e papel que aprendemos na escola. Os compiladores utilizam até mesmo as instruções de deslocamento para multiplicações por potências de dois.

Interface hardware/software

As duas instruções *multiply* do MIPS ignoram o overflow, de modo que fica a critério do software verificar se o produto é muito grande para caber nos 32 bits. Não existe overflow se *Hi* for 0 para *multu* ou o sinal replicado de *Lo* para *mult*. A instrução *move from hi* (*mfhi*) pode ser usada para transferir *Hi* para um registrador de uso geral, a fim de testar o overflow.

3.5 Divisão

A operação reciproca da multiplicação é a divisão, uma operação que é ainda menos freqüente e ainda mais peculiar. Ela oferece até mesmo a oportunidade de realizar uma operação matematicamente inválida: dividir por 0.

Vamos começar com um exemplo de divisão longa usando números decimais, para lembrar os nomes dos operandos e do algoritmo de divisão que aprendemos na escola. Por motivos semelhantes aos da seção anterior, vamos limitar os dígitos decimais a apenas 0 ou 1. O exemplo é a divisão de 1.001.010_{dec} por 1000_{dec}:

Divisor	1001 _{dec}	Quociente
Dividendo	1001010 _{dec}	Dividendo
	<u>-1000</u>	
	10	
	101	
	1010	
	<u>-1000</u>	
	10 _{dec}	Resto

Divide et impera.

Tradução do latim para "Dividir e conquistar", máxima política antiga, citada por Maquiavel, 1532

dividendo Um número sendo dividido.

divisor Um número pelo qual o dividendo é dividido.

quociente O resultado principal de uma divisão; um número que, quando multiplicado pelo divisor e somado ao resto, produz o dividendo.

resto O resultado secundário de uma divisão; um número que, quando somado ao produto do quociente pelo divisor, produz o dividendo.

Os dois operandos (**dividendo** e **divisor**) e o resultado (**quociente**) da divisão são acompanhados por um segundo resultado, chamado **resto**. Veja aqui outra maneira de expressar o relacionamento entre os componentes:

$$\text{Dividendo} = \text{Quociente} \times \text{Divisor} + \text{Resto}$$

onde o resto é menor do que o divisor. Raramente os programas utilizam a instrução de divisão só para obter o resto, ignorando o quociente.

O algoritmo básico de divisão, que aprendemos na escola, tenta ver o quanto um número pode ser subtraído, criando um dígito do quociente em cada tentativa. Nossa exemplo decimal cuidadosamente selecionado usa apenas os números 0 e 1, de modo que é fácil descobrir quantas vezes o divisor cabe na parte do dividendo: deve ser 0 ou 1. Os números binários contêm apenas 0 ou 1, de modo que a divisão binária é restrita a essas duas opções, simplificando assim a divisão binária.

Vamos supor que o dividendo e o divisor sejam positivos e, logo, o quociente e o resto sejam não-negativos. Os operandos da divisão e os dois resultados são valores de 32 bits, e ignoraremos o sinal por enquanto.

Algoritmo e hardware de divisão

A Figura 3.10 mostra o hardware para imitar nosso algoritmo da escola. Começamos com o registrador Quociente de 32 bits definido como 0. Cada iteração do algoritmo precisa deslocar o divisor para a direita um dígito, de modo que começaremos com o divisor colocado na metade esquerda do registrador Divisor de 64 bits e o deslocaremos para a direita 1 bit a cada etapa, para alinhá-lo com o dividendo. O registrador Resto é inicializado com o dividendo.

A Figura 3.11 mostra três etapas do primeiro algoritmo de divisão. Ao contrário dos humanos, o computador não é inteligente o bastante para saber, com antecedência, se o divisor é menor do que o dividendo. Ele primeiro precisa subtrair o divisor na etapa 1; lembre-se de que é assim que realizamos a comparação na instrução set on less than. Se o resultado for positivo, o divisor foi menor ou igual ao dividendo, de modo que geramos um 1 no quociente (etapa 2a). Se o resultado é negativo, a próxima etapa é restaurar o valor original, somando o divisor de volta ao resto e gerar um 0 no quociente (etapa 2b). O divisor é deslocado para a direita e depois repetimos. O resto e o quociente serão encontrados em seus registradores de mesmo nome depois que as iterações terminarem.

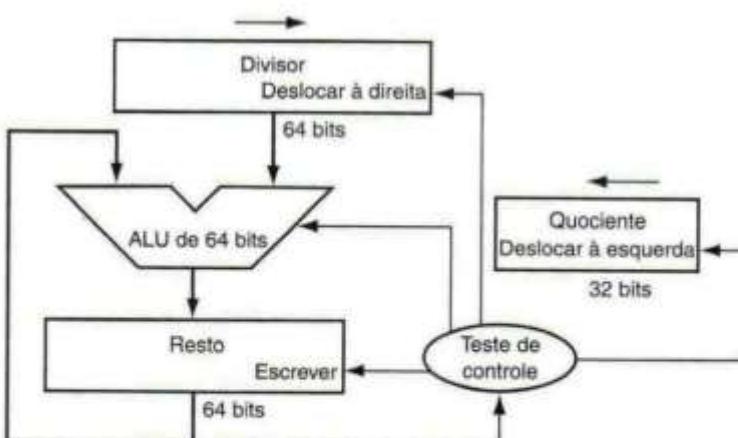


FIGURA 3.10 Primeira versão do hardware de divisão. O registrador Divisor, a ALU e o registrador Resto possuem 64 bits de largura, com apenas o registrador Quociente tendo 32 bits. O divisor de 32 bits começa na metade esquerda do registrador Divisor e é deslocado 1 bit para a direita em cada iteração. O resto é inicializado com o dividendo. O controle decide quando deslocar os registradores Divisor e Quociente e quando escrever o novo valor para o registrador Resto.

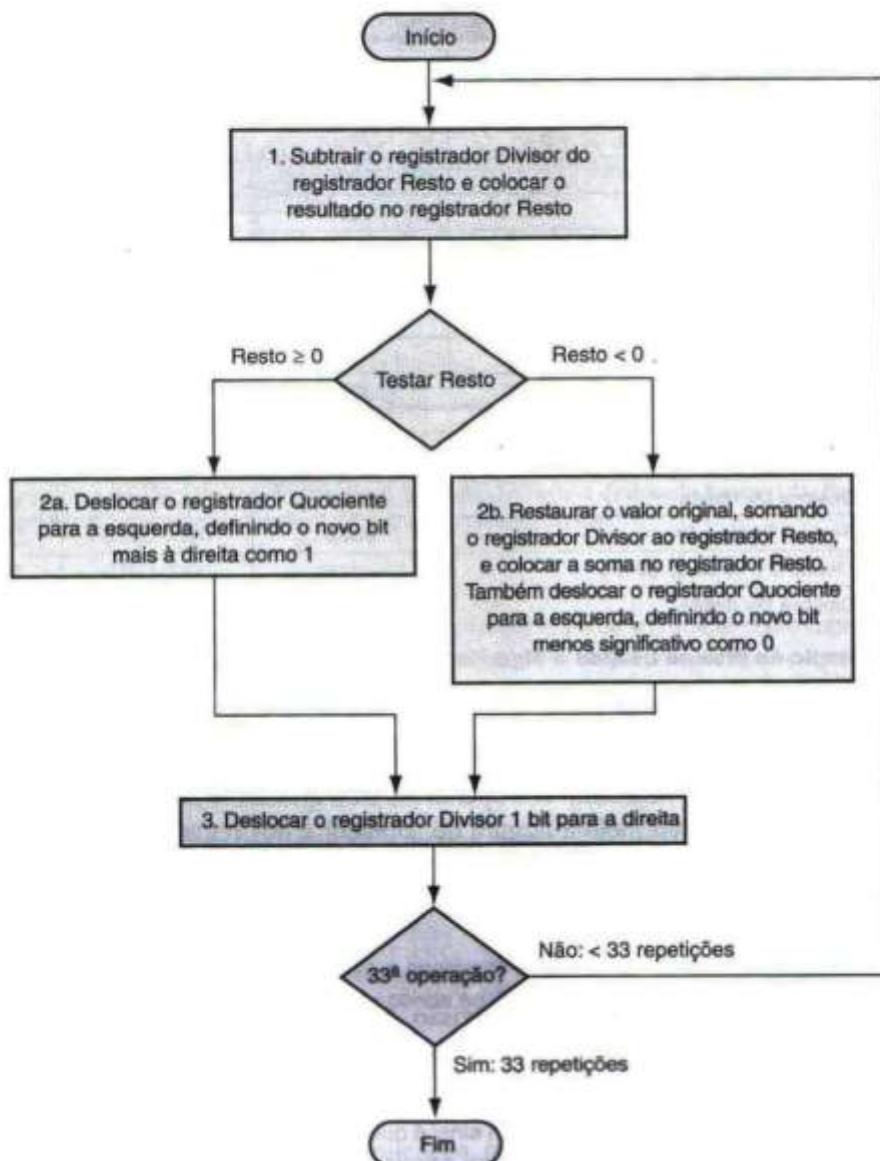


FIGURA 3.11 Um algoritmo de divisão, usando o hardware da Figura 3.10. Se o Resto é positivo, o divisor coube no dividendo, de modo que a etapa 2a gera um 1 no quociente. Um Resto negativo após a etapa 1 significa que o divisor não coube no dividendo, de modo que a etapa 2b gera um 0 no quociente e soma o divisor ao resto, revertendo assim a subtração da etapa 1. O deslocamento final, na etapa 3, alinha o divisor corretamente, em relação ao dividendo, para a próxima iteração. Essas etapas são repetidas 33 vezes.

UM ALGORITMO DE DIVISÃO

Usando uma versão de 4 bits do algoritmo para economizar páginas, vamos tentar dividir 7_{dec} por 2_{dec} , ou $0000\ 0111_{bin}$ por 0010_{bin} .

A Figura 3.12 mostra o valor de cada registrador para cada uma das etapas, com o quociente sendo 3_{dec} e o resto sendo 1_{dec} . Observe que o teste na etapa 2 (se o resto é positivo ou negativo) simplesmente testa se o bit de sinal do registrador Resto é um 0 ou um 1. O requisito surpreendente desse algoritmo é que ele utiliza $n + 1$ etapas para obter o quociente e resto corretos.

EXEMPLO

RESPOSTA

Iteração	Etapa	Quociente	Divisor	Resto
0	Valores iniciais	0000	0010 0000	0000 0111
1	1: Resto = Resto - Div	0000	0010 0000	0010 0111
	2b: Resto < 0 \Rightarrow +Div, sif Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Desloca Div direita	0000	0001 0000	0000 0111
2	1: Resto = Resto - Div	0000	0001 0000	0011 0111
	2b: Resto < 0 \Rightarrow +Div, sif Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Desloca Div direita	0000	0000 1000	0000 0111
3	1: Resto = Resto - Div	0000	0000 1000	0011 1111
	2b: Resto < 0 \Rightarrow +Div, sif Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Desloca Div direita	0000	0000 0100	0000 0111
4	1: Resto = Resto - Div	0000	0000 0100	0000 0011
	2a: Resto \geq 0 \Rightarrow sif Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Desloca Div direita	0001	0000 0010	0000 0011
5	1: Resto = Resto - Div	0001	0000 0010	0000 0001
	2a: Resto \geq 0 \Rightarrow sif Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Desloca Div direita	0011	0000 0001	0000 0001

FIGURA 3.12 Exemplo de divisão usando o algoritmo da Figura 3.11. O bit examinado para determinar a próxima etapa está em destaque.

Esse algoritmo e esse hardware podem ser refinados para que sejam mais rápidos e menos dispendiosos. A rapidez vem do deslocamento dos operandos e do quociente no mesmo momento da subtração. Essa melhoria divide ao meio a largura do somador e dos registradores, observando onde existem partes não usadas dos registradores e somadores. A Figura 3.13 mostra o hardware revisado.

Divisão com sinal

Até aqui, ignoramos os números com sinal na divisão. A solução mais simples é lembrar os sinais do divisor e do dividendo e depois negar o quociente se os sinais forem diferentes.

Detalhamento: uma complicação da divisão com sinal é que também temos de definir o sinal do resto. Lembre-se de que a seguinte equação precisa ser sempre mantida:

$$\text{Dividendo} = \text{Quociente} \times \text{Divisor} + \text{Resto}$$

Para entender como definir o sinal do resto, vejamos o exemplo da divisão de todas as combinações de $\pm 7_{\text{dec}}$ por $\pm 2_{\text{dec}}$. O primeiro caso é fácil:

$$+7 \div +2: \text{Quociente} = +3, \text{Resto} = +1$$

Verificando os resultados:

$$7 = 3 \times 2 + (+1) = 6 + 1$$

Se você mudar o sinal do dividendo, o quociente também precisa mudar:

$$-7 \div +2: \text{Quociente} = -3$$

Reescrevendo nossa fórmula básica para calcular o resto:

$$\text{Resto} = (\text{Dividendo} - \text{Quociente} \times \text{Divisor}) = -7 - (-3 \times 2) = -7 - (-6) = -1$$

Assim,

$$-7 \div +2: \text{Quociente} = -3, \text{Resto} = -1$$

Verificando os resultados novamente:

$$-7 = -3 \times 2 + (-1) = -6 - 1$$

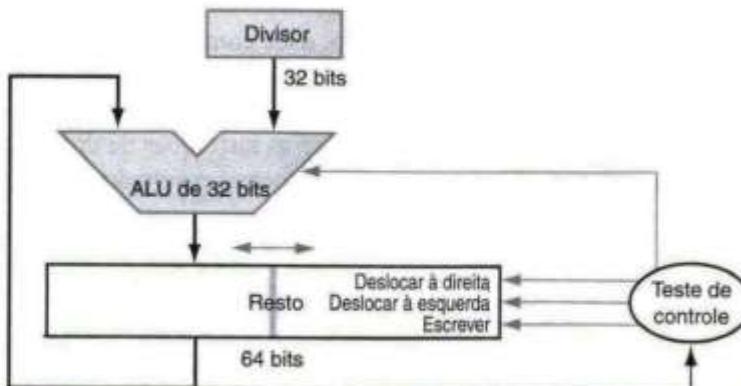


FIGURA 3.13 Uma versão melhorada do hardware de divisão. O registrador Divisor, a ALU e o registrador Quociente possuem 32 bits de largura, com apenas o registrador Resto ficando com 64 bits. Em comparação com a Figura 3.10, os registradores ALU e Divisor são divididos ao meio, e o resto é deslocado à esquerda. Essa versão também combina o registrador Quociente com a metade direita do registrador Resto.

O motivo pelo qual a resposta não é um quociente de -4 e um resto de $+1$, que também caberia nessa fórmula, é que o valor absoluto do quociente mudaria dependendo do sinal do dividendo e do divisor! Logicamente, se

$$-(x + y) \neq (-x) \div y$$

a programação seria um desafio ainda maior. Esse comportamento anômalo é evitado seguindo-se a regra de que o dividendo e o resto devem ter os mesmos sinais, não importa quais sejam os sinais do divisor e do quociente.

Calculamos as outras combinações seguindo a mesma regra:

$$+7 \div -2: \text{Quociente} = -3, \text{Resto} = +1$$

$$-7 \div -2: \text{Quociente} = +3, \text{Resto} = -1$$

Assim, o algoritmo de divisão com sinal nega o quociente se os sinais dos operandos foram opostos e faz com que o sinal do resto diferente de zero corresponda ao dividendo.

Divisão mais rápida

Usamos 32 somadores para agilizar a multiplicação, mas não podemos fazer o mesmo truque para a divisão. O motivo é que precisamos saber o sinal da diferença antes de podermos realizar a próxima etapa do algoritmo, enquanto, com a multiplicação, poderíamos calcular os 32 produtos parciais imediatamente.

Existem técnicas para produzir mais de um bit do quociente por bit. A técnica de *divisão SRT* tenta descobrir vários bits do quociente por etapa, usando uma pesquisa numa tabela baseada nos bits mais significativos do dividendo e do resto. Ela conta com as etapas subsequentes para corrigir escolhas erradas. Um valor típico hoje é 4 bits. A chave é descobrir o valor para subtrair. Com a divisão binária, existe somente uma única opção. Esses algoritmos utilizam 6 bits do resto e 4 bits do divisor para indexar uma tabela que determina a opção para cada etapa.

A precisão desse método rápido depende de haver valores apropriados na tabela de pesquisa. A falácia apresentada na Seção 3.8 mostra o que pode acontecer se a tabela estiver incorreta.

Divisão no MIPS

Você já pode ter observado que o mesmo hardware seqüencial pode ser usado para multiplicação e divisão nas Figuras 3.7 e 3.13. O único requisito é um registrador de 64 bits, que pode deslocar para a esquerda ou para a direita e uma ALU de 32 bits que soma ou subtrai. Logo, o MIPS utiliza os registradores Hi e Lo de 32 bits, tanto para multiplicação quanto para divisão. Como poderia-

mos esperar do algoritmo anterior, H_i contém o resto, e L_o contém o quociente após o término da instrução de divisão.

Para lidar com inteiros com sinal e inteiros sem sinal, o MIPS possui duas instruções: *divide* (*div*) e *divide unsigned* (*divu*). O montador MIPS permite que as instruções de divisão especifiquem três registradores, gerando as instruções *mflo* ou *mfhi* para colocar o resultado desejado em um registrador de uso geral.

Resumo

O suporte de hardware comum para multiplicação e divisão permite que o MIPS ofereça um único par de registradores de 32 bits usados tanto para multiplicar quanto para dividir. A Figura 3.14 resume os acréscimos à arquitetura MIPS das duas últimas seções.

Interface hardware/software

Instruções de divisão MIPS ignoram o overflow, de modo que o software precisa determinar se o quociente é muito grande. Além do overflow, a divisão também pode resultar em um cálculo impróprio: divisão por 0. Alguns computadores distinguem esses dois eventos anômalos. O software MIPS precisa verificar o divisor para descobrir a divisão por 0 e também o overflow.

Detalhamento: um algoritmo ainda mais rápido não soma imediatamente o divisor se o resto for negativo. Ele simplesmente soma o dividendo ao resto deslocado na etapa seguinte, pois $(r+d) \times 2 - d = r \times 2 + d \times 2 - d = r \times 2 + d$. Esse algoritmo de divisão sem restauração, que usa 1 clock por etapa, é explorado ainda mais no Exercício 3.25; o algoritmo aqui apresentado é chamado de divisão com restauração.

*A velocidade não
o leva a lugar
algum se você
estiver na
direção errada.*
Provérbio
americano

3.6

Ponto flutuante

Indo além de inteiros com e sem sinal, as linguagens de programação admitem números com frações, que são chamados *reais* na matemática. Aqui estão alguns exemplos de números reais:

$3,14159265\dots_{\text{dec}}$ (π)

$2,71828\dots_{\text{dec}}$ (e)

$0,000000001_{\text{dec}}$ ou $1,0_{\text{dec}} \times 10^{-9}$ (segundos em um nanossegundo)

$3.155.760.000_{\text{dec}}$ ou $3,15576_{\text{dec}} \times 10^9$ (segundos em um século típico)

notação científica Uma notação que apresenta números com um único dígito à esquerda do ponto decimal.

normalizado Um número na notação de ponto flutuante que não possui 0s à esquerda do ponto decimal.

Observe que, no último caso, o número não representou uma fração pequena, mas foi maior do que poderíamos representar com um inteiro de 32 bits com sinal. A notação alternativa para os dois últimos números é chamada **notação científica**, que possui um único dígito à esquerda do ponto decimal. Um número na notação científica que não possui 0s à esquerda do ponto decimal é chamado de número **normalizado**, que é o modo normal como o escrevemos. Por exemplo, $1,0_{\text{dec}} \times 10^{-9}$ está em notação científica normalizada, mas $0,1_{\text{dec}} \times 10^{-8}$ e $10,0_{\text{dec}} \times 10^{-10}$ não são.

Assim como podemos mostrar números decimais em notação científica, também podemos mostrar números binários em notação científica:

$$1,0_{\text{bin}} \times 2^{-1}$$

Assembly do MIPS

Categoría	Instrução	Exemplo	Significado	Comentários
Aritmética	add	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Três operandos; overflow detectado
	subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	Três operandos; overflow detectado
	add immediate	addi \$s1,\$s2,100	\$s1=\$s2 + 100	+ constante; overflow detectado
	add unsigned	addu \$s1, \$s2, \$s3	\$s1 = \$s2 + \$s3	Três operandos; overflow não detectado
	subtract unsigned	subu \$s1, \$s2, \$s3	\$s1 = \$s2 - \$s3	Três operandos; overflow não detectado
	add immediate unsigned	addiu \$s1, \$s2, 100	\$s1=\$s2 + 100	+ constante; overflow não detectado
	move from coprocessor register	mfc0 \$s1, \$epc	\$s1 = \$epc	Copiar PC de exceção + registradores especiais
	multiply	mult \$s2, \$s3	Hi, Lo= \$s2 x \$s3	Produto com sinal, 64 bits, em Hi, Lo
	multiply unsigned	multu \$s2, \$s3	Hi, Lo= \$s2 x \$s3	Produto sem sinal, 64 bits, em Hi, Lo
	divide	div \$s2, \$s3	Lo= \$s2/ \$s3, Hi = \$s2 mod \$s3	Lo = quociente, Hi = resto
	divide unsigned	divu \$s2, \$s3	Lo = \$s2 / \$s3, Hi = \$s2 mod \$s3	Quociente e resto sem sinal
	move from Hi	mfhi \$s1	\$s1 = Hi	Usado para obter cópia de Hi
	move from Lo	mflo \$s1	\$s1 = Lo	Usado para obter cópia de Lo
Transferência de dados	load word	lw \$s1,100(\$s2)	\$s1 = Memória[\$s2 + 100]	Dados da memória para um registrador
	store word	sw \$s1,100(\$s2)	Memória[\$s2 + 100] = \$s1	Dados de um registrador para a memória
	load half unsigned	lhu \$s1,100(\$s2)	\$s1 = Memória[\$s2 + 100]	Halfword da memória para registrador
	store half	sh \$s1,100(\$s2)	Memória[\$s2 + 100] = \$s1	Halfword de um registrador para memória
	load byte unsigned	lbu \$s1,100(\$s2)	\$s1 = Memória[\$s2 + 100]	Byte da memória para registrador
	store byte	sb \$s1,100(\$s2)	Memória[\$s2 + 100] = \$s1	Byte de um registrador para memória
	load upper immediate	lui \$s1,100	\$s1 = 100 * 2 ¹⁶	Carrega constante nos 16 bits mais altos
Lógica	and	and \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	Três operadores em registrador; AND bit a bit
	or	or \$s1,\$s2,\$s3	\$s1 = \$s2 \$s3	Três operadores em registrador; OR bit a bit
	nor	nor \$s1,\$s2,\$s3	\$s1 = ~(\$s2 \$s3)	Três operadores em registrador; NOR bit a bit
	and immediate	andi \$s1,\$s2,100	\$s1 = \$s2 & 100	AND bit a bit registrador com constante
	or immediate	ori \$s1,\$s2,100	\$s1 = \$s2 100	OR bit a bit registrador com constante
	shift left logical	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	Deslocamento à esquerda por constante
	shift right logical	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	Deslocamento à direita por constante
Desvio condicional	branch on equal	beq \$s1,\$s2,25	If (\$s1 == \$s2) go to PC + 4 + 100	Testa igualdade; desvio relativo ao PC
	branch on not equal	bne \$s1,\$s2,25	If (\$s1 != \$s2) go to PC + 4 + 100	Testa desigualdade; relativo ao PC
	set on less than	slt \$s1,\$s2,\$s3	If (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compara menor que; usado com beq, bne
	set less than immediate	slti \$s1,\$s2,100	If (\$s2 < 100) \$s1 = 1; else \$s1 = 0	Compara menor que constante
	set less than unsigned	sltu \$s1,\$s2,\$s3	If(\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compara menor que; números sem sinal
	set less than immediate unsigned	sltiu \$s1,\$s2,100	If(\$s2 < 100) \$s1 = 1; else \$s1 = 0	Compara < constante; números sem sinal
	jump	j 2500	go to 10000	Desvia para endereço de destino
Desvio Incondicional	jump register	jr \$ra	go to \$ra	Para switch e retorno de procedimento
	jump and link	jal 2500	\$ra = PC + 4; go to 10000	Para chamada de procedimento

FIGURA 3.14 Arquitetura MIPS revelada até aqui. A memória e os registradores da arquitetura MIPS não estão incluídos por questões de espaço, mas esta seção acrescentou os registradores hi e lo para dar suporte à multiplicação e à divisão. A cor indica as partes reveladas desde a Figura 3.4. A linguagem de máquina do MIPS aparece no Guia de referência rápida, no inicio do livro.

Para manter um número binário na forma normalizada, precisamos de uma base que possamos aumentar ou diminuir exatamente pelo número de bits que o número precisa ser deslocado para ter um dígito diferente de zero à esquerda do ponto decimal. Somente uma base de 2 atende a nossa necessidade. Como a base não é 10, também precisamos de um novo nome para o ponto decimal; o *ponto binário* funcionará bem.

ponto flutuante

Aritmética computacional que representa os números em que o ponto binário não é fixo.

A aritmética computacional que admite tais números é chamada **ponto flutuante** porque representa os números em que o ponto binário não é fixo, como acontece para os inteiros. A linguagem de programação C utiliza o nome *float* para esses números. Assim como na notação científica, os números são representados como um único dígito diferente de zero à esquerda do ponto binário. Em binário, o formato é

$$1,xxxxxx_{\text{bin}} \times 2^{yyyy}$$

(Embora o computador represente o expoente na base 2, bem como o restante do número, para simplificar a notação, mostramos o expoente em decimal.)

Uma notação científica padrão para os números reais no formato normalizado oferece três vantagens. Ela simplifica a troca de dados, que incluem números em ponto flutuante; simplifica os algoritmos aritméticos de ponto flutuante, por saber que os números sempre estarão nessa forma; e aumenta a precisão dos números que podem ser armazenados em uma word, pois os 0s desnecessários são substituídos por dígitos reais à direita do ponto binário.

Representação em ponto flutuante

fração O valor, geralmente entre 0 e 1, colocado no campo de fração.

expoente No sistema de representação numérica da aritmética de ponto flutuante, o valor colocado no campo de expoente.

Um projetista de uma representação em ponto flutuante precisa encontrar um compromisso entre o tamanho da **fração** e o tamanho do **expoente**, pois um tamanho de word fixo significa que você precisa tirar um bit de um para acrescentar um bit ao outro. Essa troca é entre a precisão e o intervalo: aumentar o tamanho da fração melhora a precisão da fração, enquanto aumentar o tamanho do expoente aumenta o intervalo de números que podem ser representados. Conforme nosso guia de projetos do Capítulo 2 nos lembra, um bom projeto exige um bom compromisso.

Os números em ponto flutuante normalmente são múltiplos do tamanho de uma word. A representação de um número em ponto flutuante MIPS aparece a seguir, onde *s* é o sinal do número de ponto flutuante (1 significando negativo), expoente é o valor do campo de expoente com 8 bits (incluindo o sinal do expoente) e fração é o número de 23 bits. Essa representação é chamada *sinal e magnitude*, pois o sinal possui um bit separado do restante do número.

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
<i>s</i>	expoente																									fração					
	1 bit																									23 bits					

Em geral, os números em ponto flutuante estão no formato

$$(-1)^s \times F \times 2^E$$

F envolve o valor no campo de fração, e *E* envolve o valor no campo de expoente; o relacionamento exato com esses campos será explicado em breve. (Logo veremos que o MIPS faz algo ligeiramente mais sofisticado.)

Esses tamanhos escolhidos de expoente e fração dão à aritmética do computador MIPS um intervalo extraordinário. Frações quase tão pequenas quanto $2,0_{\text{dec}} \times 10^{-38}$ e números quase tão grandes quanto $2,0_{\text{dec}} \times 10^{38}$ podem ser representados em um computador. Infelizmente, extraordinário é diferente de infinito, de modo que ainda é possível que os números sejam muito grandes. Assim, interrupções por overflow podem ocorrer na aritmética de ponto flutuante e também na aritmética

de inteiros. Observe que **overflow** aqui significa que o expoente é muito grande para ser representado no campo de expoente.

O ponto flutuante também oferece um novo tipo de evento excepcional. Assim como os programadores desejariam saber quando calcularam um número muito grande para ser representado, também desejariam saber se a fração diferente de zero que estão calculando tornou-se tão pequena que não pode ser representada; os dois eventos poderiam resultar em um programa com respostas incorretas. Para distinguir do overflow, as pessoas chamam esse evento de **underflow**. Essa situação ocorre quando o expoente negativo é muito grande para caber no campo de expoente.

Uma maneira de reduzir as chances de underflow ou overflow é oferecer outro formato que tenha um expoente maior. Em C, esse número é chamado *double*, e as operações sobre doubles são indicadas como aritmética de ponto flutuante com **precisão dupla**; o ponto flutuante com **precisão simples** é o nome do formato anterior.

A representação de um número em ponto flutuante com precisão dupla utiliza duas words MIPS, como vemos a seguir, onde *s* ainda é o sinal do número, *expoente* é o valor do campo de expoente em 11 bits, e *fração* é o número de 52 bits na fração.

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							
<i>s</i>	expoente											fração																										
1 bit	11 bits											20 bits											fração (continuação)															
	32 bits																																					

A precisão dupla do MIPS permite números quase tão pequenos quanto $2,0_{\text{dec}} \times 10^{-308}$ e quase tão grandes quanto $2,0_{\text{dec}} \times 10^{308}$. Embora a precisão dupla não aumente o intervalo do expoente, sua principal vantagem é sua maior precisão, devido ao significando maior.

Esses formatos vão além do MIPS. Eles fazem parte do *padrão de ponto flutuante IEEE 754*, encontrado em praticamente todo computador inventado desde 1980. Esse padrão melhorou bastante tanto a facilidade de portar programas de ponto flutuante quanto a qualidade da aritmética computacional.

Para colocar ainda mais bits no significando, o IEEE 754 deixa implícito o bit 1 inicial dos números binários normalizados. Logo, o número tem, na realidade, 24 bits de largura na precisão simples (1 implícito e fração de 23 bits), e 53 bits de extensão na precisão dupla (1 + 52). Para ser exato, usamos o termo *significando* para representar o número de 24 ou 53 bits que é 1 mais a fração, e *fração* quando queremos dizer o número de 23 ou 52 bits. Como 0 não possui um 1 inicial, ele recebe o valor de expoente reservado 0, de modo que o hardware não lhe acrescente um 1 inicial.

Assim, 00...00_{bin} representa 0; a representação do restante dos números usa a forma de antes, com o 1 oculto sendo acrescentado:

$$(-1)^s \times (1 + \text{Fração}) \times 2^E$$

onde os bits da fração representam um número entre 0 e 1, e *E* especifica o valor no campo de expoente, que será explicado em detalhes mais adiante. Se numerarmos os bits da fração da esquerda para a direita de *s*1, *s*2, *s*3, ..., então o valor é

$$(-1)^s \times (1 + (s_1 \times 2^{-1}) + (s_2 \times 2^{-2}) + (s_3 \times 2^{-3}) + (s_4 \times 2^{-4}) + \dots) \times 2^E$$

A Figura 3.15 mostra as codificações dos números de ponto flutuante IEEE 754. Outros recursos do IEEE 754 são símbolos especiais para representar eventos incomuns. Por exemplo, em vez de interromper em uma divisão por 0, o software pode definir o resultado para um padrão de bits que represente $+\infty$ ou $-\infty$; o maior expoente é reservado para esses símbolos especiais. Quando o programador imprime os resultados, o programa imprimirá um símbolo de infinito. (Para os que são matematicamente treinados, a finalidade do infinito é formar o fechamento topológico dos reais.)

overflow (ponto flutuante) Uma situação em que um expoente positivo torna-se muito grande para caber no campo de expoente.

underflow (ponto flutuante) Uma situação em que um expoente negativo torna-se muito grande para caber no campo de expoente.

precisão dupla Um valor de ponto flutuante representado em duas words de 32 bits.

precisão simples Um valor de ponto flutuante representado em uma única word de 32 bits.

O IEEE 754 até mesmo possui um símbolo para o resultado de operações inválidas, como 0/0, ou a subtração infinito de infinito. Esse símbolo é *NaN*, de *Not a Number* (não é um número). A finalidade dos NaNs é permitir que os programadores adiem alguns testes e decisões para outro momento no programa, quando for conveniente.

Os projetistas do IEEE 754 também queriam uma representação de ponto flutuante que pudesse ser facilmente processada por comparações de inteiros, especialmente para ordenação. Esse desejo é o motivo pelo qual o sinal está no bit mais significativo, permitindo um teste rápido de menor que, maior que ou igual a 0. (Isso é um pouco mais complicado do que uma ordenação simples de inteiros, pois essa notação é basicamente sinal e magnitude, em vez do complemento a dois.)

Precisão simples		Precisão dupla		Objeto representado
Expoente	Fração	Expoente	Fração	
0	0	0	0	0
0	não zero	0	não zero	± número desnormalizado
1-254	qualquer coisa	1-2046	qualquer coisa	± número ponto flutuante
255	0	2047		± infinito
255	não zero	2047	não zero	NaN (Not a Number)

FIGURA 3.15 Codificação IEEE 754 para os números de ponto flutuante. Um bit de sinal separado determina o sinal. Os números desnormalizados são descritos na Seção “Detalhamento” da página 164.

Colocar o expoente antes do significando simplifica a ordenação dos números de ponto flutuante usando instruções de comparação de inteiros, pois os números com expoentes maiores são maiores do que os números com expoentes menores, desde que os dois expoentes tenham o mesmo sinal.

Expoentes negativos impõem um desafio à ordenação simplificada. Se usarmos o complemento a dois ou qualquer outra notação em que os expoentes negativos têm um 1 no bit mais significativo do campo de expoente, um expoente negativo se parecerá com um número grande. Por exemplo, $1,0_{\text{bin}} \times 2^{-1}$ seria representado 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	0	.	.	.

(Lembre-se de que o 1 à esquerda do ponto é implícito no significando.) O valor $1,0_{\text{bin}} \times 2^{+1}$ seria semelhante a um número binário 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	0	.	.	.

A notação desejável, portanto, precisa representar o expoente mais negativo como $00 \dots 00_{\text{bin}}$ e o mais positivo como $11 \dots 11_{\text{bin}}$. Essa convenção é chamada *notação deslocada*, com a inclinação (bias) sendo o número subtraído da representação normal, sem sinal, para determinar o valor real.

O IEEE 754 usa um bias de 127 para a precisão simples, de modo que -1 é representado pelo padrão de bits do valor $-1 + 127_{\text{dec}}$ ou $126_{\text{dec}} = 0111\ 1110_{\text{bin}}$, e +1 é representado por $1 + 127$ ou $128_{\text{dec}} = 1000\ 0000_{\text{bin}}$. O expoente deslocado significa que o valor representado por um número em ponto flutuante é, na realidade:

$$(-1)^S \times (1 + \text{Fração}) \times 2^{(\text{Expoente} - \text{Bias})}$$

O bias do expoente para a precisão dupla é 1023.

Assim, a notação do IEEE 754 pode ser processada pelas comparações de inteiros para acelerar a ordenação dos números de ponto flutuante. Vamos mostrar a representação

REPRESENTAÇÃO DE PONTO FLUTUANTE

Mostre a representação binária IEEE 754 para o número -0.75_{dec} em precisão simples e dupla.

O número $-0,75_{dec}$ também é

EXEMPLO

RESPOSTA

$-3/4_{dec}$ ou $-3/2^2_{dec}$

Ele também é representado pela fração binária

$$-11_{\pm 0}/2^2 \approx -0.11_{\pm}$$

Em notação científica, o valor é

$$-0.11_{\pm 0.01} \times 2^0$$

e, na notação científica normalizada, ele é

$$-1, 1_{\text{vis}} \times 2^{-1}$$

A representação geral para um número de precisão simples é

$$(-1)^S \times (1 + \text{Fracão}) \times 2^{(\text{Expoente} - 1)T}$$

Quando subtraímos o bias 127 do expoente de -1.1×2^{-1} , o resultado é

$$(-1)^l \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ldots) \times 2^{(126 - l)7}$$

A representação binária de precisão simples de 0.75, portanto, é

A representação em precisão dupla é

Agora, vamos experimentar na outra direção.

CONVERTENDO PONTO FLUTUANTE BINÁRIO PARA DECIMAL

EXEMPLO

Que número decimal é representado por este float de precisão simples?

RESPOSTA

O bit de sinal é 1, o campo de expoente contém 129, e o campo de fração contém $1 \times 2^{-2} = 1/4$, ou 0,25. Usando a equação básica,

$$\begin{aligned} (-1)^S \times (1 + \text{Fração}) \times 2^{(\text{Expoente-Bias})} &= (-1)^1 \times (1 + 0,25) \times 2^{(129-127)} \\ &= -1 \times 1,25 \times 2^2 \\ &= -1,25 \times 4 \\ &= -5,0 \end{aligned}$$

Nas próximas seções, daremos os algoritmos para a adição e multiplicação em ponto flutuante. Em seu núcleo, eles utilizam operações inteiras correspondentes nos significandos, mas é preciso que haja manutenção extra para lidar com os expoentes e normalizar o resultado. Primeiro, oferecemos uma derivação intuitiva dos algoritmos em decimal, e depois uma versão mais detalhada, binária, nas figuras.

Detalhamento: em uma tentativa de aumentar o intervalo sem remover bits do significando, alguns computadores antes do padrão IEEE 754 usavam uma base diferente de 2. Por exemplo, os computadores mainframe IBM 360 e 370 usam a base 16. Como mudar o expoente no IBM em um significa deslocar o significando em 4 bits, os números de base 16 "normalizados" podem ter até 3 bits à esquerda do ponto em 0s! Logo, os dígitos hexadecimais significam que até 3 bits precisam ser removidos do significando, o que leva a problemas surpreendentes na precisão da aritmética de ponto flutuante.

Adição em ponto flutuante

Vamos somar os números na notação científica manualmente, para ilustrar os problemas na adição em ponto flutuante: $9,999_{dec} \times 10^1 + 1,610_{dec} \times 10^{-1}$. Suponha que só possamos armazenar quatro dígitos decimais do significando e dois dígitos decimais do expoente.

Etapa 1. Para poder somar esses números corretamente, temos de alinhar o ponto decimal do número que possui o menor expoente. Logo, precisamos de uma forma do número menor, $1,610_{dec} \times 10^{-1}$, que combine com o expoente maior. Obtemos isso observando que existem várias representações de um número em ponto flutuante não normalizado na notação científica:

$$1.610_{\text{dec}} \times 10^{-1} = 0.1610_{\text{dec}} \times 10^0 = 0.01610_{\text{dec}} \times 10^1$$

O número da direita é a versão que desejamos, pois seu expoente combina com o expoente do maior número, $9,999_{\text{dec}} \times 10^1$. Assim, a primeira etapa desloca o significando do menor número à direita até que seu expoente corrigido combine com o do maior número. Contudo, só podemos representar quatro dígitos decimais, de modo que, após o deslocamento, o número é, na realidade:

$$0.016_{\pm 0.001} \times 10^3$$

Etapa 3 Em seguida, vem a adição dos significados:

$$+ \frac{9,999_{\text{dec}}}{0,016_{\text{dec}}} \\ = 624,9375$$

A soma é $10.015_{des} \times 10^1$.

Etapa 3. Essa soma não está na notação científica normalizada, de modo que precisamos ajustá-la:

$$10,015_{\text{dec}} \times 10^1 = 1,0015_{\text{dec}} \times 10^2$$

Assim, depois da adição, podemos ter de deslocar a soma para colocá-la na forma normalizada, ajustando o expoente de acordo. Esse exemplo mostra o deslocamento para a direita, mas, se um número fosse positivo e o outro negativo, é possível que a soma tenha muitos 0s iniciais, exigindo deslocamentos à esquerda. Sempre que o expoente é aumentado ou diminuído, temos de verificar o overflow ou underflow – ou seja, temos de verificar se o expoente ainda cabe em seu campo.

Etapa 4. Como pressupomos que o significando só pode ter quatro dígitos de extensão (excluindo o sinal), temos de arredondar o número. Em nosso algoritmo que aprendemos na escola, as regras truncam o número se o dígito à direita do ponto desejado estiver entre 0 e 4 e somamos 1 ao dígito se o número à direita estiver entre 5 e 9. O número

$$1,0015_{\text{dec}} \times 10^2$$

é arredondado para quatro dígitos no significando, passando para

$$1,002_{\text{dec}} \times 10^2$$

Pois o quarto dígito à direita do ponto decimal estava entre 5 e 9. Observe que, se não tivermos sorte no arredondamento, como ao somar 1 a uma seqüência de 9s, a soma não pode mais ser normalizada e precisaríamos realizar a etapa 3 novamente.

A Figura 3.16 mostra o algoritmo para a adição binária de ponto flutuante que acompanha este exemplo em decimal. As etapas 1 e 2 são semelhantes ao exemplo que discutimos: ajustar o significando do número com o menor expoente e depois somar os dois significandos. A etapa 3 normaliza os resultados, forçando uma verificação de overflow ou underflow. O teste de overflow e underflow na etapa 3 depende da precisão dos operandos. Lembre-se de que o padrão de todos os bits zero no expoente é reservado e usado para a representação de ponto flutuante de zero. Além disso, o padrão de todos os bits um no expoente é reservado para indicar valores e situações fora do escopo dos números de ponto flutuante normais (veja a Seção “Detalhamento” na página 164). Assim, para a precisão simples, o expoente máximo é 127, e o expoente mínimo é -126. Os limites para precisão dupla são 1023 e -1022.

ADIÇÃO DE PONTO FLUTUANTE EM DECIMAL

Tente somar os números $0,5_{\text{dec}}$ e $-0,4375_{\text{dec}}$ em binário usando o algoritmo da Figura 3.16.

EXEMPLO

Primeiro, vejamos a versão binária dos dois números na notação científica normalizada, supondo que mantemos 4 bits de precisão:

$$\begin{aligned} 0,5_{\text{dec}} &= 1/2_{\text{dec}} = 1/2^1_{\text{dec}} \\ &= 0,1_{\text{bin}} = 0,1_{\text{bin}} \times 2^0 = 1,000_{\text{bin}} \times 2^{-1} \\ -0,4375_{\text{dec}} &= -7/16_{\text{dec}} = -7/2^4_{\text{dec}} \\ &= -0,0111_{\text{bin}} = -0,0111_{\text{bin}} \times 2^0 = -1,110_{\text{bin}} \times 2^{-2} \end{aligned}$$

RESPOSTA

Agora, seguimos o algoritmo:



FIGURA 3.16 Adição de ponto flutuante. O caminho normal é executar as etapas 3 e 4 uma vez, mas, se o arredondamento fizer com que a soma fique não-normalizada, temos de repetir a etapa 3.

Etapa 1. O significando do número com o menor expoente ($-1,11_{\text{bin}} \times 2^{-2}$) é deslocado para a direita até seu expoente combinar com o maior número:

$$-1,110_{\text{bin}} \times 2^{-2} = -0,111_{\text{bin}} \times 2^{-1}$$

Etapa 2. Some os significandos:

$$1,000_{\text{bin}} \times 2^{-1} + (-0,111_{\text{bin}} \times 2^{-1}) = 0,001_{\text{bin}} \times 2^{-1}$$

Etapa 3. Normalize a soma, verificando overflow ou underflow:

$$\begin{aligned} 0,001_{\text{bin}} \times 2^{-1} &= 0,010_{\text{bin}} \times 2^{-2} = 0,100_{\text{bin}} \times 2^{-3} \\ &= 1,000_{\text{bin}} \times 2^{-4} \end{aligned}$$

Como $127 \geq 04 \geq -126$, não existe overflow ou underflow. (O expoente deslocado seria $-4 + 127$, ou 123, que está entre 1 e 254, o menor e o maior expoente deslocado não reservado.)

Etapa 4. Arredonde a soma:

$$1,000_{\text{bin}} \times 2^{-4}$$

A soma já cabe exatamente em 4 bits, de modo que não há mudança nos bits devido ao arredondamento.

Essa soma é, então

$$\begin{aligned} 1,000_{\text{bin}} \times 2^{-4} &= 0,0001000_{\text{bin}} = 0,0001_{\text{bin}} \\ &= 1/2^4_{\text{dec}} = 1/16_{\text{dec}} = 0.0625_{\text{dec}} \end{aligned}$$

Essa soma é o que esperaríamos da soma de $0,5_{\text{dec}}$ a $-0,4375_{\text{dec}}$.

Muitos computadores dedicam o hardware para executar operações de ponto flutuante o mais rápido possível. A Figura 3.17 esboça a organização básica do hardware para a adição de ponto flutuante.

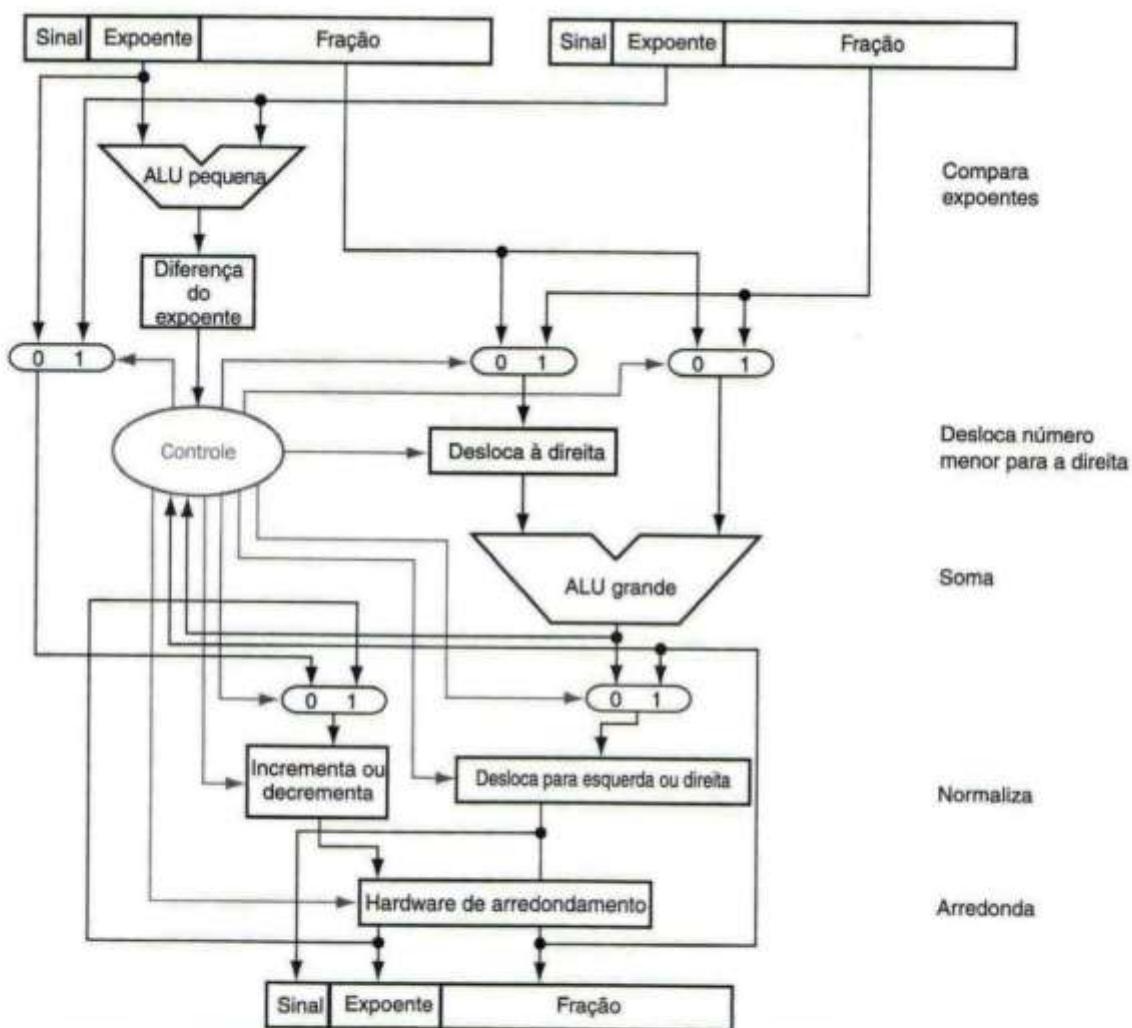


FIGURA 3.17 Diagrama de bloco de uma unidade aritmética dedicada à adição em ponto flutuante. As etapas da Figura 3.16 correspondem a cada bloco, de cima para baixo. Primeiro, o expoente de um operando é subtraído do outro usando a ALU pequena para determinar qual é maior e quanto. Essa diferença controla os três multiplexadores; da esquerda para a direita, eles selecionam o maior expoente, o significando do menor número, e o significando do maior número. O menor significando é deslocado para a direita, e depois os significandos são somados usando a ALU grande. A etapa de normalização, então, desloca a soma para a esquerda ou para a direita e incrementa ou decrementa o expoente. O arredondamento, então, cria o resultado final, que pode exigir normalização novamente para produzir o resultado final.

Multiplicação em ponto flutuante

Agora que já explicamos a adição em ponto flutuante, vamos experimentar a multiplicação em ponto flutuante. Começamos multiplicando os números decimais em notação científica na mão: $1,110_{\text{dec}} \times 10^{10} \times 9,200_{\text{dec}} \times 10^{-5}$. Suponha que possamos armazenar apenas quatro dígitos do significando e dois dígitos do expoente.

Etapa 1. Ao contrário da adição, calculamos o expoente do produto simplesmente somando os expoentes dos operandos:

$$\text{Novo expoente} = 10 + (-5) = 5$$

Vamos fazer isso com os expoentes deslocados, para obtermos o mesmo resultado: $10 + 127 = 137$, e $-5 + 127 = 122$, assim

$$\text{Novo expoente} = 137 + 122 = 259$$

Esse resultado é muito grande para o campo de expoente de 8 bits, de modo que há algo faltando! O problema é com o bias, pois estamos somando os biases e também os expoentes:

$$\text{Novo expoente} = (10 + 127) + (-5 + 127) = (5 + 2 \times 127) = 259$$

De acordo com isso, para obter a soma deslocada correta quando somamos números deslocados, temos de subtrair o bias da soma:

$$\text{Novo expoente} = 137 + 122 - 127 = 259 - 127 = 132 = (5 + 127)$$

e 5 é, na realidade, o expoente que calculamos inicialmente.

Etapa 2. Em seguida, vem a multiplicação dos significandos:

$$\begin{array}{r} 1,110_{\text{dec}} \\ \times 9,200_{\text{dec}} \\ \hline 0000 \\ 0000 \\ 2220 \\ 9990 \\ \hline 10212000_{\text{dec}} \end{array}$$

Existem três dígitos à direita do ponto decimal para cada operando, de modo que o ponto decimal é colocado seis dígitos a partir da direita no significando do produto:

$$10,212000_{\text{dec}}$$

Supondo que só possamos manter três dígitos à direita do ponto decimal, o produto é $10,212 \times 10^5$.

Etapa 3. Este produto não está normalizado, de modo que precisamos normalizá-lo:

$$10,212_{\text{dec}} \times 10^5 = 1,0212_{\text{dec}} \times 10^6$$

Assim, após a multiplicação, o produto pode ser deslocado para a direita um dígito, para colocá-lo no formato normalizado, somando 1 ao expoente. Nesse ponto, podemos verificar o overflow e o underflow. O underflow pode ocorrer se os dois operandos forem pequenos – ou seja, se ambos tiverem expoentes negativos grandes.

Etapa 4. Consideramos que o significando tem apenas quatro dígitos de largura (excluindo o sinal), de modo que devemos arredondar o número. O número

$$1,0212_{\text{dec}} \times 10^6$$

é arredondado para quatro dígitos no significando, para

$$1,021_{\text{dec}} \times 10^6$$

Etapa 5. O sinal do produto depende dos sinais dos operandos originais. Se forem iguais, o sinal é positivo; caso contrário, é negativo. Logo, o produto é

$$+1,021_{\text{dec}} \times 10^6$$

O sinal da soma no algoritmo de adição foi determinado pela adição dos significandos; porém, na multiplicação, o sinal do produto é determinado pelos sinais dos operandos.

Mais uma vez, como mostra a Figura 3.18, a multiplicação de números binários em ponto flutuante é muito semelhante às etapas que acabamos de concluir. Começamos calculando o novo expoente do produto, somando os expoentes deslocados, subtraindo um bias para obter o resultado apropriado. Em seguida, está a multiplicação de significandos, seguida por uma etapa de normalização opcional. O tamanho do expoente é verificado, em busca de overflow ou underflow, e depois o produto é arredondado. Se o arredondamento causar mais normalização, mais uma vez verificamos o tamanho do expoente. Finalmente, definimos o bit de sinal como 1 se os sinais dos operandos forem diferentes (produto negativo) ou como 0 se forem iguais (produto positivo).

MULTIPLICAÇÃO EM PONTO FLUTUANTE EM DECIMAL

Vamos tentar multiplicar os números $0,5_{\text{dec}}$ e $-0,4375_{\text{dec}}$, usando as etapas na Figura 3.18.

EXEMPLO

Em binário, a tarefa é multiplicar $1,000_{\text{bin}} \times 2^{-1}$ por $-1,110_{\text{bin}} \times 2^{-2}$.

RESPOSTA

Etapa 1. Somando os expoentes sem bias:

$$-1 + (-2) = -3$$

ou então, usando a representação deslocada:

$$\begin{aligned} (-1 + 127) + (-2 + 127) - 127 &= (-1 - 2) + (127 + 127 - 127) \\ &= -3 + 127 = 124 \end{aligned}$$

Etapa 2. Multiplicando os significandos:

$$\begin{array}{r} 1,000_{\text{bin}} \\ \times 1,110_{\text{bin}} \\ \hline 0000 \\ 1000 \\ 1000 \\ \hline 1110000_{\text{bin}} \end{array}$$

O produto é $1,110000_{\text{bin}} \times 2^{-3}$, mas precisamos mantê-lo com 4 bits, de modo que é $1,110_{\text{bin}} \times 2^{-3}$.

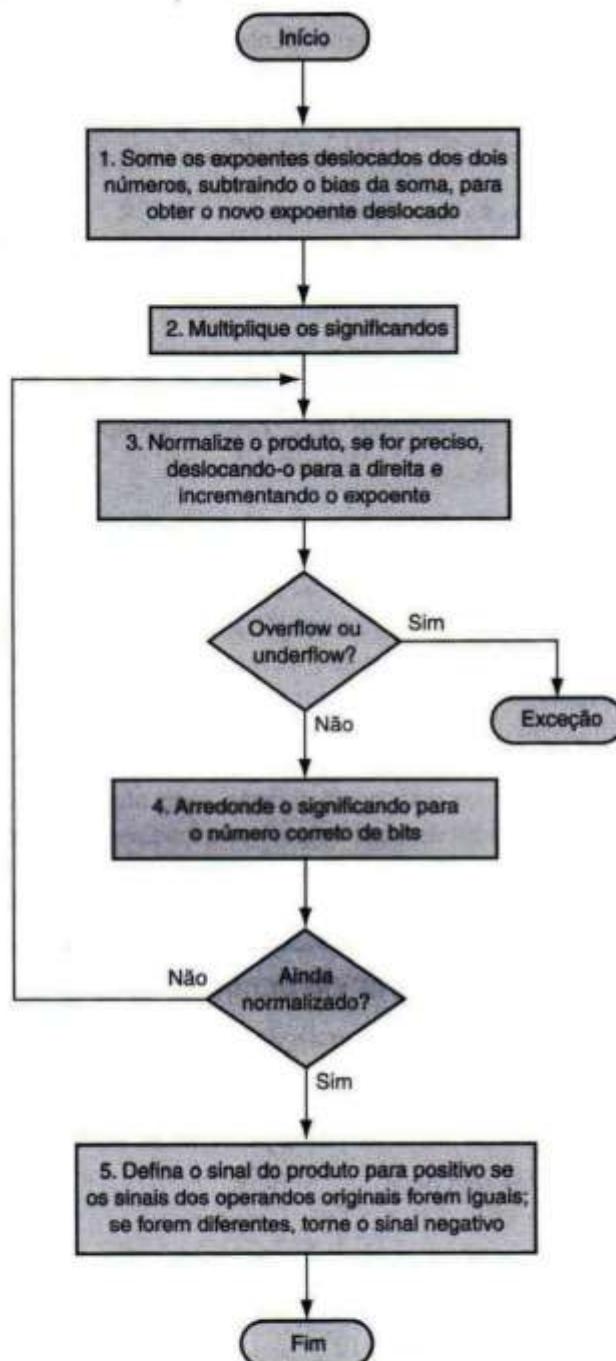


FIGURA 3.18 Multiplicação em ponto flutuante. O caminho normal é executar as etapas 3 e 4 uma vez, mas, se o arredondamento fizer com que a soma fique desnormalizada, temos de repetir a etapa 3.

Etapa 3. Agora, verificamos o produto para ter certeza de que está normalizado e depois verificamos o expoente em busca de overflow ou underflow. O produto já está normalizado e, como $127 \geq -3 \geq -126$, não existe overflow ou underflow. (Usando a representação deslocada, $254 \geq 124 \geq 1$, de modo que o expoente cabe.)

Etapa 4. O arredondamento do produto não causa mudança:

$$1,110_{\text{bin}} \times 2^{-3}$$

Etapa 5. Como os sinais dos operandos originais diferem, torne o sinal do produto negativo. Logo, o produto é

$$-1,110_{\text{bin}} \times 2^{-3}$$

Convertendo para decimal, para verificar nossos resultados:

$$\begin{aligned} -1,110_{\text{bin}} \times 2^{-3} &= -0,001110_{\text{bin}} = -0,00111_{\text{bin}} \\ &= -7/2^5_{\text{dec}} = -7/32_{\text{dec}} = -0,21875_{\text{dec}} \end{aligned}$$

O produto entre $0,5_{\text{dec}}$ e $-0,4375_{\text{dec}}$ é, na realidade, $-0,21875_{\text{dec}}$.

Instruções de ponto flutuante no MIPS

O MIPS admite os formatos de precisão simples e dupla do padrão IEEE 754 com estas instruções:

- Adição simples em ponto flutuante (add.s) e adição dupla (add.d)
- Subtração simples em ponto flutuante (sub.s) e subtração dupla (sub.d)
- Multiplicação simples em ponto flutuante (mul.s) e multiplicação dupla (mul.d)
- Divisão simples em ponto flutuante (div.s) e divisão dupla (div.d)
- Comparação simples em ponto flutuante (c.x.s) e comparação dupla (c.x.d), onde x pode ser igual (eq), diferente (neq), menor que (lt), menor ou igual (le), maior que (gt) ou maior ou igual (ge)
- Desvio, verdadeiro em ponto flutuante (be1t) e desvio falso (bc1f)

A comparação em ponto flutuante define um bit como verdadeiro ou falso, dependendo da condição de comparação, e um desvio de ponto flutuante então decide se desviará ou não, dependendo da condição.

Os projetistas do MIPS decidiram acrescentar registradores de ponto flutuante separados – chamados \$f0, \$f1, \$f2... – usados para precisão simples ou precisão dupla. Logo, eles incluiram loads e stores separados para registradores de ponto flutuante: lwc1 e swc1. Os registradores base para transferências de dados de ponto flutuante continuam sendo registradores inteiros. O código do MIPS para carregar dois números de precisão simples da memória, somá-los e depois armazenar a soma poderia se parecer com isto:

```

lwc1      $f4,x($sp)  # Lê número P.F. 32 bits em F4
lwc1      $f6,y($sp)  # Lê número P.F. 32 bits em F6
add.s    $f2,$f4,$f6  # F2 = F4 + F6 precisão simples
swc1      $f2,z($sp)  # Armazena número P.F. 32 bits de F2

```

Um registrador de precisão dupla é, na realidade, um par de registradores (par e ímpar) de precisão simples, usando o número do registrador par como seu nome.

A Figura 3.19 resume a parte de ponto flutuante da arquitetura MIPS revelada neste capítulo, com as adições para dar suporte ao ponto flutuante mostradas em cores. Semelhante à Figura 2.25, mostramos a codificação dessas instruções na Figura 3.20.

Operandos de ponto flutuante do MIPS

Nome	Exemplo	Comentários
32 registradores de ponto flutuante	\$f0, \$f1, \$f2, ..., \$f31	Registradores MIPS de ponto flutuante são usados em pares para números de precisão dupla.
2^{30} words em memória	Memória[0], Memória[4], ..., Memória[4294967292]	Acessado apenas por instruções de transferência de dados. O MIPS usa endereços em bytes, de modo que os endereços seqüenciais em words diferem em 4 vezes. A memória mantém estruturas de dados, como arrays, e spilled registers, como aqueles salvos em chamadas de procedimento.

Linguagem assembly de ponto flutuante do MIPS

Categoria	Instrução	Exemplo	Significado	Comentários
Aritmética	FP add single	add.s \$f2,\$f4,\$f6	$\$f2 = \$f4 + \$f6$	adição PF (precisão simples)
	FP subtract single	sub.s \$f2,\$f4,\$f6	$\$f2 = \$f4 - \$f6$	subtração PF (precisão simples)
	FP multiply single	mul.s \$f2,\$f4,\$f6	$\$f2 = \$f4 \times \$f6$	multiplicação PF (precisão simples)
	FP divide single	div.s \$f2,\$f4,\$f6	$\$f2 = \$f4 / \$f6$	divisão PF (precisão simples)
	FP add double	add.d \$f2,\$f4,\$f6	$\$f2 = \$f4 + \$f6$	adição PF (precisão dupla)
	FP subtract double	sub.d \$f2,\$f4,\$f6	$\$f2 = \$f4 - \$f6$	subtração PF (precisão dupla)
	FP multiply double	mul.d \$f2,\$f4,\$f6	$\$f2 = \$f4 \times \$f6$	multiplicação PF (precisão dupla)
	FP divide double	div.d \$f2,\$f4,\$f6	$\$f2 = \$f4 / \$f6$	divisão PF (precisão dupla)
Transferência de dados	load word copr. 1	lwcl \$f1,100(\$s2)	$\$f1 = \text{Memória}[\$s2 + 100]$	dados de 32 bits para um registrador FP
	store word copr. 1	swcl \$f1,100(\$s2)	$\text{Memória}[\$s2 + 100] = \$f1$	dados de 32 bits para memória
Desvio condicional	branch on FP true	belt 25	if (cond == 1) go to PC + 4 + 100	desvio relativo ao PC se PF cond.
	branch on FP false	bclf 25	if (cond == 0) go to PC + 4 + 100	desvio relativo ao PC se não cond.
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s \$f2,\$f4	if(\$f2 < \$f4) cond = 1; else cond = 0	comparação PF menor que, precisão simples
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d \$f2,\$f4	if(\$f2 < \$f4) cond = 1; else cond = 0	comparação PF menor que, precisão dupla

Linguagem de máquina de ponto flutuante do MIPS

Nome	Formato	Exemplo							Comentários
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
lwcl	I	49	20	2		100			lwcl \$f2,100(\$s4)
swcl	I	57	20	2		100			swcl \$f2,100(\$s4)
belt	I	17	8	1		25			Belt 25
bclf	I	17	8	0		25			Bclf 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
Tamanho do campo		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits		Todas as instruções MIPS de 32 bits

FIGURA 3.19 Arquitetura de ponto flutuante do MIPS revelada até aqui. Ver Apêndice A, Seção A.10, para obter mais detalhes.

op(31:26):								
8-26	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
31-29								
0(000)	<u>Rfmt</u>	<u>Bltz/qez</u>	j	jal	beq	bne	blez	bgtz
1(001)	addi	addiu	slti	sltiu	andi	ori	xori	lui
2(010)	<u>TLB</u>	<u>FIPt</u>						
3(011)								
4(100)	lb	lh	lw	lw	lbu	lhu	lw	
5(101)	sb	sh	sw	sw			sw	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26) = 010001 (FIPt), (rt(16:16)=0 => c=f, rt(16:16)=1 => c=t), rs(25:21):								
23-21	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
25-24								
0(00)	mfc1		cfc1		mtcl		ctcl	
1(01)	bc1.c							
2(10)	f-single	f-double						
3(11)								

op(31:26) = 010001 (FIPt), (f above: 10000 => f=s, 10001 => f=d), funct(5:0):								
2-0	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
5-3								
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.lo. f	c.ul. f	c.ole. f	c.uie. 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 3.20 Codificação de instruções de ponto flutuante do MIPS. Essa notação indica o valor de um campo por linha e por coluna. Por exemplo, na parte superior da figura, Tw se encontra na linha número 4 (100_{bin} para os bits de 31-29 da instrução) e na coluna número 3 (011_{bin} para os bits 28-26 da instrução), de modo que o valor correspondente do campo op (bits 31-26) é 100011_{bin}. O sublinhado indica que o campo é usado em outro lugar. Por exemplo, FIPt na linha 2 e coluna 1 (op = 010001_{bin}) está definido na parte inferior da figura. Logo, sub. f na linha 0 e coluna 1 da seção inferior significa que o campo funct (bits 5-0 da instrução) é 000001_{bin} e o campo op (bits 31-26) é 010001_{bin}. Observe que o campo rs de 5 bits, especificado na parte do meio da figura, determina se a operação é de precisão simples (f = s, de modo que rs = 10000) ou precisão dupla (f = d, de modo que rs = 10001). De modo semelhante, o bit 16 da instrução determina se a instrução bc1.c testa o estado verdadeiro (bit 16 = 1 => bc1.t) ou falso (bit 16 = 1 => bc1.f). As instruções em negrito são descritas nos Capítulos 2 ou 3, com o Apêndice A abordando todas as instruções.

Interface hardware/software

Uma questão que os projetistas de computador enfrentam no suporte à aritmética de ponto flutuante é se devem usar os mesmos registradores usados pelas instruções com inteiros ou acrescentar um conjunto especial de ponto flutuante. Como os programas normalmente realizam operações com inteiros e operações com ponto flutuante sobre dados diferentes, a separação dos registradores só aumentará ligeiramente o número de instruções necessárias para executar um programa. O maior impacto é criar um conjunto separado de instruções de transferência de dados para mover dados entre os registradores de ponto flutuante e a memória.

Os benefícios dos registradores de ponto flutuante separados são a existência do dobro dos registradores sem utilizar mais bits no formato da instrução, ter o dobro da largura de banda de registrador, com conjuntos de registradores separados para inteiros e números de ponto flutuante, e ser capaz de personalizar registradores para ponto flutuante; por exemplo, alguns computadores convertem todos os operandos dimensionados nos registradores para um único formato inteiro.

COMPILANDO UM PROGRAMA C DE PONTO FLUTUANTE EM CÓDIGO ASSEMBLY DO MIPS

EXEMPLO

Vamos converter uma temperatura em Fahrenheit para Celsius:

```
float f2c (float fahr)
{
    return ((5.0/9.0) * (fahr - 32.0));
}
```

Considere que o argumento de ponto flutuante fahr seja passado em \$f12 e o resultado deva ficar em \$f0. (Ao contrário dos registradores inteiros, o registrador de ponto flutuante 0 pode conter um número.) Qual é o código assembly do MIPS?

RESPOSTA

Consideramos que o compilador coloca as três constantes de ponto flutuante na memória para serem alcançadas facilmente por meio do ponteiro global \$gp. As duas primeiras instruções carregam as constantes 5.0 e 9.0 nos registradores de ponto flutuante:

```
f2c:
    lwcl $f16,const5($gp) # $f16 = 5.0 (5.0 na memória)
    lwcl $f18,const9($gp) # $f18 = 9.0 (9.0 na memória)
```

Depois, elas são divididas para que se obtenha a fração 5.0/9.0:

```
div.s $f16, $f16, $f18 # $f16 = 5.0 / 9.0
```

(Muitos compiladores dividiram 5.0 por 9.0 durante a compilação e guardariam uma única constante 5.0/9.0 na memória, evitando assim a divisão em tempo de execução.) Em seguida, carregamos a constante 32.0 e depois a subtraímos de fahr (\$f12):

```
lwcl $f18, const32($gp) # $f18 = 32.0
sub.s $f18, $f12, $f18 # $f18 = fahr - 32.0
```

Finalmente, multiplicamos os dois resultados intermediários, colocando o produto em \$f0 como resultado de retorno, e depois retornamos:

```
mul.s $f0, $f16, $f18 # $f0 = (5/9)*(fahr - 32.0)
jr $ra # return
```

Agora, vamos realizar operações de ponto flutuante em matrizes, código comumente encontrado em programas científicos.

COMPILANDO UM PROCEDIMENTO EM C DE PONTO FLUTUANTE COM MATRIZES BIDIMENSIONAIS NO MIPS

A maioria dos cálculos de ponto flutuante é realizada com precisão dupla. Vamos realizar uma multiplicação de matrizes $X = X + Y * Z$. Vamos supor que X , Y e Z sejam matrizes quadradas com 32 elementos em cada dimensão.

```
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];
}
```

Os endereços iniciais do array são parâmetros, de modo que estão em \$a0, \$a1 e \$a2. Suponha que as variáveis inteiras estejam em \$s0, \$s1 e \$s2, respectivamente. Qual é o código assembly do MIPS para o corpo do procedimento?

Observe que $x[i][j]$ é usado no loop mais interno. Como o índice do loop é k , o índice não afeta $x[i][j]$, de modo que podemos evitar a leitura e o armazenamento de $x[i][j]$ a cada iteração. Em vez disso, o compilador lê $x[i][j]$ em um registrador fora do loop, acumula a soma dos produtos de $y[i][k]$ e $z[k][j]$ nesse mesmo registrador, e depois armazena a soma em $x[i][j]$, ao terminar o loop mais interno.

Mantemos o código mais simples, usando as pseudo-instruções em assembly li (que carrega uma constante em um registrador), e l.d e s.d (que o montador transforma em um par de instruções de transferência de dados, lwc1 ou swc1, para um par de registradores de ponto flutuante).

O corpo do procedimento começa salvando o valor de término do loop (32) em um registrador temporário e depois inicializando as três variáveis do loop *for*:

```
mm:...
    li    $t1, 32 # $t1 = 32 (tamanho de linha/fim do loop)
    li    $s0, 0   # i=0: inicializa 1º loop for
L1:   li    $s1, 0   # j = 0: reinicia 2º loop for
L2:   li    $s2, 0   # k = 0; reinicia 3º loop for
```

Para calcular o endereço de $x[i][j]$, precisamos saber como um array bidimensional de 32×32 é armazenado na memória. Como você poderia esperar, seu layout é como se houvesse 32 arrays unidimensionais, cada um com 32 elementos. Assim, a primeira etapa é pular os i “arrays unidimensionais”, ou linhas, para obter a que desejamos. Assim, multiplicamos o índice da primeira dimensão pelo tamanho da linha, 32. Como 32 é uma potência de 2, podemos usar um deslocamento em seu lugar:

```
sll $t2, $s0, 5 # $t2 = i * 25 (tamanho de linha de x)
```

Agora, acrescentamos o segundo índice para selecionar o elemento j da linha desejada:

```
addu $t2, $t2, $s1 # $t2 = i * tamanho(linha) + j
```

Para transformar essa soma em um índice em bytes, multiplicamos pelo tamanho de um elemento da matriz em bytes. Como cada elemento tem 8 bytes para a precisão dupla, podemos deslocar à esquerda por 3:

```
sll $t2, $t2, 3 # $t2 = deslocamento em bytes de [i][j]
```

EXEMPLO

RESPOSTA

Em seguida, somamos essa soma ao endereço base de x , dando o endereço de $x[i][j]$, e depois carregamos o número de precisão dupla $x[i][j]$ em $\$f4$:

```
addu $t2, $a0, $t2 # $t2 = endereço em bytes de x[i][j]
l.d $f4, 0($t2) # $f4 = 8 bytes de x[i][j]
```

As cinco instruções a seguir são praticamente idênticas às cinco últimas: calcular o endereço e depois ler o número de precisão dupla $z[k][j]$.

```
L3: sll $t0, $s2, 5 # $t0 = k * 25 (tamanho de linha de z)
    addu $t0, $t0, $s1 # $t0 = k * tamanho(linha) + j
    sll $t0, $t0, 3 # $t0 = deslocamento em bytes de [k][j]
    addu $t0, $a2, $t0 # $t0 = endereço em bytes de z[k][j]
    l.d $f16, 0($t0) # $f16 = 8 bytes de z[k][j]
```

De modo semelhante, as cinco instruções a seguir são como as cinco últimas: calcular o endereço e depois carregar o número de precisão dupla $y[i][k]$.

```
sll      $t2, $s0, 5 # $t0 = i * 25 (tamanho de linha de y)
addu    $t0, $t0, $s2 # $t0 = i * tamanho(linha) + k
sll      $t0, $t0, 3 # $t0 = deslocamento em bytes de [i][k]
addu    $t0, $a1, $t0 # $t0 = endereço em bytes de y[i][k]
l.d      $f18, 0($t0) # $f18 = 8 bytes de y[i][k]
```

Agora que carregamos todos os dados, finalmente estamos prontos para realizar algumas operações em ponto flutuante! Multiplicamos os elementos de y e z localizados nos registradores $\$f18$ e $\$f16$, e depois acumulamos a soma em $\$f4$.

```
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]
```

O bloco final incrementa o índice k e retorna se o índice não for 32. Se for 32, ou seja, o final do loop mais interno, precisamos armazenar em $x[i][j]$ a soma acumulada em $\$f4$.

```
addiu $s2, $s2, 1 # k = k + 1
bne   $s2, $t1, L3 # se (k != 32) vai para L3
s.d   $f4, 0($t2) # x[i][j] = $f4
```

De modo semelhante, essas quatro instruções finais incrementam a variável de índice do loop do meio e do loop mais externo, voltando no loop se o índice não for 32 e saindo se o índice for 32.

```
addiu $s1, $s1, 1 # j = j + 1
bne   $s1, $t1, L2 # se (j != 32) vai para L2
addiu $s0, $s0, 1 # i = i + 1
bne   $s0, $t1, L1 # se (i != 32) vai para L1
```

Detalhamento: o layout do array discutido no exemplo, chamado *ordem linhas primeiro*, é usado pela linguagem C e muitas outras linguagens de programação. Fortran, por sua vez, usa a *ordem colunas primeiro*, pela qual o array é armazenado coluna por coluna.

Somente 16 dos 32 registradores de ponto flutuante do MIPS puderam ser usados originalmente para operações de precisão simples: $\$f0, \$f2, \$f4, \dots, \$f30$. A precisão dupla é calculada usando pares desses registradores. Os registradores de ponto flutuante com números ímpares só foram usados para carregar e armazenar a metade direita dos números de ponto flutuante de 64 bits. MIPS-32 acrescentou *l.d* e *s.d* ao conjunto de instruções. MIPS-32 também acrescentou versões "simples emparelhadas" de todas as instruções de ponto flutuante, onde uma única instrução resulta em duas operações paralelas de ponto flutuante sobre dois operandos de 32 bits dentro de registradores de 64 bits. Por exemplo, *add.ps F0, F2, F4* é equivalente a *add.s F0, F2, F4*, seguido por *add.s F1, F3, F5*.

Outro motivo para que os registradores inteiros e de ponto flutuante sejam separados é que os microprocessadores na década de 1980 não possuíam transistores suficientes para colocar a unidade ponto flutuante no mesmo chip da unidade de inteiros. Logo, a unidade de ponto flutuante, incluindo os registradores de ponto flutuante, opcionalmente estava disponível como um segundo chip. Esses chips aceleradores opcionais são chamados *co-processadores* e explicam o acrônimo para os loads de ponto flutuante no MIPS: lwc1 significa “load word to coprocessor 1” (“leia uma word para o co-processador 1”), que é a unidade de ponto flutuante. (O co-processador 0 trata da memória virtual, descrita no Capítulo 7.) Desde o início da década de 1990, os microprocessadores têm integrado o ponto flutuante (e praticamente tudo o mais) no chip, e, por isso, o termo “co-processador” reúne “acumulador” e “memória”.

Detalhamento: embora existam muitas maneiras de usar hardware na multiplicação de ponto flutuante para torná-la mais rápida, a divisão de ponto flutuante é muito mais desafiadora para se tornar rápida e precisa. Divisões lentas nos primeiros computadores levaram à remoção de divisões de muitos algoritmos, mas os computadores paralelos inspiraram a redescoberta de algoritmos com uso intenso da divisão, que funcionam melhor nesses computadores. Logo, podemos precisar de divisões mais rápidas.

Uma técnica para aproveitar um multiplicador rápido é a iteração de Newton, na qual a divisão é redefinida como a localização do zero de uma função para encontrar a recíproca $1/x$, que é multiplicada pelo outro operando. As técnicas de iteração não podem ser arredondadas corretamente sem o cálculo de muitos bits extras. Um chip TI soluciona esse problema, calculando uma recíproca de precisão extra.

Detalhamento: Java abraça o padrão IEEE 754 por nome em sua definição dos tipos de dados e operações de ponto flutuante Java. Assim, o código no primeiro exemplo poderia muito bem ter sido gerado para um método de classe que convertesse graus Fahrenheit para Celsius.

O segundo exemplo utiliza múltiplos arrays dimensionais, que não são admitidos explicitamente em Java. Java permite arrays de arrays, mas cada array pode ter seu próprio tamanho, ao contrário de vários arrays dimensionais em C. Como os exemplos no Capítulo 2, uma versão Java desse segundo exemplo exigiria muito código de verificação para os limites de array, incluindo um novo cálculo de tamanho no final da linha. Ela também precisaria verificar se a referência ao objeto não é nula.

Aritmética de precisão

Ao contrário dos inteiros, que podem representar exatamente cada número entre o menor e o maior, os números de ponto flutuante normalmente são aproximações para um número que não representam realmente. O motivo é que existe uma variedade infinita de números reais entre, digamos, 0 e 1, mas não mais do que 2^{53} podem ser representados com exatidão em ponto flutuante de precisão dupla. O melhor que podemos fazer é utilizar a representação de ponto flutuante próxima ao número real. Assim, o padrão IEEE 754 oferece vários modos de arredondamento para permitir que o programador selecione a aproximação desejada.

O arredondamento parece muito simples, mas arredondar com precisão exige que o hardware inclua bits extras no cálculo. Nos exemplos anteriores, fomos vagos com relação ao número de bits que uma representação intermediária pode ocupar, mas, claramente, se cada resultado intermediário tivesse de ser truncado ao número de dígitos exato, não haveria oportunidade para arredondar. O IEEE 754, portanto, sempre mantém 2 bits extras à direita durante adições intermediárias, chamados **guarda** e **arredondamento**, respectivamente. Vamos fazer um exemplo decimal para ilustrar o valor desses dígitos extras.

guarda O primeiro dos dois bits extras mantidos à direita durante os cálculos intermediários de números de ponto flutuante, usados para melhorar a precisão do arredondamento.

arredondamento Método para fazer com que o resultado de ponto flutuante intermediário se encaixe no formato de ponto flutuante; o objetivo normalmente é encontrar o número mais próximo que pode ser representado no formato.

ARREDONDANDO COM DÍGITOS DE GUARDA

EXEMPLO

Some $2,56_{\text{dec}} \times 10^0$ a $2,34_{\text{dec}} \times 10^2$, supondo que temos três dígitos decimais significativos. Arredonde para o número decimal mais próximo com três dígitos decimais significativos, primeiro com dígitos guarda e arredondamento, e depois sem eles.

RESPOSTA

Primeiro, temos de deslocar o número menor para a direita, a fim de alinhar os expoentes, de modo que $2,56_{\text{dec}} \times 10^0$ torna-se $0,0256_{\text{dec}} \times 10^2$. Como temos dígitos de guarda e arredondamento, somos capazes de representar os dois dígitos menos significativos quando alinharmos os expoentes. O dígito de guarda mantém 5 e o dígito de arredondamento mantém 6. A soma é

$$\begin{array}{r} 2,3400_{\text{dec}} \\ + 0,0256_{\text{dec}} \\ \hline 2,3656_{\text{dec}} \end{array}$$

Assim, a soma é $2,3656_{\text{dec}} \times 10^2$. Como temos dois dígitos para arredondar, queremos que os valores de 0 a 49 arredondem para baixo e de 51 a 99 para cima, com 50 sendo o desempate. Arredondar a soma para cima com três dígitos significativos gera $2,37_{\text{dec}} \times 10^2$.

Fazer isso *sem* dígitos de guarda e arredondamento remove dois dígitos do cálculo. A nova soma é, então,

$$\begin{array}{r} 2,34_{\text{dec}} \\ + 0,02_{\text{dec}} \\ \hline 2,36_{\text{dec}} \end{array}$$

A resposta é $2,36_{\text{dec}} \times 10^2$, arredondando no último dígito da soma anterior.

unidades na última casa (ulp)

O número de bits com erro nos bits menos significativos do significando entre o número real e o número que pode ser representado.

Como o pior caso para o arredondamento seria quando o número real está a meio caminho entre duas representações de ponto flutuante, a precisão no ponto flutuante normalmente é medida em termos do número de bits em erro nos bits mais significativos do significando; a medida é denominada número de **unidades na última casa, ou ulp (units in the last place)**. Se o número ficou defasado em 2 nos bits menos significativos, ele estaria defasado por 2 ulps. Desde que não haja qualquer overflow, underflow ou exceções de operação inválida, o IEEE 754 garante que o computador utiliza o número que está dentro de meia ulp.

Detalhamento: embora o exemplo anterior, na realidade, precisasse apenas de um dígito extra, a multiplicação pode precisar de dois. Um produto binário pode ter um bit 0 inicial; logo, a etapa de normalização precisa deslocar o produto 1 bit à esquerda. Isso desloca o dígito de guarda para o bit menos significativo do produto, deixando o bit de arredondamento para ajudar no arredondamento mais preciso do produto.

Existem quatro modos de arredondamento: sempre arredondar para cima (para $+\infty$), sempre arredondar para baixo (para $-\infty$), truncar e arredondar para o próximo par. O modo final determina o que fazer se o número estiver exatamente no meio. A Receita Federal americana sempre arredonda 0,50 dólares para cima, possivelmente para o benefício da Receita. Um modo mais imparcial seria arredondar para cima, nesse caso, na metade do tempo e arredondar para baixo na outra metade. O IEEE 754 diz que, se o bit menos significativo retido em um caso de meio do caminho fosse ímpar, some um; se fosse par, trunque. Esse método sempre cria um 0 no bit menos significativo no caso de desempate, dando nome ao arredondamento. Esse modo é o mais utilizado, e o único que o Java admite.

O objetivo dos bits de arredondamento extras é permitir que o computador obtenha os mesmos resultados, como se os resultados intermediários fossem calculados para precisão infinita e depois arredondados.

Para auxiliar nesse objetivo e arredondar para o par mais próximo, o padrão possui um terceiro bit além do bit de guarda e arredondamento; ele é definido sempre que existem bits diferentes de zero à direita do bit de arredondamento. Esse **sticky bit** permite que o computador veja a diferença entre $0,50 \dots 00_{dec}$ e $0,50 \dots 01_{dec}$ ao arredondar.

O sticky bit pode ser definido, por exemplo, durante a adição, quando o menor número é deslocado para a direita. Suponha que somemos $5,01_{dec} \times 10^{-1}$ a $2,34_{ten} \times 10^2$ no exemplo anterior. Mesmo com os bits de guarda e arredondamento, estariamos somando 0,0050 a 2,34, com uma soma de 2,3450. O sticky bit seria definido, porque existem bits diferentes de zero à direita. Sem o sticky bit para lembrar se quaisquer 1s foram deslocados, consideraríamos que o número é igual a 2.345000...00 e arredondaríamos para o par mais próximo de 2,34. Com o sticky bit para lembrar que o número é maior do que 2,345000...00, arredondaríamos para 2,35.

sticky bit Um bit usado no arredondamento além dos bits de guarda e arredondamento, definido sempre que existem bits diferentes de zero à direita do bit de arredondamento.

Resumo

A próxima Seção “Colocando em perspectiva” reforça o conceito de programa armazenado do Capítulo 2; o significado da informação não pode ser determinado simplesmente examinando-se os bits, pois os mesmos bits podem representar uma série de objetos. Esta seção mostra que a aritmética computacional é finita e, assim, pode não combinar com a aritmética natural. Por exemplo, a representação de ponto flutuante do padrão IEEE 754

$$(-1)^S \times (1 + \text{Fração}) \times 2^{(\text{Expoente} - \text{Bias})}$$

é quase sempre uma aproximação do número real. Os sistemas computacionais precisam ter o cuidado de minimizar essa lacuna entre a aritmética computacional e a aritmética no mundo real, e os programadores às vezes precisam estar cientes das implicações dessa aproximação.

Tipo C	Tipo Java	Transferências de dados	Operações
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, tut	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	lwcl, swcl	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

Padrões de bits não possuem significado inherente. Eles podem representar inteiros com sinal, inteiros sem sinal, números de ponto flutuante, instruções e assim por diante. O que é representado depende da instrução que opera sobre os bits na word.

A principal diferença entre os números no computador e os números no mundo real é que os números no computador possuem tamanho limitado e, por isso, uma precisão limitada; é possível calcular um número muito grande ou muito pequeno para ser representado em uma word. Os programadores precisam se lembrar desses limites e escrever programas de acordo.

Colocando em perspectiva

Interface hardware/software

No capítulo anterior, apresentamos as classes de armazenamento da linguagem de programação C. A tabela anterior mostra alguns dos tipos de dados C e Java junto com as instruções de transferência de dados MIPS e instruções que operam sobre aqueles tipos que aparecem nos Capítulos 2 e 3. Observe que Java omite inteiros sem sinal.

Verifique você mesmo Suponha que houvesse um formato de ponto flutuante IEEE 754 de 16 bits com 5 bits de expoente. Qual seria o intervalo provável de números que ele poderia 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 $\pm 1,1111\ 1111\ 1 \times 2^{15}, \pm 0, \pm \infty, \text{NaN}$
3. $\pm 1,0000\ 0000\ 00 \times 2^{-14}$ a $\pm 1,1111\ 1111\ 11 \times 2^{15}, \pm 0, \pm \infty, \text{NaN}$
4. $\pm 1,0000\ 0000\ 00 \times 2^{-15}$ a $\pm 1,1111\ 1111\ 11 \times 2^{14}, \pm 0, \pm \infty, \text{NaN}$

Detalhamento: para acomodar comparações que possam incluir NaNs, o padrão inclui *ordenada* e *desordenada* como opções para comparações. Logo, o conjunto de instruções MIPS inteiro possui muitos tipos de comparações para dar suporte a NaNs. (Java não admite comparações não-ordenadas.)

Em uma tentativa de espremer cada bit de precisão de uma operação de ponto flutuante, o padrão permite que alguns números sejam representados em forma não-normalizada. Em vez de ter uma lacuna entre 0 e o menor número normalizado, o IEEE permite *números não-normalizados* (também conhecidos como *denorms* ou *subnormals*). Eles têm o mesmo expoente que zero, mas um significando diferente de zero. Eles permitem que um número degrade no significado até se tornar 0, chamado *underflow gradual*. Por exemplo, o menor número normalizado positivo de precisão simples é

$$1,0000\ 0000\ 0000\ 0000\ 000_{\text{bin}} \times 2^{-126}$$

mas o menor número não-normalizado de precisão simples é

$$0,0000\ 0000\ 0000\ 0000\ 001_{\text{bin}} \times 2^{-126}, \text{ ou } 1,0_{\text{bin}} \times 2^{-149}$$

Para a precisão dupla, a lacuna denorm vai de $1,0 \times 2^{-1022}$ a $1,0 \times 2^{-1074}$.

A possibilidade de um operando ocasional não-normalizado tem dado dores de cabeça aos projetistas de ponto flutuante que estejam tentando criar unidades de ponto flutuante velozes. Logo, muitos computadores causam uma exceção se um operando for não-normalizado, permitindo que o software complete a operação. Embora as implementações de software sejam perfeitamente válidas, seu menor desempenho diminuiu a popularidade dos denorms no software de ponto flutuante portável. Além disso, se os programadores não esperarem os denorms, seus programas poderão ser surpreendidos.

3.7.

Vida real: ponto flutuante no IA-32

A arquitetura IA-32 possui instruções regulares de multiplicação e divisão, que operam inteiramente sobre os registradores, em vez de contar com Hi e Lo como no MIPS. (Na verdade, as versões posteriores do conjunto de instruções MIPS incluíram instruções semelhantes.)

As diferenças principais são encontradas nas instruções de ponto flutuante. A arquitetura de ponto flutuante IA-32 é diferente de todos os outros computadores no mundo.

A arquitetura de ponto flutuante do IA-32

O co-processador de ponto flutuante Intel 8087 foi anunciado em 1980. Essa arquitetura estendeu o 8086 com cerca de 60 instruções de ponto flutuante.

A Intel provedeu uma arquitetura de pilha com suas instruções de ponto flutuante: loads inserem números na pilha, operações encontram operandos nos dois elementos do topo da pilha e stores podem retirar elementos da pilha. A Intel complementou essa arquitetura de pilha com instruções e modos de endereçamento que permitem que a arquitetura tenha alguns dos benefícios do modelo registrador-memória. Além de localizar operandos nos dois elementos do topo da pilha, um operando pode estar na memória ou em um dos sete registradores do chip, abaixo do topo da pilha. Assim, um conjunto completo de instruções de pilha é complementado por um conjunto limitado de instruções registrador-memória.

Essa mistura é ainda um modelo registrador-memória restrito, pois os loads sempre movem dados para o topo da pilha enquanto incrementam o ponteiro do topo da pilha, e os stores só podem mover do topo da pilha para a memória. A Intel usa a notação ST para indicar o topo da pilha, e ST(i) para representar o i-ésimo registrador abaixo do topo da pilha.

Outro novo recurso dessa arquitetura é que os operandos são mais largos na pilha de registradores do que são armazenados na memória, e todas as operações são realizadas nessa precisão interna larga. Ao contrário do máximo de 64 bits no MIPS, os operandos de ponto flutuante IA-32 na pilha possuem 80 bits de largura. Os números são convertidos automaticamente para o formato interno de 80 bits em um load e convertidos de volta para o tamanho apropriado em um store. Essa *precisão dupla estendida* não é aceita pelas linguagens de programação, embora tenha sido útil aos programadores de software matemático.

Os dados da memória podem ser números de ponto flutuante de 32 bits (precisão simples) ou de 64 bits (precisão dupla). Antes de realizar a operação, a versão registrador-memória dessas instruções converterá o operando da memória para esse formato de 80 bits da Intel. As instruções de transferência de dados também converterão automaticamente inteiros de 16 e 32 bits para ponto flutuante, e vice-versa, para loads e stores de inteiros.

As operações de ponto flutuante IA-32 podem ser divididas em quatro classes principais:

1. Instruções de movimentação de dados, incluindo load, load constant e store
2. Instruções aritméticas, incluindo add, subtract, multiply, divide, square root e absolute value
3. Comparação, incluindo instruções para enviar o resultado ao processador de inteiros de modo que possa se desviar
4. Instruções transcendentais, incluindo sine, cosine, log e exponentiation

A Figura 3.21 mostra algumas das 60 operações de ponto flutuante. Observe que obtemos ainda mais combinações quando incluímos os modos de operando para essas operações. A Figura 3.22 mostra as muitas opções para a adição de ponto flutuante.

As instruções de ponto flutuante são codificadas por meio do opcode ESC do 8086 e o especificador de endereço pós-byte (ver Figura 2.46). As operações de memória reservam 2 bits para decidir se o operando é um ponto flutuante de 32 ou de 64 bits, ou um inteiro de 16 ou 32 bits. Esses mesmos 2 bits são usados em versões que não acessam a memória para decidir se o topo da pilha deve ser removido após a operação e se o topo da pilha ou um registrador inferior deve obter o resultado.

O desempenho de ponto flutuante da família IA-32 tradicionalmente tem ficado atrás de outros computadores. Se isso é simplesmente por uma falta de atenção dos engenheiros da Intel ou uma falha na sua arquitetura, é difícil saber. Podemos dizer que muitas arquiteturas novas têm sido anunciamas desde 1980, e nenhuma tem seguido as pegadas da Intel. Além disso, a Intel criou uma arquitetura de ponto flutuante mais tradicional como parte do SSE2.

Transferência de dados	Aritmética	Comparação	Transcendentais
F{I}LD mem/ST{i}	F{I}ADD{P} mem/ST{i}	F{I}COM{P}	FPTAN
F{I}ST{P} mem/ST{i}	F{I}SUB{R}{P} mem/ST{i}	F{I}UCOM{P}{P}	F2XM1
FIDPI	F{I}MUL{P} mem/ST{i}	FSTSW AX/mem	FCOS
FLDIF	{I}DIV{R}{P} mem/ST{i}		FPTAN
FLDZ	FSQRT		FPREM
	FABS		FSIN
	FRNDINT		FYL2X

FIGURA 3.21 As instruções de ponto flutuante da IA-32. Usamos as chaves {} para mostrar variações opcionais das operações básicas: {I} significa que existe uma versão inteira da instrução, {P} significa que essa variação retirará um operando da pilha após a operação, e {R} significa o reverso da ordem dos operandos nessa operação. A primeira coluna mostra as instruções de transferência de dados, que movem dados para a memória ou para um dos registradores abaixo do topo da pilha. As três últimas operações na primeira coluna colocam constantes na pilha: pi, 1,0 e 0,0. A segunda coluna contém as operações aritméticas descritas anteriormente. Observe que as três últimas operam apenas no topo da pilha. A terceira coluna contém as instruções de comparação. Como não existem instruções de desvio de ponto flutuante especiais, o resultado da comparação precisa ser transformado para a CPU de inteiros via instruções FSTSW, seja para o registrador AX ou para a memória, seguida por uma instrução SAHF para definir os códigos de condição. A comparação de ponto flutuante pode, então, ser testada por meio de instruções de desvio inteiras. A última coluna oferece as operações de ponto flutuante de mais alto nível. Nem todas as combinações sugeridas pela notação são fornecidas. Logo, operações F{I}SUB{R}{P} representam estas instruções encontradas no IA-32: FSUB, FISUB, FSUBR, FISUBR, FSUBP, FSUBRP. Para as instruções de subtração de inteiros, não existe um pop (FISUBP) ou um pop reverso (FISUBRP).

Instrução	Operandos	Comentário
FADD		Ambos os operandos na pilha; resultado substitui o topo da pilha.
FADD	ST{i}	Um operando de origem é o topo da pilha; resultado substitui registrador na posição i abaixo do topo da pilha.
FADD	ST{i}, ST	Um operando de origem é o registrador na posição i abaixo do topo da pilha; resultado substitui o topo da pilha.
FADD	mem32	Um operando de origem é um local de 32 bits na memória; resultado substitui o topo da pilha.
FADD	mem64	Um operando de origem é um local de 64 bits na memória; resultado substitui o topo da pilha.

FIGURA 3.22 As variações dos operandos para adição de ponto flutuante na arquitetura IA-32.

A arquitetura de ponto flutuante Streaming SIMD Extension 2 (SSE2) da Intel

O Capítulo 2 observa que, em 2001, a Intel acrescentou 144 instruções à sua arquitetura, incluindo registradores e operações de ponto flutuante com precisão dupla. Isso inclui oito registradores que podem ser usados como operandos de ponto flutuante, dando ao compilador um alvo diferente para as operações de ponto flutuante do que a arquitetura de pilha exclusiva. Os compiladores podem decidir usar os oito registradores SSE2 como registradores de ponto flutuante, como aqueles encontrados em outros computadores. A AMD expandiu o número para 16, como parte do AMD64, que a Intel passou a chamar de EM64T para seu uso.

Além de manter um número de precisão simples ou de precisão dupla em um registrador, a Intel permite que vários operandos de ponto flutuante sejam encaixados em um único registrador SSE2 de 128 bits: quatro de precisão simples e dois de precisão dupla. Se os operandos podem ser organizados na memória como dados alinhados em 128 bits, então as transferências de dados de 128 bits podem carregar e armazenar vários operandos por instrução. Esse formato de ponto flutuante compactado é aceito por operações aritméticas que podem operar simultaneamente sobre quatro números de precisão simples ou dupla. Essa nova arquitetura pode mais do que dobrar o desempenho em relação à arquitetura de pilha.

3.8**Falácia e armadilhas**

As falácia e armadilhas aritméticas geralmente advêm da diferença entre a precisão limitada da aritmética computacional e da precisão ilimitada da aritmética natural.

Falácia: a adição de ponto flutuante é associativa; ou seja, $x + (y + z) = (x + y) + z$.

Dado o grande intervalo de números que podem ser representados em ponto flutuante, ocorrem problemas quando se somam dois números grandes de sinais opostos mais um número pequeno. Por exemplo, suponha que $x = -1,5_{dec} \times 10^{38}$, $y = 1,5_{dec} \times 10^{38}$ e $z = 1,0$, e que esses são números de precisão simples. Então

$$\begin{aligned} x + (y + z) &= -1,5_{dec} \times 10^{38} + (1,5_{dec} \times 10^{38} + 1,0) \\ &= -1,5_{dec} \times 10^{38} + (1,5_{dec} \times 10^{38}) = 0,0 \\ (x + y) + z &= (-1,5_{dec} \times 10^{38} + 1,5_{dec} \times 10^{38}) + 1,0 \\ &= (0,0_{dec}) + 1,0 \\ &= 1,0 \end{aligned}$$

Portanto, $x + (y + z) \neq (x + y) + z$.

Como os números de ponto flutuante possuem precisão limitada e resultam em aproximações de resultados reais, $1,5_{dec} \times 10^{38}$ é tão maior que $1,0_{dec}$ que $1,5_{dec} \times 10^{38} + 1,0$ ainda é $1,5_{dec} \times 10^{38}$. É por isso que a soma de x , y e z é 0,0 ou 1,0, dependendo da ordem das adições de ponto flutuante, e, por isso, a soma em ponto flutuante *não* é associativa.

Falácia: assim como a instrução de deslocamento à esquerda pode substituir uma multiplicação de inteiros por uma potência de 2, um deslocamento à direita é o mesmo que uma divisão de inteiros por uma potência de 2.

Lembre-se de que um número binário x , onde x_i significa o bit na posição i , representa o número

$$\dots + (x_3 \times 2^3) + (x_2 \times 2^2) + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Deslocar os bits de \times para a direita de n bits pareceria ser o mesmo que dividir por 2^n . E isso é verdade para inteiros sem sinal. O problema é com os inteiros com sinal. Por exemplo, suponha que queremos dividir -5_{dec} por 4_{dec} ; o quociente seria -1_{dec} . A representação no complemento a dois para -5_{dec} é

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_{bin}$$

De acordo com essa falácia, deslocar para a direita por dois deverá dividir por 4_{dec} (2^2):

$$0011\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{bin}$$

Com um 0 no bit de sinal, esse resultado claramente está errado. O valor criado pelo deslocamento à direita é, na realidade, $1.073.741.822_{dec}$, e não -1_{dec} .

Uma solução seria ter um deslocamento aritmético à direita, que estende o bit de sinal, em vez de colocar 0s à esquerda num deslocamento à direita. Um deslocamento aritmético de 2 bits para a direita de -5_{dec} produz

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{bin}$$

O resultado é -2_{dec} , em vez de -1_{dec} ; próximo, mas não podemos comemorar.

Assim, a matemática pode ser definida como o assunto em que nunca sabemos do que estamos falando, nem se o que estamos dizendo é verdadeiro.
Bertrand Russell,
Recent Words on the Principles of Mathematics, 1901

Entretanto, o PowerPC possui uma instrução de deslocamento rápido (*shift right algebraic*) que, em conjunto com um add especial (add with carry), oferece a mesma resposta que dividir por uma potência de 2.

Armadilha: a instrução MIPS add immediate unsigned addiu estende o sinal de seu campo imediato de 16 bits.

Apesar de seu nome, add immediate unsigned (addiu) é usada para somar constantes a inteiros com sinal quando não nos importamos com o overflow. O MIPS não possui uma instrução de subtração imediata e os números negativos precisam de extensão de sinal, de modo que os arquitetos do MIPS decidiram estender o sinal do campo imediato.

Falácia: somente os matemáticos teóricos se importam com a precisão do ponto flutuante.

As manchetes dos jornais de novembro de 1994 provam que essa afirmação é uma falácia (ver Figura 3.23). A seguir, está a história por trás das manchetes.

O Pentium usa um algoritmo de divisão de ponto flutuante padrão, que gera bits de quociente múltiplos por etapa, usando os bits mais significativos do divisor e do dividendo para descobrir os 2 bits seguintes do quociente. A escolha vem de uma tabela de pesquisa contendo -2, -1, 0, +1 ou +2. A escolha é multiplicada pelo divisor e subtraída do resto para gerar um novo resto. Assim como a divisão sem restauração (ver o Exercício 3.25), se uma escolha anterior obtiver um resto muito grande, o resto parcial é ajustado em uma passada subsequente.

Evidentemente, havia cinco elementos da tabela do 80486 que a Intel pensou que nunca poderiam ser acessados, e eles otimizaram a PLA para retornar 0 no lugar de 2 nessas situações no Pentium. A Intel estava errada: embora os 11 primeiros bits sempre fossem corretos, erros apareceriam ocasionalmente nos bits de 12 a 52, ou do 4º ao 15º dígito decimal.



FIGURA 3.23 Uma amostra dos artigos de jornais e revistas de novembro de 1994, incluindo o *New York Times*, *San Jose Mercury News*, *San Francisco Chronicle* e *Infoworld*. O bug da divisão de ponto flutuante do Pentium chegou até mesmo à "Lista dos 10 maiores" do *David Letterman Late Show* na televisão. A Intel acabou tendo um custo de US\$300 milhões para substituir os chips com defeito.

A seguir está um roteiro dos fatos que aconteceram referentes ao bug do Pentium:

- *Julho de 1994:* a Intel descobre o bug no Pentium. O custo real para consertar o bug foi de várias centenas de milhares de dólares. Após os procedimentos normais de reparo do bug, levaram meses para fazer a mudança, verificar novamente e colocar o chip corrigido em produção. A Intel planejou colocar os chips bons em produção em janeiro de 1995, estimando que 3 a 5 milhões de Pentiums seriam produzidos com o bug.
- *Setembro de 1994:* um professor de matemática no Lynchburg College, na Virgínia, Thomas Nicely, descobre o bug. Depois de ligar para o suporte técnico da Intel e não receber uma posição oficial, ele posta sua descoberta na Internet. Rapidamente surgiram os seguidores, e alguns apontaram que até mesmo erros pequenos se tornam grandes ao multiplicar por grandes números: a fração de pessoas com uma doença rara vezes a população da Europa, por exemplo, poderia levar a uma estimativa errada do número de pessoas doentes.
- *7 de novembro de 1994:* o *Electronic Engineering Times* coloca a matéria em sua capa, o que logo foi seguido por outros jornais.
- *22 de novembro de 1994:* a Intel emite um comunicado oficial, chamando-o de "glitch". O Pentium "pode cometer erros no nono dígito. ... Até mesmo a maioria dos engenheiros e analistas financeiros exigem precisão apenas até a quarta ou quinta casa decimal. Usuários de planilhas eletrônicas e processadores de textos não precisam se preocupar. ... Talvez haja algumas dezenas de pessoas a quem isso afetaria. Até aqui, só ouvimos falar de uma. ... [Somente] matemáticos teóricos (com computadores Pentium comprados antes do verão) devem se preocupar". O que aborreceu a muitos foi que os clientes foram solicitados a descrever sua aplicação à Intel, e depois a Intel decidiria se sua aplicação mereceria ou não um novo Pentium sem um bug de divisão.
- *5 de dezembro de 1994:* a Intel afirma que a falha acontece uma vez em 27.000 anos para o usuário típico de planilha. A Intel considera que um usuário realiza 1.000 divisões por dia e multiplica a taxa de erro supondo que os números de ponto flutuante são aleatórios, o que é um em 9 bilhões, e depois apanha 9 milhões de dias, ou 27.000 anos. As coisas começam a acalmar, apesar de a Intel ter deixado de explicar por que um cliente típico acessaria números de ponto flutuante aleatoriamente.
- *12 de dezembro de 1994:* a IBM Research Division discute o cálculo da Intel quanto à taxa de erros (você pode acessar esse artigo visitando www.mkp.com/books_catalog/cod/links.htm). A IBM afirma que os programas comuns de planilha, calculando por 15 minutos por dia, poderiam produzir erros relacionados ao bug do Pentium com tanta freqüência quanto uma vez a cada 24 dias. A IBM considera 5.000 divisões por segundo, por 15 minutos, gerando 4,2 milhões de divisões por dia, e não considera a distribuição aleatória de números, calculando em vez disso as chances como uma em 100 milhões. Como resultado, a IBM imediatamente deixa de enviar todos os computadores pessoais IBM baseados no Pentium. As coisas se aquecem novamente para a Intel.
- *21 de dezembro de 1994:* a Intel lança o seguinte comunicado, assinado pelo presidente da Intel, pelo diretor executivo, pelo diretor de operações e pelo presidente do comitê: "Nós, da Intel, queremos sinceramente pedir desculpas por nosso tratamento da falha recentemente publicada do processador Pentium. O símbolo Intel Inside significa que seu computador possui um microprocessador que não fica atrás de nenhum outro em qualidade e desempenho. Milhares de funcionários da Intel trabalham muito para garantir que isso aconteça. Mas nenhum microprocessador é totalmente perfeito. O que a Intel continua a acreditar é que, tecnicamente, um problema extremamente pequeno assumiu vida própria. Embora a Intel mantenha a qualidade da versão atual do processador Pentium, reconhecemos que muitos usuários possuem pro-

blemas. Queremos resolvê-los. A Intel trocará a versão atual do processador Pentium por uma versão atualizada, em que essa falha de divisão de ponto flutuante está corrigida, para qualquer proprietário que o solicite, sem qualquer custo, durante toda a vida de seu computador". Os analistas estimam que essa troca custou à Intel cerca de US\$500 milhões, e os funcionários da Intel não receberam um bônus de Natal naquele ano.

Essa história nos faz refletir sobre alguns pontos. Seria mais econômico ter consertado o bug em julho de 1994? Qual foi o custo para reparar o dano causado à reputação da Intel? E qual é a responsabilidade corporativa na divulgação de bugs em um produto tão utilizado e confiado como um microprocessador?

Em abril de 1997, outro bug de ponto flutuante foi revelado nos microprocessadores Pentium Pro e Pentium II. Quando as instruções store de ponto flutuante para inteiro (fist, fistp) encontram um número de ponto flutuante negativo que seja muito grande para caber em uma word de 16 ou 32 bits sendo convertida para inteiro, elas definem o bit errado na word de status FPO (exceção de precisão, no lugar de exceção por operação inválida). Para o crédito da Intel, dessa vez, eles reconheceram publicamente o bug e ofereceram um reparo de software para contorná-lo – uma reação muito diferente da que aconteceu em 1994.

3.9

Comentários finais

A aritmética computacional é distinguida da aritmética de lápis e papel pelas restrições da precisão limitada. Esse limite pode resultar em operações inválidas, por meio do cálculo de números maiores ou menores do que os limites predefinidos. Essas anomalias, chamadas "overflow" ou "underflow", podem resultar em exceções ou interrupções, eventos de emergência, semelhantes a chamadas de sub-rotina não planejadas. O Capítulo 5 discute as exceções com mais detalhes.

A aritmética de ponto flutuante tem o desafio adicional de ser uma aproximação de números reais, e é preciso tomar cuidado para garantir que o número selecionado pelo computador seja a representação mais próxima do número real. Os desafios da imprecisão e da representação limitada fazem parte da inspiração para o campo da análise numérica.

Com o passar dos anos, a aritmética computacional tornou-se padronizada, aumentando bastante a portabilidade dos programas. A aritmética de inteiros binários com complemento a dois e a aritmética de ponto flutuante binário do padrão IEEE 754 são encontrados na grande maioria dos computadores vendidos hoje. Por exemplo, cada computador desktop vendido desde que este livro foi impresso pela primeira vez segue essas convenções.

Um efeito colateral do computador com programa armazenado é que os padrões de bits não possuem significado inherente. O mesmo padrão de bits pode representar um inteiro com sinal, um inteiro sem sinal, um número de ponto flutuante, uma instrução e assim por diante. É a instrução que opera sobre os bits que determina seu significado.

Com a explicação sobre aritmética computacional deste capítulo vem uma descrição de muito mais do conjunto de instruções do MIPS. Um ponto de confusão são as instruções explicadas neste capítulo *versus* as instruções executadas pelos chips MIPS *versus* as instruções aceitas pelos montadores MIPS. As duas figuras seguintes tentam esclarecer isso.

A Figura 3.24 lista as instruções MIPS abordadas neste capítulo e no Capítulo 2. Chamamos o conjunto de instruções da esquerda da figura de *núcleo MIPS*. As instruções à direita são chamadas *núcleo aritmético MIPS*. No lado esquerdo da Figura 3.25 estão as instruções que o processador MIPS executa que não se encontram na Figura 3.24. Chamamos o conjunto completo de instrução de hardware de *MIPS-32*. À direita da Figura 3.25 estão as instruções aceitas pelo montador, que não fazem parte do MIPS-32. Chamamos esse conjunto de instruções de *PseudoMIPS*.

Subconjunto de instruções	Inteiro	Ponto flutuante
Núcleo MIPS	95%	57%
Núcleo aritmético MIPS	0%	41%
MIPS-32 restantes	5%	2%

Instruções do núcleo MIPS	Nome	Formato	Núcleo aritmético MIPS	Nome	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	lwcl	I
load halfword unsigned	lh	I	store word to floating-point single	swcl	I
store halfword	sh	I	load word to floating-point double	ldcl	I
load byte unsigned	lb	I	store word to floating-point double	sdcl	I
store byte	sb	I	branch on floating-point true	bclt	I
branch on equal	beq	I	branch on floating-point false	bclf	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 O conjunto de instruções MIPS abordado até aqui. Este livro se concentra nas instruções da coluna da esquerda.

A Figura 3.26 indica a popularidade das instruções MIPS para os benchmarks de inteiro e ponto flutuante SPEC2000. Todas as instruções listadas foram responsáveis por pelo menos 1% das instruções executadas. A tabela a seguir resume essa informação.

Observe que, embora os programadores e escritores de compilador possam utilizar MIPS-32 para ter um menu de opções mais rico, as instruções do núcleo MIPS dominam a execução SCPEC2000 de inteiros, e o núcleo de inteiros mais aritmético domina o ponto flutuante SPEC2000.

MIPS-32 restantes	Nome	Formato	Pseudo MIPS	Nome	Formato
exclusive or ($rs \oplus rt$)	xor	R	Move	move	rd,rs
exclusive or immediate	xori	I	absolute value	abs	rd,rs
shift right arithmetic	sra	R	not ($\neg rs$)	not	rd,rs
shift left logical variable	sllv	R	negate (com ou sem sinal)	negs	rd,rs
shift right logical variable	srlv	R	rotate left	rol	rd,rs,rt
shift right arithmetic variable	srav	R	rotate right	ror	rd,rs,rt
move to Hi	mtih	R	multiply and don't check overflow (com ou sem sinal)	mult	rd,rs,rt
move to Lo	mtlo	R	multiply and check overflow (com ou sem sinal)	mulx	rd,rs,rt
load halfword	lh	I	divide and check overflow	div	rd,rs,rt
load byte	lb	I	divide and don't check overflow	divu	rd,rs,rt
load word left (não alinhado)	lw1	I	remainder (com ou sem sinal)	rem	rd,rs,rt
load word right (não alinhado)	lw2	I	load immediate	li	rd,imm
store word left (não alinhado)	sw1	I	load address	la	rd,addr
store word right (não alinhado)	sw2	I	load double	ld	rd,addr
load linked (atualização atômica)	lt	I	store double	sd	rd,addr
store cond. (atualização atômica)	sc	I	unaligned load word	ulw	rd,addr
move if zero	movz	R			
move if not zero	movn	R	unaligned store word	usw	rd,addr
multiply and add (com ou sem sinal)	madds	R			
multiply and subtract (com ou sem sinal)	msubs	I	unaligned load halfword (com ou sem sinal)	ulhx	rd,addr
branch on \geq zero and link	bgezl	I	unaligned store halfword	ush	rd,addr
branch on $<$ zero and link	bltzl	I	Branch	b	Rótulo
jump and link register	jalr	R	branch on equal zero	beqz	rs,L
branch compare to zero	bzx	I	branch on compare (com ou sem sinal)	bxx	rs,rt,L
branch compare to zero likely	bzx1	I	($x = lt, le, gt, ge$)		
($x = lt, le, gt, ge$)			set equal	seq	rd,rs,rt
branch compare reg likely	bx1	I	set not equal	sne	rd,rs,rt
trap if compare reg	tx	R	set on compare (com ou sem sinal)	sxs	rd,rs,rt
trap if compare immediate	tx1	I	($x = lt, le, gt, ge$)		
($x = eq, neq, lt, le, gt, ge$)			load to floating point (s ou d)	1,f	rd,addr
return from exception	rfe	R	store from floating point (s ou d)	s,f	rd,addr
system call	syscall	I			
break (causa exceção)	break	I			
move from FP to integer	mfcl	R			
move to FP from integer	mtcl	R			
FP move (s ou d)	mov.f	R			
FP move if zero (s ou d)	movz.f	R			
FP move if not zero (s ou d)	movn.f	R			
FP square root (s ou d)	sqrt.f	R			
FP absolute value (s ou d)	abs.f	R			
FP negate (s ou d)	neg.f	R			
FP convert (w, s ou d)	cvt.f,f	R			
FP compare un (s ou d)	c.xn.f	R			

FIGURA 3.25 Conjuntos de Instruções MIPS-32 restantes e “pseudoMIPS”. f significa instruções de ponto flutuante com precisão simples (s) ou dupla (d), e s significa versões com sinal e sem sinal (u). MIPS-32 também possui instruções de PF para multiply e add/sub (madd,f/msub,f), ceiling (ceil,f), truncate (trunc,f), round (round,f) e reciprocal (recip,f).

Núcleo MIPS	Nome	Inteiro	PF	Núcleo aritmético + MIPS-32	Nome	Inteiro	PF
add	add	0%	0%	FP add double	add.d	0%	8%
add immediate	addi	0%	0%	FP subtract double	sub.d	0%	3%
add unsigned	addu	7%	21%	FP multiply double	mul.d	0%	8%
add immediate unsigned	addiu	12%	2%	FP divide double	div.d	0%	0%
subtract unsigned	subu	3%	2%	load word to FP double	l.d	0%	15%
and	and	1%	0%	store word to FP double	s.d	0%	7%
and immediate	andi	3%	0%	shift right arithmetic	sra	1%	0%
or	or	7%	2%	load half	lh	1%	0%
or immediate	ori	2%	0%	branch less than zero	bltz	1%	0%
nor	nor	3%	1%	branch greater or equal zero	bgez	1%	0%
shift left logical	sll	1%	1%	branch less or equal zero	blez	0%	1%
shift right logical	srl	0%	0%	multiply	mul	0%	1%
load upper immediate	lui	2%	5%				
load word	lw	24%	15%				
store word	sw	9%	2%				
load byte	lb	1%	0%				
store byte	sb	1%	0%				
branch on equal (zero)	beq	6%	2%				
branch on not equal (zero)	bne	5%	1%				
jump and link	jal	1%	0%				
jump register	jr	1%	0%				
set less than	slt	2%	0%				
set less than immediate	slti	1%	0%				
set less than unsigned	sltu	1%	0%				
set less than imm. uns.	sltiu	1%	0%				

FIGURA 3.26 A freqüência das instruções MIPS para o benchmark de inteiros e ponto flutuante SPEC2000. Todas as instruções responsáveis por pelo menos 1% das instruções estão incluídas na tabela. As pseudo-instruções são convertidas em MIPS-32 antes da execução e, portanto, não aparecem aqui. Esses dados são do Capítulo 2 de *Arquitetura de Computadores: Uma abordagem quantitativa*, terceira edição.

Para o restante do livro, vamos nos concentrar nas instruções do núcleo MIPS – o conjunto de instruções de inteiros, excluindo multiplicação e divisão – para facilitar a explicação do projeto do computador. Como podemos ver, o núcleo MIPS inclui as instruções MIPS mais populares, e tenha certeza de que compreender um computador que execute o núcleo MIPS lhe dará base suficiente para entender computadores com projetos ainda mais ambiciosos.

3.10

Perspectiva histórica e leitura adicional

Esta seção estuda a história do ponto flutuante desde von Neumann, incluindo o esforço surpreendentemente controvertido dos padrões do IEEE, mais o raciocínio para a arquitetura de pilha de 80 bits para ponto flutuante do IA-32. Ver ■ Seção 3.10 no CD.

A Lei de Gresham ("dinheiro ruim expulsa o bom") para os computadores diria: "o rápido expulsa o lento, mesmo que o rápido seja errado". W. Kahan, 1992

*Nunca ceda,
nunca ceda,
nunca, nunca,
nunca – em nada,
seja grande ou
pequeno,
importante ou
insignificante –
nunca ceda.*
Winston Churchill,
discurso na Harrow
School, 1941,
Abroad, 1869

3.11**Exercícios**

- 3.1** [3] <§3.2> Converta 4096_{dec} para um número binário em complemento a dois a 32 bits.
- 3.2** [3] <§3.2> Converta -2047_{dec} para um número binário em complemento a dois a 32 bits.
- 3.3** [5] <§3.2> Converta $-2.000.000_{dec}$ para um número binário em complemento a dois a 32 bits.
- 3.4** [5] <§3.2> Que número decimal esse número binário em complemento a dois representa: 1111 1111 1111 1111 1111 0000 0110_{bin}?
- 3.5** [5] <§3.2> Que número decimal esse número binário em complemento a dois representa: 1111 1111 1111 1111 1111 1110 1111_{bin}?
- 3.6** [5] <§3.2> Que número decimal esse número binário em complemento a dois representa: 0111 1111 1111 1111 1111 1110 1111_{bin}?
- 3.7** [10] <§3.2> Encontre a seqüência mais curta de instruções MIPS para determinar o valor absoluto de um inteiro em complemento a dois. Converta essa instrução (aceita pelo montador MIPS):

```
abs $t2,$t3
```

Essa instrução significa que o registrador \$t2 tem uma cópia do registrador \$t3 se o registrador \$t3 for positivo, e o complemento a dois do registrador \$t3 se \$t3 for negativo. (Dica: isso pode ser feito com três instruções.)

- 3.8** [10] <§3.2> ■ **Aprofundando o aprendizado:** Representações numéricas
- 3.9** [10] <§3.2> Se A é um endereço de 32 bits, normalmente uma seqüência de instruções como

```
lui $t0, A_upper
ori $t0, $t0, A_lower
lw $s0, 0($t0)
```

pode ser usada para carregar a word que se encontra em A em um registrador (neste caso, \$s0). Considere a alternativa a seguir, que é mais eficiente:

```
lui $t0, A_upper_adjusted
lw $s0, A_lower($t0)
```

Descreva como A_upper é ajustado para permitir que esse código mais simples funcione. (Dica: A_upper precisa ser ajustado porque A_lower terá o sinal estendido.)

- 3.10** [10] <§3.3> Encontre a menor seqüência de instruções MIPS para determinar se existe um carry vindo da soma de dois registradores, digamos, os registradores \$t3 e \$t4. Coloque 0 ou 1 no registrador \$t2 se o carry for 0 ou 1, respectivamente. (Dica: isso pode ser feito com duas instruções.)

- 3.11** [15] <§3.3> ■ **Aprofundando o aprendizado:** Escrevendo código MIPS para realizar aritmética

- 3.12** [15] <§3.3> Suponha que todas as instruções de desvio condicional, exceto beq e bne, fossem removidas do conjunto de instruções MIPS, junto com slt e todas as suas variantes (slt, sltu, slti). Mostre como realizar

```
slt $t0, $s0, $s1
```

usando o conjunto de instruções modificado, em que slt não está disponível. (Dica: isso exige mais de duas instruções.)

- 3.13** [10] <§3.3> Desenhe as portas para o bit de soma de um somador, dada a equação da ■ página B-21.

- 3.14** [5] <§3.4> ■ **Aprofundando o aprendizado:** Escrevendo código MIPS para realizar aritmética

- 3.15** [20] <§3.4> ■ **Aprofundando o aprendizado:** Escrevendo código MIPS para realizar aritmética

- 3.16** [2 semanas] <§3.4> ■ **Aprofundando o aprendizado:** Simulando máquinas MIPS
- 3.17** [1 semana] <§3.4> ■ **Aprofundando o aprendizado:** Simulando máquinas MIPS
- 3.18** [5] <§3.4> ■ **Aprofundando o aprendizado:** Somadores com carry lookahead
- 3.19** [15] <§3.4> ■ **Aprofundando o aprendizado:** Somadores com carry lookahead
- 3.20** [10] <§3.4> ■ **Aprofundando o aprendizado:** Desempenho relativo dos somadores
- 3.21** [15] <§3.4> ■ **Aprofundando o aprendizado:** Desempenho relativo dos somadores
- 3.22** [15] <§3.4> ■ **Aprofundando o aprendizado:** Desempenho relativo dos somadores
- 3.23** [15] [30] <§3.4> [] **Aprofundando o aprendizado:** Registradores especiais do MIPS
- 3.24** <§§3.3, 3.4, 3.5> Com $x = 0000\ 0000\ 0000\ 0000\ 0000\ 0101\ 1011_{bin}$ e $y = 0000\ 0000\ 0000\ 0000\ 0000\ 0010\ 1101\ 0111_{bin}$ representando inteiros em complemento a dois com sinal, execute, mostrando todo o trabalho:

- $x + y$
- $x - y$
- $x * y$
- x / y

3.25 [20] <§§3.3, 3.4, 3.5> Execute as mesmas operações do Exercício 3.24, mas com $x = 1111\ 1111\ 1111\ 1011\ 0011\ 0101\ 0011_{bin}$ e $y = 0000\ 0000\ 0000\ 0000\ 0000\ 0010\ 1101\ 0111_{bin}$.

3.26 [30] <§3.5> O algoritmo de divisão na Figura 3.11 é denominado *divisão com restauração*, pois toda vez que o resultado da subtração do divisor pelo dividendo é negativa, você precisa somar o divisor de novo ao dividendo para restaurar o valor original. Lembre-se de que deslocar à esquerda é o mesmo que multiplicar por 2. Vejamos o valor da metade esquerda do Resto novamente, começando com a etapa 3b do algoritmo de divisão e prosseguindo até a etapa 2:

$$(Resto + Divisor) \times 2 - Divisor$$

Esse valor é criado pela restauração do Resto por meio da soma do Divisor, deslocando a soma para a esquerda e depois subtraindo o Divisor. Simplificando, o resultado que obtemos é

$$Resto \times 2 + Divisor \times 2 - Divisor = Resto \times 2 + Divisor$$

Com base nessa observação, escreva um algoritmo de *divisão sem restauração*, usando a notação da Figura 3.11, que não soma o Divisor ao Resto na etapa 3b. Mostre que seu algoritmo funciona dividindo $0000\ 1011_{bin}$ por 0011_{bin} .

3.27 [15] <§§3.2, 3.6> A Seção “Colocando em perspectiva” na página 163 menciona que os bits não possuem significado inherente. Dado o padrão de bits:

1010 1101 0001 0000 0000 0000 0010

o que ele representa, supondo que seja

- um inteiro em complemento a dois?
- um inteiro sem sinal?
- um número em ponto flutuante com precisão simples?
- uma instrução MIPS? Você poderá verificar as Figuras 3.20 e ■ A.10.2.

3.28 <§§3.2, 3.6> Este exercício é semelhante ao Exercício 3.27, mas desta vez use o padrão de bits

0010 0100 1001 0010 0100 1001 0010 0100

3.29 [10] [10] <§3.6> ■ **Aprofundando o aprendizado:** Representações de números em ponto flutuante

- 3.30** [10] <§3.6> ■ **Aprofundando o aprendizado:** Representações de números em ponto flutuante
- 3.31** [10] <§3.6> ■ **Aprofundando o aprendizado:** Escrevendo código MIPS para realizar aritmética em ponto flutuante
- 3.32** [5] <§3.6> Some $2,85_{dec} \times 10^3$ a $9,84_{dec} \times 10^4$, supondo que você tenha apenas três dígitos significativos, primeiro com dígitos de guarda e arredondamento e depois sem eles.
- 3.33** [5] <§3.6> Este exercício é semelhante ao Exercício 3.32, mas desta vez use os números $3,63_{dec} \times 10^4$ e $6,87_{dec} \times 10^3$.
- 3.34** [5] <§3.6> Mostre a representação binária no padrão IEEE 754 para o número de ponto flutuante 20_{dec} em precisão simples e dupla.
- 3.35** [5] <§3.6> Este exercício é semelhante ao Exercício 3.34, mas desta vez substitua o número 20_{dec} por $20,5_{dec}$.
- 3.36** [10] <§3.6> Este exercício é semelhante ao Exercício 3.34, mas desta vez substitua o número 20_{dec} por $0,1_{dec}$.
- 3.37** [10] <§3.6> Este exercício é semelhante ao Exercício 3.34, mas desta vez substitua o número 20_{dec} pela fração decimal $-5/6$.
- 3.38** [10] <§3.6> Suponha que seja incluída uma nova instrução que some três números em ponto flutuante. Supondo que os somemos com um somador triplo, com bits de guarda, arredondamento e sticky bit, temos resultados garantidos dentro de 1 ulp dos resultados, usando duas instruções add distintas?
- 3.39** [15] <§3.6> Com $x = 0100\ 0110\ 1101\ 1000\ 0000\ 0000\ 0000_{bin}$ e $y = 1011\ 1110\ 1110\ 0000\ 0000\ 0000\ 0000_{bin}$ representando números em ponto flutuante IEEE 754 com precisão simples, execute, mostrando todo o trabalho:
- $x + y$
 - $x * y$
- 3.40** [15] <§3.6> Com $x = 0101\ 1111\ 1011\ 1110\ 0100\ 0000\ 0000\ 0000_{bin}$, $y = 0011\ 1111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000_{bin}$ e $z = 1101\ 1111\ 1011\ 1110\ 0100\ 0000\ 0000\ 0000_{bin}$ representando números em ponto flutuante IEEE 754 com precisão simples, execute, mostrando todo o trabalho:
- $x + y$
 - (resultado de a) + z
 - Por que esse resultado não é intuitivo?
- 3.41** [20] <§§3.6, 3.7> O padrão de ponto flutuante IEEE 754 especifica precisão dupla de 64 bits com um significando de 53 bits (incluindo o 1 implícito) e um expoente de 11 bits. A arquitetura IA-32 oferece uma opção de precisão estendida com um significando de 64 bits e um expoente de 16 bits.
- Supondo que a precisão estendida é semelhante à precisão simples e dupla, qual é o bias do expoente?
 - Qual é o intervalo de números que podem ser representados pela opção de precisão estendida?
 - Até que ponto essa precisão é maior em comparação com a precisão dupla?
- 3.42** [5] <§§3.6, 3.7> A representação interna dos números de ponto flutuante na arquitetura IA-32 é de 80 bits de largura. Isso inclui um expoente de 16 bits. Contudo, também está anunciado um significando de 64 bits. Como isso é possível?
- 3.43** [10] <§3.7> Embora o IA-32 permita números em ponto flutuante de 80 bits internamente, sómente números de ponto flutuante de 64 bits podem ser lidos ou escritos na memória. Começando com apenas números de 64 bits, quantas operações são exigidas antes que o intervalo completo dos expoentes de 80 bits seja utilizado? Mostre um exemplo.

- 3.44** [25] <§3.8> ■ **Aprofundando o aprendizado:** Ponto flutuante em algoritmos
- 3.45** [30] <§3.8> ■ **Aprofundando o aprendizado:** Modos de arredondamento de ponto flutuante
- 3.46** [30] <§3.8> ■ **Aprofundando o aprendizado:** Números não-normalizados
- 3.47** [10] <§3.9> ■ **Aprofundando o aprendizado:** Avaliando freqüências de instruções
- 3.48** [10] <§3.9> ■ **Aprofundando o aprendizado:** Avaliando freqüências de instruções
- 3.49** [10] <§3.9> ■ **Aprofundando o aprendizado:** Avaliando freqüências de instruções
- 3.50** [10] <§3.9> ■ **Aprofundando o aprendizado:** Avaliando freqüências de instruções
- 3.51** [15] <§3.9> ■ **Aprofundando o aprendizado:** Avaliando o desempenho
- 3.52** [15] <§3.9> ■ **Aprofundando o aprendizado:** Avaliando o desempenho

§3.2: página 126, 3, pois cada caracter em uma string Java utiliza 16 bits mais uma word para o tamanho.

§3.3: página 131, 2.

§3.6: página 164, 3.

**Respostas
das Seções
“Verifique
você mesmo”**

Computadores no mundo real

Reconstruindo o mundo antigo

Problema: analisar e entender os sítios arqueológicos é um processo desafiador. Podemos encontrar maneiras de usar computadores para ajudar os pesquisadores a explorar os sítios e artefatos arqueológicos descobertos nesses sítios?

Solução: a arqueologia está passando por uma revolução com o uso de ferramentas digitais para mapear sítios antigos, reconstruir artefatos danificados e recriar sítios antigos em três dimensões. Entre as novas técnicas importantes sendo usadas para analisar e recriar sites estão as seguintes:

- O uso de sistemas de informações geográficas (GIS) para ajudar a medir os sítios com precisão. O GIS utiliza sistemas de posicionamento global (GPS) para medir os locais com precisão, permitindo medições rápidas e precisas de um sítio.
- Laser para obter medições precisas da estrutura bi e tridimensional dos objetos. O laser está sendo usado até mesmo com aeronaves em vôo baixo para obter medições de altura.
- Fotografia digital, para obter imagens precisas de sítios e também de objetos individuais.
- Sistemas de realidade virtual e visualização tridimensional podem usar dados fotográficos digitais e informações geoespaciais precisas para criar versões realísticas de sítios arqueológicos, permitindo que os arqueólogos tenham novas informações, além de compartilhar seu trabalho com outros pesquisadores e o público em geral.



Uma fotografia digital tirada de um modelo de realidade virtual do novo templo em Chavín de Huántar.

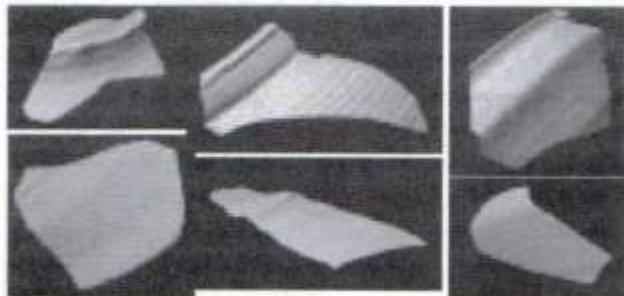
Essas técnicas foram usadas para explorar e criar um modelo de realidade virtual interativo de um sítio arqueológico, chamado Chavín de Huántar, que se encontra nos planaltos peruanos. A imagem apresentada anteriormente é uma fotografia digital tirada do modelo de realidade virtual. Chavín de Huántar foi ocupada desde cerca de 1000 a.C. e é anterior à civilização inca clássica em mais de 1.000 anos. Fotos altamente detalhadas, em conjunto com medições de mais de 25.000 pontos, permitem uma reconstrução do modelo virtual preciso. A imagem anterior é do novo templo em Chavín, que desempenhou um papel importante no estabelecimento de uma autoridade religiosa formalizada no Novo Mundo.

A modelagem e a reconstrução tridimensionais também foram usadas na reconstrução dos

artefatos a partir dos fragmentos. As imagens na parte inferior esquerda são os fragmentos de potes encontrados em Petra, o famoso sítio arqueológico na Jordânia. À direita está uma reconstrução por computador do mesmo vaso original, destacando a posição de um dos fragmentos.

Para saber mais, veja estas referências

- Reconstrução de objetos a partir de fragmentos no Laboratório SHAPE da Brown University
- A exploração de Chavín de Huántar (inclui passeio de realidade virtual pelo sítio)



Imagens de fragmentos de potes encontrados em Petra, Jordânia.



Uma reconstrução por computador dos fragmentos da foto anterior.

4

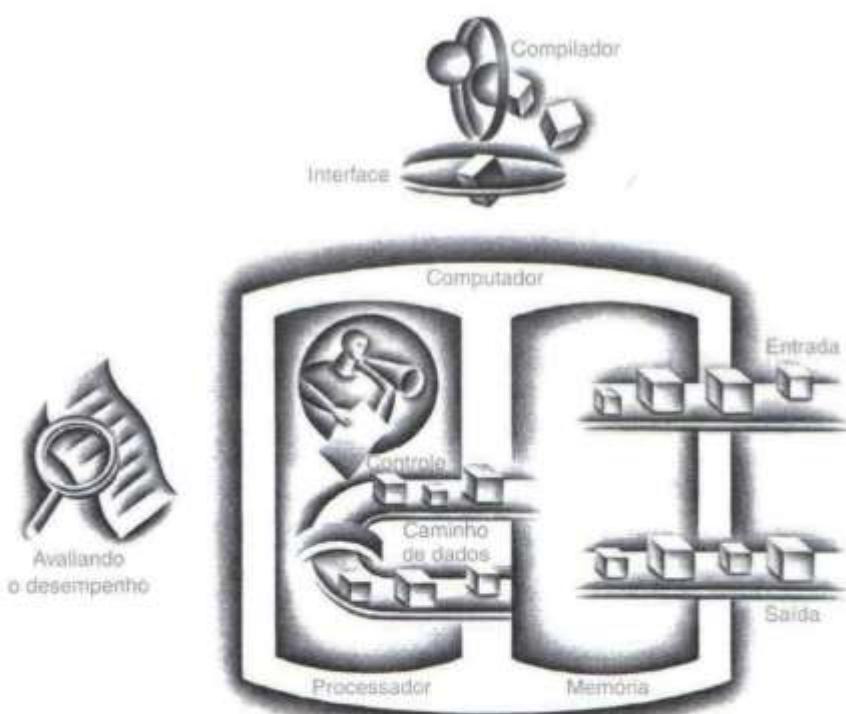
Avaliando e Compreendendo o Desempenho

O tempo descobre a verdade.

Sêneca

4.1	Introdução	182
4.2	Desempenho da CPU e seus fatores	187
4.3	Avaliando desempenho	192
4.4	Vida real: dois benchmarks SPEC e o desempenho dos recentes processadores Intel	196
4.5	Falácia e armadilhas	201
4.6	Comentários finais	204
■ 4.7	Perspectiva histórica e leitura complementar	205
4.8	Exercícios	205

Os cinco componentes clássicos de um computador



4.1

Introdução

Este capítulo explica como medir, informar e resumir o desempenho e descreve os principais fatores que determinam o desempenho de um computador. Uma importante razão para examinar o desempenho é que o desempenho do hardware normalmente é fundamental para a eficiência de um sistema inteiro, incluindo hardware e software. Uma grande parte deste capítulo, especialmente esta seção e a próxima, serão vitais para o entendimento dos três capítulos seguintes. O restante do capítulo fornece importantes conceitos e princípios que qualquer pessoa que queira avaliar o desempenho de um sistema computacional deve conhecer. O material nas Seções 4.3 a 4.5, no entanto, não é necessário para acompanhar os próximos capítulos e pode ser examinado mais tarde.

Avaliar o desempenho desse tipo de sistema pode ser bastante complicado. A escala e a complexidade dos sistemas de software modernos, em conjunto com a ampla gama de técnicas de melhoria de desempenho adotadas pelos projetistas de hardware, tornaram a avaliação de desempenho muito mais difícil. É simplesmente impossível sentar com um manual do conjunto de instruções e um sistema de software significativo e determinar, apenas por análise, a velocidade com que o software será executado no computador. Na verdade, para diferentes tipos de aplicações, diversas medições de desempenho podem ser apropriadas, e diferentes aspectos de um sistema computacional podem ser os mais significativos na determinação do desempenho geral.

É claro que ao escolher um computador, o desempenho quase sempre é um fator importante. Medir e comparar precisamente diferentes computadores é essencial para os compradores e, portanto, para os projetistas. As pessoas que vendem computadores sabem disso também. Freqüentemente, os vendedores querem que você veja o computador sob as melhores condições possíveis, mesmo que essas condições não reflitam fielmente as necessidades da aplicação do comprador. Em alguns casos, são feitas afirmações sobre computadores que não fornecem discernimento útil para aplicação real alguma. Portanto, para escolher um computador, é importante entender como medir o desempenho da melhor forma possível e conhecer as limitações dessas medições de desempenho.

Nosso interesse no desempenho, todavia, vai além dos aspectos externos da avaliação de um computador. Para compreender por que um software funciona de uma determinada maneira, por que um conjunto de instruções pode ser implementado para funcionar melhor do que outro, ou como algum recurso de hardware afeta o desempenho, precisamos entender o que determina o desempenho de um computador. Por exemplo, para melhorar o desempenho de um sistema de software, precisamos entender que fatores no hardware contribuem para o desempenho geral do sistema e a importância relativa desses fatores. Esses fatores podem incluir o quanto bem o programa usa as instruções do computador, o quanto bem o hardware associado implementa as instruções e o quanto bem a memória e os sistemas de E/S operam.

Saber determinar o impacto do desempenho desses fatores é crucial para entender a motivação por trás do projeto de determinadas partes do computador, como veremos nos próximos capítulos.

O restante desta seção descreve diferentes maneiras de determinar o desempenho. Na Seção 4.2, descrevemos a métrica para medir o desempenho tanto da perspectiva de um usuário de computador quanto da de um projetista. Também veremos como essas métricas estão relacionadas e apresentaremos a equação clássica para desempenho do processador, que usaremos em todo o texto. As Seções 4.3 e 4.4 descrevem como melhor escolher benchmarks para avaliar computadores e como resumir corretamente o desempenho de um grupo de programas. A Seção 4.4 também descreve um conjunto de benchmarks de CPU comumente usado e examina as medidas para diversos processadores Intel usando esses benchmarks. Finalmente, na Seção 4.5, examinaremos algumas das armadilhas em que têm caído alguns projetistas e aqueles que analisam e reportam desempenho.

Definindo desempenho

Quando falamos que um computador possui melhor desempenho do que outro, o que queremos dizer? Embora essa pergunta possa ser simples, uma analogia com aviões de passageiros mostra como

a questão do desempenho pode ser sutil. A Figura 4.1 ilustra alguns aviões de passageiros típicos, juntamente com suas velocidades, autonomias e capacidades. Se quiséssemos saber qual dos aviões nessa tabela teve o melhor desempenho, precisaríamos primeiro definir desempenho. Por exemplo, considerando diferentes medidas de desempenho, vemos que o avião com a maior velocidade é o Concorde, o avião com a maior autonomia é o DC-8 e o avião com a maior capacidade é o 747.

Vamos supor que tenhamos definido desempenho em termos de velocidade. Isso ainda deixa duas definições possíveis. Você poderia definir o avião mais rápido como o que tem a maior velocidade, levando um único passageiro de um ponto a outro no menor tempo. Entretanto, se você estivesse interessado em transportar 450 passageiros de um ponto a outro, o 747 claramente seria o mais rápido, como mostra a última coluna. Assim também, podemos definir desempenho de computador de várias maneiras diferentes.

Avião	Capacidade de passageiros	Autonomia de voo (milhas)	Velocidade de voo (milhas por hora)	Vazão de passageiros (passageiros × m.p.h.)
Boeing 777	375	4.630	610	228.750
Boeing 747	470	4.150	610	286.700
BAC/Sud Concorde	132	4.000	1350	178.200
Douglas DC-8-50	146	8.720	544	79.424

FIGURA 4.1 A capacidade, autonomia e velocidade de diversos aviões comerciais. A última coluna mostra o índice com que o avião transporta passageiros, que é a capacidade vezes a velocidade (ignorando a autonomia e os tempos de decolagem e de aterrissagem).

Se você estivesse executando um programa em dois computadores desktop diferentes, diria que o mais rápido é o computador que termina a tarefa primeiro. Se você estivesse gerenciando um CPD que tivesse vários servidores realizando tarefas submetidas por muitos usuários, você diria que o computador mais rápido é aquele que completou a maior quantidade de tarefas durante um dia. Como um usuário de computador individual, você está interessado em reduzir o **tempo de resposta** – o tempo entre o início e o término de uma tarefa –, também chamado de **tempo de execução**. Os gerentes de CPDs normalmente estão interessados em aumentar a **vazão** – a quantidade total de trabalho feito em um determinado tempo. Portanto, na maioria dos casos, precisaremos de diferentes métricas de desempenho e diferentes conjuntos de aplicações para comparar computadores desktop com servidores, e os computadores embutidos exigem ainda outras métricas e aplicações. Veremos exemplos disso na Seção 4.4 quando analisarmos diferentes benchmarks SPEC: um para medir desempenho de CPU (SPEC CPU) e outro para medir desempenho de servidores Web (SPECweb99).

tempo de resposta
Também chamado de **tempo de execução**, o tempo total necessário para o computador completar uma tarefa, como acesso a disco, acesso à memória, atividade de E/S, overhead do sistema operacional, tempo de execução de CPU etc.

VAZÃO E TEMPO DE RESPOSTA

As seguintes mudanças em um sistema computacional aumentam a vazão, diminuem o tempo de resposta ou as duas coisas?

EXEMPLO

1. Substituir o processador em um computador por uma versão mais rápida.
2. Incluir processadores adicionais em um sistema que usa múltiplos processadores para tarefas distintas – por exemplo, busca na Web.

Diminuir o tempo de resposta quase sempre melhora a vazão. Portanto, no caso 1, tanto o tempo de resposta quanto a vazão são melhorados. No caso 2, como nenhuma tarefa é realizada primeiro, apenas a vazão é melhorada. Se, no entanto, a demanda por processamento no segundo caso fosse quase tão grande quanto a vazão, o sistema poderia forçar as requisições a se enfileirar. Nesse caso, aumentar a vazão também poderia melhorar o tempo de resposta, já que isso reduziria o tempo de espera na fila. Conseqüentemente, em muitos sistemas computacionais reais, mudar o tempo de execução afeta a vazão e vice-versa.

RESPOSTA

Ao discutirmos o desempenho dos computadores, nos próximos capítulos, estaremos preocupados principalmente com o tempo de resposta. (No Capítulo 8, sobre sistemas de entrada/saída, discutiremos as medidas relacionadas à vazão.) Para maximizar o desempenho, devemos minimizar o tempo de execução de uma determinada tarefa. Assim, podemos relacionar o desempenho com o tempo de execução para um computador X:

$$\text{Desempenho}_X = \frac{1}{\text{Tempo de execução}_X}$$

Isso significa que para dois computadores X e Y, se o desempenho de X é maior do que o desempenho de Y, temos

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

$$\frac{1}{\text{Tempo de execução}_X} > \frac{1}{\text{Tempo de execução}_Y}$$

$$\text{Tempo de execução}_Y > \text{Tempo de execução}_X$$

Ou seja, o tempo de execução em Y é maior do que em X, se X for mais rápido do que Y.

Na análise do projeto de um computador, normalmente queremos relacionar o desempenho de dois computadores diferentes de maneira quantitativa. Usaremos a frase “X é n vezes mais rápido do que Y” – ou equivalentemente “X é n vezes tão rápido quanto Y” – para representar

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

Se X é n vezes mais rápido do que Y, então, o tempo de execução em Y é n vezes maior do que em X:

$$\frac{\text{Desempenho}_X}{\text{Desempenho}_Y} = \frac{\text{Tempo de execução}_Y}{\text{Tempo de execução}_X} = n$$

DESEMPEÑHO RELATIVO

EXEMPLO

Se um computador A executa um programa em 10 segundos e o computador B executa o mesmo programa em 15 segundos, o quanto A é mais rápido do que B?

RESPOSTA

Sabemos que A é n vezes mais rápido do que B se

$$\frac{\text{Desempenho}_A}{\text{Desempenho}_B} = \frac{\text{Tempo de execução}_B}{\text{Tempo de execução}_A} = n$$

Logo, o fator de desempenho é

$$\frac{15}{10} = 1,5$$

e A, portanto, é 1,5 vez mais rápido do que B.

No exemplo anterior, poderíamos dizer que o computador B é 1,5 vez *mais lento do que* o computador A, já que

$$\frac{\text{Desempenho}_A}{\text{Desempenho}_B} = 1,5$$

significa que

$$\frac{\text{Desempenho}_A}{1,5} = \text{Desempenho}_B$$

Para simplificar, em geral usaremos a terminologia *mais rápido do que* quando tentamos comparar computadores quantitativamente. Como o desempenho e o tempo de execução são recíprocos, aumentar o desempenho significa diminuir o tempo de execução. Para evitar a possível confusão entre os termos *aumentar* e *diminuir*, diremos “melhorar o desempenho” ou “melhorar o tempo de execução” quando quisermos nos referir a “aumentar o desempenho” ou “diminuir o tempo de execução”.

Detalhamento: o desempenho normalmente é caracterizado por limitações de tempo real: ou seja, certos eventos específicos da aplicação precisam ocorrer dentro de um período limitado. Existem dois tipos comuns de limitações de tempo real: tempo real rígido e tempo real flexível. O tempo real rígido define um limite fixo no tempo para responder ou processar algum evento. Por exemplo, o processador embutido que controla um sistema de freio antitravamento (ABS) precisa responder dentro de um limite rígido a partir do momento em que ele recebe um sinal de que as rodas estão travadas. Nos sistemas de tempo real flexível, uma resposta média ou uma resposta dentro de um tempo limitado a uma grande fração dos eventos é suficiente. Por exemplo, manipular quadros de vídeo em um sistema de reprodução de DVD seria um caso de limitação de tempo real flexível, já que é aceitável descartar um quadro se isso ocorrer muito raramente. Nas aplicações de tempo real embutidas, uma vez que a limitação de desempenho do tempo de resposta é atendida, os projetistas costumam otimizar a vazão ou tentar reduzir o custo.

Medindo o desempenho

O tempo é a medida do desempenho dos computadores: o computador que realiza a mesma quantidade de trabalho no menor tempo é o mais rápido. O *tempo de execução* do programa é medido em segundos por programa. Entretanto, o tempo pode ser definido de diferentes maneiras, dependendo do que estamos contando. A definição mais simples de tempo é chamada *tempo de relógio*, *tempo de resposta* ou *tempo decorrido*. Esses termos significam o tempo total para completar uma tarefa, incluindo acessos a disco, acessos à memória, atividades de E/S, overhead do sistema operacional – tudo.

Entretanto, os computadores freqüentemente são compartilhados, e um processador pode trabalhar em diversos programas de maneira simultânea. Nesses casos, o sistema pode tentar otimizar a vazão em vez de tentar minimizar o tempo decorrido para um programa. Portanto, em geral queremos distinguir entre o tempo decorrido e o tempo em que o processador está trabalhando para nós. O **tempo de execução na CPU**, ou simplesmente **tempo de CPU**, que reconhece essa distinção, é o tempo que a CPU gasta computando para essa tarefa e não inclui o tempo gasto esperando E/S ou executando outros programas. (Mas lembre-se de que o tempo de resposta experimentado pelo usuário será o tempo decorrido do programa, não o tempo de CPU.) O tempo de CPU ainda pode ser mais dividido no tempo que a CPU gasta efetivamente no programa, chamado **tempo de CPU do usuário**, e o tempo que a CPU gasta no sistema operacional realizando tarefas a pedido do programa, chamado **tempo de CPU do sistema**. É difícil distinguir precisamente entre o tempo de CPU do usuário e do sistema porque normalmente é difícil atribuir responsabilidade de atividades do sistema operacional para um programa em vez de outro, e devido às diferenças de funcionalidade entre os sistemas operacionais.

tempo de execução na CPU Também chamado **tempo de CPU**, o tempo real que a CPU gasta computando para uma tarefa específica.

tempo de CPU do usuário O tempo de CPU gasto efetivamente em um programa.

tempo de CPU do sistema O tempo de CPU gasto no sistema operacional realizando tarefas a pedido do programa.

Em nome da consistência, manteremos a distinção entre o desempenho baseado no tempo decorrido e o desempenho baseado no tempo de execução da CPU. Frequentemente usaremos o termo *desempenho do sistema* para nos referirmos ao tempo decorrido em um sistema sem outros aplicativos, e usaremos o termo *desempenho da CPU* para nos referirmos ao tempo de CPU. iremos nos concentrar no desempenho da CPU neste capítulo, embora nossas discussões de como resumir o desempenho possam ser aplicadas tanto às medidas do tempo decorrido quanto às do tempo de CPU.

ciclo de clock Também chamado período de clock, ciclo ou clock, o tempo para cada período de clock, normalmente do clock do processador, que opera em uma velocidade constante.

período de clock A duração de cada ciclo de clock.

Embora, como usuários de computadores, o tempo seja importante, quando examinamos os detalhes de um computador, é conveniente pensar no desempenho para a métrica. Em especial, os projetistas de computador podem querer pensar em um computador usando uma medida que se refira à rapidez com que o hardware pode realizar funções básicas. Quase todos os computadores são construídos usando um relógio que trabalha em uma velocidade constante e determina quando os eventos ocorrem no hardware. Esses intervalos de tempo são chamados de **ciclos de clock** (ou períodos de clock, ciclos ou clocks). Os projetistas se referem à duração de um **período de clock** como o tempo de cada *ciclo de clock* (por exemplo, 0,25 nanosegundo, 0,25ns, 250 picosegundos ou 250ps) e como a *velocidade de clock* (por exemplo, 4 gigahertz ou 4GHz), que é o inverso do período de clock. Na próxima seção, formalizaremos a relação entre os ciclos de clock do projetista de hardware e os segundos do usuário do computador.

Entendendo o desempenho do programas

Diferentes aplicações são sensíveis a diferentes aspectos do desempenho de um sistema computacional. Muitas aplicações, especialmente aquelas executadas em servidores, dependem ao máximo do desempenho de E/S, que, por sua vez, se baseia no hardware e no software; além disso, o tempo total decorrido, medido por um relógio, é a medida que interessa. Em algumas aplicações, o usuário pode se importar com a vazão, o tempo de resposta ou uma combinação dos dois (por exemplo, a vazão máxima com um tempo de resposta de pior caso). Para melhorar o desempenho de um programa, é preciso ter uma clara definição do que importa para a métrica do desempenho e, depois, procurar os gargalos de desempenho medindo a execução do programa e também procurar os prováveis gargalos. Nos capítulos a seguir, descreveremos como procurar gargalos e melhorar o desempenho em várias partes do sistema.

Verifique você mesmo

1. Suponha que queremos saber se uma aplicação que usa o desktop de um cliente e um servidor remoto está limitada pelo desempenho da rede. Para as mudanças a seguir, diga se apenas a vazão melhora, se o tempo de resposta e a vazão melhoram ou se nenhum dos dois melhora.
 - a. Um canal de rede extra é acrescentado entre o cliente e o servidor, aumentando a vazão total da rede e reduzindo o atraso para obter acesso à mesma (já que agora há dois canais).
 - b. O software de rede é melhorado, reduzindo, assim, o atraso da comunicação via rede, mas não melhorando a vazão.
 - c. Mais memória é acrescentada ao computador.
2. O desempenho do computador C é 4 vezes melhor do que o desempenho do computador B, que executa uma determinada aplicação em 28 segundos. Quanto tempo o computador C levará para executar essa aplicação?

4.2 Desempenho da CPU e seus fatores

Os usuários e projetistas com freqüência examinam o desempenho usando diferentes métricas. Se pudessemos relacionar essas diversas métricas, poderíamos determinar o efeito de uma mudança de projeto no desempenho, como percebido pelo usuário. Como estamos nos limitando ao desempenho da CPU nesta seção, a medida de desempenho final é o tempo de execução da CPU. Uma fórmula simples relaciona as métricas mais básicas (ciclos de clock e o tempo do ciclo de clock) ao tempo de CPU:

$$\text{Tempo de execução da CPU para um programa} = \frac{\text{ciclos de clock da CPU para um programa}}{\text{Tempo do ciclo de clock}}$$

Outra alternativa é que, como a velocidade de clock e o tempo do ciclo de clock são inversos,

$$\text{Tempo de execução da CPU para um programa} = \frac{\text{Ciclos de clock de CPU para um programa}}{\text{velocidade de clock}}$$

Essa fórmula deixa claro que o projetista de hardware pode melhorar o desempenho reduzindo a duração do ciclo de clock ou o número de ciclos de clock necessários para um programa. Como veremos neste capítulo e nos Capítulos 5, 6 e 7, o projetista normalmente precisa achar um equilíbrio entre o número de ciclos de clock necessários para um programa e a duração de cada ciclo. Muitas técnicas que diminuem o número de ciclos de clock também aumentam o tempo do ciclo.

MELHORANDO O DESEMPENHO

Nosso programa favorito é executado em 10 segundos no computador A, que possui um clock de 4GHz. Estamos tentando ajudar um projetista de computador a construir um computador B que execute esse programa em 6 segundos. O projetista determinou que um aumento substancial na velocidade de clock é possível, mas esse aumento afetará o restante do projeto da CPU, fazendo com que o computador B exija 1,2 vez mais ciclos de clock do que o computador A para esse programa. Que velocidade de clock devemos dizer para o projetista buscar?

Primeiro, vamos encontrar o número de ciclos de clock necessários para o programa em A:

$$\text{Tempo de CPU}_A = \frac{\text{Ciclos de clock da CPU}_A}{\text{Velocidade de clock}_A}$$

$$10 \text{ segundos} = \frac{\text{Ciclos de clock da CPU}_A}{4 \times 10^9 \frac{\text{ciclos}}{\text{segundo}}}$$

$$\text{Ciclos de clock da CPU}_A = 10 \text{ segundos} \times 4 \times 10^9 \frac{\text{ciclos}}{\text{segundo}} = 40 \times 10^9 \text{ ciclos}$$

O tempo de CPU para B pode ser encontrado usando a seguinte equação:

$$\text{Tempo de CPU}_B = \frac{1,2 \times \text{ciclos de clock de CPU}_A}{\text{velocidade de clock}_B}$$

EXEMPLO

RESPOSTA

$$6 \text{ segundos} = \frac{1,2 \times 40 \times 10^9 \text{ ciclos}}{\text{Velocidade de clock}_B}$$

$$\text{Velocidade de clock}_B = \frac{1,2 \times 40 \times 10^9 \text{ ciclos}}{6 \text{ segundos}} = \frac{8 \times 10^9 \text{ ciclos}}{\text{segundo}} = 8 \text{ GHz}$$

portanto, a velocidade de clock do computador B deve ser o dobro da velocidade de clock do computador A para poder executar o programa em 6 segundos.

Interface hardware/software

As equações em nossos exemplos anteriores não incluem qualquer referência ao número de instruções necessárias para o programa. Entretanto, como o compilador claramente gerou instruções para serem executadas e o computador precisou executar as instruções para rodar o programa, o tempo de execução tem de depender do número de instruções em um programa. Uma forma de pensar no tempo de execução é que ele é igual ao número de instruções executadas multiplicado pelo tempo médio gasto por cada instrução. Logo, o número de ciclos de clock necessários para um programa pode ser escrito como

ciclos de clock por instrução

instrução Um número médio de ciclos de clock por instrução para um programa ou fragmento de programa.

Ciclos de clock da CPU = Instruções para um programa × Média dos ciclos de clock por instrução

O termo **ciclos de clock por instrução**, que é o número médio de ciclos de clock que cada instrução gasta para ser executada, normalmente é abreviado como CPI. Como diferentes instruções podem gastar diferentes quantidades de tempo dependendo do que elas fazem, o CPI é uma média de todas as instruções executadas no programa. O CPI fornece uma maneira de comparar duas implementações diferentes do mesmo conjunto de instruções, já que a contagem de instruções necessária para um programa será, evidentemente, a mesma.

USANDO A EQUAÇÃO DE DESEMPENHO

EXEMPLO

Vamos supor que temos duas implementações do mesmo conjunto de instruções. O computador A tem um tempo de ciclo de clock de 250ps e um CPI de 2,0 para um determinado programa, e o computador B tem um tempo de ciclo de clock de 500ps e um CPI de 1,2 para o mesmo programa. Que computador é mais rápido para esse programa e o quanto mais rápido?

RESPOSTA

Sabemos que cada computador executa o mesmo número de instruções para o programa; vamos chamar esse número de I . Primeiro, encontre o número de ciclos de clock do processador para cada computador:

$$\text{Ciclos de clock da CPU}_A = I \times 2,0$$

$$\text{Ciclos de clock da CPU}_B = I \times 1,2$$

Agora podemos calcular o tempo de CPU para cada computador:

$$\begin{aligned}\text{Tempo de CPU}_A &= \text{Ciclos de clock da CPU}_A \times \text{tempo do ciclo de clock}_A \\ &= I \times 2,0 \times 250\text{ps} = 500 \times I\text{ps}\end{aligned}$$

Da mesma forma, para B:

$$\text{Tempo de CPU}_A = I \times 1,2 \times 500\text{ps} = 600 \times I\text{ps}$$

Claramente, o computador A é mais rápido. O quanto mais rápido é calculado pela razão dos tempos de execução:

$$\frac{\text{Desempenho da CPU}_A}{\text{Desempenho da CPU}_B} = \frac{\text{Tempo de execução}_B}{\text{Tempo de execução}_A} = \frac{600 \times I\text{ps}}{500 \times I\text{ps}} = 1,2$$

Podemos concluir que o computador A é 1,2 vez mais rápido que o computador B para esse programa.

Agora podemos escrever essa equação de desempenho básica em termos de contagem de instruções (o número de instruções executadas pelo programa), de CPI e de tempo do ciclo de clock:

$$\text{Tempo de CPU} = \text{Contagem de instruções} \times \text{CPI} \times \text{Tempo do ciclo de clock}$$

ou

$$\text{Tempo de CPU} = \frac{\text{Contagem de instruções} \times \text{CPI}}{\text{Velocidade de clock}}$$

Essas fórmulas são especialmente úteis porque elas separam os três fatores-chave que afetam o desempenho. Podemos usar essas fórmulas para comparar duas implementações diferentes ou para avaliar um projeto alternativo se soubermos seu impacto sobre esses três parâmetros.

Como podemos determinar o valor desses fatores na equação de desempenho? Podemos medir o tempo de execução da CPU executando o programa, e o tempo do ciclo de clock normalmente é publicado como parte da documentação de um computador. A contagem de instruções e o CPI podem ser mais difíceis de obter. É claro, se soubermos a velocidade de clock e o tempo de execução da CPU, precisaremos apenas da contagem de instruções ou do CPI para determinar o outro parâmetro.

Podemos medir a contagem de instruções usando ferramentas de software que traçam o perfil da execução ou usando um simulador da arquitetura. Como alternativa, podemos usar contadores de hardware, que fornecem o número de instruções executadas, o CPI médio e, muitas vezes, as origens da perda de desempenho. Como a contagem de instruções depende da arquitetura, mas não da implementação exata, podemos medir a contagem de instruções sem saber todos os detalhes da implementação. O CPI, no entanto, depende de uma grande variedade de detalhes de projeto no computador, como o sistema de memória e a estrutura do processador (como veremos nos Capítulos 5, 6 e 7), assim como da mistura de tipos de instrução executado em uma aplicação. Portanto, o CPI varia por aplicação, bem como entre implementações com o mesmo conjunto de instruções.

Os projetistas freqüentemente obtêm o CPI por meio de uma simulação detalhada de uma implementação ou usando contadores de hardware, quando uma CPU está em operação. Algumas vezes, é possível calcular os ciclos de clock da CPU olhando os diferentes tipos de instruções e usando suas contagens de ciclo de clock individuais. Nesses casos, a seguinte fórmula é útil:

$$\text{Ciclos de clock da CPU} = \sum_{i=1}^n (\text{CPI}_i \times C_i)$$

onde C_i é a contagem do número de instruções da classe i executadas, CPI_i é a média dos ciclos por instrução para essa classe de instrução e n é o número de classes de instrução. Lembre-se de que o CPI geral para um programa dependerá do número de ciclos para cada tipo de instrução e da frequência de cada tipo de instrução na execução do programa.

Colocando em perspectiva

A Figura 4.2 mostra as medidas básicas em diferentes níveis no computador e o que está sendo medido em cada caso. Podemos ver como esses fatores são combinados para fornecer o tempo de execução medido em segundos por programa:

$$\text{Tempo} = \frac{\text{Segundos}}{\text{Programa}} = \frac{\text{Instruções}}{\text{Programa}} \times \frac{\text{Ciclos de clock}}{\text{Instrução}} \times \frac{\text{Segundos}}{\text{Ciclo de clock}}$$

Sempre tenha em mente que a única medida completa e confiável do desempenho do computador é o tempo. Por exemplo, mudar o conjunto de instruções para baixar a contagem de instruções pode levar a uma organização com um tempo de ciclo de clock maior que anule a melhoria na contagem de instruções. De igual modo, como o CPI depende do tipo de instrução executada, o código que executa o menor número de instruções pode não ser o mais rápido.

Componentes do desempenho	Unidades de medida
Tempo de execução da CPU para um programa	Segundos para o programa
Contagem de instruções	Instruções executadas para o programa
Ciclos de clock por instrução (CPI)	Número médio de ciclos de clock por instrução
Tempo do ciclo de clock	Segundos por ciclo de clock

FIGURA 4.2 Os componentes básicos do desempenho e como cada um é medido.

Entendendo o desempenho dos programas

Como descrito no Capítulo 1, o desempenho de um programa depende do algoritmo, da linguagem, do compilador, da arquitetura e do hardware em si. A tabela a seguir resume como esses componentes afetam os fatores na equação de desempenho da CPU.

Componente de hardware ou software	Afeta o quê?	Como?
Algoritmo	Contagem de instruções, possivelmente o CPI	O algoritmo determina o número de instruções do programa fonte executadas e, portanto, o número de instruções do processador executadas. O algoritmo também pode afetar o CPI, favorecendo instruções mais lentas ou mais rápidas. Por exemplo, se o algoritmo usar mais operações de ponto flutuante, ele tenderá a ter um CPI mais alto.
Linguagem de programação	Contagem de instruções, CPI	A linguagem de programação certamente afeta a contagem de instruções, já que as instruções na linguagem são traduzidas em instruções do processador, que determinam a contagem de instruções. A linguagem também pode afetar o CPI devido aos seus recursos; por exemplo, uma linguagem com pesado suporte para abstração de dados (como Java) exigirá chamadas indiretas, que usarão instruções de CPI mais altos.
Compilador	Contagem de instruções, CPI	A eficiência do compilador afeta a contagem de instruções e a média de ciclos por instrução, já que o compilador determina a tradução das instruções da linguagem fonte para instruções do computador. O papel do compilador pode ser bastante complexo e afeta o CPI de maneiras complexas.
Conjunto de instruções	Contagem de instruções, velocidade de clock, CPI	O conjunto de instruções afeta os três aspectos do desempenho da CPU, uma vez que ela afeta as instruções necessárias para uma função, o custo em ciclos de cada instrução e a velocidade de clock geral do processador.

COMPARANDO SEGMENTOS DE CÓDIGO

Um projetista de compilador está tentando decidir entre duas seqüências de código para um determinado computador. Os projetistas de hardware forneceram os seguintes fatos:

EXEMPLO

	CPI para esta classe de instrução		
	A	B	C
CPI	1	2	3

Para uma determinada instrução da linguagem de alto nível, o projetista do compilador está considerando duas seqüências de código que exigem as seguintes contagens de instrução:

Seqüência de código	Contagens de instrução para classe de instrução		
	A	B	C
1	2	1	2
2	4	1	1

Qual seqüência de código executa mais instruções? Qual será a mais rápida? Qual é o CPI para cada seqüência?

A seqüência 1 executa $2 + 1 + 2 = 5$ instruções. A seqüência 2 executa $4 + 1 + 1 = 6$ instruções. Logo, a seqüência 2 executa mais instruções.

Podemos usar a equação para os ciclos de clock de CPU com base na contagem de instruções e no CPI para encontrar o número total de ciclos de clock para cada seqüência:

$$\text{Ciclos de clock de CPU} = \sum_{i=1}^n (\text{CPI}_i \times C_i)$$

Isso resulta

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

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

Portanto, a seqüência de código 2 é a mais rápida, mesmo que ela realmente execute uma instrução extra. Como a seqüência de código 2 leva menos ciclos de clock no total mas possui mais instruções, ele precisa ter um CPI menor. Os valores de CPI podem ser calculados por

$$\text{CPI} = \frac{\text{Ciclos de clock da CPU}}{\text{Contagem de instruções}}$$

$$\text{CPI}_1 = \frac{\text{Ciclos de clock da CPU}_1}{\text{Contagem de instruções}_1} = \frac{10}{5} = 2$$

$$\text{CPI}_2 = \frac{\text{Ciclos de clock da CPU}_2}{\text{Contagem de instruções}_2} = \frac{9}{6} = 1,5$$

RESPOSTA

O exemplo anterior mostra o risco de usar apenas um fator (contagem de instruções) para avaliar o desempenho. Quando estiver comparando dois computadores, você precisa considerar todos os três componentes, que se combinam para formar o tempo de execução. Se alguns desses fatores forem idênticos, como a velocidade de clock no exemplo acima, o desempenho poderá ser determinado

mix de instruções Uma medida da freqüência dinâmica de instruções por meio de um ou muitos programas.

comparando-se todos os fatores não idênticos. Como o CPI varia por **mix de instruções**, tanto a contagem de instruções quanto o CPI precisam ser comparados, ainda que as velocidades de clock sejam idênticas. Vários exercícios pedem que você avalie uma série de melhorias no computador e no compilador que afetam a velocidade de clock, CPI e contagem de instruções. Na próxima seção, examinaremos uma medida de desempenho comum que não incorpora todos os termos e, portanto, pode induzir ao engano.

Dois dos principais fatores que afetam o CPI são o desempenho do pipeline, que é a técnica usada por todos os processadores modernos para executar instruções, e o desempenho do sistema de memória. No Capítulo 6, veremos como o desempenho do pipeline é acrescentado ao CPI por meio de stalls e, no Capítulo 7, veremos como o desempenho das caches pode aumentar o CPI devido aos stalls no sistema de memória.

Verifique você mesmo

Uma determinada aplicação escrita em Java roda 15 segundos em um processador de desktop. É lançado um novo compilador Java que exige apenas seis décimos da quantidade de instruções do antigo compilador. Infelizmente, ele aumenta o CPI em 1,1. Em que velocidade podemos esperar que a aplicação seja executada usando esse novo 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. $\frac{15 \times 1,1}{0,6} = 27,5\text{s}$

4.3

Avaliando desempenho

workload Um conjunto de programas executado em um computador que é a coleção real das aplicações executadas por um usuário ou é construído a partir dos programas reais para aproximar esse misto. Um workload típico especifica os programas e as freqüências relativas.

Um usuário de computador que executa os mesmos programas todos os dias seria o candidato perfeito para avaliar um novo computador. O conjunto dos programas executados seria um **workload**. Para avaliar dois sistemas de computador, um usuário simplesmente compararia o tempo de execução do workload nos dois computadores. A maioria dos usuários, no entanto, não está nessa situação. Em vez disso, eles precisam usar outros métodos que medem o desempenho de um computador candidato, esperando que os métodos reflitam com perfeição como o computador funcionará com o workload do usuário. Essa alternativa normalmente é seguida da avaliação do computador usando um conjunto de *benchmarks* – programas especificamente escolhidos para medir desempenho. Os benchmarks formam um workload que o usuário espera que preveja o desempenho do workload real.

Hoje, é amplamente aceito que o melhor tipo de programa para benchmark são as aplicações reais. Essas podem ser aplicações que o usuário emprega regularmente ou apenas aplicações típicas. Por exemplo, em um ambiente em que os usuários sejam principalmente engenheiros, você poderia usar um conjunto de benchmarks com várias aplicações típicas de engenharia ou científicas. Se a comunidade de usuários fosse principalmente formada por engenheiros de desenvolvimento de software, os melhores benchmarks talvez incluissem aplicações como um compilador ou um sistema de processamento de documentos. Usar aplicações reais como benchmarks torna muito mais difícil encontrar meios simples de acelerar a execução do benchmark. Além disso, quando se encontram técnicas para melhorar o desempenho, essas técnicas têm muito mais chances de ajudar outros programas além do benchmark.

O uso dos benchmarks cujos desempenhos dependem de segmentos de código muito pequenos incentiva otimizações na arquitetura ou no compilador que visa a esses segmentos. As otimizações de compilador podem reconhecer fragmentos de código especiais e gerar uma seqüência de instruções

que seja eficiente para esse fragmento. Da mesma forma, um projetista poderia tentar fazer com que uma seqüência de instruções fosse executada especialmente rápido pelo fato de essa seqüência ocorrer em um benchmark. Na verdade, várias empresas introduziram compiladores com otimizações especiais visando a benchmarks específicos. Essas otimizações devem ser habilitadas explicitamente por meio de opções de compilação especiais, as quais não seriam usadas na compilação de outros programas. Se um programa de aplicação real usasse essas opções, não haveria como ter certeza de que o compilador produziria um bom código ou mesmo um código *correto*.

Algumas vezes, no afã de produzir código altamente otimizado para benchmarks, os engenheiros introduzem otimizações errôneas. Por exemplo, no final de 1995, a Intel publicou uma nova avaliação de desempenho para os benchmarks de inteiros SPEC rodando em um processador Pentium e usando um compilador interno, não usado fora da Intel.

Infelizmente, o código produzido para um dos benchmarks estava errado, fato descoberto quando um concorrente examinou o binário para entender como a Intel havia acelerado de maneira tão drástica um dos programas no pacote do benchmark. Em janeiro de 1996, a Intel admitiu o erro e reavaliou o desempenho. Programas pequenos ou que gastam quase todo o seu tempo de execução em um fragmento de código muito pequeno são especialmente vulneráveis a esses esforços.

Então, por que todo mundo não executa programas reais para medir desempenho? Uma razão é que os benchmarks pequenos são atraentes no início de um projeto, já que são pequenos o bastante para compilar e simular facilmente, às vezes manualmente. Eles são tentadores quando os projetistas estão trabalhando em um novo computador porque os compiladores podem não estar disponíveis até muito mais tarde no projeto. Embora o uso desses pequenos benchmarks no início do projeto possa ser justificado, não há um raciocínio válido para usá-los com o objetivo de avaliar sistemas computacionais funcionais.

Como mencionamos anteriormente, diferentes classes e aplicações de computadores exigirão diferentes tipos de benchmarks. Para computadores desktop, os benchmarks mais comuns são medidas de desempenho de CPU ou benchmarks focalizando uma tarefa específica, como a reprodução de um DVD ou desempenho gráfico para jogos. Na Seção 4.4, examinaremos os benchmarks SPEC CPU, que focalizam o desempenho da CPU e medem o tempo de resposta para completar um benchmark. Para servidores, a decisão de qual benchmark usar depende em grande parte da natureza da aplicação pretendida. Para servidores científicos, normalmente são usados benchmarks orientados para CPU com aplicações científicas, e o tempo de resposta para completar o benchmark é a métrica. Para outros ambientes de servidor, em geral são usados benchmarks de serviços Web, compartilhamento de arquivos e bancos de dados. Normalmente, esses benchmarks de servidor enfatizam a vazão, embora com possíveis requisitos no tempo de resposta para eventos individuais, como uma consulta a um banco de dados ou solicitação de página Web. A Seção 4.4 examina o benchmark SPECweb99, projetado para testar desempenho de servidores Web. Na computação embutida, os bons benchmarks são muito mais raros. Muitas vezes, os clientes usam suas aplicações embutidas específicas ou segmentos delas para fins de benchmark. O principal pacote de benchmark desenvolvido para computadores embutidos é o EEMBC.

Uma vez selecionado um conjunto de benchmarks adequado e obtidas as medições de desempenho, podemos escrever um relatório de desempenho. O princípio básico para o relatório das medições de desempenho deve ser a *reprodutibilidade* – isto é, devemos listar tudo que outro experimentador precisaria para duplicar os resultados. Essa lista precisa incluir a versão do sistema operacional, os compiladores e a entrada, bem como a configuração do computador. Como exemplo, a Figura 4.3 mostra a seção de descrição de sistema de um relatório do benchmark SPEC CPU2000.

Um elemento importante da reprodutibilidade é a escolha da entrada. Diferentes entradas podem gerar comportamentos bastante diferentes. Por exemplo, uma entrada pode disparar certos caminhos de execução que podem ser típicos ou pode usar partes de uma aplicação raramente usadas e, portanto, menos importantes. Alguns dos efeitos mais importantes do conjunto de entrada estão no sistema de memória. Conjuntos de entrada maiores tendem a exigir mais do sistema de memória, e o uso de workloads realisticamente dimensionados em servidores para aplicações comerciais e científicas é vital se um benchmark possui o objetivo de prever o que as aplicações reais podem ver.

Hardware	
Fornecedor de hardware	Dell
Número do modelo	Precision Workstation 360 (Pentium 4 Extreme Edition de 3.2GHz)
CPU	Intel Pentium 4 (800MHz barramento de sistema)
CPU MHz	3200
FPU	Integrado
CPU(s) habilitada(s)	1
CPU(s) possível(is)	1
Em paralelo	No
Cache primária	12K(I) micro-ops + 8KB(D) no chip
Cache secundária	512KB(I+D) no chip
Cache L3	2048KB(I+D) no chip
Outro cache	N/A
Memória	4 × 512MB ECC DDR400 SDRAM CL3
Subsistema de disco	1x 80GB ATA/100 7200 RPM
Outro hardware	

Software	
Sistema operacional	Windows XP Professional SP1
Compilador	Intel C++ Compiler 7.1 (200304022) Microsoft Visual Studio.NET (7.0.9466) MicroQuill SmartHeap Library 6.01
Tipo de sistema de arquivos	NTFS
Estado do sistema	Padrão

FIGURA 4.3 Descrição de sistema de um sistema desktop usando o Pentium 4 mais rápido disponível em 2003. Além dessa descrição obrigatória e formatada, há 23 linhas de notas descrevendo definições de flags especiais usados para portabilidade (4), otimização (2), tuning (12), temporização básica (2), uma biblioteca especial (2) e configuração de BIOS (1).

Comparando e resumindo o desempenho

Uma vez selecionados os programas a serem usados como benchmarks e depois de concordarmos sobre se estamos medindo o tempo de resposta ou a vazão, você poderia pensar que a comparação de desempenho seria muito simples. Entretanto, ainda precisamos decidir como resumir o desempenho de um grupo de benchmarks. Embora resumir um conjunto de medições resulte em menos informações, os marqueteiros e mesmo os usuários normalmente preferem ter um único número para comparar o desempenho. A questão-chave é “Como um resumo deve ser calculado?”. A Figura 4.4, retirada de um artigo sobre resumo de desempenho, ilustra as dificuldades envolvidas nesses esforços.

	Computador A	Computador B
Programa 1 (segundos)	1	10
Programa 2 (segundos)	1000	100
Tempo total (segundos)	1001	110

FIGURA 4.4 Tempos de execução de dois programas em dois computadores diferentes. Retirada da Figura 1 de Smith [1998].

Usando nossa definição de *mais rápido*, as seguintes afirmativas são válidas para as medições dos programas na Figura 4.4:

- A é 10 vezes mais rápido do que B para o programa 1.
- B é 10 vezes mais rápido do que A para o programa 2.

Individualmente, cada uma dessas afirmativas é verdadeira. Coletivamente, no entanto, apresentam um quadro confuso – o desempenho relativo dos computadores A e B não é claro.

Tempo de execução total: uma medida de resumo consistente

O método mais simples para resumir desempenho relativo é usar o tempo de execução total dos dois programas. Assim,

$$\frac{\text{Desempenho}_B}{\text{Desempenho}_A} = \frac{\text{Tempo de execução}_A}{\text{Tempo de execução}_B} = \frac{1001}{110} = 9,1$$

Ou seja, B é 9,1 vezes mais rápido do que A para os programas 1 e 2 juntos.

Esse resumo é diretamente proporcional ao tempo de execução, nossa medida final de desempenho. Se o workload consistisse em executar os programas 1 e 2 um mesmo número de vezes, essa afirmativa preveria os tempos de execução relativos para o workload em cada computador.

A média dos tempos de execução que é diretamente proporcional ao tempo de execução total é a **média aritmética (MA)**:

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

média aritmética A média dos tempos de execução diretamente proporcional ao tempo de execução total.

onde Tempo_i é o tempo de execução para o i -ésimo programa de um total de n no workload. Como ele é a média dos tempos de execução, uma média menor indica um tempo de execução médio menor e, portanto, um melhor desempenho.

A média aritmética é proporcional ao tempo de execução, considerando que cada programa no workload é executado um mesmo número de vezes. Este é o workload correto? Se não, podemos atribuir um peso w_i – para cada programa a fim de indicar a freqüência do programa nesse workload. Se, por exemplo, 20% das tarefas no workload fossem do programa 1 e 80% das tarefas no workload fossem do programa 2, então, os pesos seriam 0,2 e 0,8. Somando os produtos dos pesos pelos tempos de execução, podemos obter um quadro mais claro do desempenho do workload. Essa soma é chamada de **média aritmética ponderada**. Um método de ponderar programas é escolher pesos de modo que o tempo de execução de cada benchmark seja igual no computador usado como a base. A média aritmética padrão é um caso especial da média aritmética ponderada em que todos os pesos são iguais. Exploraremos a média ponderada em mais detalhes nos Exercícios 4.15 e 4.16.

média aritmética ponderada Uma média do tempo de execução de um workload com pesos destinados a refletir a presença dos programas em um workload; calculada como a soma dos produtos dos pesos pelos tempos de execução.

1. Suponha que você esteja escolhendo entre quatro computadores desktop diferentes: um é um Apple Macintosh e os outros três são computadores compatíveis com o PC que usam um processador Pentium 4, um processador AMD (usando o mesmo compilador do Pentium 4) e um processador Pentium 5 (que não existia ainda quando este livro foi escrito, em 2004, mas possui a mesma arquitetura do Pentium 4 e usa o mesmo compilador). Qual das seguintes afirmativas são verdadeiras?

Verifique você mesmo

- O computador mais rápido será o que tem a velocidade de clock mais alta.
- Como todos os PCs usam o mesmo conjunto de instruções compatível com Intel e executam o mesmo número de instruções para um programa, o PC mais rápido será o que tem a velocidade de clock mais alta.
- Como o AMD usa técnicas diferentes do Intel para executar instruções, eles podem ter diferentes CPIs. Entretanto, você ainda pode saber qual dos dois PCs baseados no Pentium é o mais rápido examinando a velocidade de clock.
- Somente olhando os resultados dos benchmarks para tarefas semelhantes ao seu workload é que se pode ter um quadro preciso do provável desempenho.

2. Considere as seguintes medições do tempo de execução:

Programa	Computador A	Computador B
1	2 segundos	4 segundos
2	5 segundos	2 segundos

Qual das seguintes afirmativas é verdadeira?

- a. A é mais rápido do que B para o programa 1.
- b. A é mais rápido do que B para o programa 2.
- c. A é mais rápido do que B para um workload com o mesmo número de execuções dos programas 1 e 2.
- d. A é mais rápido do que B para um workload com o programa 1 tendo o dobro das execuções do programa 2.

4.4

Vida real: dois benchmarks SPEC e o desempenho dos recentes processadores Intel

O SPEC (System Performance Evaluation Corporation) é um esforço fundado e patrocinado por diversos fornecedores de computador com a finalidade de criar conjuntos de benchmarks padronizados para sistemas computacionais modernos. Ele começou em 1989 focalizando o benchmark de estações de trabalho e servidores usando benchmarks de alto uso de CPU. (A Seção 4.7 descreve a história de maneira mais detalhada.) Hoje, o SPEC oferece uma dúzia de conjuntos de benchmark diferentes projetados para testar uma ampla variedade de ambientes computacionais usando aplicações reais e regras de execução e requisitos de relatório rigidamente especificados. Os conjuntos de benchmark SPEC incluem benchmarks para desempenho de CPU, gráficos, computação de alto desempenho, computação orientada a objetos, aplicações Java, modelos cliente-servidor, sistemas de e-mail, sistemas de arquivo e servidores Web. Nesta seção, examinaremos o desempenho de vários sistemas computacionais Dell que usam processadores Pentium III e Pentium 4 usando um benchmark de desempenho de CPU e um benchmark para sistemas orientados para a Web.

benchmark SPEC (system performance evaluation corporation): Um conjunto de benchmarks de alto uso de CPU, de inteiros e ponto flutuante, baseado em programas reais.

Desempenho com benchmarks SPEC CPU

A última versão dos benchmarks SPEC CPU é o pacote SPEC CPU2000, que consiste em 12 programas de inteiros e 14 programas de ponto flutuante, como mostra a Figura 4.5. Os benchmarks SPEC CPU são projetados para medir desempenho de CPU, embora o tempo de relógio seja a medição informada. Resumos separados são informados para os conjuntos de benchmark de ponto flutuante. As medições de tempo de execução são primeiro normalizadas dividindo-se o tempo de execução em uma Sun Ultra 5_10 com um processador de 300MHz pelo tempo de execução no computador medido; essa normalização resulta em uma medida, chamada *razão SPEC*, que tem a vantagem de que resultados numéricos maiores indicam melhor desempenho (ou seja, a razão SPEC é o inverso do tempo de execução). Uma medição de resumo CINT2000 ou CFP2000 é obtida tomando as médias geométricas das razões SPEC.

Para um determinado conjunto de instruções, os aumentos no desempenho da CPU podem vir de três origens:

1. Aumentos na velocidade de clock
2. Melhorias na organização do processador que diminuem o CPI
3. Aprimoramentos do compilador que diminuem a contagem de instruções ou geram instruções com uma média de CPI mais baixa (por exemplo, usando instruções mais simples)

Benchmarks de inteiros		Benchmarks de ponto flutuante	
Nome	Descrição	Nome	Tipo
gzip	Compactação	Wupwise	Cromodinâmica Quântica
vpr	Posicionamento e roteamento de circuitos FPGA	Swim	Modelo de água rasa
gcc	O compilador C Gnu	Mgrid	Solver multigrade no campo 3D potencial
mcf	Otimização combinatória	Applu	Equação diferencial parcial parabólica/elíptica
crafty	Programa de xadrez	Mesa	Biblioteca de gráficos tridimensionais
parser	Programa de processamento de textos	Galgel	Dinâmica computacional de fluidos
eon	Visualização por computador	Art	Reconhecimento de imagem usando redes neurais
peribmk	Aplicação Perl	Equake	Simulação de propagação de ondas sísmicas
gap	Teoria de grupo, interpretador	Facerec	Reconhecimento de imagem de rostos
vortex	Banco de dados orientado a objetos	Ammp	Química computacional
bzip2	Compactação	Lucas	Teste de primalidade
twolf	Simulador de posicionamento e roteamento de circuitos	Fma3d	Simulação de choque usando método de elementos finitos
		Sixtrack	Projeto de acelerador de física nuclear de alta energia
		Apsi	Meteorologia: distribuição de poluentes

FIGURA 4.5 Os benchmarks SPEC CPU2000. Os 12 benchmarks de inteiros no lado esquerdo da tabela são escritos em C e C++, enquanto os benchmarks de ponto flutuante no lado direito são escritos em Fortran (77 ou 90) e em C. Para obter mais informações sobre o SPEC e os benchmarks SPEC, veja www.spec.org. Os benchmarks SPEC CPU usam o tempo de relógio como métrica, mas, como há pouca E/S, eles medem o desempenho da CPU.

Para ilustrar essas melhorias de desempenho, a Figura 4.6 mostra as medições SPEC CINT2000 e CFP2000 para uma série de processadores Intel Pentium III e Pentium 4 realizadas usando os computadores desktop Dell Precision. Como o SPEC requer que os benchmarks sejam executados em hardware real e que o sistema de memória tenha um efeito significativo no desempenho, outros sistemas

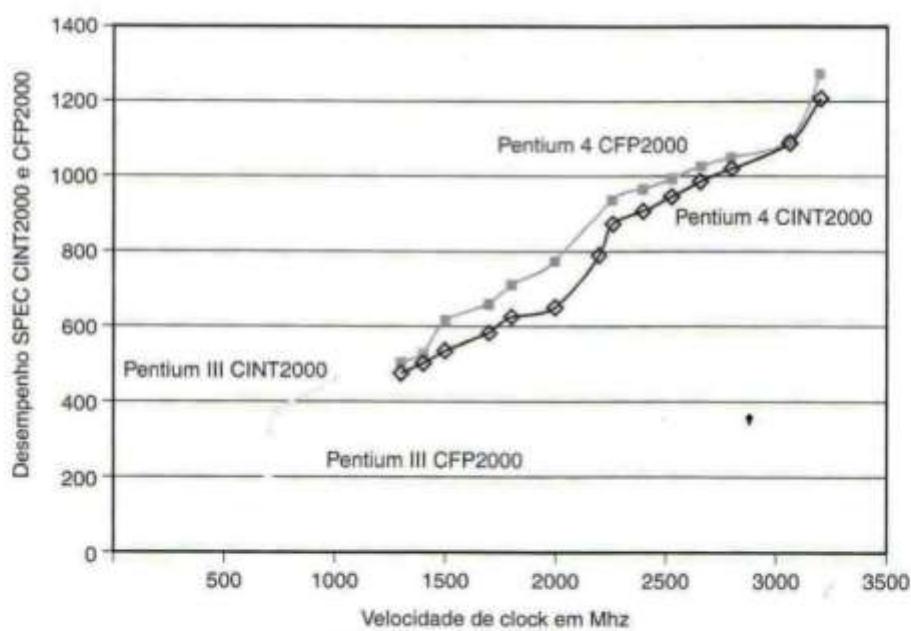


FIGURA 4.6 Avaliações SPEC CINT2000 e CFP2000 para os processadores Intel Pentium III e Pentium 4 em diferentes velocidades de clock. O SPEC exige dois conjuntos de medições: um que permita otimização agressiva com opções específicas do benchmark e um que permita apenas as opções de otimização padrão (chamadas medições "de base"). Esse gráfico contém medições de base; para os benchmarks de inteiros, as diferenças são pequenas. Para obter mais detalhes sobre o SPEC, veja www.spec.org. O Pentium 4 Extreme (uma versão do Pentium 4 introduzida no final de 2003) não está incluído nesses resultados, já que usa uma arquitetura de cache diferente dos outros processadores Pentium 4.

mas com esses processadores podem produzir níveis de desempenho diferentes. Além das diferenças na velocidade de clock, o Pentium III e o Pentium 4 usam diferentes estruturas de pipeline, que descrevemos em mais detalhes no Capítulo 6.

Existem várias observações importantes que podemos tirar desses dois gráficos de desempenho. Primeiro, repare que o desempenho de cada um desses processadores aumenta quase linearmente com os aumentos de velocidade de clock. Muitas vezes, esse não é o caso, já que as perdas no sistema de memória, discutidas no Capítulo 7, normalmente aumentam com velocidades de clock mais altas. O melhor desempenho desses processadores se deve aos sistemas de cache agressivos usados nesses processadores e à incapacidade de muitos dos benchmarks SPEC em exaurir esse sistema de memória.

Comparar os desempenhos do Pentium III e Pentium 4 fornece observações ainda mais interessantes. Em especial, repare as posições relativas das curvas de CINT2000 e CFP2000 para o Pentium III em comparação com o Pentium 4. Pode-se deduzir que o Pentium 4 é relativamente melhor do que o Pentium III nos benchmarks de ponto flutuante ou relativamente pior nos benchmarks de inteiros. Mas qual é o caso?

O Pentium 4 usa uma tecnologia de circuito integrado mais avançada, bem como uma estrutura de pipeline mais agressiva, ambos favorecendo um aumento significativo na velocidade de clock. Uma comparação particularmente interessante é a medição do SPEC CINT2000 e do SPEC CFP2000 dividida pela velocidade de clock em MHz. A tabela a seguir resume o valor médio dessa velocidade usando diferentes velocidades de clock para cada processador:

Velocidade	Pentium III	Pentium 4
CINT2000 / Velocidade de clock em MHz	0,47	0,36
CFP2000 / Velocidade de clock em MHz	0,34	0,39

Algumas vezes, métricas como desempenho de benchmark dividido pela velocidade de clock são consideradas medições da eficiência de implementação, embora, como já vimos, não se possa separar a interação da velocidade de clock das outras melhorias.

Essas medições são especialmente interessantes devido às diferenças entre os benchmarks de inteiros e de ponto flutuante. As taxas de desempenho de CINT2000 são típicas: quando uma versão mais rápida de um processador é introduzida, ela pode sacrificar um dos aspectos de um projeto (como o CPI) para melhorar outro (como a velocidade de clock). Considerando um único compilador para ambos os processadores – e, portanto, código idêntico –, as taxas de CINT2000 nos dizem que o CPI do Pentium 4 é 1,3 vez o CPI do Pentium III (0,47/0,36).

Então, como esses números podem ser revertidos para os benchmarks de ponto flutuante? A resposta é que o Pentium 4 oferece um conjunto de novas instruções (chamado Streaming SIMD Extensions 2; veja o Capítulo 3), que fornece uma melhoria significativa para o ponto flutuante. Portanto, a contagem de instruções e o CPI para o Pentium 4 irão diferir do Pentium III, produzindo um desempenho melhor.

SPECweb99: um benchmark de vazão para servidores Web

Em 1996, o SPEC introduziu seu primeiro benchmark projetado para medir desempenho de servidores Web; o benchmark foi superado por uma nova versão em 1999. O SPECweb99 difere do SPEC CPU em diversos aspectos:

- O SPECweb99 focaliza a vazão, medindo o número máximo de conexões que um sistema executando como um servidor Web pode suportar. O sistema precisa fornecer resposta a uma solicitação de um cliente dentro de um tempo limitado e com um número limitado de erros.
- Como o SPECweb99 mede a vazão, os multiprocessadores (sistemas com mais de uma CPU) são freqüentemente usados nos benchmarks.
- O SPECweb99 fornece apenas um programa para gerar solicitações para servidores Web; o software do servidor Web se torna parte do sistema sendo medido.

- O desempenho do SPECweb99 depende de uma ampla medida das características de sistema, incluindo o sistema de disco e a rede.

Para mostrar como essas características produzem uma descrição claramente variada do desempenho de um servidor Web, selecionamos os resultados do SPECweb99 para uma série de servidores Dell PowerEdge que usam as versões Xeon dos processadores Pentium III e Pentium 4. Os processadores Xeon são construídos usando a estrutura básica do Pentium III ou Pentium 4, mas suportam o multiprocessamento. Além disso, o Xeon MP suporta um terceiro nível de cache fora do chip e pode suportar mais de dois processadores. Os resultados para uma variedade desses sistemas Dell são mostrados na Figura 4.7.

Sistema	Processador	Número de unidades de disco	Número de CPUs	Número de redes	Velocidade de clock (GHz)	Resultado
1550/1000	Pentium III	2	2	2	1	2765
1650	Pentium III	3	2	1	1,4	1810
2500	Pentium III	8	2	4	1,13	3435
2550	Pentium III	1	2	1	1,26	1454
2650	Pentium 4 Xeon	5	2	4	3,06	5698
4600	Pentium 4 Xeon	10	2	4	2,2	4615
6400/700	Pentium III Xeon	5	4	4	0,7	4200
6600	Pentium 4 Xeon MP	8	4	8	2	6700
8450/700	Pentium III Xeon	7	8	8	0,7	8001

FIGURA 4.7 Desempenho do SPECweb9999 para diversos sistemas Dell PowerEdge usando as versões Xeon dos processadores Pentium III e Pentium 4.

Examinando os dados na Figura 4.7, podemos ver claramente que a velocidade de clock dos processadores não é o fator mais importante na determinação do desempenho do servidor Web. Na verdade, o 8450 possui processadores quase três vezes mais lentos do que o 6600 e ainda oferece um melhor desempenho. Esperamos que esses sistemas sejam configurados para alcançar o melhor desempenho. Ou seja, para um determinado conjunto de CPUs, discos e redes adicionais são acrescentados até o processador se tornar o gargalo.

Desempenho e eficiência do consumo

Como mencionarmos no Capítulo 1, o consumo está se tornando, cada vez mais, a principal limitação no desempenho do processador. No mercado de embutidos, em que muitos processadores entram em ambientes que se baseiam apenas em resfriamento passivo ou em energia de bateria, o consumo de energia freqüentemente se torna uma limitação tão importante quanto o desempenho e o custo.

Sem dúvida, muitos leitores terão encontrado limitações de consumo ao usar seus laptops. Definitivamente, entre os desafios de remover o excesso de calor e as limitações da vida da bateria, o consumo de energia se tornou um fator crucial no projeto dos processadores para laptops. A capacidade da bateria melhorou apenas ligeiramente com o tempo, sendo que as principais melhorias vieram dos novos materiais. Conseqüentemente, a capacidade do processador operar de maneira eficiente e conservar energia é fundamental. Para economizar energia, têm sido empregadas técnicas variando desde colocar partes do computador para dormir até reduzir a velocidade de clock e a voltagem. Na verdade, o consumo de energia é tão importante que a Intel projetou uma linha de processadores, a série Pentium M, especificamente para aplicações móveis à bateria.

Como discutimos no Capítulo 1, para a tecnologia CMOS, podemos reduzir o consumo reduzindo a freqüência. Por isso, todos os processadores recentes para laptop usam a capacidade de adaptar a freqüência para reduzir o consumo de energia, simultaneamente, é claro, reduzindo o desempenho.

Portanto, avaliar de maneira adequada a eficiência do consumo de um processador exige examinar seu desempenho no consumo máximo, em um nível intermediário que conserva bateria e em um nível que maximiza a vida da bateria. Nas linhas Intel Mobile Pentium e Pentium M, há duas velocidades de clock disponíveis: máxima e reduzida. O melhor desempenho é obtido executando na velocidade máxima, a melhor autonomia da bateria é obtida executando sempre na velocidade reduzida e o nível otimizado de desempenho-consumo é obtido trocando dinamicamente entre essas duas velocidades de clock.

A Figura 4.8 mostra o desempenho dos três processadores Intel Pentium projetados para serem usados em aplicações móveis rodando SPEC CINT2000 e SPEC CFP2000 como benchmarks. Como podemos ver, o processador mais novo, o Pentium M, possui o melhor desempenho quando executado na velocidade de clock total, bem como com o modo de velocidade de clock adaptativa. O clock de 600MHz do Pentium M o torna mais lento quando executado no modo de consumo mínimo do que o Pentium 4-M, mas ainda é mais rápido do que o projeto do Pentium III-M anterior.

Para aplicações com limitação de consumo, a métrica mais importante talvez seja a eficiência do consumo, calculada tomando-se o desempenho e dividindo pelo consumo médio de energia ao executar o benchmark. A Figura 4.9 mostra a eficiência de consumo relativa para os processadores executando os benchmarks SPEC2000. Esses dados mostram claramente a vantagem da eficiência de consumo do novo projeto Pentium M. Em todos os três modos, ele tem uma significativa vantagem na eficiência de consumo em relação ao Pentium 4-M. Esses dados mostram claramente a vantagem de um processador como o Pentium M, projetado desde o começo visando ao consumo reduzido de energia, diferente de um projeto como o Pentium III-M ou Pentium 4-M, que são versões modificadas dos processadores padrão. É claro que a medição adequada da eficiência de consumo também exige o uso de benchmarks adicionais projetados para refletir como os usuários empregam os computadores alimentados por bateria. Tanto as revistas sobre PCs quanto o boletim técnico da Intel realizam regularmente esses estudos.

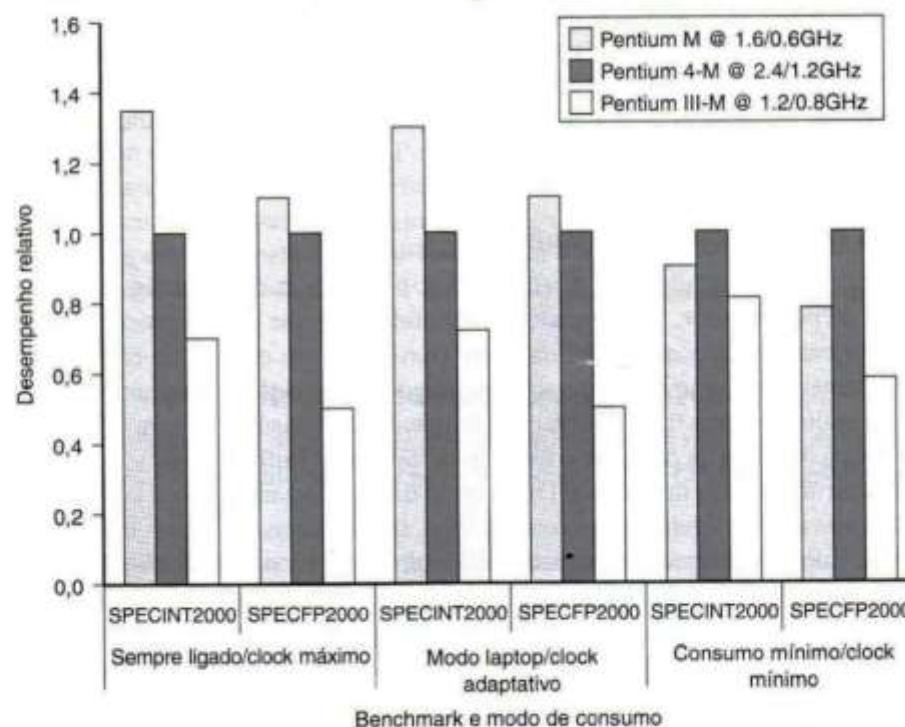


FIGURA 4.8 Desempenho relativo dos três processadores Intel em SPECINT2000 e SPECFP2000 nos três modos diferentes. Cada processador opera em duas velocidades de clock diferentes, indicadas na legenda.

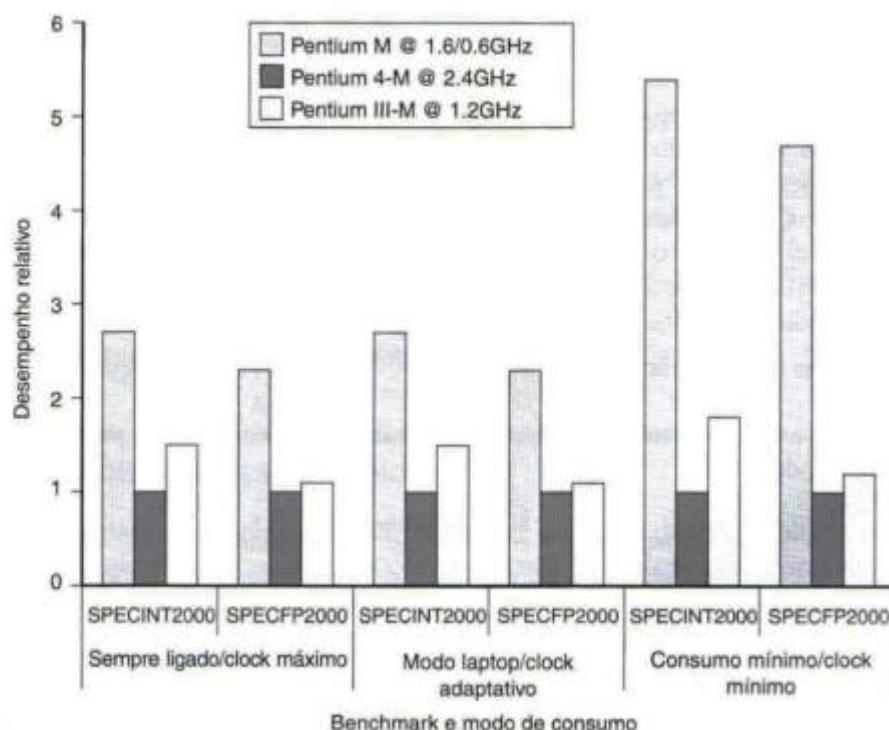


FIGURA 4.9 A eficiência relativa de consumo dos três processadores Pentium móveis executando o SPEC2000 nos três modos. A eficiência de consumo é medida como o inverso dos joules consumidos por benchmark, que é calculada dividindo o inverso do tempo de execução para um benchmark pelos watts dissipados.

Qual das seguintes configurações de Pentium III com um único processador provavelmente produzirá o melhor desempenho no SPECweb99 com base nos dados da Figura 4.7?

Verifique você mesmo

- Processador de 1,26GHz, 1 disco, 1 conexão de rede
- Processador de 1GHz, 6 discos, 3 conexões de rede
- Processador de 1,1GHz, 2 discos, 3 conexões de rede

4.5 Falácia e armadilhas

As falácias e armadilhas de custo/desempenho têm enredado muitos arquitetos de computador, inclusive a nós. Portanto, esta seção não sofre de falta de exemplos relevantes. Começamos com uma armadilha que tem apanhado muitos projetistas e revela uma importante relação no projeto de computadores.

Armadilha: esperar a melhoria de um aspecto de um computador para aumentar o desempenho por uma quantidade proporcional ao tamanho da melhoria.

Essa armadilha pegou projetistas tanto de hardware quanto de software. Um problema de projeto simples a ilustra bem. Suponha um programa executado em 100 segundos em um computador, com operações de multiplicação responsáveis por 80 segundos desse tempo. Em quanto preciso melhorar a velocidade da multiplicação se desejo que meu programa rode cinco vezes mais rápido?

Lei de Amdahl Uma regra afirmando que o aumento de desempenho possível com uma determinada melhoria é limitado pela quantidade de uso do recurso melhorado.

O tempo de execução do programa depois que eu faço a melhoria é dado pela seguinte equação simples, conhecida como **Lei de Amdahl**.

$$\text{Tempo de execução após melhoria} = \left(\frac{\text{Tempo de execução afetado pela melhoria}}{\text{Quantidade de melhoria}} + \text{Tempo de execução não-afetado} \right)$$

Para este problema:

$$\text{Tempo de execução após melhoria} = \frac{80 \text{ segundos}}{n} + (100 - 80 \text{ segundos})$$

Como queremos que o desempenho seja cinco vezes mais rápido, o novo tempo de execução deve ser de 20 segundos, dado que

$$\begin{aligned} 20 \text{ segundos} &= \frac{80 \text{ segundos}}{n} + 20 \text{ segundos} \\ 0 &= \frac{80 \text{ segundos}}{n} \end{aligned}$$

Ou seja, não há *quantidade alguma* pela qual possamos melhorar a multiplicação para conseguir um aumento de cinco vezes no desempenho, se a multiplicação for responsável por apenas 80% do workload. A melhoria de desempenho possível com determinado aumento é limitada pela quantidade em que o recurso aprimorado é usado.

Esse conceito também produz o que, na economia, é chamado de “lei dos rendimentos decrescentes”. Podemos usar a Lei de Amdahl para estimar as melhorias de desempenho quando sabemos o tempo consumido para alguma função e seu aumento de velocidade. A Lei de Amdahl, juntamente com a equação de desempenho de CPU são ferramentas úteis para avaliar as melhorias potenciais.

Um tema comum no projeto de hardware é uma consequência da Lei de Amdahl: *torne o caso comum rápido*. Essa regra simples nos lembra que, em muitos casos, a frequência na qual um evento ocorre pode ser muito mais alta do que outro. A Lei de Amdahl lembra que a oportunidade para a melhoria é afetada pela quantidade de tempo que o evento consome. Portanto, tornar o caso comum mais rápido tenderá a aumentar o desempenho melhor do que otimizar o caso raro. Ironicamente, o caso comum costuma ser mais simples do que o caso raro e, portanto, é mais fácil de melhorar.

Armadilha: usar um subconjunto da equação de desempenho como uma métrica de desempenho.

Já mostramos a falácia de prever o desempenho baseado simplesmente na velocidade de clock, contagem de instruções ou CPI. Outro erro comum é usar dois dos três fatores para comparar o desempenho. Embora essa prática possa ser válida em um contexto limitado, ela também é facilmente mal-utilizada. Na verdade, quase todas as alternativas propostas para usar o tempo como métrica de desempenho posteriormente levaram a afirmações falsas, resultados distorcidos ou interpretações incorretas.

Uma alternativa para o tempo como métrica é o **MIPS (milhão de instruções por segundo)**. Para um determinado programa, o MIPS é simplesmente

$$\text{MIPS} = \frac{\text{Contador de instruções}}{\text{Tempo de execução} \times 10^6}$$

Essa medição de MIPS também é chamada de *MIPS nativo* para distingui-la de algumas definições alternativas de MIPS que discutimos na Seção 4.8, no CD.

Como o MIPS é uma taxa de execução de instruções, ele especifica o desempenho de maneira inversa ao tempo de execução; computadores mais rápidos possuem um índice MIPS mais alto. A boa

MIPS (milhão de instruções por segundo) Uma medição da velocidade de execução de um programa baseado no número de milhões de instruções. O MIPS é calculado como a contagem de instruções dividida pelo produto do tempo de execução e 10^6 .

notícia sobre o MIPS é que ele é fácil de ser entendido e computadores mais rápidos significam MIPS maiores, o que segue a intuição natural.

Há três problemas com o uso do MIPS como uma medida para comparar computadores. Primeiro, o MIPS especifica a taxa de execução de instruções, mas não leva em conta as capacidades das instruções. Não podemos comparar computadores com diferentes conjuntos de instruções usando o MIPS, já que as contagens de instruções certamente irão diferir. Em nosso exemplo anterior examinando o desempenho SPEC CFP2000, usar o MIPS para comparar o desempenho do Pentium III com o do Pentium 4 produziria resultados enganadores. Segundo, o MIPS varia entre programas no mesmo computador; portanto, um computador não pode ter um único índice MIPS para todos os programas. Por fim, e mais importante, o MIPS pode variar inversamente com o desempenho! Existem muitos exemplos desse comportamento anômalo e um deles é dado abaixo.

MIPS COMO MEDIDA DE DESEMPEÑHO

Considere o computador com três classes de instrução e medições de CPI do último exemplo da página 191. Agora suponha que, medindo o código gerado por dois compiladores diferentes para o mesmo programa, obtivemos os seguintes dados:

Código do	Contagens de instruções (em bilhões) para cada classe de instrução		
	A	B	C
Compilador 1	5	1	1
Compilador 2	10	1	1

Considere que a velocidade de clock do computador seja de 4GHz. Que seqüência de código será executada mais rápido de acordo com o MIPS? E de acordo com o tempo de execução?

Primeiro, encontramos o tempo de execução para os dois compiladores diferentes usando a seguinte equação:

$$\text{Tempo de execução} = \frac{\text{Ciclos de clock da CPU}}{\text{Velocidade de clock}}$$

Podemos usar uma fórmula anterior para os ciclos de clock da CPU:

$$\text{Ciclos de clock da CPU} = \sum_{i=1}^n (\text{CPI}_i \times C_i)$$

$$\text{Ciclos de clock da CPU}_1 = (5 \times 1 + 1 \times 2 + 1 \times 3) \times 10^9 = 10 \times 10^9$$

$$\text{Ciclos de clock da CPU}_2 = (10 \times 1 + 1 \times 2 + 1 \times 3) \times 10^9 = 15 \times 10^9$$

Agora, encontramos o tempo de execução para os dois compiladores:

$$\text{Tempo de execução}_1 = \frac{10 \times 10^9}{4 \times 10^9} = 2,5 \text{ segundos}$$

$$\text{Tempo de execução}_2 = \frac{15 \times 10^9}{4 \times 10^9} = 3,75 \text{ segundos}$$

Portanto, concluímos que o compilador 1 gera o programa mais rápido, de acordo com o tempo de execução. Agora, vamos calcular o índice de MIPS para cada versão do programa, usando

$$\text{MIPS} = \frac{\text{Contagem de instruções}}{\text{Tempo de execução} \times 10^6}$$

EXEMPLO

RESPOSTA

$$\text{MIPS}_1 = \frac{(5+1+1) \times 10^9}{2,5 \times 10^6} = 2800$$

$$\text{MIPS}_2 = \frac{(10+1+1) \times 10^9}{3,75 (30) \times 10^6} = 3200$$

Assim, o código do compilador 2 possui um índice de MIPS mais alto, mas o código do compilador 1 é executado mais rápido!

Como esse exemplo mostra, o MIPS pode falhar em fornecer um quadro verdadeiro do desempenho – mesmo comparando duas versões do mesmo programa no mesmo computador. Na Seção 2.7, discutimos outros usos do termo *MIPS*, e como esses usos também podem ser enganadores.

Verifique você mesmo

Considere as seguintes medições de desempenho para um programa:

Medida	Computador A	Computador B
Contagem de instruções	10 bilhões	8 bilhões
Velocidade de clock	4GHz	4GHz
CPI	1,0	1,1

- Qual computador possui o índice MIPS mais alto?
- Qual computador é o mais rápido?

4.6 Comentários finais

Embora, neste capítulo, tenhamos focalizado o desempenho e como avaliá-lo, projetar apenas para o desempenho sem considerar o custo, a funcionalidade e outros requisitos é uma atitude irrealista. Todos os projetistas de computador precisam equilibrar o desempenho e o custo. Naturalmente, existe um domínio do *projeto de alto desempenho*, em que o desempenho é o principal objetivo e o custo é secundário. Uma grande parte da indústria de supercomputadores e servidores de topo de linha projeta dessa maneira. No outro extremo, está o *projeto de baixo custo*, caracterizado pelo mercado de embutidos, no qual o custo e o consumo de energia têm prioridade sobre o desempenho. Entre esses dois pontos, estão os projetos de desktop e os servidores de pouca capacidade; esses computadores exigem um *projeto de custo/desempenho*, em que o projetista equilibra o custo com o desempenho. Os exemplos da indústria de computadores desktop caracterizam os tipos de compensações com que os projetistas nessa área precisam conviver. Neste capítulo, vimos que existe um método seguro de determinar e informar o desempenho, usando o tempo de execução de programas reais como métrica. Esse tempo de execução está relacionado com outras medições importantes que podemos fazer pela seguinte equação:

$$\frac{\text{Segundos}}{\text{Programa}} = \frac{\text{Instruções}}{\text{Programa}} \times \frac{\text{Ciclos de clock}}{\text{Instrução}} \times \frac{\text{Segundos}}{\text{Ciclo de clock}}$$

Usaremos muitas vezes essa equação e seus fatores constituintes. Contudo, lembre-se de que, individualmente, os fatores não determinam o desempenho: apenas o produto, que é igual ao tempo de execução, é uma medida segura do desempenho.

Colocando em perspectiva

O tempo de execução é a única medida válida e infalível de desempenho. Muitas outras métricas foram propostas e consideradas deficientes. Algumas vezes, essas métricas falham desde o início por não refletirem o tempo de execução: outras vezes, uma métrica válida *em um contexto limitado* é estendida e usada além desse contexto ou sem o esclarecimento adicional necessário para torná-la válida.

Da mesma forma, qualquer medida que resuma o desempenho deve refletir o tempo de execução. A média aritmética ponderada resume o desempenho enquanto monitora o tempo de execução. Pelo uso dos pesos, uma média aritmética ponderada pode se ajustar para diferentes tempos de execução, equilibrando a contribuição de cada benchmark no resumo.

É claro que simplesmente conhecer essa equação não é o bastante para guiar o projeto ou a avaliação de um computador. Precisamos entender como os diferentes aspectos de um projeto afetam cada um desses parâmetros-chave. Esse conhecimento envolve uma ampla variedade de questões, desde a eficiência do compilador até os efeitos do projeto do conjunto de instruções na contagem de instruções, o impacto do pipelining e dos sistemas de memória no CPI e a interação entre a tecnologia e a organização que determinam a velocidade de clock. A arte do projeto e da avaliação de computadores reside não em jogar números em uma equação de desempenho, mas em determinar corretamente como as alternativas afetarão o desempenho e o custo.

A maioria dos usuários de computadores se importa com o custo e também com o desempenho. Embora seja difícil entender a relação entre os aspectos de um projeto e o seu desempenho, determinar o custo dos vários recursos de projeto freqüentemente é um problema ainda mais difícil. O custo de um computador é afetado não só pelo custo dos componentes, mas pelo custo do trabalho para montar o computador, da pesquisa e do desenvolvimento, das vendas e marketing e da margem de lucro. Finalmente, devido à rápida mudança nas tecnologias de implementação, a escolha mais barata de hoje em geral é menos que ideal em seis meses ou um ano.

Os projetos de computador sempre serão medidos pelo custo e desempenho, bem como por outros fatores importantes como consumo, segurança, custo de propriedade e escalabilidade. Embora este capítulo tenha focalizado o desempenho, os melhores projetos alcançarão o equilíbrio apropriado para um certo mercado entre todos esses fatores.

4.7**Perspectiva histórica e leitura adicional**

Esta seção, que examina a história da medição de desempenho e do uso de benchmarks, aparece na Seção 4.7 no CD.

4.8**Exercícios**

4.1 [5] <§4.1> Queremos comparar o desempenho de dois computadores diferentes: M1 e M2. As seguintes medições foram feitas nesses computadores:

Programa	Tempo em M1	Tempo em M2
1	2,0 segundos	1,5 segundo
2	5,0 segundos	10,0 segundos

Que computador é mais rápido para cada programa, e o quanto é mais rápido?

- 4.2** [5] <§4.1> Considere os dois computadores e programas do Exercício 4.1. As seguintes medições adicionais foram feitas:

Programa	Instruções executadas em M1	Instruções executadas em M2
1	5×10^9	6×10^9

Encontre a taxa de execução de instruções (instruções por segundo) para cada computador quando está executando o programa 1.

- 4.3** [5] <§4.1> Suponha que M1 no Exercício 4.1 custe US\$500 e M2 custe US\$800. Se precisasse executar o programa 1 um grande número de vezes, qual computador você compraria em grandes quantidades? Por quê?

- 4.4** [10] <§4.1> ■ Aprofundando o aprendizado: Computação econômica

- 4.5** [5] <§4.1> ■ Aprofundando o aprendizado: Computação econômica

- 4.6** [5] <§4.1> Outro usuário possui as seguintes necessidades para os computadores discutidas no Exercício 4.1: o P1 precisa ser executado 1.600 vezes a cada hora. Qualquer tempo restante é usado para executar o P2. Se o computador possui desempenho suficiente para executar o programa 1 a quantidade necessária de vezes por hora, então, o desempenho é medido pela vazão para o programa 2. Que computador é mais rápido para esse workload? Que computador é mais econômico?

- 4.7** [10] <§4.2> Suponha que você deseje executar um programa P com $7,5 \times 10^9$ instruções em uma máquina de 5GHz com um CPI de 0,8.

- Qual é o tempo de CPU esperado?
- Quando você executa P, ele leva 3 segundos de tempo de relógio para ser concluído. Qual é a porcentagem do tempo de CPU que P recebeu?

- 4.8** [10] <§4.2> Considere duas implementações diferentes, P1 e P2, do mesmo conjunto de instruções. Há cinco classes de instruções (A, B, C, D e E) em cada conjunto de instruções.

P1 tem uma velocidade de clock de 4GHz. P2 tem uma velocidade de clock de 6GHz. O número de ciclos médio para cada classe de instruções para P1 e P2 é o seguinte:

Classe	CPI em P1	CPI em P2
A	1	2
B	2	2
C	3	2
D	4	4
E	3	4

Considere que o desempenho de pico seja definido como a maior velocidade de clock na qual um computador pode executar qualquer sequência de instruções. Quais são os desempenhos de pico de P1 e P2 expressos em instruções por segundo?

- 4.9** [5] <§§4.1-4.2> Se o número de instruções executadas em um certo programa é dividido igualmente entre as classes das instruções no Exercício 4.8 exceto para a classe A, que ocorre com o dobro da frequência das outras, o quanto P2 é mais rápido do que P1?

- 4.10** [12] <§4.2> Considere duas implementações diferentes, I1 e I2, do mesmo conjunto de instruções. Há três classes de instruções (A, B e C) no conjunto de instruções. A implementação I1 possui uma velocidade de clock de 6GHz e I2 possui uma velocidade de clock de 3GHz. O número médio de ciclos para cada classe de instrução em I1 e I2 é dada na seguinte tabela:

Classe	CPI em I1	CPI em I2	Uso de C1	Uso de C2	Uso de C3
A	2	1	40%	40%	50%
B	3	2	40%	20%	25%
C	5	2	20%	40%	25%

A tabela também contém um resumo da proporção média de classes de instruções geradas por três compiladores diferentes. C1 é um compilador produzido pelos fabricantes de I1, C2 é produzido pelos fabricantes de I2 e o outro compilador é um produto independente. Considere que cada compilador use o mesmo número de instruções para um determinado programa mas que o mix de instruções seja como descrito na tabela. Usando C1 em I1 e I2, quanto mais rápido os fabricantes de I1 afirmam que ele é em comparação com I2? Se comprasse I1, que compilador você usaria? Se comprasse I2, que compilador você usaria? Que computador e compilador você compraria se todos os outros critérios fossem idênticos, inclusive o custo?

4.11 [5] <§4.2> Considere o programa P, executado em uma máquina de 1GHz em 10 segundos. Uma otimização é feita em P, substituindo todas as instâncias de multiplicação de um valor por 4 (mult X, X, 4) por duas instruções que definem para $\times + \times$ duas vezes (add X, X; add X, X). Chame esse novo programa otimizado de P'. O CPI de uma instrução de multiplicação é 4 e o CPI de uma soma é 1. Após recompilar, o programa agora roda em 9 segundos na máquina M. Quantas multiplicações foram substituídas pelo novo compilador?

4.12 [5] <§4.2> Sua empresa poderia tornar um programa Java mais rápido em seu novo computador acrescentando suporte de hardware para coleta de lixo. A coleta de lixo atualmente é responsável por 20% dos ciclos do programa. Você tem duas mudanças possíveis para fazer na máquina. A primeira seria manipular automaticamente a coleta de lixo no hardware. Isso causa um aumento no tempo de ciclo por um fator de 1,2. A segunda seria fornecer novas instruções de hardware para serem acrescentadas ao ISA que seriam usadas durante a coleta de lixo. Isso reduziria pela metade o número de instruções necessárias para as coletas de lixo mas aumentaria o tempo de ciclo em 1,1. Qual dessas duas opções, se houver alguma, você escolheria?

4.13 [5] <§4.2> Para o seguinte conjunto de variáveis, identifique todos os subconjuntos que podem ser usados para calcular o tempo de execução. Cada subconjunto deve ser mínimo; isto é, não deve conter variável alguma que não seja necessária.

{CPI, velocidade de clock, tempo de ciclo, MIPS, número de instruções no programa, número de ciclos no programa}

4.14 [5] <§4.2> A tabela a seguir mostra o número de operações de ponto flutuante executadas em três programas diferentes e o tempo de execução para esses programas em três computadores diferentes:

Programa	Operações de ponto flutuante	Tempo de execução em segundos		
		Computador A	Computador B	Computador C
Programa 1	5×10^9	2	5	10
Programa 2	20×10^9	20	20	20
Programa 3	40×10^9	200	50	15

Que computador é mais rápido de acordo com o tempo de execução total? O quanto ele é mais rápido comparado com os outros dois computadores?

4.15 [15] <§§4.2, 4.3> Um usuário lhe disse que os três programas do Exercício 4.14 constituem o corpo do workload dele, mas ele não os executa igualmente. O usuário quer determinar como os três computadores são comparados quando o workload consiste em diferentes arranjos desses três programas. (Você sabe que pode usar a média aritmética para encontrar o desempenho relativo.) Suponha que o número total de operações de ponto flutuante (FLOPs) executadas no workload é dividido igualmente entre os três programas. Ou seja, o programa 1 é executado 8 vezes para cada vez que o programa 3 é executado, e o programa 2 é executado duas vezes para cada vez que o programa 3 é executado. Como isso se compara com o tempo de execução total com números iguais de execuções de programa?

4.16 [15] <§§4.2, 4.3> Uma ponderação alternativa à do Exercício 4.15 é considerar que quantidades iguais de tempo serão gastos executando cada programa em um dos computadores. Que compu-

tador é mais rápido usando os dados fornecidos no Exercício 4.14 e considerando uma ponderação que gera um tempo de execução igual para cada um dos programas de benchmark no computador A? Que computador é mais rápido se considerarmos uma ponderação que gera um tempo de execução igual no computador B? E no computador C? Explique os resultados.

4.17 [5] <§§4.2-4.3> Se as velocidades de clock dos computadores M1 e M2 no Exercício 4.1 forem 4GHz e 6GHz respectivamente, encontre o CPI (ciclos por instrução) para o programa 1 em ambos os computadores usando os dados dos Exercícios 4.1 e 4.2.

4.18 [5] <§§4.2-4.3> Considerando que o CPI para o programa 2 em cada computador do Exercício 4.1 seja o mesmo do programa 1 encontrado no Exercício 4.17, encontre a contagem de instruções para o programa 2 executando em cada computador usando os tempos de execução do Exercício 4.1.

4.19 [10] <§§4.2, 4.3> ■ **Aprofundando o aprendizado:** Analisando um processador com ponto flutuante implementado no hardware ou software

4.20 [10] <§§4.2, 4.3> ■ **Aprofundando o aprendizado:** Analisando um processador com ponto flutuante implementado no hardware ou software

4.21 [5] <§§4.2, 4.3> ■ **Aprofundando o aprendizado:** Analisando um processador com ponto flutuante implementado no hardware ou software

4.22 [10] <§§4.2, 4.3> ■ **Aprofundando o aprendizado:** Analisando melhorias em um processador

4.23 [5] <§§4.2, 4.3> ■ **Aprofundando o aprendizado:** Analisando melhorias em um processador

4.24 [10] <§§4.2, 4.3> ■ **Aprofundando o aprendizado:** Analisando melhorias em um processador

4.25 [5] <§§4.2, 4.3> ■ **Aprofundando o aprendizado:** Analisando melhorias em um processador

4.26 [5] <§§4.2, 4.3> ■ **Aprofundando o aprendizado:** Analisando melhorias em um processador

4.27 [10] <§§4.2, 4.3> ■ **Aprofundando o aprendizado:** Analisando melhorias em um processador

4.28 [10] <§§4.2, 4.3> ■ **Aprofundando o aprendizado:** Analisando melhorias em um processador

4.29 [5] <§4.3> Considere que as instruções de multiplicação gastem 12 ciclos e sejam responsáveis por 15% das instruções em um programa típico, e os outros 85% das instruções exijam uma média de 4 ciclos para cada instrução. Que porcentagem do tempo a CPU gasta realizando a multiplicação?

4.30 [5] <§4.3> Sua equipe de engenharia de hardware indicou que seria possível reduzir o número de ciclos necessários para multiplicação para 8 no Exercício 4.29, mas isso exigiria um aumento de 20% no tempo de ciclo. Nada mais será afetado pela mudança. Eles devem proceder à modificação?

4.31 [10] <§4.4> Veja a lista atual dos programas SPEC na Figura 4.5. Ela inclui aplicações que correspondam às maneiras com que você normalmente usa seu computador? Que classes de programas são irrelevantes ou estão ausentes? Por que você acha que elas foram ou não foram incluídas no SPEC? O que precisaria ser feito para incluir/excluir esses programas na próxima versão do SPEC?

4.32 [5] <§4.4> Se os pacotes de benchmark foram projetados para fornecer uma métrica real para uma tarefa de computação específica, explique por que os pacotes de benchmark precisam ser atualizados.

4.33 [10] <§§4.1-4.5> Considere a seguinte notícia hipotética:

"A empresa irá revelar a primeira versão da indústria para o chip de 5GHz, que oferece um aumento de desempenho de 25% sobre o antigo campeão de velocidade da empresa, que roda em 4GHz. O novo chip pode ser conectado nas placas de sistema para o chip original mais antigo (que rodava em 1GHz) para fornecer um ganho de desempenho de 70%."

Comente sobre a definição (ou definições) de desempenho que acredita que a empresa usou. Você acha que a nota é falsa?

4.34 [indefinido] <§§4.1-4.5> Reúna um conjunto de artigos que acredita que contenham análises incorretas de desempenho ou usem métrica de desempenho suspeita para tentar persuadir leitores. Por exemplo, um artigo no *New York Times* (20 de abril de 1994, p. D1) descreveu um videogame “que irá suplantar o poder de computação até mesmo dos computadores pessoais mais poderosos” e apresentou o seguinte gráfico para apoiar o argumento de que “os videogames podem ser os super-computadores de amanhã”:

Computador	Número aproximado de instruções por segundo	Preço
Mainframe IBM de 1975	10.000.000	US\$ 10.000.000
Cray-1 de 1976	160.000.000	US\$ 20.000.000
Digital VAX de 1979	1.000.000	US\$ 200.000
PC IBM de 1981	250.000	US\$ 3.000
Sun 2 de 1984	1.000.000	US\$ 10.000
PC com chip Pentium de 1994	66.000.000	US\$ 3.000
videogame PCX da Sony de 1995	500.000.000	US\$ 500
Micronunity set-top de 1995	1.000.000.000	US\$ 500

O artigo nunca discutiu como a natureza das instruções deve impactar a definição de “poderoso”. Para cada artigo que reunir, descreva o que você acha que é suspeito ou incorreto. Bons lugares para procurar material incluem as seções de negócios ou tecnologia dos jornais, revistas (tanto em artigos quanto em anúncios) e na Internet (grupos de discussão e a Web).

§4.1: página 186, 1. a: ambos, b: latência, c: nenhum. 2,7 segundos.

§4.2: página 192, 6.

§4.3: página 195, 1. F, F, F, V. 2. V, F, F, V

§4.4: página 201, b.

§4.5: página 204, a. Computador A. b. Computador B.

**Respostas
das Seções
“Verifique
você mesmo”**

Computadores no mundo real

Movendo pessoas com mais rapidez e segurança

Problema: encontre meios de ajudar a transportar pessoas rapidamente mantendo a segurança, o conforto e a eficiência.

Solução: durante mais de 20 anos, os computadores desempenharam um papel cada vez mais importante no controle dos sistemas de transporte incluindo aviões, trens, automóveis e mesmo navios. Praticamente todos os sistemas de transporte modernos se baseiam nos computadores para aumentar sua segurança, conforto e eficiência. Os computadores também desempenham um papel fundamental na melhoria do consumo de combustível e na redução da poluição do ar. Aqui, examinamos alguns usos dos computadores nos trens e automóveis. No projeto de oleodutos, precisamos prevenir danos a todo custo e tentamos evitar atrasos. Os trens controlados por computador tentam fazer o mesmo: os danos significam riscos à vida e nunca podem ser permitidos; e os atrasos devem ser evitados!

O francês TGV (*Train a Grande Vitesse*) é um dos sistemas de trem mais rápidos do mundo, com uma velocidade de 300km/h. Tradicionalmente, os trens eram controlados por um maquinista, usando um sistema de luzes e sinais instalado no trilho que dizia ao maquinista para seguir, desacelerar ou parar. A 300km/h, era difícil ler esses sinais e fácil perder um completamente, levando a um risco de desastre. Além disso, alguns trilhos, como os que continham curvas fechadas, podiam ser inseguros em velocidades mais altas; como o TGV foi projetado para rodar em trilhos já existentes, era funda-

mental encontrar um método infalível de comunicar as condições do trilho.

Os projetistas do TGV resolveram esse problema com um sistema de sinalização inteligente, chamado TVM (*Transmission Voie-Machine*), que corre ao longo dos trilhos e é captado por antenas na locomotiva. O trilho é dividido em blocos, que normalmente possuem cerca de 1,5km de extensão. Blocos mais curtos são usados quando as condições do trilho mudam rapidamente ou quando um nível de segurança mais alto é crítico, como no Chunnel, onde a extensão de cada bloco é aproximadamente um décimo do normal. Transmissores no início de cada bloco são usados para comunicar instruções à cabine, onde elas são recebidas por um maquinista; um computador também monitora as comunicações e implementa os comandos se o maquinista falhar em fazê-lo.

Um problema é que a distância de parada para o TGV é nominalmente de quatro blocos (um



O trem do Eurostar TGV em Nice, na França.

pouco mais curta em uma emergência absoluta). O tempo para percorrer quatro blocos é de 1,2 minuto, e os novos trens do sistema de sinalização são operados com uma antecedência de 3 minutos, mesmo na neblina! Portanto, o sistema precisa monitorar a presença de todos os trens e garantir a propriedade mais importante: apenas um trem pode ocupar um bloco do trilho de cada vez! O sistema comunica constantemente a velocidade máxima segura para o bloco atual, melhorando o desempenho e a segurança.

O sistema TVM foi construído com uma forte atenção à segurança, o que significa um extenso uso de redundância para garantir a operação do sistema diante da falha de um componente. O índice de falha do TVM foi estimado em menos de 1 falha em um milhão de anos. Toda essa atenção à segurança tem valido a pena: em mais de duas décadas de funcionamento, não houve acidentes fatais causados por falha do TVM.

As imagens mostram o Eurostar TGV e a cabine e o assento do operador.

Os computadores também têm desempenhado um importante papel na fabricação de carros mais seguros, mais eficientes e menos poluentes. O automóvel moderno possui dezenas de microprocessadores controlando tudo desde o freio e a injeção eletrônica até o sistema de airbag.

Na área da segurança, os airbags e os freios antitravamento (ABS) foram duas das mais importantes inovações desde o cinto de segurança. Os freios ABS preservam a capacidade de dirigir durante uma frenagem brusca que pode ocorrer em condições de emergência. Detectando a retenção de uma roda e alternadamente aplicando e libe-

rando os freios sob controle do computador, um sistema de freio ABS pode evitar o travamento total da roda.

Os airbags usam um sensor de força para detectar a desaceleração brusca, que ocorre durante uma colisão. Os airbags são controlados por um computador que lê o sensor. A nova geração de airbags usa um processo em dois estágios: quando a desaceleração indica uma colisão de severidade moderada, o airbag infla mais lentamente, reduzindo a possibilidade de causar danos ao passageiro pelo rápido acionamento em uma colisão moderada. A confiabilidade desses sistemas de segurança foi melhorada por um teste controlado por computador que é realizado cada vez que o veículo é ligado.

Os sistemas de injeção eletrônica modernos em automóveis visam a melhorar o rendimento e reduzir a poluição. Por sorte, esses dois objetivos são mutuamente congruentes: a melhora no rendimento reduz a poluição pelo uso de menos combustível e o rendimento é melhorado por uma combustão mais eficaz, que também reduz a emissão de combustível parcialmente queimado. Os computadores controlam a injeção do combustível, a quantidade de ar injetada e o sincronismo da descarga elétrica, que precisa mudar quando o motor gira mais rápido. O controle cuidadoso desses elementos sob a faixa de operação total de 1.000 a 6.000RPM e em diferentes condições de temperatura ajudou a melhorar o rendimento e a reduzir a poluição.

Para saber mais, veja estas referências

"An investigation of the Therac-25 accidents", Nancy G. Leveson e Clark S. Turner. *IEEE Computer*, 26(7): 18-41, julho de 1993.



O interior de uma cabine do Eurostar TGV.

5

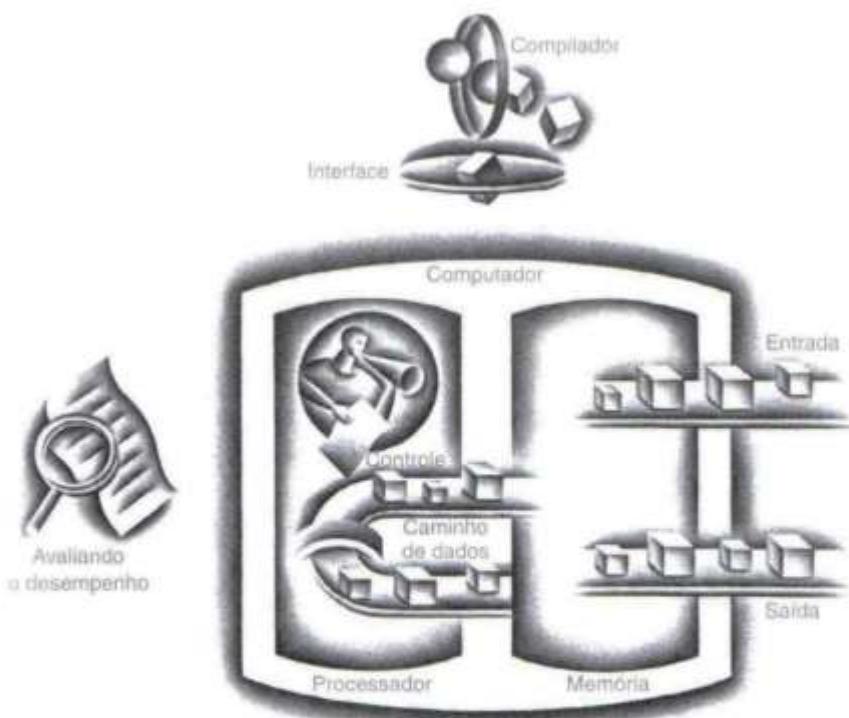
O Processador: Caminho de Dados e Controle

*Em um assunto importante,
nenhum detalhe é pequeno.*

Provérbio francês

- 5.1 Introdução 214**
- 5.2 Convenções lógicas de projeto 217**
- 5.3 Construindo um caminho de dados 220**
- 5.4 Um esquema de implementação simples 225**
- 5.5 Uma implementação multiciclo 239**
- 5.6 Exceções 257**
- 5.7 Microprogramação: simplificando o projeto de controle 261**
- 5.8 Uma introdução ao projeto digital usando uma linguagem de projeto de hardware 261**
- 5.9 Vida real: a organização das recentes implementações Pentium 261**
- 5.10 Falácia e armadilhas 264**
- 5.11 Comentários finais 265**
- 5.12 Perspectiva histórica e leitura adicional 266**
- 5.13 Exercícios 266**
-

Os cinco componentes clássicos de um computador



5.1

Introdução

No Capítulo 4, dissemos que o desempenho de uma máquina era determinado por três fatores principais: contagem de instruções, tempo de ciclo de clock e CPI (ciclos de clock por instrução). O compilador e o conjunto de instruções, que examinamos nos Capítulos 2 e 3, determinam a contagem de instruções necessária para um determinado programa. Entretanto, tanto o tempo de ciclo de clock quanto o número de ciclos de clock por instrução são determinados pela implementação do processador. Neste Capítulo, construímos o caminho de dados e a unidade de controle para duas implementações diferentes do conjunto de instruções MIPS.

Este capítulo contém uma explicação dos princípios e técnicas usados na implementação de um processador, começando com uma sinopse altamente abstrata e simplificada nesta seção, seguida de seções que desenvolvem um caminho de dados e constroem uma versão simples de um processador suficiente para implementar conjuntos de instruções como o MIPS, e, finalmente, desenvolvendo os conceitos necessários para implementar conjuntos de instruções mais complexos, como o IA-32.

Para o leitor interessado em entender a interpretação de alto nível de instruções e seu impacto sobre o desempenho do programa, esta seção inicial fornece uma base suficiente para compreender esses conceitos bem como os conceitos básicos do pipelining, que são explicados na Seção 6.1 do próximo capítulo.

Para os leitores que desejam um entendimento de como o hardware implementa instruções, as Seções 5.3 e 5.4 fornecem todo o material necessário. Além disso, essas duas seções são suficientes para entender todo o assunto apresentado no Capítulo 6 sobre pipelining. Apenas os leitores com um interesse em projeto de hardware devem ir além disso.

As outras seções deste capítulo discutem como o hardware moderno – incluindo os processadores mais complexos como a série Intel Pentium – normalmente é implementado. Os princípios básicos do controle por estados finitos são explicados e diferentes métodos de implementação, inclusive microprogramação, são examinados. Para o leitor interessado em entender o processador e seu desempenho em mais profundidade, as Seções 5.4 e 5.5 serão úteis. Para os leitores interessados no projeto de hardware moderno, a ■ Seção 5.7 discute a microprogramação, uma técnica usada para implementar um controle mais complexo como o existente nos processadores IA-32, e a ■ Seção 5.8 descreve como as linguagens de projeto de hardware e as ferramentas de CAD são usadas para implementar hardware.

Uma implementação MIPS básica

Analisaremos uma implementação que inclui um subconjunto do conjunto de instruções MIPS básico.

- As instruções de referência à memória load word (*lw*) e store word (*sw*)
- As instruções lógicas e aritméticas add, sub, and, or e slt
- As instruções branch equal (*beq*) e jump (*j*), que acrescentamos depois

Esse subconjunto não inclui todas as instruções de inteiro (por exemplo, shift, multiply e divide estão ausentes), nem inclui qualquer instrução de ponto flutuante. Entretanto, os princípios básicos usados na criação de um caminho de dados e no projeto do controle serão ilustrados. A implementação das outras instruções é semelhante.

Examinando a implementação, teremos a oportunidade de ver como o conjunto de instruções determina muitos aspectos da implementação e como a escolha de várias estratégias de implementação afeta a velocidade de clock e o CPI para a máquina. Muitos dos princípios básicos de projeto introduzidos no Capítulo 4 podem ser ilustrados considerando-se a implementação, como os princípios *Torne o caso comum mais rápido* e *A simplicidade favorece a regularidade*. Além disso, a maioria dos

conceitos usados para implementar o subconjunto MIPS neste capítulo e no próximo envolvem as mesmas idéias básicas usadas para construir um amplo espectro de computadores, desde servidores de alto desempenho até microprocessadores de finalidade geral e processadores embutidos, que estão sendo cada vez mais usados em produtos como videocassetes ou automóveis.

Uma sinopse da implementação

Nos Capítulos 2 e 3, olhamos as instruções MIPS básicas, incluindo as instruções lógicas e aritméticas, as instruções de referência à memória e as instruções de desvio. Muito do que precisa ser feito para implementar essas instruções é igual, independente da classe exata da instrução. Para cada instrução, as duas primeiras etapas são idênticas:

1. Enviar o contador de programa (PC) à memória que contém o código e buscar a instrução dessa memória.
2. Ler um ou mais registradores, usando campos da instrução para selecionar os registradores a serem lidos. Para a instrução load word, precisamos ler apenas um registrador, mas a maioria das outras instruções exige a leitura de dois registradores.

Após essas duas etapas, as ações necessárias para completar a instrução dependem da classe da instrução. Felizmente, para cada uma das três classes de instrução (referência à memória, lógica e aritmética e desvios), as ações são quase as mesmas, seja qual for o opcode exato.

Mesmo entre diferentes classes de instrução, há algumas semelhanças. Por exemplo, todas as classes de instrução, exceto jump, usam a unidade lógica e aritmética (ALU) após a leitura dos registradores. As instruções de referência à memória usam a ALU para o cálculo de endereço, as instruções lógicas e aritméticas para a execução da operação e desvios para comparação. Como podemos ver, a simplicidade e a regularidade do conjunto de instruções simplifica a implementação tornando semelhantes as execuções de muitas das classes de instrução.

Após usar a ALU, as ações necessárias para completar várias classes de instrução diferem. Uma instrução de referência à memória precisará acessar a memória para escrever dados para um store ou ler dados para um load. Uma instrução lógica e aritmética precisa escrever os dados da ALU de volta para um registrador. Finalmente, para uma instrução de desvio, podemos ter de mudar o próximo endereço de instrução com base na comparação; caso contrário, o PC deve ser incrementado em 4 para chegar ao endereço para a próxima instrução.

A Figura 5.1 mostra a visão em alto nível de uma implementação MIPS, focalizando as várias unidades funcionais e sua interconexão. Embora essa figura mostre a maioria do fluxo de dados pelo processador, ela omite dois importantes aspectos da execução da instrução.

Primeiro, em vários lugares, a Figura 5.1 mostra os dados indo para uma determinada unidade, vindo de duas origens diferentes. Por exemplo, o valor escrito no PC pode vir de dois somadores e os dados escritos no banco de registradores podem vir da ALU ou da memória de dados. Na prática, essas linhas de dados não podem simplesmente ser interligadas; precisamos adicionar um elemento que escolha dentre as diversas origens e conduza uma dessas origens para seu destino. Essa seleção normalmente é feita com um dispositivo chamado *multiplexador*, embora uma melhor denominação desse dispositivo seria *seletor de dados*. O multiplexador, descrito em detalhes no Apêndice B, seleciona dentre várias entradas com base na definição de suas linhas de controle. As linhas de controle são definidas principalmente com base nas informações extraídas da instrução executada.

Segundo, várias das unidades precisam ser controladas de acordo com o tipo da instrução. Por exemplo, a memória de dados precisa ler em um load e escrever em um store. O banco de registradores precisa ser escrito em uma instrução load e em uma instrução lógica ou aritmética. E, é claro, a ALU precisa realizar uma de várias operações, como vimos no Capítulo 3. (O Apêndice B descreve o projeto lógico detalhado da ALU.) Assim como os mux, essas operações são direcionadas por linhas de controle que são definidas baseadas nos vários campos das instruções.

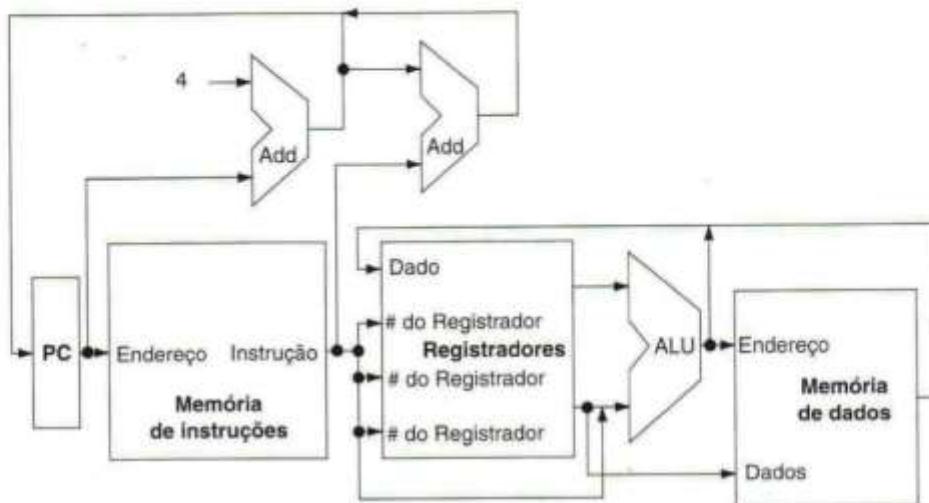


FIGURA 5.1 Uma visão abstrata da implementação do subconjunto MIPS mostrando as principais unidades funcionais e as principais conexões entre elas. Todas as instruções começam usando o contador de programa para fornecer o endereço de instrução para a memória de instruções. Depois que a instrução é buscada, os registradores usados como operandos pela instrução são especificados por campos dessa instrução. Uma vez que os operandos tenham sido buscados, eles podem ser operados para calcular um endereço de memória (para um load ou store), para calcular um resultado aritmético (para uma instrução lógica ou aritmética) ou a comparação (para um desvio). Se a instrução for uma instrução lógica ou aritmética, o resultado da ALU precisa ser escrito em um registrador. Se a operação for um load ou store, o resultado da ALU é usado como um endereço para armazenar o valor de um registrador ou ler um valor da memória para um registrador. O resultado da ALU ou memória é escrito de volta no banco de registradores. Os desvios exigem o uso da saída da ALU para determinar o próximo endereço de instrução, que vem da ALU (onde o offset do PC e do desvio são somados) ou de um somador que incrementa o PC atual em 4. As linhas finas interconectando as unidades funcionais representam barramentos, que consistem em múltiplos sinais. As linhas são usadas para guiar o leitor sobre como as informações fluem. Como as linhas de sinal podem se cruzar, mostramos explicitamente quando as linhas que se cruzam estão conectadas pela presença de um ponto no local do cruzamento.

A Figura 5.2 mostra o caminho de dados da Figura 5.1 com os três multiplexadores necessários acrescentados, bem como as linhas de controle para as principais unidades funcionais. Uma unidade de controle que tem a instrução como uma entrada é usada para determinar como definir as linhas de controle para as unidades funcionais e dois dos multiplexadores. O terceiro multiplexador – que determina se $PC + 4$ ou o endereço de destino do desvio é escrito no PC – é definido com base na saída zero da ALU, que é usada para realizar a comparação da instrução `beq`. A regularidade e a simplicidade do conjunto de instruções MIPS significam que um simples processo de decodificação pode ser usado para determinar como definir as linhas de controle.

No restante do capítulo, refinamos essa visão para preencher os detalhes, o que exige que acrescentemos mais unidades funcionais, aumentemos o número das conexões entre unidades e, é claro, adicionemos uma unidade de controle para controlar que ações são realizadas para diferentes classes de instrução. As Seções 5.3 e 5.4 descrevem uma implementação simples que usa um único ciclo de clock longo para cada instrução e segue a forma geral das Figuras 5.1 e 5.2. Nesse primeiro projeto, cada instrução começa a execução em uma transição do clock e completa a execução na próxima transição do clock.

Embora seja mais fácil de entender, esse método não é prático, já que seria mais lento do que uma implementação que permite que diferentes classes de instrução tomem números diferentes de ciclos de clock, cada um podendo ser muito mais curto. Após projetar o controle para essa máquina simples, olharemos uma implementação que usa múltiplos ciclos de clock para cada instrução. Esse projeto multiciclo será usado quando discutirmos conceitos de controle mais avançados, exceções de tratamento e o uso das linguagens de projeto de hardware nas Seções 5.5 a 5.8.

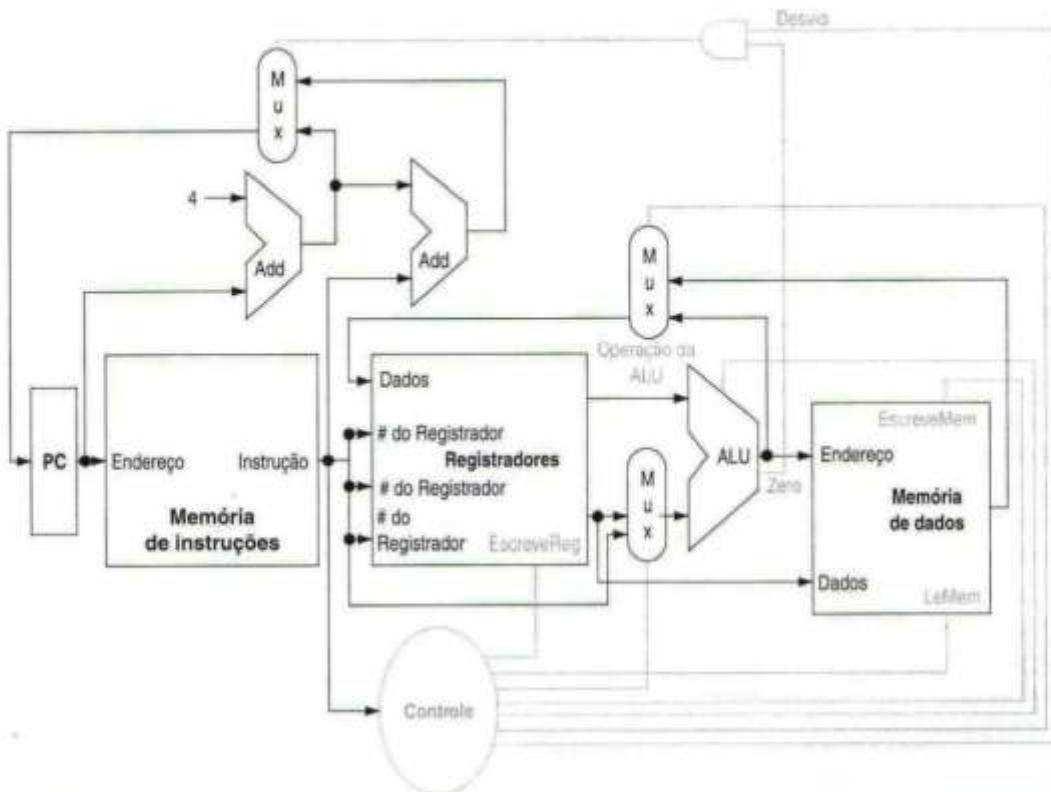


FIGURA 5.2 A implementação básica do subconjunto MIPS incluindo as linhas de controle e os multiplexadores necessários.

O multiplexador superior controla que valor substitui o PC ($PC + 4$ ou o endereço de destino do desvio); o multiplexador é controlado pela porta que realiza um AND da saída Zero da ALU com um sinal de controle que indica que a instrução é de desvio. O multiplexador cuja saída retorna para o banco de registradores é usado para conduzir a saída da ALU (no caso de uma instrução lógica ou áritmética) ou a saída da memória de dados (no caso de um load) para ser escrita no banco de registradores. Finalmente, o multiplexador da parte inferior é usado para determinar se uma segunda entrada da ALU vem dos registradores (para uma instrução lógica ou áritmética não imediata) ou do campo offset da instrução (para uma operação imediata, um load ou store, ou um desvio). As linhas de controle acrescentadas são simples e determinam a operação realizada pela ALU, se a memória de dados deve ler ou escrever e se os registradores devem realizar uma operação de escrita. As linhas de controle são mostradas em tons de cinza para que sejam vistas com mais facilidade.

O caminho de dados de ciclo único conceitualmente descrito nesta seção *precisa* ter memórias de instrução e de dados separadas porque

Verifique você mesmo

1. O formato dos dados e das instruções é diferente no MIPS, e, portanto, são necessárias diferentes memórias
2. Ter memórias separadas é menos dispendioso
3. O processador opera em um ciclo e não pode usar uma memória de porta única para dois acessos diferentes dentro desse ciclo

5.2 Convenções lógicas de projeto

Para discutir o projeto de uma máquina, precisamos decidir como a implementação lógica da máquina irá operar e como a máquina será sincronizada. Esta seção examina algumas idéias básicas na lógica digital que usaremos em todo o capítulo. Se você tiver pouco ou nenhum conhecimento em lógica digital, provavelmente será útil ler o Apêndice B antes de continuar.

As unidades funcionais na implementação MIPS consistem em dois tipos diferentes de elementos lógicos: os elementos que operam nos valores dos dados e os elementos que contêm estado. Os elementos que operam nos valores dos dados são todos *combinacionais*, significando que suas saídas dependem apenas das entradas atuais. Dada a mesma entrada, um elemento combinacional sempre produz a mesma saída. A ALU mostrada na Figura 5.1 e discutida no Capítulo 3 e no Apêndice B é um elemento combinacional. Dado um conjunto de entradas, ele sempre produz a mesma saída porque não possui qualquer armazenamento interno.

Outros elementos no projeto não são combinatórios, mas contêm *estado*. Um elemento contém estado se tiver algum armazenamento interno. Chamamos esses elementos de **elementos de estado**, pois, se desconectássemos a máquina da tomada, poderíamos reiniciá-la carregando os elementos de estado com os valores que continham antes de interrompermos a energia. Além disso, se salvássemos e armazenássemos novamente os elementos de estado, seria como se a máquina nunca tivesse perdido a energia. Na Figura 5.1, as memórias de instruções e de dados, bem como os registradores, são exemplos de elementos de estado.

Um elemento de estado possui pelo menos duas entradas e uma saída. As entradas necessárias são os valores dos dados a serem escritos no elemento e o *clock*, que determina quando o valor dos dados deve ser escrito. A saída de um elemento de estado fornece o valor escrito em um ciclo de *clock* anterior. Por exemplo, um dos elementos de estado mais simples logicamente é um flip-flop tipo D (veja o Apêndice B), que possui exatamente essas duas entradas (um valor e um *clock*) e uma saída. Além dos flip-flops, nossa implementação MIPS também usa dois outros tipos de elementos de estado: memórias e registradores, ambos aparecendo na Figura 5.1. O *clock* é usado para determinar quando se deve escrever no elemento de estado; um elemento de estado pode ser lido a qualquer momento.

Os componentes lógicos que contêm estado também são chamados de *seqüenciais* porque suas saídas dependem de suas entradas e do conteúdo do estado interno. Por exemplo, a saída da unidade funcional representando os registradores depende dos números de registrador fornecidos e do que foi escrito nos registradores anteriormente. A operação dos elementos combinatórios e seqüenciais e sua construção são discutidas em mais detalhes no Apêndice B.

Usaremos o termo *ativo* para indicar um sinal que está logicamente alto e *ativar* para especificar que um sinal deve ser determinado logicamente alto, e *inativo* ou *desativar* para representar o que é logicamente baixo.

Metodologia de clocking

metodologia de clocking O método usado para determinar quando os dados são válidos e estáveis em relação ao *clock*.

sincronização acionada por transição Um esquema de clocking em que todas as mudanças de estado ocorrem em uma transição do *clock*.

Uma **metodologia de clocking** define quando os sinais podem ser lidos e quando podem ser escritos. Ela é importante para especificar a sincronização das leituras e escritas porque, se um sinal fosse escrito ao mesmo tempo em que fosse lido, o valor da leitura poderia corresponder ao valor antigo, ao valor recém-escrito ou mesmo alguma combinação dos dois! Obviamente, os projetos de computadores não podem tolerar essa imprevisibilidade. Uma metodologia de clocking tem o objetivo de evitar essa circunstância.

Para simplificar, consideraremos uma metodologia de **sincronização acionada por transição**. Uma metodologia de sincronização acionada por transição significa que quaisquer valores armazenados em um elemento lógico seqüencial são atualizados apenas em uma transição do *clock*. Como apenas os elementos de estado podem armazenar valores de dados, qualquer coleção de lógica combinatória precisa ter suas entradas vindo de um conjunto de elementos de estado e suas saídas escritas em um conjunto de elementos de estado. As entradas são valores escritos em um ciclo de *clock* anterior, enquanto as saídas são valores que podem ser usados em um ciclo de *clock* seguinte.

A Figura 5.3 mostra os dois elementos de estado em volta de um bloco de lógica combinatória, que opera em um único ciclo de *clock*: todos os sinais precisam se propagar desde o elemento de estado 1, passando pela lógica combinatória e indo até o elemento 2 no tempo de um ciclo de *clock*. O tempo necessário para os sinais alcançarem o elemento 2 define a duração do ciclo de *clock*.

elemento de estado Um elemento com memória.

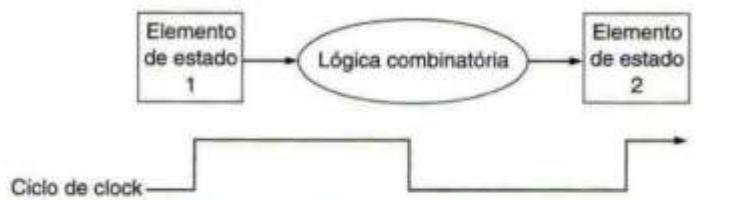


FIGURA 5.3 A lógica combinatória, os elementos de estado e o clock estão intimamente relacionados.

Em um sistema digital síncrono, o clock determina quando os elementos com estado escreverão valores no armazenamento interno. Quaisquer entradas em um único elemento precisam alcançar um valor estável (ou seja, ter alcançado um valor do qual não mudarão até após a transição do clock) antes que a transição ativa do clock faça com que o estado seja atualizado. Todos os elementos de estado, incluindo a memória, são considerados acionados por transição.

Para simplificar, não mostraremos um **sinal de controle** de escrita quando um elemento de estado é escrito em cada transição ativa de clock. Por outro lado, se um elemento de estado não for atualizado em cada clock, um sinal de controle de escrita explícito é necessário. Tanto o sinal de clock quanto o sinal de controle de escrita são entradas, e o elemento de estado só é alterado quando o sinal de controle de escrita está ativo e ocorre uma transição do clock.

Uma metodologia acionada por transição permite ler o conteúdo de um registrador, enviar o valor por meio de alguma lógica combinatória e escrever nesse registrador no mesmo ciclo de clock, como mostra a Figura 5.4. Não importa se consideramos que todas as escritas ocorrem na transição de subida do clock ou na transição de descida, já que as entradas no bloco de lógica combinatória não podem mudar exceto na transição de clock escolhida. Com uma metodologia de sincronização acionada por transição, não há qualquer feedback dentro de um único ciclo de clock, e a lógica na Figura 5.4 funciona corretamente. No Apêndice B, discutimos brevemente as outras limitações (como os tempos de setup e hold), bem como outras metodologias de sincronização.

Quase todos esses elementos de estado e lógicos terão entradas e saídas contendo 32 bits de extensão, já que essa é a extensão da maioria dos dados manipulados pelo processador. Sempre que uma unidade tiver uma entrada ou saída diferente de 32 bits de extensão, deixaremos isso claro. As figuras indicarão **barramentos** (que são sinais mais largos do que 1 bit), com linhas mais finas. Algumas vezes, desejaremos combinar vários barramentos para formar um barramento mais largo; por exemplo, podemos querer obter um barramento de 32 bits combinando dois de 16 bits. Nesses casos, rótulos nas linhas de barramento indicarão que estamos concatenando barramentos para formar um barramento mais largo. Setas também são incluídas para ajudar a esclarecer a direção do fluxo dos dados entre elementos. Finalmente, o realce indica um sinal de controle em oposição a um sinal que conduz dados; essa distinção se tornará mais clara enquanto avançarmos neste capítulo.

Verdadeiro ou falso: como o banco de registradores é lido e escrito no mesmo ciclo de clock, qualquer caminho de dados MIPS usando escritas acionadas por transição precisa ter mais de uma cópia do banco de registradores.

sinal de controle Um sinal usado para seleção de multiplexador ou para direcionar a operação de uma unidade funcional; contrasta com um sinal de dados, que contém informações operadas por uma unidade funcional.

Verifique você mesmo

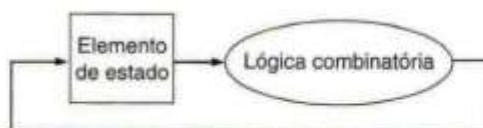


FIGURA 5.4 Uma metodologia acionada por transição permite que um elemento de estado seja lido e escrito no mesmo ciclo de clock sem criar uma disputa que poderia levar a valores de dados indeterminados.

É claro que o ciclo de clock ainda precisa ser longo o suficiente para que os valores de entrada sejam estáveis quando a transição ativa do clock ocorrer. O feedback não pode ocorrer dentro de 1 ciclo de clock devido à atualização acionada por transição do elemento de estado. Se o feedback fosse possível, esse projeto não poderia funcionar corretamente. Nossos projetos neste capítulo e no próximo se baseiam na metodologia de sincronização acionada por transição e em estruturas como a mostrada nesta figura.

5.3

Construindo um caminho de dados

elemento do caminho de dados Uma unidade funcional usada para operar sobre os dados ou contar esses dados dentro de um processador. Na implementação MIPS, os elementos do caminho de dados incluem as memórias de instruções e de dados, o banco de registradores, a unidade lógica e aritmética (ALU) e os somadores.

contador de programa (PC) O registrador contendo o endereço da instrução do programa executado.

banco de registradores Um elemento de estado que consiste em um grupo de registradores que podem ser lidos e escritos fornecendo um número de registrador a ser acessado.

Uma maneira razoável de iniciar um projeto de caminho de dados é examinar os principais componentes necessários para executar cada classe de instrução MIPS. Vamos começar olhando quais **elementos do caminho de dados** cada instrução precisa. Quando mostramos os elementos do caminho de dados, também mostramos seus sinais de controle.

A Figura 5.5 mostra o primeiro elemento de que precisamos: uma unidade de memória para armazenar as instruções de um programa e fornecer instruções dado um endereço. A Figura 5.5 também mostra um registrador, que podemos chamar de **contador de programa (PC)**, que é usado para conter o endereço da instrução atual. Finalmente, precisaremos de um somador para incrementar o PC para o endereço da próxima instrução. Esse somador, que é combinatório, pode ser construído a partir da ALU que descrevemos no Capítulo 3 e projetamos em detalhes no Apêndice B, simplesmente interligando as linhas de controle de modo que o controle sempre especifique uma operação de adição. Representaremos uma ALU desse tipo com o rótulo *Add*, como na Figura 5.5, para indicar que ela se tornou permanentemente um somador e não pode realizar as outras funções da ALU.

Para executar qualquer instrução, precisamos começar buscando a instrução na memória. Para preparar para executar a próxima instrução, também temos de incrementar o contador de programa de modo que aponte para a próxima instrução, 4 bytes depois. A Figura 5.6 mostra como os três elementos da Figura 5.5 são combinados para formar um caminho de dados que busca instruções e incrementa o PC para obter o endereço da próxima instrução seqüencial.

Agora, vamos considerar as instruções de formato R (veja a Figura 2.7). Todas elas leem dois registradores, realizam uma operação na ALU com o conteúdo dos registradores e escrevem o resultado. Chamamos essas instruções de *instruções tipo R* ou *instruções lógicas ou aritméticas* (já que elas realizam operações lógicas ou aritméticas). Essa classe de instrução inclui add, sub, and, or e slt, que foram apresentadas no Capítulo 2. Lembre-se de que um caso típico desse tipo de instrução é add \$t1, \$t2, \$t3, que lê \$t2 e \$t3 e escreve em \$t1.

Os registradores de uso geral de 32 bits do processador são armazenados em uma estrutura chamada **banco de registradores**. Um banco de registradores é uma coleção de registradores em que qualquer registrador pode ser lido ou escrito especificando o número do registrador no banco. O banco de registradores contém o estado dos registradores da máquina. Além disso, precisaremos que uma ALU opere nos valores lidos dos registradores.

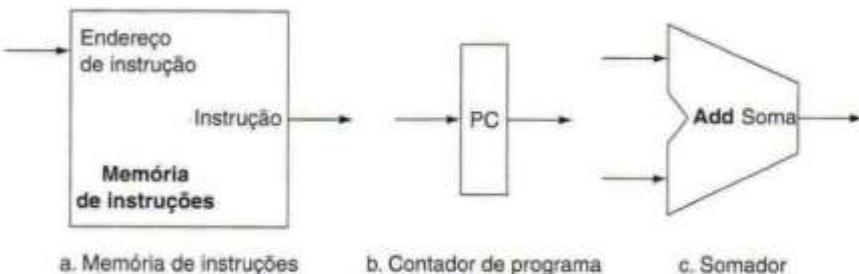


FIGURA 5.5 Dois elementos de estado são necessários para armazenar e acessar instruções, e um somador é necessário para calcular o endereço da próxima instrução. Os elementos de estado são a memória de instruções e o contador de programa. A memória de instruções só precisa fornecer acesso de leitura porque o caminho de dados não escreve instruções. Como a memória de instruções apenas é lida, nós a tratamos como lógica combinatória: a saída em qualquer momento reflete o conteúdo do local especificado pela entrada de endereço, e nenhum sinal de controle de leitura é necessário. (Precisamos escrever na memória de instruções quando carregarmos o programa; isso não é difícil de incluir e ignoramos em favor da simplicidade.) O contador de programa é um registrador de 32 bits que será escrito no final de cada ciclo de clock e, portanto, não precisa de um sinal de controle de escrita. O somador é uma ALU configurada para sempre realizar a adição das suas duas entradas de 32 bits e colocar o resultado em sua saída.

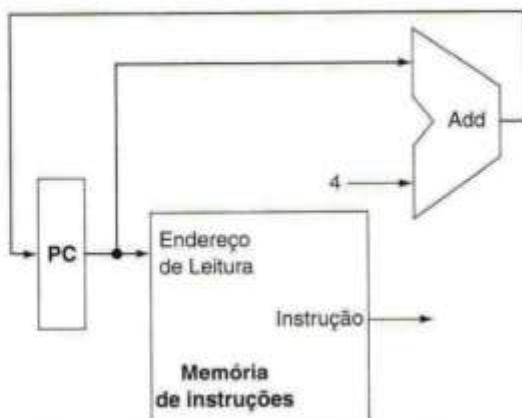


FIGURA 5.6 Uma parte do caminho de dados usada para buscar instruções e incrementar o contador do programa. A instrução buscada é usada por outras partes do caminho de dados.

Devido às instruções de formato R terem três operandos de registrador, precisaremos ler duas words de dados do banco de registradores e escrever uma word de dados no banco de registradores para cada instrução. Para que cada word de dados seja lida dos registradores, precisamos de uma entrada no banco de registradores que especifique o número do registrador a ser lido e uma saída do banco de registradores que conduzirá o valor lido dos registradores. Para escrever uma word de dados, precisaremos de duas entradas: uma para especificar o *número do registrador* a ser escrito e uma para fornecer os *dados* a serem escritos no registrador. O banco de registradores sempre gera como saída o conteúdo de quaisquer números de registrador que estejam nas entradas Registrador de leitura. As escritas, entretanto, são controladas pelo sinal de controle de escrita, que precisa ser ativo para que uma escrita ocorra na transição do clock. Portanto, precisamos de um total de quatro entradas (três para números de registrador e uma para dados) e duas saídas (ambas para dados), como mostra a Figura 5.7. As entradas de número de registrador possuem 5 bits de largura para especificar um dos 32 registradores ($32 = 2^5$), enquanto a entrada de dados e os dois barramentos de saída de dados possuem 32 bits de largura cada um.

A Figura 5.7 mostra a ALU, que usa duas entradas de 32 bits e produz um resultado de 32 bits, bem como um sinal de 1 bit se o resultado for 0. O sinal de controle de quatro bits da ALU é descrito em detalhes no Apêndice B; examinaremos o controle da ALU brevemente quando precisarmos saber como defini-lo.

A seguir, considere as instruções MIPS load word e store word, que possuem o formato `lw $t1, offset_value($t2)` ou `sw $t1, offset_value($t2)`. Essas instruções calculam um endereço de memória somando o registrador de base, que é `$t2`, com o campo offset de 16 bits com sinal contido na instrução. Se a instrução for um store, o valor a ser armazenado também precisará ser lido do banco de registradores onde ele reside em `$t1`. Se a instrução for um load, o valor lido da memória precisará ser escrito no banco de registradores no registrador especificado, que é `$t1`. Conseqüentemente, precisaremos do banco de registradores e da ALU da Figura 5.7.

Além disso, precisaremos de uma unidade para **estender o sinal** do campo offset de 16 bits da instrução para um valor com sinal de 32 bits, e de uma unidade de memória da qual ler ou na qual escrever. A memória de dados precisa ser escrita com instruções store; portanto, ela tem sinais de controle de leitura e escrita, uma entrada de endereço e uma entrada para os dados serem escritos na memória. A Figura 5.8 mostra esses dois elementos.

A instrução `beq` possui três operandos, dois registradores comparados para igualdade e um offset de 16 bits para calcular o **endereço de destino do desvio** relativo ao endereço da instrução desvio. Sua forma é `beq $t1,$t2,offset`. Para implementar essa instrução, precisamos calcular o endereço de destino somando o campo offset estendido com sinal da instrução com o PC. Há dois detalhes na definição de instruções de desvio (veja o Capítulo 2) para os quais precisamos prestar atenção:

estender o sinal

Aumentar o tamanho de um item de dados replicando o bit mais alto de sinal do item de dados original nos bits mais altos do item de dados maior de destino.

endereço de destino do desvio

O endereço especificado em um desvio, que se torna o novo contador de programa (PC) se o desvio for tomado. Na arquitetura MIPS, o destino do desvio é dado pela soma do campo offset da instrução e o endereço da instrução seguinte ao desvio.



FIGURA 5.7 Os dois elementos necessários para implementar operações para a ALU no formato R são o banco de registradores e a ALU. O banco de registradores contém todos os registradores e possui duas portas para leitura e uma porta para escrita. O projeto dos bancos de registradores de várias portas é discutido na Seção B.8 do Apêndice B. O banco de registradores sempre gera como saídas os conteúdos dos registradores correspondentes às entradas Registrador de leitura nas saídas; nenhuma outra entrada de controle é necessária. Ao contrário, uma escrita em um registrador precisa ser explicitamente indicada ativando o sinal de controle de escrita. Lembre-se de que as escritas são acionadas por transição, de modo que todas as entradas de escrita (por exemplo, o valor a ser escrito, o número do registrador e o sinal de controle de escrita) precisam ser válidas na transição do clock. Como as escritas no banco de registradores são acionadas por transição, nosso projeto pode ler e escrever sem problemas no mesmo registrador dentro de um ciclo de clock: a leitura obterá o valor escrito em um ciclo de clock anterior, enquanto o valor escrito estará disponível para uma leitura em um ciclo de clock subsequente. As entradas com o número do registrador para o banco de registradores possuem todas 5 bits de largura, enquanto as linhas com os valores de dados possuem 32 bits de largura. A operação a ser realizada pela ALU é controlada com o sinal de operação da ALU, que terá largura de 4 bits, usando a ALU projetada no Apêndice B. Em breve, usaremos a saída de detecção Zero da ALU para implementar desvios. A saída de overflow não será necessária até a Seção 5.6, quando discutiremos as exceções; até lá, elas serão omitidas.

- O conjunto de instruções especifica que a base para o cálculo do endereço de desvio é o endereço da instrução seguinte ao desvio. Como calculamos $PC + 4$ (o endereço da próxima instrução) no caminho de dados para busca de instruções, é fácil usar esse valor como a base para calcular o endereço de destino do desvio.
- A arquitetura também diz que o campo offset é deslocado 2 bits para a esquerda de modo que seja um offset de uma word; esse deslocamento aumenta a faixa efetiva do campo offset por um fator de quatro vezes.

Para lidar com a última complicação, precisaremos deslocar o campo offset de dois bits.

Além de calcular o endereço de destino do desvio, também precisamos determinar se a próxima instrução é a instrução que segue seqüencialmente ou a instrução no endereço de destino do desvio. Quando a condição é verdadeira (isto é, os operandos são iguais), o endereço de destino do desvio se torna o novo PC e dizemos que o **desvio é tomado**. Se os operandos não forem iguais, o PC incrementado deve substituir o PC atual (exatamente como para qualquer outra instrução normal); nesse caso, dizemos que o **desvio é não tomado**.

Portanto, o caminho de dados de desvio precisa de duas operações: calcular o endereço de destino do desvio e comparar o conteúdo do registrador. (Os desvios também afetam a parte da busca de instrução do caminho de dados, como veremos em breve.) Devido à complexidade da manipulação dos desvios, mostramos a estrutura do segmento do caminho de dados que lida com os desvios na Figura 5.9. Para calcular o endereço de destino do desvio, o caminho de dados de desvio inclui uma unidade de extensão de sinal, exatamente como a da Figura 5.8, e um somador. Para realizar a comparação, precisamos usar o banco de registradores mostrado na Figura 5.7 para fornecer os dois operandos (embora não precisemos escrever no banco de registradores). Além disso, a comparação pode ser feita usando a ALU que projetamos no Apêndice A. Como essa ALU fornece um sinal de saída que indica se o resultado era 0, podemos enviar os dois registradores operandos para a ALU com o con-

desvio tomado

Um desvio em que a condição de desvio é satisfeita, e o contador do programa (PC) se torna o destino do desvio. Todos os desvios incondicionais são desvios tomados.

desvio não tomado

Um desvio em que a condição de desvio é falsa e o contador do programa (PC) se torna o endereço da instrução que segue seqüencialmente o desvio.

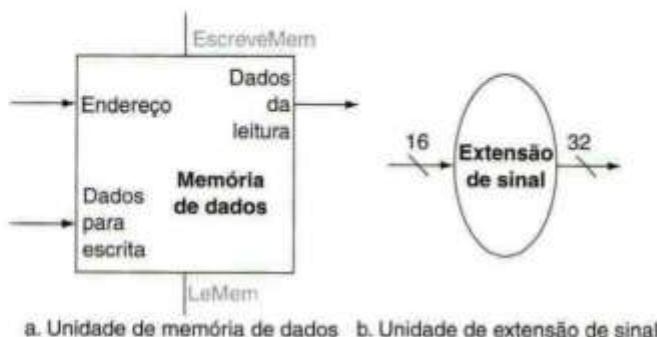


FIGURA 5.8 As duas unidades necessárias para implementar loads e stores, além do banco de registradores e da ALU da Figura 5.7, são a unidade de memória de dados e a unidade de extensão de sinal. A unidade de memória é um elemento de estado com entradas para os endereços e os dados de escrita, e uma única saída para o resultado da leitura. Existem controles de leitura e escrita separados, embora apenas um deles pode ser ativado em qualquer clock específico. A unidade de memória precisa de um sinal de leitura, já que, diferente do banco de registradores, ler o valor de um endereço inválido pode causar problemas, como veremos no Capítulo 7. A unidade de extensão de sinal possui uma entrada de 16 bits que tem o seu sinal estendido para que um resultado de 32 bits apareça na saída (veja o Capítulo 3). Consideraremos que a memória de dados é acionada por transição para as escritas. Na verdade, os chips de memória padrão possuem um sinal “write enable” que é usado para escritas. Embora o write enable não seja acionado por transição, nosso projeto acionado por transição poderia facilmente ser adaptado para funcionar com chips de memória reais. Consulte a Seção B.8 do Apêndice B para ver uma discussão mais detalhada de como funcionam os chips de memória reais.

junto de controle para fazer uma subtração. Se o sinal Zero da ALU estiver ativo, sabemos que os dois valores são iguais. Embora a saída de Zero sempre sinalize quando o resultado é 0, nós a estaremos usando apenas para implementar o teste de igualdade dos desvios. Mais adiante, mostraremos exatamente como conectar os sinais de controle da ALU para uso no caminho de dados.

A instrução jump funciona substituindo os 28 bits menos significativos do PC pelos 26 bits menos significativos da instrução deslocados de 2 bits à esquerda. Esse deslocamento é realizado simplesmente concatenando 00 ao offset do jump, como descrito no Capítulo 2.

Detalhamento: no conjunto de instruções MIPS, os desvios são **delayed**, significando que a instrução imediatamente posterior ao desvio é sempre executada, independente de a condição de desvio ser verdadeira ou falsa. Quando a condição é falsa, a execução se parece com um desvio normal. Quando a condição é verdadeira, um delayed branch primeiro executa a instrução imediatamente posterior ao desvio na ordem seqüencial antes de desviar para o endereço de destino do desvio. A motivação para os delayed branches surge de como o pipelining afeta os desvios (veja a Seção 6.6). Para simplificar, ignoramos os delayed branches neste capítulo e implementamos uma instrução beq como não sendo delayed.

delayed branch Um tipo de desvio em que a instrução imediatamente seguinte ao desvio é sempre executada, independente de a condição do desvio ser verdadeira ou falsa.

Criando um caminho de dados simples

Agora que examinamos os componentes do caminho de dados necessários para as classes de instrução individualmente, podemos combiná-los em um único caminho de dados e acrescentar o controle para completar a implementação. O caminho de dados mais simples pode tentar executar todas as instruções em um único ciclo de clock. Isso significa que nenhum recurso do caminho de dados pode ser usado mais de uma vez por instrução e, portanto, qualquer elemento necessário mais de uma vez precisa ser duplicado. Então, precisamos de uma memória para instruções separada de uma memória para dados. Embora algumas unidades funcionais precisem ser duplicadas, muitos dos elementos podem ser compartilhados por diferentes fluxos de instrução.

Para compartilhar um elemento do caminho de dados entre duas classes de instrução diferentes, talvez tenhamos de permitir múltiplas conexões com a entrada de um elemento usando um multiplexador e um sinal de controle para selecionar entre as múltiplas entradas.

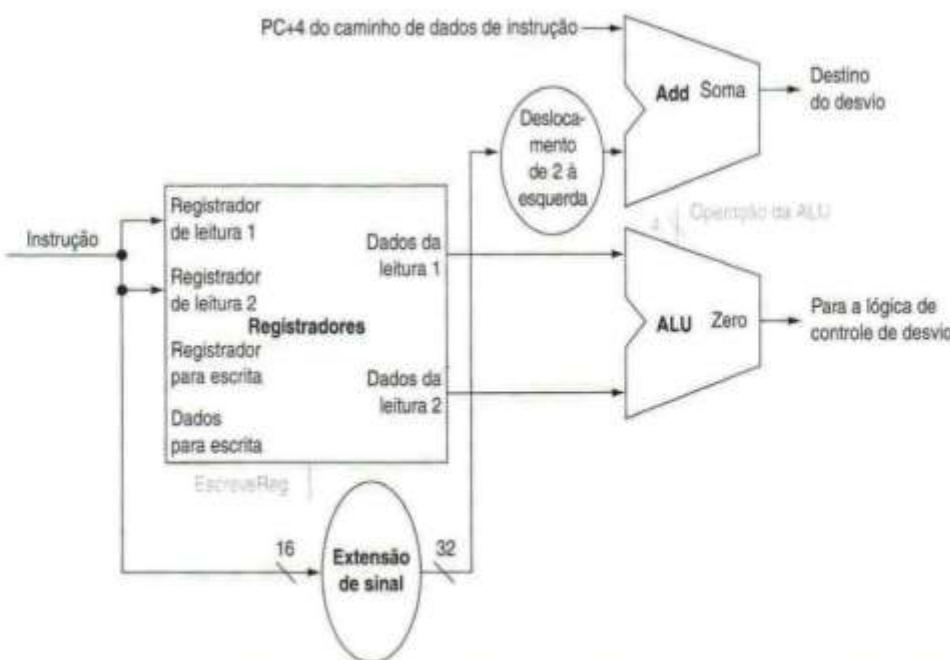


FIGURA 5.9 O caminho de dados para um desvio usa a ALU para avaliar a condição de desvio e um somador separado para calcular o destino do desvio como a soma do PC incrementado e os 16 bits mais baixos da instrução com sinal estendido (o deslocamento do desvio), deslocados de 2 bits para a esquerda. A unidade rotulada como *Deslocamento de 2 à esquerda* é simplesmente um direcionamento dos sinais entre entrada e saída que acrescenta 00_{bin} à extremidade da direita do campo offset com sinal estendido; nenhum hardware de deslocamento real é necessário, já que a quantidade de “deslocamento” é constante. Como sabemos que o offset teve o sinal dos seus 16 bits estendido, o deslocamento irá descartar apenas “bits de sinal”. A lógica de controle é usada para decidir se o PC ou o destino do desvio incrementado deve substituir o PC, com base na saída Zero da ALU.

CONSTRUINDO UM CAMINHO DE DADOS

EXEMPLO

As operações do caminho de dados das instruções lógicas e aritméticas (ou tipo R) e das instruções de acesso à memória são muito semelhantes. As principais diferenças são as seguintes:

- As instruções lógicas e aritméticas usam a ALU com as entradas vindas de dois registradores. As instruções de acesso à memória também podem usar a ALU para fazer o cálculo do endereço, embora a segunda entrada seja o campo offset de 16 bits com sinal estendido da instrução.
- O valor armazenado em um registrador de destino vem da ALU (para uma instrução tipo R) ou da memória (para um load).

Mostre como construir um caminho de dados para a parte operacional das instruções de acesso à memória e das instruções lógicas e aritméticas, que use um único banco de registradores e uma única ALU para manipular os dois tipos de instrução, incluindo quaisquer multiplexadores necessários.

RESPOSTA

Para criar um caminho de dados com apenas um único banco de registradores e uma única ALU, precisamos suportar duas origens diferentes para a segunda entrada da ALU, bem como duas origens diferentes para os dados armazenados no banco de registradores. Portanto, um multiplexador é colocado na entrada da ALU e outro na entrada de dados para o banco de registradores. A Figura 5.10 mostra a parte operacional do caminho de dados combinado.

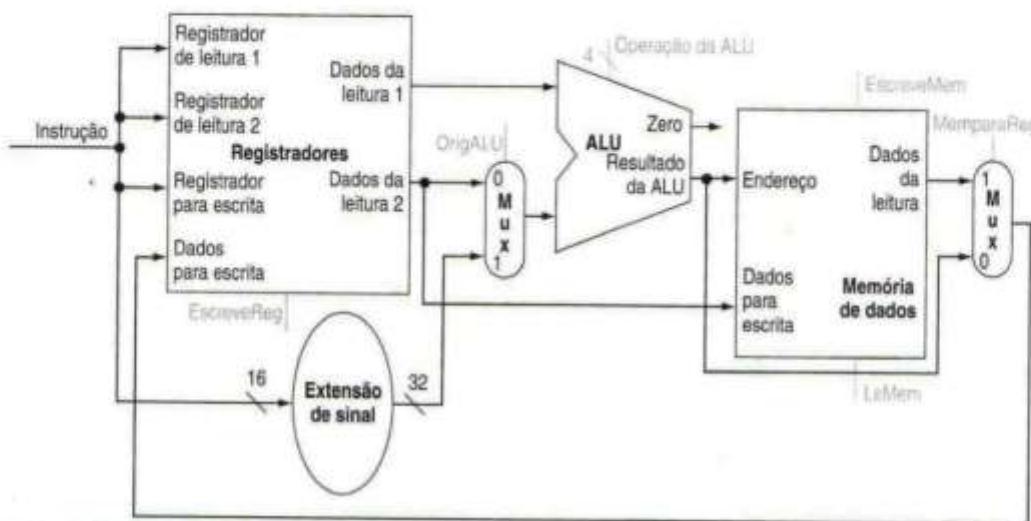


FIGURA 5.10 O caminho de dados para as instruções de acesso à memória e as instruções tipo R.

Este exemplo mostra como um único caminho de dados pode ser montado das partes nas Figuras 5.7 e 5.8 acrescentando multiplexadores. Dois multiplexadores são necessários, como descrito no exemplo.

Agora, podemos combinar todas as partes para criar um caminho de dados simples para a arquitetura MIPS incluindo um caminho de dados para busca de instruções (Figura 5.6), o caminho de dados das instruções tipo R e de acesso à memória (Figura 5.10) e o caminho de dados para desvios (Figura 5.9). A Figura 5.11 mostra o caminho de dados que obtemos compondo as partes separadas. A instrução de desvio usa a ALU principal para comparação dos registradores operandos, de modo que precisamos manter o somador da Figura 5.9 para calcular o endereço de destino do desvio. Um multiplexador adicional é necessário para selecionar o endereço de instrução seguinte ($PC + 4$) ou o endereço de destino do desvio a ser escrito no PC.

Agora que completamos este caminho de dados simples, podemos acrescentar a unidade de controle. A unidade de controle precisa ser capaz de ler entradas e gerar um sinal de escrita para cada elemento de estado, para o controle seletor de cada multiplexador e para o controle da ALU. O controle da ALU é diferente de várias maneiras e será útil projetá-lo primeiro, antes de projetarmos o restante da unidade de controle.

Qual das seguintes afirmativas é correta para uma instrução load?

Verifique você mesmo

- MemparaReg deve ser definido para fazer com que os dados da memória sejam enviados ao banco de registradores.
- MemparaReg deve ser definido para fazer com que o registrador de destino correto seja enviado ao banco de registradores.
- Não precisamos nos importar com MemparaReg.

5.4 Um esquema de implementação simples

Nesta seção, veremos o que poderia ser considerado a implementação mais simples possível do nosso subconjunto MIPS. Construímos essa implementação simples usando o caminho de dados da última seção e acrescentando uma função de controle simples. Essa implementação simples cobre as

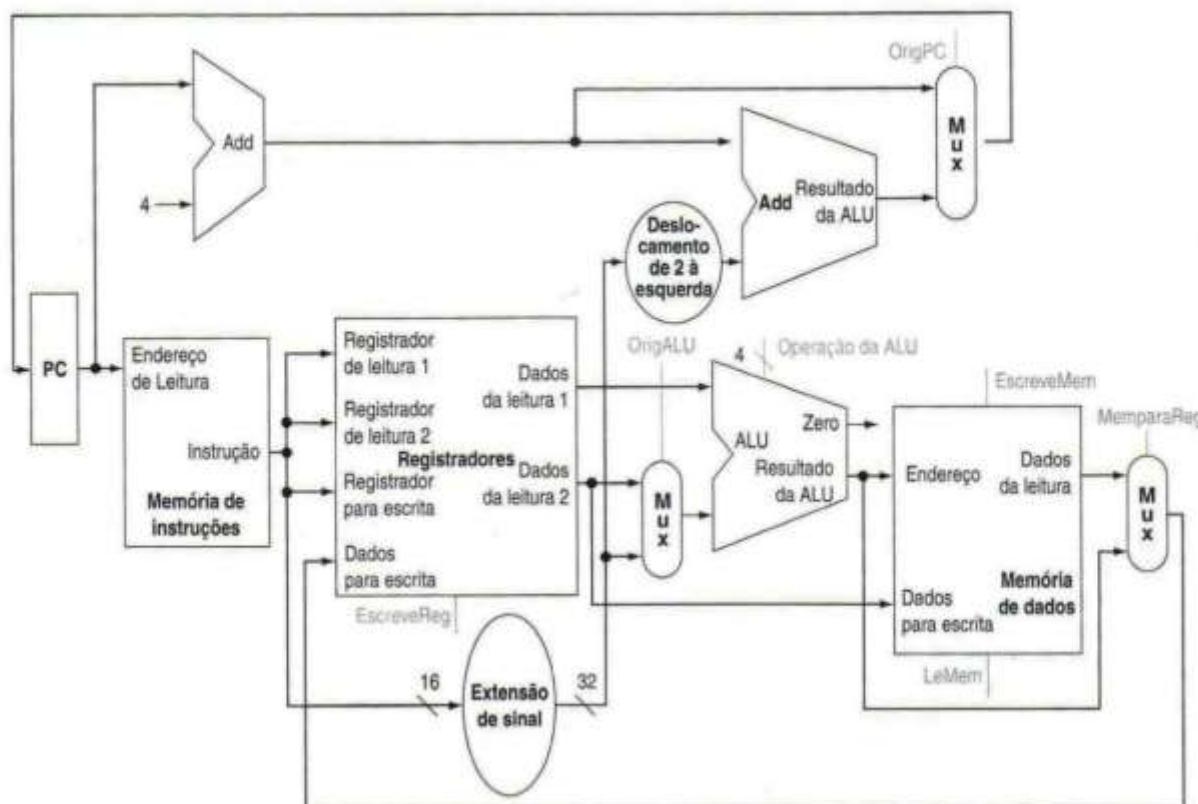


FIGURA 5.11 O caminho de dados simples para a arquitetura MIPS combina os elementos necessários para diferentes classes de instrução. Este caminho de dados pode executar as instruções básicas (load/store word, operações da ALU e desvios) em um único ciclo de clock. Um multiplexador adicional é necessário para integrar os desvios. O suporte para jumps será incluído mais tarde.

instruções load word (lw), store word (sw), branch equal (beq) e as instruções lógicas e aritméticas add, sub, and, or e set on less than. Posteriormente, desenvolveremos o projeto para incluir uma instrução jump (j).

O controle da ALU

Como pode ser visto no Apêndice B, a ALU possui quatro entradas de controle. Esses bits não foram codificados; portanto, apenas 6 das 16 combinações de entrada possíveis são usadas neste subconjunto. A ALU MIPS no Apêndice B mostra as 6 combinações a seguir:

Linhas de controle da ALU	Função
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Dependendo da classe de instrução, a ALU precisará realizar uma dessas cinco primeiras funções. (NOR é necessária para outras partes do conjunto de instruções MIPS.) Para as instruções load word e store word, usamos a ALU para calcular o endereço de memória por adição. Para instruções tipo R, a ALU precisa realizar uma das cinco ações (AND, OR, subtract, add ou set on less than), dependen-

do do valor do campo funct (ou function – função) de 6 bits nos bits menos significativos da instrução (veja o Capítulo 2). Para branch equal, a ALU precisa realizar uma subtração.

Podemos gerar a entrada do controle da ALU de 4 bits usando uma pequena unidade de controle que tenha como entradas o campo funct da instrução e um campo control de 2 bits, que chamamos de OpALU. OpALU indica se a operação a ser realizada deve ser add (00) para loads e stores, subtract (01) para beq ou determinada pela operação codificada no campo funct (10). A saída da unidade de controle da ALU é um sinal de 4 bits que controla diretamente a ALU gerando uma das combinações de 4 bits mostradas anteriormente.

Na Figura 5.12, mostramos como definir as entradas do controle da ALU com base no controle OpALU de 2 bits e no código de função de 6 bits. Para oferecer uma visão mais completa, a relação entre os bits de OpALU e o opcode da instrução também é mostrada. Mais adiante neste capítulo, veremos como os bits de OpALU são gerados na unidade de controle principal.

Opcode da instrução	OpALU	Operação da instrução	Campo funct	Ação da ALU desejada	Entrada do controle da ALU
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
tipo R	10	add	100000	add	0010
tipo R	10	subtract	100010	subtract	0110
tipo R	10	AND	100100	and	0000
tipo R	10	OR	100101	or	0001
tipo R	10	set on less than	101010	set on less than	0111

FIGURA 5.12 A forma como os bits de controle da ALU são definidos depende dos bits de controle de OpALU e dos diferentes códigos de função para as instruções tipo R. O opcode, que aparece na primeira coluna, determina a definição dos bits de OpALU. Todas as codificações são mostradas em binário. Observe que quando o código de OpALU é 00 ou 01, a ação da ALU desejada não depende do campo de código funct; nesse caso, dizemos que “não nos importamos” (don’t care) com o valor do código de função e o campo funct é mostrado como XXXXXX. Quando o valor de OpALU é 10, então o código de função é usado para definir a entrada do controle da ALU.

Esse estilo de usar vários níveis de decodificação – ou seja, a unidade de controle principal gera os bits de OpALU, que, então, são usados como entrada para o controle da ALU que gera os sinais reais para controlar a ALU – é uma técnica de implementação comum. Usar níveis múltiplos de controle pode reduzir o tamanho da unidade de controle principal. Usar várias unidades de controle menores também pode aumentar a velocidade da unidade de controle. Essas otimizações são importantes, já que a unidade de controle normalmente depende de um bom desempenho.

Há várias maneiras diferentes de implementar o mapeamento do campo OpALU de 2 bits e do campo funct de 6 bits para os três bits de controle de operação da ALU. Como apenas um pequeno número dos 64 valores possíveis do campo funct são de interesse e o campo funct é usado apenas quando os bits de OpALU são iguais a 10, podemos usar uma pequena lógica que reconhece o subconjunto dos valores possíveis e faz a definição correta dos bits de controle da ALU.

Como uma etapa no projeto dessa lógica, é útil criar uma tabela verdade para as combinações interessantes do campo de código funct e dos bits de OpALU, como fizemos na Figura 5.13; essa tabela verdade mostra como o controle da ALU de 3 bits é definido de acordo com esses dois campos de entrada. Como a tabela verdade inteira é muito grande ($2^8 = 256$ entradas) e não nos importamos com o valor do controle da ALU para muitas dessas combinações de entrada, mostramos apenas as entradas para as quais o controle da ALU precisa ter um valor específico. Em todo este capítulo, usaremos essa prática de mostrar apenas as entradas da tabela verdade que precisam ser declaradas e não mostrar as que são tudo zero ou que não nos interessam. (Essa prática possui uma desvantagem, que discutimos na Seção C.2 do Apêndice C.)

OpALU		Campo funct							Operação
OpALU1	OpALU 2	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 5.13 A tabela verdade para os três bits de controle da ALU (chamados Operação). As entradas são OpALU e o campo de código funct. Apenas as entradas para as quais o controle da ALU é ativado são mostradas. Algumas entradas don't care foram incluídas. Por exemplo, como OpALU não usa a codificação 11, a tabela verdade pode conter entradas 1X e X1, em vez de 10 e 01. Além disso, quando o campo funct é usado, os dois primeiros bits (F5 e F4) dessas instruções são sempre 10; portanto, eles são termos don't care e são substituídos por XX na tabela verdade.

termo don't care Um elemento de uma função lógica em que a saída não depende dos valores de todas as entradas. Os termos don't care podem ser especificados de diversas maneiras.

Como, em muitos casos, não nos interessamos pelos valores de algumas das entradas e para mantermos as tabelas compactas, também incluímos **termos don't care**. Um termo don't care nessa tabela verdade (representado por um X em uma coluna de entrada) indica que a saída não depende do valor da entrada correspondente a essa coluna. Por exemplo, quando os bits de OpALU são 00, como na primeira linha da tabela na Figura 5.13, sempre definimos o controle da ALU em 010, independente do código funct. Nesse caso, então, as entradas do código funct serão don't care nessa linha da tabela verdade. Depois, veremos exemplos de outro tipo de termo don't care. Se você não estiver familiarizado com o conceito de termos don't care, veja o Apêndice B para obter mais informações.

Uma vez construída a tabela, ela pode ser otimizada e, depois, transformada em portas lógicas. Esse processo é completamente mecânico. Portanto, em vez de mostrar as etapas finais aqui, descrevemos o processo e o resultado na Seção C.2 do ■ Apêndice C.

Projetando a unidade de controle principal

Agora que descrevemos como projetar uma ALU que usa o código de função e um sinal de 2 bits como suas entradas de controle, podemos voltar a considerar o restante do controle. Para começar esse processo, vamos identificar os campos de uma instrução e as linhas de controle necessárias para o caminho de dados construído na Figura 5.11. Para entender como conectar os campos de uma instrução com o caminho de dados, é útil examinar os formatos das três classes de instrução: as instruções tipo R, as instruções de desvio e as instruções de acesso à memória. A Figura 5.14 mostra esses formatos.

Existem várias observações importantes sobre esses formatos de instrução em que nos basearemos:

opcode O campo que denota a operação e o formato de uma instrução.

- O campo op, também chamado **opcode**, está sempre contido nos bits 31:26. Iremos nos referir a esse campo como Op[5:0].
- Os dois registradores a serem lidos sempre são especificados pelos campos rs e rt, nas posições 25:21 e 20:16. Isso é verdade para as instruções tipo R, branch equal e store.
- O registrador de base para as instruções load e store está sempre nas posições de bit 25:21 (rs).
- O offset de 16 bits para branch equal, load e store está sempre nas posições 15:0.
- O registrador de destino está em um de dois lugares. Para um load, ele está nas posições 20:16 (rt), enquanto para uma instrução tipo R, ele está nas posições 15:11 (rd). Portanto, precisaremos incluir um multiplexador para selecionar que campo da instrução será usado para indicar o número de registrador a ser escrito.

Usando essas informações, podemos acrescentar os rótulos de instrução e o multiplexador extra (para a entrada Registrador para escrita do banco de registradores) no caminho de dados simples. A Figura 5.15 mostra essas adições, além do bloco de controle da ALU, os sinais de escrita para ele-

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0
a. R-type instruction						
Field	35 or 43	rs	rt		address	
Bit positions	31:26	25:21	20:16		15:0	
b. Load or store instruction						
Field	4	rs	rt		address	
Bit positions	31:26	25:21	20:16		15:0	
c. Branch instruction						

FIGURA 5.14 As três classes de instrução (tipo R, acesso à memória e desvio) usam dois formatos de instrução diferentes. As instruções jump usam outro formato, que será discutido em breve. (a) Formato de instrução para instruções tipo R, as quais possuem todas opcode 0. Essas instruções possuem três registradores como operandos: rs, rt e rd. Os campos rs e rt são origens e rd é o destino. A função da ALU está no campo funct e é decodificada pelo projeto de controle da ALU da seção anterior. As instruções tipo R que implementamos são add, sub, and, or e slt. O campo shamt é usado apenas para deslocamentos; nós o ignoraremos neste capítulo. (b) Formato de instrução para instruções load (opcode = 35_{dec}) e store (opcode = 43_{dec}). O registrador rs é o registrador de base adicionado ao campo address de 16 bits para formar o endereço de memória. Para os loads, rt é o registrador de destino para o valor lido. Para os stores, rt é o registrador de origem cujo valor deve ser armazenado na memória. (c) Formato de instrução para branch equal (opcode = 4). Os registradores rs e rt são os registradores de origem que são comparados para igualdade. O campo address de 16 bits seu sinal estendido, é deslocado e somado ao PC para calcular o endereço de destino do desvio.

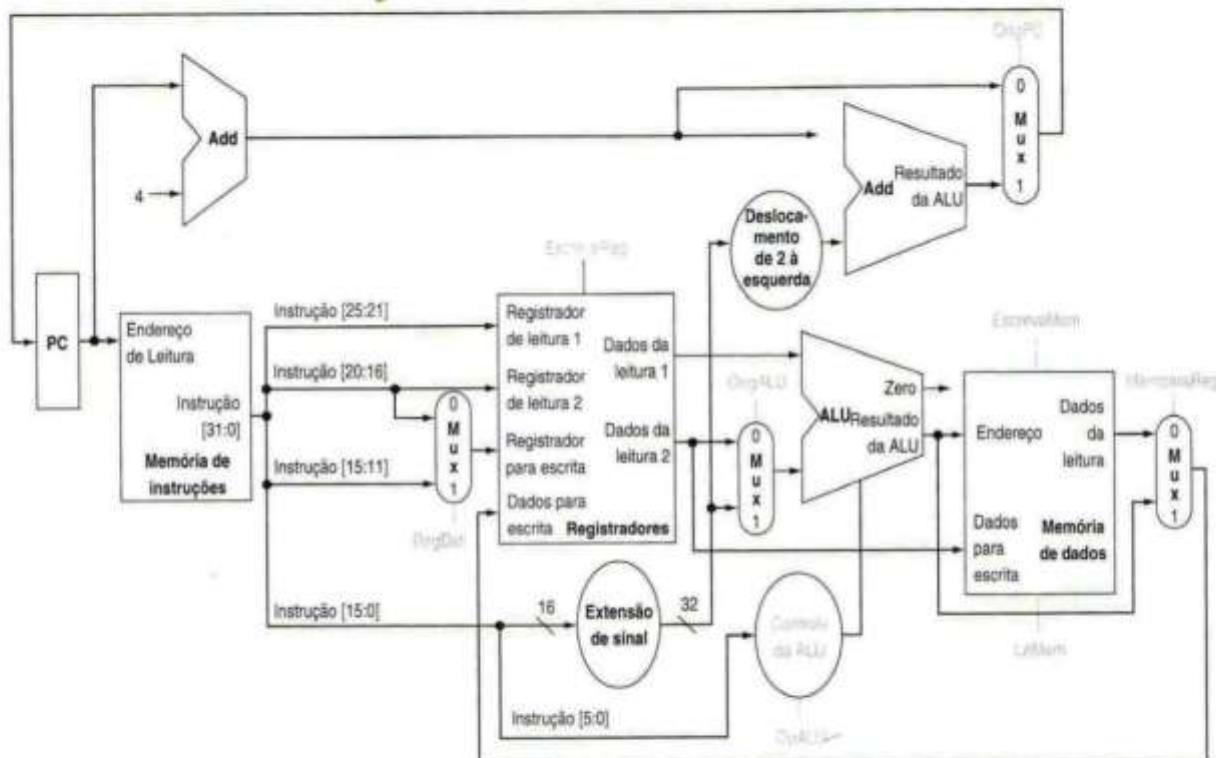


FIGURA 5.15 O caminho de dados da Figura 5.12 com todos os multiplexadores necessários e todas as linhas de controle identificadas. As linhas de controle são mostradas em cor. O bloco de controle da ALU também foi acrescentado. O PC não exige um controle de escrita, já que ele é escrito uma vez no fim de cada ciclo de clock; a lógica de controle de desvio determina se ele é escrito com o PC incrementado ou o endereço de destino do desvio.

mentos de estado, o sinal de leitura para a memória de dados e os sinais de controle para os multiplexadores. Como todos os multiplexadores possuem duas entradas, cada um deles requer uma única linha de controle.

A Figura 5.15 mostra sete linhas de controle de um único bit mais o sinal de controle OpALU de 2 bits. Já definimos como o sinal de controle OpALU funciona e é útil definir o que fazem os outros sete sinais de controle informalmente antes de determinarmos como definir esses sinais de controle durante a execução da instrução. A Figura 5.16 descreve a função dessas sete linhas de controle.

Nome do sinal	Efeito quando inativo	Efeito quando ativo
RegDst	O número do registrador destino para a entrada Registrador para escrita vem do campo rt (bits 20:16).	O número do registrador destino para a entrada Registrador para escrita vem do campo rd (bits 15:11).
EscreveReg	Nenhum.	O registrador na entrada Registrador para escrita é escrito com o valor da entrada Dados para escrita.
OrigALU	O segundo operando da ALU vem da segunda saída do banco de registradores (Dados da leitura 2).	O segundo operando da ALU consiste nos 16 bits mais baixos da Instrução com sinal estendido.
OrigPC	O PC é substituído pela saída do somador que calcula o valor de PC + 4.	O PC é substituído pela saída do somador que calcula o destino do desvio.
LeMem	Nenhum.	O conteúdo da memória de dados designado pela entrada Endereço é colocado na saída Dados da leitura.
EscreveMem	Nenhum.	O conteúdo da memória de dados designado pela entrada Endereço é substituído pelo valor na entrada Dados para escrita.
MemparaReg	O valor enviado para a entrada Dados para escrita do banco de registradores vem da ALU.	O valor enviado para a entrada Dados para escrita do banco de registradores vem da memória de dados.

FIGURA 5.16 O efeito de cada um dos sete sinais de controle. Quando o controle de bit 1 de largura, para um multiplexador com duas entradas, está ativo, o multiplexador seleciona a entrada correspondente a 1. Caso contrário, se o controle não estiver ativo, o multiplexador seleciona a entrada 0. Lembre-se de que todos os elementos de estado têm o clock como uma entrada implícita e que o clock é usado para controlar escritas. O clock nunca vem externamente para um elemento de estado, já que isso pode criar problemas de sincronização. (Veja o Apêndice B para obter mais detalhes sobre esse problema.)

Agora que examinamos a função de cada um dos sinais de controle, podemos ver como defini-los. A unidade de controle pode definir todos menos um dos sinais de controle unicamente com base no campo opcode da instrução. A exceção é a linha de controle OrigPC. Essa linha de controle deve ser ativada se a instrução for branch on equal (uma decisão que a unidade de controle pode tomar) e a saída Zero da ALU, usada para comparação de igualdade, for verdadeira. Para gerar o sinal OrigPC, precisaremos realizar um AND de um sinal da unidade de controle, que chamamos *Branch*, com o sinal Zero da ALU.

Esses nove sinais de controle (sete da Figura 5.16 e dois do OpALU) podem agora ser definidos baseados nos seis sinais de entrada da unidade de controle, que são os bits de opcode. A Figura 5.17 mostra o caminho de dados com a unidade de controle e os sinais de controle.

Antes de tentarmos escrever um conjunto de equações ou uma tabela verdade para a unidade de controle, será útil definir a função de controle informalmente. Como a definição das linhas de controle depende apenas do opcode, definimos se cada sinal de controle deve ser 0, 1 ou don't care (X) para cada um dos valores de opcode. A Figura 5.18 descreve como os sinais de controle devem ser definidos para cada opcode; essas informações seguem diretamente das Figuras 5.12, 5.16 e 5.17.

Operação do caminho de dados

Com as informações contidas nas Figuras 5.16 e 5.18, podemos projetar a lógica da unidade de controle. Antes de fazer isso, porém, vejamos como cada instrução usa o caminho de dados. Nas próximas figuras, mostramos o fluxo das três classes de instrução diferentes por meio do caminho de dados. Os sinais de controle ativos e os elementos do caminho de dados ativos são destacados em cada uma das figuras. Observe

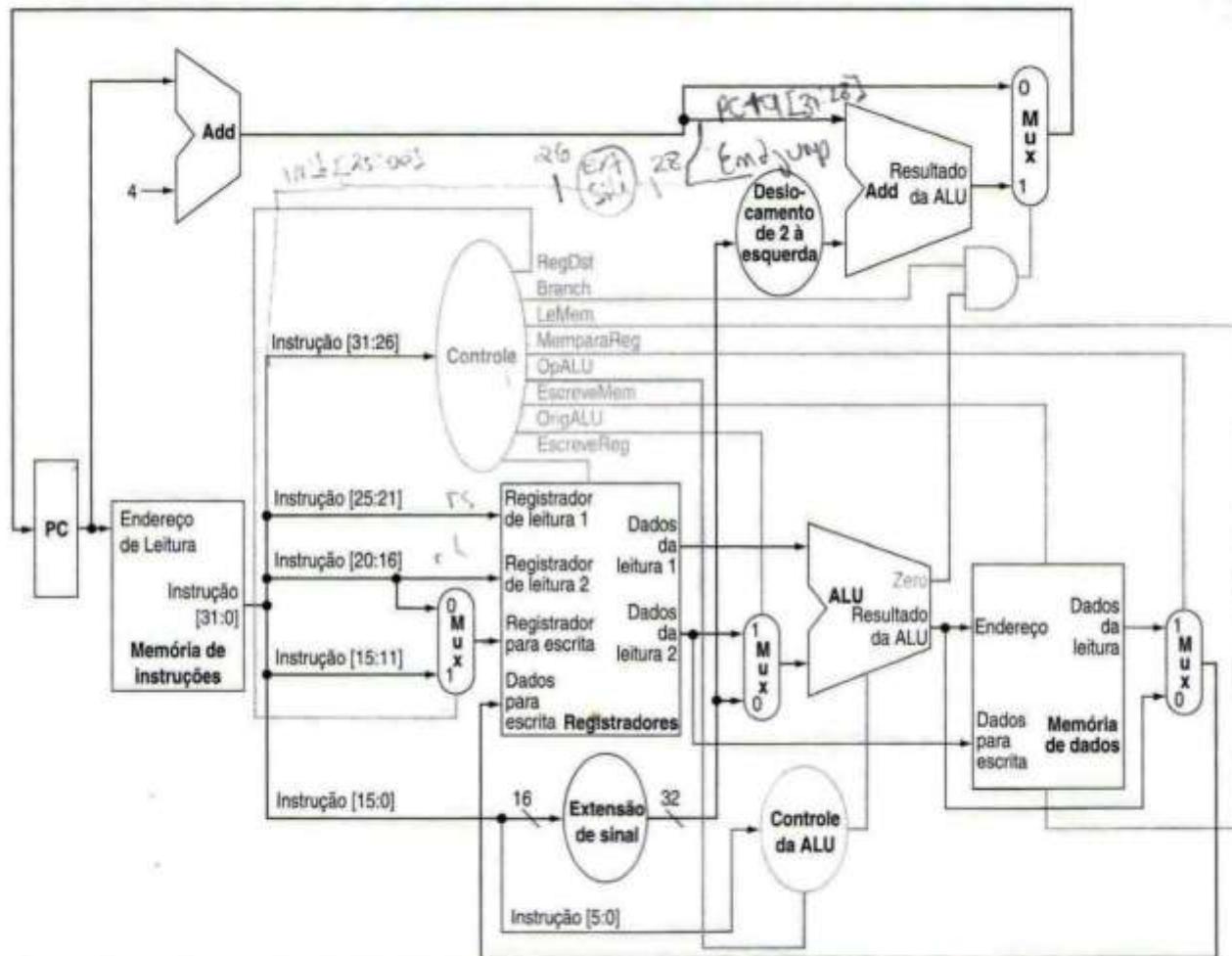


FIGURA 5.17 O caminho de dados simples com a unidade de controle. A entrada para a unidade de controle é o campo opcode de 6 bits da instrução. As saídas da unidade de controle consistem em três sinais de 1 bit usados para controlar multiplexadores (RegDst, OrigALU e MemparaReg), três sinais para controlar leituras e escritas no banco de registradores e na memória de dados (EscreveReg, LeMem e EscreveMem), um sinal de 1 bit usado na determinação de um possível desvio (Branch), e um sinal de controle de 2 bits para a ALU (OpALU). Uma porta AND é usada para combinar o sinal de controle de desvio com a saída Zero da ALU; a saída da porta AND controla a seleção do próximo PC. Observe que OrigPC é agora um sinal derivado, em vez de um sinal vindo diretamente da unidade de controle. Portanto, descartamos o nome do sinal nas próximas figuras.

que um multiplexador cujo controle é 0 tem uma ação definida, mesmo se sua linha de controle não estiver destacada. Sinais de controle de vários bits são destacados se qualquer sinal constituinte estiver ativo.

A Figura 5.19 mostra a operação do caminho de dados para uma instrução tipo R, como add \$t1,\$t2,\$t3. Embora tudo ocorra em 1 ciclo de clock, podemos pensar em quatro etapas para executar a instrução; essas etapas são ordenadas pelo fluxo da informação:

1. A instrução é buscada e o PC é incrementado.
2. Dois registradores, \$t2 e \$t3, são lidos do banco de registradores e a unidade de controle principal calcula a definição das linhas de controle também durante essa etapa.
3. A ALU opera nos dados lidos do banco de registradores, usando o código de função (bits 5:0, que é o campo funct, da instrução) para gerar a função da ALU.
4. O resultado da ALU é escrito no banco de registradores usando os bits 15:11 da instrução para selecionar o registrador de destino (\$t1).

Instrução	RegDst	OrigALU	MemparaReg	EscreveReg	LeMem	EscreveMem	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 5.18 A definição das linhas de controle é completamente determinada pelos campos opcode da instrução. A primeira linha da tabela corresponde às instruções formato R (add, sub, and, or e slt). Para todas essas instruções, os campos registradores de origem são rs e rt e o campo registrador de destino é rd; isso especifica como os sinais OrigALU e RegDst são definidos. Além disso, uma instrução tipo R escreve em um registrador (EscreveReg = 1), mas não escreve ou lê a memória de dados. Quando o sinal de controle Branch é 0, o PC é incondicionalmente substituído por PC + 4; caso contrário, o PC é substituído pelo destino do desvio se a saída Zero da ALU também está ativa. O campo OpALU para as instruções tipo R é definido como 10 para indicar que o controle da ALU deve ser gerado do campo funct. A segunda e a terceira linhas dessa tabela fornecem as definições dos sinais de controle para lw e sw. Esses campos OrigALU e OpALU são ativados para realizar o cálculo do endereço. LeMem e EscreveMem são ativados para realizar o acesso à memória. Finalmente, RegDst e EscreveReg são ativados para que um load faça o resultado ser armazenado no registrador rt. A instrução branch é semelhante à operação no formato R, já que ela envia os registradores rs e rt para a ALU. O campo OpALU para um desvio é definido como uma subtração (controle da ALU = 01), usada para testar a igualdade. Repare que o campo MemparaReg é irrelevante quando o sinal EscreveReg é 0: como o registrador não está sendo escrito, o valor dos dados na entrada Dados para escrita do banco de registradores não é usado. Portanto, a entrada MemparaReg nas duas últimas linhas da tabela é substituída por X (don't care). Os don't care também podem ser adicionados a RegDst quando EscreveReg é 0. Esse tipo de don't care precisa ser acrescentado pelo projetista, uma vez que ele depende do conhecimento de como o caminho de dados funciona.

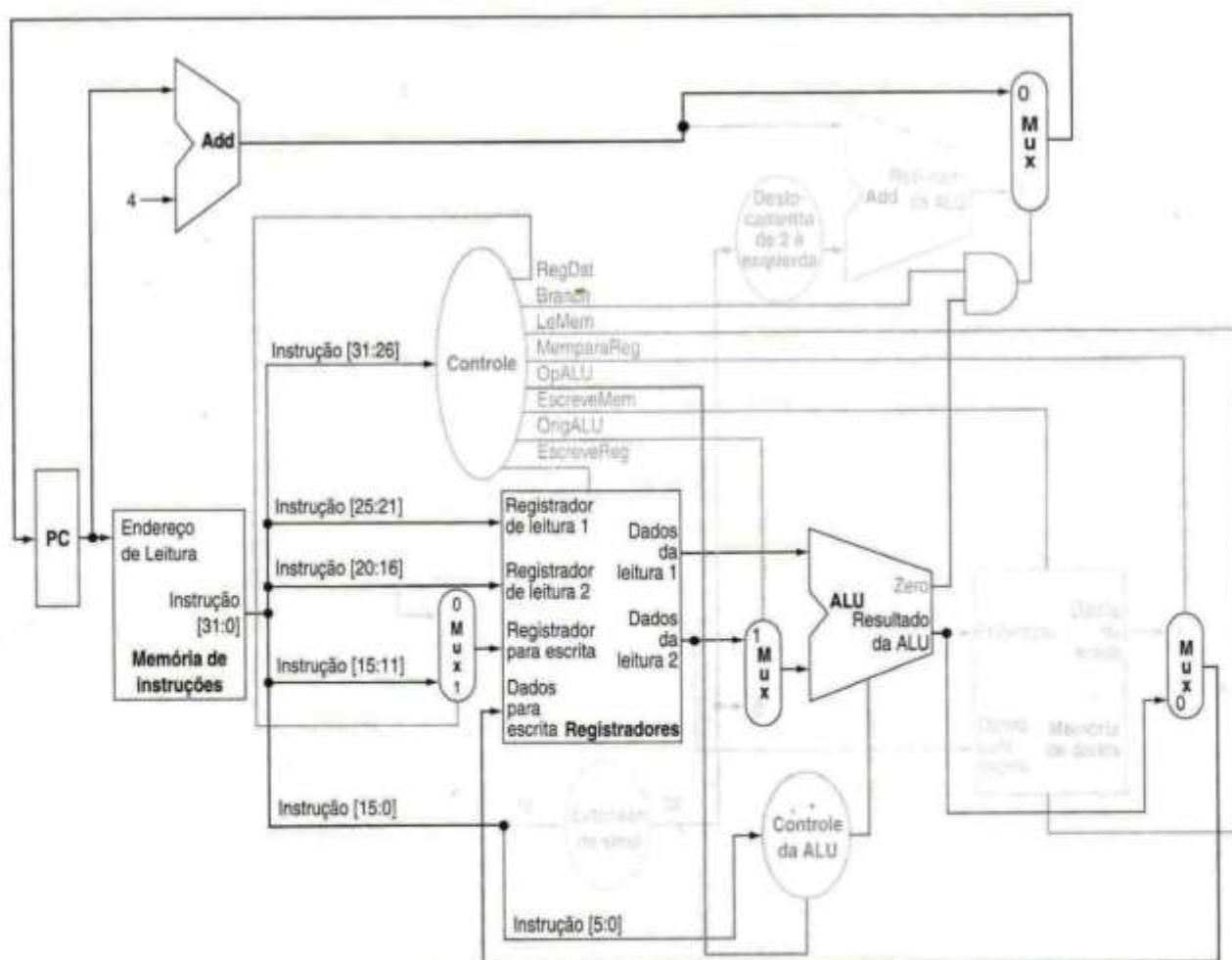


FIGURA 5.19 O caminho de dados em operação para uma instrução tipo R como add \$t1,\$t2,\$t3. As linhas de controle, as unidades do caminho de dados e as conexões que estão ativas aparecem destacadas.

Da mesma forma, podemos ilustrar a execução de um load word, como

`lw $t1, offset($t2)`

em um estilo semelhante à Figura 5.19. A Figura 5.20 mostra as unidades funcionais ativas e as linhas de controle ativas para um load. Podemos pensar em uma instrução load como operando em cinco etapas (semelhante ao tipo R executado em quatro):

1. Uma instrução é buscada da memória de instruções e o PC é incrementado.
2. Um valor de registrador ($\$t2$) é lido do banco de registradores.
3. A ALU calcula a soma do valor lido do banco de registradores com os 16 bits menos significativos com sinal estendido da instrução (offset).
4. A soma da ALU é usada como o endereço para a memória de dados.
5. Os dados da unidade de memória são escritos no banco de registradores; o registrador de destino é fornecido pelos bits 20:16 da instrução ($\$t1$).

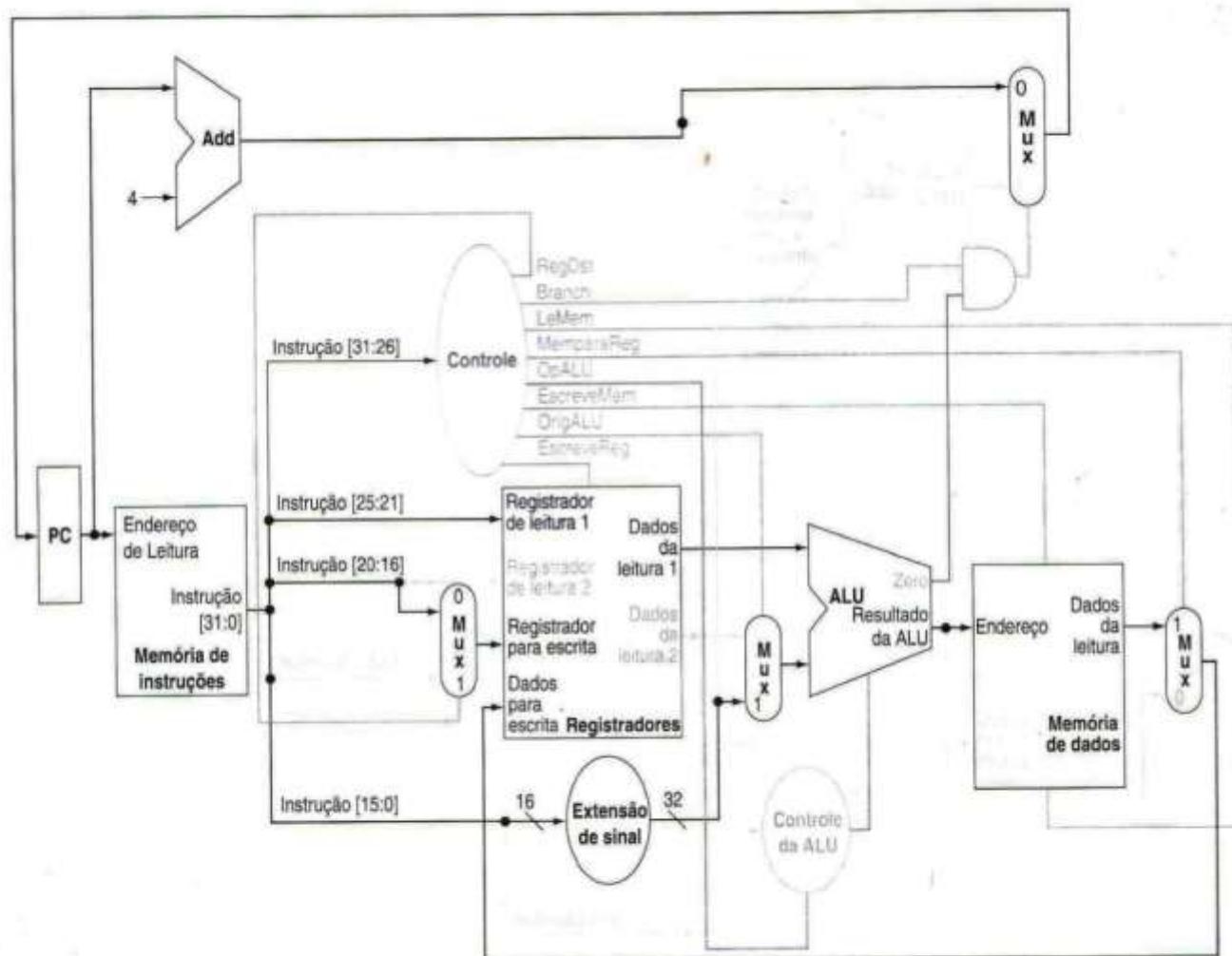


FIGURA 5.20 O caminho de dados em operação para uma instrução load. As linhas de controle, as unidades do caminho de dados e as conexões ativas aparecem destacadas. Uma instrução store operaria de maneira muito semelhante. A principal diferença seria que o controle da memória indicaria uma escrita em vez de uma leitura, a segunda leitura do valor de um registrador seria usada para os dados a serem armazenados e a operação de escrita do valor da memória de dados no banco de registradores não ocorreria.

Finalmente, podemos mostrar a operação da instrução branch-on-equal, como `beq $t1,$t2,offset`, da mesma maneira. Ela opera de forma muito parecida com uma instrução de formato R, mas a saída da ALU é usada para determinar se o PC é escrito com $PC + 4$ ou o endereço de destino do desvio. A Figura 5.21 mostra as quatro etapas da execução:

1. Uma instrução é buscada da memória de instruções e o PC é incrementado.
 2. Dois registradores, \$t1 e \$t2, são lidos do banco de registradores.
 3. A ALU realiza uma subtração dos valores de dados lidos do banco de registradores. O valor de $PC + 4$ é somado aos 16 bits menos significativos com sinal estendido (offset) deslocados de dois para a esquerda; o resultado é o endereço de destino do desvio.
 4. O resultado Zero da ALU é usado para decidir o resultado de que somador deve ser armazenado no PC.

Na próxima seção, examinaremos máquinas realmente seqüenciais, em especial aquelas em que cada uma dessas etapas é um ciclo de clock diferente.

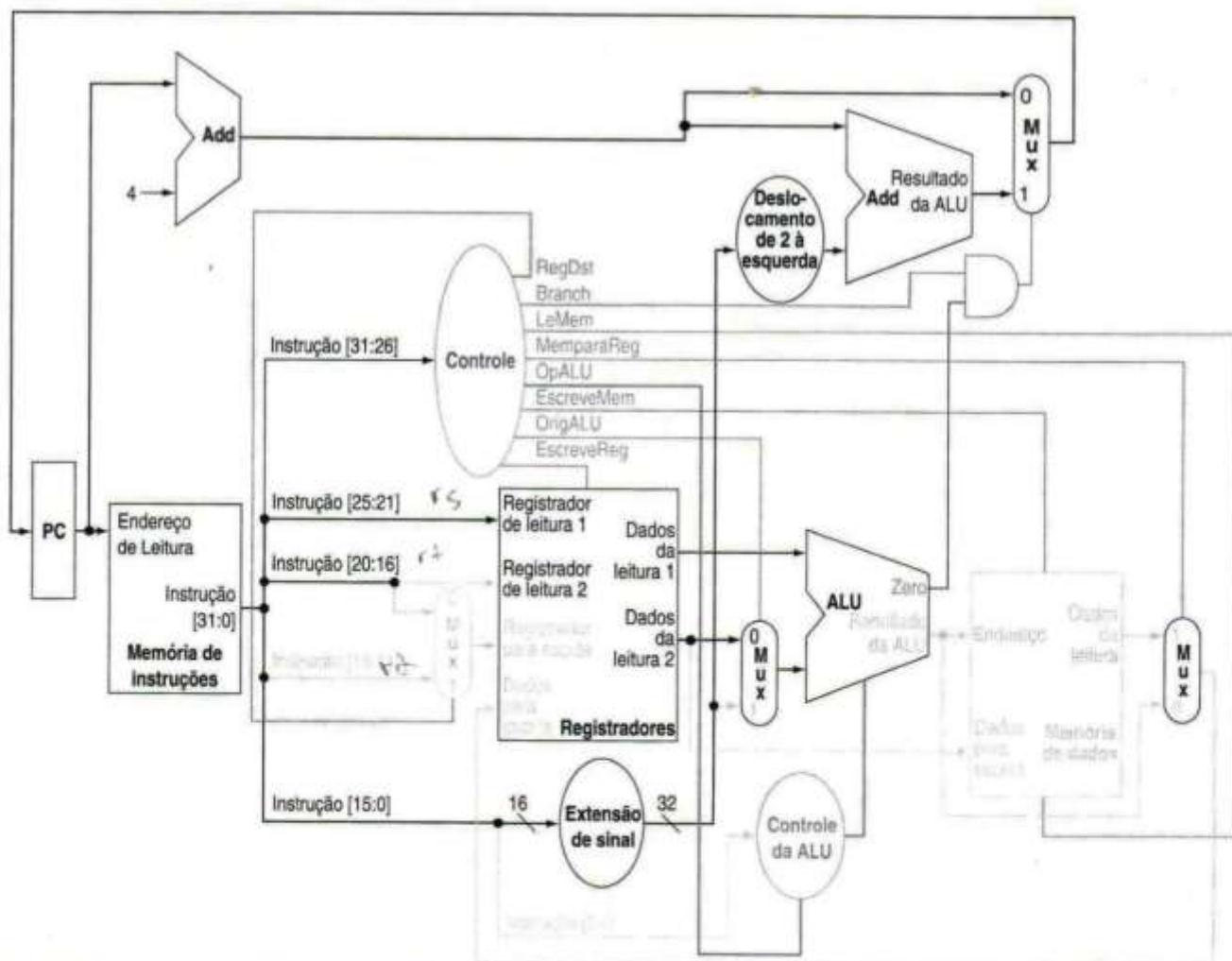


FIGURA 5.21 O caminho de dados em operação para uma instrução branch equal. As linhas de controle, as unidades do caminho de dados e as conexões que estão ativas aparecem destacadas. Após usar o banco de registradores e a ALU para realizar a comparação, a saída Zero é usada para selecionar o próximo contador de programa dentre os dois candidatos.

Finalizando o controle

Agora que vimos como as instruções operam em etapas, vamos continuar com a implementação do controle. A função de controle pode ser definida precisamente usando o conteúdo da Figura 5.18. As saídas são as linhas de controle e a entrada é o campo opcode de 6 bits, Op [5:0]. Portanto, podemos criar uma tabela verdade para cada uma das saídas com base na codificação binária dos opcodes.

A Figura 5.22 mostra a lógica na unidade de controle como uma grande tabela verdade que combina todas as saídas e que usa os bits de opcode como entradas. Ela especifica completamente a função de controle e podemos implementá-la diretamente em portas lógicas de uma maneira automatizada. Mostramos essa etapa final na Seção C.2 no Apêndice C.

Entrada ou saída	Nome do sinal	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
Saídas	RegDst	1	0	X	X
	OrigALU	0	1	1	0
	MemparaReg	0	1	X	X
	EscreveReg	1	1	0	0
	LeMem	0	1	0	0
	EscreveMem	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

FIGURA 5.22 A função de controle para a implementação simples de ciclo único é completamente especificada por essa tabela verdade. A parte superior da tabela fornece combinações de sinais de entrada que correspondem aos quatro opcodes que determinam as definições de saída do controle. (Lembre-se de que Op [5:0] corresponde aos bits 31:26 da instrução, que é o campo op.) A parte inferior da tabela fornece as saídas. Portanto, a saída EscreveReg é ativada para duas combinações diferentes das entradas. Se considerarmos apenas os quatro opcodes mostrados nessa tabela, então, poderemos simplificar a tabela verdade usando *don't care* na parte da entrada. Por exemplo, podemos detectar uma instrução no formato R com a expressão $Op5 \cdot Op2$, uma vez que isso é suficiente para distinguir as instruções no formato R das instruções lw, sw e beq. Não tiramos vantagem dessa simplificação, já que o restante dos opcodes MIPS é usado em uma implementação completa.

implementação simples de ciclo único Também chamada de implementação de ciclo de clock único. Uma implementação em que uma instrução é executada em um único ciclo de clock.

Agora, vamos acrescentar a instrução jump para mostrar como o caminho de dados básico e o controle podem ser estendidos para lidar com outras instruções no conjunto de instruções.

IMPLEMENTANDO JUMPS

A Figura 5.17 mostra a implementação de muitas das instruções vistas no Capítulo 2. Uma classe de instruções ausente é a da instrução jump. Estenda o caminho de dados e o controle da Figura 5.17 para incluir a instrução jump. Descreva como definir quaisquer novas linhas de controle.

EXEMPLO

RESPOSTA

A instrução jump se parece um pouco com uma instrução branch mas calcula o PC de destino de maneira diferente e não é condicional. Como um branch, os 2 bits menos significativos de um endereço jump são sempre 00_{bin} . Os próximos 26 bits menos significativos desse endereço de 32 bits vêm do campo de 26 bits imediato na instrução, como mostrado na Figura 5.23. Os 4 bits superiores do endereço que deve substituir o PC vêm do PC da instrução jump mais 4. Portanto, podemos implementar um jump armazenando no PC a concatenação de

- os 4 bits superiores do PC atual + 4 (esses são bits 31:28 do endereço da instrução imediatamente seguinte)
- o campo de 26 bits imediato da instrução jump
- os bits 00_{bin}

A Figura 5.24 mostra a adição do componente para jump à Figura 5.17. Um multiplexador adicional é usado para selecionar a origem para o novo valor do PC, que pode ser o PC incrementado ($PC + 4$), o PC de destino de um branch ou o PC de destino de um jump. Um sinal de controle adicional é necessário para o multiplexador adicional. Esse sinal de controle, chamado *Jump*, é ativado apenas quando a instrução é um jump – ou seja, quando o opcode é 2.

Campo	000010	address
Posições dos bits	31:26	25:0

FIGURA 5.23 Formato de instrução para a instrução jump (opcode = 2). O endereço de destino para uma instrução jump é formado pela concatenação dos 4 bits superiores do PC atual + 4 com o campo address de 26 bits na instrução jump e pela adição de 00 como os dois bits menos significativos.

Por que uma implementação de ciclo único não é usada hoje

Embora o projeto de ciclo único funcionasse corretamente, ele não seria usado nos projetos modernos porque é ineficiente. Para ver o porquê disso, observe que o ciclo de clock precisa ter a mesma duração para cada instrução nesse projeto de ciclo único e o CPI (Capítulo 4), portanto, será 1. É claro, o ciclo de clock é determinado pelo caminho mais longo possível na máquina. Esse caminho é, quase certamente, uma instrução load, que usa cinco unidades funcionais sem série: a memória de instruções, o banco de registradores, a ALU, a memória de dados e o banco de registradores. Embora o CPI seja 1, o desempenho geral de uma implementação de ciclo único provavelmente não será muito bom, já que várias das classes de instrução poderiam ficar em um ciclo de clock mais curto.

DESEMPENHO DAS MÁQUINAS DE CICLO ÚNICO

EXEMPLO

Suponha que os tempos de operação para as principais unidades funcionais nessa implementação sejam os seguintes:

- Unidades de memória: 200 picossegundos (ps)
- ALU e somadores: 100ps
- Banco de registradores (leitura ou escrita): 50ps

Considerando que os multiplexadores, a unidade de controle, os acessos do PC, a unidade de extensão de sinal e os fios não possuem atraso, qual das seguintes implementações seria mais rápida e por quanto?

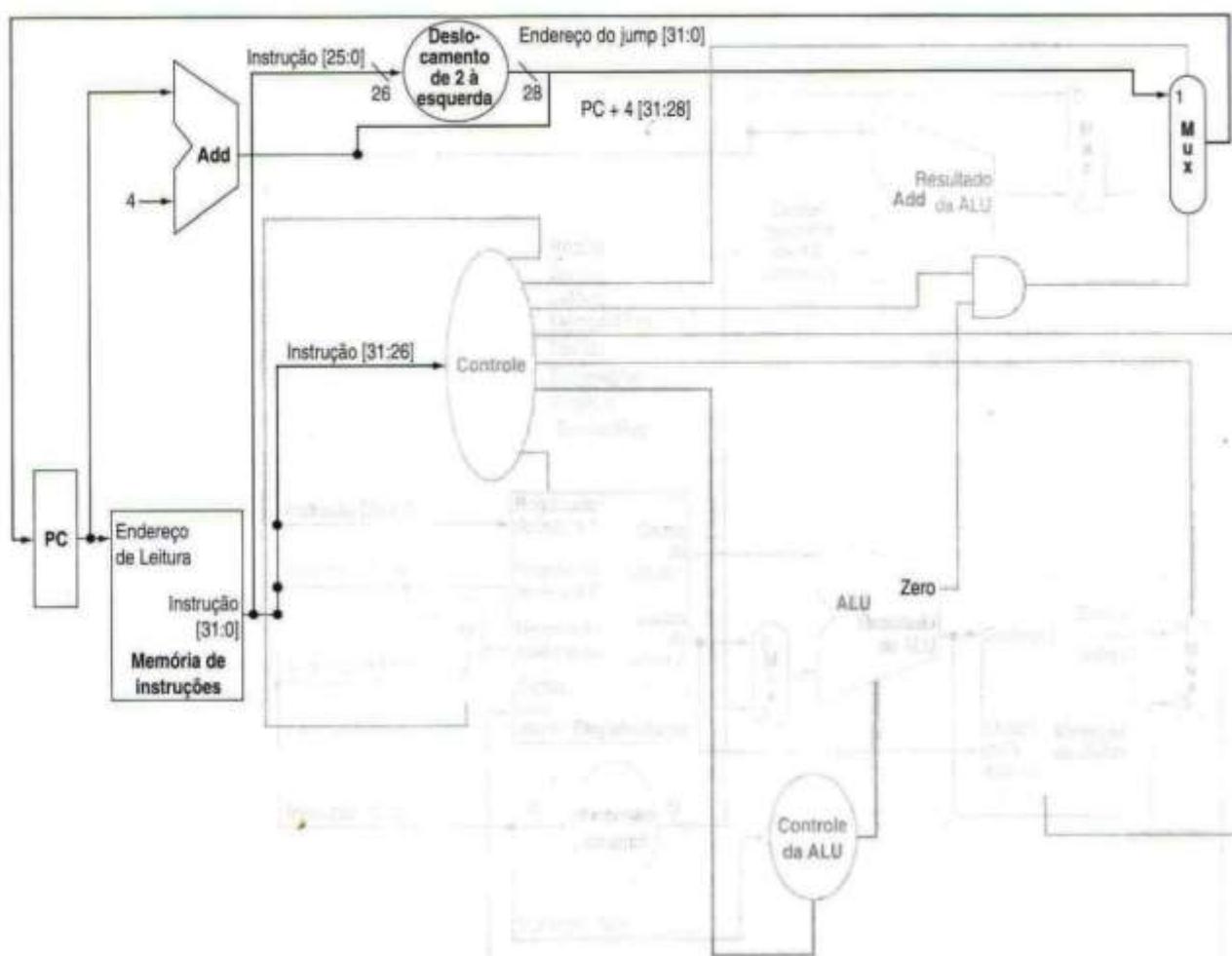


FIGURA 5.24 O controle e o caminho de dados simples são estendidos para lidar com a Instrução Jump. Um multiplexador adicional (no canto superior direito) é usado para escolher entre o destino de um jump e o destino de um desvio ou a instrução sequencial seguinte a esta. Esse multiplexador é controlado pelo sinal de controle Jump. O endereço de destino do jump é obtido deslocando-se os 26 bits inferiores da instrução jump de 2 bits para a esquerda, efetivamente adicionando 00 como os bits menos significativos, e, depois, concatenando os 4 bits mais significativos do PC + 4 como os bits mais significativos, produzindo, assim, um endereço de 32 bits.

1. Uma implementação em que toda instrução opera em 1 ciclo de clock de uma duração fixa.
2. Uma implementação em que toda instrução é executada em 1 ciclo de clock usando um clock de duração variável, que, para cada instrução, tem apenas a duração necessária. (Esse método não é incrivelmente prático, mas permitirá ver o que está sendo sacrificado quando todas as instruções precisam ser executadas com um clock único de mesma duração.)

Para comparar o desempenho, considere o seguinte mix de instruções: 25% loads, 10% stores, 45% instruções da ALU, 15% desvios e 5% jumps.

Vamos começar comparando os tempos de execução da CPU. Lembre-se do Capítulo 4 que

RESPOSTA

$$\text{Tempo de execução da CPU} = \text{Contagem de instruções} \times \text{CPI} \times \text{Tempo do ciclo de clock}$$

Como o CPI precisa ser 1, podemos simplificar isso para

$$\text{Tempo de execução da CPU} = \text{Contagem de instruções} \times \text{Tempo do ciclo de clock}$$

Precisamos apenas encontrar o tempo do ciclo de clock para as duas implementações, já que a contagem de instruções e o CPI são iguais para ambas as implementações. O caminho crítico para as diferentes classes de instrução é o seguinte:

Classe de Instrução	Unidades funcionais usadas pela classe de instrução				
tipo R	Busca de instrução	Acesso a registrador	ALU	Acesso a registrador	
Load word	Busca de instrução	Acesso a registrador	ALU	Acesso à memória	Acesso a registrador
Store word	Busca de instrução	Acesso a registrador	ALU	Acesso à memória	
Branch	Busca de instrução	Acesso a registrador	ALU		
Jump	Busca de instrução				

Usando esses caminhos críticos, podemos calcular o tamanho exigido para cada classe de instrução:

Classe de Instrução	Memória de Instrução	Leitura de registrador	Operação ALU	Memória de dados	Leitura de registrador	Total
Tipo R	200	50	100	0	50	400ps
Load word	200	50	100	200	50	600ps
Store word	200	50	100	200		550ps
Branch	200	50	100	0		350ps
Jump	200					200ps

O ciclo de clock para uma máquina com um único clock para todas as instruções será determinado pela instrução mais longa, que é 600ps. (Essa sincronização é aproximada, já que nosso modelo de sincronização é bastante simples. Na realidade, a sincronização dos sistemas digitais modernos é complexa.)

Uma máquina com um clock variável terá um ciclo de clock que varia entre 200ps e 600ps. Podemos encontrar a duração média do ciclo de clock para uma máquina com um clock de duração variável usando essas informações e a distribuição da frequência das instruções.

Portanto, o tempo médio por instrução com um clock variável é

$$\begin{aligned} \text{Ciclo de clock da CPU} &= 600 \times 25\% + 550 \times 10\% + 400 \times 45\% + 350 \times 15\% + 200 \times 5\% \\ &= 447,5 \text{ps} \end{aligned}$$

Como a implementação de clock variável possui um ciclo de clock médio mais curto, ela é claramente mais rápida. Vamos encontrar o fator de desempenho:

$$\begin{aligned} \frac{\text{Desempenho da CPU}_{\text{clock variável}}}{\text{Desempenho da CPU}_{\text{clock único}}} &= \frac{\text{Tempo de execução da CPU}_{\text{clock único}}}{\text{Tempo de execução da CPU}_{\text{clock variável}}} \\ &= \left(\frac{\text{CI} \times \text{Ciclo de clock da CPU}_{\text{clock único}}}{\text{CI} \times \text{Ciclo de clock da CPU}_{\text{clock variável}}} \right) = \frac{\text{Ciclo de clock da CPU}_{\text{clock único}}}{\text{Ciclo de clock da CPU}_{\text{clock variável}}} \\ &= \frac{600}{447,5} = 1,34 \end{aligned}$$

A implementação de clock variável seria 1,34 vez mais rápida. Infelizmente, implementar um clock de velocidade variável para cada classe de instrução é extremamente difícil e o overhead para esse método poderia ser maior do que qualquer vantagem obtida. Como veremos na próxima seção, uma alternativa é usar um ciclo de clock mais curto que realize menos trabalho e, então, variar o número de ciclos de clock para as diferentes classes de instrução.

O ônus de usar o projeto de ciclo único com um ciclo de clock fixo é significativo, mas poderia ser considerado aceitável para esse conjunto de instruções pequeno. Historicamente, as primeiras máquinas com conjuntos de instruções muito simples usavam essa tecnologia de implementação. Entretanto, se tentássemos implementar a unidade de ponto flutuante ou um conjunto de instruções com instruções mais complexas, esse projeto de ciclo único decididamente não funcionaria bem. Um exemplo disso é mostrado no Exercício 5.4 ■ Aprofundando o aprendizado.

Como precisamos considerar que o ciclo de clock é igual ao atraso de pior caso para todas as instruções, não podemos usar técnicas de implementação que reduzem o atraso do caso comum mas não melhoram o tempo de ciclo de pior caso. Uma implementação de ciclo único, portanto, viola um dos nossos principais princípios de projeto de tornar o caso comum mais rápido. Além disso, nessa implementação de ciclo único, cada unidade funcional precisa ser duplicada, o que eleva o custo da implementação. Um projeto de ciclo único é ineficiente tanto em seu desempenho quanto em seu custo de hardware!

Podemos evitar essas dificuldades usando técnicas de implementação que possuam um ciclo de clock mais curto – derivados dos atrasos da unidade funcional básica – e que exijam múltiplos ciclos de clock para cada instrução. A próxima seção explora esse esquema de implementação alternativo. No Capítulo 6, veremos outra técnica de implementação, chamada pipelining, que usa um caminho de dados muito parecido com o caminho de dados de ciclo único, mas que é muito mais eficiente. O pipelining ganha eficiência sobrepondo a execução de múltiplas instruções, aumentando a utilização do hardware e melhorando o desempenho. Para os leitores interessados principalmente nos conceitos de alto nível usados nos processadores, o material desta seção é suficiente para ler as seções introdutórias do Capítulo 6 e entender a funcionalidade básica de um processador em pipeline. Se quiser entender como o hardware realmente implementa o controle, vá em frente!

Veja os sinais de controle na Figura 5.22. Algum sinal de controle na figura pode ser substituído pelo inverso de outro? (Dica: leve em conta os don't care.) Nesse caso, você pode usar um sinal para o outro sem incluir um inverter?

**Verifique
você mesmo**

5.5 Uma implementação multiciclo

Em um exemplo anterior, dividimos cada instrução em uma série de etapas correspondentes às operações das unidades funcionais necessárias. Podemos usar essas etapas para criar uma **implementação multiciclo**. Em uma implementação multiciclo, cada *etapa* na execução levará 1 ciclo de clock. A implementação multiciclo permite que uma unidade funcional seja usada mais de uma vez por instrução, desde que seja usada em diferentes ciclos de clock. Esse compartilhamento pode ajudar a reduzir a quantidade de hardware necessária. A capacidade de permitir que instruções usem diferentes números de ciclos de clock e a capacidade de compartilhar unidades funcionais dentro da execução de uma única instrução são as principais vantagens de um projeto multiciclo. A Figura 5.25 mostra a versão abstrata do caminho de dados multiciclo. Se compararmos a Figura 5.25 com o caminho de dados para a versão de ciclo único na Figura 5.11, poderemos ver as seguintes diferenças:

Implementação multiciclo Também chamada implementação com ciclo de clock múltiplo. Uma implementação em que uma instrução é executada em múltiplos ciclos de clock.

- Uma única unidade de memória é usada para instruções e para dados.
- Existe uma única ALU, em vez de uma ALU e dois somadores.
- Um ou mais registradores são adicionados após cada unidade funcional para conter a saída dessa unidade até o valor ser usado em um ciclo de clock subsequente.

No final de um ciclo de clock, todos os dados usados nos ciclos de clock subsequentes precisam ser armazenados em um elemento de estado. Os dados usados pelas *instruções subsequentes* em um ciclo de clock posterior são armazenados em um dos elementos de estado visíveis ao programador: o



FIGURA 5.25 A visão de alto nível do caminho de dados multiciclo. Esta figura mostra os elementos-chave do caminho de dados: uma unidade de memória compartilhada, uma única ALU compartilhada entre instruções e as conexões entre essas unidades compartilhadas. O uso de unidades funcionais compartilhadas requer a adição ou o aumento da largura dos multiplexadores, bem como novos registradores temporários que contenham os dados entre ciclos de clock da mesma instrução. Os registradores adicionais são o registrador de instrução (IR), o registrador de dados da memória (MDR), A, B e SaídaALU.

banco de registradores, o PC ou a memória. Por outro lado, os dados usados pela *mesma instrução* em um ciclo posterior precisam ser armazenados em um desses registradores adicionais.

Portanto, a posição dos registradores adicionais é determinada por dois fatores: que unidades combinacionais cabem em um ciclo de clock e que dados são necessários em ciclos posteriores implementando a instrução. Nesse projeto multiciclo, consideramos que o ciclo de clock pode acomodar no máximo uma das seguintes operações: um acesso à memória, um acesso ao banco de registradores (duas leituras e uma escrita), ou uma operação da ALU. Consequentemente, quaisquer dados produzidos por uma dessas três unidades funcionais (memória, banco de registradores ou ALU) precisam ser salvos em um registrador temporário para uso em um ciclo posterior. Se eles não fossem salvos, poderia haver a possibilidade de uma disputa de sincronização, levando ao uso de um valor incorreto.

Os seguintes registradores temporários são acrescentados para atender a esses requisitos:

- O registrador de instrução (IR) e o registrador de dados da memória (MDR) são incluídos para salvar a saída da memória para uma leitura de instrução e uma leitura de dados, respectivamente. Dois registradores separados são usados, já que, como ficará claro em breve, ambos os valores são necessários durante o mesmo ciclo de clock.
- Os registradores A e B são usados para conter os valores dos registradores operandos lidos do banco de registradores.
- O registrador SaídaALU contém a saída da ALU.

Todos os registradores exceto o IR contêm dados apenas entre um par de ciclos de clock adjacentes e, portanto, não precisarão de um sinal de controle de escrita. O IR precisa conter a instrução até o fim da execução dessa instrução e, então, exigirá um sinal de controle de escrita. Essa distinção se tornará mais clara quando mostrarmos os ciclos de clock individuais para cada instrução.

Como várias unidades funcionais são compartilhadas para diferentes finalidades, precisamos de ambos: incluir multiplexadores e expandir os multiplexadores existentes. Por exemplo, como uma única memória é usada para instruções e para dados, precisamos de um multiplexador para selecionar entre as duas origens para um endereço de memória, especificamente, o PC (para acesso a instrução) e SaídaALU (para acesso a dados).

Substituir as três ALUs do caminho de dados de ciclo único por uma única ALU significa que a única ALU precisa acomodar todas as entradas que, antes, iam para as três ALUs diferentes. Manipular as entradas adicionais exige duas mudanças no caminho de dados:

- Um multiplexador adicional é incluído para a primeira entrada da ALU. O multiplexador escolhe entre o registrador A e o PC.
- O multiplexador na segunda entrada da ALU muda de duas para quatro entradas. As duas entradas adicionais para o multiplexador são a constante 4 (usada para incrementar o PC) e o campo offset com sinal estendido e deslocado (usado no cálculo do endereço de desvio).

A Figura 5.26 mostra os detalhes do caminho de dados com esses multiplexadores adicionais. Introduzindo alguns registradores e multiplexadores, podemos reduzir o número de unidades de memória de dois para um e eliminar dois somadores. Visto que os registradores e multiplexadores são muito pequenos comparados com uma unidade de memória ou ALU, isso poderia produzir uma redução substancial no custo do hardware.

Como o caminho de dados mostrado na Figura 5.26 usa múltiplos ciclos de clock por instrução, ele exigirá um conjunto diferente de sinais de controle. As unidades de estado visíveis ao programador (o PC, a memória e os registradores), bem como o IR, precisarão de sinais de controle de escrita. A memória também precisará de um sinal de leitura. Podemos usar a unidade de controle da ALU do caminho de dados de ciclo único (veja a Figura 5.13 e o Apêndice C) para controlar a ALU aqui também. Finalmente, cada um dos multiplexadores de duas entradas exige uma única linha de controle, enquanto o multiplexador de quatro entradas exige duas linhas de controle. A Figura 5.27 mostra o caminho de dados da Figura 5.26 com essas linhas de controle acrescentadas.

O caminho de dados multiciclo ainda exige adições para suportar desvios e jumps; após essas adições, veremos como as instruções são seqüenciadas e, então, geraremos o controle do caminho de dados.

Com a instrução jump e a instrução branch, há três origens possíveis para o valor a ser escrito no PC:

- A saída da ALU, que é o valor $PC + 4$ durante a busca da instrução. Esse valor deve ser armazenado diretamente no PC.
- O registrador SaídaALU, que é onde armazenaremos o endereço de destino do desvio após ele ser calculado.

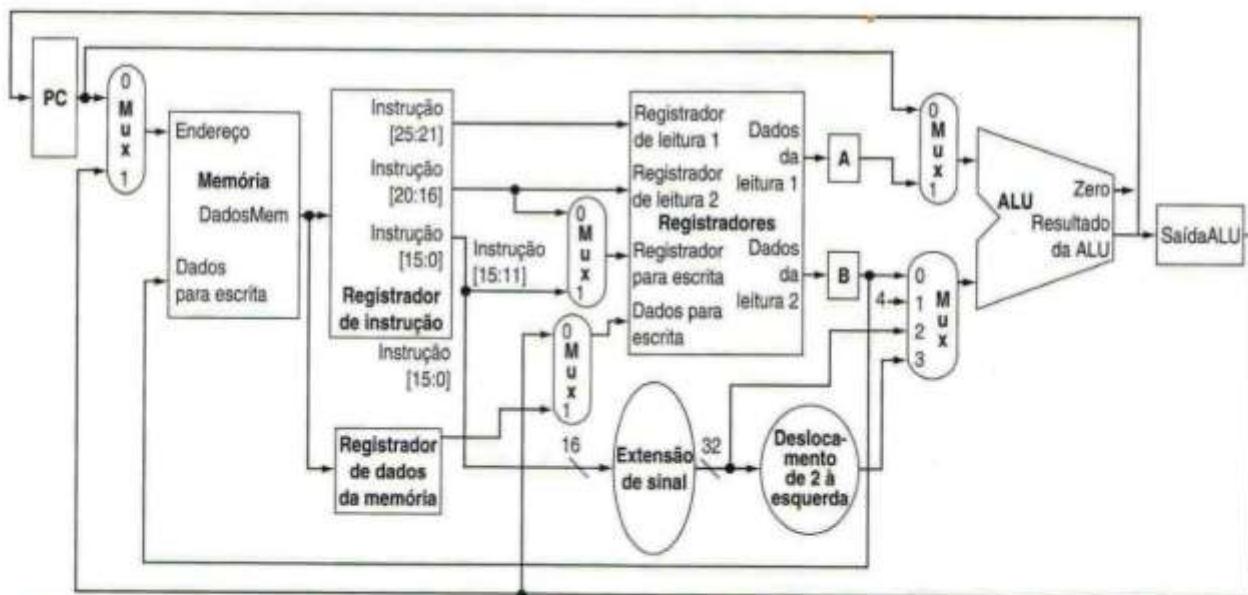


FIGURA 5.26 O caminho de dados multistep para o MIPS manipular as instruções básicas. Embora esse caminho de dados suporte incremento normal do PC, mais algumas conexões e um multiplexador serão necessários para desvios e jumps; iremos acrescentá-los em seguida. As adições comparadas com o caminho de dados de ciclo único incluem vários registradores (IR, MDR, A, B, SaídaALU), um multiplexador para o endereço de memória, um multiplexador para a entrada superior da ALU e a expansão do multiplexador na entrada inferior da ALU para um seletor de quatro entradas. Essas pequenas adições nos permitem remover dois somadores e uma unidade de memória.

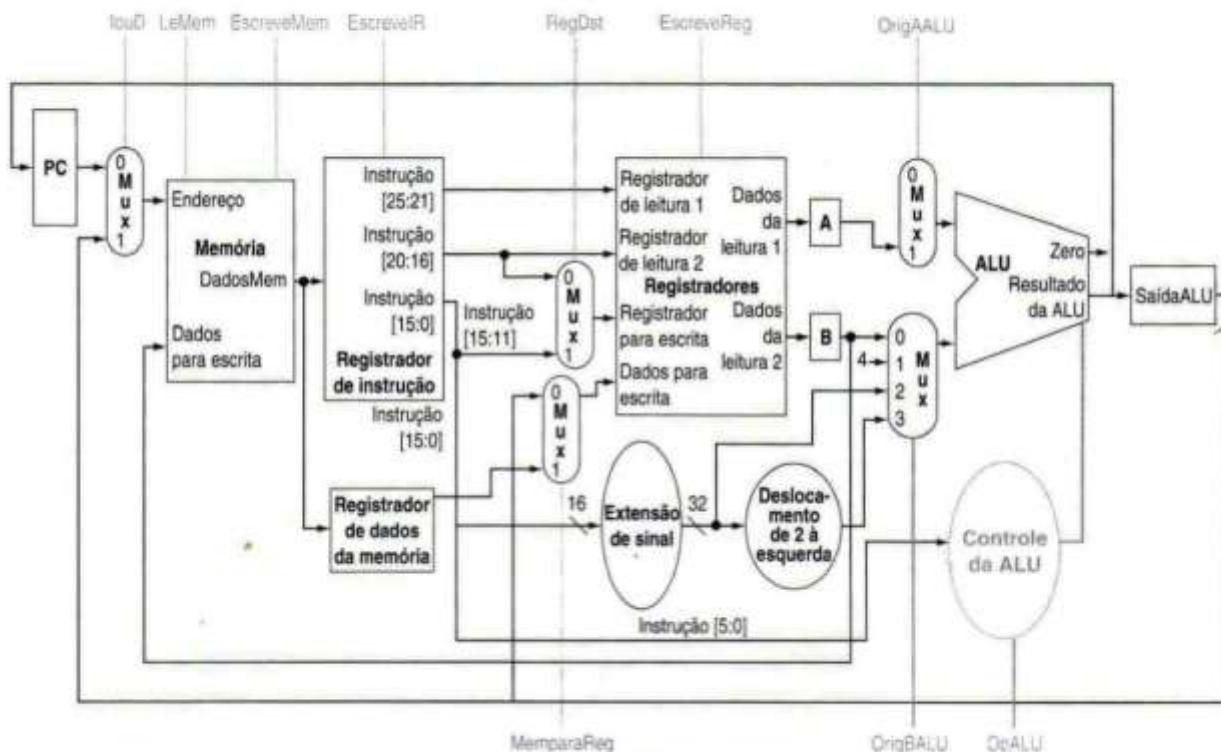


FIGURA 5.27 O caminho de dados multiciclo da Figura 5.26 com as linhas de controle indicadas. Os sinais OpALU e Orig BALU são sinais de controle de 2 bits, enquanto todas as outras linhas de controle são sinais de 1 bit. Nem o registrador A nem o registrador B requer um sinal de escrita, já que seu conteúdo é lido apenas no ciclo imediatamente após ele ser escrito. O registrador de dados da memória foi acrescentado para conter os dados de um load quando os dados retornam da memória. Os dados de um load retornando da memória não podem ser escritos diretamente no banco de registradores, já que o ciclo de clock não pode acomodar o tempo necessário para o acesso à memória e a escrita no banco de registradores. O sinal LeMem foi movido para o alto da unidade de memória para simplificar as figuras. O conjunto completo de caminhos de dados e linhas de controle para desvios será acrescentado em breve.

3. Os 26 bits menos significativos do registrador de instrução (IR) deslocados de 2 à esquerda e concatenados com os 4 bits mais significativos do PC incrementado, que é a origem quando a instrução é um jump.

Como observamos quando implementamos o controle de ciclo único, o PC é escrito incondicionalmente. Durante um incremento normal e para jumps, o PC é escrito incondicionalmente. Se a instrução é um desvio condicional, o PC incrementado é substituído pelo valor em SaidaALU apenas se os dois registradores designados forem iguais. Portanto, nossa implementação usa dois sinais de controle separados: EscrevePC, que causa uma escrita incondicional no PC, e EscrevePCCond, que causa uma escrita no PC se a condição de desvio também for verdadeira.

Precisamos conectar esses dois sinais de controle com o controle de escrita do PC. Exatamente como fizemos no caminho de dados de ciclo único, usaremos algumas portas lógicas para derivar o sinal de controle de escrita do PC de EscrevePC, de EscrevePCCond e do sinal Zero da ALU, que é usado para detectar se os dois registradores operando de um beq são iguais. Para determinar se o PC deve ser escrito durante um desvio condicional, realizamos um AND do sinal Zero da ALU com o EscrevePCCond. A saída dessa porta lógica AND, então, realiza um OR com EscrevePC, que é o sinal de escrita incondicional do PC. A saída dessa porta lógica OR é conectada com o sinal de controle de escrita para o PC.

A Figura 5.28 mostra o caminho de dados multiciclo completo e a unidade de controle, incluindo os sinais de controle e os multiplexadores adicionais para implementar a atualização do PC.

Antes de examinarmos as etapas para executar cada instrução, vamos olhar informalmente o efeito de todos os sinais de controle (como fizemos para o projeto de ciclo único da Figura 5.16). A Figura 5.29 mostra o que cada sinal de controle faz quando ativo e inativo.

Detalhamento: para reduzir o número de linhas de sinal interconectando as unidades funcionais, os projetistas podem usar *barramentos compartilhados*. Um barramento compartilhado é um conjunto de linhas que conectam várias unidades; na maioria dos casos, eles incluem diversas origens, que podem colocar dados no barramento, e diversos leitores do valor. Assim como reduzimos o número de unidades funcionais para o caminho de dados, podemos reduzir o número de barramentos interconectando essas unidades por meio do compartilhamento dos barramentos. Por exemplo, existem seis origens vindas da ALU; entretanto, apenas duas são necessárias em um determinado momento. Assim, um par de barramentos pode ser usado para conter valores enviados para a ALU. Em vez de colocar um grande multiplexador na frente da ALU, o projetista pode usar um barramento compartilhado e, então, garantir que apenas uma das origens esteja utilizando o barramento em qualquer ponto. Embora isso economize linhas de sinal, o mesmo número de linhas de controle será necessário para controlar o que ocorre no barramento. A principal desvantagem de usar essas estruturas de barramento é um possível prejuízo de desempenho, já que um barramento provavelmente não é tão rápido quanto uma conexão ponto a ponto.

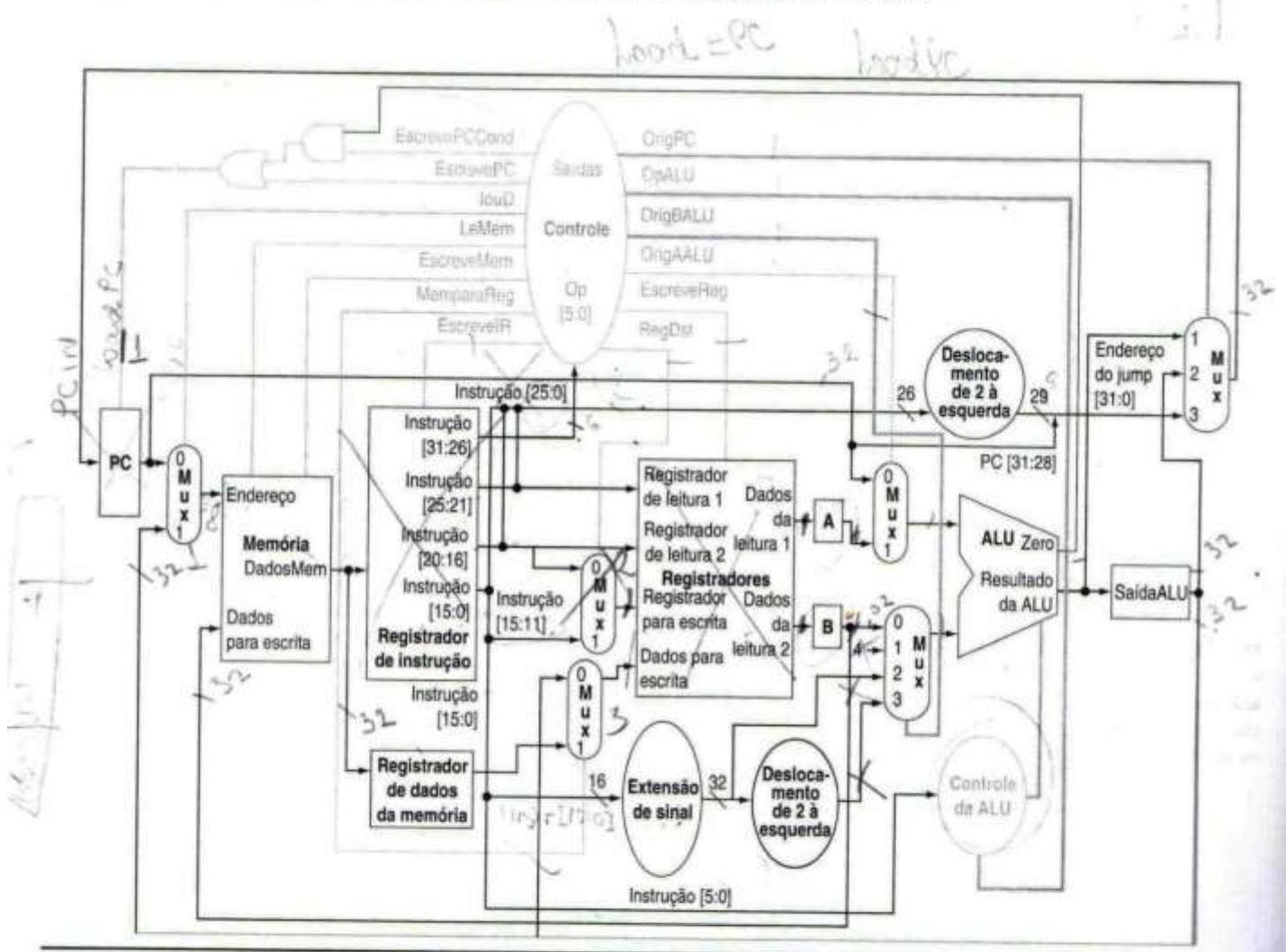


FIGURA 5.28 O caminho de dados completo para a implementação multiciclo juntamente com as linhas de controle necessárias. As linhas de controle da Figura 5.27 estão conectadas com a unidade de controle, e estão incluídos os elementos de controle e o caminho de dados necessários para efetuar as mudanças no PC. As principais adições em relação à Figura 5.27 são o multiplexador usado para selecionar a origem de um novo valor do PC, as portas lógicas usadas para combinar os sinais de escrita do PC e os sinais de controle OrigPC, EscrevePCCond. O sinal EscrevePCCond é usado para decidir se um desvio condicional deve ser tomado. Também está incluído o suporte para jumps.

Ações dos sinais de controle de 1 bit

Nome do sinal	Efeito quando inativo	Efeito quando ativo
RegDst	O número do registrador de destino do banco de registradores para a entrada Registrador para escrita vem do campo rt.	O número do registrador de destino do banco de registradores para a entrada Registrador para escrita vem do campo rd.
EscreveReg	Nenhum.	O registrador de uso geral selecionado pelo número na entrada Registrador para escrita é escrito com o valor da entrada Dados para escrita.
OrigALU	O primeiro operando da ALU é o PC.	O primeiro operando da ALU vem do registrador A.
LeMem	Nenhum.	O conteúdo da memória no local especificado pela entrada Endereço é colocado na saída Dados da memória.
EscreveMem	Nenhum.	O conteúdo da memória no local especificado pela entrada Endereço é substituído pelo valor na entrada Dados para escrita.
MemparaReg	O valor enviado para a entrada Dados para escrita do banco de registradores vem de SaídaALU.	O valor enviado para a entrada Dados para escrita do banco de registradores vem do MDR.
louD	O PC é usado para fornecer o endereço para a unidade de memória.	SaídaALU é usado para fornecer o endereço para a unidade de memória.
IR Write	Nenhum.	A saída da memória é escrita no IR.
EscrevePC	Nenhum.	O PC é escrito; a origem é controlada por OrigPC.
EscrevePCCond	Nenhum.	O PC é escrito se a saída Zero da ALU também estiver ativa.

Ações dos sinais de controle de 2 bits

Nome do sinal	Valor (binário)	Efeito
OpALU	00	A ALU realiza uma operação de adição.
	01	A ALU realiza uma operação de subtração.
	10	O campo funct da instrução determina a operação da ALU.
OrigBALU	00	A segunda entrada para a ALU vem do registrador B.
	01	A segunda entrada da ALU é a constante 4.
	10	A segunda entrada da ALU são os 16 bits menos significativos com sinal estendido do IR.
	11	A segunda entrada da ALU são os 16 bits menos significativos com sinal estendido do IR deslocados de 2 bits para a esquerda.
OrigPC	00	A saída da ALU (PC + 4) é enviada ao PC para escrita.
	01	O conteúdo de SaídaALU (o endereço de destino do desvio) é enviado ao PC para escrita.
	10	O endereço de destino do jump (IR[25:0]) deslocado de 2 bits para a esquerda e concatenado com PC + 4[31:28] é enviado ao PC para escrita.

FIGURA 5.29 A ação causada pelo valor de cada sinal de controle da Figura 5.28. A tabela superior descreve os sinais de controle de 1 bit, enquanto a tabela inferior descreve os sinais de 2 bits. Apenas as linhas de controle que afetam os multiplexadores possuem uma ação quando estão inativas. Essas informações são semelhantes às da Figura 5.16 para o caminho de dados de ciclo único, mas incluem várias linhas de controle novas (EscreveIR, EscrevePC, EscrevePCCond, OrigBALU e OrigPC) e excluem as linhas de controle que não são mais usadas ou que foram substituídas (OrigPC, Branch e Jump).

Dividindo a execução da instrução em ciclos de clock

Dado o caminho de dados na Figura 5.28, podemos agora olhar o que deve acontecer em cada ciclo de clock da execução multiciclo, já que isso determinará que sinais de controle adicionais podem ser necessários, bem como a definição dos sinais de controle. Nossa objetivo em dividir a execução em ciclos de clock deve ser maximizar o desempenho. Podemos começar dividindo a execução de qualquer instrução em uma série de etapas, cada uma usando um ciclo de clock, tentando manter a quantidade de trabalho por ciclo aproximadamente igual. Por exemplo, iremos limitar cada etapa para conter no máximo uma operação da ALU ou um acesso ao banco de registradores, ou um acesso à memória. Com essa restrição, o ciclo de clock pode ser tão curto quanto a mais longa dessas operações.

Lembre-se de que no final de cada ciclo de clock, quaisquer valores de dados necessários em um ciclo subsequente precisam ser armazenados em um registrador, que pode ser um dos elementos de estado principais (por exemplo, o PC, o banco de registradores ou a memória), um registrador temporário escrito em cada ciclo de clock (por exemplo, A, B, MDR ou SaídaALU) ou um registrador temporário com controle de escrita (por exemplo, IR). Lembre-se também de que, devido ao nosso projeto ser acionado por transição, podemos continuar a ler o valor atual de um registrador; o novo valor não parece até o próximo ciclo de clock.

No caminho de dados de ciclo único, cada instrução usa um conjunto de elementos do caminho de dados para realizar sua execução. Muitos dos elementos do caminho de dados operam em série, usando a saída de outro elemento como entrada. Alguns elementos operam em paralelo; por exemplo, o PC é incrementado e a instrução é lida ao mesmo tempo. Uma situação semelhante existe no caminho de dados multiciclo. Todas as operações listadas em uma etapa ocorrem em paralelo dentro de 1 ciclo de clock, enquanto etapas sucessivas operam em série em diferentes ciclos de clock. A limitação de uma operação da ALU, um acesso à memória e um acesso ao banco de registradores determina o que pode entrar em uma etapa.

Observe que fazemos distinção entre ler do ou escrever no PC ou em um dos registradores independentes e ler do ou escrever no banco de registradores. No primeiro caso, a ação de ler ou escrever é parte de um ciclo de clock, enquanto ler ou escrever um resultado no banco de registradores exige um ciclo de clock adicional. A razão para essa distinção é que o banco de registradores possui overhead adicional de controle e acesso se comparado com os registradores únicos independentes. Portanto, manter o ciclo de clock curto motiva dedicar ciclos de clock separados para acessos ao banco de registradores.

As possíveis etapas de execução e suas ações são descritas a seguir. Cada instrução MIPS precisa de três a cinco dessas etapas:

1. Etapa de busca da instrução

Buscar a instrução da memória e calcular o endereço da próxima instrução seqüencial:

```
IR <= Memória[PC];  
PC <= PC + 4;
```

Operação: enviar o PC para a memória como o endereço, realizar uma leitura e escrever a instrução no Registro de Instrução (IR), onde ele será armazenado. Além disso, incrementar o PC em 4. Usamos o símbolo “<=” da Verilog; ele indica que todo o lado direito é avaliado e, depois, todas as atribuições são feitas, que, efetivamente, é como o hardware é executado durante o ciclo de clock.

Para implementar essa etapa, precisaremos ativar os sinais de controle LeMem e EscreveIR e colocar IoUD em 0 para selecionar o PC como a origem do endereço. Também incrementamos o PC em 4, o que exige colocar o sinal OrigAALU em 0 (enviando o PC para a ALU), o sinal OrigBALU em 01 (enviando 4 para a ALU) e OpALU em 00 (para fazer a ALU somar). Finalmente, também desejaremos armazenar o endereço de instrução incrementado de volta no PC, o que exige colocar a origem do PC em 00 e ativar EscrevePC. O incremento do PC e o acesso à memória de instruções podem ocorrer em paralelo. O novo valor do PC não é visível até o próximo ciclo de clock. (O PC incrementado também será armazenado em SaídaALU, mas essa ação é benigna.)

2. Etapa de decodificação da instrução e busca dos registradores

Na etapa anterior e nesta etapa, ainda não sabemos qual é a instrução e, portanto, só podemos realizar ações aplicáveis a todas as instruções (como a busca da instrução na etapa 1) ou que não sejam nocivas, no caso de a instrução não ser o que pensamos que possa ser. Assim, nesta etapa, podemos ler os dois registradores indicados pelos campos de instrução rs e rt, já que não é nocivo lê-los mesmo que não seja necessário. Como os valores lidos do banco de registradores podem ser necessários em etapas futuras, nós os lemos do banco de registradores e os armazenamos nos registradores temporários A e B.

Também iremos calcular o endereço de destino do desvio com a ALU, que também não é nocivo porque podemos ignorar o valor se a instrução não for um desvio. O destino do desvio potencial é salvo em SaídaALU.

Realizar essas ações “otimistas” previamente tem a vantagem de diminuir o número de ciclos de clock necessários para executar uma instrução. Podemos realizar essas ações otimistas previamente devido à regularidade dos formatos de instrução. Por exemplo, se a instrução tiver dois registradores como entrada, eles estarão sempre nos campos rs e rt, e, se a instrução for um desvio (branch), o offset será sempre os 16 bits menos significativos:

```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
SaídaALU <= PC + (estende-sinal (IR[15:0]) << 2);
```

Operação: acessar o banco de registradores para ler os registradores rs e rt e armazenar os resultados nos registradores A e B. Como A e B são substituídos a cada ciclo, o banco de registradores pode ser lido a cada ciclo com os valores armazenados em A e B. Essa etapa também calcula o endereço de destino do desvio e armazena o endereço em SaídaALU, onde será usado no próximo ciclo de clock se a instrução for um branch. Isso exige colocar OrigAALU em 0 (de modo que o PC seja enviado para a ALU), OrigBALU em 11 (de modo que o campo de offset de sinal estendido e deslocado seja enviado para a ALU) e OpALU em 00 (de modo que a ALU realize uma soma). Os acessos ao banco de registradores e o cálculo do destino do desvio ocorrem em paralelo.

Após este ciclo de clock, determinar a ação a ser tomada pode depender do conteúdo da instrução.

3. Execução, cálculo do endereço de memória ou conclusão do desvio

Este é o primeiro ciclo durante o qual a operação do caminho de dados é determinada pela classe de instrução. Em todos os casos, a ALU está atuando nos operandos preparados na etapa anterior, realizando uma de quatro funções, dependendo da classe de instrução. Especificamos a ação a ser tomada de acordo com a classe de instrução:

Referência à memória:

```
SaídaALU <= A + estende-sinal (IR[15:0]);
```

Operação: a ALU está somando os operandos para formar o endereço de memória. Isso requer colocar OrigAALU em 1 (de modo que a primeira entrada da ALU seja o registrador A) e colocar OrigBALU em 10 (de modo que a saída da unidade de extensão de sinal seja usada para a segunda entrada da ALU). Os sinais de OpALU terão de ser colocados em 00 (fazendo a ALU ser somada).

Instrução lógica ou aritmética (tipo R):

```
SaídaALU <= A op B;
```

Operação: a ALU está realizando a operação especificada pelo código de função com dois valores lidos do banco de registradores no ciclo anterior. Isso exige colocar OrigAALU = 1 e OrigBALU = 00, que, juntos, fazem os registradores A e B serem usados como entradas da ALU. Os sinais de OpALU precisarão ser colocados em 10 (de modo que o campo funct seja usado para determinar as definições do sinal de controle da ALU).

Desvio:

```
if (A == B) PC <= SaídaALU;
```

Operação: a ALU é usada para fazer a comparação de igualdade entre os dois registradores lidos na etapa anterior. O sinal Zero da ALU é usado para determinar se o desvio será tomado ou não. Isso requer definir OrigAALU = 1 e OrigBALU = 00 (de modo que as entradas do banco de registradores

sejam as saídas da ALU). Os sinais de OpALU precisarão ser colocados em 01 (fazendo a ALU subtrair) para teste de igualdade. O sinal EscrevePCCond precisará ser ativado para atualizar o PC se a entrada Zero da ALU estiver ativa. Colocando OrigPC em 01, o valor escrito no PC virá de SaídaALU, que contém o endereço de destino do desvio calculado no ciclo anterior. Para desvios condicionais tomados, realmente escrevemos o PC duas vezes: uma para a saída da ALU (durante a decodificação da instrução e a busca dos registradores) e uma para SaídaALU (durante a etapa de conclusão do desvio). O último valor escrito no PC é o valor usado para a próxima busca de instrução.

Jump:

```
# {x, y} é a notação Verilog para concatenação dos campos de bit x e y  
PC <= {PC [31:28], (IR[25:0], 2'b00)};
```

Operação: o PC é substituído pelo endereço do jump. OrigPC é atualizado para direcionar o endereço do jump para o PC, e EscrevePC é ativado para escrever o endereço do jump no PC.

4. Etapa de acesso à memória ou conclusão de instrução tipo R

Durante esta etapa, uma instrução load ou store acessa a memória e uma instrução lógica ou aritmética escreve seu resultado. Quando um valor é lido da memória, ele é armazenado no registrador de dados da memória (MDR), onde precisa ser usado no próximo ciclo de clock.

Referência à memória:

```
MDR <= Memória [SaídaALU];
```

ou

```
Memória [SaídaALU] <= B;
```

Operação: se a instrução é um load, uma word de dados é lida da memória e é escrita no MDR. Se a instrução é um store, então, os dados são escritos na memória. Em qualquer caso, o endereço usado é o calculado durante a etapa anterior e armazenado em SaídaALU. Para um store, o operando de origem é salvo em B. (B, na verdade, é lido duas vezes, uma na etapa 2 e uma na etapa 3. Felizmente, o mesmo valor é lido ambas as vezes, já que o número do registrador – que é armazenado em IR e usado para ler do banco de registradores – não muda.) O sinal LeMem (para um load) ou EscreveMem (para um store) precisará ser ativado. Além disso, para loads e stores, o sinal IouD é colocado em 1 para forçar o endereço de memória a vir da ALU em vez do PC. Como o MDR é escrito em cada ciclo de clock, nenhum sinal de controle explícito precisa ser ativado.

Instrução lógica ou aritmética (tipo R):

```
Reg[IR[15:11]] <= SaídaALU;
```

Operação: coloque o conteúdo de SaídaALU, que corresponde à saída da operação da ALU no ciclo anterior, no registrador destino. O sinal RegDst precisa ser colocado em 1 para forçar o campo rd (bits 15:11) a ser usado para selecionar a entrada do banco de registradores a ser escrita. EscreveReg precisa ser ativado, e MemparaReg precisa ser colocado em 0 para que a saída da ALU seja escrita, em vez da saída de dados da memória.

5. Etapa de conclusão da leitura da memória

Durante esta etapa, os loads são completados escrevendo novamente o valor da memória.

Load:

```
Reg[IR[20:16]] <= MDR;
```

Etapa	Ação para instruções tipo R	Ação para instruções de acesso à memória	Ação para desvios	Ação para jumps
Busca da instrução		IR <= Memória[PC] PC <= PC + 4		
Decodificação da instrução e busca dos registradores		A <= Reg[IR[25:21]] B <= Reg[IR[20:16]] SaídaALU <= PC + (estende-sinal (IR[15:0]) << 2)		
Execução, cálculo do endereço ou conclusão do desvio/jump	SaídaALU <= A op B	SaídaALU <= A + estende-sinal (IR[15:0])	if (A == B) PC <= SaídaALU	PC <= {PC [31:28], (IR[25:0]), 2'b00})
Acesso à memória ou conclusão de instrução tipo R	Reg[IR[15:11]] <= SaídaALU	Load: MDR <= Memória[SaídaALU] ou Store: Memória [SaídaALU] <= B		
Conclusão da leitura da memória		Load: Reg[IR[20:16]] <= MDR		

FIGURA 5.30 Resumo das etapas realizadas para executar qualquer classe de instrução. As instruções levam de três a cinco etapas de execução. As duas primeiras etapas são independentes da classe de instrução. Após essas etapas, uma instrução leva ainda de um a três ciclos para ser concluída, dependendo da classe de instrução. As entradas vazias para a etapa de acesso à memória ou a etapa de conclusão da leitura da memória indicam que a classe de instrução específica leva menos ciclos. Em uma implementação multiciclo, uma nova instrução será iniciada tão logo a instrução atual seja completada; portanto, esses ciclos não são ociosos ou desperdiçados. Na verdade, como mencionado anteriormente, o banco de registradores lê a cada ciclo, mas, como o IR não muda, os valores lidos do banco de registradores são idênticos. Em especial, o valor lido para o registrador B durante a etapa de decodificação da instrução, para uma instrução de desvio ou tipo R, é o mesmo valor armazenado em B durante a etapa de execução e, depois, usado na etapa de acesso à memória para uma instrução store word.

Operação: escrever os dados do load, armazenados no MDR no ciclo anterior, no banco de registradores. Para isso, colocamos MemparaReg = 1 (para escrever o resultado na memória), ativamos EscreveReg (para causar uma escrita) e tornamos RegDst = 0 para escolher o campo rt (bits 20:16) como o número do registrador.

Essa seqüência de cinco etapas é resumida na Figura 5.30. Para essa seqüência, podemos determinar o que o controle precisa fazer em cada ciclo de clock.

Definindo o controle

Agora que determinamos quais são os sinais de controle e quando eles precisam ser ativados, podemos implementar a unidade de controle. Para projetar a unidade de controle para o caminho de dados de ciclo único, usamos um conjunto de tabelas verdade que especificaram a definição dos sinais de controle com base na classe de instrução. Para o caminho de dados multiciclo, o controle é mais complexo porque a instrução é executada em uma série de etapas. O controle para o caminho de dados multiciclo precisa especificar os sinais a serem definidos em qualquer etapa e também a próxima etapa na seqüência.

Nesta subseção e na Seção 5.7, veremos duas técnicas diferentes para especificar o controle. A primeira técnica é baseada em máquinas de estados finitos, que em geral são representadas graficamente. A segunda técnica, chamada **micropogramação**, usa uma representação de programação para o controle. As duas técnicas representam o controle de uma forma que permite a implementação detalhada – usando portas lógicas, ROMs ou PLAs – a ser sintetizada por um sistema de CAD. Neste capítulo, focalizaremos o projeto do controle e sua representação nessas duas formas.

A Seção 5.8 mostra como as linguagens de projeto de hardware são usadas para projetar processadores modernos com exemplos do caminho de dados multiciclo e o controle de estados finitos. No projeto de sistemas digitais moderno, a última etapa de levar uma descrição de hardware a portas lógicas reais é realizada por ferramentas de síntese de lógica e caminho de dados. O Apêndice C mostra como esse processo opera traduzindo a unidade de controle multiciclo para uma implementa-

micropogramação

Uma representação simbólica do controle na forma de instruções, chamadas microinstruções, que são executadas em uma micromáquina simples.

ção de hardware detalhada. As idéias principais do controle podem ser obtidas deste capítulo sem examinar o material da Seção 5.8 ou do Apêndice C. Entretanto, se você deseja realmente desenvolver algum projeto de hardware, a Seção 5.9 é útil, e o Apêndice C pode mostrar como as implementações provavelmente se parecerão no nível de portas lógicas.

Obtida essa implementação, e o conhecimento de que cada estado exige 1 ciclo de clock, podemos encontrar o CPI para um mix de instruções típico.

CPI EM UMA CPU MULTICICLO

Usando o mix de instruções SPECINT2000 mostrado na Figura 3.26, qual é o CPI, considerando que cada estado na CPU multiciclo exige 1 ciclo de clock?

O mix possui 25% de loads (1% load byte + 24% load word), 10% de stores (1% store byte + 9% store word), 11% de branches (6% beq, 5% bne), 2% de jumps (1% jal + 1% jr) e 52% de ALU (todo o restante do mix, que consideraremos ser instruções da ALU). Pela Figura 5.30, o número de ciclos de clock para cada classe de instrução é o seguinte:

- Loads: 5
- Stores: 4
- Instruções da ALU: 4
- Branches: 3
- Jumps: 3

O CPI é obtido pelo seguinte:

$$\text{CPI} = \frac{\text{Ciclos de clock da CPU}}{\text{Contagem de instruções}} = \frac{\sum \text{Contagem de instruções}_i \times \text{CPI}_i}{\text{Contagem de instruções}}$$
$$= \frac{\sum \text{Contagem de instruções}_i}{\text{Contagem de instruções}} \times \text{CPI}_i$$

A razão

$$\frac{\text{Contagem de instruções}_i}{\text{Contagem de instruções}}$$

é simplesmente a freqüência de instruções da classe de instrução i . Portanto, podemos substituir para obter

$$\text{CPI} = 0,25 \times 5 + 0,10 \times 4 + 0,52 \times 4 + 0,11 \times 3 + 0,02 \times 3 = 4,12$$

Esse CPI é melhor do que o CPI de pior caso de 5,0 quando todas as instruções usam o mesmo número de ciclos de clock. É claro que overheads nos dois projetos podem reduzir ou aumentar essa diferença. O projeto multiciclo provavelmente também é mais econômico, já que usa menos componentes separados no caminho de dados.

EXEMPLO

RESPOSTA

máquina de estados finitos

Uma função de lógica seqüencial consistindo em um conjunto de entradas e saídas, uma função de próximo estado que mapeia o estado atual e as entradas para um novo estado, e uma função de saída que mapeia o estado atual e possivelmente as entradas para uma série de saídas ativas.

função de próximo estado

Uma função combinatória que, dadas as entradas e o estado atual, determina o próximo estado de uma máquina de estados finitos.

O primeiro método usado para especificar o controle multiciclo é uma **máquina de estados finitos**. Uma máquina de estados finitos consiste em um conjunto de estados e diretrizes sobre como mudar de estado. As diretrizes são definidas por uma **função de próximo estado**, que mapeia o estado atual e as entradas para um novo estado. Quando usamos uma máquina de estados finitos para controle, cada estado também especifica um conjunto de saídas ativadas quando a máquina está nesse estado. A implementação de uma máquina de estados finitos em geral considera que todas as saídas que não são explicitamente ativadas estão inativas. Da mesma forma, a correta operação do caminho de dados depende do fato de que um sinal que não é explicitamente ativado está inativo, em vez de agir como um don't care. Por exemplo, o sinal EscreveReg deve ser ativado apenas quando uma entrada do banco de registradores deve ser escrita; quando ele não é explicitamente ativado, ele só pode estar inativo.

Os controles dos multiplexadores são um pouco diferentes, já que selecionam uma das entradas, quer seja 0 ou 1. Portanto, na máquina de estados finitos, sempre especificamos a definição de todos os controles dos multiplexadores que nos interessam. Quando implementamos a máquina de estados finitos com lógica, colocar um controle em 0 pode ser o padrão e, logo, pode não exigir portas lógicas. Um exemplo simples de uma máquina de estados finitos aparece no Apêndice B e, se você não está familiarizado com o conceito de máquina de estados finitos, talvez seja útil examinar o Apêndice B antes de continuar.

O controle de estados finitos basicamente corresponde às cinco etapas de execução mostradas nas páginas 245 a 247; cada estado na máquina de estados finitos usará 1 ciclo de clock. A máquina de estados finitos consistirá em várias partes. Como as duas primeiras etapas da execução são idênticas para cada instrução, os dois estados iniciais da máquina de estados finitos serão comuns para todas as instruções. As etapas 3 a 5 diferem, dependendo do opcode. Após a execução da primeira etapa para uma determinada classe de instrução, a máquina de estados finitos retornará ao estado inicial para começar a buscar a próxima instrução.

A Figura 5.31 mostra essa representação abstrata da máquina de estados finitos. Para preencher os detalhes da máquina de estados finitos, primeiro expandiremos a parte da busca e decodificação da instrução e, depois, mostraremos os estados (e ações) para as diferentes classes de instrução.

Mostramos os dois primeiros estados da máquina de estados finitos na Figura 5.32 usando uma representação gráfica tradicional. Numeramos os estados para simplificar a explicação, embora os números sejam arbitrários. O estado 0, correspondente à etapa 1, é o estado inicial da máquina.

Os sinais ativados em cada estado são mostrados dentro do círculo que representa o estado. Os arcos entre os estados definem o próximo estado e são rotulados com condições que selecionam um próximo estado específico quando vários próximos estados são possíveis. Após o estado 1, os sinais ativados dependem da classe de instrução. Portanto, a máquina de estados finitos possui quatro arcos partindo do estado 1, correspondentes às quatro classes de instrução: acesso à memória, tipo R,

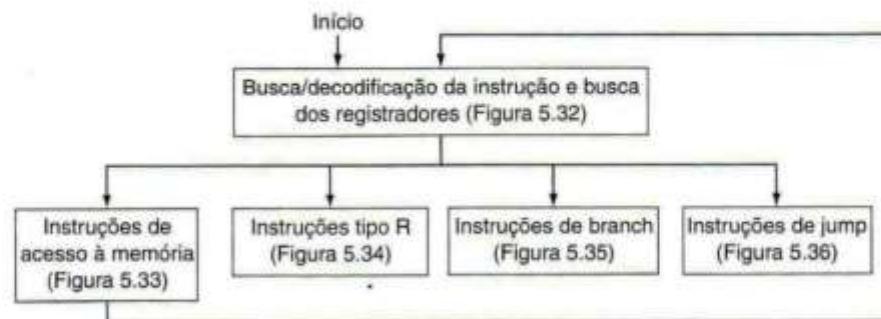


FIGURA 5.31 A visão de alto nível do controle da máquina de estados finitos. As primeiras etapas são independentes da classe de instrução; depois, uma série de sequências que dependem do opcode da instrução é usada para completar cada classe de instrução. Após completar as ações necessárias para essa classe de instrução, o controle retorna para buscar uma nova instrução. Cada retângulo nesta figura pode representar um ou vários estados. O arco rotulado como *Início* indica o estado onde começar quando a primeira instrução está para ser buscada.

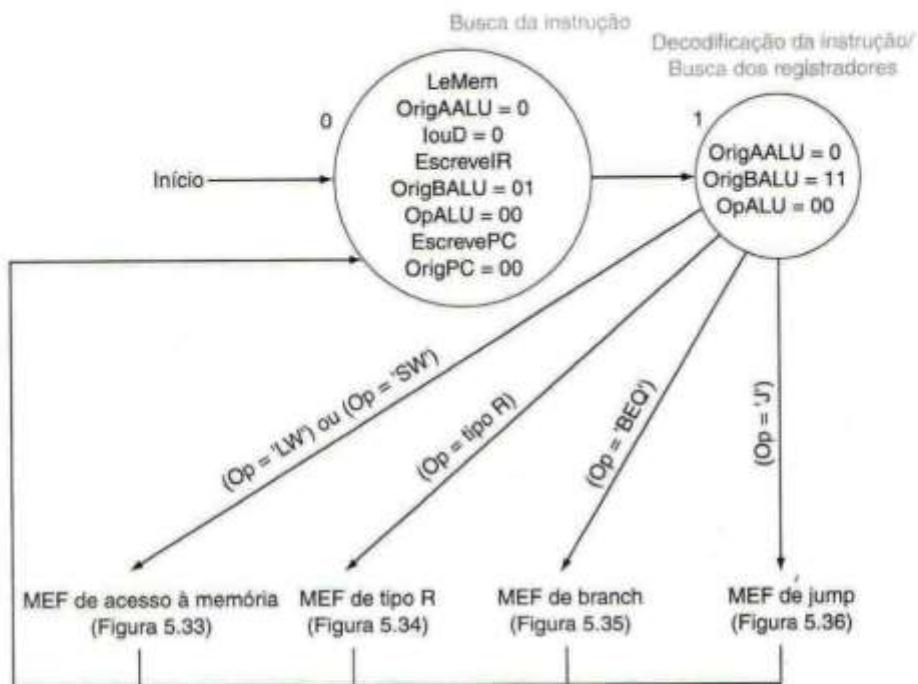


FIGURA 5.32 A parte da busca e decodificação da instrução é idêntica em todas as instruções. Esses estados correspondem ao retângulo superior na máquina de estados finitos na Figura 5.31. No primeiro estado, ativamos dois sinais para fazer com que a memória leia uma instrução e a escreva no registrador de instrução (LeMem e EscreveIR) e colocamos IouD em 0 para escolher o PC como a origem do endereço. Os sinais OrigAALU, OrigBALU, OpALU, EscrevePC e OrigPC são definidos para calcular PC + 4 e armazená-lo no PC. (Ele também será armazenado em SaídaALU, mas nunca será usado de lá.) No próximo estado, calculamos o endereço de destino do desvio colocando OrigBALU em 11 (fazendo com que os 16 bits menos significativos com sinal estendido e deslocados do IR sejam enviados para a ALU), colocando OrigAALU em 0 e OpALU em 00; armazenaremos o resultado no registrador SaídaALU, que é escrito em cada ciclo. Existem quatro próximos estados que dependem da classe da instrução, que é conhecida durante esse estado. A entrada da unidade de controle, chamada Op, é usada para determinar qual desses arcos seguir. Lembre-se de que todos os sinais não ativados explicitamente estão inativos; isso é muito importante para os sinais que controlam escritas. Para controles de multiplexadores, a falta de uma definição específica indica que não nos importamos com a definição do multiplexador.

branch on equal e jump. Esse processo de ramificação para diferentes estados dependendo da instrução é chamado de *decodificação*, uma vez que a escolha do próximo estado e, portanto, das ações que se seguem dependem da classe de instrução.

A Figura 5.33 mostra a parte da máquina de estados finitos necessária para implementar as instruções de acesso à memória. Para as instruções de acesso à memória, o primeiro estado após buscar a instrução e os registradores calcula o endereço de memória (estado 2). Para calcular o endereço de memória, os multiplexadores de entrada da ALU precisam ser definidos de forma que a primeira entrada seja o registrador A, enquanto a segunda entrada é o campo de deslocamento com sinal estendido; o resultado é escrito no registrador SaídaALU. Após o cálculo do endereço de memória, a memória deve ser lida ou escrita; isso exige dois estados diferentes. Se o opcode da instrução é lw, então, o estado 3 (correspondente à etapa do acesso à memória) faz a leitura da memória (LeMem é ativado). Nos estados 3 e 5, o sinal IouD é colocado em 1 para forçar o endereço de memória a vir da ALU. Após realizar uma escrita, a instrução sw completou a execução e o próximo estado é 0. Contudo, se a instrução é um load, outro estado (o estado 4) é necessário para escrever o resultado da memória no banco de registradores. Colocar os controles dos multiplexadores MemparaReg em 1 e RegDst em 0 enviará o valor carregado no MDR para ser escrito no banco de registradores, usando rt como o número de registrador. Após esse estado, correspondente à etapa da conclusão da leitura da memória, o próximo estado é 0.

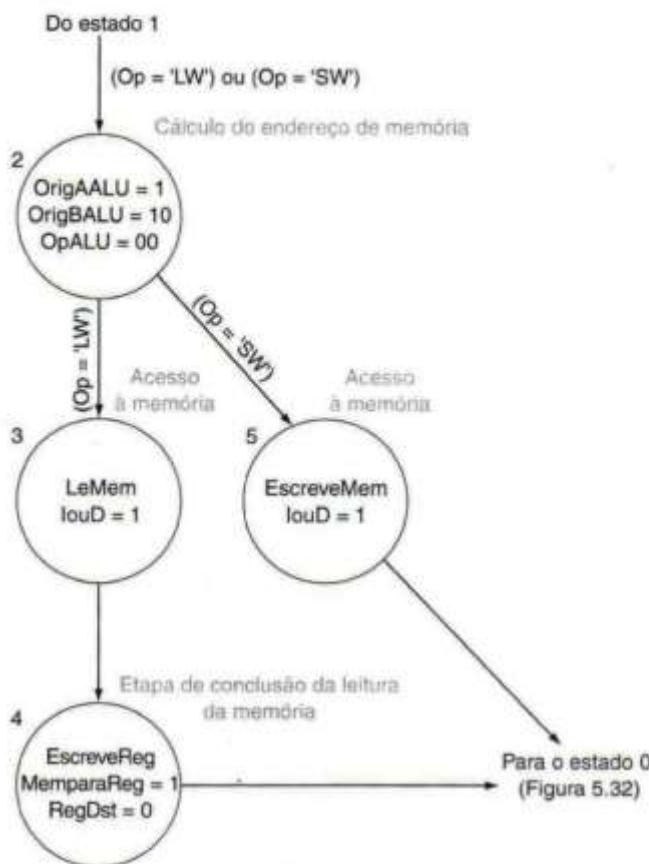


FIGURA 5.33 A máquina de estados finitos para controlar as instruções de referência à memória possui quatro estados. Esses estados correspondem ao retângulo rotulado como “Instruções de acesso à memória” na Figura 5.31. Após realizar um cálculo de endereço de memória, uma sequência separada é necessária para load e store. A definição dos sinais de controle OrigAALU, OrigBALU e OpALU é usada para causar o cálculo de endereço de memória no estado 2. Os loads exigem um estado extra para escrever o resultado do MDR (onde o resultado é escrito no estado 3) no banco de registradores.

Implementar as instruções tipo R exige dois estados correspondentes às etapas 3 (execução) e 4 (conclusão das instruções tipo R). A Figura 5.34 mostra essa parte de dois estados da máquina de estados finitos. O estado 6 ativa OrigAALU e coloca os sinais de OrigBALU em 00; isso força os dois registradores lidos do banco de registradores a serem usados como entradas para a ALU. Colocar OpALU em 10 faz com que a unidade de controle da ALU use o campo funct para definir os sinais de controle da ALU. No estado 7, EscreveReg é ativado para fazer a escrita no banco de registradores, RegDst é ativado para fazer o campo rd ser usado como o número do registrador destino, e MemparaReg é desativado para selecionar SaídaALU como a origem do valor a ser escrito no banco de registradores.

Para desvios, apenas um único estado adicional é necessário, pois completam a execução durante a terceira etapa da execução da instrução. Durante esse estado, os sinais de controle que fazem a ALU comparar o conteúdo dos registradores A e B precisam ser definidos, e os sinais que fazem o PC ser escrito condicionalmente com o endereço no registrador SaídaALU também são definidos. Para realizar a comparação, é necessário ativar OrigAALU, colocar OrigBALU em 00 e colocar o valor de OpALU em 01 (forçando uma subtração). (Usamos apenas a saída Zero da ALU, não o resultado da subtração.) Para controlar a escrita do PC, ativamos EscrevePCCond e colocamos OrigPC em 01, que fará o valor no registrador SaídaALU (contendo o endereço de desvio calculado no estado 1, Figura 5.32) ser escrito no PC se o bit Zero da ALU estiver ativo. A Figura 5.35 mostra esse estado único.

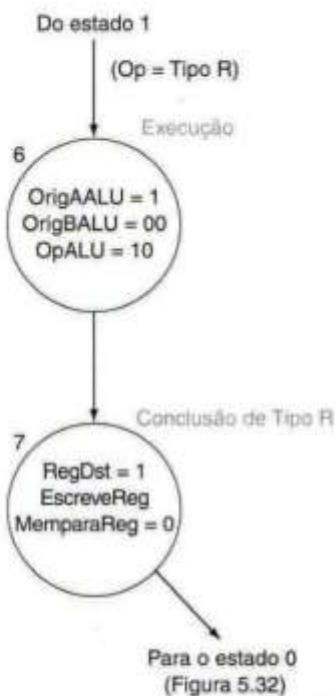


FIGURA 5.34 As instruções tipo R podem ser implementadas com uma máquina de estados finitos simples de dois estados. Esses estados correspondem ao retângulo rotulado como “Instruções tipo R” na Figura 5.31. O primeiro estado faz com que a operação da ALU ocorra, enquanto o segundo estado faz com que o resultado da ALU (que está em SaídaALU) seja escrito no banco de registradores. Os três sinais ativados durante a etapa 7 levam o conteúdo de SaídaALU a ser escrito no banco de registradores na entrada especificada pelo campo rd do registrador de instrução.

Agora, podemos juntar essas partes da máquina de estados finitos para formar uma especificação para a unidade de controle, como mostra a Figura 5.38. Em cada estado, os sinais ativos são mostrados. O próximo estado depende dos bits do opcode da instrução e, portanto, rotulamos os arcos com uma comparação para os opcodes de instrução correspondentes.

Uma máquina de estados finitos pode ser implementada com um registrador temporário que contenha o estado atual e um bloco de lógica combinacional que determine os sinais do caminho de dados a serem ativados bem como o próximo estado. A Figura 5.37 mostra como poderia se parecer uma implementação desse tipo. O Apêndice C descreve em detalhes como a máquina de estados finitos é implementada usando essa estrutura. Na Seção C.3, a lógica de controle combinacional para a máquina de estados finitos da Figura 5.38 é implementada com uma ROM (read-only memory – memória somente de leitura) e um PLA (programmable logic array – array lógico programável). (Veja também o Apêndice B para uma descrição desses elementos lógicos.) Na próxima seção deste capítulo, consideraremos outra forma de representar o controle. Essas duas técnicas são simplesmente representações diferentes da mesma informação de controle.

A última classe de instrução é jump; assim como o branch, ela requer apenas um único estado (mostrado na Figura 5.36) para completar sua execução. Nesse estado, o sinal EscrevePC é ativado para fazer o PC ser escrito. Colocando OrigPC em 10, o valor fornecido para escrita será os 26 bits menos significativos do registrador de instrução com 00_{bin} acrescentado como os bits menos significativos concatenados com os 4 bits mais significativos do PC.

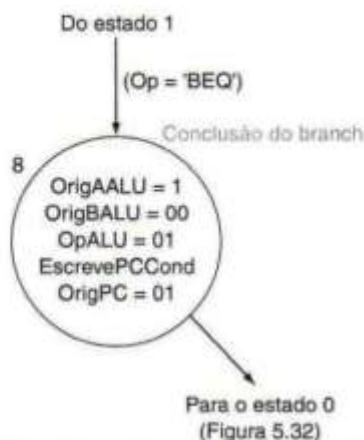


FIGURA 5.35 A instrução branch requer um estado único. As três primeiras saídas que estão ativas fazem a ALU comparar os registradores (OrigAALU, OrigBALU e OpALU), enquanto os sinais OrigPC e EscrevePCCond realizam a escrita condicional se a condição de desvio for verdadeira. Observe que não usamos o valor escrito em SaídaALU; em vez disso, usamos apenas a saída Zero da ALU. O endereço de destino do desvio é lido de SaídaALU, onde ele é salvo no final do estado 1.

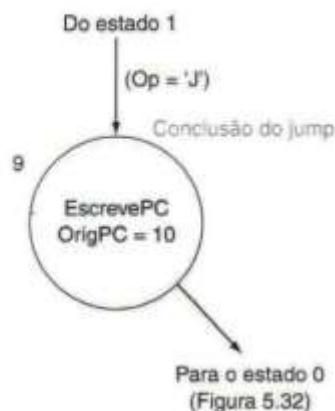


FIGURA 5.36 A instrução jump requer um único estado que ative dois sinais de controle para escrever o PC com os 26 bits menos significativos do registrador de instrução deslocados 2 bits para a esquerda e concatenados com os 4 bits mais significativos do PC dessa instrução.

O pipelining, o assunto do Capítulo 6, é quase sempre usado para acelerar a execução das instruções. Para instruções simples, o pipelining é capaz de atingir a velocidade de clock mais alta de um projeto multiciclo e um CPI de ciclo único de um projeto de clock único. Na maioria dos processadores com pipeline, entretanto, algumas instruções levam mais tempo do que um único ciclo e exigem controle multiciclo. Instruções de ponto flutuante são um exemplo universal. Há muitos exemplos na arquitetura IA-32 que exigem o uso do controle multiciclo.

Detalhamento: o estilo de máquina de estados finitos na Figura 5.37 é chamado de uma máquina de Moore (nome derivado de Edward Moore). Sua característica identificadora é que a saída depende apenas do estado atual. Para uma máquina de Moore, o retângulo rotulado como "lógica de controle combinacional" pode ser dividido em duas partes. Uma parte possui a saída do controle e apenas a entrada do estado, enquanto a outra tem apenas a saída do próximo estado.

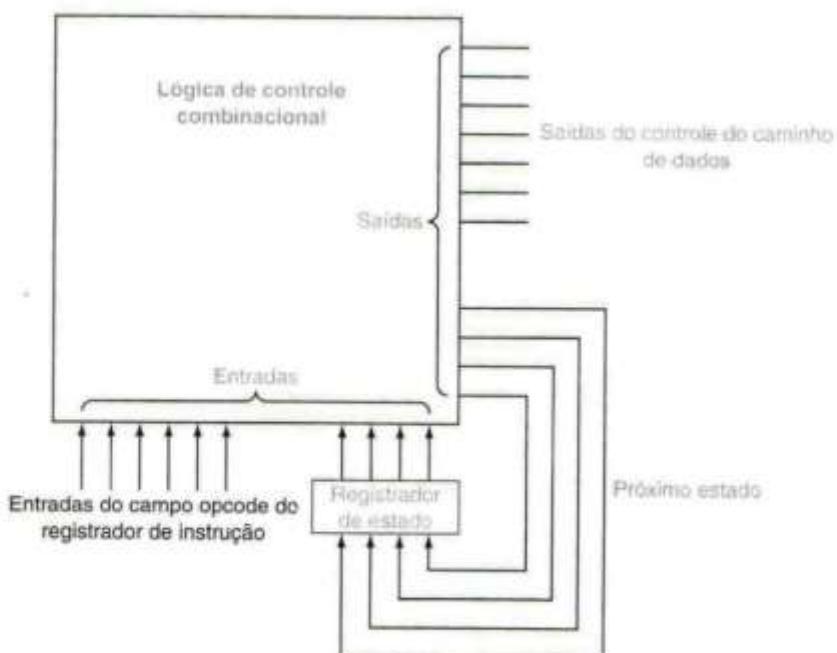


FIGURA 5.37 Os controles da máquina de estados finitos normalmente são implementados usando um bloco de lógica combinacional e um registrador para conter o estado atual. As saídas da lógica combinacional são o número do próximo estado e os sinais de controle a serem ativados para o estado atual. As entradas para a lógica combinacional são o estado atual e quaisquer entradas usadas para determinar o próximo estado. Nesse caso, as entradas são os bits de opcode do registrador de instrução. Observe que na máquina de estados finitos usada neste capítulo, as saídas só dependem do estado atual, não das entradas. A seção “Detalhamento” a seguir explica isso mais a fundo.

Um estilo de máquina alternativo é uma máquina de Mealy (nome derivado de George Mealy). A máquina de Mealy permite que tanto a entrada quanto o estado atual sejam usados para determinar a saída. As máquinas de Moore possuem vantagens de implementação potenciais na velocidade e no tamanho da unidade de controle. As vantagens de velocidade surgem porque as saídas de controle, necessárias no início do ciclo de clock, não dependem das entradas, mas apenas do estado atual. No Apêndice C, quando a implementação dessa máquina de estados finitos é levada até as portas lógicas, a vantagem do tamanho pode ser vista claramente. A possível desvantagem de uma máquina de Moore é que ela pode exigir estados adicionais. Por exemplo, em situações em que há uma diferença de um estado entre duas seqüências de estados, a máquina de Mealy pode unificar os estados tornando as saídas dependentes das entradas.

Entendendo o desempenho dos programas

Para um processador com um determinado ciclo de clock, o desempenho relativo entre dois segmentos de código será determinado pelo produto do CPI pela contagem de instruções para executar cada segmento. Como temos visto aqui, as instruções podem variar em seu CPI, mesmo para um processador simples. Nos próximos dois capítulos, veremos que a introdução do pipelining e o uso de caches criam oportunidades ainda maiores para variação no CPI. Embora muitos fatores que afetam o CPI sejam controlados pelo projetista do hardware, o processador, o compilador e o sistema de software ditam as instruções a serem executadas, e é esse processo que determina qual será o CPI efetivo do programa. Os programadores buscando melhorar o desempenho precisam entender o papel do CPI e os fatores que o afetam.

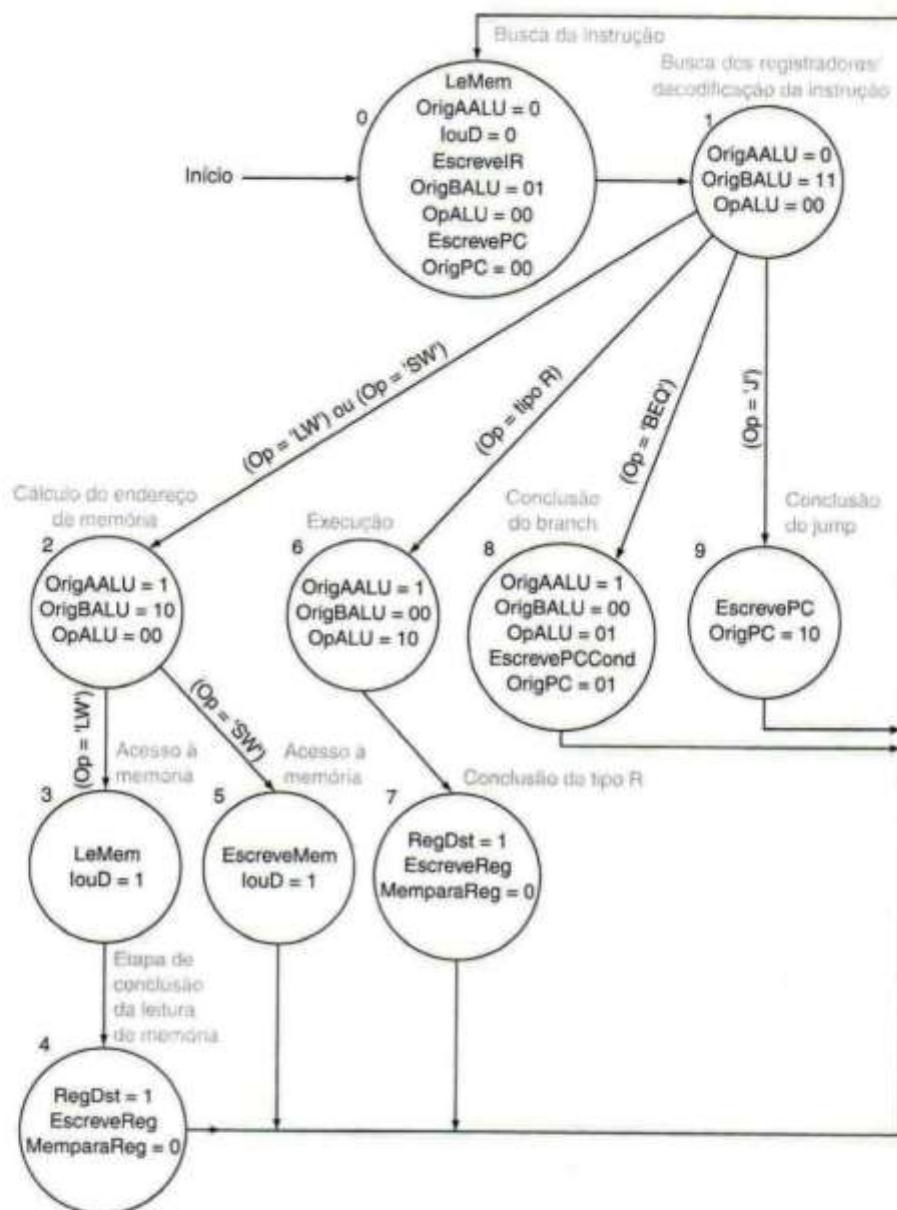


FIGURA 5.38 O controle da máquina de estados finitos completo para o caminho de dados mostrado na Figura 5.28.

Figura 5.28. Os rótulos nos arcos são condições testadas para determinar que estado é o próximo estado; quando o próximo estado é incondicional, nenhum rótulo é fornecido. Os rótulos dentro dos nós indicam os sinais de saída ativados durante esse estado; sempre especificamos a definição de um sinal de controle de multiplexador se a operação correta o exigir. Conseqüentemente, em alguns estados, um controle de multiplexador será definido em 0.

Verifique você mesmo

1. Verdadeiro ou falso: como a instrução de jump não depende dos valores dos registradores ou do cálculo do endereço de destino do desvio, ela pode ser completada durante o segundo estado, em vez de esperar até o terceiro.
2. Verdadeiro, falso ou talvez: o sinal de controle EscrevePCCond pode ser substituído por OrigPC[0].

5.6 Exceções

O controle é o aspecto mais complicado do projeto do processador: esta é a parte mais difícil de realizar corretamente e a parte mais difícil de tornar rápida. Uma das partes mais difíceis do controle é implementar **exceções e interrupções** – eventos diferentes de desvios ou jumps que mudam o fluxo normal de execução das instruções. Uma exceção é um evento inesperado de dentro do processador; o overflow aritmético é um exemplo de exceção. Uma interrupção é um evento que também causa uma mudança inesperada no fluxo de controle mas vem de fora do processador. As interrupções são usadas pelos dispositivos de E/S para se comunicarem com o processador, como veremos no Capítulo 8.

Muitas arquiteturas e autores não distinguem entre interrupções e exceções, freqüentemente usando o termo *interrupção* para se referirem aos dois tipos de evento. Seguimos a convenção MIPS, usando o termo *exceção* para se referir a *qualquer* mudança inesperada no fluxo de controle sem distinguir se a causa é interna ou externa; usamos o termo *interrupção* apenas quando o evento é causado externamente. A arquitetura Intel IA-32 usa a palavra *interrupção* para todos esses eventos.

As interrupções foram criadas inicialmente para tratar os eventos inesperados, como overflow aritmético, e para sinalizar requisições de serviço dos dispositivos de E/S. O mesmo mecanismo básico foi estendido para tratar exceções geradas internamente. Aqui estão alguns exemplos mostrando se a situação é gerada internamente pelo processador ou se é gerada externamente:

Tipo de evento	Origem	Terminologia MIPS
Requisição de dispositivo de E/S	Externa	Interrupção
Chamada ao sistema operacional pelo programa do usuário	Interna	Exceção
Overflow aritmético	Interna	Exceção
Uso de uma instrução indefinida	Interna	Exceção
Mal-funcionamentos do hardware	Ambas	Exceção ou Interrupção

Muitos dos requisitos para suportar exceções são provenientes da situação específica que faz com que uma exceção ocorra. Então, voltaremos a esse assunto no Capítulo 7, quando discutiremos as hierarquias de memória, e no Capítulo 8, quando discutiremos a E/S e entenderemos melhor a motivação para as capacidades adicionais no mecanismo de exceção. Nesta seção, lidaremos com a implementação do controle para detectar dois tipos de exceções que surgem de parte do conjunto de instruções e da implementação que já discutimos.

A detecção de condições excepcionais e a tomada da ação apropriada freqüentemente estão no caminho crítico da sincronização de uma máquina, que determina o tempo do ciclo de clock e, portanto, o desempenho. Sem uma atenção apropriada às exceções durante o projeto da unidade de controle, as tentativas de incluir exceções em uma implementação complexa podem reduzir significativamente o desempenho, bem como complicar a tarefa de obter o projeto correto.

Como as exceções são tratadas

Os dois tipos de exceções que nossa implementação atual pode gerar são a execução de uma instrução indefinida e um overflow aritmético. A ação básica que a máquina precisa realizar quando ocorre uma exceção é salvar o endereço da instrução problemática no contador de programa para exceções (EPC) e, depois, transferir o controle para o sistema operacional em alguns endereços especificados.

O sistema operacional, então, pode tomar a ação apropriada, que pode envolver fornecer algum serviço ao programa do usuário, tomar alguma ação predefinida em resposta a um overflow ou interromper a execução do programa e retornar um erro. Depois de realizar qualquer ação necessária devido à exceção, o sistema operacional pode terminar o programa ou continuar sua execução usando o

exceção Também chamada de interrupção. Um evento não programado que interrompe a execução do programa; usada para detectar overflows.

interrupção Uma exceção que vem de fora do processador. (Algumas arquiteturas usam o termo *interrupção* para todas as exceções.)

EPC para determinar onde reiniciar a execução do programa. No Capítulo 7, veremos em mais detalhes o problema do reinício da execução.

interrupção vetorializada Uma interrupção em que o endereço para o qual o controle é transferido é determinado pela causa da exceção.

Para que o sistema operacional trate a exceção, ele precisa conhecer o motivo da exceção, além da instrução que a causou. Existem dois métodos principais usados para comunicar o motivo de uma exceção. O método usado na arquitetura MIPS é incluir um registrador de status (chamado *registrar Cause*), que contém um campo indicando o motivo da exceção.

Um segundo método é usar **interrupções vetorializadas**. Em uma interrupção vetorializada, o endereço para o qual o controle é transferido é determinado pela causa da exceção. Por exemplo, para acomodar os dois tipos de exceção relacionados anteriormente, podemos definir os dois endereços do vetor de exceções a seguir:

Tipo de exceção	Endereço do vetor de exceções (em hexa)
Instrução indefinida	C000 0000 _{hex}
Overflow aritmético	C000 0020 _{hex}

O sistema operacional sabe a causa da exceção pelo endereço em que ela é iniciada. Os endereços são separados por 32 bytes ou 8 instruções, e o sistema operacional precisa registrar a causa da exceção e pode realizar algum processamento limitado nessa sequência. Quando a exceção não é vetorializada, um único ponto de entrada para todas as exceções pode ser usado e o sistema operacional decodifica o registrador de status para encontrar a causa.

Podemos executar o processamento exigido para as exceções incluindo alguns registradores e sinais de controle extras em nossa implementação básica e estendendo ligeiramente a máquina de estados finitos. Vamos considerar que estamos implementando o sistema de exceção usado na arquitetura MIPS. (Implementar exceções vetorializadas não é mais difícil.) Precisaremos acrescentar dois registradores adicionais ao caminho de dados:

- *EPC*: um registrador de 32 bits usado para conter o endereço da instrução afetada. (Esse registrador é necessário mesmo quando as exceções são vetorializadas.)
- *Cause*: um registrador usado para registrar a causa da exceção. Na arquitetura MIPS, esse registrador é de 32 bits, embora alguns bits atualmente não sejam usados. Considere que o bit menos significativo desse registrador codifica as duas origens de exceção possíveis mencionadas anteriormente: instrução indefinida = 0 e overflow aritmético = 1.

Precisamos incluir dois sinais de controle para fazer com que os registradores EPC e Cause sejam escritos; chame esses sinais de controle *EscreveEPC* e *EscreveCause*. Além disso, precisaremos de um sinal de controle de 1 bit para definir corretamente o bit menos significativo do registrador Cause; chame esse sinal *CausalInt*. Finalmente, precisaremos ser capazes de escrever no PC o *endereço de exceção*, que é o ponto de entrada do sistema operacional para o tratamento da exceção; na arquitetura MIPS, esse endereço é 8000 0180_{hex}. Então, OrigPC pode ser colocado em 11_{bin} para selecionar esse valor para ser escrito no PC.

Como o PC é incrementado durante o primeiro ciclo de cada instrução, não podemos simplesmente escrever o valor do PC no EPC, já que o valor no PC será o endereço da instrução mais 4. Entretanto, podemos usar a ALU para subtrair 4 do PC e escrever a saída no EPC. Isso não requer sinais de controle ou caminhos adicionais, uma vez que podemos usar a ALU para subtrair, e a constante 4 já é uma possível entrada da ALU. A porta de escrita de dados do EPC, portanto, está conectada à saída da ALU. A Figura 5.39 mostra o caminho de dados multiciclo com as adições necessárias para implementar exceções.

Usando o caminho de dados da Figura 5.39, a ação a ser tomada para cada tipo diferente de exceção pode ser tratada em um estado para cada uma. Em cada caso, o estado define o registrador Cause, calcula e salva o PC original no EPC e escreve o endereço de exceção no PC. Portanto, para tratar os dois tipos de exceção considerados, precisaremos incluir apenas os dois estados. Todavia, antes de incluirmos, precisamos determinar como verificar exceções, já que essas verificações irão controlar os arcos para os novos estados.

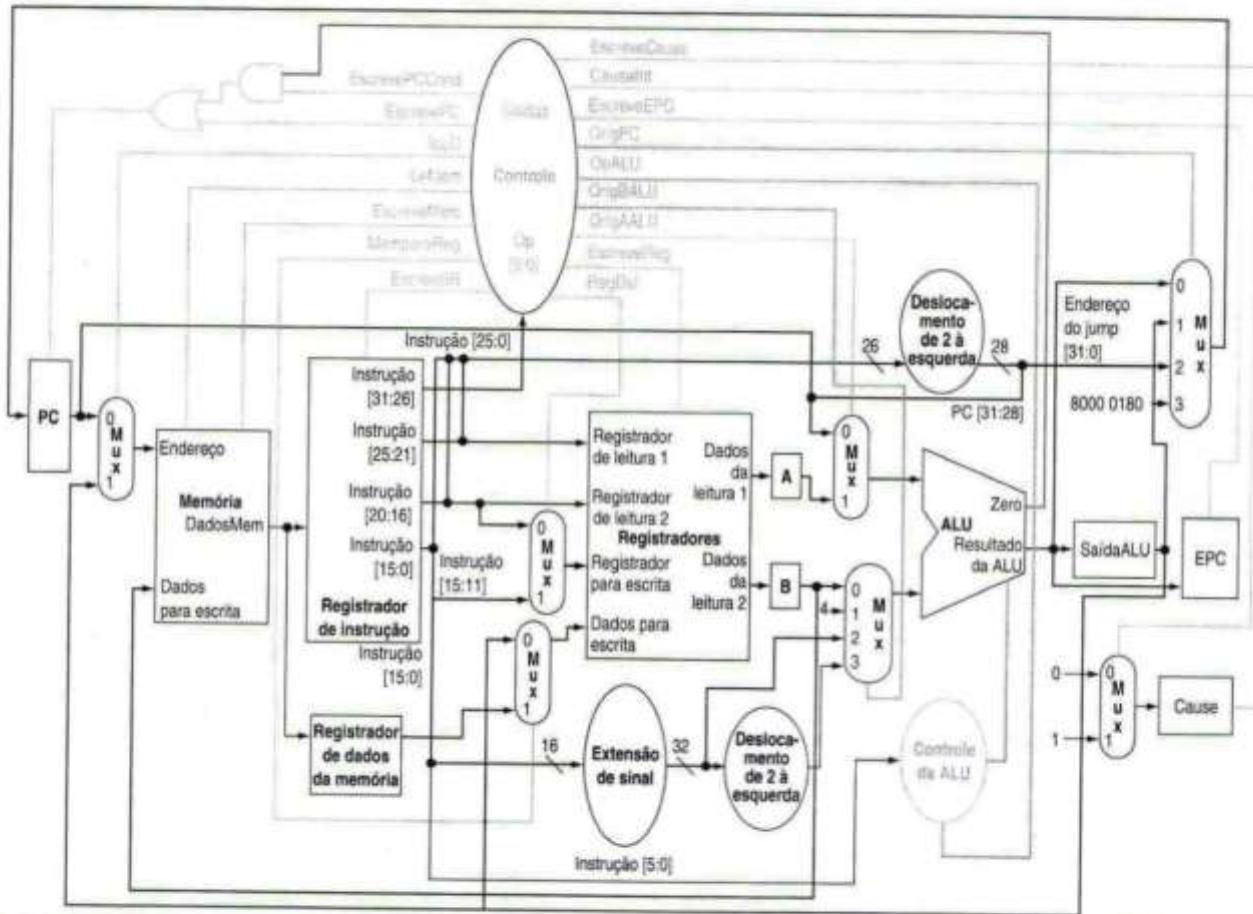


FIGURA 5.39 O caminho de dados multiciclo com as adições necessárias para implementar exceções. As adições específicas incluem os registradores Cause e EPC, um multiplexador para controlar o valor enviado ao registrador Cause, uma expansão do multiplexador controlando o valor escrito no PC e linhas de controle para o multiplexador e os registradores acrescentados. Para maior simplicidade, essa figura não mostra o sinal de overflow da ALU, que precisaria ser armazenado em um registrador de 1 bit e distribuído como uma entrada adicional para a unidade de controle (consulte a Figura 5.40 para ver como ele é usado).

Como o controle verifica exceções

Agora, precisamos projetar um método de detectar essas exceções e transferir o controle para o estado apropriado nos estados de exceção. A Figura 5.40 mostra os dois novos estados (10 e 11), assim como sua conexão com o restante do controle de estados finitos. Cada uma das duas exceções possíveis é detectada diferentemente:

- **Instrução indefinida:** isso é detectado quando nenhum próximo estado é definido do estado 1 para o valor de Op. Tratamos essa exceção definindo como o estado 10 o valor do próximo estado para todos os valores de Op diferentes de lw, sw, 0 (tipo R), j e beq. Mostramos isso usando simbolicamente *outro* para indicar que o campo Op não corresponde a nenhum dos opcodes que rotulam arcos do estado 1 para o novo estado 10, que é usado para essa exceção.
- **Overflow aritmético:** a ALU, projetada no Apêndice B, incluiu lógica para detectar overflow e um sinal chamado *Overflow* é fornecido como uma saída da ALU. Esse sinal é usado na máquina de estados finitos para especificar um possível próximo estado adicional (estado 11) para o estado 7, como mostra a Figura 5.40.

A Figura 5.40 representa uma especificação completa do controle para esse subconjunto MIPS com dois tipos de exceção. Lembre-se de que o problema de projetar o controle de uma máquina real é lidar

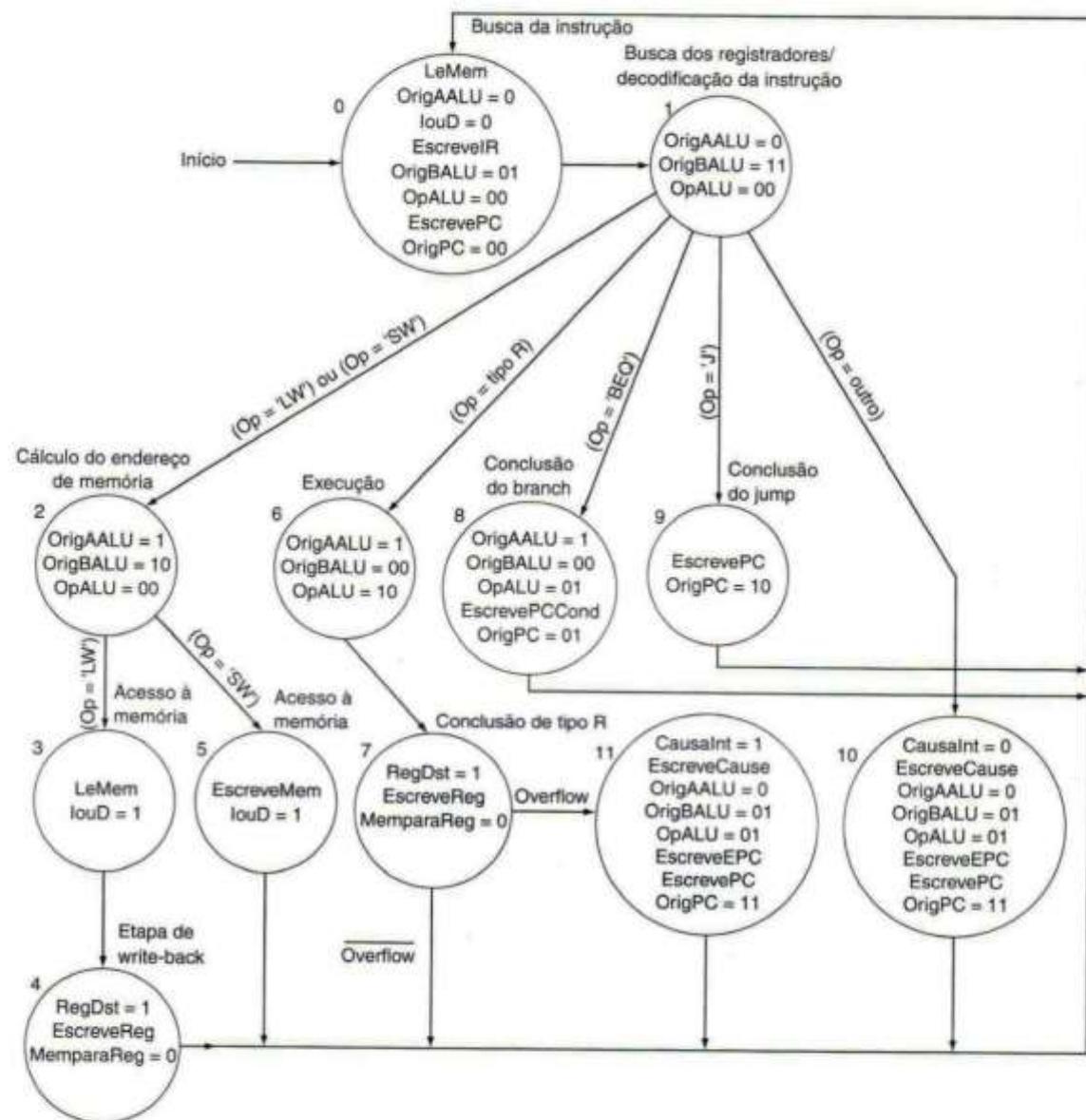


FIGURA 5.40 Isso mostra a máquina de estados finitos com as adições para tratar a detecção de exceções. Os estados 10 e 11 são os novos estados que geram o controle apropriado para exceções. O desvio do estado 1 rotulado como (*Op = other*) indica o próximo estado quando a entrada não corresponde aos opcodes de 1w, sw, 0 (tipo R), j ou beq. O desvio do estado 7 rotulado como *Overflow* indica a ação a ser tomada quando a ALU sinalizar um overflow.

com a variedade de interações diferentes entre instruções e outros eventos causadores de exceção de maneira que a lógica de controle permaneça pequena e rápida. As interações complexas que são possíveis são o que torna a unidade de controle o aspecto mais complicado do projeto de hardware.

Detalhamento: se você examinar de perto a máquina de estados finitos na Figura 5.40, poderá ver que alguns problemas podem ocorrer na forma como as exceções são tratadas. Por exemplo, no caso do overflow aritmético, a instrução que causa o overflow completa a escrita do seu resultado porque o desvio do overflow está no mesmo estado em que a escrita é completada. Entretanto, é possível que a arquitetura defina a instrução como não tendo efeito algum se a instrução causar uma exceção; é isso que o conjunto de instruções MIPS especifica. No Capítulo 7, veremos que certas classes de exceções exigem evitar que a instrução mude o estado da máquina e veremos que esse aspecto do tratamento de exceções se torna complexo e potencialmente limita o desempenho.

A otimização proposta na seção “Verifique você mesmo” da página 256 com relação a OrigPC ainda é válida no controle estendido para as exceções mostradas na Figura 5.40? Justifique.

**Verifique
você mesmo**

5.7

Microprogramação: simplificando o projeto de controle

A microprogramação é uma técnica para projetar unidades de controle complexas. Ela usa um mecanismo de hardware simples que pode, então, ser programado para implementar um conjunto de instruções mais complexo. A microprogramação é usada hoje para implementar algumas partes de um conjunto de instruções complexo, como um Pentium, bem como em processadores de uso especial. Esta seção, que aparece no CD, explica os conceitos básicos e como eles podem ser usados para implementar o controle multiciclo MIPS.

5.8

Uma introdução ao projeto digital usando uma linguagem de projeto de hardware

O projeto digital moderno é feito usando linguagens de descrição de hardware e ferramentas de síntese auxiliadas por computador que podem criar projetos de hardware detalhados das descrições usando bibliotecas e síntese de lógica. Livros inteiros são escritos sobre essas linguagens e seu uso no projeto digital. Esta seção, que aparece no CD, fornece uma breve introdução e mostra como uma linguagem de projeto de hardware, a Verilog nesse caso, pode ser usada para descrever o controle multiciclo MIPS comportalmente e em uma forma adaptável para síntese de hardware.

5.9

Vida real: a organização das recentes implementações Pentium

As técnicas descritas neste capítulo para construir caminhos de dados e unidades de controle são o coração de todo computador. Entretanto, todos os computadores modernos vão além das técnicas deste capítulo e usam pipelining. O *pipelining*, assunto do próximo capítulo, melhora o desempenho sobrepondo a execução de múltiplas instruções, atingindo uma vazão de uma instrução por ciclo de clock (como nossa implementação de ciclo único de clock) com um tempo de ciclo de clock determinado pelo atraso de unidades funcionais individuais em vez de todo o caminho de execução de uma instrução (como nosso projeto multiciclo). O último processador Intel IA-32 sem pipelining foi o 80386, introduzido em 1985; o primeiro processador MIPS, o R2000, também introduzido em 1985, usava pipeline.

Os recentes processadores Intel IA-32 (o Pentium II, III e 4) empregam métodos de pipelining sofisticados. Esses processadores, no entanto, ainda se vêem diante do desafio de implementar o controle para o complexo conjunto de instruções IA-32, descrito no Capítulo 2. As unidades funcionais e os caminhos de dados básicos em uso nos processadores modernos, embora significativamente mais complexos do que os descritos neste capítulo, possuem a mesma funcionalidade básica e tipos semelhantes de sinais de controle. Portanto, a tarefa de projetar uma unidade de controle explora os mesmos princípios usados neste capítulo.

Desafios da implementação de arquiteturas mais complexas

Diferente da arquitetura MIPS, a arquitetura IA-32 contém instruções muito complexas que podem levar dezenas, se não centenas de ciclos para serem executados. Por exemplo, a instrução move

controlador

microprogramado Um método de especificar controle que usa microcódigo em vez de uma representação de estados finitos.

controle

hardwired Uma implementação de controle da máquina de estados finitos normalmente usando arrays lógicos programáveis (PLAs) ou grupos de PLAs e lógica aleatória.

microcódigo

O conjunto das microinstruções que controlam um processador.

superescalar Uma técnica de pipelining avançada que permite ao processador executar mais de uma instrução por ciclo de clock.

microinstrução Uma representação do controle usando instruções de baixo nível, cada uma ativando um conjunto de sinais de controle em um determinado ciclo de clock, bem como especificando que microinstrução executar em seguida.

microoperações As instruções do tipo RISC diretamente executadas pelo hardware nas recentes implementações Pentium.

string (MOV\$) requer calcular e atualizar dois endereços de memória diferentes, bem como carregar e armazenar um byte da string. A maior quantidade e a complexidade dos modos de endereçamento na arquitetura IA-32 complicam a implementação mesmo das instruções simples semelhantes às do MIPS. Felizmente, um caminho de dados multiciclo é bem-estruturado para se adaptar às variações na quantidade de trabalho necessário por instrução, que são inerentes às instruções IA-32. Essa adaptabilidade vem de duas capacidades:

1. Um caminho de dados multiciclo permite que instruções demorem números variáveis de ciclos de clock. As instruções IA-32 simples que são semelhantes às da arquitetura MIPS podem ser executadas em 3 ou 4 ciclos de clock, enquanto instruções mais complexas podem levar dezenas de ciclos.
2. Um caminho de dados multiciclo pode usar componentes do caminho de dados mais de uma vez por instrução. Isso é fundamental para manipular modos de endereçamento mais complexos, assim como para implementar operações mais complexas, ambos presentes na arquitetura IA-32. Sem essa capacidade, o caminho de dados precisaria ser estendido para tratar as demandas das instruções mais complexas sem reutilizar componentes, o que poderia ser completamente impraticável. Por exemplo, um caminho de dados de ciclo único (que não reutiliza componentes) para a IA-32 exigiria várias memórias de dados e uma enorme quantidade de ALUs.

O uso do caminho de dados multiciclo e de um **controlador microprogramado** fornece uma estrutura para implementar o conjunto de instruções IA-32. A tarefa difícil, no entanto, é criar uma implementação de alto desempenho, que exige lidar com a diversidade dos requisitos que surgem das diferentes instruções. Basicamente, uma implementação de alto desempenho precisa garantir que as instruções simples serão executadas rapidamente e que o transtorno das complexidades do conjunto de instruções penalizam principalmente as instruções complexas e menos utilizadas.

Para atingir esse objetivo, todas as implementações Intel da arquitetura IA-32 desde o 486 usaram uma combinação do **controle hardwired** para tratar instruções simples e o controle **microcodificado** para tratar as instruções mais complexas. Para as instruções que podem ser executadas em uma única passagem pelo caminho de dados – aquelas com complexidade semelhante a uma instrução MIPS –, o controle hardwired gera as informações de controle e executa a instrução em uma única passagem pelo caminho de dados que usa um pequeno número de ciclos de clock. As instruções que exigem múltiplas passagens pelo caminho de dados e seqüênciação complexa são manipuladas pelo controlador microcodificado, que exige um número maior de ciclos e múltiplas passagens pelo caminho de dados para completar a execução da instrução. A vantagem desse método é que ele permite ao projetista obter baixas contagens de ciclo para as instruções simples sem ter de construir o caminho de dados extremamente complexo que seria necessário para manipular a grande maioria das instruções mais complexas.

A estrutura da implementação do Pentium 4

Os recentes processadores Pentium são capazes de executar mais de uma instrução por clock usando uma técnica de pipelining avançada, chamada **superescalar**. Descrevemos como funciona um processador superescalar no próximo capítulo. A coisa importante a entender aqui é que executar mais de uma instrução por clock exige duplicar os recursos do caminho de dados. A maneira mais simples de pensar nisso é que o processador possui vários caminhos de dados, embora sejam adequados para tratar uma classe de instruções – por exemplo, loads e stores, operações da ALU ou desvios. Dessa forma, o processador é capaz de executar um load ou store no mesmo ciclo de clock em que está executando um desvio ou uma operação da ALU. Os Pentiums III e 4 permitem que até três instruções IA-32 sejam executadas em um ciclo de clock.

Os Pentiums III e Pentium 4 executam **microinstruções** simples semelhantes às instruções MIPS, chamadas **microoperações** na terminologia da Intel. Essas microinstruções são operações totalmen-

te autônomas que inicialmente possuem 70 bits de largura. O controle do caminho de dados para implementar essas microinstruções é completamente hardwired. Esse último nível de controle expande até três microinstruções em cerca de 120 linhas de controle para os caminhos de dados de inteiros e de 275 a 400 linhas de controle para o caminho de dados de ponto flutuante – o último número é para as novas instruções SSE2 incluídas no Pentium 4. Essa última etapa da expansão das microinstruções em linhas de controle é muito semelhante à geração de controle para o caminho de dados de ciclo único ou para o controle da ALU.

Como a tradução entre as instruções IA-32 e as microinstruções são realizadas? Nas implementações Pentium anteriores (por exemplo, o Pentium Pro, o Pentium II e o Pentium III), a unidade de decodificação de instruções olharia até três instruções IA-32 de cada vez e usaria um conjunto de PLAs para gerar até seis microinstruções por ciclo. Com a velocidade de clock significativamente mais alta introduzida no Pentium 4, essa solução deixou de ser adequada e um método inteiramente novo de gerar microinstruções tornou-se necessário.

A solução adotada no Pentium 4 é incluir um **cache de trace** de microinstruções, que é acessado pelo contador de programa da IA-32. Um cache de trace é uma forma sofisticada de cache de instruções, que explicaremos em detalhes no Capítulo 7. Por enquanto, pense nele como um buffer contendo as microinstruções que implementam uma determinada instrução IA-32. Quando a cache de trace é acessada com o endereço da próxima instrução IA-32 a ser executada, um dos seguintes eventos ocorre:

- A tradução da instrução IA-32 está na cache de trace. Nesse caso, até três microinstruções são produzidas por meio da cache de trace. Essas três microinstruções representam de uma a três instruções IA-32. O PC da IA-32 é incrementado em uma a três instruções dependendo de quantas couberem na seqüência de três microinstruções.
- A tradução da instrução IA-32 está na cache de trace, mas requer mais de quatro microinstruções para ser implementada. Para essas instruções IA-32 complexas, existe uma ROM de microcódigo; a unidade de controle transfere o controle para o microprograma até que a instrução IA-32 mais complexa tenha sido completada. A ROM de microcódigo fornece um total de mais de 8.000 microinstruções, com um número de seqüências sendo compartilhadas entre instruções IA-32. O controle, então, é transferido de volta para buscar instruções da cache de trace.
- A tradução da instrução IA-32 designada não está na cache de trace. Nesse caso, um decodificador de instruções IA-32 é usado para decodificar a instrução IA-32. Se o número de microinstruções for quatro ou menos, as microinstruções decodificadas são colocadas na cache de trace, onde podem ser encontradas na próxima execução dessa instrução. Caso contrário, a ROM de microcódigo é usada para completar a seqüência.

cache de trace Uma cache de instruções que contém uma seqüência de instruções com um endereço inicial conhecido; nas recentes implementações Pentium, a cache de trace contém microoperações em vez de instruções IA-32.

De uma a três microinstruções são enviadas da cache de trace para o pipeline de microinstruções do Pentium 4, que descrevemos em detalhes no final do Capítulo 6. O uso do controle hardwired simples de baixo nível e dos caminhos de dados simples para manipular as microinstruções juntamente com a cache de trace das instruções decodificadas permite ao Pentium 4 obter velocidades de clock impressionantes, semelhantes às dos microprocessadores implementando conjunto de instruções mais simples. Além disso, o processo de tradução, que combina controle hardwired direto para instruções simples com controle microcodificado para instruções complexas, permite que o Pentium 4 execute as instruções simples de alta freqüência no conjunto de instruções IA-32 em uma alta velocidade, produzindo um CPI baixo e bastante competitivo.

Entendendo o desempenho dos programas

Embora a maioria do desempenho do Pentium 4, ignorando o sistema de memória, dependa da eficiência das microoperações em pipeline, a eficácia do front-end em decodificar instruções IA-32 pode ter um efeito significativo sobre o desempenho. Em especial, devido à estrutura do decodifica-

despacho Uma operação em uma unidade de controle microprogramada em que a próxima microinstrução é selecionada baseada em um ou mais campos de uma macroinstrução, normalmente criando uma tabela contendo os endereços das microinstruções de destino e indexando a tabela usando um campo da macroinstrução. As tabelas de despacho em geral são implementadas em ROM ou em array lógico programável (PLA). O termo *despacho* também é usado em processadores dinamicamente escalonados em referência ao processo de enviar uma instrução para uma fila.

dor, usar instruções IA-32 mais simples que exigem quatro ou menos microoperações e, portanto, evitar um **despacho** de microcódigo provavelmente leva a um melhor desempenho. Por causa dessa estratégia de implementação (e uma semelhante no Pentium III), escritores de compiladores e os programadores assembly devem procurar fazer uso de seqüências de instruções IA-32 simples em vez de alternativas mais complexas.

5.10 Falácia e armadilhas

Armadilha: incluir uma instrução complexa implementada com microprogramação pode não ser mais rápido do que uma seqüência usando instruções mais simples.

A maioria das máquinas com um conjunto de instruções grande e complexo é implementada, pelo menos em parte, usando microcódigo armazenado em ROM. Surpreendentemente, nessas máquinas, as seqüências de instruções individuais mais simples, algumas vezes, são tão rápidas ou até mais rápidas do que a seqüência de microcódigo personalizada para uma instrução específica.

Como isso pode ser verdade? Antigamente, o microcódigo tinha a vantagem de ser buscado de uma memória muito mais rápida do que as instruções do programa. Desde que as caches entraram em uso em 1968, o microcódigo deixou de ter esse aspecto consistente no tempo de busca. Entretanto, o microcódigo ainda tem a vantagem de usar registradores temporários internos no cálculo, o que pode ser útil em máquinas com poucos registradores de uso geral. A desvantagem do microcódigo é que os algoritmos precisam ser selecionados antes de a máquina ser anunciada e não podem ser mudados até o próximo modelo da arquitetura. As instruções em um programa, por outro lado, podem usar melhorias nesses algoritmos a qualquer hora durante a vida da máquina. Da mesma forma, a seqüência de microcódigo provavelmente não é ótima para todas as combinações possíveis de operandos.

Um exemplo dessas instruções nas implementações IA-32 é a instrução move string (MOVS) usada com um prefixo de repetição que discutimos no Capítulo 2. Essa instrução normalmente é mais lenta do que um loop que move words de cada vez, como vemos nesta seção.

Outro exemplo envolve a instrução LOOP, que decrementa um registrador e desvia para o rótulo especificado se o registrador decrementado não for igual a zero. Essa instrução deve ser usada como o desvio no final dos loops que possuem um número fixo de iterações (por exemplo, muitos loops *for*). Essa instrução, além de conter algum trabalho extra, oferece vantagens em minimizar as perdas potenciais do desvio em máquinas com pipeline (como veremos quando discutirmos os desvios no próximo capítulo).

Infelizmente, em todas as recentes implementações Intel IA-32, a instrução LOOP é sempre mais lenta do que a seqüência de macrocódigo consistindo em instruções individuais mais simples (considerando que a pequena diferença no tamanho de código não é um fator relevante). Portanto, a otimização de compiladores focalizando a velocidade nunca gera a instrução LOOP. Isso, por sua vez, dificulta a motivação para tornar o LOOP rápido nas futuras implementações, já que ele é usado muito raramente!

Falácia: se houver espaço no armazenamento de controle, novas instruções não terão custo algum.

Uma das vantagens de um método microprogramado é que o armazenamento de controle implementado em ROM não é muito caro e, como a quantidade de transistores cresceu, a ROM extra se tornou praticamente de graça. A analogia aqui é construir uma casa e descobrir, quase no término da obra, que ainda sobrou espaço e materiais para acrescentar um cômodo. Esse cômodo, no entanto,

não seria gratuito, já que haveria os custos do trabalho e da manutenção por toda a vida da casa. A tentação de acrescentar instruções “gratuitas” pode ocorrer apenas quando o conjunto de instruções não é fixo, como provavelmente seria o caso no primeiro modelo de um computador. Como a compatibilidade futura dos programas binários é um recurso altamente desejável, todos os futuros modelos dessa máquina serão forçados a incluir essas chamadas instruções gratuitas, mesmo se o espaço, mais tarde, for escasso.

Durante o projeto do 80286, muitas instruções foram acrescentadas ao conjunto de instruções. A disponibilidade de mais recurso de silício e o uso da implementação micropogramada tornaram essas adições aparentemente simples. Possivelmente, a maior adição foi um sofisticado mecanismo de proteção, que é pouquíssimo utilizado hoje, mas que ainda precisa ser implementado em versões mais recentes. Essa adição foi motivada por uma necessidade percebida desse tipo de mecanismo e o desejo de aperfeiçoar as arquiteturas de microprocessador para fornecer funcionalidade igual à dos computadores maiores. Da mesma forma, diversas instruções decimais foram incluídas para fornecer aritmética decimal em bytes. Essas instruções raramente são utilizadas hoje porque usar aritmética binária em 32 bits e converter de e para representação decimal é consideravelmente mais rápido. Assim como os mecanismos de proteção, as instruções decimais precisam ser implementadas nos processadores mais novos, mesmo se usadas raramente.

5.11 Comentários finais

Como vimos neste capítulo, o caminho de dados e o controle para um processador podem ser projetados começando com o conjunto de instruções e um entendimento das características básicas da tecnologia. Na Seção 5.3, vimos como o caminho de dados para um processador MIPS poderia ser construído com base na arquitetura e na decisão de construir uma implementação de ciclo único. Naturalmente, a tecnologia básica também afeta muitas decisões de projeto determinando que componentes podem ser usados no caminho de dados e até mesmo se uma implementação de ciclo único faz sentido. Do mesmo modo, na primeira parte da Seção 5.5, vimos como a decisão de dividir o ciclo de clock em uma série de etapas levou ao caminho de dados multiciclo revisado. Nos dois casos, a organização de nível superior – uma máquina multiciclo ou de ciclo único –, juntamente com o conjunto de instruções, recomendou muitas características do projeto do caminho de dados.

O controle pode ser projetado usando uma das várias representações iniciais. A escolha do controle de seqüenciação e de como a lógica é representada pode, então, ser determinada de forma independente; o controle, então, pode ser implementado com diversos métodos usando uma técnica de lógica estruturada. A Figura 5.41 mostra a variedade de métodos para especificar o controle e para ir da especificação para uma implementação usando alguma forma de lógica estruturada.

Colocando em perspectiva

De modo semelhante, o controle é principalmente definido pelo conjunto de instruções, pela organização e pelo projeto do caminho de dados. Na organização de ciclo único, esses três aspectos basicamente definem como os sinais de controle precisam ser definidos. No projeto multiciclo, a decomposição exata da execução das instruções em ciclos, que é baseada no conjunto de instruções, juntamente com o caminho de dados, define as necessidades do controle.

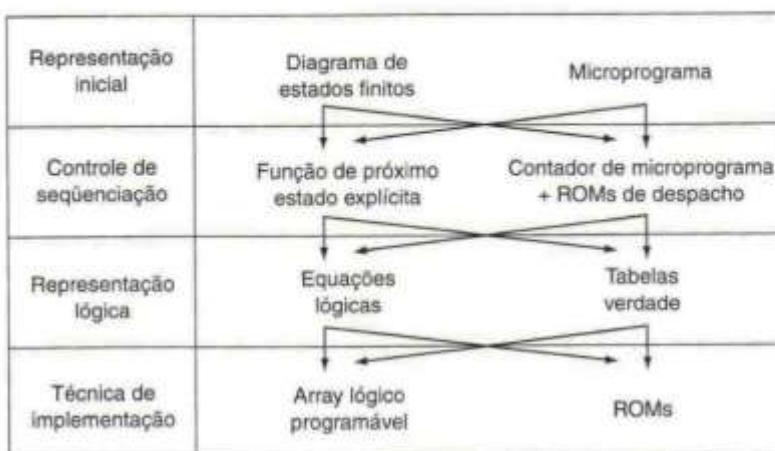


FIGURA 5.41 Métodos alternativos para especificar e implementar controle. As setas indicam possíveis caminhos de projeto: qualquer caminho da representação inicial até a tecnologia de implementação final é viável. Tradicionalmente, “controle hardwired” significa que as técnicas no lado esquerdo são usadas, e “controle microprogramado” significa que as técnicas no lado direito são usadas.

O controle é um dos aspectos mais difíceis do projeto de um computador. Uma importante razão é que projetar o controle requer um entendimento de como funcionam todos os componentes do processador. Para ajudar a satisfazer a esse requisito, examinamos duas técnicas de especificar controle: diagramas de estados finitos e microprogramação. Essas representações de controle permitem abstrair a especificação do controle dos detalhes de como implementá-la. Usar esse tipo de abstração é o principal recurso de que dispomos para fazer frente diante da complexidade dos projetos de computadores.

Uma vez especificado o controle, podemos mapeá-lo detalhadamente em hardware. Os detalhes exatos da implementação do controle dependerão da estrutura do controle e da tecnologia básica usada para implementá-lo. A abstração da especificação do controle também é valiosa porque as decisões de como implementar o controle são dependentes da tecnologia e provavelmente mudam com o tempo.

5.12

Perspectiva histórica e leitura adicional

O surgimento da micropogramação e seu efeito sobre o projeto do conjunto de instruções e sobre o desenvolvimento dos computadores é uma das interações mais interessantes nas primeiras décadas do computador eletrônico. A história é o foco da seção de perspectivas históricas no CD.

5.13

Exercícios

5.1 [6] <§5.2> Precisamos de lógica combinacional, lógica seqüencial ou uma combinação das duas para implementar cada um dos seguintes itens?

- multiplexador
- comparador
- incrementador/decrementador
- deslocador (barrel shifter)
- multiplicador com deslocadores e somadores

- f. registrador
- g. memória
- h. ALU (aqueles nos caminhos de dados de ciclo único e multiciclo)
- i. somador com carry look-ahead
- j. latch
- k. máquina de estados finitos genérica (FSM)

5.2 [10] <§5.4> Descreva o efeito que teria uma falha “stuck-at-0” (ou seja, independente do que deveria ser, o sinal é sempre 0) para os sinais mostrados a seguir, no caminho de dados de ciclo único da Figura 5.17. Que instruções, se houver, não funcionarão corretamente? Explique por quê. Considere cada uma das falhas separadamente:

- a. EscreveReg = 0
- b. ALUop0 = 0
- c. ALUop1 = 0
- d. Branch = 0
- e. LeMem = 0
- f. EscreveMem = 0

5.3 [5] <§5.4> Este exercício é semelhante ao Exercício 5.2, mas, desta vez, considere falhas stuck-at-1 (o sinal é sempre 1).

5.4 [5] <§5.4> ■ **Aprofundando o aprendizado:** Caminhos de dados de ciclo único com ponto flutuante.

5.5 [5] <§5.4> ■ **Aprofundando o aprendizado:** Caminhos de dados de ciclo único com ponto flutuante.

5.6 [10] <§5.4> ■ **Aprofundando o aprendizado:** Caminhos de dados de ciclo único com ponto flutuante.

5.7 [2 a 3 meses] <§§5.1–5.4> Usando peças comuns, construa uma máquina que implemente a máquina de ciclo único neste capítulo.

5.8 [15] <§5.4> Desejamos acrescentar a instrução `jr` (jump register) ao caminho de dados de ciclo único descrito neste capítulo. Inclua quaisquer caminhos de dados e sinais de controle necessários no caminho de dados de ciclo único da Figura 5.17 e mostre as adições necessárias à Figura 5.18. Você pode tirar cópias dessas figuras para que seja mais rápido mostrar essas adições.

5.9 [10] <§5.4> Esta questão é semelhante ao Exercício 5.8, exceto que desejamos incluir a instrução `sll` (shift left logical), descrita na Seção 2.5.

5.10 [15] <§5.4> Esta questão é semelhante ao Exercício 5.8, exceto que desejamos incluir a instrução `lui` (load upper immediate), descrita na Seção 2.9.

5.11 [20] <§5.4> Esta questão é semelhante ao Exercício 5.8, exceto que desejamos incluir uma variante da instrução `lw` (load word), que incrementa o registrador de índice após ler words da memória. Essa instrução (`l_inc`) corresponde a estas duas instruções:

```
lw $rs,L($rt)
addi $rt,$rt,1
```

5.12 [5] <§5.4> Explique por que não é possível modificar a implementação de ciclo único para implementar a instrução load with increment descrita no Exercício 5.11 sem modificar o banco de registradores.

5.13 [7] <§5.4> Considere o caminho de dados de ciclo único na Figura 5.17. Um amigo está proposto a modificar esse caminho de dados de ciclo único eliminando o sinal de controle `MemparaReg`.

O multiplexador que possui MemparaReg como entrada usará, no lugar dele, o sinal de controle OrigALU ou o sinal LeMem. A modificação do seu amigo funcionará? Um dos dois sinais (LeMem e OrigALU) pode substituir o outro? Explique.

5.14 [10] <§5.4> O MIPS escolhe simplificar a estrutura de suas instruções. A forma como implementamos instruções complexas por meio do uso de instruções MIPS é decompor essas instruções complexas em múltiplas instruções MIPS mais simples. Mostre como o MIPS pode implementar a instrução swap \$rs, \$rt, que troca o conteúdo dos registradores \$rs e \$rt. Considere o caso em que há um registrador disponível que pode ser destruído, assim como o caso em que nenhum registrador desses existe.

Se a implementação dessa instrução em hardware aumentar o período de clock da implementação de uma única instrução em 10%, que porcentagem das operações de troca no mix de instruções recomendaria implementá-la em hardware?

5.15 [5] <§5.4> **■ Aprofundando o aprendizado:** Efeitos das falhas nos multiplexadores de controle

5.16 [5] <§5.4> **■ Aprofundando o aprendizado:** Efeitos das falhas nos multiplexadores de controle

5.17 [5] <§5.5> **■ Aprofundando o aprendizado:** Efeitos das falhas nos multiplexadores de controle

5.18 [5] <§5.5> **■ Aprofundando o aprendizado:** Efeitos das falhas nos multiplexadores de controle

5.19 [15] <§5.4> **■ Aprofundando o aprendizado:** Acrescentando instruções ao caminho de dados

5.20 [15] <§5.4> **■ Aprofundando o aprendizado:** Acrescentando instruções ao caminho de dados

5.21 [8] <§5.4> **■ Aprofundando o aprendizado:** Acrescentando instruções ao caminho de dados

5.22 [8] <§5.4> **■ Aprofundando o aprendizado:** Acrescentando instruções ao caminho de dados

5.23 [10] <§5.4> **■ Aprofundando o aprendizado:** Sinais de controle do caminho de dados

5.24 [10] <§5.4> **■ Aprofundando o aprendizado:** Sinais de controle do caminho de dados

5.25 [15] <§5.4> **■ Aprofundando o aprendizado:** Modificando o caminho de dados e o controle

5.26 [8] <§5.4> Repita o Exercício 5.14, mas aplique sua solução na instrução load with increment: l_incr \$rt, Endereço(\$rs).

5.27 [5] <§5.4> O conceito de “caminho crítico”, o caminho mais longo possível na máquina, foi introduzido em 5.4. Com base em nosso entendimento da implementação de ciclo único, mostre que unidades podem tolerar mais atrasos (ou seja, não estão no caminho crítico) e que unidades podem se beneficiar da otimização do hardware. Quantifique suas respostas usando os mesmos números apresentados na Seção 5.4, “Exemplo: Desempenho das Máquinas de Ciclo Único”.

5.28 [5] <§5.5> Este exercício é semelhante ao Exercício 5.2, mas, desta vez, considere o efeito que as falhas stuck-at-0 teriam sobre o caminho de dados multiciclo na Figura 5.27. Considere cada uma das seguintes falhas:

- EscreveReg = 0
- LeMem = 0
- EscreveMem = 0
- EscreveIR = 0
- EscrevePC = 0
- EscrevePCCond = 0.

5.29 [5] <§5.5> Este exercício é semelhante ao Exercício 5.28, mas, desta vez, considere falhas stuck-at-1 (o sinal é sempre 1).

5.30 [15] <§§5.4, 5.5> Este exercício é semelhante ao Exercício 5.13, porém mais geral. Determine se qualquer um dos sinais de controle na implementação de ciclo único pode ser eliminado e substituído por outro sinal de controle existente ou seu inverso. Note que essa redundância existe porque

temos um conjunto muito pequeno de instruções nesse momento, e ela desaparecerá (ou será mais difícil de encontrar) quando implementarmos um número maior de instruções.

5.31. [15] <§5.5> Queremos acrescentar a instrução `lui` (load upper immediate) descrita no Capítulo 3 no caminho de dados multiciclo descrito neste capítulo. Use a mesma arquitetura do caminho de dados multiciclo da Figura 5.28. Você pode achar útil examinar as etapas de execução mostradas nas páginas 245 a 247 e considerar as etapas que precisarão ser realizadas para executar a nova instrução. Quantos ciclos são necessários para implementar essa instrução?

5.32 [15] <§5.5> Você foi solicitado a modificar a implementação do `lui` no Exercício 5.31 para reduzir o tempo de execução em 1 ciclo. Inclua quaisquer caminhos de dados e sinais de controle necessários no caminho de dados multiciclo da Figura 5.28. Você pode tirar cópias das figuras existentes para que seja mais rápido mostrar suas modificações. Você precisa manter a suposição de que não sabe qual é a instrução antes do final do estado 1 (fim do segundo ciclo). Diga explicitamente quantos ciclos são necessários para executar a nova instrução em seu caminho de dados e máquina de estados finitos modificados.

5.33 [20] <§5.5> Esta questão é semelhante ao Exercício 5.31, exceto que desejamos implementar uma nova instrução `ldi` (load immediate) que lê um valor imediato de 32 bits do local da memória seguinte ao endereço da instrução.

5.34 [15] <§5.5> Considere uma mudança na implementação multiciclo que altere o banco de registradores de modo que ele tenha apenas uma porta de leitura. Descreva (por meio de um diagrama) quaisquer mudanças adicionais que precisarão ser feitas no caminho de dados para suportar essa alteração. Modifique a máquina de estados finitos para indicar como as instruções funcionarão, dado seu novo caminho de dados.

5.35 [15] <§5.5> Dois importantes parâmetros controlam o desempenho de um processador: o tempo de ciclo e o número de ciclos por instrução. Existe um compromisso permanente entre esses dois parâmetros no processo de projeto dos microprocessadores. Embora alguns projetistas prefiram aumentar a frequência do processador à custa de um CPI alto, outros projetistas seguem uma escola de pensamento diferente em que reduzir o CPI ocorre à custa de uma menor frequência do processador.

Considere as seguintes máquinas e compare seu desempenho usando os dados do SPEC CPUint 2000 da Figura 3.26.

M1: o caminho de dados multiciclo do Capítulo 5 com um clock de 1GHz.

M2: uma máquina com o caminho de dados multiciclo do Capítulo 5, exceto que as atualizações de registradores são feitas no mesmo ciclo de clock de uma leitura de memória ou operação da ALU.

Portanto, na Figura 5.38, os estados 6 e 7 e os estados 3 e 4 são combinados. Essa máquina possui um clock de 3,2GHz, já que a atualização dos registradores aumenta a duração do caminho crítico.

M3: uma máquina como a M2 exceto que os cálculos de endereço efetivo são feitos no mesmo ciclo de clock de um acesso à memória. Assim, os estados 2, 3 e 4 podem ser combinados, como podem os estados 2 e 5 e também 6 e 7. Essa máquina tem um clock de 2,8GHz devido ao longo ciclo criado pela combinação do cálculo de endereço e o acesso à memória.

Descubra qual das máquinas é a mais rápida. Existem mix de instruções que tornariam outra máquina mais rápida? Se existem, quais são eles?

5.36 [20] <§5.5> Seus amigos da C³ (Creative Computer Corporation) determinaram que o caminho crítico que define a duração do ciclo de clock do caminho de dados multiciclo é o acesso à memória para loads e stores (não para buscar instruções). Isso fez com que sua mais nova implementação do MIPS 30000 fosse executada em uma velocidade de clock de 4,8GHz em vez da velocidade de clock almejada de 5,6GHz. Entretanto, Clara da C₃ tem uma solução. Se todos os ciclos que acessam a memória forem divididos em dois ciclos de clock, então, a máquina poderá rodar em sua velocidade de clock almejada.

Usando os mix do SPEC CPUint 2000 mostrados no Capítulo 3 (Figura 3.26), determine o quanto é mais rápida a máquina com os acessos à memória de dois ciclos, comparada com a máquina de 4,8GHz com acesso à memória de ciclo único. Considere que todos os jumps e branches levam o mesmo número de ciclos e que as instruções set e as instruções aritméticas immediate são implemen-

tadas como instruções tipo R. Você consideraria a etapa extra de dividir a busca de instrução em dois ciclos se isso aumentasse a velocidade de clock para até 6,4GHz? Por quê?

5.37 [20] <§5.5> Suponha que houvesse uma instrução MIPS, chamada `bcmp`, que comparasse dois blocos de words em dois endereços de memória. Considere que essa instrução exige que o endereço inicial do primeiro bloco esteja no registrador `$t1` e o endereço inicial do segundo bloco esteja em `$t2`, e que o número de words a comparar esteja em `$t3` (que é $\$t3=0$). Considere que a instrução pode deixar o resultado (o endereço da primeira desigualdade ou zero se é uma completa igualdade) em `$t1` e/ou `$t2`. Além disso, considere que os valores desses registradores bem como os registradores `$t4` e `$t5` podem ser destruídos em executar essa instrução (para que os registradores possam ser usados como temporários para executar a instrução).

Escreva o programa em assembly MIPS para implementar a (emular o comportamento da) comparação de blocos. Quantas instruções serão executadas para comparar dois blocos de 100 words? Usando o CPI das instruções na implementação multiciclo, quantos ciclos são necessários para a comparação de blocos de 100 words?

5.38 [2 a 3 meses] <§§5.1–5.5> Usando peças comuns, construa uma máquina que implemente a máquina multiciclo deste capítulo.

5.39 [15] <§5.5> ■ **Aprofundando o aprendizado:** Acrescentando instruções ao caminho de dados

5.40 [15] <§5.5> ■ **Aprofundando o aprendizado:** Acrescentando instruções ao caminho de dados

5.41 [15] <§5.5> ■ **Aprofundando o aprendizado:** Acrescentando instruções ao caminho de dados

5.42 [15] <§5.5> ■ **Aprofundando o aprendizado:** Acrescentando instruções ao caminho de dados

5.43 [20] <§5.5> ■ **Aprofundando o aprendizado:** Acrescentando instruções ao caminho de dados

5.44 [10] <§5.5> ■ **Aprofundando o aprendizado:** Acrescentando instruções ao caminho de dados

5.45 [15] <§§5.1–5.5> ■ **Aprofundando o aprendizado:** Comparando o desempenho do processador

5.46 [20] <§5.5> ■ **Aprofundando o aprendizado:** Implementando instruções no MIPS

5.47 [30] <§5.6> Queremos acrescentar a instrução `eret` (retorno de exceção) ao caminho de dados multiciclo descrito neste capítulo. Uma tarefa importante da instrução `eret` é recarregar o PC com o endereço de retorno em que ocorreu uma exceção ou trap de erro. Suponha que, se o processador estiver servindo uma trap de erro, então, o PC precisa ser carregado de um registrador `ErroPC`. Caso contrário, o processador está servindo uma exceção (o PC precisa ser carregado do EPC). Suponha que há um bit no registrador `Cause` chamado `trap` para codificar uma trap de erro quando ela ocorre e para salvar o PC no registrador `ErroPC`. Inclua quaisquer caminhos de dados e sinais de controle necessários no caminho de dados multiciclo da Figura 5.39 para acomodar a chamada e o retorno do trap/exceção, e mostre as modificações necessárias na máquina de estados finitos da Figura 5.40 para implementar a instrução `eret`. Você pode tirar cópias das figuras para facilitar a demonstração de suas alterações.

5.48 [6] <§5.6> As exceções ocorrem quando é necessária uma mudança no fluxo de controle para manipular um evento inesperado no processador. Como a causa e a instrução que causou a exceção podem ser representadas pelo hardware em uma máquina MIPS? Forneça dois exemplos de condições que um processador pode tratar reiniciando a execução da instrução após tratar a exceção, e dois outros exemplos para exceções que levam ao término do programa.

5.49 [6] <§5.6> A detecção de exceção é um importante aspecto do tratamento de exceções. Tente identificar o ciclo em que as seguintes exceções podem ser detectadas para o caminho de dados multiciclo da Figura 5.28.

Considere as seguintes exceções:

- Divisão por zero (suponha que usamos a mesma ALU para divisão em um ciclo e que ela é reconhecida pelo restante do controle)
- Overflow

- c. Instrução inválida
- d. Interrupção externa
- e. Endereço de memória de instrução inválido
- f. Endereço de memória de dados inválido

5.50 [15] <§5.6> ■ **Aprofundando o aprendizado:** Acrescentando instruções ao caminho de dados

5.51 [30] <§5.7> O microcódigo tem sido usado para incluir instruções mais poderosas em um conjunto de instruções; vamos explorar as possíveis vantagens desse método. Crie uma estratégia para implementar a instrução `bcmp` descrita no Exercício 5.37 usando o caminho de dados multiciclo e o microcódigo. Você provavelmente precisará fazer algumas mudanças no caminho de dados para implementar eficientemente a instrução `bcmp`. Forneça uma descrição das duas mudanças propostas e descreva como a instrução `bcmp` funcionará. Existe alguma vantagem que pode ser obtida incluindo registradores internos no caminho de dados para ajudar a suportar a instrução `bcmp`? Estime a melhora de desempenho que você pode obter implementando a instrução em hardware (em comparação à solução de software obtida no Exercício 5.37) e explique de onde vem o aumento de desempenho.

5.52 [30] <§5.7> ■ **Aprofundando o aprendizado:** Microcódigo

5.53 [30] <§5.7> ■ **Aprofundando o aprendizado:** Microcódigo

5.54 [5] <§5.7> ■ **Aprofundando o aprendizado:** Microcódigo

5.55 [30] <§5.8> Usando a estratégia desenvolvida no Exercício 5.51, modifique o formato de microinstrução MIPS descrito na ■ Figura 5.7.1 e forneça o microprograma completo para a instrução `bcmp`. Descreva em detalhes como você estendeu o microcódigo de modo a suportar a criação de estruturas de controle mais complexas (como um loop) dentro do microcódigo. O suporte para a instrução `bcmp` alterou o tamanho do microcódigo? Outras instruções além da `bcmp` serão afetadas pela mudança no formato da microinstrução?

5.56 [5] <§5.8> A e B são registradores definidos por meio do seguinte código de inicialização Verilog:

```
reg A,B  
initial begin  
    A = 1;  
    B = 2;  
end
```

Analise os dois segmentos de descrição Verilog a seguir, e compare os resultados das variáveis A e B e a operação feita em cada exemplo.

a) `always @(negedge clock) begin`
 `A = B;`
 `B = A;`
 `end`

b) `always @(negedge clock) begin`
 `A <= B;`
 `B <= A;`
 `end`

5.57 [15] <§§5.4, 5.8> Escreva o módulo ALUControl em Verilog combinacional usando a seguinte forma como base:

```
module ALUControl (OpALU, FuncCode, ALUControl);  
    input OpALU[1:0], FuncCode[5:0];  
    output ALUControl[3:0];  
    ....  
endmodule
```

5.58 [1 semana] <§§5.3, 5.4, 5.8> Usando uma linguagem de simulação de hardware como Verilog, implemente um simulador funcional para a versão de ciclo único. Construa seu simulador usando uma biblioteca de peças existentes, se estiver disponível. Se as peças contiverem informações de sincronização, determine qual será o tempo de ciclo de sua implementação.

5.59 [2 a 4 horas] <§§4.7, 5.5, 5.8, 5.8> Estenda a descrição Verilog de multiciclo em § 5.8 acrescentando uma implementação da instrução multiply sem sinal do MIPS; considere que ela é implementada usando a ALU MIPS e uma operação shift and add.

5.60 [2 a 4 horas] <§§4.7, 5.5, 5.8, 5.9> Estenda a descrição Verilog de multiciclo em § 5.8 acrescentando uma implementação da instrução divide sem sinal do MIPS; considere que ela é implementada usando a ALU MIPS com um algoritmo de um bit por vez.

5.61 [1 semana] <§§5.5, 5.8> Usando uma linguagem de simulação de hardware como Verilog, implemente um simulador funcional para uma implementação multiciclo do projeto de um processador PowerPC. Construa seu simulador usando uma biblioteca de peças existente, se estiver disponível. Se as peças contiverem informações de sincronização, determine qual será o tempo de ciclo de sua implementação.

Como o MIPS, as instruções PowerPC possuem 32 bits cada uma. Considere que seu conjunto de instruções suporta os seguintes formatos de instrução:

tipo R

	op	Rd	Rt	Rs	0	Func	RC
0	5 6	10 11	15 16	20 21 22			30 31

Load/store & immediate

	op	Rd	Rt	Address			
0	5 6	10 11	15 16				31

Desvio condicional

	op	0	BI	BD	AA LK	
0	5 6	10 11	15 16		29	30 31

Jump

	op	Address					AA LK
0	5 6	10 11	15 16				29 30 31

RC-reg

	LT	GT	EQ	OV	
0 1 2 3					

Campo Func (22:30): semelhante ao MIPS, identifica o código de função.

Bit RC (31): se for (1), atualiza os bits de controle RC-reg para refletirem os resultados da instrução (todas as tipo R)

AA(30): 1 indica que o endereço dado é um endereço absoluto; 0 indica endereço relativo

LK: se 1, atualiza LNKR (o registrador de link), que pode ser usado mais tarde para implementação de retorno de sub-rotina

BI: codifica a condição de desvio (por exemplo, beg -> BI = 2, blt -> BI = 0 etc.)

BD: destino do desvio relativo.

Sua implementação PowerPC simplificada deve ser capaz de implementar as seguintes instruções:

```
add:      add $Rd, $Rt, $Rs      ($Rd <- $Rt + $Rs)
          addi $Rd, $Rt, #n     ($Rd <- $Rt + #n)
subtract: sub $Rd, $Rt, $Rs      ($Rd <- $Rt - $Rs)
          subi $Rd, $Rt, #n    ($Rd <- $Rt - #n)
load:     lw $Rd, Addr($Rt)    ($Rd <- Memória[$Rt + Addr])
store:    sw $Rd, Addr($Rt)    (Memória[$Rt + Addr] <- $Rd)
AND, OR:   and/or $Rd, $Rt, $Rs   ($Rd <- $Rt AND/OR $Rs)
          andi/ori $Rd, $Rt, #n   ($Rd <- $Rt AND/OR #n)
Jump:     jmp Addr           (PC <- Addr)
Desvio condicional: Beq Addr  (CR[2]==1? PC<- PC+BD : PC <- PC+4)
chamada de subrotina: jal Addr (LNKR <- PC+4; PC<- Addr)
retorno de subrotina: Ret      (PC <- LNK)
```

5.62 [Discussão] <§§5.7, 5.10, 5.11> Hipótese: se a primeira implementação de uma arquitetura usar microprogramação, ela afetará o conjunto de instruções. Por que isso poderia ser verdade? Você pode encontrar uma arquitetura que provavelmente usará sempre microcódigo? Por quê? Que máquinas nunca usarão microcódigo? Por quê? Que implementação você acha que o arquiteto tinha em mente quando projetou o conjunto de instruções?

5.63 [Discussão] <§§5.7, 5.12> Wilkes inventou a microprogramação principalmente para simplificar a construção do controle. Desde 1980, tem havido uma explosão de software de projeto auxiliado por computador, cujo objetivo também é simplificar a construção do controle. Isso tornou o projeto do controle muito mais fácil. Você pode encontrar evidências, com base nas ferramentas ou em projetos reais, que apóiam ou refutam essa hipótese?

5.64 [Discussão] <§5.12> As instruções MIPS e as microinstruções MIPS possuem muitas semelhanças. O que dificultaria para um compilador produzir microcódigo MIPS em vez de macrocódigo? Que mudanças na microarquitetura tornariam o microcódigo mais útil para essa aplicação?

§5.1: página 217, 3.

§5.2: página 219, falso.

§5.3: página 225, A.

§5.4: página 239, sim, MemparaReg e RegDst são inversos um do outro. Sim, simplesmente use o **você mesmo** "outro sinal e inverta a ordem das entradas do multiplexador!

§5.5: página 256, 1. Falso. 2. Talvez: se o sinal OrigPC[0] estiver sempre com valor zero quando ele é um don't care (que é a maioria dos estados), então, ele será idêntico a EscrevePCCond.

§5.6: página 261, não, já que o valor 11, que antigamente não era usado, agora é usado!

§5.7: página 5.7-13, Quatro tabelas com 55 entradas (não se esqueça do despacho primário!)

**Respostas
das Seções
“Verifique
você mesmo”**

Computadores no mundo real

Capacitando o deficiente

Problema: superar os obstáculos encontrados pelas pessoas portadoras de deficiência física.

Solução: usar a robótica, sensores e controle computadorizado para substituir ou suplementar membros ou órgãos danificados.

A fotografia à direita mostra um sistema desenvolvido para um bombeiro acidentado durante um combate a um incêndio. Os sensores nos dedos de látex registram instantaneamente o calor e o frio, e uma interface eletrônica em seu órgão artificial estimula os terminais nervosos em seu braço, que, então, transferem as informações para o cérebro. O sistema de US\$3.000 permite que seu braço sinta a pressão e o peso; assim, pela primeira vez desde que perdeu seu braço em 1986, Ken pode apanhar uma lata de refrigerante sem esmagá-la ou deixá-la escorregar entre seus dedos. O principal dispositivo que possibilita isso é uma interface eletrônica capaz de transmitir sinais aos terminais nervosos do braço de Ken, que, então, transferem as informações ao seu cérebro.

Harvey Fishman e Mark Peterman, da Universidade de Stanford, deram passos em direção à tecnologia da informação que, algum dia, poderá tratar da cegueira relacionada à idade. Seu méto-



O bombeiro Ken Whitten exibe orgulhosamente seu novo braço biônico.

do é evitar os fotorreceptores do olho com um sinal de uma câmera digital conectada diretamente ao sistema visual. Eles estão desenvolvendo uma interface neural com o sistema visual chamada chip de sinapse artificial. O desafio é transformar sinais elétricos nas substâncias químicas que as células usam para se comunicarem. Esse chip é conectado às células e, da perspectiva da célula, a sinapse artificial é simplesmente um furo no silício. Esse furo é conectado a um reservatório de neurotransmissor. Quando um campo elétrico é aplicado no chip, o neurotransmissor é bombeado

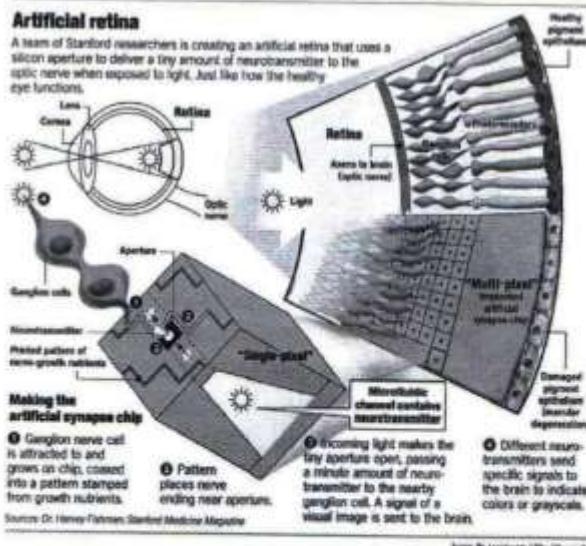
através do furo, estimulando as células vizinhas. Em 2003, eles criaram quatro sinapses artificiais em um chip de um centímetro de largura.

Embora esse trabalho ainda esteja em sua fase embrionária, o potencial não está limitado aos problemas de visão. Segundo Fishman, "em qualquer lugar onde haja um rompimento de nervo, há um potencial para o reconectarmos."

Para saber mais, veja estas referências

Rick Smolan e Jennifer Erwitt, *One Digital Day: How the Microchip Is Changing Our World*, Times Publishing, 1998.

Peterman et al., "The artificial synapse chip: A flexible retinal interface based on directed retinal cell growth and neurotransmitter stimulation", *Artificial Organs*: 27(11), 18 de novembro de 2003.



Retina artificial usando chips de sinapse artificial.
Do *The San Francisco Chronicle*, 5 de janeiro de 2004.

6

Melhorando o Desempenho com Pipelining

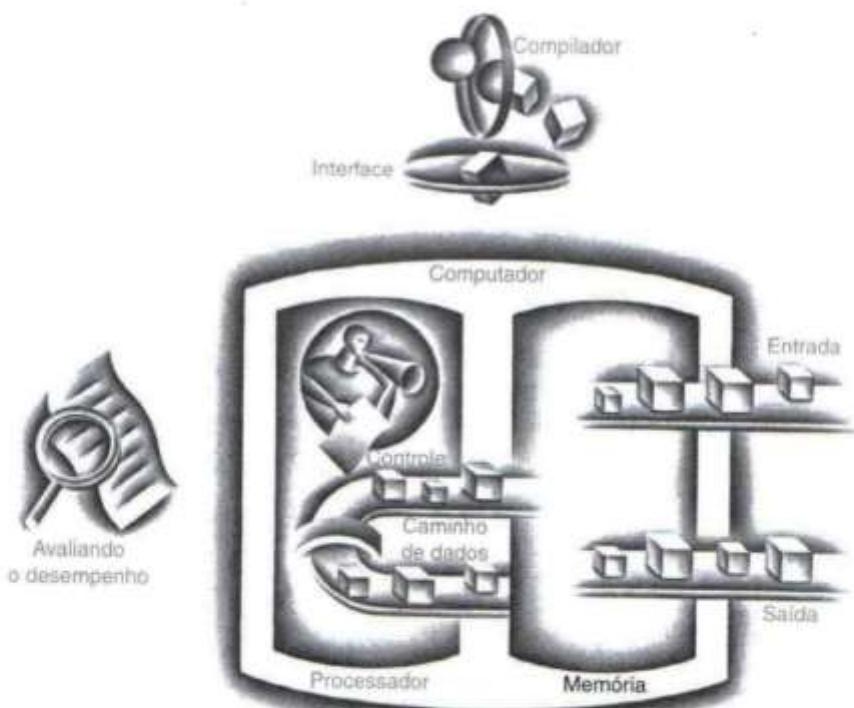
*Enquanto o tempo passa,
todas as coisas permanecem;
novas coisas acontecem,
enquanto as antigas envelhecem.*

Robert Herrick

Hesperides: Ceremonies for Christmas Eve, 1648

- 6.1 Visão geral de pipelining 278**
 - 6.2 Um caminho de dados usando pipeline 289**
 - 6.3 Controle de um pipeline 300**
 - 6.4 Hazards de dados e forwarding 303**
 - 6.5 Hazards de dados e stalls 311**
 - 6.6 Hazards de desvio 313**
 - 6.7 Usando uma linguagem de descrição de hardware para descrever e modelar um pipeline 322**
 - 6.8 Exceções 322**
 - 6.9 Pilelining avançado: extraíndo mais desempenho 326**
 - 6.10 Vida real: o pipeline do Pentium 4 338**
 - 6.11 Falácia e armadilhas 340**
 - 6.12 Comentários finais 341**
 - 6.13 Perspectiva histórica e leitura adicional 343**
 - 6.14 Exercícios 343**
-

Os cinco componentes clássicos de um computador



Nunca perca
tempo.
Provérbio
americano

pipelining Uma técnica de implementação em que várias instruções são sobrepostas na execução, semelhante a uma linha de montagem.

6.1 Visão geral de pipelining

Pipelining é uma técnica de implementação em que várias instruções são sobrepostas na execução. Hoje, a técnica de pipelining é fundamental para tornar os processadores mais rápidos.

Esta seção utiliza bastante uma analogia para dar uma visão geral dos termos e aspectos da técnica de pipelining. Se você estiver interessado apenas no quadro geral, deverá se concentrar nesta seção e depois pular para as Seções 6.9 e 6.10, para ver uma introdução às técnicas de pipelining avançadas, utilizadas nos processadores mais recentes, como o Pentium III e 4. Se estiver interessado em explorar a anatomia de um computador com pipeline, esta seção é uma boa introdução às Seções de 6.2 a 6.8.

Qualquer um que tenha lavado muitas roupas intuitivamente já usou pipelining. A técnica *sem pipeline* para lavar roupas seria

1. Colocar a trouxa suja de roupas na lavadora.
2. Quando a lavadora terminar, colocar a trouxa molhada na secadora (se houver).
3. Quando a secadora terminar, colocar a trouxa seca na mesa e passar.
4. Quando terminar de passar, pedir ao seu colega de quarto para guardar as roupas.

Quando seu colega terminar, então comece novamente com a próxima trouxa suja.

A técnica *com pipeline* leva muito menos tempo, como mostra a Figura 6.1. Assim que a lavadora terminar com a primeira trouxa e ela for colocada na secadora, você carrega a lavadora com a segunda trouxa suja. Quando a primeira trouxa estiver seca, você a coloca na tábua para começar a passar e dobrar, move a trouxa molhada para a secadora e a próxima trouxa suja para a lavadora. Em seguida,

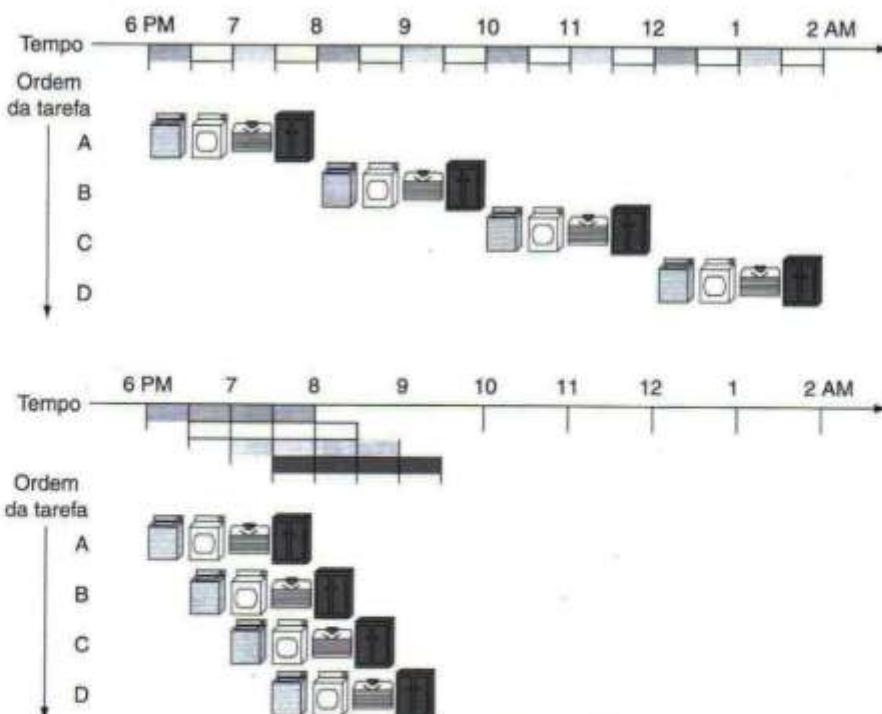


FIGURA 6.1 A analogia da lavagem de roupas para pipelining. Ana, Beto, Catarina e Davi possuem roupas sujas para serem lavadas, secadas, passadas e guardadas. O lavador, o secador, o passador e o guardador levam 30 minutos para sua tarefa. A lavagem sequencial levaria 8 horas para quatro trouxas de roupas, enquanto a lavagem com pipeline levaria apenas 3,5 horas. Mostramos o estágio do pipeline de diferentes trouxas com o passar do tempo mostrando cópias dos quatro recursos nessa linha de tempo bidimensional, mas na realidade temos apenas um de cada recurso.

você pede a seu colega para guardar a primeira remessa, começa a passar e dobrar a segunda, a seca-dora está com a terceira remessa e você coloca a quarta na lavadora. Nesse ponto, todas as etapas – denominadas *estágios* em pipelining – estão operando simultaneamente. Desde que haja recursos separados para cada estágio, podemos usar um pipeline para as tarefas.

O paradoxo da técnica de pipelining é que o tempo desde a colocação de uma única trouxa de roupa suja na lavadora até que ela seja secada, passada e guardada não é mais curto para a técnica de pipelining; o motivo pelo qual a técnica de pipelining é mais rápida para muitas trouxas é que tudo está trabalhando em paralelo, de modo que mais trouxas são terminadas por hora. A técnica de pipelining melhora a vazão do sistema de lavanderia sem melhorar o tempo para concluir uma única trouxa. Logo, a técnica de pipelining não diminuiria o tempo para concluir uma trouxa de roupas, mas, quando temos muitas trouxas para lavar, a melhoria na vazão diminui o tempo total para concluir o trabalho.

Se todos os estágios levarem aproximadamente o mesmo tempo e houver trabalho suficiente para realizar, então o ganho de velocidade devido à técnica de pipelining será igual ao número de estágios do pipeline, neste caso, quatro: lavar, secar, passar e guardar. Assim, a lavanderia com pipeline é potencialmente quatro vezes mais rápida do que a sem pipeline: 20 trouxas levariam cerca de 5 vezes o tempo de 1 trouxa, enquanto 20 trouxas de lavagem seqüencial levariam 20 vezes o tempo de 1 trouxa. O ganho foi de apenas 2,3 vezes na Figura 6.1 porque mostramos apenas 4 trouxas. Observe que, no início e no final da carga de trabalho na versão com pipeline da Figura 6.1, o pipeline não está completamente cheio. Esse efeito no inicio e no fim afeta o desempenho quando o número de tarefas não é grande em comparação com a quantidade de estágios do pipeline. Se o número de trouxas for muito maior que 4, então os estágios estarão cheios na maior parte do tempo e o aumento na vazão será muito próximo de 4.

Os mesmos princípios se aplicam a processadores em que usamos pipeline para a execução da instrução. As instruções MIPS normalmente exigem cinco etapas:

1. Buscar instrução da memória.
2. Ler registradores enquanto a instrução é decodificada. O formato das instruções MIPS permite que a leitura e a decodificação ocorram simultaneamente.
3. Executar a operação ou calcular um endereço.
4. Acessar um operando na memória de dados.
5. Escrever o resultado em um registrador.

Logo, o pipeline MIPS que exploramos neste capítulo possui cinco estágios. O exemplo a seguir mostra que a técnica de pipelining agiliza a execução da instrução, assim como agiliza a lavagem de roupas.

DESEMPENHO DE CICLO ÚNICO VERSUS DESEMPENHO COM PIPELINE

Para tornar esta discussão concreta, vamos criar um pipeline. Neste exemplo, e no restante deste capítulo, vamos limitar nossa atenção a oito instruções: load word (*lw*), store word (*sw*), add (*add*), subtract (*sub*), and (*and*), or (*or*), set-less-than (*slt*) e branch-on-equal (*beq*).

Compare o tempo médio entre as instruções de uma implementação em ciclo único, em que todas as instruções levam 1 ciclo de clock, com uma implementação com pipeline. Os tempos de operação para as principais unidades funcionais neste exemplo são de 200ps para acesso à memória, 200ps para operação com ALU e 100ps para leitura ou escrita de registradores. Como dissemos no Capítulo 5, no modelo de ciclo único, cada instrução leva exatamente 1 ciclo de clock, de modo que o ciclo precisa ser esticado para acomodar a instrução mais lenta.

EXEMPLO

RESPOSTA

A Figura 6.2 mostra o tempo exigido para cada uma das oito instruções. O projeto de ciclo único precisa contemplar a instrução mais lenta – na Figura 6.2, ela é 1w – de modo que o tempo exigido para cada instrução é 800ps. Assim como na Figura 6.1, a Figura 6.3 compara a execução sem pipeline e com pipeline de três instruções load word. Desse modo, o tempo entre a primeira e a quarta instrução no projeto sem pipeline é 3 × 800ns, ou 2.400ps.

Todos os estágios do pipeline utilizam um único ciclo de clock, de modo que o ciclo de clock precisa ser grande o suficiente para acomodar a operação mais lenta. Assim como o projeto de ciclo único de clock precisa levar o tempo do ciclo de clock no pior caso, de 800ps, embora algumas instruções possam ser tão rápidas quanto 500ps, o ciclo de clock da execução com pipeline precisa ter o ciclo de clock no pior caso de 200ps, embora alguns estágios levem apenas 100ps. O uso de pipeline ainda oferece uma melhoria de desempenho de quatro vezes: o tempo entre a primeira e a quarta instruções é de 3 × 200ps, ou 600ps.

Agora, podemos converter a discussão sobre ganho de velocidade com a técnica de pipelining em uma fórmula. Se os estágios forem perfeitamente balanceados, então o tempo entre as instruções no processador com pipeline – assumindo condições ideais – é igual a

$$\text{Tempo entre instruções}_{\text{com pipeline}} = \frac{\text{Tempo entre instruções}_{\text{sem pipeline}}}{\text{Número de estágios do pipe}}$$

Sob condições ideais e com uma grande quantidade de instruções, o ganho de velocidade com a técnica de pipelining é aproximadamente igual ao número de estágios do pipe; um pipeline de cinco estágios é quase cinco vezes mais rápido.

A fórmula sugere que um pipeline de cinco estágios deve oferecer uma melhoria de quase cinco vezes sobre o tempo sem pipeline de 800ps, ou um ciclo de clock de 160ps. Entretanto, o exemplo mostra que os estágios podem ser mal平衡ados. Além disso, a técnica de pipelining envolve algum overhead, cuja origem se tornará mais clara adiante. Assim, o tempo por instrução no processador com pipeline será superior ao mínimo possível, e o ganho de velocidade será menor que o número de estágios do pipeline.

Além do mais, até mesmo nossa afirmação de uma melhoria de quatro vezes para nosso exemplo não está refletida no tempo de execução total para as três instruções: são 1.400ps *versus* 2.400ps. Naturalmente, isso acontece porque o número de instruções não é grande. O que aconteceria se aumentássemos o número de instruções? Poderíamos estender os valores anteriores para 1.000.003 instruções. Acrescentaríamos 1.000.000 instruções no exemplo com pipeline; cada instrução acrescenta 200ps ao tempo de execução total. O tempo de execução total seria $1.000.000 \times 200\text{ps} + 1.400\text{ps}$, ou 200.001.400ps. No exemplo sem pipeline, acrescentaríamos 1.000.000 instruções, cada uma exigindo 800ps, de modo que o tempo de execução total seria $1.000.000 \times 800\text{ps} + 2.400\text{ps}$, ou 800.002.400ps. Sob essas condições ideais, a razão entre os tempos de execução total para os programas reais nos processadores sem pipeline e com pipeline é próximo da razão de tempos entre as instruções:

Classe de Instrução	Busca de Instruções	Leitura de registradores	Operação da ALU	Acesso a dados	Escrita de registradores	Tempo total
Load word(1w)	200ps	100ps	200ps	200ps	100ps	800ps
Store word (sw)	200ps	100ps	200ps	200ps		700ps
Formato R (add, sub, and, or, srl)	200ps	100ps	200ps		100ps	600ps
Branch (beq)	200ps	100ps	200ps			500ps

FIGURA 6.2 Tempo total para cada instrução calculada a partir do tempo para cada componente.

Esse cálculo considera que os multiplexadores, unidade de controle, acessos ao PC e unidade de extensão de sinal não possuem atraso.

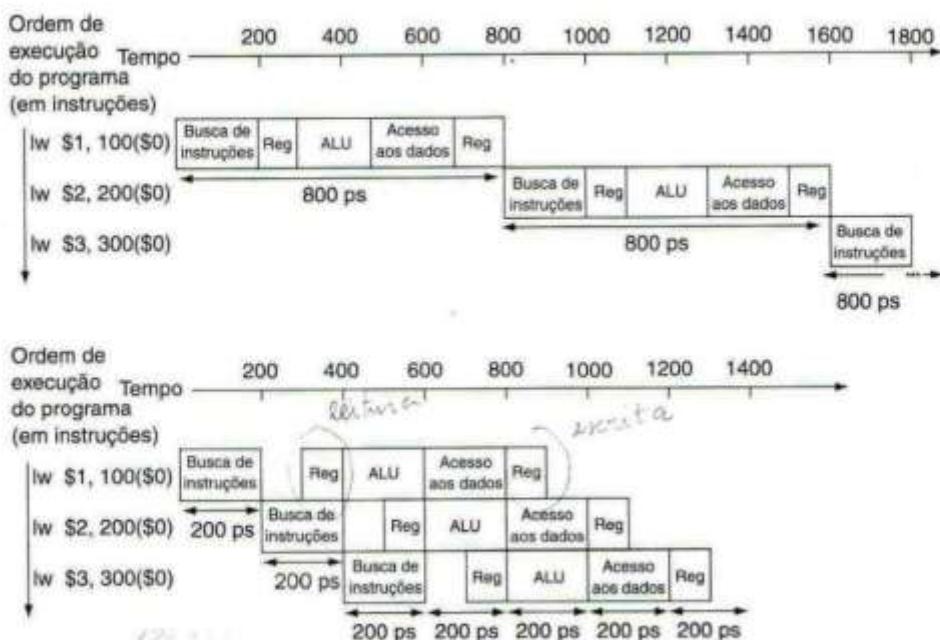


FIGURA 6.3 Acima, execução em ciclo único, sem pipeline, versus execução com pipeline abaixo.

Ambas utilizam os mesmos componentes de hardware, cujo tempo está listado na Figura 6.2. Neste caso, vemos um ganho de velocidade de 4 vezes no tempo médio entre as instruções, de 800ps para 200ps. Compare com a Figura 6.1. Para a lavanderia, consideramos que todos os estágios eram iguais. Se a secadora fosse mais lenta, então o estágio da secadora definiria o tempo do estágio. Os tempos de estágio do pipeline dos computadores são limitados pelo recurso mais lento, seja a operação da ALU ou o acesso à memória. Consideramos que a escrita no banco de registradores ocorre na primeira metade do ciclo de clock e a leitura do banco de registradores ocorre na segunda metade. Usamos essa suposição por todo este capítulo.

$$\frac{800.002.400\text{ps}}{200.001.400\text{ps}} \approx 4.00 \approx \frac{800\text{ps}}{200\text{ps}}$$

A técnica de pipelining melhora o desempenho *aumentando a vazão de instruções, em vez de diminuir o tempo de execução de uma instrução individual*, mas a vazão de instruções não é a medida importante, pois os programas reais executam bilhões de instruções.

Projetando conjuntos de instruções para pipelining

Mesmo com essa explicação simples sobre pipelining, podemos entender melhor o projeto do conjunto de instruções MIPS, projetado para execução com pipeline.

Primeiro, todas as instruções MIPS têm o mesmo tamanho. Essa restrição torna muito mais fácil buscar instruções no primeiro estágio do pipeline e decodificá-las no segundo estágio. Em um conjunto de instruções como o IA-32, no qual as instruções variam de 1 byte a 17 bytes, a técnica de pipelining é muito mais desafiadora. Como vimos no Capítulo 5, todas as implementações recentes da arquitetura IA-32 na realidade traduzem instruções IA-32 em microoperações simples, que se parecem com instruções MIPS. Conforme veremos na Seção 6.10, na verdade, o Pentium 4 usa um pipeline de microoperações, no lugar das instruções IA-32 nativas!

Em segundo lugar, o MIPS tem apenas alguns poucos formatos de instrução, com os campos de registrador de origem localizados no mesmo lugar em cada instrução. Essa simetria significa que o segundo estágio pode começar a ler o banco de registradores ao mesmo tempo em que o hardware está determinando que tipo de instrução foi lida. Se os formatos de instrução do MIPS não fossem si-

métricos, precisaríamos dividir o estágio 2, resultando em seis estágios de pipeline. Logo veremos a desvantagem dos pipelines mais longos.

Em terceiro lugar, os operandos em memória só aparecem em loads ou stores no MIPS. Essa restrição significa que podemos usar o estágio de execução para calcular o endereço de memória e depois acessar a memória no estágio seguinte. Se pudéssemos operar sobre os operandos na memória, como na arquitetura IA-32, os estágios 3 e 4 se expandiriam para estágio de endereço, estágio de memória e, em seguida, estágio de execução.

Em quarto lugar, conforme discutimos no Capítulo 2, os operandos precisam estar alinhados na memória. Logo, não precisamos nos preocupar com uma única instrução de transferência de dados exigindo dois acessos à memória de dados; os dados solicitados podem ser transferidos entre o processador e a memória em um único estágio do pipeline.

Pipeline hazards

Existem situações em pipelining em que a próxima instrução não pode ser executada no ciclo de clock seguinte. Esses eventos são chamados *hazards*, e existem três tipos diferentes.

Hazards estruturais

hazard estrutural Uma ocorrência em que uma instrução planejada não pode ser executada no ciclo de clock correto, pois o hardware não admite a combinação de instruções definidas para executar em determinado ciclo de clock.

O primeiro hazard é chamado **hazard estrutural**. Ele significa que o hardware não pode admitir a combinação de instruções que queremos executar no mesmo ciclo de clock. Um hazard estrutural na lavanderia aconteceria se usássemos uma combinação lavadora-secadora no lugar de lavadora e secadora separadas, ou se nosso colega estivesse ocupado com outra coisa e não pudesse guardar as roupas. Nossa pipeline, cuidadosamente programado, fracassaria.

Como dissemos, o conjunto de instruções MIPS foi projetado para ser executado em um pipeline, tornando muito fácil para os projetistas evitar hazards estruturais quando projetaram o pipeline. Contudo, suponha que tivéssemos uma única memória, em vez de duas. Se o pipeline da Figura 6.3 tivesse uma quarta instrução, veríamos que, no mesmo ciclo de clock em que a primeira instrução está acessando dados da memória, a quarta instrução está buscando uma instrução dessa mesma memória. Sem duas memórias, nossa pipeline poderia ter um hazard estrutural.

Hazards de dados

hazard de dados Também chamado hazard de dados do pipeline. Uma ocorrência em que uma instrução planejada não pode ser executada no ciclo de clock correto porque os dados necessários para executar a instrução ainda não estão disponíveis.

Os **hazards de dados** ocorrem quando o pipeline precisa ser interrompido porque uma etapa precisa esperar até que outra seja concluída. Suponha que você tenha encontrado uma meia na estação de passar para a qual não existe um par. Uma estratégia possível é correr até o seu quarto e procurar em sua gaveta para ver se consegue encontrar o par. Obviamente, enquanto você está procurando, as roupas que ficaram secas e estão prontas para serem passadas, e aquelas que acabaram de ser lavadas e estão prontas para serem secadas deverão esperar.

Em um pipeline de computador, os hazards de dados surgem quando uma instrução depende de uma anterior que ainda está no pipeline (um relacionamento que não existe realmente quando se lava roupas). Por exemplo, suponha que tenhamos uma instrução add seguida imediatamente por uma instrução subtract que usa a soma (\$\$0):

```
add    $$0, $t0, $t1
sub    $$t2, $$0, $t3
```

Sem intervenção, um hazard de dados poderia prejudicar o pipeline severamente. A instrução add não escreve seu resultado até o quinto estágio, significando que teríamos de acrescentar três bolhas ao pipeline.

Embora pudéssemos contar com compiladores para remover todos esses hazards, os resultados não seriam satisfatórios. Essas dependências acontecem com muita freqüência, e o atraso simplesmente é muito longo para se esperar que o compilador nos tire desse dilema.

A solução principal é baseada na observação de que não precisamos esperar que a instrução termine antes de tentar resolver o hazard de dados. Para a seqüência de código anterior, assim que a ALU cria a soma para o add, podemos fornecê-la como uma entrada para a subtração. O acréscimo de hardware extra para ter o item que falta antes do previsto, diretamente dos recursos internos, é chamado de **forwarding** ou **bypassing**.

forwarding Também chamado **bypassing**. Um método para resolver um hazard de dados utilizando o elemento de dados que falta a partir de buffers internos, em vez de esperar que chegue nos registradores visíveis ao programador ou na memória.

FORWARDING COM DUAS INSTRUÇÕES

Para as duas instruções anteriores, mostre quais estágios do pipeline estariam conectados pelo forwarding. Use o desenho da Figura 6.4 para representar o caminho de dados durante os cinco estágios do pipeline. Alinhe a cópia do caminho de dados para cada instrução, semelhante ao pipeline da lavanderia, na Figura 6.1.

A Figura 6.5 mostra a conexão para o forwarding do valor em \$s0 após o estágio de execução da instrução add como entrada para o estágio de execução da instrução sub.

EXEMPLO

RESPOSTA

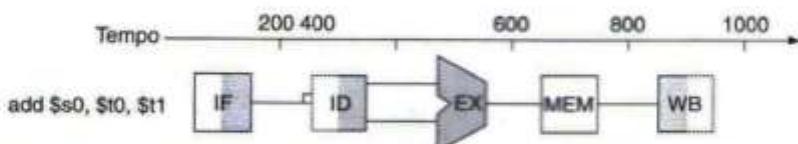


FIGURA 6.4 Representação gráfica do pipeline de instruções, semelhante ao objetivo do pipeline da lavanderia na Figura 6.1. Aqui, usamos símbolos representando os recursos físicos com as abreviações para os estágios do pipeline usados no decorrer do capítulo. Os símbolos para os cinco estágios são: *IF* para o estágio de busca de instruções (Instruction Fetch), com a caixa representando a memória de instruções; *ID* para o estágio de decodificação de instruções (Instruction Decode)/leitura do banco de registradores, com o desenho mostrando o banco de registradores sendo lido; *EX* para o estágio de execução, com o desenho representando a ALU; *MEM* para o estágio de acesso à memória, com a caixa representando a memória de dados; e *WB* para o estágio de escrita do resultado (Write Back), com o desenho mostrando o banco de registradores sendo escrito. O sombreado indica que o elemento é usado pela instrução. Logo, *MEM* possui um fundo branco porque *add* não acessa a memória de dados. O sombreado na metade direita do banco de registradores ou na memória significa que o elemento é lido nesse estágio, e o sombreado na metade esquerda significa que ele é escrito nesse estágio. Logo, a metade direita de *ID* é sombreada no segundo estágio porque o banco de registradores é lido, e a metade esquerda de *WB* é sombreada no quinto estágio porque o banco de registradores é escrito.

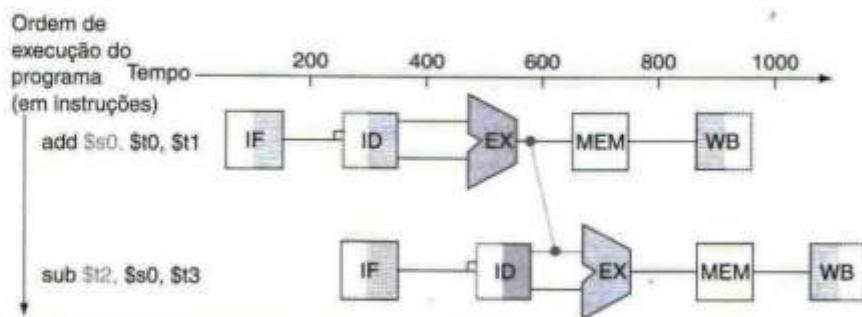


FIGURA 6.5 Representação gráfica do forwarding. A conexão mostra o caminho do forwarding desde a saída do estágio EX de add até a entrada do estágio EX para sub, substituindo o valor do registrador \$s0 lido no segundo estágio de sub.

Nessa representação gráfica dos eventos, os caminhos de forwarding só são válidos se o estágio de destino estiver mais adiante no tempo do que o estágio de origem. Por exemplo, não pode haver um caminho de forwarding válido da saída do estágio de acesso à memória na primeira instrução para a entrada do estágio de execução da instrução seguinte, pois isso significaria voltar no tempo.

hazard de dados no uso de load Uma forma específica de hazard de dados em que os dados solicitados por uma instrução load ainda não estão disponíveis quando requisitados.

pipeline stall Também chamado bolha. Um stall iniciado a fim de resolver um hazard.

O forwarding funciona muito bem e é descrito com detalhes na Seção 6.4. Entretanto, ele não pode impedir todos os stalls do pipeline. Por exemplo, suponha que a primeira instrução fosse um load de \$s0 em vez de um add. Como podemos imaginar examinando a Figura 6.5, os dados desejados só estariam disponíveis *depois* do quarto estágio da primeira instrução na dependência, que é muito tarde para a *entrada* do terceiro estágio da sub. Logo, até mesmo com o forwarding, teríamos de atrasar um estágio para um **hazard de dados no uso de load**, como mostra a Figura 6.6. Essa figura mostra um conceito importante de pipeline, conhecido oficialmente como **pipeline stall**, mas normalmente recebendo o apelido de *bolha*. Veremos os stalls em outros lugares do pipeline. A Seção 6.5 mostra como podemos tratar de casos assim, usando a detecção de hardware e stalls ou software que trata o atraso do load como um atraso de desvio.

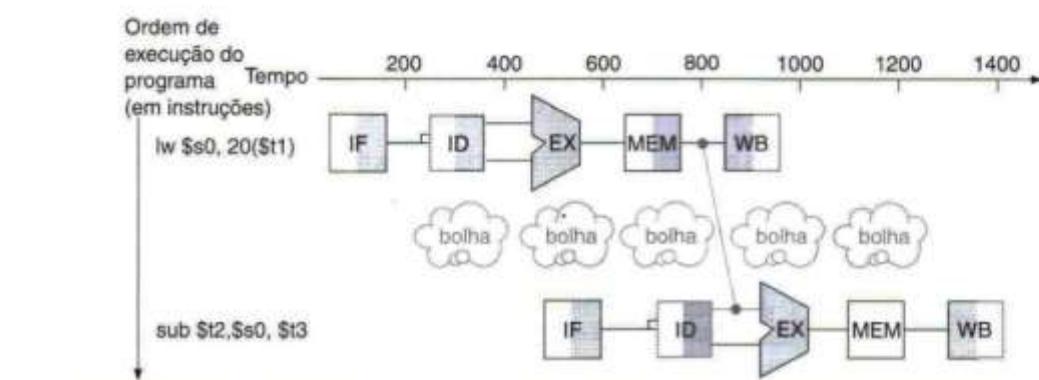


FIGURA 6.6 Precisamos de um stall até mesmo com forwarding quando uma instrução do formato R após um load tenta usar os dados. Sem o stall, o caminho da saída do estágio de acesso à memória para a entrada do estágio de execução estaria ao contrário no tempo, o que é impossível. Essa figura, na realidade, é uma simplificação, pois não podemos saber, antes que a instrução de subtração seja lida e decodificada, se um stall será necessário. A Seção 6.5 mostra os detalhes do que realmente acontece no caso de um hazard.

REORDENANDO O CÓDIGO PARA EVITAR PIPELINE STALLS

EXEMPLO

Considere o seguinte segmento de código em C:

$$\begin{aligned} A &= B + E; \\ C &= B + F; \end{aligned}$$

Aqui está o código MIPS gerado para esse segmento, supondo que todas as variáveis estejam na memória e sejam endereçáveis como offsets a partir de \$t0:

```

lw      $t1, 0($t0)
lw      $t2, 4($t0) ×
add   $t3, $t1,$t2
sw      $t3, 12($t0)
lw      $t4, 8($t0)
add   $t5, $t1,$t4
sw      $t5, 16($t0)

```

Encontre os hazards no segmento de código a seguir e reordene as instruções para evitar quaisquer pipeline stalls.

As duas instruções add possuem um hazard, devido à respectiva dependência da instrução lw imediatamente anterior. Observe que o bypassing elimina vários outros hazards em potencial, incluindo a dependência do primeiro add no primeiro lw e quaisquer hazards para instruções store. Subir a terceira instrução lw elimina os dois hazards:

```
lw    $t1, 0($t0)
lw    $t2, 4($t1)
lw    $t4, 8($t1)
add  $t3, $t1,$t2
sw    $t3, 12($t0)
add  $t5, $t1,$t4
sw    $t5, 16($t0)
```

Em um processador com pipeline com forwarding, a seqüência reordenada será completada em dois ciclos a menos do que a versão original.

RESPOSTA

O forwarding leva a outro detalhe da arquitetura MIPS, além dos quatro mencionados nas páginas 281-282. Cada instrução MIPS escreve no máximo um resultado e faz isso quase no final do pipeline. O forwarding é mais difícil se houver vários resultados para encaminhar por instrução, ou se precisarem escrever um resultado mais cedo na execução da instrução.

Detalhamento: o nome "forwarding" vem da idéia de que o resultado é passado adiante a partir de uma instrução anterior para uma instrução posterior. "Bypassing" vem de passar o resultado pelo banco de registradores à unidade desejada.

Hazards de controle

O terceiro tipo de hazard é chamado **hazard de controle**, vindo da necessidade de tomar uma decisão com base nos resultados de uma instrução enquanto outras estão sendo executadas.

Suponha que nosso pessoal da lavanderia receba a tarefa feliz de limpar os uniformes de um time de futebol. Como a roupa é muito suja, temos de determinar se o detergente e a temperatura da água que selecionamos são fortes o suficiente para limpar os uniformes, mas não tão forte para desgastá-los antes do tempo. Em nosso pipeline de lavanderia, temos de esperar até o segundo estágio para examinar o uniforme seco para ver se precisamos ou não mudar as opções da lavadora. O que fazer?

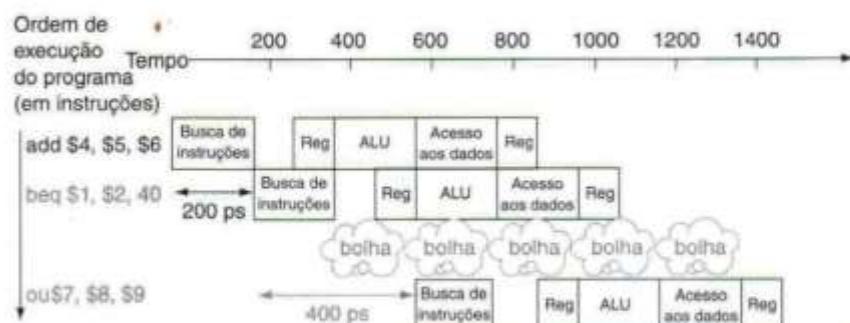
Aqui está a primeira das duas soluções para controlar os hazards na lavanderia e seu equivalente nos computadores.

hazard de controle
Também chamado **hazard de desvio**. Um acontecimento em que a instrução apropriada não pode ser executada no seu devido ciclo de clock porque a instrução buscada não é aquela necessária; ou seja, o fluxo de endereços de instrução não é o que o pipeline esperava.

Stall: Basta operar seqüencialmente até que o primeiro lote esteja seco e depois repetir até você ter a fórmula correta. Essa opção conservadora certamente funciona, mas é lenta.

A tarefa de decisão equivalente em um computador é a instrução de desvio. Observe que temos de começar a buscar a instrução após o desvio no próximo ciclo de clock. Contudo, o pipeline possivelmente não saberá qual deve ser a próxima instrução, pois ele só recebeu da memória a instrução de desvio! Assim como na lavanderia, uma solução possível é ocasionar um stall no pipeline imediatamente após buscarmos um desvio, esperando até que o pipeline determine o resultado do desvio para saber de que endereço apanhar a próxima instrução.

Vamos supor que colocamos hardware extra suficiente de modo que possamos testar registradores, calcular o endereço de desvio e atualizar o PC durante o segundo estágio do pipeline (ver Seção 6.6 para obter mais detalhes). Até mesmo com esse hardware extra, o pipeline envolvendo desvios condicionais se pareceria com a Figura 6.7. A instrução lw, executada se o desvio não for tomado, fica em stall durante um ciclo de clock extra de 200ps antes de iniciar.

**FIGURA 6.7 Pipeline mostrando o stall em cada desvio condicional como solução para controlar os hazards.**

Existe um stall de um estágio no pipeline, ou bolha, após o desvio. Na realidade, o processo de criação de um stall é ligeiramente mais complicado, conforme veremos na Seção 6.6. No entanto, o efeito sobre o desempenho é o mesmo que ocorreria se uma bolha fosse inserida.

DESEMPENHO DO "STALL NO DESVIO"

EXEMPLO

Estime o impacto nos ciclos de clock por instrução (CPI) do stall nos desvios. Suponha que todas as outras instruções tenham um CPI de 1.

RESPOSTA

A Figura 3.26, mostra que os desvios são 13% das instruções executadas no SPECint2000. Como outras instruções possuem um CPI de 1 e os desvios tomaram um ciclo de clock extra para o stall, então veríamos um CPI de 1,13 e, portanto, um stall de 1,13 em relação ao caso ideal. Observe que isso inclui apenas branches e que os jumps também poderiam ocasionar um stall.

Se não pudermos resolver o desvio no segundo estágio, como normalmente acontece para pipelines maiores, então veríamos um atraso ainda maior se ocorresse um stall nos desvios. O custo dessa opção é muito alto para a maioria dos computadores utilizar, e isso motiva uma segunda solução para o hazard de controle:

Prever: se você estiver certo de que tem a fórmula correta para lavar os uniformes, então basta prever que ela funcionará e lavar a segunda remessa enquanto espera que a primeira seque. Essa opção não atrasa o pipeline quando você estiver correto. Entretanto, quando estiver errado, você terá de refazer a remessa que foi lavada enquanto pensa na decisão.

desvio não tomado

Um que segue para a instrução sucessiva. Um desvio tomado é aquele que causa a transferência da execução para o destino do desvio.

previsão de desvio Um método de resolver um hazard de desvio que considera um determinado resultado para o desvio e prossegue a partir dessa suposição, em vez de esperar para verificar o resultado real.

Os computadores realmente utilizam a *previsão* para tratar dos desvios. Uma técnica simples é sempre prever que os **desvios não serão tomados**. Quando você estiver certo, o pipeline prosseguirá a toda velocidade. Somente quando os desvios são tomados é que o pipeline sofre um stall. A Figura 6.8 mostra um exemplo assim.

Uma versão mais sofisticada de **previsão de desvio** teria alguns desvios previstos como tomados e alguns como não tomados. Em nossa analogia, os uniformes escuros ou de casa poderiam usar uma fórmula, enquanto os uniformes claros ou de sair poderiam usar outra. Como um exemplo de computador, no final dos loops existem desvios que voltam para o inicio do loop. Como provavelmente serão tomados e desviam para trás, sempre poderíamos prever como tomados os desvios para um endereço anterior.

Essas técnicas rígidas para o desvio contam com o comportamento estereotipado e não são responsáveis pela individualidade de uma instrução de desvio específica. Previsores de hardware *dinâmicos*, ao contrário, fazem suas escolhas dependendo do comportamento de cada desvio, e podem mudar as previsões para um desvio durante a vida de um programa. Segundo nossa analogia, na previsão dinâmica, uma pessoa veria como o uniforme estava sujo e escolheria a fórmula, ajustando a próxima escolha dependendo do sucesso das escolhas recentes. Uma técnica popular para a previsão

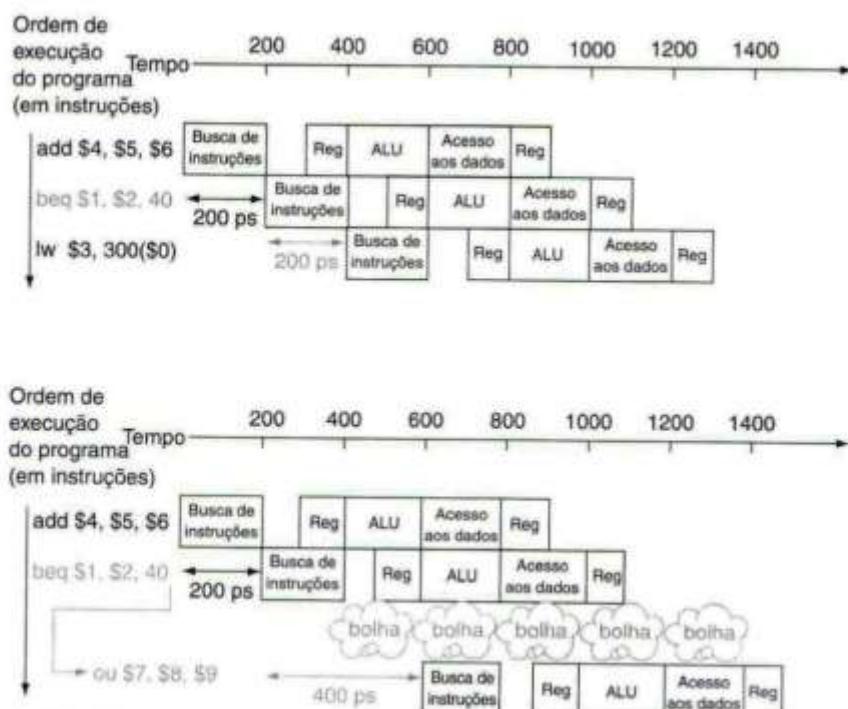


FIGURA 6.8 Prevendo que os desvios não serão tomados como solução para o hazard de controle. O desenho superior mostra o pipeline quando o desvio não é tomado. O desenho inferior mostra o pipeline quando o desvio é tomado. Conforme observamos na Figura 6.7, a inserção de uma bolha nesse padrão simplifica o que realmente acontece, pelo menos durante o primeiro ciclo de clock imediatamente após o desvio. A Seção 6.6 esclarecerá os detalhes.

dinâmica de desvios é manter um histórico de cada desvio como tomado ou não tomado, e depois usar o comportamento passado recente para prever o futuro. Como veremos mais adiante, a quantidade e o tipo de histórico mantido têm se tornado extensos, resultando em previsores de desvio dinâmicos que podem prever os desvios corretamente, com uma precisão superior a 90% (ver Seção 6.6). Quando a escolha estiver errada, o controle do pipeline terá de garantir que as instruções após o desvio errado não tenham efeito, devendo reiniciar o pipeline a partir do endereço de desvio apropriado. Em nossa analogia de lavanderia, temos de deixar de aceitar novas remessas para poder reiniciar a remessa prevista incorretamente.

Como no caso de todas as outras soluções para controlar hazards, pipelines mais longos aumentam o problema, neste caso, aumentando o custo do erro de previsão. As soluções para controlar os hazards são descritas com mais detalhes na Seção 6.6.

Detalhamento: existe uma terceira técnica para o hazard de controle, chamada *decisão adiada*. Em nossa analogia, sempre que você tiver de tomar uma decisão sobre a lavanderia, basta colocar uma remessa de roupas que não sejam de futebol na lavadora, enquanto espera que os uniformes de futebol sequem. Desde que você tenha roupas sujas suficientes, que não sejam afetadas pelo teste, essa solução funcionará bem.

Chamado de *delayed branch* (desvio adiado) nos computadores, essa é a solução realmente usada pela arquitetura MIPS. O delayed branch sempre executa a próxima instrução seqüencial, com o desvio ocorrendo após esse atraso de uma instrução. Isso fica escondido do programador assembly do MIPS, pois o montador pode arrumar as instruções automaticamente para conseguir o comportamento de desvio desejado pelo programador. O software MIPS colocará uma instrução imediatamente após a instrução de delayed branch, que não é afetada pelo desvio, e um desvio tomado muda o endereço da instrução que vem após essa instrução segura. Em nosso exemplo, a instrução add antes do desvio na Figura 6.7 não afeta o desvio, e pode ser movida para depois do desvio, para esconder totalmente o atraso no desvio. Como os delayed branches são úteis quando os desvios são curtos, nenhum processador usa um delayed branch de mais de 1 ciclo. Para atrasos em desvios maiores, a previsão de desvio baseada em hardware normalmente é usada.

Resumo da visão geral de pipelining

Pipelining é uma técnica que explora o paralelismo entre as instruções em um fluxo de instruções seqüenciais. Ela tem a vantagem substancial de que, diferente de algumas técnicas para ganhar velocidade (ver Capítulo 9), ela é fundamentalmente invisível ao programador.

Nas próximas seções deste capítulo, abordamos o conceito de pipelining usando o subconjunto de instruções MIPS `lw`, `sw`, `add`, `sub`, `and`, `or`, `slt` e `beq` (o mesmo do Capítulo 5) e uma versão simplificada do seu pipeline. Depois, examinamos os problemas que a técnica de pipelining introduz e o desempenho alcançável sob situações típicas.

Se você quiser saber mais sobre o software e as implicações de desempenho da técnica de pipelining, agora terá base suficiente para pular para a Seção 6.9. A Seção 6.9 introduz conceitos avançados de pipelining, como o escalonamento superescalar e dinâmico, e a Seção 6.10 examina o pipeline do microprocessador Pentium 4.

Como alternativa, se você estiver interessado em entender como a técnica de pipelining é implementada e os desafios de lidar com hazards, poderá prosseguir para examinar o projeto de um caminho de dados com pipeline, explicado na Seção 6.2, e o controle básico, explicado na Seção 6.3. Depois, você poderá usar esse conhecimento para explorar a implementação do forwarding na Seção 6.4, e a implementação de stalls na Seção 6.5. Você poderá, então, ler a Seção 6.6 para aprender mais sobre soluções para hazards de desvio, e depois ver como as exceções são tratadas, na Seção 6.8.

Colocando em perspectiva

latência (pipeline) O número de estágios em um pipeline ou o número de estágios entre duas instruções durante a execução.

A técnica de pipelining aumenta o número de instruções em execução simultânea e a velocidade em que as instruções são iniciadas e concluídas. A técnica de pipelining não reduz o tempo gasto para completar uma instrução individual, também chamado de **latência**. Por exemplo, o pipeline de cinco estágios ainda usa 5 ciclos de clock para completar a instrução. Nos termos usados no Capítulo 4, a técnica de pipelining melhora a *vazão* de instruções, e não o *tempo de execução* ou *latência* das instruções individualmente.

Os conjuntos de instruções podem simplificar ou dificultar a vida dos projetistas do pipeline, que já precisam enfrentar hazards estruturais, de controle e de dados. A previsão de desvio, o forwarding e os stalls ajudam a tornar um computador rápido enquanto ainda gera as respostas certas.

Entendendo o desempenho dos programas

Fora do sistema de memória, a operação eficaz do pipeline normalmente é o fator mais importante para determinar o CPI do processador e, portanto, seu desempenho. Conforme veremos na Seção 6.9, compreender o desempenho de um processador moderno com múltiplos problemas é algo complexo e exige a compreensão de mais do que apenas as questões que surgem em um processador com pipeline simples. Apesar disso, os hazards estruturais, de dados e de controle continuam sendo importantes em pipelines simples e mais sofisticados.

Para pipelines modernos, os hazards estruturais costumam girar em torno da unidade de ponto flutuante, que pode não ser totalmente implementada com pipeline, enquanto os hazards de controle costumam ser um problema maior nos programas de inteiros, que costumam ter maiores freqüências de desvio, além de desvios menos previsíveis. Os hazards de dados podem ser gargalos de desempenho em programas de inteiros e de ponto flutuante. Em geral, é mais fácil lidar com hazards de dados em programas de ponto flutuante porque a menor freqüência de desvios e os padrões de acesso mais regulares permitem que o compilador tente escalonar instruções para evitar os hazards. É mais difícil realizar essas otimizações em programas de inteiros, que possuem acesso menos regular e envolvem um maior uso de ponteiros. Conforme veremos na Seção 6.9, existem técnicas de compilação e de hardware mais ambiciosas para reduzir as dependências de dados para o escalonamento.

Para cada sequência de código a seguir, indique se ela deverá sofrer stall, pode evitar stalls usando apenas forwarding, ou pode ser executada sem stall ou forwarding:

Verifique você mesmo

Seqüência 1	Seqüência 2	Seqüência 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

6.2 Um caminho de dados usando pipeline

A Figura 6.9 mostra o caminho de dados de ciclo único do Capítulo 5. A divisão de uma instrução em cinco estágios significa um pipeline de cinco estágios, que, por sua vez, significa que até cinco instruções estarão em execução durante qualquer ciclo de clock. Assim, temos de separar o caminho de dados em cinco partes, com cada parte possuindo um nome correspondente a um estágio da execução da instrução:

1. IF (Instruction Fetch): Busca de instruções
2. ID (Instruction Decode): Decodificação de instruções e leitura do banco de registradores
3. EX: Execução ou cálculo de endereço
4. MEM: Acesso à memória de dados
5. WB (Write Back): Escrita do resultado

Há menos
nissos do que
chega aos olhos.

Tallulah Bankhead,
observação para
Alexander
Wollcott, 1922

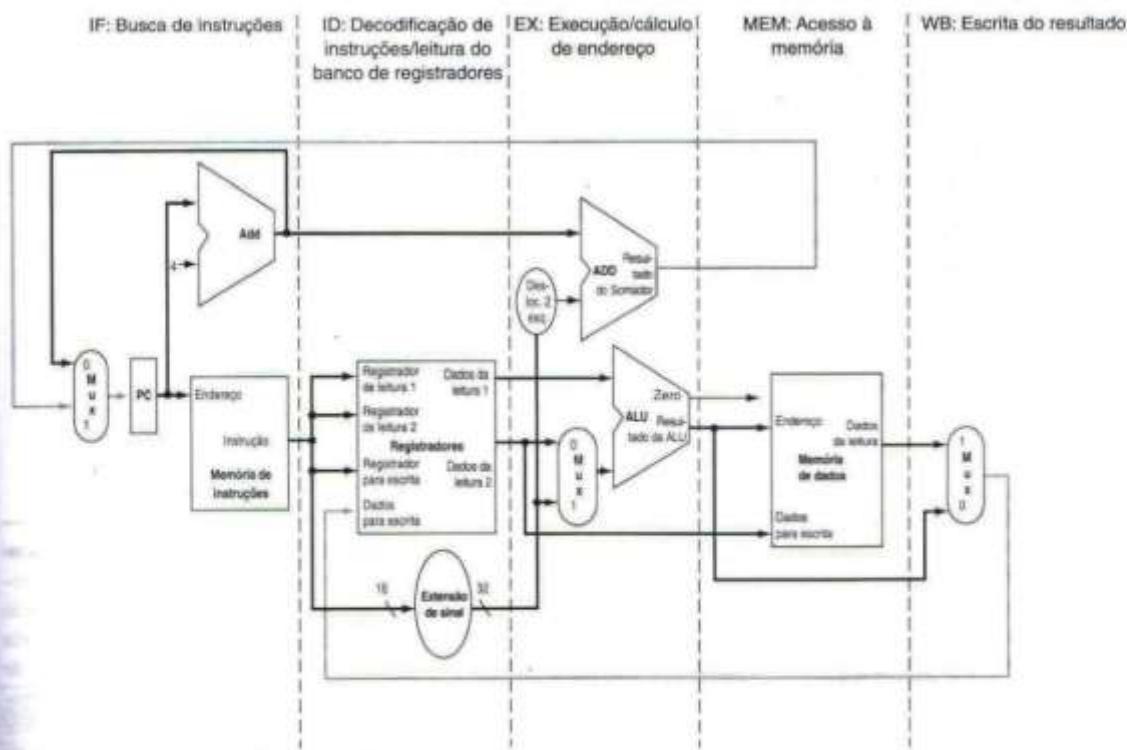


FIGURA 6.9 O caminho de dados de ciclo único do Capítulo 5 (semelhante à Figura 5.17). Cada etapa da instrução pode ser mapeada no caminho de dados da esquerda para a direita. As únicas exceções são a atualização do PC e a etapa de escrita do resultado, mostrada em cores, que envia o resultado da ALU ou os dados da memória para a esquerda, para serem escritos no banco de registradores. (Normalmente, usamos linhas coloridas para controle, mas são linhas de dados.)

Na Figura 6.9, esses cinco componentes correspondem aproximadamente ao modo como o caminho de dados é desenhado; as instruções e os dados em geral se movem da esquerda para a direita pelos cinco estágios enquanto completam a execução. Voltando à nossa analogia da lavanderia, as roupas ficam mais limpas, mais secas e mais organizadas à medida que prosseguem na fila, e nunca se movem para trás.

Entretanto, existem duas exceções para esse fluxo de informações da esquerda para a direita:

- O estágio de escrita do resultado, que coloca o resultado de volta no banco de registradores, no meio do caminho de dados
- A seleção do próximo valor do PC, escolhendo entre o PC incrementado e o endereço de desvio do estágio MEM

Os dados fluindo da direita para a esquerda não afetam a instrução atual; somente as instruções seguintes no pipeline são influenciadas por esses movimentos de dados reversos. Observe que a primeira seta da direita para a esquerda pode levar a hazards de dados e a segunda ocasiona hazards de controle.

Uma maneira de mostrar o que acontece na execução com pipeline é fingir que cada instrução tem seu próprio caminho de dados, e depois colocar esses caminhos de dados em uma linha de tempo para mostrar seu relacionamento. A Figura 6.10 mostra a execução das instruções na Figura 6.3, exibindo seus caminhos de dados privados em uma linha de tempo comum. Usamos uma versão estilizada do caminho de dados na Figura 6.9 para mostrar os relacionamentos na Figura 6.10.

A Figura 6.10 parece sugerir que três instruções precisam de três caminhos de dados. No Capítulo 5, acrescentamos registradores para manter dados de modo que partes do caminho de dados pudessem ser compartilhadas durante a execução da instrução; usamos a mesma técnica aqui para compartilhar os múltiplos caminhos de dados. Por exemplo, como mostra a Figura 6.10, a memória de ins-

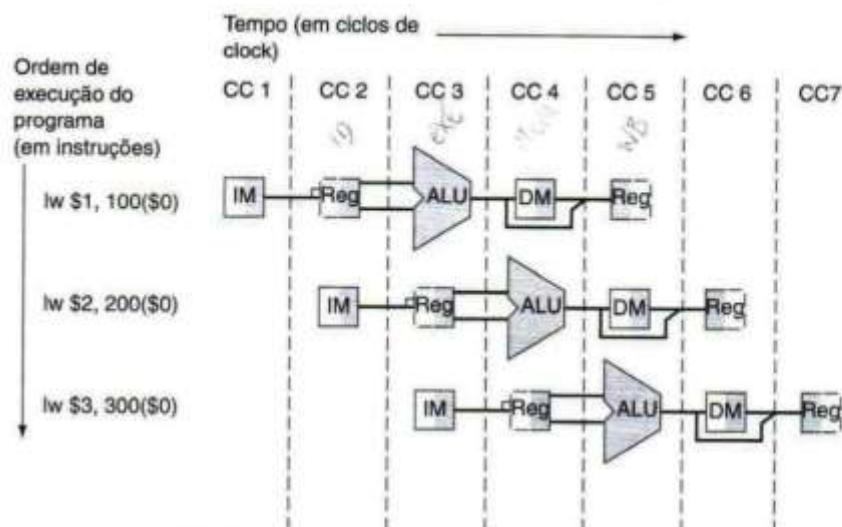


FIGURA 6.10 Instruções executadas usando o caminho de dados de ciclo único na Figura 6.9, assumindo a execução com pipeline. Semelhante às Figuras de 6.4 a 6.6, esta figura finge que cada instrução possui seu próprio caminho de dados e pinta cada parte de acordo com o uso. Ao contrário daquelas figuras, cada estágio é rotulado pelo recurso físico usado nesse estágio, correspondendo às partes do caminho de dados na Figura 6.9. IM representa a memória de instruções e o PC no estágio de busca da instrução, Reg significa banco de registradores e extensor de sinal no estágio de decodificação de instruções/leitura do banco de registradores (ID), e assim por diante. Para manter a ordem de tempo correta, esse caminho de dados estilizado divide o banco de registradores em duas partes lógicas: leitura de registradores durante a busca de registradores (ID) e registradores escritos durante a escrita do resultado (WB). Esse uso dual é representado pelo desenho da metade esquerda não sombreada do banco de registradores, usando linhas tracejadas no estágio ID, quando ele não estiver sendo escrito, e a metade direita não sombreada usando linhas tracejadas do estágio WB, quando não estiver sendo lido. Como antes, consideraremos que o banco de registradores é escrito na primeira metade do ciclo de clock e o banco de registradores é lido durante a segunda metade.

truções é usada durante apenas um dos cinco estágios de uma instrução, permitindo que seja compartilhada por outras instruções durante os outros quatro estágios.

Para reter o valor de uma instrução individual para seus outros quatro estágios, o valor lido da memória de instruções precisa ser salvo em um registrador. Argumentos semelhantes se aplicam a cada estágio do pipeline, de modo que precisamos colocar registradores sempre que existam linhas divisorias entre os estágios na Figura 6.9. Essa mudança é semelhante aos registradores acrescentados no Capítulo 5, quando passamos de um ciclo único para um caminho de dados de múltiplos ciclos. Retornando à nossa analogia da lavanderia, poderíamos ter um cesto entre cada par de estágios para conter as roupas para a próxima etapa.

A Figura 6.11 mostra o caminho de dados usando pipeline com os registradores do pipeline destacados. Todas as instruções avançam durante cada ciclo de clock de um registrador do pipeline para o seguinte. Os registradores recebem os nomes dos dois estágios separados por esse registrador. Por exemplo, o registrador do pipeline entre os estágios IF e ID é chamado de IF/ID.

Observe que não existe um registrador de pipeline no final do estágio de escrita do resultado (WB). Todas as instruções precisam atualizar algum estado no processador – o banco de registradores, memória ou o PC –, assim, um registrador de pipeline separado é redundante para o estado atualizado. Por exemplo, uma instrução load colocará seu resultado em 1 dos 32 registradores, e qualquer instrução posterior que precise desses dados simplesmente lerá o registrador apropriado.

Naturalmente, cada instrução atualiza o PC, seja incrementando-o ou atribuindo a ele o endereço de destino de um desvio. O PC pode ser considerado um registrador de pipeline: um que alimenta o estágio IF do pipeline. Contudo, diferente dos registradores de pipeline sombreados na Figura 6.11, o PC faz parte do estado arquitetônico visível; seu conteúdo precisa ser salvo quando ocorre uma exceção, enquanto o conteúdo dos registradores de pipeline pode ser descartado. Na analogia da lavanderia, você poderia pensar no PC como correspondendo ao cesto que mantém a remessa de roupas sujas antes da etapa de lavagem!

Para mostrar como funciona a técnica de pipelining, no decorrer deste capítulo, apresentamos sequências de figuras para demonstrar a operação com o tempo. Essas páginas extras parecem exigir

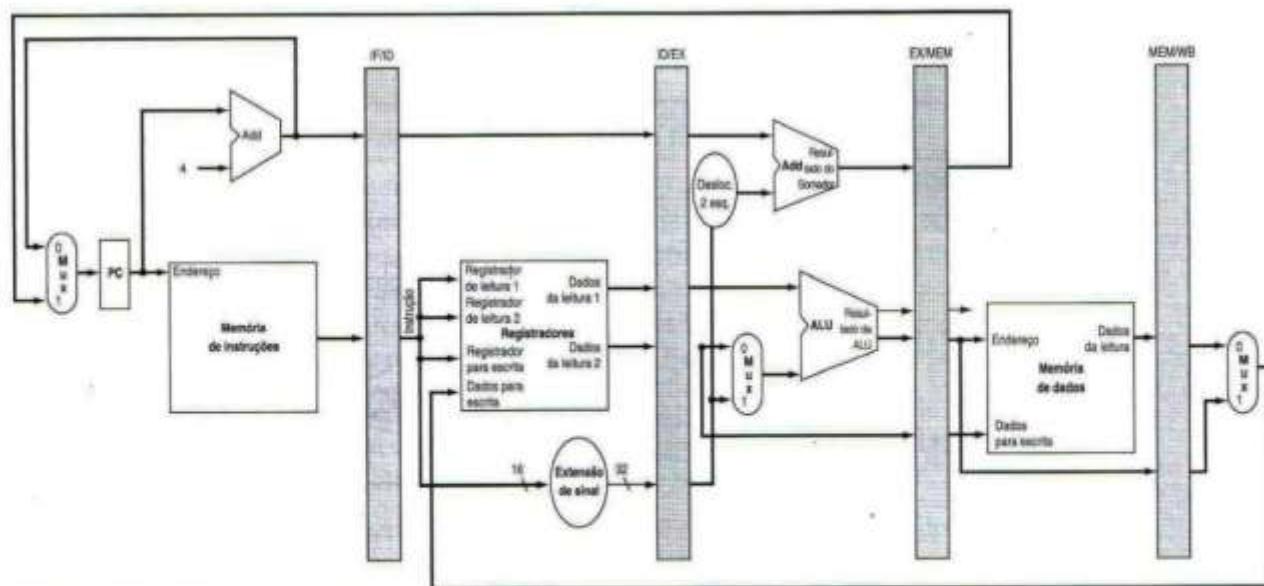


FIGURA 6.11 A versão com pipeline do caminho de dados na Figura 6.9. Os registradores do pipeline, em cinza, separam cada estágio do pipeline. Eles são rotulados pelos nomes dos estágios que separam; por exemplo, o primeiro é rotulado com IF/ID porque separa os estágios de busca de instruções e decodificação de instruções. Os registradores precisam ser grandes o suficiente para armazenar todos os dados correspondentes às linhas que passam por eles. Por exemplo, o registrador IF/ID precisa ter 64 bits de largura, pois precisa manter a instrução de 32 bits lida da memória e o endereço incrementado de 32 bits no PC. Vamos expandir esses registradores no decorrer deste capítulo, mas, por enquanto, os outros três registradores de pipeline contêm 128, 97 e 64 bits, respectivamente.

muito mais tempo para você entender. Mas não tema; as seqüências levam muito menos tempo do que parece, pois você pode compará-las para ver que mudanças ocorrem em cada ciclo do clock. As Seções 6.4 e 6.5 descrevem o que acontece quando existem hazards de dados entre instruções em um pipeline; ignore-as por enquanto.

As Figuras 6.12 a 6.14, nossa primeira seqüência, mostram as partes ativas do caminho de dados destacadas enquanto uma instrução de load passa pelos cinco estágios de execução do pipeline. Mostramos um load primeiro porque ele ativa todos os cinco estágios. Como nas Figuras de 6.4 a 6.11, destacamos a metade direita dos registradores ou memória quando estão sendo lidos e destacamos a metade esquerda quando estão sendo escritos. Mostramos a abreviação da instrução lw com o nome do estágio do pipeline que está ativo em cada figura. Os cinco estágios são os seguintes:

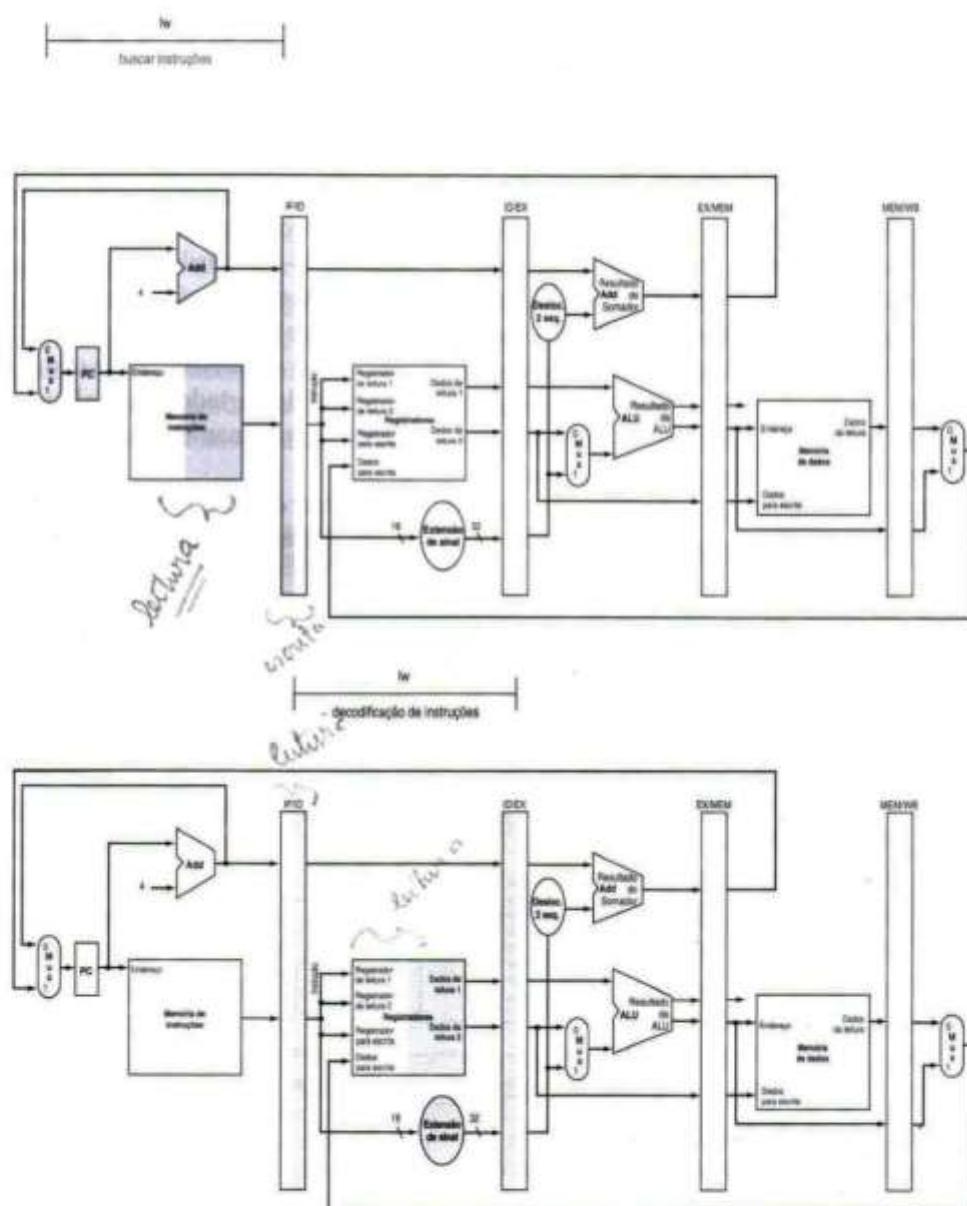


FIGURA 6.12 IF e ID: primeiro e segundo estágios do pipe de uma instrução, com as partes ativas do caminho de dados da Figura 6.11 em destaque. A convenção de destaque é a mesma utilizada na Figura 6.4. Como no Capítulo 5, não há confusão quando se lê e escreve nos registradores, pois o conteúdo só muda na transição do clock. Embora o load só precise do registrador de cima no estágio 2, o processador não sabe que instrução está sendo decodificada, de modo que estende o sinal da constante de 16 bits e lê os dois registradores para o registrador de pipeline ID/EX. Não precisamos de todos os três operandos, mas simplifica o controle manter todos os três.

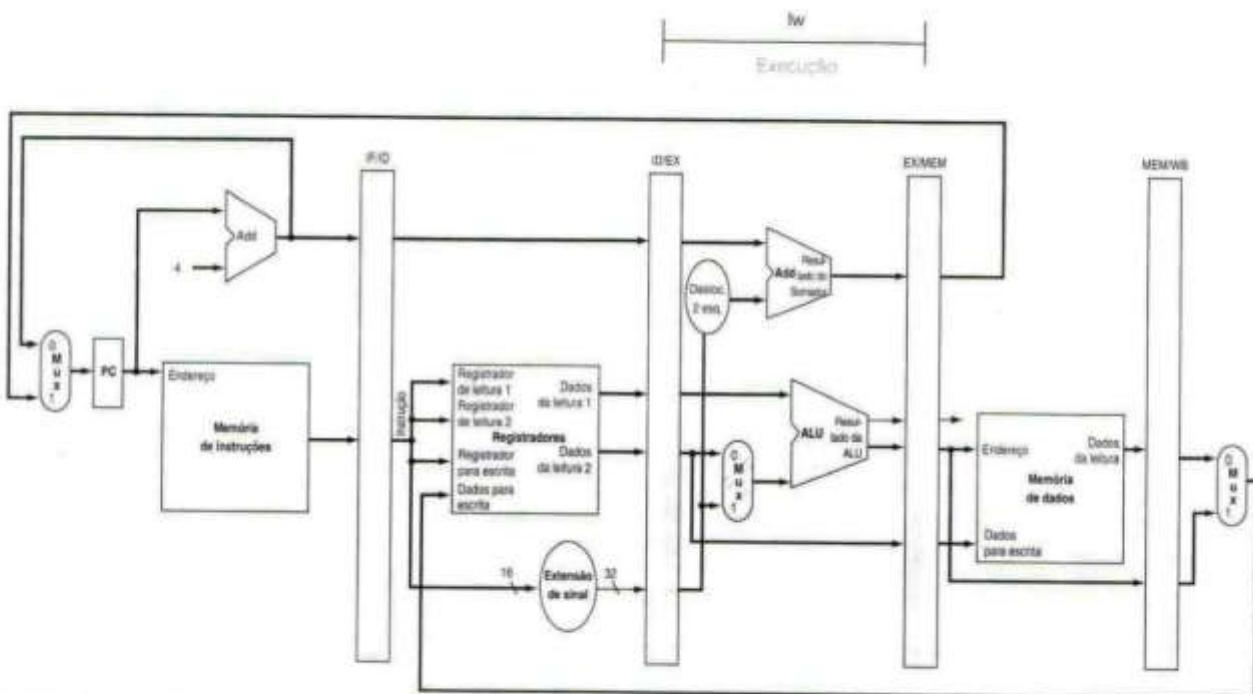


FIGURA 6.13 EX: o terceiro estágio do pipe de uma instrução load, destacando as partes do caminho de dados da Figura 6.11 usadas neste estágio do pipe. O registrador é acrescentado ao imediato com sinal estendido, e a soma é colocada no registrador de pipeline EX/MEM.

1. *Busca de instruções:* a parte superior da Figura 6.12 mostra a instrução sendo lida da memória usando o endereço no PC e depois colocada no registrador de pipeline IF/ID. O registrador de pipeline IF/ID é semelhante ao registrador de instrução na Figura 5.26. O endereço do PC é incrementado em 4 e depois escrito de volta ao PC, para que fique pronto para o próximo ciclo de clock. Esse endereço incrementado também é salvo no registrador de pipeline IF/ID caso seja necessário mais tarde para uma instrução, como beq. O computador não tem como saber que tipo de instrução está sendo buscada, de modo que precisa se preparar para qualquer instrução, passando informações potencialmente necessárias pelo pipeline.
2. *Decodificação de instruções e leitura do banco de registradores:* A parte inferior da Figura 6.12 mostra a parte relativa à instrução do registrador de pipeline IF/ID, fornecendo o campo imediato de 16 bits, que tem seu sinal estendido para 32 bits, e os números dos dois registradores para leitura. Todos os três valores são armazenados no registrador de pipeline ID/EX, assim como o endereço no PC incrementado. Novamente, transferimos tudo que possa ser necessário por qualquer instrução durante um ciclo de clock posterior.
3. *Execução ou cálculo de endereço:* a Figura 6.13 mostra que a instrução load lê o conteúdo do registrador 1 e o imediato com o sinal estendido do registrador de pipeline ID/EX e os soma usando a ALU. Essa soma é colocada no registrador de pipeline EX/MEM.
4. *Acesso à memória:* a parte superior da Figura 6.14 mostra a instrução load lendo a memória de dados por meio do endereço vindo do registrador de pipeline EX/MEM e carregando os dados no registrador de pipeline MEM/WB.
5. *Escrita do resultado:* a parte inferior da Figura 6.14 mostra a etapa final: lendo os dados do registrador de pipeline MEM/WB e escrevendo-os no banco de registradores, no meio da figura.

Essa revisão da instrução load mostra que qualquer informação necessária em um estágio posterior do pipe precisa ser passada a esse estágio por meio de um registrador de pipeline. A revisão de

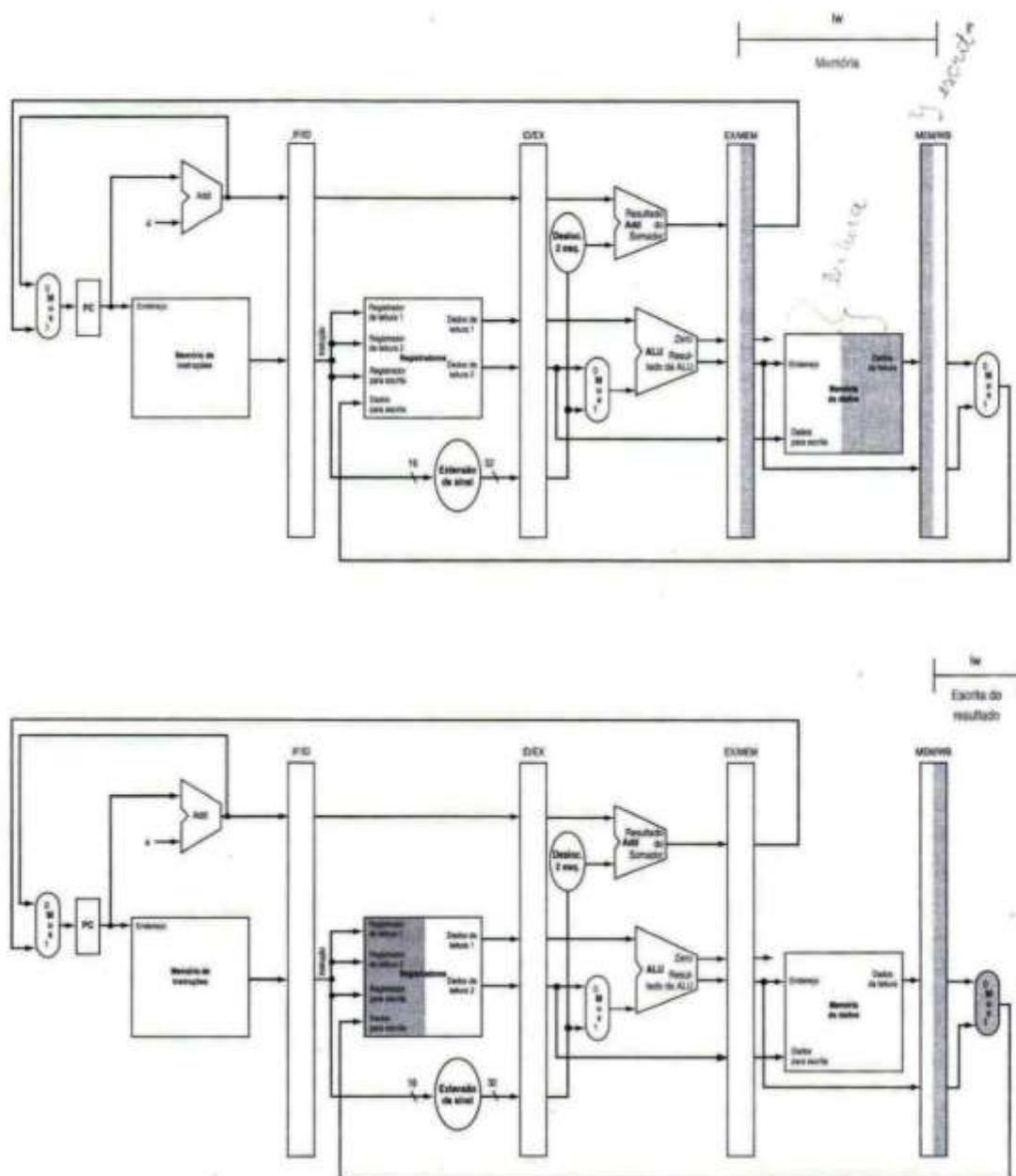


FIGURA 6.14 MEM e WB: o quarto e quinto estágios do pipe de uma instrução load, destacando as partes do caminho de dados da Figura 6.11 usadas nesses estágios do pipe. A memória de dados é lida por meio do endereço no registrador de pipeline EX/MEM, e os dados são colocados no registrador de pipeline MEM/WB. Em seguida, os dados são lidos do registrador de pipeline MEM/WB e escritos no banco de registradores, no meio do caminho de dados.

uma instrução store mostra a semelhança na execução da instrução, bem como a passagem da informação para os estágios posteriores. Aqui estão os cinco estágios do pipe da instrução store:

1. *Busca de instruções*: a instrução é lida da memória usando o endereço no PC e depois é colocada no registrador de pipeline IF/ID. Esse estágio ocorre antes que a instrução seja identificada, de modo que a parte superior da Figura 6.12 funciona para store e também para load.
2. *Decodificação de instruções e leitura do banco de registradores*: a instrução no registrador de pipeline IF/ID fornece os números de dois registradores para leitura e estende o sinal do

imediato de 16 bits. Esses três valores de 32 bits são armazenados no registrador de pipeline ID/EX. A parte inferior da Figura 6.12 para instruções load também mostra as operações do segundo estágio para stores. Esses dois primeiros estágios são executados por todas as instruções, pois é muito cedo para saber o tipo da instrução.

3. Execução e cálculo de endereço: a Figura 6.15 mostra a terceira etapa; o endereço efetivo é colocado no registrador de pipeline EX/MEM.
4. Acesso à memória: a parte superior da Figura 6.16 mostra os dados sendo escritos na memória. Observe que o registrador contendo os dados a serem armazenados foi lido em um estágio anterior e armazenado no ID/EX. A única maneira de disponibilizar os dados durante o estágio MEM é colocar os dados no registrador de pipeline EX/MEM no estágio EX, assim como armazenar o endereço efetivo em EX/MEM.
5. Escrita do resultado: a parte inferior da Figura 6.16 mostra a última etapa do store. Para essa instrução, nada acontece no estágio de escrita do resultado. Como cada instrução por trás do store já está em progresso, não temos como acelerar essas instruções. Logo, uma instrução passa por um estágio mesmo que não haja nada a fazer, pois as instruções posteriores já estão prosseguindo em velocidade máxima.

A instrução store novamente ilustra que, para passar algo de um estágio anterior do pipe para um estágio posterior, a informação precisa ser colocada em um registrador de pipeline; caso contrário, a informação é perdida quando a próxima instrução entrar nesse estágio do pipeline. Para a instrução store, precisamos passar um dos registradores lidos no estágio ID para o estágio MEM, onde é armazenado na memória. Os dados foram colocados inicialmente no registrador de pipeline ID/EX e depois passados para o registrador de pipeline EX/MEM.

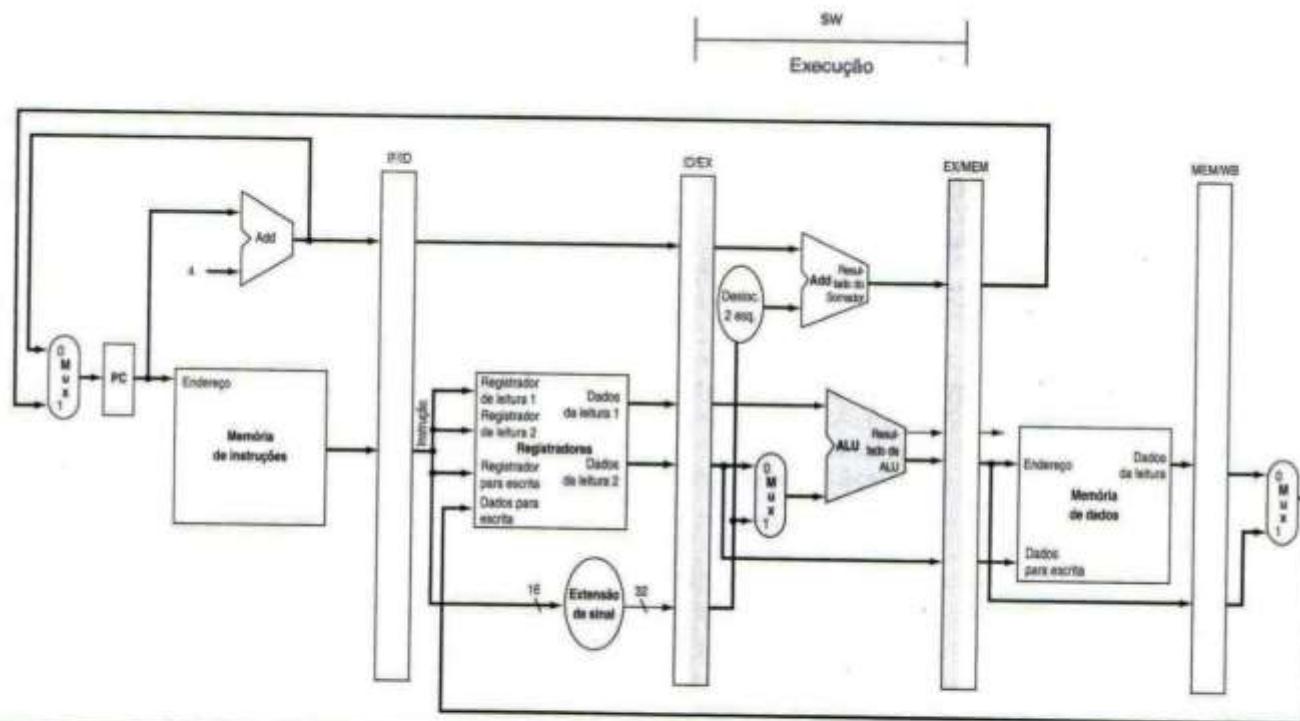


FIGURA 6.15 EX: o terceiro estágio do pipe de uma instrução store. Ao contrário do terceiro estágio da instrução load na Figura 6.13, o segundo valor do registrador é carregado no registrador de pipeline EX/MEM a ser usado no próximo estágio. Embora não faça mal algum sempre escrever esse segundo registrador no registrador de pipeline EX/MEM, escrevemos o segundo registrador apenas em uma instrução store para tornar o pipeline mais fácil de entender.

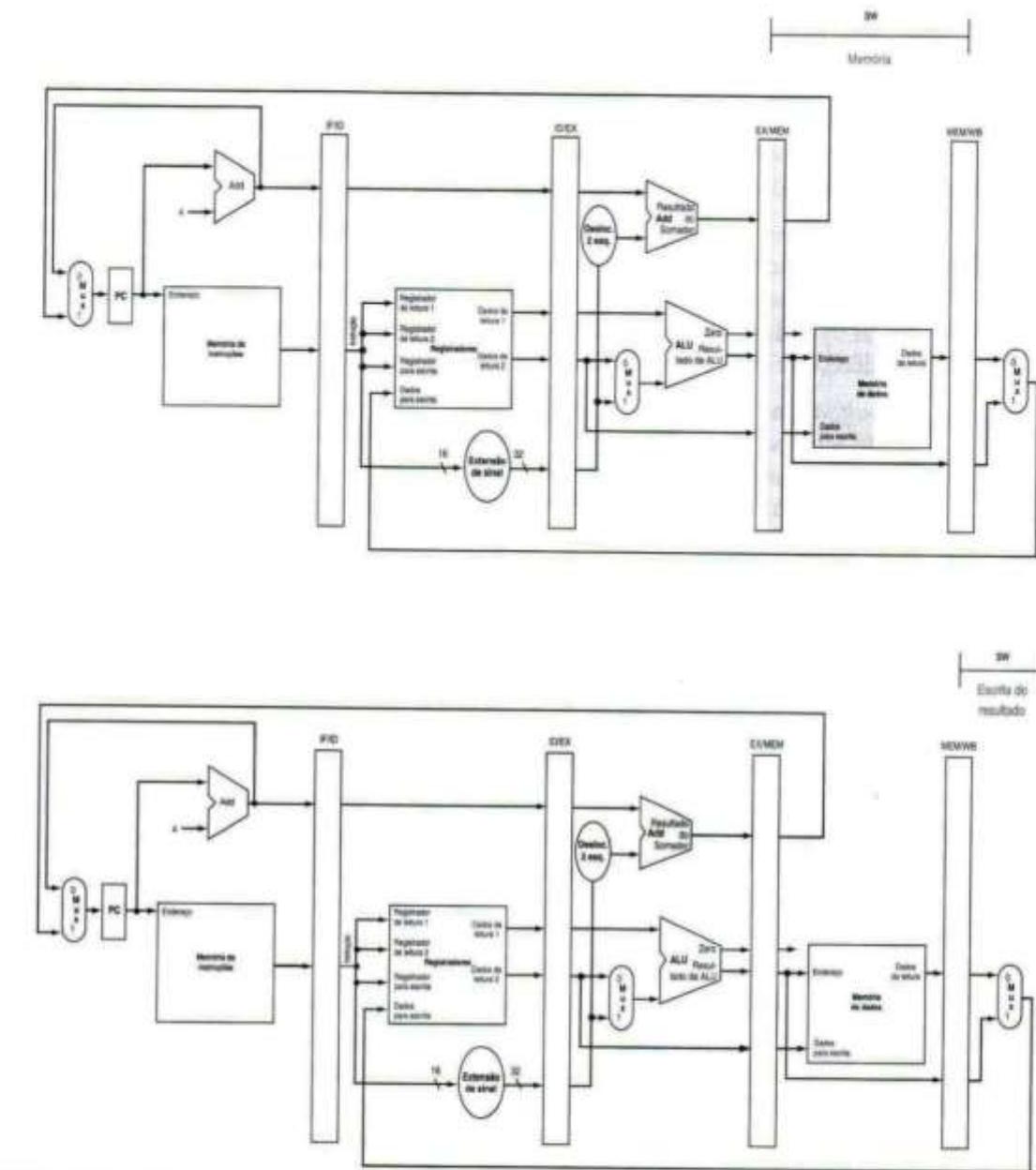


FIGURA 6.16 MEM e WB: o quarto e quinto estágios do pipe de uma instrução store. No quarto estágio, os dados são escritos na memória de dados para o store. Observe que os dados vêm do registrador de pipeline EX/MEM e que nada é mudado no registrador de pipeline MEM/WB. Quando os dados são escritos na memória, não há nada mais para a instrução store fazer, de modo que nada acontece no estágio 5.

Load e store ilustram um segundo ponto importante: cada componente lógico do caminho de dados – como memória de instruções, portas para leitura de registradores, ALU, memória de dados e porta para escrita de registradores – só pode ser usado dentro de um único estágio do pipeline. Caso contrário, teríamos um *hazard estrutural* (ver página 282). Logo, esses componentes e seu controle podem ser associados a um único estágio do pipeline.

Agora, podemos descobrir um bug no projeto da instrução load. Você conseguiu ver? Qual registrador é alterado no estágio final da leitura? Mais especificamente, qual instrução fornece o número do registrador de escrita? A instrução no registrador de pipeline IF/ID fornece o número do registrador de escrita, embora essa instrução ocorra consideravelmente *depois* da instrução load!

Logo, precisamos preservar o número do registrador de destino da instrução load. Assim como store passou o *conteúdo* do registrador do ID/EX para o registrador de pipeline EX/MEM para uso no estágio MEM, load precisa passar o *número* do registrador de ID/EX por EX/MEM para o registrador de pipeline MEM/WB, para o uso no estágio WB. Outra maneira de pensar sobre a passagem do número de registrador é que, para compartilhar o caminho de dados em pipeline, precisávamos preservar a instrução lida durante o estágio IF, de modo que cada registrador de pipeline contenha uma parte da instrução necessária para esse estágio e para os estágios posteriores.

A Figura 6.17 mostra a versão correta do caminho de dados, passando o número do registrador de escrita primeiro para o registrador ID/EX, depois para o registrador EX/MEM, e finalmente

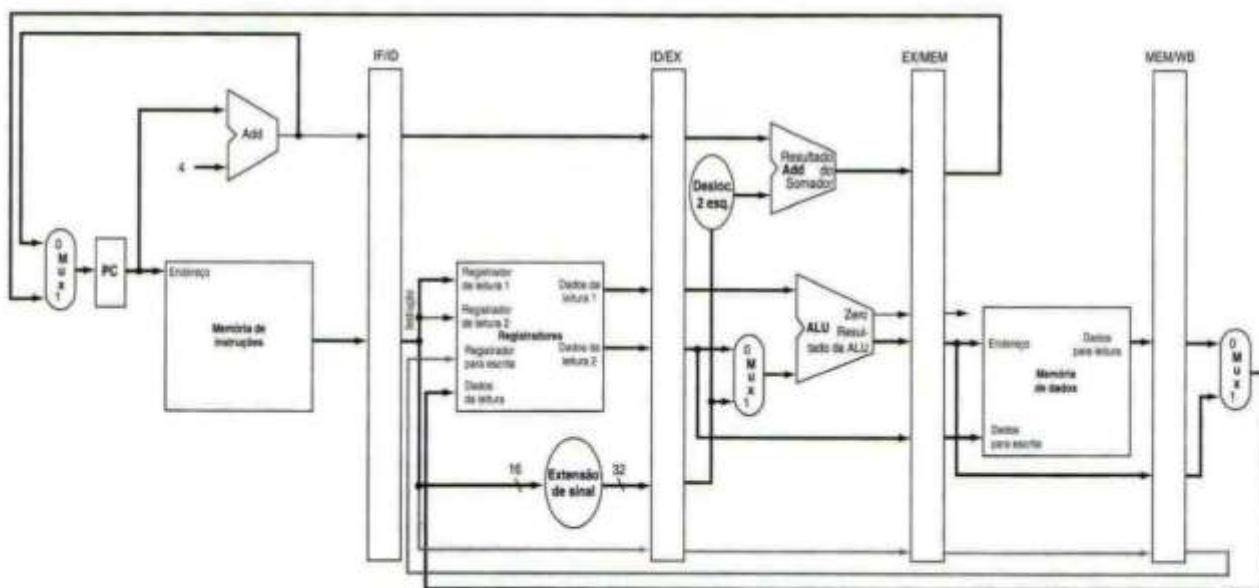


FIGURA 6.17 O caminho de dados em pipeline corrigido para lidar corretamente com a instrução load. O número do registrador de escrita agora vem do registrador de pipeline MEM/WB junto com os dados. O número do registrador é passado do estágio do pipe ID até alcançar o registrador de pipeline MEM/WB, acrescentando mais 5 bits aos três últimos registradores de pipeline. Esse novo caminho aparece em destaque.

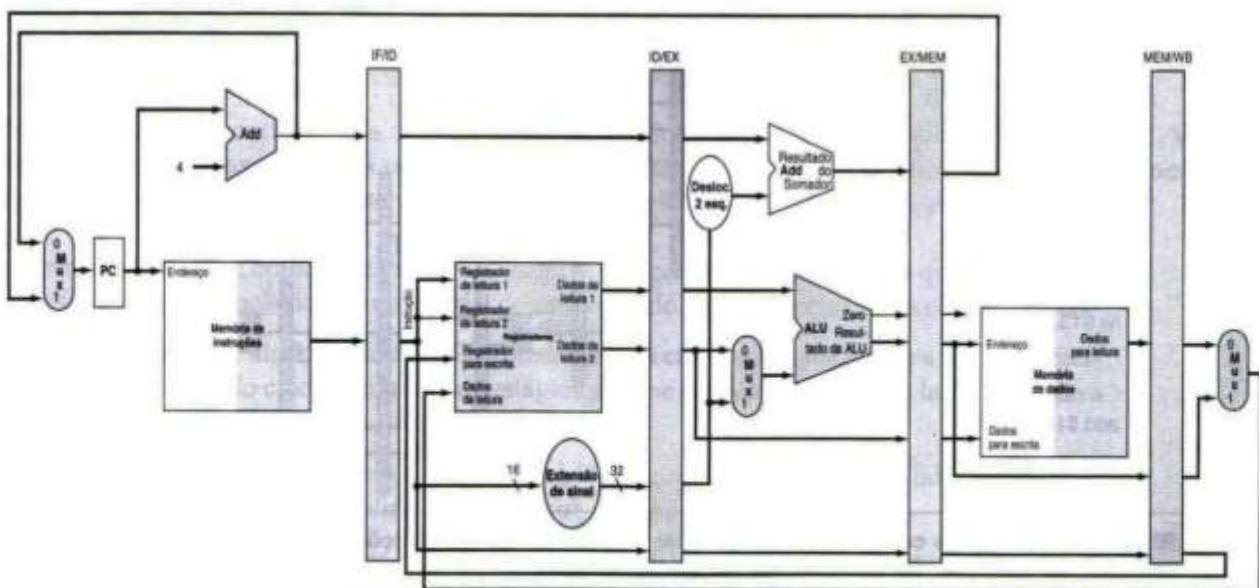


FIGURA 6.18 A parte do caminho de dados na Figura 6.17 usada em todos os cinco estágios de uma instrução load.

para o registrador MEM/WB. O número do registrador é usado durante o estágio WB para especificar o registrador a ser escrito. A Figura 6.18 é um único desenho do caminho de dados correto, destacando o hardware utilizado em todos os cinco estágios da instrução load word nas Figuras de 6.12 a 6.14. Ver na Seção 6.6 uma explicação de como fazer a instrução branch funcionar como esperado.

Representando pipelines graficamente

Pipelining pode ser difícil de entender, pois muitas instruções estão executando simultaneamente em um único caminho de dados em cada ciclo de clock. Para ajudar na compreensão, existem dois estilos básicos de figuras de pipeline: *diagramas de pipeline com múltiplos ciclos de clock*, como a Figura 6.10, e *diagramas de pipeline com único ciclo de clock*, como as Figuras de 6.12 a 6.16. Os diagramas com múltiplos ciclos de clock são mais simples, mas não contêm todos os detalhes. Por exemplo, considere a seguinte sequência de cinco instruções:

lw	\$10, 20(\$1)
sub	\$11, \$2, \$3
add	\$12, \$3, \$4
lw	\$13, 24(\$1)
add	\$14, \$5, \$6

A Figura 6.19 mostra o diagrama de pipeline com múltiplos ciclos de clock para essas instruções. O tempo avança da esquerda para a direita na horizontal, semelhante ao pipeline da lavanderia, na Figura 6.1. Uma representação dos estágios do pipeline é colocada em cada parte do eixo de instruções, ocupando os ciclos de clock apropriados. Esses caminhos de dados estilizados representam os cinco estágios do nosso pipeline, mas um retângulo indicando o nome de cada estágio do pipe também fun-

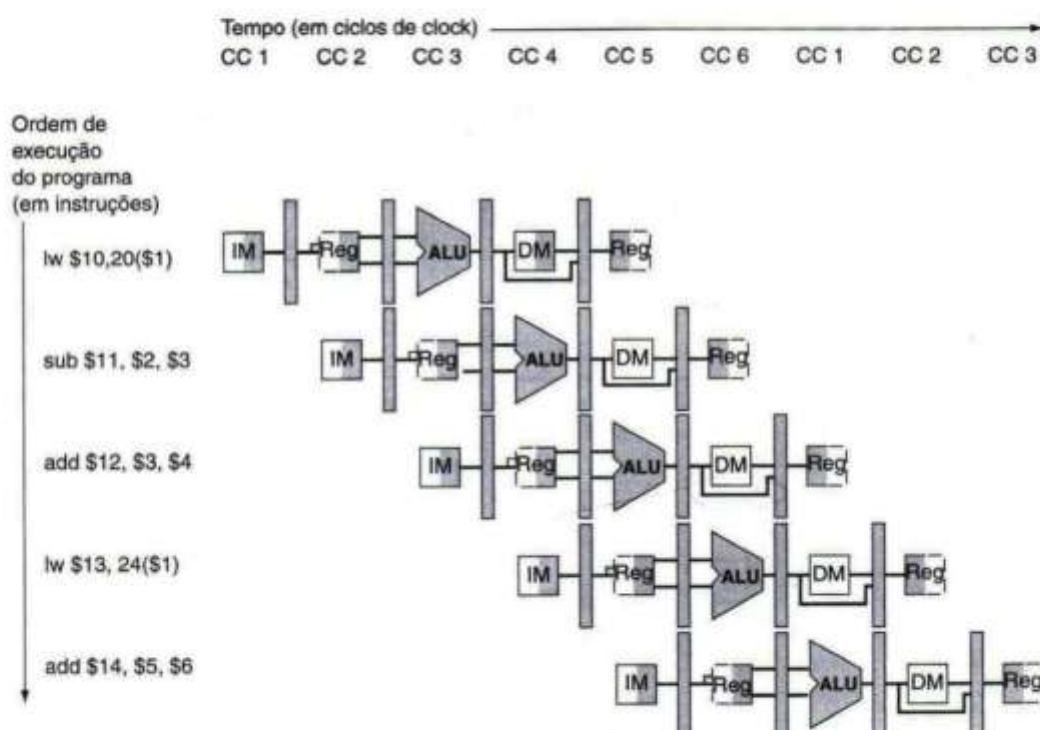


FIGURA 6.19 Diagrama de pipeline com múltiplos ciclos de clock das cinco instruções. Esse estilo de representação de pipeline mostra a execução completa das instruções em uma única figura. As instruções são listadas por ordem de execução, de cima para baixo, e os ciclos de clock se movem da esquerda para a direita. Ao contrário da Figura 6.4, aqui, mostramos os registradores de pipeline entre cada estágio. A Figura 6.20 mostra a maneira tradicional de desenhar esse diagrama.



FIGURA 6.20 Diagrama de pipeline com múltiplos ciclos de clock tradicional, com as cinco instruções da Figura 6.19.

ciona bem. A Figura 6.20 mostra a versão mais tradicional do diagrama de pipeline com múltiplos ciclos de clock. Observe que a Figura 6.19 mostra os recursos físicos utilizados em cada estágio, enquanto a Figura 6.20 usa o *nome* de cada estágio. Usamos os diagramas com múltiplos ciclos de clock para oferecer uma idéia das situações de pipelining.

Os diagramas de pipeline de ciclo único de clock mostram o estado do caminho de dados inteiro durante um único ciclo de clock, e normalmente todas as cinco instruções no pipeline são identificadas por rótulos acima de seus respectivos estágios do pipeline. Usamos esse tipo de figura para mostrar os detalhes do que está acontecendo dentro do pipeline durante cada ciclo de clock; normalmente, os desenhos aparecem em grupos, para mostrar a operação do pipeline durante uma seqüência de ciclos de clock. Um diagrama de ciclo único de clock representa uma fatia vertical de um conjunto do diagrama com múltiplos ciclos de clock, mostrando o uso do caminho de dados em cada uma das instruções do pipeline no ciclo de clock designado. Por exemplo, a Figura 6.21 mostra o diagrama com ciclo único de clock correspondente ao ciclo de clock 5 das Figuras 6.19 e 6.20. Obviamente, os diagramas com único ciclo de clock possuem mais detalhes e ocupam muito mais espaço para mostrar o mesmo número de ciclos de clock. A Seção ■ “Aprofundando o aprendizado” incluída no CD contém os diagramas com ciclo único de clock correspondentes a essas duas instruções, além de exercícios pedindo que você crie tais diagramas para outra seqüência de código.

Um grupo de alunos discutia sobre a eficiência de um pipeline de cinco estágios quando um deles apontou que nem todas as instruções estão ativas em cada estágio do pipeline. Depois de decidir ignorar os efeitos dos hazards, eles fizeram as cinco afirmações a seguir. Quais delas estão corretas?

Verifique você mesmo

1. Permitir que jumps, branches e instruções da ALU utilizem menos estágios do que os cinco necessários pela instrução load aumentará o desempenho do pipeline sob todas as circunstâncias.
2. Tentar permitir que algumas instruções utilizem menos ciclos não-ajuda, pois a vazão é determinada pelo ciclo do clock; o número de estágios do pipe por instrução afeta a latência, e não a vazão.
3. Permitir que jumps, branches e operações da ALU utilizem menos ciclos só ajuda quando nem loads nem stores estão no pipeline, de modo que os benefícios são pequenos.
4. Você não pode fazer com que as instruções da ALU utilizem menos ciclos, devido à escrita do resultado, mas os branches e jumps podem utilizar menos ciclos, de modo que existe alguma oportunidade de melhoria.

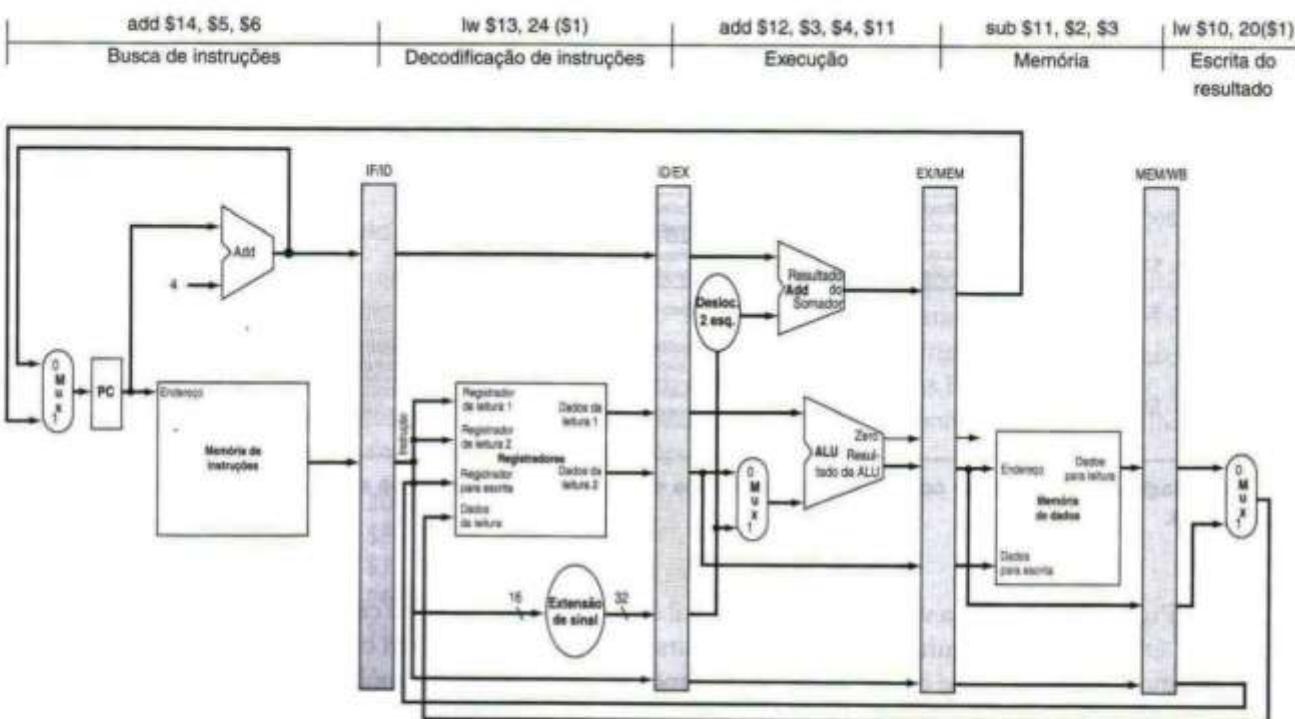


FIGURA 6.21 O diagrama com ciclo único de clock correspondente ao ciclo de clock 5 do pipeline das Figuras 6.19 e 6.20.
Como você pode ver, uma figura com ciclo único de clock é uma fatia vertical de um diagrama com múltiplos ciclos de clock.

5. Em vez de tentar fazer com que as instruções utilizem menos ciclos de clock, devemos explorar um meio de tornar o pipeline mais longo, de modo que as instruções utilizem mais ciclos, porém com ciclos mais curtos. Isso poderia melhorar o desempenho.

No computador 6600, talvez ainda mais do que em qualquer computador anterior, o sistema de controle é a diferença.

James Thornton,
Design of a Computer: The Control Data 6600,
1970

6.3 Controle de um pipeline

Assim como acrescentamos controle ao caminho de dados simples na Seção 5.4, agora acrescentamos controle ao caminho de dados de um pipeline. Começamos com um projeto simples, que vê o problema por meio de óculos cor de rosa; nas Seções de 6.4 a 6.8, removemos os óculos para revelar os hazards do mundo real.

O primeiro passo é rotular as linhas de controle no caminho de dados existente. A Figura 6.22 mostra essas linhas. Pegamos o máximo possível emprestado do controle para o caminho de dados simples da Figura 5.17. Em particular, usamos a mesma lógica de controle da ALU, lógica de desvio, multiplexador do registrador destino e linhas de controle. Essas funções são definidas na Figura 5.12, Figura 5.16 e Figura 5.18. Reproduzimos as principais informações nas Figuras 6.23 a 6.25 para facilitar o acompanhamento do restante do texto.

Quanto à implementação com ciclo único discutida no Capítulo 5, consideramos que o PC é escrito a cada ciclo de clock, de modo que não existe um sinal de escrita separado para o PC. Pelo mesmo argumento, não existem sinais de escrita para os registradores de pipeline (IF/ID, ID/EX, EX/MEM e MEM/WB), pois os registradores de pipeline também são escritos durante cada ciclo de clock.

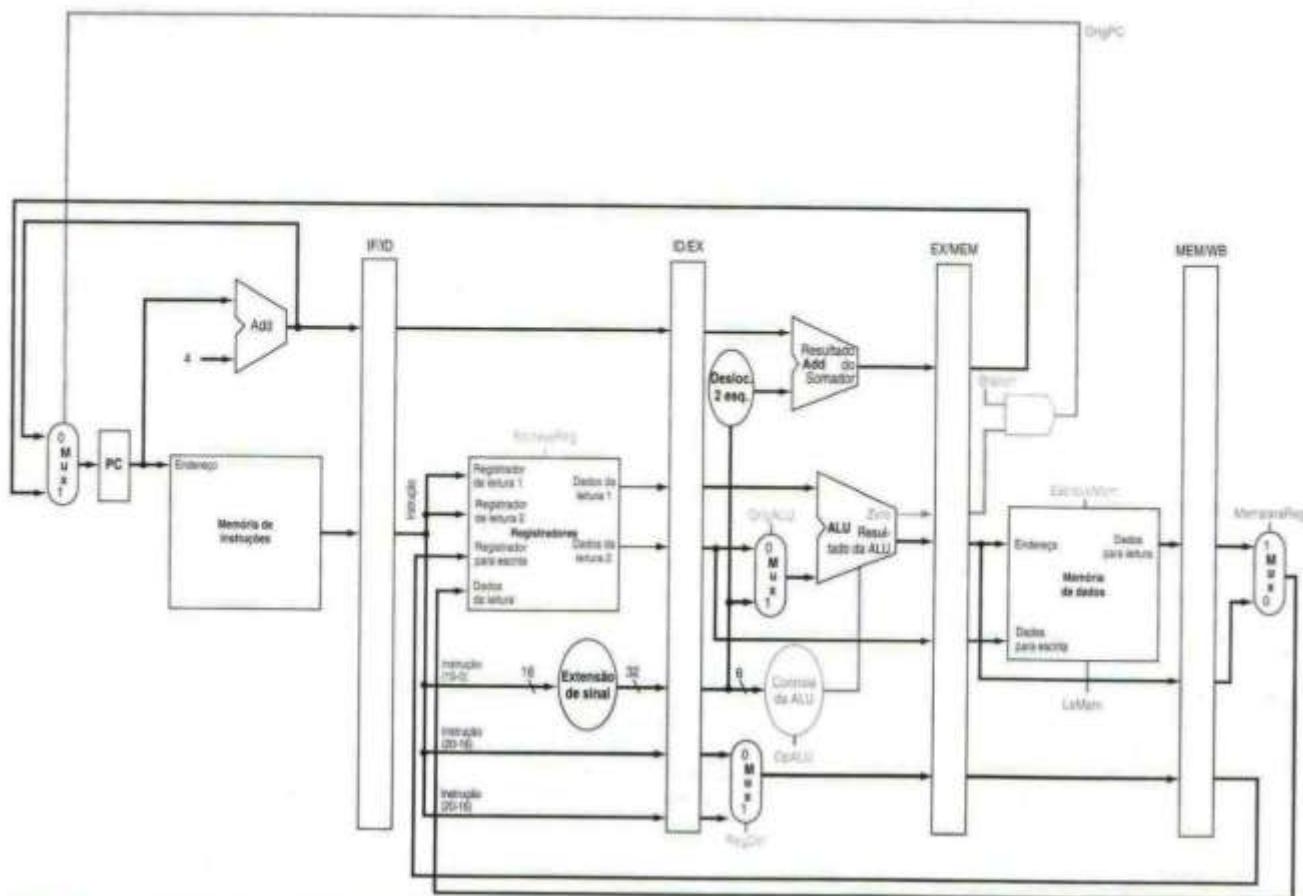


FIGURA 6.22 O caminho de dados em pipeline da Figura 6.17 com sinais de controle identificados. Esse caminho de dados toma emprestado a lógica de controle para a origem do PC, o número do registrador destino e o controle da ALU, do Capítulo 5. Observe que agora precisamos do campo funct (código de função) de 6 bits da instrução no estágio EX como entrada para o controle da ALU, de modo que esses bits também precisam ser incluídos no registrador de pipeline ID/EX. Lembre-se de que esses 6 bits também são os bits menos significativos do campo imediato da instrução, de modo que o registrador de pipeline ID/EX pode fornecê-los a partir do campo imediato, já que a extensão do sinal deixa esses bits inalterados.

Opcode da instrução	OpALU	Operação da instrução	Campo funct	Ação da ALU desejada	Entrada do controle da ALU
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
Tipo R	10	add	100000	add	0010
Tipo R	10	subtract	100010	subtract	0110
Tipo R	10	AND	100100	and	0000
Tipo R	10	OR	100101	or	0001
Tipo R	10	set on less than	101010	set on less than	0111

FIGURA 6.23 Uma cópia da Figura 5.12. Essa figura mostra como os bits do controle da ALU são definidos dependendo dos bits de controle OpALU e dos diferentes códigos de função para instruções tipo R.

Nome do sinal	Efeito quando inativo	Efeito quando ativo
RegDst	O número do registrador destino para a entrada Registrador para escrita vem do campo rt (bits 20:16).	O número do registrador destino para a entrada Registrador para escrita vem do campo rd (bits 15:11).
EscreveReg	Nenhum.	O registrador na entrada Registrador para escrita é escrito com o valor na entrada Dados para escrita.
OrigALU	O segundo operando da ALU vem da segunda saída do banco de registradores (Dados da leitura 2).	O segundo operando da ALU consiste nos 16 bits mais baixos da instrução com sinal estendido.
OrigPC	O PC é substituído pela saída do somador que calcula o valor de PC + 4.	O PC é substituído pela saída do somador que calcula o destino do desvio.
LeMem	Nenhum.	O conteúdo da memória de dados designado pela entrada Endereço é colocado na saída Dados da leitura.
EscreveMem	Nenhum.	O conteúdo da memória de dados designado pela entrada Endereço é substituído pelo valor na entrada Dados para escrita.
MemparaReg	O valor enviado para a entrada Dados para escrita do banco de registradores vem da ALU.	O valor enviado para a entrada Dados para escrita do banco de registradores vem da memória de dados.

FIGURA 6.24 Uma cópia da Figura 5.16. A função de cada um dos sete sinais de controle é definida. As linhas de controle da ALU (OpALU) são definidas na segunda coluna da Figura 6.23. Quando um controle de 1 bit para um multiplexador bidirecional é ativado, o multiplexador seleciona a entrada correspondente a 1. Caso contrário, se o controle for desativado, o multiplexador seleciona a entrada 0. Observe que OrigPC é controlado por uma porta lógica AND na Figura 6.22. Se o sinal Branch e o sinal Zero da ALU estiverem ativos, então OrigPC é 1; caso contrário, ele é 0. O controle define o sinal Branch somente durante uma instrução beq; caso contrário, o OrigPC é 0.

Instrução	Linhas de controle do estágio de execução/cálculo de endereço				Linhas de controle do estágio de acesso à memória			Linhas de controle do estágio de escrita do resultado	
	RegDst	OpALU1	OpALU0	OrigALU	Branch	LeMem	EscreveMem	EscreveReg	MemparaReg
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 6.25 Os valores das linhas de controle são iguais aos da Figura 5.18, mas foram reorganizados em três grupos, correspondentes aos três últimos estágios do pipeline.

Para especificar o controle para o pipeline, só precisamos definir os valores de controle durante cada estágio do pipeline. Como cada linha de controle está associada a um componente ativo em apenas um estágio do pipeline, podemos dividir as linhas de controle em cinco grupos, de acordo com o estágio do pipeline.

1. *Busca de instruções:* os sinais de controle para ler a memória de instruções e escrever o PC sempre são ativados, de modo que não existe nada de especial para controlar nesse estágio do pipeline.
2. *Decodificação de instruções/leitura do banco de registradores:* como no estágio anterior, a mesma coisa acontece em cada ciclo de clock, de modo que não existem linhas de controle opcionais para definir.
3. *Execução/cálculo de endereço:* os sinais a serem definidos são RegDst, OpALU e OrigALU (ver Figuras 6.23 e 6.24). Os sinais selecionam o registrador destino, a operação da ALU e Dados da leitura 2 ou um imediato com sinal estendido para a ALU.
4. *Acesso à memória:* as linhas de controle definidas nesse estágio são Branch, LeMem e EscreveMem. Esses sinais são definidos pelas instruções branch equal, load e store, respectivamente. Lembre-se de que o OrigPC na Figura 6.24 seleciona o próximo endereço seqüencial, a menos que o controle ative Branch e o resultado da ALU seja zero.

5. *Escrita do resultado:* as duas linhas de controle são MemparaReg, que decide entre enviar o resultado da ALU ou o valor da memória para o banco de registradores, e EscreveReg, que escreve o valor escolhido.

Como a utilização de um pipeline o caminho de dados deixa inalterado o significado das linhas de controle, podemos usar os mesmos valores de controle de antes. A Figura 6.25 tem os mesmos valores do Capítulo 5, mas agora as nove linhas de controle estão agrupadas por estágio do pipeline.

A implementação do controle significa definir as nove linhas de controle para esses valores em cada estágio, para cada instrução. A maneira mais simples de fazer isso é estender os registradores do pipeline para incluir informações de controle.

Como as linhas de controle começam com o estágio EX, podemos criar a informação de controle durante a decodificação da instrução. A Figura 6.26 mostra que esses sinais de controle são usados no respectivo estágio do pipeline à medida que a instrução se move pelo pipeline, assim como o número do registrador destino para loads se move pelo pipeline da Figura 6.17. A Figura 6.27 mostra o caminho de dados completo, com os registradores de pipeline estendidos e com as linhas de controle conectadas ao estágio apropriado.

6.4 Hazards de dados e forwarding

Os exemplos da seção anterior mostram o poder da execução em pipeline e como o hardware realiza a tarefa. Agora é hora de retirarmos os óculos cor de rosa e examinarmos o que acontece com os programas reais. As instruções nas Figuras de 6.19 a 6.21 eram independentes; nenhuma delas usava os resultados calculados por qualquer uma das outras. Mesmo assim, na Seção 6.1, vimos que os hazards de dados são obstáculos para a execução em pipeline.

Vejamos uma seqüência com muitas dependências, indicadas com realce:

O que você quer dizer, por que isso precisa ser criado? É uma alternativa. Você precisa criar alternativas.
Douglas Adams,
Hitchhikers Guide to the Galaxy,
1979

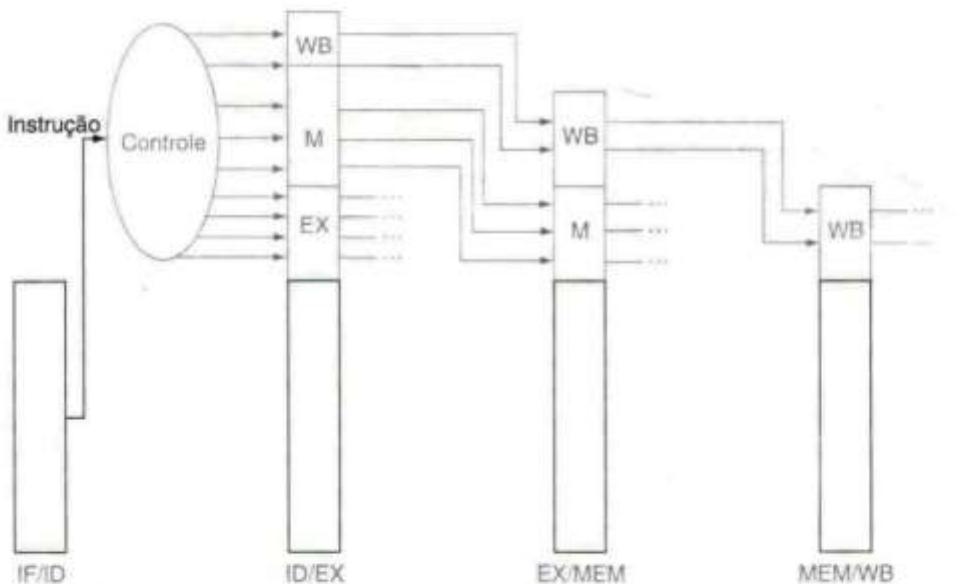


FIGURA 6.26 As linhas de controle para os três estágios finais. Observe que quatro das nove linhas de controle são usadas na fase EX, com as cinco linhas de controle restantes passadas adiante para o registrador de pipeline EX/MEM, para manter as linhas de controle; três são usadas durante o estágio MEM, e as duas últimas são passadas para MEM/WB, para uso no estágio WB.

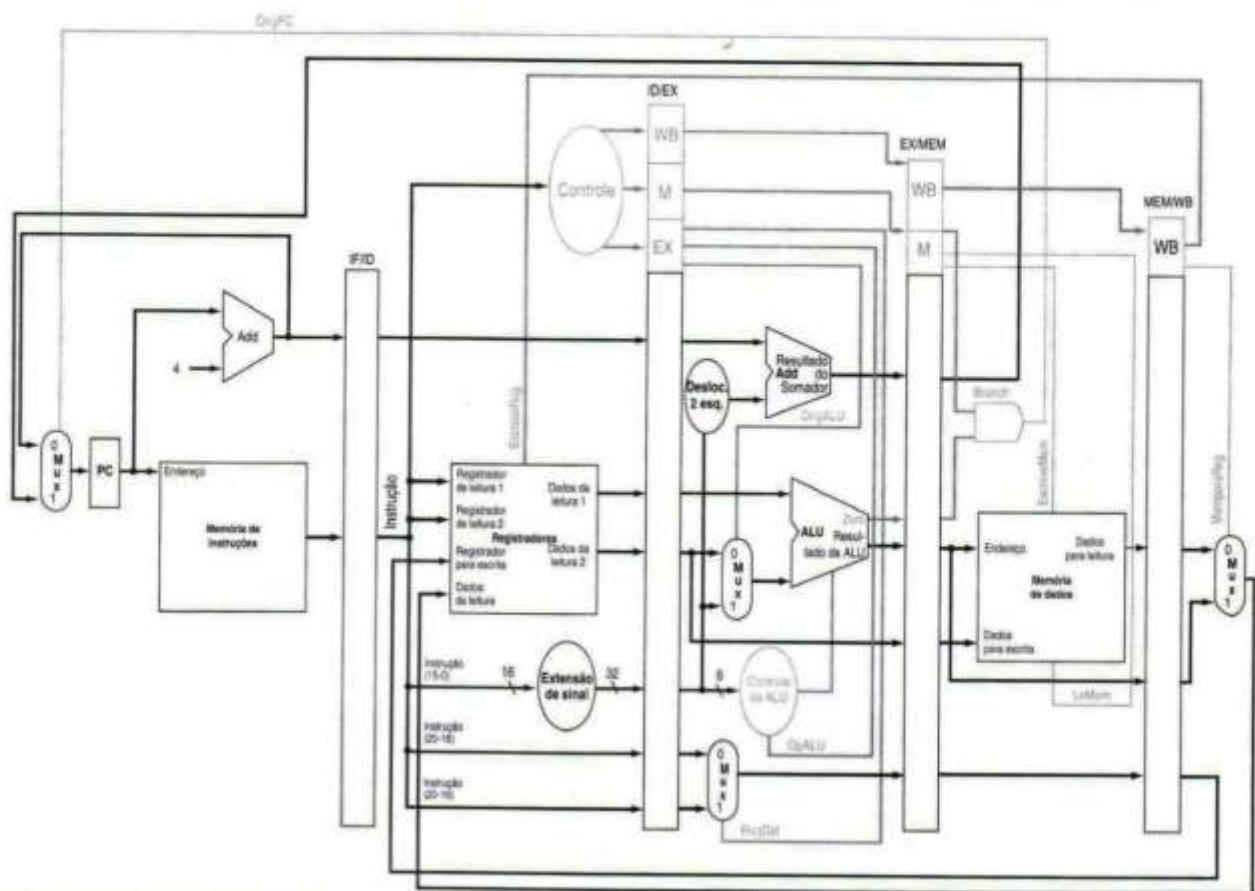


FIGURA 6.27 O caminho de dados em pipeline da Figura 6.22, com os sinais de controle conectados às partes de controle dos registradores de pipeline. Os valores de controle para os três últimos estágios são criados durante o estágio de decodificação de instruções e depois colocados no registrador de pipeline ID/EX. As linhas de controle para cada estágio do pipe são usadas, e as linhas de controle restantes depois disso são passadas para o próximo estágio do pipeline.

```

sub $2, $1,$3      # Registrador $2 escrito por sub
and $12,$2,$5      # 1º operando ($2) depende de sub
or $13,$6,$2       # 2º operando ($2) depende de sub
add $14,$2,$2      # 1º ($2) & 2º ($2) dependem de sub
sw $15,100($2)    # Base ($2) depende de sub
  
```

As quatro últimas instruções são todas dependentes do resultado no registrador \$2 da primeira instrução. Se o registrador \$2 tivesse o valor 10 antes da instrução subtract e -20 depois dela, o programador desejaria que -20 fosse usado nas instruções seguintes que se referem ao registrador \$2.

Como essa seqüência funcionaria com nosso pipeline? A Figura 6.28 ilustra a execução dessas instruções usando uma representação de pipeline com múltiplos ciclos de clock. Para demonstrar a execução dessa seqüência de instruções em nosso pipeline atual, o topo da Figura 6.28 mostra o valor do registrador \$2, que muda durante o ciclo de clock 5, quando a instrução sub escreve seu resultado.

Um hazard em potencial pode ser resolvido pelo projeto do hardware do banco de registradores: o que acontece quando um registrador é lido e escrito no mesmo ciclo de clock? Consideramos que a escrita está na primeira metade do ciclo de clock e a leitura está na segunda metade, de modo que a leitura fornece o que foi escrito. Como acontece para muitas implementações dos bancos de registradores, não temos hazard de dados nessa situação.

A Figura 6.28 mostra que os valores lidos para o registrador \$2 *não* seriam o resultado da instrução sub, a menos que a leitura ocorresse durante o ciclo de clock 5 ou posterior. Assim, as instruções que receberiam o valor correto de -20 são add e sw; as instruções and e or receberiam o valor incorreto

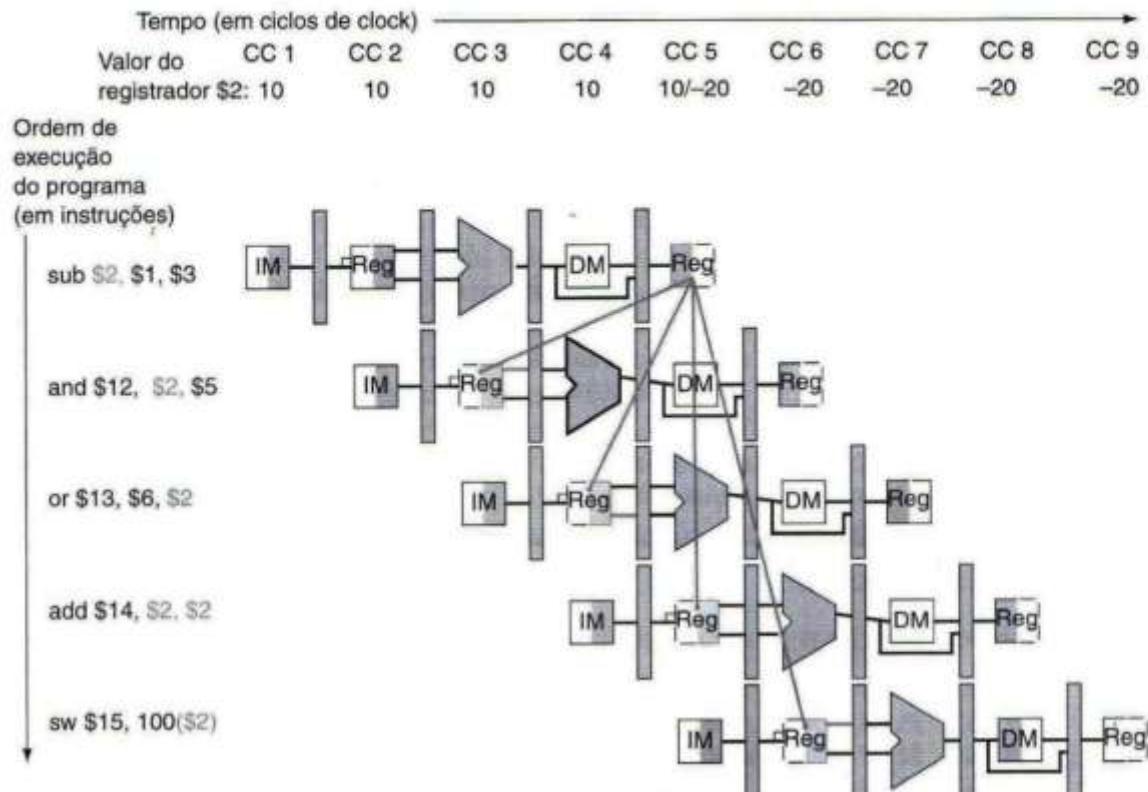


FIGURA 6.28 Dependências em pipeline em uma seqüência de cinco instruções usando caminhos de dados simplificados para mostrar as dependências. Todas as ações dependentes são mostradas em cinza, e “CC *i*” no alto da figura significa o ciclo de clock *i*. A primeira instrução escreve em \$2, e todas as instruções seguintes leem de \$2. Esse registrador é escrito no ciclo de clock 5, de modo que o valor correto está indisponível antes do ciclo de clock 5. (Uma leitura de um registrador durante um ciclo de clock retorna o valor escrito no final da primeira metade do ciclo, quando ocorre tal escrita.) As linhas coloridas do caminho de dados do topo para os inferiores mostram as dependências. Aquelas que precisam retornar no tempo são os *hazards de dados do pipeline*.

de 10! Usando esse estilo de desenho, esses problemas se tornam aparentes quando uma linha de dependência retorna no tempo.

Mas examine cuidadosamente a Figura 6.28. Quando é produzido realmente os dados da instrução sub? O resultado está disponível no final do estágio EX, ou ciclo de clock 3. Quando os dados são realmente necessários pelas instruções and e or? No inicio do estágio EX, ou nos ciclos de clock 4 e 5, respectivamente. Assim, podemos executar esse segmento sem stalls se simplesmente os dados sofrerem *forwarding* assim que estiverem disponíveis para quaisquer unidades que precisam deles antes de estarem disponíveis para leitura do banco de registradores.

Como funciona o forwarding? Para simplificar o restante desta seção, consideramos apenas o desafio de forwarding para uma operação no estágio EX, que pode ser uma operação da ALU ou um cálculo de endereço efetivo. Isso significa que, quando uma instrução tenta usar um registrador em seu estágio EX, que uma instrução anterior pretende escrever em seu estágio WB, na realidade precisamos dos valores como entradas para a ALU.

Uma notação que nomeia os campos dos registradores de pipeline permite uma notação mais precisa das dependências. Por exemplo, “ID/EX.RegistradorRs” refere-se ao número de um registrador cujo valor se encontra no registrador de pipeline ID/EX; ou seja, aquele da primeira porta de leitura do banco de registradores. A primeira parte do nome, à esquerda do ponto, é o nome do registrador de pipeline; a segunda parte é o nome do campo nesse registrador. Usando essa notação, os dois pares de condições de hazard são

- 1a. EX/MEM.RegistradorRd = ID/EX.RegistradorRs
- 1b. EX/MEM.RegistradorRd = ID/EX.RegistradorRt
- 2a. MEM/WB.RegistradorRd = ID/EX.RegistradorRs
- 2b. MEM/WB.RegistradorRd = ID/EX.RegistradorRt

O primeiro hazard na seqüência da página 304 está no registrador \$2, entre o resultado de `sub $2,$1,$3` e o primeiro operando de leitura de `and $12,$2,$5`. Esse hazard pode ser detectado quando a instrução `and` está no estágio EX e a instrução anterior está no estágio MEM, de modo que este é o hazard 1a:

$$\text{EX/MEM.RegistradorRd} = \text{ID/EX.RegistradorRs} = \$2$$

DETECÇÃO DE DEPENDÊNCIA

EXEMPLO

Classifique as dependências nesta seqüência da página 304:

```
sub  $2, $1, $3 # Registrador $2 escrito por sub
and  $12, $2, $5 # 1º operando ($2) depende de sub
or   $13, $6, $2 # 2º operando ($2) depende de sub
add  $14, $2, $2 # 1º ($2) & 2º ($2) dependem de sub
sw   $15, 100($2) # Index ($2) depende de sub
```

RESPOSTA

Conforme já mencionamos, o `sub-and` é um hazard tipo 1a. Os outros hazards são:

- O `sub-or` é um hazard tipo 2b:

$$\text{MEM/WB.RegistradorRd} = \text{ID/EX.RegistradorRt} = \$2$$

- As duas dependências em `sub-add` não são hazards, pois o banco de registradores fornece os dados apropriados durante o estágio ID de `add`.
- Não existe hazard de dados entre `sub` e `sw`, porque `sw` lê \$2 no ciclo de clock *depois* que `sub` escreve \$2.

Como algumas instruções não escrevem em registradores, essa política não é exata; às vezes, poderia haver forwarding desnecessário. Uma solução é simplesmente verificar se o sinal `EscreveReg` estará ativo: examinando o campo de controle WB do registrador de pipeline durante os estágios EX e MEM, é possível determinar se `EscreveReg` está ativo. Além disso, o MIPS exige que cada uso de \$0 como operando deve gerar um valor de operando 0. Se uma instrução no pipeline tiver \$0 como seu destino (por exemplo, `sll $0,$1,2`), queremos evitar o forwarding do seu valor possivelmente diferente de zero. Não encaminhar os resultados destinados a \$0 libera o programador assembly e o compilador de qualquer requisito para evitar o uso de \$0 como destino. As condições anteriores, portanto, funcionam corretamente desde que acrescentemos `EX/MEM.RegistradorRd ≠ 0` à primeira condição de hazard e `MEM/WB.RegistradorRd ≠ 0` à segunda.

Agora que podemos detectar os hazards, metade do problema está resolvido – mas ainda precisamos fazer o forwarding dos dados corretos.

A Figura 6.29 mostra as dependências entre os registradores de pipeline e as entradas da ALU para a mesma seqüência de código da Figura 6.28. A mudança é que a dependência começa por um registrador de *pipeline*, em vez de esperar pelo estágio WB para escrever no banco de registradores.

Assim, os dados exigidos existem a tempo para as instruções posteriores, com os registradores de pipeline mantendo os dados para forwarding.

Se pudermos pegar as entradas da ALU de *qualquer* registrador de pipeline, e não apenas de ID/EX, então podemos fazer o forwarding dos dados corretos. Acrescentando multiplexadores à entrada da ALU e com os controles apropriados, podemos executar o pipeline em velocidade máxima na presença dessas dependências de dados.

Por enquanto, vamos considerar que as únicas instruções para as quais precisamos de forwarding são as quatro instruções no formato R: add, sub, and e or. A Figura 6.30 mostra um detalhe da ALU e do registrador de pipeline antes e depois de acrescentar o forwarding. A Figura 6.31 mostra os valores das linhas de controle para os multiplexadores da ALU que selecionam os valores do banco de registradores ou um dos valores de forwarding.

Esse controle de forwarding estará no estágio EX porque os multiplexadores de forwarding da ALU são encontrados nesse estágio. Assim, temos de passar os números dos registradores operandos do estágio ID por meio do registrador de pipeline ID/EX, para determinar se os valores devem sofrer forwarding. Já temos o campo rt (bits 20-16). Antes do forwarding, o registrador ID/EX não precisava incluir espaço para manter o campo rs. Logo, rs (bits 25-21) é acrescentado a ID/EX.

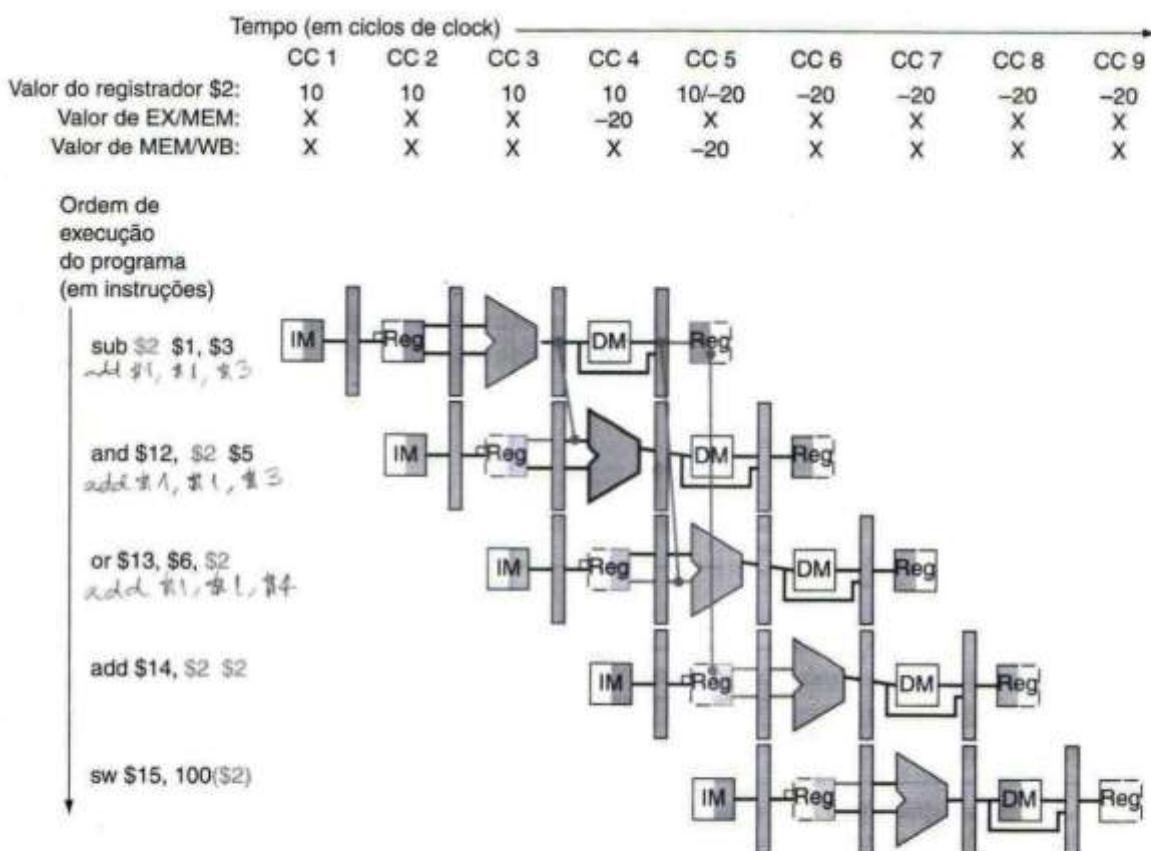
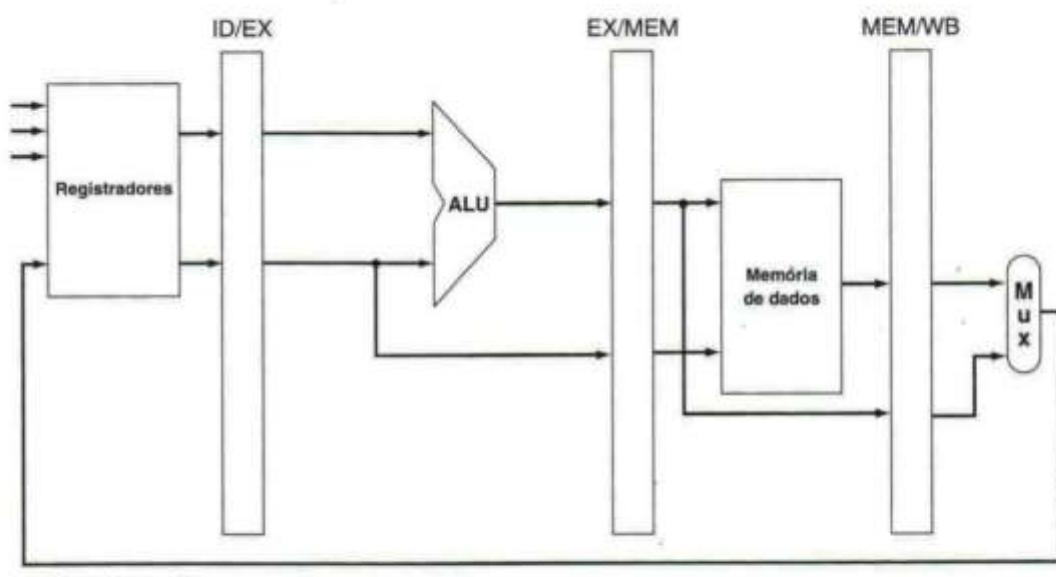
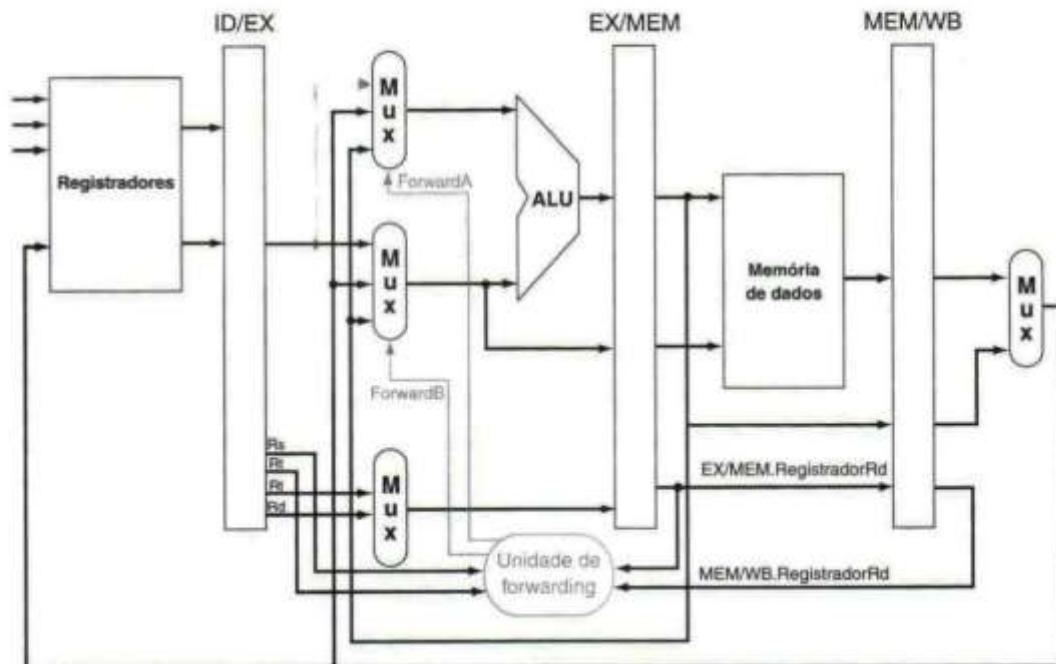


FIGURA 6.29 As dependências entre os registradores de pipeline se movem para a frente no tempo, de modo que é possível fornecer as entradas para a ALU necessárias pela instrução and e pela instrução or fazendo forwarding dos resultados encontrados nos registradores de pipeline. Os valores nos registradores de pipeline mostram que o valor desejado está disponível antes de ser escrito no banco de registradores. Consideraremos que o banco de registradores encaminha valores lidos e escritos durante o mesmo ciclo de clock, de modo que add não causa stall, mas os valores vêm do banco de registradores, e não de um registrador de pipeline. O “forwarding” do banco de registradores – ou seja, a leitura apanha o valor da escrita nesse ciclo de clock – é o motivo pelo qual o ciclo de clock 5 mostra o registrador \$2 tendo o valor 10 no inicio e -20 no final do ciclo de clock. Como no restante desta seção, tratamos de todo o forwarding, exceto para o valor a ser armazenado por uma instrução store.



a. Sem forwarding



b. Com forwarding

FIGURA 6.30 Acima estão a ALU e os registradores de pipeline antes da inclusão do forwarding. Abaixo, os multiplexadores foram expandidos para acrescentar os caminhos de forwarding, e mostramos a unidade de forwarding. O hardware novo aparece em um destaque. No entanto, essa figura é um desenho estilizado, omitindo os detalhes do caminho de dados completo, como o hardware de extensão de sinal. Observe que o campo ID/EX.RegistradorRt aparece duas vezes, uma para conectar ao mux e uma para a unidade de forwarding, mas esse é um único sinal. Como na discussão anterior, isso ignora o forwarding de um valor armazenado por uma instrução store.

Controle do Mux	Origem	Explicação
ForwardA = 00	ID/EX	O primeiro operando da ALU vem do banco de registradores.
ForwardA = 10	EX/MEM	O primeiro operando da ALU sofre forwarding do resultado anterior da ALU.
ForwardA = 01	MEM/WB	O primeiro operando da ALU sofre forwarding da memória de dados ou de um resultado anterior da ALU.
ForwardB = 00	ID/EX	O segundo operando da ALU vem do banco de registradores.
ForwardB = 10	EX/MEM	O segundo operando da ALU sofre forwarding do resultado anterior da ALU.
ForwardB = 01	MEM/WB	O segundo operando da ALU sofre forwarding da memória de dados ou de um resultado anterior da ALU.

FIGURA 6.31 Os valores de controle para os multiplexadores de forwarding da Figura 6.30. O imediato com sinal que é outra entrada da ALU é descrito na Seção "Detalhamento" ao final desta seção.

Agora, vamos escrever as duas condições para detectar hazards e os sinais de controle para resolvê-los:

1. Hazard EX:

```

if (EX/MEM.EscreveReg
and (EX/MEM.RegistradorRd ≠ 0)
and (EX/MEM.RegistradorRd = ID/EX.RegistradorRs)) ForwardA = 10

if (EX/MEM.EscreveReg
and (EX/MEM.RegistradorRd ≠ 0)
and (EX/MEM.RegistradorRd = ID/EX.RegistradorRt)) ForwardB = 10
    
```

Esse caso faz o forwarding do resultado da instrução anterior para qualquer entrada da ALU. Se a instrução anterior tiver de escrever no banco de registradores e o número do registrador de escrita combinar com o número do registrador de leitura das entradas A ou B da ALU, desde que não seja o registrador 0, então direcione o multiplexador para pegar o valor, e não do registrador de pipeline EX/MEM.

2. Hazard MEM:

```

if (MEM/WB.EscreveReg
and (MEM/WB.RegistradorRd ≠ 0)
and (MEM/WB.RegistradorRd = ID/EX.RegistradorRs)) ForwardA = 01
if (MEM/WB.EscreveReg
and (MEM/WB.RegistradorRd ≠ 0)
and (MEM/WB.RegistradorRd = ID/EX.RegistradorRt)) ForwardB = 01
    
```

Como dissemos, não existe hazard no estágio WB porque consideramos que o banco de registradores fornece o resultado correto se a instrução no estágio ID ler o mesmo registrador escrito pela instrução no estágio WB. Tal banco de registradores realiza outra forma de forwarding, mas isso ocorre dentro do banco de registradores.

Uma complicação são os hazards de dados em potencial entre o resultado da instrução no estágio WB, o resultado da instrução no estágio MEM e o operando de origem da instrução no estágio ALU. Por exemplo, ao somar um vetor de números em um único registrador, uma seqüência de instruções lerá e escreverá no mesmo registrador:

```

add $1, $1, $2
add $1, $1, $3
add $1, $1, $4
    *
    
```

Nesse caso, o resultado sofre forwarding do estágio MEM, pois o resultado no estágio MEM é o resultado mais recente. Assim, o controle para o hazard em MEM seria (com os acréscimos destacados)

```

if (MEM/WB.EscreveReg
and (MEM/WB.RegistradorRd ≠ 0)
and (EX/MEM.RegistradorRd ≠ ID/EX.RegistradorRs)
and (MEM/WB.RegistradorRd = ID/EX.RegistradorRs)) ForwardA = 01

if (MEM/WB.EscreveReg
and (MEM/WB.RegistradorRd ≠ 0)
and (EX/MEM.RegistradorRd ≠ ID/EX.RegistradorRt)
and (MEM/WB.RegistradorRd = ID/EX.RegistradorRt)) ForwardB = 01

```

*Forwarding
ne=1*

A Figura 6.32 mostra o hardware necessário para dar suporte ao forwarding para operações que utilizam resultados durante o estágio EX.

Detalhamento: o forwarding também pode ajudar com hazards quando instruções store dependem de outras instruções. Como elas utilizam apenas um valor de dados durante o estágio MEM, o forwarding é fácil. Mas considere os loads imediatamente seguidos por stores. Precisamos acrescentar mais hardware de forwarding para fazer com que as cópias de memória para memória se tornem mais rápidas. Se tivéssemos que redesenhar a Figura 6.29, substituindo as instruções sub e and por lw e sw, veríamos que é possível evitar um stall, pois os dados existem no registrador MEM/WB de uma instrução load em tempo para seu uso no estágio MEM de uma instrução store. Para essa opção, teríamos de acrescentar o forwarding para o estágio de acesso à memória. Deixamos essa modificação como um exercício.

Além disso, a entrada imediata com sinal para a ALU, necessária para loads e stores, não existe no caminho de dados da Figura 6.32. Como o controle central decide entre registrador e imediato, e como a unidade de forwarding escolhe o registrador de pipeline para uma entrada de registrador para a ALU, a solução mais fácil é acrescentar um multiplexador 2:1 que escolha entre a saída do multiplexador ForwardB e o imediato com sinal. A Figura 6.33 mostra esse acréscimo. Observe que essa solução é diferente daquela que aprendemos no Capítulo 5, onde o multiplexador controlado pela linha OrigALUB foi expandido para incluir a saída imediata.

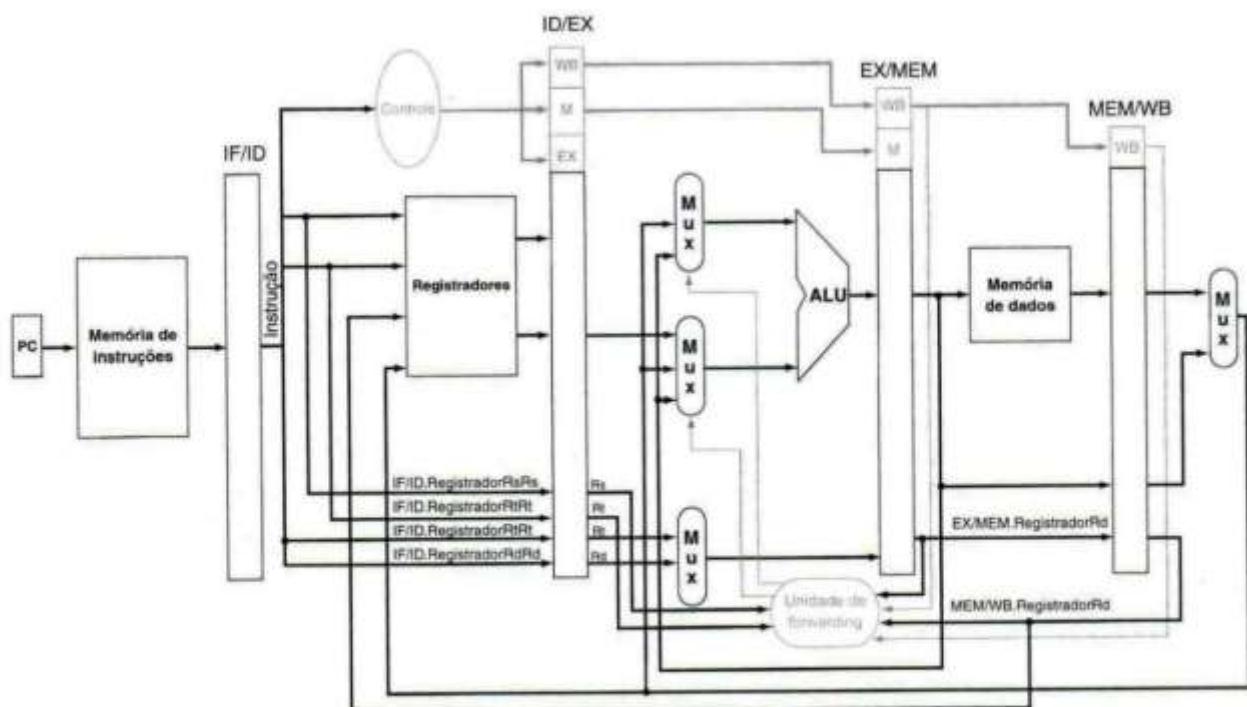


FIGURA 6.32 O caminho de dados modificado para resolver os hazards via forwarding. Em comparação com o caminho de dados da Figura 6.27, os acréscimos são os multiplexadores para as entradas da ALU. Contudo, essa figura é um desenho mais estilizado, omitindo detalhes do caminho de dados completo, como o hardware de desvio e o hardware de extensão de sinal.

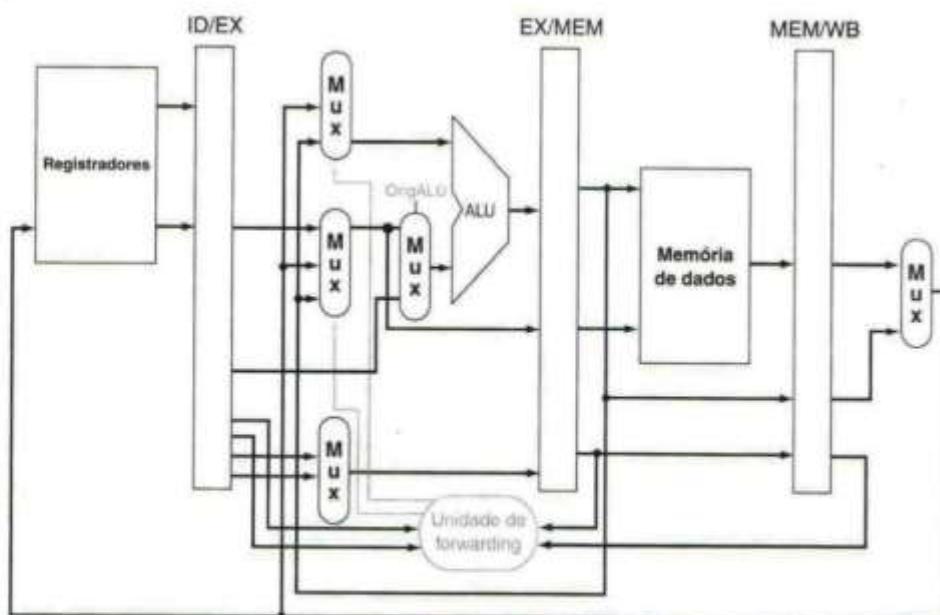


FIGURA 6.33 Uma visão de perto do caminho de dados da Figura 6.30 mostra um multiplexador 2:1, acrescentado para selecionar o imediato com sinal como uma entrada para a ALU.

6.5 Hazards de dados e stalls

Conforme dissemos na Seção 6.1, um caso em que o forwarding não pode salvar o dia é quando uma instrução tenta ler um registrador após uma instrução load que escreve no mesmo registrador. A Figura 6.34 ilustra o problema. Os dados ainda são lidos da memória no ciclo de clock 4, enquanto a ALU está realizando a operação para a instrução seguinte. Algo precisa ocorrer para evitar que a ALU realize a operação incorreta. Para isso, precisamos de uma unidade de detecção de hazard.

Logo, além de uma unidade de forwarding, precisamos de uma *unidade de detecção de hazard*. Ela opera durante o estágio ID, de modo que pode inserir um stall entre o load e seu uso. Verificando as instruções load, o controle para a unidade de detecção de hazard é esta condição única:

```
if (ID/EX.LeMem and
    ((ID/EX.RegistradorRt = IF/ID.RegistradorRs) or
     (ID/EX.RegistradorRt = IF/ID.RegistradorRt)))
    ocasiona stall no pipeline
```

A primeira linha testa se a instrução é um load: a única instrução que lê a memória de dados é um load. As duas linhas seguintes verificam se o campo do registrador destino do load no estágio EX combina com qualquer registrador origem da instrução no estágio ID. Se a condição permanecer, a instrução ocasiona um stall de 1 ciclo de clock no pipeline. Depois desse stall de 1 ciclo, a lógica de forwarding pode lidar com a dependência e a execução prossegue. (Se não houvesse forwarding, então as instruções na Figura 6.34 precisariam de outro ciclo de stall.)

Se a instrução no estágio ID sofrer um stall, então a instrução no estágio IF também precisa sofrer; caso contrário, perderíamos a instrução lida da memória. Evitar que essas duas instruções tenham progresso é algo feito simplesmente impedindo-se que o registrador PC e o registrador de pipeline IF/ID sejam alterados. Desde que esses registradores sejam preservados, a instrução no estágio IF continuará a ser lida usando o mesmo PC, e os registradores no estágio ID continuarão a ser lidos usando os mesmos campos de instrução no registrador de pipeline IF/ID. Retornando à nossa analogia favorita, é como se você reiniciasse a lavadora com as mesmas roupas e deixasse a secadora con-

Se você não tiver sucesso a princípio, redefina o sucesso.

Anônimo

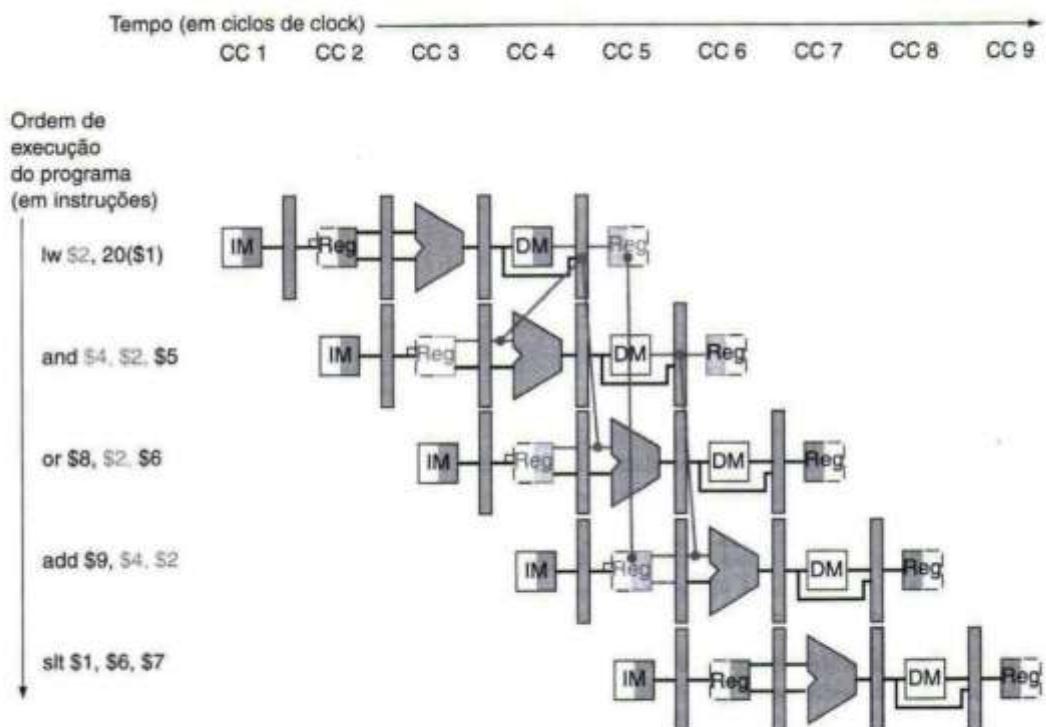


FIGURA 6.34 Uma seqüência de instruções em pipeline. Como a dependência entre o load e a instrução seguinte (and) recua no tempo, esse hazard não pode ser resolvido pelo forwarding. Logo, essa combinação precisa resultar em um stall pela unidade de detecção de hazard.

nop Uma instrução que não realiza operação alguma para mudar de estado. Continuando a trabalhar vazia. Naturalmente, assim como a secadora, a metade do pipeline que começa com o estágio EX precisa estar fazendo alguma coisa; o que ela está fazendo é executar instruções que não têm efeito algum: **nops**.

Como podemos inserir esses nops, que atuam como bolhas, no pipeline? Na Figura 6.25, vimos que a desativação de todos os nove sinais de controle (colocando-os em 0) nos estágios EX, MEM e WB criará uma instrução que “não faz nada”, ou **nop**. Identificando o hazard no estágio ID, podemos inserir uma bolha no pipeline alterando os campos de controle EX, MEM e WB do registrador de pipeline ID/EX para 0. Esses valores de controle benignos são filtrados adiante em cada ciclo de clock com o efeito correto: nenhum registrador ou memória serão modificados se os valores forem todos 0.

A Figura 6.35 mostra o que realmente acontece no hardware: a execução do slot do pipeline associado com a instrução **and** transforma-se em um **nop** e todas as instruções começando com a instrução **and** são atrasadas um ciclo. O hazard força as instruções **and** e **or** a repetirem no ciclo de clock 4 o que elas fizeram no ciclo de clock 3: and lê e decodifica registradores, enquanto or é buscado novamente na memória de instruções. Esse trabalho repetido é o próprio stall, mas seu efeito é esticar o tempo das instruções **and** e **or** e atrasar a busca da instrução **add**. Assim como uma bolha de ar em um cano de água, uma bolha de stall retarda tudo o que está atrás dela e prossegue pelo pipe de instruções um estágio a cada ciclo, até que saia no final.

A Figura 6.36 destaca as conexões do pipeline para a unidade de detecção de hazard e a unidade de forwarding. Como antes, a unidade de forwarding controla os multiplexadores da ALU para substituir o valor de um registrador de uso geral pelo valor do registrador de pipeline apropriado. A unidade de detecção de hazard controla a escrita dos registradores PC e IF/ID mais o multiplexador que escolhe entre os valores de controle reais e 0s. A unidade de detecção de hazard insere um stall e desativa os campos de controle se o teste de hazard do uso do load for verdadeiro. Mostramos os diagramas com ciclo único de clock na Seção ■ “Aprofundando o aprendizado”, no CD.

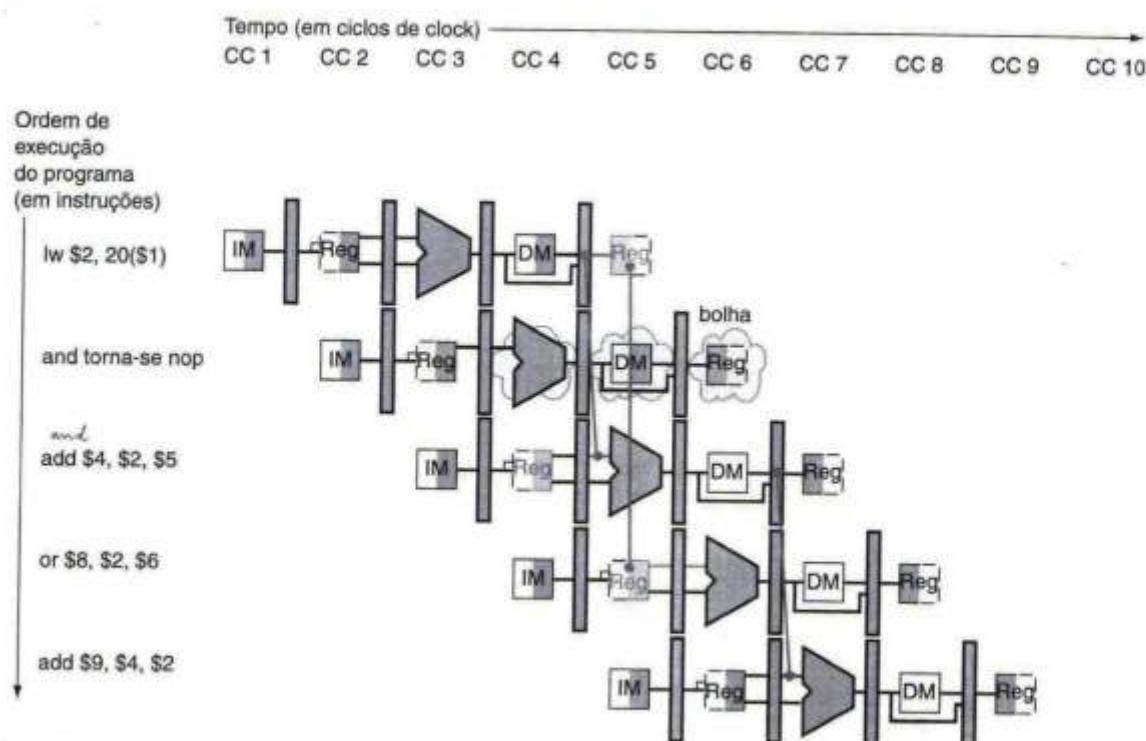


FIGURA 6.35 O modo como os stalls são realmente inseridos no pipeline. Uma bolha é inserida a partir do ciclo de clock 4, alterando a instrução AND para um NOP. Observe que a instrução AND na realidade é buscada e decodificada nos ciclos de clock 2 e 3, mas seu estágio EX é atrasado até o ciclo de clock 5 (ao contrário da posição sem stall no ciclo de clock 4). Da mesma forma, a instrução OR é apanhada no ciclo de clock 3, mas seu estágio IF é atrasado até o ciclo de clock 5 (ao contrário da posição não atrasada no ciclo de clock 4). Após a inserção da bolha, todas as dependências seguem à frente no tempo e nenhum outro hazard acontece.

Embora o hardware possa ou não contar com o compilador para resolver dependências de hazard, para garantir a execução correta, o compilador precisa compreender o pipeline para alcançar o melhor desempenho. Caso contrário, stalls inesperados reduzirão o desempenho do código compilado.

Colocando em perspectiva

Detalhamento: com relação ao comentário anterior sobre a colocação das linhas de controle em 0 para evitar a escrita de registradores ou memória: somente os sinais EscreveReg e EscreveMem precisam ser 0, enquanto os outros sinais de controle podem ser don't care.

6.6

Hazards de desvio

Até aqui, limitamos nossa preocupação aos hazards envolvendo operações aritméticas e transferências de dados. Entretanto, como vimos na Seção 6.1, também existem hazards de pipeline envolvendo desvios. A Figura 6.37 mostra uma sequência de instruções e indica quando o desvio ocorreria nesse pipeline. Uma instrução precisa ser buscada a cada ciclo de clock para sustentar o pipeline, embora, em nosso projeto, a decisão sobre o desvio não ocorra até o estágio MEM do pipeline. Conforme mencionamos na Seção 6.1, esse atraso para determinar a instrução própria a ser buscada é chamado de *hazard de controle* ou *hazard de desvio*, ao contrário dos *hazards de dados*, que acabamos de examinar.

Existem milhares pendurados nos galhos do mal para cada um que está batendo na raiz.

Henry David Thoreau, *Walden*, 1854

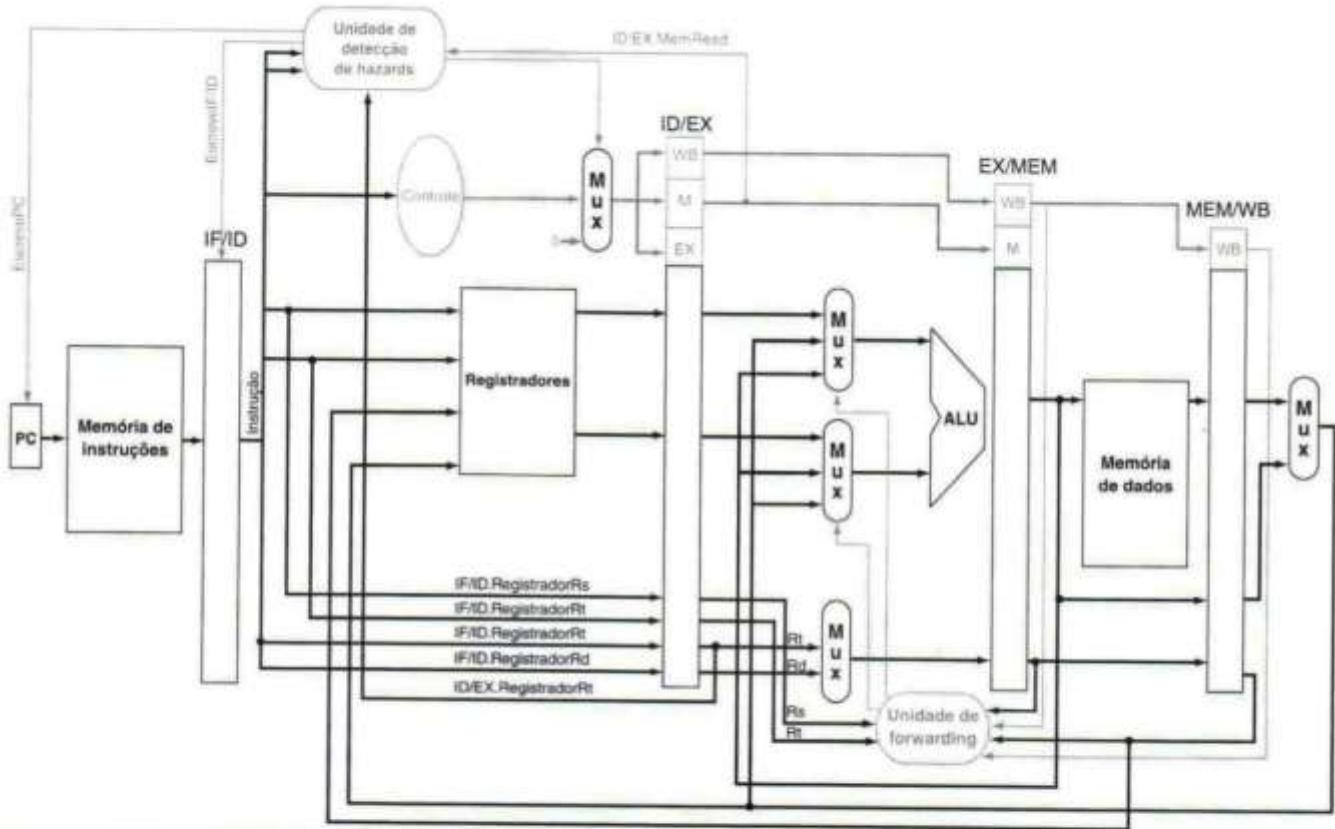


FIGURA 6.36 Visão geral do controle em pipeline, mostrando os dois multiplexadores para forwarding, a unidade de detecção de hazard e a unidade de forwarding. Embora os estágios ID e EX tenham sido simplificados – a lógica de extensão de sinal do imediato e de desvio estão faltando –, este desenho mostra a essência dos requisitos do hardware de forwarding.

Esta seção sobre hazards de controle é mais curta do que as seções anteriores, sobre hazards de dados. Os motivos são que os hazards de controle são relativamente simples de entender, eles ocorrem com menos freqüência que os hazards de dados, e não há nada tão eficiente contra os hazards de controle quanto o forwarding para os hazards de dados. Logo, usamos esquemas mais simples. Veremos dois esquemas para resolver os hazards de controle e uma otimização para melhorar esses esquemas.

Considere que o desvio não foi tomado

Como vimos na Seção 6.1, fazer um stall até que o desvio termine é muito lento. Uma melhoria comum ao stall do desvio é considerar que o desvio não será tomado e, portanto, continuar no fluxo sequencial das instruções. Se o desvio for tomado, as instruções que estão sendo buscadas e decodificadas precisam ser descartadas. A execução continua no destino do desvio. Se os desvios não são tomados na metade das vezes, e se custar pouco descartar as instruções, essa otimização reduz ao meio o custo dos hazards de controle.

Para descartar instruções, simplesmente alteramos os valores de controle para 0, assim como fizemos para o stall no hazard de dados para o caso do load. A diferença é que também precisamos alterar as três instruções nos estágios IF, ID e EX quando o desvio atingir o estágio MEM; para os stalls no uso de load, simplesmente alteramos o controle para 0 no estágio ID e o deixamos prosseguir no pipeline. Descartar instruções, então, significa que precisamos ser capazes de dar **flush nas instruções** nos estágios IF, ID e EX do pipeline.

flush (instruções)

Descartar instruções em um pipeline, normalmente devido a um evento inesperado.

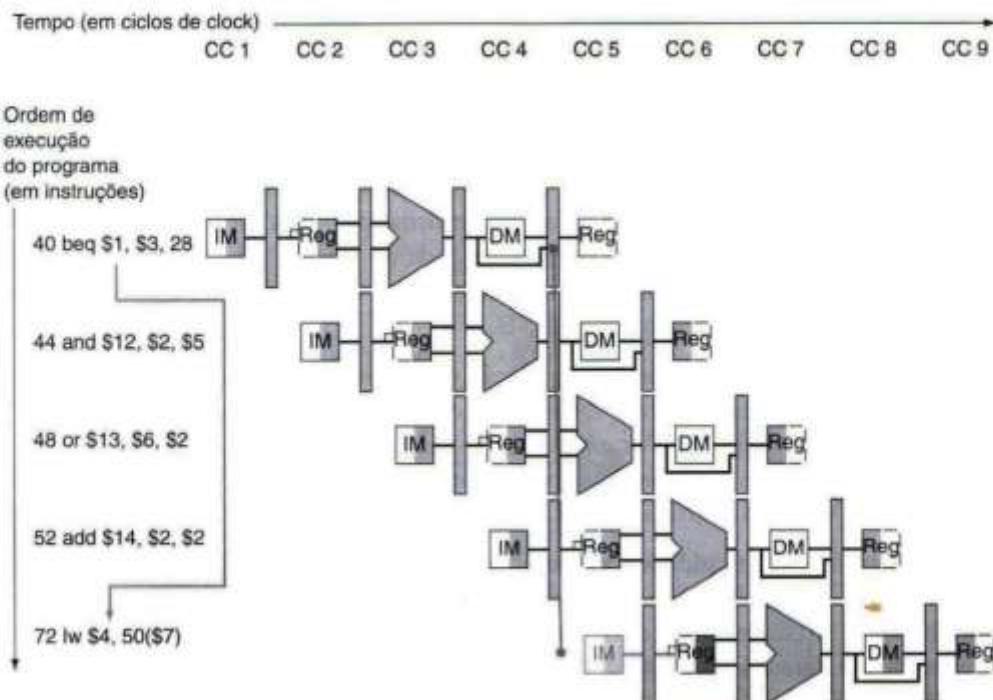


FIGURA 6.37 O impacto do pipeline sobre a instrução branch. Os números à esquerda da instrução (40, 44, ...) são os endereços das instruções. Como a instrução branch decide se deve desviar no estágio MEM – ciclo de clock 4 para a instrução beq, anterior –, as três instruções seqüenciais que seguem o branch serão buscadas e iniciarão sua execução. Sem intervenção, essas três instruções seguirão a executar antes que o beq desvie para lw na posição 72. (A Figura 6.7 considerou um hardware extra para reduzir o hazard de controle a 1 ciclo de clock; essa figura usa o caminho de dados não otimizado.)

Reduzindo o atraso dos desvios

Uma forma de melhorar o desempenho do desvio é reduzir o custo do desvio tomado. Até aqui, consideramos que o próximo PC para um desvio é selecionado no estágio MEM, mas, se movermos a execução do desvio para um estágio anterior do pipeline, então menos instruções precisam sofrer flush. A arquitetura do MIPS foi criada para dar suporte a desvios rápidos de ciclo único, que poderiam passar pelo pipeline com uma pequena penalidade no desvio. Os projetistas observaram que muitos desvios contam apenas com testes simples (igualdade ou sinal, por exemplo) e que esses testes não exigem uma operação completa da ALU, mas podem ser feitos com no máximo algumas portas lógicas. Quando uma decisão de desvio mais complexa é exigida, uma instrução separada, que usa uma ALU para realizar uma comparação, é requisitada – uma situação semelhante ao uso de códigos de condição para os desvios.

Levar a decisão do desvio para cima exige que duas ações ocorram mais cedo: calcular o endereço de destino do desvio e avaliar a decisão do desvio. A parte fácil dessa mudança é subir com o cálculo do endereço de desvio. Já temos o valor do PC e o campo imediato no registrador de pipeline IF/ID, de modo que só movemos o somador do desvio do estágio EX para o estágio ID; naturalmente, o cálculo do endereço de destino do desvio será realizado para todas as instruções, mas só será usado quando for necessário.

A parte mais difícil é a própria decisão do desvio. Para branch equal, comparariamos os dois registradores lidos durante o estágio ID para ver se são iguais. A igualdade pode ser testada primeiro realizando um OR exclusivo de seus respectivos bits e depois um OR de todos os resultados. Mover o teste de desvio para o estágio ID implica hardware adicional de forwarding e detecção de hazard, visto que um desvio dependente de um resultado ainda no pipeline ainda precisará funcionar corretamente com essa optimização. Por exemplo, para implementar branch-on-equal (e seu inverso), tere-

mos de fazer um forwarding dos resultados para a lógica do teste de igualdade que opera durante o estágio ID. Existem dois fatores que complicam as coisas:

1. Durante o estágio ID, temos de decodificar a instrução, decidir se um bypass para a unidade de igualdade é necessário e completar a comparação de igualdade de modo que, se a instrução for um desvio, possamos atribuir ao PC o endereço de destino do desvio. O forwarding para os operandos dos desvios foi tratado anteriormente pela lógica de forwarding da ALU, mas a introdução da unidade de teste de igualdade no estágio ID exigirá nova lógica de forwarding. Observe que os operandos fonte de um desvio que sofreram bypass podem vir dos latches do pipeline ALU/MEM ou MEM/WB.
2. Como os valores em uma comparação de desvio são necessários durante o estágio ID, mas podem ser produzidos mais adiante no tempo, é possível que ocorra um hazard de dados e um stall seja necessário. Por exemplo, se uma instrução da ALU imediatamente antes de um desvio produz um dos operandos para a comparação no desvio, um stall será exigido, já que o estágio EX para a instrução da ALU ocorrerá depois do ciclo de ID do desvio.

Apesar dessas dificuldades, mover a execução do desvio para o estágio ID é uma melhoria, pois reduz a penalidade de um desvio a apenas uma instrução se o desvio for tomado, a saber, aquela sendo buscada atualmente. Os exercícios exploram os detalhes da implementação do caminho de forwarding e a detecção do hazard.

Para fazer um flush das instruções no estágio IF, acrescentamos uma linha de controle, chamada IF.Flush, que zera o campo de instrução do registrador de pipeline IF/ID. Apagar o registrador transforma a instrução buscada em um nop, uma instrução que não possui ação e não muda estado algum.

DESVIOS NO PIPELINE

EXEMPLO

Mostre o que acontece quando o desvio é tomado nesta seqüência de instruções, considerando que o pipeline está otimizado para desvios que não são tomados e que movemos a execução do desvio para o estágio ID:

```
36 sub $10, $4, $8
40 beq $1, $3, 7 # desvio relativo ao PC para 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)
```

RESPOSTA

A Figura 6.38 mostra o que acontece quando um desvio é tomado. Diferente da Figura 6.37, há somente uma bolha no pipeline para o desvio tomado.

Previsão dinâmica de desvios

Supor que um desvio não seja tomado é uma forma simples de *previsão de desvios*. Nesse caso, preveremos que os desvios não são tomados, fazendo um flush no pipeline quando estivermos errados. Para o pipeline simples, com cinco estágios, essa técnica, possivelmente acoplada com a previsão baseada no compilador, deverá ser adequada. Com pipelines mais profundos, a penalidade do desvio aumenta quando medida em ciclos de clock. Da mesma forma, com a questão múltipla, a penalidade do desvio aumenta em termos de instruções perdidas. Essa combinação significa que, em um pipeli-

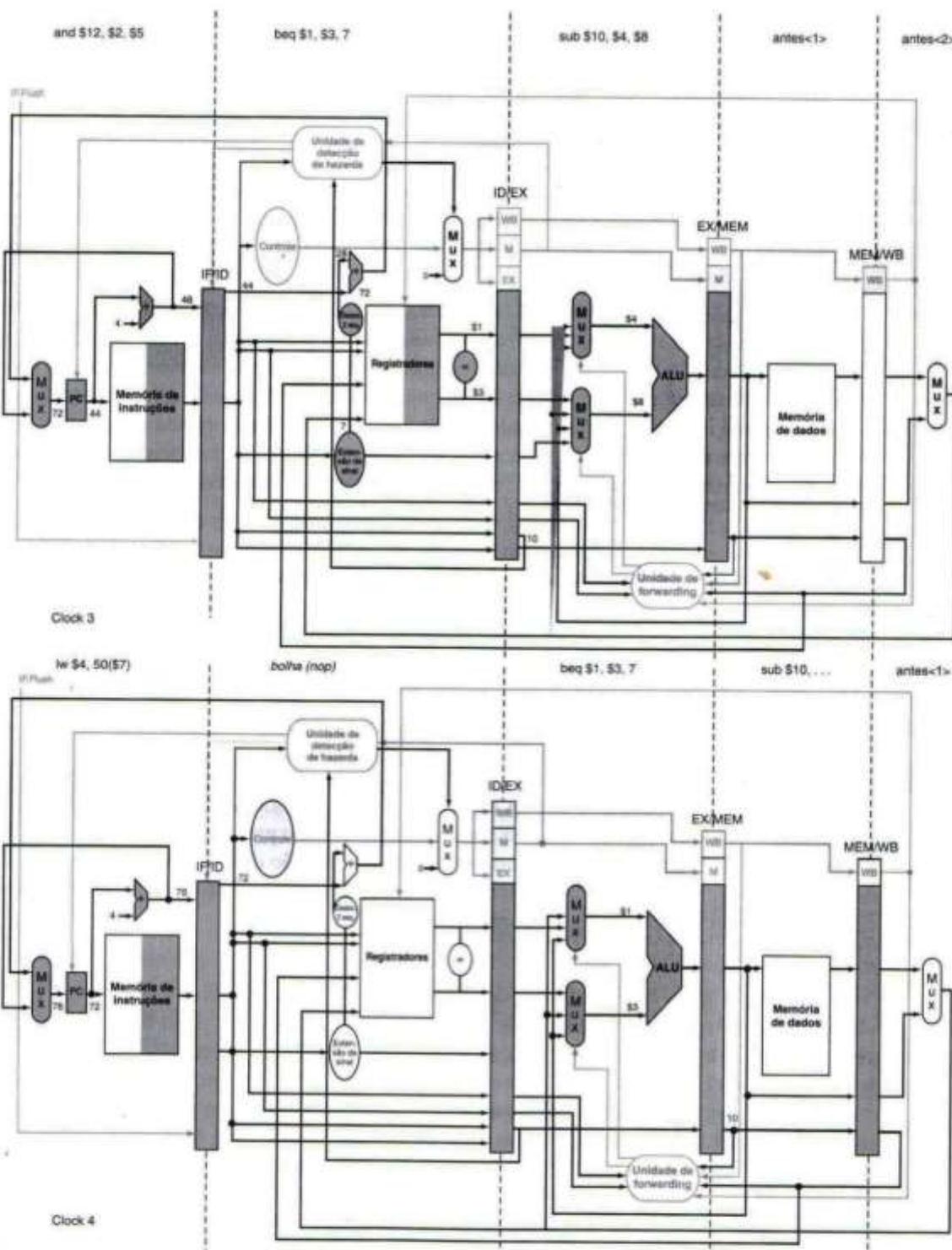


FIGURA 6.38 O estágio ID do ciclo de clock 3 determina que um desvio precisa ser tomado, de modo que seleciona 72 como próximo endereço do PC e zera a instrução buscada para o próximo ciclo de clock. O ciclo de clock 4 mostra a instrução no local 72 sendo buscada e a única bolha ou instrução nop no pipeline como resultado do desvio tomado. (Como o nop na realidade é \$11 \$0, \$0, 0, é discutível se o estágio ID no clock 4 deve ou não ser destacado.)

visão dinâmica de desvios Previsão de desvios durante a execução, usando informações em tempo de execução.

buffer de previsão de desvios Também chamado **tabela de histórico de desvios**. Uma pequena memória indexada pela parte menos significativa do endereço da instrução de desvio e que contém um ou mais bits indicando se o desvio foi tomado recentemente ou não.

ne agressivo, um esquema de previsão estática provavelmente desperdiçará muito desempenho. Como mencionamos na Seção 6.1, com mais hardware, é possível tentar prever o comportamento do desvio durante a execução do programa.

Uma técnica é pesquisar o endereço da instrução para ver se um desvio foi tomado na última vez que essa instrução foi executada e, se foi, começar a buscar novas instruções a partir do mesmo lugar da última vez. Essa técnica é chamada **visão dinâmica de desvios**.

Uma implementação dessa técnica é um **buffer de previsão de desvios**, ou **tabela de histórico de desvios**. Um buffer de previsão de desvios é uma pequena memória indexada pela parte menos significativa do endereço da instrução de desvio. A memória contém um bit que diz se o desvio foi tomado recentemente ou não.

Esse é o tipo de buffer mais simples; na verdade, não sabemos se a previsão é a correta – ela pode ter sido colocada lá por outro desvio, que tem os mesmos bits de endereço menos significativos. Mas isso não afeta a exatidão. A previsão é apenas um palpite considerado correto, de modo que a busca começa na direção prevista. Se o palpite estiver errado, as instruções previstas incorretamente são excluídas, o bit de previsão é invertido e armazenado de volta, e a seqüência apropriada é buscada e executada.

Esse esquema de previsão de 1 bit tem um problema de desempenho: mesmo que um desvio quase sempre seja tomado, provavelmente faremos uma previsão incorreta duas vezes, em vez de uma, quando ele não for tomado. O exemplo a seguir mostra esse dilema.

LOOPS E PREVISÃO

EXEMPLO

Considere um desvio de loop que se desvia nove vezes seguidas, depois não é tomado uma vez. Qual é a exatidão da previsão para esse desvio, supondo que o bit de previsão para o desvio permaneça no buffer de previsão?

RESPOSTA

O comportamento da previsão de estado fixo fará uma previsão errada na primeira e última iterações do loop. O erro de previsão na última iteração é inevitável, pois o bit de previsão dirá “tomado”; o desvio foi tomado nove vezes seguidas nesse ponto. O erro de previsão na primeira iteração acontece porque o bit é invertido na execução anterior da última iteração do loop, pois o desvio não foi tomado nessa iteração final. Assim, a exatidão da previsão para esse desvio tomado 90% do tempo é apenas de 80% (duas previsões incorretas contra oito corretas).

O ideal é que a previsão do sistema combine com a frequência de desvio tomado para esses desvios altamente regulares. Para remediar esse ponto fraco, os esquemas de previsão de 2 bits são utilizados com frequência. Em um esquema de 2 bits, uma previsão precisa estar errada duas vezes antes de ser alterada. A Figura 6.39 mostra a máquina de estados finitos para um esquema de previsão de 2 bits.

Um buffer de previsão de desvio pode ser implementado como um pequeno buffer especial, acessado com o endereço da instrução durante o estágio do pipe IF. Se a instrução for prevista como tomada, a busca começa a partir do destino assim que o PC for conhecido; isso pode ser até mesmo no estágio ID. Caso contrário, a busca e a execução seqüencial continuam. Se a previsão for errada, os bits de previsão são trocados, como mostra a Figura 6.39.

Detalhamento: conforme descrevemos na Seção 6.1, em um pipeline de cinco estágios, podemos tornar o hazard de controle em um recurso, redefinindo o desvio. Um delayed branch sempre executa a seguinte instrução, mas a segunda instrução após o desvio será afetada pelo desvio.

Os compiladores e os montadores tentam colocar uma instrução que sempre executa após o desvio no **delay slot do desvio**. A tarefa do software é tornar as instruções sucessoras válidas e úteis. A Figura 6.40 mostra as três maneiras como o delay slot do desvio pode ser escalonado.

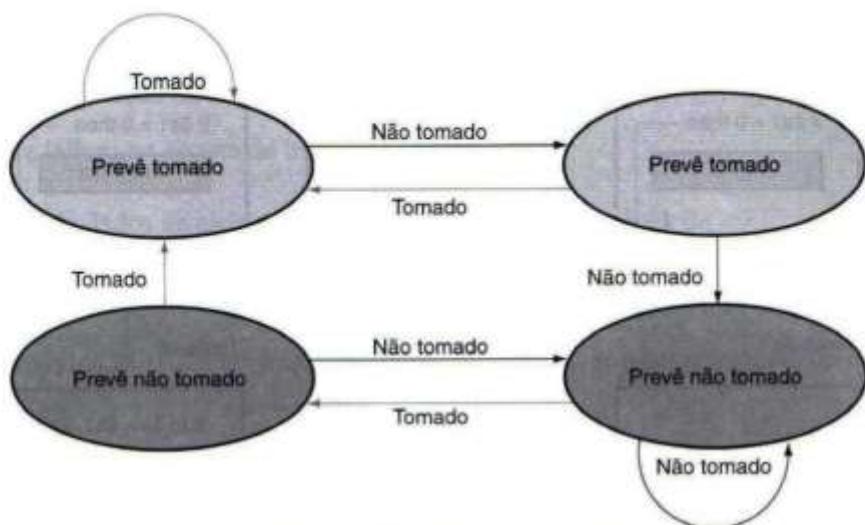


FIGURA 6.39 Os estados em um esquema de previsão de 2 bits. Usando 2 bits em vez de 1, um desvio que favoreça bastante a situação “tomado” ou “não tomado” – como muitos desvios fazem – será previsto incorretamente apenas uma vez. Os 2 bits são usados para codificar os quatro estados no sistema. O esquema de dois bits é um caso geral de uma previsão baseada em contador, incrementado quando a previsão é exata e decrementado em caso contrário, e utiliza o ponto intermediário desse intervalo como divisão entre desvio tomado e não tomado.

As limitações sobre o escalonamento com delayed branch surgem de (1) as restrições sobre as instruções escalonadas nos delay slots e (2) nossa capacidade de prever durante a compilação se um desvio provavelmente será tomado ou não.

O delayed branch foi uma solução simples e eficaz para um pipeline de cinco estágios despachando uma instrução a cada ciclo de clock. À medida que os processadores utilizam pipelines maiores, despachando múltiplas instruções por ciclo de clock (ver Seção 6.9), o atraso do desvio torna-se maior e um único delay slot é insuficiente. Logo, o delayed branch perdeu popularidade em comparação com as técnicas mais dispendiosas, porém mais flexíveis. Simultaneamente, o crescimento em transistores disponíveis por chip tornou a previsão dinâmica relativamente mais barata.

Detalhamento: um previsor de desvios nos diz se um desvio é tomado ou não, mas ainda exige o cálculo do destino do desvio. No pipeline de cinco estágios, esse cálculo leva 1 ciclo, significando que os desvios tomados terão uma penalidade de 1 ciclo. Os delayed branches são uma técnica para eliminar essa penalidade. Outra técnica é usar uma cache para manter o contador de destino ou instrução de destino, usando um **buffer de destino de desvios**.

Detalhamento: o esquema de previsão dinâmica de 2 bits usa apenas informações sobre um determinado desvio. Os pesquisadores notaram que o uso de informações sobre um desvio local e um comportamento global de desvios executados recentemente, juntos, geram maior exatidão da previsão para o mesmo número de bits de previsão. Essas técnicas são chamadas de **previsor correlato**. Um previsor correlato simples poderia ter dois previsores de 2 bits para cada desvio, com a escolha entre os previsores feita com base em se o último desvio executado foi tomado ou não. Assim, o comportamento de desvio global pode ser imaginado como acrescentando bits de índice adicionais para a previsão.

Uma inovação mais recente na previsão de desvios é o uso de previsões de torneio. Um **previsor de torneio** utiliza vários previsores, acompanhando, para cada desvio, qual previsor gera os melhores resultados. Um previsor de torneio típico poderia conter duas previsões para cada índice de desvio: uma baseada em informações locais e uma baseada no comportamento do desvio global. Um seletor escolheria qual previsor usar para qualquer previsão dada. O seletor pode operar semelhantemente a um previsor de 1 ou 2 bits, favorecendo qualquer um dos dois previsores que tenha sido mais preciso. Muitos microprocessadores avançados mais recentes utilizam esses previsores rebuscados.

buffer de destino de desvios Uma estrutura que coloca em cache o PC de destino ou a instrução de destino para um desvio. Ele normalmente é organizado como uma cache com tags, tornando-o mais dispendioso do que um buffer de previsão simples.

previsor correlato Um previsor de desvio que combina o comportamento local de determinado desvio e informações globais sobre o comportamento de algum número recente de desvios executados.

previsor de torneio Um previsor de desvios com múltiplas previsões para cada desvio é um mecanismo de seleção que escolhe qual previsor deve ser usado para determinado desvio.

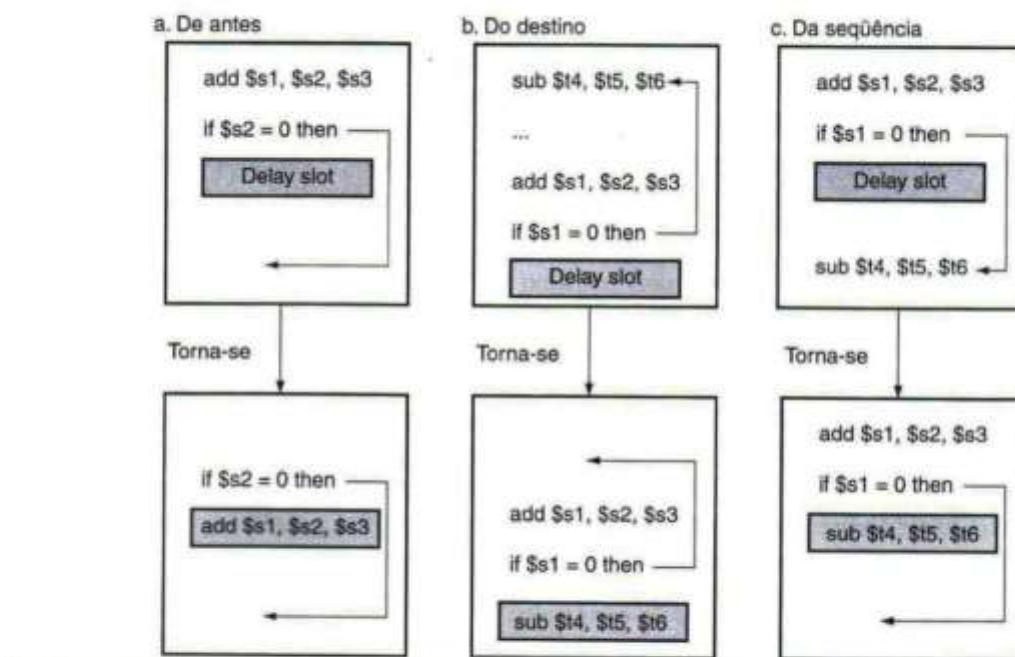


FIGURA 6.40 Escalonando o delay slot do desvio. Para cada par de quadros, o quadro de cima mostra o código antes do escalonamento; o quadro de baixo mostra o código escalonado. Em (a), o delay slot é escalonado com uma instrução independente de antes do desvio. Essa é a melhor opção. As estratégias (b) e (c) são usadas quando (a) não é possível. Nas seqüências de código para (b) e (c), o uso de $\$s1$ na condição de desvio impede que a instrução add (cujo destino é $\$s1$) seja movida para o delay slot do desvio. Em (b), o delay slot de desvio é escalonado a partir do destino do desvio; normalmente, a instrução de destino precisará ser copiada, pois pode ser alcançada por outro caminho. A estratégia (b) é preferida quando o desvio é tomado com alta probabilidade, como em um desvio de loop. Finalmente, o desvio pode ser escalonado a partir da seqüência não tomada, como em (c). Para tornar essa otimização válida para (b) ou (c), deve ser “OK” executar a instrução sub quando o desvio seguir na direção inesperada. Com “OK”, queremos dizer que o trabalho é desperdiçado, mas o programa ainda será executado corretamente. Esse é o caso, por exemplo, se $\$t4$ fosse um registrador temporário não utilizado quando o desvio entrasse na direção inesperada.

Resumo de pipeline

Até aqui, vimos três modelos de execução: ciclo único, múltiplos ciclos e pipeline. O controle em pipeline se esforça para conseguir 1 ciclo de clock por instrução, como o ciclo único, mas também por um ciclo de clock rápido, como nos múltiplos ciclos. Vamos revisar a comparação dos exemplos de processadores de ciclo único e múltiplos ciclos.

COMPARANDO O DESEMPENHO DE VÁRIOS ESQUEMAS DE CONTROLE

EXEMPLO

Compare o desempenho para os controle de ciclo único, múltiplos ciclos e em pipeline usando o mix de instruções SPECint2000 (ver exemplos nas páginas 237 e 249) e considerando os mesmos tempos de ciclo por unidade do exemplo da página 237. Para a execução em pipeline, considere que metade das instruções load é seguida imediatamente por uma instrução que usa o resultado, que o atraso do desvio na previsão errada é de 1 ciclo de clock, e que a quarta parte dos desvios tem previsão errada. Considere que os jumps sempre pagam um ciclo de clock inteiro de atraso, de modo que seu tempo médio é de 2 ciclos de clock. Ignore quaisquer outros hazards.

RESPOSTA

Pelo exemplo da página 237 (Desempenho das Máquinas de Ciclo Único), obtemos os seguintes tempos unitários funcionais:

- 200ps para acesso à memória
- 100ps para operação da ALU
- 50ps para leitura ou escrita do banco de registradores

Para o caminho de dados de ciclo único, isso leva a um ciclo de clock de

$$200 + 50 + 100 + 200 + 50 = 600\text{ps}$$

O exemplo na página 249 (CPI em uma CPU Multiciclo) tem as seguintes freqüências de instrução:

- 25% de loads
- 10% de stores
- 11% de branches
- 2% de jumps
- 52% de instruções da ALU

Além do mais, o exemplo na página 249 mostrou que o CPI para o projeto múltiplo foi de 4,12. O ciclo de clock para o caminho de dados multiciclo e o projeto em pipeline precisam ser iguais à unidade funcional mais longa: 200ps.

Para o projeto em pipeline, os loads exigem 1 ciclo de clock quando não existe dependência de uso de load e 2 quando ela existe. Logo, a média dos ciclos de clock por instrução load é de 1,5. Stores exigem 1 ciclo de clock, assim como as instruções da ALU. Os branches exigem 1 quando previstos corretamente e 2 quando isso não acontece, de modo que a média dos ciclos de clock por instrução de branch é 1,25. O CPI dos jumps é 2. Logo, o CPI médio é

$$1,5 \times 25\% + 1 \times 10\% + 1 \times 52\% + 1,25 \times 11\% + 2 \times 2\% = 1,17$$

Vamos comparar os três projetos pelo tempo médio de instrução. Para o projeto de ciclo único, ele é fixado em 600ps. Para o projeto multiciclo, ele é $200 \times 4,12 = 824\text{ps}$. Para o projeto em pipeline, o tempo médio da instrução é de $1,17 \times 200 = 234\text{ps}$, tornando-o quase duas vezes mais rápido do que qualquer outra técnica.

O leitor inteligente notará que o tempo de ciclo longo da memória é um gargalo no desempenho para os projetos em pipeline e multiciclo. A divisão dos acessos à memória em dois ciclos de clock e, portanto, permitir que o ciclo de clock tenha 100ps melhoraria o desempenho nos dois casos. Vamos explorar isso nos exercícios.

Este capítulo começou na lavanderia, mostrando princípios da técnica de pipelining em um ambiente do dia-a-dia. Usando essa analogia como um guia, explicamos a técnica de pipelining de instruções passo a passo, começando com o caminho de dados de ciclo único e depois acrescentando registradores de pipeline, caminhos de forwarding, detecção de hazards, previsão de desvios e flush de instruções nas exceções. A Figura 6.41 mostra o caminho de dados e o controle final.

Considere três esquemas de previsão de desvios: desvio não tomado, previsão tomada e previsão dinâmica. Suponha que todos eles tenham penalidade zero quando prevêem corretamente e 2 ciclos quando estão errados. Suponha que a exatidão média da previsão do previsor dinâmico seja de 90%. Qual previsor é a melhor escolha para os seguintes desvios?

**Verifique
você mesmo**

1. Um desvio tomado com freqüência de 5%
2. Um desvio tomado com freqüência de 95%
3. Um desvio tomado com freqüência de 70%

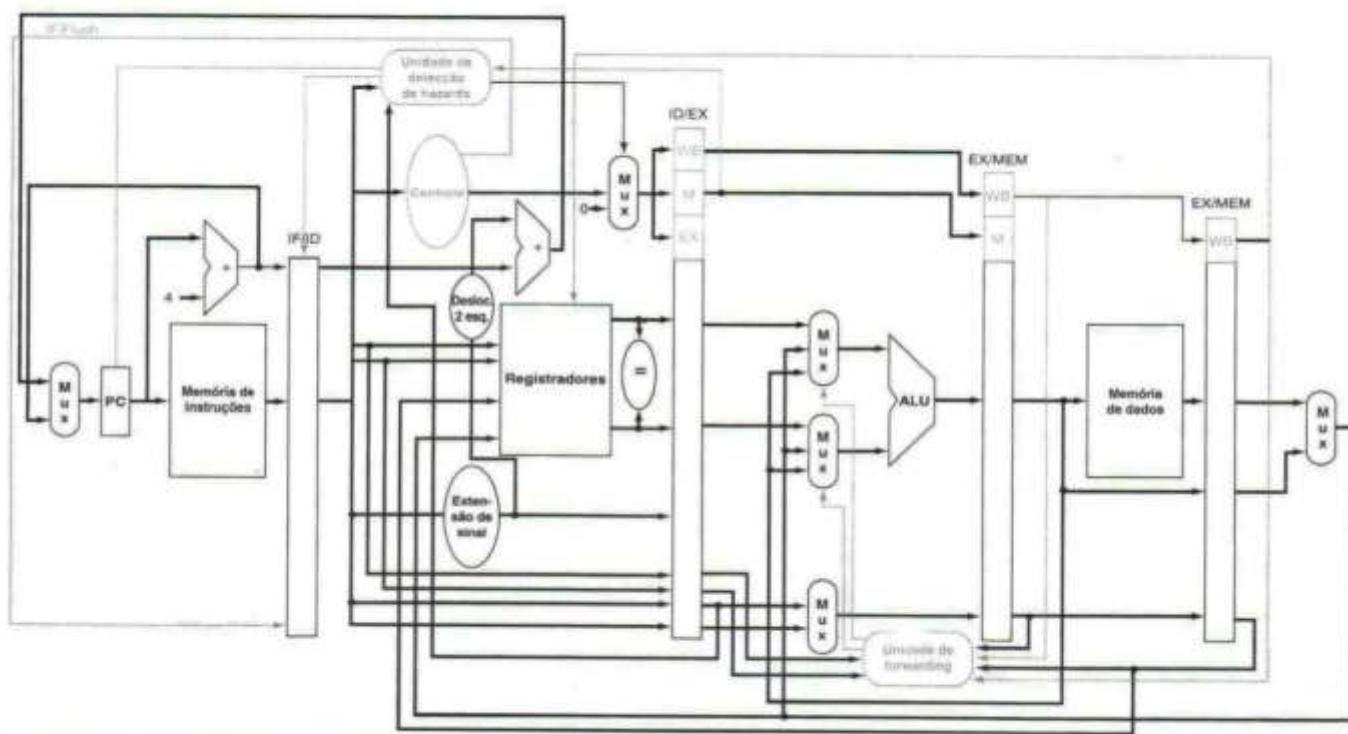


FIGURA 6.41 O caminho de dados e controle final para este capítulo.

6.7

Usando uma linguagem de descrição de hardware para descrever e modelar um pipeline

Fazer um computador com facilidades automáticas de interrupção de programa se comportar [seqüencialmente] não foi uma tarefa fácil, pois o número de instruções em diversos estágios do processamento quando um sinal de interrupção ocorre pode ser muito grande.

Fred Brooks Jr.,
Planning a Computer System:
Project Stretch,
1962

6.8

Exceções

Outra forma de hazard de controle envolve exceções. Por exemplo, suponha que a instrução a seguir

`add $1,$2,$1`

ocasione um overflow aritmético. Preferimos transferir o controle para a rotina de exceção imediatamente após essa instrução, pois não gostaríamos que esse valor inválido contaminasse outros registradores ou locais da memória.

Assim como fizemos para o desvio tomado na seção anterior, temos de dar flush nas instruções que vêm após a instrução add do pipeline e começar a buscar instruções do novo endereço. Usaremos o mesmo mecanismo que usamos para os desvios tomados, mas, dessa vez, a exceção causa a desativação das linhas de controle.

Quando lidamos com um desvio mal previsto, vimos como dar flush na instrução no estágio IF, transformando-a em um nop. Para dar flush nas instruções no estágio ID, usamos o multiplexador já presente no estágio ID que zera os sinais de controle para stalls. Um novo sinal de controle, chamado ID.Flush, realiza um OR com o sinal de stall da Unidade de Detecção de Hazards para dar flush du-

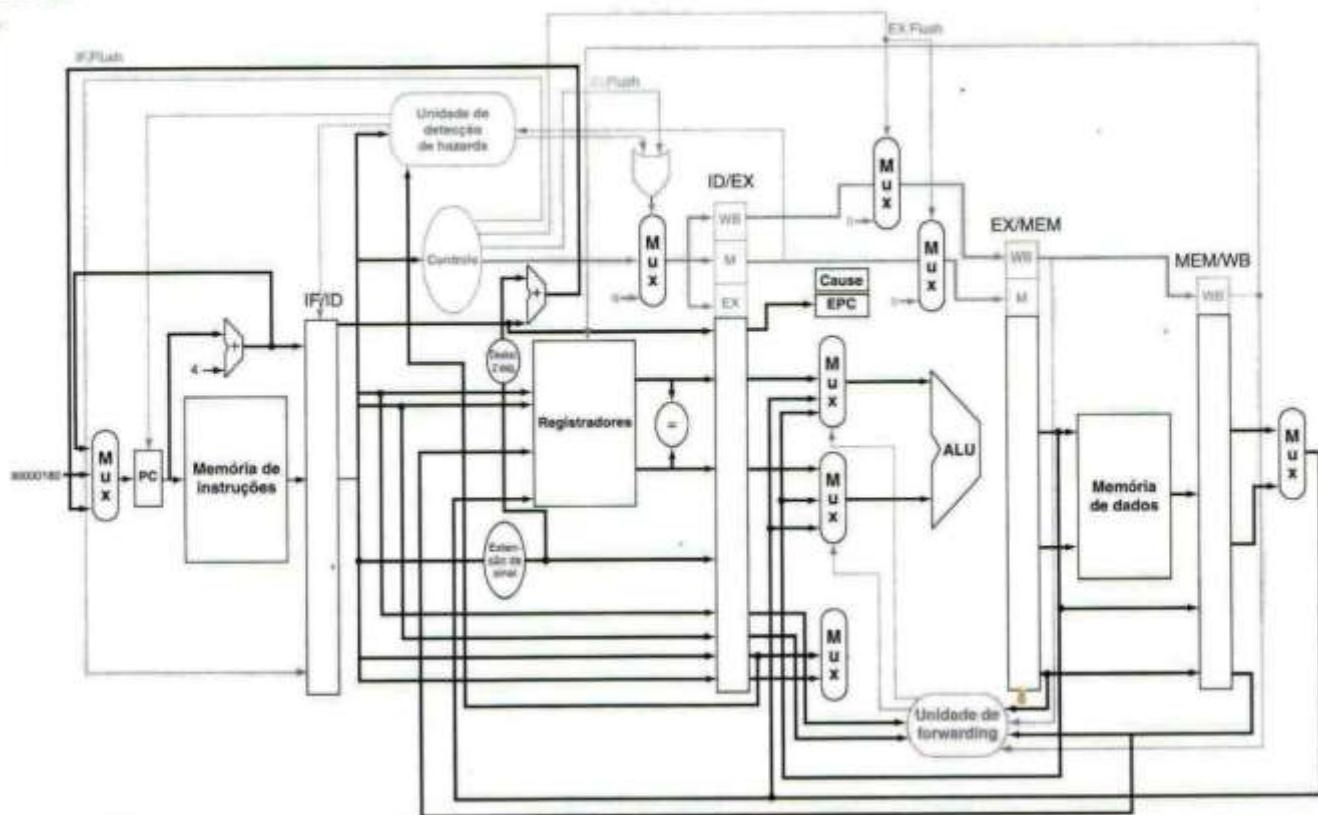


FIGURA 6.42 O caminho de dados com controles para lidar com exceções. Os principais acréscimos incluem uma nova entrada, com o valor 8000 0180_{hex}, no multiplexador que fornece o novo valor do PC; um registrador Cause para registrar a causa da exceção; e um registrador PC de Exceção (Exception Program Counter – EPC) para salvar o endereço da instrução que causou a exceção. A entrada 8000 0180_{hex} para o multiplexador é o endereço inicial para começar a buscar instruções no caso de uma exceção. Embora não apareça, o sinal de overflow da ALU é uma entrada para a unidade de controle.

rante o ID. Para dar flush na instrução em EX, usamos um novo sinal, chamado EX.Flush para fazer com que novos multiplexadores zerem as linhas de controle. Para começar a buscar instruções do local 8000 0180_{hex}, que é o local da exceção para o overflow aritmético, simplesmente acrescentamos uma entrada adicional ao multiplexador do PC, que envia 8000 0180_{hex} ao PC. A Figura 6.42 mostra essas mudanças.

Este exemplo aponta um problema com as exceções: se não paramos a execução no meio da instrução, o programador não poderá ver o valor original do registrador \$1 que ajudou a causar o overflow, pois funcionará como registrador de destino da instrução add. Devido ao planejamento cuidadoso, a exceção de overflow é detectada durante o estágio EX; logo, podemos usar o sinal EX.Flush para impedir que a instrução no estágio EX escreva seu resultado no estágio WB. Muitas exceções exigem que, por fim, completemos a instrução que causou a exceção como se ela fosse executada normalmente. O modo mais fácil de fazer isso é dar flush na instrução e reiniciá-la desde o inicio após a exceção ser tratada.

A etapa final é salvar o endereço da instrução problemática no Exception Program Counter (EPC), como fizemos no Capítulo 5. Na realidade, salvamos o endereço + 4, de modo que a rotina de tratamento da exceção primeiro deve subtrair 4 do valor salvo. A Figura 6.42 mostra uma versão estilizada do caminho de dados, incluindo o hardware de desvio e as acomodações necessárias para tratar das exceções.

EXCEÇÃO EM UM COMPUTADOR COM PIPELINE

EXEMPLO

Dada esta seqüência de instruções

```

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)

```

considere que as instruções a serem invocadas em uma exceção começem desta forma:

```

40000040hex sw $25, 1000($0)
40000044hex sw $26, 1004($0)

```

Mostre o que acontece no pipeline se houver uma exceção de overflow na instrução add.

RESPOSTA

A Figura 6.43 mostra os eventos, começando com a instrução add no estágio EX. O overflow é detectado durante essa fase, e 4000 0040_{hex} é forçado para o PC. O ciclo de clock 7 mostra que o add e as instruções seguintes sofrem flush, e a primeira instrução do código de exceção é buscada. Observe que o endereço da instrução *seguinte* ao add é salvo: 4C_{hex} + 4 = 50_{hex}.

O Capítulo 5 lista algumas outras causas de exceções:

- Solicitação de dispositivo de E/S
- Chamada de um serviço do sistema operacional a partir de um programa do usuário
- Uso de uma instrução indefinida
- Defeito do hardware

Com cinco instruções ativas em qualquer ciclo de clock, o desafio é associar uma exceção à instrução apropriada. Além do mais, várias exceções podem ocorrer simultaneamente em um único ciclo de clock. A solução normal é priorizar as exceções de modo que seja fácil determinar qual será atendida primeiro; essa estratégia também funciona para processadores em pipeline. Na maioria das implementações MIPS, o hardware ordena as exceções de modo que a instrução mais antiga seja interrompida.

Solicitações de dispositivos de E/S e defeitos do hardware não estão associados a uma instrução específica, de modo que a implementação possui alguma flexibilidade quanto ao momento de interromper o pipeline. Logo, usar o mecanismo utilizado para outras exceções funciona muito bem.

O EPC captura o endereço das instruções interrompidas, e o registrador Cause do MIPS registra todas as exceções possíveis em um ciclo de clock, de modo que o software de exceção precisa combinar a exceção à instrução. Uma dica importante é saber em que estágio do pipeline um tipo de exceção pode ocorrer. Por exemplo, uma instrução indefinida é descoberta no estágio ID, e a chamada ao sistema operacional ocorre no estágio EX. As exceções são coletadas no registrador Cause, de modo que o hardware possa interromper com base em exceções posteriores, uma vez que a mais antiga tenha sido atendida.

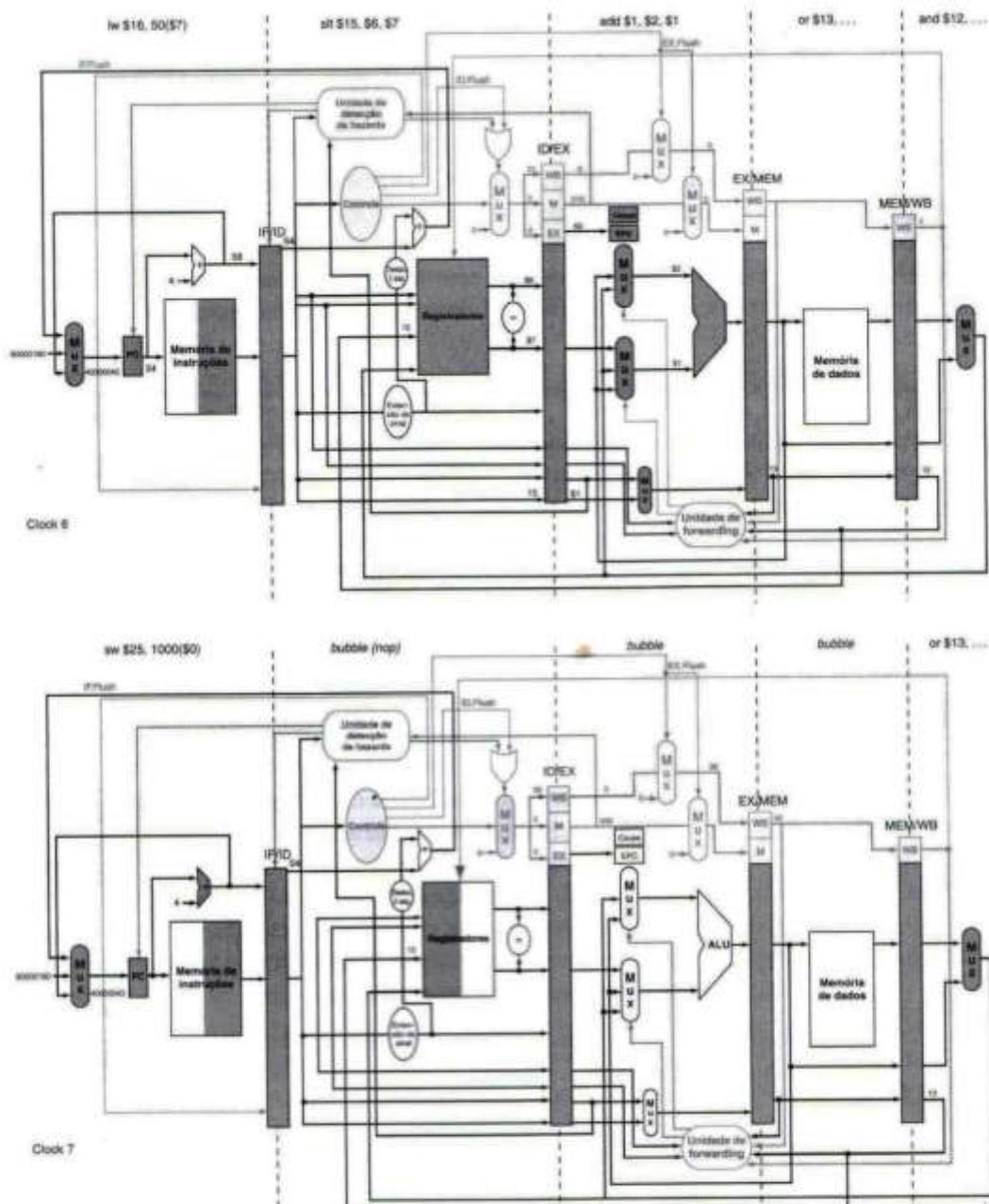


FIGURA 6.43 O resultado de uma exceção devido a um overflow aritmético na instrução add. O overflow é detectado durante o estágio EX do clock 6, salvando o endereço após o add no registrador EPC ($4C + 4 = 50_{hex}$). O overflow faz com que todos os sinais Flush sejam ativados perto do final desse ciclo de clock, desativando os valores de controle (colocando-os em 0) para o add. O ciclo de clock 7 mostra as instruções convertidas para bolhas no pipeline mais a busca da primeira instrução da rotina de exceção – `sw $25, 1000($0)` – a partir do local da instrução $4000\ 0040_{hex}$. Observe que as instruções `and` e `or`, que estão antes do add, ainda completam. Embora não apareça, o sinal de overflow da ALU é uma entrada para a unidade de controle.

Interface hardware/software

O hardware e o sistema operacional precisam trabalhar em conjunto para que as exceções se comportem conforme você esperaria. O contrato do hardware normalmente é interromper a instrução problemática no meio do caminho, deixar que todas as instruções anteriores terminem, dar flush em todas as instruções seguintes, definir um registrador para mostrar a causa da exceção, salvar o endereço da instrução problemática e depois desviar para um endereço previamente arranjado. O contrato

do sistema operacional é examinar a causa da exceção e atuar de forma apropriada. Para uma instrução indefinida, falha de hardware ou exceção por overflow aritmético, o sistema operacional normalmente encerra o programa e retorna um indicador do motivo. Para uma solicitação de dispositivo de E/S ou uma chamada de serviço ao sistema operacional, o sistema operacional salva o estado do programa, realiza a tarefa desejada e, em algum ponto no futuro, restaura o programa para continuar a execução. No caso das solicitações do dispositivo de E/S, normalmente podemos escolher executar outra tarefa antes de retomar a tarefa que requisitou a E/S, pois essa tarefa em geral pode não ser capaz de prosseguir até que a E/S termine. É por isso que a capacidade de salvar e restaurar o estado de qualquer tarefa é crítica. Um dos usos mais importantes e freqüentes das exceções é o tratamento de faltas de página e exceções de TLB; o Capítulo 7 descreve essas exceções e seu tratamento com mais detalhes.

Interrupção imprecisa
Também chamada **exceção imprecisa**. As interrupções ou exceções nos computadores em pipeline não estão associadas à instrução exata que foi a causa da interrupção ou exceção.

Interrupção precisa Também chamada **exceção precisa**. Uma interrupção ou exceção que está sempre associada à instrução correta nos computadores em pipeline.

A dificuldade de sempre associar a exceção correta à instrução correta nos computadores em pipeline levou alguns projetistas de computador a relaxarem esse requisito em casos não críticos. Alguns processadores são considerados como tendo **interrupções imprecisas** ou **exceções imprecisas**. No exemplo anterior, o PC normalmente teria 58_{hex} no início do ciclo de clock, depois que a exceção for detectada, embora a instrução com problema esteja no endereço 4C_{hex}. Um processador com exceções imprecisas poderia colocar 58_{hex} no EPC e deixar que o sistema operacional determinasse qual instrução causou o problema. O MIPS e a grande maioria dos computadores de hoje admitem **interrupções precisas** ou **exceções precisas**. (Um motivo é para dar suporte à memória virtual, que veremos no Capítulo 7.)

Verifique você mesmo

Os projetistas do MIPS queriam que as instruções de multiplicação e divisão de inteiros operassem em paralelo com outras instruções de inteiros. Como a multiplicação e a divisão exigem múltiplos ciclos de clock, um grupo de alunos está discutindo se é possível implementar exceções precisas. Quais dos seguintes argumentos são completamente precisos?

1. É impossível implementar exceções precisas, pois uma multiplicação ou divisão pode gerar uma exceção após as instruções que a seguem.
2. É trivial implementar exceções precisas, pois a multiplicação e a divisão não podem gerar uma exceção depois de iniciadas, e, por isso, a temporização de todas as exceções é obviamente precisa.
3. Não importa se a multiplicação ou a divisão podem gerar uma exceção. O fato de que ainda poderiam estar executando e não completadas quando alguma outra instrução gerasse uma exceção torna impossível implementar exceções precisas.
4. Embora seja verdade que uma multiplicação ou divisão ainda poderia estar executando, é certo que ela terminará em breve, e quando isso acontecer, qualquer exceção gerada para uma instrução após uma multiplicação ou divisão será precisa.

6.9

Pipelining avançado: extraíndo mais desempenho

Esteja avisado de que as Seções 6.9 e 6.10 são apenas introduções de assuntos fascinantes, porém avançados. Se você quiser saber mais detalhes, deverá consultar nosso livro mais avançado, *Arquitetura de Computadores: Uma abordagem quantitativa*, terceira edição, no qual o material explicado nas próximas páginas é expandido para mais de 200 páginas!

A técnica de pipelining explora o paralelismo em potencial entre as instruções. Esse paralelismo é chamado de **paralelismo em nível de instrução** (ILP – Instruction-Level Parallelism). Existem dois métodos principais para aumentar a quantidade em potencial de paralelismo em nível de instrução. O primeiro é aumentar a profundidade do pipeline para sobrepor mais instruções. Usando nossa analogia da lavanderia e considerando que o ciclo da lavadora fosse maior do que os outros, poderíamos dividir nossa lavadora em três máquinas que lavam, enxágam e centrifugam, como as etapas de uma lavadora tradicional. Poderíamos, então, passar de um pipeline de quatro para seis estágios. Para ganhar o máximo de velocidade, precisamos rebalancear as etapas restantes de modo que tenham o mesmo tamanho, nos processadores ou na lavanderia. A quantidade de paralelismo sendo explorada é maior, pois existem mais operações sendo sobrepostas. O desempenho é potencialmente maior, pois o ciclo de clock pode ser encurtado.

Outra técnica é replicar os componentes internos do computador de modo que ele possa iniciar várias instruções em cada estágio do pipeline. O nome geral para essa técnica é **despacho múltiplo**. Uma lavanderia com despacho múltiplo substituiria nossa lavadora e secadora doméstica por, digamos, três lavadoras e três secadoras. Você também teria de recrutar mais auxiliares para passar e guardar três vezes a quantidade de roupas no mesmo período. A desvantagem é o trabalho extra para manter todas as máquinas ocupadas e transferir as trouxas de roupa para o próximo estágio do pipeline.

Disparar várias instruções por estágio permite que a velocidade de execução da instrução exceda a velocidade de clock ou, de forma alternativa, que o CPI seja menor do que 1. Às vezes, é útil inverter a métrica e usar o *IPC*, ou *instrução por ciclo de clock*, principalmente quando os valores se tornam menores do que 1! Logo, um microprocessador de despacho múltiplo quádruplo de 6GHz pode executar uma velocidade de pico de 24 bilhões de instruções por segundo e ter um CPI de 0,25 no melhor dos casos, ou um IPC de 4. Considerando um pipeline de cinco estágios, esse processador teria 20 instruções em execução em determinado momento. Os microprocessadores mais potentes de hoje tentam despachar de três a oito instruções a cada ciclo de clock. Entretanto, normalmente existem muitas restrições sobre os tipos das instruções que podem ser executadas simultaneamente e o que acontece quando surgem dependências.

Existem duas maneiras importantes de implementar um processador de despacho múltiplo, com a principal diferença sendo a divisão de trabalho entre o compilador e o hardware. Como a divisão do trabalho indica se as decisões estão sendo feitas estaticamente (ou seja, durante a compilação) ou dinamicamente (ou seja, durante a execução), as técnicas às vezes são chamadas de **despacho múltiplo estático** e **despacho múltiplo dinâmico**. Como veremos, as duas técnicas possuem outros nomes, usados mais comumente, que podem ser menos precisos ou mais reticutivos.

Existem duas responsabilidades principais e distintas que precisam ser lidadas em um pipeline de despacho múltiplo:

1. Empacotar as instruções em **slots de despacho**: como o processador determina quantas instruções e quais instruções podem ser despachadas em determinado ciclo de clock? Na maioria dos processadores de despacho estático, esse processo é tratado pelo menos parcialmente pelo compilador; nos projetos de despacho dinâmico, isso normalmente é tratado durante a execução pelo processador, embora o compilador em geral já tenha tentado ajudar a melhorar a velocidade do despacho colocando as instruções em uma ordem benéfica.
2. Lidar com hazards de dados e de controle: em processadores de despacho estático, algumas ou todas as consequências dos hazards de dados e controle são tratadas estaticamente pelo compilador. Ao contrário, a maioria dos processadores de despacho dinâmico tenta aliviar pelo menos algumas classes de hazards usando técnicas de hardware operando durante a execução.

Embora as tenhamos descrito como técnicas distintas, na realidade, cada técnica pega algo emprestado da outra, e nenhuma pode afirmar ser perfeitamente pura.

paralelismo em nível de instrução O paralelismo entre as instruções.

despacho múltiplo Um esquema pelo qual múltiplas instruções são disparadas em 1 ciclo de clock.

despacho múltiplo estático Uma técnica para implementar um processador de despacho múltiplo em que muitas decisões são tomadas pelo compilador antes da execução.

despacho múltiplo dinâmico Uma técnica para implementar um processador de despacho múltiplo em que muitas decisões são tomadas durante a execução pelo processador.

slots de despacho As posições das quais as instruções poderiam ser despachadas em determinado ciclo de clock; por analogia, correspondem a posições nos blocos iniciais para um sprint.

O conceito de especulação

especulação Uma técnica pela qual o compilador ou processador adivinha o resultado de uma instrução para removê-la como uma dependência na execução de outras instruções.

Um dos métodos mais importantes para localizar e explorar mais ILP é a especulação. **Especulação** é uma técnica que permite que o compilador ou o processador “adivinhem” as propriedades de uma instrução, de modo a permitir que a execução comece para outras instruções que possam depender da instrução especulada. Por exemplo, poderíamos especular a respeito do resultado de um desvio, de modo que as instruções após o desvio pudessem ser executadas mais cedo. Ou então, poderíamos especular que um store que precede um load não se refere ao mesmo endereço, o que permitiria que o load fosse executado antes do store. A dificuldade com a especulação é que ela pode estar errada. Assim, qualquer mecanismo de especulação deve incluir tanto um método para verificar se a escolha foi certa quanto um método para retornar ou retroceder os efeitos das instruções executadas de forma especulativa. A implementação dessa capacidade de retrocesso aumenta a complexidade de qualquer processador que ofereça suporte à especulação.

A especulação pode ser feita pelo compilador ou pelo hardware. Por exemplo, o compilador pode usar a especulação para reordenar as instruções, fazendo uma instrução passar por um desvio ou um load passar por um store. O hardware do processador pode realizar a mesma transformação durante a execução, usando técnicas que discutiremos mais adiante nesta seção.

Os mecanismos de recuperação usados para a especulação incorreta são bem diferentes. No caso da especulação em software, o compilador normalmente insere instruções adicionais que verificam a precisão da especulação e oferecem uma rotina de reparo para usar quando a especulação tiver sido incorreta. Na especulação em hardware, o processador normalmente coloca os resultados especulativos em um buffer até que saiba que não são mais especulativos. Se a especulação foi correta, as instruções são concluídas, permitindo que o conteúdo dos buffers seja escrito nos registradores ou na memória. Se a especulação foi incorreta, o hardware faz um flush nos buffers e executa novamente, mas na seqüência de instruções correta.

A especulação introduz outro problema possível: especular sobre certas instruções pode introduzir exceções que anteriormente não estavam presentes. Por exemplo, suponha que uma instrução load seja movida de uma maneira especulativa, mas o endereço que usa não é válido quando a especulação for incorreta. O resultado é que ocorrerá uma exceção que não deveria ter ocorrido. O problema é complicado pelo fato de que, se a instrução load não fosse especulativa, então, a exceção deveria ocorrer! Na especulação feita pelo compilador, esses problemas são evitados pelo acréscimo de suporte especial à especulação, que permite que tais exceções sejam ignoradas até que esteja claro que elas realmente devam ocorrer. Na especulação por hardware, as exceções são simplesmente mantidas em um buffer até que fique claro que a instrução que as causa não é mais especulativa e está pronta para terminar; nesse ponto, a exceção é gerada, e prossegue o tratamento normal da exceção.

Como a especulação pode melhorar o desempenho quando realizada corretamente e diminuir o desempenho quando feita descuidadamente, é preciso haver muito esforço na decisão de quando a especulação é apropriada. Mais adiante, nesta seção, vamos examinar as técnicas estática e dinâmica para a especulação.

Despacho múltiplo estático

pacote de despacho
O conjunto de instruções despachadas juntas em 1 ciclo de clock; o pacote pode ser determinado estaticamente, pelo compilador, ou dinamicamente, pelo processador.

Todos os processadores de despacho múltiplo estático utilizam o compilador para ajudar no empacotamento de instruções e no tratamento de hazards. Em um processador de despacho estático, você pode pensar no conjunto de instruções despachadas em determinado ciclo de clock, o que é chamado **pacote de despacho**, como uma grande instrução com várias operações. Essa visão é mais do que uma analogia. Como um processador de despacho múltiplo estático normalmente restringe o mix de instruções que podem ser iniciadas em determinado ciclo de clock, é útil pensar no pacote de despacho como uma única instrução, permitindo várias operações em certos campos predefinidos. Essa visão levou ao nome original para essa técnica: VLIW (Very Long Instruction Word – palavra de instrução muito longa). A arquitetura Intel IA-64 utiliza essa técnica, que possui um nome próprio:

EPIC (Explicitly Parallel Instruction Computer – computador de instruções explicitamente paralelas). Os processadores Itanium e Itanium 2, disponíveis em 2000 e 2002, respectivamente, são as primeiras implementações da arquitetura IA-64.

A maioria dos processadores de despacho estático também conta com o compilador para assumir alguma responsabilidade por tratar de hazards de dados e controle. As responsabilidades do compilador podem incluir previsão estática de desvios e escalonamento de código, para reduzir ou impedir todos os hazards.

Vejamos uma versão simples do despacho estático de um processador MIPS, antes de descrevermos o uso dessas técnicas em processadores mais agressivos. Depois de usar esse exemplo simples para rever os comentários, discutimos os destaques da arquitetura Intel IA-64.

Um exemplo: despacho múltiplo estático com a ISA do MIPS

Para que você tenha uma idéia do despacho múltiplo estático, consideramos um processador MIPS simples capaz de despachar duas instruções por ciclo, sendo que uma das instruções pode ser uma operação da ALU com inteiros e a outra pode ser um load ou um store. Esse projeto é como aquele utilizado em alguns processadores MIPS embutidos. O despacho de duas instruções por ciclo exigirá a busca e a decodificação de 64 bits de instruções. Em muitos processadores de despacho múltiplo, e basicamente em todos os processadores VLIW, o layout do despacho de instruções simultâneas é restrito para simplificar a decodificação e o despacho da instrução. Logo, exigiremos que as instruções sejam emparelhadas e alinhadas em um limite de 64 bits, com a parte da ALU ou desvio aparecendo primeiro. Além do mais, se uma instrução do par não puder ser usada, exigimos que ela seja substituída por um nop. Assim, as instruções sempre são despachadas em pares, possivelmente com um nop em um slot. A Figura 6.44 mostra como as instruções aparecem enquanto entram no pipeline em pares.

Tipo de instrução	Estágios do pipe					
Instrução da ALU ou desvio	IF	ID	EX	MEM	WB	
Instrução load ou store	IF	ID	EX	MEM	WB	
Instrução da ALU ou desvio		IF	ID	EX	MEM	WB
Instrução load ou store		IF	ID	EX	MEM	WB
Instrução da ALU ou desvio			IF	ID	EX	MEM
Instrução load ou store			IF	ID	EX	WB
Instrução da ALU ou desvio				IF	ID	EX
Instrução load ou store				IF	ID	WB

FIGURA 6.44 Pipeline com despacho estático de duas instruções em operação. As instruções da ALU e de transferência de dados são despachadas ao mesmo tempo. Aqui, consideramos a mesma estrutura de cinco estágios utilizada para o pipeline de despacho único. Embora isso não seja estritamente necessário, possui algumas vantagens. Em particular, manter as escritas de registrador no final do pipeline simplifica o tratamento de exceções e a manutenção de um modelo de exceção preciso, que se torna mais difícil em processadores de despacho múltiplo.

Os processadores de despacho múltiplo estático variam no modo como lidam com hazards de dados e controle em potencial. Em alguns projetos, o compilador tem responsabilidade completa por remover *todos* os hazards, escalonando o código e inserindo nops de modo que o código execute sem qualquer necessidade de detecção de hazard ou stalls gerados pelo hardware. Em outros, o hardware detecta os hazards de dados e gera stalls entre dois pacotes de despacho, enquanto exige que o compilador evite todas as dependências dentro de um par de instruções. Mesmo assim, um hazard geralmente força o pacote de despacho inteiro contendo a instrução dependente a sofrer stall. Se o software precisa lidar com todos os hazards ou apenas tentar reduzir a fração de hazards entre pacotes de despacho separados, a aparência de haver uma única grande instrução com várias operações é reforçada. Ainda assumiremos a segunda técnica para esse exemplo.

Para emitir uma operação da ALU e uma operação de transferência de dados em paralelo, a primeira necessidade para o hardware adicional – além da lógica normal de detecção de hazard e stall – são portas extras no banco de registradores (ver Figura 6.45). Em 1 ciclo de clock, podemos ter de ler dois registradores para a operação da ALU e mais dois para um store, e também uma porta de escrita para uma operação da ALU e uma porta de escrita para um load. Como a ALU está presa à operação da ALU, também precisamos de um somador separado para calcular o endereço efetivo para as transferências de dados. Sem esses recursos extras, nosso pipeline com despacho duplo seria atrapalhado pelos hazards estruturais.

Claramente, esse processador com despacho duplo pode melhorar o desempenho por um fator de até 2. Entretanto, fazer isso exige que o dobro de instruções seja superposto na execução, e essa sobreposição adicional aumenta a perda de desempenho relativa aos hazards de dados e controle. Por exemplo, em nosso pipeline simples de cinco estágios, os loads possuem uma latência de uso de 1 ciclo de clock, o que impede que uma instrução use o resultado sem sofrer stall. No pipeline com despacho duplo e cinco estágios, o resultado de uma instrução load não pode ser usado no próximo *ciclo de clock*. Isso significa que as *duas* instruções seguintes não podem usar o resultado do load sem sofrer stall. Além do mais, as instruções da ALU que não tiveram latência de uso no pipeline simples de cinco estágios agora possuem uma latência de uso de uma instrução, pois os resultados não podem ser usados no load ou store emparelhados. Para explorar com eficiência o paralelismo disponível em um processador de com despacho múltiplo, é preciso utilizar técnicas mais ambiciosas de escalonamento de compilador ou hardware, e o despacho múltiplo estático requer que o compilador assuma essa função.

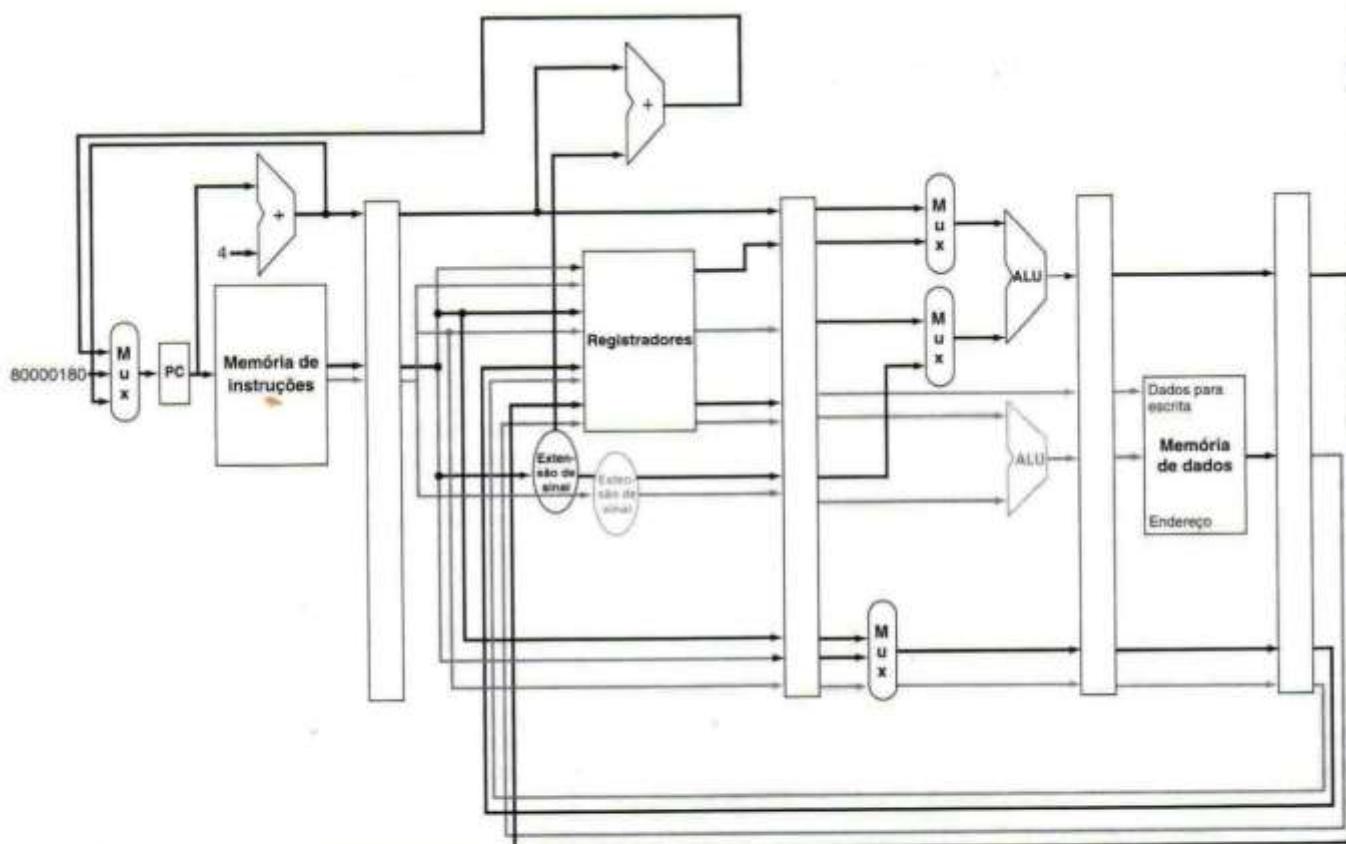


FIGURA 6.45 Um caminho de dados com despacho duplo estático. Os acréscimos necessários para o despacho duplo estão destacados: outros 32 bits da memória de instruções, mais duas portas de leitura e mais uma porta de escrita no banco de registradores, e outra ALU. Suponha que a ALU inferior trate dos cálculos de endereço para transferências de dados e a ALU superior trate de tudo o mais.

ESCALONAMENTO DE CÓDIGO SIMPLES PARA DESPACHO MÚLTIPLA

Como este loop seria escalonado em um pipeline com despacho duplo estático para o MIPS?

```
Loop: lw $t0, 0($s1)    # $t0=elemento do array
      addu $t0,$t0,$s2   # add escalar em $s2
      sw $t0, 0($s1)     # resultado do store
      addi $s1, $s1, -4    # decrementa ponteiro
      bne $s1,$zero,Loop  # desvia se $s1!=0
```

Reordene as instruções para evitar o máximo de stalls do pipeline possível. Considere que os desvios são previstos, de modo que os hazards de controle sejam tratados pelo hardware.

As três primeiras instruções possuem dependências de dados, bem como as duas últimas. A Figura 6.46 mostra o melhor escalonamento para essas instruções. Observe que apenas um par de instruções possui os dois slots utilizados. São necessários 4 clocks por iteração do loop; em 4 clocks para executar 5 instruções, obtemos o CPI decepcionante de 0,8 versus o melhor caso de 0,5, ou um IPC de 1,25 versus 2,0. Observe que, no cálculo do CPI ou do IPC, não contamos quaisquer nops executados como instruções úteis. Isso melhoraria o CPI, mas não o desempenho!

EXEMPLO

RESPOSTA

	Instrução da ALU ou desvio	Instrução de transferência de dados	Ciclo de clock
Loop:		lw \$t0, 0(\$s1)	1
	addi \$s1,\$s1,-4		2
	addu \$t0,\$t0,\$s2		3
	bne \$s1,\$zero,Loop	sw \$t0, 0(\$s1)	4

FIGURA 6.46 O código escalonado conforme apareceria em um pipeline MIPS com despacho duplo. Os slots vazios são nops.

Uma técnica de compilador importante para conseguir mais desempenho dos loops é o **desdobramento de loop (loop unrolling)**, uma técnica em que são feitas várias cópias do corpo do loop. Após o desdobramento, haverá mais ILP disponível pela sobreposição de instruções de diferentes iterações.

desdobramento de loop (loop unrolling) Uma técnica para conseguir mais desempenho dos loops que acessam arrays, em que são feitas várias cópias do corpo do loop e instruções de diferentes iterações são escalonadas juntas.

DESOBRAMENTO DE LOOP PARA PIPELINES COM DESPACHO MÚLTIPLA

Veja como o trabalho de desdobramento do loop e escalonamento funciona no exemplo anterior. Suponha que o índice do loop seja um múltiplo de quatro, para simplificar.

EXEMPLO

RESPOSTA

Para escalar o loop sem quaisquer atrasos, acontece que precisamos fazer quatro cópias do corpo do loop. Depois de desdobrar e eliminar as instruções de overhead de loop desnecessárias, o loop terá quatro cópias de lw, add e sw, mais um addi e um bne. A Figura 6.47 mostra o código desdobrado e escalonado.

Durante o processo de desdobramento, o compilador introduziu registradores adicionais (\$t1, \$t2, \$t3). O objetivo desse processo, chamado **renomeação de registradores**, é eliminar dependências que não são dependências de dados verdadeiras, mas que poderiam levar a hazards em potencial ou impedir que o compilador escalonasse o código de forma flexível. Considere como o código não desdobrado apareceria usando apenas \$t0. Haveria instâncias repetidas de lw \$t0,0(\$s1), addu \$t0,\$t0,\$s2 seguidas por sw \$t0,4(\$s1), mas essas sequências, apesar do uso de \$t0, na realidade são completamente independentes – nenhum valor de dados flui entre um par dessas instruções e o par seguinte. É isso que é chamado de **antide-**

renomeação de registradores O restante dos registradores é usado, pelo compilador ou hardware, para remover antidependências.

antidependência

Também chamada **dependência de nome**. Uma ordenação forçada pela reutilização de um nome, normalmente um registrador, em vez de uma dependência verdadeira que transporta um valor entre duas instruções.

pendência ou dependência de nome, que é uma ordenação forçada puramente pela reutilização de um nome, em vez de uma dependência de dados real.

Renomear os registradores durante o processo de desdobramento permite que o compilador move subsequentemente essas instruções independentes de modo a escalonar melhor o código. O processo de renomeação elimina as dependências de nome, enquanto preserva as verdadeiras dependências.

Observe agora que 12 das 14 instruções no loop são executadas como um par. São necessários 8 clocks para quatro iterações do loop, ou 2 clocks por iteração, o que gera um CPI de $8/14 = 0,57$. O desdobramento e o escalonamento do loop com despacho dual nos deu um fator de melhoria de dois, parcialmente pela redução das instruções de controle de loop e parcialmente pela execução do despacho dual. O custo dessa melhoria de desempenho é usar quatro registradores temporários em vez de um, além de um aumento significativo no tamanho do código.

	Instrução da ALU ou desvio	Instrução de transferência de dados	Ciclo de clock
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 6.47 O código desdoblado e escalonado da Figura 6.46 conforme apareceria no pipeline MIPS com despacho duplo estático. Os slots vazios são nops. Como a primeira instrução no loop decrementa \$s1 em 16, os endereços lidos são o valor original de \$s1, depois esse endereço menos 4, menos 8 e menos 12.

A arquitetura IA-64 da Intel

A arquitetura IA-64 é um conjunto de instruções registrador-registrador, estilo RISC, como a versão de 64 bits da arquitetura MIPS (chamada MIPS-64), mas com vários recursos exclusivos para dar suporte à exploração explícita do ILP, controlada por compilador. A Intel chama a técnica de EPIC (Explicitly Parallel Instruction Computer). As principais diferenças entre a IA-64 e a arquitetura MIPS são as seguintes:

1. A IA-64 possui muito mais registradores do que a MIPS, incluindo 128 registradores inteiros e 128 registradores de ponto flutuante, além de 8 registradores especiais para desvios e 64 registradores de condição de 1 bit. Além disso, a IA-64 admite janelas de registradores em um padrão semelhante à arquitetura RISC original de Berkeley RISC e à arquitetura SPARC da Sun.
2. A IA-64 coloca instruções em pacotes que possuem um formato fixo e designação explícita de dependências.
3. A IA-64 inclui instruções e capacidades especiais para especulação e para eliminação de desvio, o que aumenta a quantidade de ILP que pode ser explorado.

A arquitetura IA-64 é projetada para alcançar os maiores benefícios de um paralelismo implícito da técnica VLIW entre operações em uma instrução e formatação fixa dos campos da operação – enquanto mantém maior flexibilidade do que uma VLIW normalmente permite. A arquitetura IA-64 utiliza dois conceitos diferentes para conseguir essa flexibilidade: grupos de instruções e pacotes.

Um **grupo de instruções** é uma seqüência de instruções consecutivas sem dependências de dados em registradores entre elas. Todas as instruções em um grupo poderiam ser executadas em paralelo se houvesse recursos de hardware suficientes e se quaisquer dependências por meio da memória fossem preservadas. Um grupo de instruções pode ser arbitrariamente longo, mas o compilador precisa indicar explicitamente o limite entre um grupo de instruções e outro. Esse limite é indicado incluindo uma **parada** entre duas instruções que pertencem a diferentes grupos.

grupo de instruções

Em IA-64, uma seqüência de instruções consecutivas sem dependências de dados em registradores entre elas.

parada Em IA-64, um indicador explícito de uma interrupção entre instruções independentes e dependentes.

As instruções IA-64 são codificadas em *pacotes*, que possuem 128 bits de largura. Cada pacote consiste em um campo de modelo de 5 bits e três instruções, cada uma com 41 bits de largura. Para simplificar o processo de decodificação e despacho de instruções, o campo de modelo de um pacote especifica quais das diferentes unidades de execução cada instrução do pacote necessita. As cinco unidades de execução diferentes são ALU de inteiros, ALU para não inteiros (inclui operações de deslocamento e multimídia), unidade de memória, unidade de ponto flutuante e unidade de desvio.

O campo de modelo de 5 bits dentro de cada pacote descreve *tanto* a presença de quaisquer paradas associadas ao pacote *quanto* o tipo da unidade de execução exigida em cada instrução dentro do pacote. Os formatos de pacote só podem especificar um subconjunto de todas as combinações possíveis dos tipos de instruções e paradas.

Para melhorar a quantidade de ILP que pode ser explorado, a IA-64 oferece suporte extenso para predicação e para especulação (ver a seção “Detalhamento” na página seguinte). **Predicação** é uma técnica que pode ser usada para eliminar desvios, tornando a execução de uma instrução dependente de um predicado e não dependente de um desvio. Como já vimos, os desvios reduzem a oportunidade de explorar o ILP, restringindo a movimentação de código. O desdobramento do loop funciona bem para eliminar desvios de loop, mas um desvio dentro de um loop – surgindo, por exemplo, de uma instrução *if-then-else* – não pode ser eliminado pelo desdobramento do loop. Contudo, a predicação oferece um método para eliminar o desvio, permitindo a exploração mais flexível do paralelismo.

predicação Uma técnica para tornar as instruções dependentes de predicados, e não de desvios.

Por exemplo, suponha que tenhamos uma seqüência de código como

```
if (p) {comando 1} else {comando 2}
```

Usando métodos de compilação normais, este segmento seria compilado usando dois desvios: um após o desvio de condição para a parte *else* e um após a instrução 1 desviando para a próxima instrução seqüencial. Com a predicação, ele poderia ser compilado como

```
(p) comando 1  
(-p) comando 2
```

onde o uso de (*condição*) indica que a instrução só é executada se a condição for verdadeira e, caso contrário, torna-se um no-op. Observe que a predicação pode ser usada como um meio de especular, bem como um método para eliminar desvios.

A arquitetura IA-64 oferece suporte abrangente para a predicação: quase toda instrução na arquitetura IA-64 pode ser predicada especificando um registrador de predicado, cuja identidade é colocada nos 6 bits menos significativos de um campo de instrução. Uma consequência da predicação completa é que um desvio condicional é simplesmente um desvio com predicado de proteção!

A IA-64 é o exemplo mais sofisticado de um conjunto de instruções com suporte para a exploração baseada em compilador do ILP. Os processadores Itanium e Itanium 2 da Intel implementam essa arquitetura. Um breve resumo das características desses processadores pode ser visto na Figura 6.48.

Processador	Máximo de instruções despachadas/clock	Unidades funcionais	Máximo de operações por clock	Velocidade máxima de clock	Transistores (milhões)	Potência (watts)	SPEC int2000	SPEC fp2000
Itanium	6	4 inteiros/mídia 2 memória 3 desvio 2 FP	9	0,8 GHz	25	130	379	701
Itanium 2	6	6 integer/mídia 4 memória 3 desvio 2 FP	11	1,5 GHz	221	130	810	1427

FIGURA 6.48 Um resumo das características do Itanium e Itanium 2, as duas primeiras implementações da Intel para a arquitetura IA-64. Além de velocidades de clock mais alta e mais unidades funcionais, o Itanium 2 inclui uma cache de nível 3 no chip *versus* uma cache nível 3 fora do chip no Itanium.

veneno Um resultado gerado quando um load especulativo gera uma exceção, ou uma instrução utiliza um operando envenenado.

advanced load No IA-64, uma instrução load especulativa com suporte para verificar aliases que poderiam invalidar o load.

superescalár Uma técnica de pipelining avançada que permite que o processador execute mais de uma instrução por ciclo de clock.

escalonamento dinâmico em pipeline Suporte do hardware para modificar a ordem de execução das instruções de modo a evitar stalls.

Detalhamento: o suporte para especulação na arquitetura IA-64 consiste em suporte separado para especificação de controle, que lida com o adiamento de exceções para as instruções especuladas, e especulação de referências de memória, que admite especulação de instruções load. O tratamento de exceções adiadas é admitido pela inclusão de instruções load especulativas que, quando ocorre uma exceção, marca o resultado como **veneno**. Quando um resultado venenoso é usado por uma instrução, o resultado também é venenoso. O software pode, então, verificar um resultado venenoso quando souber que a execução não é mais especulativa.

No IA-64, também podemos especular sobre referências de memória movendo loads antes de stores dos quais podem depender. Isso é feito com uma instrução **advanced load**. Um advanced load é executado normalmente, mas utiliza uma tabela especial para acompanhar o endereço do qual o processador leu. Todos os stores subsequentes verificam essa tabela e geram um flag na entrada se o endereço do store combinar com o endereço do load. Uma instrução subsequente terá de ser usada para verificar o status da entrada depois que o load não for mais especulativo. Se um store para o mesmo endereço intervir, a instrução de verificação especifica uma rotina de reparo que executa novamente o load e quaisquer outras instruções dependentes antes de continuar com a execução; se nenhum store desse tipo ocorrer, a entrada da tabela simplesmente será apagada, indicando que o load não é mais especulativo.

Processadores com despacho múltiplo dinâmico

Os processadores de despacho múltiplo dinâmico também são conhecidos como processadores **superescalares**, ou simplesmente superescalares. Nos processadores superescalares mais simples, as instruções são despachadas em ordem, e o processador decide se zero, uma ou mais instruções podem ser despachadas em determinado ciclo de clock. Obviamente, conseguir um bom desempenho em tal processador ainda exige que o compilador tente escalonar instruções para separar as dependências e, com isso, melhorar a velocidade de despacho de instruções. Mesmo com esse escalonamento de compilador, existe uma diferença importante entre essa arquitetura superescalar simples e um processador VLIW: o código, seja ele escalonado ou não, é garantido pelo hardware que será executado corretamente. Além do mais, o código compilado sempre será executado corretamente, independente da velocidade de despacho ou estrutura do pipeline do processador. Em alguns projetos VLIW, isso não tem acontecido, e a recompilação foi necessária quando da mudança por diferentes modelos de processador; em outros processadores de despacho estático, o código seria executado corretamente em diversas implementações, mas constantemente de uma forma tão pouco eficiente que torna a compilação necessária.

Muitas arquiteturas superescalares estendem a estrutura básica das decisões de despacho dinâmico para incluir **escalonamento dinâmico em pipeline**. O escalonamento dinâmico em pipeline escolhe quais instruções serão executadas em determinado ciclo de clock, enquanto tenta evitar hazards e stalls. Vamos começar com um exemplo simples de impedimento de um hazard de dados. Considere a seguinte sequência de código:

```
lw    $t0, 20($s2)
addu $t1, $t0, $t2
sub  $s4, $s4, $t3
slti $t5, $s4, 20
```

Embora a instrução sub esteja pronta para executar, ela precisa esperar que lw e addu terminem primeiro, o que poderia exigir muitos ciclos de clock se a memória for lenta. (O Capítulo 7 explica as caches, motivo pelo qual os acessos à memória às vezes são muito lentos.) O escalonamento dinâmico em pipeline permite que tais hazards sejam evitados total ou parcialmente.

Escalonamento dinâmico em pipeline

O escalonamento dinâmico em pipeline escolhe quais instruções serão executadas em seguida, possivelmente reordenando-as para evitar stalls. Nesses processadores, o pipeline é dividido em três unidades principais: uma unidade de busca e despacho de instruções, várias unidades funcionais (10

ou mais nos projetos de alto nível em 2004) e uma **unidade de commit**. A Figura 6.49 mostra o modelo. A primeira unidade busca instruções, decodifica-as e envia cada instrução a uma unidade funcional correspondente para execução. Cada unidade funcional possui buffers, chamados **estações de reserva**, que mantêm os operandos e a operação. (Na próxima seção, discutiremos uma alternativa às estações de reserva utilizadas por muitos processadores recentes.) Assim que o buffer tiver todos os seus operandos e a unidade funcional estiver pronta para executar, o resultado será calculado. Quando o resultado for completado, ele será enviado a quaisquer estações de reserva esperando por esse resultado em particular, bem como a unidade de commit, que mantém o resultado em um buffer até que seja seguro colocar o resultado no banco de registradores ou, para um store, na memória. O buffer na unidade de commit, normalmente chamado de **buffer de reordenação**, também é usado para fornecer operandos, mais ou menos da mesma maneira como a lógica de forwarding faz em um pipeline escalonado estaticamente. Quando um resultado é submetido ao banco de registradores, ele pode ser apanhado diretamente de lá, como em um pipeline normal.

A combinação de operandos em buffers nas estações de reserva e os resultados no buffer de reordenação oferece uma forma de renomeação de registradores, assim como aquela utilizada pelo compilador em nosso exemplo anterior de desdobramento de loop, na página 331. Para ver como isso funciona conceitualmente, considere as seguintes etapas:

1. Quando uma instrução é despachada, se um de seus operandos estiver no banco de registradores ou no buffer de reordenação, ele será copiado para a estação de reserva imediatamente, onde será colocado em um buffer até que todos os operandos e a unidade de execução estejam disponíveis. Para a instrução despachada, a cópia do registrador operando não é mais necessária, e se houvesse uma escrita nesse registrador, o valor poderia ser reescrito.
2. Se um operando não estiver no banco de registradores ou no buffer de reordenação, ele terá de estar esperando para ser produzido por uma unidade funcional. O nome da unidade funcional que produzirá o resultado é acompanhado. Quando essa unidade por fim produz o resultado, ele é copiado diretamente para a estação de reserva, que estava aguardando, a partir da unidade funcional, sem passar pelos registradores.

Essas etapas efetivamente utilizam o buffer de reordenação e as estações de reserva para implementar a renomeação de registradores.

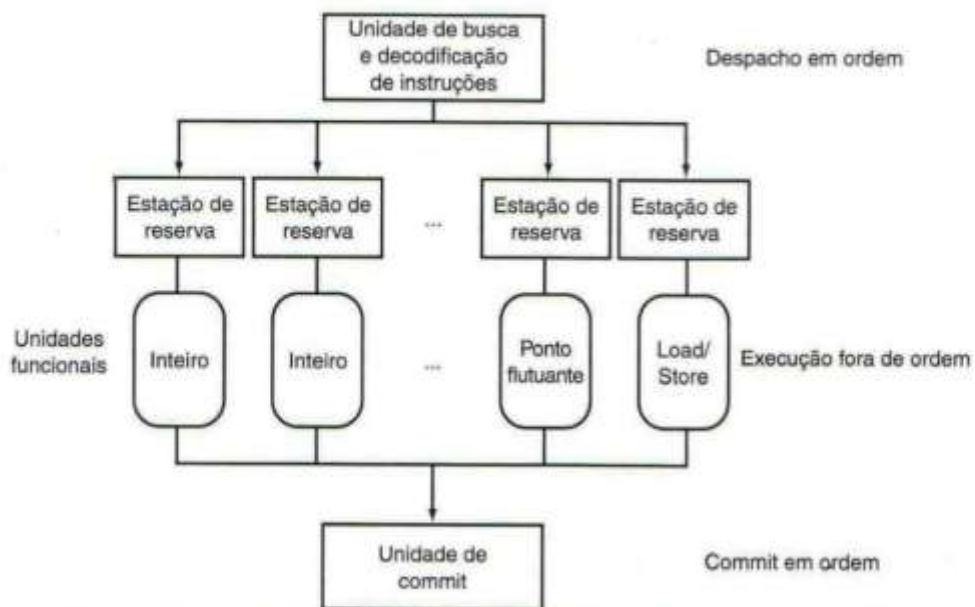


FIGURA 6.49 As três unidades principais de um pipeline escalonado dinamicamente. A etapa final da atualização do estado também é chamada de reforma ou graduação.

unidade de commit A unidade em um pipeline de execução dinâmica ou fora de ordem que decide quando é seguro liberar o resultado de uma operação aos registradores e memória visíveis ao programador.

estação de reserva Um buffer dentro de uma unidade funcional que mantém os operandos e a operação.

buffer de reordenação O buffer que mantém resultados em um processador escalonado dinamicamente até que seja seguro armazenar os resultados na memória ou em um registrador.

Conceitualmente, você pode pensar em um pipeline escalonado de forma dinâmica como uma análise da estrutura de fluxo de dados de um programa, conforme vimos quando discutimos a análise de fluxo de dados dentro de um compilador no Capítulo 2. O processador executa as instruções em alguma ordem que preserva a ordem do fluxo de dados do programa. Para fazer com que os programas se comportem como se estivessem executando em um pipeline simples em ordem, a unidade de busca e decodificação de instruções precisa despachar instruções em ordem, o que permite que as dependências sejam acompanhadas, e a unidade de commit precisa escrever resultados nos registradores e na memória na ordem de execução do programa. Esse modo conservador é chamado de término em ordem. Logo, se houver uma exceção, o computador poderá apontar para a última instrução executada, e os únicos registradores atualizados serão aqueles escritos pelas instruções antes da instrução que causa a exceção. Apesar de o front end (busca e despacho) e o back end (commit) do pipeline executarem em ordem, as unidades funcionais são livres para iniciar a execução sempre que os dados de que precisam estiverem disponíveis. Hoje, todos os pipelines escalonados dinamicamente utilizam o término em ordem, embora isso nem sempre seja verdadeiro.

Em geral, o escalonamento dinâmico é estendido pela inclusão da especulação baseada em hardware, especialmente para resultados de desvios. Prevendo a direção de um desvio, um processador escalonado dinamicamente pode continuar a buscar e executar instruções ao longo do caminho previsto. Como as instruções possuem um **commit em ordem**, sabemos se o desvio foi previsto corretamente ou não antes que quaisquer instruções do caminho previsto tenham seus resultados atualizados pelas unidades de commit. Um pipeline especulativo, escalonado dinamicamente, também pode admitir especulação nos endereços de load, permitindo uma reordenação load-store e usando a unidade de commit para evitar a especulação incorreta. Na próxima seção, veremos o uso do escalonamento dinâmico com especulação no projeto do Pentium 4.

commit em ordem: Um commit em que os resultados da execução em pipeline sejam escritos no estado visível ao programador na mesma ordem em que as instruções são buscadas.

execução fora de ordem: Uma situação na execução em pipeline quando uma instrução com execução bloqueada não faz com que as instruções seguintes esperem.

Detalhamento: uma unidade de commit controla as atualizações no banco de registradores e na memória. Alguns processadores escalonados dinamicamente atualizam o banco de registradores imediatamente durante a execução, usando registradores extras para implementar a função de renomeação e preservar a cópia antiga de um registrador até que a instrução que atualiza o registrador não seja mais especulativa. Outros processadores colocam os resultados em buffers, normalmente em estruturas chamadas de buffer de reordenação, e a atualização real no banco de registradores ocorre mais tarde, como parte do commit. Os stores na memória precisam ser colocados em buffers até o momento do commit, seja no buffer de store (ver Capítulo 7) ou no buffer de reordenação. A unidade de commit permite que o store escreva na memória a partir do buffer quando o buffer tiver um endereço válido e dados válidos, e quando o store não for mais dependente dos desvios previstos.

Detalhamento: os acessos à memória se beneficiam das caches sem bloqueio, que continuam atendendo aos acessos à cache durante uma falha de cache (ver Capítulo 7). Processadores com **execução fora de ordem** precisam de caches sem bloqueio para permitir que as instruções sejam executadas durante uma falha.

Interface hardware/software

Dado que os compiladores também podem escalonar o código em torno das dependências de dados, você poderia perguntar por que um processador superescalar usaria o escalonamento dinâmico. Existem três motivos principais. Primeiro, nem todos os stalls são previsíveis. Em particular, as falhas de cache (ver Capítulo 7) causam stalls imprevisíveis. O escalonamento dinâmico permite que o processador oculte alguns desses stalls continuando a executar instruções enquanto esperam que o stall termine.

Segundo, se o processador especula sobre resultados de desvio usando a previsão de desvio dinâmica, ele não pode saber a ordem exata das instruções durante a compilação, pois isso depende do comportamento previsto e real dos desvios. A incorporação da especulação dinâmica para explorar mais ILP sem incorporar o escalonamento dinâmico restringiria significativamente os benefícios de tal especulação. Terceiro, como a latência do pipeline e a largura do despacho mudam de uma implementação para outra, a melhor maneira de compilar uma sequência de código também muda. Por exemplo, como escalar

uma sequência de instruções dependentes é algo afetado tanto pela largura quanto pela latência do despacho. A estrutura do pipeline afeta o número de vezes que um loop precisa ser desdobrado para evitar stalls e também o processo de renomeação de registradores feito pelo compilador. O escalonamento dinâmico permite que o hardware oculte a maioria desses detalhes. Assim, os usuários e os distribuidores de software não precisam se preocupar em ter várias versões de um programa para diferentes implementações do mesmo conjunto de instruções. De modo semelhante, o código antigo legado receberá grande parte do benefício de uma nova implementação sem a necessidade de recompilação.

Colocando em perspectiva

Tanto a técnica de pipelining quanto a execução com despacho múltiplo aumentam a vazão máxima de instruções e a tentativa de explorar o ILP. No entanto, as dependências de dados e controle nos programas oferecem um limite superior sobre o desempenho sustentado, pois o processador às vezes precisa esperar que uma dependência seja resolvida. As técnicas centradas no software para a exploração do ILP contam com a capacidade do compilador de encontrar e reduzir os efeitos de tais dependências, enquanto as técnicas centradas no hardware contam com extensões para o pipeline e mecanismos de despacho. A especulação, realizada pelo compilador ou pelo hardware, pode aumentar a quantidade de ILP que pode ser explorada, embora se deva ter cuidado, visto que a especulação incorreta provavelmente reduzirá o desempenho.

Entendendo o desempenho dos programas

Processadores modernos, de alto desempenho, são capazes de despachar várias instruções por clock; infelizmente, é muito difícil sustentar essa taxa de despacho. Por exemplo, apesar da existência de processadores com despacho de quatro a seis instruções por clock, muito poucas aplicações podem sustentar mais do que duas instruções por clock. Existem dois motivos principais para isso.

Primeiro, dentro do pipeline, os principais gargalos no desempenho surgem das dependências que não podem ser aliviadas, reduzindo assim o paralelismo entre as instruções e a velocidade de despacho sustentada. Embora pouca coisa possa ser feita sobre as verdadeiras dependências dos dados, normalmente o compilador ou o hardware não sabe exatamente se uma dependência existe ou não e, por isso, precisa considerar de forma conservadora que a dependência existe. Por exemplo, o código que utiliza ponteiros, principalmente os que criam mais aliasing, levará a dependências em potencial mais implícitas. Ao contrário, a maior regularidade dos acessos a um array normalmente permite que um compilador deduza que não existem dependências. De modo semelhante, os desvios que não podem ser previstos com precisão, seja em tempo de execução ou de compilação, limitarão a capacidade de explorar o ILP. Em geral, o ILP adicional está disponível, mas a capacidade de o compilador ou o hardware encontrar ILP que possa estar bastante separado (às vezes pela execução de milhares de instruções) é limitada.

Em segundo lugar, as perdas no sistema da memória (o tópico do Capítulo 7) também limitam a capacidade de manter o pipeline cheio. Alguns stalls do sistema de memória podem ser escondidos, mas quantidades limitadas de ILP também limitam a extensão à qual esses stalls podem ser escondidos.

Indique se as técnicas ou componentes a seguir estão associados principalmente a uma técnica baseada em software ou hardware para a exploração do ILP. Em alguns casos, a resposta pode ser “ambos”.

Verifique você mesmo

1. Previsão de desvio
2. Despacho múltiplo

3. VLIW
4. Superescalar
5. Escalonamento dinâmico
6. Execução fora de ordem
7. Especulação
8. EPIC
9. Buffer de reordenação
10. Renomeação de registradores
11. Predição

6.10

Vida real: o pipeline do Pentium 4

No capítulo anterior, discutimos como o Pentium 4 buscava e traduzia instruções IA-32 em microoperações. As microoperações, então, são executadas por um pipeline especulativo, sofisticado, escalonado dinamicamente, capaz de sustentar uma velocidade de execução de três microoperações por ciclo de clock. Esta seção focaliza esse pipeline de microoperação. O Pentium 4 combina o despacho múltiplo com um pipeline profundo a fim de conseguir um CPI baixo e uma velocidade de clock alta.

Quando consideramos o projeto de processadores sofisticados, escalonados dinamicamente, o projeto de unidades funcionais, a cache e o banco de registradores, o despacho de instruções e o controle geral do pipeline se misturam, dificultando a separação entre o caminho de dados e o pipeline. Por causa disso, muitos engenheiros e pesquisadores adotaram o termo **microarquitetura** para se referirem à arquitetura interna detalhada de um processador. A Figura 6.50 mostra a microarquitetura do Pentium 4, focalizando as estruturas para execução das microoperações.

Outra maneira de examinar o Pentium 4 é ver os estágios do pipeline pelos quais uma instrução típica passa. A Figura 6.51 mostra a estrutura do pipeline e o número típico de ciclos de clock gastos em cada estágio; naturalmente, o número de ciclos de clock varia devido à natureza do escalonamento dinâmico e também aos requisitos das microoperações individuais.

O Pentium 4 e seus antecessores Pentium III e Pentium Pro utilizam a técnica de decodificar instruções IA-32 em microoperações e executar essas microoperações usando um pipeline especulativo com diversas unidades funcionais. Na verdade, a microarquitetura básica é semelhante, e todos esses processadores podem completar até três microoperações por ciclo. O Pentium 4 ganha sua vantagem no desempenho em relação ao Pentium III devido a diversas melhorias:

1. Um pipeline com aproximadamente o dobro da profundidade (aproximadamente 20 ciclos, contra 10) e que pode executar com quase o dobro da velocidade na mesma tecnologia
2. Mais unidades funcionais (7 contra 5)
3. Suporte para um número maior de operações pendentes (126 contra 40)
4. O uso de uma cache de trace (ver Capítulo 7) e um previsor de desvios muito melhor (4K entradas *versus* 512)
5. Outras melhorias para o sistema de memória, que discutiremos no Capítulo 7

microarquitetura

A organização do processador, incluindo as principais unidades funcionais, sua interconexão e controle.

registradores da arquitetura

Os registradores visíveis do conjunto de instruções de um processador; por exemplo, no MIPS, estes são os 32 registradores inteiros e 16 de ponto flutuante.

Detalhamento: o Pentium 4 usa um esquema para resolver antidependências e especulação incorreta, que utiliza um buffer de reordenação junto com a renomeação de registradores. A renomeação de registradores renomeia explicitamente os **registradores da arquitetura** de um processador (8 no caso do IA-32) para um conjunto maior de registradores físicos (128 no Pentium 4). O Pentium 4 utiliza a renomeação de registradores para remover antidependências. A renomeação de registradores exige que o processador mantenha um mapa entre os re-

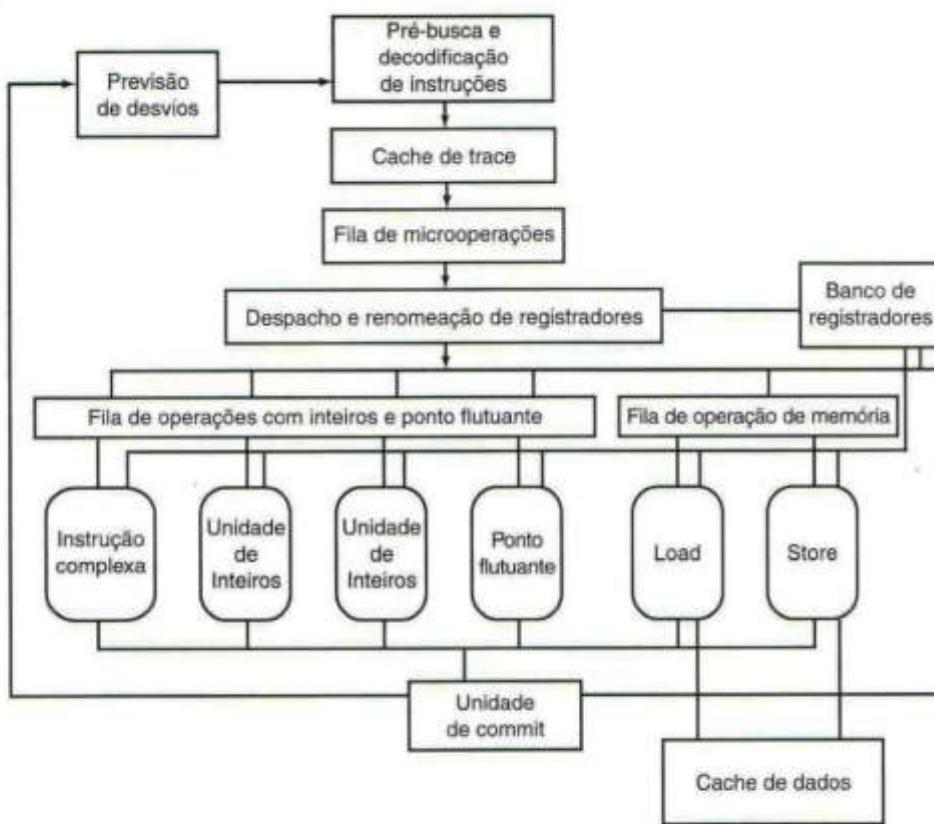


FIGURA 6.50 A microarquitetura do Intel Pentium 4. As extensas filas permitem que até 126 microoperações estejam pendentes em qualquer ponto do tempo, incluindo 48 loads e 24 stores. Na realidade, existem sete unidades funcionais, pois a unidade FP inclui uma unidade dedicada separada para moves de ponto flutuante. As unidades load e store, na realidade, são separadas em duas partes, com a primeira parte tratando do cálculo do endereço e a segunda parte responsável pela referência real à memória. As ALUs de inteiros operam com o dobro da frequência de clock, permitindo que duas operações da ALU com inteiros sejam concluídas para cada uma das duas unidades de inteiros em um único ciclo de clock. Conforme descrevemos no Capítulo 5, o Pentium 4 usa uma cache especial, chamada cache de trace, para manter sequências precodificadas de microoperações, correspondendo a instruções IA-32. A operação de uma cache de trace é explicada com mais detalhes no Capítulo 7. A unidade de FP também trata de instruções de multimídia MMX e SSE2. Existe uma extensa rede entre as unidades funcionais para efetuar bypass; como o pipeline é dinâmico, e não estático, o bypass é feito marcando resultados e fazendo um trace dos operandos origem, de modo a permitir uma combinação quando um resultado é produzido para uma instrução que está em uma das filas e que precisa do resultado. A Intel pretendia liberar duas novas versões do Pentium 4 na época em que este livro foi escrito, que provavelmente teriam mudanças na microarquitetura.

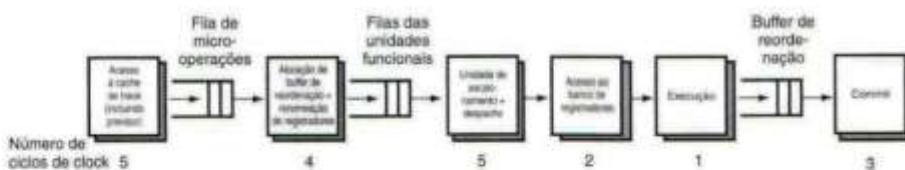


FIGURA 6.51 O pipeline do Pentium 4 mostrando o fluxo do pipeline para uma instrução típica e o número de ciclos de clock para as principais etapas no pipeline. Também aparecem os principais buffers onde as instruções esperam.

gistradores da arquitetura e os registradores físicos, indicando qual registrador físico é a cópia mais atualizada de um registrador da arquitetura. Registrando as renomeações que ocorreram, a renomeação de registradores oferece outra técnica para a recuperação no caso de especulação incorreta: basta desfazer os mapeamentos que ocorreram desde a primeira instrução especulada incorretamente. Isso fará com que o estado do processador retorne à última instrução executada corretamente, mantendo o mapeamento correto entre os registradores da arquitetura e os registradores físicos.

Entendendo o desempenho dos programas

O Pentium 4 combina um pipeline profundo (em média, 20 ou mais estágios de pipe por instrução) e despacho múltiplo agressivo para conseguir alto desempenho. Mantendo baixas as latências para operações back-to-back (0 para operações da ALU e 2 para loads), o impacto das dependências de dados é reduzido. Quais são os gargalos de desempenho em potencial mais sérios para os programas executados nesse processador? A lista a seguir inclui alguns problemas de desempenho em potencial, com os três últimos podendo se aplicar de alguma forma a qualquer processador com pipeline de alto desempenho.

- O uso de instruções IA-32 que não são mapeadas para três ou menos microoperações simples
- Desvios que são difíceis de se prever, causando stalls e reinícios mal previstos quando a especulação falha
- Localidade de instruções fraca, o que faz com que a cache de trace não funcione de forma eficaz
- Dependências longas – normalmente causadas por instruções duradouras ou falhas de cache de dados –, que causam stalls
- Atrasos de desempenho que surgem no acesso à memória (ver Capítulo 7), fazendo com que o processador sofra stall

Verifique você mesmo

As afirmações a seguir são verdadeiras ou falsas?

1. O Pentium 4 pode despachar mais instruções por clock do que o Pentium III.
2. O pipeline de despacho múltiplo do Pentium 4 executa instruções IA-32 diretamente.
3. O Pentium 4 utiliza o escalonamento dinâmico, mas não a especulação.
4. A microarquitetura do Pentium 4 possui muito mais registradores do que o IA-32 requer.
5. O pipeline do Pentium 4 possui menos estágios do que o do Pentium III.
6. A cache de trace do Pentium 4 é exatamente o mesmo que uma cache de instruções.

6.11

Falácia e armadilhas

Falácia: pipelining é fácil.

Nossos livros comprovam a sutileza da execução correta de um pipeline. Nossa livro avançado tinha um bug no pipeline em sua primeira edição, apesar de ter sido revisado por mais de 100 pessoas e testado nas salas de aula de 18 universidades. O bug só foi descoberto quando alguém tentou montar um computador com aquele livro. O fato de que o Verilog para descrever um pipeline como esse do Pentium 4 terá milhares de linhas é uma indicação da complexidade. Esteja atento!

Falácia: as idéias de pipelining podem ser implementadas independentes da tecnologia.

Quando o número de transistores no chip e a velocidade dos transistores tornou um pipeline de cinco estágios a melhor solução, então o delayed branch (veja a seção “Detalhamento” na página 318-319) foi uma solução simples para controlar os hazards. Com pipelines maiores, a execução superescalar

e a previsão de desvios, agora isso é redundante. No início da década de 1990, o escalonamento dinâmico em pipeline exigia muitos recursos e não era necessário para o alto desempenho, mas, à medida que a quantidade de transistores continuava a dobrar, a lógica se tornava muito mais rápida do que a memória, então as múltiplas unidades funcionais e os pipelines dinâmicos fizeram mais sentido. Hoje, todos os processadores de alto nível utilizam despacho múltiplo, e a maioria também escolhe implementar a especulação agressiva.

Armadilha: a falha em considerar o projeto do conjunto de instruções pode afetar o pipeline de forma adversa.

Muitas das dificuldades em pipelining surgem por causa das complicações do conjunto de instruções. Aqui estão alguns exemplos:

- Tamanhos de instrução e tempos de execução muito variáveis podem causar desequilíbrio entre estágios do pipeline e complicar bastante a detecção de hazards em um projeto com pipeline, no nível do conjunto de instruções. Esse problema foi contornado, inicialmente no DEC VAX 8500, no final da década de 1980, usando o esquema de micropipeline que o Pentium 4 emprega hoje. Naturalmente, o overhead da tradução e a manutenção da correspondência entre as microoperações e as instruções permanecem.
- Modos de endereçamento sofisticados podem levar a diferentes tipos de problemas. Os modos de endereçamento que atualizam registradores (ver Capítulo 3) complicam a detecção de hazards. Outros modos de endereçamento que exigem múltiplos acessos à memória complicam bastante o controle do pipeline e tornam difícil manter o pipeline fluindo tranquilamente.

Talvez o melhor exemplo seja o DEC Alpha e o DEC NVAX. Em uma tecnologia comparável, o conjunto de instruções mais recente do Alpha permitiu uma implementação cujo desempenho tem mais do que o dobro da velocidade do NVAX. Em outro exemplo, Bhandarkar e Clark [1991] compararam o MIPS M/2000 e o DEC VAX 8700 contando os ciclos de clock dos benchmarks SPEC; eles concluíram que, embora o MIPS M/2000 execute mais instruções, o VAX na média executa 2,7 vezes mais ciclos de clock, de modo que o MIPS é mais rápido.

6.12

Comentários finais

A técnica de pipelining melhora o tempo de execução médio por instrução. Dependendo se você começa com um caminho de dados de ciclo único ou de ciclo múltiplo, essa redução pode ser considerada uma diminuição do tempo de ciclo do clock ou do número de ciclos de clock por instruções (CPI). Começamos com o caminho de dados simples de ciclo único, de modo que a técnica de pipelining foi apresentada como reduzindo o tempo do ciclo de clock do caminho de dados simples. O despacho múltiplo, em comparação, focaliza claramente a redução do CPI (ou aumento do IPC). A Figura 6.52 mostra o efeito sobre o CPI e a velocidade de clock para cada uma das microarquiteturas dos Capítulos 5 e 6. O desempenho aumenta quando se sobe e se segue para a direita, pois é o produto do IPC e da velocidade de clock que determina o desempenho para determinado conjunto de instruções.

A técnica de pipelining melhora a vazão, mas não o tempo de execução inherente (ou *latência*) das instruções; a latência é semelhante, em duração, à técnica de multiciclo. Ao contrário dessa técnica, que utiliza o mesmo hardware repetidamente durante a execução da instrução, a técnica de pipelining inicia uma instrução a cada ciclo de clock com um hardware dedicado. De modo semelhante, o despacho múltiplo acrescenta um hardware adicional ao caminho de dados para permitir que várias instruções sejam iniciadas a cada ciclo de clock, mas com um aumento na latência efetiva. A Figura 6.53 mostra os caminhos de dados da Figura 6.52 colocados de acordo com a quantidade de compartilhamento de hardware e **latência de instrução**.

Noventa por
cento da
sabedoria
consiste em ser
sensato no
tempo.
Provérbio
americano

Latência de Instrução
O tempo de execução
inerente para uma
instrução.

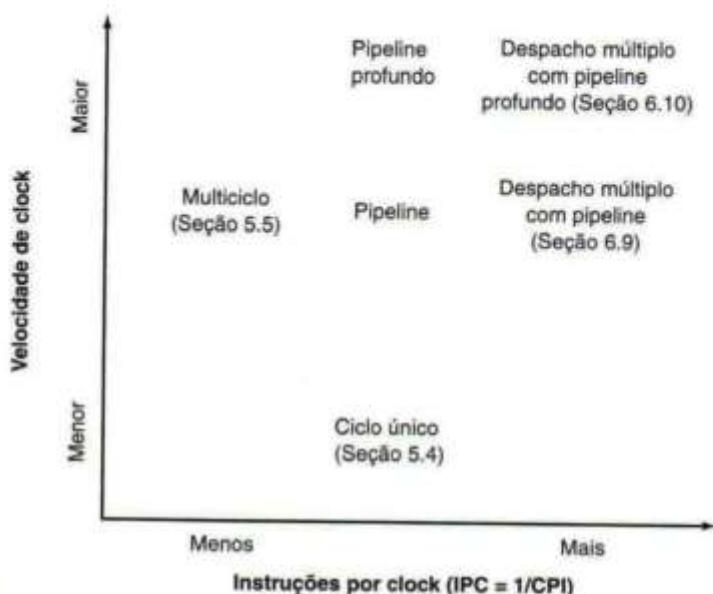


FIGURA 6.52 As consequências no desempenho do caminho de dados simples (ciclo único) e do caminho de dados multiciclo do Capítulo 5 e o modelo de execução em pipeline do Capítulo 6. Lembre-se de que o desempenho da CPU é uma função do IPC vezes a velocidade de clock, de modo que a passagem para o canto superior direito aumenta o desempenho. Embora as instruções por ciclo de clock sejam ligeiramente maiores no caminho de dados simples, o caminho de dados em pipeline é próximo e utiliza a taxa de clock tão rapidamente quanto o caminho de dados multiciclo.

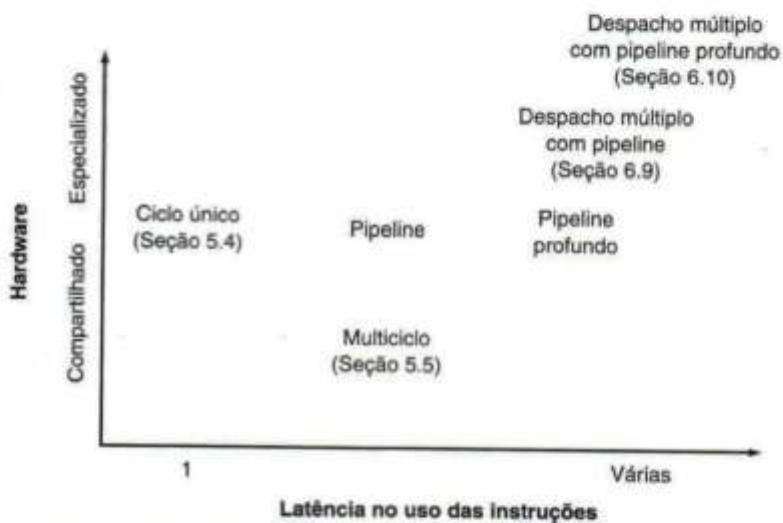


FIGURA 6.53 O relacionamento básico entre os caminhos de dados na Figura 6.52. Observe que o eixo x é “Latência no uso das instruções”, que é o que determina a facilidade de manter o pipeline cheio. O caminho de dados em pipeline aparece como vários ciclos de clock para a latência da instrução, pois o tempo de execução de uma instrução não é mais curto; é a vazão da instrução melhorada.

A técnica de pipelining e o despacho múltiplo tentam explorar o paralelismo no nível de instrução. A presença de dependências de dados e o controle, que podem se tornar hazards, são as principais limitações para a exploração do paralelismo. Escalonamento e especulação, ambos no hardware e no software, são as principais técnicas utilizadas para reduzir o impacto das dependências sobre o desempenho.

A passagem para pipelines maiores, despacho de instruções múltiplas e escalonamento dinâmico em meados da década de 1990 ajudou a sustentar os 60% de aumento de desempenho dos processadores por ano, do qual temos nos beneficiado desde o inicio da década de 1980. No passado, parecia que a escolha era entre os processadores com taxa de clock mais alta e os processadores escalares mais sofisticados. Como vimos, o Pentium 4 combina ambos e consegue um desempenho marcante.

Com avanços marcantes no processamento, a Lei de Amdahl sugere que outra parte do sistema se tornará o gargalo. Esse gargalo é o assunto do próximo capítulo: o sistema de memória.

Uma alternativa a empurrar uniprocessadores para explorar o paralelismo automaticamente no nível de instrução é experimentar os multiprocessadores, que exploram o paralelismo em níveis muito maiores. O processamento em paralelo é o assunto do Capítulo 9, que aparece no CD.

6.13

Perspectiva histórica e leitura adicional

Esta seção, que aparece no CD, discute a história dos primeiros processadores em pipeline, os superescalares mais antigos, o desenvolvimento de técnicas para execuções fora de ordem e especulativas, além de desenvolvimentos importantes na tecnologia de compiladores que acompanha tudo isso.

6.14

Exercícios

6.1 [5] <§6.1> Se o tempo para a operação da ALU puder ser encurtado para 25% (em comparação com a descrição na Figura 6.2,), então

- Isso afetará o ganho de velocidade obtido pela técnica de pipelining? Nesse caso, em quanto? Caso contrário, por quê?
- E se a operação da ALU agora exigir 25% mais tempo?

6.2 [10] <§6.1> Uma arquiteta de computadores precisa projetar o pipeline de um novo microprocessador. Ela tem um núcleo de um programa de exemplo com 10^6 instruções. Cada instrução exige 100ps para terminar.

- Quanto tempo será necessário para executar esse núcleo de programa em um processador sem pipeline?
- O microprocessador mais moderno tem cerca de 20 estágios de pipeline. Suponha que ele tenha um pipeline perfeito. Quanto é o ganho de velocidade conseguido em comparação com o processador sem pipeline?
- O pipelining real não é perfeito, pois sua implementação introduz algum overhead por estágio do pipeline. Esse overhead afetará a latência das instruções, a vazão das instruções ou ambos?

6.3 [5] <§6.1> Usando um desenho semelhante ao da Figura 6.5, mostre os caminhos de forwarding necessários para executar as quatro instruções a seguir:

```
add $3, $4, $6
sub $5, $3, $2
lw $7, 100($5)
add $8, $7, $2
```

6.4 [10] <§6.1> Identifique todas as dependências de dados no código a seguir. Quais dependências são hazards de dados que serão resolvidos por meio de forwarding? Quais dependências são hazards de dados que causarão stall?

```

add $3, $4, $2
sub $5, $3, $1
lw $6, 200($3)
add $7, $3, $6

```

6.5 [5] <§6.1> ■ Aprofundando o aprendizado: Delayed Branches

6.6 [10] <§6.2> Usando a Figura 6.22 como um guia, use canetas ou marcadores coloridos para mostrar quais partes do caminho de dados estão ativos e quais estão inativos em cada um dos cinco estágios da instrução sw. Sugerimos usar cinco cópias da Figura 6.22 para responder a esse exercício. (Concedemos permissão para violar a Lei de Direitos Autorais para realizar os exercícios nos Capítulos 5 e 6!) Não se esqueça de incluir uma legenda para explicar seu esquema de cores.

6.7 [5] <§6.2> ■ Aprofundando o aprendizado: Compreendendo pipelines por meio de desenhos

6.8 [5] <§6.2> ■ Aprofundando o aprendizado: Compreendendo pipelines por meio de desenhos

6.9 [15] <§6.2> ■ Aprofundando o aprendizado: Compreendendo pipelines por meio de desenhos

6.10 [5] <§6.2> ■ Aprofundando o aprendizado: Registradores em pipeline

6.11 [15] <§§4.8, 6.2> ■ Aprofundando o aprendizado: Pipelining de ponto flutuante

6.12 [15] <§6.3> A Figura 6.37 e a Figura 6.35 são dois estilos diferentes para desenhar pipelines. Para ter certeza de que você entende o relacionamento entre esses dois estilos, desenhe as informações contidas nas Figuras 6.31 a 6.35, usando o estilo da Figura 6.37. Destaque as partes ativas dos caminhos de dados na figura.

6.13 [20] <§6.3> A Figura 6.14.10 é semelhante à Figura 6.14.7, da Seção ■ “Aprofundando o aprendizado”, mas as instruções não estão identificadas. Determine o máximo que você puder sobre as cinco instruções nos cinco estágios do pipeline. Se você não puder preencher um campo de uma instrução, explique por quê. Para alguns campos, será mais fácil decodificar as instruções de máquina para assembly, usando como referências a Figura 3.18 e a Figura A.10.2. Para outros campos, será mais fácil examinar os valores dos sinais de controle, usando as Figuras 6.26 a 6.28 como referências. Você pode ter de examinar cuidadosamente as Figuras 6.14.5 a 6.14.9 para entender como as coleções de valores de controle são apresentadas (ou seja, o bit mais à esquerda em um ciclo se tornará o bit mais significativo no outro ciclo). Por exemplo, o valor de controle EX para a instrução subtract, 1100, calculado durante o estágio ID do ciclo 3 na Figura 6.14.6, torna-se três valores separados especificando RegDst (1), OpALU (10) e OrigALU (0) no ciclo 4.

6.14 [40] <§6.3> O trecho de código a seguir é executado usando-se o pipeline mostrado na Figura 6.30:

```

lw $5, 40($2)
add $6, $3, $2
or $7, $2, $1
and $8, $4, $3
sub $9, $2, $1

```

No ciclo 5, imediatamente antes que as instruções sejam executadas, o estado do processador é o seguinte:

- O PC tem o valor 100_{dec} , o endereço da instrução sub.
- Cada registrador possui o valor inicial 10_{dec} mais o número do registrador (por exemplo, o registrador \$8 tem o valor inicial 18_{dec}).
- Cada word da memória acessada como dado tem o valor inicial 1000_{dec} mais o endereço em bytes da word (por exemplo, Memória[8] tem o valor inicial 1008_{dec}).

Determine o valor de cada campo nos quatro registradores de pipeline do ciclo 5.

6.15 [20] <§6.3> ■ Aprofundando o aprendizado: Rotulando diagramas de pipeline com controle

6.16 [20] <§6.4> ■ **Aprofundando o aprendizado:** Ilustrando diagramas com forwarding

6.17 [5] <§§6.4, 6.5> Considere a execução do código a seguir no caminho de dados em pipeline da Figura 6.36:

add	\$2,	\$3,	\$1
sub	\$4,	\$3,	\$5
add	\$5,	\$3,	\$7
add	\$7,	\$6,	\$1
add	\$8,	\$2,	\$6

Ao final do quinto ciclo da execução, quais registradores estão sendo lidos e qual registrador será escrito?

6.18 [5] <§§6.4, 6.5> Com relação ao programa no Exercício 6.17, explique o que a unidade de forwarding está fazendo durante o quinto ciclo da execução. Se quaisquer comparações estiverem sendo feitas, mencione-as.

6.19 [5] <§§6.4, 6.5> Com relação ao programa no Exercício 6.17, explique o que a unidade de detecção de hazards está fazendo durante o quinto ciclo da execução. Se quaisquer comparações estiverem sendo feitas, mencione-as.

6.20 [20] <§§6.4, 6.5> ■ **Aprofundando o aprendizado:** Forwarding na memória

6.21 Temos um programa de 10^3 instruções no formato de “lw, add, lw, add, ...” A instrução add depende (e só depende) da instrução lw imediatamente antes dela. A instrução lw também depende (e só depende) da instrução add imediatamente antes dela. Se o programa for executado no caminho de dados em pipeline da Figura 6.36:

- Qual seria o CPI real?
- Sem o forwarding, qual seria o CPI real?

6.22 [5] <§§6.4, 6.5> Considere a execução do seguinte código no caminho de dados em pipeline da Figura 6.36:

lw	\$4, 100(\$2)
sub	\$6, \$4, \$3
add	\$2, \$3, \$5

Quantos ciclos serão necessários para executar esse código? Desenhe um diagrama como o da Figura 6.34, que ilustre as dependências que precisam ser resolvidas, e ofereça outro diagrama como o da Figura 6.35, que ilustre como o código realmente será executado (incorporando quaisquer stalls ou forwardings) a fim de resolver os problemas identificados.

6.23 [15] <§6.5> Liste todas as entradas e saídas da unidade de forwarding da Figura 6.36. Dê os nomes, o número de bits e um uso resumido para cada entrada e saída.

6.24 [20] <§6.5> ■ **Aprofundando o aprendizado:** Ilustrando diagramas com forwarding e stalls

6.25 [20] <§6.5> ■ **Aprofundando o aprendizado:** Impacto sobre o forwarding de mover-lo para o estágio ID

6.26 [15] <§§6.2-6.5> ■ **Aprofundando o aprendizado:** Impacto do modo de endereçamento de memória sobre o pipeline

6.27 [10] <§§6.2-6.5> ■ **Aprofundando o aprendizado:** Impacto das operações aritméticas com operandos na memória sobre o pipeline

6.28 [30] <§6.5, Apêndice C> ■ **Aprofundando o aprendizado:** Projeto de hardware da unidade de forwarding

6.29 [1 semana] <§§6.4, 6.5> Usando o simulador fornecido com este livro, reúna estatísticas sobre hazards de dados para um programa em C (fornecido pelo instrutor ou com o software). Você es-

creverá uma sub-rotina que receberá a instrução a ser executada, e essa rotina precisa modelar o pipeline de cinco estágios nesse capítulo. Faça com que seu programa colete as seguintes estatísticas:

- Número de instruções executadas.
- Número de hazards de dados não resolvidos pelo forwarding e número de hazards resolvidos pelo forwarding.
- Se o compilador MIPS C que você está usando despachar instruções *nop* para evitar hazards, conte também o número de instruções *nop*.

Supondo que os acessos à memória sempre levam 1 ciclo de clock, calcule o número médio de ciclos de clock por instrução. Classifique as instruções *nop* como stalls inseridos pelo software, depois subtraia-os do número de instruções executadas no cálculo do CPI.

6.30 [7] <§§6.4, 6.5> No exemplo da página 320, vimos que a vantagem no desempenho do projeto multiciclo era limitada pelo tempo maior exigido para acessar a memória *versus* o uso da ALU. Suponha que o acesso à memória passou a levar 2 ciclos de clock. Encontre o desempenho relativo dos projetos de ciclo único e multiciclo. Nos próximos exercícios, estendemos isso para o projeto em pipeline, que exige muito mais trabalho!

6.31 [10] <§6.6> ■ **Aprofundando o aprendizado:** Codificando com moves condicionais

6.32 [10] <§6.6> ■ **Aprofundando o aprendizado:** Vantagem no desempenho do move condicional

6.33 [20] <§§6.2-6.6> No exemplo da página 320, vimos que a vantagem no desempenho dos projetos de multiciclo e pipeline era limitada pelo maior tempo exigido para o acesso à memória em relação ao uso da ALU. Suponha que o acesso à memória passe a levar 2 ciclos de clock. Desenhe o pipeline modificado. Liste todas as novas situações de forwarding possíveis e todos os novos hazards possíveis, com sua extensão.

6.34 [20] <§§6.2-6.6> Refaça o exemplo da página 320 usando o pipeline reestruturado do Exemplo 6.33, para comparar o ciclo único e o multiciclo. Para os desvios, considere a mesma previsão de exatidão, mas aumente a penalidade conforme a necessidade. Para loads, considere que as instruções subsequentes dependem da leitura com uma probabilidade de 1/2, 1/4, 1/8, 1/16 e assim por diante. Ou seja, a instrução duas posições após um load tem uma probabilidade de 25% de usar o resultado do load como uma de suas origens. Ignorando quaisquer outros hazards de dados, encontre o desempenho relativo do projeto em pipeline para o projeto de ciclo único com o pipeline reestruturado.

6.35 [10] <§§6.4-6.6> Conforme indicado na página 314, mover a comparação de desvio para cima, até o estágio de ID, ocasiona uma oportunidade tanto para o forwarding quanto para hazards que não podem ser resolvidos pelo forwarding. Indique um conjunto de sequências de código que mostrem os caminhos possíveis que são exigidos para o forwarding e os hazards que precisam ser detectados, considerando apenas um dos dois operandos. O número de casos deverá ser igual ao tamanho máximo do hazard se não houvesse forwarding.

6.36 [15] <§6.6> Temos um núcleo de programa consistindo em cinco desvios condicionais. O núcleo do programa será executado milhares de vezes. A seguir estão os resultados de cada desvio para uma execução do núcleo do programa (T para tomado, N para não tomado).

Branch 1: T-T-T

Branch 2: N-N-N-N

Branch 3: T-N-T-N-T-N

Branch 4: T-T-T-N-T

Branch 5: T-T-N-T-T-N-T

Considere que o comportamento de cada desvio permanece o mesmo para cada execução do núcleo do programa. Para esquemas dinâmicos, suponha que cada desvio tenha seu próprio buffer de previsão e que cada buffer seja inicializado com o mesmo estado antes de cada execução. Liste as previsões para os seguintes esquemas de previsão de desvios:

- a. Sempre tomado
- b. Sempre não tomado
- c. Previsor de 1 bit, inicializado para prever “tomado”
- d. Previsor de 2 bits, inicializado para prever “tomado” levemente Qual é a exatidão das previsões?

6.37 [10] <§§6.4-6.6> Esboce todas as vias de forwarding para as entradas de desvio e mostre quando elas precisam ser habilitadas (como fizemos na página 309).

6.38 [10] <§§6.4-6.6> Escreva a lógica para detectar quaisquer hazards sobre as origens de desvios, como fizemos na página 309.

6.39 [10] <§§6.4-6.6> O exemplo na página 284 mostra como *maximizar* o desempenho em nosso caminho de dados em pipeline com forwarding e stalls no uso após um load. Reescreva o código a seguir para *minimizar* o desempenho sobre esse caminho de dados – ou seja, reordene as instruções de modo que essa seqüência utilize *mais* ciclos de clock para executar, enquanto ainda obtém o mesmo resultado.

```
lw    $2, 100($6)
lw    $3, 200($7)
add  $4, $2, $3
add  $6, $3, $5
sub  $8, $4, $6
lw    $7, 300($8)
beq  $7, $8, Loop
```

6.40 [20] <§6.6> Considere o caminho de dados em pipeline na Figura 6.54. Uma tentativa de fazer um flush e uma tentativa de fazer um stall podem ocorrer simultaneamente? Se for possível, elas resultam em conflito de ações e/ou cooperação de ações? Se houver cooperação de ações, como elas funcionam juntas? Se houver conflito de ações, quais devem ter prioridade? Existe alguma mudança simples que possa ser feita no caminho de dados para garantir a prioridade necessária? Você pode considerar a seqüência de código a seguir para ajudar na resposta a essa pergunta:

```
beq $1, $2, ALVO # considere que o desvio foi tomado
lw  $3, 40($4)
add $2, $3, $4
sw  $2, 40($4)
ALVO: or $1, $1, $2
```

6.41 [15] <§§ 6.4, 6.7> O Verilog para a implementação do forwarding na Figura 6.7.2 não considera o forwarding de um resultado o valor a ser armazenado por uma instrução SW. Acrescente isso ao código Verilog.

6.42 [5] <§§6.5, 6.7> O Verilog para a implementação do forwarding na Figura 6.7.3 não considera o forwarding de um resultado para usar em um cálculo de endereço. Faça esse acréscimo simples ao código do Verilog.

6.43 [15] <§§6.6,6.7> O código Verilog para implementar a detecção de hazards de desvio e stalls na Figura 6.7.3 não detecta a possibilidade de hazards de dados para os dois registradores origem de uma instrução BEQ. Estenda o Verilog da Figura 6.7.3 para lidar com todos os hazards de dados para os operandos do desvio. Escreva a lógica de forwarding e stall necessária para completar os desvios durante o estágio ID.

6.44 [10] <§§6.6, 6.7> Reescreva o código Verilog da Figura 6.7.3 para implementar uma estratégia de delayed branch.

6.45 [20] <§§6.6, 6.7> Reescreva o código Verilog da Figura 6.7.3 para implementar um buffer de destino de desvios. Considere que o buffer é implementado com um módulo tendo a seguinte definição:

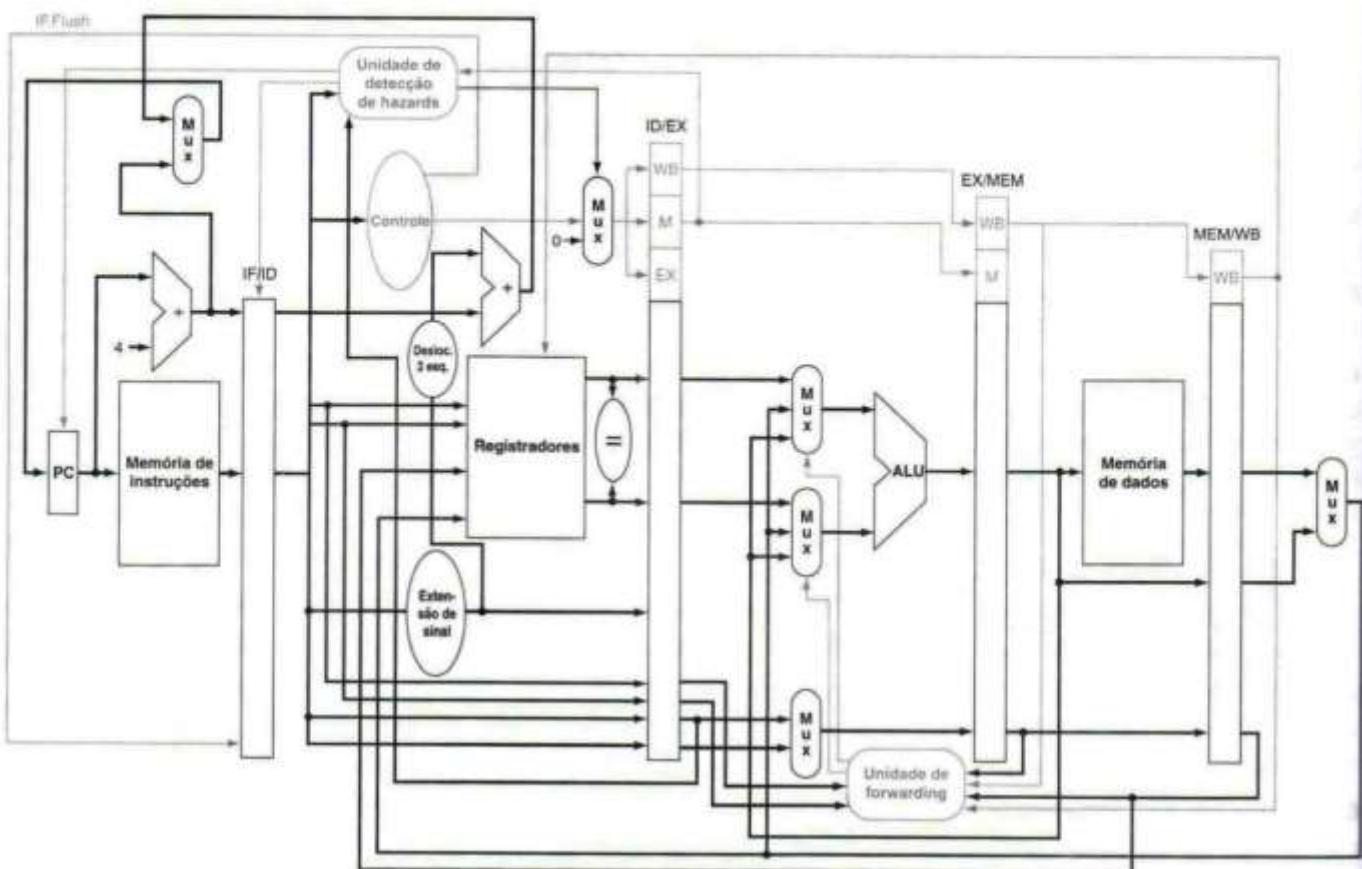


FIGURA 6.54 Caminho de dados para desvio, incluindo o hardware para fazer flush na instrução que vem após o desvio.

Essa otimização move a decisão do desvio do quarto estágio do pipeline para o segundo; somente uma instrução que vem após o desvio estará no pipeline nesse momento. A linha de controle IF.Flush transforma a instrução buscada em um nop, zerando o registrador de pipeline IF/ID. Embora a linha flush apareça vinda da Unidade de controle nessa figura, na realidade, ela vem do hardware que determina se um desvio foi tomado, rotulado com um sinal de igual à direita dos registradores no estágio ID. Os multiplexadores de forwarding (mux) e os caminhos também precisam ser acrescentados a esse estágio, mas não aparecem para simplificar a figura.

```
module PredictPC (currentPC,nextPC,miss,update.destination);
    input currentPC,
          update, // true se previsão anterior não disponível ou incorreta
          destination; // usado com update para corrigir uma previsão
    output nextPC, // retorna próximo PC se a previsão for exata
           miss; // true significa nenhuma previsão no buffer
endmodule;
```

Não se esqueça de acomodar todas as três possibilidades: uma previsão correta, uma falha no buffer (ou seja, miss = true) e uma previsão incorreta. Nos dois últimos casos, também é preciso atualizar a previsão.

6.46 [1 mês] <§§5.4, 6.3-6.8> Se você tem acesso a um sistema de simulação como o Verilog ou o ViewLogic, primeiro crie o caminho de dados e controle de ciclo único do Capítulo 5. Depois, evoluia esse projeto para uma organização em pipeline, como fizemos neste capítulo. Não se esqueça de executar os programas do MIPS em cada etapa para garantir que seu projeto refinado continua a operar corretamente.

6.47 [10] <§6.9> O código a seguir foi desdoblado uma vez, mas ainda não foi escalonado. Suponha que o índice do loop seja um múltiplo de dois (ou seja, \$10 é um múltiplo de oito):

```

Loop:  lw    $2,   0($10)
       sub   $4,   $2, $3
       sw    $4,   0($10)
       lw    $5,   4($10)
       sub   $6,   $5, $3
       sw    $6,   4($10)
       addi  $10,  $10, 8
       bne   $10,  $30, Loop
    
```

Escalone esse código para uma execução rápida no pipeline padrão do MIPS (considere que ele admite a instrução `addi`). Suponha inicialmente que `$10` é 0 e `$30` é 400, e que os desvios são resolvidos no estágio MEM. Como o código escalonado se compara com o código não escalonado original?

6.48 [20] <§6.9> Este exercício é semelhante ao Exercício 6.47, exceto que, desta vez, o código deve ser desdobrado duas vezes (criando três cópias do código). Contudo, não se sabe que o índice do loop é um múltiplo de três, e, por isso, você precisará inventar um meio de garantir que o código ainda será executado corretamente. (Dica: considere a inclusão de algum código no início ou no final do loop, que cuide dos casos não tratados pelo loop.)

6.49 [20] <§6.9> Usando o código no Exercício 6.47, desdobre o código quatro vezes e escalone-o para a versão de despacho múltiplo estático do processador MIPS descrito nas páginas 329-330. Você pode considerar que o loop é executado para um múltiplo de quatro vezes.

6.50 [10] <§§6.1-6.9> À medida que a tecnologia ocasiona recursos com tamanhos cada vez menores, os fios se tornam relativamente mais lentos (em comparação com a lógica). À medida que a lógica se torna mais rápida com o encolhimento do tamanho dos recursos e o aumento das velocidades de clock, os stalls no fio consomem mais ciclos de clock. É por isso que o Pentium 4 possui vários estágios de pipeline dedicados a transferir dados ao longo dos fios de uma parte do pipeline para outra. Quais são as desvantagens de ter de incluir estágios de pipe para os stalls no fio?

6.51 [30] <§6.10> Novos processadores são introduzidos mais rapidamente do que novas versões de livros-texto. Para manter seu livro-texto atualizado, investigue alguns dos desenvolvimentos mais recentes nessa área e escreva uma seção “Detalhamento” de uma página para inserir no final da Seção 6.10. Como ponto de partida, use a World Wide Web para explorar as características dos processadores mais recentes da Intel e da AMD.

§6.1: página 289, 1. Stall no resultado de LW. 2. Fazer forwarding com o resultado de ADD. 3. Não é necessário forwarding nem stall.

§6.2: página 299, afirmações 2 e 5 estão corretas; o restante está incorreto.

§6.6: página 321, 1. desvio não tomado. 2. Previsão tomada. 3. Previsão dinâmica.

§6.7, ■: página 6.7-3, afirmações 1 e 3 são verdadeiras.

§6.7, ■: página 6.7-7, somente a afirmação 3 é completamente precisa.

§6.8: página 326, somente a número 4 é totalmente precisa. A afirmação 2 é parcialmente precisa.

§6.9: página 337, especulação: ambos; buffer de reordenação: hardware; renomeação de registradores: ambos; execução fora de ordem: hardware; predição: software; previsão de desvios: ambos; VLIW: software; superescalar: hardware; EPIC: ambos, pois existe suporte substancial do hardware; despacho múltiplo: ambos; escalonamento dinâmico: hardware.

§6.10: página 340, todas as afirmações são falsas.

**Respostas
das Seções
“Verifique
você mesmo”**

Computadores no mundo real

Comunicação em massa sem gatekeepers

Problema a solucionar: oferecer à sociedade fontes de notícias e opiniões além daquelas encontradas na mídia de massa tradicional.

Solução: use a Internet e a World Wide Web para selecionar e publicar fontes de notícias não tradicionais e não locais.

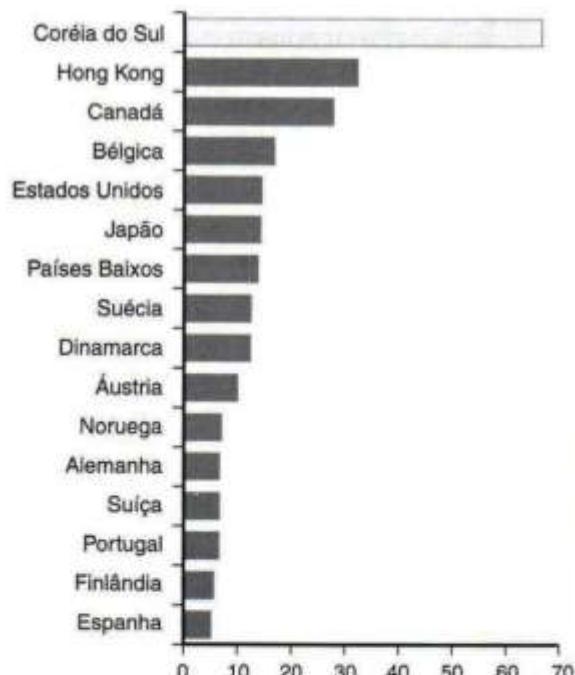
A Internet mantém a promessa de permitir que os cidadãos se comuniquem sem que a informação primeiramente seja interpretada pela mídia de massa tradicional, como televisão, jornais e revistas. Para ver qual poderia ser o futuro, poderíamos examinar países que possuem acesso generalizado e de alta velocidade à Internet.

Um deles é a Coréia do Sul. Em 2002, 68% das casas sul coreanas tinha acesso de banda larga, em comparação com os 15% nos Estados Unidos e 8% na Europa ocidental. (Banda larga geralmente tem velocidades de DSL ou cabo, cerca de 300 a 1.000Kbps.) O motivo principal para a maior penetração é que 70% das famílias moram em grandes cidades, e quase metade está em apartamentos. Logo, o setor de telecomunicações coreano conseguiu oferecer rapidamente a banda larga a 90% dos lares.

Qual foi o impacto do acesso geral de alta velocidade sobre a sociedade coreana? Os sites de notícias da Internet se tornaram extremamente populares. Um exemplo é o OhMyNews, que publi-

ca artigos de *qualquer um* após primeiro verificar se os fatos no artigo estão corretos.

Muitos acreditam que os serviços de notícias na Internet influenciaram o resultado da eleição presidencial coreana de 2002. Primeiro, eles incentivaram os mais jovens a votar. Segundo, o candidato vencedor defendeu políticos que estavam mais próximos daqueles que são populares nos serviços de notícias da Internet. Juntos, eles



Porcentagem de lares com conexões de banda larga por país em 2002. Fonte: The Yankee Group, Boston.

contornaram a desvantagem que a maioria das organizações da mídia dava ao seu oponente.

O Google News é outro exemplo de acesso não tradicional a notícias, que vai além da mídia de massa de um país. Ele procura serviços de notícias internacionais em busca de tópicos, e depois os resume e apresenta por popularidade. Em vez de deixar a decisão sobre quais artigos devem estar nas manchetes para os redatores de jornais locais, a mídia do mundo inteiro decide. Além disso, oferecendo links para histórias de muitos países, o leitor tem uma visão internacional, e não local. Ele também é atualizado muitas vezes por dia, diferente de um jornal diário. A figura a seguir compara a página de manchete do *New York Times* com o site Google News no mesmo dia.

O grande impacto dessas tecnologias nos lembra de que os engenheiros de computadores possuem responsabilidades com suas comu-

nidades. Temos de estar cientes dos valores sociais com relação a privacidade, segurança, liberdade de expressão e assim por diante, para garantir que novas inovações tecnológicas melhorem esses valores, em vez de comprometê-los inadvertidamente.

Para saber mais, veja estas referências

- "Seriously wired", *The Economist*, 17 de abril de 2003.
- OhMyNews, www.ohmynews.com
- Google News, www.news.google.com

<i>New York Times</i> Front Page	Google News
<p>Judge Rules Out a Death Penalty for 9/11 Suspect Rebuke for Justice Dept.</p> <p>Poll Shows Drop in Confidence on Bush Skill in Handling Crises Country on Wrong Track, Says Solid Majority</p> <p>Revised Admission for High Schools City Says Students Will Get First Preference</p> <p>No Illicit Arms Found in Iraq, U.S. Inspector Tells Congress</p> <p>U.S. Practice How to Down Hijacked Jets Coetzee, Writer of Apartheid as Bleak Mirror, Wins Nobel</p> <p>Sexual Accusations Lead to an Apology by Schwarzenegger</p> <p>Interim Chief Accepts Stock Exchange Shift</p> <p>Yankees Even with Twins Agency Warns of Fake Drugs Limbaugh Fallback Position</p>	<p>Top Stories</p> <p>More than 1000 rally behind Schwarzenegger AP - 5 minutes ago</p> <p>Maria Shriver defends husband CNN Can accusations hurt Arnold's campaign? KESQ</p> <p>and 1252 related</p> <p>Bush: Hussein 'A Danger to the World' ABC news - 5 hours ago</p> <p>Bush Stands By Decision Voice of America</p> <p>Hunt for weapons yields no evidence The Canberra Times</p> <p>and 598 related</p> <p>World Stories</p> <p>Defiant UN chief announces rival blueprint for Iraq The Times (UK) - 2 hours ago</p> <p>France, Russia Assail US Draft on Iraq Reuters</p> <p>and 782 related</p>

New York Times versus Google News em 3 de outubro de 2003, às 18 horas. A página de manchete do jornal precisa balancear histórias grandes com notícias nacionais, notícias locais e esportes. O Google News possui muitas histórias por assunto, do mundo inteiro, com links que o leitor pode acompanhar. As histórias do Google variam segundo a hora do dia e, portanto, são mais atualizadas.

7

Grande e Rápida: Explorando a Hierarquia de Memória

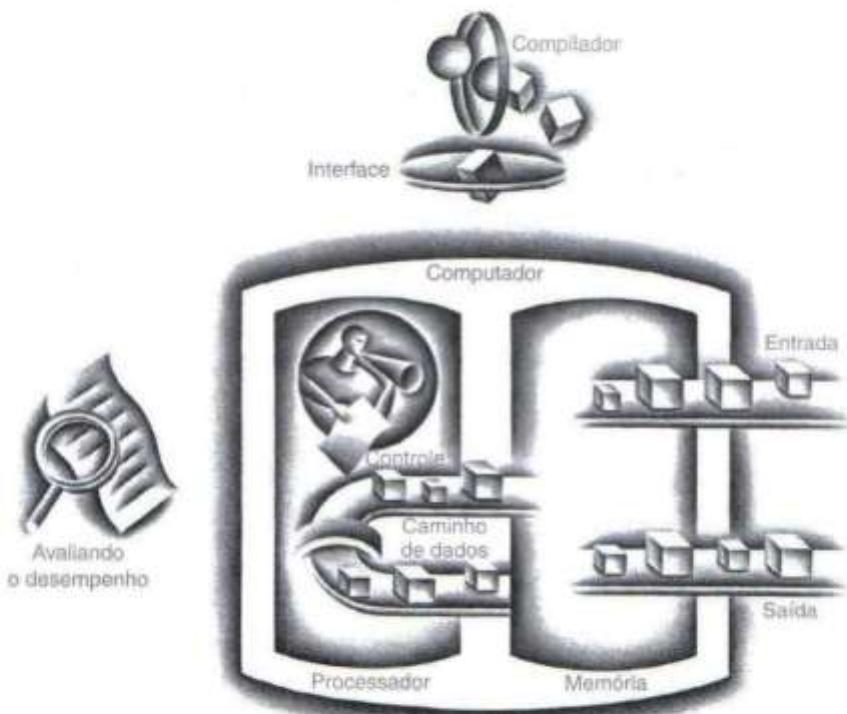
O ideal seria ter uma capacidade de memória infinitamente grande a ponto de qualquer word específica ... estar imediatamente disponível. ... Somos ... forçados a reconhecer a possibilidade de construir uma hierarquia de memórias, cada uma com capacidade maior do que a anterior, mas com acessibilidade menos rápida.

A. W. Burks, H. H. Goldstine e J. von Neumann

Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, 1946

7.1	Introdução	354
7.2	Princípios básicos de cache	358
7.3	Medindo e melhorando o desempenho da cache	371
7.4	Memória virtual	385
7.5	Uma estrutura comum para hierarquias de memória	406
7.6	Vida real: as hierarquias de memória do Pentium P4 e do AMD Opteron	412
7.7	Falácias e armadilhas	415
7.8	Comentários finais	417
■ 7.9	Perspectiva histórica e leitura adicional	419
7.10	Exercícios	419

Os cinco componentes clássicos de um computador



7.1 Introdução

Desde os primeiros dias da computação, os programadores têm desejado quantidades ilimitadas de memória rápida. Os tópicos deste capítulo ajudam os programadores a criar a ilusão de memória rápida ilimitada. Antes de vermos como a ilusão é realmente criada, vamos considerar uma analogia simples que ilustra os princípios e mecanismos-chave que usamos.

Suponha que você fosse um estudante fazendo um trabalho sobre os importantes desenvolvimentos históricos no hardware dos computadores. Você está sentado em uma biblioteca examinando uma pilha de livros retirada das estantes. Você descobre que vários computadores importantes, sobre os quais precisa escrever, são descritos nos livros encontrados, mas não há nada sobre o EDSAC. Então, volta às estantes e procura um outro livro. Você encontra um livro sobre computadores britânicos que fala sobre o EDSAC. Com uma boa seleção de livros sobre a mesa à sua frente, existe uma boa probabilidade de que muitos dos tópicos de que precisa possam ser encontrados neles. Com isso, você pode gastar mais do seu tempo apenas usando os livros na mesa sem voltar às estantes. Ter vários livros na mesa economiza seu tempo em comparação a ter apenas um livro e constantemente precisar voltar às estantes para devolvê-lo e apanhar outro.

O mesmo princípio nos permite criar a ilusão de uma memória grande que podemos acessar tão rapidamente quanto uma memória muito pequena. Assim como você não precisou acessar todos os livros da biblioteca ao mesmo tempo com igual probabilidade, um programa não acessa todo o seu código ou dados ao mesmo tempo com igual probabilidade. Caso contrário, seria impossível tornar rápida a maioria dos acessos à memória e ainda ter memória grande nos computadores, assim como seria impossível você colocar todos os livros da biblioteca em sua mesa e ainda encontrar o desejado rapidamente.

Esse princípio da localidade sustenta a maneira como você fez seu trabalho na biblioteca e o modo como os programas funcionam. O princípio da localidade diz que os programas acessam uma parte relativamente pequena do seu espaço de endereçamento em qualquer instante do tempo, exatamente como você acessou uma parte bastante pequena da coleção da biblioteca. Há dois tipos diferentes de localidade:

■ **Localidade temporal** (localidade no tempo): se um item é referenciado, ele tenderá a ser referenciado novamente em breve. Se você trouxe um livro para sua mesa para examiná-lo, é provável que precise examiná-lo novamente em breve.

■ **Localidade espacial** (localidade no tempo): se um item é referenciado, os itens cujos endereços estão próximos tenderão a ser referenciados em breve. Por exemplo, ao trazer o livro sobre os primeiros computadores ingleses para pesquisar sobre o EDSAC, você também percebeu que havia outro livro ao lado dele na estante sobre computadores mecânicos; então, resolver trazer também esse livro, no qual, mais tarde, encontrou algo útil. Os livros sobre o mesmo assunto são colocados juntos na biblioteca para aumentar a localidade espacial. Veremos como a localidade espacial é usada nas hierarquias de memória um pouco mais adiante neste capítulo.

Assim como os acessos aos livros na estante exibem naturalmente a localidade, a localidade nos programas surge de estruturas de programa simples e naturais. Por exemplo, a maioria dos programas contém loops e, portanto, as instruções e os dados provavelmente são acessados de modo repetitivo, mostrando altas quantidades de localidade temporal. Como em geral as instruções são acessadas seqüencialmente, os programas mostram alta localidade espacial. Os acessos a dados também exibem uma localidade espacial natural. Por exemplo, os acessos aos elementos de um array ou de um registro terão altos índices de localidade espacial.

Tiramos vantagem do princípio da localidade implementando a memória de um computador como uma **hierarquia de memória**. Uma hierarquia de memória consiste em múltiplos níveis de memória com diferentes velocidades e tamanhos. As memórias mais rápidas são mais caras por isso que as memórias mais lentas e, portanto, são menores.

localidade temporal O princípio em que se um local de dados é referenciado, então, ele tenderá a ser referenciado novamente em breve.

localidade espacial O princípio da localidade em que se um local de dados é referenciado, então, os dados com endereços próximos tenderão a ser referenciados em breve.

hierarquia de memória Uma estrutura que usa múltiplos níveis de memórias; conforme a distância da CPU aumenta, o tamanho das memórias e o tempo de acesso também aumentam.

Hoje, existem três tecnologias principais usadas na construção das hierarquias de memória. A memória principal é implementada por meio de DRAM (Dynamic Random Access Memory), enquanto os níveis mais próximos do processador (caches) usam SRAM (Static Random Access Memory). A DRAM é mais barata por bit do que a SRAM, embora seja substancialmente mais lenta. A diferença de preço ocorre porque a DRAM usa significativamente menos área por bit de memória e as DRAMs, portanto, têm maior capacidade para a mesma quantidade de silício; a diferença de velocidade ocorre devido a diversos fatores descritos na Seção B.8 do Apêndice B. A terceira tecnologia, usada para implementar o maior e mais lento nível na hierarquia, é o disco magnético. O tempo de acesso e o preço por bit variam muito entre essas tecnologias, como mostra a tabela a seguir, usando valores típicos em 2004, quando este livro foi escrito:

Tecnologia de memória	Tempo de acesso típico	US\$ por GB em 2004
SRAM	0,5 a 5ns	4000 a 10.000
DRAM	50 a 70ns	100 a 200
Disco magnético	5.000.000 a 20.000.000ns	0,50 a 2

Devido a essas diferenças no custo e no tempo de acesso, é vantajoso construir memória como uma hierarquia de níveis. A Figura 7.1 mostra que a memória mais rápida está próxima do processador e a memória mais lenta e barata está abaixo dele. O objetivo é oferecer ao usuário o máximo de memória disponível na tecnologia mais barata, enquanto se fornece acesso na velocidade oferecida pela memória mais rápida.

O sistema de memória é organizado como uma hierarquia: um nível mais próximo do processador em geral é um subconjunto de qualquer nível mais distante, e todos os dados são armazenados no nível mais baixo. Por analogia, os livros em sua mesa formam um subconjunto da biblioteca onde você está trabalhando, que, por sua vez, é um subconjunto de todas as bibliotecas do campus. Além disso, conforme nos afastamos do processador, os níveis levam cada vez mais tempo para serem acessados, exatamente como poderíamos encontrar em uma hierarquia de bibliotecas de campus.

Uma hierarquia de memória pode consistir em múltiplos níveis, mas os dados são copiados apenas entre dois níveis adjacentes ao mesmo tempo, de modo que podemos concentrar nossa atenção em apenas dois níveis. O nível superior – o que está mais perto do processador – é menor e mais rápido (já que usa tecnologia mais cara) do que o nível inferior. A Figura 7.2 mostra que a unidade de informação mínima que pode estar presente ou ausente na hierarquia de dois níveis é denominada um **bloco** ou uma *linha*; em nossa analogia da biblioteca, um bloco de informação seria um livro.

bloco A unidade mínima de informação que pode estar presente ou ausente na hierarquia de dois níveis.

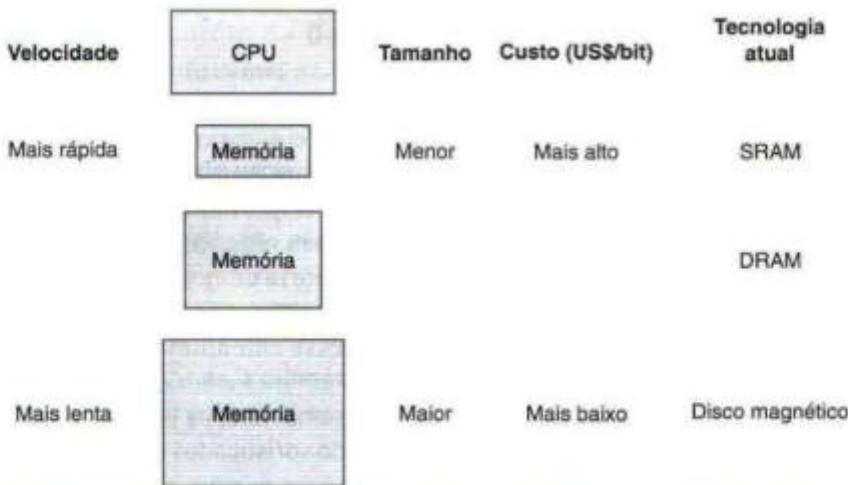


FIGURA 7.1 A estrutura básica de uma hierarquia de memória. Implementando o sistema de memória como uma hierarquia, o usuário tem a ilusão de uma memória que é tão grande quanto o maior nível da hierarquia, mas pode ser acessada como se fosse totalmente construída com a memória mais rápida.

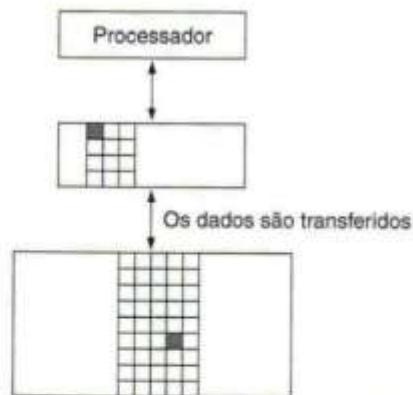


FIGURA 7.2 Cada par de níveis na hierarquia de memória pode ser imaginado como tendo um nível superior e um nível inferior. Dentro de cada nível, a unidade de informação que está presente ou não é chamada de um *bloco*. Em geral, transferimos um bloco inteiro quando copiamos algo entre os níveis.

Se os dados requisitados pelo processador aparecerem em algum bloco no nível superior, isso é chamado um *acerto* (análogo a encontrar a informação em um dos livros em sua mesa). Se os dados não forem encontrados no nível superior, a requisição é chamada uma *falha*. O nível inferior em uma hierarquia é, então, acessado para recuperar o bloco com os dados requisitados. (Continuando com nossa analogia, você vai da sua mesa até as estantes para encontrar o livro desejado.) A **taxa de acertos** é a fração dos acessos à memória encontrados no nível superior; ela normalmente é usada como uma medida do desempenho da hierarquia de memória. A **taxa de falhas** ($1 -$ taxa de acertos) é a proporção dos acessos à memória não encontrados no nível superior.

Como o desempenho é o principal objetivo de ter uma hierarquia de memória, o tempo para servir acertos e falhas é um aspecto importante. O **tempo de acerto** é o tempo para acessar o nível superior da hierarquia de memória, que inclui o tempo necessário para determinar se o acesso é um acerto ou uma falha (ou seja, o tempo necessário para consultar os livros na mesa). A **penalidade de falha** é o tempo para substituir um bloco no nível superior pelo bloco correspondente do nível inferior, mais o tempo para transferir esse bloco para o processador (ou, o tempo para apanhar outro livro das estantes e colocá-lo na mesa). Como o nível superior é menor e construído usando peças de memória mais rápidas, o tempo de acerto será muito menor do que o tempo para acessar o próximo nível na hierarquia, que é o principal componente da penalidade de falha. (O tempo para examinar os livros na mesa é muito menor do que o tempo para se levantar e apanhar um novo livro nas estantes.)

Como veremos neste capítulo, os conceitos usados para construir sistemas de memória afetam muitos outros aspectos de um computador, inclusive como o sistema operacional gerencia a memória e a E/S, como os compiladores geram código e mesmo como as aplicações usam o computador. É claro que como todos os programas gastam muito do seu tempo acessando a memória, o sistema de memória é necessariamente um importante fator na determinação do desempenho. A confiança nas hierarquias de memória para obter desempenho tem significado que os programadores (que costumavam pensar na memória como um dispositivo de armazenamento plano e de acesso aleatório) agora precisam entender as hierarquias de memória para alcançarem um bom desempenho. Para mostrar como esse entendimento é importante, vamos fornecer dois exemplos.

Como os sistemas de memória são essenciais para o desempenho, os projetistas de computadores têm dedicado muita atenção a esses sistemas e desenvolvido sofisticados mecanismos para melhorar o desempenho do sistema de memória. Neste capítulo, veremos as principais idéias conceituais, embora muitas simplificações e abstrações tenham sido usadas para manter o material praticável em tamanho e complexidade. Poderíamos facilmente ter escrito centenas de páginas apenas sobre os sistemas de memória, como dezenas de recentes teses de doutorado têm feito.

taxa de acertos

A proporção dos acessos à memória encontrados em uma cache.

taxa de falhas

A proporção de acessos à memória não encontrados em um nível da hierarquia de memória.

tempo de acerto O tempo necessário para acessar um nível da hierarquia de memória, incluindo o tempo necessário para determinar se o acesso é um acerto ou uma falha.

penalidade de falha O tempo necessário para buscar um bloco de um nível inferior para um nível superior da hierarquia de memória, incluindo o tempo para acessar o bloco, transmiti-lo de um nível para outro e inseri-lo no nível que experimentou a falha.



FIGURA 7.3 Este diagrama mostra a estrutura de uma hierarquia de memória: conforme a distância do processador aumenta, o tamanho também aumenta. Essa estrutura com os mecanismos de operação apropriados permite que o processador tenha um tempo de acesso determinado principalmente pelo nível 1 da hierarquia e ainda tenha uma memória tão grande quanto o nível n . Manter essa ilusão é o assunto deste capítulo. Embora o disco local normalmente seja a parte inferior da hierarquia, alguns sistemas usam fita ou um servidor de arquivos numa rede local como os próximos níveis da hierarquia.

Quais das seguintes afirmações costumam ser verdadeiras?

1. As caches tiram proveito da localidade temporal.
2. Em uma leitura, o valor retornado depende de quais blocos estão na cache.
3. A maioria do custo da hierarquia de memória está no nível mais alto.

Verifique você mesmo

Os programas apresentam localidade temporal (a tendência de reutilizar itens de dados recentemente acessados) e localidade espacial (a tendência de referenciar itens de dados que estão próximos a outros itens recentemente acessados). As hierarquias de memória tiram proveito da localidade temporal mantendo mais próximos do processador os itens de dados acessados mais recentemente. As hierarquias de memória tiram proveito da localidade especial movendo blocos consistindo em múltiplas words contíguas na memória para níveis superiores na hierarquia.

A Figura 7.3 mostra que uma hierarquia de memória usa tecnologias de memória menores e mais rápidas perto do processador. Portanto, os acessos de acerto no nível mais alto da hierarquia podem ser processados rapidamente. Os acessos de falha vão para os níveis mais baixos da hierarquia, que são maiores, porém mais lentos. Se a taxa de acertos for bastante alta, a hierarquia de memória terá um tempo de acesso efetivo próximo ao tempo de acesso do nível mais alto (e mais rápido) e um tamanho igual ao do nível mais baixo (e maior).

Na maioria dos sistemas, a memória é uma hierarquia verdadeira, o que significa que os dados não podem estar presentes no nível i a menos que também estejam presentes no nível $i + 1$.

Colocando em perspectiva

Cache: um lugar seguro para esconder ou guardar coisas.

Webster's New World Dictionary of the American Language, Third College Edition (1988)

7.2

Princípios básicos de cache

Em nosso exemplo da biblioteca, a mesa servia como uma cache – um lugar seguro para guardar coisas (livros) que precisávamos examinar. *Cache* foi o nome escolhido para representar o nível da hierarquia de memória entre o processador e a memória principal no primeiro computador comercial a ter esse nível extra. Hoje, embora permaneça o uso dominante da palavra *cache*, o termo também é usado para referenciar qualquer armazenamento usado para tirar proveito da localidade de acesso. As caches apareceram inicialmente nos computadores de pesquisa no início da década de 1960 e nos computadores de produção mais tarde nessa mesma década; todo computador de uso geral construído hoje, dos servidores aos processadores embutidos de baixa capacidade, possui caches.

Nesta seção, começaremos a ver uma cache muito simples na qual cada requisição do processador é uma word e os blocos também consistem em uma única word. (Os leitores que já estão familiarizados com os fundamentos de cache podem pular para a Seção 7.3.)

A Figura 7.4 mostra essa cache simples, antes e depois de requisitar um item de dados que não está inicialmente na cache. Antes de requisitar, a cache contém uma coleção de referências recentes, X_1 , X_2 , ..., X_{n-1} , e o processador requisita uma word X_n que não está na cache. Essa requisição resulta em uma falha, e a word X_n é trazida da memória para a cache.

Olhando o cenário na Figura 7.4, existem duas perguntas a serem respondidas: como sabemos se o item de dados está na cache? Além disso, se estiver, como encontrá-lo? As respostas a essas duas questões estão relacionadas. Se cada word pode ficar exatamente em um lugar na cache, então, é fácil encontrar a word se ela estiver na cache. A maneira mais simples de atribuir um local na cache para cada word da memória é atribuir um local na cache baseado no *endereço* da word na memória. Essa estrutura de cache é chamada de **mapeamento direto**, já que cada local da memória é mapeado diretamente para um local exato na cache. O mapeamento típico entre endereços e locais de cache para uma cache diretamente mapeada é simples. Por exemplo, quase todas as caches diretamente mapeadas usam o mapeamento

mapeamento

direto Uma estrutura de cache em que cada local da memória é mapeado exatamente para um local na cache.

(Endereço de bloco) módulo (Número de blocos de cache na cache)

Esse mapeamento é atraente porque se o número de entradas na cache for uma potência de dois, então, o módulo pode ser calculado simplesmente usando os \log_2 bits menos significativos (tamanho da cache em blocos); assim, a cache pode ser acessada diretamente com os bits menos significativos.

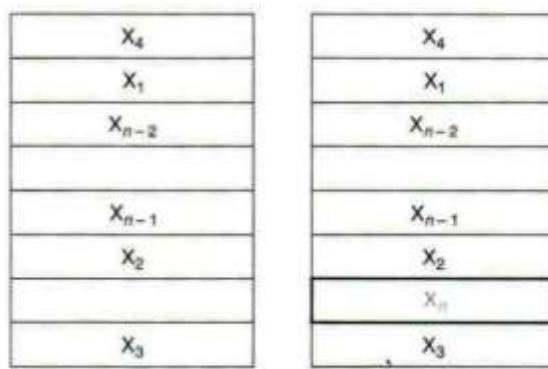


FIGURA 7.4 A cache, imediatamente antes e após uma referência a uma word X_n que não está inicialmente na cache. Essa referência causa uma falha que força a cache a buscar X_n na memória e inseri-la na cache.

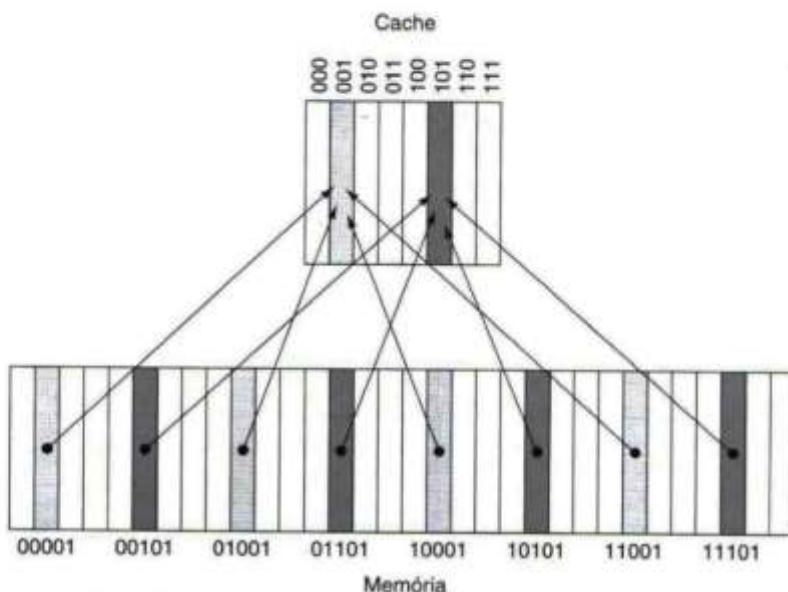


FIGURA 7.5 Uma cache diretamente mapeada com oito entradas mostrando os endereços das words de memória entre 0 e 31 que são mapeadas para os mesmos locais de cache. Como há oito words na cache, um endereço X é mapeado para a word de cache X módulo 8. Ou seja, os $\log_2(8) = 3$ bits menos significativos são usados como o índice da cache. Assim, os endereços 00001_{bin}, 01001_{bin}, 10001_{bin} e 11001_{bin} são todos mapeados para a entrada 001_{bin} da cache, enquanto os endereços 00101_{bin}, 01101_{bin}, 10101_{bin} e 11101_{bin} são todos mapeados para a entrada 101_{bin} da cache.

Por exemplo, a Figura 7.5 mostra como os endereços de memória entre 1_{dec} (00001_{bin}) e 29_{dec} (11101_{bin}) são mapeados para as posições 1_{dec} (001_{bin}) e 5_{dec} (101_{bin}) em uma cache diretamente mapeada de oito words.

Como cada local da cache pode armazenar o conteúdo de diversos locais da memória diferentes, como podemos saber se os dados na cache correspondem a uma word requisitada? Ou seja, como sabemos se uma word requisitada está na cache ou não? Respondemos a essa pergunta incluindo um conjunto de **tags** na cache. As tags contêm as informações de endereço necessárias para identificar se uma word na cache corresponde à word requisitada. A tag precisa apenas conter a parte superior do endereço, correspondente aos bits que não são usados como índice para a cache. Por exemplo, na Figura 7.5, precisamos apenas ter os 2 bits mais significativos dos 5 bits de endereço na tag, já que o campo índice com os 3 bits menos significativos do endereço seleciona o bloco. Excluímos os bits de índice porque eles são redundantes, uma vez que, por definição, o campo índice de cada endereço precisa ter o mesmo valor.

Também precisamos de uma maneira de reconhecer se um bloco de cache não possui informações válidas. Por exemplo, quando um processador é iniciado, a cache não tem dados válidos e os campos de tag não terão significado. Mesmo após executar muitas instruções, algumas entradas de cache podem ainda estar vazias, como na Figura 7.4. Portanto, precisamos saber se a tag deve ser ignorada para essas entradas. O método mais comum é incluir um **bit de validade** para indicar se uma entrada contém um endereço válido. Se o bit não estiver ligado, não pode haver uma correspondência para esse bloco.

Para o restante desta seção, vamos nos concentrar em explicar como funcionam as leituras em uma cache e como o controle de cache funciona para leituras. Em geral, a manipulação de leituras é um pouco mais simples do que a manipulação de escritas, já que as leituras não precisam mudar o conteúdo da cache. Após vermos os aspectos básicos de como as leituras funcionam e como as falhas de cache podem ser tratadas, examinaremos os projetos de cache para computadores reais e detalharemos como essas caches manipulam as escritas.

tag Um campo em uma tabela usado para uma hierarquia de memória que contém as informações de endereço necessárias para identificar se o bloco associado na hierarquia corresponde a uma word requisitada.

bit de validade Um campo nas tabelas de uma hierarquia de memória que indica que o bloco associado na hierarquia contém dados válidos.

Acessando uma cache

A Figura 7.6 mostra o conteúdo de uma cache diretamente mapeada de oito words ao responder a uma série de requisições do processador. Como há oito blocos na cache, os 3 bits menos significativos de um endereço fornecem o número do bloco.

Índice	V	Tag	Dados
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. Estado inicial da cache após a inicialização

Índice	V	Tag	Dados
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	S	10 _{bin}	Memória (10110 _{bin})
111	N		

b. Após tratar uma falha no endereço (10110_{bin})

Índice	V	Tag	Dados
000	N		
001	N		
010	S	11 _{bin}	Memória (11010 _{bin})
011	N		
100	N		
101	N		
110	S	10 _{bin}	Memória (10110 _{bin})
111	N		

c. Após tratar uma falha no endereço (11010_{bin})

Índice	V	Tag	Dados
000	S	10 _{bin}	Memória (10000 _{bin})
001	N		
010	S	11 _{bin}	Memória (11010 _{bin})
011	N		
100	N		
101	N		
110	S	10 _{bin}	Memória (10110 _{bin})
111	N		

d. Após tratar uma falha no endereço (10000_{bin})

Índice	V	Tag	Dados
000	S	10 _{bin}	Memória (10000 _{bin})
001	N		
010	S	11 _{bin}	Memória (11010 _{bin})
011	S	00 _{bin}	Memória (00011 _{bin})
100	N		
101	N		
110	S	10 _{bin}	Memória (10110 _{bin})
111	N		

e. Após tratar uma falha no endereço (00011_{bin})

Índice	V	Tag	Dados
000	S	10 _{bin}	Memória (10000 _{bin})
001	N		
010	S	10 _{bin}	Memória (10010 _{bin})
011	S	00 _{bin}	Memória (00011 _{bin})
100	N		
101	N		
110	S	10 _{bin}	Memória (10110 _{bin})
111	N		

f. Após tratar uma falha no endereço (10010_{bin})

FIGURA 7.6 O conteúdo da cache é mostrado para cada requisição de referência que falha, com os campos índice e tag mostrados em binário. A cache inicialmente está vazia, com todos os bits de validade (entrada V da cache) inativos (N). O processador requisita os seguintes endereços: 10110_{bin} (falha), 11010_{bin} (falha), 10110_{bin} (acerto), 11010_{bin} (acerto), 10000_{bin} (falha), 00011_{bin} (falha), 10000_{bin} (acerto) e 10010_{bin} (falha). As figuras mostram o conteúdo da cache após cada falha na seqüência ter sido tratada. Quando o endereço 10010_{bin} (18) é referenciado, a entrada para o endereço 11010_{bin} (26) precisa ser substituída e uma referência a 11010_{bin} causará uma falha subsequente. O campo tag conterá apenas a parte superior do endereço. O endereço completo de uma word contida no bloco de cache i com o campo tag j para essa cache é $j \times 8 + i$ ou, de forma equivalente, a concatenação do campo tag j e o campo índice i . Por exemplo, na cache f anterior, o índice 010 possui tag 10 e corresponde ao endereço 10010.

Aqui está a ação para cada referência:

Endereço decimal da referência	Endereço binário da referência	Acerto ou falha na cache	Bloco de cache atribuído (onde foi encontrado ou inserido)
22	10110 _{bin}	falha (7.6b)	(10110 _{bin} mod 8) = 110 _{bin}
26	11010 _{bin}	falha (7.6c)	(11010 _{bin} mod 8) = 010 _{bin}
22	10110 _{bin}	acerto	(10110 _{bin} mod 8) = 110 _{bin}
26	11010 _{bin}	acerto	(11010 _{bin} mod 8) = 010 _{bin}
16	10000 _{bin}	falha (7.6d)	(10000 _{bin} mod 8) = 000 _{bin}
3	00011 _{bin}	falha (7.6e)	(00011 _{bin} mod 8) = 011 _{bin}
16	10000 _{bin}	acerto	(10000 _{bin} mod 8) = 000 _{bin}
18	10010 _{bin}	falha (7.6f)	(10010 _{bin} mod 8) = 010 _{bin}

Quando a word no endereço 18 (10010_{bin}) é trazida para o bloco de cache 2 (010_{bin}), a word no endereço 26 (11010_{bin}), que estava no bloco de cache 2 (010_{bin}), precisa ser substituída pelos dados recém-requisitados. Esse comportamento permite que uma cache tire proveito da localidade temporal: words recentemente acessadas substituem words menos referenciadas recentemente. Essa situação é análoga a precisar de um livro da estante e não ter mais espaço na mesa para colocá-lo – algum livro precisa ser devolvido à estante. Em uma cache diretamente mapeada, há apenas um lugar para colocar o item recém-requisitado e, portanto, apenas uma escolha do que substituir.

Agora, sabemos onde olhar na cache para cada endereço possível: os bits menos significativos de um endereço podem ser usados para encontrar a entrada de cache única para a qual o endereço poderia ser mapeado. A Figura 7.7 mostra como um endereço referenciado é dividido em

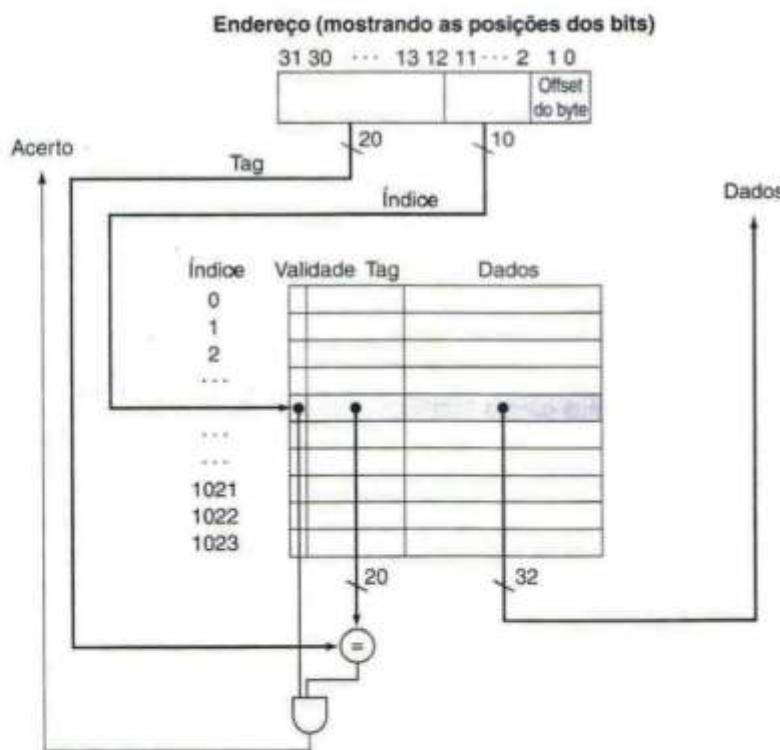


FIGURA 7.7 Para esta cache, a parte inferior do endereço é usada para selecionar uma entrada de cache consistindo em uma word de dados e uma tag. A tag da cache é comparada com a parte superior do endereço para determinar se a entrada na cache corresponde ao endereço requisitado. Como a cache tem 2^{10} (ou 1024) words e um tamanho de bloco de 1 word, 10 bits são usados para indexar a cache, deixando $32 - 10 - 2 = 20$ bits para serem comparados com a tag. Se a tag e os 20 bits superiores do endereço forem iguais e o bit de validade estiver ligado, então, a requisição é um acerto na cache e a word é fornecida para o processador. Caso contrário, ocorre uma falha.

- um índice de cache, usado para selecionar o bloco
- um campo tag, usado para ser comparado com o valor do campo tag da cache

O índice de um bloco de cache, juntamente com o conteúdo da tag desse bloco, especifica de modo único o endereço de memória da word contida no bloco de cache. Como o campo índice é usado como um endereço para acessar a cache e como um campo de n bits possui 2^n valores, o número total de entradas em uma cache diretamente mapeada será uma potência de dois. Na arquitetura MIPS, uma vez que as words são alinhadas como múltiplos de 4 bytes, os 2 bits menos significativos de cada endereço especificam um byte dentro de uma word e, portanto, são ignorados ao selecionar a word no bloco.

O número total de bits necessários para uma cache é uma função do tamanho da cache e do tamanho do endereço, pois a cache inclui o armazenamento para os dados e as tags. O tamanho do bloco mencionado anteriormente era de uma word, mas normalmente é de várias words. Considerando o endereço em bytes de 32 bits, uma cache diretamente mapeada de 2^n blocos de tamanho com 2^m words (2^{m+2} bytes) por bloco exigirá um campo tag cujo tamanho é $32 - (n + m + 2)$ bits, pois n bits são usados para o índice, m bits são usados para a word dentro do bloco e 2 bits são usados para a parte do byte do endereço. O número total de bits em uma cache diretamente mapeada é $2^n \times (\text{tamanho do bloco} + \text{tamanho do tag} + \text{tamanho do campo de validade})$. Como o tamanho do bloco é 2^m words (2^{m+5} bits) e o tamanho do endereço é 32 bits, o número de bits nessa cache é $2^n \times (m \times 32 + (32 - n - 2) + 1) = 2^n \times (m \times 32 + 31 - n - m)$. Entretanto, por convenção de nomeação deve-se excluir o tamanho da tag e do campo de validade e contar apenas o tamanho dos dados.

BITS EM UMA CACHE

EXEMPLO

Quantos bits no total são necessários para uma cache diretamente mapeada com 16KB de dados em blocos de 4 words, considerando um endereço de 32 bits?

RESPOSTA

Sabemos que 16KB são 4K words, o que equivale a 2^{12} words e, com um tamanho de bloco de 4 words (2^2), 2^{10} blocos. Cada bloco possui 4×32 ou 128 bits de dados mais uma tag, que é $32 - 10 - 2 = 2$ bits, mais um bit de validade. Portanto, o tamanho de cache total é

$$2^{10} \times (128 + (32 - 10 - 2 - 2) + 1) = 2^{10} \times 147 = 147\text{Kbits}$$

ou 18,4KB para uma cache de 16KB. Para essa cache, o número total de bits na cache é aproximadamente 1,15 vez o necessário apenas para o armazenamento dos dados.

MAPEANDO UM ENDEREÇO PARA UM BLOCO DE CACHE MULTIWORD

EXEMPLO

Considere uma cache com 64 blocos e um tamanho de bloco de 16 bytes. Para qual número de blocos o endereço em bytes 1200 é mapeado?

RESPOSTA

A fórmula vista na página 358. O bloco é dado por

$$(\text{Endereço do bloco}) \bmod (\text{Número de blocos de cache})$$

onde o endereço do bloco é

$$\frac{\text{Endereço em bytes}}{\text{Bytes por bloco}}$$

Observe que esse endereço de bloco é o bloco contendo todos os endereços entre

$$\left\lfloor \frac{\text{Endereço em bytes}}{\text{Bytes por bloco}} \right\rfloor \times \text{Bytes por bloco}$$

e

$$\left\lfloor \frac{\text{Endereço em bytes}}{\text{Bytes por bloco}} \right\rfloor \times \text{Bytes por bloco} + (\text{Bytes por bloco} - 1)$$

Portanto, com 16 bytes por bloco, o endereço em bytes 1200 é o endereço de bloco

$$\left\lfloor \frac{1200}{16} \right\rfloor = 75$$

que é mapeado para o número de bloco de cache ($75 \bmod 64$) = 11. Na verdade, esse bloco mapeia todos os endereços entre 1200 e 1215.

Blocos maiores exploram a localidade espacial para diminuir as taxas de falhas. Como mostra a Figura 7.8, aumentar o tamanho de bloco normalmente diminui a taxa de falhas. A taxa de falhas pode subir posteriormente se o tamanho de bloco se tornar uma fração significativa do tamanho de cache, uma vez que o número de blocos que pode ser armazenado na cache se tornará pequeno e haverá uma grande competição entre esses blocos. Como resultado, um bloco será retirado da cache antes que muitas de suas words sejam acessadas. Como alternativa, a localidade espacial entre as words em um bloco diminui com um bloco muito grande; assim, os benefícios na taxa de falhas se tornam menores.

Um problema mais sério associado com apenas aumentar o tamanho de bloco é que o custo de uma falha aumenta. A penalidade de falha é determinada pelo tempo necessário para buscar o bloco do próximo nível mais baixo na hierarquia e carregá-lo na cache. O tempo para buscar o bloco possui duas partes: a latência até a primeira word e o tempo de transferência para o restante do bloco. Claramente, a menos que mudemos o sistema de memória, o tempo de transferência – e, portanto, a penali-

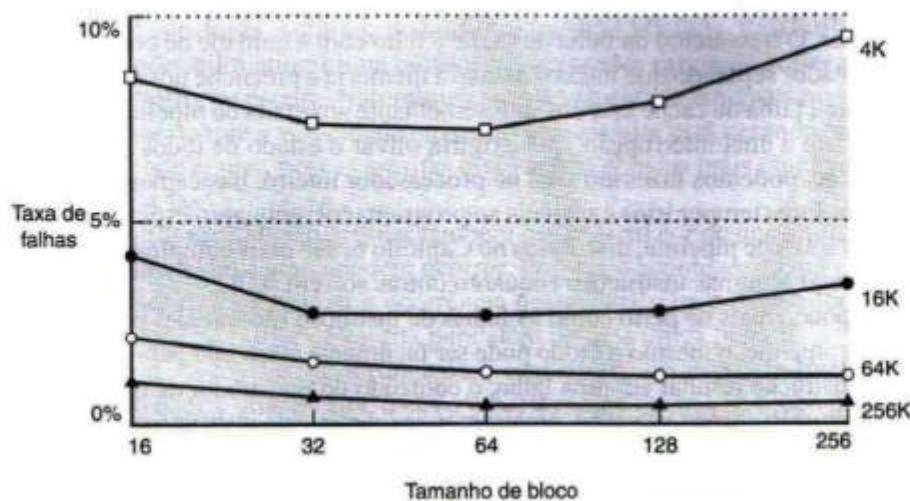


FIGURA 7.8 Taxa de falhas versus tamanho de bloco. Note que a taxa de falhas realmente sobe se o tamanho de bloco for muito grande em relação ao tamanho da cache. Cada linha representa uma cache de tamanho diferente. (Esta figura é independente da associatividade, que será discutida em breve.) Infelizmente, os tracés do SPEC2000 levariam tempo demais se o tamanho de bloco fosse incluído; portanto, esses dados são baseados no SPEC92.

dade de falha – aumentará conforme o tamanho do bloco aumenta. Além disso, o aumento na taxa de falhas começa a decrescer conforme os blocos se tornam maiores. O resultado é que o aumento na penalidade de falha suplanta o decréscimo na taxa de falhas para grandes blocos, diminuindo, assim, o desempenho da cache. Discutiremos esse assunto na próxima seção.

Detalhamento: a principal desvantagem de aumentar o tamanho do bloco é que a penalidade de falha da cache aumenta. Embora seja difícil fazer qualquer coisa sobre o componente de latência da penalidade de falha, podemos ser capazes de ocultar um pouco de transferência de modo que a penalidade de falha seja efetivamente menor. O método mais fácil de fazer isso, chamado *reinício precoce*, é simplesmente retomar a execução assim que a word requisitada do bloco o seja retornada, em vez de esperar o bloco inteiro. Muitos processadores usam essa técnica para acesso a instruções, que é onde ela funciona melhor. Como os acessos a instruções são extremamente seqüenciais, se o sistema de memória puder entregar uma word a cada ciclo de clock, o processador poderá ser capaz de reiniciar sua operação quando a word requisitada for retornada, com o sistema de memória entregando novas words de instrução em tempo. Essa técnica normalmente é menos eficaz para caches de dados porque é provável que as words sejam requisitadas do bloco de uma maneira menos previsível; além disso, a probabilidade de que o processador precise de outra word de um bloco de cache diferente antes que a transferência seja concluída é alta. Se o processador não puder acessar a cache de dados porque uma transferência está em andamento, então, ele precisará sofrer stall.

Um esquema ainda mais sofisticado é organizar a memória de modo que a word requisitada seja transferida da memória para a cache primeiro. O restante do bloco, então, é transferido, começando com o endereço após a word requisitada e retornando para o início do bloco. Essa técnica, chamada *word requisitada primeiro*, ou *word crítica primeiro*, pode ser um pouco mais rápida do que o reinício precoce, mas ela é limitada pelas mesmas propriedades que limitam o reinício precoce.

Tratando falhas de cache

falta de cache Uma requisição de dados da cache que não pode ser atendida porque os dados não estão presentes na cache.

Antes de olharmos a cache de um sistema real, vamos ver como a unidade de controle lida com as **falhas de cache**. A unidade de controle precisa detectar uma falha e processá-la buscando os dados requisitados da memória (ou, como veremos, de uma cache de nível inferior). Se a cache reportar um acerto, o computador continua usando os dados como se nada tivesse acontecido. Consequentemente, podemos usar o mesmo controle básico que desenvolvemos no Capítulo 5 e aumentamos para acomodar pipelining no Capítulo 6. As memórias no caminho de dados dos Capítulos 5 e 6 são simplesmente substituídas por caches.

Modificar o controle de um processador para tratar um acerto é fácil; as falhas, no entanto, exigem um trabalho maior. O tratamento da falha de cache é feito com a unidade de controle do processador e com um controlador separado que inicia o acesso à memória e preenche novamente a cache. O processamento de uma falha de cache cria um stall semelhante aos stalls de pipeline discutidos no Capítulo 6, como oposto a uma interrupção, que exigiria salvar o estado de todos os registradores. Para uma falha de cache, podemos fazer um stall no processador inteiro, basicamente congelando o conteúdo dos registradores temporários e visíveis ao programador, enquanto esperamos a memória. Em contrapartida, os stalls de pipeline, discutidos no Capítulo 6, são mais complexos porque precisamos continuar a executar algumas instruções enquanto outras sofrem stall.

Vejamos um pouco mais de perto como as falhas de instrução são tratadas pelo caminho de dados multiciclo ou em pipeline; o mesmo método pode ser facilmente estendido para tratar falhas de dados. Se um acesso à instrução resultar em uma falha, o conteúdo do registrador de instrução será inválido. Para colocar a instrução correta na cache, precisamos ser capazes de instruir o nível inferior na hierarquia de memória a realizar uma leitura. Como o contador do programa é incrementado no primeiro ciclo de clock da execução nos processadores multiciclo e em pipeline, o endereço da instrução que gera uma falha de cache de instruções é igual ao valor do contador de programa menos 4. Uma vez tendo o endereço, precisamos instruir a memória principal a realizar uma leitura. Esperamos a memória responder (já que o acesso levará vários ciclos) e, então, escrevemos as words na cache.

Agora, podemos definir as etapas a serem realizadas em uma falha de cache de instruções:

1. Enviar o valor do PC original (PC atual - 4) para a memória.
2. Instruir a memória principal a realizar uma leitura e esperar que a memória complete seu acesso.
3. Escrever na entrada da cache, colocando os dados da memória na parte dos dados da entrada, escrevendo os bits mais significativos do endereço (vindo da ALU) no campo tag e ligando o bit de validade.
4. Reiniciar a execução da instrução na primeira etapa, o que buscará novamente a instrução, dessa vez encontrando-a na cache.

O controle da cache sobre um acesso de dados é basicamente idêntico: em uma falha, simplesmente suspendemos o processador até que a memória responda com os dados.

Tratando escritas

As escritas funcionam de maneira um pouco diferente. Suponha que, em uma instrução store, escrevemos os dados apenas na cache de dados (sem alterar a memória principal); então, após a escrita na cache, a memória teria um valor diferente do valor na cache. Nesse caso, dizemos que a cache e a memória estão *inconsistentes*. A maneira mais simples de manter consistentes a memória principal e a cache é sempre escrever os dados na memória e na cache. Esse esquema é chamado **write-through**.

O outro aspecto importante das escritas é o que ocorre em uma falha de dados. Primeiro, buscamos as words do bloco da memória. Após o bloco ser buscado e colocado na cache, podemos substituir (sobrescrever) a word que causou a falha no bloco de cache. Também escrevemos a word na memória principal usando o endereço completo.

Embora esse projeto trate das escritas de maneira muito simples, ele não oferece um desempenho muito bom. Com um esquema de write-through, toda escrita faz com que os dados sejam escritos na memória principal. Essas escritas levarão muito tempo, talvez mais de 100 ciclos de clock de processador, e tornariam o processador consideravelmente mais lento. Para os benchmarks de inteiro SPEC2000, por exemplo, 10% das instruções são stores. Se o CPI sem falhas de cache fosse 1,0, gastar 100 ciclos extras em cada escrita levaria a um CPI de $1,0 + 100 \times 10\% = 11$, reduzindo o desempenho em mais de 10%.

Uma solução para esse problema é usar um **buffer de escrita** (ou write buffer). Um buffer de escrita armazena os dados enquanto estão esperando para serem escritos na memória. Após escrever os dados na cache e no buffer de dados, o processador pode continuar a execução. Quando uma escrita na memória principal é concluída, a entrada no buffer de escrita é liberada. Se o buffer de escrita estiver cheio quando o processador atingir uma escrita, o processador precisará sofrer stall até que haja uma posição vazia no buffer de escrita. Naturalmente, se a velocidade em que a memória pode completar escritas for menor do que a velocidade em que o processador está gerando escritas, nenhuma quantidade de buffer pode ajudar, pois as escritas estão sendo geradas mais rápido do que o sistema de memória pode aceitá-las.

A velocidade em que as escritas são geradas também pode ser menor do que a velocidade em que a memória pode aceitá-las, e stalls ainda podem ocorrer. Isso pode acontecer quando as escritas ocorrem em bursts (ou rajadas). Para reduzir a ocorrência desses stalls, os processadores normalmente aumentam a profundidade do buffer de escrita para além de uma única entrada.

A alternativa para um esquema write-through é um esquema chamado **write-back**. Em um esquema write-back, quando ocorre uma escrita, o novo valor é escrito apenas no bloco da cache. O bloco modificado é escrito no nível inferior da hierarquia quando ele é substituído. Os esquemas write-back podem melhorar o desempenho, especialmente quando os processadores podem gerar escritas tão rápido ou mais rápido do que as escritas podem ser tratadas pela memória principal; entretanto, um esquema write-back é mais complexo de implementar do que um esquema write-through.

No restante desta seção, descreveremos as caches de processadores reais e examinaremos como elas tratam leituras e escritas. Na Seção 7.5, descreveremos o tratamento de escritas em mais detalhes.

write-through Um esquema em que as escritas sempre atualizam a cache e a memória, garantindo que os dados sejam sempre consistentes entre os dois.

buffer de escrita Uma fila que contém os dados enquanto estão esperando para serem escritos na memória.

write-back Um esquema que manipula escritas atualizando valores apenas no bloco da cache e, depois, escrevendo o bloco modificado no nível inferior da hierarquia quando o bloco é substituído.

Detalhamento: as escritas introduzem várias complicações nas caches que não estão presentes para leituras. Discutiremos aqui duas delas: a política nas falhas de escrita e a implementação eficiente das escritas em caches write-back.

Considere uma falha em uma cache write-through. A estratégia adotada na maioria dos projetos de cache write-through, chamada *buscar na falha, buscar na escrita ou, algumas vezes, alocar na falha*, aloca um bloco de cache para o endereço que falhou e busca o restante do bloco para a cache antes de escrever os dados e continuar a execução. Como alternativa, poderíamos alocar o bloco na cache mas não buscar os dados (o que se chama *não buscar na escrita*), ou mesmo não alocar o bloco (o que se chama *não alocar na escrita*). Outro nome para essas estratégias que não colocam os dados escritos na cache é *write-around*, já que os dados são escritos em torno da cache para chegar na memória. A motivação para esses esquemas é a observação de que, às vezes, os programas escrevem blocos de dados inteiros antes de lê-los. Nesses casos, a busca associada com a falha de escrita inicial pode ser desnecessária. Existem diversos problemas sutis envolvidos na implementação desses esquemas para blocos multiword, incluindo a complicação do tratamento dos acertos de escrita exigindo mecanismos semelhantes aos usados para as caches write-back.

Implementar stores de modo realmente eficaz em uma cache que usa uma estratégia write-back é mais complexo do que em uma cache write-through. Em uma cache write-back, precisamos escrever o bloco novamente na memória se os dados na cache estiverem modificados e tivermos uma falha de cache. Se simplesmente substituíssemos o bloco em uma instrução store antes de sabermos se o store teve acerto na cache (como poderíamos fazer para uma cache write-through), destruiríamos o conteúdo do bloco, que não é copiado na memória. Uma cache write-through pode escrever os dados na cache e ler a tag; se a tag não corresponder, então, ocorre uma falha. Como a cache é write-through, a substituição do bloco na cache não é catastrófica, já que a memória possui o valor correto.

Em uma cache write-back, como não podemos substituir o bloco, os stores exigem dois ciclos (um ciclo para verificar um acerto seguido de um ciclo para efetivamente realizar a escrita) ou exigem um buffer extra, chamado *buffer de store*, para conter esses dados – na prática, permitindo que o store leve apenas um ciclo por meio de um pipeline de memória. Quando um buffer de store é usado, o processador realiza a consulta de cache e coloca os dados no buffer de store durante o ciclo de acesso de cache normal. Considerando um acerto de cache, os novos dados são escritos do buffer de store para a cache no próximo ciclo de acesso de cache não usado.

Por comparação, em uma cache write-through, as escritas sempre podem ser feitas em um ciclo. No entanto, existem algumas complicações extras com os blocos multiword, uma vez que não podemos simplesmente substituir a tag quando escrevemos os dados. Em vez disso, lemos a tag e escrevemos a parte dos dados do bloco selecionado. Se a tag corresponder ao endereço do bloco escrito, o processador pode continuar normalmente, já que o bloco correto foi atualizado. Se a tag não corresponder, o processador gera uma falha de escrita para buscar o resto do bloco correspondente a esse endereço. Como é sempre seguro substituir os dados, os acertos de escrita ainda levam um ciclo.

Muitas caches write-back também incluem buffers de escrita usados para reduzir a penalidade de falha quando uma falha substitui um bloco modificado. Em casos como esse, o bloco modificado é movido para um buffer write-back associado com a cache enquanto o bloco requisitado é lido da memória. Depois, o buffer write-back é escrito novamente na memória. Considerando que outra falha não ocorra imediatamente, essa técnica divide a penalidade de falha quando um bloco modificado precisa ser substituído.

Uma cache de exemplo: o processador Intrinsity FastMATH

O Intrinsity FastMATH é um microprocessador embutido veloz que usa a arquitetura MIPS e uma implementação de cache simples. Próximo ao final do capítulo, examinaremos o projeto de cache mais complexo do Intel Pentium 4, mas começaremos com este exemplo simples, mas real, por questões didáticas. A Figura 7.9 mostra a organização da cache de dados do Intrinsity FastMATH.

Esse processador possui um pipeline de 12 estágios, semelhante ao discutido no Capítulo 6. Quando está operando na velocidade de pico, o processador pode requisitar uma word de instrução e uma word de dados em cada clock. Para satisfazer às demandas do pipeline sem stalls, são usadas caches de instruções e de dados separadas. Cada cache possui 16KB, ou 4K words, com blocos de 16 words.

As requisições de leitura para a cache são simples. Como existem caches de dados e de instruções separadas, sinais de controle separados serão necessários para ler e escrever em cada cache. (Lem-

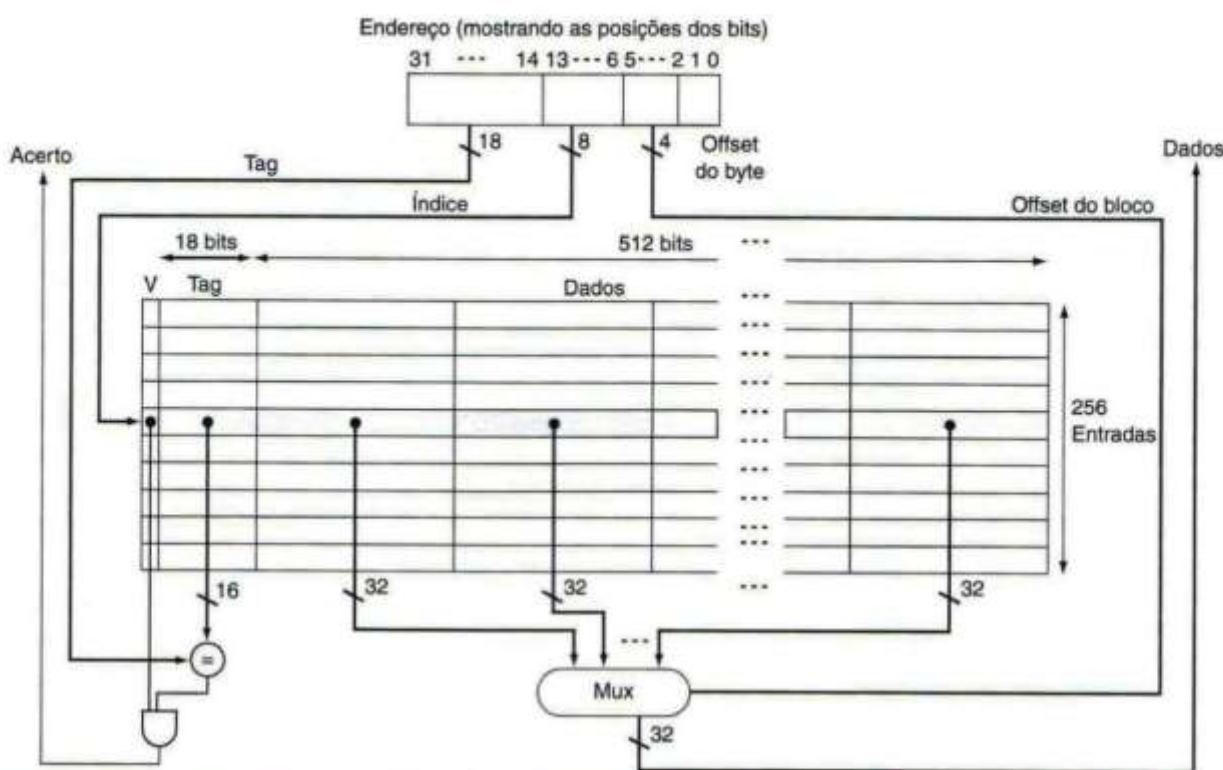


FIGURA 7.9 Cada cache de 16KB no Intrinsity FastMATH contém 256 blocos com 16 words por bloco. O campo tag possui 18 bits de largura e o campo índice possui 8 bits de largura, enquanto um campo de 4 bits (bits 5 a 2) é usado para indexar o bloco e selecionar a word do bloco usando um multiplexador de 16 para 1. Na prática, para eliminar o multiplexador, as caches usam uma RAM grande separada para os dados e uma RAM menor para as tags, com o offset de bloco fornecendo os bits de endereço extras para a RAM grande de dados. Nesse caso, a RAM grande possui 32 bits de largura e precisa ter 16 vezes o número de words como blocos na cache.

bre-se de que precisamos atualizar a cache de instruções quando ocorre uma falha.) Portanto, as etapas para uma requisição de leitura para qualquer uma das caches são as seguintes:

1. Enviar o endereço para a cache apropriada. O endereço vem do PC (para uma instrução) ou da ALU (para dados).
2. Se a cache sinalizar acerto, a word requisitada estará disponível nas linhas de dados. Como existem 16 words no bloco desejado, precisamos selecionar a word correta. Um campo índice de bloco é usado para controlar o multiplexador (mostrado na parte inferior da Figura), que seleciona a word requisitada das 16 words do bloco indexado.
3. Se a cache sinalizar falha, enviaremos o endereço para a memória principal. Quando a memória retorna com os dados, nós os escrevemos na cache e, então, os lemos para atender à requisição.

Para escritas, o Intrinsity FastMATH oferece write-through e write-back, deixando a cargo do sistema operacional decidir qual estratégia usar para cada aplicação. Ele possui um buffer de escrita de uma entrada.

Que taxas de falhas de cache são atingidas com uma estrutura de cache como a usada pelo Intrinsity FastMATH? A Figura 7.10 mostra as taxas de falhas para as caches de instruções e de dados para os benchmarks de inteiro do SPEC2000. A taxa de falhas combinada é a taxa de falhas efetiva por referência para cada programa após considerar a freqüência diferente dos acessos a instruções e a dados.

Taxa de falhas de instruções	Taxa de falhas de dados	Taxa de falhas combinada efetiva
0,4%	11,4%	3,2%

FIGURA 7.10 Taxas de falhas de instruções e dados aproximadas para o processador Intrinsity FastMATH para benchmarks SPEC2000. A taxa de falhas combinada é a taxa de falhas efetiva para a combinação da cache de instruções de 16KB e da cache de dados de 16KB. Ela é obtida ponderando as taxas de falhas individuais de instruções e de dados pela freqüência das referências a instruções e dados.

Embora a taxa de falhas seja uma característica importante dos projetos de cache, a medida decisiva será o efeito do sistema de memória sobre o tempo de execução do programa; em breve veremos como a taxa de falhas e o tempo de execução estão relacionados.

cache dividida Um esquema em que um nível da hierarquia de memória é composto de duas caches independentes que operam em paralelo uma com a outra, com uma tratando instruções e a outra tratando dados.

Detalhamento: uma cache combinada com um tamanho total igual à soma das duas **caches divididas** normalmente terá uma taxa de acertos melhor. Essa taxa mais alta ocorre porque a cache combinada não divide rigidamente o número de entradas que podem ser usadas por instruções daquelas que podem ser usadas por dados. Entretanto, muitos processadores usam uma instrução split e uma cache de dados para aumentar a largura de banda da cache.

Aqui estão taxas de falhas para caches do tamanho dos encontrados no processador Intrinsity FastMATH, e para uma cache combinada cujo tamanho é igual ao total das duas caches:

- Tamanho total da cache: 32KB
- Taxa de falhas efetiva da cache dividida: 3,24%
- Taxa de falhas da cache combinada: 3,18%

A taxa de falhas da cache dividida é apenas ligeiramente pior.

A vantagem de dobrar a largura de banda da cache, suportando acessos a instruções e a dados simultaneamente, logo suplanta a desvantagem de uma taxa de falhas um pouco maior. Essa constatação é outro lembrete de que não podemos usar a taxa de falhas como a única medida de desempenho de cache, como mostra a Seção 7.3.

Projetando o sistema de memória para suportar caches

As falhas de cache são satisfeitas pela memória principal, que é construída com DRAMs. Na Seção 7.1, vimos que as DRAMs são projetadas com a principal ênfase na densidade e não no tempo de acesso. Embora seja difícil reduzir a latência para buscar a primeira word da memória, podemos reduzir a penalidade de falha se aumentarmos a largura de banda da memória para a cache. Essa redução permite que tamanhos de bloco maiores sejam usados enquanto mantemos uma baixa penalidade de falhas, semelhante àquela para um bloco menor.

O processador normalmente é conectado à memória por meio de um barramento. A velocidade de clock do barramento geralmente é muito mais lenta do que a do processador (tanto quanto um fator de 10 vezes). A velocidade desse barramento afeta a penalidade de falha.

Para entender o impacto das diferentes organizações de memória, vamos definir um conjunto hipotético de tempos de acesso à memória. Considere

- 1 ciclo de clock de barramento de memória para enviar o endereço
- 15 ciclos de clock de barramento de memória para cada acesso a DRAM iniciado
- 1 ciclo de clock de barramento de memória para enviar uma word de dados

Se tivermos um bloco de cache de quatro words e um banco de DRAMs com a largura de uma word, a penalidade de falha seria $1 + 4 \times 15 + 4 \times 1 = 65$ ciclos de clock de barramento de memória. Portanto, o número de bytes transferidos por ciclo de clock de barramento para uma única falha seria

$$\frac{4 \times 4}{65} = 0,25$$

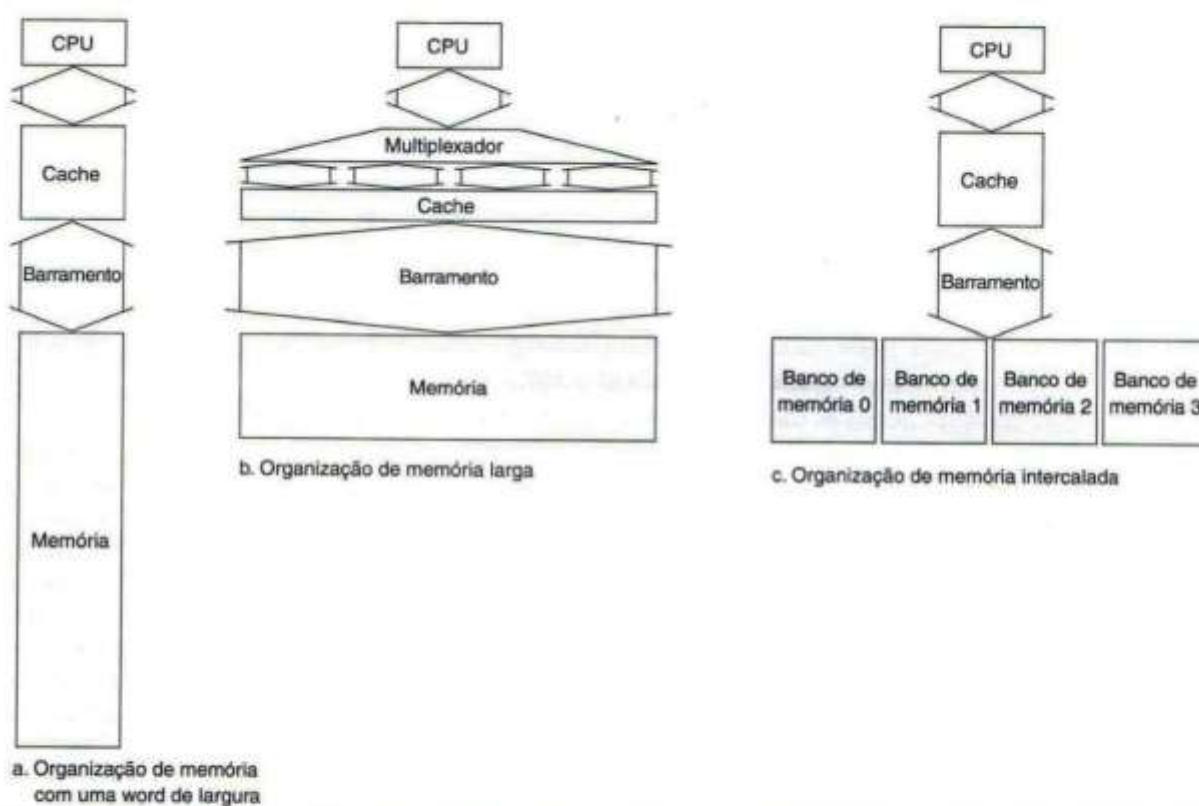


FIGURA 7.11 O principal método para obter largura de banda de memória mais alta é aumentar a largura física ou lógica do sistema de memória. Nesta figura, a largura de banda da memória é melhorada de duas maneiras. O projeto mais simples (a) usa uma memória na qual todos os componentes possuem uma word de largura; (b) mostra uma memória, um barramento e uma cache mais largos; enquanto (c) mostra um barramento e uma cache mais estreitos com uma memória intercalada. Em (b), a lógica entre a cache e o processador consiste em um multiplexador usado em leituras e lógica de controle para atualizar as words apropriadas da cache nas escritas.

A Figura 7.11 mostra três opções para projetar o sistema de memória. A primeira delas segue o que temos considerado: a memória possui uma word de largura e todos os acessos são feitos seqüencialmente. A segunda opção aumenta a largura de banda para a memória alargando a memória e os barramentos entre o processador e a memória; isso permite acessos paralelos a todas as words do bloco. A terceira opção aumenta a largura de banda alargando a memória mas não o barramento de interconexão. Portanto, ainda pagamos um custo para transmitir cada word, mas podemos evitar pagar o custo da latência de acesso mais de uma vez. Vejamos em quanto essas outras duas opções melhoraram a penalidade de falha de 65 ciclos que veríamos para a primeira opção (Figura 7.11a).

Aumentar a largura da memória e do barramento aumentará a largura de banda da memória proporcionalmente, diminuindo as partes do tempo de acesso e do tempo de transferência da penalidade de falha. Com uma largura de memória principal de duas words, a penalidade de falha cai de 65 ciclos de clock de barramento de memória para $1 + 2 \times 15 + 2 \times 1 = 33$ ciclos de clock de barramento de memória. A largura de banda para uma única falha é, então, 0,48 (quase duas vezes maior) byte por ciclo de clock de barramento para uma memória que tem duas words de largura, e 0,94 byte por ciclo de clock de barramento quando a memória tem quatro words de largura (quase quatro vezes maior). Os maiores custos dessa melhoria são o barramento mais largo e o possível aumento no tempo de acesso da cache devido ao multiplexador e à lógica de controle entre o processador e a cache.

Em vez de tornar todo o caminho entre a memória e a cache mais largo, os chips de memória podem ser organizados em bancos para ler ou escrever múltiplas words em um único tempo de acesso em vez de ler ou escrever uma única word em cada vez. Cada banco poderia ter uma word de largura para que a largura do barramento e da cache não precisassem mudar, mas enviar um endereço para

vários bancos permite que todos eles leiam simultaneamente. Esse esquema, chamado de *intercalação* (interleaving), conserva a vantagem de incorrer a latência de memória completa apenas uma vez. Por exemplo, com quatro bancos, o tempo para obter um bloco de quatro words consistiria em 1 ciclo para transmitir o endereço e a requisição de leitura para os bancos, 15 ciclos para que todos os quatro bancos accessem a memória e 4 ciclos para enviar as quatro words de volta para a cache. Isso produz uma penalidade de falha de $1 + 1 \times 15 + 4 \times 1 = 20$ ciclos de clock de barramento de memória. Essa é uma largura de banda efetiva por falha de 0,80 byte por clock, ou cerca de três vezes a largura de banda para a memória e barramento de uma word de largura. Os bancos também são valiosos nas escritas. Cada banco pode escrever independentemente, quadruplicando a largura de banda de escrita e gerando menos stalls em uma cache write-through. Como veremos, uma estratégia alternativa para escritas torna a intercalação ainda mais atraente.

Detalhamento: os chips de memória são organizados para produzir vários bits de saída, normalmente de 4 a 32, sendo que 8 ou 16 eram os mais comuns em 2004, quando este livro foi escrito. Descrevemos a organização da RAM como $d \times w$, onde d é o número dos locais endereçáveis (a profundidade), e w é a saída (ou a largura de cada local). Um caminho para melhorar a velocidade em que transferimos dados da memória para as caches é tirar proveito da estrutura das DRAMs. As DRAMs são organizadas logicamente como arrays retangulares, e o tempo de acesso é dividido em acesso de linha e acesso de coluna. As DRAMs colocam uma linha de bits em um buffer dentro da DRAM para o acesso de coluna. Elas também fornecem sinais de sincronização opcionais que permitem acessos repetidos ao buffer sem um tempo de acesso de linha. Essa capacidade, originalmente chamada de *modo de página*, passou por uma série de avanços. No modo de página, o buffer age como uma SDRAM; mudando o endereço de coluna, bits aleatórios podem ser acessados no buffer até o próximo acesso de linha. Essa capacidade muda significativamente o tempo de acesso, já que o tempo de acesso para bits na mesma linha é muito menor. A Figura 7.12 mostra como a densidade, o custo e o tempo de acesso das DRAMs mudaram através dos anos.

O desenvolvimento mais recente são as SDRAMs DDR (DRAMs síncronas de velocidade de dados dupla). As SDRAMs fornecem um acesso em rajada aos dados de locais sequenciais na DRAM. Uma SDRAM recebe um endereço inicial e o tamanho da rajada. Os dados da rajada são transferidos sob o controle de um sinal de clock, que, em 2004, podia operar em até 300MHz. As duas principais vantagens das SDRAMs são o uso de um clock que elimina a necessidade de sincronização e a eliminação da necessidade de fornecer endereços sucessivos para uma rajada. A parte DDR do nome significa que os dados são transferidos tanto na transição de subida quanto na transição de descida do

Ano de surgimento	Tamanho do chip	USS por MB	Tempo de acesso total a uma nova linha/coluna	Tempo de acesso de coluna a uma linha existente
1980	64 Kbit	US\$1.500	250ns	150ns
1983	256 Kbit	US\$500	185ns	100ns
1985	1 Mbit	US\$200	135ns	40ns
1989	4 Mbit	US\$50	110ns	40ns
1992	16 Mbit	US\$15	90ns	30ns
1996	64 Mbit	US\$10	60ns	12ns
1998	128 Mbit	US\$4	60ns	10ns
2000	256 Mbit	US\$1	55ns	7ns
2002	512 Mbit	US\$0,25	50ns	5ns
2004	1.024 Mbit	US\$0,10	45ns	3ns

FIGURA 7.12 Tamanho da DRAM aumentado por múltiplos de quatro aproximadamente uma vez a cada três anos até 1996 e, daí em diante, dobrando aproximadamente a cada dois anos. As melhorias no tempo de acesso têm sido mais lentas, porém contínuas, e o custo quase acompanha as melhorias na densidade, embora seja freqüentemente afetado por outros fatores, como a disponibilidade e a demanda. O custo por megabyte não está ajustado pela inflação.

clock, gerando, assim, o dobro da largura de banda que você poderia esperar com base na velocidade de clock e na largura de dados. Para oferecer uma largura de banda tão grande, a DRAM interna é organizada em bancos de memória intercalados.

A vantagem dessas otimizações é que elas usam os circuitos já presentes amplamente nas DRAMs, adicionando pouco custo ao sistema enquanto atinge uma significativa melhoria na largura de banda. A arquitetura interna das DRAMs e como essas otimizações são implementadas são descritas na Seção B.8 do Apêndice B.

Resumo

Começamos a seção anterior examinando a mais simples das caches: uma cache diretamente mapeada com um bloco de uma word. Nesse tipo de cache, tanto os acertos quanto as falhas são simples, já que uma word pode estar localizada exatamente em um lugar e existe uma tag separada para cada word. Para manter a cache e a memória consistentes, um esquema de write-through pode ser usado, de modo que toda escrita na cache também faz com que a memória seja atualizada. A alternativa ao write-through é um esquema write-back que copia um bloco de volta para a memória quando ele é substituído; discutiremos esse esquema mais detalhadamente em seções futuras.

Para tirar vantagem da localidade espacial, uma cache precisa ter um tamanho de bloco maior do que uma word. O uso de um bloco maior diminui a taxa de falhas e melhora a eficiência da cache reduzindo a quantidade de armazenamento de tag em relação à quantidade de armazenamento de dados na cache. Embora um tamanho de bloco maior diminua a taxa de falhas, ele também pode aumentar a penalidade de falha. Se a penalidade de falha aumentasse linearmente com o tamanho de bloco, blocos maiores poderiam facilmente levar a um desempenho menor. Para evitar isso, a largura de banda da memória principal é aumentada para transferir blocos de cache de maneira mais eficiente. Os dois métodos comuns para fazer isso são tornar a memória mais larga e a intercalação. Nos dois casos, reduzimos o tempo para buscar o bloco minimizando o número de vezes que precisamos iniciar um novo acesso à memória para buscar um bloco; e, com um barramento mais largo, também podemos diminuir o tempo necessário para enviar o bloco da memória para a cache.

A velocidade do sistema de memória afeta a decisão do projetista sobre o tamanho do bloco de cache. Quais dos seguintes princípios de projeto de cache normalmente são válidos?

**Verifique
você mesmo**

1. Quanto mais curta for a latência da memória, menor será o bloco de cache.
2. Quanto mais curta for a latência da memória, maior será o bloco de cache.
3. Quanto maior for a largura de banda da memória, menor será o bloco de cache.
4. Quanto maior for a largura de banda da memória, maior será o bloco de cache.

7.3

Medindo e melhorando o desempenho da cache

Nesta seção, começamos examinando como medir e analisar o desempenho da cache; depois, exploramos duas técnicas diferentes para melhorar o desempenho da cache. Uma delas focaliza o decréscimo da taxa de falhas reduzindo a probabilidade de dois blocos de memória diferentes disputarem o mesmo local da cache. A segunda técnica reduz a penalidade de falha acrescentando um nível adicional na hierarquia. Essa técnica, chamada *caching multinível*, apareceu inicialmente nos computadores de topo de linha sendo vendidos por mais de US\$100.000 em 1990; desde então, ela se tornou comum nos computadores desktop vendidos por menos de US\$1.000!

O tempo de CPU pode ser dividido nos ciclos de clock que a CPU gasta executando o programa e os ciclos de clock que gasta esperando o sistema de memória. Normalmente, consideramos que os custos dos acesso à cache que são acertos são parte dos ciclos de execução normais da CPU. Portanto,

$$\text{Tempo de CPU} = (\text{ciclos de clock de execução da CPU} + \text{Ciclos de clock de stall de memória}) \times \text{Tempo de ciclo de clock}$$

Os ciclos de clock de stall de memória vêm principalmente das falhas de cache e é isso que iremos considerar aqui. Também limitamos a discussão a um modelo simplificado do sistema de memória. Nos processadores reais, os stalls gerados por leituras e escritas podem ser muito complexos, e a previsão correta do desempenho normalmente exige simulações extremamente detalhadas do processador e do sistema de memória.

Os ciclos de clock de stall de memória podem ser definidos como a soma dos ciclos de stall vindo das leituras mais os provenientes das escritas:

$$\text{Ciclos de clock de stall de memória} = \text{Ciclos de stall de leitura} + \text{Ciclos de stall de escrita}$$

Os ciclos de stall de leitura podem ser definidos em função do número de acessos de leitura por programa, a penalidade de falha nos ciclos de clock para uma leitura e a taxa de falhas de leitura:

$$\text{Ciclos de stall de leitura} = \frac{\text{Leituras}}{\text{Programa}} \times \text{Taxa de falhas de leitura} \times \text{Penalidade de falha de leitura}$$

As escritas são mais complicadas. Para um esquema write-through, temos duas origens de stalls: as falhas de escrita, que normalmente exigem que busquemos o bloco antes de continuar a escrita (veja a seção “Detalhamento” na página 366 para obter mais informações sobre como lidar com escritas), e os stalls do buffer de escrita, que ocorrem quando o buffer de escrita está cheio ao ocorrer uma escrita. Assim, os ciclos de stall para escritas são iguais à soma desses dois fatores:

$$\text{Ciclos de stall de escrita} = \left(\frac{\text{Escritas}}{\text{Programa}} \times \text{Taxa de falhas de escrita} \times \text{Penalidade de falha de escrita} \right) + \text{Stalls do buffer de escrita}$$

Como os stalls do buffer de escrita dependem da sincronização das escritas, e não apenas da freqüência, não é possível fornecer uma equação simples para calcular esses stalls. Felizmente, nos sistemas com um buffer de escrita razoável (por exemplo, quatro ou mais words) e uma memória capaz de aceitar escritas em uma velocidade que excede significativamente a freqüência de escrita média em programas (por exemplo, por um fator de duas vezes), os stalls do buffer de escrita serão pequenos e podemos ignorá-los. Se um sistema não atendesse a esse critério, ele não seria bem projetado; ao contrário, o projetista deveria ter usado um buffer de escrita mais profundo ou uma organização write-back.

Os esquemas write-back também possuem stalls potenciais extras surgindo da necessidade de escrever um bloco de cache novamente na memória quando o bloco é substituído. Discutiremos mais isso na Seção 7.5.

Na maioria das organizações de cache write-back, as penalidades de falha de leitura e escrita são iguais (o tempo para buscar o bloco da memória). Se considerarmos que os stalls do buffer de escrita são insignificantes, podemos combinar as leituras e escritas usando uma única taxa de falhas e a penalidade de falha:

$$\text{Ciclos de clock de stall de memória} = \frac{\text{Acessos à memória}}{\text{Programa}} \times \text{Taxa de falhas} \times \text{Penalidade de falha}$$

Também podemos fatorar isso como

$$\text{Ciclos de clock de stall de memória} = \frac{\text{Instruções}}{\text{Programa}} \times \frac{\text{Falhas}}{\text{Instrução}} \times \text{Penalidade de falha}$$

Vamos considerar um exemplo simples para ajudar a entender o impacto no desempenho da cache sobre o desempenho do processador.

CALCULANDO O DESEMPENHOO DA CACHE

Suponha que uma taxa de falhas de cache de instruções para um programa seja de 2% e que uma taxa de falhas de cache de dados seja de 4%. Se um processador possui um CPI de 2 sem qualquer stall de memória e a penalidade de falha é de 100 ciclos para todas as falhas, determine o quanto mais rápido um processador executaria com uma cache perfeita que nunca falhasse. Use as freqüências de instruções do SPECint2000 do Capítulo 3, Figura 3.26.

O número de ciclos de falha da memória para instruções em termos da contagem de instruções (I) é

$$\text{Ciclos de falha de instrução} = I \times 2\% \times 100 = 2,00 \times I$$

A freqüência de todos os loads e stores no SPECint2000 é de 36%. Logo, podemos encontrar o número de ciclos de falha da memória para referências de dados:

$$\text{Ciclos de falha de dados} = I \times 36\% \times 4\% \times 100 = 1,44 \times I$$

O número total de ciclos de stall da memória é $2,00I + 1,44I = 3,44I$. Isso é mais do que 3 ciclos de stall da memória por instrução. Portanto, o CPI com stalls da memória é $2 + 3,44 = 5,44$. Como não há mudança alguma na contagem de instruções ou na velocidade de clock, a taxa dos tempos de execução da CPU é

$$\frac{\text{Tempo de CPU com stalls}}{\text{Tempo de CPU com cache perfeita}} = \frac{I \times \text{CPI}_{\text{stall}} \times \text{Ciclo de clock}}{I \times \text{CPI}_{\text{perfeito}} \times \text{Ciclo de clock}}$$

$$\frac{\text{CPI}_{\text{stall}}}{\text{CPI}_{\text{perfeito}}} = \frac{5,44}{2}$$

O desempenho com a cache perfeita é melhor por um fator de $\frac{5,44}{2} = 2,72$

O que acontece se o processador for tornado mais rápido mas o sistema de memória não? A quantidade de tempo gasto nos stalls da memória tomará uma fração cada vez maior do tempo de execução; a Lei de Amdahl, que examinamos no Capítulo 4, nos lembra desse fato. Alguns exemplos simples mostram como esse problema pode ser sério. Suponha que aceleremos o computador do exemplo anterior reduzindo seu CPI de 2 para 1 sem mudar a velocidade de clock, o que pode ser feito com um pipeline melhorado. O sistema com falhas de cache, então, teria um CPI de $1 + 3,44 = 4,44$, e o sistema com a cache perfeita seria

$$\frac{4,44}{1} = 4,44 \text{ vezes mais rápido}$$

A quantidade de tempo de execução gasto em stalls da memória teria subido de

EXEMPLO

RESPOSTA

$$\frac{3,44}{5,44} = 63\%$$

para

$$\frac{3,44}{4,44} = 77\%$$

Da mesma forma, aumentar a velocidade de clock sem mudar o sistema de memória também aumenta a perda de desempenho devido às falhas de cache, como mostra o próximo exemplo.

DESEMPENHO DA CACHE COM VELOCIDADE DE CLOCK AUMENTADA

EXEMPLO

Suponha que aumentemos o desempenho do computador do exemplo anterior dobrando sua velocidade de clock. Como a velocidade da memória principal é improvável de ser alterada, considere que o tempo absoluto para manipular uma falha de cache não mude. O quanto mais rápido será o computador com o clock mais rápido, considerando a mesma taxa de falhas do exemplo anterior?

RESPOSTA

Medida com ciclos de clock mais rápidos, a nova penalidade de falha será o dobro dos ciclos de clock, ou 200 ciclos de clock. Portanto:

$$\text{Total de ciclos de falha por instrução} = (2\% \times 200) + 36\% \times (4\% \times 200) = 6,88$$

Logo, o computador mais rápido com falhas de cache terá um CPI de $2 + 6,88 = 8,88$, comparado com um CPI com falhas de cache de 5,44 para o computador mais lento.

Usando a fórmula para o tempo de CPU do exemplo anterior, podemos calcular o desempenho relativo como

$$\begin{aligned} \frac{\text{Desempenho com clock rápido}}{\text{Desempenho com clock lento}} &= \frac{\text{Tempo de execução com clock lento}}{\text{Tempo de execução com clock rápido}} \\ &= \frac{\text{CI} \times \text{CPI}_{\text{clock lento}} \times \text{Ciclo de clock}}{\text{CI} \times \text{CPI}_{\text{clock rápido}} \times \frac{\text{Ciclo de clock}}{2}} \\ &= \frac{5,44}{8,88 \times \frac{1}{2}} = 1,23 \end{aligned}$$

Portanto, o computador com clock mais rápido é cerca de 1,2 vez mais rápido, e não 2 vezes mais rápido, que ele teria sido se ignorássemos as falhas de cache.

Como demonstram esses exemplos, as penalidades de cache relativas aumentam à medida que um processador se torna mais rápido. Além disso, se um processador melhorar a velocidade de clock e o CPI, ele experimentará uma consequência dupla:

1. Quanto menor o CPI, maior será o impacto dos ciclos de stall.
2. É improvável que o sistema de memória principal seja melhorado tão rápido quanto o tempo de ciclo do processador, principalmente porque o desempenho da DRAM básica não está se tornando muito mais rápido. Ao calcular o CPI, a penalidade de falha da cache é medida em ciclos de clock do processador para uma falha. Portanto, se as memórias principais de dois processadores tiverem os mesmos tempos de acesso absolutos, uma velocidade de clock de processador mais alta produzirá uma penalidade de falha maior.

Assim, a importância do desempenho da cache para os processadores com baixo CPI e altas velocidades de clock é maior; e, consequentemente, o perigo de ignorar o comportamento da cache na avaliação do desempenho desses processadores também é maior. Como veremos na Seção 7.6, o uso de processadores em pipeline rápidos nos PCs desktop e nas estações de trabalho tem levado ao uso de sofisticados sistemas de cache mesmo em computadores vendidos por menos de US\$1.000.

Os exemplos e equações anteriores consideram que o tempo de acerto não é um fator na determinação do desempenho da cache. Claramente, se o tempo de acerto aumentar, o tempo total para acessar uma word do sistema de memória aumentará, possivelmente causando um aumento no tempo de ciclo do processador. Embora vejamos em breve outros exemplos do que pode aumentar o tempo de acerto, um exemplo é aumentar o tamanho da cache. Uma cache maior pode ter um tempo de acesso maior, exatamente como se sua mesa na biblioteca fosse muito grande (digamos, 3 metros quadrados): você levaria mais tempo para localizar um livro. Com pipelines mais profundos do que cinco estágios, um aumento no tempo de acerto provavelmente acrescenta outro estágio ao pipeline, já que podem ser necessários vários ciclos para um acerto de cache. Embora seja mais complexo calcular o impacto de desempenho de um pipeline mais profundo, em algum ponto, o aumento no tempo de acerto para uma cache maior pode dominar a melhoria na taxa de acertos, levando a uma redução no desempenho do processador.

A próxima subseção discute organizações de cache alternativas que diminuem a taxa de falhas mas pode, algumas vezes, aumentar o tempo de acerto; outros exemplos aparecem em Falácias e armadilhas (Seção 7.7).

Reduzindo as falhas de cache com um posicionamento de blocos mais flexível

Até agora, quando colocamos um bloco na cache, usamos um esquema de posicionamento simples: um bloco só pode entrar exatamente em um local na cache. Como já dissemos, esse esquema é chamado de *mapeamento direto* porque qualquer endereço de bloco na memória é diretamente mapeado para um único local no nível superior da hierarquia. Existe, na verdade, toda uma faixa de esquemas para posicionamento de blocos. Em um extremo está o mapeamento direto, onde um bloco só pode ser posicionado exatamente em um local.

No outro extremo está um esquema em que um bloco pode ser posicionado em *qualquer* local na cache. Esse esquema é chamado de **totalmente associativo** porque um bloco na memória pode ser associado com qualquer entrada da cache. Para encontrar um determinado bloco em uma cache totalmente associativa, todas as entradas da cache precisam ser pesquisadas, pois um bloco pode estar posicionado em qualquer uma delas. Para tornar a pesquisa exequível, ela é feita em paralelo com um comparador associado a cada entrada da cache. Esses comparadores aumentam muito o custo do hardware, na prática, tornando o posicionamento totalmente associativo viável apenas para caches com pequenos números de blocos.

A faixa intermediária de projetos entre a cache diretamente mapeada e a cache totalmente associativa é chamada de **associativa por conjunto**. Em uma cache associativa por conjunto, existe um número fixo de locais (pelo menos dois) onde cada bloco pode ser colocado; uma cache associativa por conjunto com n locais para um bloco é chamado de cache associativa por conjunto de n vias. Uma cache associativa por conjunto de n vias consiste em diversos conjuntos, cada um consistindo em n blocos. Cada bloco na memória é mapeado para um *conjunto* único na cache, determinado pelo campo índice, e um bloco pode ser colocado em *qualquer* elemento desse conjunto. Portanto, um posicionamento associativo por conjunto combina o posicionamento diretamente mapeado e o posicionamento totalmente associativo: um bloco é diretamente mapeado para um conjunto e, então, uma correspondência é pesquisada em todos os blocos no conjunto.

cache totalmente associativa Uma estrutura de cache em que um bloco pode ser posicionado em qualquer local da cache.

cache associativa por conjunto Uma cache que possui um número fixo de locais (no mínimo dois) onde cada bloco pode ser colocado.

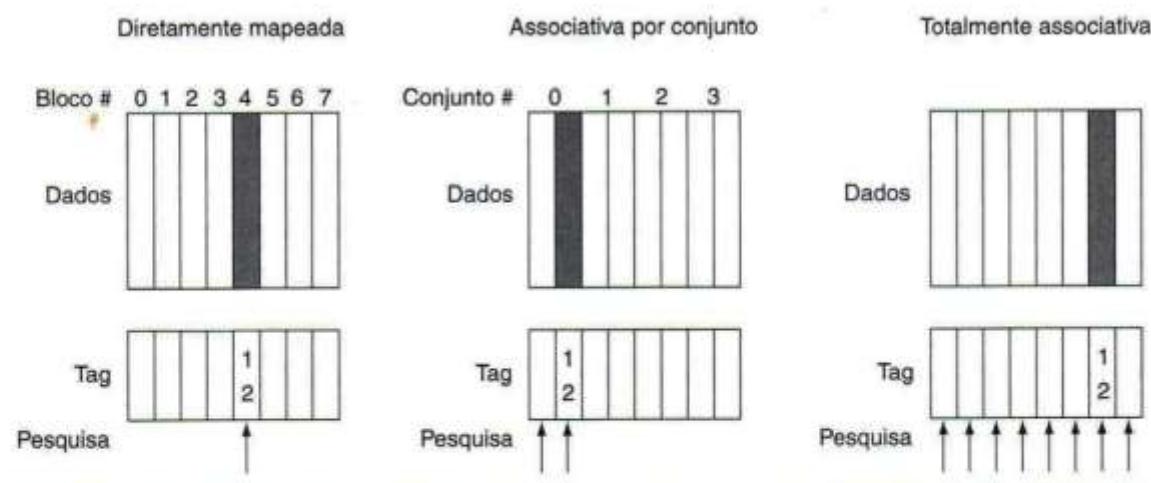


FIGURA 7.13 O local de um bloco de memória cujo endereço é 12 em uma cache com 8 blocos varia para posicionamento diretamente mapeado, associativo por conjunto e totalmente associativo. No posicionamento diretamente mapeado, há apenas um bloco de cache onde o bloco de memória 12 pode ser encontrado e esse bloco é dado por $(12 \bmod 8) = 4$. Em uma cache associativa por conjunto de duas vias, haveria quatro conjuntos e o bloco de memória 12 precisa estar no conjunto $(12 \bmod 4) = 0$; o bloco de memória pode estar em qualquer elemento do conjunto. Em um posicionamento totalmente associativo, o bloco de memória para o endereço de bloco 12 pode aparecer em qualquer um dos oito blocos de cache.

Lembre-se de que, em uma cache diretamente mapeada, a posição de um bloco de memória é determinada por

$$(\text{Número do bloco}) \bmod (\text{Número de blocos de cache})$$

Em uma cache associativa por conjunto, o conjunto contendo um bloco de memória é determinado por

$$(\text{Número do bloco}) \bmod (\text{Número de conjuntos na cache})$$

Como o bloco pode ser colocado em qualquer elemento do conjunto, *todas as tags de todos os elementos do conjunto precisam ser pesquisadas*. Em uma cache totalmente associativa, o bloco pode entrar em qualquer lugar e *todas as tags de todos os blocos na cache precisam ser pesquisadas*. Por exemplo, a Figura 7.13 mostra onde o bloco 12 pode ser colocado em uma cache com oito blocos no total, conforme a política de posicionamento de blocos para caches diretamente mapeadas, associativas por conjunto de duas vias e totalmente associativas.

Podemos pensar em cada estratégia de posicionamento de bloco como uma variação da associatividade por conjunto. A Figura 7.14 mostra as possíveis estruturas de associatividade para uma cache de oito blocos. Uma cache diretamente mapeada é simplesmente uma cache associativa por conjunto de uma via: cada entrada de cache contém um bloco e cada conjunto possui um elemento. Uma cache totalmente associativa com m entradas é simplesmente uma cache associativa por conjunto de m vias; ele tem um conjunto com m blocos e uma entrada pode residir em qualquer bloco dentro desse conjunto.

A vantagem de aumentar o grau da associatividade é que ela normalmente diminui a taxa de falhas, como mostra o próximo exemplo. A principal desvantagem, que veremos em mais detalhes em breve, é um aumento no tempo de acerto.



FIGURA 7.14 Uma cache de oito blocos configurada como diretamente mapeada, associativa por conjunto de duas vias, associativa por conjunto de quatro vias e totalmente associativa. O tamanho total da cache em blocos é igual ao número de conjuntos multiplicado pela associatividade. Portanto, para uma cache de tamanho fixo, aumentar a associatividade diminui o número de conjuntos enquanto aumenta o número de elementos por conjunto. Com oito blocos, uma cache associativa por conjunto de oito vias é igual a uma cache totalmente associativa.

FALHAS E ASSOCIATIVIDADE NAS CACHES

Considere três caches pequenas, cada uma consistindo em quatro blocos de uma word cada. Uma cache é totalmente associativa, uma segunda cache é associativa por conjunto de duas vias e a terceira cache é diretamente mapeada. Encontre o número de falhas para cada organização de cache, dada a seguinte seqüência de endereços de bloco: 0, 8, 0, 6, 8.

EXEMPLO

O caso diretamente mapeado é mais fácil. Primeiro, vamos determinar para qual bloco de cache cada endereço de bloco é mapeado:

RESPOSTA

Endereço do bloco	Bloco de cache
0	(0 módulo 4) = 0
6	(6 módulo 4) = 2
8	(8 módulo 4) = 0

Agora podemos preencher o conteúdo da cache após cada referência, usando uma entrada em branco para indicar que o bloco é inválido, texto colorido para mostrar uma nova entrada incluída na cache para a referência associada e um texto normal para mostrar uma entrada existente na cache:

Endereço do bloco de memória associado	Acerto ou falha	Conteúdo dos blocos de cache após referência			
		0	1	2	3
0	falha	Memória[0]			*
8	falha	Memória[8]			
0	falha	Memória[0]			
6	falha	Memória[0]		Memória[6]	
8	falha	Memória[8]		Memória[6]	

A cache diretamente mapeada gera cinco falhas para os cinco acessos.

A cache associativa por conjunto possui dois conjuntos (com índices 0 e 1) com dois elementos por conjunto. Primeiro, vamos determinar para qual conjunto cada endereço de bloco é mapeado:

Endereço do bloco	Bloco de cache
0	(0 módulo 2) = 0
6	(6 módulo 2) = 0
8	(8 módulo 2) = 0

Já que temos uma escolha de qual entrada em um conjunto substituir em uma falha, precisamos de uma regra de substituição. As caches associativas por conjunto normalmente substituem o bloco menos recentemente usado dentro de um conjunto; ou seja, o bloco usado há mais tempo é substituído. (Discutiremos as regras de substituição mais detalhadamente em breve.) Usando essa regra de substituição, o conteúdo da cache associativa por conjunto após cada referência se parece com o seguinte:

Endereço do bloco de memória associado	Acerto ou falha	Conteúdo dos blocos de cache após referência			
		Conjunto 0	Conjunto 0	Conjunto 1	Conjunto 1
0	falha	Memória[0]			
8	falha	Memória[0]	Memória[8]		
0	acerto	Memória[0]	Memória[8]		
6	falha	Memória[0]	Memória[6]		
8	falha	Memória[8]	Memória[6]		

Observe que quando o bloco 6 é referenciado, ele substitui o bloco 8, já que o bloco 8 foi referenciado menos recentemente do que o bloco 0. A cache associativa por conjunto de duas vias possui quatro falhas, uma a menos do que a cache diretamente mapeada.

A cache totalmente associativa possui quatro blocos de cache (em um único conjunto); qualquer bloco de memória pode ser armazenado em qualquer bloco de cache. A cache totalmente associativa possui o melhor desempenho, com apenas três falhas:

Endereço do bloco de memória associado	Acerto ou falha	Conteúdo dos blocos de cache após referência			
		Bloco 0	Bloco 1	Bloco 2	Bloco 3
0	falha	Memória[0]			
8	falha	Memória[0]	Memória[8]		
0	acerto	Memória[0]	Memória[8]		
6	falha	Memória[0]	Memória[8]		
8	falha	Memória[0]	Memória[8]	Memória[6]	

Para essa série de referências, três falhas é o melhor que podemos fazer porque três endereços de bloco únicos são acessados. Repare que se tivéssemos oito blocos na cache, não haveria qualquer substituição na cache associativa por conjunto de duas vias (confira isso você mesmo), e ele teria o mesmo número de falhas da cache totalmente associativa. Da mesma forma, se tivéssemos 16 blocos, todas as três caches teriam o mesmo número de falhas. Essa mudança na taxa de falhas mostra que o tamanho da cache e a associatividade não são independentes para a determinação do desempenho da cache.

Quanta redução na taxa de falhas é obtida pela associatividade? A Figura 7.15 mostra a melhoria para os benchmarks SPEC2000 para uma cache de dados de 64KB com um bloco de 16 words e mostra a associatividade mudando do mapeamento direto para oito vias. Passar da associatividade de uma via para duas vias diminui a taxa de falhas em aproximadamente 15%, mas há pouca melhora adicional em passar para uma associatividade mais alta.

Associatividade	Taxa de falhas de dados
1	10,3%
2	8,6%
4	8,3%
8	8,1%

FIGURA 7.15 As taxas de falhas da cache de dados para uma organização como o processador Intrinsity FastMATH para benchmarks SPEC2000 com associatividade variando de uma via a oito vias. Esses resultados para 10 programas SPEC2000 são de Hennessy e Patterson [2003].

Localizando um bloco na cache

Agora, vamos considerar a tarefa de encontrar um bloco em uma cache que é associativa por conjunto. Assim como em uma cache diretamente mapeada, cada bloco em uma cache associativa por conjunto inclui uma tag de endereço que fornece o endereço do bloco. A tag de cada bloco de cache dentro do conjunto apropriado é verificada para ver se corresponde ao endereço de bloco vindo do processador. A Figura 7.16 mostra como o endereço é decomposto. O valor de índice é usado para selecionar o conjunto contendo o endereço de interesse, e as tags de todos os blocos no conjunto precisam ser pesquisadas. Como a velocidade é a essência da pesquisa, todas as tags no conjunto selecionado são pesquisadas em paralelo. Assim como em uma cache totalmente associativa, uma pesquisa sequencial tornaria o tempo de acerto de uma cache associativa por conjunto muito lento.

Se o tamanho de cache total for mantido igual, aumentar a associatividade aumenta o número de blocos por conjunto, que é o número de comparações simultâneas necessárias para realizar a pesquisa em paralelo: cada aumento por um fator de dois na associatividade dobra o número de blocos por conjunto e divide por dois o número de conjuntos. Assim, cada aumento pelo dobro na associatividade diminui o tamanho do índice em 1 bit e aumenta o tamanho da tag em 1 bit. Em uma cache totalmente associativa, existe apenas um conjunto e todos os blocos precisam ser verificados em paralelo. Portanto, não há qualquer índice, e o endereço inteiro, excluindo o offset do bloco, é comparado com a tag de cada bloco. Em outras palavras, a cache inteira é pesquisada sem qualquer indexação.

Em uma cache diretamente mapeada, como na Figura 7.7, apenas um único comparador é necessário, pois a entrada pode estar apenas em um bloco, e acessamos a cache indexando. A Figura 7.17 mostra que em uma cache associativa por conjunto de quatro vias, quatro comparadores são necessários, juntamente com um multiplexador de 4 para 1 para escolher entre os quatro números possíveis do conjunto selecionado. O acesso de cache consiste em indexar o conjunto apropriado e, depois, pesquisar as tags do conjunto. Os custos de uma cache associativa são os comparadores extras e qualquer atraso pela necessidade de comparar e selecionar entre os elementos do conjunto.

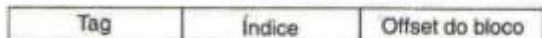


FIGURA 7.16 As três partes de um endereço em uma cache associativa por conjunto ou diretamente mapeada. O índice é usado para selecionar o conjunto e, depois, a tag é usada para escolher o bloco por comparação com os blocos no conjunto selecionado. O offset do bloco é o endereço dos dados desejados dentro do bloco.

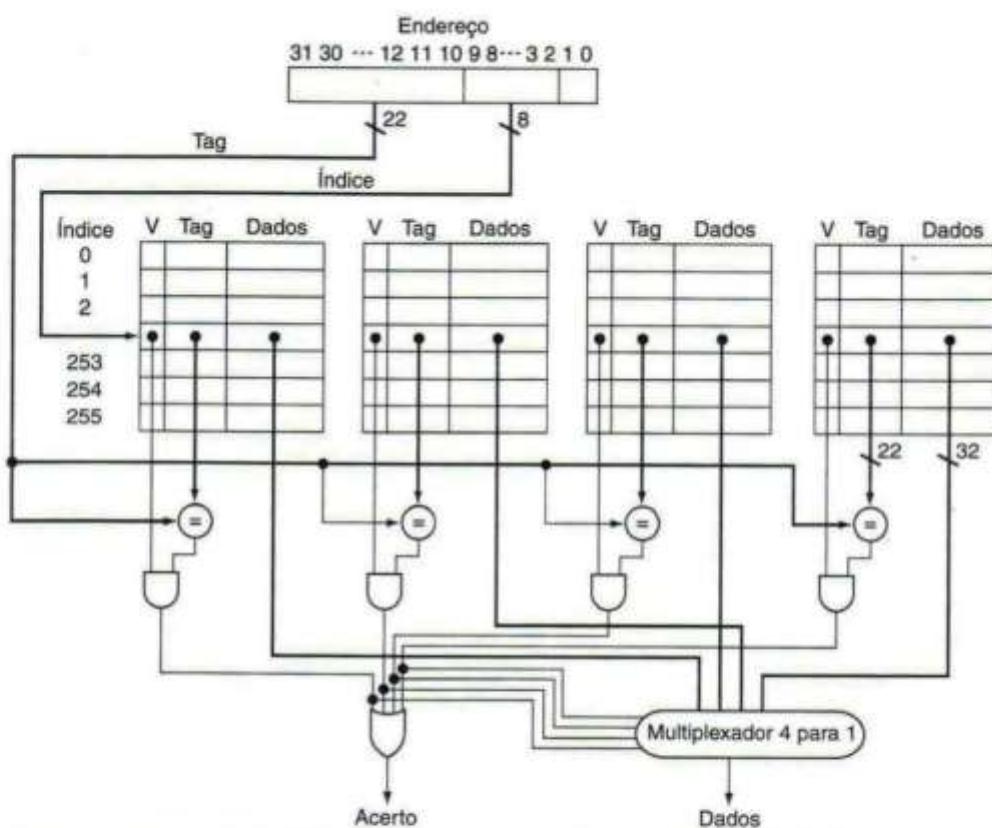


FIGURA 7.17 A implementação de uma cache associativa por conjunto de quatro vias exige quatro comparadores e um multiplexador de 4 para 1. Os comparadores determinam qual elemento do conjunto selecionado (se houver) corresponde à tag. A saída dos comparadores é usada para selecionar os dados de um dos quatro blocos do conjunto indexado, usando um multiplexador com um sinal de seleção decodificado. Em algumas implementações, a saída permite que sinais nas partes de dados das RAMs de cache possam ser usados para selecionar a entrada no conjunto que controla a saída. A saída permite que o sinal venha dos comparadores, fazendo com que o elemento correspondente controle as saídas de dados. Essa organização elimina a necessidade do multiplexador.

A escolha entre mapeamento direto, associativo por conjunto ou totalmente associativo em qualquer hierarquia de memória dependerá do custo de uma falha em comparação com o custo da implementação da associatividade, ambos em tempo e em hardware extra.

TAMANHO DAS TAGS VERSUS ASSOCIATIVIDADE DO CONJUNTO

EXEMPLO

O acréscimo da associatividade requer mais comparadores e mais bits de tag por bloco de cache. Considerando uma cache de 4K blocos, um tamanho de bloco de quatro words e um endereço de 32 bits, encontre o número total de conjuntos e o número total de bits de tag para caches que são diretamente mapeadas, associativas por conjunto de duas e quatro vias e totalmente associativas.

RESPOSTA

Como existem 16 ($=2^4$) bytes por bloco, um endereço de 32 bits produz $32 - 4 = 28$ bits para serem usados para índice e tag. A cache diretamente mapeada possui um mesmo número de conjuntos e blocos e, portanto, 12 bits de índice, já que $\log_2(4K) = 12$; logo, o número total de bits de tag é $(28 - 12) \times 4K = 16 \times 4K = 64$ Kbits.

Cada grau de associatividade diminui o número de conjuntos por um fator de dois e, portanto, diminui o número de bits usados para indexar a cache por um e aumenta o número de bits na tag por um. Conseqüentemente, para uma cache associativa por conjunto de duas vias, existem 2K de con-

juntos, e o número total de bits de tag é $(28 - 11) \times 2 \times 2K = 34 \times 2K = 68$ Kbits. Para uma cache associativa por conjunto de quatro vias, o número total de conjuntos é 1K, e o número total de bits de tag é $(28 - 10) \times 4 \times 1K = 72 \times 1K = 72$ Kbits.

Para uma cache totalmente associativa, há apenas um conjunto com blocos de 4K de blocos, e a tag possui 28 bits, produzindo um total de $28 \times 4K \times 1 = 112K$ de bits de tag.

Escolhendo que bloco substituir

Quando uma falha ocorre em uma cache diretamente mapeada, o bloco requisitado só pode entrar em exatamente uma posição, e o bloco ocupando essa posição precisa ser substituído. Em uma cache associativa, temos uma escolha de onde colocar o bloco requisitado e, portanto, uma escolha de qual bloco substituir. Em uma cache totalmente associativa, todos os blocos são candidatos à substituição. Em uma cache associativa por conjunto, precisamos escolher entre os blocos do conjunto selecionado.

O esquema mais comum é o **LRU** (**Least Recently Used – usado menos recentemente**), que usamos no exemplo anterior. Em um esquema LRU, o bloco substituído é aquele que não foi usado há mais tempo. A substituição LRU é implementada monitorando quando cada elemento em um conjunto foi usado em relação aos outros elementos no conjunto. Para uma cache associativa por conjunto de duas vias, o controle de quando os dois elementos foram usados pode ser implementado mantendo um único bit em cada conjunto e definindo o bit para indicar um elemento sempre que o elemento é referenciado. Conforme a associatividade aumenta, a implementação do LRU se torna mais difícil; na Seção 7.5, veremos um esquema alternativo para substituição.

LRU (Least Recently Used – usado menos recentemente) Um esquema de substituição em que o bloco substituído é aquele que não foi usado há mais tempo.

Reduzindo a penalidade de falha usando caches multiníveis

Todos os computadores modernos fazem uso de caches. Na maioria dos casos, essas caches são implementadas no mesmo die do microprocessador que forma o processador. Para diminuir a diferença entre as rápidas velocidades de clock dos processadores modernos e o tempo relativamente longo necessário para acessar as DRAMs, muitos microprocessadores suportam um nível adicional de cache. Essa cache de segundo nível, que pode estar no mesmo chip ou fora do chip em um conjunto de SRAMs separado, é acessada sempre que ocorre uma falha na cache primária. Se a cache de segundo nível contiver os dados desejados, a penalidade de falha para a cache de primeiro nível será o tempo de acesso à cache de segundo nível, que será muito menor do que o tempo de acesso à memória principal. Se nem a cache primária nem a secundária contiverem os dados, um acesso à memória principal será necessário, e uma penalidade de falha maior será observada.

Em que grau é significante a melhora de desempenho pelo uso de uma cache secundária? O próximo exemplo nos mostra.

DESEMPENHO DAS CACHES MULTINÍVEL

Suponha que tenhamos um processador com um CPI básico de 1,0, considerando que todas as referências acertem na cache primária e uma velocidade de clock de 5GHz. Considere um tempo de acesso à memória principal de 100ns, incluindo todo o tratamento de falhas. Suponha que a taxa de falhas por instrução na cache primária seja de 2%. O quanto mais rápido será o processador se acrescentarmos uma cache secundária que tenha um tempo de acesso de 5ns para um acerto ou uma falha e que seja grande o suficiente para reduzir a taxa de falhas para a memória principal para 0,5%?

EXEMPLO

A penalidade de falha para a memória principal é

$$\frac{100\text{ns}}{0,2 \frac{\text{ns}}{\text{ciclo de clock}}} = 500 \text{ ciclos de clock}$$

RESPOSTA

O CPI efetivo com um nível de cache é dado por

$$\text{CPI total} = \text{CPI básico} + \text{Ciclos de stall de memória por instrução}$$

Para o processador com um nível de cache,

$$\text{CPI total} = 1,0 + \text{Ciclos de stall de memória por instrução} = 1,0 + 2\% \times 500 = 11,0$$

Com dois níveis de cache, uma falha na cache primária (ou de primeiro nível) pode ser satisfeita pela cache secundária ou pela memória principal. A penalidade da falha para um acesso à cache de segundo nível é

$$\frac{5\text{ns}}{0,2 \frac{\text{ns}}{\text{ciclo de clock}}} = 25 \text{ ciclos de clock}$$

Se a falha for satisfeita na cache secundária, essa será toda a penalidade de falha. Se a falha precisar ir à memória principal, então, a penalidade de falha total será a soma do tempo de acesso à cache secundária e do tempo de acesso à memória principal.

Logo, para uma cache de dois níveis, o CPI total é a soma dos ciclos de stall dos dois níveis de cache e o CPI básico:

$$\begin{aligned}\text{CPI total} &= 1 + \text{Stalls primários por instrução} \\ &\quad + \text{Stalls secundários por instrução} \\ &= 1 + 2\% \times 25 + 0,5\% \times 500 = 1 + 0,5 + 2,5 = 4,0\end{aligned}$$

Portanto, o processador com a cache secundária é mais rápido por um fator de

$$\frac{11,0}{4,0} = 2,8$$

Como alternativa, poderíamos ter calculado os ciclos de stall somando os ciclos de stall das referências que acertam na cache secundária ($(2\% - 0,5\%) \times 25 = 0,4$) e as referências que vão à memória principal, que precisam incluir o custo para acessar a cache secundária, bem como o tempo de acesso à memória principal ($0,5\% \times (25 + 500) = 2,6$). A soma, $1,0 + 0,4 + 2,6$, é novamente 4,0.

As considerações de projeto para uma cache primária e secundária são significativamente diferentes porque a presença da outra cache muda a melhor escolha em comparação com uma cache de nível único. Em especial, uma estrutura de cache de dois níveis permite que a cache primária se concentre em minimizar o tempo de acerto para produzir um ciclo de clock mais curto, enquanto permite que a cache secundária focalize a taxa de falhas para reduzir a penalidade dos longos tempos de acesso à memória.

A interação das duas caches permite esse foco. A penalidade de falha da cache primária é significativamente reduzida pela presença da cache secundária, permitindo que a primária seja menor e tenha uma taxa de falhas mais alta. Para a cache secundária, o tempo de acesso se torna menos importante com a presença da cache primária, já que o tempo de acesso da cache secundária afeta a penalidade de falha da cache primária, em vez de afetar diretamente o tempo de acerto da cache primária ou o tempo de ciclo do processador.

O efeito dessas mudanças nas duas caches pode ser visto comparando cada cache com o projeto ótimo para um nível único de cache. Em comparação com uma cache de nível único, a cache primária de uma **cache multinível** normalmente é menor. Além disso, a cache primária freqüentemente usa um tamanho de bloco menor, para se adequar ao tamanho de cache menor e à penalidade de falha reduzida. Em comparação, a cache secundária normalmente será maior do que em uma cache de nível único, já que o tempo de acesso da cache secundária é menos importante. Com um tamanho total maior, a cache secundária geralmente usará um tamanho de bloco maior do que o apropriado com uma cache de nível único.

cache multinível Uma hierarquia de memória com múltiplos níveis de cache, em vez de apenas uma cache e a memória principal.

Entendendo o desempenho dos programas

No Capítulo 2, vimos que o Quicksort tinha uma vantagem algorítmica sobre o Bubble Sort que não podia ser superada pela otimização de linguagem ou compilador. A Figura 7.18(a) mostra as instruções executadas por item pesquisado pelo Radix Sort em comparação com o Quicksort. Decididamente, para arrays grandes, o Radix Sort possui uma vantagem algorítmica sobre o Quicksort em termos do número de operações. A Figura 7.18(b) mostra o tempo por chave em vez das instruções executadas. Podemos ver que as linhas começam na mesma trajetória da Figura 7.18(a), mas, então, a linha do Radix Sort diverge conforme os dados a serem ordenados aumentam. O que está ocorrendo?

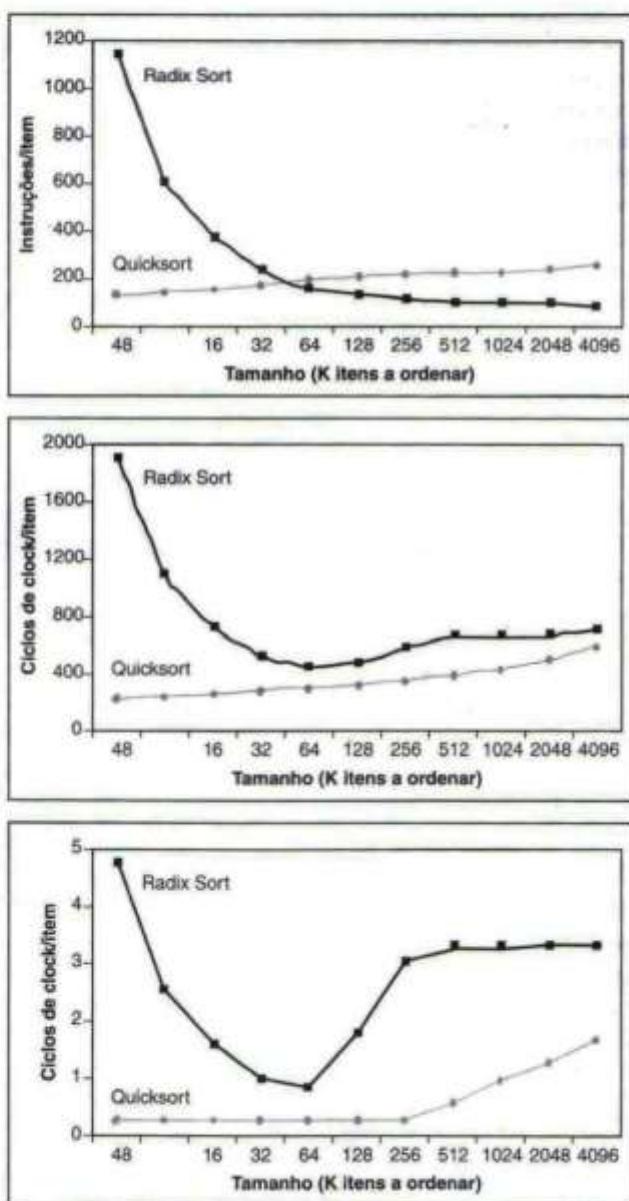


FIGURA 7.18 Comparando o Quicksort e o Radix Sort por (a) instruções executadas por item ordenado, (b) tempo por item ordenado e (c) falhas de cache por item ordenado. Esses dados são de um documento de LaMarca e Ladner [1996]. Embora os números mudassem para computadores mais novos, a idéia ainda permanece. Devido a esses resultados, foram criadas novas versões do Radix Sort que levam a hierarquia de memória em consideração, para readquirir suas vantagens logarítmicas (veja a Seção 7.7). A idéia básica das otimizações de cache é usar todos os dados em um bloco repetidamente antes de serem substituídos em uma falha.

A Figura 7.18 (c) responde olhando as falhas de cache por item ordenado: o Quicksort possui muito menos falhas por item a ser ordenado.

Infelizmente, a análise algorítmica padrão ignora o impacto da hierarquia de memória. À medida que velocidades de clock mais altas e a Lei de Moore permitem aos arquitetos compactarem todo o desempenho de um fluxo de instruções, um uso correto da hierarquia de memória é fundamental para a obtenção de um alto desempenho. Como dissemos na introdução, entender o comportamento da hierarquia de memória é vital para compreender o desempenho dos programas nos computadores atuais.

taxa de falhas global

A fração das referências que falham em todos os níveis de uma cache multinível.

taxa de falhas local

A fração das referências a um nível de uma cache que falham; usada em hierarquias multiníveis.

Detalhamento: caches multiníveis envolvem diversas complicações. Primeiro, agora existem vários tipos diferentes de falhas e taxas de falhas correspondentes. No exemplo da página 377, vimos a taxa de falhas da cache primária e a **taxa de falhas global** – a fração das referências que falharam em todos os níveis de cache. Há também uma taxa de falhas para a cache secundária, que é a taxa de todas as falhas na cache secundária dividida pelo número de acessos. Essa taxa de falhas é chamada de **taxa de falhas local** da cache secundária. Como a cache primária filtra os acessos, especialmente aqueles com boa localidade espacial e temporal, a taxa de falhas local da cache secundária é muito mais alta do que a taxa de falhas global. Para o exemplo na página 377, podemos calcular a taxa de falhas local da cache secundária como: $0,5\% / 2\% = 25\%$. Felizmente, a taxa de falhas global determina a frequência com que precisamos acessar a memória principal.

Complicações adicionais surgem porque as caches podem ter diferentes tamanhos de bloco para corresponder ao tamanho total maior ou menor. De forma semelhante, a associatividade da cache pode mudar. As caches on-chip em geral são construídas com associatividade de quatro ou mais, enquanto caches off-chip raramente têm associatividade maior que dois. As caches on-chip L1 tendem a ter associatividade mais baixa do que as caches on-chip L2, já que o tempo de acerto rápido é mais importante para caches L1. Essas mudanças no tamanho de bloco e na associatividade introduzem complicações na modelagem das caches, que geralmente significam que todos os níveis precisam ser simulados juntos para que o comportamento seja compreendido.

Detalhamento: com processadores que usam execução fora de ordem, o desempenho é mais complexo, já que executam instruções durante a penalidade de falha. Em vez da taxa de falhas de instruções e da taxa de falhas de dados, usamos falhas por instrução e esta fórmula:

$$\frac{\text{Ciclos de stall de memória}}{\text{Instrução}} = \frac{\text{Falhas}}{\text{Instrução}} \times (\text{Latência de falha total} - \text{Latência de falha sobreposta})$$

Não há uma maneira geral de calcular a latência de falha sobreposta; portanto, as avaliações das hierarquias de memória para processadores com execução fora de ordem inevitavelmente exigem simulações do processador e da hierarquia de memória. Somente vendo a execução do processador durante cada falha é que podemos ver se o processador sofre stall esperando os dados ou simplesmente encontra outro trabalho para fazer. Uma regra é que o processador muitas vezes oculta a penalidade de falha para uma falha de cache L1 que acerta na cache L2, mas raramente oculta uma falha para a cache L2.

Detalhamento: o desafio do desempenho para algoritmos é que a hierarquia de memória varia entre diferentes implementações da mesma arquitetura no tamanho de cache, na associatividade, no tamanho de bloco e no número de caches. Para fazer frente a essa variabilidade, algumas bibliotecas numéricas recentes parametrizam os seus algoritmos e, então, pesquisam o espaço de parâmetros em tempo de execução para encontrar a melhor combinação para um determinado computador.

Verifique você mesmo

Qual das afirmações a seguir geralmente é verdadeira sobre um projeto com múltiplos níveis de cache?

- As caches de primeiro nível são mais focalizadas no tempo de acerto e as caches de segundo nível se preocupam mais com a taxa de falhas.
- As caches de primeiro nível são mais focalizadas na taxa de falhas e as caches de segundo nível se preocupam mais com o tempo de acerto.

Resumo

Nesta seção, vamos nos concentrar em três tópicos: o desempenho da cache, o uso da associatividade para reduzir as taxas de falhas e o uso das hierarquias de cache multinível para reduzir as penalidades de falha.

Como o número total de ciclos gastos em um programa é a soma dos ciclos de processador e os ciclos de stall de memória, o sistema de memória pode ter um efeito significativo sobre o tempo de execução do programa. Na verdade, à medida que os processadores se tornam mais rápidos (reduzindo o CPI ou aumentando a velocidade de clock, ou ambos), aumenta o efeito relativo dos ciclos de stall de memória, tornando os bons sistemas de memória fundamentais para alcançar um alto desempenho. O número de ciclos de stall de memória depende da taxa de falhas e da penalidade de falha. O desafio, como veremos na Seção 7.5, é reduzir um desses fatores sem afetar significativamente os outros fatores críticos na hierarquia de memória.

Para reduzir a taxa de falhas, examinamos o uso dos esquemas de posicionamento associativos. Esses esquemas podem reduzir a taxa de falhas de uma cache permitindo um posicionamento mais flexível dos blocos dentro da cache. Os esquemas totalmente associativos permitem que os blocos sejam posicionados em qualquer lugar, mas também exigem que todos os blocos da cache sejam pesquisados para atender a uma requisição. Essa pesquisa normalmente é implementada usando um comparador por bloco de cache e pesquisando as tags em paralelo. O custo dos comparadores torna as caches totalmente associativas inviáveis. As caches associativas por conjunto são uma alternativa prática, já que precisamos pesquisar apenas entre os elementos de um único conjunto, escolhido por indexação. As caches associativas por conjunto apresentam taxas de falhas mais altas, mas são mais rápidas de serem acessadas. O grau de associatividade que produz o melhor desempenho depende da tecnologia e dos detalhes da implementação.

Finalmente, examinamos as caches multiníveis como uma técnica para reduzir a penalidade de falha permitindo uma cache secundária maior para tratar falhas na cache primária. As caches de segundo nível se tornaram comuns quando os projetistas descobriram que o silício limitado e as metas de altas velocidades de clock impedem que as caches primárias se tornem grandes. A cache secundária, que normalmente é 10 ou mais vezes maior do que a cache primária, trata muitos acessos que falham na cache primária. Nesses casos, a penalidade de falha é aquela do tempo de acesso à cache secundária (em geral, menos de 10 ciclos de processador) contra o tempo de acesso à memória (normalmente mais de 100 ciclos de processador). Assim como na associatividade, as negociações de projeto entre o tamanho da cache secundária e seu tempo de acesso dependem de vários aspectos de implementação.

7.4 Memória virtual

...foi inventado um sistema para fazer a combinação entre os sistemas centrais de memória e os tambores de discos aparecer para o programador como um depósito de nível único, com as transferências necessárias ocorrendo automaticamente.

Kilburn et al., *One-level storage system*, 1962

Na seção anterior, vimos como as caches fornecem acesso rápido às partes recentemente usadas do código e dos dados de um programa. Da mesma forma, a memória principal pode agir como uma “cache” para o armazenamento secundário, normalmente implementado com discos magnéticos. Essa técnica é chamada de **memória virtual**. Historicamente, houve duas motivações principais para a memória virtual: permitir o compartilhamento seguro e eficiente da memória entre vários programas e remover os transtornos de programação de uma quantidade pequena e limitada de memória principal. Quatro décadas após sua invenção, o primeiro motivo é o que ainda predomina.

Considere um grupo de programas executados ao mesmo tempo em um computador. A memória total exigida por todos os programas pode ser muito maior do que a quantidade de memória principal

memória virtual Uma técnica que usa a memória principal como uma “cache” para armazenamento secundário.

disponível no computador, mas apenas uma fração dessa memória está sendo usada ativamente em um dado momento. A memória principal precisa conter apenas as partes ativas dos muitos programas, exatamente como uma cache contém apenas a parte ativa de um programa. Portanto, o princípio da localidade possibilita a memória virtual e as caches, e a memória virtual nos permite compartilhar eficientemente o processador e a memória principal. É claro que, para permitir que vários programas compartilhem a mesma memória, precisamos ser capazes de proteger os programas uns dos outros, garantindo que um programa só possa ler e escrever as partes da memória principal atribuídas a ele.

Não podemos saber quais programas irão compartilhar a memória com outros programas quando os compilamos. Na verdade, os programas que compartilham a memória mudam dinamicamente enquanto estão sendo executados. Devido a essa interação dinâmica, gostaríamos de compilar cada programa para o seu próprio *espaço de endereçamento* – faixa distinta dos locais de memória acessível apenas a esse programa. A memória virtual implementa a tradução do espaço de endereçamento de um programa para os **endereços físicos**. Esse processo de tradução impõe a **proteção** do espaço de endereçamento de um programa contra outros programas.

A segunda motivação para a memória virtual é permitir que um único programa do usuário exceda o tamanho da memória principal. Antigamente, se um programa se tornasse muito grande para a memória, cabia ao programador fazê-lo se adequar. Os programadores dividiam os programas em partes e, então, identificavam aquelas mutuamente exclusivas. Esses *overlays* eram carregados ou descarregados sob o controle do programa do usuário durante a execução, com o programador garantindo que o programa nunca tentaria acessar um overlay que não estivesse carregado e que os overlays carregados nunca excederiam o tamanho total da memória. Os overlays eram tradicionalmente organizados como módulos, cada um contendo código e dados. As chamadas entre procedimentos em módulos diferentes levavam um módulo a se sobrepor a outro.

Como você pode bem imaginar, essa responsabilidade era uma carga substancial para os programadores. A memória virtual, criada para aliviar os programas dessa dificuldade, gerencia automaticamente os dois níveis da hierarquia de memória representados pela memória principal (às vezes, chamada de *memória física* para distingui-la da memória virtual) e pelo armazenamento secundário.

Embora os conceitos aplicados na memória virtual e nas caches sejam os mesmos, suas diferentes raízes históricas levaram ao uso de uma terminologia diferente. Um bloco de memória virtual é chamado de *página* e uma falha da memória virtual é chamada de **falta de página**. Com a memória virtual, o processador produz um **endereço virtual**, traduzido por uma combinação de hardware e software para um *endereço físico*, que, por sua vez, pode ser usado para acessar a memória principal. A Figura 7.19 mostra a memória endereçada virtualmente com páginas mapeadas na memória principal. Esse processo é chamado de *mapeamento de endereço* ou **tradução de endereço**. Hoje, os dois níveis de hierarquia de memória controlados pela memória virtual são as DRAMs e os discos magnéticos (veja o Capítulo 1). Se voltarmos à nossa analogia da biblioteca, podemos pensar no endereço virtual como o título de um livro e no endereço físico como seu local na biblioteca.

A memória virtual também simplifica o carregamento do programa para execução fornecendo *relocação*. A relocação mapeia os endereços virtuais usados por um programa para diferentes endereços físicos antes que os endereços sejam usados para acessar a memória. Essa relocação nos permite carregar o programa em qualquer lugar na memória principal. Além disso, todos os sistemas de memória virtual em uso atualmente relocam o programa como um conjunto de blocos (páginas) de tamanho fixo, eliminando, assim, a necessidade de encontrar um bloco contíguo de memória para alojar um programa; em vez disso, o sistema operacional só precisa encontrar um número suficiente de páginas na memória principal. Antes, os problemas de relocação exigiam hardware especial e suporte especial do sistema operacional; hoje, a memória virtual também oferece essa função.

Na memória virtual, o endereço é desmembrado em um *número de página virtual* e um *offset de página*. A Figura 7.20 mostra a tradução do número de página virtual para um *número de página física*. O número de página física constitui a parte mais significativa do endereço físico, enquanto o offset de página, que não é alterado, constitui a parte menos significativa. O número de bits no campo offset de página determina o tamanho da página. O número de páginas endereçáveis com o endereço

endereço físico

Um endereço na memória principal.

proteção Um conjunto de mecanismos para garantir que múltiplos processos compartilhando processador, memória ou dispositivos de E/S não possam interferir, intencionalmente ou não, um com o outro, lendo ou escrevendo dados no outro. Esses mecanismos também isolam o sistema operacional de um processo de usuário.

falta de página

Um evento que ocorre quando uma página acessada não está presente na memória principal.

endereço virtual

Um endereço que corresponde a um local no espaço virtual e é traduzido pelo mapeamento de endereço para um endereço físico quando a memória é acessada.

tradução de endereço

Também chamada de **mapeamento de endereço**. O processo pelo qual um endereço virtual é mapeado para um endereço usado para acessar a memória.

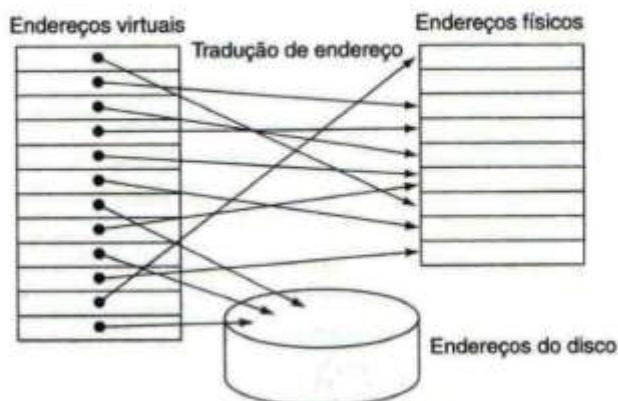


FIGURA 7.19 Na memória virtual, os blocos de memória (chamados de páginas) são mapeados de um conjunto de endereços (chamados de endereços virtuais) em outro conjunto (chamado de endereços físicos). O processador gera endereços virtuais enquanto a memória é acessada usando endereços físicos. Tanto a memória virtual quanto a memória física são desmembradas em páginas, de modo que uma página virtual é realmente mapeada em uma página física. Naturalmente, também é possível que uma página virtual esteja ausente da memória principal e não seja mapeada para um endereço físico, residindo no disco em vez disso. As páginas físicas podem ser compartilhadas fazendo dois endereços virtuais apontarem para o mesmo endereço físico. Essa capacidade é usada para permitir que dois programas diferentes compartilhem dados ou código.

virtual não precisa corresponder ao número de páginas endereçáveis com o endereço físico. Ter um número de páginas virtuais maior do que as páginas físicas é a base para a ilusão de uma quantidade de memória virtual essencialmente ilimitada.

Muitas escolhas de projeto nos sistemas de memória virtual são motivadas pelo alto custo de uma falha, que, na memória virtual, tradicionalmente é chamada de *falta de página*. Uma falta de página levará milhões de ciclos de clock para ser processada. (A tabela na página 355 mostra que a memória principal é aproximadamente 100.000 vezes mais rápida do que o disco.) Essa enorme penalidade de falha, dominada pelo tempo para obter a primeira word para tamanhos de página típicos, leva a várias decisões importantes nos sistemas de memória virtual:

- As páginas devem ser grandes o suficiente para tentar amortizar o longo tempo de acesso. Tamanhos de 4KB a 16KB são comuns atualmente. Novos sistemas de desktop e servidor estão sendo desenvolvidos para suportar páginas de 32KB e 64KB, embora novos sistemas embutidos estejam indo na outra direção, para páginas de 1KB.
- Organizações que reduzem a taxa de faltas de página são atraentes. A principal técnica usada aqui é permitir o posicionamento totalmente associativo das páginas na memória.
- As faltas de página podem ser tratadas em nível de software porque o overhead será pequeno se comparado com o tempo de acesso ao disco. Além disso, o software pode se dar ao luxo de usar algoritmos inteligentes para escolher como posicionar as páginas, já que mesmo pequenas reduções na taxa de falhas compensarão o custo desses algoritmos.
- O write-through não funcionará para a memória virtual, visto que as escritas levam muito tempo. Em vez disso, os sistemas de memória virtual usam write-back.

As próximas subseções tratam desses fatores no projeto de memória virtual.

Detalhamento: embora normalmente imaginemos os endereços virtuais como muito maiores do que os endereços físicos, o contrário pode ocorrer quando o tamanho de endereço do processador é pequeno em relação ao estado da tecnologia de memória. Nenhum programa único pode se beneficiar, mas um grupo de programas executados ao mesmo tempo pode se beneficiar de não precisar ser trocado para a memória ou de ser executado em processadores paralelos. Haja vista que a Lei de Moore se aplica à DRAM, os processadores de 32 bits já são problemáticos para servidores e logo serão para desktops.

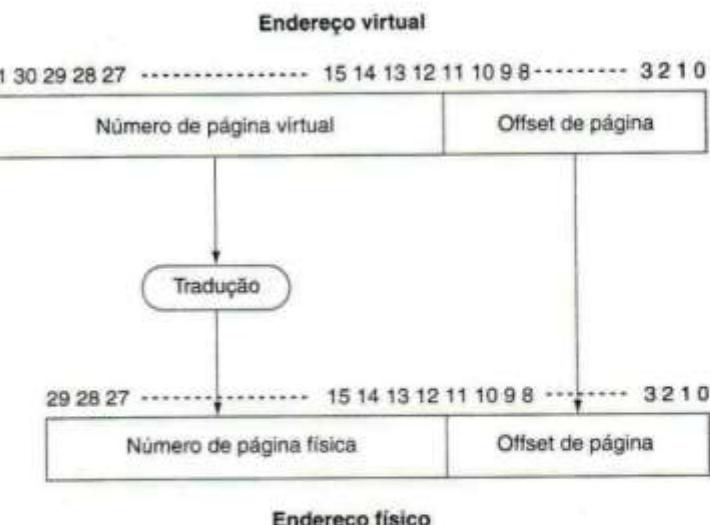


FIGURA 7.20 Mapeamento de um endereço virtual em um endereço físico. O tamanho de página é $2^{12} = 4KB$. O número de páginas físicas permitido na memória é 2^{18} , já que o número de página física contém 18 bits. Portanto, a memória principal pode ter, no máximo, 1GB, enquanto o espaço de endereço virtual possui 4GB.

segmentação Um esquema de mapeamento de endereço de tamanho variável em que um endereço consiste em duas partes: um número de segmento, que é mapeado para um endereço físico, e um offset de segmento.

Detalhamento: a discussão da memória virtual neste livro focaliza a paginação, que usa blocos de tamanho fixo. Há também um esquema de blocos de tamanho variável chamado **segmentação**. Na segmentação, um endereço consiste em duas partes: um número de segmento e um offset de segmento. O registrador de segmento é mapeado para um endereço físico e o offset é somado para encontrar o endereço físico real. Como o segmento pode variar em tamanho, uma verificação de limites é necessária para garantir que o offset esteja dentro do segmento. O principal uso da segmentação é suportar métodos de proteção mais avançados e compartilhar um espaço de endereçamento. A maioria dos livros de sistemas operacionais contém extensas discussões sobre a segmentação comparada com a paginação e sobre o uso da segmentação para compartilhar logicamente o espaço de endereçamento. A principal desvantagem da segmentação é que ela divide o espaço de endereço em partes logicamente separadas que precisam ser manipuladas como um endereço de duas partes: o número de segmento e o offset. A paginação, por outro lado, torna o limite entre o número de página e o offset invisível aos programadores e compiladores.

Os segmentos também têm sido usados como um método para estender o espaço de endereçamento sem mudar o tamanho da word do computador. Essas tentativas têm sido mal-sucedidas devido à dificuldade e ao ônus de desempenho inerentes a um endereço de duas partes, dos quais os programadores e compiladores precisam estar cientes.

Muitas arquiteturas dividem o espaço de endereçamento em grandes blocos de tamanho fixo que simplificam a proteção entre o sistema operacional e os programas de usuário e aumentam a eficiência da paginação. Embora essas divisões normalmente sejam chamadas de "segmentos", esse mecanismo é muito mais simples do que a segmentação de tamanho de bloco variável e não é visível aos programas do usuário; discutiremos isso em mais detalhes em breve.

Posicionando uma página e a encontrando novamente

Em razão da penalidade incrivelmente alta de uma falta de página, os projetistas reduzem a freqüência das faltas de página otimizando o posicionamento das páginas. Se permitirmos que uma página virtual seja mapeada em qualquer página física, o sistema operacional, então, pode escolher substituir qualquer página que desejar quando ocorrer uma falta de página. Por exemplo, o sistema operacional pode usar um sofisticado algoritmo e complexas estruturas de dados, que monitoram o uso de páginas, para tentar escolher uma página que não será necessária por um longo tempo. A capacidade de usar um esquema de substituição inteligente e flexível reduz a taxa de faltas de página e simplifica o uso do posicionamento de páginas totalmente associativo.

Como mencionamos na Seção 7.3, a dificuldade em usar posicionamento totalmente associativo está em localizar uma entrada, já que ela pode estar em qualquer lugar no nível superior da hierarquia. Uma pesquisa completa é impraticável. Nos sistemas de memória virtual, localizamos páginas usando uma tabela que indexa a memória; essa estrutura é chamada de **tabela de páginas** e reside na memória. Uma tabela de páginas é indexada pelo número de página do endereço virtual para descobrir o número da página física correspondente. Cada programa possui sua própria tabela de páginas, que mapeia o espaço de endereçamento virtual desse programa para a memória principal. Em nossa analogia da biblioteca, a tabela de páginas corresponde a um mapeamento entre os títulos dos livros e os locais da biblioteca. Exatamente como o catálogo de cartões pode conter entradas para livros em outra biblioteca ou campus em vez da biblioteca local, veremos que a tabela de páginas pode conter entradas para páginas não presentes na memória. Para indicar o local da tabela de páginas na memória, o hardware inclui um registrador que aponta para o início da tabela de páginas; esse registrador é chamado de *registrador de tabela de páginas*. Por enquanto, considere que a tabela de páginas esteja em uma área fixa e contigua da memória.

tabela de páginas

A tabela com as traduções de endereço virtual para físico em um sistema de memória virtual. A tabela, armazenada na memória, normalmente é indexada pelo número de página virtual; cada entrada na tabela contém o número da página física para essa página virtual se a página estiver atualmente na memória.

Interface hardware/software

A tabela de páginas, juntamente com o contador de programa e os registradores, especifica o *estado* de um programa. Se quisermos permitir que outro programa use o processador, precisamos salvar esse estado. Mais tarde, após restaurar esse estado, o programa pode continuar a execução. Frequentemente nos referimos a esse estado como um *processo*. O processo é considerado *ativo* quando está de posse do processador; caso contrário, ele é considerado *inativo*. O sistema operacional pode tornar um processo ativo carregando o estado do processo, incluindo o contador de programa, o que irá iniciar a execução no valor salvo do contador de programa.

O espaço de endereçamento do processo, e, consequentemente, todos os dados que ele pode acessar na memória, é definido pela sua tabela de páginas, que reside na memória. Em vez de salvar a tabela de páginas inteira, o sistema operacional simplesmente carrega o registrador de tabela de páginas para apontar para a tabela de páginas do processo que ele quer tornar ativo. Cada processo possui sua própria tabela de páginas, já que diferentes processos usam os mesmos endereços virtuais. O sistema operacional é responsável por alocar a memória física e atualizar as tabelas de páginas, de modo que os espaços de endereço virtuais dos diferentes processos não colidam. Como veremos em breve, o uso de tabelas de páginas separadas também fornece proteção de um processo contra outro.

A Figura 7.21 usa o registrador de tabela de páginas, o endereço virtual e a tabela de páginas indicada para mostrar como o hardware pode formar um endereço físico. Um bit de validade é usado em cada entrada de tabela de páginas, exatamente como fariam em uma cache. Se o bit estiver desligado, a página não está presente na memória principal e ocorre uma falta de página. Se o bit estiver ligado, a página está na memória e a entrada contém o número de página física.

Como a tabela de páginas contém um mapeamento para toda página virtual possível, nenhuma tag é necessária. Na terminologia da cache, o índice usado para acessar a tabela de páginas consiste no endereço de bloco inteiro, que é o número de página virtual.

Faltas de página

Se o bit de validade para uma página virtual estiver desligado, ocorre uma falta de página. O sistema operacional precisa receber o controle. Essa transferência é feita pelo mecanismo de exceção, que abordaremos posteriormente nesta seção. Quando o sistema operacional obtém o controle, ele precisa encontrar a página no próximo nível da hierarquia (geralmente o disco magnético) e decidir onde colocar a página requisitada na memória principal.

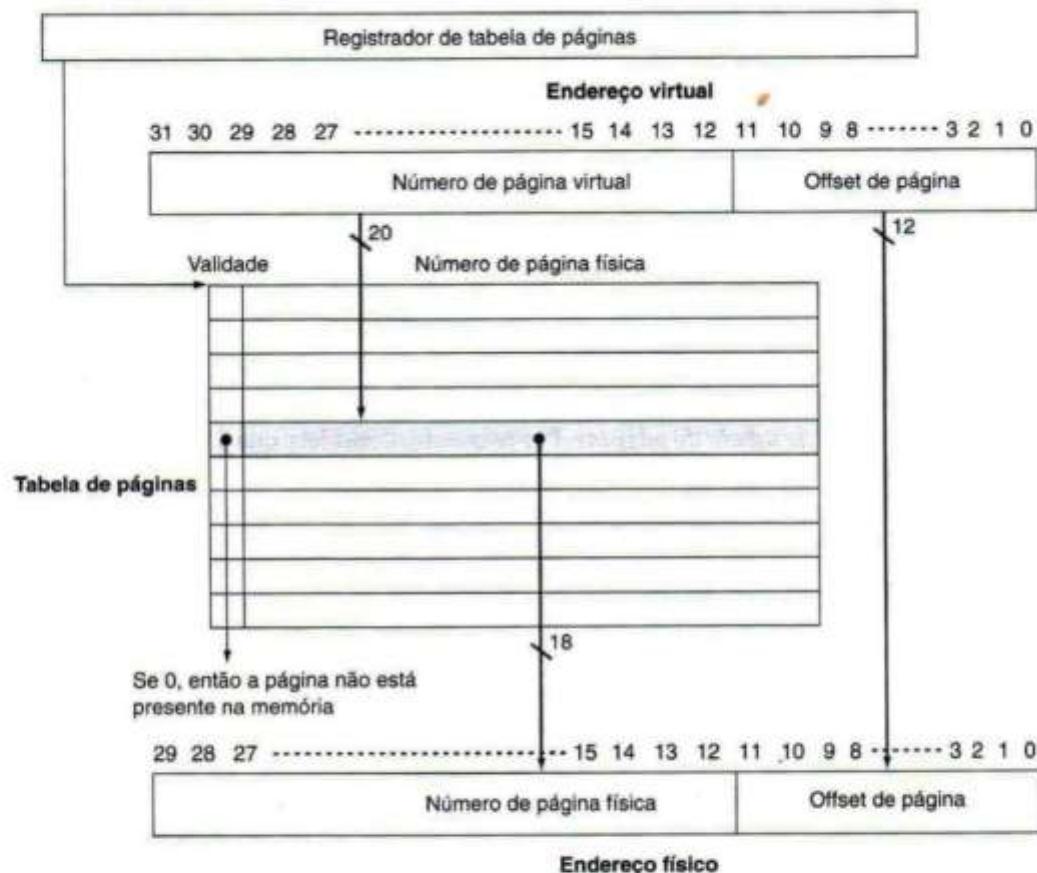


FIGURA 7.21 A tabela de páginas é indexada pelo número de página virtual para obter a parte correspondente do endereço físico. O endereço inicial da tabela de páginas é dado pelo ponteiro da tabela de páginas. Nessa figura, o tamanho de página é 2^{12} bytes, ou 4KB. O espaço de endereço virtual é 2^{32} bytes, ou 4GB, e o espaço de endereçamento físico é 2^{30} bytes, que permite uma memória principal de até 1GB. O número de entradas na tabela de páginas é 2^{20} , ou um milhão de entradas. O bit de validade para cada entrada indica se o mapeamento é legal. Se ele estiver desligado, a página não está presente na memória. Embora a entrada de tabela de páginas mostrada aqui só precise ter 19 bits de largura, ela normalmente seria arredondada para 32 bits para facilitar a indexação. Os bits extras seriam usados para armazenar informações adicionais que precisam ser mantidas página a página, como a proteção.

O endereço virtual por si só não diz imediatamente onde está a página no disco. Voltando à nossa analogia da biblioteca, não podemos encontrar o local de um livro nas estantes apenas sabendo seu título. Precisamos ir ao catálogo e consultar o livro, obter um endereço para o local nas estantes. Da mesma forma, em um sistema de memória virtual, precisamos monitorar o local no disco de cada página em um espaço de endereçamento virtual.

Como não sabemos de antemão quando uma página na memória será escolhida para ser substituída, o sistema operacional normalmente cria o espaço no disco para todas as páginas de um processo no momento em que ele cria o processo. Esse espaço do disco é chamado de **área de swap**. Nesse momento, o sistema operacional também cria uma estrutura para registrar onde cada página virtual está armazenada no disco. Essa estrutura de dados pode ser parte da tabela de páginas ou pode ser uma estrutura de dados auxiliar indexada da mesma maneira que a tabela de páginas. A Figura 7.22 mostra a organização quando uma única tabela contém o número de página física ou o endereço de disco.

O sistema operacional também cria uma estrutura de dados que controla quais processos e quais endereços virtuais usam cada página física. Quando ocorre uma falta de página, se todas as páginas na memória principal estiverem em uso, o sistema operacional precisa escolher uma página para substituir. Como queremos minimizar o número de faltas de página, a maioria dos sistemas operacionais tenta escolher uma página que supostamente não será necessária no futuro próximo. Usando o

área de swap O espaço no disco reservado para o espaço de memória virtual completo de um processo.

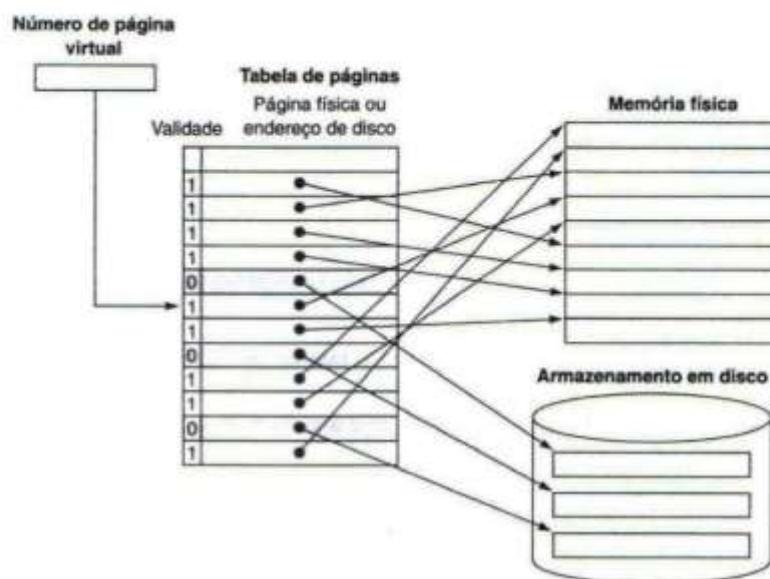


FIGURA 7.22 A tabela de páginas mapeia cada página na memória virtual em uma página na memória principal ou em uma página armazenada em disco, que é o próximo nível na hierarquia. O número de página virtual é usado para indexar a tabela de páginas. Se o bit de validade estiver ligado, a tabela de páginas fornece o número de página física (ou seja, o endereço inicial da página na memória) correspondente à página virtual. Se o bit de validade estiver desligado, a página reside atualmente apenas no disco, em um endereço de disco especificado. Em muitos sistemas, a tabela de endereços de página física e endereços de página de disco, embora sendo logicamente uma única tabela, é armazenada em duas estruturas de dados separadas. As tabelas duplas se justificam, em parte, porque precisamos manter os endereços de disco de todas as páginas, mesmo que elas estejam atualmente na memória principal. Lembre-se de que as páginas na memória principal e as páginas no disco são idênticas em tamanho.

passado para prever o futuro, os sistemas operacionais seguem o esquema de substituição LRU (Least Recently Used – usado menos recentemente), que mencionamos na Seção 7.3. O sistema operacional procura a página usada menos recentemente, fazendo a suposição de que uma página que não foi usada por um longo período é menos provável de ser usada do que uma página acessada mais recentemente. As páginas substituídas são escritas na área de swap do disco. Caso você esteja curioso, o sistema operacional é apenas outro processo, e essas tabelas controlando a memória estão na memória; os detalhes dessa aparente contradição serão explicados em breve.

Por exemplo, suponha que as referências às páginas (na ordem) fossem 10, 12, 9, 7, 11, 10 e, então, referenciamos a página 8, que não estava presente na memória. A página LRU é a 12; na substituição LRU, substituiríamos a página 12 na memória principal pela página 8. Se a próxima referência também gerasse uma falta de página, substituiríamos a página 9, já que ela seria, então, a LRU entre as páginas presentes na memória.

Interface hardware/software

Implementar um esquema de LRU completamente preciso é muito caro, uma vez que isso requer atualizar uma estrutura de dados a *cada* referência à memória. Como alternativa, a maioria dos sistemas operacionais aproxima a LRU monitorando que páginas foram e que páginas não foram usadas recentemente. Para ajudar o sistema operacional a estimar as páginas LRU, alguns computadores fornecem um **bit de referência** ou **bit de uso**, que é ligado sempre que uma página é acessada. O sistema operacional limpa periodicamente os bits de referência e, depois, os registra para que ele possa determinar que páginas foram tocadas durante um determinado período. Com essas informações de uso, o sistema operacional pode selecionar uma página que está entre as referenciadas menos recentemente (detectadas tendo seu bit de referência desligado). Se esse bit não for fornecido pelo hardware, o sistema operacional precisará encontrar outra maneira de estimar que páginas foram acessadas.

bit de referência

Também chamado de **bit de uso**. Um campo que é ligado sempre que uma página é acessada e que é usado para implementar LRU ou outros esquemas de substituição.

Detalhamento: com um endereço virtual de 32 bits, página de 4KB e 4 bytes por entrada da tabela de páginas, podemos calcular o tamanho total da tabela de páginas:

$$\begin{aligned} \text{Número de entradas da tabela de páginas} &= \frac{2^{32}}{2^{12}} = 2^{20} \\ \text{Tamanho da tabela de páginas} &= 2^{20} \text{ entradas da tabela de páginas} \times 2^2 \frac{\text{bytes}}{\text{entrada de tabela de páginas}} \\ &= 4\text{MB} \end{aligned}$$

Ou seja, precisaríamos usar 4MB da memória para cada programa em execução em um dado momento. Em um computador com dezenas ou centenas de programas ativos e uma tabela de páginas de tamanho fixo, a maioria ou toda a memória estaria ocupada com tabelas de páginas!

Diversas técnicas são usadas para reduzir a quantidade de armazenamento necessário para a tabela de páginas. As cinco técnicas a seguir visam a reduzir o armazenamento máximo total necessário, bem como minimizar a memória principal dedicada às tabelas de páginas:

1. A técnica mais simples é manter um registrador de limite que restrinja o tamanho da tabela de páginas para um determinado processo. Se o número de página virtual se tornar maior do que o conteúdo do registrador de limite, entradas precisarão ser incluídas na tabela de páginas. Essa técnica permite que a tabela de páginas cresça à medida que um processo consome mais espaço. Assim, a tabela de páginas só será maior se o processo estiver usando muitas páginas do espaço de endereçamento virtual. Essa técnica exige que o espaço de endereçamento se expanda apenas em uma direção.
2. Permitir o crescimento apenas em uma direção não é o bastante, já que a maioria das linguagens exige duas áreas cujo tamanho seja expansível: uma área contém a pilha e a outra contém o heap. Devido à sua dualidade, é conveniente dividir a tabela de páginas e deixá-la crescer do endereço mais alto para baixo, assim como do endereço mais baixo para cima. Isso significa que haverá duas tabelas de páginas separadas e dois limites separados. O uso de duas tabelas de páginas divide o espaço de endereçamento em dois segmentos. O bit mais significativo de um endereço normalmente determina que segmento – e, portanto, que tabela de páginas – deve ser usado para esse endereço. Como o segmento é especificado pelo bit de endereço mais significativo, cada segmento pode ter a metade do tamanho do espaço de endereçamento. Um registrador de limite para cada segmento especifica o tamanho atual do segmento, que cresce em unidades de páginas. Esse tipo de segmentação é usado por muitas arquiteturas, inclusive MIPS. Diferente do tipo de segmentação abordado na seção “Detalhamento” da página 387, essa forma de segmentação é invisível ao programa de aplicação, embora não para o sistema operacional. A principal desvantagem desse esquema é que ele não funciona bem quando o espaço de endereçamento é usado de uma maneira esparsa e não como um conjunto contíguo de endereços virtuais.
3. Outro método para reduzir o tamanho da tabela de páginas é aplicar uma função de hashing no endereço virtual de modo que a estrutura de dados da tabela de páginas precise ser apenas do tamanho do número de páginas físicas na memória principal. Essa estrutura é chamada de *tabela de páginas invertida*. É claro que o processo de consulta é um pouco mais complexo com uma tabela de páginas invertida porque não podemos mais simplesmente indexar a tabela de páginas.
4. Múltiplos níveis de tabelas de páginas também podem ser usados para reduzir a quantidade total de armazenamento para a tabela de páginas. O primeiro nível mapeia grandes blocos de tamanho fixo do espaço de endereçamento virtual, talvez de 64 a 256 páginas no total. Esses grandes blocos são, às vezes, chamados de segmentos, e essa tabela de mapeamento de primeiro nível é chamada de tabela de segmentos, embora os segmentos sejam invisíveis ao usuário. Cada entrada na tabela de segmentos indica se alguma página nesse segmento está alocada e, se estiver, aponta para uma tabela de páginas desse segmento. A tradução de endereços ocorre primeiramente olhando na tabela de segmentos, usando os bits mais significativos do endereço. Se o endereço do segmento for válido, o próximo conjunto de bits mais significativos é usado para indexar a tabela de páginas indicada pela entrada da tabela de segmentos. Esse esquema permite que o espaço de endereçamento seja usado de uma maneira esparsa (vários segmentos não contíguos podem estar ativos), sem precisar alocar a tabela de páginas inteira. Esses esquemas são particularmente úteis com espaços de endereçamento muito grandes e em sistemas de software que exigem alocação não-contígua. A principal desvantagem desse mapeamento de dois níveis é o processo mais complexo para a tradução de endereços.

5. Para reduzir a memória principal real consumida pelas tabelas de páginas, a maioria dos sistemas modernos também permite que as tabelas de páginas sejam paginadas. Embora isso pareça complicado, esse esquema funciona usando os mesmos conceitos básicos da memória virtual e simplesmente permite que as tabelas de páginas residam no espaço de endereçamento virtual. Entretanto, há alguns problemas pequenos mas cruciais, como uma série interminável de faltas de página, que precisam ser evitadas. Como esses problemas são resolvidos é um tema muito detalhado e, em geral, altamente específico ao processador. Em poucas palavras, esses problemas são evitados colocando todas as tabelas de páginas no espaço de endereçamento do sistema operacional e colocando pelo menos algumas das tabelas de páginas para o sistema em uma parte da memória principal que é fisicamente endereçada e está sempre presente – e, portanto, nunca no disco.

E quanto às escritas?

A diferença entre o tempo de acesso para a cache e para a memória principal é de dezenas a centenas de ciclos, e os esquemas write-through podem ser usados, embora precisemos de um buffer de escrita para ocultar ao processador a latência da escrita. Em um sistema de memória virtual, as escritas no próximo nível de hierarquia (disco) levam milhões de ciclos de clock de processador; portanto, construir um buffer de escrita para permitir que o sistema escreva diretamente no disco seria impraticável. Em vez disso, os sistemas de memória virtual precisam usar write-back, realizando as escritas individuais para a página na memória e copiando a página novamente para o disco quando ela é substituída na memória. Essa cópia de volta ao nível inferior na hierarquia é a origem do outro nome para essa técnica de tratamento de escritas, a saber, *copy-back* (ou copiar de volta).

Interface hardware/software

Um esquema write-back possui outra importante vantagem em um sistema de memória virtual. Como o tempo de transferência de disco é pequeno comparado com seu tempo de acesso, copiar de volta uma página inteira é muito mais eficiente do que escrever words individuais novamente no disco. Uma operação write-back, embora mais eficiente do que transferir páginas individuais, ainda é onerosa. Portanto, gostaríamos de saber se uma página *precisa* ser copiada de volta quando escolhemos substituí-la. Para monitorar se uma página foi escrita desde que foi lida para a memória, um *bit de modificação* (dirty bit) é acrescentado à tabela de páginas. O bit de modificação é ligado quando qualquer word em uma página é escrita. Se o sistema operacional escolher substituir a página, o bit de modificação indica se a página precisa ser escrita no disco antes que seu local na memória possa ser cedido a outra página.

Tornando a tradução de endereços rápido: a TLB

Como as tabelas de páginas são armazenadas na memória principal, cada acesso à memória por um programa pode levar, no mínimo, o dobro do tempo: um acesso à memória para obter o endereço físico e um segundo acesso para obter os dados. O segredo para melhorar o desempenho de acesso é bascular-se na localidade da referência à tabela de páginas. Quando uma tradução para um número de página virtual é usada, ela provavelmente será necessária novamente no futuro próximo, pois as referências às words dessa página possuem localidade temporal e também espacial.

Assim, os processadores modernos incluem uma cache especial que controla as traduções usadas recentemente. Essa cache especial de tradução de endereços é tradicionalmente chamada de **TLB** (*translation-lookaside buffer*), embora seja mais correto chamá-la de *cache de tradução*. A TLB corresponde àquele pequeno pedaço de papel que normalmente usamos para registrar o local de um conjunto de livros que consultamos no catálogo; em vez de pesquisar continuamente o catálogo inteiro, registramos o local de vários livros e usamos o pedaço de papel como uma cache da biblioteca.

TLB (translation-lookaside buffer) Uma cache que monitora os mapeamentos de endereços recentemente usados para evitar um acesso à tabela de páginas.

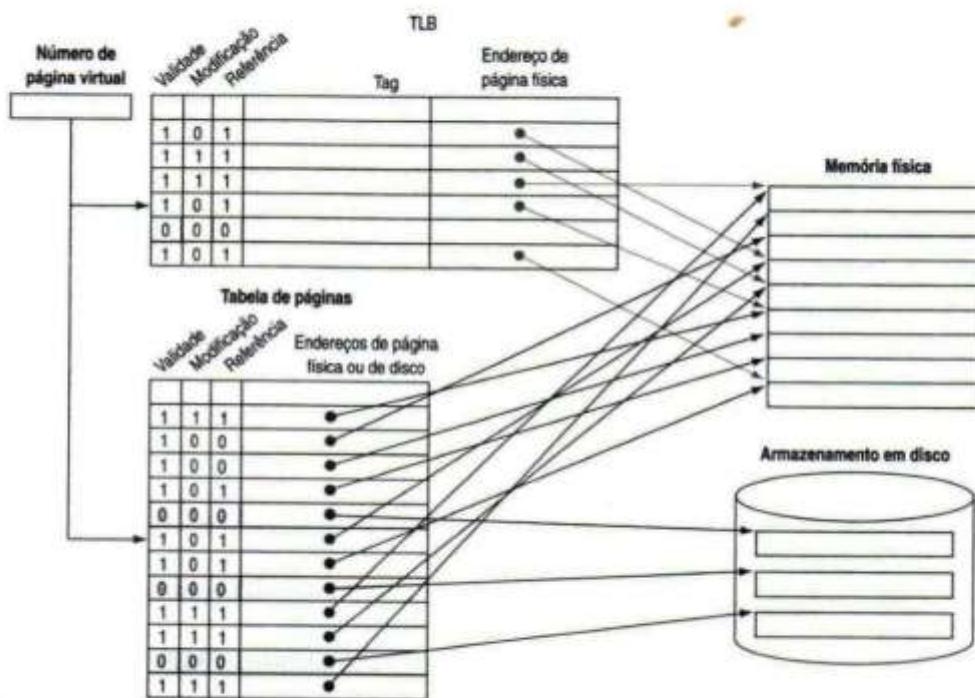


FIGURA 7.23 A TLB age como uma cache da tabela de páginas apenas para as entradas que mapeiam as páginas físicas. A TLB contém um subconjunto dos mapeamentos de página virtual para física que estão na tabela de páginas. Os mapeamentos da TLB são mostrados em destaque. Como a TLB é uma cache, ela precisa ter um campo tag. Se não houver uma entrada correspondente na TLB para uma página, a tabela de páginas precisa ser examinada. A tabela de páginas fornece um número de página física para a página (que pode, então, ser usado para construir uma entrada da TLB) ou indica que a página reside em disco, caso em que ocorre uma falta de página. Como a tabela de páginas possui uma entrada para cada página virtual, nenhum campo tag é necessário; ou seja, ela *não* é uma cache.

A Figura 7.23 mostra que cada entrada de tag na TLB contém uma parte do número de página virtual e cada entrada de dados da TLB contém um número de página física. Como não iremos mais acessar a tabela de páginas a cada referência, em vez disso acessaremos a TLB, que precisará incluir outros bits, como o bit de modificação e o bit de referência.

Em cada referência, consultamos o número de página virtual na TLB. Se tivermos um acerto, o número de página física é usado para formar o endereço e o bit de referência correspondente é ligado. Se o processador estiver realizando uma escrita, o bit de modificação também é ligado. Se ocorrer uma falha na TLB, precisamos determinar se ela é uma falta de página ou simplesmente uma falha de TLB. Se a página existir na memória, então a falha de TLB indica apenas que a tradução está faltando. Nesse caso, o processador pode tratar a falha de TLB lendo a tradução da tabela de páginas para a TLB e, depois, tentando a referência novamente. Se a página não estiver presente na memória, então a falha de TLB indica uma falta de página verdadeira. Nesse caso, o processador chama o sistema operacional usando uma exceção. Como a TLB possui muito menos entradas do que o número de páginas na memória principal, as falhas de TLB serão muito mais freqüentes do que as faltas de página verdadeiras.

As falhas de TLB podem ser tratadas no hardware ou no software. Na prática, com cuidado, pode haver pouca diferença de desempenho entre os dois métodos, uma vez que as operações básicas são iguais nos dois casos.

Depois que uma falha de TLB tiver ocorrido e a tradução faltando tiver sido recuperada da tabela de páginas, precisaremos selecionar uma entrada da TLB para substituir. Como os bits de referência e de modificação estão contidos na entrada da TLB, precisamos copiar esses bits de volta para a entrada da tabela de páginas quando substituirmos uma entrada. Esses bits são a única parte da entrada da TLB que pode ser modificada. O uso de write-back – ou seja, copiar de volta essas entradas no

momento da falha e não quando são escritas – é muito eficiente, já que esperamos que a taxa de falhas da TLB seja pequena. Alguns sistemas usam outras técnicas para aproximar os bits de referência e de modificação, eliminando a necessidade de escrever na TLB exceto para carregar uma nova entrada da tabela em uma falha.

Alguns valores comuns para uma TLB poderiam ser

- Tamanho da TLB: 16 a 512 entradas
- Tamanho do bloco: 1 a 2 entradas da tabela de páginas (geralmente 4 a 8 bytes cada uma)
- Tempo de acerto: 0,5 a 1 ciclo de clock
- Penalidade de falha: 10 a 100 ciclos de clock
- Taxa de falhas: 0,01% a 1%

Os projetistas têm usado uma ampla gama de associatividades em TLBs. Alguns sistemas usam TLBs pequenas e totalmente associativas porque um mapeamento totalmente associativo possui uma taxa de falhas mais baixa; além disso, como a TLB é pequena, o custo de um mapeamento totalmente associativo não é tão alto. Outros sistemas usam TLBs grandes, normalmente com pequena associatividade. Com um mapeamento totalmente associativo, escolher a entrada a ser substituída se torna difícil, pois é muito caro implementar um esquema de LRU de hardware. Além do mais, como as falhas de TLB são muito mais freqüentes do que as faltas de página e, portanto, precisam ser tratadas de modo mais econômico, podemos utilizar um algoritmo de software caro, como para as falhas. Como resultado, muitos sistemas fornecem algum suporte para escolher aleatoriamente uma entrada a ser substituída. Iremos examinar os esquemas de substituição mais detalhadamente na Seção 7.5.

A TLB do Intrinsity FastMATH

Para ver essas idéias em um processador real, vamos dar uma olhada mais de perto na TLB do Intrinsity FastMATH. O sistema de memória usa páginas de 4KB e um espaço de endereçamento de 32 bits; portanto, o número de página virtual tem 20 bits de largura, como no topo da Figura 7.24. O endereço físico é do mesmo tamanho do endereço virtual. A TLB contém 16 entradas, é totalmente associativa e é compartilhada entre as referências de instruções e de dados. Cada entrada possui 64 bits de largura e contém uma tag de 20 bits (que é o número de página virtual para essa entrada de TLB), o número de página física correspondente (também 20 bits), um bit de validade, um bit de modificação e outros bits de contabilidade.

A Figura 7.24 mostra a TLB e uma das caches, enquanto a Figura 7.25 mostra as etapas no processamento de uma requisição de leitura ou escrita. Quando ocorre uma falha de TLB, o hardware MIPS salva o número de página da referência em um registrador especial e gera uma exceção. A exceção chama o sistema operacional, que trata a falha no software. Para encontrar o endereço físico da página ausente, a rotina de falha de TLB indexa a tabela de páginas usando o número de página do endereço virtual e o registrador de tabela de páginas, que indica o endereço inicial da tabela de páginas do processo ativo. Usando um conjunto especial de instruções de sistema que podem atualizar a TLB, o sistema operacional coloca o endereço físico da tabela de páginas na TLB. Uma falha de TLB leva cerca de 13 ciclos de clock, considerando que o código e a entrada da tabela de páginas estejam na cache de instruções e na cache de dados, respectivamente. (Veremos o código TLB MIPS na página 403). Uma falta de página verdadeira ocorre se a entrada da tabela de páginas não possuir um endereço físico válido. O hardware mantém um índice que indica a entrada recomendada a ser substituída; a entrada recomendada é escolhida aleatoriamente.

Existe uma complicaçāo extra para requisições de escrita: o bit de acesso de escrita na TLB precisa ser verificado. Esse bit impede que o programa escreva em páginas para as quais tenha apenas acesso de leitura. Se o programa tentar uma escrita e o bit de acesso de escrita estiver desligado, uma exceção é gerada. O bit de acesso de escrita faz parte do mecanismo de proteção, que abordaremos em breve.

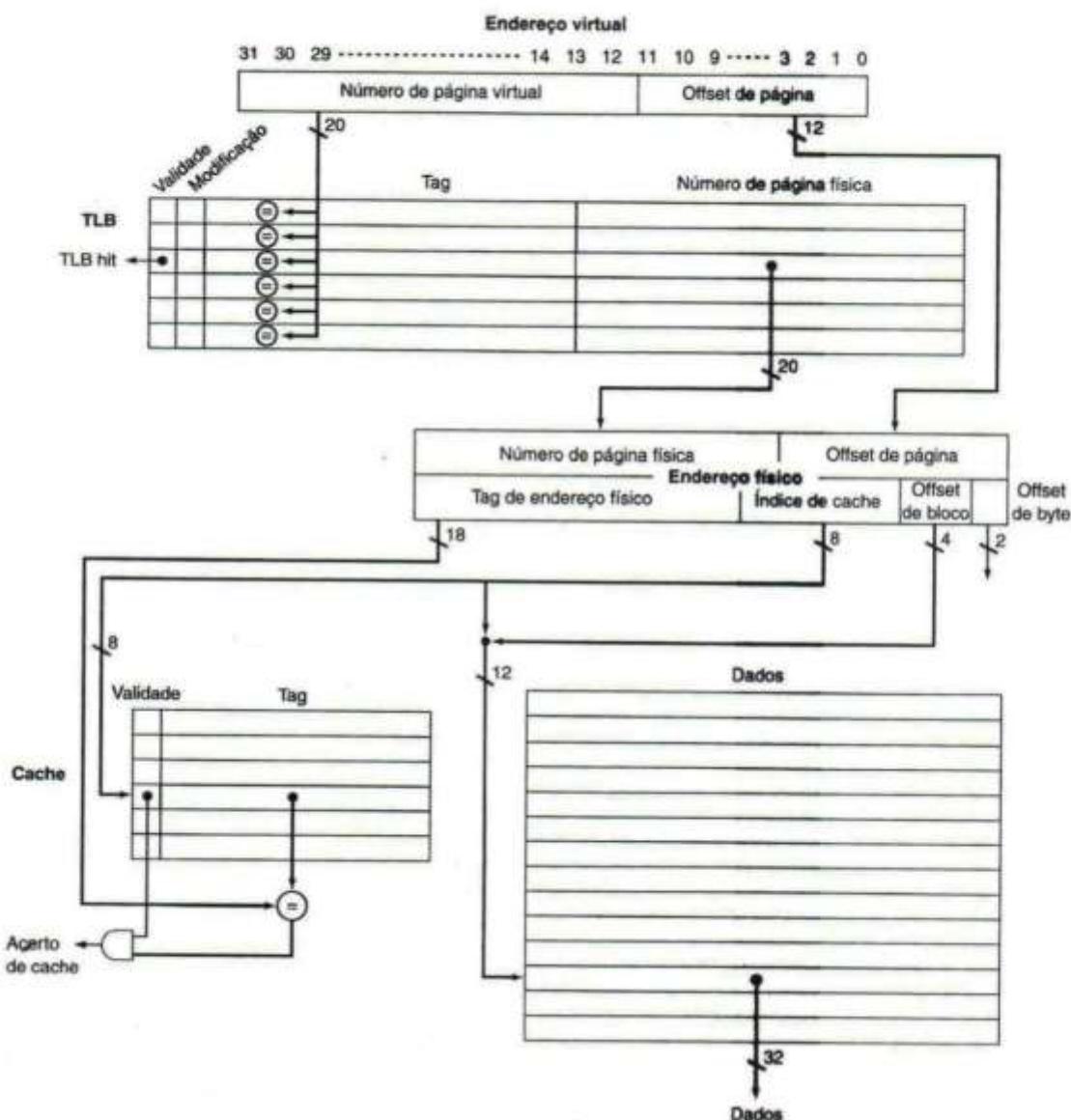


FIGURA 7.24 A TLB e a cache implementam o processo de ir de um endereço virtual para um item de dados no Intrinsity FastMATH. Essa figura mostra a organização da TLB e a cache de dados considerando um tamanho de página de 4KB. Este diagrama focaliza uma leitura; a Figura 7.25 descreve como tratar escritas. Repare que, diferente da Figura 7.9, as RAMs de tag e de dados são divididas. Endereçando a longa, mas estreita, RAM de dados com o índice de cache concatenado com o offset de bloco, selecionamos a word desejada no bloco sem um multiplexador 16:1. Embora a cache seja diretamente mapeada, a TLB é totalmente associativa. A implementação de uma TLB totalmente associativa exige que toda tag TLB seja comparada com o número de página virtual, já que a entrada desejada pode estar em qualquer lugar na TLB. Se o bit de validade da entrada correspondente estiver ligado, o acesso será um acerto de TLB e os bits do número de página física acrescidos aos bits do offset de página formarão o índice usado para acessar a cache. (O Intrinsity FastMATH na realidade possui um tamanho de página de 16KB; a seção “Detalhamento” na página 399 explica como ele funciona.)

Integrando memória virtual, TLBs e caches

Nossos sistemas de memória virtual e de cache funcionam em conjunto como uma hierarquia, de modo que os dados não podem estar na cache a menos que estejam presentes na memória principal. O sistema operacional desempenha um importante papel na manutenção dessa hierarquia removendo o conteúdo de qualquer página da cache quando decide migrar essa página para o disco. Ao mesmo tempo, o sistema operacional modifica as tabelas de páginas e a TLB de modo que uma tentativa de acessar quaisquer dados na página gere uma falta de página.

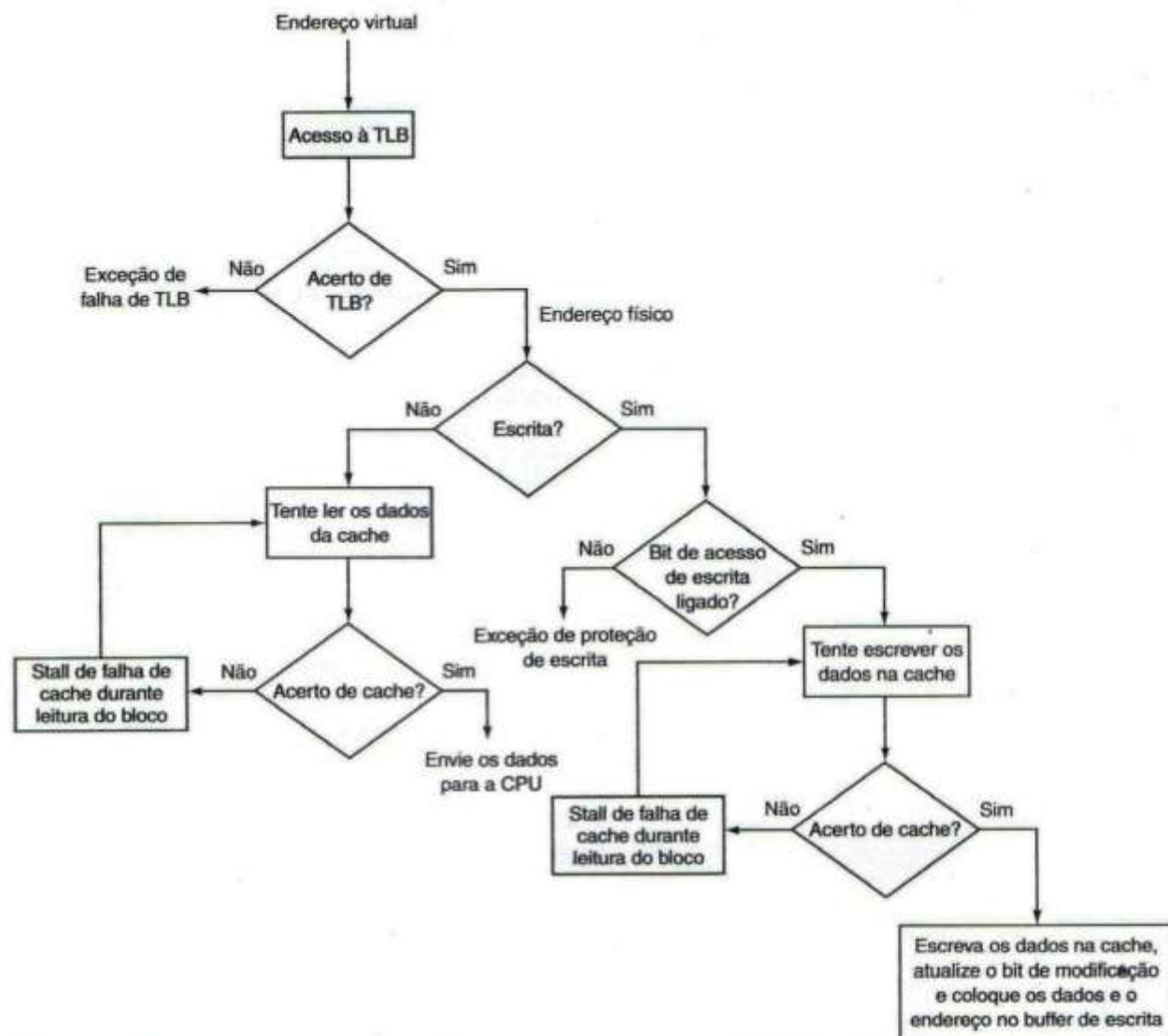


FIGURA 7.25 Processando uma leitura ou uma escrita direta na TLB e na cache do Intrinsity FastMATH. Se a TLB gerar um acerto, a cache pode ser acessada com o endereço físico resultante. Para uma leitura, a cache gera um acerto ou uma falha e fornece os dados ou causa um stall enquanto os dados são trazidos da memória. Se a operação for uma escrita, uma parte da entrada de cache é substituída por um acerto e os dados são enviados ao buffer de escrita se considerarmos uma cache write-through. Uma falha de escrita é exatamente como uma falha de leitura exceto que o bloco é modificado após ser lido da memória. Uma cache write-back requer que as escritas liguem um bit de modificação para o bloco de cache; além disso, um buffer de escrita é carregado com o bloco inteiro apenas em uma falha de leitura ou falha de escrita se o bloco a ser substituído estiver com o bit de modificação ligado. Observe que um acerto de TLB e um acerto de cache são eventos independentes, mas um acerto de cache só pode ocorrer após um acerto de TLB, o que significa que os dados precisam estar presentes na memória. A relação entre as falhas de TLB e as falhas de cache é examinada mais a fundo no exemplo a seguir e nos exercícios no final do capítulo.

Sob as circunstâncias ideais, um endereço virtual é traduzido pela TLB e enviado para a cache onde os dados apropriados são encontrados, recuperados e devolvidos ao processador. No pior caso, uma referência pode falhar em todos os três componentes da hierarquia de memória: a TLB, a tabela de páginas e a cache. O exemplo a seguir ilustra essas interações em mais detalhes.

OPERAÇÃO GERAL DE UMA HIERARQUIA DE MEMÓRIA

EXEMPLO

Em uma hierarquia de memória como a da Figura 7.24, que inclui uma TLB e uma cache organizada como mostrado, uma referência de memória pode encontrar três tipos de falhas diferentes: uma falha de TLB, uma falta de página e uma falha de cache. Considere todas as combinações desses três eventos com uma ou mais ocorrendo (sete possibilidades). Para cada possibilidade, diga se esse evento realmente pode ocorrer e sob que circunstâncias.

RESPOSTA

A Figura 7.26 mostra as circunstâncias possíveis e se elas podem ou não surgir na prática.

Detalhamento: a Figura 7.26 considera que todos os endereços de memória são traduzidos para endereços físicos antes que a cache seja acessada. Nessa organização, a cache é *fisicamente indexada e fisicamente rotulada* (tanto o índice quanto a tag de cache são endereços físicos em vez de virtuais). Nesse sistema, a quantidade de tempo para acessar a memória, considerando um acerto de cache, precisa acomodar um acesso de TLB e um acesso de cache; naturalmente, esses acessos podem ser em pipeline.

Como alternativa, o processador pode indexar a cache com um endereço que seja completa ou parcialmente virtual. Isso é chamado de **cache virtualmente endereçada** e usa tags que são endereços virtuais; portanto, esse tipo de cache é *virtualmente indexado e virtualmente rotulado*. Nessas caches, o hardware de tradução de endereço (TLB) não é usado durante o acesso de cache normal, já que a cache é acessada com um endereço virtual que não foi traduzido para um endereço físico. Isso tira a TLB do caminho crítico, reduzindo a latência da cache. Quando ocorre uma falha de cache, no entanto, o processador precisa traduzir o endereço para um endereço físico de modo que ele possa buscar o bloco de cache da memória principal.

Quando a cache é acessada com um endereço virtual e páginas são compartilhadas entre programas (que podem acessá-las com diferentes endereços virtuais), há a possibilidade de **aliasing**. O aliasing ocorre quando o mesmo objeto possui dois nomes – nesse caso, dois endereços virtuais para a mesma página. Essa ambiguidade cria um problema porque uma word nessa página pode ser colocada na cache em dois locais diferentes, cada um correspondendo a diferentes endereços virtuais. Essa ambiguidade permitiria que um programa escrevesse os dados sem que o outro programa soubesse que os dados foram mudados. As caches endereçadas completamente por endereços virtuais introduzem limitações de projeto na cache e na TLB para reduzir o aliasing ou exigem que o sistema operacional (e possivelmente o usuário) tome ações para garantir que o aliasing não ocorra.

Uma conciliação comum entre esses dois pontos de projeto são as caches virtualmente indexadas (algumas vezes, usando apenas a parte do offset de página do endereço, que é um endereço físico, já que não é traduzida), mas usam tags físicas. Esses projetos, que são *virtualmente indexados mas fisicamente rotulados*, tentam unir as vantagens de desempenho das caches virtualmente indexadas às vantagens da arquitetura mais simples de uma **cache fisicamente endereçada**. Por exemplo, não existe qualquer problema de aliasing nesse caso. A cache de dados L1 do Pentium 4 é um exemplo, como seria o Intrinsity FastMATH se o tamanho de página fosse de 4KB. Para realizar esse truque, precisa haver uma cuidadosa coordenação entre o tamanho de página mínimo, o tamanho da cache e a associatividade.

cache virtualmente endereçada Uma cache acessada com um endereço virtual em vez de um endereço físico.

aliasing Uma situação em que o mesmo objeto é acessado por dois endereços; pode ocorrer na memória virtual quando existem dois endereços virtuais para a mesma página física.

cache fisicamente endereçada Uma cache endereçada por um endereço físico.

TLeB	Tabela de páginas	Cache	Possível? Caso seja, sob que circunstâncias?
Acerto	Acerto	Falha	Possível, embora a tabela de páginas nunca seja realmente verificada se a TLB acertar.
Falha	Acerto	Acerto	TLB falha, mas entrada encontrada na tabela de páginas; após nova tentativa, os dados são encontrados na cache.
Falha	Acerto	Falha	TLB falha, mas entrada encontrada na tabela de páginas; após nova tentativa, os dados falham na cache.
Falha	Falha	Falha	TLB falha e é seguida de uma falta de página; após nova tentativa, os dados só podem falhar na cache.
Acerto	Falha	Falha	Impossível: não pode haver uma tradução na TLB se a página não estiver presente na memória.
Acerto	Falha	Acerto	Impossível: não pode haver uma tradução na TLB se a página não estiver presente na memória.
Falha	Falha	Acerto	Impossível: os dados não podem ser permitidos na cache se a página não estiver na memória.

FIGURA 7.26 As possíveis combinações de eventos na TLB, no sistema de memória virtual e na cache. Três dessas combinações são impossíveis e uma é possível (acerto de TLB, acerto de memória virtual, falta de cache) mas nunca detectada.

Detalhamento: a TLB do Intrinsity FastMATH é um pouco mais complicada do que mostra a Figura 7.24. O MIPS inclui dois mapeamentos de página física por número de página virtual, mapeando, assim, um par “par-impar” de números de página virtual para dois números de página física. Portanto, a tag é 1 bit mais estreita, já que cada entrada corresponde a duas páginas. Existem bits de contabilidade separados para cada página física. Essa otimização dobra a quantidade de memória mapeada por entrada de TLB. Como explica a seção “Detalhamento” da página a seguir, o campo tag realmente inclui um campo ID de espaço de endereço de 8 bits para reduzir o custo das trocas de contexto. Para suportar os tamanhos de página variáveis mencionados na página 405, há também um campo máscara de 32 bits que determina a linha divisoria entre o endereço de página virtual e o offset de página.

Implementando proteção com memória virtual

Uma das funções mais importantes da memória virtual é permitir o compartilhamento de uma única memória principal por diversos processos, enquanto fornece proteção de memória entre esses processos e o sistema operacional. O mecanismo de proteção precisa garantir que, embora vários processos estejam compartilhando a mesma memória principal, um processo rebelde não pode escrever no espaço de endereçamento de outro processo do usuário ou no sistema operacional, intencionalmente ou não. Por exemplo, se o programa que mantém as notas dos alunos de uma universidade estivesse sendo executado em um computador ao mesmo tempo em que programas dos alunos no primeiro curso de programação, não desejariam que o programa errante de um iniciante escrevesse sobre as notas de outro aluno. O bit de acesso de escrita na TLB pode proteger uma página de ser escrita. Sem esse nível de proteção, os vírus de computador seriam ainda mais comuns.

Interface hardware/software

Para permitir que o sistema operacional implemente proteção no sistema de memória virtual, o hardware precisa fornecer pelo menos três capacidades básicas resumidas a seguir.

1. Suportar pelo menos dois modos que indicam se o processo em execução é um processo de usuário ou um processo de sistema operacional, normalmente chamado de processo **supervisor**, processo de **kernel** ou processo **executivo**.
2. Fornecer uma parte do estado do processador que um processo de usuário pode ler mas não escrever. Isso inclui o bit de modo usuário/supervisor, que determina se o processador está no modo usuário ou supervisor, o ponteiro para a tabela de páginas e a TLB. Para escrever esses elementos, o sistema operacional usa instruções especiais que só estão disponíveis no modo supervisor.
3. Fornecer mecanismos pelos quais o processador pode passar do modo usuário para o modo supervisor e vice-versa. A primeira direção normalmente é conseguida por uma exceção de **chamada ao sistema**, implementada como uma instrução especial (*syscall* no conjunto de instruções MIPS) que transfere o controle para um local dedicado no espaço de código supervisor. Como em qualquer outra exceção, o contador de programa do ponto da chamada de sistema é salvo no PC de exceção (EPC), e o processador é colocado no modo supervisor. Para retornar ao modo usuário da exceção, use a instrução *return from exception* (ERET), que reinicializa para o modo usuário e desvia para o endereço no EPC.

Usando esses mecanismos e armazenando as tabelas de páginas no espaço de endereçamento do sistema operacional, o sistema operacional pode mudar as tabelas de páginas enquanto impede que um processo do usuário as modifique, garantindo que um processo do usuário só possa acessar o armazenamento fornecido pelo sistema operacional.

modo de kernel

Também chamado de **modo supervisor**. Um modo que indica que um processo executado é um processo do sistema operacional.

chamada ao sistema

Uma instrução especial que transfere o controle do modo usuário para um local dedicado no estágio de código supervisor, chamando o mecanismo de exceção no processo.

Também queremos evitar que um processo leia os dados de outro processo. Por exemplo, não desejamos que o programa de um aluno leia as notas enquanto elas estiverem na memória do processador. Uma vez que começamos a compartilhar a memória principal, precisamos fornecer a capacidade de um processo proteger seus dados de serem lidos e escritos por outro processo; caso contrário, a memória principal será um poço de permissividade!

Lembre-se de que cada processo possui seu próprio espaço de endereçamento virtual. Portanto, se o sistema operacional mantiver as tabelas de páginas organizadas de modo que as páginas virtuais independentes mapeiem as páginas físicas separadas, um processo não será capaz de acessar os dados de outro. É claro que isso exige que um processo de usuário seja incapaz de mudar o mapeamento da tabela de páginas. O sistema operacional pode garantir segurança se ele impedir que o processo do usuário modifique suas próprias tabelas de páginas. No entanto, o sistema operacional precisa ser capaz de modificar as tabelas de páginas. Colocar as tabelas de páginas no espaço de endereçamento protegido do sistema operacional satisfaz a ambos os requisitos.

Quando os processos querem compartilhar informações de uma maneira limitada, o sistema operacional precisa assisti-los, já que o acesso às informações de outro processo exige mudar a tabela de páginas do processo que está acessando. O bit de acesso de escrita pode ser usado para restringir o compartilhamento apenas à leitura e, como o restante da tabela de páginas, esse bit pode ser mudado apenas pelo sistema operacional. Para permitir que outro processo, digamos, P1, leia uma página pertencente ao processo P2, P2 pediria ao sistema operacional para criar uma entrada na tabela de páginas para uma página virtual no espaço de endereço de P1 que aponte para a mesma página física que P2 deseja compartilhar. O sistema operacional poderia usar o bit de proteção de escrita para impedir que P1 escrevesse os dados, se esse fosse o desejo de P2. Quaisquer bits que determinam os direitos de acesso para uma página precisam ser incluídos na tabela de páginas e na TLB, pois a tabela de páginas é acessada apenas em uma *falha* de TLB.

troca de contexto
Uma mudança no estado interno do processador para permitir que um processo diferente use o processador, o que inclui salvar o estado necessário para retornar ao processo sendo atualmente executado.

Detalhamento: quando o sistema operacional decide deixar de executar o processo P1 para executar o processo P2 (o que chamamos de **troca de contexto** ou **troca de processo**), ele precisa garantir que P2 não possa ter acesso às tabelas de páginas de P1 porque isso comprometeria a proteção. Se não houver uma TLB, basta mudar o registrador de tabela de páginas para que aponte para a tabela de páginas de P2 (em vez da de P1); com uma TLB, precisamos limpar as entradas de TLB que pertencem a P1 – tanto para proteger os dados de P1 quanto para forçar a TLB a carregar as entradas para P2. Se a taxa de troca de processos fosse alta, isso poderia ser bastante ineficiente. Por exemplo, P2 poderia carregar apenas algumas entradas de TLB antes que o sistema operacional trocasse novamente para P1. Infelizmente, P1, então, descobriria que todas as suas entradas de TLB desapareceram e precisaria pagar falhas de TLB para recarregá-las. Esse problema ocorre porque os endereços virtuais usados por P1 e P2 são iguais e precisamos limpar a TLB para evitar confundir esses endereços.

Uma alternativa comum é estender o espaço de endereçamento virtual acrescentando um *identificador de processo* ou *identificador de tarefa*. O Intrinsic FastMATH possui um campo ID do espaço de endereçamento (ASID) de 8 bits para essa finalidade. Esse pequeno campo identifica o processo que está atualmente sendo executado; ele é mantido em um registrador carregado pelo sistema operacional quando ele muda de processo. O identificador de processo é concatenado com a parte da tag da TLB, de modo que um acerto de TLB ocorra apenas se o número de página e o identificador de processo corresponderem. Essa combinação elimina a necessidade de limpar a TLB, exceto em raras ocasiões.

Problemas semelhantes podem ocorrer para uma cache, já que, em uma troca de processo, a cache conterá dados do processo em execução. Esses problemas surgem de diferentes maneiras para caches física e virtualmente endereçadas; além disso, uma variedade de soluções diferentes, como identificadores de processo, são usadas para garantir que um processo obtenha seus próprios dados.

Tratando falhas de TLB e faltas de página

Embora a tradução de endereços físicos para virtuais com uma TLB seja simples quando temos um acerto de TLB, o tratamento de falhas de TLB e de faltas de página é mais complexo. Uma falha de TLB ocorre quando nenhuma entrada na TLB corresponde a um endereço virtual. Uma falha de TLB pode indicar uma de duas possibilidades:

1. A página está presente na memória e precisamos apenas criar a entrada de TLB ausente.
2. A página não está presente na memória e precisamos transferir o controle para o sistema operacional para lidar com uma falta de página.

Como saber qual dessas duas circunstâncias ocorreu? Quando processarmos a falha de TLB, iremos procurar uma entrada na tabela de páginas para ser trazida para a TLB. Se a entrada na tabela de páginas correspondente tiver um bit de validade que esteja desligado, a página correspondente não está na memória e temos uma falta de página em vez de uma simples falha de TLB. Se o bit de validade estiver ligado, podemos simplesmente recuperar a entrada desejada.

Uma falha de TLB pode ser tratada por software ou por hardware, pois ela exigirá apenas uma curta sequência de operações para copiar uma entrada válida da tabela de páginas da memória para a TLB. O MIPS tradicionalmente trata uma falha de TLB por software. Ele traz a entrada da tabela de páginas da memória e, depois, executa novamente a instrução que causou a falha de TLB. Na reexecução, ele terá um acerto de TLB. Se a entrada da tabela de páginas indicar que a página não está na memória, dessa vez ele terá uma exceção de falta de página.

Tratar uma falha de TLB ou uma falta de página requer o uso do mecanismo de exceção para interromper o processo ativo, transferir o controle para o sistema operacional e, depois, retomar a execução do processo interrompido. Uma falta de página será reconhecida em algum momento durante o ciclo de clock usado para acessar a memória. Para reiniciar a instrução após a falta de página ser tratada, o contador de programa da instrução que causou a falta de página precisa ser salvo. Assim como nos Capítulos 5 e 6, o contador de programa de exceção (EPC) é usado para conter esse valor.

Além disso, uma falha de TLB ou uma exceção de falta de página precisa ser sinalizada no final do mesmo ciclo de clock em que ocorre o acesso à memória, de modo que o próximo ciclo de clock começará o processamento da exceção em vez de continuar a execução normal das instruções. Se a falta de página não fosse reconhecida nesse ciclo de clock, uma instrução load poderia substituir um registrador, e isso poderia ser desastroso quando tentássemos reiniciar a instrução. Por exemplo, considere a instrução `lw $1,0($1)`: o computador precisa ser capaz de impedir que o estágio de escrita do resultado do pipeline ocorra; caso contrário, ele não poderia reiniciar corretamente a instrução, já que o conteúdo de `$1` teria sido destruído. Uma complicação parecida surge nos stores. Precisamos impedir que a escrita na memória realmente seja concluída quando há uma falta de página; isso normalmente é feito desativando a linha de controle de escrita para a memória.

Interface hardware/software

Entre o momento em que começamos a executar o tratamento de exceção no sistema operacional e o momento em que o sistema operacional salvou todo o estado do processo, o sistema operacional se torna particularmente vulnerável. Por exemplo, se outra exceção ocorresse quando estivéssemos processando a primeira exceção no sistema operacional, a unidade de controle substituiria o contador de programa de exceção, tornando impossível voltar para a instrução que causou a falta de página! Podemos evitar esse desastre fornecendo a capacidade de inabilitar e **habilitar exceções**. Assim que uma exceção ocorre, o processador liga um bit que desativa todas as outras exceções; isso poderia acontecer ao mesmo tempo em que o processador liga o bit de modo supervisor. O sistema operacional, então, salva o estado apenas suficiente para lhe permitir se recuperar se outra exceção ocorrer – a saber, o contador de programa de exceção e o registrador Cause. EPC e Cause são dois dos registradores de controle especiais que ajudam com exceções, falhas de TLB e faltas de página; a Figura 7.27 mostra o restante. O sistema operacional, então, pode habilitar novamente as exceções. Essas etapas asseguram que as exceções não façam com que o processador perca qualquer estado e, portanto, sejam incapazes de reiniciar a execução da instrução interruptora.

habilitar exceção

Também chamado de "habilitar interrupção". Uma ação ou sinal que controla se o processo responde ou não a uma exceção; necessário para evitar a ocorrência de exceções durante intervalos antes que o processador tenha seguramente salvado o estado necessário para a reinicialização.

Registrador	Número do registrador CPO	Descrição
EPC	14	Onde reiniciar após exceção
Cause	13	Causa da exceção
BadVAddr	8	Endereço que causou exceção
Index	0	Local no TLB a ser lido ou escrito
Random	1	Local pseudo-aleatório no TLB
EntryLo	2	Endereço da página física e flags
EntryHi	10	Endereço da página virtual
Context	4	Endereço da tabela de página e número de página

Figura 7.27 Registradores de controle MIPS. Considera-se que estes estejam no co-processador O, e por isso são lidas com mfc0 e escritos com mtc0.

Uma vez que o sistema operacional conhece o endereço virtual que causou a falta de página, ele precisa completar três etapas:

1. Consultar a entrada de tabela de páginas usando o endereço virtual e encontrar o local em disco da página referenciada.
2. Escolher uma página física a ser substituída; se a página escolhida estiver com o bit de modificação ligado, ela precisará ser escrita no disco antes que possamos definir uma nova página virtual para essa página física.
3. Iniciar uma leitura para trazer a página referenciada do disco para a página física escolhida.

É claro que essa última etapa levará milhões de ciclos de clock de processador (assim como a segunda, se a página substituída estiver com o bit de modificação ligado); portanto, o sistema operacional normalmente selecionará outro processo para executar no processador até que o acesso ao disco seja concluído. Como o sistema operacional salvou o estado do processo, ele pode passar o controle do processador à vontade para outro processo.

Quando a leitura da página do disco está completa, o sistema operacional pode restaurar o estado do processo que causou originalmente a falta de página e executar a instrução que retorna da exceção. Essa instrução irá redefinir o processador do modo kernel para o modo usuário, bem como restaurar o contador de programa. O processo do usuário, então, reexecuta a instrução que causou a falta de página, acessa a página requisitada com sucesso e continua a execução.

As exceções de falta de página para acessos a dados são difíceis de implementar corretamente em um processador devido a uma combinação de três características:

1. Elas ocorrem no meio das instruções, diferente das faltas de página de instruções.
2. A instrução não pode ser completada antes que a exceção seja tratada.
3. Após tratar a exceção, a instrução precisa ser reinicializada como se nada tivesse ocorrido.

Instrução reinicializável
Uma instrução que pode retomar a execução após uma exceção ser resolvida sem que a exceção afete o resultado da instrução.

Tornar **instruções reinicializáveis**, de modo que a exceção possa ser tratada e a instrução possa ser continuada, é relativamente fácil em uma arquitetura como o MIPS. Como cada instrução escreve apenas um item de dados e essa escrita ocorre no final do ciclo da instrução, podemos simplesmente impedir que a instrução seja concluída (não escrevendo) e reinicializar a instrução no começo.

Para processadores com instruções muito mais complexas que podem tocar muitos locais da memória e escrever muitos itens de dados, é muito mais difícil tornar as instruções reinicializáveis. O processamento de uma instrução pode gerar diversas faltas de página no meio da instrução. Por exemplo, alguns processadores possuem instruções de movimento de blocos que tocam milhares de words de dados. Nesses processadores, as instruções normalmente não podem ser reinicializadas do início, como fazemos para instruções MIPS. Em vez disso, a instrução precisa ser interrompida e depois continuada em sua execução. Retomar uma instrução no meio de sua execução em geral requer salvar algum estado especial, processar a exceção e restaurar esse estado especial. Fazer esse tra-

lho corretamente exige uma coordenação cuidadosa e detalhada entre o código de tratamento de exceção no sistema operacional e no hardware.

Vejamos o MIPS mais de perto. Quando uma falha de TLB ocorre, o hardware do MIPS salva o número de página da referência em um registrador especial chamado `BadVAddr` e gera uma exceção.

A exceção chama o sistema operacional, que trata a falha por software. O controle é transferido para o endereço 8000 0000_{hex} (o local do **handler** da falha de TLB). Para encontrar o endereço físico para a página ausente, a rotina de falha de TLB indexa a tabela de páginas usando o número de página do endereço virtual e o registrador de tabela de páginas, que indica o endereço inicial da tabela de páginas do processo ativo. Para tornar essa indexação rápida, o hardware do MIPS coloca tudo que você precisa no registrador especial `Context`: os 12 bits mais significativos têm o endereço da base da tabela de páginas e os próximos 18 bits têm o endereço virtual da página ausente. Como cada entrada de tabela de páginas possui uma word, os últimos 2 bits são 0. Portanto, as duas primeiras instruções copiam o registrador `Context` para o registrador temporário do kernel `$k1` e, depois, carregam a entrada de tabela de páginas desse endereço em `$k1`. Lembre-se de que `$k0` e `$k1` são reservados para uso do sistema operacional sem salvamento; um importante motivo para essa convenção é tornar rápido o handler de falha de TLB. A seguir está o código MIPS para um handler de falha de TLB típico:

```
FalhaDeTLB:  
    mfc0 $k1,Context      # copia o endereço de PTE para temp $k1  
    lw $k1, 0($k1)        # coloca PTE em temp $k1  
    mtc0 $k1,EntryLo     # coloca PTE no registrador especial EntryLo  
    tlbwr                # coloca EntryLo na entrada de TLB em Random  
    eret                 # retorna da exceção de falha de TLB
```

Como mostrado anteriormente, o MIPS possui um conjunto especial de instruções de sistema para atualizar a TLB. A instrução `tlbwr` copia o registrador de controle `EntryLo` para a entrada de TLB selecionada pelo registrador de controle `Random`. `Random` implementa uma substituição aleatória e, portanto, é basicamente um contador de execução livre. Uma falha de TLB leva cerca de 12 ciclos de clock.

Observe que o handler de falha de TLB não verifica se a entrada de tabela de páginas é válida. Como a exceção para a entrada de TLB ausente é muito mais frequente do que uma falta de página, o sistema operacional carrega a TLB da tabela de páginas sem examinar a entrada e reinicializa a instrução. Se a entrada for inválida, ocorre outra exceção diferente e o sistema operacional reconhece a falta de página. Esse método torna rápido o caso frequente de uma falha de TLB, com uma pequena penalidade de desempenho para o raro caso de uma falta de página.

Uma vez que o processo que gerou a falta de página tenha sido interrompido, ele transfere o controle para 8000 0180_{hex}, um endereço diferente do handler de falha de TLB. Esse é o endereço geral para exceção; a falha de TLB possui um ponto de entrada especial para reduzir a penalidade para uma falha de TLB. O sistema operacional usa o registrador `Cause` de exceção para diagnosticar a causa da exceção. Como a exceção é uma falta de página, o sistema operacional sabe que será necessário um processamento extenso. Portanto, diferente de uma falha de TLB, ele salva todo o estado do processo ativo. Esse estado inclui todos os registradores de uso geral e de ponto flutuante, o registrador de endereço de tabela de páginas, o EPC e o registrador `Cause` de exceção. Como os handlers de exceção normalmente não usam os registradores de ponto flutuante, o ponto de entrada geral não os salva, deixando isso para os poucos handlers que precisam deles.

A Figura 7.28 esboça o código MIPS de um handler de exceção. Note que salvamos e restauramos o estado no código MIPS, tomando cuidado quando habilitamos e inabilitamos exceções, mas chamamos código C para tratar a exceção em particular.

O endereço virtual que causou a falta de página depende se essa foi uma falta de instruções ou de dados. O endereço da instrução que gerou a falta está no EPC. Se ela fosse uma falta de página de instruções, o EPC contém o endereço virtual da página que gerou a falta; caso contrário, o endereço virtual que gerou a falta pode ser calculado examinando a instrução (cujo endereço está no EPC) para encontrar o registrador base e o campo offset.

handler Nome de uma rotina de software chamada para "tratar" uma exceção ou interrupção.

Salva estado			
Salva GPR	addi \$k1,\$sp, -XCP SIZE sw \$sp, XCT_SP(\$k1) sw \$v0, XCT_VO(\$k1) ... sw \$ra, XCT_RA(\$k1)	# reserva espaço na pilha para o estado # salva \$sp na pilha # salva \$v0 na pilha # salva \$v1, \$a1, \$s1, \$t1, ...na pilha # salva \$ra na pilha	
Salva Hi, Lo	mfhi \$v0 mflo \$v1 sw \$v0, XCT_HI(\$k1) sw \$v1, XCT_LI(\$k1)	# copia Hi # copia Lo # salva valor de Hi na pilha # salva valor de Lo na pilha	
Salva registradores de exceção	mfc0 \$a0, \$cr sw \$a0, XCT_CR(\$k1) mfc0 \$a3, \$sr sw \$a3, XCT_SR(\$k1)	# copia registrador Cause # salva valor de \$cr na pilha # copia registrador de status # salva \$sr na pilha	
Atribui novo valor a sp	move \$sp, \$k1	# sp = sp - XCP SIZE	
Habilita exceções aninhadas			
	andi \$v0, \$a3, MASK1 mtc0 \$v0, \$sr	# \$v0 = \$sr & MASK1, habilita exceções # \$sr = valor que habilita exceções	
Chama handler de exceção em C			
Atribui novo valor a \$gp	move \$gp, GPINIT	# \$gp aponta para área do heap	
Chama código em C	move \$a0, \$sp jal xcpt_deliver	# arg1 = ponteiro para pilha de exceção # chama código em C para tratar exceção	
Restaura estado			
Restaura a maioria dos registradores, Hi, Lo	lw \$at, \$sp lw \$ra, XCT_RA(\$at) ... lw \$a0, XCT_A0(\$k1)	# valor temporário de \$sp # restaura \$ra da pilha # restaura \$t0, ..., \$a1 # restaura \$a0 da pilha	
Restaura registrador de status	lw \$v0, XCT_SR(\$at) li \$v1, MASK2 and \$v0, \$v0, \$v1 mtc0 \$v0, \$sr	# carrega \$sr antigo da pilha # máscara para inabilitar exceções # \$v0 = \$sr & MASK2, inabilita exceções # restaura o valor do registrador de status	
Retorna da exceção			
Restaura \$sp e o restante dos registradores usados como registradores temporários	lw \$sp, XCT_SP(\$at) lw \$v0, XCT_VO(\$at) lw \$v1, XCT_V1(\$at) lw \$k1, XCT_EPC(\$at) lw \$at, XCT_AT(\$at)	# restaura \$sp da pilha # restaura \$v0 da pilha # restaura \$v1 da pilha # copia \$epc antigo da pilha # restaura \$at da pilha	
Restaura EPC e retorna	mtc0 \$k1, \$epc eret \$ra	# restaura \$epc # retorna para a instrução interrompida	

FIGURA 7.28 Código MIPS para salvar e restaurar o estado em uma exceção.

não mapeada Uma parte do espaço de endereçamento que não pode ter faltas de página.

Detalhamento: essa versão simplificada considera que o stack pointer (sp) é válido. Para evitar o problema de uma falta de página durante esse código de exceção de baixo nível, o MIPS separa uma parte do seu espaço de endereçamento que não pode ter faltas de página, chamada **não mapeada** (unmapped). O sistema operacional insere o código para o ponto de entrada do tratamento de exceções e a pilha de exceção na memória não mapeada. O hardware MIPS traduz os endereços virtuais 8000 0000_{hex} a BFFF FFFF_{hex} para endereços físicos simplesmente ignorando os bits superiores do endereço virtual, colocando, assim, esses endereços na parte inferior da memória física. Portanto, o sistema operacional coloca os pontos de entrada dos tratamentos de exceções e as pilhas de exceção na memória não mapeada.

Detalhamento: o código na Figura 7.28 mostra a seqüência de retorno de exceção do MIPS-32. O MIPS-I usa rfe e jr em vez de eret.

Resumo

Memória virtual é o nome para o nível da hierarquia de memória que controla a cache entre a memória principal e o disco. A memória virtual permite que um único programa expanda seu espaço de endereçamento para além dos limites da memória principal. Mais importante, nos sistemas computacionais recentes, a memória virtual suporta o compartilhamento da memória principal entre vários processos simultaneamente ativos, que, juntos, exigem muito mais memória principal física do que a disponível. Para suportar o compartilhamento, a memória virtual também fornece mecanismos para proteção de memória.

Gerenciar a hierarquia de memória entre a memória principal e o disco é uma tarefa difícil devido ao alto custo das faltas de página. Várias técnicas são usadas para reduzir a taxa de falhas:

1. Os blocos, chamados de páginas, são ampliados para tirar proveito da localidade espacial e para reduzir a taxa de falhas.
2. O mapeamento entre endereços virtuais e endereços físicos, que é implementado com uma tabela de páginas, é feito totalmente associativo para que uma página virtual possa ser colocada em qualquer lugar na memória principal.
3. O sistema operacional usa técnicas, como LRU e um bit de referência, para escolher que páginas substituir.

Como as gravações no disco são caras, a memória virtual usa um esquema write-back e também monitora se uma página foi modificada (usando um bit de modificação) para evitar gravar páginas não alteradas novamente no disco.

O mecanismo de memória virtual fornece tradução de endereços de um endereço virtual usado pelo programa para o espaço de endereçamento físico usado para acessar a memória. Essa tradução de endereços permite compartilhamento protegido da memória principal e oferece várias vantagens adicionais, como a simplificação da alocação de memória. Para garantir que os processos sejam protegidos uns dos outros, é necessário que apenas o sistema operacional possa mudar as traduções de endereços, o que é implementado impedindo que programas de usuário alterem as tabelas de páginas. O compartilhamento controlado das páginas entre processos pode ser implementado com a ajuda do sistema operacional e dos bits de acesso na tabela de páginas que indicam se o programa do usuário possui acesso de leitura ou escrita à página.

Se um processador precisasse acessar uma tabela de páginas residente na memória para traduzir cada acesso, a memória virtual teria muito overhead e a cache não teria sentido! Em vez disso, uma TLB age como uma cache para traduções da tabela de páginas. Os endereços são, então, traduzidos do virtual para o físico usando as traduções na TLB.

As caches, a memória virtual e as TLBs se baseiam em um conjunto comum de princípios e políticas. A próxima seção aborda essa estrutura comum.

Entendendo o desempenho dos programas

Embora a memória virtual tenha sido criada para permitir que uma memória pequena aja como uma grande, a diferença de desempenho entre o disco e a memória significa que se um programa acessa rotineiramente mais memória virtual do que a memória física que possui, sua execução será muito lenta. Esse programa estaria continuamente trocando páginas entre a memória e o disco, o que chamamos de *thrashing*. O thrashing, embora raro, é um desastre quando ocorre. Se seu programa realiza thrashing, a solução mais fácil é executá-lo em um computador com mais memória ou comprar mais memória para o computador. Uma opção mais complexa é reexaminar suas estruturas de dados e algoritmo para ver se você pode mudar a localidade e, portanto, reduzir o número de páginas que seu programa usa simultaneamente. Esse conjunto de páginas é informalmente chamado de *working set*.

Um problema de desempenho mais comum são as falhas de TLB. Como uma TLB pode tratar apenas de 32 a 64 entradas de página ao mesmo tempo, um programa poderia facilmente ver uma alta

taxa de falhas de TLB, já que o processador pode acessar menos de um quarto de megabyte diretamente: $64 \times 4\text{KB} = 0.25\text{MB}$. Por exemplo, as falhas de TLB normalmente são um problema para o Radix Sort. Para tentar amenizar esse problema, a maioria das arquiteturas de computadores agora suporta tamanhos de página variáveis. Por exemplo, além da página de 4KB padrão, o hardware do MIPS suporta páginas de 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, 64MB e 256MB. Consequentemente, se um programa usa grandes tamanhos de página, ele pode acessar mais memória diretamente sem falhas de TLB.

Na prática, o problema é fazer o sistema operacional permitir que os programas selecionem esses tamanhos de página maiores. Mais uma vez, a solução mais complexa para reduzir as falhas de TLB é reexaminar as estruturas de dados e os algoritmos para reduzir o working set de páginas.

Verifique você mesmo

Associe o elemento da hierarquia de memória à esquerda com a frase correspondente à direita.

- | | |
|----------------------|---|
| 1. Cache L1 | a. Uma cache para uma cache |
| 2. Cache L2 | b. Uma cache para discos |
| 3. Memória principal | c. Uma cache para uma memória principal |
| 4. TLB | d. Uma cache para entradas de tabela de páginas |

7.5

Uma estrutura comum para hierarquias de memória

Agora você reconhece que os diferentes tipos de hierarquias de memória compartilham muita coisa em comum. Embora muitos aspectos das hierarquias de memória difiram quantitativamente, muitas das políticas e recursos que determinam como uma hierarquia funciona são semelhantes em qualidade. A Figura 7.29 mostra como algumas características quantitativas das hierarquias de memória podem diferir. No restante desta seção, discutiremos os aspectos operacionais comuns das hierarquias de memória e como determinar seu comportamento. Examinaremos essas políticas como uma série de questões que se aplicam entre quaisquer dos níveis de uma hierarquia de memória, embora usemos principalmente terminologia de caches por motivo de simplicidade.

Questão 1: onde um bloco pode ser colocado?

Vimos que o posicionamento de bloco no nível superior da hierarquia pode usar uma gama de esquemas, do diretamente mapeado ao associativo por conjunto e ao totalmente associativo. Como mencionamos anteriormente, toda essa faixa de esquemas pode ser imaginada como variações em um esquema associativo por conjunto no qual o número de conjuntos e o número de blocos por conjunto variam.

Nome do esquema	Número de conjuntos	Blocos por conjunto
Diretamente mapeado	Número de blocos na cache	1
Associativo por conjunto	Número de blocos na cache Associatividade	Associatividade (normalmente 2 a 16)
Totalmente associativo	1	Número de blocos na cache

A vantagem de aumentar o grau de associatividade é que normalmente isso diminui a taxa de falhas. A melhoria da taxa de falhas deriva da redução das falhas que disputam o mesmo local. Examinaremos essas falhas mais detalhadamente em breve. Antes, vejamos quanta melhoria é obtida. A Figura 7.30 mostra os dados para um workload consistindo nos benchmarks SPEC2000 com caches de

4KB a 512KB, variando de diretamente mapeado a associativo por conjunto de oito vias. Os maiores ganhos são obtidos com a passagem do mapeamento direto para a associatividade por conjunto de duas vias, o que produz uma redução de 20% a 30% na taxa de falhas. Conforme crescem os tamanhos de cache, a melhoria relativa da associatividade aumenta apenas ligeiramente; como a perda geral de uma cache maior é menor, a oportunidade de melhorar a taxa de falhas diminui e a melhoria absoluta na taxa de falhas da associatividade é reduzida significativamente. As possíveis desvantagens da associatividade, como já mencionado, são o custo mais alto e o tempo de acesso mais longo.

Recurso	Valores típicos para caches L1	Valores típicos para caches L2	Valores típicos para memória paginada	Valores típicos para um TLB
Tamanho total em blocos	250-2000	4000-250.000	16.000-250.000	16-512
Tamanho total em kilobytes	16-64	500-8000	250.000-1.000.000.000	0,25-16
Tamanho do bloco em bytes	32-64	32-128	4000-64.000	4-32
Penalidade da perda em clocks	10-25	100-1000	10.000.000-100.000.000	10-1000
Taxas de perda (global para L2)	2%-5%	0,1%-2%	0,00001%-0,0001%	0,01%-2%

FIGURA 7.29 Os principais parâmetros quantitativos do projeto que caracterizam os principais elementos da hierarquia de memória em um computador. Estes são valores típicos para esses níveis em 2004. Embora o intervalo de valores seja maior, isso é parcialmente porque muitos dos valores que mudaram com o tempo estão relacionados; por exemplo, à medida que os caches se tornam maiores para contornar maiores penalidades de perda, os tamanhos de bloco também crescem.

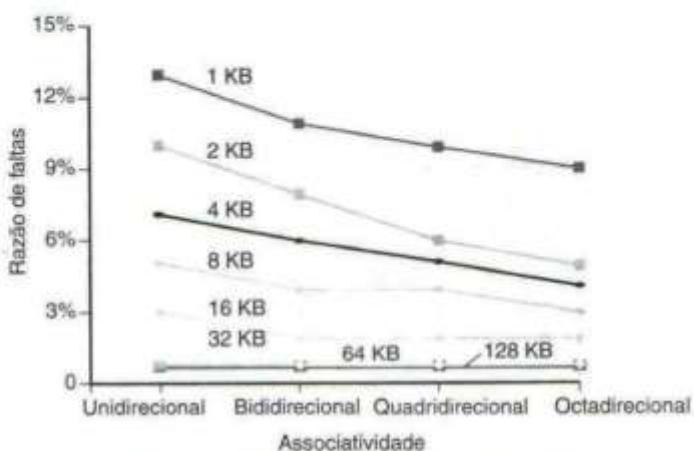


FIGURA 7.30 As razões de falhas da cache de dados para cada um dos oito tamanhos melhora à medida que a associatividade aumenta Embora o benefício de passar de associação por conjunto de uma via (mapeamento direto) para de 2 vias seja significativo, os benefícios de maior associatividade são menores (por exemplo, 1%-10% passando de 2 vias para 4 vias contra 20%-30% de melhoria passando de 1 via para 2 vias). Há ainda menos melhoria ao passar de 4 vias para 8 vias, que, por sua vez, é muito próximo das taxas de falhas de uma cache totalmente associativa. As caches menores obtêm um benefício absoluto muito maior com a associatividade, pois a taxa de falhas básica de uma cache pequena é maior. A Figura 7.15 explica como esses dados foram coletados.

Questão 2: como um bloco é encontrado?

A escolha de como localizamos um bloco depende do esquema de posicionamento do bloco, já que isso determina o número de locais possíveis. Poderíamos resumir os esquemas da seguinte maneira:

Associatividade	Método de localização	Comparações necessárias
Diretamente mapeado	Indexação	1
Associativo por conjunto	Indexação do conjunto, pesquisa entre os elementos	Grau de associatividade
Total	Pesquisa de todas as entradas de cache	Tamanho da cache
	Tabela de consulta separada	0

A escolha entre os métodos diretamente mapeado, associativo por conjunto ou totalmente associativo em qualquer hierarquia de memória dependerá do custo de uma falha comparado com o custo de implementar a associatividade, ambos em termos de tempo e de hardware extra. Incluir a cache L2 no chip permite uma associatividade muito mais alta, pois os tempos de acerto não são tão importantes e o projetista não precisa se basear nos chips SRAM padrão como blocos de construção. As caches totalmente associativas são proibitivas exceto para pequenos tamanhos, nos quais o custo dos comparadores não é grande e as melhorias da taxa de falhas absoluta são as melhores.

Nos sistemas de memória virtual, uma tabela de mapeamento separada (a tabela de páginas) é mantida para indexar a memória. Além do armazenamento necessário para a tabela, usar um índice exige um acesso extra à memória. A escolha da associatividade total para o posicionamento de página e da tabela extra é motivada por quatro fatos:

1. A associatividade total é benéfica, já que as falhas são muito caras.
2. A associatividade total permite que softwares usem esquemas sofisticados de substituição projetados para reduzir a taxa de falhas.
3. O mapa completo pode ser facilmente indexado sem a necessidade de pesquisa e de qualquer hardware extra.
4. O tamanho de página maior significa que o overhead do tamanho da tabela de páginas é relativamente pequeno. (O uso de uma tabela de consulta separada, como uma tabela de páginas para a memória virtual, não é viável para uma cache, pois a tabela seria muito maior do que uma tabela de páginas, que pode não ser acessada rapidamente.)

Portanto, os sistemas de memória virtual quase sempre usam posicionamento totalmente associativo.

O posicionamento totalmente associativo é muitas vezes usado para caches e TLBs, no qual o acesso combina indexação e a pesquisa de um conjunto pequeno. Alguns sistemas têm usado caches diretamente mapeadas devido às suas vantagens no tempo de acesso e da simplicidade. A vantagem no tempo de acesso ocorre porque a localização do bloco requisitado não depende de uma comparação. Essas escolhas de projeto dependem de muitos detalhes da implementação, como se a cache é on-chip, a tecnologia usada para implementar a cache e o papel vital do tempo de acesso na determinação do tempo de ciclo do processador.

Questão 3: que bloco deve ser substituído em uma falha de cache?

Quando uma falha ocorre em uma cache associativa, precisamos decidir que bloco substituir. Em uma cache totalmente associativa, todos os blocos são candidatos à substituição. Se a cache for associativa por conjunto, precisamos escolher entre os blocos do conjunto. É claro que a substituição é fácil em uma cache diretamente mapeada porque existe apenas um candidato.

Já mencionamos as duas principais estratégias para substituição nas caches associativas por conjunto ou totalmente associativas:

- *Substituição aleatória*: os blocos candidatos são selecionados aleatoriamente, talvez usando alguma assistência do hardware. Por exemplo, o MIPS suporta substituição aleatória para falhas de TLB.
- *Substituição LRU (Least Recently Used)*: o bloco substituído é o que não foi usado há mais tempo.

Na prática, o LRU é muito oneroso de ser implementado para hierarquias com mais do que um pequeno grau de associatividade (geralmente, dois a quatro), já que é oneroso controlar o uso das informações. Mesmo para a associatividade por conjunto de quatro vias, o LRU normalmente é aproximado – por exemplo, monitorando qual par de blocos é o LRU (o que requer 1 bit) e, depois, monitorando que bloco em cada par é o LRU (o que requer 1 bit por par).

Para maior associatividade, ou o LRU é aproximado ou a substituição aleatória é usada. Nas caches, o algoritmo de substituição está no hardware, o que significa que o esquema deve ser fácil de implementar. A substituição aleatória é simples de construir em hardware e, para uma cache associativa por conjunto de duas vias, a substituição aleatória possui uma taxa de falhas cerca de 1,1 vez mais alta do que a substituição LRU. Conforme as caches se tornam maiores, a taxa de falhas para as duas estratégias de substituição cai e a diferença absoluta se torna pequena. Na verdade, a substituição aleatória, algumas vezes, pode ser melhor do que as aproximações simples de LRU que são facilmente implementadas em hardware.

Na memória virtual, alguma forma de LRU é sempre aproximada, já que mesmo uma pequena redução na taxa de falhas pode ser importante quando o custo de uma falha é enorme. Os bits de referência ou funcionalidade equivalente costumam ser fornecidos para facilitar que o sistema operacional monitore um conjunto de páginas usadas menos recentemente. Como as falhas são muito caras e relativamente raras, é aceitável aproximar essa informação, em especial, em nível de software.

Questão 4: o que acontece em uma escrita?

Uma importante característica de qualquer hierarquia de memória é como ela lida com as escritas. Já vimos as duas opções básicas:

- *Write-through*: as informações são escritas no bloco da cache e no bloco do nível inferior da hierarquia de memória (memória principal para uma cache). As caches na Seção 7.2 usaram esse esquema.
- *Write-back* (também chamada de *copy-back*): as informações são escritas apenas no bloco da cache. O bloco modificado é escrito no nível inferior da hierarquia apenas quando ele é substituído. Os sistemas de memória virtual sempre usam write-back, pelas razões explicadas na Seção 7.4.

Tanto write-back quanto write-through têm suas vantagens. As principais vantagens do write-back são as seguintes:

- As words individuais podem ser escritas pelo processador na velocidade em que a cache, não a memória, pode aceitar.
- Diversas escritas dentro de um bloco exigem apenas uma escrita no nível inferior da hierarquia.
- Quando blocos são escritos com write-back, o sistema pode fazer uso efetivo de uma transferência de alta largura de banda, já que o bloco inteiro é escrito.

O write-through possui estas vantagens:

- As falhas são mais simples e baratas porque nunca exigem que um bloco seja escrito de volta no nível inferior.
- O write-through é mais fácil de ser implementado do que o write-back, embora, para ser prática em um sistema de alta velocidade, uma cache write-through precisaria usar um buffer de escrita.

Em sistemas de memória virtual, apenas uma política write-back é viável devido à longa latência de uma escrita no nível inferior da hierarquia (o disco). À medida que os processadores continuam melhorando o desempenho em uma taxa mais rápida do que a memória principal baseada em DRAM, a taxa em que as escritas são geradas por um processador excederá a taxa em que o sistema de memória pode processá-las, até mesmo permitindo memórias físicas e logicamente mais largas. Como consequência, cada vez mais caches estão usando uma estratégia write-back.

Colocando em perspectiva

Embora as caches, as TLBs e a memória virtual inicialmente possam parecer muito diferentes, elas se baseiam nos mesmos dois princípios de localidade e podem ser entendidos examinando como lidam com quatro questões:

Questão 1: Onde um bloco pode ser colocado?

Resposta: Em um local (mapeamento direto), em alguns locais (associatividade por conjunto) ou em qualquer local (associatividade total).

Questão 2: Como um bloco é encontrado?

Resposta: Existem quatro métodos: indexação (como em uma cache diretamente mapeada), pesquisa limitada (como em uma cache associativa por conjunto), pesquisa completa (como em uma cache totalmente associativa) e tabela de consulta separada (como em uma tabela de páginas).

Questão 3: Que bloco é substituído em uma falha?

Resposta: Em geral, no bloco usado menos recentemente ou um bloco aleatório.

Questão 4: Como as escritas são tratadas?

Resposta: Cada nível na hierarquia pode usar write-through ou write-back.

modelo dos três Cs

Um modelo de cache em que todas as falhas são classificadas em uma de três categorias: falhas compulsórias, falhas de capacidade e falhas de conflito.

fallha compulsória

Também chamada de **fallha de partida a frio**. Uma falha de cache causada pelo primeiro acesso a um bloco que nunca esteve na cache.

fallha de capacidade

Uma falha de cache que ocorre porque a cache, mesmo com associatividade total, não pode conter todos os blocos necessários para satisfazer à requisição.

fallha de conflito

Também chamada de **fallha de colisão**. Uma falha de cache que ocorre em uma cache associativa por conjunto ou diretamente mapeada quando vários blocos competem pelo mesmo conjunto e que é eliminada em uma cache totalmente associativa do mesmo tamanho.

Os Três Cs: um modelo intuitivo para entender o comportamento das hierarquias de memória

Nesta seção, vamos examinar um modelo que esclarece as origens das falhas em uma hierarquia de memória e como as falhas serão afetadas por mudanças na hierarquia. Explicaremos as idéias em termos de caches, embora as idéias se apliquem diretamente a qualquer outro nível na hierarquia. Nesse modelo, todas as falhas são classificadas em uma de três categorias (os **três Cs**):

- **Falhas compulsórias:** são falhas de cache causadas pelo primeiro acesso a um bloco que nunca esteve na cache. Também são chamadas de **fallhas de partida a frio**.
- **Falhas de capacidade:** são falhas de cache causadas quando a cache não pode conter todos os blocos necessários durante a execução de um programa. As falhas de capacidade ocorrem quando os blocos são substituídos e, depois, recuperados.
- **Falhas de conflito:** são falhas de cache que ocorrem em caches associativas por conjunto ou diretamente mapeadas quando vários blocos disputam o mesmo conjunto. As falhas de conflito são aquelas falhas em uma cache diretamente mapeada ou associativa por conjunto que são eliminadas em uma cache totalmente associativa do mesmo tamanho. Essas falhas de cache também são chamadas de **fallhas de colisão**.

A Figura 7.31 mostra como a taxa de falhas se divide nas três origens. Essas origens de falhas podem ser diretamente atacadas mudando algum aspecto do projeto da cache. Como as falhas de conflito surgem diretamente da disputa pelo mesmo bloco de cache, aumentar a associatividade reduz as falhas de conflito. Entretanto, a associatividade pode aumentar o tempo de acesso, levando a um menor desempenho geral.

As falhas de capacidade podem facilmente ser reduzidas aumentando a cache; na verdade, as caches de segundo nível têm se tornado constantemente maiores durante muitos anos. É claro que, quando tornamos a cache maior, também precisamos ser cautelosos quanto ao aumento no tempo de acesso, que pode levar a um desempenho geral mais baixo. Por isso, as caches de primeiro nível cresceram lentamente ou nem isso.

Como as falhas compulsórias são geradas pela primeira referência a um bloco, a principal maneira de um sistema de cache reduzir o número de falhas compulsórias é aumentando o tamanho do bloco. Isso irá reduzir o número de referências necessárias para tocar cada bloco do programa uma vez por-

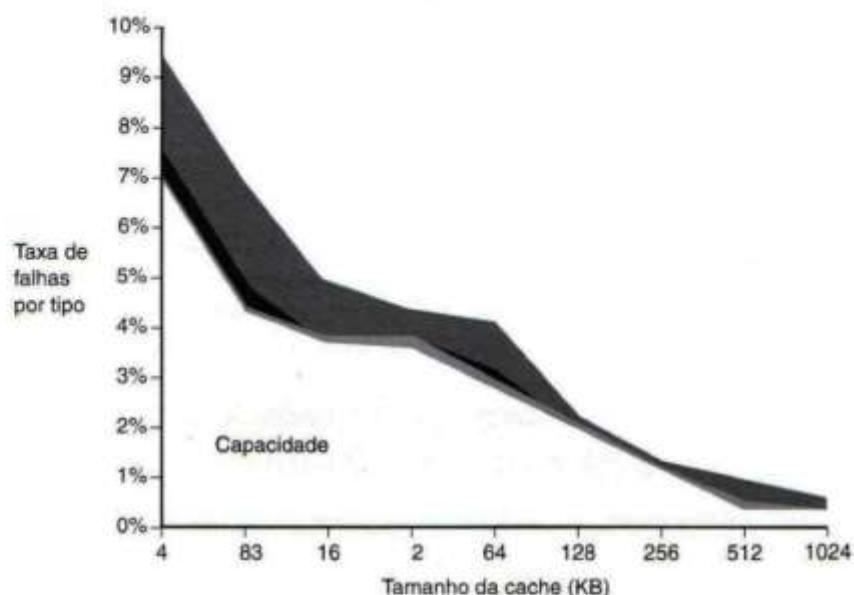


FIGURA 7.31 A taxa de falhas pode ser dividida em três origens de falha. Esse gráfico mostra a taxa de falhas total e seus componentes para uma faixa de tamanhos de cache. Esses dados são para os benchmarks de inteiro e ponto flutuante do SPEC2000 e são da mesma fonte dos dados na Figura 7.30. O componente da falha compulsória é de 0,006% e não pode ser visto nesse gráfico. O próximo componente é a taxa de falhas de capacidade, que depende do tamanho da cache. A parte do conflito, que depende da associatividade e do tamanho da cache, é mostrada para uma faixa de associatividades, de uma via a oito vias. Em cada caso, a seção rotulada corresponde ao aumento na taxa de falhas que ocorre quando a associatividade é alterada do próximo grau mais alto para o grau rotulado de associatividade. Por exemplo, a seção rotulada como *duas vias* indica as falhas adicionais surgindo quando o cache possui associatividade de dois em vez de quatro. Portanto, a diferença na taxa de falhas incorrida por uma cache diretamente mapeada em relação a uma cache totalmente associativa do mesmo tamanho é dada pela soma das seções rotuladas como *oito vias, quatro vias, duas vias e uma via*. A diferença entre oito vias e quatro vias é tão pequena que mal pode ser vista nesse gráfico.

que o programa consistirá em menos blocos de cache. Aumentar demais o tamanho do bloco pode ter um efeito negativo sobre o desempenho devido ao aumento na penalidade de falha.

A decomposição das falhas nos três Cs é um modelo qualitativo útil. Nos projetos de cache reais, muitas das escolhas de projeto interagem e mudar uma característica de cache freqüentemente afetará vários componentes da taxa de falhas. Apesar dessas deficiências, esse modelo é uma maneira útil de adquirir conhecimento sobre o desempenho dos projetos de cache.

A dificuldade de projetar hierarquias de memória é que toda mudança que melhore potencialmente a taxa de falhas também pode afetar negativamente o desempenho geral, como mostra a Figura 7.32. Essa combinação de efeitos positivos e negativos é o que torna o projeto de uma hierarquia de memória interessante.

Quais das seguintes afirmativas (se houver) normalmente são verdadeiras?

1. Não há um meio de reduzir as falhas compulsórias.
2. As caches totalmente associativas não possuem falhas de conflito.
3. Na redução de falhas, a associatividade é mais importante do que a capacidade.

Colocando em perspectiva

Verifique você mesmo

Mudança de projeto	Efeito sobre a taxa de falhas	Possível efeito negativo no desempenho
Aumentar o tamanho da cache	Diminui as falhas de capacidade	Pode aumentar o tempo de acesso
Aumentar a associatividade	Diminui a taxa de falhas devido a falhas de conflito	Pode aumentar o tempo de acesso
Aumentar o tamanho de bloco	Diminui a taxa de falhas para uma ampla faixa de tamanhos de bloco devido à localidade espacial	Aumenta a penalidade de falha. Blocos muito grandes podem aumentar a taxa de falhas

FIGURA 7.32 Dificuldades do projeto de hierarquias de memória.

7.6

Vida real: as hierarquias de memória do Pentium P4 e do AMD Opteron

Nesta seção, veremos a hierarquia de memória de dois microprocessadores modernos: o Intel Pentium P4 e o AMD Opteron. Em 2004, o P4 era usado em uma diversidade de desktops PC e pequenos servidores. O processador AMD Opteron está sendo encontrado nos servidores e clusters de topo de linha.

A Figura 7.33 mostra a fotografia do die do Opteron e a Figura 1.9 no Capítulo 1 mostra a fotografia do die do P4. Ambos possuem caches secundárias no die do processador principal. Essa integração reduz o tempo de acesso à cache secundária e também reduz o número de pinos do chip, já que não existe necessidade de um barramento para uma cache secundária.

As hierarquias de memória do P4 e do Opteron

A Figura 7.34 resume os tamanhos de endereço e as TLBs dos dois processadores.

Observe que o AMD Opteron possui quatro TLBs enquanto o P4 possui duas. Note também que os endereços virtuais e físicos não precisam corresponder ao tamanho de word. O AMD implementa

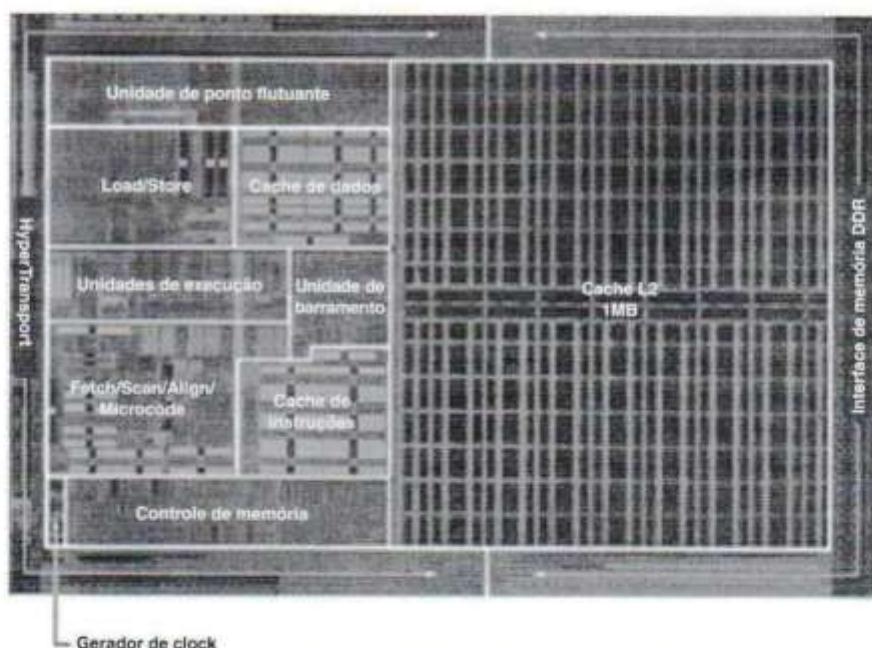


FIGURA 7.33 Uma fotografia do die do processador AMD Opteron com os componentes indicados. A cache L2 ocupa 42% do die. Os outros componentes, em ordem de tamanho, são HyperTransport: 13%, memória DDR: 10%, Fetch/Scan/Align/Microcode: 6%, controlador de memória: 4%, FPU: 4%, cache de instruções: 4%, cache de dados: 4%, unidades de execução: 3%, unidade de barramento: 2% e gerador de clock: 0,2%. Em uma tecnologia de 0,13, este die possui 193mm².

Característica	Intel Pentium P4	AMD Opteron
Endereço virtual	32 bits	48 bits
Endereço físico	36 bits	40 bits
Tamanho de página	4KB, 2/4MB	4KB, 2/4MB
Organização de TLB	1 TLB para instruções e 1 TLB para dados Ambas são associativas por conjunto de quatro vias Ambas usam substituição pseudo-LRU Ambas têm 128 entradas Falhas de TLB tratadas por hardware	2 TLBs para instruções e 2 TLBs para dados Ambas as TLBs L1 são totalmente associativas, substituição LRU Ambas as TLBs L2 são associativas por conjunto de quatro vias, LRU com round-robin Ambas as TLBs L1 têm 40 entradas Ambas as TLBs L2 têm 512 entradas Falhas de TLB tratadas por hardware

FIGURA 7.34 Tradução de endereços e hardware TLB para o Intel Pentium P4 e o AMD Opteron. O tamanho de word define o tamanho máximo do endereço virtual, mas um processador não precisa usar todos esses bits. O tamanho de endereço físico é independente do tamanho de word. O P4 tem uma TLB para instruções e uma TLB idêntica separada para dados, enquanto o Opteron possui uma TLB L1 e uma TLB L2 para instruções e TLBs L1 e L2 idênticas para dados. Os dois processadores fornecem suporte para páginas grandes, que são usadas para coisas como o sistema operacional ou para mapear um buffer de quadro. O esquema de página grande evita o uso de um grande número de entradas para mapear um único objeto que está sempre presente.

apenas 48 dos 64 bits possíveis do seu espaço virtual e 40 dos 64 bits possíveis do seu espaço de endereço físico. O Intel aumenta o espaço de endereço físico para 36 bits, embora nenhum programa isolado possa endereçar mais de 32 bits.

A Figura 7.35 mostra suas caches. Repare que a cache de dados L1 e as caches L2 são maiores no Opteron e que o P4 usa um tamanho de bloco maior para sua cache L2 do que sua cache de dados L1.

Embora o Opteron execute os mesmos programas IA-32 que o Pentium P4, sua maior diferença é que ele acrescentou um modo de endereçamento de 64 bits. Assim como o 80386 acrescentou um espaço de endereçamento plano de 32 bits e registradores de 32 bits à arquitetura 80286 de 16 bits anterior, o Opteron acrescenta à arquitetura IA-32 um novo modo, chamado *AMD64*, com espaço de endereçamento plano de 64 bits e registradores de 64 bits. Ele aumenta o contador de programa para 64 bits, estende os registradores de 32 bits para 64 bits, adiciona oito novos registradores de 64 bits e dobra o número de registradores SSE2. Em 2004, a Intel anunciou que os futuros processadores IA-32 incluiriam sua extensão de endereçamento de 64 bits.

Característica	Intel Pentium P4	AMD Opteron
Organização de cache L1	Caches de instruções e de dados divididos	Caches de instruções e de dados divididos
Tamanho de cache L1	8KB para dados, cache de trace de 96KB para instruções RISC (operações RISC de 12K)	64KB cada para instruções/dados
Associatividade de cache L1	Associativa por conjunto de 4 vias	Associativa por conjunto de 2 vias
Substituição L1	Substituição LRU aproximada	Substituição LRU
Tamanho de bloco L1	64 bytes	64 bytes
Política de escrita L1	Write-through	Write-back
Organização de cache L2	Unificada (instruções e dados)	Unificada (instruções e dados)
Tamanho de cache L2	512KB	1024KB (1MB)
Associatividade de cache L2	Associativa por conjunto de 8 vias	Associativa por conjunto de 16 vias
Substituição L2	Substituição LRU aproximada	Substituição LRU aproximada
Tamanho de bloco L2	128 bytes	64 bytes
Política de escrita L2	Write-back	Write-back

FIGURA 7.35 Caches de primeiro nível e segundo nível do Intel Pentium P4 e do AMD Opteron. As caches primárias do P4 são fisicamente indexadas e rotuladas; para uma descrição das alternativas, veja a Seção “Detalhamento” na página 398.

Técnicas para reduzir as penalidades de falha

Tanto o Pentium 4 quanto o AMD Opteron possuem otimizações adicionais que permitem reduzir a penalidade de falha. A primeira delas é o retorno da word requisitada primeiro em uma falha, como descrito na Seção “Detalhamento” da página 370. Ambos permitem que o processador continue executando instruções que acessam a cache de dados durante uma falha de cache. Essa técnica, chamada **cache não bloqueante**, é comumente usada quando os projetistas tentam ocultar a latência da falha de cache usando processadores com execução fora de ordem. Eles implementam dois tipos de não-bloqueio. *Acerto sob falha* permite acertos de cache adicionais durante uma falha, enquanto *falha sob acerto* permite múltiplas falhas de cache pendentes. O objetivo do primeiro desses dois é ocultar alguma latência de falha com outro trabalho, enquanto o objetivo do segundo é sobrepor a latência de duas falhas diferentes.

A sobreposição de uma grande fração dos tempos de falha para múltiplas falhas pendentes requer um sistema de memória de alta largura de banda, capaz de tratar múltiplas falhas em paralelo. Em sistemas desktop, a memória pode apenas ser capaz de tirar proveito limitado dessa capacidade, mas grandes servidores e multiprocessadores freqüentemente possuem sistemas de memória capazes de tratar mais de uma falha pendente em paralelo.

Os dois microprocessadores fazem prefetch de instruções e possuem um mecanismo de prefetch embutido no hardware para acessos a dados. Eles olham um padrão de falhas de dados e usam essas informações para tentar prever o próximo endereço a fim de começar a buscar os dados antes que a falha ocorra. Essas técnicas geralmente funcionam melhor ao acessar arrays em loops.

Uma grande dificuldade enfrentada pelos projetistas de cache é suportar processadores como o P4 e o Opteron, que podem executar mais de uma instrução de acesso à memória por ciclo de clock. Várias requisições podem ser suportadas na cache de primeiro nível por duas técnicas diferentes. A cache pode ser multiporta, permitindo mais de um acesso simultâneo ao mesmo bloco de cache. Entretanto, as caches multiporta normalmente são muito caras, já que as células de RAM em uma memória multiporta precisam ser muito maiores do que as células uniporta. O esquema alternativo é desmembrar a cache em bancos e permitir acessos múltiplos e independentes, desde que os acessos sejam a bancos diferentes. A técnica é semelhante à memória principal intercalada (veja a Figura 7.11).

Para reduzir o tráfego de memória em uma configuração de multiprocessador, a Intel possuía outras versões do P4 com caches on-chip muito maiores em 2004. Por exemplo, o Intel Pentium P4 Xeon vem com cache de *terceiro* nível *on-chip* de 1MB e destina-se a servidores de processadores duais. Um exemplo mais radical é o Intel Pentium P4 Extreme Edition, que vem com 2MB de cache L3 mas sem suporte para multiprocessamento. Esses dois chips são muito maiores e mais caros. Por exemplo, em 2004, um Precision Workstation 360 com um P4 de 3,2GHz custava cerca de US\$1.900. Um upgrade para o processador Extreme Edition acrescentava US\$500 ao preço. O Dell Precision Workstation 450, que permite processadores duais, custava cerca de US\$2.000 para um Xeon de 3,2GHz com 1MB de cache L3. Incluir um segundo processador como esse acrescentava US\$1.500 ao preço.

As sofisticadas hierarquias de memória desses chips e a grande fração dos dies dedicada às caches e às TLBs mostram o significativo esforço de projeto despendido para tentar diminuir a diferença entre tempos de ciclo de processador e latência de memória. Futuros avanços nos projetos de pipeline dos processadores, juntamente com o maior uso do multiprocessamento – que apresenta seus próprios problemas nas hierarquias de memória – fornecem muitos desafios novos para os projetistas.

Detalhamento: talvez a maior diferença entre os chips AMD e Intel seja o uso de uma cache de trace para a cache de instruções do P4, enquanto o AMD Opteron usa uma cache de instruções mais tradicional.

Em vez de organizar as instruções em um bloco de cache seqüencialmente para promover localidade espacial, uma cache de trace encontra uma sequência dinâmica de instruções incluindo desvios tomados para carregar em um bloco de cache. Portanto, os blocos de cache contêm traces dinâmicos das instruções executadas como determinado pela CPU, em vez de seqüências estáticas das instruções como determinado pelo layout da memória. Como isso acrescenta a previsão de desvios (Capítulo 6) à cache, os desvios precisam ser validados juntamente com os endereços para que tenham uma busca válida. Além disso, o P4 coloca as microoperações na cache (veja o Capítulo 5) em vez das instruções IA-32 como o Opteron.

Claramente, as caches de trace possuem mecanismos de mapeamento de endereços muito mais complicados, uma vez que os endereços não são mais alinhados às potências de dois do tamanho de word.

Entretanto, as caches de trace podem melhorar a utilização dos blocos de cache. Por exemplo, blocos muito longos nas caches convencionais podem ser adentrados a partir de um desvio tomado e, portanto, a primeira parte do bloco ocupa espaço na cache que pode não ser buscado. Da mesma forma, esses blocos podem deixar de ser utilizados por desvios tomados, de modo que a última parte do bloco pode ser desperdiçada. Uma vez que os desvios tomados ou jumps ocorrem a cada 5 a 10 instruções, a utilização de bloco efetiva é um verdadeiro problema para processadores como o Opteron, cujo bloco de 64 bytes provavelmente incluiria 16 a 24 instruções 80x86. As caches de trace armazenam instruções apenas do ponto de entrada até a saída do trace, evitando, assim, esse overhead do início e do final do bloco. Uma desvantagem das caches de trace é que elas potencialmente armazenam as mesmas instruções várias vezes na cache: desvios condicionais fazendo diferentes escolhas levam as mesmas instruções a se tornarem parte de trases separados, cada um aparecendo na cache.

Para levar em conta o tamanho maior das microoperações e a redundância inherente em uma cache de trace, a Intel afirma que a taxa de falhas da cache de trace de 96KB do P4, que contém 12K microoperações, é aproximadamente igual à de uma cache de 8KB, que contém cerca de 2 a 3K instruções IA-32.

7.7

Falácia e armadilhas

Como um dos aspectos mais naturalmente quantitativos da arquitetura de um computador, a hierarquia de memória pareceria ser menos vulnerável às falácias e armadilhas. Não só houve muitas falácias propagadas e armadilhas encontradas, mas algumas levaram a grandes resultados negativos. Começamos com uma armadilha que freqüentemente pega estudantes em exercícios e exames.

Armadilha: esquecer-se de considerar o endereço em bytes ou o tamanho de bloco de cache ao simular uma cache.

Quando estamos simulando uma cache (manualmente ou por computador), precisamos levar em conta o efeito de um endereçamento em bytes e blocos multiword ao determinar para qual bloco de cache um determinado endereço é mapeado. Por exemplo, se tivermos uma cache diretamente mapeada de 32 bytes com um tamanho de bloco de 4 bytes, o endereço em bytes 36 é mapeado no bloco 1 da cache, já que o endereço em bytes 36 é o endereço de bloco 9 e $(9 \bmod 8) = 1$. Por outro lado, se o endereço 36 for um endereço em words, então, ele é mapeado no bloco $(36 \bmod 8) = 4$. O problema deve informar claramente a base do endereço.

De modo semelhante, precisamos considerar o tamanho do bloco. Suponha que tenhamos uma cache com 256 bytes e um tamanho de bloco de 32 bytes. Em que bloco o endereço em bytes 300 se encontra? Se dividirmos o endereço 300 em campos, poderemos ver a resposta:

31	30	29	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	1	0	1	1	0	0
Endereço de bloco										Número de bloco de cache		Offset de bloco de cache					

O endereço em bytes 300 é o endereço de bloco

$$\left\lfloor \frac{300}{32} \right\rfloor = 9$$

O número de blocos na cache é

$$\left\lceil \frac{256}{32} \right\rceil = 8$$

O bloco número 9 cai no bloco de cache número $(9 \bmod 8) = 1$.

Esse erro pega muitas pessoas, incluindo os autores (nos rascunhos anteriores) e instrutores que esquecem se pretendiam que os endereços estivessem em words, bytes ou números de bloco. Lembrar-se dessa armadilha ao realizar os exercícios.

Armadilha: ignorar o comportamento do sistema de memória ao escrever programas ou gerar código em um compilador.

Isso poderia facilmente ser escrito como uma armadilha: “Os programadores podem ignorar as hierarquias de memória ao escrever código.” Ilustramos com um exemplo usando multiplicação de matrizes, para complementar a comparação de ordenações na Figura 7.18.

Aqui está o loop interno da versão da multiplicação de matrizes do 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];
```

Quando executado com entradas que são matrizes de dupla precisão de 500×500 , o tempo de execução de CPU dos loops anteriores em uma CPU MIPS com uma cache secundária de 1MB foi aproximadamente metade da velocidade comparada a quando a ordem dos loops é alterada para k, j, i (de modo que i seja o mais interno)! A única diferença é como o programa acessa a memória e o efeito resultante na hierarquia de memória. Outras otimizações de compilador usando uma técnica chamada *blocagem* podem resultar em um tempo de execução que é outras quatro vezes mais rápido para esse código!

Armadilha: usar tempo médio de acesso à memória para avaliar a hierarquia de memória de um processador com execução fora de ordem.

Se um processador é suspenso durante uma falha de cache, você pode calcular separadamente o tempo de stall de memória e o tempo de execução do processador, e, portanto, avaliar a hierarquia de memória de forma independente usando o tempo médio de acesso à memória.

Se o processador continuar executando instruções e puder até sustentar mais falhas de cache durante uma falha de cache, então, a única avaliação precisa da hierarquia de memória é simular o processador *com execução* fora de ordem juntamente com a hierarquia de memória.

Armadilha: estender um espaço de endereçamento acrescentando segmentos sobre um espaço de endereçamento não segmentado.

Durante a década de 1970, muitos programas ficaram tão grandes que nem todo o código e dados podiam ser endereçados apenas com um endereço de 16 bits. Os computadores, então, foram revisados para oferecer endereços de 32 bits, quer por meio de um espaço de endereçamento de 32 bits não segmentado (também chamado de *espaço de endereçamento plano*), quer acrescentando 16 bits de segmento ao endereço de 16 bits existente. De uma perspectiva de marketing, acrescentar segmentos que fossem visíveis ao programador e que forçassem o programador e o compilador a decomponerem programas em segmentos podia resolver o problema de endereçamento. Infelizmente, existe problema toda vez que uma linguagem de programação quer um endereço que seja maior do que um segmento, como índices para grandes arrays, ponteiros irrestritos ou parâmetros por referência. Além disso, acrescentar segmentos pode transformar todos os endereços em duas words – uma para o número do segmento e outra para o offset do segmento –, causando problemas no uso dos endereços em registradores. Dado o tamanho das DRAMs e a Lei de Moore, muitos dos sistemas de 32 bits atuais estão enfrentando problemas semelhantes.

7.8 Comentários finais

A dificuldade de construir um sistema de memória para fazer frente aos processadores mais rápidos é acentuada pelo fato de que a matéria-prima para a memória principal, DRAMs, é essencialmente a mesma nos computadores mais rápidos que nos computadores mais lentos e baratos. A Figura 7.36 compara a hierarquia de memória dos microprocessadores destinados a aplicações de desktop, de servidor e embutidas. As caches L1 são semelhantes entre aplicações, com as principais diferenças sendo o tamanho da cache L2, o tamanho do die, a velocidade de clock do processador e as instruções despachadas por clock.

É o princípio da localidade que nos dá uma chance de superar a longa latência do acesso à memória – e a confiabilidade dessa técnica é demonstrada em todos os níveis da hierarquia de memória. Embora esses níveis da hierarquia pareçam muito diferentes em termos quantitativos, eles seguem estratégias semelhantes em sua operação e exploram as mesmas propriedades da localidade.

Como as velocidades de processador continuam a aumentar mais rapidamente do que os tempos de acesso à DRAM ou os tempos de acesso a disco, a memória, cada vez mais, será o fator limitador do desempenho. Os processadores melhoram em desempenho em uma alta velocidade, e as DRAMs agora estão dobrando sua densidade aproximadamente a cada dois anos. O *tempo de acesso* das DRAMs, no entanto, está melhorando em uma velocidade muito mais lenta – menos de 10% por ano. A Figura 7.37 mostra uma representação gráfica do desempenho do processador em relação a uma

MPU	AMD Opteron	Intrinsity FastMATH	Intel Pentium 4	Intel PXA250	Sun UltraSPARC IV
Conjunto de instruções	IA-32, AMD64	MIPS32	IA-32	ARM	SPARC v9
Aplicação pretendida	Servidor	Embutido	Desktop	Embutido de baixo consumo	Servidor
Tamanho do die (mm^2) (2004)	193	122	217		356
Instruções despachadas por clock	3	2	3 operações RISC	1	4 × 2
Velocidade de clock (2004)	2,0GHz	2,0GHz	3,2GHz	0,4GHz	1,2GHz
Cache de instruções	64KB, associativa por conjunto de 2 vias	16KB, diretamente mapeada	cache de trace com 12.000 operações RISC (~96 KB)	32KB, associativa por conjunto de 32 vias	32KB, associativa por conjunto de 4 vias
Latência (clocks)	3	4	4	1	2
Cache de dados	64KB, associativa por conjunto de 2 vias	16KB, associativa por conjunto de 1 via	8KB, associativa por conjunto de 4 vias	32KB, associativa por conjunto de 32 vias	64KB, associativa por conjunto de 4 vias
Latência (clocks)	3	3	2	1	2
Entradas de TLB (TLB I/D/L2)	40/40/512/512	16	128/128	32/32	128/512
Tamanho de página mínimo	4KB	4KB	4KB	1KB	8KB
Cache L2 on-chip	1.024KB, associativa por conjunto de 16 vias	1.024KB, associativa por conjunto de 4 vias	512KB, associativa por conjunto de 8 vias	–	–
Cache L2 off-chip	–	–	–	–	16MB, associativa por conjunto de 2 vias
Tamanho de bloco (L1/L2, bytes)	64	64	64/128	32	32

FIGURA 7.36 Microprocessadores para desktops, embutidos e para servidores em 2004. De uma perspectiva da hierarquia de memória, a principal diferença entre as categorias é a cache L2. Não há uma cache L2 para o embutido de baixo consumo, uma grande L2 on-chip para o embutido e para o desktop e uma off-chip de 16MB para o servidor. As velocidades de clock de processador também variam: 0,4GHz para embutido de baixo consumo, 1GHz ou mais alta para o restante. Observe que o UltraSPARC IV possui dois processadores no chip.

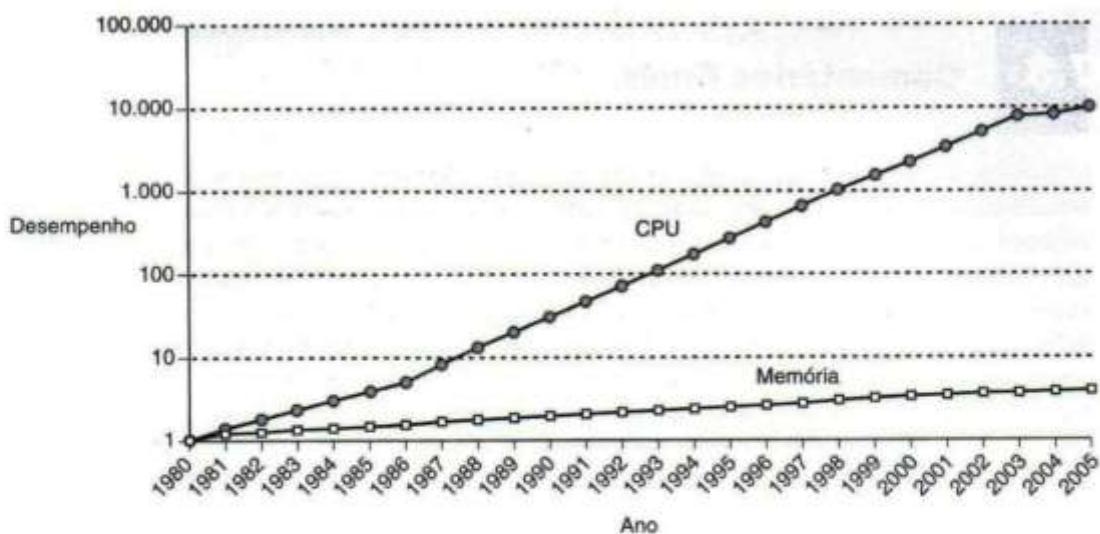


FIGURA 7.37 Usando seus desempenhos em 1980 como base, o tempo de acesso das DRAMs comparado com o desempenho dos processadores está representado ao longo do tempo. Note que o eixo vertical precisa estar em uma escala logarítmica para registrar o tamanho da diferença de desempenho processador-DRAM. A linha de base da memória é de 64KB DRAM em 1980, com três anos para a próxima geração até 1996 e dois anos daí em diante, com uma melhoria de desempenho de 7% por ano na latência. A linha do processador considera uma melhoria de 35% por ano até 1986 e uma melhoria de 55% até 2003. Ela se torna mais lenta a partir daí.

melhoria de desempenho anual de 7% na latência da DRAM. Enquanto a latência melhora lentamente, os recentes avanços na tecnologia da DRAM (DRAMs de dupla velocidade de dados e técnicas relacionadas) levaram a maiores aumentos na largura de banda da memória. Essa largura de banda de memória potencialmente maior permitiu que os projetistas aumentassem os tamanhos de bloco de cache com menores aumentos na penalidade de falha.

Tendências recentes

O desafio de projetar hierarquias de memória para eliminar essa diferença de crescimento, como observamos na Seção “Colocando em perspectiva” da página 411, é que todas as escolhas de projeto de hardware para hierarquias de memória têm um efeito positivo e um negativo sobre o desempenho. Isso significa que, para cada nível da hierarquia, existe um ponto de desempenho ótimo por programa, que precisa incluir algumas falhas. Se esse for o caso, como podemos solucionar a diferença de crescimento entre as velocidades de processador e os níveis mais baixos da hierarquia? Essa questão é atualmente objeto de muita pesquisa.

As caches de primeiro nível inicialmente ajudavam a diminuir a diferença que estava crescendo entre tempo de ciclo de clock de processador e o tempo de ciclo de clock de SRAM off-chip. Para reduzir a diferença entre caches on-chip pequenas e DRAM, as caches de segundo nível se tornaram comuns. Hoje, todos os computadores desktop usam caches de segundo nível on-chip, e as caches de terceiro nível estão se tornando populares em alguns segmentos. As caches multinível também possibilitam o uso mais fácil de outras otimizações por dois motivos. Primeiro, os parâmetros de projeto de uma cache de segundo ou terceiro nível são diferentes dos de uma cache de primeiro nível. Por exemplo, como uma cache de segundo ou terceiro nível será muito maior, é possível usar tamanhos de bloco maiores. Segundo, uma cache de segundo ou terceiro nível não está constantemente sendo usada pelo processador como uma cache de primeiro nível. Isso nos permite considerar fazer com que, quando estiver ociosa, uma cache de segundo ou terceiro nível realize alguma tarefa que possa ser útil para evitar futuras falhas.

Outra direção possível é recorrer à ajuda de software. Controlar eficientemente a hierarquia de memória usando uma variedade de transformações de programa e recursos de hardware é um impor-

tante foco dos avanços dos compiladores. Duas idéias diferentes estão sendo exploradas. Uma é reorganizar o programa para melhorar sua localidade espacial e temporal. Esse método focaliza os programas orientados para loops que usam grandes arrays como a principal estrutura de dados; grandes problemas de álgebra linear são um exemplo típico. Reestruturando os loops que acessam os arrays, podemos obter uma localidade – e, portanto, um desempenho de cache – substancialmente melhor. O exemplo na página 416 mostrou como poderia ser eficaz até mesmo uma simples mudança da estrutura de loop.

Outra direção é tentar usar **prefetching** direcionado pelo compilador. Em prefetching, um bloco de dados é trazido para a cache antes de ser realmente referenciado. O compilador tenta identificar os blocos de dados necessários no futuro e, usando instruções especiais, diz à hierarquia de memória para mover os blocos para a cache. Quando o bloco é efetivamente referenciado, ele é encontrado na cache, em vez de causar uma falha de cache. O uso de caches secundárias tornou o prefetching ainda mais atraente, já que a cache secundária pode estar envolvida em um prefetching, enquanto a cache primária continua a servir requisições do processador.

Como veremos no Capítulo 9, os sistemas de memória também são um importante tópico de projeto para processadores paralelos. A crescente importância da hierarquia de memória na determinação do desempenho do sistema em sistemas uni ou multiprocessador significa que essa relevante área continuará a ser objeto de projetistas e pesquisadores ainda por vários anos.

prefetching Uma técnica em que os blocos de dados necessários no futuro são colocados na cache pelo uso de instruções especiais que especificam o endereço do bloco.

7.9

Perspectiva histórica e leitura adicional

Esta seção de história oferece um resumo das tecnologias de memória, das linhas de atraso de mercúrio à DRAM, a invenção da hierarquia de memória e os mecanismos de proteção, e conclui com uma breve história dos sistemas operacionais, incluindo CTSS, MULTICS, UNIX, BSD UNIX, MS-DOS, Windows e Linux.

7.10

Exercícios

7.1 [5] <§7.1> A SRAM é comumente usada para implementar caches on-chip pequenas e rápidas enquanto a DRAM é usada para memória principal maior e mais rápida. No passado, um projeto comum para supercomputadores era construir máquinas sem caches e memórias principais criadas inteiramente de SRAM (o Cray C90, por exemplo, um computador muito rápido em sua época). Se o custo não fosse um problema, você ainda desejaria projetar um sistema dessa maneira?

7.2 [10] <§7.2> Descreva as características gerais de um programa que apresentaria muito pouca localidade temporal e espacial com relação aos acessos a dados. Forneça um programa de exemplo (pseudocódigo está bom).

7.3 [10] <§7.2> Descreva as características gerais de um programa que apresentaria muita localidade temporal mas muito pouca localidade espacial com relação aos acessos a dados. Forneça um programa de exemplo (pseudocódigo está bom).

7.4 [10] <§7.2> Descreva as características gerais de um programa que apresentaria muito pouca localidade temporal mas muita localidade espacial com relação aos acessos a dados. Forneça um programa de exemplo (pseudocódigo está bom).

7.5 [3/3] <§7.2> Um novo processador pode usar uma cache write-through ou write-back, selecionada por meio de software.

- a. Considere que o novo processador executará aplicações com alta exigência de dados e com um grande número de operações load e store. Explique qual política de escrita de cache deve ser usada.
- b. Considere a mesma questão mas, desta vez, para um sistema cuja segurança é um fator crítico e em que a integridade dos dados é mais importante do que o desempenho da memória.

7.6 [10] <§7.2> ■ Aprofundando o aprendizado: Localidade

7.7 [10] <§7.2> ■ Aprofundando o aprendizado: Localidade

7.8 [10] <§7.2> ■ Aprofundando o aprendizado: Localidade

7.9 [10] <§7.2> Aqui está uma série de referências de endereços dados como endereços em words: 2, 3, 11, 16, 21, 13, 64, 48, 19, 11, 3, 22, 4, 27, 6 e 11. Considerando uma cache diretamente mapeada com 16 blocos de uma word cada, que está inicialmente vazia, indique cada referência na lista como um acerto ou uma falha e mostre o conteúdo final da cache.

7.10 [10] <§7.2> Usando a série de referências fornecida no Exercício 7.9, mostre os acertos e falhas e o conteúdo final da cache para uma cache diretamente mapeada com blocos de quatro words cada e um *tamanho total* de 16 words.

7.11 [15] <§7.2> Dado o seguinte pseudocódigo:

```
int array[10000,100000];
para cada elemento do array[i][j] {
    array[i][j] = array[i][j]*2;
}
```

escreva dois programas em C que implementem esse algoritmo: um deve acessar os elementos do array na ordem linha primeiro, e o outro deve acessá-los na ordem coluna primeiro. Compare os tempos de execução dos dois programas. O que isso lhe diz sobre os efeitos do layout de memória no desempenho da cache?

7.12 [10] <§7.2> Calcule o número total de bits necessários para implementar a cache da Figura 7.9. Esse número é diferente do tamanho da cache, que normalmente se refere ao número de bytes de dados armazenados na cache. O número de bits necessários para implementar a cache representa a quantidade total de memória necessária para armazenar todos os dados, tags e bits de validade.

7.13 [10] <§7.2> Encontre um método para eliminar a porta lógica AND no bit de validade da Figura 7.7. (Dica: você precisa mudar a comparação.)

7.14 [10] <§7.2> Considere uma hierarquia de memória usando uma das três organizações para a memória principal mostradas na Figura 7.11. Suponha que o tamanho de bloco da cache é de 16 words, que a largura da organização (b) da figura é de quatro words e que o número de bancos na organização (c) é quatro. Se a latência da memória principal para um novo acesso é 10 ciclos de clock de barramento de memória e o tempo de transferência é de um ciclo de clock de barramento de memória, quais são as penalidades de memória para cada uma dessas organizações?

7.15 [10] <§7.2> ■ Aprofundando o aprendizado: Desempenho da cache.

7.16 [15] <§7.2> A cache C1 é diretamente mapeada com 16 blocos de uma word cada. A cache C2 é diretamente mapeada com 4 blocos de quatro words cada. Considere que a penalidade de falha para C1 é de 8 ciclos de clock de barramento de memória e a penalidade de falha para C2 é de 11 ciclos de clock de barramento de memória. Considerando que as caches estão inicialmente vazias, encontre uma seqüência de referências para a qual C2 tenha uma taxa de falhas mais baixa, mas gaste mais ciclos de clock de barramento de memória em falhas de cache do que C1. Use endereços em words.

7.17 [10] <§7.2> Considere um sistema de memória que suporte intercalação de quatro leituras ou quatro escritas. Dados os endereços de memória a seguir na ordem em que aparecem no barramento de memória: 3, 9, 17, 2, 51, 37, 13, 4, 8, 41, 67, 10, quais desses resultarão em um conflito de banco?

7.18 [3 horas] <§7.3> Use um simulador de cache para simular várias organizações de cache diferentes para as primeiras 1 milhão de referências em um trace do gcc. Considere uma cache de instruções de 32KB e uma cache de dados de 32KB usando a mesma organização. Você deve escolher pelo menos dois tipos de associatividade e dois tamanhos de bloco. Desenhe um diagrama como o da Figura 7.17 que mostre a organização da cache de dados com a melhor taxa de acertos.

7.19 [1 dia] <§7.3> Você foi incumbido de projetar uma cache para um novo sistema. Ele tem um endereço físico em bytes de 32 bits e requer caches de instruções e de dados separados. As SRAMs possuem um tempo de acesso de 1,5ns e um tamanho de $32K \times 8$ bits, e você tem um total de 16 SRAMs para usar. A penalidade de falha para o sistema de memória é de $8 + 2 \times$ tamanho de bloco em words. Usar associatividade por conjunto acrescenta 0,2ns ao tempo de acesso à cache. Usando as primeiras 1 milhão de referências do gcc, encontre as melhores organizações de cache I e D, dadas as SRAMs disponíveis.

7.20 [10] <§§7.2, B.5> ■ **Aprofundando o aprendizado:** Configurações de cache

7.21 [10] <§§7.2, B.5> ■ **Aprofundando o aprendizado:** Configurações de cache

7.22 [10] <§7.3> ■ **Aprofundando o aprendizado:** Operação de cache

7.23 [10] <§7.3> ■ **Aprofundando o aprendizado:** Operação de cache

7.24 [10] <§7.3> ■ **Aprofundando o aprendizado:** Operação de cache

7.25 [5] <§7.3> A associatividade normalmente melhora a taxa de falhas, mas nem sempre. Forneça uma curta série de referências de endereços para a qual uma cache associativa por conjunto de duas vias com substituição LRU experimentaria mais falhas do que uma cache diretamente mapeada do mesmo tamanho.

7.26 [15] <§7.3> Suponha que o tamanho de endereço de um computador seja k bits (usando endereçamento em bytes), o tamanho de cache seja S bytes, o tamanho de bloco seja B bytes e a cache seja associativa por conjunto de A vias. Considere que B é uma potência de dois e, portanto, $B = 2^b$. Descubra quais são as seguintes quantidades em termos de S , B , A e k : o número de conjuntos na cache, o número de bits de índice no endereço e o número de bits necessários para implementar a cache (veja o Exercício 7.12).

7.27 [10] <§7.3> ■ **Aprofundando o aprendizado:** Configurações de cache

7.28 [10] <§7.3> ■ **Aprofundando o aprendizado:** Configurações de cache

7.29 [20] <§7.3> Considere três processadores com diferentes configurações de cache:

- *Cache 1*: diretamente mapeada com blocos de uma word
- *Cache 2*: diretamente mapeada com blocos de quatro words
- *Cache 3*: associativa por conjunto de duas vias com blocos de quatro words

As seguintes medições de taxa de falhas foram feitas:

- *Cache 1*: a taxa de falhas de instruções é 4%; a taxa de falhas de dados é 6%.
- *Cache 2*: a taxa de falhas de instruções é 2%; a taxa de falhas de dados é 4%.
- *Cache 3*: a taxa de falhas de instruções é 2%; a taxa de falhas de dados é 3%.

Para esses processadores, a metade das instruções contém uma referência de dados. Considere que a penalidade de falha da cache é $6 + o$ tamanho de bloco em words. O CPI para esse workload foi medido em um processador com o Cache 1 e foi determinado como 2,0. Descubra que processador gasta mais ciclos nas falhas de cache.

7.30 [5] <§7.3> Os tempos de ciclo para os processadores do Exercício 7.29 são 420ps para o primeiro e segundo processadores e 310ps para o terceiro processador. Determine qual processador é o mais rápido e qual é o mais lento.

7.31 [15] <§7.3> Suponha que a cache para o sistema descrito no Exercício 7.29 seja associativa por conjunto de duas vias e tenha blocos de oito words e um tamanho total de 16KB. Mostre a organização e o acesso de cache usando o mesmo formato da Figura 7.17.

7.32 [10] <§§7.2, 7.4> O seguinte programa em C é executado (sem quaisquer otimizações) em um processador com uma cache que tem blocos de oito words (32 bytes) e contém 256 bytes de dados:

```
int i,j,c,stride,array[512];
...
for (i=0; i<10000; i++)
    for (j=0; j<512; j=j+stride)
        c = array[j]+17;
```

Se considerarmos apenas a atividade de cache gerada pelas referências ao array e considerarmos que os inteiros são words, qual é a taxa de falhas esperada quando a cache é diretamente mapeada e stride é igual a 256? E se stride for igual a 255? Algum destes mudaria se a cache fosse associativa por conjunto de duas vias?

7.33 [10] <§§7.3, B.5> ■ Aprofundando o aprendizado: Configurações de cache

7.34 [5] <§§7.2–7.4> ■ Aprofundando o aprendizado: Interações da hierarquia de memória

7.35 [4 horas] <§§7.2–7.4> Queremos usar um simulador de cache para simular várias organizações de TLB e memória virtual. Use o primeiro 1 milhão de referências do gcc para essa avaliação. Queremos saber a taxa de falhas de TLB para cada uma das seguintes TLBs e tamanhos de página:

1. TLB de 64 entradas com associatividade total e páginas de 4KB
2. TLB de 32 entradas com associatividade total e páginas de 8KB
3. TLB de 64 entradas com associatividade de oito vias e páginas de 4KB
4. TLB de 128 entradas com associatividade de quatro vias e páginas de 4KB

7.36 [15] <§7.4> Considere um sistema de memória virtual com as seguintes propriedades:

- Endereço de byte virtual de 40 bits
- Páginas de 16KB
- Endereço de byte físico de 36 bits

Qual é o tamanho total da tabela de páginas para cada processo nesse processador, considerando que os bits de validade, de proteção, de modificação e de uso usam um total de 4 bits, e que todas as páginas virtuais estão em uso? (Considere que os endereços de disco não são armazenados na tabela de páginas.)

7.37 [15] <§7.4> Considere que o sistema de memória virtual do Exercício 7.36 seja implementado com uma TLB associativa por conjunto de duas vias com um total de 256 entradas de TLB. Mostre o mapeamento virtual para físico com uma figura como a Figura 7.24. Indique a largura de todos os campos e sinais.

7.38 [10] <§7.4> Um processador possui uma TLB de 16 entradas e usa páginas de 4KB. Quais são as consequências de desempenho desse sistema de memória se um programa acessar pelo menos 2MB da memória ao mesmo tempo? Alguma coisa pode ser feita para melhorar o desempenho?

7.39 [10] <§7.4> Os overflows de buffer são uma falha comum usada para ganhar controle de um sistema. Se um buffer estiver localizado na pilha, um hacker poderá causar o overflow do buffer e inserir uma seqüência de instruções maliciosas comprometendo o sistema. Você pode imaginar um mecanismo de hardware que poderia ser usado para impedir isso?

7.40 [15] <§7.4> ■ Aprofundando o aprendizado: Tabelas de páginas hierárquicas

7.41 [15] <§7.4> ■ Aprofundando o aprendizado: Tabelas de páginas hierárquicas

7.42 [5] <§7.5> Se todas as falhas são classificadas em uma de três categorias – compulsórias, de capacidade ou de conflito (como abordado na página 410) –, que falhas provavelmente são reduzidas quando um programa é reescrito de modo que exija menos memória? E se a velocidade de clock do processador em que o programa está sendo executado for aumentada? E se a associatividade da cache existente for aumentada?

7.43 [5] <§7.5> O seguinte programa em C poderia ser usado para ajudar a construir um simulador de cache. Muitos dos tipos de dados não foram definidos, mas o código descreve precisamente as ações que ocorrem durante um acesso de leitura a uma cache diretamente mapeada.

```
word LerCacheDiretamenteMapeada(endereço a)
    static Entrada cache[CACHE_SIZE_IN_WORDS];
    Entrada e = cache[a.index];
    if (e.valid == FALSE || e.tag != a.tag) {
        e.valid = true;
        e.tag = a.tag;
        e.data = ler_memória(a);
    }
    return e.data;
```

Sua tarefa é modificar esse código para produzir uma correta descrição das ações que ocorrem durante um acesso de leitura a uma cache diretamente mapeada com blocos de múltiplas words.

7.44 [8] <§7.5> Este exercício é semelhante ao Exercício 7.43, exceto que, desta vez, escreva o código para acessos de leitura a uma cache associativa por conjunto de n vias com blocos de uma word. Note que seu código provavelmente sugerirá que as comparações são de natureza seqüencial quando, na verdade, elas seriam realizadas em paralelo pelo hardware real.

7.45 [8] <§7.5> Estenda sua solução do Exercício 7.43 incluindo a especificação de um novo procedimento para tratar acessos de escrita, considerando uma política write-through. Considere se sua solução para tratar acessos de leitura precisa ou não ser modificada.

7.46 [8] <§7.5> Estenda sua solução do Exercício 7.43 incluindo a especificação de um novo procedimento para tratar acessos de escrita, considerando uma política write-back. Considere se sua solução para tratar acessos de leitura precisa ou não ser modificada.

7.47 [8] <§7.5> Este exercício é semelhante ao Exercício 7.45, mas, desta vez, estenda sua solução do Exercício 7.44. Considere que a cache usa substituição aleatória.

7.48 [8] <§7.5> Este exercício é semelhante ao Exercício 7.46, mas, desta vez, estenda sua solução do Exercício 7.44. Considere que a cache usa substituição aleatória.

7.49 [5] <§§7.7–7.8> Por que um compilador poderia realizar a seguinte otimização?

```
/* Antes */
for (j = 0; j < 20; j++)
    for (i = 0; i < 200; i++)
        x[i][j] = x[i][j] + 1;

/* Depois */
for (i = 0; i < 200; i++)
    for (j = 0; j < 20; j++)
        x[i][j] = x[i][j] + 1;
```

§7.1: página 357, 1.

§7.2: página 371, 1 e 4: Uma penalidade de falha menor pode levar a blocos menores, mas uma largura de banda de memória mais alta normalmente leva a blocos maiores, já que a penalidade de falha é apenas ligeiramente maior.

§7.3: página 384, 1.

§7.4: página 406, 1-a, 2-c, 3-c, 4-d.

§7.5: página 411, 2.

**Respostas
das Seções
“Verifique
você mesmo”**

Computadores no mundo real

Salvando os tesouros artísticos do mundo

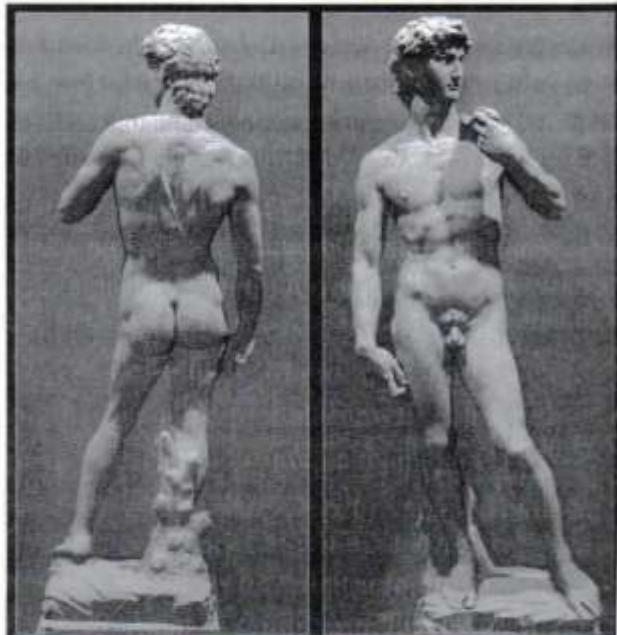
Problema: encontrar uma maneira de ajudar a conservar obras de arte ameaçadas por fatores ambientais e envelhecimento ou danificadas por tentativas de restauração anteriores sem causar mais danos às obras de arte insubstituíveis.

Solução: usar computadores e instrumentação científica para analisar a obra de arte e sua montagem, permitindo que conservadores e restauradores de arte realizem uma preservação mais instruída e bem-sucedida de uma obra de arte.

A conservação e a restauração artísticas desenvolveram-se em campos de alta tecnologia que fazem extensivo uso da computação e da instrumentação científica. Por exemplo, uma das formas de arte mais difíceis de serem restauradas e mantidas são os afrescos, que são pintados no reboco molhado de uma parede ou teto. A umidade e o calor modificam a superfície e causam deterioração; assim também, a poluição do ar, a fumaça de velas e outros contaminantes atacam diretamente a pintura e acrescentam sujeira e manchas que cobrem a obra de arte original.

Durante a restauração dos afrescos de Michelangelo na Capela Sistina, foram usados computadores para examinar o teto, localizar rachaduras e mapear precisamente a superfície e os afrescos.

Como a espessura do teto e das paredes varia de cerca de noventa centímetros a quase dois metros, existem diferenças significantes no comportamento térmico, o que, por sua vez, afeta a pintura da superfície. Os computadores foram usados para modelar toda a estrutura, incluindo a alta umidade produzida quando milhares de pessoas permanecem dentro da capela em um dia quente! Isso levou a um sistema de climatização controlada



Uma varredura a laser da estátua de Davi de Michelangelo.

do por computador que usa sensores instalados em locais estratégicos. O objetivo é manter os visitantes frios enquanto se preserva a obra de arte de Michelangelo para as próximas gerações.

Possivelmente, a área da conservação artística mais beneficiada pela disponibilidade de computação de alto desempenho e baixo custo é a restauração de quadros. Três técnicas – reflectografia infravermelha, geração de imagem ultravioleta e radiografia – são as mais utilizadas. Devido à necessidade da geração altamente precisa de imagens de alta resolução, câmeras ou scanners de raios-X controlados por computador são usados em todas essas técnicas. Isso resulta em um mosaico de imagens, que são, então, montadas por um computador. A combinação do movimento controlado por computador de uma câmera ou scanner de raios-X e a composição computadorizada subsequente de dezenas a milhares de imagens permite a digitalização de grandes superfícies em altíssima resolução.

A reflectografia infravermelha usa a luz no espectro próximo ao infravermelho e uma câmera digital para detectar a intensidade da reflexão da luz a partir da superfície de um quadro, mural ou

afresco. Essa técnica é útil para localizar o desenho de base que a maioria dos artistas usa para esboçar inicialmente as formas em uma pintura. O desenho de base, normalmente feito em preto, usando carvão, absorve a luz infravermelha. Na figura a seguir estão duas imagens de um quadro: uma mostrada na luz normal (à esquerda) e outra usando a técnica de reflectografia infravermelha (à direita).

Os restauradores usam a geração de imagens ultravioleta para descobrir as cores originais de uma pintura retocada. A radiografia fornece informações semelhantes, já que os pigmentos branco e amarelo que foram cobertos ou repintados aparecem mais escuros devido ao seu conteúdo.

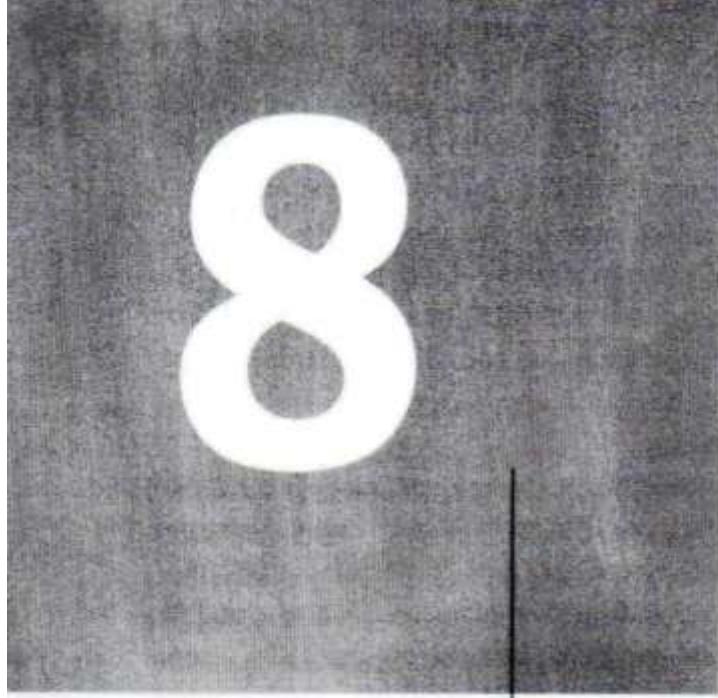
As tecnologias de digitalização também têm sido aplicadas em objetos de arte tridimensionais, como esculturas. O Davi de Michelangelo foi digitalizado usando um laser por um grupo liderado pelo Professor Marc Levoy, da Universidade de Stanford. O banco de dados resultante para uma digitalização com 0,29mm de resolução consiste em mais de 2 bilhões de polígonos e 32 gigabytes de dados. O projeto Michelangelo Digital criou um modelo detalhado da famosa escultura, que será útil tanto para conservação quanto como uma ferramenta acadêmica para estudantes de todo o mundo. Duas das muitas imagens que podem ser derivadas da digitalização tridimensional, mostradas de forma oposta.



Uma imagem da Capela Sistina na luz normal (esquerda) e em infravermelho (direita).

Para saber mais, veja estas referências

Conserving paintings, um site dedicado ao laboratório de imagens digitais da Universidade de Harvard.
Sistine Chapel, uma breve história da Capela Sistina.
O projeto Michelangelo Digital.



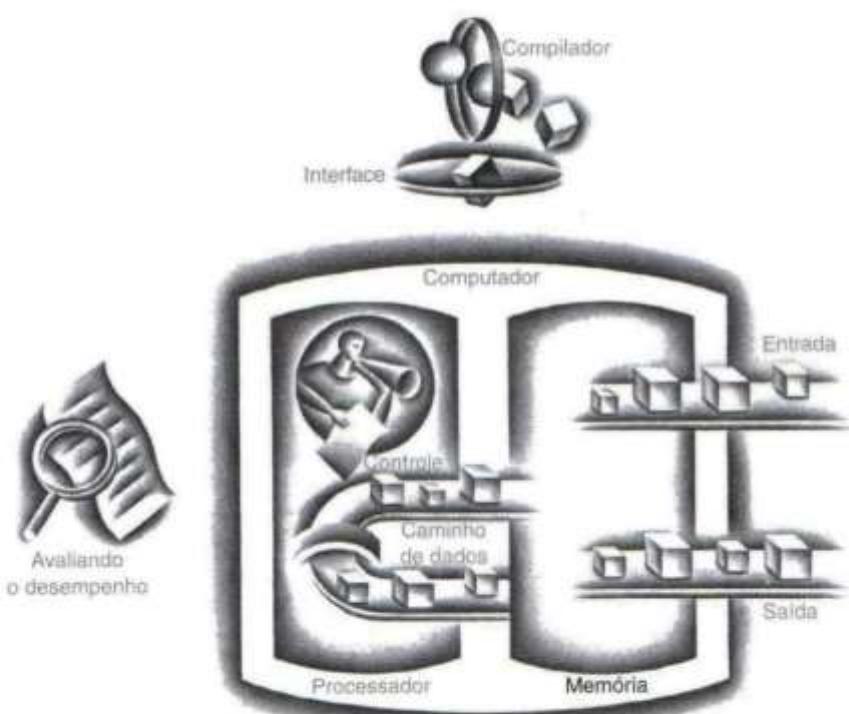
Armazenamento, Redes e Outros Periféricos

*Combinar largura de banda e armazenamento...
permite acesso veloz e confiável às trovas
de conteúdo em expansão nos discos e...
repositórios que se proliferam na Internet.*

George Gilder
The End is Drawing Nigh, 2000

- 8.1 Introdução 428**
- 8.2 Armazenamento em disco e confiabilidade 430**
- 8.3 Redes 439**
- 8.4 Barramentos e outras conexões entre processadores, memória e dispositivos de E/S 439**
- 8.5 Interface dos dispositivos de E/S com processador, memória e sistema operacional 445**
- 8.6 Medidas de desempenho de E/S: exemplos de sistemas de disco e de arquivos 452**
- 8.7 Projetando um sistema de E/S 455**
- 8.8 Vida real: uma câmera digital 457**
- 8.9 Falácia e armadilhas 460**
- 8.10 Comentários finais 463**
- 8.11 Perspectiva histórica e leitura adicional 463**
- 8.12 Exercícios 464**
-

Os cinco componentes clássicos de um computador



8.1

Introdução

Embora os usuários possam se frustrar se seus computadores travarem e tiverem de ser reinicializados, eles ficam irados se seu sistema de armazenamento falhar e informações forem perdidas. Assim, a barra para a confiabilidade é muito mais alta para o armazenamento do que para a computação. As redes também são planejadas para tratar falhas na comunicação, incluindo diversos mecanismos para detectar e recuperar-se de tais falhas. Logo, os sistemas de E/S geralmente colocam muito mais ênfase sobre a confiabilidade e custo, enquanto os processadores e a memória focalizam o desempenho e o custo.

Os sistemas de E/S também precisam planejar a facilidade de expansão e a diversidade de dispositivos, o que não é um problema para os processadores. A facilidade de expansão está relacionada à capacidade de armazenamento, que é outro parâmetro de projeto para os sistemas de E/S; os sistemas podem precisar de um limite inferior de capacidade de armazenamento para cumprir seu papel.

Embora o desempenho tenha um papel secundário para E/S, ele é mais complexo. Por exemplo, com alguns dispositivos, precisamos cuidar principalmente da latência de acesso, enquanto em outros a vazão é fundamental. Além do mais, o desempenho depende de muitos aspectos do sistema, da hierarquia de memória e do sistema operacional. A Figura 8.1 mostra a estrutura de um sistema simples com sua E/S. Todos os componentes, dos dispositivos de E/S individuais ao processador e software de sistemas, afetam a confiabilidade, a facilidade de expansão e o desempenho de tarefas que incluem E/S.

Os dispositivos de E/S são incrivelmente diversificados. Três características são úteis na organização dessa grande variedade:

- *Comportamento*: entrada (somente leitura), saída (somente escrita, não pode ser lido) ou armazenamento (pode ser relido e normalmente reescrito).
- *Parceria*: um humano ou uma máquina está na outra extremidade do dispositivo de E/S, seja alimentando a entrada de dados ou lendo-os na saída.

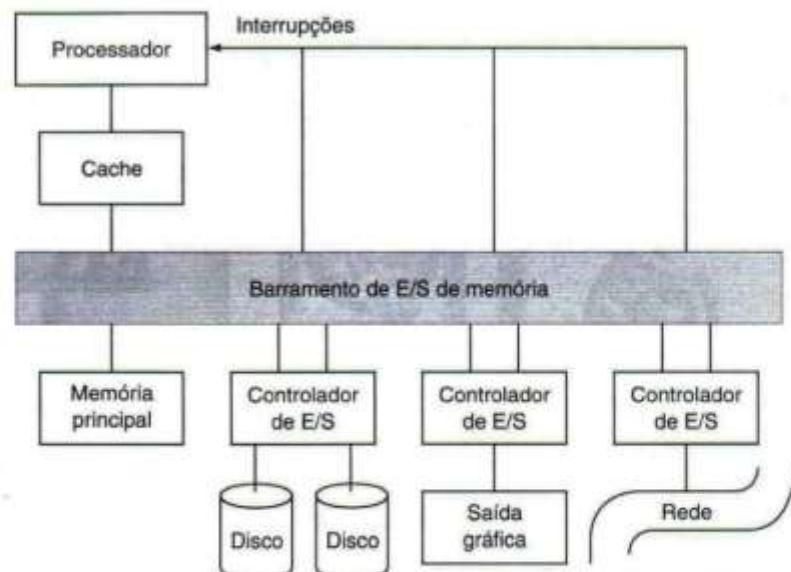


FIGURA 8.1 Uma coleção típica de dispositivos de E/S. As conexões entre os dispositivos de E/S, processador e memória normalmente são chamadas de *barramentos*. A comunicação entre os dispositivos e o processador utiliza interrupções e protocolos no barramento, conforme veremos neste capítulo. A Figura 8.11 mostra a organização para um PC desktop.

- **Taxa de dados:** a taxa de pico em que os dados podem ser transferidos entre o dispositivo de E/S e a memória principal ou processador. É útil saber qual é a demanda máxima que o dispositivo pode gerar.

Por exemplo, um teclado é um dispositivo de *entrada* usado por um *humano* com uma *taxa de dados máxima* de 10 bytes por segundo. A Figura 8.2 mostra alguns dos dispositivos de E/S conectados aos computadores.

Dispositivo	Comportamento	Parceiro	Taxa de dados (Mbit/seg)
Teclado	Entrada	humano	0,0001
Mouse	entrada	humano	0,0038
Entrada de voz	entrada	humano	0,2640
Entrada de som	entrada	máquina	3,0000
Scanner	entrada	humano	3,2000
Saída de voz	saída	humano	0,2640
Saída de som	saída	humano	8,0000
Impressora a laser	saída	humano	3,2000
Monitor gráfico	saída	humano	800,0000-8000,0000
Modem	Entrada ou saída	máquina	0,0160-0,0640
Rede/LAN	Entrada ou saída	máquina	100,0000-1000,0000
Rede/LAN sem fio	Entrada ou saída	máquina	11,0000-54,0000
Disco óptico	Armazenamento	máquina	80,0000
Fita magnética	Armazenamento	máquina	32,0000
Disco magnético	Armazenamento	máquina	240,0000-2560,0000

FIGURA 8.2 A diversidade de dispositivos de E/S. Os dispositivos de E/S podem ser distinguidos analisando se servem como dispositivos de entrada, saída ou armazenamento; seu parceiro de comunicação (pessoas ou outros computadores); e suas taxas de comunicação de pico. As taxas de dados se espalham por oito ordens de grandeza. Observe que uma rede pode ser um dispositivo de entrada ou saída, mas não pode ser usada para armazenamento. As taxas de transferência para os dispositivos sempre são indicadas na base 10, de modo que 10 Mbits/seg = 10.000.000 bits/seg.

No Capítulo 1, discutimos rapidamente sobre quatro dispositivos de E/S importantes e característicos: mouses, monitores gráficos, discos e redes. Neste capítulo, vamos nos aprofundar no armazenamento em disco e nas redes.

O modo como devemos avaliar o desempenho da E/S normalmente depende da aplicação. Em alguns ambientes, podemos nos importar principalmente com a vazão do sistema. Nesses casos, a largura de banda de E/S será mais importante. Até mesmo a largura de banda de E/S pode ser medida de duas maneiras diferentes:

1. Quantos dados podemos mover pelo sistema em determinado momento?
2. Quantas operações de E/S podemos realizar por unidade de tempo?

A decisão sobre a melhor medida de desempenho pode depender do ambiente. Por exemplo, em muitas aplicações de multimídia, a maioria das requisições de E/S é para fluxos de dados longos, e a largura de banda de transferência é a característica importante. Em outro ambiente, podemos querer processar um número maior de acessos pequenos e não relacionados a um dispositivo de E/S. Um exemplo desse ambiente poderia ser um escritório de processamento de impostos do National Income Tax Service (NITS). O NITS cuida principalmente do processamento de uma grande quantidade de formulários em determinado momento; cada formulário de imposto é armazenado separadamente e é muito pequeno. Um sistema orientado para transferência de arquivos grandes pode ser satisfatório, mas um sistema de E/S que possa admitir a transferência simultânea de muitos arquivos pequenos pode ser mais barato e mais rápido para processar milhões de formulários de imposto.

Requisições de E/S Leituras ou escritas em dispositivos de E/S.

Em outras aplicações, importamo-nos principalmente com o tempo de resposta, que você deve se lembrar que é o tempo total gasto para realizar uma tarefa em particular. Se as **requisições de E/S** forem extremamente grandes, o tempo de resposta dependerá muito da largura de banda, mas em muitos ambientes a maioria dos acessos será pequena, e o sistema de E/S com a menor latência por acesso oferecerá o melhor tempo de resposta. Em máquinas de monousuário, como computadores desktop e laptops, o tempo de resposta é a principal característica do desempenho.

Uma grande quantidade de aplicações, especialmente no vasto mercado comercial para a computação, exige alta vazão e pouco tempo de resposta. Alguns exemplos incluem caixas eletrônicos de banco, sistemas de entrada de pedidos e acompanhamento de estoque, servidores de arquivos e servidores Web. Nesses ambientes, preocupamo-nos com o tempo usado para cada tarefa e quantas tarefas podemos processar em um segundo. A quantidade de solicitações de caixas eletrônicos que você pode processar por hora não importa se cada uma exige 15 minutos – você ficará sem clientes! De modo semelhante, se você puder processar cada solicitação dos caixas eletrônicos rapidamente, mas só pode lidar com uma pequena quantidade de requisições ao mesmo tempo, não poderá dar suporte a muitos caixas eletrônicos, ou então o custo do computador por caixa eletrônico será muito alto.

Resumindo, as três classes, desktops, servidores e computadores embutidos são sensíveis à confiabilidade e ao custo da E/S. Sistemas de desktop e sistemas embutidos se concentram mais no tempo de resposta e na diversidade dos dispositivos de E/S, enquanto outros sistemas focalizam mais a vazão e a facilidade de expansão dos dispositivos de E/S.

8.2

Armazenamento em disco e confiabilidade

não-volátil Dispositivo de armazenamento em que os dados retêm seu valor mesmo quando a alimentação é removida.

trilha Um dos milhares de círculos concêntricos que compõem a superfície de um disco magnético.

setor Um dos segmentos que compõem uma trilha em um disco magnético; um setor é a menor quantidade de informação lida ou escrita em um disco.

seek O processo de posicionar uma cabeça de leitura/gravação na trilha correta de um disco.

Como mencionamos no Capítulo 1, os discos magnéticos contam com um prato giratório coberto por uma superfície magnética e utiliza uma cabeça de leitura/escrita móvel para acessar o disco. O armazenamento em disco é **não-volátil** – os dados permanecem mesmo quando a alimentação é removida. Um disco magnético consiste em uma coleção de pratos (1-4), cada qual com duas superfícies de disco graváveis. A pilha de pratos gira a uma velocidade entre 5.400 a 15.000RPM e tem um diâmetro entre 2,5cm e 9cm. Cada superfície do disco é dividida em círculos concêntricos, chamados **trilhas**. Normalmente, existem de 10.000 a 50.000 trilhas por superfície. Cada trilha, por sua vez, é dividida em **setores** que contêm as informações; cada trilha pode ter de 100 a 500 setores. Os setores normalmente possuem 512 bytes de tamanho, embora exista uma iniciativa para aumentar o tamanho do setor para 4.096 bytes. A sequência gravada em mídia magnética é um número de setor, um gap, a informação para esse setor incluindo o código de correção de erro (ver Apêndice B), um gap, o número de setor do próximo setor, e assim por diante. Originalmente, todas as trilhas tinham o mesmo número de setores e, portanto, o mesmo número de bits, mas com a introdução da ZBR (Zone Bit Recording – registro de bits por zona) no início da década de 1990, as unidades de disco passaram para um número variável de setores (portanto, bits) por trilha, em vez de manter constante o espaçamento entre os bits. O ZBR aumenta o número de bits nas trilhas externas e, assim, aumenta a capacidade da unidade.

Como vimos no Capítulo 1, para ler e escrever informações, as cabeças de leitura/escrita precisam ser movidas de modo que estejam sobre o local correto. As cabeças de disco para cada superfície são conectadas e se movem em conjunto, de modo que cada cabeça esteja sobre a mesma trilha de cada superfície. O termo **cilindro** é usado para se referir a todas as trilhas sob as cabeças em determinado ponto para todas as superfícies.

Para acessar dados, o sistema operacional precisa direcionar o disco por um processo em três estágios. O primeiro passo é posicionar a cabeça sobre a trilha apropriada. Essa operação é chamada **seek**, e o tempo para mover a cabeça até a trilha apropriada é chamado *tempo de seek*.

Os fabricantes de disco informam o tempo de seek mínimo, o tempo de seek máximo e o tempo de seek médio em seus manuais. Os dois primeiros são fáceis de medir, mas a média está aberta a interpretações, pois ela depende da distância do seek. Os fabricantes decidiram calcular o tempo de seek médio como a soma do tempo para todos os seeks possíveis dividido pelo número de seeks possíveis. Os tempos de seek médios normalmente são anunciados como entre 3ms a 14ms, mas, dependendo da aplicação e do escalonamento das requisições de disco, o tempo de seek médio real pode ser de apenas 25% a 33% do número anunciado, devido à localidade das referências de disco. Essa localidade surge tanto por causa de acessos sucessivos ao mesmo arquivo quanto porque o sistema operacional tenta escalarizar esses acessos juntos.

Quando a cabeça tiver atingido a trilha correta, temos de esperar até o setor desejado girar sob a cabeça de leitura/escrita. Esse tempo é chamado de **latência rotacional** ou **atraso rotacional**. A latência média para a informação desejada está a meio caminho ao redor do disco. Como os discos giram entre 5.400RPM a 15.000RPM, a latência rotacional média é entre

$$\text{Latência rotacional média} = \frac{0,5 \text{ rotação}}{5.400 \text{ RPM}} = \frac{0,5 \text{ rotação}}{5.400 \text{ RPM} / \left(60 \frac{\text{seg}}{\text{min}} \right)}$$

$$= 0,0056 \text{ segundos} = 5,6\text{ms}$$

e

$$\text{Latência rotacional média} = \frac{0,5 \text{ rotação}}{15.000 \text{ RPM}} = \frac{0,5 \text{ rotação}}{15.000 \text{ RPM} / \left(60 \frac{\text{seg}}{\text{min}} \right)}$$

$$= 0,0020 \text{ segundos} = 2,0\text{ms}$$

O último componente de um acesso ao disco, o *tempo de transferência*, é o tempo para transferir um bloco de bits. O tempo de transferência é uma função do tamanho do setor, da velocidade de rotação e da densidade de gravação de uma trilha. As taxas de transferência quando este livro foi escrito estavam entre 30 e 80MB/seg. A única complicação é que a maioria dos controladores de disco possui uma cache interna que armazena setores enquanto eles passam; as taxas de transferência da cache normalmente são maiores e poderiam chegar até 320MB/seg em 2004. Hoje, a maioria das transferências de disco possui o tamanho de múltiplos setores.

Uma *controladora de discos* normalmente trata do controle detalhado do disco e da transferência entre o disco e a memória. A controladora acrescenta o componente final do tempo de acesso ao disco, o *tempo da controladora*, que é o overhead que a controladora impõe na realização do acesso de E/S. O tempo médio para realizar uma operação de E/S consistirá nesses quatro tempos mais qualquer espera que ocorra porque outros processos estão utilizando o disco.

TEMPO DE LEITURA DO DISCO

Qual é o tempo médio para ler ou escrever um setor de 512 bytes em um disco típico girando a 10.000RPM? O tempo de seek médio anunciado é de 6ms, a taxa de transferência é de 50MB/seg e o overhead da controladora é de 0,2ms. Suponha que o disco esteja ocioso, de modo que não existe um tempo de espera.

O tempo médio de acesso ao disco é igual ao Tempo médio de seek + Atraso rotacional médio + Tempo de transferência + Overhead da controladora. Usando o tempo de seek médio anunciado, a resposta é

latência rotacional
Também chamada de atraso. O tempo exigido para que o setor desejado de um disco gire sob a cabeça de leitura/escrita; normalmente considerado metade do tempo de rotação.

EXEMPLO

RESPOSTA

$$6,0\text{ms} + \frac{0,5 \text{ rotação}}{10.000 \text{ RPM}} = \frac{0,5 \text{ KB}}{50\text{MB/sec}} + 0,2\text{ms} = 6,0 + 3,0 + 0,01 + 0,2 = 9,2\text{ms}$$

Se o tempo médio de seek medido for 25% do tempo médio anunciado, a resposta é

$$1,5\text{ms} + 3,0\text{ms} + 0,01\text{ms} + 0,2\text{ms} = 4,7\text{ms}$$

Observe que, quando consideramos o tempo médio de seek medido, ao contrário do tempo médio de seek anunciado, a latência rotacional pode ser o maior componente do tempo de acesso.

As densidades de disco têm continuado a aumentar há mais de 50 anos. O impacto dessa melhoria na densidade e na redução do tamanho físico de uma unidade de disco tem sido incríveis, como mostra a Figura 8.3. Os objetivos de diferentes projetistas de discos têm levado a uma grande variedade de unidades disponíveis em determinado momento. A Figura 8.4 mostra as características de três discos magnéticos. Na época do lançamento deste livro, esses discos de um único fabricante custavam entre US\$0,50 e US\$5 por gigabyte, dependendo do tamanho, da interface e do desempenho. A menor unidade tem vantagens em consumo e volume por byte.

Detalhamento: a maioria das controladoras de disco inclui caches. Essas caches permitem um acesso rápido aos dados lidos recentemente entre transferências solicitadas pela CPU. Elas utilizam write-through e não atualizam quando há falha na escrita. Elas normalmente também incluem algoritmos de prefetch para tentar antecipar a demanda. Naturalmente, essas capacidades complicam a medida de desempenho do disco e aumentam a importância da escolha do workload.

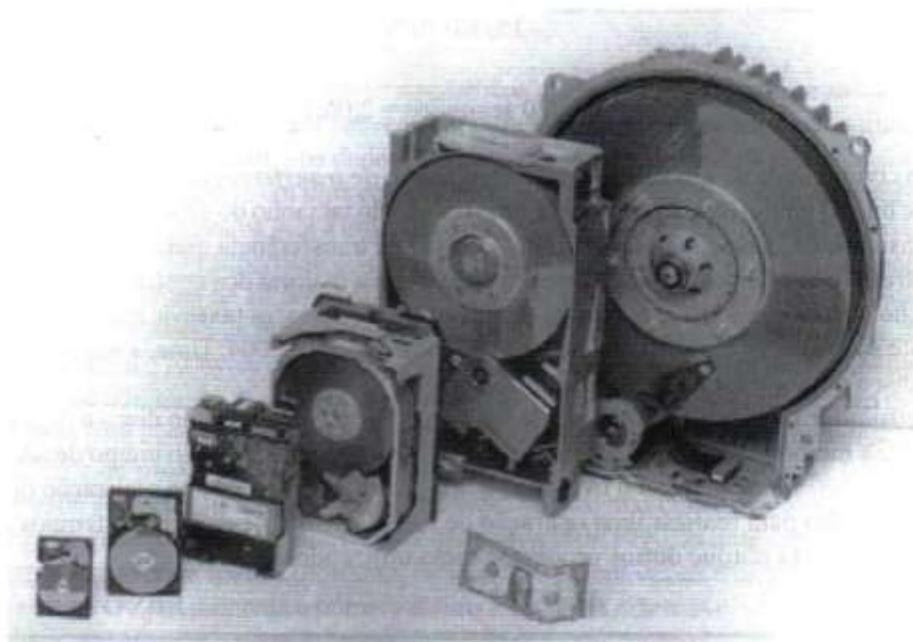


FIGURA 8.3 Seis discos magnéticos, variando em diâmetro de 35cm até 4,5cm. O microdrive IBM, que não aparece, possui um diâmetro de 2,5cm. Os discos da figura foram introduzidos há mais de 15 anos e, portanto, não representam a melhor capacidade dos discos modernos desses mesmos diâmetros. Contudo, essa fotografia representa com precisão seus tamanhos físicos relativos. O maior dos discos é o DEC R81, contendo quatro pratos de 35,5cm de diâmetro e armazenando 456MB. Ele foi fabricado em 1985. O disco com diâmetro de 20cm vem da Fujitsu, e esse disco de 1984 armazena 130MB em seis pratos. O Micropolis RD53 possui cinco pratos de 13,3cm e armazena 85MB. O IBM 0361 também possui cinco pratos, mas possuem apenas 8,8cm de diâmetro. Esse disco de 1988 tem 320MB de capacidade. Em 2004, o disco de 8,8cm mais denso tinha 2 pratos e tinha 200GB no mesmo espaço, ocasionando um aumento de densidade de aproximadamente 600 vezes! O Comer CP 2045 possui dois pratos de 6,35cm, contendo 40MB, e foi fabricado em 1990. O menor disco desta fotografia é o Integral 1820. Esse disco de um único prato de 4,5cm contém 20MB e foi fabricado em 1992.

Características	Seagate ST373453	Seagate ST3200822	Seagate ST94811A
Diâmetro do disco (cm)	8,89	8,89	6,35
Capacidade do disco formatado (GB)	73,4	200,0	40,0
Número de superfícies de disco (cabeças)	8	4	2
Velocidade de rotação (RPM)	15.000	7.200	5.400
Tamanho da cache de disco interna (MB)	8	8	8
Interface externa, largura de banda (MB/seg)	Ultra320 SCSI, 320	Serial ATA, 150	ATA, 100
Taxa de transferência sustentada (MB/seg)	57,86	32,58	34
Tempo de seek mínimo (leitura/escrita) (ms)	0,2/0,4	1,0/1,2	1,5/2,0
Tempo médio de seek para leitura/escrita (ms)	3,6/3,9	8,5/9,5	12,0/14,0
Tempo médio para falha (MTTF) (horas)	1.200.000 a 25°C	600.000 a 25°C	330.000 a 25°C
Garantia (anos)	5	3	-
Erros de leitura não recuperáveis por bits lidos	<1 por 10 ¹⁵	<1 por 10 ¹⁴	<1 por 10 ¹⁴
Temperatura, limites de vibração (operando)	5°-55°C, 400Hz @ 0,5 G	0°-60°C, 350Hz @ 0,5 G	5°-55°C, 400Hz @ 1 G
Tamanho: dimensões (cm), peso (gramas)	2,5 x 10,1 x 14,7, 861,8g	2,5 x 10,1 x 14,7, 635g	1,0 x 6,8 x 9,9, 90,7g
Potência: operando/ocioso/standby (watts)	20/12/-	12/8/1	2,4/1,0/0,4
GB/pol. cúb., GB/watt	3 GB/pol. cúb., 4 GB/W	9 GB/pol. cúb., 16 GB/W	10 GB/pol. cúb., 17 GB/W
Preço em 2004, \$/GB	US\$400, US\$5/GB	US\$100, US\$0,5/GB	US\$100, US\$2,50/GB

FIGURA 8.4 Características de três discos magnéticos de um único fabricante em 2004. Os discos mostrados aqui possuem ou interface SCSI, um barramento de E/S padrão para muitos sistemas, ou ATA, um barramento de E/S padrão para PCs. O primeiro disco é usado para servidores de arquivos, o segundo para PCs de desktop e o último para laptops. Cada disco possui uma cache de 8MB. A taxa de transferência da cache é de 3-6 vezes mais rápida do que a taxa de transferência da superfície do disco. O custo muito mais baixo da unidade de 8,8cm ATA ocorre principalmente devido ao mercado hipercompetitivo dos PCs, embora existam diferenças em desempenho e confiabilidade entre ele e a unidade SCSI. A vida útil para esses discos é de 5 anos, embora a Seagate ofereça uma garantia de 5 anos apenas na unidade. Observe que o MTTF cotado considera potência e temperatura normais. Os tempos de vida do disco podem ser muito mais curtos se a temperatura e a vibração não forem controlados. Veja o link da Seagate em www.seagate.com para obter mais informações sobre essas unidades.

Confiança, confiabilidade e disponibilidade

Os usuários imploram por armazenamento confiável, mas como podemos definir isso? No setor de computação, isso é mais difícil do que consultar o dicionário. Após um considerável debate, a definição considerada padrão é a seguinte (Laprie, 1985):

Confiança de um sistema computacional é a qualidade do serviço entregue de modo que a confiança possa ser justificadamente depositada sobre esse serviço. O serviço entregue por um sistema é o seu comportamento real observado como percebido por outro(s) sistema(s) interagindo com os usuários desse sistema. Cada módulo possui um comportamento especificado ideal, no qual uma especificação de serviço é uma descrição combinada do comportamento esperado. Uma falha do sistema ocorre quando o comportamento real se desvia do comportamento especificado.

Assim, você precisa que uma especificação de referência do comportamento esperado seja capaz de determinar a confiança. Os usuários podem, então, ver um sistema alternando entre dois estados de serviço fornecido com relação à especificação do serviço:

1. Realização do serviço, na qual o serviço é entregue conforme especificado
2. Interrupção do serviço, na qual o serviço entregue é diferente do serviço especificado

As transições do estado 1 para o estado 2 são causadas por falhas, e as transições do estado 2 para o estado 1 são causadas por restaurações. As falhas podem ser permanentes ou intermitentes. O último é o caso mais difícil de diagnosticar quando um sistema oscila entre os dois estados; as falhas permanentes são muito mais fáceis de diagnosticar. Essa definição ocasiona dois termos relacionados: confiabilidade e disponibilidade.

Interface SCSI (Small Computer Systems Interface) Um barramento utilizado como padrão para dispositivos de E/S.

Confiabilidade é uma medida da realização contínua do serviço – ou, de forma equivalente, do tempo para a falha – de um ponto de referência. Logo, o tempo médio para a falha (MTTF) dos discos na Figura 8.4 é uma medida de confiabilidade. A interrupção do serviço é medida como o tempo médio para o reparo (MTTR). O *tempo médio entre falhas* (MTBF) é simplesmente a soma MTTF + MTTR. Embora o MTBF seja muito utilizado, o MTTF normalmente é o termo mais apropriado.

Disponibilidade é uma medida da realização do serviço com relação à alternância entre os dois estados de realização e interrupção. A disponibilidade é quantificada estaticamente como

$$\text{Disponibilidade} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}$$

Observe que a confiabilidade e a disponibilidade são medidas quantificáveis, e não apenas sinônimos de confiança.

Qual é a causa das falhas? A Figura 8.5 resume muitos documentos que coletaram dados sobre motivos para falhas de sistemas computacionais e sistemas de telecomunicações. Logicamente, os operadores humanos são uma fonte de falhas significativa.

Operador	Software	Hardware	Sistema	Ano do dado coletado
42%	25%	18%	Centro de dados (Tandem)	1985
15%	55%	14%	Centro de dados (Tandem)	1989
18%	44%	39%	Centro de dados (DEC VAX)	1985
50%	20%	30%	Centro de dados (DEC VAX)	1993
50%	14%	19%	Rede telefônica pública dos EUA	1996
54%	7%	30%	Rede telefônica pública dos EUA	2000
60%	25%	15%	Serviços de Internet	2002

FIGURA 8.5 Resumo dos estudos dos motivos para falhas. Embora seja difícil coletar dados para determinar se os operadores são a causa dos erros, como os operadores normalmente registram os motivos para as falhas, esses estudos capturaram esses dados. Constantemente havia outras categorias, como motivos ambientais para cortes de energia, mas eles em geral eram pequenos. As duas linhas iniciais vêm de um artigo clássico de Jim Gray [1990], que ainda é muito citado, quase 20 anos após a coleta dos dados. As duas linhas seguintes são de um artigo de Murphy e Gent, que estudaram casos de cortes em sistemas VAX com o tempo ("Measuring system and software reliability using an automated data collection process", *Quality and Reliability Engineering International* 11:5, setembro–outubro de 1995, 341-53). As quinta e sexta linhas são estudos de dados de falhas do FCC sobre a rede telefônica pública dos Estados Unidos, por Kuhn ("Sources of failure in the public switched telephone network", *IEEE Computer* 30:4, abril de 1997, 31-36) e por Patty Enriquez. O estudo mais recente de três servidores de Internet vem de Oppenheimer, Ganapath e Patterson [2003].

Para aumentar o MTTF, você pode melhorar a qualidade dos componentes ou projetar sistemas para que continuem a operação na presença de componentes que falharam. Logo, a falha precisa ser definida em relação a um contexto. Uma falha em um componente pode não ocasionar uma falha do sistema. Para esclarecer essa distinção, o termo *falha* é usado para indicar falha de um componente. Aqui estão três maneiras de melhorar o MTTF:

1. *Impedimento de falha*: evitar a ocorrência da falta pela construção
2. *Tolerância a falhas*: uso de redundância para permitir que o serviço cumpra com a especificação de serviço apesar da ocorrência de falhas, o que se aplica principalmente a falhas do hardware
3. *Previsão de falha*: prever a presença e criação de falhas, o que se aplica a falhas do hardware e do software

Diminuir o MTTR pode ajudar na disponibilidade tanto quanto aumentar o MTTF. Por exemplo, ferramentas para detecção, diagnóstico e reparo de falhas podem ajudar a reduzir o tempo para reparar falhas ocasionadas por pessoas, software e hardware.

RAID

Aproveitar a redundância para melhorar a disponibilidade do armazenamento em disco é algo capturado na frase **Redundant Arrays of Inexpensive Disks**, abreviada como **RAID**. No momento em que o termo foi criado, a alternativa eram discos grandes e caros, como os maiores da Figura 8.3. O argumento era que, substituindo alguns discos grandes por muitos discos pequenos, o desempenho melhoraria porque haveria mais cabeças de leitura, e haveria vantagens no custo, consumo e espaço ocupado, pois discos menores são muito mais eficientes por gigabyte do que discos maiores. A redundância foi necessária porque muito mais discos menores tinham menor confiabilidade do que alguns poucos discos grandes.

Tendo muitos discos pequenos, o custo da redundância extra para melhorar a confiabilidade é pequeno em relação aos discos grandes. Assim, a confiabilidade era mais econômica se você construísse um array redundante de discos mais baratos. Em retrospecto, essa era a principal vantagem.

De quanta redundância você precisa? Você precisa de informações extras para encontrar as falhas? Importa como você organiza os dados e as informações de verificação extra nesses discos? O artigo que criou o termo deu uma resposta evolutiva a essas questões, começando com a solução mais simples, porém mais dispendiosa. A Figura 8.6 mostra a evolução e um exemplo de custo no número de discos de verificação extras. Para acompanhar a evolução, os autores numeraram os estágios do RAID, e eles ainda são usados hoje.

RAID (Redundant Arrays of Inexpensive Disks) Uma organização de discos que usa um array de discos pequenos e baratos para aumentar o desempenho e a confiabilidade.

Nenhuma redundância (RAID 0)

O simples espalhamento dos dados por vários discos, chamado **striping**, força automaticamente os acessos a vários discos. O striping por um conjunto de discos faz com que a coleção apareça ao soft-

striping: Alocação de blocos logicamente seqüenciais por discos separados para permitir maior desempenho do que um único disco pode oferecer.

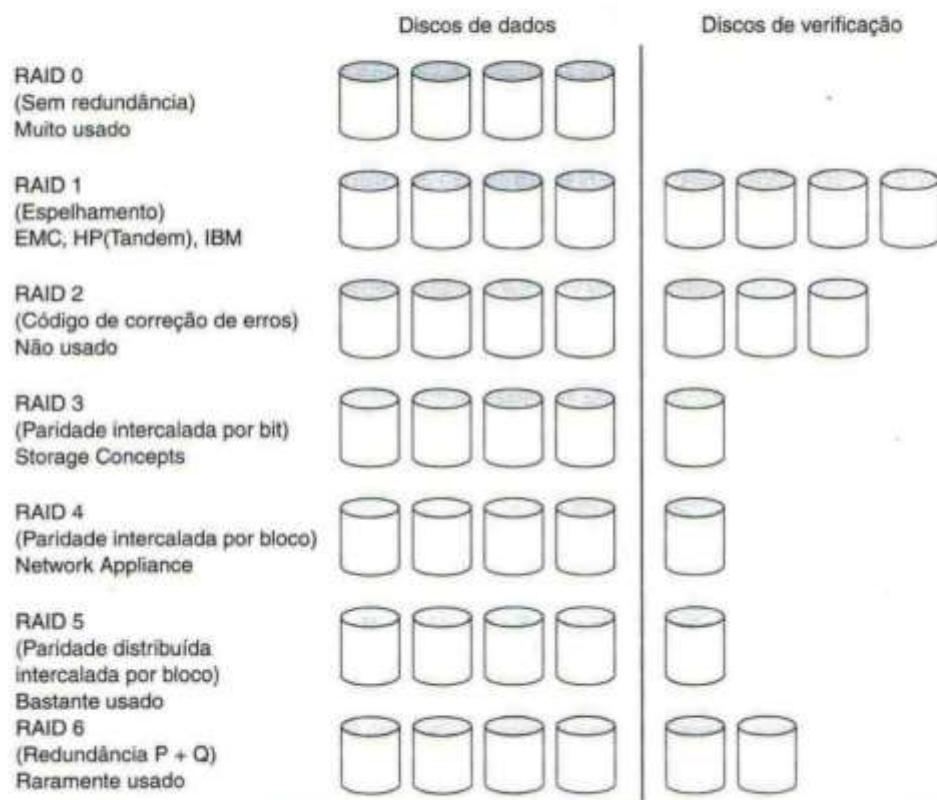


FIGURA 8.6 RAID para um exemplo de quatro discos de dados, mostrando discos de verificação extras por nível de RAID e empresas que utilizam cada nível. As Figuras 8.7 e 8.8 explicam a diferença entre RAID 3, RAID 4 e RAID 5.

ware como um único disco grande, que simplifica o gerenciamento do armazenamento. Isso também melhora o desempenho para acessos grandes, pois muitos discos podem operar ao mesmo tempo. Os sistemas de edição de vídeo, por exemplo, normalmente repartem seus dados e podem não se preocupar com a confiabilidade tanto quanto, digamos, os bancos de dados.

RAID 0 é um nome errado, pois não existe redundância. Entretanto, os níveis de RAID normalmente são deixados para o operador definir ao criar um sistema de armazenamento, e RAID 0 normalmente está listado como uma das opções. Logo, o termo *RAID 0* tornou-se muito utilizado.

Espelhamento (RAID 1)

espelhamento Escrever dados idênticos em vários discos, para aumentar a disponibilidade dos dados.

Esse esquema tradicional para tolerar falhas de disco, chamado **espelhamento** ou *shadowing*, utiliza o dobro da quantidade de discos do RAID 0. Sempre que os dados são gravados em um disco, esses dados também são gravados em um disco redundante, de modo que sempre existem duas cópias da informação. Se um disco falhar, o sistema simplesmente vai ao “espelho” e lê seu conteúdo para obter a informação desejada. O espelhamento é a solução de RAID mais dispendiosa, pois exige mais discos.

Código de detecção e correção de erros (RAID 2)

RAID 2 utiliza um esquema de detecção e correção de erros que é mais utilizado para memórias (ver **Apêndice B**). Como RAID 2 caiu em desuso, não iremos descrevê-lo aqui.

Paridade intercalada por bit (RAID 3)

grupo de proteção
O grupo de discos de dados ou blocos que compartilham um disco ou bloco de verificação comum.

O custo da disponibilidade mais alta pode ser reduzido para $1/N$, onde N é o número de discos em um **grupo de proteção**. Em vez de ter uma cópia completa dos dados originais para cada disco, só precisamos acrescentar informações redundantes suficientes para restaurar a informação perdida em uma falha. Leituras ou escritas vão para todos os discos no grupo, com um disco extra para manter as informações de verificação caso haja uma falha. RAID 3 é popular em aplicações com grandes conjuntos de dados, como multimídia e alguns códigos científicos.

Paridade é um esquema desse tipo. Os leitores não acostumados com a paridade podem pensar no disco redundante como aquele com a soma de todos os dados dos outros discos. Quando um disco falha, então você subtrai todos os dados nos discos bons do disco de paridade; a informação restante deverá ser a informação que falta. A paridade é simplesmente a soma módulo dois.

Diferente de RAID 1, muitos discos precisam ser lidos para determinar os dados que faltam. A disposição por trás dessa técnica é a de que levar mais tempo para recuperar-se de uma falha, mas gastar menos com armazenamento redundante, é uma boa escolha.

Paridade intercalada por bloco (RAID 4)

RAID 4 usa a mesma razão de discos de dados e discos de verificação do RAID 3, mas eles acessam dados de formas diferentes. A paridade é armazenada como blocos e associada a um conjunto de blocos de dados.

Em RAID 3, cada acesso ia para todos os discos. Contudo, algumas aplicações preferem acessos menores, permitindo que acessos independentes ocorram em paralelo. Essa é a finalidade do RAID níveis 4 a 6. Como a informação de detecção de erro em cada setor é verificada nas leituras para ver se os dados estão corretos, essas “leituras pequenas” a cada disco podem ocorrer de forma independente, desde que o acesso mínimo seja de um setor. No contexto do RAID, um acesso pequeno vai para apenas um disco em um grupo de proteção, enquanto um acesso grande vai para todos os discos em um grupo de proteção.

As escritas são outro problema. Pode parecer que cada escrita pequena exigiria que todos os outros discos fossem acessados para ler o restante das informações necessárias para recalcular a nova

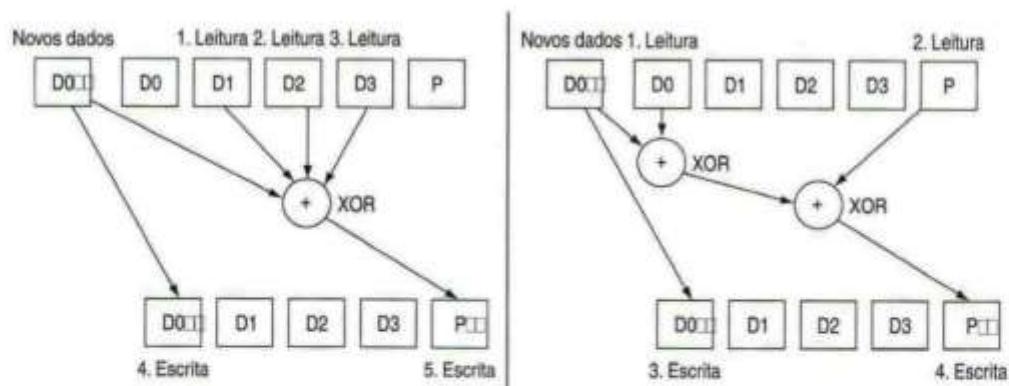


FIGURA 8.7 Pequena atualização de escrita em RAID 3 versus RAID 4. Essa otimização para pequenas escritas reduz a quantidade de acessos ao disco, bem como a quantidade de discos ocupados. Essa figura considera que temos quatro blocos de dados e um bloco de paridade. O cálculo de paridade do RAID 3 direto à esquerda da figura lê os blocos D1, D2 e D3 antes de acrescentar o bloco D0' para calcular a nova paridade P'. (Caso você esteja questionando, os novos dados D0' vêm diretamente da CPU, de modo que os discos não estão envolvidos na sua leitura.) O atalho RAID 4 à direita lê o valor antigo D0' e o compara com o novo valor D0' para ver quais bits mudaram. Depois, você lê a paridade antiga P e depois muda os bits correspondentes para formar P'. A função lógica OR exclusivo faz exatamente o que queremos. Esse exemplo substitui três leituras de disco (D1, D2, D3) e duas escritas (D0', P') envolvendo todos os discos para duas leituras de disco (D0', P') e duas escritas de disco (D0', P'), que envolvem apenas dois discos. Aumentar o tamanho do grupo de paridade aumenta as economias do atalho. RAID 5 utiliza o mesmo atalho.

paridade, como na Figura 8.7. Uma “escrita pequena” exigiria a leitura dos dados antigos e da paridade antiga, o acréscimo das novas informações e depois a escrita da nova paridade no disco de paridade e os novos dados no disco de dados.

A idéia principal para reduzir esse overhead é que a paridade é simplesmente uma soma de informações; observando quais bits mudam quando escrevemos as novas informações, só precisamos mudar os bits correspondentes no disco de paridade. A Figura 8.7 mostra o atalho. Temos de ler os dados antigos do disco sendo escrito, comparar os dados antigos com os novos para ver quais bits mudam, ler a paridade antiga, alterar os bits correspondentes, depois escrever os novos dados e a nova paridade. Assim, a pequena escrita envolve quatro acessos de disco a dois discos, em vez de acessar todos os discos. Essa organização é RAID 4.

Paridade distribuída intercalada por bloco (RAID 5)

RAID 4 aceita de forma eficiente uma mistura de leituras grandes, escritas grandes e leituras pequenas, e também permite escritas pequenas. Uma desvantagem para o sistema é que o disco de paridade precisa ser atualizado em cada escrita, de modo que o disco de paridade é o gargalo para escritas back-to-back.

Para resolver o gargalo da escrita de paridade, a informação de paridade pode ser espalhada por todos os discos, de modo que não haja um único gargalo para escritas. A organização da paridade distribuída é RAID 5.

A Figura 8.8 mostra como os dados são distribuídos no RAID 4 versus RAID 5. Como vemos na organização da direita, em RAID 5, a paridade associada a cada linha de blocos de dados não é mais restrita a um único disco. Essa organização permite que várias escritas ocorram simultaneamente, desde que os blocos de paridade não estejam localizados no mesmo disco. Por exemplo, uma escrita no bloco 8 à direita também precisa acessar seu bloco de paridade P2, ocupando assim o primeiro e terceiro discos. Uma segunda escrita no bloco 5, à direita, implicando uma atualização no seu bloco de paridade P1, acessa o segundo e quarto discos e, assim, poderia ocorrer simultaneamente com a escrita no bloco 8. Essas mesmas escritas na organização à esquerda resultam em mudanças nos blocos P1 e P2, ambas no quinto disco, que é um gargalo.

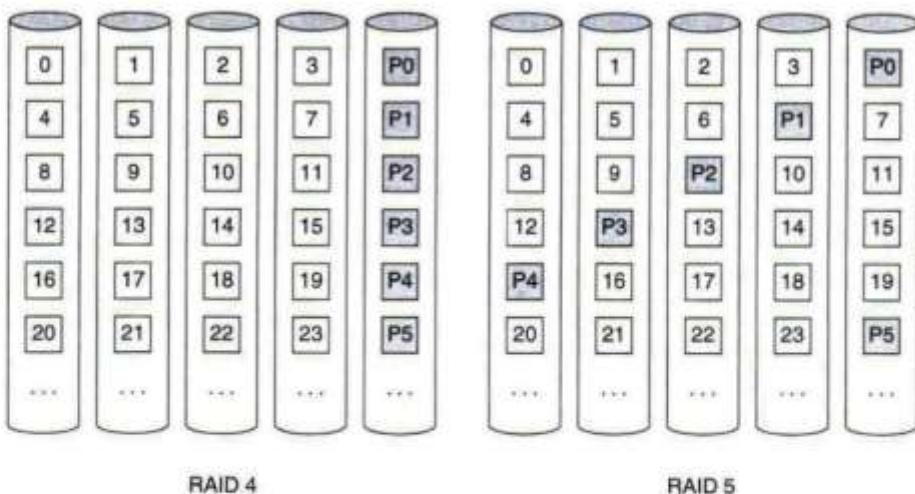


FIGURA 8.8 Paridade Intercalada por bloco (RAID 4) versus paridade distribuída intercalada por bloco (RAID 5). Distribuindo os blocos de paridade a todos os discos, algumas escritas pequenas podem ser realizadas em paralelo.

Redundância P + Q (RAID 6)

Os esquemas baseados em paridade protegem contra uma única falha auto-identificável. Quando uma correção de única falha não é suficiente, a paridade pode ser generalizada para ter um segundo cálculo sobre os dados e outro disco de verificação de informações. Esse segundo bloco de verificação permite a recuperação de uma segunda falha. Assim, o overhead do armazenamento é o dobro daquele do RAID 5. O atalho de escrita pequena da Figura 8.7 também funciona, exceto que agora existem seis acessos a disco, em vez de quatro para atualizar as informações de P e Q.

Resumo de RAID

hot swapping

Substituição de um componente de hardware enquanto o sistema está em execução.

reservas em standby

Recursos de hardware de reserva que podem imediatamente tomar o lugar de um componente defeituoso.

RAID 1 e RAID 5 são bastante utilizados em servidores; uma estimativa é de que 80% de discos nos servidores se encontrem em algum sistema RAID.

Um ponto fraco dos sistemas RAID é o reparo. Primeiro, para evitar tornar os dados indisponíveis durante o reparo, o array precisa ser designado para permitir que os discos que falharam sejam substituídos sem ter de desligar o sistema. RAIDs possuem redundância suficiente para permitir a operação continua, mas o **hot swapping** de discos impõe demandas sobre o projeto físico e elétrico do array e as interfaces de disco. Segundo, outra falha poderia ocorrer durante o reparo, de modo que o tempo de reparo afeta as chances de perder dados: quanto maior for o tempo de reparo, maiores as chances de outra falha que causará perda de dados. Em vez de ter de esperar que o operador traga um disco bom, alguns sistemas incluem **reservas em standby**, de modo que os dados podem ser reconstruídos imediatamente na descoberta da falha. O operador pode, então, substituir os discos que falharam sem tanta pressa. Em terceiro lugar, embora os fabricantes de disco indiquem um MTTF muito alto para seus produtos, esses números estão sob condições nominais. Se determinado array de disco estiver sujeito a ciclos de temperatura devido, digamos, a falhas no sistema de ar-condicionado, ou a sacudidas devido a projeto, construção ou instalação imperfeita do rack, as taxas de falha serão muito maiores. O cálculo da confiabilidade do RAID considera independência entre falhas de disco, mas as falhas de disco poderiam estar co-relacionadas, pois tal dano devido ao ambiente provavelmente aconteceria em todos os discos no array. Finalmente, um operador humano, por fim, determina quais discos devem ser removidos. Como mostra a Figura 8.5, os operadores são apenas humanos, de modo que ocasionalmente poderão remover um disco bom no lugar do disco com defeito, ocasionando uma falha de disco irrecuperável.

Embora o RAID 6 raramente seja usado hoje, um operador cuidadoso poderá querer sua redundância extra para proteger contra falhas de hardware esperadas e mais uma margem de segurança, para proteger contra erro humano e falhas relacionadas a problemas com o ambiente.

Quais das seguintes afirmações são verdadeiras sobre a confiança?

**Verifique
você mesmo**

1. Se um sistema estiver ativo, então todos os seus componentes estão realizando seu serviço esperado.
2. A disponibilidade é uma medida quantitativa da porcentagem de tempo que um sistema está realizando seu serviço esperado.
3. A confiabilidade é uma medida quantitativa da realização continua do serviço por um sistema.
4. A principal fonte de interrupções hoje é o software.

Quais das seguintes afirmações são verdadeiras sobre os níveis RAID 1, 3, 4, 5 e 6?

1. Os sistemas RAID contam com a redundância para conseguir a alta disponibilidade.
2. RAID 1 (espelhamento) possui o mais alto overhead de disco de verificação.
3. Para pequenas escritas, RAID 3 (paridade intercalada por bit) possui a pior vazão.
4. Para grandes escritas, RAID 3, 4 e 5 possuem a mesma vazão.

Detalhamento: uma questão é como o espelhamento interage com o striping. Suponha que você tivesse, digamos, quatro discos de dados para armazenar e oito discos físicos para usar. Você criaria quatro pares de discos – cada um organizado como RAID 1 – e depois faria striping dos dados nos quatro pares RAID 1? Como alternativa, você criaria dois conjuntos de quatro discos – cada um organizado como RAID 0 – e depois espelharia as escritas nos dois conjuntos RAID 0? A terminologia RAID evoluiu para chamar o primeiro de RAID 1 + 0, ou RAID 10 ("espelhos com striping") e o segundo de RAID 0 + 1 ou RAID 1 ("striping espelhado").

8.3**Redes**

As redes estão ganhando mais popularidade com o passar do tempo e, diferente de outros dispositivos de E/S, existem muitos livros e cursos sobre elas. Para os leitores que não fizeram nenhum curso nem leram livros sobre redes, a Seção 8.3, no CD, oferece uma visão geral dos tópicos e da terminologia, incluindo interligação de redes, o modelo OSI, famílias de protocolos, como TCP/IP, redes de longa distância, como ATM, redes locais, como Ethernet, e redes sem fio, como IEEE 802.11.

8.4**Barramentos e outras conexões entre processadores, memória e dispositivos de E/S**

Em um sistema computacional, os diversos subsistemas precisam ter interfaces entre si. Por exemplo, a memória e o processador precisam se comunicar, assim como o processador e os dispositivos de E/S. Durante muitos anos, isso tem sido feito com um *barramento*. Um barramento é um link de comunicação compartilhado, que utiliza um conjunto de fios para conectar diversos subsistemas. As duas vantagens principais da organização do barramento são versatilidade e baixo custo. Definindo um único esquema de conexão, novos dispositivos podem ser facilmente acrescentados, e os periféricos podem ainda ser movidos entre os sistemas computacionais que utilizam o mesmo tipo de bar-

ramento. Além do mais, os barramentos são eficazes porque um único conjunto de fios é compartilhado de várias maneiras.

A principal desvantagem de um barramento é que ele cria um gargalo de comunicação, possivelmente limitando a vazão máxima de E/S. Quando a E/S tiver de passar por um único barramento, a largura de banda desse barramento limita a vazão máxima da E/S. O principal desafio é projetar um sistema de barramento capaz de atender às demandas do processador e também conectar grandes quantidades de dispositivos de E/S à máquina.

Um motivo para o projeto de barramento ser tão difícil é que a velocidade máxima do barramento é limitada principalmente pelos fatores físicos: a extensão do barramento e o número de dispositivos. Esses limites físicos nos impedem de executar o barramento arbitrariamente rápido. Além disso, a necessidade de dar suporte a uma gama de dispositivos com latências e taxas de transferência de dados muito variáveis também torna o projeto do barramento desafiador.

Como se torna difícil trabalhar com muitos fios paralelos em alta velocidade devido a variações de clock e reflexão, o setor está em transição, passando de barramentos paralelos compartilhados para interconexões seriais ponto a ponto de alta velocidade com switches. Assim, essas redes estão gradualmente substituindo os barramentos em nossos sistemas.

Como resultado dessa transição, esta seção foi revisada nesta edição para enfatizar o problema geral de conectar dispositivos de E/S, processadores e memória, em vez de focalizar exclusivamente os barramentos.

Fundamentos sobre barramento

Classicamente, um barramento em geral contém um conjunto de linhas de controle e um conjunto de linhas de dados. As linhas de controle são usadas para sinalizar solicitações e confirmações, e também para indicar que tipo de informação se encontra nas linhas de dados. As linhas de dados do barramento transportam informações entre a origem e o destino. Essa informação pode consistir em dados, comandos complexos ou endereços. Por exemplo, se um disco quiser escrever algum dado na memória a partir de um setor do disco, as linhas de dados serão usadas para indicar o endereço na memória em que serão colocados os dados, além de transportar os dados reais do disco. As linhas de controle serão usadas para indicar que tipo de informação está contida nas linhas de dados do barramento em cada ponto da transferência. Alguns barramentos possuem dois conjuntos de linhas de sinal para comunicar separadamente dados e endereço em uma única transmissão de barramento. De qualquer forma, as linhas de controle são usadas para indicar o que o barramento contém e implementar o protocolo do barramento. E como o barramento é compartilhado, também precisamos de um protocolo para decidir quem o usará em seguida; vamos discutir esse problema em breve.

Vamos considerar uma **transação de barramento** típica. Uma transação de barramento inclui duas partes: enviar o endereço e receber ou enviar os dados. As transações de barramento normalmente são definidas pelo que fazem com a memória. Uma transação de *leitura* transfere dados *da* memória (para o processador ou para um dispositivo de E/S), e uma transação de *escrita* escreve dados *na* memória. Logicamente, essa terminologia é confusa. Para evitar isso, vamos tentar usar os termos *entrada* e *saida*, que sempre são definidos do ponto de vista do processador: uma operação de entrada significa entrar dados do dispositivo para a memória, onde o processador os poderá ler, e uma operação de saída significa sair com dados para um dispositivo a partir da memória, onde o processador os escreve.

Os barramentos tradicionalmente são classificados como **barramentos processador-memória**, ou **barramentos de E/S**. Os barramentos processador-memória são curtos, geralmente de alta velocidade e correspondentes ao sistema de memória, de modo a maximizar a largura de banda memória-processador. Os barramentos de E/S, ao contrário, podem ser extensos, podem ter muitos tipos de dispositivos conectados a eles e normalmente possuem uma grande faixa de largura de banda de dados dos dispositivos conectados a eles. Os barramentos de E/S normalmente não realizam interface direta com a memória, mas utilizam um barramento processador-memória ou um **barramento backplane** para a conexão com a memória. Outros barramentos com características diferentes surgiram para funções especiais, como barramentos gráficos.

transação de barramento

Uma sequência de operações de barramento que inclui uma solicitação e pode incluir uma resposta, ambas podendo transportar dados. Uma transação é iniciada por uma única solicitação e pode exigir várias operações de barramento individuais.

barramento processador-memória

Um barramento que conecta processador e memória, e que é curto, geralmente de alta velocidade, e correspondente ao sistema de memória, de modo a maximizar a largura de banda memória-processador.

barramento backplane

E/S coexistem em um único barramento.

O barramento de E/S serve como um modo de expandir a máquina e conectar novos periféricos. Para facilitar isso, o setor de computadores desenvolveu diversos padrões. Os padrões servem como uma especificação para o fabricante de computador e para o fabricante de periféricos. Um padrão garante ao projetista do computador que os periféricos estarão disponíveis para uma nova máquina, e garante ao montador do periférico que os usuários poderão se conectar ao seu novo equipamento. A Figura 8.9 resume as principais características dos dois padrões de barramento de E/S dominantes: Firewire e USB. Eles conectam uma série de dispositivos aos computadores desktop, desde teclados a câmeras e discos.

Características	Firewire (1394)	USB 2.0
Tipo de barramento	E/S	E/S
Largura básica do barramento de dados (sinais)	4	2
Clock	assíncrono	assíncrono
Largura de banda máxima teórica	50MB/seg (Firewire 400) ou 100MB/seg (Firewire 800)	0,2MB/seg (baixa velocidade), 1,5MB/seg (velocidade plena) ou 60MB/seg (velocidade alta)
Conectável "a quente"	sim	sim
Número máximo de dispositivos	63	127
Tamanho máximo do barramento (fio de cobre)	4,5 metros	5 metros
Nome do padrão	IEEE 1394, 1394b	USB Implementors Forum

FIGURA 8.9 Principais características dos dois padrões de barramento de E/S dominantes.

Os dois esquemas básicos para comunicação no barramento são **síncrono** e **assíncrono**. Se um barramento for síncrono, ele inclui um clock nas linhas de controle e um protocolo fixo para comunicação, que é relativo ao clock. Por exemplo, para um barramento processador-memória realizando uma leitura da memória, poderíamos ter um protocolo que transmite o endereço e o comando read no primeiro clock, usando as linhas de controle para indicar o tipo de solicitação. A memória poderia, então, ter de responder com a word de dados no quinto clock. Esse tipo de protocolo pode ser facilmente implementado com uma pequena máquina de estados finitos. Como o protocolo é predeterminado e envolve pouca lógica, o barramento pode ser executado muito rapidamente e a lógica da interface será pequena.

Entretanto, os barramentos síncronos possuem duas grandes desvantagens. Primeiro, cada dispositivo no barramento precisa executar na mesma velocidade de clock. Segundo, devido a problemas de variação de clock, os barramentos síncronos não podem ser longos se forem rápidos (ver o Apêndice B para obter uma discussão sobre variação do clock). Os barramentos processador-memória normalmente são síncronos, pois os dispositivos que se comunicam estão próximos, são poucos em número e estão preparados para operar em velocidades de clock altas.

Um barramento assíncrono não utiliza clock. Por não ter clock, um barramento assíncrono pode acomodar uma grande variedade de dispositivos, e o barramento pode ser estendido sem preocupação com problemas de variação de clock ou sincronismo. Tanto Firewire quanto USB 2.0 são assíncronos. Para coordenar a transmissão de dados entre o emissor e o receptor, um barramento assíncrono utiliza um **protocolo de handshaking**. Um protocolo de handshaking consiste em uma série de etapas em que o emissor e o receptor prosseguem para a próxima etapa apenas quando as duas partes concordarem. O protocolo é implementado com um conjunto adicional de linhas de controle.

Um exemplo simples ilustrará como funcionam os barramentos assíncronos. Vamos considerar um dispositivo requisitando uma word de dados ao sistema de memória. Suponha que existam três linhas de controle:

barramento síncrono
Um barramento que inclui um clock nas linhas de controle e um protocolo fixo para comunicação, que é relativo ao clock.

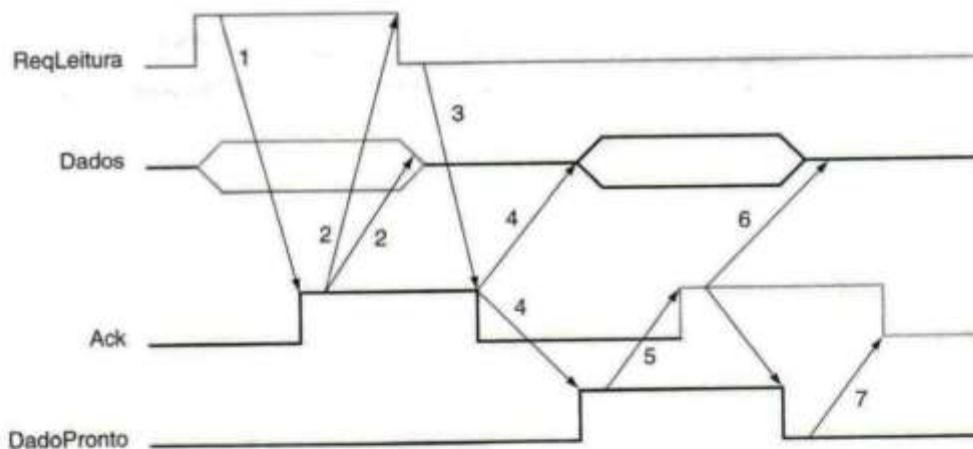
barramento assíncrono
Um barramento que utiliza um protocolo de handshaking para coordenar o uso, em vez de um clock; pode acomodar uma grande variedade de dispositivos, de diferentes velocidades.

protocolo de handshaking
Uma série de etapas usadas para coordenar as transferências em barramentos assíncronos em que o emissor e o receptor só prosseguem para a próxima etapa quando as duas partes concordarem que a etapa atual foi concluída.

1. *ReqLeitura*: usado para indicar uma requisição de leitura para a memória. O endereço é colocado nas linhas de dados ao mesmo tempo.
2. *DadoPronto*: usado para indicar que a word de dados agora está pronta nas linhas de dados. Em uma transação de saída, a memória ativará esse sinal, pois está fornecendo os dados. Em uma transação de entrada, um dispositivo de E/S ativaría esse sinal, pois ele forneceria dados. De qualquer forma, os dados são colocados nas linhas de dados ao mesmo tempo.
3. *Ack*: usado para confirmar o sinal *ReqLeitura* ou *DadoPronto* da outra parte.

Em um protocolo assíncrono, os sinais de controle *ReqLeitura* e *DadoPronto* são ativados até que a outra parte (a memória ou o dispositivo) indique que as linhas de controle foram vistas e as linhas de dados foram lidas; essa indicação é feita ativando-se a linha *Ack*. Esse processo completo é chamado de *handshaking*. A Figura 8.10 mostra como esse protocolo opera, descrevendo as etapas na comunicação.

Embora grande parte da largura de banda de um barramento seja decidida pela escolha de um protocolo síncrono ou assíncrono e as características de temporização do barramento, vários outros fatores afetam a largura de banda que pode ser alcançada por uma única transferência. Os mais importantes deles são a largura do barramento de dados e se ele admite transferências em bloco ou transfere uma word de cada vez.



As etapas no protocolo começam imediatamente após o dispositivo sinalizar uma requisição, levantando *ReqLeitura* e colocando o endereço nas linhas de *Dados*:

1. Quando a memória vê a linha *ReqLeitura*, ela lê o endereço do barramento de dados e levanta *Ack* para indicar que ele foi visto.
2. O dispositivo de E/S vê a linha *Ack* alta e libera as linhas *ReqLeitura* e de *Dados*.
3. A memória vê que *ReqLeitura* está baixa e abaixa a linha *Ack* para confirmar o sinal *ReqLeitura*.
4. Essa etapa começa quando a memória possui dados prontos. Ela coloca os dados da requisição de leitura nas linhas de dados e levanta *DadoPronto*.
5. O dispositivo de E/S vê *DadoPronto*, lê os dados do barramento e sinaliza que possui os dados levantando *Ack*.
6. A memória vê o sinal *Ack*, abaixa *DadoPronto* e libera as linhas de dados.
7. Finalmente, o dispositivo de E/S, vendo *DadoPronto* baixar, abaixa a linha *Ack*, o que indica que a transmissão está concluída.

Uma nova transação no barramento pode ser iniciada agora.

FIGURA 8.10 O protocolo de handshaking assíncrono consiste em sete etapas para ler uma word da memória e recebê-la em um dispositivo de E/S. Os sinais em destaque são aqueles ativados pelo dispositivo de E/S, enquanto a memória ativa os sinais mostrados em preto. As setas rotulam as sete etapas e o evento que dispara cada etapa. O símbolo mostrando duas linhas (alto e baixo) ao mesmo tempo nas linhas de dados indica que as linhas de dados possuem dados válidos nesse ponto. (O símbolo indica que os dados são válidos, mas o valor não é conhecido.)

Detalhamento: outro método para aumentar a largura de banda efetiva do barramento é liberar o barramento quando ele não estiver sendo usado para transmitir informações. Esse tipo de protocolo é chamado de protocolo de transação repartida. A vantagem de tal protocolo é que, liberando o barramento durante o momento em que os dados não estão sendo transmitidos, o protocolo permite que outro solicitante use o barramento. Isso pode melhorar a largura de banda efetiva do barramento para o sistema inteiro, se a memória for sofisticada o suficiente para tratar com várias transações superpostas. Os multiprocessadores compartilhando um barramento de memória podem usar **protocolos de transação repartida**.

protocolo de transação repartida Um protocolo em que o barramento é liberando durante uma transação do barramento enquanto o solicitante está aguardando que os dados sejam transmitidos, o que libera o barramento para acesso por outro solicitante.

Barramentos e redes do Pentium 4

A Figura 8.11 mostra o sistema de E/S de um PC baseado no Pentium 4. O processador se conecta a periféricos por meio de dois chips principais. O chip próximo ao processador é o hub controlador da memória, normalmente chamado *bridge norte*, e aquele conectado a ele é o hub controlador de E/S, chamado de *bridge sul*.

A bridge norte é basicamente um controlador de DMA, conectando o processador à memória, ao barramento gráfico AGP e ao chip da bridge sul. A bridge sul conecta a bridge norte a uma abundância de barramentos de E/S. A Intel e outros fabricantes oferecem uma grande variedade de chip sets para conectar o Pentium 4 ao mundo exterior. Para exemplificar as opções, a Figura 8.12 mostra dois desses chip sets.

À medida que a Lei de Moore continua a vigorar, um número cada vez maior de controladoras de E/S, que antes estavam disponíveis como placas opcionais conectadas aos barramentos de E/S, tem sido incorporadas por esses chip sets. Por exemplo, o chip bridge sul da Intel, o 875, inclui uma controladora RAID com striping, e o chip bridge norte da Intel, o 845GL, inclui uma controladora gráfica.

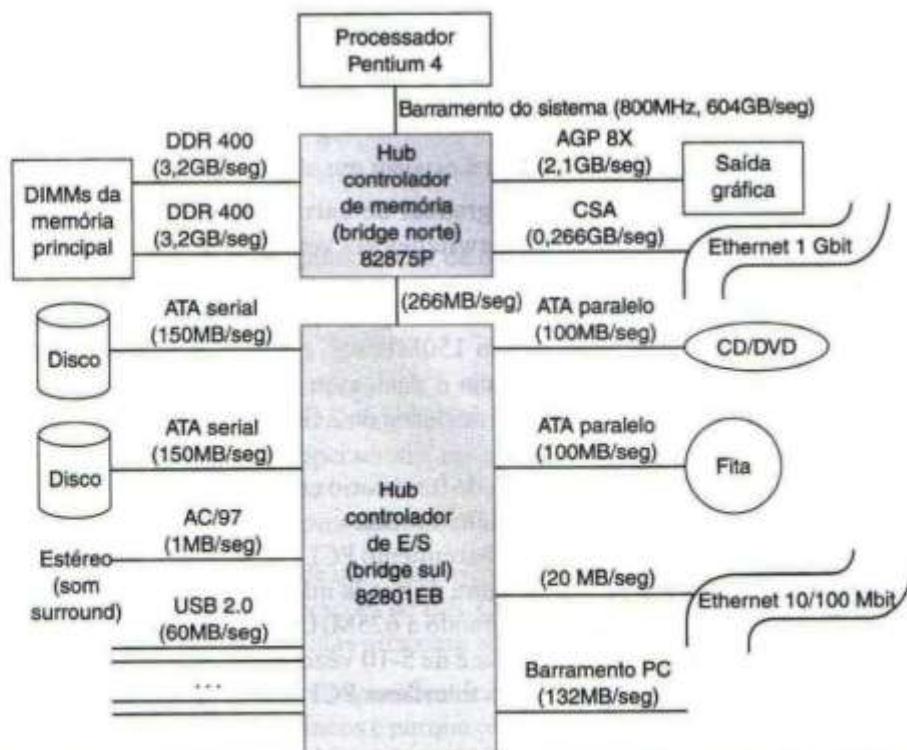


FIGURA 8.11 Organização do sistema de E/S em um PC Pentium 4 usando o chip set Intel 875.

Observe que a taxa de transferência máxima entre a bridge norte (hub de memória) e a bridge sul (hub de E/S) é de 266MB/seg, motivo pelo qual a Intel colocou o barramento AGP e o Gigabit Ethernet na bridge norte.

	Chip set 875P	Chip set 845GL
Segmento de destino	Performance PC	Value PC
Barramento do sistema (64 bits)	800/533MHz	400MHz
Hub controlador de memória ("bridge norte")		
Tamanho do pacote, pinos	42,5 x 42,5mm, 1005	37,5 x 37,5mm, 760
Velocidade da memória	DDR 400/333/266 SDRAM	DDR 266/200, PC133 SDRAM
Barramentos de memória, larguras	2x72	1x64
Número de DIMMs, suporte a Mbit da DRAM	4, 128/256/512Mbits	2, 128/256/512Mbits
Capacidade máxima da memória	4GB	2GB
Correção de erro da memória disponível?	sim	não
Barramento gráfico AGP, velocidade	sim, 8X ou 4X	não
Controlador gráfico	Externo	Interno (Extreme Graphics)
Interface CSA Gigabit Ethernet	sim	não
Velocidade de interface da bridge sul (8 bits)	266MHz	266MHz
Hub controlador de E/S ("bridge sul")		
Tamanho do pacote, pinos	31 x 31mm, 460	31 x 31mm, 421
Barramento PCI: largura, velocidade, masters	32 bits, 33MHz, 6 masters	32 bits, 33MHz, 6 masters
Controlador MAC Ethernet, interface	100/10Mbits	100/10Mbits
Portas USB 2.0, controladoras	8, 4	6, 3
Portas ATA 100	2	2
Controlador ATA serial 150, portas	sim, 2	não
Controlador RAID 0	sim	não
Controlador de áudio AC-97, interface	sim	sim
Gerenciamento de E/S	SMBus 2.0, GPIO	SMBus 2.0, GPIO

FIGURA 8.12 Dois chip sets de E/S Pentium 4 da Intel. A bridge norte 845GL utiliza muito menos pinos do que o 875, com apenas um barramento de memória e omitindo o barramento AGP e a interface Gigabit Ethernet. Observe que a natureza serial do USB e da ATA serial significa que mais duas portas USB e mais duas portas ATA serial precisam de apenas mais 39 pinos na bridge sul do chip set 875 *versus* o chip set 845GL.

Esses dois chips demonstram a evolução gradual de barramentos compartilhados paralelos para interconexões ponto a ponto seriais com switches via versões passada e futura do ATA e do PCI.

ATA serial é um sucessor serial do barramento ATA paralelo, usado por discos magnéticos e ópticos nos PCs. A primeira geração transfere em 150MB/seg, em comparação com os 100MB/seg do barramento paralelo ATA-100. Sua distância é de 1 metro, o dobro do tamanho máximo do ATA-100. Ele usa apenas 7 fios, com um canal de dados de 2 fios em cada direção, em comparação com os 80 para ATA-100.

A bridge sul na Figura 8.11 demonstra o período transitório entre os barramentos paralelos e as redes seriais, oferecendo barramentos ATA paralelo e serial.

PCI Express é um sucessor serial do popular barramento PCI. No lugar de 32-64 fios compartilhados operando a 33MHz-133MHz com uma largura de banda máxima de 132-1064MB/seg, PCI Express utiliza apenas 4 fios em cada direção, operando a 625MHz para oferecer 300MB/seg por direção. A largura de banda por pino do PCI Express é de 5-10 vezes a de seus predecessores. Um computador pode, então, se dar ao luxo de ter várias interfaces PCI Express para conseguir uma largura de banda ainda mais alta.

Embora os chips da Figura 8.11 mostrem o barramento PCI paralelo, a Intel planeja substituir o barramento gráfico AGP e o barramento entre a bridge norte e a bridge sul por PCI Express na próxima geração desses chips.

Os barramentos e as redes oferecem interconexão elétrica entre os dispositivos de E/S, processadores e memória, e também definem o protocolo de mais baixo nível para a comunicação. Acima desse nível básico, temos de definir os protocolos de hardware e software para controlar as transferências de dados entre os dispositivos de E/S e a memória, e para o processador especificar comandos aos dispositivos de E/S. Esses assuntos serão abordados na próxima seção.

Redes e barramentos conectam componentes. Quais das seguintes afirmações são verdadeiras sobre eles?

Verifique você mesmo

1. As redes e os barramentos de E/S são quase sempre padronizados.
2. As redes de mídia compartilhada e os barramentos multimaster precisam de um esquema de arbitragem.
3. As redes locais e os barramentos processador-memória são quase sempre síncronos.
4. As redes e os barramentos de alto desempenho utilizam técnicas semelhantes, comparadas com suas alternativas de desempenho inferior: elas são mais largas, enviam muitas words por transação e possuem linhas separadas de endereço e dados.

8.5

Interface dos dispositivos de E/S com processador, memória e sistema operacional

Um protocolo de barramento ou de rede define como uma word ou bloco de dados devem ser comunicados em um conjunto de fios. Isso ainda deixará várias outras tarefas que precisam ser realizadas para realmente fazer com que os dados sejam transferidos de um dispositivo para o espaço de endereçamento da memória de algum programa de usuário. Esta seção focaliza essas tarefas e responderá a perguntas como estas:

- Como uma solicitação de E/S de um usuário é transformada em um comando de dispositivo e comunicada ao dispositivo?
- Como os dados são realmente transferidos de ou para um local da memória?
- Qual é o papel do sistema operacional?

Como veremos na resposta a essas perguntas, o sistema operacional desempenha um papel importante no tratamento da E/S, atuando como interface entre o hardware e o programa que solicita a E/S. As responsabilidades do sistema operacional surgem de três características dos sistemas de E/S:

1. Diversos programas usando o processador compartilham o sistema de E/S.
2. Os sistemas de E/S normalmente usam interrupções (exceções geradas externamente) para comunicar informações sobre operações de E/S. Como as interrupções causam uma transferência ao modo kernel ou supervisor, elas precisam ser tratadas pelo sistema operacional (SO).
3. O controle de baixo nível de um dispositivo de E/S é complexo, pois exige o gerenciamento de um conjunto de eventos simultâneos e porque os requisitos para o controle correto do dispositivo normalmente são muito detalhados

Interface hardware/software

As três características dos sistemas de E/S anteriores levam a diversas funções diferentes que o SO precisa oferecer:

- O sistema operacional garante que o programa de um usuário acessa apenas as partes de um dispositivo de E/S para as quais o usuário possui direitos. Por exemplo, o sistema operacional não pode permitir que um programa leia ou escreva num arquivo no disco se o proprietário do arquivo não tiver acesso a esse programa. Em um sistema com dispositivos de E/S compartilhados, a proteção não poderia ser fornecida se os programas de usuário pudessem realizar E/S diretamente.
- O sistema operacional oferece abstrações para acessar dispositivos fornecendo rotinas que tratam as operações de baixo nível dos dispositivos.
- O sistema operacional trata as interrupções geradas pelos dispositivos de E/S, assim como trata as exceções geradas por um programa.
- O sistema operacional tenta oferecer acesso equilibrado aos recursos de E/S, além de escalonar acessos a fim de melhorar a vazão do sistema.

Para realizar essas funções em favor dos programas de usuário, o sistema operacional precisa ser capaz de se comunicar com os dispositivos de E/S e impedir que o programa do usuário se comunique com os dispositivos de E/S diretamente. Três tipos de comunicação são necessários:

1. O sistema operacional precisa ser capaz de dar comandos aos dispositivos de E/S. Esses comandos incluem não apenas operações como ler e escrever, mas também outras operações a serem feitas no dispositivo, como uma busca em um disco.
2. O dispositivo precisa ser capaz de notificar o sistema operacional quando o dispositivo de E/S tiver completado uma operação ou tiver encontrado um erro. Por exemplo, quando um disco completar uma busca, ele notificará o sistema operacional.
3. Os dados precisam ser transferidos entre a memória e um dispositivo de E/S. Por exemplo, o bloco sendo lido em uma leitura de disco precisa ser movido do disco para a memória.

Nas próximas seções, veremos como essas comunicações são realizadas.

Dando comandos a dispositivos de E/S

Para dar um comando a um dispositivo de E/S, o processador precisa ser capaz de endereçar o dispositivo e fornecer uma ou mais palavras de comando. Dois métodos são usados para endereçar o dispositivo: E/S mapeada em memória e instruções de E/S especiais. Na **E/S mapeada em memória**, partes do espaço de endereçamento são atribuídas a dispositivos de E/S. Leituras e escritas para esses endereços são interpretadas como comandos aos dispositivos de E/S.

Por exemplo, uma operação de escrita pode ser usada para enviar dados a um dispositivo de E/S, onde os dados serão interpretados como um comando. Quando o processador coloca o endereço e os dados no barramento da memória, o sistema de memória ignora a operação, porque o endereço indica uma parte do espaço de memória usado para E/S. O controlador de dispositivos, porém, vê a operação, registra os dados e os transmite ao dispositivo como um comando. Os programas de usuário são impedidos de realizar operações de E/S diretamente, pois o sistema operacional não oferece acesso ao espaço de endereçamento atribuído aos dispositivos de E/S e, assim, os endereços são protegidos pela tradução de endereços. A E/S mapeada em memória também pode ser usada para transmitir dados, escrevendo ou lendo para selecionar endereços. O dispositivo utiliza o endereço para

E/S mapeada em memória Um esquema de E/S em que partes do espaço de endereçamento são atribuídas a dispositivos de E/S e leituras e escritas para esses endereços são interpretadas como comandos aos dispositivos de E/S.



ELSEVIER

determinar o tipo de comando, e os dados podem ser fornecidos por uma escrita ou obtidos por uma leitura. De qualquer forma, o endereço codifica a identidade do dispositivo e o tipo de transmissão entre o processador e o dispositivo.

Na realidade, realizar uma leitura ou escrita de dados para cumprir uma solicitação do programa normalmente exige várias operações de E/S separadas. Além do mais, o processador pode ter de interrogar o status do dispositivo entre comandos individuais para determinar se o comando foi concluído com sucesso. Por exemplo, uma simples impressora possui dois registradores de dispositivo de E/S – um para informações de status e um para dados a serem impressos. O registrador de status contém um *bit de pronto*, ligado pela impressora quando ela tiver impresso um caractere, e um *bit de erro*, indicando que a impressora está com papel preso ou sem papel. Cada byte de dados a ser impresso é colocado no registrador de dados. O processador precisa, então, esperar até que a impressora ligue o bit pronto antes que possa colocar outro caractere no buffer. O processador também precisa verificar o bit de erro para determinar se houve um problema. Cada uma dessas operações exige um acesso separado ao dispositivo de E/S.

Detalhamento: a alternativa à E/S mapeada em memória é usar instruções de E/S dedicadas no processador. Essas **Instruções de E/S** podem especificar o número do dispositivo e a word de comando (ou o local da word de comando na memória). O processador comunica o endereço do dispositivo por meio de um conjunto de fios normalmente incluídos como parte do barramento de E/S. O comando real pode ser transmitido pelas linhas de dados do barramento. Exemplos de computadores com instruções de E/S são os computadores Intel IA-32 e o IBM 370. Tornando as instruções de E/S ilegais para serem executadas quando fora do modo kernel ou supervisor, os programas de usuário são impedidos de acessar os dispositivos diretamente.

Comunicação com o processador

O processo de verificar periodicamente os bits de status para ver se é hora da próxima operação de E/S, como no exemplo anterior, é chamado de **polling**. O polling é a forma mais simples para um dispositivo de E/S se comunicar com o processador. O dispositivo de E/S simplesmente coloca a informação no registrador de status, e o processador deve vir e apanhar a informação. O processador está totalmente no controle e realiza todo o trabalho.

O polling pode ser usado de várias maneiras diferentes. As aplicações embutidas de tempo real sondam os dispositivos de E/S porque as taxas de E/S são predeterminadas e isso torna o overhead da E/S mais previsível, o que é útil para tempo real. Como veremos, isso permite que a sondagem seja usada mesmo quando a taxa de E/S é um pouco maior.

A desvantagem do polling é que ele pode desperdiçar muito tempo de processador, pois os processadores são muito mais rápidos do que os dispositivos de E/S. O processador pode ler o registrador de status muitas vezes, para descobrir que o dispositivo não completou uma operação de E/S comparativamente lenta, ou que o mouse não saiu do lugar desde a última vez em que foi sondado. Quando o dispositivo completar uma operação, ainda teremos de ler o status para determinar se ele teve sucesso.

O overhead em uma interface de polling foi reconhecido há muito tempo, levando à invenção de interrupções para notificar o processador quando um dispositivo de E/S exigir atenção do processador. A **E/S controlada por interrupção**, usada por quase todos os sistemas pelo menos para alguns dispositivos, emprega interrupções de E/S para indicar ao processador que um dispositivo de E/S precisa de atenção. Quando um dispositivo deseja notificar o processador de que completou alguma operação ou que precisa de atenção, isso faz com que o processador seja interrompido.

Uma interrupção de E/S é exatamente como as exceções vistas nos Capítulos 5, 6 e 7, com duas exceções importantes:

- I. Uma interrupção de E/S é assíncrona com relação à execução da instrução. Ou seja, a interrupção não é associada a qualquer instrução e não impede o término da instrução. Isso é muito diferente de quaisquer exceções de falta de página ou exceções como overflow aritmético. Nossa unidade de controle só precisa verificar uma interrupção de E/S pendente no momento em que iniciar uma nova instrução.

instrução de E/S Uma instrução dedicada, usada para dar um comando a um dispositivo de E/S e que especifica o número do dispositivo e a word de comando (ou o local da word de comando na memória).

polling O processo de verificar periodicamente o status de um dispositivo de E/S para determinar a necessidade de atender ao dispositivo.

E/S controlada por interrupção Um esquema de E/S que emprega interrupções para indicar ao processador que um dispositivo de E/S precisa de atenção.

2. Além do fato de que uma interrupção de E/S ocorreu, gostaríamos de transmitir informações adicionais, como a identidade do dispositivo gerando a interrupção. Além do mais, as interrupções representam dispositivos que podem ter diferentes prioridades e cujas solicitações de interrupção possuem diferentes urgências associadas a elas.

Para comunicar informações ao processador, como a identidade do dispositivo que gera a interrupção, um sistema pode usar interrupções veteadas ou um registrador de causa da exceção. Quando o processador reconhece a interrupção, o dispositivo pode enviar o endereço do vetor ou um campo de status para colocar no registrador de causa. Como resultado, quando o sistema operacional adquire o controle, ele sabe a identidade do dispositivo que causou a interrupção e pode interrogar imediatamente o dispositivo. Um mecanismo de interrupção elimina a necessidade de o processador sondar o dispositivo e, em vez disso, permite que o processador seja focalizado nos programas em execução.

Níveis de prioridade de interrupção

Para lidar com as diferentes prioridades dos dispositivos de E/S, a maioria dos mecanismos de interrupção possui vários níveis de prioridade: sistemas operacionais UNIX utilizam de quatro a seis níveis. Essas prioridades indicam a ordem em que o processador deverá processar interrupções. Exceções geradas internamente e interrupções de E/S externas possuem prioridades; em geral, as interrupções de E/S possuem prioridade menor do que as exceções internas. Pode haver várias prioridades de interrupção de E/S, com dispositivos de alta velocidade associados às prioridades mais altas.

Para dar suporte a níveis de prioridade para interrupções, o MIPS oferece as primitivas que deixam o sistema operacional implementar a política, de modo semelhante ao modo como o MIPS trata de falhas de TLB. A Figura 8.13 mostra os principais registradores, e a Seção A.7 no Apêndice A oferece mais detalhes.

O registrador Status determina quem pode interromper o computador. Se o bit Interrupções habilitadas for 0, então ninguém poderá interromper. Um bloqueio de interrupções mais refinado está disponível no campo de máscara de interrupções. Existe um bit na máscara correspondente a cada bit no campo interrupções pendentes do registrador Cause. Para habilitar a interrupção correspondente, é preciso haver um 1 no campo de máscara no bit dessa posição. Quando ocorre uma interrupção, o sistema operacional pode encontrar o motivo no campo de código de exceção do registrador Status: 0 significa que uma interrupção ocorreu, com outros valores para as exceções mencionadas no Capítulo 7.

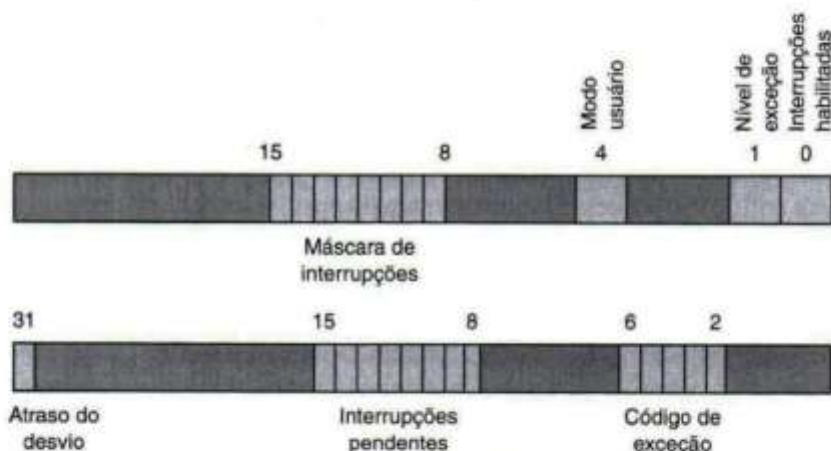


FIGURA 8.13 Os registradores Cause e Status. Essa versão do registrador Cause corresponde à arquitetura MIPS-32. A arquitetura MIPS I mais antiga tinha três conjuntos aninhados de bits kernel/usuário e de bits de habilitação de interrupções para dar suporte a interrupções aninhadas. A Seção A.7 no Apêndice A possui mais detalhes sobre esses registradores.

Aqui estão as etapas que precisam ocorrer no tratamento de uma exceção:

1. Realize um AND lógico entre o campo interrupções pendentes e o campo máscara de interrupções para ver quais interrupções ativas poderiam ser as culpadas. São feitas cópias desses dois registradores usando a instrução mfc0.
2. Selecione a prioridade mais alta dessas interrupções. A convenção do software é que a mais à esquerda seja a prioridade mais alta.
3. Salve o campo de máscara de interrupções do registrador Status.
4. Mude o campo de máscara de interrupções para desativar todas as interrupções de prioridade igual ou inferior.
5. Salve o estado do processador necessário para lidar com a interrupção.
6. Para permitir interrupções de prioridade mais alta, coloque o bit interrupções habilitadas do registrador Cause em 1.
7. Chame a rotina de interrupção apropriada.
8. Antes de restaurar o estado, coloque o bit interrupções habilitadas do registrador Cause em 0. Isso permite restaurar o campo de máscara de interrupções.

■ O Apêndice A mostra um handler de exceções para uma tarefa de E/S simples.

Como os níveis de prioridade de interrupção (IPL – Interrupt Priority Levels) correspondem a esses mecanismos? O IPL é uma invenção do sistema operacional. Ele é armazenado na memória do processo, e cada processo recebe um IPL. No IPL mais baixo, todas as interrupções são permitidas. Ao contrário, no IPL mais alto, todas as interrupções são bloqueadas. Levantar e reduzir o IPL envolve mudanças no campo de máscara de interrupção do registrador Status.

Detalhamento: os dois bits menos significativos dos campos interrupções pendentes e máscara de interrupções são para interrupções de software, que são de prioridade inferior. Eles normalmente são usados por interrupções de prioridade mais alta para deixar trabalho para interrupções de menor prioridade realizarem depois que o motivo imediato da interrupção for tratado. Quando a interrupção de maior prioridade terminar, as tarefas de prioridade inferior serão observadas e tratadas.

Transferindo os dados entre um dispositivo e a memória

Vimos dois métodos diferentes que permitem que um dispositivo se comunique com o processador. Essas duas técnicas – polling e interrupções de E/S – formam a base para dois métodos de implementação da transferência de dados entre o dispositivo de E/S e a memória. Essas duas técnicas funcionam melhor com dispositivos de menor largura de banda, nos quais estamos mais interessados em reduzir o custo do controlador de dispositivo e interface do que oferecer uma transferência com largura de banda alta. Tanto o polling quanto as transferências controladas por interrupção colocam o trabalho de mover dados e gerenciar a transferência sobre os ombros do processador. Depois de examinar esses dois esquemas, veremos um esquema mais adequado para dispositivos de maior desempenho ou coleções de dispositivos.

Podemos usar o processador para transferir dados entre um dispositivo e a memória com base no polling. Em aplicações de tempo real, o processador carrega dados dos registradores do dispositivo de E/S e os armazena na memória.

Um mecanismo alternativo é fazer a transferência de dados controlada por interrupção. Nesse caso, o sistema operacional ainda transferiria dados em pequenos números de bytes de ou para o dispositivo. Entretanto, como a operação de E/S é controlada por interrupção, o sistema operacional simplesmente atua sobre outras tarefas enquanto os dados estão sendo lidos ou escritos no dispositivo.

vo. Quando o sistema operacional reconhece uma interrupção a partir do dispositivo, ele lê o status para verificar a ocorrência de erros. Se não houver, o sistema operacional poderá fornecer a próxima parte dos dados, por exemplo, por uma seqüência de escritas mapeadas em memória. Quando o último byte de uma solicitação de E/S tiver sido transmitido e a operação de E/S for concluída, o sistema operacional poderá informar ao programa. O processador e o sistema operacional realizam todo o trabalho nesse processo, acessando o dispositivo e a memória para cada item de dados transferido.

A E/S controlada por interrupção libera o processador de ter de esperar por cada evento de E/S, embora, se usássemos esse método para transferir dados de ou para um disco rígido, o overhead ainda poderia ser intolerável, pois isso poderia consumir uma grande fração do processador quando o disco estivesse transferindo. Para dispositivos com alta largura de banda, como discos rígidos, as transferências consistem principalmente em blocos de dados relativamente grandes (centenas a milhares de bytes). Assim, os projetistas de computadores inventaram um mecanismo para desafogar o processador e fazer com que o controlador de dispositivo transfira dados diretamente de ou para a memória sem envolver o processador. Esse mecanismo é chamado de **acesso direto à memória** (DMA – Direct Memory Access). O mecanismo de interrupção ainda é usado pelo dispositivo para a comunicação com o processador, mas somente no término da transferência de E/S ou quando ocorre um erro.

O DMA é implementado com um controlador especializado, que transfere dados entre um dispositivo de E/S e a memória, independente do processador. O controlador de DMA torna-se o **bus master** e direciona as leituras e escritas entre si mesmo e a memória. Existem três etapas em uma transferência de DMA:

1. O processador configura o DMA fornecendo a identidade do dispositivo, a operação a realizar no dispositivo, o endereço de memória que é a origem ou o destino dos dados a serem transferidos e o número de bytes a transferir.
2. O DMA inicia a operação no dispositivo e arbitra o acesso ao barramento. Quando os dados estão disponíveis (do dispositivo ou da memória), ele transfere os dados. O dispositivo de DMA fornece o endereço de memória para a leitura ou a escrita. Se a solicitação exigir mais de uma transferência no barramento, a unidade de DMA gera o próximo endereço de memória e inicia a próxima transferência. Usando esse mecanismo, a unidade de DMA pode completar uma transferência inteira, que pode ter milhares de bytes de tamanho, sem incomodar o processador. Muitos controladores de DMA contêm alguma memória para permitir que eles tratem de modo flexível atrasos na transferência ou aqueles ocorridos na espera para se tornar o bus master.
3. Quando a transferência de DMA termina, o controlador interrompe o processador, que pode então determinar, interrogando o dispositivo de DMA ou examinando a memória, se a operação inteira foi concluída com sucesso.

Pode haver vários dispositivos de DMA em um sistema de computador. Por exemplo, em um sistema com um único barramento processador-memória e vários barramentos de E/S, cada controlador de barramento de E/S normalmente terá um processador de DMA que trata de quaisquer transferências entre um dispositivo no barramento de E/S e a memória.

Ao contrário do polling ou da E/S controlada por interrupção, o DMA pode ser usado para realizar interface de um disco rígido sem consumir todos os ciclos de processador para uma única E/S. Naturalmente, se o processador também estiver brigando pela memória, ele será atrasado quando a memória estiver ocupada realizando uma transferência de DMA. Usando caches, o processador pode evitar ter de acessar a memória na maior parte do tempo, deixando assim a maior parte da largura de banda da memória livre para uso por dispositivos de E/S.

Detalhamento: para reduzir ainda mais a necessidade de interromper o processador e ocupá-lo no tratamento de uma solicitação de E/S que possa envolver a realização de várias operações reais, o controlador de E/S pode se tornar mais inteligente. Controladores inteligentes normalmente são chamados de *processadores de E/S* (bem como *controladores de E/S* ou *controladores de canal*). Esses processadores especializados executam

acesso direto à memória (DMA) Um mecanismo que oferece a um controlador de dispositivo a capacidade de transferir dados diretamente da memória ou para ela sem envolver o processador.

bus master Uma unidade no barramento que pode iniciar solicitações ao barramento.

uma série de operações de E/S, chamadas de *programa de E/S*. O programa pode estar armazenado no processador de E/S, ou pode estar armazenado na memória e ser buscado pelo processador de E/S. Ao usar um processador de E/S, o sistema operacional normalmente configura um programa de E/S que indica as operações de E/S a serem realizadas, além do tamanho e do endereço de transferência para quaisquer leituras ou escritas. O processador de E/S, então, busca as operações do programa de E/S e interrompe o processador apenas quando o programa inteiro estiver completo. Os processadores de DMA são processadores de uso especial (normalmente, de único chip e não-programáveis), enquanto os processadores de E/S normalmente são implementados com microprocessadores de uso geral, que executam um programa de E/S especializado.

Acesso direto à memória e o sistema de memória

Quando o DMA é incorporado a um sistema de E/S, o relacionamento entre o sistema de memória e o processador muda. Sem DMA, todos os acessos ao sistema de memória vêm do processador e, assim, prosseguem pela tradução de endereços e acesso à cache como se o processador gerasse as referências. Com DMA, existe outro caminho para o sistema de memória – que não passa pelo mecanismo de tradução de endereços ou pela hierarquia de cache. Essa diferença gera alguns problemas nos sistemas de memória virtual e em sistemas com caches. Esses problemas normalmente são solucionados com uma combinação de técnicas de hardware e suporte do software.

As dificuldades de ter DMA em um sistema de memória virtual surgem porque as páginas possuem um endereço físico e um endereço virtual. O DMA também cria problemas para sistemas com caches, pois pode haver duas cópias de um item de dados: uma na cache e uma na memória. Como o processador de DMA realiza solicitações de memória diretamente à memória, e não pela cache do processador, o valor de um local de memória visto pela unidade de DMA e pelo processador pode ser diferente. Considere uma leitura do disco que a unidade de DMA coloque diretamente na memória. Se alguns dos locais em que o DMA escreve estiverem na cache, o processador receberá o valor antigo quando fizer uma leitura. De modo semelhante, se a cache for write-back, o DMA poderá ler um valor diretamente da memória quando um valor mais novo estiver na cache, e o valor não foi escrito de volta. Isso é chamado de *problema de dados antigos*, ou *problema de coerência*.

Interface hardware/software

Em um sistema com memória virtual, o DMA deverá funcionar com endereços virtuais ou com endereços físicos? O problema óbvio com os endereços virtuais é que a unidade de DMA precisará traduzir os endereços virtuais em endereços físicos. O problema principal com o uso de um endereço físico em uma transferência de DMA é que a transferência não pode cruzar com facilidade um limite de página. Se uma solicitação de E/S cruzasse um limite de página, então os locais de memória para os quais ela estava sendo transferida não necessariamente seriam contíguos na memória virtual. Consequentemente, se usarmos endereços físicos, teremos de restringir todas as transferências de DMA para permanecerem dentro de uma página.

Um método para permitir que o sistema inicie transferências de DMA que cruzam limites de página é fazer com que o DMA funcione em endereços virtuais. Nesse sistema, a unidade de DMA possui um pequeno número de entradas de mapa que oferecem mapeamento virtual-para-físico para uma transferência. O sistema operacional provê o mapeamento quando a E/S for iniciada. Usando esse mapeamento, a unidade de DMA não precisa se preocupar com o local das páginas virtuais envolvidas na transferência.

Outra técnica é que o sistema operacional divida a transferência de DMA em uma série de transferências, cada uma confinada dentro de uma única página física. As transferências, então, são *encadeadas* e entregues a um processador de E/S ou unidade de DMA inteligente, que executa a seqüência inteira de transferências; como alternativa, o sistema operacional pode solicitar as transferências individualmente.

Qualquer que seja o método utilizado, o sistema operacional ainda precisa cooperar não remapeando as páginas enquanto uma transferência de DMA que envolve essa página estiver em andamento.

Vimos três métodos diferentes para transferir dados entre um dispositivo de E/S e a memória. Ao passar do polling para uma E/S controlada por interrupção e para uma interface de DMA, mudamos o peso do gerenciamento de uma operação de E/S do processador para um controlador de E/S progressivamente mais inteligente. Esses métodos têm a vantagem de liberar os ciclos do processador. Sua desvantagem é que eles aumentam o custo do sistema de E/S. Por causa disso, determinado sistema computacional pode escolher qual ponto nesse espectro é apropriado para os dispositivos de E/S se conectarem a ele.

Antes de discutirmos o projeto dos sistemas de E/S, vejamos rapidamente as medidas de desempenho deles.

Interface hardware/software

O problema de coerência para dados de E/S é evitado pelo uso de uma de três técnicas importantes. Uma técnica é rotear a atividade de E/S por meio da cache. Isso garante que as leituras vejam o valor mais recente enquanto as escritas atualizam quaisquer dados na cache. O roteamento de toda a E/S pela cache é dispendioso e possui um grande impacto potencial negativo no desempenho do processador, pois os dados de E/S raramente são usados de imediato e podem deslocar dados úteis de que um programa em execução precisa. Uma segunda opção é ter o sistema operacional invalidando a cache seletivamente para uma leitura de E/S ou forçar a ocorrência de write-backs para uma escrita de E/S (normalmente chamado de *flush* de cache). Essa técnica exige uma pequena quantidade de suporte do hardware e provavelmente é mais eficiente se o software puder realizar a função de forma fácil e eficiente. Como esse flush de grandes partes da cache só precisa acontecer nos acessos em bloco ao DMA, ele será relativamente pouco frequente. A terceira técnica é oferecer um mecanismo de hardware para fazer o flush (ou invalidar) seletivamente as entradas de cache. A invalidação do hardware para garantir coerência da cache é típica em sistemas multiprocessador, e a mesma técnica pode ser usada para E/S; discutimos esse assunto com detalhes no Capítulo 9.

Verifique você mesmo

Na avaliação das três maneiras de realizar E/S, quais afirmações são verdadeiras?

1. Se quisermos a menor latência para uma operação de E/S a um único dispositivo de E/S, a ordem é polling, DMA e E/S controlada por interrupção.
2. Em termos de menor impacto na utilização do processador a partir de um único dispositivo de E/S, a ordem é DMA, E/S controlada por interrupção e polling.

8.6

Medidas de desempenho de E/S: exemplos de sistemas de disco e de arquivos

Como devemos comparar sistemas de E/S? Essa é uma pergunta complexa, porque o desempenho da E/S depende de muitos aspectos do sistema e diferentes aplicações enfatizam diferentes aspectos do sistema de E/S. Além do mais, um projeto pode fazer escolhas complexas entre tempo de resposta e vazão, tornando impossível medir apenas um aspecto isoladamente. Por exemplo, tratar um pedido o mais cedo possível em geral minimiza o tempo de resposta, embora uma vazão maior possa ser alcançada se tentarmos lidar com solicitações relacionadas juntas. De acordo com isso, podemos aumentar a vazão em um disco agrupando solicitações que acessam locais que estão próximos. Essa política aumentará o tempo de resposta para algumas solicitações, provavelmente levando a uma va-

rição maior no tempo de resposta. Embora a vazão seja maior, alguns benchmarks restringem o tempo de resposta máximo a qualquer solicitação, tornando tais otimizações potencialmente problemáticas.

Nesta seção, damos alguns exemplos de medidas propostas para determinar o desempenho dos sistemas de disco. Esses benchmarks são afetados por uma variedade de recursos do sistema, incluindo tecnologia de disco, como os discos são conectados, o sistema de memória, o processador e o sistema de arquivos fornecido pelo sistema operacional.

Antes de discutirmos esses benchmarks, precisamos explicar um ponto confuso sobre terminologia e unidades. O desempenho dos sistemas de E/S depende da velocidade em que o sistema transfere dados. A velocidade de transferência depende da velocidade do clock, que normalmente é dada em GHz = 10^9 ciclos por segundo. A taxa de transferência normalmente é cotada em GB/seg. Nos sistemas de E/S, GBs são medidos usando a base 10 (ou seja, 1GB = 10^9 = 1.000.000.000 bytes), diferente da memória principal, onde a base 2 é utilizada (ou seja, 1GB = 2^{30} = 1.073.741.824). Além de aumentar a confusão, essa diferença introduz a necessidade de conversão entre a base 10 (1K = 1000) e a base 2 (1K = 1024), porque muitos acessos à E/S são para blocos de dados que possuem um tamanho que é uma potência de dois. Em vez de complicar todos os nossos exemplos, convertendo com precisão uma das duas medidas, ressaltamos aqui essa distinção e o fato de que tratar as duas medidas como se as unidades fossem idênticas introduz um pequeno erro. Ilustramos esse erro na Seção 8.9.

Benchmarks de E/S de processamento de transações

Aplicações de **processamento de transações** (TP – Transaction Processing) envolvem um requisito de tempo de resposta e uma medida de desempenho baseada na vazão. Além do mais, a maioria dos acessos de E/S é pequena. Por causa disso, as aplicações de TP tratam principalmente da **tакса de E/S**, medida como o número de acessos ao disco por segundo, ao contrário da **tакса de dados**, medida como bytes de dados por segundo. As aplicações de TP geralmente envolvem mudanças em um banco de dados grande, com o sistema atendendo a alguns requisitos de tempo de resposta e tratando de forma controlada certos tipos de falhas. Essas aplicações são muito críticas e sensíveis ao custo. Por exemplo, os bancos normalmente utilizam sistemas de TP porque se preocupam com uma série de características, entre elas: garantir que as transações não são perdidas, tratar das transações rapidamente e minimizar o custo do processamento de cada transação. Embora a confiabilidade em face da falha seja um requisito absoluto em tais sistemas, o tempo de resposta e a vazão são críticos para criar sistemas econômicos.

Diversos benchmarks de processamento de transações foram desenvolvidos. O conjunto mais conhecido de benchmarks é uma série desenvolvida pelo Transaction Processing Council (TPC).

O TPC-C, inicialmente criado em 1992, simula um ambiente de consulta complexo. O TPC-H modela o apoio à decisão ocasional – as consultas não são relacionadas e o conhecimento de consultas passadas não pode ser usado para otimizar futuras consultas; o resultado é que os tempos de execução da consulta podem ser muito longos. TPC-R simula um sistema de apoio à decisão comercial, no qual os usuários realizam um conjunto padrão de consultas. No TPC-R, o preconhecimento das consultas é considerado sem questionamento, e o DBMS pode ser otimizado para executar essas consultas. O TPC-W é um benchmark de aplicações baseadas na Web, que simula as atividades de um servidor Web transacional orientado a negócios. Ele exercita o sistema de banco de dados e também o software básico do servidor Web. Os benchmarks TPC são descritos em [www\(tpc.org\)](http://www(tpc.org)).

Todos os benchmarks de TCP medem o desempenho em transações por segundo. Além disso, eles incluem um requisito de tempo de resposta, de modo que o desempenho da vazão é medido apenas quando o limite do tempo de resposta é atendido. Para modelar sistemas do mundo real, as velocidades de transação mais altas também estão associadas a sistemas maiores, tanto em termos de usuários quanto o tamanho do banco de dados ao qual as transações são aplicadas. Finalmente, o custo do sistema para um sistema de benchmark também precisa ser incluído, permitindo comparações precisas de custo-desempenho.

processamento de transações Um tipo de aplicação que envolve o tratamento de pequenas operações curtas (chamadas transações) que normalmente exigem tanto E/S quanto cálculo. As aplicações de processamento de transações normalmente possuem requisitos de tempo de resposta e uma medida de desempenho baseada na vazão das transações.

такса de E/S A medida de desempenho das E/S por unidade de tempo, como leituras por segundo.

такса de dados Medida de desempenho de bytes por unidade de tempo, como GB/segundo.

Benchmarks de E/S para sistema de arquivos e para Web

Os sistemas de arquivo, armazenados em discos, possuem um padrão de acesso diferente. Por exemplo, medições dos sistemas de arquivo do UNIX em um ambiente de engenharia demonstraram que 80% dos acessos são feitos a arquivos de menos de 10KB e que 90% de todos os acessos a arquivo são para dados com endereços seqüenciais no disco. Além do mais, 67% dos acessos eram de leitura, 27% eram de escritas, e 6% eram acessos de leitura-modificação-escrita, que lêem, modificam e depois reescrevem os dados no mesmo local. Essas medidas levaram à criação de benchmarks sintéticos de sistema arquivos. Um dos mais populares desses benchmarks possui cinco fases, usando 70 arquivos:

- *MakeDir*: constrói uma sub-árvore de diretório idêntica em estrutura à sub-árvore de diretórios dada
- *Copy*: copia cada arquivo da sub-árvore de origem para a sub-árvore de destino
- *ScanDir*: percorre recursivamente uma sub-árvore de diretórios e examina o status de cada arquivo nela
- *ReadAll*: lê cada byte de cada arquivo em uma sub-árvore uma vez
- *Make*: compila e link-edita todos os arquivos em uma sub-árvore

Como veremos na Seção 8.7, o projeto de um sistema de E/S envolve saber qual é o workload.

Além de benchmarks de processador, o SPEC oferece um benchmark de servidor de arquivos (SPECFS) e um benchmark de servidor Web (SPECWeb). O SPECFS é um benchmark para medir o desempenho do NFS (Network File System) usando um script de solicitações para servidores de arquivos; ele testa o desempenho do sistema de E/S, incluindo disco e rede, além do processador. SPECFS é um benchmark orientado a vazão, mas com requisitos importantes de tempo de resposta. SPECWeb é um benchmark de servidor Web que simula vários clientes solicitando páginas estáticas e dinâmicas de um servidor, além de clientes postando dados ao servidor.

Desempenho de E/S versus desempenho do processador

A Lei de Amdahl no Capítulo 2 nos lembra que negligenciar E/S é perigoso. Um exemplo simples demonstra isso.

IMPACTO DA E/S SOBRE O DESEMPENHOO DO SISTEMA

EXEMPLO

Suponha um benchmark que executa em 100 segundos de tempo decorrido, onde 90 segundos é o tempo de CPU e o restante é o tempo de E/S. Se o tempo de CPU melhorar em 50% por ano pelos próximos cinco anos, mas o tempo de E/S não melhorar, o quanto mais rápido nosso programa será executado no final de cinco anos?

RESPOSTA

Sabemos que

$$\begin{aligned}\text{Tempo decorrido} &= \text{tempo de CPU} + \text{tempo de E/S} \\ 100 &= 90 + \text{tempo de E/S} \\ \text{tempo de E/S} &= 10 \text{ segundos}\end{aligned}$$

Os novos tempos de CPU e os tempos decorridos resultantes são calculados na seguinte tabela:

Após n anos	Tempo de CPU	Tempo de E/S	Tempo gasto	% de tempo com E/S
0	90 segundos	10 segundos	100 segundos	10%
1	$\frac{90}{1,5} = 60$ segundos	10 segundos	70 segundos	14%
2	$\frac{60}{1,5} = 40$ segundos	10 segundos	50 segundos	20%
3	$\frac{40}{1,5} = 27$ segundos	10 segundos	37 segundos	27%
4	$\frac{27}{1,5} = 18$ segundos	10 segundos	28 segundos	36%
5	$\frac{18}{1,5} = 12$ segundos	10 segundos	22 segundos	45%

A melhoria no desempenho da CPU durante cinco anos é

$$\frac{90}{12} = 7,5$$

No entanto, a melhoria no tempo decorrido é de apenas

$$\frac{100}{22} = 4,5$$

e o tempo de E/S aumentou de 10% para 45% do tempo decorrido.

As seguintes afirmativas são verdadeiras ou falsas? Ao contrário dos benchmarks de processador, os benchmarks de E/S

Verifique você mesmo

1. concentram-se na vazão, em vez da latência
2. podem exigir que os dados definam a escala em tamanho ou número de usuários para conseguir os marcos de desempenho
3. venham de organizações, em vez de indivíduos

8.7 Projetando um sistema de E/S

Existem dois tipos principais de especificação que os projetistas encontram nos sistemas de E/S: restrições de latência e restrições de largura de banda. Nos dois casos, o conhecimento do padrão de tráfego afeta o projeto e a análise.

As restrições de latência envolvem garantir que a latência para completar uma operação de E/S esteja limitada por uma certa quantidade. No caso simples, o sistema pode ser descarregado, e o projetista também precisa garantir que algum limite de latência seja realizado, pois isso é crítico para a aplicação ou porque o dispositivo precisa receber certo serviço garantido para impedir erros. Os exemplos disso são semelhantes à análise que vimos na seção anterior. Da mesma forma, determinar a latência de um sistema não carregado é relativamente fácil, pois envolve rastrear o caminho da operação de E/S e somar as latências individuais.

Encontrar a latência média (ou a distribuição da latência) sob uma carga é um problema muito mais complexo. Esses problemas são resolvidos ou por teoria de filas (quando o comportamento das solicitações do workload e os tempos de atendimento de E/S podem ser aproximados por distribuições simples) ou por simulação (quando o comportamento dos eventos de E/S é complexo). Os dois tópicos estão além do escopo deste texto.

Projetar um sistema de E/S para atender a um conjunto de restrições de largura de banda dado um workload é o outro problema típico que os projetistas enfrentam. Como alternativa, o projetista pode receber um sistema de E/S parcialmente configurado e ser solicitado a balancear o sistema para manter a largura de banda máxima alcançável conforme ditado pela parte pré-configurada do sistema. Esse último problema de projeto é uma versão simplificada do primeiro.

A técnica geral para projetar tal sistema é a seguinte:

1. Encontrar o elo mais fraco no sistema de E/S, que é o componente no caminho da E/S que restringirá o projeto. Dependendo do workload, esse componente pode estar em qualquer lugar, incluindo na CPU, no sistema de memória, no barramento backplane, nos controladores de E/S ou nos dispositivos. Os limites de workload e de configuração podem ditar onde está localizado o elo mais fraco.
2. Configurar esse componente para sustentar a largura de banda exigida.
3. Determinar os requisitos para o restante do sistema e configurá-los para dar suporte a essa largura de banda.

O modo mais fácil de entender essa metodologia é com um exemplo.

PROJETO DO SISTEMA DE E/S

EXEMPLO

Considere o seguinte sistema computacional:

- Uma CPU que sustenta 3 bilhões de instruções por segundo e tem uma média de 100.000 instruções do sistema operacional por operação de E/S
- Um barramento backplane de memória capaz de sustentar uma velocidade de transferência de 1.000MB/seg
- Controladoras SCSI Ultra320 com uma velocidade de transferência de 320MB/seg e acomodando até 7 discos
- Unidades de disco com uma largura de banda de leitura/escrita de 75MB/seg e um seek médio mais latência rotacional de 6ms

Se o workload consiste em 64KB de leituras (onde o bloco é seqüencial em uma trilha) e o programa do usuário precisa de 200.000 instruções por operação de E/S, encontre a taxa de E/S sustentável máxima e a quantidade de discos e controladoras SCSI exigidas. Suponha que as leituras sempre possam ser feitas em um disco ocioso, se existir (por exemplo, ignore os conflitos de disco).

RESPOSTA

Os dois componentes fixos do sistema são o barramento de memória e a CPU. Primeiro, vamos encontrar a velocidade de E/S que esses dois componentes podem sustentar e determinar quais deles é o gargalo. Cada E/S utiliza 200.000 instruções de usuário e 100.000 instruções do sistema operacional, de modo que

Velocidade de E/S máxima da CPU =

$$\frac{\text{Velocidade execução instrução}}{\text{Instruções por ES}} = \frac{3 \times 10^9}{(200 + 100) \times 10^3} = 10.000 \frac{\text{E/Ss}}{\text{segundo}}$$

Cada E/S transfere 64KB, de modo que

$$\text{Taxa máxima de E/S do barramento} = \frac{\text{Largura de banda do barramento}}{\text{Bytes por E/S}}$$

$$= \frac{1000 \times 10^6}{64 \times 10^3} = 15.625 \frac{\text{E/Ss}}{\text{segundo}}$$

A CPU é o gargalo, de modo que agora podemos configurar o restante do sistema para trabalhar no nível ditado pela CPU, 10.000E/Ss por segundo.

Vamos determinar quantos discos precisamos para poder acomodar 10.000E/Ss por segundo. Para encontrar o número de discos, primeiro encontramos o tempo por operação de E/S no disco:

$$\text{Tempo por E/S no disco} = \text{Seek} + \text{tempo de rotação} + \text{tempo de transferência}$$

$$= 6 \text{ ms} + \frac{64\text{KB}}{75\text{MB/seg}} = 6,9\text{ms}$$

Assim, cada disco pode completar 1000ms/6,9ms ou 146E/Ss por segundo. Para saturar a CPU são necessárias 10.000E/Ss por segundo, ou $10.000/146 = 69$ discos.

Para calcular o número de barramentos SCSI, precisamos verificar a taxa de transferência média por disco para ver se podemos saturar o barramento, o que é dado por

$$\text{Taxa de transferência} = \frac{\text{Tamanho da transferência}}{\text{Tempo de transferência}} = \frac{64\text{KB}}{6,9\text{ms}} \approx 9,56\text{MB/seg}$$

O número máximo de discos por barramento SCSI é 7, que não saturará esse barramento. Isso significa que precisaremos de $69/7$, ou 10 barramentos e controladoras SCSI.

Observe o número significativo de suposições simplificadas necessárias para realizar esse exemplo. Na prática, muitas dessas simplificações poderiam não ser verdadeiras para aplicações críticas com uso intenso de E/S (como bancos de dados). Por esse motivo, a simulação normalmente é a única maneira realista de prever o desempenho da E/S de um workload realista.

8.8 Vida real: uma câmera digital

As câmeras digitais são basicamente computadores embutidos com armazenamento removível, gravável, não-volátil e dispositivos de E/S interessantes. A Figura 8.14 mostra nosso exemplo.

Quando ligado, o microprocessador primeiro executa diagnósticos em todos os componentes e escreve quaisquer mensagens de erro na tela de cristal líquido (LCD) na parte traseira da câmera. Essa câmera usa uma tela LCD colorida de polissilício TFT de baixa temperatura com 4,6cm. Quando os fotógrafos tiram fotos, primeiro eles mantêm o obturador aberto até a metade, para que o microprocessador possa fazer uma leitura da luz. O microprocessador mantém o obturador aberto para receber a luz necessária, capturada por um dispositivo de carga acoplada (CCD) como pixels vermelhos, verdes e azuis.

Para a câmera da Figura 8.14, o CCD é um chip de 1,2cm, com 1.360×1.024 pixels, e varredura progressiva. Os pixels são varridos linha por linha e depois passados por rotinas para equilíbrio de branco, cor e correção de serrilhado, e depois armazenados em um buffer de quadro de 4MB. O próximo passo é compactar a imagem em um formato padrão, como JPEG, e armazená-la na memória Flash removível. O fotógrafo escolhe a compactação, nessa câmera chamada de fina ou normal, com a razão de compactação de 10 a 20 vezes. Uma imagem compactada na qualidade fina ocupa menos de 0,5MB, e uma imagem compactada na qualidade normal ocupa cerca de 0,25MB. O microprocessador, então, atualiza a tela do LCD para mostrar que existe espaço para mais uma figura.



FIGURA 8.14 A câmera Sanyo VPC-SX500 com cartão de memória Flash e Microdrive IBM. Embora as câmeras mais novas ofereçam mais pixels por imagem, os princípios são os mesmos. Essa câmera digital de 1.360 x 1.024 pixels armazena imagens usando memória CompactFlash ou usando um Microdrive IBM. Essa foto foi tirada usando um microdrive de 340MB e uma memória CompactFlash de 80MB. Como mostra a Figura 8.15, em 2004, as capacidades chegavam até 4GB. Ela possui 11cm de largura x 6cm de altura x 4cm de profundidade e pesa em torno de 200 gramas. Além de tirar uma foto parada e convertê-la para o formato JPEG a cada 0,9 segundos, ela pode registrar um clipe de vídeo Quick Time em tamanho VGA (640 x 480). Uma vantagem tecnológica é o uso de um sistema personalizado em um chip para reduzir o tamanho e o consumo, de modo que a câmera só precisa de duas pilhas pequenas para operar, ao contrário das quatro em outras câmeras digitais.

Embora o parágrafo anterior aborde os fundamentos de uma câmera digital, existem muito mais recursos incluídos: mostrar as imagens gravadas na tela LCD colorida; modo sleep para economizar pilha; monitoração da carga da pilha; buffer para permitir o registro de uma seqüência rápida de imagens não compactadas; e, nessa câmera, gravação de vídeo usando o formato MPEG e gravação de áudio usando o formato WAV.

Essa câmera permite que o fotógrafo use um disco Microdrive no lugar da memória CompactFlash. A Figura 8.15 compara a memória CompactFlash e o Microdrive IBM.

O pacote padrão CompactFlash foi proposto pela Sandisk Corporation em 1994 para os cartões PCMCIA-ATA de PCs portáteis. Por seguir a interface ATA, ele simula uma interface de disco incluindo comandos de seek, trilhas lógicas e assim por diante. Ele inclui uma controladora embutida para dar suporte a muitos tipos de memória Flash e para ajudar com o rendimento do chip para memórias Flash, mapeando blocos com defeito.

O cérebro eletrônico dessa câmera é um computador embutido com diversas funções especiais embutidas no chip. A Figura 8.16 mostra o diagrama em blocos de um chip semelhante ao utilizado na câmera. Tais chips foram chamados de *sistemas em um chip* (SOC) porque integram em um único chip todas as partes encontradas em uma pequena placa de circuito impresso no passado. SOC geralmente reduz o tamanho e diminui o consumo em comparação com soluções menos integradas. O fabricante afirma que o SOC permite que a câmera opere com metade do número de pilhas e ofereça um tamanho menor do que as câmeras dos concorrentes.

Para aumentar o desempenho, ele possui dois barramentos. O barramento de 16 bits é para os dispositivos de E/S muito mais lentos: Smart Media interface, memória de programa e dados, e DMA. O barramento de 32 bits é para a SDRAM, o processador de sinais (que está conectado ao CCD), o

Características	Sandisk Type I CompactFlash SDCFB-128-768	Sandisk Type II CompactFlash SDCFB-1000-768	Hitachi 4 GB Microdrive DSCM-10340
Capacidade de dados formatados (MB)	128	1000	4000
Bytes por setor	512	512	512
Taxa de transferência de dados (MB/sec)	4 (rajada)	4 (rajada)	4-7
Velocidade do link para o buffer (MB/sec)	6	6	33
Potência standby/operando (W)	0.15/0.66	0.15/0.66	0.07/0.83
Tamanho: altura x largura x profundidade (cm)	3,6 x 4,3 x 0,3	3,6 x 4,3 x 0,3	3,6 x 4,3 x 0,3
Peso em gramas	11,4	13,5	16
Ciclos de escrita antes de desgaste do setor	300.000	300.000	não aplicável
Tempo médio entre falhas – MTBF (horas)	> 1.000.000	> 1.000.000	(ver legenda)
Melhor preço (2004)	US\$40	US\$200	US\$480

FIGURA 8.15 Características de três alternativas de armazenamento para câmeras digitais. A Hitachi corresponde ao tamanho Type II no Microdrive, enquanto a placa CompactFlash utiliza esse espaço para incluir muito mais chips Flash. A Hitachi não indica o MTTF para as unidades de 2,5cm, mas a vida de serviço é de cinco anos ou 8.800 horas ligadas, o que vier primeiro. Elas giram em 3.600RPM e possuem 12ms de tempo de seek.

codificador Motion JPEG, e o codificador NTSC/PAL (que está conectado ao LCD). Diferente dos microprocessadores de desktop, observe a grande variedade de barramentos de E/S que esse chip precisa integrar. A MPU RISC de 32 bits é um projeto proprietário e executa em 28,8MHz, a mesma velocidade de clock dos barramentos. Esse chip de 700mW contém 1,8M transistores em um die de 10,5 x 10,5mm implementado por meio de um processo de 0,35 micron.

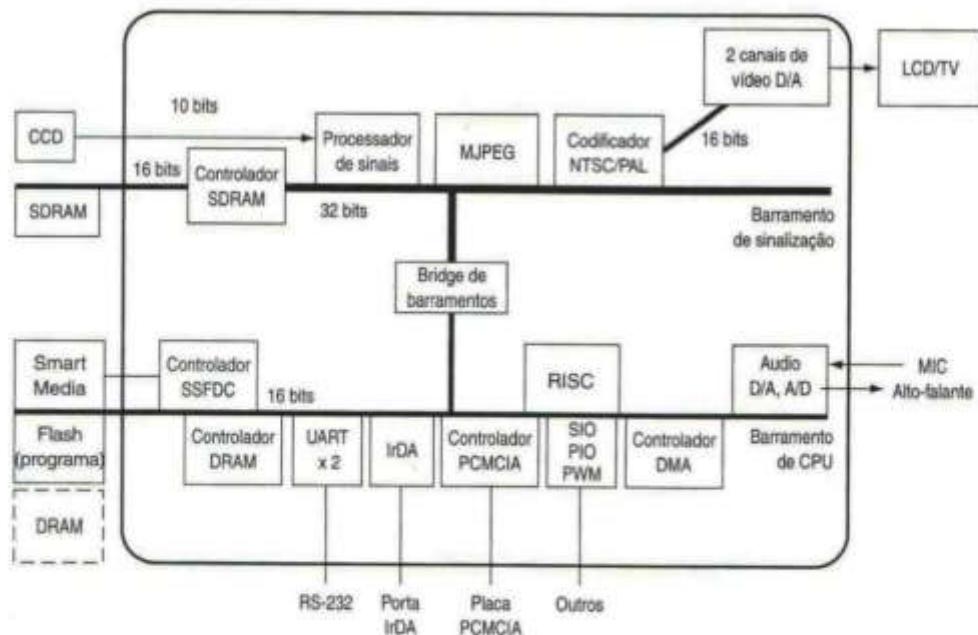


FIGURA 8.16 O sistema em um chip (SOC) encontrado nas câmeras digitais Sanyo. Esse diagrama em blocos é para o processador do SOC na câmera da Figura 8.14. O SOC sucessor, chamado Super Advanced IC, utiliza três barramentos no lugar de dois, opera a 60MHz, consome 800mW e contém 3,1M transistores em um die de 10,2 x 10,2mm usando um processo de 0,35 micron. Observe que esse sistema embutido possui o dobro de transistores do microprocessador mais poderoso, de alto desempenho em 1990! O SOC na figura é limitado a processar 1.024 x 768 pixels, mas seu sucessor admite 1.360 x 1.024 pixels. (Ver Okada, Matsuda, Yamada e Kobayashi [1999]).

8.9

Falácia e armadilhas

Falácia: o tempo médio para falha (MTTF) indicado para discos é 1.200.000 horas ou quase 140 anos, de modo que os discos praticamente nunca falham.

As práticas de marketing atuais dos fabricantes de disco podem enganar os usuários. Como esse MTTF é calculado? No inicio do processo, os fabricantes colocam milhares de discos em uma sala, os colocam para trabalhar por alguns meses, e contam a quantidade que falha. Eles calculam o MTTF como o número total de horas que os discos estiveram acumuladamente ativos dividido pelo número que falhou.

Um problema é que esse número é muito superior ao tempo de vida de um disco, que normalmente é cinco anos ou 43.000 horas. Para esse grande MTTF fazer algum sentido, esses fabricantes argumentam que o cálculo corresponde a um usuário que compra um disco, e depois continua substituindo o disco a cada cinco anos – o tempo de vida planejado do disco. A reivindicação é que, se muitos clientes (e seus bisnetos) fizessem isso para o próximo século, na média eles substituiriam um disco 27 vezes antes de uma falha, ou cerca de 140 anos.

Uma medida mais útil seria a porcentagem de discos que falham. Considere 1.000 discos com um MTTF de 1.200.000 horas e que os discos sejam usados 24 horas por dia. Se você substituisse os discos que falharam por um novo com as mesmas características de confiabilidade, o número que falharia durante cinco anos (43.800 horas) é

$$\text{Discos falhos} = 1.000 \text{ unidades} \times 43.800 \text{ horas/unidade} / 1.200.000 \text{ horas/falha} = 36$$

Explicando de uma forma alternativa, 3,6% falharia no periodo de 5 anos.

Armadilha: usar uma taxa de transferência de pico de uma parte do sistema de E/S para fazer projeções de desempenho ou comparações de desempenho.

Muitos dos componentes de um sistema de E/S, desde os dispositivos até os controladores e barramentos, são especificados por meio de suas larguras de banda de pico. Na prática, essas medidas de largura de banda de pico normalmente são baseadas em suposições irrealistas sobre o sistema ou não são alcançáveis, devido a outras limitações do sistema. Por exemplo, cotando o desempenho do barramento, a velocidade de transferência de pico às vezes é especificada usando um sistema de memória impossível de criar. Para sistemas em rede, o overhead do software para iniciar a comunicação é ignorado.

O barramento PCI de 32 bits, 33MHz, possui uma largura de banda de pico de cerca de 133MB/seg. Na prática, até mesmo para transferências longas, é difícil sustentar mais do que cerca de 80MB/seg para sistemas de memória reais. Como já dissemos, os usuários de redes sem fio normalmente conseguem apenas cerca de um terço da largura de banda de pico.

A Lei de Amdahl também nos lembra que a vazão de um sistema de E/S será limitada pelo componente de menor desempenho no caminho de E/S.

Falácia: o armazenamento em disco magnético está com os dias contados e logo será substituído.

Isso é tanto uma falácia quanto uma armadilha. Essas afirmações foram feitas constantemente nos últimos 20 anos, embora a seqüência de alternativas fracassadas nos últimos anos pareça ter reduzido o nível de afirmações para a morte do armazenamento magnético. Entre os competidores malsucedidos estão as memórias de bolha magnética, armazenamento óptico e armazenamento holográfico. Nenhum desses sistemas conseguiu a combinação de características que favorecem os discos magnéticos: alta confiabilidade, não-volatilidade, baixo custo, tempo de acesso razoável e melhoria rápida. A tecnologia de armazenamento magnético continua a melhorar no mesmo ritmo – ou mais rápido – que esteve sustentando pelos últimos 25 anos.

Armadilha: usar fitas magnéticas para o backup de discos.

Mais uma vez, isso é uma falácia e uma armadilha.

As fitas magnéticas têm feito parte dos sistemas de computador desde tanto quanto os discos, pois utilizam tecnologia semelhante aos discos e, por isso, historicamente têm seguido as mesmas melhorias na densidade. A diferença de custo-desempenho histórica entre discos e fitas é baseada em um disco selado, rotativo, com menor tempo de acesso do que o acesso seqüencial à fita, mas os spools removíveis de fita magnética significam que muitas fitas podem ser usadas por leitora e que elas podem ser muito longas, de modo que possuem alta capacidade. Logo, no passado, uma única fita magnética poderia manter o conteúdo de muitos discos, e por ser de 10 a 100 vezes mais barata por gigabyte do que os discos, esse era um meio de backup útil.

A alegação foi de que as fitas magnéticas precisam acompanhar os discos, pois as inovações nos discos precisam ajudar as fitas. Essa alegação foi importante porque as fitas eram um pequeno mercado e não poderiam dispor de um grande esforço de pesquisa e desenvolvimento separado. Um motivo para o mercado ser pequeno é que os proprietários de desktop geralmente não fazem backup de discos em fita, e assim, enquanto os desktops são um grande mercado para discos, eles são um pequeno mercado para fitas.

Infelizmente, o maior mercado levou os discos a melhorarem muito mais rapidamente do que as fitas. Entre 2000 a 2002, o disco muito mais popular era maior do que a maior fita popular. Nesse mesmo espaço de tempo, o preço por gigabyte de discos ATA caiu para menos do que o das fitas. Os defensores da fita agora alegam que elas possuem requisitos de compatibilidade que não são impostos sobre os discos; as leitoras de fita precisam ler ou escrever a geração atual e anterior de fitas e precisam ler as quatro últimas gerações de fitas. Como os discos são sistemas fechados, as cabeças de disco só precisam ler os pratos embutidos, e essa vantagem explica por que os discos estão melhorando muito mais rapidamente.

Hoje, algumas organizações retiraram as fitas, usando redes e discos remotos para replicar os dados geograficamente. Os locais são selecionados de modo que os desastres não prejudiquem os dois locais, permitindo um tempo de recuperação instantâneo. (Um tempo de recuperação longo é outra desvantagem séria da natureza serial das fitas magnéticas.) Essa solução depende dos avanços na capacidade do disco e na largura de banda da rede, para fazer sentido economicamente, mas esses dois estão recebendo um investimento muito maior e, portanto, possuem registros de realização recentes melhores do que a fita.

Falácia: um barramento de 100MB/seg pode transferir 100MB de dados em 1 segundo.

Primeiro, você em geral não pode usar 100% de qualquer recurso do computador. Para um barramento, você ficaria satisfeito em conseguir 70% a 80% da largura de banda de pico. O tempo para enviar o endereço, o tempo para confirmar os sinais e os atrasos enquanto se espera para usar um barramento ocupado estão entre os motivos para você não poder usar 100% de um barramento.

Segundo, a definição de um megabyte de armazenamento e um megabyte por segundo de largura de banda não correspondem. Conforme discutimos na página 452, as medidas de largura de banda de E/S normalmente são cotadas em base 10 (ou seja, 1MB/seg = 10^6 bytes/seg), enquanto 1MB de dados normalmente é uma medida na base 2 (ou seja, 1MB = 2^{20} bytes). Qual é o significado dessa distinção? Se pudéssemos usar 100% do barramento para a transferência de dados, o tempo para transferir 100MB de dados em um barramento de 100MB/seg é, na realidade,

$$100 \times 2^{20} / 100 \times 10^6 = 1.048.576 / 1.000.000 = 1,048576 = 1,05 \text{ segundo}$$

Um erro semelhante, porém maior, é introduzido quando tratamos um gigabyte de dados transferidos ou armazenados como equivalentes, significando 10^9 versus 2^{30} bytes.

Armadilha: tentar oferecer recursos apenas dentro da rede versus fim a fim.

O problema é fornecer em um nível inferior recursos que só podem ser cumpridos no nível mais alto, satisfazendo assim apenas parcialmente à demanda da comunicação. Saltzer, Reed e Clark [1984] explicam o argumento de *fim a fim* como

A função em questão só pode ser especificada completa e corretamente com o conhecimento e a ajuda da aplicação que fica nas extremidades do sistema de comunicação. Portanto, não é possível oferecer essa função questionada como um recurso do próprio sistema de comunicação.

Seu exemplo da armadilha foi uma rede no MIT que usava vários gateways, cada qual acrescentando uma soma de verificação de um gateway para o seguinte. Os programadores da aplicação assumiram a precisão garantida pela soma de verificação, acreditando incorretamente que a mensagem estava protegida enquanto armazenada na memória de cada gateway. Um gateway tinha uma falha intermitente que trocava um par de bytes para cada milhão de bytes transferidos. Com o tempo, o código-fonte de um sistema operacional era repetidamente passado pelo gateway, adulterando, dessa forma, o código. A única solução foi corrigir os arquivos-fonte infectados, comparando as listagens em papel e reparando o código manualmente! Se as somas de verificação tivessem sido calculadas e verificadas pela aplicação rodando nos sistemas na ponta, a segurança teria sido garantida.

No entanto, existe uma função útil para verificações intermediárias, desde que a verificação fim a fim esteja disponível. A verificação fim a fim pode mostrar que *algo* está errado entre dois nós, mas não aponta onde se encontra o problema. As verificações intermediárias podem descobrir *o que* está errado. Você precisa de ambos para reparar.

Armadilha: mover funções da CPU para o processador de E/S, esperando melhorar o desempenho sem uma análise cuidadosa.

Existem muitos exemplos dessa armadilha pegando as pessoas, embora os processadores de E/S, quando usados corretamente, certamente podem melhorar o desempenho. Um caso frequente dessa falácia é o uso de interfaces de E/S inteligentes que, devido ao maior overhead para configurar uma requisição de E/S, pode ter uma latência pior do que uma atividade de E/S controlada pelo processador (embora, se o processador for liberado suficientemente, a vazão do sistema ainda possa aumentar). Constantemente, o desempenho cai quando o processador de E/S tem um desempenho muito inferior ao do processador principal. Como consequência, uma quantidade pequena do tempo de processador principal é substituída por uma quantidade maior de tempo do processador de E/S. Os projetistas de estações de trabalho têm visto esses dois fenômenos repetidamente.

Myer e Sutherland [1968] escreveram um artigo clássico sobre a escolha entre complexidade e desempenho nos controladores de E/S. Apanhando emprestado o conceito religioso da “roda da reencarnação”, eles, por fim, observaram que eram apanhados em um loop de aumentar continuamente a potência de um processador de E/S até que ele precisasse do seu próprio co-processador mais simples:

Enfrentamos a tarefa começando com um esquema simples e depois acrescentando comandos e recursos que achamos que melhorariam a potência da máquina. Gradualmente, o processador [de vídeo] tornava-se mais complexo... Finalmente, o processador de vídeo ficou semelhante a um computador completo, com alguns recursos gráficos especiais. E depois aconteceu uma coisa estranha. Sentimo-nos compelidos a acrescentar ao processador um segundo processador subsidiário que, por si só, começou a aumentar em complexidade. Foi então que descobrimos a verdade perturbadora. Projetar um processador de vídeo pode se tornar um processo cílico sem fim. Na verdade, descobrimos que o processo era tão frustrante que passamos a chamá-lo de “roda da reencarnação”.

8.10 Comentários finais

Os sistemas de E/S são avaliados em diversas características diferentes: confiança; a variedade de dispositivos de E/S aceitos; o número máximo de dispositivos de E/S; custo; e desempenho, medidos tanto em latência quanto em vazão. Esses objetivos levam a esquemas bastante variados para interface de dispositivos de E/S. Nos sistemas inferiores e intermediários, o DMA com buffer provavelmente será o mecanismo de transferência dominante. Nos sistemas de alto nível, a latência e a largura de banda podem ser ambos importantes, e o custo pode ser secundário. Vários caminhos para dispositivos de E/S com buffer limitado normalmente caracterizam sistemas de E/S de alto nível. Em geral, ser capaz de acessar os dados em um dispositivo de E/S a qualquer tempo (alta disponibilidade) torna-se mais importante quando os sistemas crescem. Como resultado, a redundância e os mecanismos de correção de erros tornam-se mais e mais prevalentes enquanto ampliamos o sistema.

As demandas de armazenamento e rede estão crescendo em velocidades sem precedentes, em parte devido às demandas crescentes para que toda a informação esteja na ponta dos seus dedos. Uma estimativa é que a quantidade de informação criada em 2002 foi de 5 exabytes – equivalente a 500.000 cópias do texto da Biblioteca do Congresso dos Estados Unidos –, e essa quantidade total de informações no mundo dobrou nos últimos três anos [Lyman e Varian, 2003].

As direções futuras da E/S incluem expandir o alcance das redes com e sem fio, com quase todo dispositivo potencialmente tendo um endereço IP, e a transformação contínua de barramentos paralelos em redes e switches seriais. Todavia, a consolidação no setor de discos poderá levar a um atraso na melhoria da capacidade de disco para as taxas anteriores, que dobraram a cada ano entre 2000 e 2004.

Entendendo o desempenho dos programas

O desempenho de um sistema de E/S, seja ele medido por largura de banda ou latência, depende de todos os elementos no caminho entre o dispositivo e a memória, incluindo o sistema operacional que gera os comandos de E/S. A largura de banda desses barramentos, a memória e o dispositivo determinam a velocidade de transferência máxima do dispositivo ou para ele. De modo semelhante, a latência depende da latência do dispositivo, junto com qualquer latência imposta pelo sistema de memória ou barramentos. A largura de banda efetiva e a latência de resposta também dependem de outras requisições de E/S que podem causar disputa por algum recurso no caminho. Finalmente, o sistema operacional é um gargalo. Em alguns casos, o sistema operacional leva muito tempo para entregar uma solicitação de E/S de um programa de usuário para um dispositivo de E/S, levando a uma alta latência. Em outros casos, o sistema operacional efetivamente limita a largura de banda de E/S, devido às limitações no número de operações de E/S simultâneas que ele pode admitir.

Lembre-se de que, embora o desempenho possa ajudar a vender um sistema de E/S, os usuários, em sua maioria, exigem confiabilidade e capacidade dos seus sistemas de E/S.

8.11 Perspectiva histórica e leitura adicional

A história dos sistemas de E/S é fascinante. Esta Seção 8.11 oferece um breve histórico dos discos magnéticos, RAID, bancos de dados, a Internet, a World Wide Web e como a Ethernet continua a triunfar sobre seus desafiantes.

8.12 Exercícios

8.1 [10] <§§8.1-8.2> Aqui estão dois sistemas de E/S diferentes intencionados para uso no processamento de transações:

- O sistema A pode aceitar 1.500 operações de E/S por segundo.
- O sistema B pode aceitar 1.000 operações de E/S por segundo.

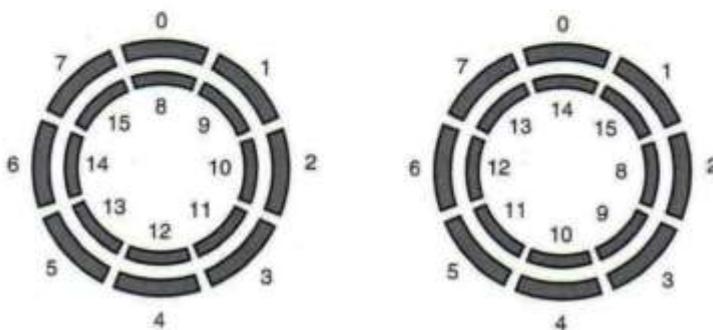
Os sistemas utilizam o mesmo processador que executa 500 milhões de instruções por segundo. Suponha que cada transação exija 5 operações de E/S e que cada operação de E/S exija 10.000 instruções. Ignorando o tempo de resposta e supondo que as transações possam ser superpostas arbitrariamente, qual é a velocidade máxima em transações-por-segundo que cada máquina pode sustentar?

8.2 [15] <§§8.1-8.2> A latência de uma operação de E/S para os dois sistemas do Exercício 8.1 é diferente. A latência para uma E/S no sistema A é igual a 20ms, enquanto para o sistema B a latência é de 18ms para as 500 primeiras E/S por segundo e 25ms por E/S para cada E/S entre 500 e 1.000 E/S por segundo. No workload, cada 10^a transação depende da transação imediatamente anterior e precisa esperar pelo seu término. Qual é a velocidade de transação máxima que ainda permite que cada transação complete em 1 segundo e que não excede a largura de banda de E/S da máquina? (Para simplificar, considere que todas as requisições de transação chegam no início de um intervalo de 1 segundo.)

8.3 [5] <§§8.1-8.2> Suponha que queiramos usar um laptop para enviar 100 arquivos de aproximadamente 40MB cada para outro computador por uma conexão sem fio de 5Mb/seg. A bateria do laptop atualmente mantém 100.000 joules de energia. A placa de rede sem fio sozinha consome 5 watts enquanto transmite, enquanto o restante do laptop sempre consome 35 watts. Antes de cada transferência de arquivo precisamos de 10 segundos para escolher qual arquivo enviar. Quantos arquivos completos podemos transferir antes que a bateria do laptop chegue a zero?

8.4 [10] <§§8.1-8.2> Considere o consumo de energia do disco rígido do laptop do Exercício 8.3. Suponha que ele não seja mais constante, mas varie entre 6 watts quando está girando e 1 watt quando não está girando. A potência consumida pelo laptop fora o disco rígido e a placa de rede sem fio são 32 watts constantes. Suponha que a taxa de transferência do disco rígido seja de 50MB/seg, seu atraso antes de poder começar a transferir seja de 20ms, e que todas as outras vezes ele não gira. Quantos arquivos completos podemos transferir antes que a bateria do laptop chegue a zero? De quanta energia precisaríamos para enviar todos os 100 arquivos? (Considere que a placa de rede sem fio não pode enviar dados até que eles estejam na memória.)

8.5 [5] <§8.3> O seguinte diagrama simplificado mostra duas maneiras em potencial de numerar os setores dos dados em um disco (apenas duas trilhas são mostradas e cada trilha possui oito setores). Supondo que as leituras típicas sejam contíguas (por exemplo, todos os 16 setores são lidos em ordem), que forma de numeração dos setores provavelmente resultará no desempenho mais alto? Por quê?



8.6 [20] <§8.3> Neste exercício, executaremos um programa para avaliar o comportamento de uma unidade de disco. Os setores de disco são endereçados seqüencialmente dentro de uma trilha, as trilhas seqüencialmente dentro de cilindros, e os cilindros seqüencialmente dentro do disco. É difícil determinar o tempo de troca de cabeça e o tempo de troca de cilindro, devido aos efeitos da rotação. Até mesmo determinar a contagem de pratos, setores/trilha e atraso rotacional é difícil com base na observação dos workloads de disco típicos.

A chave é fatorar os efeitos rotacionais do disco fazendo buscas consecutivas a setores individuais com endereços que diferem por uma quantidade linearmente crescente, começando com 0, 1, 2 e assim por diante. O algoritmo Skippy, do trabalho de Nisha Talagala e seus colegas da U.C. Berkeley [2000], é

```
fd = open("dispositivo do disco em modo raw");
for (i = 0; i < measurements; i++) {
    //temporize a seqüência seguinte e imprima <i, tempo>
    lseek(fd, i * SINGLE_SECTOR, SEEK_CUR);
    write(fd, buffer, SINGLE_SECTOR);
}
close(fd);
```

O algoritmo básico salta pelo disco, aumentando a distância de seek em um setor antes de cada escrita, e atualiza a distância e o tempo para cada escrita. A interface do dispositivo em modo raw é usada para evitar otimizações. SINGLE_SECTOR é o tamanho de um único setor em bytes. O argumento SEEK_CUR de lseek move o ponteiro do arquivo por uma quantidade relativa ao ponteiro atual. Um relatório técnico descrevendo o Skippy e dois outros benchmarks de unidade disco (executados em segundos ou minutos, em vez de horas ou dias) está em <http://sunsite.berkeley.edu/Dienst/UI/2.0/Describe/ncstrl.ucb/CSD-99-1063>.

Execute o algoritmo Skippy em uma unidade de disco à sua escolha.

- Qual é o número de cabeças?
- O número de pratos?
- Qual é a latência rotacional?
- Qual é o tempo de troca de cabeça (o tempo para trocar a cabeça que está lendo de uma superfície do disco para outra sem mover o braço; ou seja, no mesmo cilindro)?
- Qual é o tempo de troca de cilindro? (Esse é o tempo para mover o braço para o próximo cilindro seqüencial.)

8.7 [20] <§8.3> A Figura 8.17 mostra a saída da execução do benchmark Skippy em um disco.

- Qual é o número de cabeças?
- O número de pratos?
- Qual é a latência rotacional?
- Qual é o tempo de troca de cabeça (o tempo para trocar a cabeça que está lendo de uma superfície do disco para outra sem mover o braço; ou seja, no mesmo cilindro)?
- Qual é o tempo de troca de cilindro? (Esse é o tempo para mover o braço para o próximo cilindro seqüencial.)

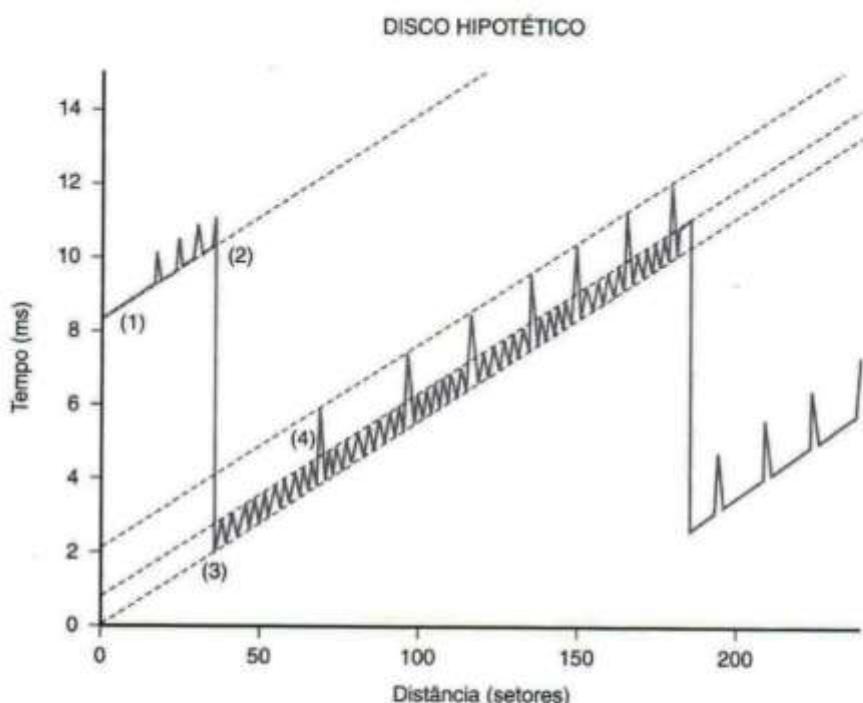


FIGURA 8.17 Saída de exemplo do Skippy para um disco hipotético.

8.8 [10] <§8.3> Considere dois sistemas de disco RAID que pretendem armazenar 10 terabytes de dados (sem contar qualquer redundância). O sistema A utiliza a tecnologia RAID 1, e o sistema B utiliza tecnologia RAID 5 com quatro discos em um “grupo de proteção”.

- Quantos terabytes de armazenamento a mais são necessários no Sistema A do que no Sistema B?
- Suponha que uma aplicação escreva um bloco de dados no disco. Se a leitura ou a escrita de um bloco leva 30ms, quanto tempo a escrita levará no Sistema A no pior caso? E no sistema B no pior caso?
- O Sistema A é mais confiável do que o Sistema B? Por quê ou por que não?

8.9 [15] <§8.3> O que pode acontecer com um sistema RAID 5 se a energia faltar entre a atualização da escrita no bloco de dados e a atualização da escrita no bloco de verificação, de modo que somente um dos dois seja gravado com sucesso? O que poderia ser feito para evitar isso desde o início?

8.10 [5] <§8.3> A velocidade da luz é de aproximadamente 3×10^8 metros por segundo, e os sinais elétricos atravessam em cerca de 50% dessa velocidade em um condutor. Quando o termo *alta velocidade* é aplicado a uma rede, é a largura de banda que é mais alta, e não necessariamente a velocidade dos sinais elétricos. Qual fator é o “tempo de vôo” real para os sinais elétricos? Considere dois computadores afastados em 20 metros e dois computadores afastados em 2.000 quilômetros. Compare seus resultados com as latências informadas no exemplo da ■ Seção 8.3.

8.11 [5] <§8.3> O número de bytes em trânsito em uma rede é definido como o tempo de vôo (descrito no Exercício 8.10) multiplicado pela largura de banda entregue. Calcule o número de bytes em trânsito para as duas redes descritas no Exercício 8.10, considerando uma largura de banda entregue de 6MB/seg.

8.12 [5] <§8.3> Uma agência secreta monitora simultaneamente 100 conversas de telefone celular e multiplexa os dados em uma rede com uma largura de banda de 5MB/seg e uma latência de overhead de 150s por mensagem de 1KB. Calcule o tempo de transmissão por mensagem e determine se existe largura de banda suficiente para dar suporte a essa aplicação. Suponha que os dados da conversa telefônica consistam em 2 bytes amostrados a uma taxa de 4KHz.

8.13 [5] <§8.3> A rede sem fio possui uma taxa de erro de bit (BER – Bit Error Rate) muito maior do que a rede com fio. Um modo de lidar com uma BER mais alta é usar um código de correção de erro (ECC – Error Correcting Code) nos dados transmitidos. Um ECC muito simples é triplicar cada byte, codificando cada zero como 000 e cada um como 111. Quando um padrão de 3 bits é recebido, o sistema escolhe o bit original mais provável.

- Se o sistema recebeu 001, qual é o valor mais provável do bit original?
- Se 000 foi enviado, mas um erro de bit duplo fizer com que ele seja recebido como 110, o que o receptor considerará o valor do bit original?
- Quantos erros de bit esse ECC simples pode corrigir?
- Quantos erros de bit esse ECC pode detectar?
- Se 1 entre cada 100 bits enviados pela rede estiver incorreto, que porcentagem de erros de bit um receptor usando esse ECC não detectaria?

8.14 [5] <§8.3> Existem dois tipos de paridade: par e ímpar. Uma word binária com paridade par e nenhum erro terá um número par de 1s nela, enquanto uma word com paridade ímpar e nenhum erro terá um número ímpar de 1s nela. Calcule o bit de paridade para cada uma das seguintes words de 8 bits se a paridade par for usada:

- 01100111
- 01010101

8.15 [10] <§8.3>

- Se um sistema utiliza paridade par, e a word 0111 for lida do disco, podemos dizer que existe um erro de único bit?
- Se um sistema utiliza paridade ímpar, e a word 0101 aparece no barramento processador-memória, suspeitamos que ocorreu um erro de único bit. Podemos dizer em qual bit o erro ocorre? Por quê ou por que não?
- Se um sistema utiliza paridade par e a word 0101 aparece no barramento processador-memória, podemos saber se existe um erro de duplo bit?

8.16 [10] <§8.3> Um programa realiza repetidamente um processo em três etapas: ele lê um bloco de 4KB de dados do disco, faz algum processamento nesses dados e depois escreve o resultado como outro bloco de 4KB em outro lugar no disco. Cada bloco é contíguo e localizado aleatoriamente em uma única trilha no disco. A unidade de disco gira a 10.000RPM, possui um tempo de seek médio de 8ms e uma velocidade de transferência de 50MB/seg. O overhead do controlador é de 2ms. Nenhum outro programa está usando o disco ou o processador, e não existe sobreposição de operação de disco com processamento. A etapa de processamento utiliza 20 milhões de ciclos de clock, e a velocidade do clock é de 5GHz. Qual é a velocidade geral do sistema em blocos processados por segundo?

8.17 [5] <§8.4> Os protocolos de rede OSI formam uma hierarquia de camadas de abstração, criando uma interface entre aplicações de rede e os fios físicos. Isso é semelhante aos níveis de abstração utilizados na interface ISA entre o software e o hardware. Cite três vantagens do uso da abstração no projeto de protocolos de rede.

8.18 [5] <§§8.3, 8.5> Suponha que tenhamos um sistema com as seguintes características:

- Um sistema de memória e barramento dando suporte ao acesso em bloco de 4 a 16 words de 32 bits.
- Um barramento síncrono de 64 bits com clock de 200MHz, com cada transferência de 64 bits utilizando 1 ciclo de clock, e 1 ciclo de clock exigido para enviar um endereço à memória.
- Dois ciclos de clock necessários entre cada operação de barramento. (Suponha que o barramento esteja ocioso antes de um acesso.)
- Um tempo de acesso à memória de 200ns para as quatro primeiras words; cada conjunto adicional de quatro words pode ser lido em 20ns.

Suponha que os sistemas de barramento e memória descritos anteriormente sejam usados para lidar com acessos ao disco a partir de discos como aquele descrito no Exemplo da página 431. Se a E/S tiver permissão para consumir 100% do barramento e largura de banda da memória, qual é o número máximo de transferências de disco simultâneas que podem ser sustentadas para os dois tamanhos de bloco?

8.19 [5] <§8.5> No sistema descrito no Exercício 8.18, o sistema de memória exigiu 200ns para ler as quatro primeiras words, e cada quatro words adicionais exigiu 20ns. Supondo que o sistema de memória exija 150ns para ler as quatro primeiras words e 30ns para ler cada quatro words adicionais, encontre a largura de banda sustentada e a latência para uma leitura de 256 words para transferências que utilizam blocos de 4 words e para transferências que utilizam blocos de 16 words. Calcule também o número efetivo de transações de barramento por segundo para cada caso.

8.20 [5] <§8.5> O Exercício 8.19 demonstra que usar tamanhos de bloco maiores resulta em um aumento na largura de banda máxima sustentada que pode ser alcançada. Sob que condições um projetista poderia tentar favorecer tamanhos de bloco menores? Especificamente, por que um projetista escolheria um tamanho de bloco igual a 4 em vez de 16 (supondo que todas as características sejam conforme identificadas no Exercício 8.19)?

8.21 [15] <§8.5> Esta pergunta examina com mais detalhes como o aumento do tamanho de bloco para transações de barramento diminui a latência total exigida e aumenta a largura de banda máxima sustentável. No Exercício 8.19, dois tamanhos de bloco diferentes são considerados (4 words e 16 words). Calcule a latência total e a largura de banda máxima para todos os tamanhos de bloco possíveis (entre 4 e 16) e mostre os resultados graficamente. Resuma o que você aprendeu examinando seu gráfico.

8.22 [15] <§8.5> Este exercício é semelhante ao Exercício 8.21. Desta vez, fixe o tamanho do bloco em 4 e 16 (como no Exercício 8.19), mas calcule as latências e as larguras de banda para leituras de diferentes tamanhos. Especificamente, considere leituras de 4 a 256 words, e use tantos pontos de dados quantos precisar para construir um gráfico significativo. Use seu gráfico para ajudar a determinar em que ponto os tamanhos de bloco iguais a 16 resultam em uma latência reduzida quando comparados com tamanhos de bloco de 4.

8.23 [10] <§8.5> Este exercício examina uma alternativa de projeto ao sistema descrito no Exercício 8.18, que pode melhorar o desempenho das escritas. Para as escritas, suponha que todas as características relatadas no Exercício 8.18 e também o seguinte:

As 4 primeiras words são escritas a 200ns após o endereço estar disponível, e cada nova escrita leva 20ns. Considere uma transferência de barramento dos dados mais recentes a serem escritos, e que uma escrita das 4 words anteriores pode ser sobreposta.

A análise de desempenho informada no exemplo permaneceria inalterada para escritas (na realidade, pode haver algumas mudanças menores devido à necessidade de calcular códigos de detecção de erro etc., mas vamos ignorar isso). Um esquema de barramento alternativo conta com linhas de endereço e dados separadas de 32 bits. Isso permitirá que um endereço e dados sejam transmitidos no mesmo ciclo. Para essa alternativa de barramento, qual será a latência da transferência inteira de 256 words? Qual é a largura de banda sustentada? Considere tamanhos de bloco de 4 e 8 words. Quando você imagina que o esquema alternativo seria altamente favorecido?

8.24 <20> <§8.5> Considere um barramento assíncrono usado para realizar a interface de um dispositivo de E/S com o sistema de memória descrito no Exercício 8.18. Cada requisição de E/S pede 16 words de dados da memória, que, juntamente com o dispositivo de E/S, possui um barramento de 4 words. Considere o mesmo tipo de protocolo de handshaking que aparece na Figura 8.10, exceto que é estendido para que a memória possa continuar a transação enviando blocos adicionais de dados até a transação estar concluída. Modifique a Figura 8.10 (tanto as etapas quanto o diagrama) para indicar como essa transferência poderia ocorrer. Supondo que cada etapa de handshaking gaste 20ns e

o acesso à memória gaste 60ns, quanto tempo é necessário para completar uma transferência? Qual é a largura de banda sustentada máxima para esse barramento assíncrono, e como ela se compara com o barramento assíncrono do exemplo?

8.25 [1 dia-1 semana] <§§8.2-8.5> ■ **Aprofundando o aprendizado:** Escrevendo código para benchmark de desempenho de E/S

8.26 [15] <§8.5> Queremos comparar a largura de banda máxima para um barramento síncrono e um assíncrono. O barramento síncrono possui um tempo de ciclo de clock de 50ns, e cada transmissão no barramento leva 1 ciclo de clock. O barramento assíncrono exige 40ns por handshake. A parte de dados dos dois barramentos é de 32 bits de largura. Encontre a largura de banda para cada barramento quando realizar leituras de uma word de uma memória de 200ns.

8.27 [20] <§8.5> Suponha que tenhamos um sistema com as seguintes características:

1. Um sistema de memória e barramento com suporte ao acesso em bloco de 4 a 16 words de 32 bits.
2. Um barramento síncrono de 64 bits com clock de 200MHz, com cada transferência de 64 bits ocupando 1 ciclo de clock, e 1 ciclo de clock exigido para enviar um endereço à memória.
3. Dois ciclos de clock necessários entre cada operação de barramento. (Suponha que o barramento esteja ocioso antes de um acesso.)
4. Um tempo de acesso à memória para as quatro primeiras words de 200ns; cada conjunto adicional de quatro words pode ser lida em 20ns. Considere que uma transferência no barramento de dados lidos mais recentemente e uma leitura das próximas quatro words possam ser sobrepostos.

Encontre a largura de banda sustentada e a latência para uma leitura de 256 words para transferências que utilizam blocos de 4 words e para transferências que utilizam blocos de 16 words. Calcule também o número efetivo de transações de barramento por segundo para cada caso. Lembre-se de que uma única transação de barramento consiste em uma transmissão de endereço seguida por dados.

8.28 [10] <§8.5> Vamos determinar o impacto do overhead do polling para três dispositivos diferentes. Suponha que o número de ciclos de clock para uma operação de polling – incluindo a transferência para a rotina de polling, acesso ao dispositivo e reinício do programa do usuário – seja de 400 e que o processador execute com um clock de 500MHz.

Determine a fração de tempo de CPU consumida para os três casos a seguir, supondo que você sonde com freqüência suficiente para que nenhum dado seja perdido e supondo que os dispositivos estejam potencialmente sempre ocupados:

1. O mouse precisa ser sondado 30 vezes por segundo para garantir que não perderemos qualquer movimento feito pelo usuário.
2. O disquete transfere dados ao processador em unidades de 16 bits e possui uma velocidade de dados de 50KB/seg. Nenhuma transferência de dados pode ser perdida.
3. O disco rígido transfere dados em pedaços de quatro words e pode transferir a 4MB/seg. Novamente, nenhuma transferência pode ser perdida.

8.29 [15] <§§8.3-8.6> Para o sistema de E/S descrito no Exercício 8.41, encontre a largura de banda instantânea máxima em que os dados podem ser transferidos do disco para a memória usando tantos discos quantos forem necessários. De quantos discos e barramentos de E/S (o mínimo de cada) você precisa para alcançar a largura de banda? Como você só precisa dessa largura de banda por um instante, as latências não precisam ser consideradas.

8.30 [5] <§8.6> Suponha que você esteja projetando um microprocessador que use instruções especiais para acessar dispositivos de E/S (em vez de mapear os dispositivos em endereços da memória). Que instruções especiais você precisaria incluir? Que linhas de barramento adicionais você precisaria que esse microprocessador aceitasse para endereçar os dispositivos de E/S?

8.31 <§8.6> Uma vantagem importante das interrupções em relação ao polling é a capacidade de o processador realizar outras tarefas enquanto espera a comunicação de um dispositivo de E/S. Suponha que um processador de 1GHz precise ler 1.000 bytes de dados de determinado dispositivo de E/S. O dispositivo de E/S fornece 1 byte de dados a cada 0,02ms. O código para processar os dados e armazená-los em um buffer utiliza 1.000 ciclos.

- Se o processador detecta que um byte de dados está pronto por meio do polling, e uma interrupção de polling utiliza 60 ciclos, quantos ciclos a operação inteira utiliza?
- Se, em vez disso, o processador for interrompido quando um byte estiver pronto, e o processador gastar o tempo entre as interrupções em outra tarefa, quantos ciclos dessa outra tarefa o processador pode completar enquanto a comunicação de E/S está ocorrendo? O overhead para tratar uma interrupção é de 200 ciclos.

8.32 [15] <§§8.3-8.6> ■ **Aprofundando o aprendizado:** Encontrando gargalos de largura de banda na E/S

8.33 [15] <§§7.3, 7.5, 8.5, 8.6> ■ **Aprofundando o aprendizado:** Operação do sistema de E/S

8.34 [10] <§8.6> Escreva um parágrafo identificando algumas das suposições de simplificação feitas na análise a seguir.

Suponha que tenhamos um processador que executa com um clock de 500MHz e o número de ciclos de clock para uma operação de polling – incluindo transferir para a rotina de polling, acessar o dispositivo e reiniciar o programa do usuário – seja 400. O disco rígido transfere dados em pedaços de quatro words e pode transferir a 4MB/seg. Suponha que você sonde com freqüência suficiente para que nenhum dado seja perdido e suponha que o disco rígido esteja potencialmente sempre ocupado. A configuração inicial de uma transferência de DMA leva 1.000 ciclos de clock do processador, e o tratamento da interrupção no término do DMA exija 500 ciclos de clock do processador. O disco rígido possui uma velocidade de transferência de 4MB/seg e utiliza DMA. Ignore qualquer impacto da disputa do barramento entre o processador e o controlador de DMA. Portanto, se a transferência média do disco for de 8KB, a fração do processador de 500MHz consumida se o disco estiver ativamente transferindo 100% do tempo é de 0,2%.

8.35 [8] <§8.6> Suponha que tenhamos o mesmo disco rígido e processador usados no Exercício 8.18, mas usemos a E/S controlada por interrupção. O overhead para cada transferência, incluindo a interrupção, é de 500 ciclos de clock. Encontre a fração do processador consumida se o disco rígido estiver transferindo dados apenas em 5% do tempo.

8.36 [8] <§8.6> Suponha que tenhamos o mesmo processador e disco rígido do Exercício 8.18. Suponha que a configuração inicial de uma transferência de DMA utilize 1.000 ciclos de clock do processador, e suponha que o tratamento da interrupção no término do DMA exija 500 ciclos de clock do processador. O disco rígido possui uma taxa de transferência de 4MB/seg e utiliza DMA. Se a transferência média do disco é de 8KB, que fração do processador de 500MHz é consumida se o disco estiver ativamente transferindo 100% do tempo? Ignore qualquer impacto de disputa de barramento entre o processador e o controlador de DMA.

8.37 [2 dias-1 semana] <§8.6, Apêndice A> ■ **Aprofundando o aprendizado:** Usando SPIM para explorar a E/S

8.38 [3 dias-1 semana] <§8.6, Apêndice A> ■ **Aprofundando o aprendizado:** Escrevendo código para realizar E/S

8.39 [3 dias-1 semana] <§8.6, Apêndice A> ■ **Aprofundando o aprendizado:** Escrevendo código para realizar E/S

8.40 [15] <§§8.3-8.7> Refaça o exemplo na página 456, mas considere que as leituras são aleatórias em 8KB. Você pode assumir que as leituras sempre são feitas em um disco ocioso, se houver um disponível.

8.41 [20] <§§8.3-8.7> Aqui estão os diversos blocos de montagem usados de um sistema de E/S que possui um barramento processador-memória síncrono executando a 800MHz e um ou mais adaptadores de E/S que realizam a interface dos barramentos de E/S com o barramento processador-memória.

- *Sistema de memória:* o sistema de memória possui uma interface de 32 bits e trata das transferências de quatro palavras. O sistema de memória possui linhas separadas de endereço e dados e, para escritas na memória, aceita uma word a cada ciclo de clock para 4 ciclos de clock e depois exige outros 4 ciclos de clock antes que as words tenham sido armazenadas e possa aceitar outra transação.
- *Interfaces de DMA:* os adaptadores de E/S utilizam DMA para transferir os dados entre os barramentos de E/S e o barramento processador-memória. A unidade de DMA disputa o barramento processador-memória e envia/recebe blocos de quatro words de/para o sistema de memória. O controlador de DMA pode acomodar até oito discos. O início de uma nova operação de E/S (incluindo o seek e o acesso) leva 0,1ms, durante os quais outra E/S não pode ser iniciada por esse controlador (mas as operações pendentes podem ser tratadas).
- *Barramento de E/S:* o barramento de E/S é um barramento síncrono com uma largura de banda sustentável de 100MB/seg; cada transferência possui uma word de extensão.
- *Discos:* os discos possuem um seek médio medido mais latência rotacional de 8ms. Os discos possuem uma largura de banda de leitura/escrita de 40MB/seg, quando estão transferindo.

Descubra o tempo exigido para ler um setor de 16KB de um disco para a memória, supondo que essa seja a única atividade no barramento.

8.42 [5] <§8.7> Para realizar um acesso ao disco ou à rede, normalmente é necessário que o usuário faça com que o sistema operacional se comunique com os controladores de disco ou de rede. Suponha que, em um determinado computador de 5GHz, sejam necessários 10.000 ciclos para o sistema operacional tratar uma trap, 20ms para o sistema operacional realizar um acesso ao disco e 25s para o sistema operacional realizar o acesso à rede. Em um acesso ao disco, que porcentagem do tempo de atraso é gasta no tratamento da trap no sistema operacional? E em um acesso à rede?

8.43 [5] <§8.7> Suponha que, no computador do Exercício 8.42, possamos de alguma forma reduzir o tempo para o sistema operacional se comunicar com o controlador de disco em 60%, e que possamos reduzir o tempo para o sistema operacional se comunicar com a rede em 40%. Por qual porcentagem podemos reduzir o tempo total para um acesso à rede? Por qual porcentagem podemos reduzir o tempo total para um acesso ao disco? Vale a pena gastar muito esforço para melhorar a latência do tratamento da trap no sistema operacional em um computador que realiza muitos acessos ao disco? E em um computador que realiza muitos acessos à rede?

§8.2: página 439, Confiança: 2 e 3. RAID: Todas são verdadeiras.

§8.3: página 8.3-8, 1.

§8.4: página 445, 1 e 2.

§8.5: página 452, 1 e 2.

§8.6: página 455, 1 e 2.

**Respostas
das Seções
“Verifique
você mesmo”**

Computadores no mundo real

Salvando vidas com diagnósticos melhores

Problema: encontre um meio de examinar os órgãos internos para diagnosticar problemas psicológicos sem o uso de cirurgias ou radiação prejudicial.

Solução: o desenvolvimento das imagens por ressonância magnética (MRI – Magnetic Resonance Imaging), uma tecnologia de varredura tridimensional, tem sido uma das descobertas mais importantes na tecnologia médica moderna. MRI utiliza uma combinação de pulsos de freqüência de rádio e campos magnéticos para varrer o tecido. O órgão a ser apresentado é varrido em uma série de fatias bidimensionais, que são então compostas para criar uma imagem tridimensional.

Além dessa tarefa com uso intensivo da computação, para compor as fatias e criar uma imagem volumétrica, uma computação extensiva é

utilizada para extrair as imagens bidimensionais iniciais, pois a relação sinal-ruido normalmente é baixa. O desenvolvimento da MRI tem permitido a varredura de tecidos moles, como o cérebro, para os quais os raios-X não são tão eficazes e a cirurgia exploratória é perigosa. Sem uma capacidade de computação pouco dispendiosa, o MRI continuaria sendo lento e muito caro.

As duas ilustrações mostram uma série de imagens de MRI do cérebro humano. Quando uma imagem está em formato digital, um médico pode manipular a imagem, remover camadas externas, examinar a imagem por pontos de vista diferentes ou examinar a estrutura tridimensional para ajudar no diagnóstico.

Os principais benefícios do MRI são dois:

- Ele pode reduzir a necessidade de cirurgia exploratória desnecessária. Um médico



Imagens de MRI de um cérebro humano, na visão bidimensional

pode ser capaz de determinar se um paciente com dores de cabeça possui um tumor no cérebro, o que exige cirurgia, ou simplesmente precisa de medicação para uma dor de cabeça.

- Oferecendo a um cirurgião uma imagem tridimensional precisa, o MRI pode melhorar o processo de planejamento cirúrgico e, portanto, o resultado. Por exemplo, ao operar o cérebro para remover um tumor sem imagens precisas do tumor, o cirurgião provavelmente teria de entrar no cérebro e depois criar um plano na hora, dependendo do tamanho e local exatos do tumor. Além do mais, técnicas pouco invasivas (por exemplo, cirurgia endoscópica), que se tornaram muito eficientes, seriam impossíveis sem imagens precisas.

Existem muitos usos interessantes novos da tecnologia de MRI, que contam com computação mais rápida e mais econômica. Alguns dos mais promissores são

- Imagens em tempo real do coração e vasos sanguíneos para melhorar o diagnóstico de doenças cardíacas e cardiovasculares;

- Combinação de imagens em tempo real e imagens de MRI durante a cirurgia para ajudar os cirurgiões a realizar a cirurgia com precisão, particularmente quando se usa técnicas minimamente invasivas.
- MRI funcional (fMRI): um novo tipo de aplicação que utiliza MRI para examinar a função do cérebro, principalmente analisando o fluxo sanguíneo em diversas partes do cérebro. fMRI está sendo usada para diversas aplicações, incluindo a exploração de bases fisiológicas para problemas cognitivos, como dislexia, gerenciamento de dores, planejamento de neurocirurgias e conhecimento de distúrbios neurológicos.

Para saber mais, veja estas referências

Varreduras de MRI do projeto Visible Human do National Institutes of Health

Princípios de MRI e sua aplicação em imagens médicas (longo e razoavelmente detalhado, mas apenas um pouco matemático)

Uso de MRI para realizar imagens cardíacas de tempo real e angiografia (imagem dos vasos sanguíneos)

MRI funcional, www.fmri.org/fmri.htm

Visualização e imagens (incluindo imagens MRI e CT): computação de alto desempenho para imagens complexas



Imagens de MRI de um cérebro humano em três dimensões

Índice

As informações no CD estão listadas por número de capítulo e seção, seguidas pelos intervalos de página (CD9.1:1-2). As referências de página precedidas por uma única letra referem-se aos apêndices.

A

abstrações, 16, 19
acerto(s)
definição, 356
taxa/razão, 356
tempo, 356
acrônimos, 8
ACS, CD6.13:12
acumulador estendido, CD2.19:1
acumulador, arquiteturas, CD2.19:1
Ada, 130
add immediate unsigned, 129
add imediato, 43
add unsigned, 129
add, 37-38, 226
adição, 128-132
advanced load, 334
carry antecipado, B29-37
ponto flutuante, 148-151
Advanced Research Project Agency (ARPA), CD7.9:6, CD8.3:4, CD8.11:5
Agarwala, Tilak, CD6.13:3
Aho, Al, CD2.19:6
Aiken, Howard, CD1.7:2
air bags, 211
álgebra booleana, B4
Algol, CD2.19:5
algoritmo de Tomasulo, CD6.13:2
aliasing, 398
Allan, Fran, CD2.19:6
alocar na falha, 366
Alpha, arquitetura, CD5.12:2
Alto, 16, CD1.7:7-13, CD7.9:8, CD8.11:5
ALU. *Ver* Arithmetic Logic Unit
AMD Opteron, hierarquia de memória, 412-415
AMD, 101
Amdahl, Gene, CD5.12:1
análise semântica, CD2.12:1
and (AND), 51, 226, 242, B4
and imediato, 52

AND, porta lógica, CD3.10:4
Andreessen, Marc, CD8.11:5
anéis, CD7.9:5
antidependência, 331-332
Antifuse, B60
Apple II, CD1.7:5
Application Binary Interface (ABI), 17
aproveitamento, 23
Arithmetic Logic Unit (ALU), 133-134, 138, 140, 151
1 bit, B20-22
32 bits, B22-28
caminhos de dados, 215, 220, 221, 222, 223
construindo, B20-22
controle, 226-228, C3-6
implementação de ciclo único, 225-239
implementação de ciclos múltiplos, 239-256
MIPS, B24-29
OpALU, 226-230
SaídaALU, 240, 241, 246
somadores, 220, 221
aritmética
adição, 128-132
divisão, 137-142
falárias e armadilhas, 167-170
média, 195
multiplicação, 132-137
números com sinal e sem sinal, 120-128
ponto flutuante, 142, 144-166
subtração, 128-132
ARM, D29
armadilhas, 26
armazenamento
disco, 430-439
para câmeras digitais, 457-459
ARPANET, CD8.3:4, CD8.11:5
arquitetura peer-to-peer, CD8.3:7-8
arquitetura. *Ver* arquitetura do conjunto de instruções
arquivo executável, 80
arquivos objeto
definição, 80, A7
formato, A10
link-editando, 81-82
arrays
de elementos lógicos, B14
versus ponteiros, 96-99

arredondando, 214-215
arte, restauração, 424-425
ASCII (American Standard Code for Information Interchange), 66-67
versus números binários, 121-122
assembly, 10, 79, A2-7
desvantagens, A7
quando usar, A6
Ver também MIPS, assembly
associatividade nas caches, 377-379
AT&T Bell Labs, CD7.9:7
atalho para negação, 124
atalho para verificação de limites, 126
Atanasoff, John, CD1.7:2
ativar, sinal, 218
ativo, sinal, 218, B3
atraso rotacional, 431
atribuição bloqueante, B18
atribuição não bloqueante, B18

B

Bachman, Charles, CD8.11:3
backpatching, A9
backplane, 440
Backus, John, CD2.19:5
banco de registradores, 220-221, B39, 42-43
bancos de dados relacionais, CD8.11:3
bancos de dados, história dos, CD8.11:3
barramento assíncrono, 441
barramento síncrono, 441
barramentos, 219
assíncronos, 441
backplane, 440
compartilhados, 243-244
definição, 440, B14
fundamentos, 440-443
mestre, 450
Pentium 4, 443-445
processador-memória ou E/S, 441
síncronos, 441
transação, 441
vantagens/desvantagens, 440
base 2 para representar números, 120-121
Basket, Forrest, CD7.9:6
Bell Labs, CD7.9:6
benchmarks para servidor Web, 454

benchmarks para sistema de arquivos, 454
 benchmarks, 192-193
 SPECweb99, 198-201
 Berkeley Computer Corp. (BCC), CD7.9:6
 Berkeley Software Distribution (BSD),
 CD7.9:6
 Berners-Lee, Tim, CD8.11:5
 Big Endian, 41, A31
 Bigelow, Julian, CD1.7:2
 BINAC, CD1.7:3
 bit de modificação, 393
 bit de referência, 391
 bit de sinal, 122
 bit de uso, 391
 bit de validade da cache, 359
 Bit Error Rate (BER), CD8.3:9
 bit mais significativo, 121
 bit menos significativo, 121
 bits, 9-10, 44
 de modificação, 393
 em uma cache, 362
 mais significativos, 121
 mapa, 14
 menos significativos, 121
 referência/uso, 391
 sinal, 122
 sticky, 163
 Blaauw, Gerrit, CD6.13:2
 Block, Barbara, 116-117
 bloco básico, 55
 blocos
 definição, 355
 localizando nas caches, 379-381
 localizando, 407-408
 posicionamento, 406-408
 reduzindo as falhas de cache com,
 375-379
 substituindo, 381, 408-409
 branch equal (beq), 221, 223, 225-239
 Brooks, Fred, Jr., CD6.13:2
 Bubble Sort, 95
 buffer de atualização de varredura, 14
 buffer de escrita, 365
 buffer de quadros, 14
 buffer de reordenação, 335
 Burks, Arthur W., 36, CD1.7:2
 busca de instruções, 289, 292-293, 295, 301
 busca, 430
 buscar na falha/escrita, 366
 bypassing, 283
 byte, endereçamento, 41
 byte, ordem, A31

C

C
 convertendo pontos flutuantes para
 código assembly MIPS, 158-161
 desenvolvimento, CD2.19:5
 exemplo de ordenação, 89-96

hierarquia de tradução, 78-82
 loop while, 54
 operações lógicas, 50-52
 overflows, 129
 procedimentos, 59-66
 strings, 67-68
 C++, CD2.19:5
 cabeça de leitura/gravação, 17-18
 cache associativa por conjunto, 375, 380
 cache de trace, 263
 cache endereçada fisicamente, 398
 cache endereçada virtualmente, 398
 cache mapeada diretamente, 358-359, 375
 cache totalmente associativa, 375
 caches bloqueantes, 336
 caches divididas, 368
 caches
 acessando, 360-364
 associativas por conjunto, 375, 380
 associatividade, 377-379
 bit de validade, 359
 bits, 362
 Blocos usados para reduzir falhas, 375-379
 Blocos, localizando, 379-381
 definição, 358
 desempenho com taxa de clock
 aumentada, 374
 desempenho, medindo e melhorando,
 371-385
 divididos, 368
 endereço de mapeamento para bloco
 multiword, 362-363
 fundamentos, 358-371
 Intrinsic FastMATH, exemplo de
 processador, 366-368
 mapeamento direto, 358-359, 375-376
 memória, 15-16
 modelo dos três Cs, 410-411
 multinível, 372, 381-384
 não bloqueante, 336, 414
 projeto do sistema de memória para dar
 suporte, 368-371
 reduzindo a penalidade de falha com
 caches multiníveis, 381-384
 simples, exemplo, 358-359
 tags, 359, 380
 totalmente associativas, 376
 tratando das falhas, 364-365, 375-379
 tratando de escritas, 365-366
 caching multinível, 371, 381-384
 Cal TSS, CD7.9:6
 câmeras digitais, 178-179, 457-459
 caminho de dados, 15
 com pipeline, 289-300
 convenções lógicas de projeto, 217-219
 criando, 220-226
 elementos, 220
 falácias e armadilhas, 263-265
 implementação de ciclo único, 225-239
 implementação multiciclo, 239-256
 jumps, 236
 operação, 230-234
 campos
 definição, 45
 MIPS, 46
 caracteres em Java, 69-70
 carga, A14
 Carnegie Mellon University, CD6.13:4
 carry rápido, B29-37
 case, instrução, 55-56
 caso comum, agilizando, 202, 214
 Cathode Ray Tubes (CRTs), 14
 Central Processing Unit (CPU), 15
 desempenho, 186-192
 tempo de execução, 185-186
 tempo, 185-186
 Cerf, Vint, CD8.11:5
 chamada ao sistema, 399, A32-33
 Chamberlin, Donald, CD8.11:4
 Chavin de Huántar, 178-179
 chips, 15, 23
 ciclos de clock por instrução (CPI),
 188-190
 CPU multicielo, 249
 ciclos de clock, 186, B37
 implementação de ciclo único, 225-239
 implementação multiciclo, 239-256
 máquinas de estados finitos, 250
 ciclos de clock, dividindo a execução em
 leitura da memória, 247-248
 busca de instruções, 245
 decodificação de instruções e busca de
 registradores, 245-246
 desvios, 246-247
 instruções aritméticas e lógicas, 246,
 247-248
 jump, 247
 referência à memória, 246, 247
 cilindro, uso do termo, 430
 circuitos integrados
 como são fabricados, 22-26
 definição, 15, 21-22
 classe de armazenamento automática, 62
 classe de armazenamento estática, 62
 classes de armazenamento, tipos de, 62
 clock sensível ao nível, B58
 clocks, B37-38
 CLU, CD2.19:5
 CMOS (Complementary Metal Oxide
 Semiconductor), 24, 199
 Cobol, CD2.19:5, CD8.11:3
 Cocke, John, CD2.19:6, CD6.13:2, 3
 Codd, Ted, CD8.11:3, 4
 codificação de instruções, ponto flutuante
 MIPS, 157
 codificador, B6
 código de função, 46
 código de máquina, 45
 códigos de condição, 104
 códigos de correção de erro, B51
 códigos de detecção de erro, B51-52
 Colossus, CD1.7:2
 comentários, 37

commit em ordem, 336
Compact Disks (CDs), 19
Compaq Computers, CD8.11:4
compilação separada, A13
compiladores
 C, 79
 como funcionam, CD2.12:1-6
 desenvolvimento histórico, CD2.19:6
 estrutura, 85
 funções, 9
 Java, 84
 otimização, 85-89
 traduzindo a linguagem de alto nível
 para instruções que o hardware possa
 executar, 9-11, A4
complemento a dois, representação, 122
computadores embutidos, 5-6, CD1.7:7, A6
computadores pessoais, primeiros,
 CD1.7:4-7
computadores
 aparência interna, 14-17
 aplicações, 4-6
 componentes, 12
 desenvolvimento histórico, CD1.7:1-8
 organização, 13
comunicação por rádio, CD8.3:6-7
confiabilidade, 434
confiança do disco, 430-439
conjuntos de instruções
 arquitetura, 17, 19
 definição, 36
 desenvolvimento histórico, CD2.19:1-6
endereçando, 68-77
instruções para tomada de decisões,
 52-54
operações de hardware, 37-38
operações lógicas, 50-52
operандos de hardware, 39-44
otimizações do compilador, 85-89
processando texto, 66-70
projeto para pipelining, 281-282
representando instruções para o
 computador, 44-49
suporte a procedimentos, 58-66
traduzindo e iniciando um programa,
 78-85
constante de 32 bits, carregando, 71
constantes, 42
 32 bits, carregando, 71
consumo, 24
 problemas com consumo, 199-201
Contador de Programa (PC), 59, 220
Control Data Corp. (CDC), CD1.7:4,
 CD6.13:2
controle hardwired, 262, **CD5.12:2**
controle, 15
 fixo, 262
 por pipeline, 300-303
convenções de chamada de procedimentos,
 A16
convenções lógicas de projeto, 217-219

Coonen, Jerome T., CD3.10:5
Copy-back, 393
cor, 220
Corbato, John, CD7.9:5, 9
cores de gráficos, CD2.12:6
CPU. Ver unidade central de processamento
Cray Research, Inc., CD1.7:4
Cray, Seymour, CD1.7:4
CTSS (Compatible Time-Sharing System),
 CD7.9:6-8
Culler Scientific, CD6.13:3
Culler, David, 117
Cutler, David, CD7.9:7
Cydrome Co., CD6.13:3, 4

D

D, flip-flop, B40-42
dados dinâmicos, A16
Dahl, Ole-Johan, CD2.19:5
Data General, CD8.11:4
Dawson, Todd, 117
DEC (Digital Equipment Corp.), CD1.7:4,
 CD4.7:2, CD7.9:7, CD8.11:9
decodificação de instruções, 289, 293, 294,
 303
decodificação, 251
decodificadores, B6
defeitos, 23
deficientes, tecnologia para, 274-275
delayed branch, 223, 287, 315-316, A30
Dell Computer Corp.
 benchmark SPECweb99, 198-201
 benchmarks de CPU SPEC, 192-193,
 196-201
DeMorgan, leis, B4
DeMorgan, teoremas, B8
dependência de nome, 332
desativar, sinal, 218
desdobramento de constante, 87
desempenho do sistema, 186
desempenho
 avaliando, 192-196
 caches, 191-192
 caches, medindo e melhorando, 372-385
 como o hardware e o software afetam, 8
 comparando, 191, 194-196, 321
 CPU, 186-191
 definição, 182-185
 E/S, 452-455
 equação, 188-189
 falácias e armadilhas, 201-204
 fatores afetando, 190
 máquinas de ciclo único, 236-239
 medindo, 185-186
 pipeline, 192
 por custo unitário das tecnologias, 21
 relativo, 184-185
 relatórios, 193-194
 revisão histórica, CD4.7:1-3

sistema, 186
SPECweb99, benchmark, 198-201
Ver também pipelining
versus economia de energia, 199-201
desktop, computadores, 4
 benchmarks de desempenho, 193
despacho múltiplo dinâmico, 327, 334-336
despacho múltiplo estático, 327, 328-334,
 CD6.13:3
despacho múltiplo
 definição, 327
 dinâmico, 327, 334-336
 estático, 327, 328-334
 trabalho da IBM, CD6.13:3
despacho, 263-264
desvio/controle, riscos, 285-287, 313-320
 delayed branch, 223, 315-316
 não tomado, 222, 286, 315
 previsão dinâmica de desvios, 318-319
 Verilog, CD6.7:6-7
desvios condicionais, 53
desvios incondicionais, 53
desvios
 buffer de destino, 319
 buffer de previsão, 318
 delay slot, 318
 delayed branch, 223, 287, 315-316, A30
 endereçamento, 71-73, 221-222
 endereço de destino, 221-223
 implementação de ciclo múltiplo,
 246-247, 253
 loop, 318
 não tomados, 222, 315
 previsão, 287, 318-319
 tabela de histórico, 318
 tomados, 222
detecção de dependência, 306
Deutsch, Peter, CD7.9:6
dígitos binários (números), 10, 44
 ASCII versus, 121
convertendo para decimais, 123
convertendo para ponto flutuante
 decimal, 148
notação científica, 144
somando e subtraindo, 128-132
tabela de conversão
 hexadecimal-binário, 45
uso de, 120-121
DIMMs. Ver Dual Inline Memory Modules
Direct Memory Access (DMA), 450-452
diretivas, layout de dados, A10
disco rígido magnético, 18
disco(s)
 armazenamento e dependência, 430-439
 controlador, 431, 432
 falácias e armadilhas, 460-462
 tempo de leitura, 431-432
 unidades, 15
discos magnéticos, 18, 430
 diferenças entre memória principal e, 18
hierarquias de memória, 354, 386

discos ópticos, 19
disponibilidade, 434
dispositivo de armazenamento não volátil, 430
dispositivos de entrada, 12, A28-29
dispositivos de saída, 12, A28-29
disquetes, 19, CD1.7:5
dissipador de calor, 17
dividendo, 137
divisão com sinal, 140-141
divisão sem sinal, 141-142
divisão, 137-142
divisor, 183
Double Data Rate Synchronous DRAMs (DDDR DRAMs), 490-491
DRAM. Ver Dynamic Random Access Memory
Dual Inline Memory Modules (DIMMs), 17
DVD, unidade, 15
DVDs (Digital Video Disks), 19
Dynamic Random Access Memory (DRAM), 15, 355, 368, 370-371, 386, B47, 49-51
desenvolvimento histórico, CD7.9:2-3
Dynamically Linked Libraries (DLLs), 82-84

E

E/S
barramentos, 440
comunicando com o processador, 447-448
controlada por interrupção, 447-448
dando comandos a dispositivos, 446-447
desempenho, 452-455
desenvolvimento histórico, CD8.11:1-6
dispositivos, 12, 428, A28-29
diversidade, 430
exemplo de câmera digital, 457-459
falácias e armadilhas, 460-462
instruções, 447
interface de dispositivos com processador, memória e sistema operacional, 445-452
mapeada em memória, 446-447
medindo o desempenho, 429
níveis de prioridade de interrupção, 448-449
projetando um sistema, 455-457
solicitações, 430
transferindo dados entre dispositivos e memória, 449-451
velocidade, 453
Eckert, J. Presper, CD1.7:1, 2, 3, CD7.9:1
Eckert-Mauchly Computer Corp., CD1.7:3
EDSAC (Electronic Delay Storage Automatic Calculator), CD1.7:2, CD5.12:1

EDVAC (Electronic Discrete Variable Automatic Computer), CD1.7:1-2
Eispack, CD3.10:2
elaborações, 6
elementos combinacionais, 217
elementos de estado, 217-218, B37
elementos seqüenciais, 218
eliminação de código morto, 87
eliminação de local de armazenamento morto, 87
eliminação de subexpressões comuns global, 87
eliminação de subexpressões comuns, 86
Ellison, Larry, CD8.11:3
EMC, CD8.11:4
emulação, CD5.12:1-2
endereçamento de base, 40, 73
endereçamento de deslocamento, 73
endereçamento em registrador, 73
endereçamento imediato, 73
endereçamento no MIPS
decodificando a linguagem de máquina, 74-78
em desvios condicionais e jumps, 71-73, 221-223
operandos imediatos de 32 bits, 70
resumo de modos, 74
endereçamento pseudodireto, 74
endereçamento relativo ao PC, 72, 74
endereço (endereçamento)
absoluto, A9
base, 40
cálculo, 289, 293, 295, 303
exceção, 258
físico, 386-387
memória, 40
relativo ao PC, 72
tradução, 386, 393-395
virtual, 386
endereço de retorno, 59
endereço virtual, 386
endereços absolutos, A9
endereços físicos, 386-387
Engelbart, Doug, 13
ENIAC (Electronic Numerical Integrator and Calculator), CD1.7:1-3, CD7.9:1
entradas assíncronas, B59-60
escalonamento dinâmico em pipeline, 334-336
EscreveCause, 258
EscreveEPC, 258
EscrevePC, 242
EscrevePCCond, 242
escritas
tratamento da memória virtual, 393
tratando na cache, 365-366
especificação comportamental, B16
especificação estrutural, B16
especulação, 328
espelhamento, 436

esquema de implementação de ciclo único, 225-239
desempenho em pipeline *versus*, 279-280
estações base, CD8.3:7
estações de reserva, 335
exceções, 130, A24-27
definição, 257
endereço, 258
imprecisas, 326
pipeline, 322-326
tratamento, 257-258, A26-27
verificação de controle, 259-261
Exception Program Counter (EPC), 130, 257-258, 324-326
execução fora de ordem, 336
expoente, 144
extensão de sinal, 123, 125-126, 221, 222
extensões para multimídia dos RISCs
desktop/servidor, D13-14
extensões para processamento de sinais digitais, D15

F

falácias, 25
falha, 356
falha, penalidade, 356
reduzindo com caches multinível, 381-384
falha, taxa/razão, 356
global, 384
local, 384
falhas
cache, 364-365, 375-379
capacidade, 410
colisão, 410
compulsórias, 410
conflito, 410
partida a frio, 410
TBL, 401
falhas
Mean Time Between Failures (MTBF), 434
Mean Time To Failure (MTTF), 434
Mean Time To Repair (MTTR), 434
motivos, 434
sincronizador, B59
faltas de página, 386, 387, 389-393, 401
Field Programmable Devices (FPGDs), B60-61
Field Programmable Gate Arrays (FPGAs), B60
fios, B16
Firewire, 441
firmware, CD5.12:2
Fisher, Josh, CD6.13:3
Fishman, Harvey, 274-275
fita magnética, 19
flags, 104

- FLASH, 18, 19
flip-flops, 218, CD40-42
flush, instruções, 315
Flynn, Michael, CD6.13:2
folha, procedimentos, 61, 68
formato de instrução, 44-45
Forrester, J., CD7.9:1
Fortran, CD2.19:6
 overflows, 129, 130
forwarding, 283, 303-310, CD6.7:2
fração, 144, 145
frame de chamada de procedimento, A17
frame pointer, 63
freios ABS, 211
front-end dos compiladores, CD2.12:1-7
- G**
- gateways, CD8.3:4
General-Purpose Register (GPR), 100, 102, CD2.19:2
geração de código, CD2.12:7
Gibson, Garth, CD8.11:3
Goldstine, Herman H., 36, CD1.7:1-2, CD3.10:1
Google,
 News, 351
Gosling, James, CD2.19:5
grafo de interferência, CD2.12:5
Gray, Jim, CD8.11:3
grupo de instruções, 332
grupo de proteção, 436
guarda, 162
H
habilitação de exceção, 401
half words, 69
Hamming, código, B52
handler, 402
hardware
 desempenho afetado, 8
 ferramentas de síntese, B16
 funções, 12
 linguagem de descrição, B15-19
Harvard, arquitetura, CD1.7:2
hazards de controle. *Ver hazards de desvio*
hazards de dados de pipelining
 definição, 283-285
 estruturais, 282
 forwarding, 303-310
 stalls, 311-313
 uso de load, 284
hazards de dados
 definição, 283-285
 forwarding, 303-310
 stalls, 311-313
 uso de load, 284
hazards de desvio não tomado, 286
hazards de desvio/controle de pipelining, 285-287, 313-320, **CD6.7:8-9**
hazards estruturais, 282
- hazards
 unidade de detecção, 311-313
 Ver também hazards de pipelining
heap, alocando espaço para dados, 64
Held, Gerald, CD8.11:3
Hewlett-Packard, CD2.19:4, CD3.10:5, CD4.7:2
 PA-RISC 2.0, D27-28
hexadecimal-binário, tabela de conversão, 45
Hi, 137
hierarquia de tradução para C, 78
 compilador, 79
 linker, 80-82
 loader, 82
 montador, 79
hierarquia de tradução para Java, 84
 compilador Just-in-Time, 85
 compilador, 84
 Java Virtual Machine, 84
Hitachi, SuperH, D30-31
hot swapping, 438
hubs, CD8.3:5
- I**
- IBM
 armazenamento em disco, CD8.11:1-3
 computadores antigos, CD1.7:4
 despacho múltiplo, CD6.13:3
 discos Winchester, CD8.11:2, 3
 disquetes, CD1.7:5, CD8.11:1-2
 histórico das linguagens de
 programação, CD2.19:5
 memória virtual, CD7.9:3-6
 microprogramação, CD5.12:1-2
 ponto flutuante, CD3.10:2-3
 RAID, CD8.11:4
 Stretch, computador, CD6.13:1-2
IEEE 754, padrão de ponto flutuante, 145-148, CD3.10:5-7
if-then-else, compilando instruções para
 desvios condicionais, 53
implementação, 17, 19
IMS, CD8.11:3
inclinação do clock, B57
infinito, 145
informações de depuração, A10
informações de relocação, A10
Ingres, CD8.11:3
instrução indefinida, detecção de exceção, 259
instrução reiniciável, 402
instruções aritméticas e lógicas, 220-221, 224
 implementação de ciclo múltiplo, 246, 248
 implementação de ciclo único, 226-239
instruções de transferência de dados, 40
instruções dependentes, 304
- instruções para tomada de decisão, 52-54
instruções tipo R, 220-221, 224
Integrated Data Store (IDS), CD8.11:2-3
inteiros, com sinal *versus* sem sinal, 124
Intel IA-32, 43
 codificação de instruções, 104-106
 complexidade, 261-262
 conclusões, 106
 desenvolvimento histórico, 99-101, CD2.19:3-4
 falárias e armadilhas, 106-107
 modos de endereçamento, 103
 operações com inteiros, 103-104
 ponto flutuante, 164-166
 registradores, 101-102
Intel IA-64, 328-329, CD5.12:2
 arquitetura, 332-334, CD6.13:3
Intel Streaming SIMD Extension 2 (SSE2), 101
 ponto flutuante, 166
Intel Streaming SIMD Extensions (SSE), 100-101
Intel, CD1.7:4, CD8.11:6
 80286, 100, CD2.19:4
 80386, CD2.19:4
 80486, 100, CD2.19:4
 8086, 100, CD2.19:2, 4
 8087, 100, CD3.10:5
 Pentium e Pentium Pro, 100, CD2.19:4
 SPEC, benchmarks de CPU, 192-193, 196-201
 SPECweb99, benchmark, 198-201
 Ver também Pentium 4
Interface Message Processor (IMP), CD8.3:3
interligação de redes, CD8.3:1-3
Internet, CD8.11:5
 novos serviços, 350-351
interrupções, 130, A24-27
 handler, A24
 imprecisa, 326, CD6.13:2
 níveis de prioridade, 448-449
 uso do termo, 257
interrupções/exceções imprecisas, 3265, CD6.13:3
intervalo vivo, CD2.12:5
intercalação, 320
Intrinsic FastMATH, exemplo de
 processador, 366-369
- J**
- J, tipo, 71
Java Virtual Machine (JVM), 84, CD2.14:2
Java
 bytecode, 84, CD2.14:1, 2
 caracteres e strings, 69-70
 compilando, CD2.14:4-5
 desenvolvimento da, CD2.19:5
 interpretando, CD2.14:1-2

- loop while, CD2.14:3-4
métodos de chamada, CD2.14:5
operações lógicas, 50-52
sort e swap, CD2.14:5-10
traduzindo a hierarquia, 84
Jhai Foundation, rede de PCs, 33
Jobs, Steven, CD1.7:5
Johnson, Reynold B., CD8.11:1
Joy, Bill, CD7.9:5
jump register, 56
jump, 53, 58, 65, 222
caminho de dados e controle, 235-236,
242, 247, 253
endereçamento no, 71-73
jump-and-link, 58-59, 65
Just-in-Time (JIT), compilador, 84
- K**
- Kahan, William, CD3.10:3-6
Kahn, Robert, CD8.11:5
Karnaugh, mapas, B13
Katz, Randy, CD8.11:4
Kay, Alan, CD2.19:6
kernel, processo, 399
Knuth, Donald, CD2.19:7
- L**
- Lampson, Butler, CD7.9:6, 8
laptops, desempenho *versus* consumo de
energia, 199-201
latência de instrução, 341
latência rotacional, 431
latência
instrução, 341
pipeline, 288
Least Recently Used (LRU), 381, 391
leitura, 289, 293, 303
linguagem de destino, A4
linguagem de máquina, 44-45, A2
arquivo-objeto, 80
decodificando, 74-78
ponto flutuante MIPS, 156
linguagem fonte, A4
linguagem orientada a objetos
definição, 96, CD2.14:1
Java, CD2.14:1-10
línguagens de programação de alto nível
arquiteturas, CD2.19:3
definição, 10
traduzindo para instruções que o
hardware pode executar, 9-11
vantagens, 11
línguagens de programação, histórica,
CD2.19:5
link-editor, 80
linkers, 80-82, A3, 13-14
Linpack, CD3.10:2, CD4.7:2
- Linux, 9, CD7.9:11
Liquid Crystal Displays (LCDs), 14
Lisp, CD2.19:4
lista de sensitividade, B18
Little Endian, 41, A31
Livermore, loops, CD4.7:2
Lo, 137
load, 40, 42
byte unsigned, 123
byte, 67, 123
half, 69, 123
halfword unsigned, 123
upper immediate, 70
word, 40, 42, 43, 221, 225-239
- loader, 82
Local Area Networks (LANs), 20,
CD8.3:4-6, CD8.11:6
localidade espacial, 354-355
localidade temporal, 354
localidade, princípio da, 354-355
lógica combinatória, B3, 6-15, 17-19
lógica de dois níveis, B7-10
lógica seqüencial, B3
lógica
arrays de elementos lógicos, B14
combinacional, B3, 6-15, 17-19
dois níveis, B9-10
equações, B4-5
seqüencial, B3, 43-45
Long Instruction Word (LIW), CD6.13:3
long-haul, redes, CD8.3:3
Lookup Tables (LUTs), B61
loops, 54
desdobrando, 86, 330-332
desvio, 318
Lorie, Raymond, CD8.11:3
- M**
- M32R, D31-32
MacOS, 9
macros, A2, 11-12
Magnetic Resonance Imaging (MRI),
472-473
máquina virtual, simulação, A30
máquinas de estados finitos, 249-256,
B37-56, C6-19
Mark, máquinas, CD1.7:2
máscara, 51
Mauchly, John, CD1.7:1, 2, 3
McKeeman, William, CD2.19:6
Mealy, George, 255
Mealy, máquina de, 255, 257, B53
Mean Time Between Failures (MTBF), 434
Mean Time to Failure (MTTF), 434, 460
Mean Time to Repair (MTTR), 434
média aritmética ponderada, 195
megabyte, 17
memória não volátil, 18
memória primária, 18
- memória principal, 18
diferenças entre discos magnéticos e, 18
memória secundária, 18
memória virtual
definição, 386
faltas de página, 386, 387, 389-393
implementando a proteção, 399-400
motivos, 386
offset de página, 386, 387
overlays, 386
página, 386
página, posicionando e encontrando,
388-389
projeto, 387-393
tabela de páginas, 388-389
tradução de endereços, 386, 393-395
Translation-Lookaside Buffer (TLB),
393-403
write-backs, 393
memória volátil, 18
memória, 6
acesso, 289, 293, 295, 302
alocação, 64
cache, 15
cartões, 19
definição, 15, 18
desenvolvimento histórico, CD7.9:1-9
Direct Memory Access (DMA), 450-452
Dynamic Random Access (DRAM), 16,
355, 368, 370-371, 386, B47, 49-51
mapeando, 386
não volátil, 18
operandos, 40
placa, 15
primária, 18
principal, 18
Random Access (RAM), 16
Read Only (ROM), B11, 12, C11-14
secundária, 18
Static Random Access (SRAM), 16,
355, B45-47
transferindo dados entre dispositivos e,
449-451
unidade, 220
uso, A14-16
virtual, 385-406
volátil, 18
memória, E/S mapeada, 446-447
memória, elementos
latches, flip-flops e bancos de
registradores, B39-45
SRAMs e DRAMs, B45-52
memória, hierarquia
caches, 358-385
definição, 354
desenvolvimento histórico, CD7.9:3-6
estrutura, 406-411
falácias e armadilhas, 415-416
métodos para criação, 355
níveis, 355-356
operação geral, 398-399

- Pentium P4 e AMD Opteron, 412-415
 tendências, 418-419
 virtual, 385-406
 memória, referência, 246, 247, 251-252
 memórias, 218
 Memory Data Register (MDR), 240, 247
 metaestabilidade, B59
 metodologia de clock, 218-219, B37
 acionada por transição, 218-219, B37
 metodologias de temporização, B56-60
 sensível ao nível, B58
 metodologia de clocking acionada por
 transição, 218-219, B37
 metodologias de temporização, B56-60
 microarquitetura, 448
 microcódigo, 262
 microcomputadores, primeiros, CD1.7:4
 microinstruções, 262-263
 campos, CD5.7:3, 4-5
 formato, CD5.7:3-3
 microoperações, 262
 microprocessadores
 primeiros, CD1.7:4
 microprogramação
 controlador, 262
 criando um programa, CD5.7:3-7
 definição de formato de microinstrução, CD5.7:2-3
 definição, 249, 261
 desenvolvimento histórico, CD5.12:1-3
 falárias e armadilhas, 264-265
 implementando o programa, CD5.7:8
 simplificando o projeto com, CD5.7:1-8
 Microsoft Corp., CD1.7:4, CD7.9:8,
 CD8.11:4
 Mintermos, B9
 MIPS (Million Instructions Per Second)
 equação, 202
 problema com o uso como medida de
 desempenho, 203-204
 MIPS, 36
 alocação de memória, 64
 Arithmetic Logic Unit (ALU), B24-29
 campos, 46
 código de exceção, 404
 compilando instruções para, 37-38
 conjunto de instruções, 37
 endereçamento, 70-78
 implementação, 215-217
 instruções para tomada de decisão,
 52-53
 linguagem de máquina, resumo, 48, 57,
 66
 mapeando registros em números, 44-50
 operações lógicas, 50-52
 operandos, resumo, 43, 48, 51, 65, 77,
 127
 ponto flutuante, 155-161
 registradores, 39, 58-59, 62, 64-65, 401
 RISC, instruções para MIPS16, D32-33
 RISC, instruções para MIPS64, D19-21
 RISC, subconjunto básico, D7-13, 15-19
 tabela de codificação de instruções, 47,
 75
 traduzindo assembly para linguagem de
 máquina, 47-48
 MIPS, assembly
 instruções de comparação, A42-44
 add immediate unsigned, 129
 add immediate, 42, 43
 add unsigned, 129
 add, 36-38
 and imediato, 52, 65
 AND, 50, 51
 codificação de instruções, A36
 deslocamentos, 50
 desvios condicionais e incondicionais,
 53
 divide unsigned, 142
 divide, 142
 formato de instruções, A36-37
 instruções aritméticas e lógicas,
 A37-42
 instruções de desvio, A44-47
 instruções de exceção e interrupção,
 A59-60
 instruções de jump, A46-47
 instruções de load, A49-50
 instruções de manipulação de
 constantes, A42
 instruções de movimentação de dados,
 A52-53
 instruções de ponto flutuante, A54-59
 instruções de store, A50-52
 instruções de trap, A47-49
 jump, 53, 59
 jump-and-link, 58-59
 load word, 40-43, 221
 modos de endereçamento, A33-34
 move from hi, 137
 move from lo, 137
 multiply unsigned, 137
 multiply, 137
 nor (NOR), 50, 51
 or (OR), 50, 51
 or imediato, 51, 65
 ponto flutuante, 156
 resumo, 38, 43, 48, 51-52, 56, 65, 77,
 127, 131, 143, 156, 171-173
 set on less than immediate unsigned,
 124
 set on less than immediate, 56, 124
 set on less than unsigned, 124
 set on less than, 55
 sintaxe do montador, A34-35
 store word, 40-43
 subtract unsigned, 129
 subtract, 37-38
 tabela de endereços de desvio, 56
 Mitsubishi, M32R, D31-32
 mix de instruções, 192
 modelo dos três Cs, 410-411
 modos de endereçamento
 IA-32, 102
 MIPS, 74
 RISC, D4-7
 monitor de matriz ativa, 14
 monitor em tons de cinza, 14
 monitor gráfico, 14
 montador, diretivas, A3
 montadores, 10, 79, A2, 7-12
 Moore, Edward, 255
 Moore, Gordon, 21
 Moore, lei, 21, 137
 Moore, máquina, 255, B53
 Mosaic, CD8.11:5
 Motorola
 68881, CD3.10:6
 mouse, 13
 move from hi, 137
 move from lo, 137
 move from system control (mfsc), 130
 movimentação de código, 87
 multiciclos, implementação, 239-256
 MULTICS (Multiplexed Information and
 Computing Service), CD7.9:6
 Multiflow Co., CD6.13:3
 multiplexadores, 215, B7
 multiplicação com sinal, 135
 multiplicação, 132-137
 ponto flutuante, 152-154
 multiplicação, 137
 multiplicador, 132
 multiplicando, 132
 multiply unsigned, 137
- N**
- NaN (Not a Number), 146
 NAND, porta lógica, B5
 não alocar na escrita, 366
 não buscar na escrita, 366
 não mapeada, 404
 NCR, CD8.11:5
 Netscape, CD8.11:6
 nop, 312
 nor (NOR), 51, 226, B5
 Northrop, CD1.7:3
 NOT, 51, B4
 notação científica, 142, 144
 notação deslocada, 128, 146
 número de página física, 386
 número de página virtual, 386
 número normalizado, 142
 números com sinal, 128
 números decimais, 44, 121
 convertendo números binários, 123
 convertendo ponto flutuante de binário
 para decimal, 148
 dividindo, 137
 multiplicando, 132-133
 notação científica, 142

números positivos, multiplicando, 132-135
números sem sinal, 120-128
números

 ASCII *versus* binário, 121
 atalhos, 124-126
 base para representar, 120-121
 bit de sinal, 122
 com e sem sinal, 120-128
 convertendo binário para decimal, 123
 loads, 123
 negativo e positivo, 123-125
 representação em complemento a dois,
 122
 sinal e magnitude, 121
Nygaard, Kristen, CD2.19:5

O

Oak, CD2.19:5
offset de página, 386, 387
offset, 40, 41
OpALU, 226-230
opcode, 46, 228, 230
Open Source Foundation, CD7.9:7
Open Systems Interconnect (OSI),
 CD8.3:1
operação de entrada, 440
operação de saída, 440
operações inválidas, 145
operações, para hardware de computador,
 37-38
operadores, Verilog, B16
operando
 constantes ou imediatos, 42
 do hardware do computador, 39-44
 memória, 40
 MIPS, ponto flutuante, 156
 MIPS, resumo, 43, 48, 51, 65, 77, 127
or (OR), 51, 226, 241
or imediato, 51
Oracle, CD8.11:4
ordenação
 alocação de registrador, 90
 código para o corpo, 91-93
 corpo para loop for, 93-94
 Java, CD2.14:5-10
 passando parâmetros, 94
 preservando registradores, 94
 procedimento completo, 94
OrigPC, controle e sinal, 230
otimização de alto nível, 85-86
otimização global, 86-89, CD2.12:3-5
otimização local, 86-89, CD2.12:2
otimizações
 alto nível, 85-86
 locais e globais, 86-88, CD2.12:3-5
 resumo, 88
overflow, CD3.10:3
 adição e subtração, 128-131
 divisão, 142

exceções, detecção, 259
multiplicação, 136-137
ponto flutuante, 145
overlays, 386

P

pacotes, CD8.3:3
padrão 802.11, CD8.3:7
página, 386
 posicionando e encontrando, 388-389
Palmer, John F., CD3.10:5
parada, 332
parâmetro formal, A12
paridade distribuída intercalada por bloco
 (RAID 5), 437
paridade intercalada por bit (RAID 3), 436
paridade intercalada por bloco (RAID 4),
 436-437
paridade
 distribuída intercalada por bloco (RAID
 5), 437
 intercalada por bloco (RAID 4),
 436-437
 intercalada por bit (RAID 3), 436
PA-RISC 2.0, D26-28
parsing, CD2.12:1
Pascal, CD2.19:5
PCSpim, A31
Pentium 4
 barramentos e redes, 443-445
 fabricando, 22-25
 hierarquias de memória, 412-415
 implementação, 262-263
 pipeline, 338-340
Pentium, processadores
 SPEC, benchmarks de CPU, 192-193,
 196-201
 SPECweb99, benchmark, 198-201
periódico de clock, 186, B37
Peterman, Mark, 274-275
pilha de protocolos, CD8.3:2
pilha, 59
 alocando espaço para dados, 63
 ponteiro, 59
 segmento, A16
pipeline, stall, 284-285, 311-313,
 CD6.7:5-6
pipelining
 caminho de dados, 289-300
 controle, 300-303
 definição, 278-279
 desenvolvimento histórico, CD6.13:1-5
 exceções, 322-326
 execução de instruções mais rápida com,
 279-281
 exemplo do Pentium 4, 338
 falácias e armadilhas, 340-343
 forwarding, 283, 303-310, CD6.7:2
 latência, 288
métodos avançados para extrair mais
 desempenho, 326-336
projetando conjuntos de instruções,
 281-282
representação gráfica, 298-300
stalls, 284-285, 311-313, CD6.7:5-6
Verilog usado para descrever e modelar,
 CD6.7:1-5
 visão geral, 278
pixels, 14
placa-mãe, 15
polling, 447
ponteiro global, 62
ponteiros, arrays *versus*, 96-97
ponto binário, 144
ponto flutuante, 142, 144-166
 adição, 148-151
 arredondamento, 162
 convertendo ponto flutuante de binário
 para decimal, 148
 definição, 144
 desenvolvimento histórico, CD3.10:1-7
 IA-32, 164-166
 MIPS, 155-161
 multiplicação, 152-154
 representação, 144-148
pop, 59
Pop-up Satellite Archival Tags (PSATs),
 116-117
portas lógicas, B5, C3-6
PowerPC
 instruções, D25-26
precisão dupla estendida, 165
precisão dupla, 145
precisão simples, 145
previsão dinâmica de desvios, 287,
 316-319
previsão, 287, 316-319
 IA-64, 333
previsor correlato, 319
previsores de desvio de torneio, 319
problema de coerência, 451
problema de dados antigos, 451
problemas ambientais, tecnologia e,
 116-117
problemas de economia de energia,
 199-201
procedimento callee, 59, A16
procedimento caller, 59, A16
procedimentos aninhados, 61-62
procedimentos recursivos, A19, 21
procedimentos
 alocando espaço para dados no heap,
 64-65
 alocando espaço para dados na
 pilha, 63
 aninhados, 61-62
 C, 59-65
 definição, 58
 etapas, 51
 folha, 61, 68

- frame, 63
inline, 85
preservados versus não preservados, 62
recursivos, A19, 21
processador, 15
barramentos com a memória, 441
comunicação com, 447-448
núcleos, 5-6
processadores superescalares, 262,
334-336, CD6.13:3
Processamento de Transações (TP), 453
processo executivo, 399
processo supervisor, 399
produto de somas, B7-9
programa armazenado, conceito, 36, 163
Programmable Logic Arrays (PLAs),
B9-11, C5, 14-16
Programmable Logic Devices (PLDs), B60
Programmable Read Only Memory
(PROM), B11, 12
propagação de constante, 87
propagação de cópia, 87
propagação do carry, B30-37
protocolo de handshaking, 441-442
protocolo de transação repartida, 443
protocolos, famílias/suites, CD8.3:1-2
próximo estado, função de, 249, B52, C8-9,
16-21
pseudo-instruções, 79, A12
push, 59
Putzolu, Gianfranco, CD8.11:4
- Q**
- Quicksort, 96, 383
quociente, 137-138
- R**
- Radix Sort, 383
RAID (Redundant Arrays of Inexpensive
Disks)
código de detecção e correção de erros
(RAID 2), 436
desenvolvimento histórico, CD8.11:4
espelhamento (RAID 1), 435
nenhuma redundância (RAID 0), 435
paridade distribuída intercalada por
bloco (RAID 5), 437
paridade intercalada por bit (RAID 3),
436
paridade intercalada por bloco (RAID
4), 436
redundância P + Q (RAID 6), 438
resumo, 438
uso do termo, 435
Random Access Memory (RAM), 16
Raster Cathode Ray Tubes (CRTs), 14
Rau, Bob, CD6.13:3
- Read Only Memory (ROM), B11, 12,
C11-14
reais, 142
redes comutadas, CD8.3:4
redes, 20-21
características, CD8.3:1
interconexão de, CD8.3:1-3
locais sem fio, CD8.3:5-6
locais, CD8.3:5-6
long-haul, CD8.3:4
Pentium 4, 443-445
redução de força, 87
Reduced Instruction Set Computer (RISC),
CD2.19:3
Alpha, D21-22
ARM, D29
arquitetura, CD5.12:2
desktop *versus* embutido, D2-4
extensões para multimídia, D13-14
extensões para processamento de sinais
digitais, D15
M32R, D31-32
MIPS, D7-13, 15-19
MIPS16, D32-33
MIPS64, D19-21
modos de endereçamento e formatos de
instrução, D4-7
PA-RISC 2.0, D26-28
PowerPC, D25-26
SPARCv.9, D23-25
SuperH, D30-31
Thumb, D30
redundância. *Ver* RAID (Redundant Arrays
of Inexpensive Disks)
referência direta, A8
referências não resolvidas, A2
reg, B16
registrador base, 41
registrador Cause, 258
Registrador de Instrução (IR), 240, 242
registrador destino, 47
registrador, convenções de uso, A16-24
registradores da arquitetura, 338
registradores dedicados, CD2.19:2
registradores, 39, 43, 64, 218, 401
alocação, CD2.12:5-7
arquitetura, 338
dedicados, CD2.19:2
destino, 47
IA-32, 101-102
jump, 56, 59
mapeando para números, 44-49
números, 221
ponteiro global, 62
renomeando, 331-332
spilling, 42, 59
uso especial, CD2.19:2
uso geral, 100, 102, CD2.19:2
registro de ativação, 63
reinício precoce, 364
relocação, 386
- Remington-Rand, CD1.7:3
representação intermediária, CD2.12:2
reprodutibilidade, 193-194
reservas em standby, 438
restaurações, 433
resto, 137-138
restrição de alinhamento, 41
ripple carry, B30, 34-35
RISC. *Ver* Reduced Instruction Set
Computer
Ritchie, Dennis, CD2.19:5, CD7.9:6, 8
roteadores, CD8.3:4
rótulos externos, A8
rótulos globais, A8
rótulos locais, A8
rótulos, externos/globais e locais, A8
- S**
- saída don't care, B12
SaidaALU, 240, 241, 246
Sandisk Corp., 458
Segmentação, 388
segmento de dados estático, 64, A14-16
segmento de dados, A9, 14
segmento de texto, 64, A9, 14
seletor de dados, 215
Selinger, Patricia, CD8.11:3-4
semicondutor, 22
servidores, 4
set on less than immediate unsigned, 124, 127
set on less than immediate, 56, 124
set on less than unsigned, 124, 127
set on less than, 55, 56, 124, 226
setores, 430
shadowing, 435
shift amount, 50
shifts, 50
significando, 145
silício, 22
Silicon Graphics. *Ver* MIPS
Simple Programmable Logic Devices
(SPLDs), B60
simplicidade, 214-215
Simula-67, CD2.19:5
sinais de controle
escrita, 218, 221
lista, 230, 244
sinal de controle de escrita, 218, 221
sinal de portadora, CD8.3:6
sinal e magnitude, 122, 144
sincronizadores, B59-60
sistema síncrono, B38
sistemas de ponto flutuante, CD6.13:4
sistemas de tempo compartilhado,
CD7.9:5-8
sistemas operacionais
desenvolvimento histórico, CD7.9:5-8
exemplos, 9
funções, 9, 446

sítios arqueológicos, 178-179
 Small Computer Systems Interface (SCSI), 434
 Smalltalk, CD2.19:5
 Smith, Jim, CD6.13:1-2
 software de aplicações, 9
 software de sistemas, 9
 software
 aplicações, 9
 desempenho afetado por, 8
 sistemas, 9
 soldagem, 23
 soma de produtos, B7-9
 somador, 220-
 SPARCv.9, D23-25
 SPARCv.9, D23-25
 SPEC (System Performance Evaluation Corp.)
 benchmarks de servidor de arquivos, 454
 benchmarks de servidor Web, 454
 SPEC, razão, 196
 SPECweb99, benchmark, 198-201
 spilling registers, 42, 59
 SPIM, A29-33
 opções da linha de comando, A31
 SRAM. Ver Static Random Access Memory
 SRT, divisão, 141
 Stallman, Richard, CD2.19:6
 Static Random Access Memory (SRAM), 15, 355, B45-47
 Stewart, Robert G., CD3.10:5
 sticky, bit, 163
 Stone, Harold S., CD3.10:5
 Stonebraker, Mike, CD8.11:4
 store half, 69
 store word, 42-43, 221, 225-239
 store, 42
 store, buffer, 336, 366
 store, byte, 67
 Stretch, computador, CD6.13:1-2
 strings
 C, 67-68
 Java, 69-70
 striping, 435
 Stroustrup, Bjarne, CD2.19:5
 Structured Query Language (SQL), CD8.11:3-4
 subtract, 37-38, 226
 sub-rotinas, CD5.7:2
 subtração, 128-132
 subtract unsigned, 129
 Sun Microsystems, CD4.7:2, CD7.9:7
 supercomputadores
 definição, 4
 primeiro, CD1.7:4
 SuperH, D30-31
 swap
 alocação de registradores, 89
 área, 390

 código para o corpo, 90
 Java, CD2.14:5-10
 procedimento completo, 90
 switch, instrução, 56
 switches, CD8.3:5
 Sybase, CD8.11:3-4
 System R, CD8.11:3, 4

T

tabela de páginas, 389
 tabela de símbolos, 79, A9
 tabelas verdade, 227-228, B3, C3, 10, 11, 12
 tags, cache, 359, 380
 taxa de atualização, 14
 taxa de clock, 186
 taxa de dados, 453
 taxa de perda global, 384
 taxa de perda local, 384
 Taylor, George S., CD3.10:6-7
 Taylor, Robert, CD7.9:7
 TCP/IP, CD8.3:3, CD8.11:5
 tecnologia de computador, avanços, 4
 tecnologia sem fio, 21, 116-117
 tela plana, 14
 tempo de busca, 430-431
 tempo de CPU do sistema, 186
 tempo de CPU do usuário, 186
 tempo de execução, 183, 185-186
 uso de total, 195-196
 tempo de preparação, B41
 tempo de propagação, B60
 tempo de relógio, 185
 tempo de resposta, 183, 185
 tempo de suspensão, B41
 tempo de transferência, 431
 tempo decorrido, 185
 tempo do controlador, 431
 tempo, definições de, 185
 terabytes, 4
 término em ordem, 336
 termos do produto, B9
 termos don't care, 228, B12-13
 Thacker, Chuck, CD7.9:6
 Thompson, Ken, CD7.9:6, 8
 Thornton, J. E., CD6.13:2
 thrashing, 405
 thumb, D30
 tipos de dados, Verilog, B16
 tipos
 exemplos, 62
 verificando, CD2.12:1
 Tomasulo, Robert, CD6.13:2, 3
 Torvald, Linus, CD7.9:8
 traduzindo microprograma para hardware, C20-24
 Traiger, Irving, CD8.11:3
 Transaction Processing Council (TPC), 453
 transistores, 21, 22

Translation-Lookaside Buffer (TLB), 393-403, CD7.9:3
 transporte, tecnologia e, 210-211
 trens controlados por computador, 210-211
 trilhas, 430
 troca de contexto, 400
 troca de processo, 400
 Tucker, Stewart, CD5.12:1
 Turing, Alan, CD1.7:2
 TVM (Transmission Voie-Machine), 210-211

U

Ullman, Jeff, CD2.19:6
 underflow, 192, CD3.10:4
 Unicode, 69
 unidade de commit, 335
 unidade de controle
 combinacional, C3-6
 exceções, 257-261
 falárias e armadilhas, 263-265
 implementação de multiciclo, 239-256
 implementação de único ciclo, 225-239
 interrupções, 257
 jumps, 235-236
 máquinas de estados finitos, 248-256,
 C6-16
 microprogramação, 248, CD5.7:4-8
 principal, projetando, 228-234
 somando, 225
 unidade de disco rígido, 15
 unidades de controle combinacionais, C3-6
 unidades de disquete, 19
 unidades na última casa (ulp), 162
 UNIVAC I (Universal Automatic Computer), CD1.7:3
 UNIX
 arquivo objeto, 80
 desenvolvimento, CD2.19:5, CD7.9:6-8
 loader, 82
 USB, 441

V

valor seletor, B7
 variável de indução, eliminação, 88
 varredura, CD2.12:1
 VAX, CD5.12:2-3, CD7.9:7
 vazão, 184
 veneno, 334
 ventilação, 24
 Verilog, CD5.8:1-6
 Descrição, B15-19
 estrutura do programa, B17
 lógica combinacional, B17-19
 lógica sequencial, B43-45
 MIPS, Arithmetic Logic Unit (ALU), B28-29

tipos de dados e operadores, B16
usado para descrever e modelar um
pipeline, CD6.7:1-6
Very Large Scale Integrated (VLSIs),
circuitos, 16, 21, 22
Very Long Instruction Word (VLIW),
CD6.13:3
vetor, interrupções, 258
vetores flutuantes, CD3.10:2
VHDL, B15, 16
von Neumann, John, 36, CD1.7:1-2,
CD3.10:1-2
Vyssotsky, Victor, CD2.19:6

W

wafers, 22-23
WARP, CD6.13:3
while, loop, 54, 72-73
em Java, CD2.14:3-5

Whirlwind, projeto, CD1.7:3, CD7.9:1
Wide Area Networks (WANs), 20,
CD8.11:6
WiFi, 32-33, CD8.3:6
Wilkes, Maurice, CD1.7:2, CD5.12:1,
CD7.9:4
Wilkinson, James H., CD3.10:2
Windows, 9
Wired Equivalent Privacy, CD8.3:8
Wireless Local Area Networks (WLANS),
CD8.3:8-8
Wirth, Niklaus, CD2.19:5
Wong, Gene, CD8.11:3
word crítica primeiro, 364
word requisitada primeiro, 364
word, na arquitetura MIPS, 38
working set, 405
workload, 192
World Wide Web, CD8.11:5
Wozniak, Stephen, CD1.7:5
write-around, 365

write-back, 289, 295, 303, 366, 393, 409
write-through, 365, 409

X

Xerox Palo Alto Research Center (PARC),
CD1.7:5, CD7.9:7, CD8.11:5, 6
xspim, A31

Z

zerando words na memória
arrays e, 96-98
comparando os dois métodos, 99
ponteiros, 98-99
Zip, unidade, 15, 19
Zone Bit Recording (ZBR), 430
Zuse, Konrad, CD1.7:2

JOHN L. HENNESSY é o reitor da Stanford University, de onde é membro da universidade desde 1977, nos departamentos de Engenharia Elétrica e Ciência da Computação. É colaborador do IEEE e da ACM, membro da National Academy of Engineering, da National Academy of Science, da American Academy of Arts and Sciences, e da Spanish Royal Academy of Engineering. Recebeu o Eckert-Mauchly Award de 2001 por suas contribuições à tecnologia RISC, o 2001 Seymour Cray Computer Engineering Award e compartilhou o prêmio John von Neumann em 2000 com David Patterson.

Depois de completar o projeto MIPS em Stanford, em 1984, saiu da faculdade por um ano para ajudar a fundar a MIPS Computer Systems, que desenvolveu um dos primeiros microprocessadores RISC comerciais. Depois de ser adquirida pela Silicon Graphics em 1991, a MIPS Technologies tornou-se uma empresa independente em 1998, focalizando microprocessadores para o mercado embutido. Em 2004, mais de 300 milhões de microprocessadores MIPS foram entregues em dispositivos variando desde videogames e computadores palmtop até impressoras-laser e switches de rede.

Sua pesquisa mais recente em Stanford focaliza a área de projeto e exploração de multiprocessadores. Ele ajudou a liderar o projeto da arquitetura de multiprocessadores BASH, os primeiros multiprocessadores de memória compartilhada distribuída a aceitar coerência de cache, e a base para vários projetos comerciais de multiprocessadores, incluindo os multiprocessadores Silicon Graphics Origin. Desde que se tornou reitor de Stanford, revisar e atualizar este texto e o mais avançado *Arquitetura de Computadores: Uma abordagem competitiva* tornou-se uma forma principal de recreação e passatempo.

Consulte nosso catálogo completo e últimos lançamentos em: www.campus.com.br

ORGANIZAÇÃO E PROJETO DE COMPUTADORES

A INTERFACE HARDWARE / SOFTWARE

3

DAVID A. PATTERSON • JOHN L. HENNESSY

Software e Hardware são os dois pilares dos sistemas computacionais. Somente através da distinção da função de cada um, foi possível conceber a evolução que culmina com os computadores que conhecemos hoje. Cada qual evoluiu de tal forma que são necessários profissionais com perfis diferentes para o desenvolvimento de cada um.

Organização e Projeto de Computadores: A Interface Hardware/Software, 3^a Edição, de David Patterson e John Hennessy, dispensa maiores apresentações. Dos mesmos autores do consagrado *Arquitetura de Computadores: Uma Abordagem Quantitativa*, este livro traz em sua 3^a Edição, de forma didática e repleta de exercícios, o conhecimento necessário para a construção da ponte entre os dois mundos hoje tão distintos: o Hardware e o Software.

Este é um livro-texto abrangente, indicado para cursos de graduação em Informática, Ciência da Computação e Engenharia de Sistemas, em disciplinas como Organização de Computadores, Computadores e Programação.

Mario João Jr.

Professor do NCE/UFRJ e Doutorando do IM-NCE/UFRJ

Patterson e Hennessy demonstram, nessa 3^a Edição, o motivo pelo qual estão na lista dos mais importantes autores em Ciência da Computação.

Neste livro, eles exploram o fato de que o desempenho de um software de sistema é fortemente afetado pela compreensão que os projetistas de software têm das tecnologias de hardware que compõem uma dada plataforma computacional, assim como o projetista de hardware precisa compreender bem quais os impactos no desempenho das aplicações que as suas escolhas terão.

Para conseguir construir essa ponte entre dois mundos tão próximos, os autores apresentam caminhos distintos, mas igualmente bem elaborados, fundamentando as discussões através da arquitetura MIPS 32. Com equilíbrio, os autores iluminam os conceitos de aritmética do computador, pipelining, hierarquias de memória, E/S, multiprocessing e clusterização.

Esta edição foi atualizada em termos de clareza e conteúdo. Em especial nota-se um foco maior na relação entre o hardware e o desempenho de programas.

Organização e Projeto de Computadores: A Interface Hardware/Software, 3^a Edição, é indicado para os cursos de graduação em Informática, Ciência da Computação, Engenharias Eletrônica e de Sistemas na disciplina Organização e Arquitetura de Computadores. Também pode ser utilizado por alunos de pós-graduação e por profissionais de tecnologia da informação.

Sergio Guedes de Souza

Pesquisador - Núcleo de Computação Eletrônica -NCE - UFRJ

Prof. Colaborador - Departamento de Ciência da Computação, Instituto de Matemática - DCC/IM - UFRJ



Uma empresa Elsevier
www.campus.com.br

