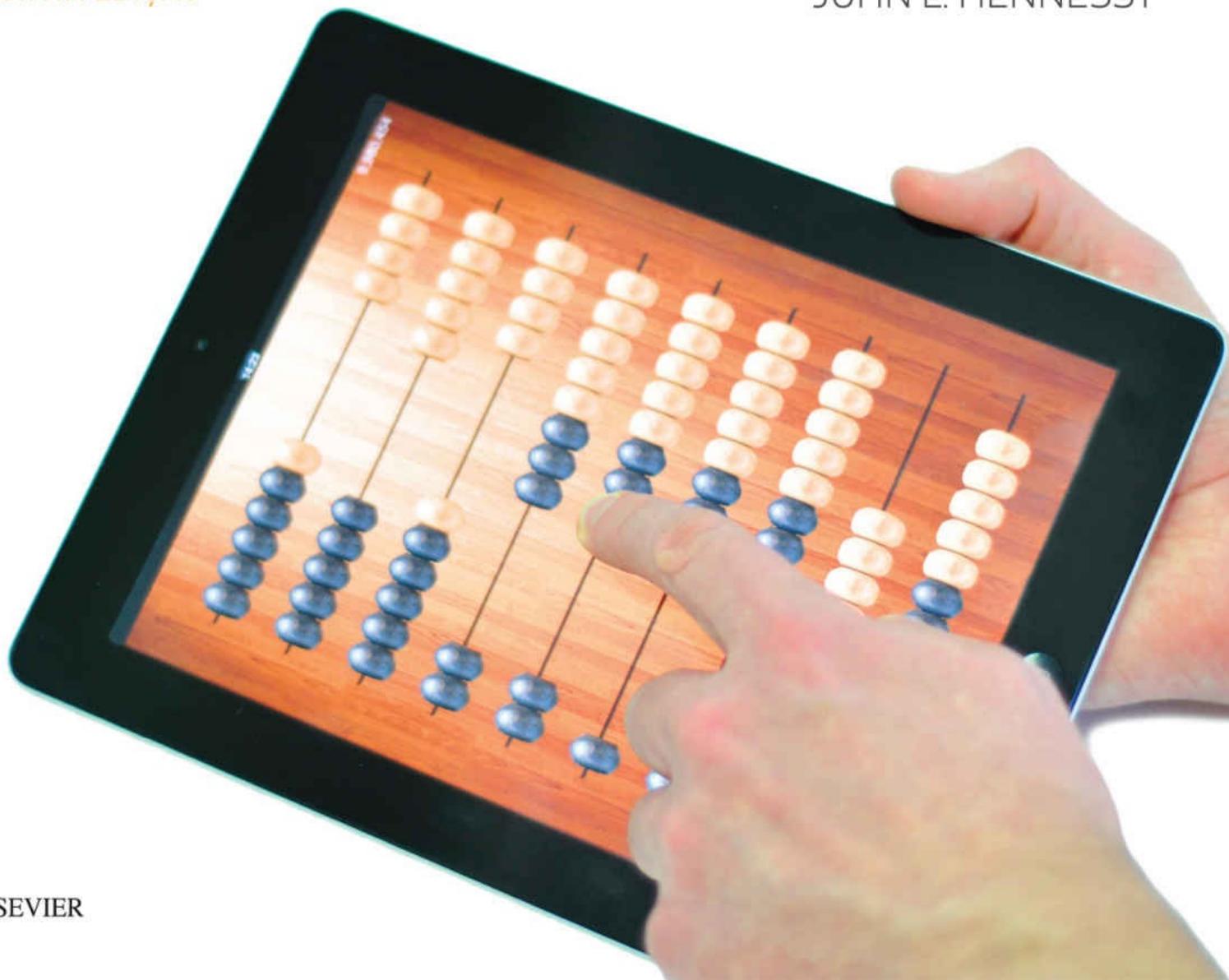


ORGANIZAÇÃO E PROJETO DE COMPUTADORES

A INTERFACE HARDWARE/SOFTWARE

TRADUÇÃO DA
QUINTA EDIÇÃO

DAVID A. PATTERSON
JOHN L. HENNESSY



Organização e Projeto de Computadores

Interface Hardware/Software

TRADUÇÃO DA 5^a EDIÇÃO

David A. Patterson

University of California, Berkeley

John L. Hennessy

Stanford University

Com a colaboração de

Perry Alexander
The University of Kansas

Peter J. Ashenden
Ashenden Designs Pty Ltd

Jason D. Bakos

University of South Carolina

Javier Bruguera
Universidade de Santiago de Compostela

Jichuan Chang
Hewlett-Packard

Matthew Farrens
University of California, Davis

David Kaeli
Northeastern University

Nicole Kaiyan
University of Adelaide

David Kirk
NVIDIA

James R. Larus
School of Computer and Communications Science at EPFL

Jacob Leverich
Hewlett-Packard

Kevin Lim
Hewlett-Packard

John Nickolls
NVIDIA

John Oliver
Cal Poly, San Luis Obispo

Milos Prvulovic
Georgia Tech

Partha Ranganathan
Hewlett-Packard

Tradução
Daniel Vieira

ELSEVIER



Sumário

[Capa](#)

[Folha de rosto](#)

[Copyright](#)

[Dedicatória](#)

[Agradecimentos](#)

[Prefácio](#)

1: Abstrações e Tecnologias Computacionais

- [1.1. Introdução](#)
- [1.2. Oito grandes ideias sobre arquitetura de computadores](#)
- [1.3. Por trás do programa](#)
- [1.4. Sob as tampas](#)
- [1.5. Tecnologias para construção de processadores e memórias](#)
- [1.6. Desempenho](#)
- [1.7. A barreira da potência](#)
- [1.8. Mudança de mares: Passando de processadores para multiprocessadores](#)

- 1.9. Vida real: Fabricação e benchmarking do Intel Core i7
- 1.10. Falácia e armadilhas
- 1.11. Comentários finais
- 1.12. Exercícios

2: Instruções: A Linguagem dos Computadores

Os cinco componentes clássicos de um computador

3: Aritmética Computacional

Os cinco componentes clássicos de um computador

4: O Processador

Os cinco componentes clássicos de um computador

5: Grande e Rápida: Explorando a Hierarquia de Memória

Os cinco componentes clássicos de um computador

6: Processadores paralelos do cliente à nuvem

Organização de multiprocessador ou cluster

Apêndice A: Assemblers, Link-editores e o Simulador SPIM

Apêndice B: Fundamentos do Projeto Lógico

Índice

MIPS Guia de Referência

Copyright

Do original *Computer Organization and Design: The Hardware/Software Interface*, 5th edition.

Tradução autorizada do idioma inglês da edição publicada por Morgan Kaufmann Publishers, an imprint of Elsevier.

Copyright © 2014 Elsevier Inc. All rights reserved

© 2017, Elsevier Editora Ltda.

Todos os direitos reservados e protegidos pela Lei nº 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.

ISBN original: 978-0-12-407726-3

ISBN: 978-85-352- 8793-6

ISBN (versão digital): 978-85-352- 8794-3

Copidesque: Augusto Coutinho

Revisão: Elaine dos S. Batista

Editoração Eletrônica: Thomson Digital

Elsevier Editora Ltda.

Conhecimento sem Fronteiras

Edifício City Tower
Rua da Assembleia, nº 100 – 6º andar – Sala 601
20011-904 – Centro – Rio de Janeiro – RJ

Rua Quintana, 753 – 8º andar
04569-011 – Brooklin – São Paulo – SP

Serviço de Atendimento ao Cliente
0800-0265340
atendimento1@elsevier.com

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 ao nosso Serviço de Atendimento ao Cliente, 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.

**CIP-BRASIL. CATALOGAÇÃO NA PUBLICAÇÃO
SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ**

P344o

5. ed.

Patterson, David

Organização e projeto de computadores / David Patterson, John L. Hennessy. - 5. ed. - Rio de Janeiro : Elsevier, 2017.

il.

Tradução de: Computer organization and design

Inclui bibliografia e índice

ISBN: 978-85-352-8793-6

1. Sistemas operacionais (Computação). I. Hennessy, John L. II. Título.

17-42533

CDD: 005.43

CDU: 004.451



Dedicatória

Para Linda, que foi, é, e sempre será o amor da minha vida.

David A. Patterson leciona arquitetura de computadores na University of California, Berkeley, desde que ingressou na faculdade em 1977, onde é diretor de Ciência da Computação. Seu trabalho como professor lhe rendeu o Distinguished Teaching Award da University of California, o Karlstrom Award da ACM, a Mulligan Education Medal e o Undergraduate Teaching Award do IEEE. Patterson recebeu o IEEE Technical Achievement Award e o ACM Eckert-Mauchly Award por suas contribuições ao RISC, e compartilhou o IEEE Johnson Information Storage Award pelas contribuições ao RAID. Também compartilhou a IEEE John von Neumann Medal e o C&C Prize com John Hennessy. Como seu coautor, Patterson é Fellow da American Academy of Arts and Sciences, do Computer History Museum, ACM e IEEE, e foi eleito para a National Academy of Engineering, a National Academy of Sciences e o Silicon Valley Engineering Hall of Fame. Trabalhou no Information Technology Advisory Committee para o presidente dos Estados Unidos, como presidente da divisão de Ciência da Computação no departamento EECS, em Berkeley, como presidente da Computing Research Association, e como presidente da ACM. Esse registro o fez receber o Distinguished Service Awards da ACM e da CRA.

Em Berkeley, Patterson liderou o projeto e a implementação do RISC I, provavelmente o primeiro computador VLSI com conjunto de instruções reduzido, e a base comercial da arquitetura SPARC. Foi líder do projeto Redundant Arrays of Inexpensive Disks (RAID), que levou aos sistemas de armazenamento confiáveis de muitas empresas. Também esteve envolvido no projeto Network of Workstations (NOW), que levou à tecnologia de cluster usada por empresas da Internet e, mais tarde, à computação em nuvem. Esses projetos receberam três prêmios de dissertação da ACM. Seus projetos de pesquisa atuais são Algoritmo Homem-Máquina e Algoritmos e Especializadores para Implementações Provavelmente Ideais com Resiliência e Eficiência. O AMP Lab está desenvolvendo algoritmos de aprendizado de máquina escaláveis, modelos de programação amigáveis ao computador em escala de warehouse e ferramentas de crowdsourcing para obter rapidamente percepções valiosas, a partir de big data na nuvem. O ASPIRE Lab utiliza o ajuste profundo de hardware e software para alcançar os mais altos níveis de desempenho e eficiência de energia possíveis para sistemas de computação móveis e em escala de rack.

John L. Hennessy é o décimo presidente da Stanford University, de onde é

membro desde 1977, nos departamentos de engenharia elétrica e ciência da computação. Hennessy é Fellow do IEEE e da ACM; membro da National Academy of Engineering, da National Academy of Science e da American Philosophical Society; e Fellow da American Academy of Arts and Sciences. Entre seus muitos prêmios estão o 2001 Eckert-Mauchly Award por suas contribuições à tecnologia RISC, o 2001 Seymour Cray Computer Engineering Award e o 2000 John von Neumann Award, em conjunto com David Patterson. Também obteve sete doutorados honorários.

Em 1981, iniciou o projeto MIPS, em Stanford com alguns poucos alunos de graduação. Depois de concluir o projeto em 1984, afastou-se da universidade para ser o cofundador da MIPS Computer Systems (atualmente MIPS Technologies), que desenvolveu um dos primeiros microprocessadores comerciais RISC. Em 2006, mais de 2 bilhões de microprocessadores MIPS haviam sido instalados em dispositivos, desde videogames e computadores palmtop até impressoras a laser e switches de rede. Mais tarde, Hennessy liderou o projeto DASH (Director Architecture for Shared Memory), que produziu o protótipo do primeiro multiprocessador com coerência de cache escalável; muitas de suas principais ideias foram adotadas nos multiprocessadores modernos. Além de suas atividades técnicas e responsabilidades na universidade, continuou a trabalhar com diversas startups, como conselheiro no estágio inicial ou como investidor.

Agradecimentos

Figuras 1.7, 1.8 Cortesia da iFixit (www.ifixit.com).

Figura 1.9 Cortesia da Chipworks ([www\(chipworks.com](http://www(chipworks.com))).

Figura 1.13 Cortesia da Intel.

Figuras 1.10.1, 1.10.2, 4.15.2 Cortesia de Charles Babbage Institute, University of Minnesota Libraries, Minneapolis.

Figuras 1.10.3, 4.15.1, 4.15.3, 5.12.3, 6.14.2 Cortesia da IBM.

Figura 1.10.4 Cortesia da Cray Inc.

Figura 1.10.5 Cortesia da Apple Computer, Inc.

Figura 1.10.6 Cortesia do Computer History Museum.

Figuras 5.17.1, 5.17.2 Cortesia do Museum of Science, Boston.

Figura 5.17.4 Cortesia da MIPS Technologies, Inc.

Figura 6.15.1 Cortesia da NASA Ames Research Center.

Prefácio

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

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 apresentar 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 ideias 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.

A passagem recente do processador único para microprocessadores *multicore* confirmou a solidez desse ponto de vista, dado desde a primeira edição. Embora os programadores pudessem ignorar o aviso e confiar em arquitetos de computador, escritores de compilador e engenheiros de silício para fazer os seus programas executarem mais rápido ou com menos energia, sem mudanças, essa era já terminou. Para os programas executarem mais rápido, eles precisam se tornar paralelos. Embora o objetivo de muitos pesquisadores seja possibilitar que os programadores não precisem saber a natureza paralela subjacente do hardware que estão programando, serão necessários muitos anos para se concretizar essa visão. Nossa visão é que, pelo menos na próxima década, a maioria dos programadores terá de entender a interface hardware/software se quiser que os programas executem de modo eficiente em computadores paralelos.

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 se comporta 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 os fundamentos sólidos de engenharia e a relação quantitativa custo/benefício. Usamos um enfoque que combinava exemplos e medições, baseado 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. O livro era voltado para profissionais de computação sérios, que desejavam um conhecimento detalhado sobre computadores.

A maioria dos leitores deste livro não planeja se tornar arquiteto de computador. Contudo, o desempenho e o baixo consumo dos futuros sistemas de software será drasticamente afetado pela forma como os projetistas de software entendem as técnicas de hardware básicas, em funcionamento, dentro de um sistema. Assim, aqueles que escrevem compiladores, projetistas de sistemas operacionais, 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 atender esse público-alvo diferente. Ficamos tão satisfeitos com o resultado que as edições seguintes de *Arquitetura de Computadores* foram revisadas para remover a maior parte do material introdutório; logo, há muito menos repetição entre eles hoje, do que nas primeiras edições dos dois livros.

Mudanças para a 5^a edição

Tivemos cinco objetivos principais para esta 5^a edição de *Organização e Projeto de Computadores*: demonstrar a importância de conhecer o hardware por meio de um exemplo comum; destacar os principais temas no decorrer dos tópicos, usando ícones na margem; atualizar os exemplos para refletir a passagem da era do PC para a era pós-PC; espalhar o material sobre E/S por todo o livro, em vez de isolá-lo em um único capítulo; e atualizar o conteúdo técnico para refletir as mudanças ocorridas na área desde a publicação da 4^a edição em 2009.

Antes de discutirmos os objetivos com detalhes, vejamos a tabela a seguir. Ela mostra as sequências de hardware e software no decorrer do livro. Os [Capítulos 1, 4, 5 e 6](#) são encontrados nas duas sequências, não importando a experiência ou o foco. O [Capítulo 1](#) inclui uma discussão sobre a importância da potência e como ela motiva a mudança de microprocessadores de *core* (núcleo) único para *multicore*, e apresenta as oito grandes ideias na arquitetura de computadores. O [Capítulo 2](#) provavelmente servirá de material de revisão para os focados em hardware, mas é uma leitura essencial para aqueles voltados para o software, especialmente para os leitores interessados em aprender mais sobre os compiladores e as linguagens de programação orientadas a objeto. O [Capítulo 3](#) está voltado para os leitores interessados em construir um caminho de dados ou aprender mais sobre aritmética de ponto flutuante. Alguns pularão partes do [Capítulo 3](#), ou porque não precisam delas ou porque são uma revisão. No entanto, apresentamos o exemplo continuado de multiplicação matricial neste capítulo, mostrando como o paralelismo de *subword* oferece uma melhoria quádrupla, portanto, não deixe de ler as Seções [3.6](#) a [3.8](#). O [Capítulo 4](#) explica os processadores em pipeline. As [Seções 4.1, 4.5 e 4.10](#) oferecem resumos e a [Seção 4.12](#) oferece o próximo aumento de desempenho para a multiplicação matricial, para os voltados ao software. Porém, aqueles mais interessados em hardware, descobrirão que esse capítulo apresenta um material básico; eles também podem, dependendo de sua base, querer ler primeiro o Apêndice B, sobre projeto lógico. O último capítulo sobre *multicores*, multiprocessadores e clusters é um material basicamente novo e deve ser lido por todos. Ele foi significativamente reorganizado nesta edição, tornando o fluxo de ideias mais natural e incluindo muito mais profundidade sobre GPUs, computadores em escala de warehouse e interface hardware-software das placas de interface de rede, a base para os clusters.

Capítulo ou Apêndice	Seções	Foco do software	Foco do hardware
1. Abstrações e Tecnologias Computacionais	1.1 a 1.11		
2. Instruções: A Linguagem dos Computadores	2.1 a 2.14		
	2.15 a 2.19		
3. Aritmética Computacional	3.1 a 3.5		
	3.6 a 3.8 (Paralelismo subword)		
	3.9 a 3.10 (Falácia)		
B. Fundamentos de Projeto Lógico	B.1 a B.13		
4. O Processador	4.1 (Visão geral)		
	4.2 (Convenções lógicas)		
	4.3 a 4.4 (Implementação simples)		
	4.5 (Visão geral do pipelining)		
	4.6 (Caminho de dados em pipeline)		
	4.7 a 4.9 (Hazards, exceções)		
	4.10 a 4.12 (Paralelo, vida real)		
5. Grande e Rápida: Explorando a Hierarquia de Memória	5.1 a 5.10		
	5.11 a 5.14		
6. Processadores paralelos do cliente à nuvem	6.1 a 6.8		
	6.9 a 6.13		
A. Montadores, Link-editores e o Simulador SPIM	A.1 a A.11		



Leia cuidadosamente



Leia se tiver tempo



Revise ou leia

O primeiro dos cinco objetivos desta quinta edição foi demonstrar a importância de compreender o hardware moderno para obter bom desempenho e baixo consumo de energia com um exemplo concreto. Como já dissemos, começamos com o paralelismo subword no [Capítulo 3](#), para melhorar a multiplicação matricial por um fator de 4. Dobramos o desempenho no [Capítulo 4](#), desdobrando o laço para demonstrar o valor do paralelismo em nível de instrução. O [Capítulo 5](#) dobra o desempenho novamente, otimizando para caches com uso de bloqueio. Por fim, o [Capítulo 6](#) demonstra um ganho de velocidade de 14 dos 16 processadores usando o paralelismo em nível de thread. Todas as quatro otimizações no total acrescentam apenas 24 linhas de código C ao nosso exemplo inicial de multiplicação matricial.

O segundo objetivo foi ajudar os leitores a separar a floresta das árvores, identificando desde cedo as oito grandes ideias da arquitetura do computador e depois indicando todos os lugares em que elas ocorrem pelo restante do livro. Usamos (espera-se) ícones fáceis de lembrar na margem, e destacamos a palavra correspondente no texto, para que os leitores se lembrem desses oito temas. Existem quase 100 citações no livro. Nenhum capítulo possui menos de sete exemplos de grandes ideias, e nenhuma ideia é citada menos de cinco vezes. O desempenho por meio de paralelismo, pipelining e predição são as três mais populares das grandes ideias, seguidas de perto pela Lei de Moore. O capítulo sobre o processador (4) é aquele com mais exemplos, o que não é uma surpresa, pois provavelmente recebeu a maior atenção dos arquitetos de computador. A grande ideia que aparece em todos os capítulos é o desempenho através do paralelismo, que é uma observação aprazível, dada a ênfase recente no paralelismo na área e nas edições deste livro.

O terceiro objetivo foi reconhecer, nesta edição, a mudança de geração na computação desde a era do PC até a era pós-PC, com nossos exemplos e material. Assim, o [Capítulo 1](#) aprofunda-se nas entradas de um tablet e não em um PC, e o [Capítulo 6](#) descreve a infraestrutura de computação da nuvem. Também incluímos o ARM, que é o conjunto de instruções escolhido nos dispositivos pessoais móveis da era pós-PC, bem como o conjunto de instruções x86, que dominou a era PC e (até aqui) domina a computação em nuvem.

O quarto objetivo foi espalhar o material sobre E/S por todo o livro, em vez de

incliuí-lo em seu próprio capítulo, assim como espalhamos o paralelismo por todos os capítulos na 4^a edição. Logo, o material sobre E/S nesta edição pode ser encontrado nas [Seções 1.4, 4.9, 5.2 e 5.5](#). A ideia é que os leitores (e instrutores) provavelmente abordarão a E/S se ela não estiver segregada em seu próprio capítulo.

Esta é uma área em rápida mudança e, como sempre acontece a cada nova edição, um objetivo importante é atualizar o conteúdo técnico. O exemplo corrente é o ARM Córtext A8 e o Intel Core i7, refletindo nossa era pós-PC. Outros destaques incluem uma visão geral do novo conjunto de instruções de 64 bits do ARMv8, um tutorial sobre GPUs, que explica sua terminologia exclusiva, uma abordagem mais profunda sobre os computadores em escala de warehouse, que compõem a nuvem, e um mergulho nas placas Ethernet de 10 Gigabytes.

Por fim, atualizamos todos os exercícios. Embora alguns elementos tenham mudado, preservamos os elementos úteis das edições anteriores. Para fazer com que o livro funcione melhor como referência, colocamos as definições dos novos termos nas margens, em sua primeira ocorrência. A seção “Entendendo o Desempenho do Programa”, presente em cada capítulo, ajuda os leitores a entenderem o desempenho de seus programas e como melhorá-lo, assim como o elemento “Interface de Hardware/Software” ajudou os leitores a entenderem as escolhas nessa interface. A seção “Colocando em Perspectiva” continua, de modo que o leitor verá a floresta, apesar de todas as árvores. As seções “Verifique Você Mesmo” ajudam os leitores a confirmarem sua compreensão do material na primeira leitura com as respostas fornecidas ao final de cada capítulo. Esta edição ainda inclui o cartão de referência MIPS, inspirado pelo “Green Card” do IBM System/360. Esse cartão foi atualizado e deverá ser uma referência prática na escrita de programas em linguagem assembly MIPS.

Comentários finais

Ao ler a seção de agradecimentos a seguir, você verá que trabalhamos bastante para corrigir erros. Como um livro passa por muitas tiragens, temos a oportunidade de fazer ainda mais correções. Se você descobrir quaisquer outros erros, por favor, entre em contato com a editora por correio eletrônico em atendimento1@elsevier.com, ou pelo correio tradicional, usando o endereço encontrado na página de copyright.

Esta edição marca uma quebra na duradoura colaboração entre Hennessy e Patterson, que começou em 1989. As demandas da condução de uma das maiores universidades do mundo significam que o Presidente Hennessy não poderia mais se comprometer a escrever uma nova edição. O outro autor se sentiu como um malabarista caminhando sem uma rede de segurança. Assim, as pessoas nos agradecimentos e os colegas da Berkeley desempenharam um papel ainda maior na formação do conteúdo deste livro. Apesar disso, desta vez só existe um único autor a culpar pelo novo material que você está para ler.

Agradecimentos da 5^a edição

A cada edição deste livro, tivemos a sorte de receber ajuda de muitos leitores, revisores e colaboradores. Cada uma dessas pessoas ajudou a tornar este livro melhor.

O [Capítulo 6](#) foi tão intensamente revisado que fizemos uma revisão separada para ideias e conteúdo, e fiz mudanças com base no retorno recebido de cada revisor. Gostaria de agradecer a **Christos Kozyrakis**, da Stanford University, por sugerir o uso da interface de rede para clusters, a fim de demonstrar a interface hardware-software da E/S e pelas sugestões sobre a organização do restante do capítulo; a **Mario Flagsilk**, da Stanford University, por fornecer detalhes, diagramas e medições de desempenho da NIC NetFPGA; e pelas sugestões sobre como melhorar o capítulo, a **David Kaeli**, da Northeastern University, **Partha Ranganathan**, da HP Labs, **David Wood**, da University of Wisconsin, e meus colegas da Berkeley **Siamak Faridani**, **Shoaib Kamil**, **Yunsup Lee**, **Zhangxi Tan** e **Andrew Waterman**.

Gostaria de agradecer especialmente a **Rimas Avizenis**, da UC Berkeley, que desenvolveu as diversas versões da multiplicação matricial e também forneceu os números de desempenho. Como trabalhei com seu pai enquanto era graduando na UCLA, foi uma bela simetria trabalhar com Rimas na UCB.

Também gostaria de agradecer ao meu colaborador de vários anos **Randy Katz**, da UC Berkeley, que ajudou a desenvolver o conceito das grandes ideias em arquitetura de computadores como parte da extensa revisão que realizamos juntos em uma turma de graduação.

Gostaria de agradecer a **David Kirk**, **John Nickolls** e seus colegas na NVIDIA (Michael Garland, John Montrym, Doug Voorhies, Lars Nyland, Erik Lindholm, Paulius Micikevicius, Massimiliano Fatica, Stuart Oberman e Vasily Volkov) por escreverem o primeiro apêndice detalhado sobre GPUs. Gostaria de expressar novamente meu apreço a **Jim Larus**, recentemente nomeado diretor da School of Computer and Communications Science na EPFL, por sua disposição em contribuir com sua experiência em programação na linguagem assembly, além de aceitar que os leitores deste livro usem o simulador que ele desenvolveu e mantém.

Também sou grato a **Jason Bakos**, da University of South Carolina, que atualizou e criou novos exercícios para esta edição, trabalhando com os originais preparados para a 4^a edição por **Perry Alexander** (The University of Kansas);

Javier Bruguera (Universidade de Santiago de Compostela); **Matthew Farrens** (University of California, Davis); **David Kaeli** (Northeastern University); **Nicole Kaiyan** (University of Adelaide); **John Oliver** (Cal Poly, San Luis Obispo); **Milos Prvulovic** (Georgia Tech); e **Jichuan Chang, Jacob Leverich, Kevin Lim e Partha Ranganathan** (todos da Hewlett-Packard).

Um agradecimento especial a **Jason Bakos**, por desenvolver os novos slides para palestras.

Sou grato aos muitos instrutores que responderam às pesquisas da editora, revisaram nossas propostas e participaram de grupos de foco para analisar e responder aos nossos planos para esta edição. Entre eles estão: *Grupo de foco em 2012*: Bruce Barton (Suffolk County Community College), Jeff Braun (Montana Tech), Ed Gehringer (North Carolina State), Michael Goldweber (Xavier University), Ed Harcourt (St. Lawrence University), Mark Hill (University of Wisconsin, Madison), Patrick Homer (University of Arizona), Norm Jouppi (HP Labs), Dave Kaeli (Northeastern University), Christos Kozyrakis (Stanford University), Zachary Kurmas (Grand Valley State University), Jae C. Oh (Syracuse University), Lu Peng (LSU), Milos Prvulovic (Georgia Tech), Partha Ranganathan (HP Labs), David Wood (University of Wisconsin), Craig Zilles (University of Illinois at Urbana-Champaign). *Inspeções e críticas*: Mahmoud Abou-Nasr (Wayne State University), Perry Alexander (The University of Kansas), Hakan Aydin (George Mason University), Hussein Badr (State University of New York at Stony Brook), Mac Baker (Virginia Military Institute), Ron Barnes (George Mason University), Douglas Blough (Georgia Institute of Technology), Kevin Bolding (Seattle Pacific University), Miodrag Bolic (University of Ottawa), John Bonomo (Westminster College), Jeff Braun (Montana Tech), Tom Briggs (Shippensburg University), Scott Burgess (Humboldt State University), Fazli Can (Bilkent University), Warren R. Carithers (Rochester Institute of Technology), Bruce Carlton (Mesa Community College), Nicholas Carter (University of Illinois at Urbana-Champaign), Anthony Cocchi (The City University of New York), Don Cooley (Utah State University), Robert D. Cupper (Allegheny College), Edward W. Davis (North Carolina State University), Nathaniel J. Davis (Air Force Institute of Technology), Molisa Derk (Oklahoma City University), Derek Eager (University of Saskatchewan), Ernest Ferguson (Northwest Missouri State University), Rhonda Kay Gaede (The University of Alabama), Etienne M. Gagnon (UQAM), Costa Gerousis (Christopher Newport University), Paul Gillard (Memorial University of Newfoundland), Michael Goldweber (Xavier University), Georgia

Grant (College of San Mateo), Merrill Hall (The Master's College), Tyson Hall (Southern Adventist University), Ed Harcourt (St. Lawrence University), Justin E. Harlow (University of South Florida), Paul F. Hemler (Hampden-Sydney College), Martin Herboldt (Boston University), Steve J. Hodges (Cabrillo College), Kenneth Hopkinson (Cornell University), Dalton Hunkins (St. Bonaventure University), Baback Izadi (State University of New York — New Paltz), Reza Jafari, Robert W. Johnson (Colorado Technical University), Bharat Joshi (University of North Carolina, Charlotte), Nagarajan Kandasamy (Drexel University), Rajiv Kapadia, Ryan Kastner (University of California, Santa Barbara), E.J. Kim (Texas A&M University), Jihong Kim (Seoul National University), Jim Kirk (Union University), Geoffrey S. Knauth (Lycoming College), Manish M. Kochhal (Wayne State), Suzan Koknar-Tezel (Saint Joseph's University), Angkul Kongmunvattana (Columbus State University), April Kontostathis (Ursinus College), Christos Kozyrakis (Stanford University), Danny Krizanc (Wesleyan University), Ashok Kumar, S. Kumar (The University of Texas), Zachary Kurmas (Grand Valley State University), Robert N. Lea (University of Houston), Baoxin Li (Arizona State University), Li Liao (University of Delaware), Gary Livingston (University of Massachusetts), Michael Lyle, Douglas W. Lynn (Oregon Institute of Technology), Yashwant K Malaiya (Colorado State University), Bill Mark (University of Texas at Austin), Ananda Mondal (Claflin University), Alvin Moser (Seattle University), Walid Najjar (University of California, Riverside), Danial J. Neebel (Loras College), John Nestor (Lafayette College), Jae C. Oh (Syracuse University), Joe Oldham (Centre College), Timour Paltashev, James Parkerson (University of Arkansas), Shaunak Pawagi (SUNY at Stony Brook), Steve Pearce, Ted Pedersen (University of Minnesota), Lu Peng (Louisiana State University), Gregory D Peterson (The University of Tennessee), Milos Prvulovic (Georgia Tech), Partha Ranganathan (HP Labs), Dejan Raskovic (University of Alaska, Fairbanks) Brad Richards (University of Puget Sound), Roman Rozanov, Louis Rubinfield (Villanova University), Md Abdus Salam (Southern University), Augustine Samba (Kent State University), Robert Schaefer (Daniel Webster College), Carolyn J. C. Schauble (Colorado State University), Keith Schubert (CSU San Bernardino), William L. Schultz, Kelly Shaw (University of Richmond), Shahram Shirani (McMaster University), Scott Sigman (Drury University), Bruce Smith, David Smith, Jeff W. Smith (University of Georgia, Athens), Mark Smotherman (Clemson University), Philip Snyder (Johns Hopkins University), Alex Sprintson (Texas A&M), Timothy D. Stanley (Brigham Young University),

Dean Stevens (Morningside College), Nozar Tabrizi (Kettering University), Yuval Tamir (UCLA), Alexander Taubin (Boston University), Will Thacker (Winthrop University), Mithuna Thottethodi (Purdue University), Manghui Tu (Southern Utah University), Dean Tullsen (UC San Diego), Rama Viswanathan (Beloit College), Ken Vollmar (Missouri State University), Guoping Wang (Indiana-Purdue University), Patricia Wenner (Bucknell University), Kent Wilken (University of California, Davis), David Wolfe (Gustavus Adolphus College), David Wood (University of Wisconsin, Madison), Ki Hwan Yum (University of Texas, San Antonio), Mohamed Zahran (City College of New York), Gerald D. Zarnett (Ryerson University), Nian Zhang (South Dakota School of Mines & Technology), Jiling Zhong (Troy University), Huiyang Zhou (The University of Central Florida) e Weiyu Zhu (Illinois Wesleyan University).

Um agradecimento especial também a **Mark Smotherman**, que fez uma revisão final cuidadosa, e descobriu 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 **Todd Green** e **Nate McFadden**: certamente, eu não conseguia terminar este livro sem eles. Também queremos estender os agradecimentos a **Lisa Jones**, que gerenciou o processo de produção do livro, e a **Russell Purdy**, que executou o projeto da capa. A nova capa é uma conexão inteligente entre o conteúdo da era pós-PC desta edição e a capa da 1^a edição.

As contribuições de quase 150 pessoas que mencionamos aqui tornaram esta edição nosso melhor livro até agora.

Divirta-se!

David A. Patterson

Abstrações e Tecnologias Computacionais

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

Alfred North Whitehead Uma Introdução à Matemática, 1911

1.1 Introdução

1.2 Oito grandes ideias sobre arquitetura de computadores

1.3 Por trás do programa

1.4 Sob as tampas

1.5 Tecnologias para a montagem de processadores e memória

1.6 Desempenho

1.7 A barreira da potência

1.8 Mudança de mares: Passando de processadores para multiprocessadores

1.9 Vida real: Fabricação e benchmarking do Intel Core i7

1.10 Falácia e armadilhas

1.11 Comentários finais

1.12 Exercícios

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 ideias 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, e cuja economia em parte tornou-se dependente dos rápidos avanços na tecnologia da informação, prometidos pela Lei de Moore. Essa área incomum abraça a inovação com uma velocidade surpreendente. Nos últimos 30 anos, surgiram inúmeros novos computadores que prometiam revolucionar a indústria da computação; essas revoluções foram interrompidas porque alguém sempre 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 Nova York até Londres em aproximadamente um segundo por apenas alguns centavos. Imagine, por alguns instantes, como esse progresso mudaria a sociedade – morar no Taiti e trabalhar em São Francisco, indo para Moscou no início da noite a fim de 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 intelectual do ser humano naturalmente afetou muito nossas vidas cotidianas, além de ter mudado a maneira como conduzimos a busca de novos conhecimentos. Agora, existe uma nova veia 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, em um passado recente, eram “ficção científica para a computação”:

- *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, além de aumentarem a segurança por meio da prevenção de derrapagens perigosas e pela ativação de air-bags para proteger os passageiros em caso de colisão.

- *Telefones celulares*: quem sonharia que os avanços dos sistemas computacionais levariam aos telefones portáteis, permitindo a comunicação entre pessoas em quase todo lugar do mundo?
- *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 15 ou 25 anos. Além do mais, os custos continuam a cair; você poderá adquirir seu próprio genoma, permitindo que a assistência médica seja ajustada a você mesmo.
- *World Wide Web*: ainda não existente na época da primeira edição deste livro, a World Wide Web transformou nossa sociedade. Para muitos, a Web substituiu as bibliotecas e os jornais.
- *Motores de busca*: à medida que o conteúdo da Web crescia em tamanho e em valor, encontrar informações relevantes tornou-se cada vez mais importante. Hoje, muitas pessoas contam com ferramentas de busca para tantas coisas em suas vidas que seria muito difícil viver sem elas.

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. A ficção científica de hoje sugere as aplicações que fazem sucesso amanhã: já a caminho estão os mundos virtuais, a sociedade sem dinheiro em espécie e carros que podem dirigir sem auxílio humano.

Classes de aplicações de computador e suas características

Embora um conjunto comum de tecnologias de hardware (discutidas nas [Seções 1.4](#) e [1.5](#)) 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 (PCs)** são possivelmente os modelos mais conhecidos 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. A evolução de muitas tecnologias de computação é motivada por essa classe da computação, que só tem cerca de 35 anos!

computadores desktop

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

Os **servidores** são a forma moderna do que, antes, eram computadores muito maiores, 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, frequentemente, 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. 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.

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 pouco mais 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 ([Seção 6.10](#)). No outro extremo, estão os **supercomputadores**, que, atualmente, consistem em dezenas de milhares de processadores e, em geral, de **terabytes** de memória, e custam desde dezenas 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 máximo da capacidade de computação, eles são uma fração relativamente pequena dos servidores e do mercado de computadores em termos de receita total.

supercomputador

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

terabyte (TB)

Originalmente, $1.099.511.627.776$ (2^{40}) bytes, embora alguns desenvolvedores de sistemas de comunicações e de armazenamento secundário o tenham redefinido como significando $1.000.000.000.000$ (10^{12}) bytes. Para diminuir a confusão, agora usamos o termo **tebibyte (TiB)** para 2^{40} bytes, definindo *terabyte (TB)* para indicar 10^{12} bytes. A Figura 1.1 mostra a faixa completa de valores decimais e binários, com seus nomes.

Termo decimal	Abreviação	Valor	Termo binário	Abreviação	Valor	% maior
kilobyte	KB	10^3	kibibyte	KiB	2^{10}	2%
megabyte	MB	10^6	mebibyte	MiB	2^{20}	5%
gigabyte	GB	10^9	gibibyte	GiB	2^{30}	7%
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}	10%
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}	13%
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}	15%
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}	18%
zottabyte	YB	10^{24}	yobibyte	YiB	2^{80}	21%

FIGURA 1.1 A ambiguidade 2^x versus 10^y bytes foi resolvida acrescentando-se uma notação binária para todos os termos de tamanho comuns.

Na última coluna, observamos como o termo binário é maior do que seu correspondente decimal, o que é visto quando descemos na tabela. Esses prefixos funcionam para bits e também como bytes, portanto *gigabit* (Gb) é 10^9 bits, enquanto *gibibits* (GiB) é 2^{30} bits.

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 seu carro, os computadores em um aparelho de 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!

computador embutido

Um computador dentro de outro dispositivo, usado para executar uma aplicação predeterminada ou um conjunto de softwares.

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 frequentemente possuem menor tolerância a falhas, já que os resultados podem variar desde um simples incômodo, quando sua nova televisão falha, até o completo desastre que poderia ocorrer quando o computador em um avião ou em um navio falha. Nas aplicações embutidas orientadas ao 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, em geral, são empregadas as mesmas técnicas de redundância utilizadas no mundo dos 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.

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 (Capítulo 4). O núcleo permite que um projetista integre outro hardware específico de uma aplicação com o núcleo do processador para a fabricação em um único chip.

Bem-vindo à era pós-PC

O andar contínuo da tecnologia ocasiona mudanças de geração no hardware de computador que agitam todo o setor de tecnologia da informação. Desde a última edição do livro, passamos por essa mudança, tão significativa no passado quanto a mudança que começou há 30 anos com os computadores pessoais. O PC está sendo substituído pelo **dispositivo móvel pessoal (PMD — Personal Mobile Device)**. PMDs operam com bateria, com conectividade sem fios com a Internet, e normalmente custam centenas de dólares; como os PCs, os usuários podem baixar software (“apps”) para serem executados neles. Ao contrário dos PCs, eles não possuem mais um teclado e mouse, e provavelmente utilizam uma tela sensível ao toque ou até mesmo entrada de voz. O PMD de hoje é um

smartphone ou um computador tablet, mas amanhã poderá incluir óculos eletrônicos. A Figura 1.2 mostra o rápido crescimento dos tablets e smartphones em comparação ao dos PCs e telefones celulares tradicionais.

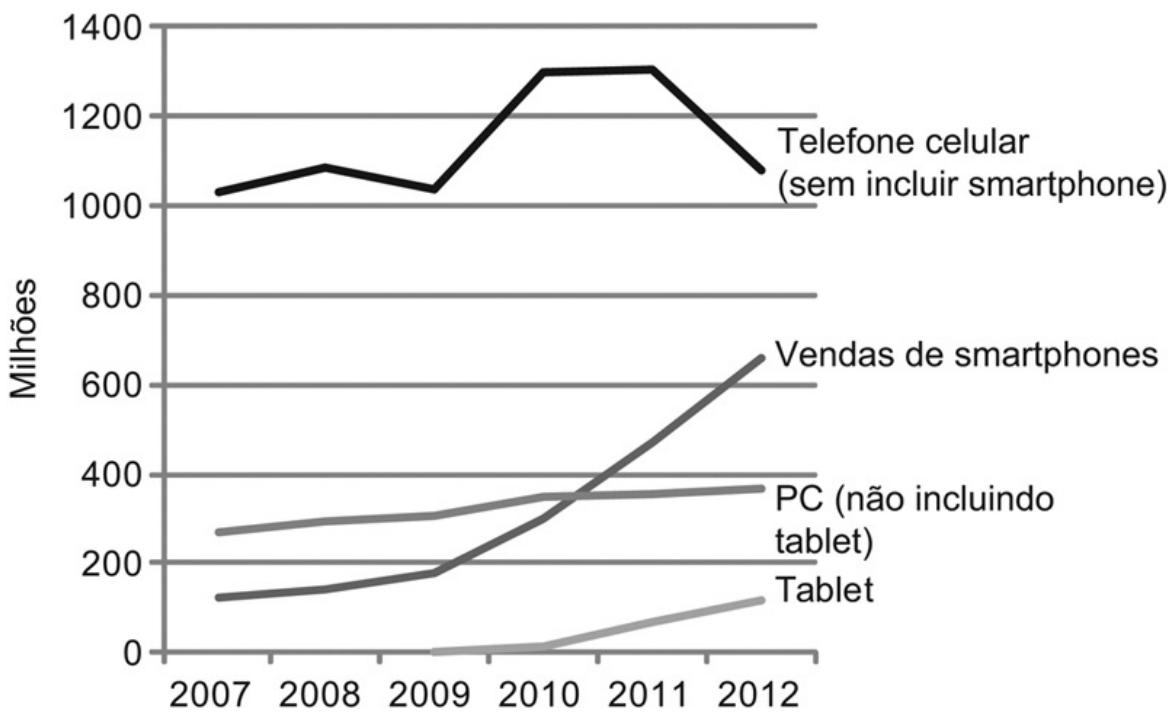


FIGURA 1.2 O número de tablets e smartphones fabricados por ano, refletindo a era pós-PC, contra os computadores pessoais e telefones celulares tradicionais.

Smartphones representam o crescimento recente no setor de telefone celular, e ultrapassaram os PCs em 2011. Os tablets são a categoria com crescimento mais rápido, quase dobrando entre 2011 e 2012. Recentemente, as categorias de PCs e celulares tradicionais estão relativamente planas ou em declínio.

Personal Mobile Devices (PMDs)

são pequenos dispositivos sem fio para realizar a conexão com a Internet; eles utilizam baterias para gerar energia e o software é instalado baixando aplicativos. Alguns exemplos comuns são smartphones e tablets.

Apoderando-se do servidor tradicional está a **computação em nuvem**, que conta com gigantes centros de dados, conhecidos como *Computadores em*

Escala de Warehouse (WSCs — Warehouse Scale Computadores). Empresas como Amazon e Google montam esses WSCs contendo 100.000 servidores e depois permitem que as empresas aluguem partes deles para que possam oferecer serviços de software para DMPs, sem a necessidade de montar WSCs próprios. Em vez disso, o **Software as a Service (SaaS)** implementado por meio da nuvem está revolucionando o setor de software, assim como DMPs e WSCs estão revolucionando o setor de hardware. Os desenvolvedores de software atuais normalmente possuem uma parte de sua aplicação rodando no PMD e uma parte rodando na nuvem.

Computação em nuvem

se refere a um grande conjunto de servidores que prestam serviço através da Internet. Alguns provedores fornecem um número dinâmico variante de servidores como um serviço.

Software as a Service (SaaS)

oferece software e dados como um serviço pela Internet, normalmente através de um programa magro, como um navegador, que roda em dispositivos clientes locais, em vez de um código binário que precisa ser instalado e executado totalmente nesse dispositivo. Alguns exemplos são a busca na Web e as redes sociais.

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 em geral 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 1960: a

natureza paralela dos processadores e a natureza hierárquica das memórias. Além do mais, conforme explicamos na [Seção 1.7](#), os programadores de hoje precisam se preocupar com a eficiência em termos de consumo de energia dos seus programas rodando no PMD ou na nuvem, o que também requer conhecer o que está por trás do seu código. Os programadores que desejam construir versões competitivas do software 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:

- 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*.
- Que técnicas podem ser usadas pelos projetistas de hardware para aumentar a economia de energia? O que o programador pode fazer para ajudar ou impedir esse processo?
- Quais são os motivos e as consequências da mudança recente do processamento sequencial para o processamento paralelo? Este livro oferece a motivação, descreve os mecanismos de hardware atuais para dar suporte ao paralelismo e estuda a nova geração de **microprocessadores “multicore”** ([Capítulo 6](#)).
- Desde o primeiro computador comercial em 1951, que grandes ideias os

arquitetos de computador tiveram para estabelecer a base da computação moderna?

microprocessador multicore

Um microprocessador contendo múltiplos processadores (“cores” ou núcleos) em um único circuito integrado.

Sem entender as respostas a essas perguntas, melhorar o desempenho do seu programa em um computador moderno ou avaliar quais 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 é a base para o restante do livro. Ele apresenta as ideias e definições básicas, coloca os principais componentes de software e hardware em perspectiva, mostra como avaliar o desempenho e a potência, apresenta os circuitos integrados, a tecnologia que estimula a revolução dos computadores, e explica a mudança para núcleos múltiplos (multicores).

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 computador (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, incluímos na margem uma definição **destacada** de cada termo novo na primeira vez que aparece no texto. Após um pequeno período trabalhando com a terminologia, você será fluente e seus amigos ficarão impressionados quando você usar corretamente palavras como BIOS, CPU, DIMM, DRAM, PCIe, SATA e muitas outras.

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).

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, para resumir importantes conceitos quanto ao desempenho do programa. A primeira aparece a seguir.

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 entrada/saída (E/S). 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 4, 5 e 6
Sistema de E/S (hardware e sistema operacional)	Determina a velocidade em que as operações de E/S podem ser executadas	Capítulos 4, 5 e 6

Para demonstrar o impacto das ideias neste livro, melhoramos o desempenho de um programa em C que multiplica uma matriz por um vetor em uma sequência de capítulos. Cada etapa é baseada no conhecimento de como o hardware subjacente realmente funciona em um microprocessador moderno para melhorar o desempenho por um fator de 200!

- Na categoria de *parallelismo em nível de dados*, no [Capítulo 3](#), usamos o *parallelismo de subword por meio de C intrínseco* para aumentar o desempenho por um fator de 3,8.
- Na categoria de *parallelismo em nível de instrução*, no [Capítulo 4](#), usamos o *desdobramento de loop para explorar a questão de instruções múltiplas e hardware de execução fora de ordem* para melhorar o desempenho por outro fator de 2,3.

- Na categoria de *otimização da hierarquia de memória*, no [Capítulo 5](#), usamos o *bloqueio de cache* para aumentar o desempenho em grandes matrizes por outro fator de 2,5.
- Na categoria de *parallelismo em nível de thread*, no [Capítulo 6](#), usamos *loops for paralelos no OpenMP para explorar o hardware multicore* a fim de aumentar o desempenho por outro fator de 14.

Verifique você mesmo

As Seções “Verifique você mesmo” destinam-se a ajudar os leitores a avaliar se compreenderam os principais conceitos apresentados 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, fazendo com que fique mais fácil pulá-las se você estiver certo de que entendeu o assunto.

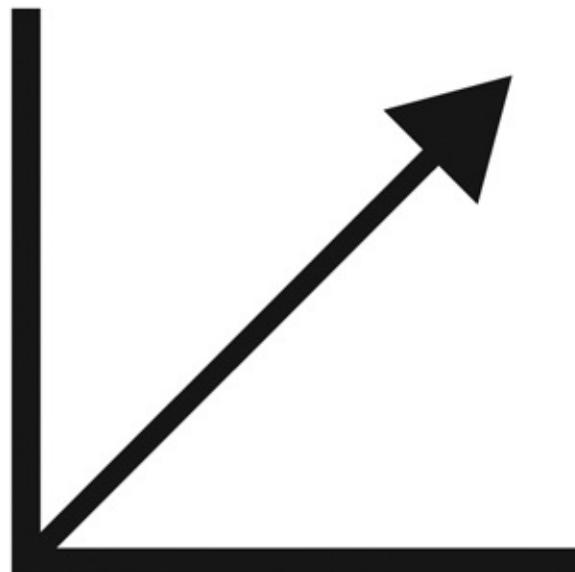
1. O número de processadores embutidos vendidos a cada ano supera, e muito, o número de processadores para PC e até mesmo pós-PC. Você pode confirmar ou negar isso com base em sua própria experiência? Tente contar o número de processadores embutidos na sua casa. Compare esse número com o número de computadores convencionais em sua casa.
2. Como mencionado anteriormente, tanto o software quanto o hardware afetam o desempenho de um programa. Você pode pensar em exemplo nos quais 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. Oito grandes ideias sobre arquitetura de computadores

Agora, apresentamos oito grandes ideias que os arquitetos de computador inventaram nos últimos 60 anos de projetos de computadores. Essas ideias são tão poderosas que duraram muito tempo depois do primeiro computador que as usaram, com os arquitetos mais novos demonstrando sua admiração ao imitar seus predecessores. Essas grandes ideias são temas que estarão entrelaçados durante este e os próximos capítulos, quando surgirem os exemplos. Para indicar sua influência, nesta seção apresentamos ícones e termos destacados, que representam as grandes ideias, usando-os para identificar as quase 100 seções do livro que apresentam o uso das grandes ideias.

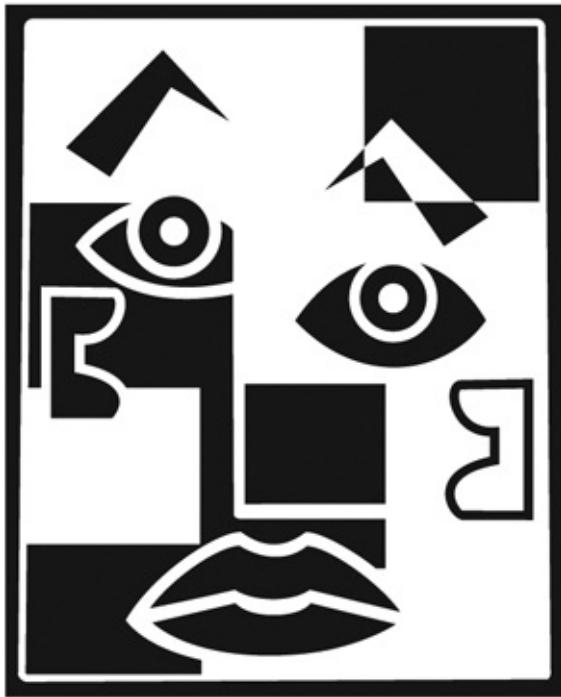
Projete pensando na Lei de Moore

A única constante para os projetistas de computador é a mudança rápida, que é controlada em grande parte pela **Lei de Moore**. Ela declara que os recursos do circuito integrado dobram a cada 18 a 24 meses. A Lei de Moore resultou de uma previsão desse crescimento na capacidade do CI em 1965, por Gordon Moore, um dos fundadores da Intel. Como os projetos de computador podem durar anos, os recursos disponíveis por chip podem facilmente dobrar ou quadruplicar entre o início e o final do projeto. Assim como um atirador, os arquitetos de computador precisam antecipar onde estará a tecnologia quando o projeto terminar, e não quando ele começar. Usamos um gráfico da Lei de Moore “para cima e para a direita”, representando o projeto para a mudança rápida.



Use a abstração para simplificar o projeto

Os arquitetos e os programadores de computador tiveram que inventar técnicas para que se tornassem mais produtivos, pois, de outra forma, o tempo de projeto aumentaria de modo insustentável quando os recursos aumentassem pela Lei de Moore. Uma técnica de produtividade importante para o hardware e o software é usar **abstrações** para representar o projeto em diferentes níveis de representação; os detalhes de nível mais baixo serão ocultados, para oferecer um modelo mais simples nos níveis mais altos. Usaremos o ícone de pintura abstrata para representar essa segunda grande ideia.



A B S T R A Ç Ã O

Torne o caso comum veloz

Tornar o **caso comum veloz** costuma melhorar mais o desempenho do que otimizar o caso raro. Ironicamente, o caso comum normalmente é mais simples do que o caso raro e, portanto, geralmente é mais fácil de melhorar. Esse conselho do senso comum implica que você saberá qual é o caso comum, o que é possível apenas com experimentação e medição cuidadosas ([Seção 1.6](#)). Usamos um carro esportivo como ícone para tornar o caso comum veloz, pois a viagem mais comum tem um ou dois passageiros, e certamente é mais fácil tornar um carro esportivo veloz do que um utilitário!



CASO COMUM VELOZ

Desempenho pelo paralelismo

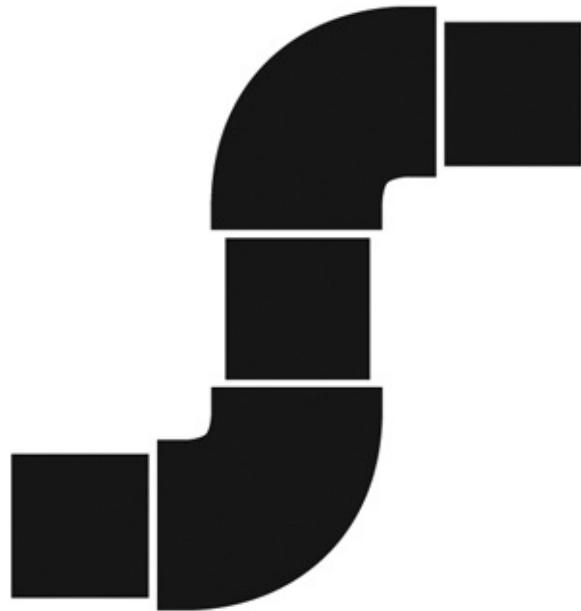
Desde o nascimento da computação, os arquitetos de computador têm oferecido projetos que geram mais desempenho realizando operações em paralelo. Veremos muitos exemplos de paralelismo neste livro. Usamos um avião a jato com vários motores como nosso ícone para **desempenho paralelo**.



PARALELISMO

Desempenho pelo pipelining

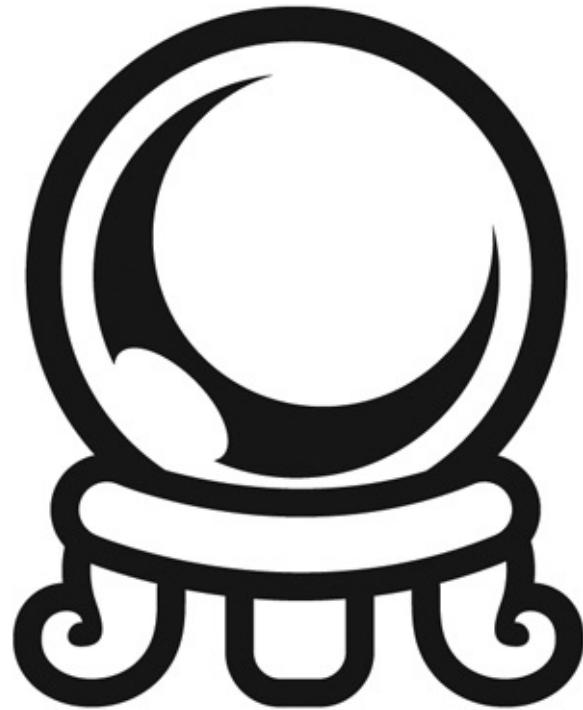
Um padrão de paralelismo em particular é tão prevalecente na arquitetura de computação que merece seu próprio nome: **pipelining**. Por exemplo, antes dos dispositivos contra incêndio, uma “brigada de baldes” respondia a um incêndio, a qual muitos filmes de cowboy mostram em resposta a um ato covarde de um vilão. Os moradores formam uma corrente humana para carregar uma fonte de água a um incêndio, pois eles poderiam mover os baldes muito mais rapidamente pela corrente, em vez de indivíduos correndo de um lado para outro. Nossa ícone de tubulação é uma sequência de tubos, com cada seção representando um estágio da tubulação.



PIPELINING

Desempenho pela predição

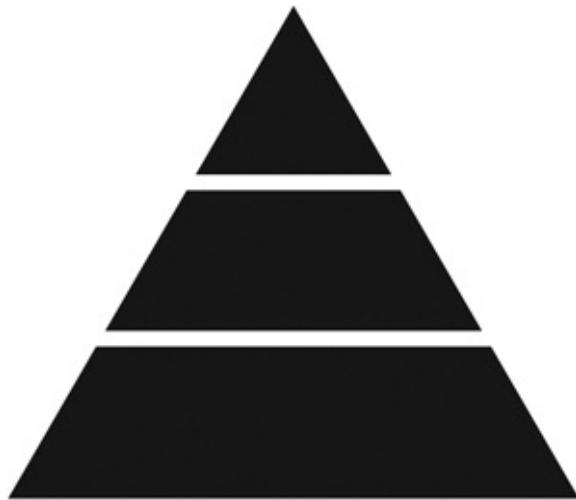
Seguindo o ditado de que pode ser melhor pedir perdão do que pedir permissão, a última grande ideia é a **predição**. Em alguns casos, na média, pode ser mais rápido prever e começar a trabalhar do que esperar até que você saiba ao certo, supondo que o mecanismo para se recuperar de um erro de previsão não seja dispendioso e sua predição seja relativamente precisa. Usamos a bola de cristal de uma cartomante como nosso ícone de predição.



P R E D I Ç Ã O

Hierarquia de memórias

Os programadores desejam que a memória seja rápida, grande e barata, pois a velocidade da memória geralmente modela o desempenho, a capacidade limita o tamanho dos problemas que podem ser resolvidos e o custo da memória, hoje, geralmente é o maior custo do computador. Os arquitetos descobriram que podem resolver essas demandas em conflito com uma **hierarquia de memórias**, com a memória mais rápida, menor e mais cara por bit no topo da hierarquia e a mais lenta, maior e mais barata por bit no fundo. Conforme veremos no [Capítulo 5](#), as caches dão ao programador a ilusão de que a memória principal é quase tão rápida quanto o topo da hierarquia e quase tão grande e barata quanto o fundo da hierarquia. Usamos um triângulo em camadas para representar a hierarquia da memória. A forma indica velocidade, custo e tamanho: quanto mais perto do topo, mais rápida e mais cara por bit é a memória; quanto mais larga a base da camada, maior é a memória.



HIERARQUIA

Estabilidade pela redundância

Os computadores não apenas precisam ser rápidos; eles precisam ser estáveis. Como qualquer dispositivo pode falhar, montamos sistemas **estáveis** incluindo componentes redundantes, que podem assumir o controle quando uma falha ocorre e ajudar a detectá-la. Usamos o caminhão de reboque como nosso ícone, pois os duplos pneus em cada lado de seus eixos traseiros permitem que o caminhão continue seguindo, mesmo quando um pneu fura. (Supõe-se que o motorista do caminhão seguirá imediatamente para um borracheiro, para que o pneu seja consertado e a redundância restaurada!)



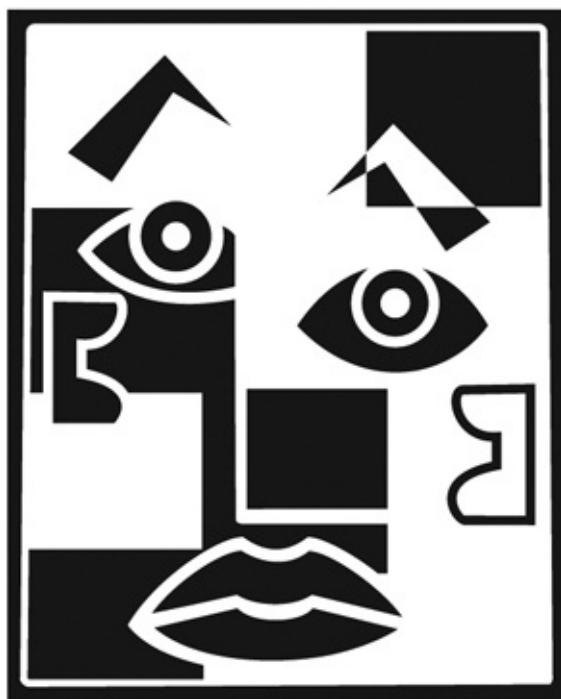
ESTABILIDADE

1.3. Por trás do programa

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

Mark Twain, The Innocents Abroad, 1869

Uma aplicação típica, como um processador de textos ou um grande sistema de banco de dados, pode consistir em milhões de linhas de código e se basear em bibliotecas de software sofisticadas que implementam funções complexas no apoio à 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, um exemplo da grande ideia da **abstração**.



A B S T R A Ç Ã O

A [Figura 1.3](#) mostra que essas camadas de software são organizadas principalmente de maneira hierárquica, na qual as aplicações são o anel mais externo e uma variedade de **software de sistemas** situa-se entre o hardware e as aplicações.

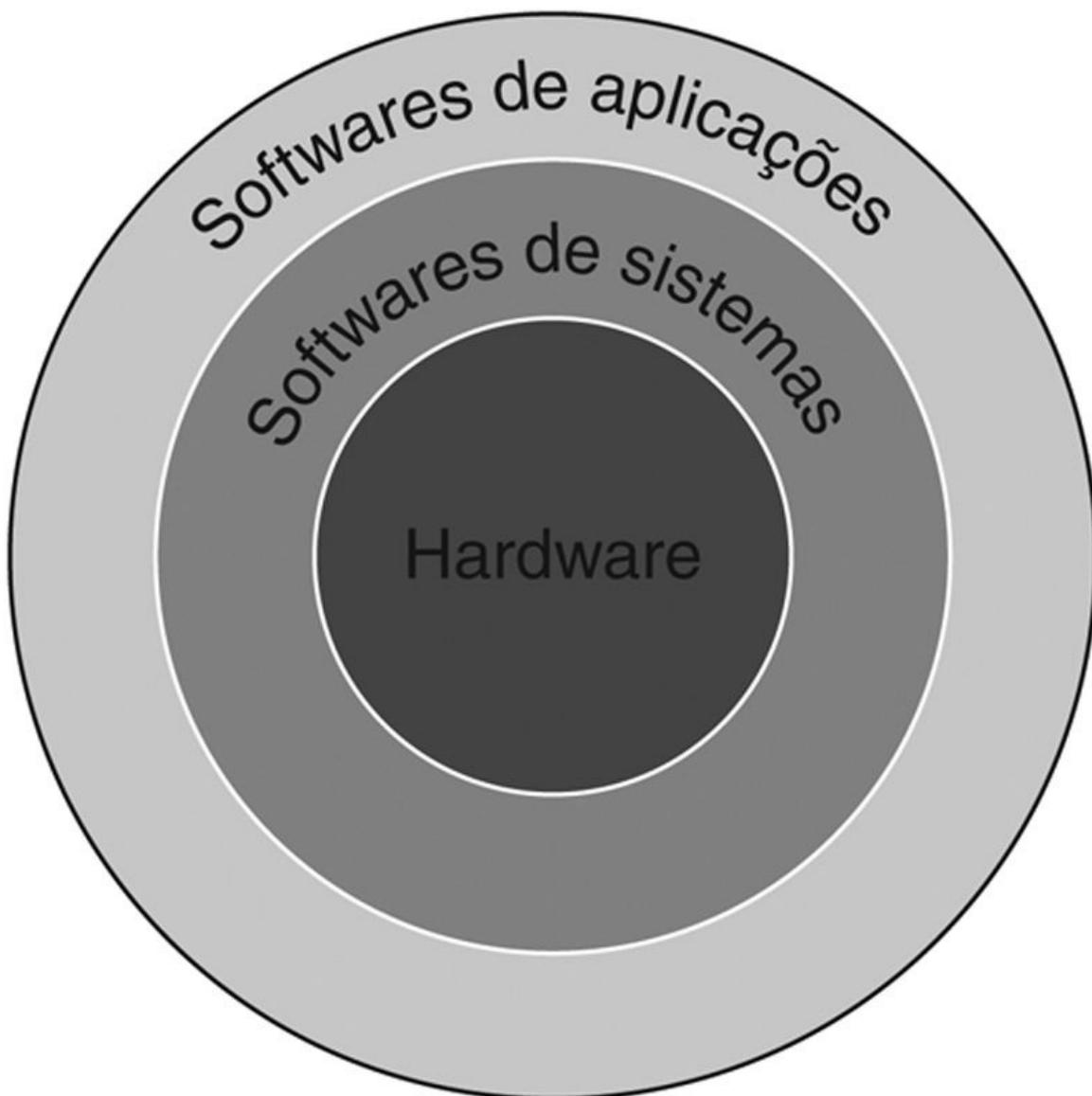


FIGURA 1.3 Uma visão simplificada do hardware e software como camadas hierárquicas, mostradas como círculos concêntricos, em que o hardware está no centro e as aplicações aparecem externamente.

Nas aplicações complexas, muitas vezes existem diversas camadas de software de aplicação. Por exemplo, um sistema de

banco de dados pode “rodar” sobre o software de sistemas hospedando uma aplicação, que, por sua vez, roda sobre o banco de dados.

software de sistemas

Software que fornece serviços normalmente úteis, incluindo sistemas operacionais, compiladores, carregadores, e montadores.

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 do 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
- Alocar armazenamento e memória
- Providenciar o compartilhamento protegido do computador entre as diversas aplicações que o utilizam simultaneamente

sistema operacional

Programa de supervisão que gerencia os recursos de um computador, em favor dos programas executados nessa máquina.

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

Os **compiladores** realizam outra função fundamental: a tradução de um programa escrito em uma linguagem de alto nível, como C, C + +, Java ou Visual Basic, em instruções que o hardware possa executar. Em razão da sofisticação das linguagens de programação modernas e das instruções simples executadas pelo hardware, a tradução de um programa de linguagem de alto nível para instruções de hardware é complexa. Daremos um breve resumo do processo aqui, e depois entraremos em mais detalhes, no [Capítulo 2](#) e no Apêndice A.

compilador

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

De uma linguagem de alto nível para a linguagem do hardware

Para poder realmente falar com uma máquina eletrônica, você precisa enviar sinais elétricos. Os sinais mais fáceis de serem entendidos pelas máquinas são ligado (*on*) e desligado (*off*); portanto, o alfabeto da máquina se resume a apenas duas letras. Assim como as 26 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 e obedece, podem ser imaginadas como números. Por exemplo, os bits

1000110010100000

dizem ao computador para somar dois números. O [Capítulo 2](#) 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.

dígito binário

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

Instrução

Um comando que o hardware do computador entende e obedece.

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 que traduzem 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

montador (assembler)

Um programa que traduz uma versão simbólica de instruções para a versão binária.

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**. Em contraste, a linguagem binária que a máquina entende é a **linguagem de máquina**.

linguagem assembly

Uma representação simbólica das instruções de máquina.

linguagem de máquina

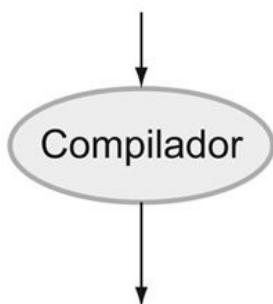
Uma representação binária das instruções de máquina.

Embora seja um fantástico avanço, a linguagem assembly ainda está longe da notação que um cientista gostaria de 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. A [Figura 1.4](#) mostra os relacionamentos entre esses programas e linguagens, que são outros exemplos do poder da **abstração**.

Programa
em linguagem
de alto
nível (em C)

```
swap(int v[], int k)
{int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

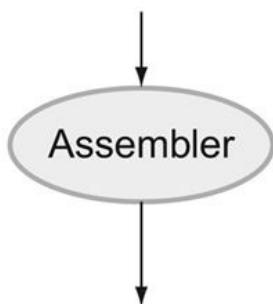


Programa em linguagem de máquina binária (para o MIPS)

```

swap:
    multi $2, $5,4
    add   $2, $4,$2
    lw    $15, 0($2)
    lw    $16, 4($2)
    sw    $16, 0($2)
    sw    $15, 4($2)
    jr   $31

```

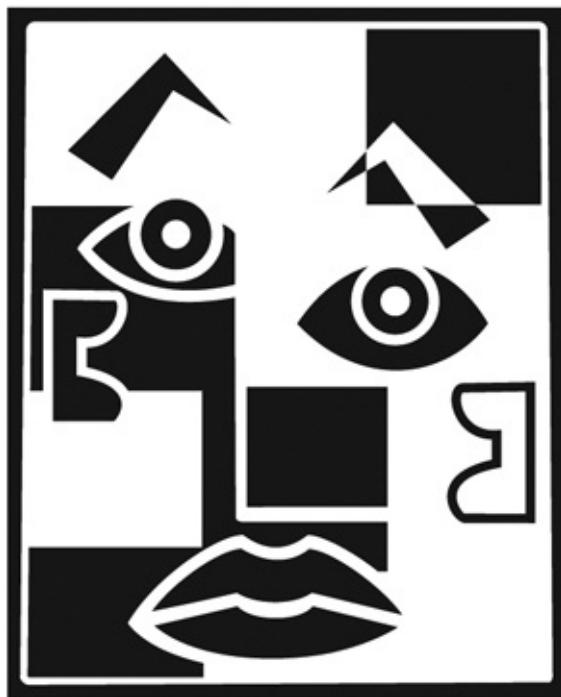


Programa em linguagem de máquina binária (para o MIPS)

```
00000000101000100000000100011000  
0000000010000010001000000100001  
10001101111000100000000000000000  
1000111000010010000000000000000100  
1010111000010010000000000000000000  
1010110111100010000000000000000100  
0000001111100000000000000000001000
```

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 removem a fase intermediária e produzem linguagem de máquina diretamente. Essas linguagens e esse programa são analisados com mais detalhes no [Capítulo 2](#).



A B S T R A Ç Ã O

linguagem de programação de alto nível

Uma linguagem portável, como C, C++, Java ou Visual Basic, composta de palavras e notação algébrica, que pode ser traduzida por um compilador para a linguagem assembly.

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
```

Como podemos ver, o montador (assembler) traduziria essa instrução para a instrução binária, que diz ao computador para somar os dois números, A e B.

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 ([Figura 1.4](#)). Além disso, elas permitem que linguagens sejam projetadas de acordo com o uso pretendido. É por isso que a linguagem Fortran foi projetada para computação científica, Cobol para processamento de dados comerciais, Lisp para manipulação de símbolos e assim por diante. Há também linguagens específicas de domínio para grupos ainda mais estreitos de usuários, como aqueles interessados em simulação de fluidos, por exemplo.

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 necessita-se de menos tempo para desenvolver programas quando são escritos em linguagens que exigem menos linhas para expressar uma ideia. 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.

1.4. Sob as tampas

Agora que olhamos por trás do 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: entrada, saída, processamento e armazenamento de dados. A forma como essas funções são realizadas é o principal tema deste livro, e os capítulos subsequentes lidam com as diferentes partes destas 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”. Há aproximadamente uma dúzia desses itens no livro; o primeiro descreve os cinco componentes de um computador que realizam as tarefas de entrada, saída, processamento e armazenamento de dados.

Dois dos principais componentes dos computadores são: os **dispositivos de entrada**, como o teclado e o mouse, e os **dispositivos de saída**, como a caixa de som. Como o nome sugere, a entrada alimenta o computador, e a saída é o resultado da computação, enviado para o usuário. Alguns dispositivos, como redes sem fio, fornecem tanto entrada quanto saída para o computador.

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.

Os [capítulos 5 e 6](#) descrevem dispositivos de entrada e saída (E/S) em mais detalhes, mas vamos dar um passeio preliminar pelo hardware do computador, começando com os dispositivos de E/S externos.

Colocando em perspectiva

Os cinco componentes de um computador são: entrada, saída, memória,

caminho de dados e controle; 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.

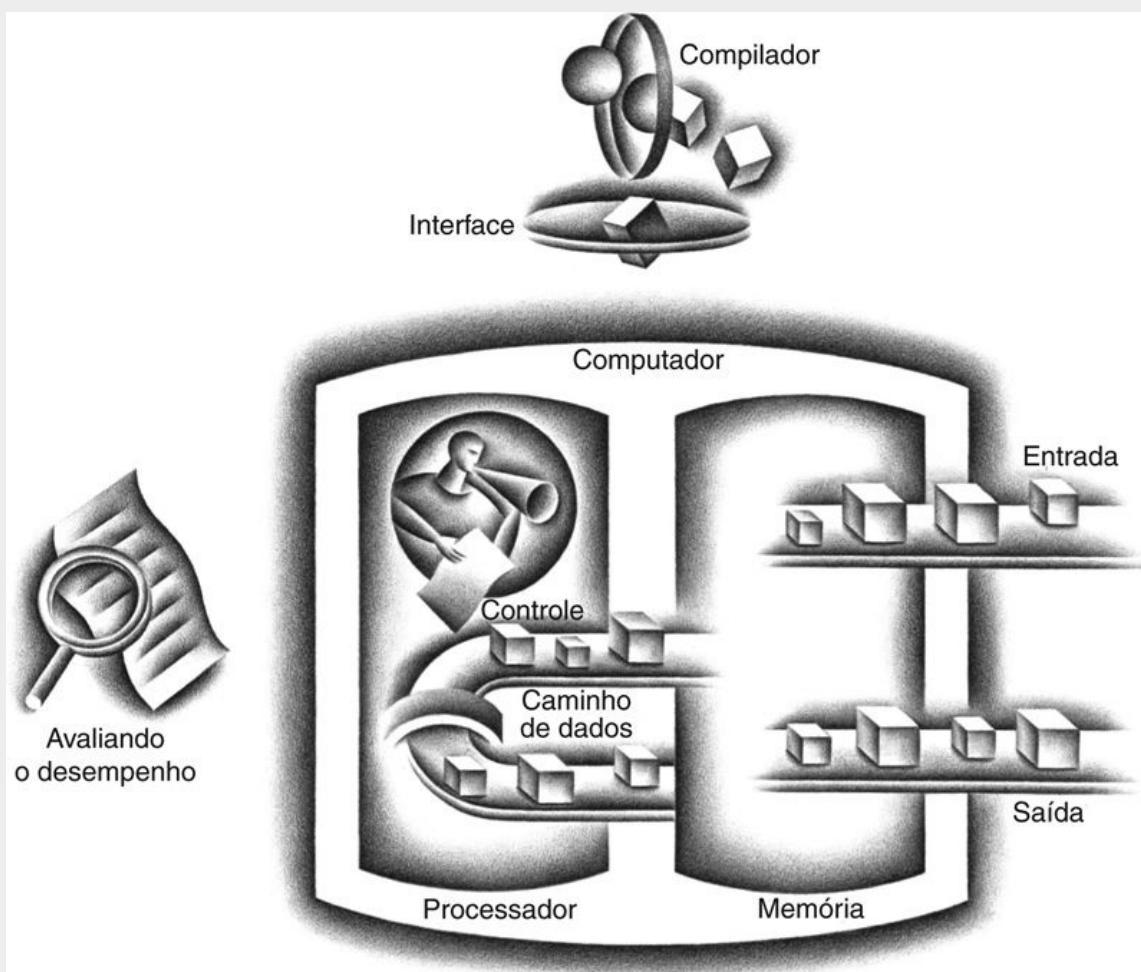


FIGURA 1.5 A organização de um computador, mostrando os cinco componentes clássicos.

O processador obtém instruções e dados da memória. A entrada escreve dados na memória e a saída lê os dados desta. O controle envia os sinais que determinam as operações do caminho de dados, da memória, da entrada e da saída.

Através do espelho

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, Scientific American, 1984

Talvez o dispositivo de E/S mais fascinante seja o monitor gráfico. A maioria dos dispositivos móveis pessoais utiliza **monitores de cristal líquido (LCDs)** para obterem uma tela fina, com baixa potência. O LCD não é a fonte da luz; em vez disso, ele controla a transmissão da luz. Um LCD típico inclui moléculas em forma de bastão em um líquido que forma uma hélice giratória que encurva a luz que entra na tela, de uma fonte de luz atrás da tela ou, menos frequentemente, da luz refletida. Os bastões se esticam quando uma corrente é aplicada e não encurvam mais a luz. Como o material de cristal líquido está entre duas telas polarizadas a 90 graus, a luz não pode passar a não ser que esteja encurvada. Hoje, a maioria dos monitores LCD utiliza uma **matriz ativa** que tem uma minúscula chave de transistor em cada pixel para controlar a corrente com precisão e gerar imagens mais nítidas. Uma máscara vermelha-verde-azul associada a cada ponto na tela determina a intensidade dos três componentes de cor na imagem final; em um LCD de matriz ativa colorida, existem três chaves de transistores em cada ponto.

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.

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 1024×768 a 2048×1536 . 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.

pixel

O menor elemento individual da imagem. A tela é composta de centenas de milhares a milhões de pixels, organizados em uma matriz.

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 1.6](#) mostra um buffer de quadros com 4 bits por pixel.

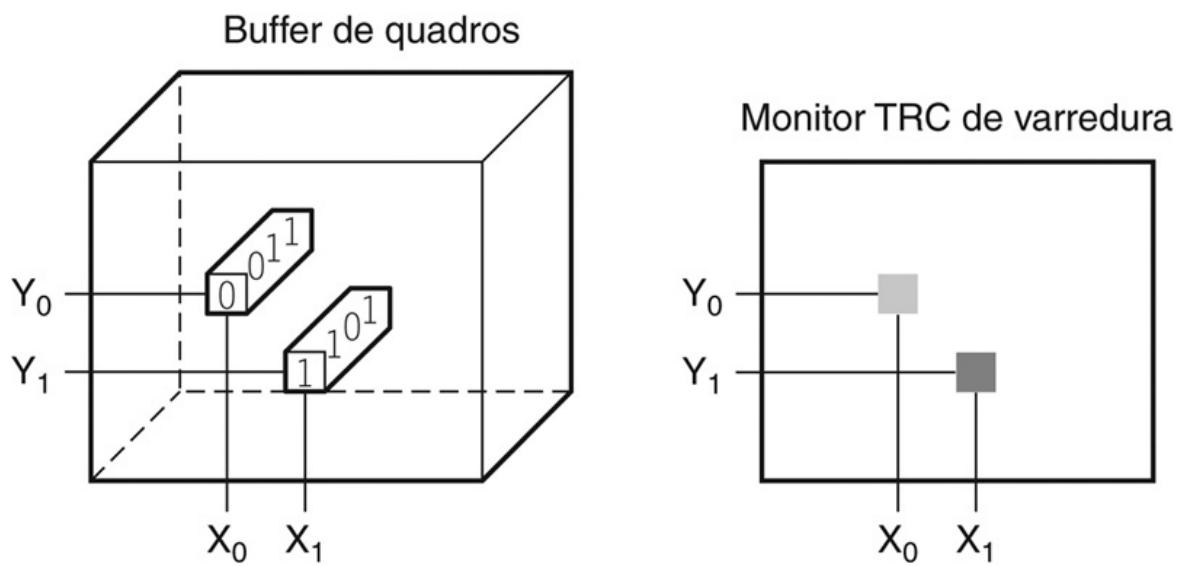


FIGURA 1.6 Cada coordenada no buffer de quadros à esquerda determina o tom da coordenada correspondente para o monitor TRC de varredura à direita.

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).

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 até mesmo as mais sutis mudanças na tela.

Touchscreen

Embora os PCs também usem monitores LCD, os tablets e smartphones da era pós-PC substituíram o mouse e o teclado por telas sensíveis ao toque, que tem a maravilhosa vantagem da interface do usuário para que este aponte diretamente para o que está interessado, em vez de fazer isso indiretamente com um mouse.

Embora existam diversas maneiras de implementar uma tela sensível ao toque, muitos tablets hoje utilizam a sensação capacitiva. Como as pessoas são condutores elétricos, se um isolante como o vidro for coberto com um condutor transparente, o toque distorce o campo eletrostático da tela, que resulta em uma mudança na capacidade. Essa tecnologia pode permitir múltiplos toques simultaneamente, o que permite gestos que possam levar a interfaces de usuário atraentes.

Abrindo o gabinete

A Figura 1.7 mostra o conteúdo do computador tablet iPad 2, da Apple. Não é surpresa que, dos cinco componentes clássicos do computador, a E/S domine esse dispositivo de leitura. A lista de dispositivos de E/S inclui uma tela LCD capacitiva multitoque, câmera frontal, câmera traseira, microfone, conector de headphone, alto-falantes, acelerômetro, giroscópio, rede Wi-Fi e rede Bluetooth. O caminho de dados, o controle e a memória são uma minúscula parte dos componentes.



FIGURA 1.7 Componentes do Apple iPad 2 A1395.

O fundo metálico do iPad (com o logo da Apple invertido no meio) está no centro. No topo está a tela multitoque capacitiva e o visor LCD. No canto direito está a bateria de polímero de 3,8 V e 25 W, que consiste em três capas de células de Li-ion e oferece 10 horas de vida da bateria. No canto esquerdo está o quadro metálico que conecta a LCD ao fundo do iPad. Os pequenos componentes em torno do fundo de metal no centro são o que consideramos o computador; eles normalmente têm a forma de um L para que se ajustem de modo compacto dentro da capa ao lado da bateria. A [Figura 1.8](#) mostra uma visão de perto da placa em forma de L no canto inferior esquerdo da capa metálica, que é a placa do circuito impresso lógico, que contém o processador e a memória. O minúsculo retângulo abaixo da placa lógica contém um chip que fornece a comunicação sem fio: Wi-Fi, Bluetooth e sintonizador de FM. Ele se encaixa em um pequeno conector no canto inferior esquerdo da placa lógica. Perto do canto superior esquerdo da capa existe outro

componente em forma de L, que é a montagem da câmera frontal, que inclui a câmera, conector do headphone e microfone. Perto do canto superior direito da capa está a placa contendo o controle de volume e o botão de silêncio e bloqueio de rotação de tela, junto com um giroscópio e acelerômetro. Esses dois últimos chips se combinam para permitir que o iPad reconheça o movimento em 6 eixos. O pequeno retângulo perto dele é a câmera traseira. Perto do canto inferior direito da capa está a montagem do alto-falante em forma de L. O cabo no fundo é o conector entre a placa lógica e a placa de controle da câmera/volume. A placa entre o cabo e a montagem do alto-falante é o controlador para a tela sensível capacitiva. (Cortesia da iFixit, www.ifixit.com)

Os pequenos retângulos na [Figura 1.8](#) contêm os dispositivos que impulsionam nossa tecnologia avançada, os **circuitos integrados**, apelidados de **chips**. O pacote A5 visto no meio da [Figura 1.8](#) contém dois processadores ARM que operam com uma taxa de clock de 1 GHz. 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. Ocasionalmente, as pessoas chamam o processador de **CPU**, que significa o termo pomposo **unidade central de processamento**.

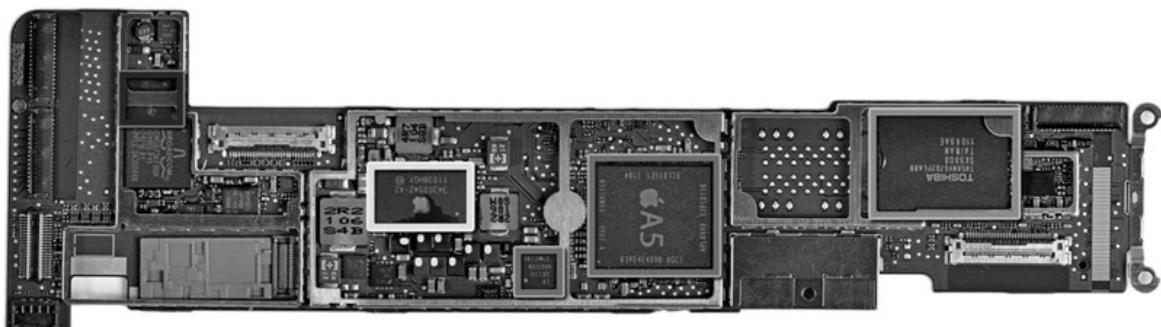


FIGURA 1.8 A placa lógica do Apple iPad 2 na [Figura 1.7](#).

A foto destaca cinco circuitos integrados. O circuito integrado grande no meio é o chip A5 da Apple, que contém os núcleos do processador ARM dual que rodam a 1 GHz, bem como a memória principal de 512 MB dentro do pacote. A [Figura 1.9](#) mostra uma fotografia do chip processador dentro do pacote A5. O chip de tamanho semelhante à esquerda é o chip de memória flash de 32 GB para o armazenamento não volátil. Há um espaço vazio entre os dois chips, onde um segundo chip flash pode ser

instalado para dobrar a capacidade de armazenamento do iPad. Os chips à direita do A5 incluem: controlador de potência e chips controladores de E/S. (Cortesia da iFixit, www.ifixit.com)

círculo integrado

Também chamado **chip**, é um dispositivo que combina de dezenas a milhões de transistores.

unidade central de processamento (CPU)

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

Penetrando ainda mais no hardware, a [Figura 1.9](#) revela detalhes de um microprocessador. O processador contém logicamente dois componentes principais: o caminho de dados e o controle, correspondendo, 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 4](#) explica o caminho de dados e o controle para um projeto de desempenho mais alto.



FIGURA 1.9 O circuito integrado do processador dentro do pacote A5. O tamanho do chip é 12,1 por 10,1 mm, e foi fabricado originalmente em um processo de 45 nm ([Seção 1.5](#)). Ele possui dois processadores ARM idênticos, ou núcleos, na metade esquerda do chip, e uma unidade de processamento gráfico (GPU) PowerVR com quatro caminhos de dados no quadrante superior esquerdo. No canto inferior esquerdo dos núcleos ARM estão as interfaces com a memória principal

(DRAM). (Cortesia da Chipworks, www.chipworks.com)

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.

O pacote A5 na [Figura 1.8](#) também inclui dois chips de memória, cada um com 2 gibibits de capacidade, fornecendo assim 512 MiB. A **memória** é onde os programas são mantidos quando estão sendo executados; ela também contém os dados necessários aos programas em execução. A memória é constituída 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 sequencial, como as fitas magnéticas, a parte *RAM* do termo DRAM significa que os acessos à memória levam o mesmo tempo, independentemente da parte da memória lida.

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.

Dynamic Random Access Memory (DRAM)

Memória construída como um circuito integrado para fornecer acesso aleatório a qualquer local. Os tempos de acesso são de 50 nanosegundos e o custo por gigabyte em 2012 era de US\$ 5 a US\$ 10.

Descer até as profundezas de qualquer componente de hardware revela os interiores da máquina. 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 RAM estática – **Static Random Access Memory (SRAM)**. A SRAM é mais rápida, mas menos densa e, portanto, mais cara do que a DRAM ([Capítulo 5](#)). SRAM e DRAM são duas camadas da **hierarquia de memória**.



HIERARQUIA

memória cache

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

Static Random Access Memory (SRAM)

Também uma memória montada como um circuito integrado, porém mais rápida e menos densa que a DRAM.

Como mencionado anteriormente, uma das grandes ideias para melhorar o projeto é a abstração. Uma das abstrações mais importantes é a interface entre o hardware e o software de nível mais baixo. Em decorrência de 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 o 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 da interface do sistema operacional fornecida para os programadores das aplicações é chamada de **interface binária de aplicação (ABI)**.



A B S T R A Ç Ã O

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.

Uma arquitetura do conjunto de instruções permite aos projetistas de computador falarem sobre funções, independentemente 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”.

implementação

Hardware que obedece à abstração de uma arquitetura.

Colocando em perspectiva

Tanto o hardware quanto o software consistem em camadas hierárquicas usando abstração, com cada camada inferior ocultando detalhes do nível acima. Uma interface-chave 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* com custo e desempenho variáveis executem um software idêntico.

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, um DVD não se esquece do filme quando você desliga o aparelho de DVD e, portanto, é uma tecnologia de **memória não volátil**.

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. Um disco de DVD é não volátil.

Para distinguir entre a memória usada para armazenar dados e programas enquanto estão sendo executados e essa memória não volátil usada para armazenar programas entre as execuções, o termo **memória principal** ou **memória primária** é usado para o primeiro e o termo **memória secundária** é usado para o último. A memória secundária forma a próxima camada inferior da hierarquia de memória. As DRAMs dominam a memória principal desde 1975; e os **discos magnéticos** dominam a memória secundária há mais tempo ainda. Devido ao seu tamanho e formato, os dispositivos móveis pessoais utilizam **memória flash**, uma memória semicondutora não volátil, no lugar dos discos. A [Figura 1.8](#) mostra o chip contendo a memória flash do iPad 2. Embora mais lenta que a DRAM, ela é muito mais barata, além de ser não volátil. Embora custando mais por bit do que os discos, ela é menor, vem em capacidades muito menores, é mais resistente e utiliza menos energia do que os discos. Logo, a memória flash é a memória secundária padrão para os PMDs. Infelizmente, diferente de discos e da DRAM, os bits da memória flash se desgastam após 100.000 a 1.000.000 de escritas. Assim, os sistemas de arquivos precisam acompanhar o número de escritas e ter uma estratégia para evitar o desgaste do armazenamento, por exemplo, movendo dados mais populares. O [Capítulo 5](#) descreve os discos e a memória flash com mais detalhes.



HIERARQUIA

memória principal

Também chamada **memória primária**. A memória usada para armazenar os programas enquanto estão sendo executados; normalmente consiste na DRAM nos computadores atuais.

memória secundária

Memória não volátil usada para armazenar programas e dados entre execuções; normalmente consiste em memória flash nos PMDs e discos magnéticos nos servidores.

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. Por serem dispositivos mecânicos rotativos, os tempos de acesso são cerca de 5 a 20 milissegundos e o custo por gigabyte em 2012 era de US\$ 0,05 a US\$ 0,10.

memória flash

Uma memória semicondutora não volátil. Ela é mais barata e mais lenta que a

DRAM, porém mais cara por bit e mais rápida que os discos magnéticos. Os tempos de acesso são cerca de 5 a 50 microsegundos, e o custo por gigabyte em 2012 era de US\$ 0,75 a US\$ 1,00.

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](#), que 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 próximo ao 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, transferindo até 40 gigabits por segundo. 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 espinha dorsal da Internet, que é o suporte da World Wide Web. Elas costumam ser baseadas em cabos de fibra ótica e são alugadas de empresas de telecomunicações.

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.

As redes mudaram a cara da computação nos últimos 30 anos, por se tornarem muito mais comuns e aumentarem drasticamente o desempenho. Na década de 1970, poucas pessoas tinham acesso ao correio eletrônico (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 bastante 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 30 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 1 a 40 gigabits por segundo, em geral compartilhada por, no máximo, alguns computadores. 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 drástico aumento no emprego das redes e em sua capacidade, tornaram a tecnologia de redes o ponto central para a revolução da informação nos últimos 30 anos.

Na última década, outra inovação na tecnologia de redes está reformulando a maneira como os computadores se comunicam. A tecnologia sem fio se tornou amplamente utilizada, o que permitiu a era pós-PC. 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.

Verifique você mesmo

- A memória semicondutora DRAM, a memória flash e o armazenamento de disco diferem significativamente. Para cada tecnologia, descreva a principal diferença quanto a cada um dos seguintes aspectos: volatilidade, tempo de acesso relativo aproximado e custo relativo aproximado em comparação com a DRAM.

1.5. 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 a fim de tentar vencer a corrida para projetar um computador melhor. A [Figura 1.10](#) mostra as tecnologias usadas ao longo do tempo, com uma estimativa do desempenho relativo por custo unitário para cada tecnologia. 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	Circuito integrado	900
1995	Circuito VLSI (Very Large Scale Integrated)	2.400.000
2013	Circuito ULSI (Ultra Large Scale Integrated)	250.000.000.000

FIGURA 1.10 Desempenho relativo, por custo unitário, das tecnologias usadas nos computadores ao longo do tempo.

Fonte: Computer Museum, Boston.

Um **transistor** é simplesmente uma chave liga/desliga controlada por eletricidade. O *circuito integrado (CI)* combinou dezenas a centenas de transistores em um único chip. Quando Gordon Moore previu a duplicação contínua dos recursos, ele estava prevendo a taxa de crescimento do número de transistores por chip. Para descrever o incrível aumento no número de transistores de centenas para milhões, a característica *escala muito grande* é acrescentado ao termo, criando a abreviação **VLSI** (de **circuito Very Large Scale Integrated**).

transistor

Uma chave liga/desliga controlada por um sinal elétrico.

circuito Very Large Scale Integrated (VLSI)

Um dispositivo com centenas de milhares a milhões de transistores.

Essa taxa de integração crescente tem se mantido notavelmente estável. A Figura 1.11 mostra o crescimento na capacidade da DRAM desde 1977. Durante décadas, a indústria quadruplicou consistentemente a capacidade a cada três anos, resultando em um aumento de mais de 16.000 vezes!

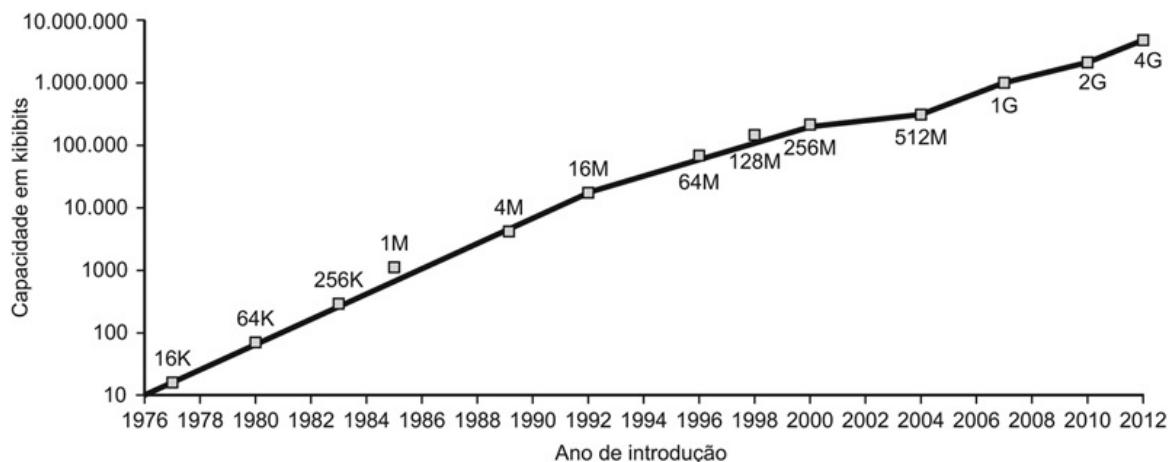


FIGURA 1.11 Crescimento da capacidade por chip de DRAM ao longo do tempo.

O eixo y é medido em kibibits (2^{10} bits). A indústria de DRAM quadruplicou a capacidade a cada quase três anos, um aumento de 60% por ano, durante 20 anos. Nos últimos anos, essa taxa diminuiu um pouco e está próximo do dobro a cada dois anos.

Para entender como fabricar circuitos integrados, começamos do início. A fabricação de um chip começa com o **silício**, uma substância encontrada na areia. Como o silício não conduz bem a eletricidade, ele 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)

silício

Um elemento natural que é um semicondutor.

semicondutor

Uma substância que não conduz eletricidade muito bem.

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 1.12](#) 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 30 cm de diâmetro e cerca de 30 a 60 cm de comprimento. Um lingote é finamente fatiado em **wafers**, com até 0,25 cm 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.

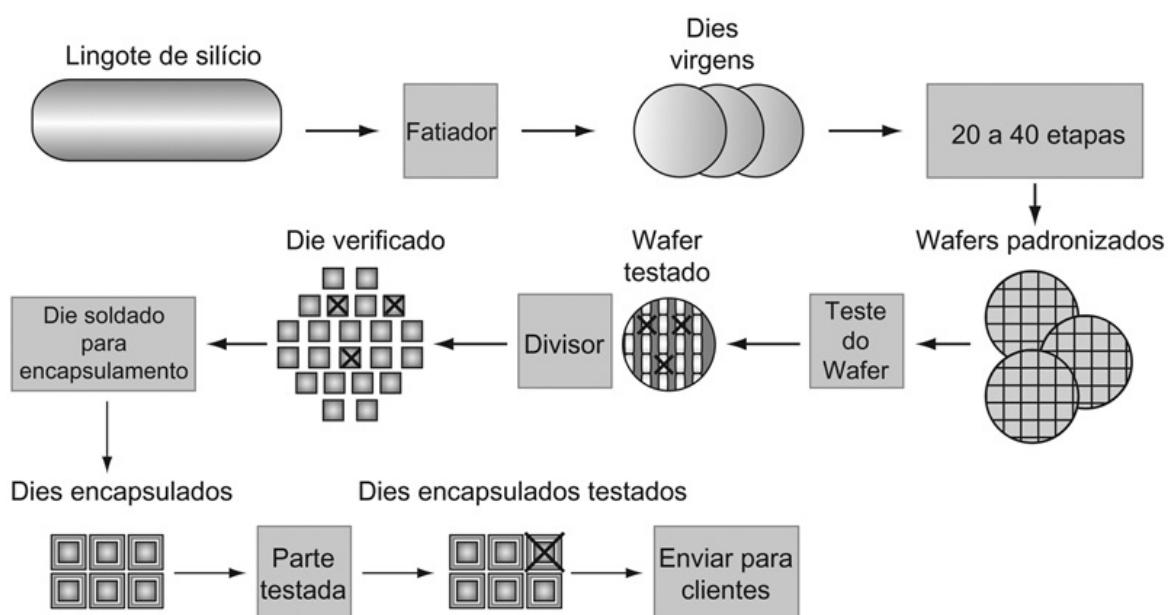


FIGURA 1.12 Processo de fabricação de um chip.

Após ser fatiado de um lingote de silício, os wafers virgens passam por 20 a 40 passos para criar wafers padronizados ([Figura 1.13](#)). Esses wafers padronizados são testados com um testador de wafers e é criado um mapa das partes boas. Depois, os wafers são divididos em dies (moldes) ([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, neste 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.

lingote de cristal de silício

Uma barra composta de um cristal de silício que possui entre 20 e 30cm de diâmetro e cerca de 30 a 60cm de comprimento.

wafer

Uma fatia de um lingote de silício de não mais que 2,5mm de espessura, usada para criar chips.

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. A estratégia mais simples para lidar com a imperfeição 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 1.13](#) é uma fotografia de um wafer com microprocessadores antes de serem cortados; anteriormente, a [Figura 1.9](#) mostrou um die individual do microprocessador e seus principais componentes.

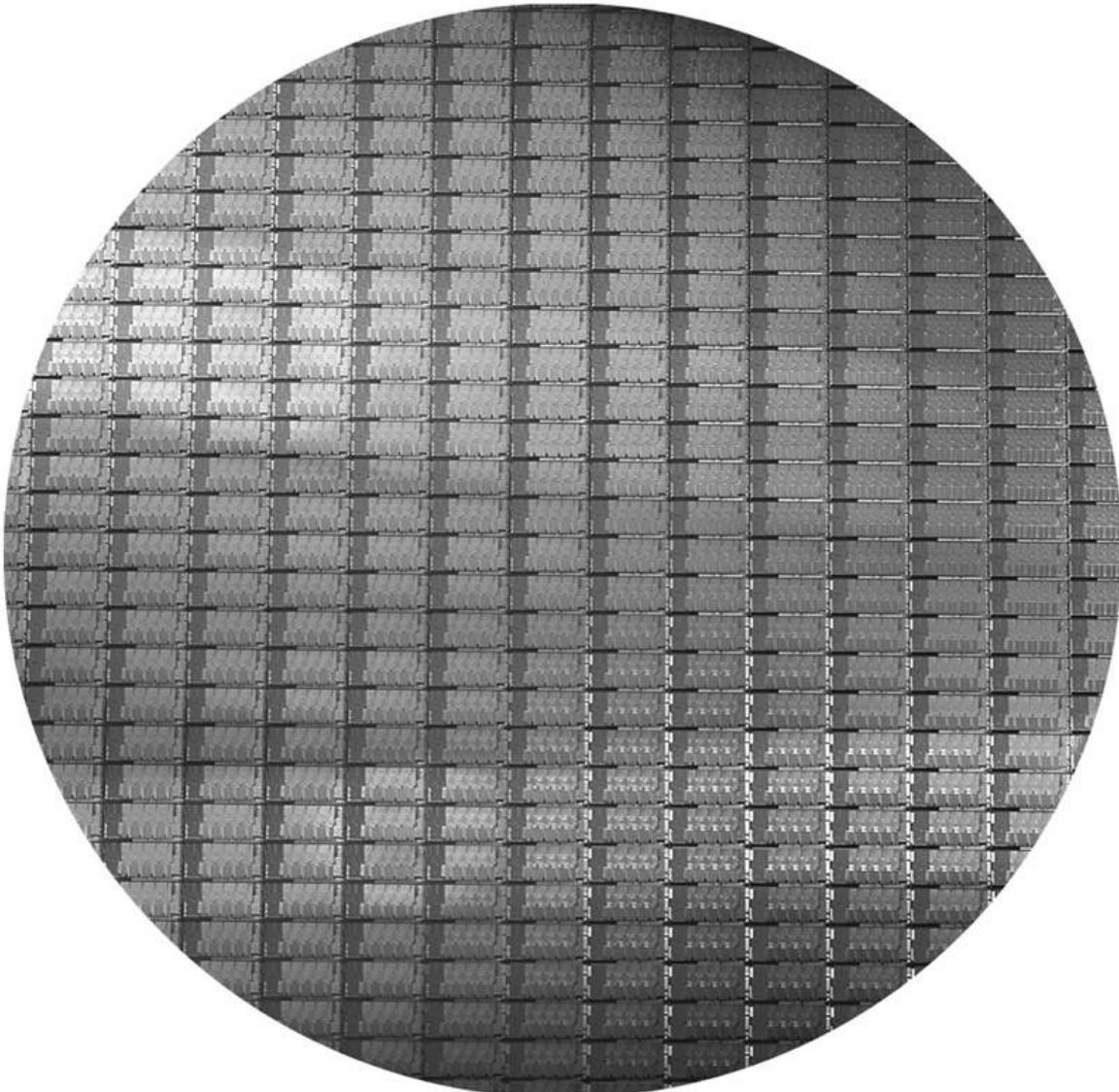


FIGURA 1.13 Um wafer de 300 mm de diâmetro dos chips Intel Core i7 (Cortesia da Intel).

O número de dies nesse wafer de 300 mm em 100% de aproveitamento é 280, cada um com 20,7 por 10,5 mm. 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. Esse die usa uma tecnologia de 32 nanômetros, o que significa que os menores recursos possuem um tamanho de aproximadamente 32 nm, embora normalmente sejam um pouco menores do que o tamanho real do recurso, que se refere ao tamanho dos transistores como “desenhados” versus o tamanho final fabricado.

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, mais informalmente conhecidos como chips.

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.

aproveitamento

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, em razão do aproveitamento mais baixo e do menor número de dies que pode 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. Um processo de 32 nanômetros (nm) era comum em 2012, o que significa basicamente que o menor tamanho de recurso no die é 32 nm.

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 aos clientes.

Detalhamento

O custo de um circuito integrado pode ser expresso em três equações simples:

$$\text{Custo por die} = \frac{\text{Custo por wafer}}{\text{Dies por wafer} \times \text{aproveitamento}}$$

$$\text{Dies por wafer} = \frac{\text{Área do wafer}}{\text{Área do die}}$$

$$\text{Aproveitamento} = \frac{1}{(1 + (\text{Defeitos por área} \times \text{Área do die} / 2))^2}$$

A primeira equação é simples de se derivar. A segunda é uma aproximação, pois não subtrai a área perto da borda do wafer arredondado que não pode acomodar os dies retangulares (Figura 1.13). A equação final é baseada nas observações empíricas dos aproveitamentos nas fábricas de circuito integrado, com o expoente relacionado ao número de etapas de processamento crítico.

Logo, dependendo da taxa de defeito e do tamanho do die e wafer, os custos geralmente não são lineares em relação à área do die.

Verifique você mesmo

Um fator chave para determinar o custo de um circuito integrado é o volume. Quais dos seguintes são motivos pelos quais um chip fabricado em grande volume custaria menos?

1. Com grandes volumes, o processo de manufatura pode ser ajustado para um projeto em particular, aumentando o aproveitamento.
2. É menos trabalhoso projetar uma peça com alto volume do que uma peça com baixo volume.
3. As máscaras usadas para criar o chip são caras, de modo que o custo por chip é mais baixo para volumes mais altos.
4. Os custos de desenvolvimento da engenharia são altos e, em grande parte, não dependem do volume; assim, o custo de desenvolvimento por die é mais baixo com peças de alto volume.
5. Peças de alto volume normalmente possuem menores tamanhos de die do que as peças de baixo volume e, portanto, possuem maior

aproveitamento por wafer.

1.6. Desempenho

Avaliar o desempenho dos computadores pode ser desafiador. A escala e a complexidade dos sistemas de software modernos, junto com a grande variedade de técnicas de melhoria de desempenho empregadas por projetistas de hardware, tornaram a avaliação do desempenho muito mais difícil.

Ao tentar escolher entre diferentes computadores, o desempenho é um atributo importante. Comparar e avaliar com precisão diferentes computadores é crítico para compradores e por consequência, também para os projetistas. O pessoal que vende computadores também sabe disso. Normalmente, os vendedores desejam que você veja seu computador da melhor maneira possível, não importa se isso reflete ou não as necessidades da aplicação do comprador. Logo, ao escolher um computador, é importante entender como medir melhor o desempenho e as limitações das medições de desempenho.

O restante desta seção descreve diferentes maneiras como o desempenho pode ser determinado; depois, descrevemos as métricas para avaliar o desempenho do ponto de vista de um usuário do computador e de um projetista. Também analisamos como essas métricas estão relacionadas e apresentamos a equação clássica de desempenho do processador, que usaremos no decorrer do texto.

Definindo o desempenho

Quando dizemos que um computador tem melhor desempenho que outro, o que queremos dizer? Embora essa pergunta possa parecer simples, uma analogia com aviões de passageiros mostra como a questão de desempenho pode ser sutil. A Figura 1.14 mostra alguns aviões de passageiros típicos, juntamente com a velocidade de cruzeiro, autonomia e capacidade. Se você quisesse saber qual dos aviões nessa tabela tem o melhor desempenho, primeiro precisaríamos definir o desempenho. Por exemplo, considerando diferentes medidas de desempenho, vemos que o avião com a maior velocidade de cruzeiro é o Concorde (que saiu de serviço em 2003), o avião com a maior autonomia é o DC-8, e o avião com a maior capacidade é o 747.

Aeronave	Capacidade	Autonomia (milhas)	Velocidade (m.p.h.)	Passenger throughput (passageiros × m.p.h.)
Boeing 777	375	4630	610	228.750
Boeing 747	470	4150	610	286.700
BAC/Sul Concorde	132	4000	1350	178.200
Douglas DC-8-50	146	8720	544	79.424

FIGURA 1.14 A capacidade, autonomia e velocidade de uma série de aviões comerciais.

A última coluna mostra a taxa com que o avião transporta passageiros, que é a capacidade vezes a velocidade de voo (ignorando a autonomia e os tempos de decolagem e pouso).

Vamos supor que o desempenho seja definido em termos de velocidade. Isso ainda deixa duas definições possíveis. Você poderia definir o avião mais rápido como aquele com a velocidade de voo mais alta, levando um único passageiro de um ponto para outro com o menor tempo. Porém, se você estivesse interessado em transportar 450 passageiros de um ponto para outro, o 747 certamente seria o mais rápido, como mostra a última coluna da figura. De modo semelhante, podemos definir o desempenho do computador de diferentes maneiras.

Se você estivesse rodando um programa em dois computadores desktop diferentes, diria que o mais rápido é o computador que termina o trabalho primeiro. Se estivesse gerenciando um centro de dados com diversos servidores rodando tarefas submetidas por muitos usuários, você diria que o computador mais rápido é aquele que completasse o máximo 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 conhecido como **tempo de execução**. Os gerentes de centro de dados normalmente estão interessados em aumentar o **throughput** ou **largura de banda** — a quantidade total de trabalho realizado em determinado tempo. Logo, na maioria dos casos, ainda precisaremos de diferentes métricas de desempenho, além de diferentes conjuntos de aplicações para avaliar computadores embutidos e de desktop, que são mais voltados para o tempo de resposta, contra servidores, que são mais voltados para o throughput.

tempo de resposta

Também chamado **tempo de execução**. O tempo total exigido para o computador completar uma tarefa, incluindo acessos ao disco, acessos à memória, atividades de E/S, overhead do sistema operacional, tempo de

execução de CPU e assim por diante.

throughput

Também chamado **largura de banda**. Outra medida de desempenho, é o número de tarefas completadas por unidade de tempo.

Throughput e tempo de resposta

Exemplo

As mudanças a seguir em um sistema de computador aumentam o throughput, diminuem o tempo de resposta ou ambos?

1. Substituir o processador em um computador por uma versão mais rápida.
2. Acrescentar processadores adicionais a um sistema que utiliza múltiplos processadores para tarefas separadas — por exemplo, busca na Web.

Resposta

Diminuir o tempo de resposta quase sempre melhora o throughput. Logo, no caso 1, o tempo de resposta e o throughput são melhorados. No caso 2, ninguém realiza o trabalho mais rapidamente, de modo que somente o throughput aumenta.

Porém, se a demanda para processamento no segundo caso fosse quase tão grande quanto o throughput, o sistema poderia forçar as solicitações a se enfileirarem. Nesse caso, aumentar o throughput também poderia melhorar o tempo de resposta, pois poderia reduzir o tempo de espera na fila. Assim, em muitos sistemas de computadores reais, mudar o tempo de execução ou o throughput normalmente afeta o outro.

Na discussão sobre o desempenho dos computadores, vamos nos preocupar principalmente com o tempo de resposta nos primeiros capítulos. Para maximizar o desempenho, queremos minimizar o tempo de resposta ou o tempo de execução para alguma tarefa. Assim, podemos relacionar desempenho e tempo de execução para o 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 for maior 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 que o de X, se X for mais rápido que Y.

Na discussão de um projeto de computador, normalmente queremos relacionar o desempenho de dois computadores diferentes quantitativamente. Usaremos a frase “X é n vezes mais rápido que Y” — ou, de modo equivalente, “X tem n vezes a velocidade de Y” — para indicar

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

Se X for n vezes mais rápido que Y, então o tempo de execução em Y é n vezes maior do que em X:

$$\frac{\text{Desempenho}_X}{\text{Desempenho}_Y} = \frac{\text{Tempo de execução}_Y}{\text{Tempo de execução}_X} = n$$

Desempenho relativo

Exemplo

Se o 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 que B?

Resposta

Sabemos que A é n vezes mais rápido que B se

$$\frac{\text{Desempenho}_A}{\text{Desempenho}_B} = \frac{\text{Tempo de execução}_B}{\text{Tempo de execução}_A} = n$$

Assim, a razão de desempenho é

$$\frac{15}{10} = 1,5$$

e A, portanto, é 1,5 vez mais rápido que B.

No exemplo anterior, também poderíamos dizer que o computador B é 1,5 vez *mais lento* que o computador A, pois

$$\frac{\text{Desempenho}_A}{\text{Desempenho}_B} = 1,5$$

significando que

$$\frac{\text{Desempenho}_A}{1,5} = \text{Desempenho}_B$$

Para simplificar, normalmente usaremos a terminologia *mais rápido que* quando tentamos comparar computadores quantitativamente. Como o desempenho e o tempo de execução são recíprocos, aumentar o desempenho requer diminuir o tempo de execução. Para evitar a confusão em potencial entre os termos *aumentar* e *diminuir*, normalmente dizemos “melhorar o desempenho” ou “melhorar o tempo de execução” quando queremos dizer “aumentar o desempenho” e “diminuir o tempo de execução”.

Medindo o desempenho

O tempo é a medida de desempenho do computador: 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. Porém, o tempo pode ser definido de diferentes maneiras, dependendo do que contamos. A definição mais clara de tempo é chamada de *tempo do relógio*, *tempo de resposta* ou *tempo decorrido*. Esses termos significam o tempo total para completar uma tarefa, incluindo acessos ao disco, acessos à memória, atividades de *entrada/saída* (E/S), overhead do sistema operacional — tudo.

Contudo, os computadores normalmente são compartilhados e um processador pode trabalhar em vários programas simultaneamente. Nesses casos, o sistema pode tentar otimizar o throughput em vez de tentar minimizar o tempo decorrido para um programa. Logo, normalmente queremos distinguir entre o tempo decorrido e o tempo que o processador está trabalhando em nosso favor. **Tempo de execução de 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 pela E/S ou pela execução de outros programas. (Lembre-se, porém, de que o tempo de resposta experimentado pelo usuário será o tempo decorrido do programa, e não o tempo de CPU.) O tempo de CPU pode ser dividido ainda mais em tempo de CPU gasto no programa, chamado **tempo de CPU do usuário**, e o tempo de CPU gasto no sistema operacional, realizando

tarefas em favor do programa, chamado **tempo de CPU do sistema**. A diferenciação entre o tempo de CPU do sistema e do usuário é difícil de se realizar com precisão, pois normalmente é difícil atribuir a responsabilidade pelas atividades do sistema operacional a um programa do usuário em vez do outro, e por causa das diferenças de funcionalidade entre os sistemas operacionais.

tempo de execução de CPU

Também chamado tempo de CPU. O tempo real que a CPU gasta calculando para uma tarefa específica.

tempo de CPU do usuário

O tempo de CPU gasto em um programa propriamente dito.

tempo de CPU do sistema

O tempo de CPU gasto no sistema operacional realizando tarefas em favor do programa.

Por uma questão de consistência, mantemos uma distinção entre o desempenho baseado no tempo decorrido e baseado no tempo de execução da CPU. Usaremos o termo *desempenho do sistema* para nos referirmos ao tempo decorrido em um sistema não carregado e *desempenho da CPU* para nos referirmos ao tempo de CPU do usuário. Vamos focalizar o desempenho da CPU neste capítulo, embora nossas discussões de como resumir o desempenho possam ser aplicadas às medições de tempo decorrido ou tempo de CPU.

Entendendo o desempenho do programa

Diferentes aplicações são sensíveis a diferentes aspectos do desempenho de um sistema de computador. Muitas aplicações, especialmente aquelas rodando em servidores, dependem muito do desempenho da E/S, que, por sua vez, conta com o hardware e o software. O tempo decorrido total medido por um relógio comum é a medida de interesse. Em alguns ambientes de aplicação, o usuário pode se importar com o throughput, tempo de resposta ou uma combinação complexa dos dois (por exemplo, o throughput máximo com o

tempo de resposta, no pior caso). Para melhorar o desempenho de um programa, deve-se ter uma definição clara de qual métrica de desempenho interessa e depois prosseguir para procurar gargalos de desempenho medindo a execução do programa e procurando os prováveis gargalos. Nos próximos capítulos, vamos descrever como procurar gargalos e melhorar o desempenho em diversas partes do sistema.

Embora, como usuários de computador, nos importemos com o tempo, quando examinamos os detalhes de um computador, é conveniente pensar sobre o desempenho em outras métricas. Em particular, os projetistas de computação podem querer pensar a respeito de um computador usando uma medida que se relaciona à velocidade com que o hardware pode realizar suas funções básicas. Quase todos os computadores são construídos usando-se um clock que determina quando os eventos ocorrem no hardware. Esses intervalos de tempo discretos são chamados de **ciclos de clock** (ou **batidas, batidas de clock, períodos de clock, clocks, ciclos**). Os projetistas referem-se à extensão de um **período de clock** como o tempo para um *ciclo de clock* completo (por exemplo, 250 picosegundos ou 250 ps) e como a *taxa de clock* (por exemplo, 4 gigahertz ou 4 GHz), que é o inverso do período de clock. Na próxima subseção, formalizaremos o relacionamento entre os ciclos de clock do projetista de hardware e os segundos do usuário do computador.

ciclo de clock

Também chamado **batida, batida de clock, período de clock, clock, ciclo**. O tempo para um período de clock, normalmente do clock do processador, que trabalha a uma taxa constante.

período de clock

A extensão de cada ciclo de clock.

Verifique você mesmo

1. Suponha que saibamos que uma aplicação que usa dispositivos pessoais móveis e a nuvem seja limitada pelo desempenho da rede. Para as mudanças a seguir, indique se: somente o throughput melhora, o tempo de resposta e o throughput melhoram, ou nenhum destes melhora.

- a. Um canal de rede extra é acrescentado entre o PMD e a nuvem, aumentando o throughput total da rede e reduzindo o atraso para obter o acesso à rede (pois agora existem dois canais).
 - b. O software de rede é melhorado, reduzindo assim o atraso na comunicação da rede, mas não aumentando o throughput.
 - c. Mais memória é acrescentada ao computador.
2. O desempenho do computador C é quatro vezes mais rápido que o desempenho do computador B, que executa determinada aplicação em 28 segundos. Quanto tempo o computador C levará para executar esta aplicação?

Desempenho da CPU e seus fatores

Usuários e projetistas normalmente examinam o desempenho usando diferentes métricas. Se pudéssemos relacionar essas diferentes métricas, poderíamos determinar o efeito de uma mudança de projeto sobre o desempenho experimentado pelo usuário. Como estamos interessados no desempenho da CPU neste ponto, a medida de desempenho final é o tempo de execução da CPU. Uma fórmula simples relaciona as métricas mais básicas (ciclos de clock e tempo do ciclo de clock) ao tempo da CPU:

Tempo de execução da CPU para um programa = Ciclos de clock da CPU para um programa
X Tempo do ciclo de clock

Como alternativa, como a taxa de clock e o tempo do ciclo de clock são inversos,

$$\text{Tempo de execução da CPU para um programa} = \frac{\text{Ciclos de clock da CPU para um programa}}{\text{Taxa de clock}}$$

Essa fórmula deixa claro que o projetista de hardware pode melhorar o desempenho reduzindo o número de ciclos de clock exigidos para um programa ou o tamanho do ciclo de clock. Conforme veremos em outros capítulos, os projetistas normalmente têm de escolher entre o número de ciclos de clock necessários para um programa e a extensão de cada ciclo. Muitas técnicas que diminuem o número de ciclos de clock podem também aumentar o tempo do

ciclo de clock.

Melhorando o desempenho

Exemplo

Nosso programa favorito executa em 10 segundos no computador A, que tem um clock de 2 GHz. Estamos tentando ajudar um projetista de computador a montar um computador B, que executará esse programa em 6 segundos. O projetista determinou que é possível haver um aumento substancial na taxa de clock, mas esse aumento afetará o restante do projeto da CPU, fazendo com que o computador B exija 1,2 vez a quantidade de ciclos de clock do computador A para esse programa. Que taxa de clock o projetista deve ter como alvo?

Resposta

Vamos primeiro achar o número de ciclos de clock exigidos para o programa em A:

$$\text{Tempo de CPU}_A = \frac{\text{Ciclos de clock de CPU}_A}{\text{Taxa de clock}_A}$$

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

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

O tempo de CPU para B pode ser encontrado por meio desta equação:

$$\text{Tempo de CPU}_B = \frac{1,2 \times \text{Ciclos de CPU}_A}{\text{Taxa de clock}_B}$$

$$6 \text{ segundos} = \frac{1,2 \times 20 \times 10^9 \text{ ciclos}}{\text{Taxa de clock}_B}$$

$$\text{Taxa de clock}_B = \frac{1,2 \times 20 \times 10^9 \text{ ciclos}}{6 \text{ segundos}} = \frac{0,2 \times 20 \times 10^9 \text{ ciclos}}{\text{segundo}} = \frac{4 \times 10^9 \text{ ciclos}}{\text{segundo}} = 4 \text{ GHz}$$

Para executar o programa em 6 segundos, B deverá ter o dobro da taxa de clock de A.

Desempenho da instrução

Essas equações de desempenho não incluíram qualquer referência ao número de instruções necessárias para o programa. Porém, como o compilador claramente gerou instruções para executar, e o computador teve de rodá-las para executar o programa, o tempo de execução dependerá do número de instruções em um programa. Um modo de pensar a respeito do tempo de execução é que ele é igual ao número de instruções executadas multiplicado pelo tempo médio por instrução. Portanto, o número de ciclos de clock exigido para um programa pode ser escrito como

$$\text{Ciclos de clock de CPU} = \text{Instruções para um programa} \times \text{Ciclos de clock médios por instrução}$$

O termo **ciclos de clock por instrução**, que é o número médio de ciclos de clock que cada instrução leva para executar, normalmente é abreviado como **CPI**. Como diferentes instruções podem exigir diferentes quantidades de tempo, dependendo do que elas fazem, CPI é uma média de todas as instruções executadas no programa. CPI oferece um modo de comparar duas implementações diferentes da mesma arquitetura do conjunto de instruções, pois

o número de instruções executadas para um programa, logicamente, será o mesmo.

ciclos de clock por instruções (CPI)

Número médio de ciclos de clock por instrução para um programa ou fragmento de programa.

Usando a equação de desempenho

Exemplo

Suponha que tenhamos duas implementações da mesma arquitetura de conjunto de instruções. O computador A tem um tempo de ciclo de clock de 250 ps e um CPI de 2,0 para algum programa, e o computador B tem um tempo de ciclo de clock de 500 ps e um CPI de 1,2 para o mesmo programa. Qual computador é mais rápido para este programa e por quanto?

Resposta

Sabemos que cada computador executa o mesmo número de instruções para o programa; vamos chamar esse número de I . Primeiro, encontramos o número de ciclos de clock do processador para cada computador:

$$\text{Ciclos de clock de CPU}_A = I \times 2,0$$

$$\text{Ciclos de clock de CPU}_B = I \times 1,2$$

Agora, podemos calcular o tempo de CPU para cada computador:

$$\begin{aligned}\text{Tempo de CPU}_A &= \text{Ciclos de clock de CPU}_A \times \text{Tempo de ciclo de clock} \\ &= I \times 2,0 \times 250 \text{ ps} = 500 \times I \text{ ps}\end{aligned}$$

De modo semelhante, para B:

$$\text{Tempo de CPU}_B = I \times 1,2 \times 500 \text{ ps} = 600 \times I \text{ ps}$$

Claramente, o computador A é mais rápido. O resultado da diferença de velocidade é dado 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.

A equação clássica de desempenho da CPU

Agora, podemos escrever essa equação de desempenho básica em termos do **contador de instrução** (o número de instruções executadas pelo programa), CPI e tempo de ciclo do clock:

Tempo de CPU = Contador de instrução X CPU X Tempo de ciclo de clock

ou então, como a taxa de clock é o inverso do tempo de ciclo de clock:

$$\text{Tempo de CPU} = \frac{\text{Contador de instrução} \times \text{CPI}}{\text{Taxa de clock}}$$

contador de instrução

O número de instruções executadas pelo programa.

Essas fórmulas são particularmente úteis porque separam os três fatores principais que afetam o desempenho. Podemos usá-las para comparar duas implementações diferentes ou para avaliar uma alternativa de projeto se soubermos seu impacto sobre esses três parâmetros.

Comparando segmentos de código

Exemplo

Um projetista de compilador está tentando decidir entre duas sequências de código para determinado computador. Os projetistas de hardware forneceram os seguintes fatos:

	CPI para cada classe de instrução		
	A	B	C
CPI	1	2	3

Para determinada instrução na linguagem de alto nível, o escritor do compilador está considerando duas sequências de código que exigem as seguintes contagens de instruções:

Sequência de código	Contagens de instruções para cada classe de instrução		
	A	B	C
1	2	1	2
2	4	1	1

Qual sequência de código executa mais instruções? Qual será mais rápida? Qual é o CPI para cada sequência?

Resposta

A sequência 1 executa $2 + 1 + 2 = 5$ instruções. A sequência 2 executa $4 + 1 + 1 = 6$ instruções. Portanto, a sequência 1 executa menos instruções.

Podemos usar a equação para ciclos de clock de CPU com base na contagem de instruções e CPI, a fim de descobrir o número total de ciclos de clock para cada sequência:

$$\text{Ciclos de clock de CPU} = \sum_{i=1}^n (\text{CPI}_i \times C_i)$$

Isso gera:

Ciclos de clock de CPU₁ = (2 × 1) + (1 × 2) + (2 × 3) = 2 + 2 + 6 = 10 ciclos

Ciclos de clock de CPU₂ = (4 × 1) + (1 × 2) + (1 × 3) = 4 + 2 + 3 = 9 ciclos

Assim, a sequência de código 2 é mais rápida, embora execute uma instrução extra. Como a sequência de código 2 leva menos ciclos de clock em geral, mas tem mais instruções, ela deverá ter um CPI menor. Os valores de CPI podem ser calculados por

$$CPI = \frac{\text{Ciclos de clock de CPU}}{\text{Contagem de instruções}}$$

$$CPI_1 = \frac{\text{Ciclos de clock de CPU}_1}{\text{Contagem de instruções}_1} = \frac{10}{5} = 2,0$$

$$CPI_2 = \frac{\text{Ciclos de clock de CPU}_2}{\text{Contagem de instruções}_2} = \frac{9}{6} = 1,5$$

Colocando em perspectiva

A Figura 1.15 mostra as medições básicas em diferentes níveis no computador e o que está sendo medido em cada caso. Podemos ver como esses fatores são combinados para gerar um tempo de execução medido em segundos por programa:

Componentes de desempenho	Unidades de medida
Tempo de execução de um programa pela CPU	Programa pelo segundo
Contagem de instruções	Instruções pelo programa
Instruções por ciclos de Clock (CPI)	Número médio de ciclos de clock por instrução
Tempo de ciclo de clock	Segundos por ciclos de clock

FIGURA 1.15 Os componentes básicos do desempenho e como cada um é medido.

$$\text{Tempo} = \frac{\text{Segundos}}{\text{Programa}} = \frac{\text{Instruções}}{\text{Programa}} \times \frac{\text{Ciclos de clock}}{\text{Instrução}} \times \frac{\text{Segundos}}{\text{Ciclo de clock}}$$

Lembre-se sempre de que a única medida completa e confiável do desempenho do computador é o tempo. Por exemplo, mudar o conjunto de instruções para reduzir sua contagem, pode levar a uma organização com tempo de ciclo de clock menor ou CPI maior, que compensa a melhoria na contagem de instruções. De modo semelhante, como o CPI depende do tipo das instruções executadas, o código que executa o menor número de instruções pode não ser o mais rápido.

Como determinar o valor desses fatores na equação de desempenho? Podemos medir o tempo de execução da CPU rodando 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 se obter. Naturalmente, se soubermos a taxa de clock e o tempo de execução da CPU, só precisamos da contagem de instruções ou do CPI para determinar o outro.

Podemos medir a contagem de instruções usando ferramentas de software que determinam o perfil da execução ou usando um simulador da arquitetura. Como alternativa, podemos usar contadores de hardware, que estão incluídos na maioria dos processadores, para registrar uma série de medidas, incluindo o número de instruções executadas, o CPI médio e, frequentemente, 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 conhecer todos os detalhes da implementação. Porém, o CPI depende de

diversos detalhes de projeto no computador, incluindo o sistema de memória e a estrutura do processador (conforme veremos nos [Capítulos 4 e 5](#)), além da mistura de tipos de instruções executados em uma aplicação. Assim, o CPI varia por aplicação, bem como entre implementações com o mesmo conjunto de instruções.

O exemplo anterior mostra o perigo de usar apenas um fator (contagem de instruções) para avaliar o desempenho. Ao comparar dois computadores, você precisa examinar todos os três componentes, que se combinam para formar o tempo de execução. Se alguns dos fatores forem idênticos, como a taxa de clock no exemplo anterior, o desempenho pode ser determinado 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, mesmo que as taxas de clock sejam idênticas. Vários exercícios ao final deste capítulo lhe pedem para avaliar uma série de melhorias de computador e compilador, que afetam a taxa de clock, CPI e contagem de instruções. Na [Seção 1.10](#), examinaremos uma medida de desempenho comum, que não incorpora todos os termos e, portanto, pode ser enganosa.

mix de instruções

Uma medida da frequência dinâmica das instruções por um ou muitos programas.

Entendendo o desempenho do programa

O desempenho de um programa depende do algoritmo, da linguagem, do compilador, da arquitetura e do hardware real. 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	O algoritmo determina o número de instruções do programa-fonte executadas e, portanto, o número de instruções de processador executadas. O algoritmo também pode afetar o CPI, favorecendo instruções mais lentas ou mais rápidas. Por exemplo, se o algoritmo utiliza mais operações de ponto flutuante, ele tenderá a ter um CPI mais alto.

	„ possivelmente CPI	
Linguagem de programação	Contagem de instruções, CPI	A linguagem de programação certamente afeta a contagem de instruções, pois as instruções na linguagem são traduzidas para instruções de processador, o que determina a contagem de instruções. A linguagem também pode afetar o CPI por causa dos seus recursos; por exemplo, uma linguagem com um suporte intenso para abstração de dados (por exemplo, Java) exigirá chamadas indiretas, que usarão instruções de CPI mais alto.
Compilador	Contagem de instruções, CPI	A eficiência do compilador afeta a contagem de instruções e os ciclos médios por instruções, pois o compilador determina a tradução das instruções da linguagem-fonte para instruções do computador. O papel do compilador pode ser muito complexo e afetar o CPI de maneiras complexas.
Arquitetura do conjunto de instruções	Contagem de instruções, taxa de clock, CPI	A arquitetura do conjunto de instruções afeta todos os três aspectos do desempenho da CPU, pois afeta as instruções necessárias para uma função, o custo em ciclos de cada instrução e a taxa de clock geral do processador.

Detalhamento

Embora você possa esperar que o CPI mínimo seja 1,0, conforme veremos no Capítulo 4, alguns processadores buscam e executam múltiplas instruções por ciclo de clock. Para refletir essa técnica, alguns projetistas invertem o CPI para falar sobre *IPC*, ou *instruções por ciclo de clock*. Se um processador executa, em média, duas instruções por ciclo de clock, então ele tem um IPC de 2 e, portanto, um CPI de 0,5.

Detalhamento

Embora o ciclo de clock tenha sido tradicionalmente fixo, para economizar energia ou aumentar temporariamente o desempenho, os processadores atuais podem variar suas taxas de clock, de modo que precisaríamos usar a taxa de clock *média* para um programa. Por exemplo, o Intel Core i7 aumentará

temporariamente a taxa de clock em cerca de 10% até que o chip aqueça muito. A Intel chama isso de *modo Turbo*.

Verifique você mesmo

Determinada aplicação escrita em Java roda por 15 segundos em um processador de desktop. Um novo compilador Java é lançado, exigindo apenas 60% das instruções do compilador antigo. Infelizmente, isso aumenta o CPI em 1,1. Com que velocidade podemos esperar que a aplicação rode usando esse novo compilador? Escolha a resposta certa a partir das três opções a seguir:

$$\frac{15 \times 0,6}{1,1} = 8,2 \text{ seg}$$

a. $1,1$

b. $15 \times 0,6 \times 1,1 = 9,9 \text{ seg}$

$$\frac{15 \times 1,1}{0,6} = 27,5 \text{ seg}$$

c. $0,6$

1.7. A barreira da potência

A Figura 1.16 mostra o aumento na taxa de clock e na potência de oito gerações de microprocessadores Intel durante 30 anos. Tanto a taxa de clock quanto a potência aumentaram rapidamente durante décadas e depois se estabilizaram recentemente. O motivo pelo qual elas cresceram juntas é que estão correlacionadas e o motivo para o seu recuo recente é que chegamos ao limite de potência prática para o resfriamento dos microprocessadores comuns.

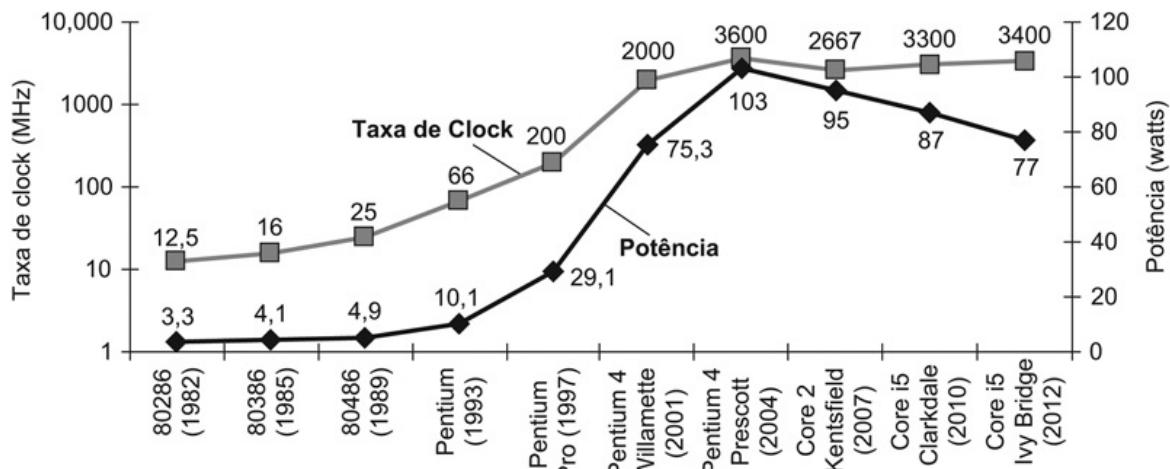


FIGURA 1.16 Taxa de clock e potência para microprocessadores Intel x86 durante oito gerações e 25 anos.

O Pentium 4 fez um salto drástico na taxa de clock e potência, porém menor em desempenho. Os problemas térmicos do Prescott levaram ao abandono da linha Pentium 4. A linha Core 2 retorna a uma pipeline mais simples, com menores taxas de clock e múltiplos processadores por chip. Os pipelines do Core i5 seguem suas pegadas.

Embora a potência ofereça um limite para o que podemos resfriar, na era pós-PC, o recurso realmente crítico é a energia. A vida da bateria pode superar o desempenho no dispositivo móvel pessoal, e os arquitetos de computadores em escala de warehouse tentam reduzir os custos da alimentação e resfriamento de 100.000 servidores, pois os custos são muito altos nessa escala. Assim como a medição do tempo em segundos é uma medida mais segura do desempenho do programa do que uma taxa como MIPS (Seção 1.10), a métrica de energia em

joules é uma medida melhor do que uma potência como watts, que é simplesmente joules/segundo.

A tecnologia dominante para circuitos integrados é denominada *Complementary Metal Oxide Semiconductor* (CMOS). Para CMOS, a principal fonte de dissipação de potência é a chamada potência dinâmica — ou seja, a potência que é consumida quando os transistores mudam do estado 0 para 1 e vice-versa. A energia dinâmica depende da carga capacitiva de cada transistor, da tensão elétrica aplicada:

$$Energia \propto Carga\,capacitiva \times Tensão^2$$

Esta equação é a energia de um pulso durante a transição lógica de 0 → 1 → 0 ou 1 → 0 → 1. A energia de uma única transição é, então,

$$Energia \propto 1/2 \times Carga\,capacitiva \times Tensão^2$$

A potência exigida por transistor é simplesmente o produto da energia de uma transição e a frequência das transições:

$$Potência \propto 1/2 \times Carga\,capacitiva \times Tensão^2 \times Frequência\,comutada$$

A frequência comutada é uma função da taxa de clock. A carga capacitiva por transistor é uma função do número de transistores conectados a uma saída (chamado de *fanout*) e da tecnologia, que determina a capacidade dos fios e dos transistores.

Com relação à [Figura 1.16](#), como as taxas de clock poderiam crescer por um fator de 1.000 enquanto a potência crescia por um fator apenas de 30? A potência pode ser diminuída reduzindo-se a tensão elétrica, o que ocorreu a cada nova geração da tecnologia, e a potência é uma função da tensão elétrica ao quadrado. Normalmente, a tensão elétrica foi reduzida em 15% por geração. Em 20 anos, as tensões passaram de 5V para 1V, motivo pelo qual o aumento na potência é de apenas 30 vezes.

Potência relativa

Exemplo

Suponha que tenhamos desenvolvido um novo processador, mais simples, que tem 85% da carga capacitiva do processador mais antigo e mais complexo. Além do mais, considere que ele tenha tensão ajustável, de modo que pode reduzir a tensão em 15% em comparação com o processador B, o que resulta em um encolhimento de 15% na frequência. Qual é o impacto sobre a potência dinâmica?

Resposta

$$\frac{\text{Potência}_{\text{nova}}}{\text{Potência}_{\text{antiga}}} = \frac{\langle \text{Carga capacitiva} \times 0,85 \rangle \times \langle \text{Tensão} \times 0,85 \rangle^2 \times \langle \text{Frequência comutada} \times 0,85 \rangle}{\text{Carga capacitiva} \times \text{Tensão}^2 \times \text{Frequência comutada}}$$

Assim, a razão de potência é

$$0,85^4 = 0,52$$

Logo, o novo processador usa cerca de metade da potência do processador antigo.

O problema hoje é que reduzir ainda mais a tensão parece causar muito vazamento nos transistores, como torneiras de água que não conseguem ser completamente fechadas. Até mesmo hoje, cerca de 40% do consumo de potência nos chips de servidor é decorrente de vazamentos. Se os transistores começassem a vazar mais, o processo inteiro poderia se tornar incontrolável.

Para tentar resolver o problema de potência, os projetistas já conectaram grandes dispositivos a fim de aumentar o resfriamento e depois desligaram partes do chip que não são usadas em determinado ciclo de clock. Embora existam muitas maneiras mais dispendiosas de resfriar os chips e, portanto, aumentar a potência para, digamos, 300 watts, essas técnicas são muito caras para computadores de desktop e até mesmo servidores, sem falar nos dispositivos móveis pessoais.

Como os projetistas de computador bateram contra a barreira da potência, eles precisaram de uma nova maneira de prosseguir e escolheram um caminho diferente do modo como projetavam microprocessadores nos primeiros 30 anos.

Detalhamento

Embora a energia dinâmica seja a principal fonte de dissipação de potência na CMOS, a dissipação de potência estática ocorre devido à corrente de vazamento que flui mesmo quando um transistor está desligado. Nos servidores, o vazamento normalmente é responsável por 40% do consumo de potência. Assim, aumentar o número de transistores aumenta a dissipação de potência, mesmo que os transistores estejam sempre desligados. Diversas técnicas de projeto e inovações de tecnologia estão sendo implantadas para controlar o vazamento, mas é difícil reduzir mais a tensão.

Detalhamento

A potência é um desafio para os circuitos integrados por dois motivos. Primeiro, ela precisa ser trazida e distribuída em torno do chip; os microprocessadores modernos utilizam centenas de pinos somente para potência e aterramento! De modo semelhante, diversos níveis de interconexão no chip são usados unicamente para distribuição de potência e aterramento às partes do chip. Segundo, a potência é dissipada como calor, e precisa ser removida. Os chips de servidor podem queimar mais de 100 watts, e o resfriamento do chip e do sistema ao seu redor é um custo importante nos Computadores em Escala de Warehouse (Capítulo 6).

1.8. Mudança de mares: Passando de processadores para multiprocessadores

Até agora, a maioria dos softwares têm sido como música escrita para um solista; com a geração atual de chips, estamos adquirindo alguma experiência com duetos e quartetos e outros pequenos grupos; mas compor um trabalho para grande orquestra e coro é um tipo de desafio diferente.

Brian Hayes, Computing in a Parallel Universe, 2007.

O limite de potência forçou uma mudança drástica no projeto dos microprocessadores. A Figura 1.17 mostra a melhoria no tempo de resposta dos programas para microprocessadores de desktop ao longo dos anos. Desde 2002, a taxa se reduziu de um fator de 1,5 por ano para um fator de menos de 1,2 por ano.

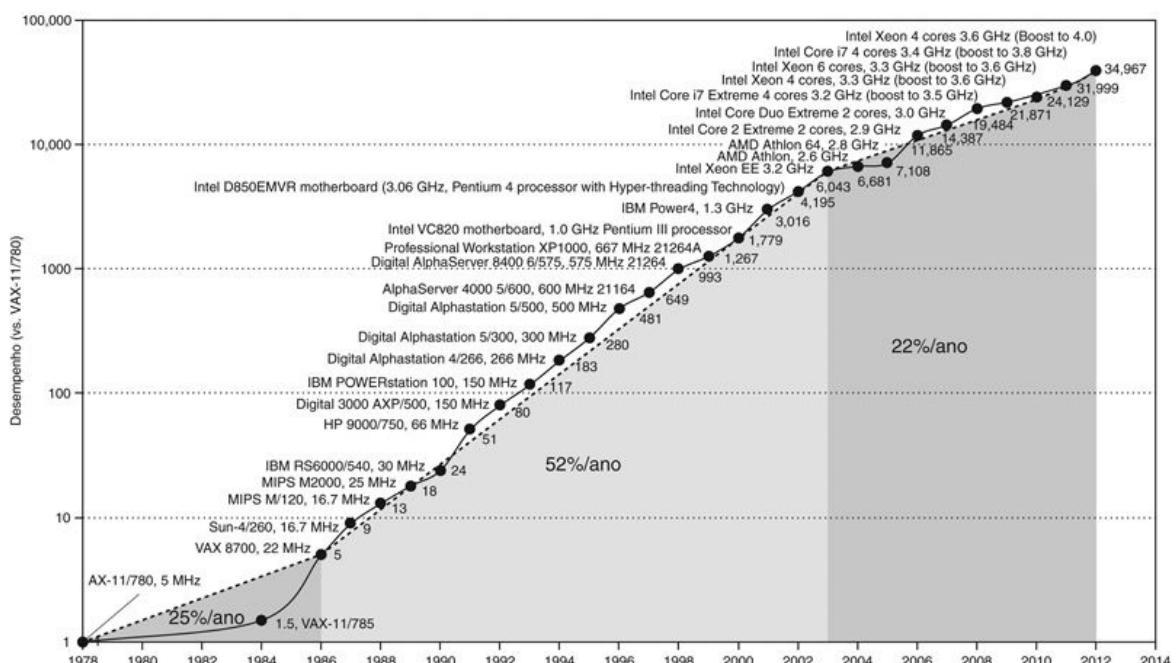


FIGURA 1.17 Crescimento do desempenho do processador desde meados da década de 1980.

Este gráfico representa o desempenho relativo ao VAX 11/780

medido pelos benchmarks SPECint (Seção 1.10). Antes de meados da década de 1980, o crescimento do desempenho do processador foi em grande parte controlado pela tecnologia e teve uma média de 25% por ano. O aumento no crescimento para cerca de 52% desde então é atribuído a ideias arquiteturais e organizacionais mais avançadas. A melhoria de desempenho anual mais alta, de 52% desde meados da década de 1980, significou que o desempenho aumentou por um fator de cerca de sete mais alto em 2002 do que teria sido se permanecesse em 25%. Desde 2002, os limites de potência, o paralelismo disponível em nível de instrução e a latência de memória longa reduziram o desempenho de processadores únicos recentemente, para cerca de 22% por ano.

Em vez de continuar diminuindo o tempo de resposta de um único programa executando em um único processador, em 2006 todas as empresas de desktop e servidor estavam usando microprocessadores com múltiplos processadores por chip, em que o benefício normalmente está mais no throughput do que no tempo de resposta. Para reduzir a confusão entre as palavras processador e microprocessador, as empresas se referem aos processadores como “cores” (ou núcleos), e esses microprocessadores são chamados genericamente de microprocessadores “multicore” (ou múltiplos núcleos). Logo, um microprocessador “quadcore” é um chip que contém quatro processadores, ou quatro núcleos.

No passado, os programadores podiam contar com inovações no hardware, na arquitetura e nos compiladores para dobrar o desempenho de seus programas a cada 18 meses sem ter de mudar uma linha de código. Hoje, para os programadores obterem uma melhoria significativa no tempo de resposta, eles precisam reescrever seus programas de modo que tirem proveito de múltiplos processadores. Além do mais, para obter o benefício histórico de rodar mais rapidamente nos microprocessadores mais novos, os programadores terão de continuar a melhorar o desempenho de seu código à medida que dobra o número de núcleos.

Para reforçar como os sistemas de software e hardware trabalham lado a lado, usamos uma seção especial, *Interface hardware/software*, no livro inteiro, com a primeira aparecendo logo a seguir. Essas seções resumem ideias importantes nessa interface crítica.



PARALELISMO

Interface de hardware/software

O **paralelismo** sempre foi fundamental para o desempenho na computação, mas normalmente esteve oculto. O Capítulo 4 explicará sobre o **pipelining**, uma técnica elegante que roda programas mais rapidamente sobrepondo a execução de instruções. Este é um exemplo de *paralelismo em nível de instrução*, em que a natureza paralela do hardware é retirada de modo que o programador e o compilador possam pensar no hardware como executando instruções sequencialmente.

Forçar os programadores a estarem cientes do hardware paralelo e reescrever explicitamente seus programas para serem paralelos foi a “terceira trilha” da arquitetura de computadores, pois empresas no passado, que dependiam dessa mudança no comportamento, fracassaram. Do ponto de vista histórico, é surpreendente que a indústria inteira de TI tenha apostado seu futuro nos programadores finalmente passarem com sucesso para a programação explicitamente paralela.



PIPELINING

Por que tem sido tão difícil para os programadores escreverem programas explicitamente paralelos? O primeiro motivo é que a programação paralela é, por definição, programação de desempenho, o que aumenta a dificuldade da programação. Não apenas o programa precisa estar correto, solucionar um problema importante e oferecer uma interface útil às pessoas ou outros programas que o chamam, mas ele também precisa ser rápido. Caso contrário, se você não precisasse de desempenho, bastaria escrever um programa sequencial.

O segundo motivo é que rapidez, para o hardware paralelo, significa que o programador precisa dividir uma aplicação, de modo que cada processador tenha, aproximadamente, a mesma quantidade de coisas a fazer ao mesmo tempo, e que o overhead do escalonamento e coordenação não afasta os benefícios de desempenho em potencial do paralelismo.

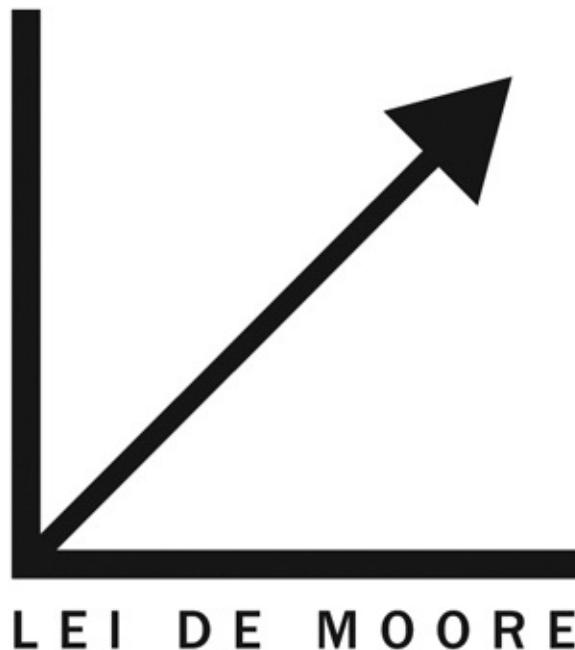
Como uma analogia, suponha que a tarefa fosse escrever um artigo de jornal. Oito repórteres trabalhando no mesmo artigo poderiam potencialmente escrever um artigo oito vezes mais rápido. Para conseguir essa velocidade aumentada, seria preciso desmembrar a tarefa de modo que cada repórter tivesse algo para fazer ao mesmo tempo. Assim, temos de *escalonar* as subtarefas. Se algo saísse errado e apenas um repórter levasse mais tempo do que os sete outros levaram,

então o benefício de ter oito escritores seria diminuído. Assim, temos de *balancear a carga* por igual para obter o ganho de velocidade desejado. Outro perigo seria se os repórteres tivessem de gastar muito tempo falando uns com os outros para escrever suas seções. Você também se atrasaria se uma parte do artigo, como a conclusão, não pudesse ser escrita até que todas as outras partes fossem concluídas. Assim, deve-se ter o cuidado para *reduzir o overhead de comunicação e sincronização*. Para essa analogia e para a programação paralela, os desafios incluem escalonamento, balanceamento de carga, tempo para sincronismo e overhead para comunicação entre as partes. Como você poderia imaginar, o desafio é ainda maior com mais repórteres de um artigo de jornal e mais processadores na programação paralela.

Para refletir essa mudança de mares no setor, os próximos cinco capítulos desta edição do livro possuem uma seção sobre as implicações da revolução paralela relacionadas a cada capítulo:

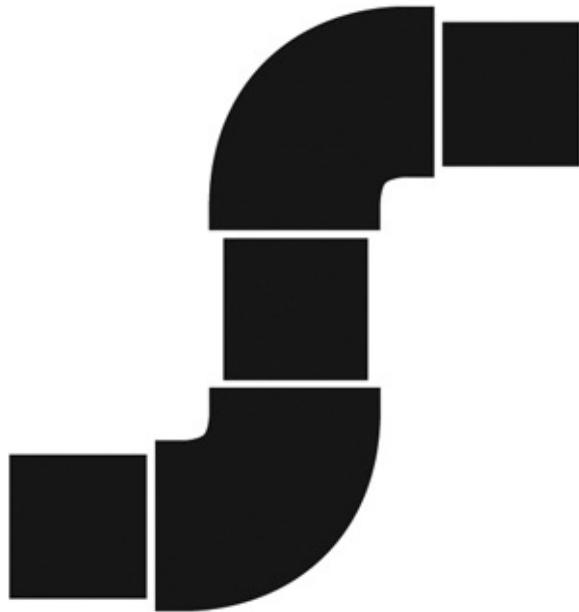
- [Capítulo 2, Seção 2.11: Paralelismo e instruções: sincronização](#).

Normalmente, tarefas paralelas independentes, às vezes, precisam ser coordenadas como, por exemplo, dizer quando elas completaram seu trabalho. Esse capítulo explica as instruções usadas por processadores multicore para sincronizar tarefas.



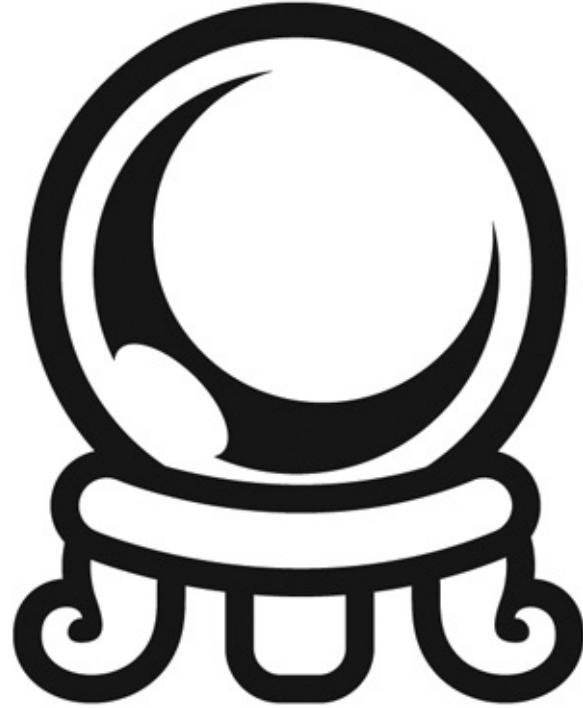
- [Capítulo 3, Seção 3.6: Paralelismo e aritmética de computador: Paralelismo](#)

de subword. Talvez a forma mais simples de se criar paralelismo envolva a computação com elementos em paralelo, como ao multiplicar dois vetores. O paralelismo de subword tira proveito dos recursos fornecidos pela **Lei de Moore** para fornecer unidades aritméticas mais largas, que podem operar sobre muitos, simultaneamente.



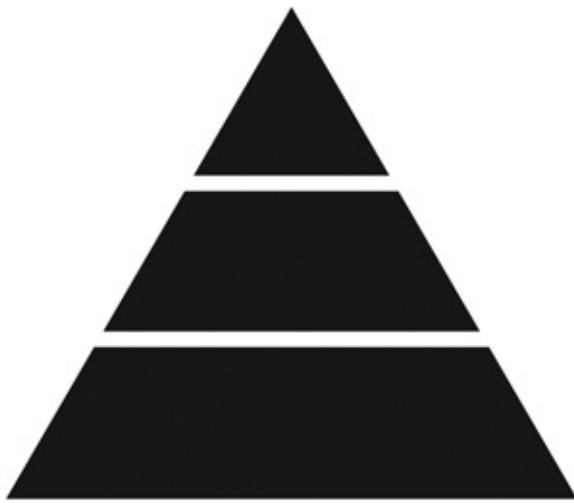
PIPELINING

- *Capítulo 4, Seção 4.10: Paralelismo e paralelismo avançado em nível de instrução.* Dada a dificuldade da programação explicitamente paralela, um esforço tremendo foi investido na década de 1990, para que o hardware e o compilador revelassem o paralelismo implícito, inicialmente por meio do **pipelining**. Esse capítulo descreve algumas dessas técnicas agressivas, incluindo a busca e a execução simultâneas de múltiplas instruções e a estimativa dos resultados das decisões, com a execução especulativa das instruções usando a **predição**.



P R E D I Ç Ã O

- *Capítulo 5, Seção 5.10: Paralelismo e hierarquias de memória: coerência do cache.* Um modo de reduzir o custo da comunicação é fazer com que todos os processadores usem o mesmo espaço de endereço, de modo que qualquer processador possa ler ou gravar quaisquer dados. Visto que todos os processadores atuais utilizam caches para manter uma cópia temporária dos dados na memória mais rápida, mais próxima do processador, é fácil imaginar que a programação paralela seria ainda mais difícil se os caches associados a cada processador tivessem valores inconsistentes dos dados compartilhados. Esse capítulo descreve os mecanismos que mantêm coerentes os dados em todos os caches.



HIERARQUIA

- *Capítulo 5, Seção 5.11: Paralelismo e hierarquias de memória: Redundant Arrays of Inexpensive Disks.* Esta seção descreve como o uso de muitos discos em conjunto pode oferecer um throughput muito maior, que foi a inspiração original dos *Redundant Arrays of Inexpensive Disks* (RAID). A popularidade real do RAID provou ter uma segurança muito maior, incluindo um número modesto de discos redundantes. Esta seção explica as diferenças no desempenho, custo e fidelidade entre os diferentes níveis de RAID.



PARALELISMO

Além dessas seções, existe um capítulo inteiro sobre processamento paralelo. O [Capítulo 6](#) entra em mais detalhes sobre os desafios da programação paralela; apresenta as duas técnicas contrastantes para a comunicação de endereçamento compartilhado e passagem explícita de mensagens; descreve o modelo restrito de paralelismo que é mais fácil de programar; discute a dificuldade do benchmarking de processadores paralelos; apresenta um novo modelo de desempenho simples para microprocessadores multicore e finalmente descreve e avalia quatro exemplos de microprocessadores multicore usando esse modelo.

Como já dissemos, os Capítulos de [3](#) a [6](#) utilizam a multiplicação vetorial de matrizes como um exemplo recorrente, para mostrar como cada tipo de paralelismo pode aumentar o desempenho de forma significativa.

1.9. Vida real: Fabricação e benchmarking do Intel Core i7

Eu acreditava que [os computadores] seriam uma ideia 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 coinventor do Eniac, falando em 1991

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 baseiam os computadores modernos. Nesta primeira “Vida Real”, veremos como os circuitos integrados são fabricados e como o desempenho e a potência são medidos com o Intel Core i7, como exemplo.

Benchmark de CPU SPEC

Um usuário de computador que executa os mesmos programas todos os dias seria o candidato perfeito para avaliar um novo computador. O conjunto de programas executados formaria uma **carga de trabalho**. Para avaliar dois sistemas, um usuário simplesmente compararia o tempo de execução da carga de trabalho nos dois computadores. A maioria dos usuários, porém, não está nesta situação. Em vez disso, eles precisam contar com outros métodos que medem o desempenho de um computador candidato, esperando que os métodos refletem como o computador funcionará com a carga de trabalho do usuário. Essa alternativa normalmente é seguida pela avaliação do computador usando um conjunto de **benchmarks** — programas escolhidos especificamente para medir o desempenho. Os benchmarks formam uma carga de trabalho que o usuário acredita que irá prever o desempenho da carga de trabalho real. Conforme observamos, para criar o **caso comum veloz**, você primeiro precisa saber exatamente qual caso é comum, portanto, os benchmarks desempenham um

papel crítico na arquitetura de computadores.

carga de trabalho

Um conjunto de programas executados em um computador, que é a coleção real das aplicações executadas por um usuário ou construídas a partir de programas reais para aproximar tal mistura. Uma carga de trabalho típica especifica os programas e as frequências relativas.

benchmark

Um programa selecionado para uso na comparação do desempenho de computadores.



CASO COMUM VELOZ

O System Performance Evaluation Cooperative (SPEC) é um esforço com patrocínio e suporte de uma série de fornecedores de computadores, a fim de criar conjuntos padrão de benchmarks para sistemas de computador modernos. Em 1989, o SPEC criou originalmente um conjunto de benchmark focalizando o desempenho do processador (agora chamado SPEC89), que evoluiu por cinco gerações. A mais recente é SPEC CPU2006, que consiste em um conjunto de 12 benchmarks de inteiros (CINT2006) e 17 benchmarks de ponto flutuante (CFP2006). Os benchmarks de inteiros variam desde parte de um compilador C até um programa de xadrez e uma simulação de computador quântico. Os benchmarks de ponto flutuante incluem códigos de grade estruturados para modelagem de elemento finito, códigos de método de partículas para dinâmica

molecular e códigos de álgebra linear esparsa para dinâmica de fluidos.

A [Figura 1.18](#) descreve os benchmarks de inteiros SPEC e seu tempo de execução no Intel Core i7, mostrando os fatores que explicam o tempo de execução: contagem de instruções, CPI e tempo do ciclo de clock. Observe que o CPI varia por um fator de 5.

Descrição	Nome	Contagem de instruções x 10 ⁹	CPI	Tempo do ciclo de clock (seg x 10 ⁻⁹)	Tempo de execução (seg)	Tempo de referência (seg)	SPE-Cratio
Processamento de string interpretado	perl	2252	0,60	0,376	508	9770	19,2
Compactação de classificação em bloco	bzip2	2390	0,70	0,376	629	9650	15,4
Compilador C GNU	gcc	794	1,20	0,376	358	8050	22,5
Otimização combinatória	mcf	221	2,66	0,376	221	9120	41,2
Jogo de Go (IA)	go	1274	1,10	0,376	527	10490	19,9
Pesquisa de sequência genética	hmmer	2616	0,60	0,376	590	9330	15,8
Jogo de xadrez (IA)	sjeng	1948	0,80	0,376	586	12100	20,7
Simulação de computador quântico	libquantum	659	0,44	0,376	109	20720	190,0
Compactação de vídeo	h264avc	3793	0,50	0,376	713	22130	31,0
Biblioteca de simulação de evento discreto	omnetpp	367	2,10	0,376	290	6250	21,5
Jogos/descoberta de caminho	astar	1250	1,00	0,376	470	7020	14,9
Análise XML	xalancbmk	1045	0,70	0,376	275	6900	25,1
Média geométrica	—	—	—	—	—	—	25,7

FIGURA 1.18 Benchmarks SPECINTC2006 executando no Intel Core i7 920 de 2,66 GHz.

Conforme explica a equação na seção “A equação clássica de desempenho da CPU”, anteriormente neste capítulo, o tempo de execução é o produto dos três fatores nesta tabela: contagem de instruções em bilhões, clocks por instrução (CPI) e tempo do ciclo de clock em nanossegundos. SPECratio é simplesmente o tempo de referência, que é fornecido pelo SPEC, dividido pelo tempo de execução medido. O único número mencionado como SPECINTC2006 é a média geométrica dos SPECratios.

Para simplificar o marketing dos computadores, o SPEC decidiu informar um único número para resumir todos os 12 benchmarks de inteiros. As medidas do tempo de execução são primeiro normalizadas dividindo-se o tempo de execução em um processador de referência pelo tempo de execução no computador mensurado; essa normalização gera uma medida, chamada *SPECratio*, que tem a vantagem de usar resultados numéricos maiores para indicar desempenho melhor. Ou seja, o SPECratio é o inverso do tempo de execução. Uma medição de resumo CINT2006 ou CFP2006 é obtida usando-se a média geométrica dos

SPECratios.

Detalhamento

Ao comparar dois computadores usando SPECratios, use a média geométrica, de modo que ela informe a mesma resposta relativa, não importando o computador utilizado para normalizar os resultados. Se calculássemos a média dos valores de tempo de execução normalizados com uma média aritmética, os resultados variariam dependendo do computador que escolhêssemos como referência.

A fórmula para a média geométrica é

$$\sqrt[n]{\prod_{i=1}^n \text{Razão do tempo de execução}_i}$$

em que Razão do tempo de execução_i é o tempo de execução, normalizado ao computador de referência, para o iº programa de um total de n na carga de trabalho, e

$$\prod_{i=1}^n a_i \text{ significa o produto } a_1 \times a_2 \times \dots \times a_n$$

Benchmark de potência SPEC

Dada a importância cada vez maior do consumo de energia e potência, o SPEC acrescentou um benchmark para medir a potência. Ele informa o consumo de potência dos servidores em diferentes níveis de carga de trabalho, dividido em incrementos de 10%, por um período de tempo. A [Figura 1.19](#) mostra os resultados para um servidor usando processadores Intel Nehalem, semelhantes aos anteriores.

Carga de destino %	Desempenho (ssj_ops)	Potência média (Watts)
100%	865.618	258
90%	786.688	242
80%	698.051	224
70%	607.826	204
60%	521.391	185
50%	436.757	170
40%	345.919	157
30%	262.071	146
20%	176.061	135
10%	86.784	121
0%	0	80
Soma geral	4.787.166	1922
$\sum \text{ssj_ops} / \sum \text{potência} =$		2490

FIGURA 1.19 SPECpower_ssj2008 executando no Intel Xeon X5620 a 2,66 GHz e soquete dual com 16 GB de DRAM DDR2-667 e um disco SSD de 100 GB.

SPECpower começou com o benchmark SPEC para aplicações comerciais em Java (SPECJBB2005), que exercita processadores, caches e memória principal, além da máquina virtual Java, compilador, coletor de lixo e partes do sistema operacional. O desempenho é medido em throughput e as unidades são operações de negócios por segundo. Mais uma vez, para simplificar o marketing dos computadores, o SPEC resume esses números em um único número, chamado “ssj_ops geral por Watt”. A fórmula para essa única métrica de resumo é

$$\text{ssj_ops geral por Watt} = \left(\sum_{i=0}^{10} \text{ssj_ops}_i \right) / \left(\sum_{i=0}^{10} \text{potência}_i \right)$$

em que ssj_ops_i é o desempenho em cada incremento de 10% e potência_i é a potência consumida em cada nível de desempenho.

1.10. Falácia e armadilhas

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

Sir Karl Popper, The Philosophy of Science, 1957

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 contraexemplo. Também discutiremos *armadilhas* ou erros facilmente cometidos. Em geral, as armadilhas são generalizações 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 e armadilhas de custo/desempenho têm confundido muitos arquitetos de computador, incluindo nós. Consequentemente, esta seção não poupa exemplos relevantes. Vamos começar com uma armadilha que engana muitos projetistas e revela um relacionamento importante no projeto de computadores.

Armadilha: Esperar que a melhoria de um aspecto de um computador aumente o desempenho geral por uma quantidade proporcional ao tamanho da melhoria.

A grande ideia de tornar o **caso comum veloz** tem um corolário desmoralizante que tem assolado os projetistas de hardware e de software. Ele nos lembra que a oportunidade de melhoria é afetada pelo tempo que o evento consome.



CASO COMUM VELOZ

Um problema simples de projeto ilustra isso muito bem. Suponha que um programa execute em 100 segundos em um computador, com operações de multiplicação responsáveis por 80 segundos desse tempo. Quanto terei de melhorar a velocidade da multiplicação se eu quiser que meu programa execute cinco vezes mais rápido?

O tempo de execução do programa depois de fazer a melhoria é dado pela seguinte equação simples, conhecida como **lei de Amdahl**:

$$= \frac{\text{Tempo de execução afetado pelo aprimoramento}}{\text{Quantidade de aprimoramento}} + \text{Tempo de execução não afetado}$$

lei de Amdahl

Uma regra indicando que a melhoria de desempenho possível com determinado aprimoramento é limitada pela quantidade de utilização do recurso aprimorado. Essa é uma versão quantitativa da lei dos retornos decrescentes.

Para este problema:

$$\text{Tempo de execução após o aprimoramento} = \frac{80 \text{ seg}}{n} + (100 - 80 \text{ segundos})$$

Como queremos que o desempenho seja cinco vezes mais rápido, o novo tempo de execução deverá ser 20 segundos, gerando

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

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

Ou seja, *não existe quantidade* pela qual podemos melhorar a multiplicação para conseguir um aumento quíntuplo no desempenho, se a multiplicação é responsável por apenas 80% da carga de trabalho. A melhoria de desempenho possível com determinado aprimoramento é limitada pela quantidade de utilização do recurso aprimorado. Esse conceito também gera o que chamamos de lei dos retornos decrescentes na vida diária.

Podemos usar a lei de Amdahl para estimar os aprimoramentos no desempenho quando sabemos o tempo consumido para alguma função e seu ganho de velocidade em potencial. A lei de Amdahl, junto com a equação de desempenho da CPU, é uma ferramenta prática para avaliar melhorias em potencial. A lei de Amdahl é explorada com mais detalhes nos exercícios.

A lei de Amdahl também é usada para se demonstrar limites práticos do número de processadores paralelos. Examinamos esse argumento na seção de Falácia e Armadilhas do [Capítulo 6](#).

Falácia: Os computadores com pouca utilização demandam menos potência.

A eficiência de potência importa em baixas utilizações, pois as cargas de trabalho do servidor variam. A utilização de servidores no computador em escala de warehouse no Google, por exemplo, está entre 10% e 50% na maior parte do tempo e em 100% em menos de 1% do tempo. Mesmo com cinco anos para aprender a executar bem o benchmark SPECpower, o computador configurado especialmente para isso, com os melhores resultados em 2012, ainda usava 33%

da potência de pico a 10% da carga. Os sistemas em campo que não estão configurados para o benchmark SPECpower certamente são piores.

Como as cargas de trabalho dos servidores variam, mas utilizam uma grande fração da potência máxima, Barroso et al. (2007) argumenta que deveríamos reprojetar o hardware para alcançar a “computação proporcional à energia”. Se os servidores futuros usassem, digamos, 10% da potência máxima a 10% de carga de trabalho, poderíamos reduzir a conta de eletricidade dos centros de dados e nos tornarmos bons cidadãos corporativos em uma era de preocupação crescente com as emissões de CO₂.

Falácia: Projetar para o desempenho e projetar para eficiência de energia são objetivos não relacionados.

Como a energia é potência com o passar do tempo, normalmente acontece que as otimizações de hardware ou software que levam menos tempo economizam energia em geral, mesmo que a otimização gaste um pouco mais de energia quando é usada. Um motivo é que todo o restante do computador está consumindo energia enquanto o programa está em execução, mesmo que a parte otimizada use um pouco mais de energia, o tempo reduzido poderá economizar energia do sistema inteiro.

Armadilha: Usar um subconjunto da equação de desempenho como uma métrica de desempenho.

Já advertimos sobre o perigo de prever o desempenho com base simplesmente na taxa de clock, ou na contagem de instruções ou no CPI. Outro erro comum é usar apenas dois dos três fatores para comparar o desempenho. Embora o uso de dois dos três fatores possa ser válido em um contexto limitado, o conceito facilmente também é mal utilizado. Sem dúvida, quase todas as alternativas propostas para o uso do tempo como métrica de desempenho por fim levaram a afirmações enganosas, resultados distorcidos ou interpretações incorretas.

Uma alternativa ao tempo é o **MIPS (milhões de instruções por segundo)**. Para determinado programa, o MIPS é simplesmente

$$\text{MIPS} = \frac{\text{Contagem de instruções}}{\text{Tempo de execução} \times 10^6}$$

milhões de instruções por segundo (MIPS)

Uma medida da velocidade de execução do programa baseada no número de milhões de instruções. MIPS é calculado como a contagem de instruções dividida pelo produto do tempo de execução e 10^6 .

Como MIPS é uma taxa de execução de instruções, MIPS especifica o desempenho inversamente ao tempo de execução; computadores mais rápidos possuem uma taxa de MIPS mais alta. A boa notícia sobre MIPS é que ele é fácil de entender e computadores mais rápidos significam um MIPS maior, que corresponde à intuição.

Existem três problemas com o uso do MIPS como uma medida para comparar computadores. Primeiro, 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 MIPS, pois as contagens de instruções certamente serão diferentes. Segundo, MIPS varia entre os programas no mesmo computador; assim, um computador não pode ter uma única avaliação MIPS. Por exemplo, substituindo o tempo de execução, vemos o relacionamento entre MIPS, taxa de clock e CPI:

$$\text{MIPS} = \frac{\text{Contagem de instruções}}{\frac{\text{Contagem de instruções} \times \text{CPI}}{\text{Taxa de clock}} \times 10^6} = \frac{\text{Taxa de clock}}{\text{CPI} \times 10^6}$$

O CPI variou em 5 para SPEC CPU2006 em um computador Intel Core i7 na [Figura 1.18](#), de modo que o MIPS também varia. Finalmente, e mais importante, se um novo programa executa mais instruções, e uma é mais rápida que a outra, o MIPS pode variar independentemente do desempenho!

Verifique você mesmo

Considere as seguintes medidas de desempenho para um programa:

Medida	Computador A	Computador B
Número de instruções	10 bilhões	8 bilhões
Taxa de clock	4 GHz	4 GHz
CPI	1,0	1,1

- a. Que computador tem a avaliação MIPS mais alta?
- b. Qual computador é mais rápido?

1.11. Comentários finais

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

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 várias questões.

Os projetistas de hardware e de software constroem sistemas computacionais em camadas hierárquicas; 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 abstração. 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. No lado negativo, a arquitetura pode impedir a introdução de inovações que exijam a mudança da interface.



A B S T R A Ç Ã O

Existe um método confiável para determinar e informar o desempenho usando o tempo de execução dos programas reais como métrica. Esse tempo de execução está relacionado a outras medições importantes que podemos fazer pela seguinte equação:

$$\frac{\text{Segundos}}{\text{Programa}} = \frac{\text{Instruções}}{\text{Programa}} \times \frac{\text{Ciclos de clock}}{\text{Instrução}} \times \frac{\text{Segundos}}{\text{Ciclo de clock}}$$

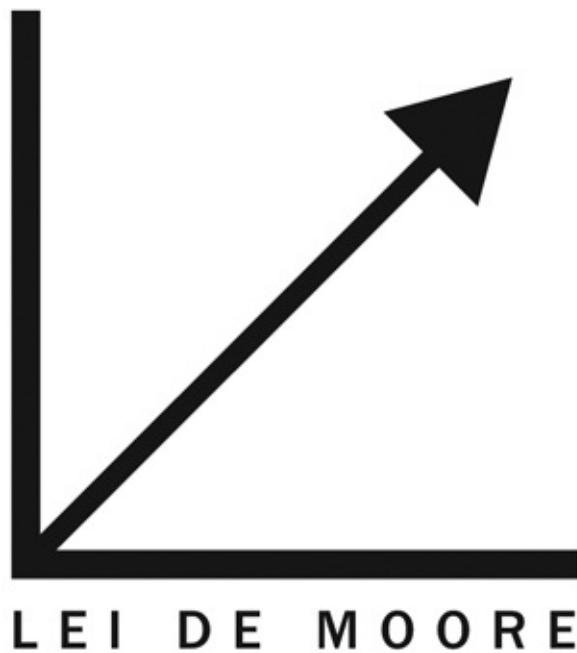
Usaremos essa equação e seus fatores constituintes muitas vezes. Lembre-se, porém, de que individualmente os fatores não determinam o desempenho: somente o produto, que é igual ao tempo de execução, é uma medida confiável do desempenho.

Colocando em perspectiva

O tempo de execução é a única medida válida e incontestável do desempenho. Muitas outras métricas foram propostas e desapareceram. Às vezes, essas

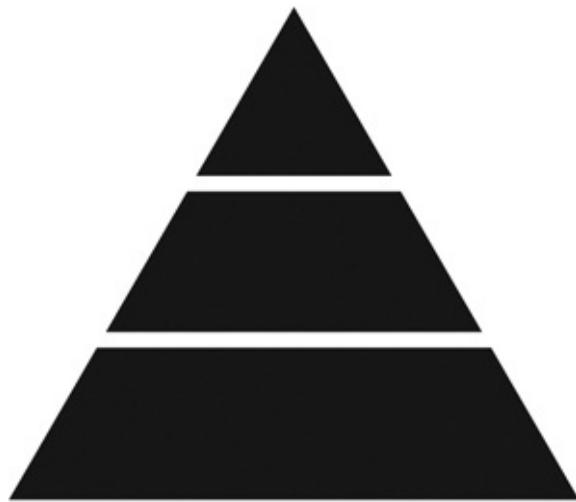
métricas possuem falhas desde o início, não refletindo o tempo de execução; outras vezes, uma métrica que é válida em um contexto limitado é estendida e usada além desse contexto ou sem o esclarecimento adicional necessário para torná-la válida.

A tecnologia de hardware vital para os processadores modernos é o silício. De igual importância para uma compreensão da tecnologia de circuito integrado é o conhecimento das taxas de mudança tecnológica esperadas, conforme previsto pela **Lei de Moore**. Enquanto o silício impulsiona o rápido avanço do hardware, novas ideias na organização dos computadores melhoraram seu custo/desempenho. Duas das principais ideias são a exploração do paralelismo no programa, normalmente por meio de processadores múltiplos, e a exploração da localidade dos acessos a uma **hierarquia de memória**, em geral por meio de caches.



A eficiência no uso de energia substituiu a área do die como o recurso mais crítico do projeto de microprocessadores. Conservar energia enquanto se tenta aumentar o desempenho tem迫使 o setor de hardware a passar para microprocessadores multicore, forçando, assim, o setor de software a passar para a programação do hardware em paralelo. Agora é preciso que haja **paralelismo**

para melhorar o desempenho.



HIERARQUIA

Os projetos de computadores sempre foram medidos pelo custo e desempenho, além de outros fatores importantes, como energia, confiabilidade, custo de proprietário e escalabilidade (ou facilidade de expansão). Embora este capítulo tenha focalizado o custo, o desempenho e a energia, os melhores projetos buscarão o equilíbrio apropriado para determinado mercado entre todos esses fatores.



PARALELISMO

Mapa para este livro

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

- *Caminho de dados*: [Capítulos 3, 4 e 6](#)
- *Controle*: [Capítulos 4 e 6](#)
- *Memória*: [Capítulo 5](#)
- *Entrada*: [Capítulos 5 e 6](#)
- *Saída*: [Capítulos 5 e 6](#)

Como dissemos, o [Capítulo 4](#) descreve como os processadores exploram o paralelismo implícito; e o [Capítulo 6](#) descreve os microprocessadores multicore explicitamente paralelos, que estão no núcleo da revolução paralela. O [Capítulo 5](#) descreve como a hierarquia de memória explora a localidade. 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 Apêndice A oferece uma referência para o conjunto de instruções do [Capítulo 2](#). O [Capítulo 3](#) descreve como os computadores tratam dos dados aritméticos. O Apêndice B apresenta o projeto lógico.

1.12. 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, <1.4> significa que você deve ler a [Seção 1.4](#), “Sob as tampas”, para ajudar a resolver esse exercício.

- 1.1** [2] <§1.1> Sem considerar os smartphones usados por um bilhão de pessoas, liste e descreva quatro outros tipos de computadores.
- 1.2** [5] <§1.2> As oito grandes ideias na arquitetura do computador são semelhantes às ideias de outras áreas. Faça a correspondência das oito ideias da arquitetura de computadores, “Projete pensando na Lei de Moore”, “Use a abstração para simplificar o projeto”, “Torne o caso comum veloz”, “Desempenho pelo paralelismo”, “Desempenho pelo pipelining”, “Desempenho pela predição”, “Hierarquia de memórias”, “Estabilidade pela redundância”, com as seguintes ideias de outras áreas:
- a.** Linhas de montagem na fabricação de automóveis
 - b.** Cabos de pontes suspensas
 - c.** Sistemas de navegação aérea e marinha que incorporam informações de vento
 - d.** Elevadores expressos nos prédios
 - e.** Central de reserva de biblioteca
 - f.** Aumento da área de porta em um transistor CMOS para diminuir seu tempo de comutação
 - g.** Acréscimo de catapultas eletromagnéticas de aeronaves (que são alimentadas eletricamente, ao contrário dos atuais modelos alimentados por vapor), possibilitadas pela maior geração de potência oferecida pela nova tecnologia de reator
 - h.** Montagem de carros com piloto automático, cujos sistemas de controle contam parcialmente com sistemas sensores existentes, já instalados no veículo base, como sistemas de afastamento da pista e sistemas inteligentes de controle de navegação
- 1.3** [2] <§1.3> Descreva as etapas que transformam um programa escrito em linguagem de alto nível, como C, em uma representação que é executada diretamente por um processador de computador.

1.4 [2] <§1.4> Imagine uma tela colorida usando 8 bits para cada uma das cores primárias (vermelho, verde, azul) por pixel e com uma resolução de 1280×1024 pixels.

- a. Qual deve ser o tamanho (em bytes) do buffer de frame a fim de armazenar um frame?
- b. Quanto tempo levará, no mínimo, para que o frame seja enviado por uma rede de 100 Mbits/s?

1.5 [4] <§1.6> Considere três processadores diferentes, P1, P2 e P3, executando o mesmo conjunto de instruções. P1 tem uma taxa de clock de 3 GHz e um CPI de 1,5. P2 tem uma taxa de clock de 2,5 GHz e um CPI de 1,0. P3 tem uma taxa de clock de 4,0 GHz e um CPI de 2,2.

- a. Qual processador possui o desempenho mais rápido expressado pelas instruções por segundo?
- b. Se cada processador executa um programa em 10 segundos, encontre o número de ciclos e o número de instruções.
- c. Ao tentar reduzir o tempo de execução em 30%, a CPI aumenta em 20%. Qual a taxa de clock que deve ser utilizada para a redução de tempo?

1.6 [20] <§1.6> Considere duas implementações diferentes da mesma arquitetura do conjunto de instruções. Existem quatro classes de instruções, de acordo com seu CPI (classes A, B, C e D). P1 com uma taxa de clock de 2,5 GHz e CPIs de 1, 2, 3 e 3, e P2 com uma taxa de clock de 3 GHz e CPIs de 2, 2, 2 e 2.

Dado um programa com uma contagem de instruções dinâmicas de $1,06E6$ divididas em classes das seguintes formas: 10% classe A, 20% classe B, 50% classe C e 20% classe D, qual implementação é mais rápida?

- a. Qual é o CPI global para cada implementação?
- b. Encontre os ciclos de clock exigidos nos dois casos.

1.7 [15] <§1.6> Os compiladores podem ter um impacto profundo sobre o desempenho de uma aplicação. Suponha que, para um programa, o compilador A resulte em uma contagem de instruções dinâmicas de $1,0E9$ e tenha um tempo de execução de 1,1 s, enquanto o compilador B resulte em uma contagem de instruções dinâmicas de $1,2E9$ e um tempo de execução de 1,5 s.

- a. Ache o CPI médio para cada programa dado que o processador possui um tempo de ciclo de clock de 1 ns.
- b. Suponha que os programas compilados sejam executados em dois processadores diferentes. Se os tempos de execução nos dois

processadores são iguais, o quanto mais rápido é o clock do processador que executa o código do compilador A em relação ao clock do processador que executa o código do compilador B?

- c. Um novo compilador é desenvolvido para usar apenas 6,0E8 instruções e possui um CPI médio de 1,1. Qual é o ganho de velocidade com o uso deste novo compilador, em vez de usar o compilador A ou B no processador original?

1.8 O processador Pentium 4 Prescott, lançado em 2004, tinha uma taxa de clock de 3,6 GHz e tensão de 1,25 V. Suponha que, na média, ele consumisse 10 W de potência estática e 90 W de potência dinâmica.

O Core i5 Ivy Bridge, lançado em 2012, tinha uma taxa de clock de 3,4 GHz e tensão de 0,9 V. Suponha que, na média, ele consumisse 30 W de potência estática e 40 W de potência dinâmica.

1.8.1 [5] <§1.7> Para cada processador, ache as cargas capacitivas médias.

1.8.2 [5] <§1.7> Ache a porcentagem da potência dissipada total composta pela potência estática e a razão entre a potência estática e a potência dinâmica de cada tecnologia.

1.8.3 [15] <§1.7> Se a potência total dissipada for reduzida em 10%, quanto a tensão deve ser reduzida para que a corrente de vazamento continue igual?

Nota: a potência é definida como o produto entre tensão e corrente.

1.9 Suponha que, para instruções aritméticas, load/store e desvio, um processador tenha CPIs de 1, 12 e 5, respectivamente. Suponha também que, em um único processador, um programa exija a execução de 2,56E9 instruções aritméticas, 1,28E9 instruções load/store, e 256 milhões de instruções de desvio. Suponha que cada processador tenha uma frequência de clock de 2 GHz.

Suponha que, quando o programa tem execução paralela por vários núcleos, o número de instruções aritméticas e de load/store por processador seja dividido por $0,7 \times p$ (onde p é o número de processadores), mas o número de instruções de desvio por processador permaneça igual.

1.9.1 [5] <§1.7> Ache o tempo de execução total para esse programa em 1, 2, 4 e 8 processadores, e mostre o ganho de velocidade relativo do resultado com 2, 4 e 8 processadores, em relação ao resultado com único processador.

1.9.2 [10] >§§1.6, 1.8> Se o CPI das instruções aritméticas fosse dobrado, qual seria o impacto sobre o tempo de execução do programa em 1, 2, 4 ou 8 processadores?

1.9.3 [10] <§§1.6, 1.8> Para quanto o CPI das instruções de load/store deveria

ser reduzido, de modo que um único processador corresponda ao desempenho de quatro processadores usando os valores de CPI originais?

1.10 Suponha que um wafer com 15 cm de diâmetro tenha um custo de 12, contenha 84 dies e tenha 0,020 defeitos/cm². Suponha que um wafer com 20 cm de diâmetro tenha um custo de 15, contenha 100 dies e tenha 0,031 defeitos/cm².

1.10.1 [10] <§1.5> Ache o aproveitamento para os dois wafers.

1.10.2 [5] <§1.5> Ache o custo por die para os dois wafers.

1.10.3 [5] <§1.5> Se o número de dies por wafer for aumentado em 10% e os defeitos por unidade de área aumentarem em 15%, ache a área do die e o aproveitamento.

1.10.4 [5] <§1.5> Suponha que um processo de fabricação melhore o aproveitamento de 0,92 para 0,95. Ache os defeitos por unidade de área para cada versão da tecnologia dada uma área de die de 200 mm².

1.11 Os resultados do benchmark bzip2 do SPEC CPU2006 executando em um AMD Barcelona tem uma contagem de instruções de 2,389E2, um tempo de execução de 750 s e um tempo de referência de 9650 s.

1.11.1 [5] <§§1.6, 1.9> Ache o CPI se o tempo de ciclo de clock for 0,333 ns.

1.11.2 [5] <§1.9> Ache o SPECratio.

1.11.3 [5] <§§1.6, 1.9> Ache o aumento no tempo de CPU se o número de instruções do benchmark for aumentado em 10% sem afetar o CPI.

1.11.4 [5] <§§1.6, 1.9> Ache o aumento no tempo de CPU se o número de instruções do benchmark for aumentado em 10% e o CPI for aumentado em 5%.

1.11.5 [5] <§§1.6, 1.9> Ache a mudança no SPECratio para essa mudança.

1.11.6 [10] <§1.6> Suponha que estejamos desenvolvendo uma nova versão do processador AMD Barcelona com uma taxa de clock de 4 GHz. Acrescentamos algumas instruções adicionais ao conjunto de instruções, de modo que o número de instruções foi reduzido em 15%. O tempo de execução é reduzido para 700 s e o novo SPECratio é 13,7. Ache o novo CPI.

1.11.7 [10] <§1.6> Esse valor de CPI é maior do que o obtido em 1.11.1, pois a taxa de clock foi aumentada de 3 GHz para 4 GHz. Determine se o aumento no CPI é semelhante ao da taxa de clock. Se forem diferentes, explique o motivo.

1.11.8 [5] <§1.6> Em quanto o tempo de CPU foi reduzido?

1.11.9 [10] <§1.6> Para um segundo benchmark, libquantum, considere um

tempo de execução de 960 ns, CPI de 1,61 e taxa de clock de 3 GHz. Se o tempo de execução for reduzido por outros 10% sem afetar o CPI e com uma taxa de clock de 4 GHz, determine o número de instruções.

1.11.10 [10] <§1.6> Determine a taxa de clock exigida para dar outra redução de 10% no tempo de CPU enquanto o número de instruções é mantido, e com o CPI inalterado.

1.11.11 [10] <§1.6> Determine a taxa de clock se o CPI for reduzido em 15% e o tempo de CPU em 20%, enquanto o número de instruções permanece inalterado.

1.12 A [Seção 1.10](#) cita como armadilha a utilização de um subconjunto da equação de desempenho como uma métrica de desempenho. Para ilustrar isso, considere os dois processadores a seguir. P1 tem uma taxa de clock de 4 GHz, CPI médio de 0,9 e requer a execução de 5,0E9 instruções. P2 tem uma taxa de clock de 3 GHz, um CPI médio de 0,75 e requer a execução de 1,0E9 instruções.

1.12.1 [5] <§§1.6, [1.10](#)> Uma falácia comum é considerar o computador com a maior taxa de clock como tendo o maior desempenho. Verifique se isso é verdade para P1 e P2.

1.12.2 [10] <§§1.6, 1.10> Outra falácia é considerar que o processador executando o maior número de instruções precisará de um tempo de CPU maior. Considerando que o processador P1 está executando uma sequência de 1,0E9 instruções e que o CPI dos processadores P1 e P2 não muda, determine o número de instruções que P2 pode executar ao mesmo tempo em que P1 precisa para executar 1,0E9 instruções.

1.12.3 [10] <§§1.6, 1.10> Uma falácia comum é usar milhões de instruções por segundo (MIPS) para comparar o desempenho de dois processadores diferentes e considerar que o processador com o maior valor de MIPS tem o maior desempenho. Verifique se isso é verdade para P1 e P2.

1.12.4 [10] <§1.10> Outro valor de desempenho comum é milhões de operações de ponto flutuante por segundo (MFLOPS), definido como:

$$\text{MFLOPS} = \text{Nº de operações de PF} / (\text{Tempo de execução} \times 1\text{E}6)$$

mas esse valor tem os mesmos problemas do MIPS. Considere que 40% das instruções executadas em P1 e P2 sejam instruções de ponto flutuante. Determine os valores de MFLOPS para os programas.

1.13 Outra armadilha citada na [Seção 1.10](#) é esperar aprimorar o desempenho geral de um computador melhorando apenas um aspecto do computador. Considere um computador rodando um programa que requer 250 s, com 70 s

gastos executando instruções de ponto flutuante, 85 s executando instruções de L/S, e 40 s gastos executando instruções de desvio.

1.13.1 [5] <§1.10> Em quanto será reduzido o tempo total se o tempo para as operações de PF for reduzido em 20%?

1.13.2 [5] <1.10> Em quanto o tempo para operações INT será reduzido se o tempo total for reduzido em 20%?

1.13.3 [5] <1.10> O tempo total pode ser reduzido em 20% reduzindo-se apenas o tempo para as instruções de desvio?

1.14 Suponha que um programa exija a execução de 50×10^6 instruções de PF, 110×10^6 instruções INT, 80×10^6 instruções de L/S e 16×10^6 instruções de desvio. O CPI para cada tipo de instrução é 1, 1, 4 e 2, respectivamente.

Suponha que o processador tenha uma taxa de clock de 2 GHz.

1.14.1 [10] <§1.10> Em quanto devemos melhorar o CPI para instruções de PF se quisermos que o programa seja executado duas vezes mais rápido?

1.14.2 [10] <§1.10> Em quanto devemos melhorar o CPI para instruções de L/S se quisermos que o programa seja executado duas vezes mais rápido?

1.14.3 [5] <§1.10> Em quanto o tempo de execução do programa deve ser melhorado se o CPI das instruções INT e PF for reduzido em 40% e o CPI de L/S e desvio for reduzido em 30%?

1.15 [5] <§1.8> Quando um programa é adaptado para ser executado em vários processadores em um sistema multiprocessador, o tempo de execução em cada processador é composto de tempo de execução e o tempo de overhead exigido para as seções críticas bloqueadas e/ou para enviar dados de um processador para outro.

Suponha que um programa exija $t = 100$ s de tempo de execução em um processador. Quando p processadores estiverem em execução, cada processador requer t/p s, além de mais 4 s de overhead, independentemente do número de processadores. Calcule o tempo de execução por processador para 2, 4, 8, 16, 32, 64 e 128 processadores. Para cada caso, liste o ganho de velocidade correspondente em relação a um único processador e a razão entre o ganho de velocidade real e o ganho de velocidade ideal (ganho de velocidade se não houvesse overhead).

Respostas das Seções “Verifique você mesmo”

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

§1.4, página 20: Memória DRAM: volátil, tempo de acesso curto de 50 a 70 nanossegundos, e custo por GB é US\$ 5 a US\$ 10. Memória em disco: não

volátil, tempos de acesso são de 100.000 a 400.000 vezes mais lento que a DRAM, e custo por GB é 100 vezes mais barato que a DRAM. Memória flash: não volátil, tempos de acesso são de 100 a 1000 vezes mais lentos que a DRAM, e custo por GB é 7 a 10 vezes mais barato que a DRAM.

§1.5, página 24: 1, 3 e 4 são motivos válidos. A resposta 5 geralmente pode ser verdadeira, pois o alto volume pode tornar o investimento extra para reduzir o tamanho do die em, digamos, 10%, uma boa decisão econômica, mas isso não precisa ser verdadeiro.

§1.6, página 28: 1. a: ambos, b: latência, c: nem um nem outro. 7 segundos.

§1.6, página 33: b.

§1.10, página 43: a. Computador A tem a maior avaliação MIPS. b.

Computador B é mais rápido.

Instruções: A Linguagem dos Computadores

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

Charles V, imperador romano (1500-1558)

- 2.1 Introdução
- 2.2 Operações do hardware do computador
- 2.3 Operandos do hardware do computador
- 2.4 Números com sinal e sem sinal
- 2.5 Representando instruções no computador
- 2.6 Operações lógicas
- 2.7 Instruções para tomada de decisões
- 2.8 Suporte a procedimentos no hardware do computador
- 2.9 Comunicando-se com as pessoas
- 2.10 Endereçamento no MIPS para operandos imediatos e endereços de 32 bits
- 2.11 Paralelismo e instruções: Sincronização
- 2.12 Traduzindo e iniciando um programa
- 2.13 Um exemplo de ordenação em C para juntar tudo isso
- 2.14 Arrays *versus* ponteiros
- 2.15 Vida real: instruções ARMv7 (32 bits)

2.16 Vida real: Instruções x86

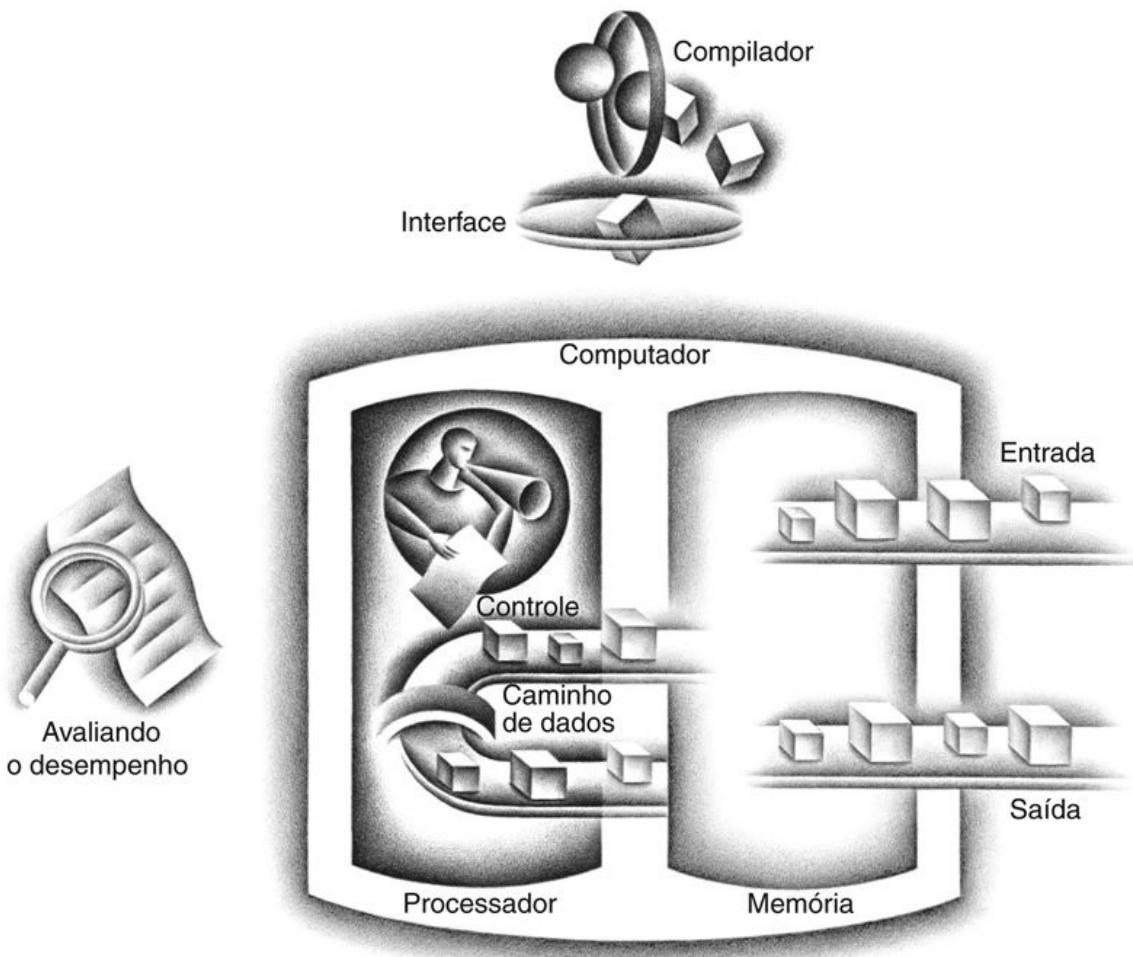
2.17 Vida real: instruções ARMv8 (64 bits)

2.18 Falácia e armadilhas

2.19 Comentários finais

2.20 Exercícios

Os cinco componentes clássicos de um computador



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 nosso caminho, expondo o hardware para a aritmética e a representação dos números de ponto flutuante.

conjunto de instruções

O vocabulário dos comandos entendidos por uma determinada arquitetura.

Você poderia pensar que as linguagens dos computadores fossem tão diversificadas quanto as dos humanos, mas, na realidade, as linguagens de computação são muito semelhantes, mais parecidas com dialetos regionais do que linguagens independentes. Logo, quando você aprender uma, será fácil entender as outras.

O conjunto de instruções escolhido vem da *MIPS Technologies*, e é um exemplo elegante dos conjuntos de instruções projetados desde a década de 1980. Para demonstrar como é fácil selecionar outros conjuntos de instruções, daremos uma olhada rápida em três outros conjuntos de instruções populares.

1. ARMv7 é semelhante ao MIPS. Mais de 9 bilhões de chips com processadores ARM foram fabricados em 2011, tornando-o o conjunto de instruções mais popular no mundo.
2. O segundo exemplo é o Intel x86, que controla tanto o PC quanto a nuvem da era pós-PC.
3. O terceiro exemplo é o ARMv8, que amplia o tamanho de endereçamento do ARMv7 de 32 bits para 64 bits. Ironicamente, conforme veremos, esse conjunto de instruções de 2013 está mais próximo do MIPS do que do ARMv7.

A semelhança dos conjuntos de instruções 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. Este 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 de hoje, 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.

Aprendendo como representar as instruções, você também descobrirá o segredo da computação: o **conceito de programa armazenado**. 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.

conceito de programa armazenado

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

Revelamos o conjunto de instruções do MIPS aos poucos, mostrando o raciocínio em conjunto com as estruturas do computador. Esse tutorial passo a passo entrelaça os componentes com suas explicações, tornando a linguagem de máquina mais fácil de digerir. A [Figura 2.1](#) oferece uma prévia do conjunto de instruções abordado neste capítulo.

Operandos do MIPS

Nome	Exemplo	Comentários
32 registradores	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Localizações rápidas para dados. No MIPS, os dados precisam estar em regiões para realizar aritmética. O registrador \$zero sempre é igual a 0 e o registrador \$at é reservado pelo montador para lidar com constantes grandes.
2^{30} palavras de memória	Memória[0], Memória[4], ..., Memória[4294967292]	Acessada apenas pelas instruções de transferência de dados. O MIPS utiliza endereços de byte, de modo que os endereços sequenciais de dados diferem em 4. A memória mantém estruturas de dados, matrizes e registros espalhados.

Linguagem 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
	add immediate	addi \$s1,\$s2,20	$\$s1 = \$s2 + 20$	Usada para somar constantes
Transferência de dados	load word	lw \$s1,20(\$s2)	$\$s1 = \text{Memória}[\$s2 + 20]$	Dados da memória para o registrador
	store word	sw \$s1,20(\$s2)	$\text{Memória}[\$s2 + 20] = \$s1$	Dados do registrador para a memória
	load half	lh \$s1,20(\$s2)	$\$s1 = \text{Memória}[\$s2 + 20]$	Halfword da memória para registrador
	load half unsigned	lhu \$s1,20(\$s2)	$\$s1 = \text{Memória}[\$s2 + 20]$	Halfword da memória para registrador
	store half	sh \$s1,20(\$s2)	$\text{Memória}[\$s2 + 20] = \$s1$	Halfword de um registrador para memória
	load byte	lb \$s1,20(\$s2)	$\$s1 = \text{Memória}[\$s2 + 20]$	Byte da memória para registrador
	load byte unsigned	lbu \$s1,20(\$s2)	$\$s1 = \text{Memória}[\$s2 + 20]$	Byte da memória para registrador
	store byte	sb \$s1,20(\$s2)	$\text{Memória}[\$s2 + 20] = \$s1$	Byte de um registrador para memória
	load linked word	ll \$s1,20(\$s2)	$\$s1 = \text{Memória}[\$s2 + 20]$	Carrega word como 1ª metade do swap atômico
	store condition, word	sc \$s1,20(\$s2)	$\text{Memória}[\$s2 + 20] = \$s1; \$s1 = 0 \text{ or } 1$	Armazena word como 2ª metade do swap atômico
Lógica	load upper immed.	lui \$s1,20	$\$s1 = 20 * 2^{16}$	Carrega constante nos 16 bits mais altos
	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	$\$s1 = \sim (\$s2 \$s3)$	Três operadores em registrador; NOR bit a bit
	and immediate	andi \$s1,\$s2,20	$\$s1 = \$s2 & 20$	AND bit a bit registrador com constante
	or immediate	ori \$s1,\$s2,20	$\$s1 = \$s2 20$	OR bit a bit registrador com constante
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 << 10$	Deslocamento à esquerda por constante
Desvio condicional	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 >> 10$	Deslocamento à direita por constante
	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 ($\$s2 < \$s3$) $\$s1 = 1$; $\$s1 = 0$	Compara menor que; usado com beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	If ($\$s2 < \$s3$) $\$s1 = 1$; $\$s1 = 0$	Compara menor que sem sinal
	set less than immediate	slti \$s1,\$s2,20	If ($\$s2 < 20$) $\$s1 = 1$; $\$s1 = 0$	Compara menor que constante
	set less than immediate unsigned	sltiu \$s1,\$s2,20	If ($\$s2 < 20$) $\$s1 = 1$; $\$s1 = 0$	Compara menor que constante 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	$\$ra = PC + 4; \text{go to } 10000$	Para chamada de procedimento

FIGURA 2.1 Assembly do MIPS revelado neste capítulo.

Esta informação também pode ser encontrada na Coluna 1 do Guia de Referência do MIPS no final deste livro.

2.2. Operações do hardware do computador

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

Burks, Goldstine e von Neumann, 1947

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.)

Esta sequência de instruções 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.
```

Portanto, são necessárias três instruções para somar quatro variáveis.

As palavras à direita do símbolo (#) em cada linha acima são *comentários* para o leitor humano, e os computadores os ignoram. Note que, diferentemente de outras linguagens de programação, cada linha desta linguagem pode conter, no máximo, uma instrução. Outra diferença para a linguagem C é que comentários sempre terminam no final da 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 em C no MIPS

Exemplo

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.

Resposta

Uma instrução MIPS opera sobre 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
```

Compilando uma atribuição em 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)
```

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, o Java foi idealizado originalmente como 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 em C, esses compiladores Java normalmente são denominados compiladores JIT (*Just In Time* — no momento exato). A Seção 2.12 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.

2.3. Operandos do hardware do computador

Ao contrário dos programas em 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 primitivas usadas no projeto do hardware que também são visíveis ao programador quando o projeto do computador é concluído, portanto, você pode pensar nos registradores como os “tijolos” na construção do computador. 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.

Word (palavra)

A unidade de acesso natural de um computador, normalmente um grupo de 32 bits; corresponde ao tamanho de um registrador 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, como o MIPS. Assim, continuando em nossa evolução 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.

A razão para o limite dos 32 registradores pode ser encontrada no segundo dos três 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 2.5](#).

O [Capítulo 4](#) mostra 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 2.8](#) 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.

Compilando uma atribuição em C usando registradores

Exemplo

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?

Resposta

O programa compilado é muito semelhante ao exemplo anterior, exceto que substituímos 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)
```

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 — arrays (ou sequências) e estruturas. 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 no início deste capítulo. 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 palavra 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.

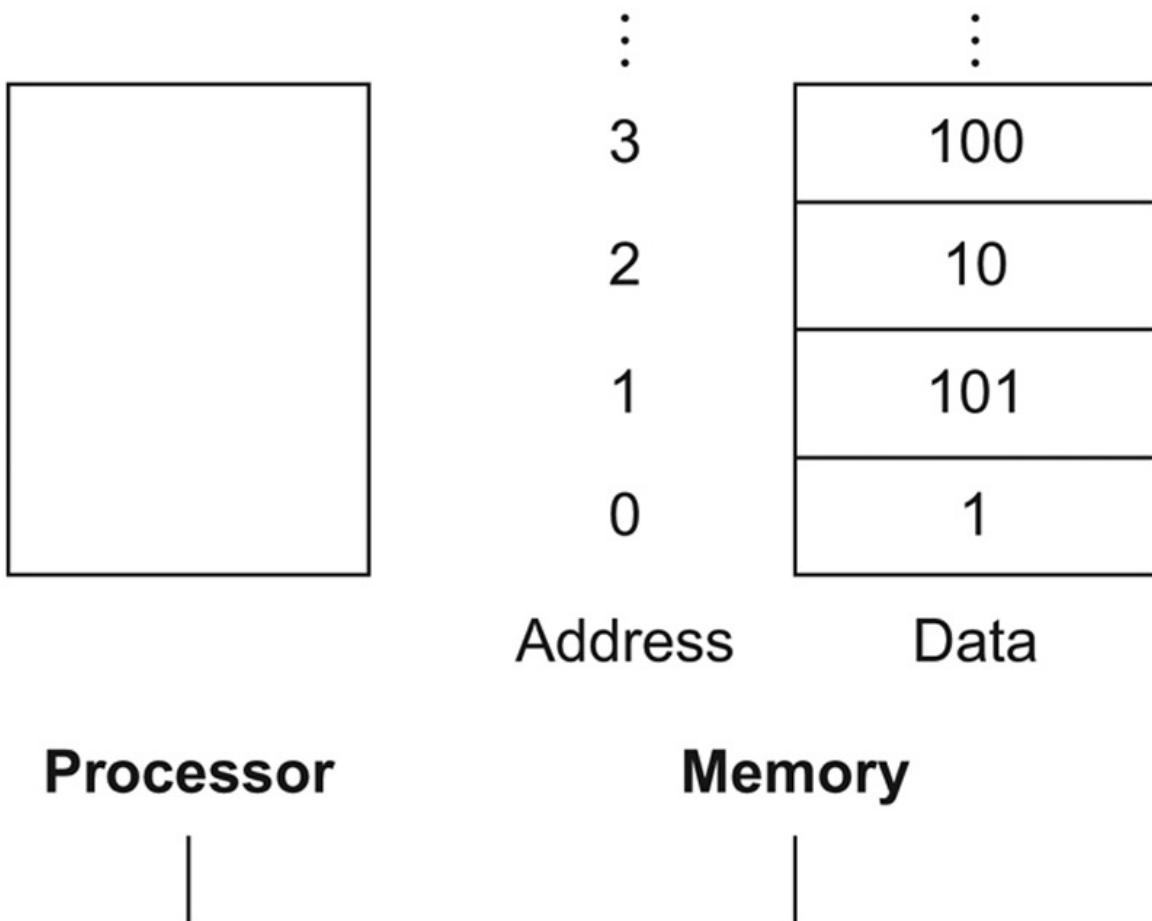


FIGURA 2.2 Endereços de memória e conteúdo da memória nesses locais.

Se esses elementos fossem palavras, esses endereços estariam incorretos, pois o MIPS, na realidade, usa endereços de bytes, com cada palavra representando quatro bytes. A [Figura 2.3](#) mostra o endereçamento para palavras sequenciais na memória.

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 específicos dentro de uma sequência da memória.

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).

Compilando uma atribuição quando um operando está na memória

Exemplo

Vamos supor que *A* seja uma sequência de 100 palavras 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 8. 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]
```

(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 (8) é chamada de *offset* e o registrador acrescentado para formar o endereço (\$s3) 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 atuais endereça bytes individuais. Portanto, o endereço de uma palavra combina os endereços dos 4 bytes dentro da palavra. Logo, os endereços sequenciais das palavras diferem em quatro vezes. Por exemplo, a Figura 2.3 mostra os endereços MIPS reais para a Figura 2.2; o endereço em bytes da terceira palavra é 8.

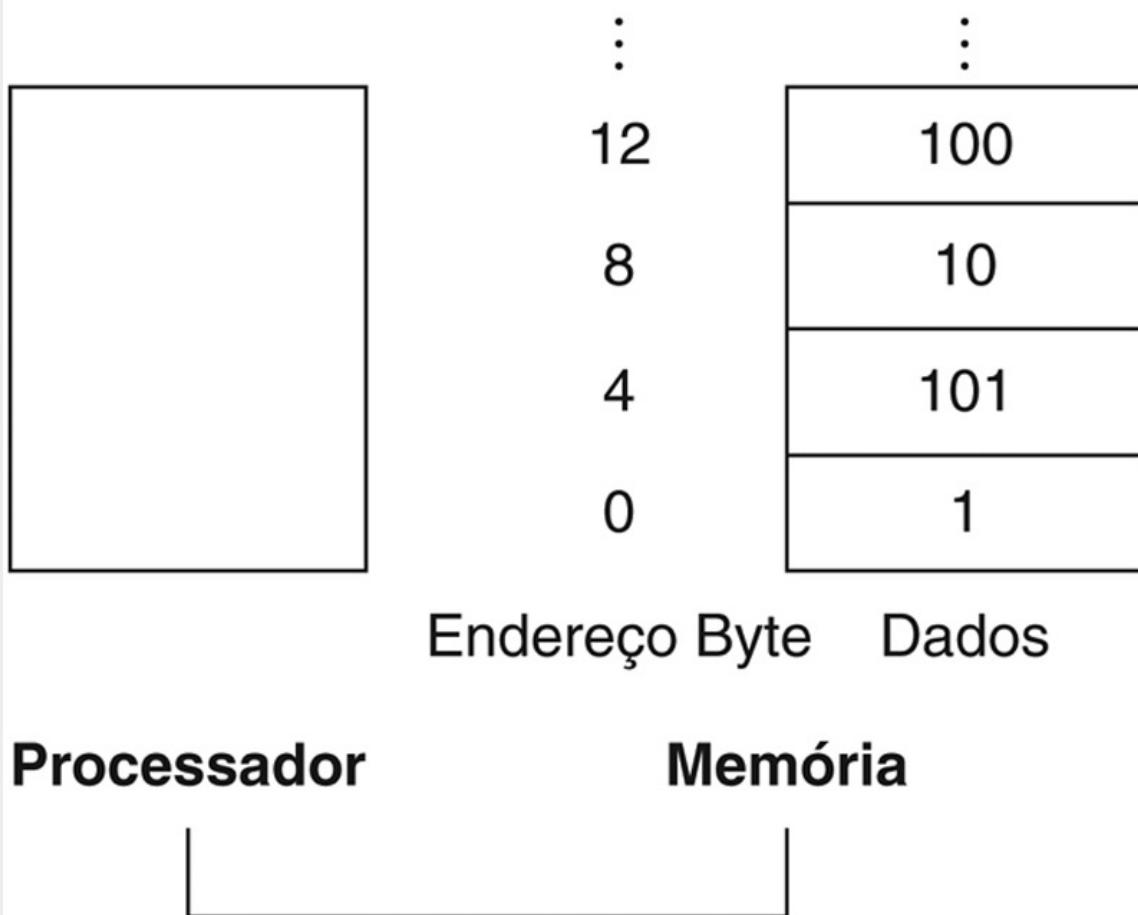


FIGURA 2.3 Endereços reais de memória do MIPS e conteúdo da memória para essas palavras.

A mudança de endereços está destacada para comparar com a Figura 2.2. Como o MIPS endereça cada byte, endereços de palavras são múltiplos de 4: existem 4 bytes em uma palavra.

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

restrição de alinhamento

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

Os computadores se dividem entre aqueles que utilizam o endereço do byte mais à esquerda, ou *big end*, como endereço da palavra e os que utilizam o

byte mais à direita, ou *little end*. O MIPS está no campo do *big endian*. Visto que a ordem só importa se você acessar os dados idênticos tanto como uma palavra quanto como quatro bytes, poucos precisam estar cientes da ordenação dos bytes. (O Apêndice A mostra as duas opções para numerar os bytes de uma palavra.)

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 2.18.)

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* (armazenar palavra).

Interface hardware/software

Como os endereços nos loads e stores são números binários, podemos ver por que a DRAM para a memória vem em tamanhos binários, e não em tamanhos decimais. Ou seja, em gibibytes (2^{30}) ou tebibytes (2^{40}), e não em gigabytes (10^9) ou terabytes (10^{12}) (Figura 1.1).

Compilando com load e store

Exemplo

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, desta vez, usamos o offset apropriado para o endereçamento do byte na instrução load word, a fim de 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 (4×12) como offset e o registrador \$s3 como registrador base.

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

Load word e store word são as instruções que copiam words entre memória e registradores na arquitetura MIPS. Outras marcas de computadores utilizam outras instruções juntamente com load e store para transferir dados. Uma arquitetura com essas alternativas é a Intel x86, descrita na [Seção 2.16](#).

Interface hardware/software

Muitos programas possuem mais variáveis do que os computadores possuem registradores. Consequentemente, o compilador tenta manter as variáveis usadas com mais frequência nos registradores e coloca o restante na memória, usando loads e stores para mover variáveis entre os registradores e a memória. O processo de colocar as variáveis menos utilizadas (ou aquelas necessárias mais adiante) na memória é chamado de *spilled registers* (ou *registradores derramados*).

O princípio de hardware relacionando tamanho e velocidade sugere que a memória deve ser mais lenta que os registradores, pois existem menos registradores. Isso realmente acontece; os acessos aos dados são mais rápidos se os dados estiverem nos registradores, ao invés de estarem na memória.

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

Assim, os registradores MIPS levam menos tempo para serem acessados e possuem maior vazão do que a memória, tornando os dados nos registradores mais rápidos de acessar e mais simples de usar. O acesso aos registradores também usa menos energia do que o acesso à memória. Para conseguir o melhor desempenho e economizar energia, uma arquitetura de conjunto de instruções precisa ter um número suficiente de registradores, e os compiladores precisam usar os registradores de modo eficaz.

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 SPEC2006.

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

supondo que \$s1 + AddrConstant4 seja o endereço de memória da constante 4.

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
```

Os operandos constantes ocorrem com frequência e, incluindo constantes dentro das instruções aritméticas, as operações são muito mais rápidas e usam menos energia do que se as constantes fossem lidas da memória.

A constante zero tem outro emprego, que é simplificar o conjunto de instruções por oferecer variações úteis. Por exemplo, a operação move é apenas uma instrução de soma na qual cada operando é zero. Portanto, o MIPS dedica o registrador \$zero para ter sempre o valor zero. (Como você deve esperar, ele é o registrador número 0.) O uso da frequência para justificar as inclusões de constantes é outro exemplo da grande ideia de tornar o **caso comum veloz**.



CASO COMUM VELOZ

Verifique você mesmo

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

1. Muito rápida: eles aumentam tão rapidamente quanto a Lei de Moore, que prevê a duplicação 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

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. As Seções 2.16 e 2.18 mostram a diferença muito maior entre o ARMv7 com endereços de 32 bits e o seu sucessor de 64 bits, o ARMv8.

Detalhamento

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.

Detalhamento

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.

Detalhamento

Como o MIPS admite constantes negativas, a subtração imediata não é necessária no MIPS.

2.4. Números com sinal e sem sinal

Primeiro, vamos revisar rapidamente como um computador representa números. Os humanos são ensinados a pensar na base 10, mas os números podem ser representados em qualquer base. Por exemplo, 123 base 10 = 1111011 base 2.

Os números são mantidos no hardware do computador como uma série de sinais eletrônicos altos e baixos, e por isso são considerados números de base 2.

(Assim como os números de base 10 são chamados números *decimais*, os números de 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 de **dígitos binários** ou *bits*. Esse bloco de montagem fundamental pode assumir dois valores, que podem ser imaginados como várias alternativas: alto ou baixo, ligado ou desligado, verdadeiro ou falso ou 1 ou 0.

dígito binário

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

Generalizando, em qualquer base numérica, o valor do i -ésimo dígito d é

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

em que 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. Subscritamos os números decimais com *dec* e os números binários com *bin*. Por exemplo,

$$1011_{\text{bin}}$$

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 palavra. 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	0	1	0	1	1	

(32 bits wide)

Como as palavras 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).

bit menos significativo

O bit mais à direita em uma palavra MIPS.

bit mais significativo

O bit mais à esquerda em uma palavra MIPS.

Cada palavra no MIPS possui 32 bits de largura, de modo que podemos representar 2^{32} padrões diferentes de 32 bits. É natural deixar que essas representações mostrem os números de 0 a $2^{32} - 1$ ($4.294.967.295_{\text{dec}}$):

$$\begin{aligned} 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000_{\text{bin}} &= 0_{\text{dec}} \\ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0001_{\text{bin}} &= 1_{\text{dec}} \\ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0010_{\text{bin}} &= 2_{\text{dec}} \\ \dots &\quad \dots \\ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1101_{\text{bin}} &= 4.294.967.293_{\text{dec}} \\ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1110_{\text{bin}} &= 4.294.967.294_{\text{dec}} \\ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111_{\text{bin}} &= 4.294.967.295_{\text{dec}} \end{aligned}$$

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):

$$(x_{31} \times 2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \dots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Por motivos que veremos em breve, esses números positivos são denominados números sem sinal.

Interface hardware/software

A base 2 não é natural para os seres humanos; temos 10 dedos e, portanto, a base 10 é natural. Por que os computadores não usaram decimal? Na verdade, o primeiro computador comercial *oferecia* aritmética decimal. O problema foi que o computador ainda usava sinais do tipo ligado e desligado, de modo que um dígito decimal era simplesmente representado por vários dígitos binários. O sistema decimal mostrou ser tão ineficaz que os computadores subsequentes passaram a usar apenas binário, convertendo para a base 10 apenas para eventos de entrada/saída relativamente pouco frequentes.

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* (ou *estouro*). Fica a critério da linguagem de programação, do sistema operacional e do programa determinar o que fazer quando isso ocorre.

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 óvia 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 sequência de 0s à esquerda, de modo que o resultado seria uma sequência 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 significa positivo e 1s iniciais significa negativo. Essa convenção para representar os números binários com sinal é chamada representação por *complemento de dois*:

0000 0000 0000 0000 0000 0000 0000 0000 _{bin}	= 0 _{dec}
0000 0000 0000 0000 0000 0000 0000 0001 _{bin}	= 1 _{dec}
0000 0000 0000 0000 0000 0000 0000 0010 _{bin}	= 2 _{dec}
...	...
0111 1111 1111 1111 1111 1111 1111 1101 _{bin}	= 2.147.483.645 _{dec}
0111 1111 1111 1111 1111 1111 1111 1110 _{bin}	= 2.147.483.646 _{dec}
0111 1111 1111 1111 1111 1111 1111 1111 _{bin}	= 2.147.483.647 _{dec}
1000 0000 0000 0000 0000 0000 0000 0000 _{bin}	= - 2.147.483.648 _{dec}
1000 0000 0000 0000 0000 0000 0000 0001 _{bin}	= - 2.147.483.647 _{dec}
1000 0000 0000 0000 0000 0000 0000 0010 _{bin}	= - 2.147.483.646 _{dec}
...	...
1111 1111 1111 1111 1111 1111 1111 1101 _{bin}	= - 3 _{dec}
1111 1111 1111 1111 1111 1111 1111 1110 _{bin}	= - 2 _{dec}
1111 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 de dois possui um número negativo, -2.147.483.648_{dec}, que não possui um número positivo correspondente. Este desequilíbrio era uma preocupação para o programador desatento, mas a

representação com sinal e magnitude gerava problemas para o programador e para o projetista do hardware. Consequentemente, todo computador, hoje em dia, utiliza a representação de números binários por complemento de dois para os números com sinal.

A representação por complemento de dois tem a vantagem de que todos os números negativos possuem 1 no bit mais significativo. Consequentemente, 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 é multiplicado pelas versões positivas de seus respectivos valores de base.

Conversão de binário para decimal

Exemplo

Qual é o valor decimal deste número em complemento de dois com 32 bits?

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

Resposta

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 de negativo para positivo.

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 de 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 e sem sinal se aplica 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 do sinal* —, mas sua finalidade é colocar uma 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 é sem sinal.

Ao carregar uma word de 32 bits em 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* (1b) trata o byte como um número com sinal e, assim, estende por sinal para preencher os 24 bits mais à esquerda do registrador, enquanto *load byte unsigned* (1bu) trabalha com inteiros sem sinal. Como os programas em C quase sempre usam bytes para representar caracteres, em vez de considerar bytes como inteiros com sinal muito curtos, 1bu é usada de modo praticamente exclusivo para os loads de byte.

Interface hardware/software

Diferente dos números discutidos anteriormente, os endereços de memória naturalmente 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 `sighned int`, para deixar a distinção clara.

Vamos examinar dois atalhos úteis quando trabalhamos com os números em complemento de dois. O primeiro atalho é um modo rápido de negar um número binário no complemento de dois. Basta inverter cada 0 para 1 e cada 1 para 0, depois somar um ao resultado. Este 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 $x + \bar{x} = -1$, portanto, $x + \bar{x} + 1 = 0$ ou $\bar{x} + 1 = -x$. (Usamos a notação \bar{x} para significar inverter cada bit em x de 0 para 1 e vice-versa.)

Atalho para negação

Exemplo

Negue 2_{dec} e depois verifique o resultado negando -2_{dec} .

Resposta

$$2_{\text{dec}} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{bin}}$$

Negando esse número, invertendo os bits e somando um,

$$\begin{array}{r} 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{\text{bin}} \\ + \qquad 1_{\text{bin}} \\ \hline = \qquad 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{bin}} \\ = \qquad -2_{\text{dec}} \end{array}$$

Na outra direção,

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{bin}}$$

primeiro é invertido e depois incrementado:

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{bin}} \\
 + \hspace{10em} 1_{\text{bin}} \\
 \hline
 = \hspace{1em} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{bin}} \\
 = \hspace{1em} 2_{\text{dec}}
 \end{array}$$

O próximo 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 de 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

Exemplo

Converta as versões binárias de 16 bits de 2_{dec} e -2_{dec} para números binários de 32 bits.

Resposta

A versão binária de 16 bits do número 2 é

$$0000\ 0000\ 0000\ 0010_{\text{bin}} = 2_{\text{dec}}$$

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:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{bin}} = 2_{\text{dec}}$$

Vamos negar a versão de 16 bits de 2 usando o atalho anterior. Assim,

$$0000\ 0000\ 0000\ 0010_{\text{bin}}$$

torna-se

$$\begin{array}{r} 1111\ 1111\ 1111\ 1101_{\text{bin}} \\ + \qquad \qquad \qquad 1_{\text{bin}} \\ \hline = \ 1111\ 1111\ 1111\ 1110_{\text{bin}} \end{array}$$

Criar uma versão de 32 bits do número negativo significa copiar o bit de sinal 16 vezes e colocá-lo à esquerda:

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{bin}} = -2_{\text{dec}}$$

Esse truque funciona porque os números positivos em complemento de dois realmente possuem uma quantidade infinita de 0s à esquerda e os que são negativos em complemento de 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.

Resumo

O ponto principal desta seção é que precisamos representar inteiros positivos e negativos dentro de uma palavra do computador e, embora existam prós e contras a qualquer opção, a escolha predominante desde 1965 tem sido o complemento de dois.

Detalhamento

Para números decimais sem sinal, usamos “—” para representar negativo, pois não existem limites para o tamanho de um número decimal. Dado um tamanho de word fixo, strings com bits binários e hexadecimais (Figura 2.4) podem codificar o sinal; logo, normalmente não usamos “ + ” ou “—” com notação binária ou hexadecimal.

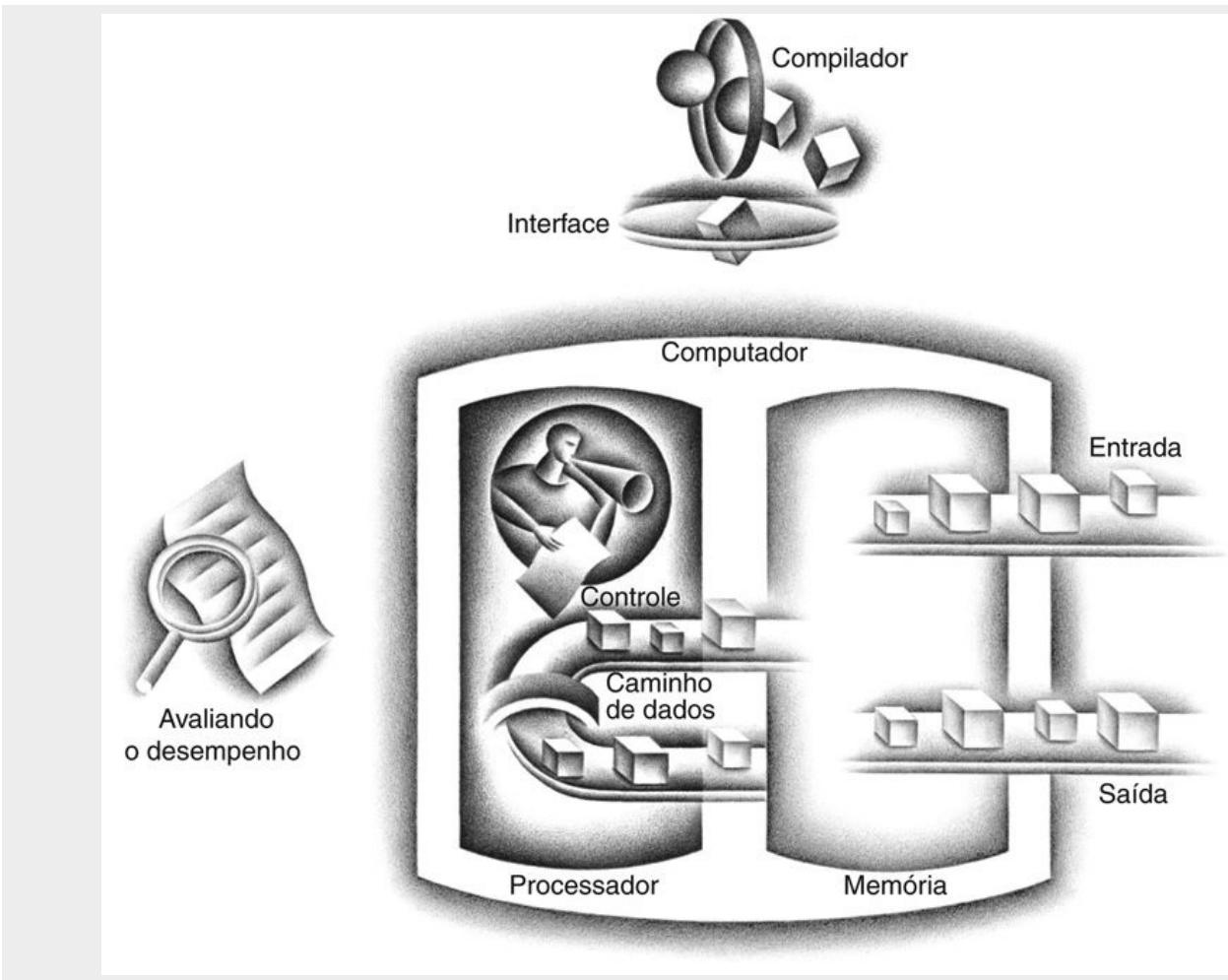
Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal
0 _{hexa}	0000 _{bin}	4 _{hexa}	0100 _{bin}	8 _{hexa}	1000 _{bin}	c _{hexa}	1100 _{bin}
1 _{hexa}	0001 _{bin}	5 _{hexa}	0101 _{bin}	9 _{hexa}	1001 _{bin}	d _{hexa}	1101 _{bin}
2 _{hexa}	0010 _{bin}	6 _{hexa}	0110 _{bin}	a _{hexa}	1010 _{bin}	e _{hexa}	1110 _{bin}
3 _{hexa}	0011 _{bin}	7 _{hexa}	0111 _{bin}	b _{hexa}	1011 _{bin}	f _{hexa}	1111 _{bin}

FIGURA 2.4 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.

Verifique você mesmo

Qual é o valor decimal deste número de 64 bits em complemento de dois?



1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
 1111 1000_{bin}

- 1) -4_{dec}
- 2) -8_{dec}
- 3) -16_{dec}
- 4) 18.446.744.073.709.551.609_{dec}

Detalhamento

O complemento de dois recebe esse nome em decorrência da regra de que a soma sem sinal de um número de n bits e seu negativo de n bits é 2^n ; logo, o

complemento ou a negação de um número em complemento de dois x é $2^n - x$.

Uma terceira representação alternativa para o complemento de dois, de sinal e magnitude é chamada **complemento de um**. O negativo de um complemento de um é encontrado invertendo-se cada bit, de 0 para 1 e de 1 para 0 ou x^- . Essa relação ajuda a explicar seu nome, pois o complemento de x é $2^n - x - 1$. 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 de dois, exceto que também possui dois 0s: $00\dots00_{\text{bin}}$ é o 0 positivo, e $11\dots11_{\text{bin}}$ é o 0 negativo. O maior número negativo $10\dots000_{\text{bin}}$ representa $-2.147.483.647_{\text{dec}}$ e, por isso, os positivos e negativos são balanceados. Os que aderiram ao complemento de um precisaram de uma etapa extra para subtrair um número e, por isso, o complemento de dois domina hoje.

complemento de um

Uma notação que representa o valor mais negativo por $10 \dots 000_{\text{bin}}$ e o valor mais positivo por $01 \dots 11_{\text{bin}}$, deixando um número igual de negativos e positivos, mas terminando com dois zeros, um positivo ($00 \dots 00_{\text{bin}}$) e um negativo ($11 \dots 11_{\text{bin}}$). O termo também é usado para significar a inversão de cada bit em um padrão: 0 para 1 e 1 para 0.

Uma notação final, que veremos quando tratarmos de ponto flutuante no Capítulo 3, é representar o valor mais negativo por $00\dots000_{\text{bin}}$ e o valor mais positivo por $11\dots11_{\text{bin}}$, com 0 normalmente tendo o valor $10\dots00_{\text{bin}}$. 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

Uma notação que representa o valor mais negativo por $00 \dots 000_{\text{bin}}$ e o valor mais positivo por $11 \dots 11_{\text{bin}}$, com 0 normalmente tendo o valor $10 \dots 00_{\text{bin}}$, deslocando assim o número, de modo que o número mais o deslocamento têm uma representação não negativa.

2.5. Representando instruções no computador

Agora, estamos prontos para explicar a diferença entre o modo como os humanos instruem os computadores e como os computadores veem as instruções.

As instruções 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 são referenciados por quase todas as instruções, é preciso haver uma convenção para mapear nomes de registrador em números. Na linguagem assembly do 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.

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 fonte 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

Esse layout da instrução é chamado **formato de instrução**. Como você pode ver pela contagem do número de bits, essa instrução MIPS ocupa exatamente 32 bits — o mesmo tamanho da palavra de dados. Acompanhando nosso princípio de projeto, de que a simplicidade favorece a regularidade, todas as instruções MIPS possuem 32 bits de extensão.

formato de instrução

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

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

linguagem de máquina

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

Pode parecer que agora você estará lendo e escrevendo sequências 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 comuns. Como a base 16 é uma potência de 2, podemos converter facilmente substituindo cada grupo de quatro dígitos binários por um único dígito hexadecimal e vice-versa. A [Figura 2.4](#) converte hexadecimal para binário e vice-versa.

hexadecimal

Números na base 16.

Visto que frequentemente 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

Exemplo

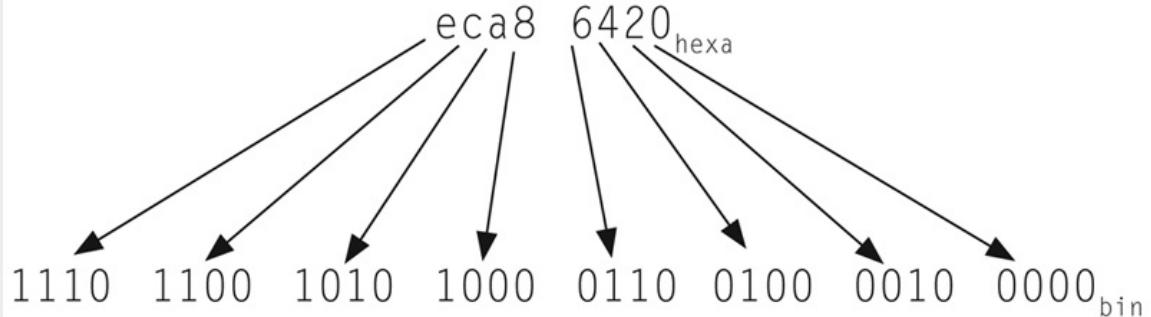
Converta os seguintes números hexadecimais e binários para a outra base:

eca8 6420_{hexa}

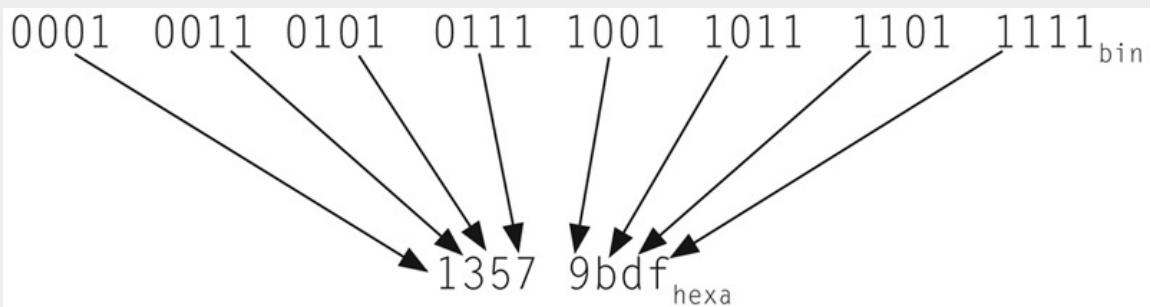
0001 0011 0101 0111 1001 1011 1101 1111_{bin}

Resposta

Usando a Figura 2.4, temos a solução ao olhar na tabela em uma direção:



E depois na outra direção:



Campos do MIPS

Os campos do MIPS recebem nomes para facilitar seu tratamento:

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 nos campos das instruções MIPS:

- **op**: operação básica da instrução, tradicionalmente chamado de **opcode**.
- **rs**: o primeiro registrador do operando fonte.
- **rt**: o segundo registrador do operando fonte.
- **rd**: o registrador do operando de destino. Ele recebe o resultado da operação.
- **shamt**: “Shift amount” (quantidade de deslocamento). (A [Seção 2.6](#) explica as instruções de shift e esse termo; ele não será usado até lá, e, por isso, o campo contém zero nesta seção.)
- **funct**: função. Esse campo, normalmente chamado *código de função*,

seleciona a variante específica da operação no campo op.

opcode

O campo que denota a operação e 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. Esta 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 3: 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 palavra 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 $\pm 2^{15}$. 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 palavra.

Vejamos a instrução load word apresentada anteriormente:

```
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 tamanho do quarto campo no tipo I é igual à soma dos tamanhos 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 2.5](#) mostra os números utilizados em cada campo para as instruções MIPS descritas aqui.

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 2.5 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

Exemplo

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?

Resposta

Por conveniência, primeiro vamos representar as instruções em linguagem de máquina usando os números decimais. Pela Figura 2.5, 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 (Figura 2.5) 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).

A instrução add, que vem em seguida, é especificada com 0 no primeiro campo (op) e 32 no último campo (funct). Os três operandos de registrador (18, 8 e 8) aparecem no segundo, no terceiro e no 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.

Como $1200_{dec} = 0000\ 0100\ 1011\ 0000_{bin}$, o equivalente binário ao formato

decimal é o seguinte:

10 0 011	01001	01000	0000 0100 1011 0000			
000000	10010	01000	01000		00000	100000
10 1 011	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, que está destacado.

Interface hardware/software

O desejo de manter todas as instruções com o mesmo tamanho está em conflito com o desejo de ter o máximo de registradores possível. Qualquer aumento no número de registradores usa, pelo menos, um bit a mais em cada campo de registrador do formato da instrução. Dadas essas restrições e o princípio de projeto de que menor é mais rápido, a maior parte dos conjuntos de instruções hoje possui 16 ou 32 registradores de uso geral.

A [Figura 2.6](#) resume as partes do assembly do MIPS descritas nesta seção. Como veremos no [Capítulo 4](#), a semelhança das representações binárias de instruções relacionadas simplifica o projeto do hardware. Essas instruções são outro exemplo da regularidade da arquitetura MIPS.

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
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 2.6 Arquitetura MIPS revelada até a Seção 2.5.

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, em que 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 2.6](#); e o campo *funct*, que particulariza a operação específica das instruções no formato R. O formato I mantém os 16 bits finais como um único campo de *endereço*.

Linguagem de máquina do MIPS

Colocando em perspectiva

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 2.7 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.

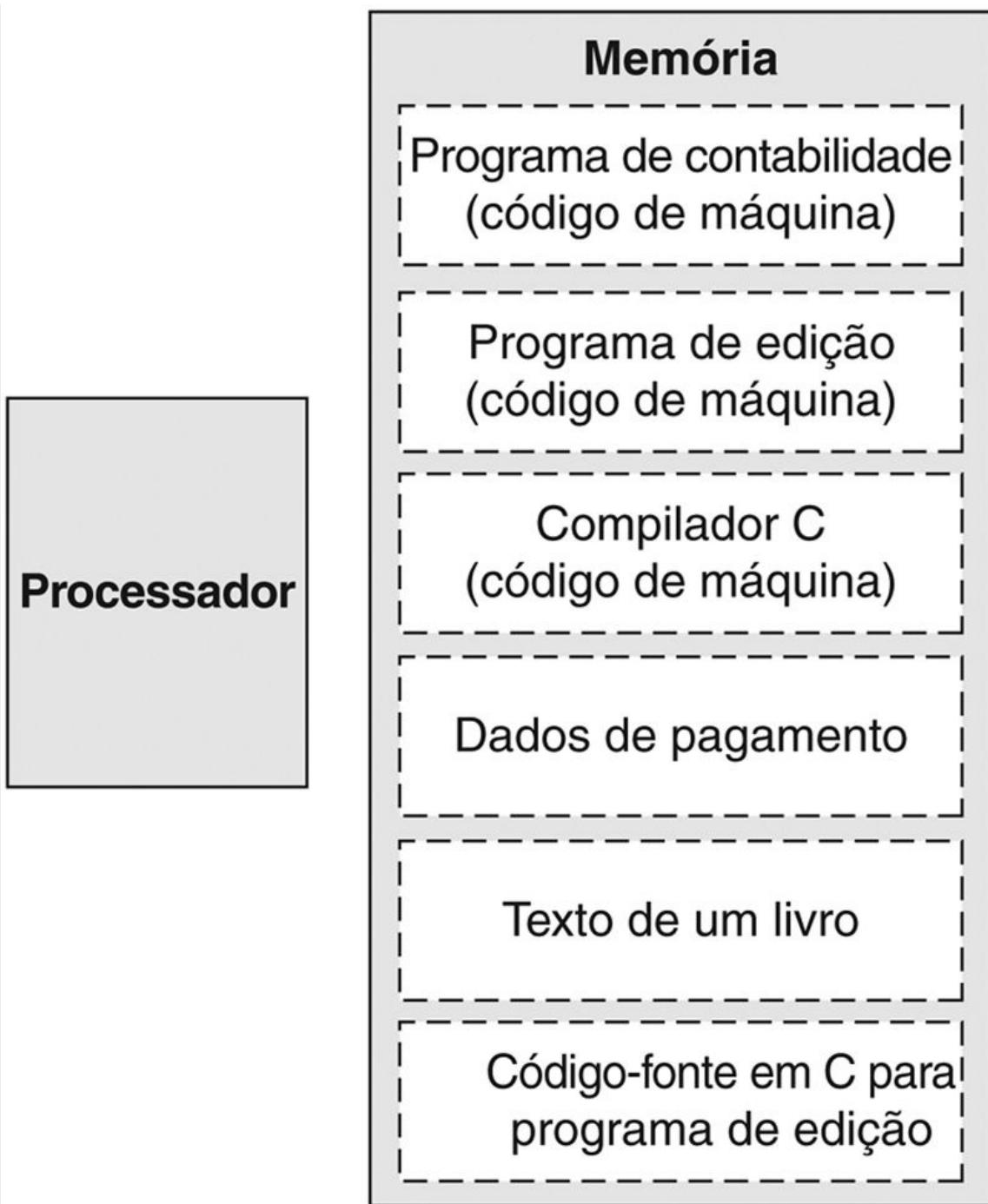


FIGURA 2.7 O conceito de programa armazenado.

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.

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.

Verifique você mesmo

Que instrução MIPS isto representa? Escolha entre uma das quatro opções a seguir.

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

1. sub \$s0, \$s1, \$s2
2. add \$s2, \$s0, \$s1
3. add \$s2, \$s1, \$s0
4. sub \$s2, \$s0, \$s1

2.6. Operações lógicas

“Ao contrário”, continuou Tweedledee, “se foi assim, poderia ser; e se assim fosse, seria; mas como não é, então não é. Isso é lógico.”

Lewis Carroll, Alice no país das maravilhas, 1865

Embora os primeiros computadores se concentrassem em palavras completas, logo ficou claro que era útil atuar sobre campos de bits dentro de uma palavra ou até mesmo sobre bits individuais. Examinar os caracteres dentro de uma palavra, cada um dos quais armazenados como 8 bits, é um exemplo dessa operação ([Seção 2.9](#)). Instruções foram acrescentadas às linguagens de programação e às arquiteturas de conjunto de instruções para simplificar, entre outras coisas, o empacotamento e o desempacotamento dos bits em words. Essas instruções são chamadas operações lógicas. A [Figura 2.8](#) mostra as operações lógicas em C, Java e MIPS.

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 2.8 Operadores lógicos em C e Java e suas instruções MIPS correspondentes.

MIPS implementa NOT usando um NOR com um operando sendo zero.

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 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 dual 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 valor original estava no

registrador \$t0 e o resultado deverá 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 10 (Registrador \$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 , assim como o deslocamento de um número decimal por i dígitos é equivalente a multiplicar por 10^i . Por exemplo, a instrução sll 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**, 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 tiver

AND

Uma operação lógica bit a bit com dois operandos, que calcula um 1 somente se houver um 1 em *ambos* os operandos.

0000 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 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 dual 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 0000 0011 1101 1100 0000_{bin}

OR

Uma operação lógica bit a bit com dois operandos, que calcula um 1 se houver um 1 em *qualquer um* dos operandos.

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. Usando nossa notação anterior, ele calcula \bar{X} .

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.

NOT

Uma operação lógica bit a bit com dois operandos, que calcula o NOT do OR dos dois operandos. Ou seja, ela calcula um 1 somente se houver um 0 em *ambos* os operandos.

Acompanhando o formato de três 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: $A \text{ NOR } 0 = \text{NOT}(A)$.

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 2.8](#) 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, a arquitetura do conjunto de instruções do MIPS não possui uma versão imediata do NOR.

Detalhamento

O conjunto de instruções MIPS completo também inclui exclusive or (XOR), que define o bit como 1 quando dois bits correspondentes diferem, e como 0 quando eles são iguais. C permite que *campos de bit* ou *campos* sejam definidos dentro das palavras, ambos permitindo que os objetos sejam empacotados com uma palavra e combinem com uma interface imposta externamente, como um dispositivo de E/S. Todos os campos precisam caber dentro de uma única palavra. Os campos recebem inteiros sem sinal que podem ser tão curtos quanto 1 bit. Os compiladores C inserem e extraem campos usando instruções lógicas no MIPS: and, or, sll e srl.

Detalhamento

AND lógico imediato e OR lógico imediato colocam 0s nos 16 bits superiores para formar uma constante de 32 bits, diferente do add imediato, que realiza extensão de sinal.

Verifique você mesmo

Que operações podem isolar um campo em uma word?

1. AND
2. Um deslocamento à esquerda seguido por um deslocamento à direita

2.7. Instruções para tomada de decisões

A utilidade de um computador automático se encontra na possibilidade de usar determinada sequência de instruções repetidamente; o número de vezes em que ela é repetida depende dos resultados do cálculo. Essa escolha pode depender do sinal de um número (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 chamada *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
```

Ela significa ir até a instrução chamada *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 requer a 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.

Compilando if-then-else em desvios condicionais

Exemplo

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;
```

Resposta

A Figura 2.9 é um fluxograma de como deve ser o código MIPS. A primeira expressão compara a igualdade, de modo que poderíamos querer desviar se os registradores forem a instrução de igual (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) e assim usamos o desvio se os registradores forem a instrução de *não igual* (bne):

```
bne $s3,$s4,Else # vá para Else se I ≠ j
```

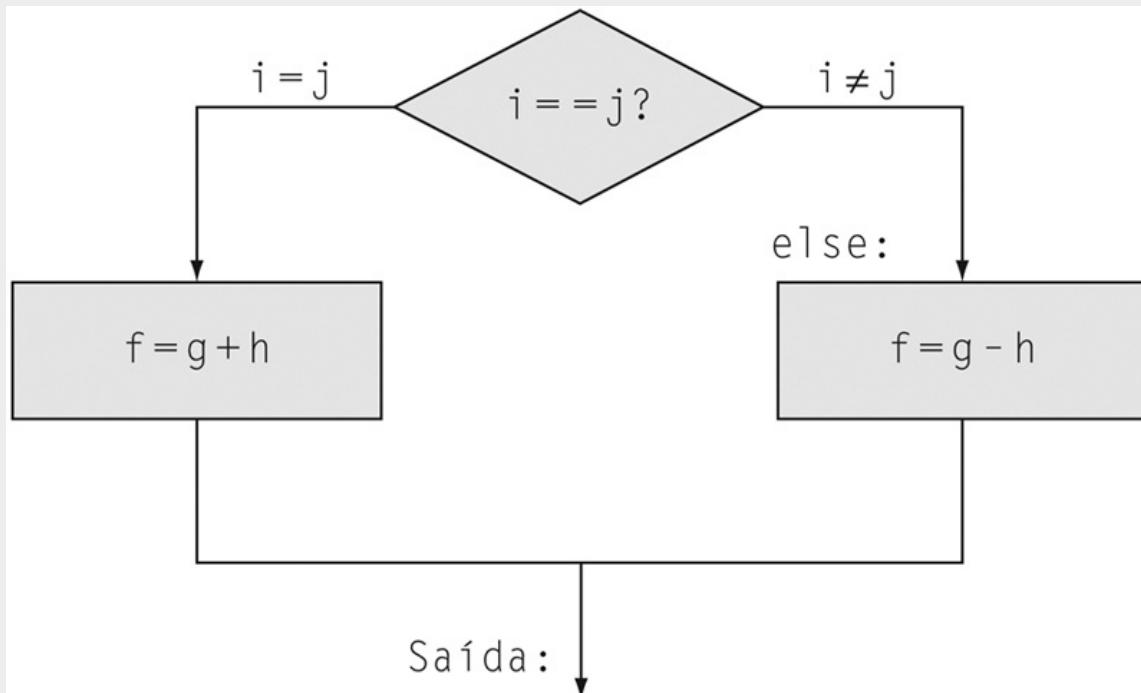


FIGURA 2.9 Ilustração das opções na instrução if acima.

A caixa da esquerda corresponde à parte *then* da instrução *if*, e a caixa da direita corresponde à parte *else*.

A próxima instrução de atribuição realiza uma única operação, e se todos os

operandos estiverem em registradores, 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 # vá 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:
```

Observe que o montador alivia o compilador e o programador assembly do trabalho de calcular endereços para os desvios, assim como evita o cálculo dos endereços de dados para loads e stores ([Seção 2.12](#)).

Interface hardware/software

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

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.

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 carregar *save[i]* em um registrador temporário. Antes que possamos carregar *save[i]* em um registrador temporário, precisamos ter seu endereço. Antes que possamos somar *i* à base do array *save* para formar o endereço, temos de multiplicar o índice *i* por 4, em razão do problema do endereçamento em bytes. Felizmente, podemos usar o deslocamento lógico à esquerda, pois o deslocamento em 2 bits à esquerda multiplica por 2^2 ou 4 (Seção “Operações Lógicas”, anteriormente neste capítulo). 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 = i * 4
```

Para obter o endereço de *save[i]*, temos de somar *\$t1* e a base do array *save* em *\$t6*:

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

Agora, podemos usar esse endereço para carregar `save[i]` em 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    # vá 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      # vá para o Loop  
Exit:
```

(Veja nos exercícios uma otimização para essa sequência.)

Interface hardware/software

Essas sequências de instruções que terminam em um desvio são tão fundamentais para a compilação que recebem seu próprio termo: um **bloco básico** é uma sequência 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.

bloco básico

Uma sequência de instruções sem desvios (exceto, possivelmente, no final) e sem destinos de desvio ou rótulos de desvio (exceto, possivelmente, no início).

O teste de igualdade ou desigualdade provavelmente é o teste mais comum, mas às vezes é útil ver se uma variável é menor do que outra. 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    # $t0 = 1 se $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 comuns nas comparações, de modo que existe uma versão imediata da instrução “set on less than”. 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
```

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.

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.

Interface hardware/software

As instruções de comparação precisam lidar com a dicotomia entre números com sinal e sem sinal. Às vezes, um padrão de bits com 1 no bit mais significativo representa um número negativo e, naturalmente, é menor que qualquer número positivo, que precisa ter um 0 no bit mais significativo. Com inteiros sem sinal, por outro lado, um 1 no bit mais significativo representa um número que é *maior* que qualquer um que comece com um 0. (Logo tiraremos proveito desse significado dual do bit mais significativo para reduzir o custo da verificação dos limites do array.)

MIPS oferece duas versões da comparação “set on less than” para tratar dessas alternativas. *Set on less than* (slt) e *set on less than immediate* (slti) funcionam 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* (sltiu).

Comparação de números com sinal e sem sinal

Exemplo

Suponha que o registrador \$s0 tenha o número binário

1111 1111 1111 1111 1111 1111 1111 1111 1111_{bin}

e que o registrador \$s1 tenha o número binário

0000 0000 0000 0000 0000 0000 0000 0001_{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_{dec} se for um inteiro e $4.294.967.295_{dec}$ se for um inteiro sem sinal. O valor no registrador $\$s1$ representa 1_{dec} em qualquer caso. O registrador $\$t0$ tem o valor 1, pois $-1_{dec} < 1_{dec}$, e o registrador $\$t1$ tem o valor 0, desde $4.294.967.295_{dec} > 1_{dec}$.

Tratar números com sinal como se fossem sem sinal é um modo de baixo custo para verificar se $0 \leq x < y$, que corresponde à verificação de índice fora de limite dos arrays. O principal é que os inteiros negativos na notação de complemento de 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 grande parte do número na segunda. Assim, uma comparação sem sinal de $x < y$ também verifica se x é negativo, bem como se x é menor que y .

Atalho para verificação de limites

Exemplo

Use este atalho para reduzir uma verificação de índice fora dos limites: salte para `IndexOutOfBoundsException` se $\$s1 \geq \$t2$ ou se $\$s1$ é negativo.

Resposta

O código de verificação só usa `sltu` para realizar as duas verificações:

```
sltu $t0,$s1,$t2 # $t0 = 0 se $s1 >= tamanho ou $s1 < 0  
beq $t0,$zero,IndexOutOfBoundsException #se erro, goto Error
```

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 sequência de testes condicionais, transformando a instrução *switch* em uma cadeia de instruções *if-then-else*.

Às vezes, as alternativas podem ser codificadas de forma mais eficiente como uma tabela de endereços de sequências de instruções alternativas, chamada **tabela de endereços de desvio** ou **tabela de desvio**, e o programa só precisa indexar na tabela e depois desviar para a sequência apropriada. A tabela de desvios é, então, apenas um array de palavras com endereços que correspondem aos rótulos no código. O programa carrega a entrada apropriada a partir da tabela de desvio para um registrador. Depois, ele precisa desviar usando o endereço no registrador. 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. Depois desvia para o endereço apropriado usando essa instrução. Veremos um uso ainda mais popular de *jr* na próxima seção.

tabela de endereços de desvio

Também chamada de **tabela de desvios**. Uma tabela de endereços de sequências de instruções alternativas.

Interface hardware/software

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

Detalhamento

Se você já ouviu falar em *delayed branches*, explicados no Capítulo 4, não se preocupe: o montador do MIPS os torna invisíveis ao programador assembly.

Verifique você mesmo

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

desequilíbrio? Por quê?

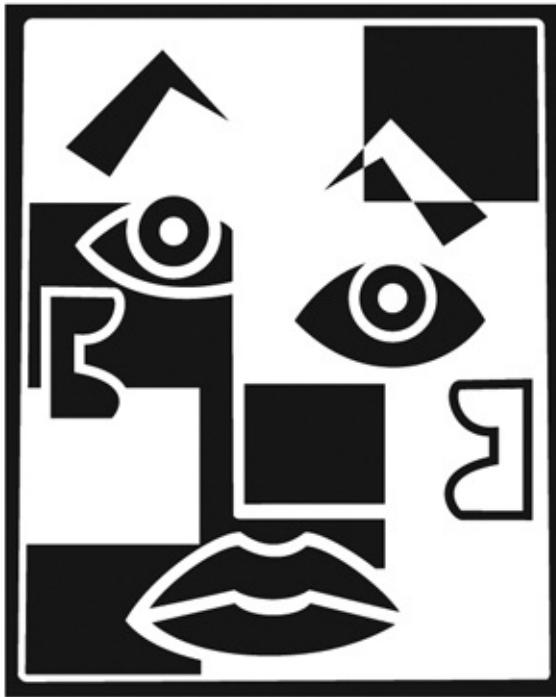
1. Mais instruções de decisão tornam o código mais fácil de ler e entender.
 2. Menos instruções de decisão simplificam a tarefa da camada inferior responsável pela execução.
 3. Mais instruções de decisão significam menos linhas de código, o que geralmente reduz o tempo de codificação.
 4. Mais instruções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

2.8. Suporte a procedimentos no hardware do computador

Um **procedimento** ou função é uma ferramenta que os programadores utilizam a fim de 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 interface entre o procedimento e o restante do programa e dos dados, pois eles passam valores e retornam resultados. Os procedimentos são uma forma de implementar **abstração** no software.

procedimento

Uma sub-rotina armazenada que realiza uma tarefa com base nos parâmetros que lhe são passados.



A B S T R A Ç Ã O

Você pode pensar em um procedimento como um espião que sai com um plano secreto, adquire recursos, realiza a tarefa, cobre seus rastros e depois retorna ao ponto de origem com o resultado desejado. Nada mais deverá ter sido alterado 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.

1. De modo semelhante, na execução de um procedimento, o programa precisa seguir estas seis etapas:
2. Colocar parâmetros em um lugar onde o procedimento possa acessá-los.
3. Transferir o controle para o procedimento.
4. Adquirir os recursos de armazenamento necessários para o procedimento.
5. Realizar a tarefa desejada.
6. Colocar o valor de retorno em um local onde o programa que o chamou possa acessá-lo.
7. 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. O software do

MIPS utiliza a seguinte convenção na alocação de seus 32 registradores:

- \$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
```

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).

A parte do *link* no nome da instrução significa que um endereço ou link é formado de modo a 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.

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.

Para dar suporte a tais situações, computadores como o MIPS utilizam uma instrução de *jump register* (jr), apresentada anteriormente para ajudar com as instruções case, 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.

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.

Num programa armazenado é necessário 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.

contador de programa (PC)

O registraror que contém o endereço da instrução sendo executada no programa.

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 desta 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”.

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, a fim de 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 palavra para cada registrador salvo ou restaurado. O software MIPS reserva o registrador 29 para o stack pointer, dando-lhe o nome óbvio \$sp. As pilhas são tão comuns 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**.

pilha (stack)

Uma estrutura de dados utilizada para armazenar os registradores, 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 os registradores devem ser armazenados ou onde os valores antigos dos registradores podem ser localizados.

push

Acrescentar elemento à pilha.

pop

Remover elemento da pilha.

Por motivos históricos, as pilhas “crescem” de endereços maiores para endereços menores. Essa convenção significa que você põe valores na 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

Exemplo

Vamos transformar o exemplo da Seção 2.2 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?

Resposta

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 Seção 2.2, que usa dois registradores temporários. Assim, precisamos salvar três registradores: `$s0`, `$t0` e `$t1`. “Empilhamos” os valores antigos, criando espaço para três palavras (12 bytes) na pilha e depois as armazenamos:

```

addi $sp, $sp, -12    # ajusta 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 2.10 mostra a pilha antes, durante e após a chamada do procedimento.

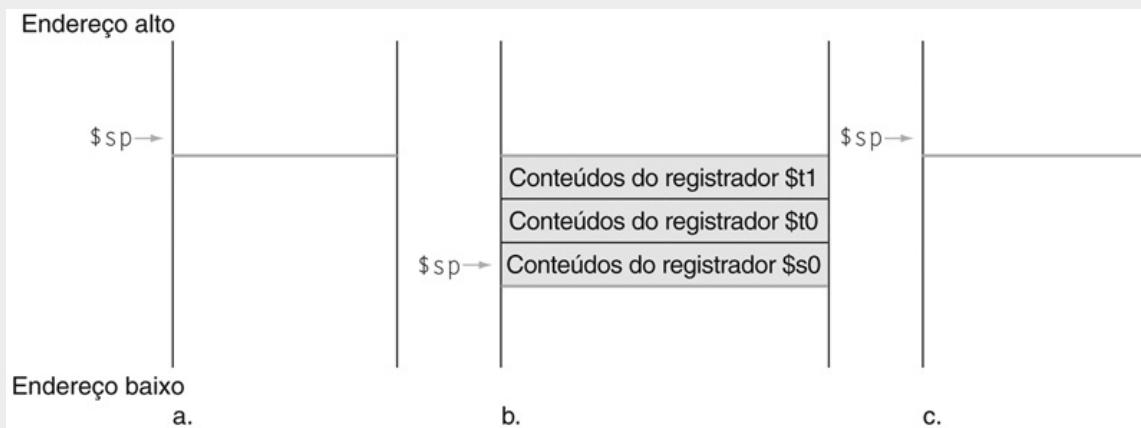


FIGURA 2.10 Os valores do stack pointer e a pilha (a) antes, (b) durante e (c) após a chamada do procedimento.

O stack pointer sempre aponta para o “topo” da pilha ou para a última palavra na pilha neste desenho.

As três instruções seguintes correspondem ao corpo do procedimento, que segue o exemplo da Seção 2.2:

```

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:

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

Essa convenção simples reduz o armazenamento de registradores. 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 deve 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雇用 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 registrador \$a0 e depois usando `jal A`. Depois, suponha que o procedimento A chame o procedimento B por meio de `jal 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

Exemplo

Vamos realizar um procedimento recursivo que calcula o fatorial:

```

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

```

Qual é o código assembly do MIPS?

Resposta

A variável de 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*:

```

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 *\$v0*. 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 quem chamou
```

Antes de retirar dois itens da pilha, poderíamos ter restaurado \$a0 e \$ra. Como \$a0 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 diminuído e depois fact é chamado novamente com o valor reduzido.

```
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 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 \$v0 recebe o produto do argumento antigo \$a0 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*. Alguns exemplos são inteiros e caracteres (Seção 2.9). 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

O registrador reservado para apontar para a área estática.

A [Figura 2.11](#) resume o que é preservado em uma chamada de procedimento. Observe que vários esquemas preservam a pilha, garantindo que o caller receberá os mesmos dados em um load da pilha que foram armazenados nela. 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.

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 2.11 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 serão 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 2.12](#) mostra o estado da pilha antes, durante e após a chamada de um procedimento.

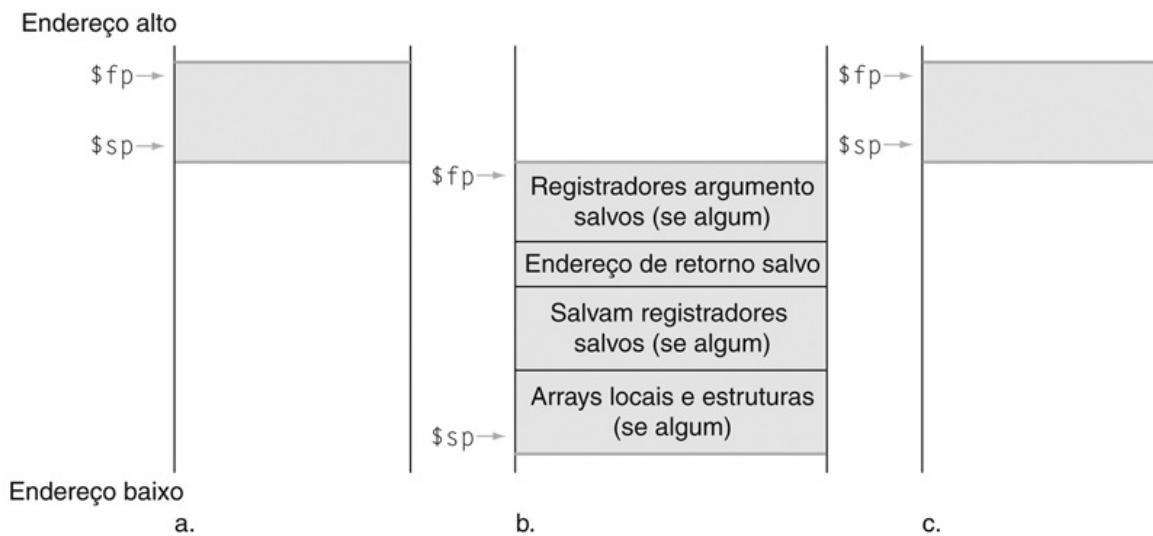


FIGURA 2.12 Ilustração da alocação de pilha (a) antes, (b) durante e (c) após a chamada de um procedimento.

O frame pointer (\$fp) aponta para a primeira palavra do frame, normalmente um registrador de argumento salvo, e o stack pointer (\$sp) aponta para o topo da pilha. A pilha é ajustada de modo a 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. Essa informação também aparece na Coluna 4 do Guia de Referência do MIPS, no final deste livro.

frame de procedimento

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

Alguns softwares MIPS utilizam o **frame pointer** (\$fp) a fim de 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 independentemente de o frame pointer explícito ser utilizado. Evitamos o \$fp impedindo mudanças no \$sp dentro de um procedimento: em nossos exemplos, a pilha é ajustada apenas na entrada e na saída do procedimento.

frame pointer

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

Alocando espaço para novos dados no heap

Além das variáveis automáticas que são locais aos procedimentos, os programadores de C precisam de espaço na memória para as variáveis globais e para estruturas de dados dinâmicas. A Figura 2.13 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 lar do código de máquina 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, correspondam 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.

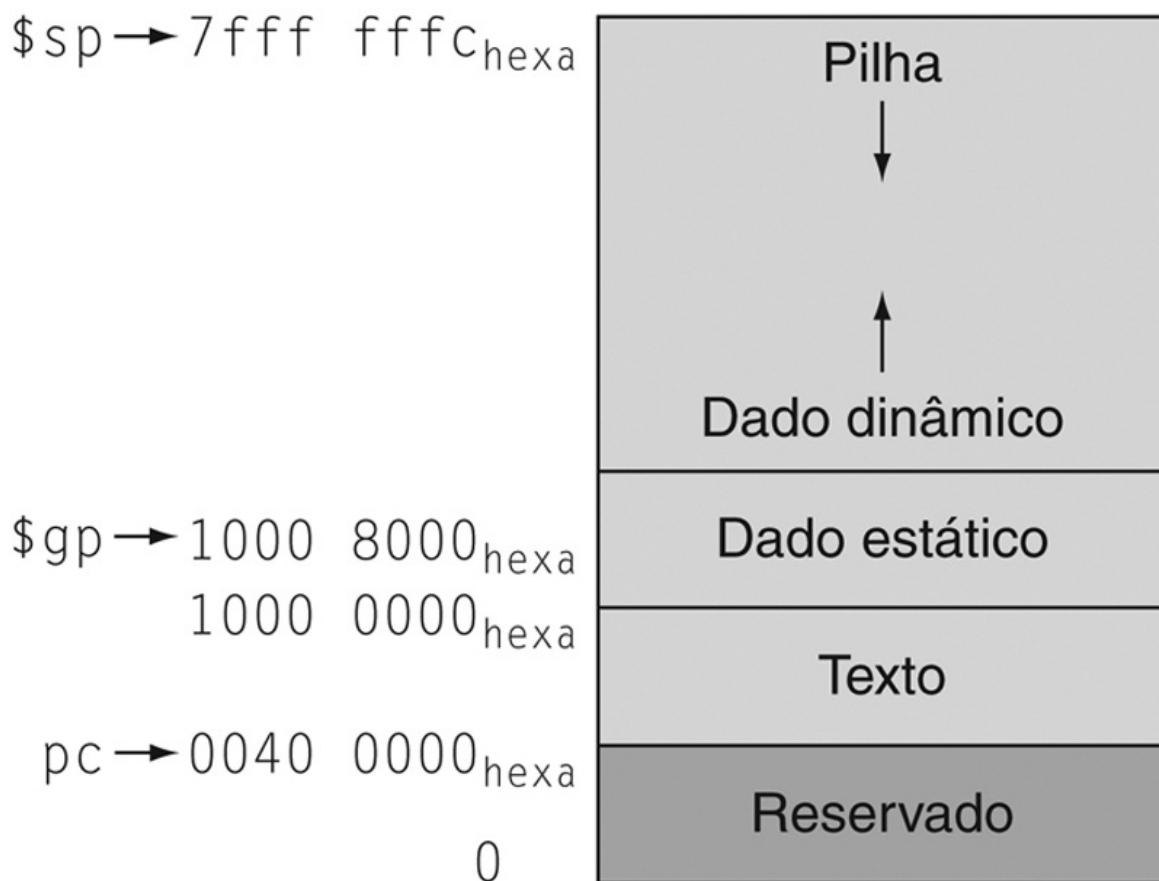


FIGURA 2.13 A alocação de memória do MIPS para programas e dados.

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 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. Essa informação também aparece na Coluna 4 do Guia de Referência do MIPS, no final deste livro.

segmento de texto

O segmento de um arquivo-objeto Unix que contém o código em linguagem de máquina para as rotinas do arquivo-fonte.

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 no heap 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 “vazamento de memória”, que, por fim, pode ocupar tanta memória que venha 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. Java utiliza alocação de memória e coleta de lixo automáticas, justamente para evitar esses bugs.

A [Figura 2.14](#) resume as convenções de registrador para o assembly do MIPS. Esta convenção é outro exemplo de tornar o **caso comum veloz**: a maioria dos procedimentos pode ser satisfeita com até 4 argumentos, 2 registradores para um valor de retorno, 8 registradores salvos e 10 registradores temporários sem sequer ir para a memória.

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 2.14 Convenções de registradores MIPS.

O registrador 1, chamado \$at, é reservado para o montador ([Seção 2.12](#)), e os registradores 26-27, chamados \$k0-\$k1, são reservados para o sistema operacional. Essa informação também aparece na Coluna 2 do Guia de Referência do MIPS, no final deste livro.

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 2.12, 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 do MIPS; ele trata o registrador 30 como outro registrador de valor salvo (\$a8).

Detalhamento

Alguns procedimentos recursivos podem ser implementados iterativamente sem o uso de recursão. A iteração pode melhorar significativamente o desempenho, removendo o overhead associado a chamadas de procedimento. Por exemplo, considere um procedimento usado para acumular uma soma:

```

int sum(int n, int acc) {
    if (n > 0)
        return sum(n - 1, acc + n);
    else
        return acc;
}

```

Considere a chamada de procedimento `sum(3,0)`. Isso resultará em chamadas recursivas a `sum(2,3)`, `sum(1,5)` e `sum(0,6)`, e depois o resultado 6 será retornado quatro vezes. Essa chamada recursiva de `sum` é conhecida como *tail call* e esse exemplo de uso da recursão tail pode ser implementado de modo muito eficiente (suponha que $\$a0 = n$ e $\$a1 = acc$):

```

sum: slti $t0, $a0, 1          # testa se n <= 0
      bne $t0, $zero, sum_exit # vai para sum_exit se n <= 0
      add$a1, $a1, $a0         # soma n a acc
      addi$a0, $a0, -1         # subtrai 1 de n
      j sum                   # vai para sum
sum_exit:
      add$v0, $a1, $zero       # retorna valor acc
      jr $ra                  # retorna ao caller

```

Verifique você mesmo

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

1. Os programadores C gerenciam os dados explicitamente, enquanto isso é automático em Java.
2. A linguagem C leva a mais problemas de ponteiro e vazamento de memória do que Java.

2.9. Comunicando-se com as pessoas

!(@ |= > (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)

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; o *American Standard Code for Information Interchange* (ASCII) é a representação que quase todos seguem. A [Figura 2.15](#) resume o código ASCII.

Valor ASCII	Sinal										
32	Espaç	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 2.15 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 caractere de tabulação e 13, um *carriage return*. Outro valor útil é 0 para null, o valor que a linguagem de programação C utiliza para marcar o final de uma string. Essa informação também aparece na Coluna 3 do Guia de Referência do MIPS, no final deste livro.

ASCII e números binários

Exemplo

Poderíamos representar números como strings de dígitos ASCII em vez de inteiros. Em quanto o armazenamento aumenta se o número 1 bilhão for representado em ASCII em vez de inteiro com 32 bits?

Resposta

Um bilhão é 1.000.000.000, de modo que ele precisaria de 10 dígitos ASCII, cada um com 8 bits de extensão. Assim, a expansão no armazenamento seria $(10 \times 8)/32$ ou 2,5. Além da expansão no armazenamento, o hardware para somar, subtrair, multiplicar e dividir esses números decimais é difícil e consumiria mais energia. Essas dificuldades explicam por que os profissionais de computação são levados a crer que o binário é natural e que o computador decimal ocasional é bizarro.

Diversas instruções podem extrair um byte de uma palavra, de modo que load word e store word são suficientes para transferir bytes e também palavras. Entretanto, em razão da popularidade do texto em alguns programas, o MIPS oferece instruções para mover bytes. *Load byte* (lb) lê um byte da memória, 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 sequência

```
lb $t0,0($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 caractere 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. (Como veremos, Java utiliza a

primeira opção.)

Compilando um procedimento de cópia de string, para demonstrar o uso de strings em C

Exemplo

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;

}
```

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 palavras, como nos exemplos anteriores.

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

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

Um cálculo de endereço semelhante coloca o endereço de $x[i]$ em $$t3$, e depois o carácter em $$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 carácter foi 0; ou seja, esse é o último carácter 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 caracter 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 palavra 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 2.14, 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. Por essa razão, em vez de pensar nos registradores \$t como sendo apenas para valores 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 2.16](#) 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 a fim de representar um caracter.

Latim	Malaiala	Apurahuano	Pontuação Geral
Grego	Sinhala	Khmer	Letras de Modificação de Espaço
Cirílico	Tailandês	Mongol	Símbolos de Moedas
Armênio	Laociano	Limbu	Marcas Diacríticas Combinadas
Hebraico	Tibetano	Tai Le	Marcas para Símbolos Combinadas
Árabe	Birmanês	Kangxi	Superescritos e Subescritos
Sírio	Georgiano	Hiragana	Formas de Números
Taana	Hangul	Katakana	Operadores Matemáticos
Devanágari	Etíope	Bopomofo	Símbolos Matemáticos Alfanuméricos
Bengali	Cherokee	Kanbun	Braille
Gurmukhi	Silábico Unificado dos Aborígenes Canadenses	Shavian	Reconhecimento Ótico de Caracteres
Gujarati	Ogham	Osmanya	Símbolos Musicais Bizantinos
Oriá	Runic	Silabário Cipriota	Símbolos Musicais
Tamil	Tagalo	Símbolos Tai Xuan Jing	Setas
Telugu	Hanunoo	Símbolos I Ching	Elementos de Bloco
Canarês	Buhid	Números Egeus	Formas Geométricas

FIGURA 2.16 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, Grego começa em 0370_{hexa} e Cirílico 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 multilíngues 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 ou 32 bits para os outros caracteres. UTF-32 utiliza 32 bits por caracter. 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. Assim como load byte, *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 halfword unsigned* (lhu) trabalha com inteiros sem sinal. Assim, lhu é o mais comum dos dois. 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 sequência

```
lhu $t0,0($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, o Java inclui uma palavra que indica o tamanho da string, semelhante aos arrays Java.

Detalhamento

O software do MIPS tenta manter a pilha alinhada em endereços de palavra, 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 palavra, e uma variável `string` ou array de shorts em Java agrupará 2 halfwords por palavra.

Detalhamento

Refletindo a natureza internacional da Web, a maioria das páginas web hoje utiliza Unicode ao invés de ASCII.

Verifique você mesmo

- I. 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. As strings em 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.
- II. Que tipo de variável que pode conter $1.000.000.000_{\text{dec}}$ ocupa mais espaço na memória?
 1. `int` em C

- 2. string em C
- 3. string em Java

2.10. 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 2.17](#) mostra a operação de *lui*.

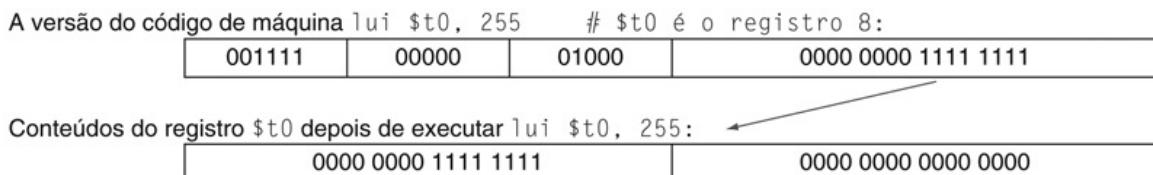


FIGURA 2.17 O efeito da instrução *lui*.

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.

Carregando uma constante de 32 bits

Exemplo

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
```

Resposta

Primeiro, carregaríamos 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
```

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
```

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 (assembler temporary), 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 (Seção 2.12). 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.

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 palavra. O operador *lógico ou imediato*, da Seção 2.6, carrega 0s nos 16 bits superiores e, portanto, é usado 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):

2	10000
6 bits	26 bits

em que 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:

Contador de programa = Registrador + 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. Eles 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 SPEC vão para locais a menos de 16 instruções de distância. Como o *contador de programa* (PC) contém o endereço da instrução atual, podemos desviar dentro de $\pm 2^{15}$ palavras da instrução atual se usarmos o PC como o registrador a ser somado ao endereço. Quase todos os loops e as instruções *if* são muito menores do que 2^{16} palavras, de modo que o PC é a opção ideal.

Essa forma de endereçamento de desvio é denominada **endereçamento relativo ao PC**. Conforme veremos no [Capítulo 4](#), é conveniente que o hardware incremente o PC desde cedo, a fim de 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). Este é outro exemplo de tornar o **caso comum veloz**, que neste caso significa endereçar instruções próximas.

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.



CASO COMUM VELOZ

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 *palavras* 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 palavra, e não um endereço relativo a byte. De modo semelhante, o campo de 26 bits nas instruções de *jump* também é um endereço de palavra, significando que representa um endereço de byte com 28 bits.

Detalhamento

Como o contador de programa (PC) utiliza 32 bits, 4 bits precisam vir de outro lugar para os jumps. 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 2.12) 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 Seção 2.7 foi compilado para este código em assembly do MIPS:

```
Loop:s11    $t1,$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] ≠ k
            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?

Resposta

As instruções montadas e seus endereços são:

80000	0	0	19	9	2	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 palavras sequenciais diferem em 4, a quantidade de bytes em uma palavra. A instrução `bne` na quarta linha acrescenta 2 palavras 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

A maioria dos desvios condicionais é feita para um local nas proximidades, mas, ocasionalmente, eles se desviam para um ponto mais distante 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 se irá pular o jump.

Desviando para um lugar mais distante

Exemplo

Dado um desvio em que o registrador `$s0` é igual ao registrador `$s1`,

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

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

muito maior.

Resposta

Estas instruções substituem o desvio condicional com endereço curto:

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

Resumo dos modos de endereçamento no MIPS

As diversas formas de endereçamento geralmente são denominadas **modos de endereçamento**. A [Figura 2.18](#) mostra como os operandos são identificados para cada modo de endereçamento. Os modos de endereçamento do MIPS são os seguintes:

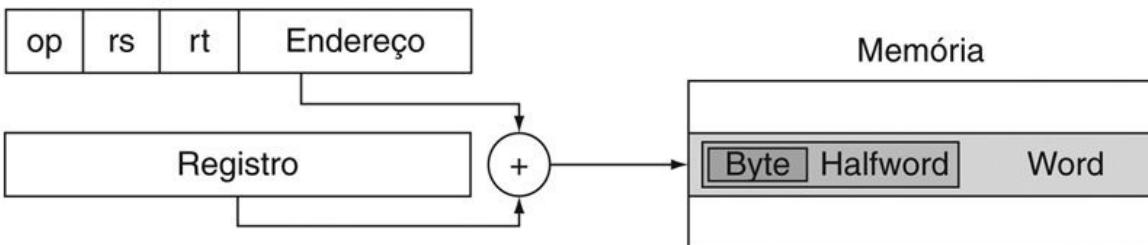
1. Endereçamento imediato



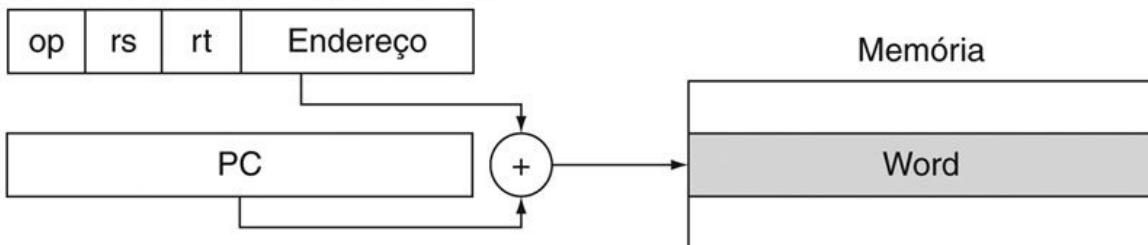
2. Endereçamento em registrador



3. Endereçamento de base



4. Endereçamento relativo ao PC



5. Endereçamento pseudodireto

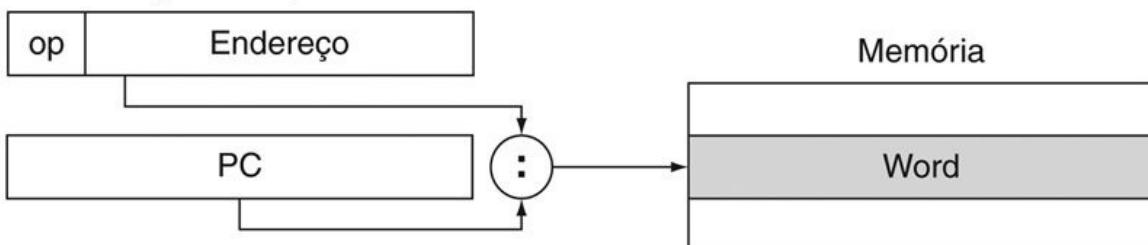


FIGURA 2.18 Ilustração dos cinco modos de endereçamento do MIPS.

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 palavras. 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 ao PC, e o modo 5 concatenando um endereço de 26 bits deslocado à esquerda em 2 bits com os 4 bits superiores do PC. 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`).

1. *Endereçamento imediato*, em que o operando é uma constante dentro da própria instrução.
2. *Endereçamento em registrador*, no qual o operando é um registrador.
3. *Endereçamento de base ou deslocamento*, em que o operando está no local de memória cujo endereço é a soma de um registrador e uma constante na instrução.
4. *Endereçamento relativo ao PC*, no qual o endereço de desvio é a soma do PC e uma constante na instrução.
5. *Endereçamento pseudodireto*, em que o endereço de jump são os 26 bits da instrução concatenados com os bits mais altos do PC.

modo de endereçamento

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

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 ([Seção 2.17](#)). 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.

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 2.19](#) 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	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
31-29								
0(000)	formato R	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	F1Pt						
3(011)								
4(100)	load byte	load half	lw	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	sw	store word			swr	
6(110)	load linked word	lwcl						
7(111)	store cond. word	swcl						

op(31:26)=010000 (TLB), 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)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

op(31:26)=000000 (format R), 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)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set 1.t.	set 1.t. unsigned				
6(110)								
7(111)								

FIGURA 2.19 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 do floating point na linha 2, coluna 1, é definido na [Figura 3.18](#), no [Capítulo 3](#). Bltz/gez é o opcode para quatro instruções encontradas no Apêndice A: bltz, bgez, bltzal e bgezal. Este capítulo descreve as instruções indicadas com nome completo usando destaque, enquanto o [Capítulo 3](#) descreve as instruções indicadas em mnemônicos do mesmo jeito. 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?

00af8020hexa

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 2.19, 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 2.20:

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

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 2.20 Formatos das instruções do MIPS.

A parte inferior da Figura 2.19 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 2.14 diz que esses números representam os registradores \$a1, \$a7 e \$s0. Agora, podemos mostrar a instrução assembly:

```
add $s0, $a1, $t7
```

A [Figura 2.20](#) mostra todos os formatos de instrução do MIPS. A [Figura 2.1](#) mostrou a linguagem assembly do MIPS revelada neste capítulo; a outra parte ainda oculta das instruções MIPS trata principalmente de aritmética e números reais, que serão abordados no próximo capítulo.

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
- 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

4. Endereços desde cerca de 128M antes do desvio até cerca de 128M depois
5. Qualquer lugar dentro de um bloco de 64M endereços, em que o PC fornece os 6 bits mais altos
6. Qualquer lugar dentro de um bloco de 256M endereços, em que o PC fornece os 4 bits mais altos

III. 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.

2.11. Paralelismo e instruções: Sincronização

A **execução paralela** é mais fácil quando as tarefas são independentes, mas frequentemente elas precisam cooperar. A cooperação normalmente significa que algumas tarefas estão escrevendo novos valores que outras precisam ler. Para saber quando uma tarefa terminou de escrever, de modo que é seguro que outra tarefa leia, elas precisam de sincronização. Se elas não estiverem sincronizadas, haverá um perigo de **data race**, em que os resultados do programa podem mudar, dependendo de como os eventos ocorram.



PARALELISMO

data race

Dois acessos à memória formam uma data race se eles forem de threads diferentes para o mesmo local, pelo menos um é de escrita, e eles ocorrem um após o outro.

Por exemplo, lembre-se da analogia citada no [Capítulo 1](#) dos oito repórteres escrevendo um artigo. Suponha que um repórter precise ler todas as seções anteriores antes de escrever uma conclusão. Logo, temos de saber quando os

outros repórteres terminaram suas seções, de modo que ele não se preocupe se será alterado depois disso. Ou seja, é melhor que eles sincronizem a escrita e leitura de cada seção, para que a conclusão seja coerente com o que é impresso nas seções anteriores.

Na computação, os mecanismos de sincronização normalmente estão embutidos em rotinas de software em nível de usuário que contam com as instruções de sincronização fornecidas pelo hardware. Nesta seção, focalizamos a implementação das operações de sincronização *lock* e *unlock*. Lock e unlock podem ser usados facilmente para criar regiões nas quais apenas um único processador possa operar, algo chamado *exclusão mútua*, além de implementar mecanismos de sincronização mais complexos.

A habilidade fundamental que exigimos para implementar a sincronização em um multiprocessador é um conjunto de primitivos de hardware com a capacidade de ler e modificar um local de memória *atomicamente*. Ou seja, nada mais pode se interpor entre a leitura e a escrita do local da memória. Sem essa capacidade, o custo da montagem de primitivos de sincronização básicos será muito alto e aumentará à medida que crescer a quantidade de processadores.

Existem diversas formulações alternativas das primitivas de hardware básicas, todas oferecendo a capacidade de ler e modificar um local atomicamente, junto com algum modo de dizer se a leitura e escrita foram realizadas atomicamente. Em geral, os arquitetos não esperam que os usuários empreguem as primitivas de hardware básicas, mas em vez disso esperam que as primitivas sejam usadas pelos programadores de sistemas para montar uma biblioteca de sincronização, um processo que normalmente é complexo e intrincado.

Vamos começar com uma primitiva de hardware desse tipo, mostrando como ela pode ser usada para montar uma primitiva de sincronização básica. Uma operação típica para a montagem de operações de sincronização é a *troca atômica* ou *swap atômico*, que troca um valor em um registrador por um valor na memória.

Para ver como usar isso a fim de montar uma primitiva de sincronização básica, suponha que queremos montar um bloqueio simples, em que o valor 0 é usado para indicar que o bloqueio está livre e 1 é usado para indicar que o bloqueio está indisponível. Um processador tenta definir o bloqueio realizando uma troca de 1, que está em um registrador, com o endereço de memória correspondendo ao bloqueio. O valor retornado da instrução de troca é 1 se algum outro processador já tiver solicitado acesso, e 0 em caso contrário. No segundo caso, o valor também é trocado para 1, impedindo que qualquer outra

troca em outro processador também recupere um 0.

Por exemplo, considere dois processadores que tentam cada um realizar a troca simultaneamente; essa race é interrompida, pois exatamente um dos processadores realizará a troca primeiro, retornando 0, e o segundo processador retornará 1 quando realizar a troca. A chave para o uso da primitiva de troca para implementar a sincronização é que a operação seja atômica: a troca é indivisível, e duas trocas simultâneas serão ordenadas pelo hardware. É impossível que dois processadores tentando definir a variável de sincronização dessa maneira pensem que definiram a variável simultaneamente.

A implementação de uma única operação de memória atômica apresenta alguns desafios no projeto do processador, pois requer uma leitura e uma escrita na memória em uma única instrução ininterrupta.

Uma alternativa é ter um par de instruções em que a segunda instrução retorna um valor, mostrando se o par de instruções foi executado como se fosse atômico. O par de instruções é efetivamente atômico se parecer que todas as outras operações executadas por qualquer processador ocorreram antes ou depois do par. Assim, quando um par de instruções é efetivamente atômico, nenhum outro processador pode alterar o valor entre o par de instruções.

No MIPS, esse par de instruções inclui um load especial, chamado *load vinculado*, e um store especial, chamado *store condicional*. Essas instruções são usadas em sequência: se o conteúdo do local de memória especificado pelo load vinculado for alterado antes que ocorra o store condicional para o mesmo endereço, então o store condicional falha. O store condicional é definido para armazenar o valor de um registrador na memória e alterar o valor desse registrador para 1 se tiver sucesso e para 0 se falhar. Como o load vinculado retorna o valor inicial, e o store condicional retorna 1 somente se tiver sucesso, a sequência a seguir implementa uma troca atômica no local de memória especificado pelo conteúdo de \$s1:

```
again: addi $t0,$zero,1      ;copia valor da troca
      ll    $t1,0($s1)        ;load linked
      sc    $t0,0($s1)        ;store condicional
      beq   $t0,$zero,again   ;desvia se store falhar
      add   $s4,$zero,$t1      ;coloca valor de load em $s4
```

Sempre que um processador intervém e modifica o valor na memória entre as instruções `ll` e `sc`, o script retorna 0 em `$t0`, fazendo com que a sequência de código tente novamente. Ao final dessa sequência, o conteúdo de `$s4` e o local de memória especificado por `$s1` foram trocados atomicamente.

Detalhamento

Embora fosse apresentada para sincronização de multiprocessador, a troca atômica também é útil para o sistema operacional lidar com múltiplos processos em um único processador. A fim de garantir que nada interfira em um único processador, o store condicional também falha se o processador realizar uma troca de contexto entre as duas instruções (Capítulo 5).

Uma vantagem do mecanismo de load vinculado/store condicional é que ele pode ser usado para montar outras primitivas de sincronização, como *compare and swap atômicos* ou *fetch-and-increment atômicos*, que são usados em alguns modelos de programação paralela. Estes envolvem mais instruções entre o `ll` e o `sc`.

Como o store condicional falhará após outro store atraído ao endereço do load vinculado ou em qualquer exceção, deve-se ter o cuidado na escolha de quais instruções são inseridas entre as duas instruções. Em particular, somente instruções registrador-registrador podem ser permitidas com segurança; caso contrário, é possível criar situações de impasse, em que o processador nunca possa completar o `sc`, em consequência das faltas de páginas repetidas. Além disso, o número de instruções entre o load vinculado e o store condicional deve ser pequeno, para minimizar a probabilidade de que um evento não relacionado ou um processador concorrente faça com que o store condicional falhe com frequência.

Verifique você mesmo

Quando você usará primitivas como load vinculado e store condicional?

1. Quando threads em cooperação de um programa paralelo precisarem ser sincronizados para obter um comportamento apropriado para leitura e escrita de dados compartilhados
2. Quando processos em cooperação em um processador precisarem ser sincronizados para a leitura e escrita de dados compartilhados

2.12. 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 2.21](#) mostra a hierarquia de tradução. Alguns sistemas combinam estas 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.

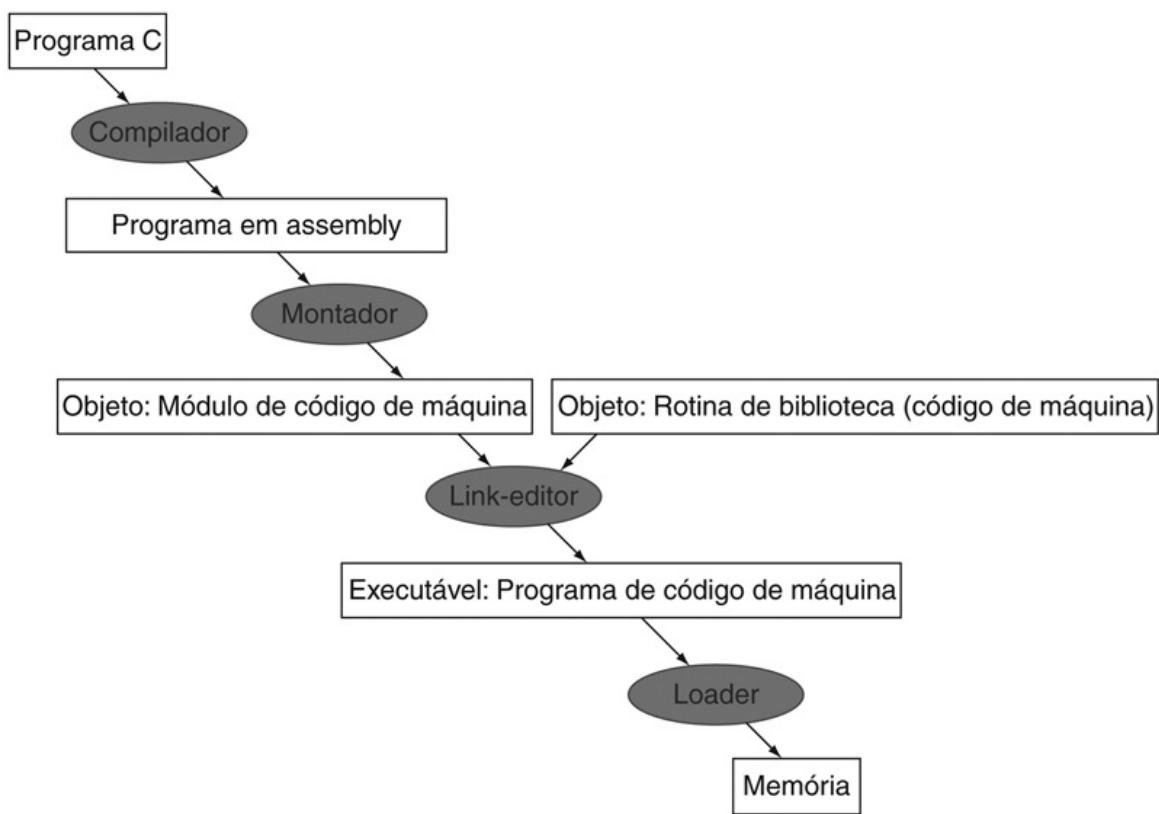


FIGURA 2.21 Uma hierarquia de tradução para a linguagem C.

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, de modo a 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. A fim de 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.a`, as rotinas de biblioteca link-editadas dinamicamente são `x.so`, e os arquivos executáveis, como padrão, são chamados `a.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 a linguagem assembly, de modo que a produtividade do programador é muito mais alta.

Em 1975, muitos sistemas operacionais e montadores foram escritos em **linguagem assembly**, pois as memórias eram pequenas e os compiladores eram ineficientes. O aumento de um milhão de vezes na capacidade de memória em um único chip de DRAM reduziu os problemas com tamanho de programas 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.

linguagem assembly

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

Montador

Como a linguagem assembly é a interface com o software de nível superior, o montador (ou *assembler*) 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 **pseudoinstruções**.

pseudoinstrução

Uma variação comum das instruções em assembly, normalmente tratada como se fosse uma instrução propriamente dita.

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 em linguagem 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 registrador $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 segundo exemplo da [Seção 2.5](#). 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 em linguagem 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 de modo a 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 2.1 e 2.19](#).

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. Montadores MIPS

utilizam hexadecimal.

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 a fim de 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.

tabela de símbolos

Uma tabela que combina nomes de rótulos aos endereços das palavras 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. Veja a [Figura 2.13](#).)
- As *informações de relocação* identificam instruções e palavras 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 apanha todos os programas em linguagem de máquina montados independentes e os “remenda”.

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.

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 2.13](#), na [Seção 2.8](#), 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* a fim de 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.

arquivo executável

Um programa funcional no formato de um arquivo-objeto, que não contém referências não resolvidas. Ele pode conter tabelas de símbolos e informações de depuração. Um “executável desrido” não contém essa informação. As informações de relocação podem ser incluídas para o loader.

Link-edição de arquivos-objeto

Exemplo

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 comprehensível; 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.

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 da instrução	Dependência

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 \$al, 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	—	

Resposta

O procedimento A precisa encontrar o endereço para a variável cujo rótulo é X, a fim de 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 2.13, 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}.

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)

A Figura 2.13 também mostra que o segmento de texto começa no endereço 40 0000_{hexa} e o segmento de dados no 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 é 100_{hexa} bytes e seus dados 20_{hexa} bytes, então o começo do endereço do texto do procedimento B é 40 0100_{hexa}, e seus dados começam em 1000 0020_{hexa}.

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 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 2.13 mostra que \$gp é inicializado com 1000 8000_{hexa}. Para obter o endereço 1000 0000_{hexa} (o endereço da palavra x), colocamos 8000_{hexa} no campo de endereço da instrução lw, no endereço 40 0000_{hexa}. 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 palavra Y).

Detalhamento

Lembre-se de que as instruções MIPS são alinhadas na palavra, de modo que ja1 remove os dois bits da direita para aumentar a faixa de endereços da instrução. Assim, ele usa 26 bits para criar um endereço de byte de 28 bits. Logo, o endereço real nos 26 bits inferiores da instrução ja1 neste exemplo é $10\ 0040_{hexa}$, em vez de $40\ 0100_{hexa}$.

Loader

Agora que o arquivo executável está no disco, o sistema operacional o lê para a memória e o inicia. O **loader** 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 inicialização, que copia os parâmetros para os registradores de argumento e chama a rotina principal do programa.
Quando a rotina principal retorna, a rotina de inicialização termina o programa com uma chamada exit do sistema.

loader

Um programa de sistema que coloca o programa-objeto na memória principal, de modo que esteja pronto para ser executado.

As Seções A.3 e A.4 no Apêndice A descrevem os link-editores e os loaders com mais detalhes.

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 todas as rotinas na biblioteca que são chamadas de qualquer lugar no executável, mesmo que essas chamadas não sejam executadas. 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 Linked 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.

Dynamically Linked Libraries (DLLs)

Rotinas de bit que são vinculadas a um programa durante a execução.

A desvantagem da versão inicial das DLLs era que elas ainda link-editavam 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 muitas inovações em nosso campo, esse truque conta com um certo nível de indireção. A [Figura 2.22](#) 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.

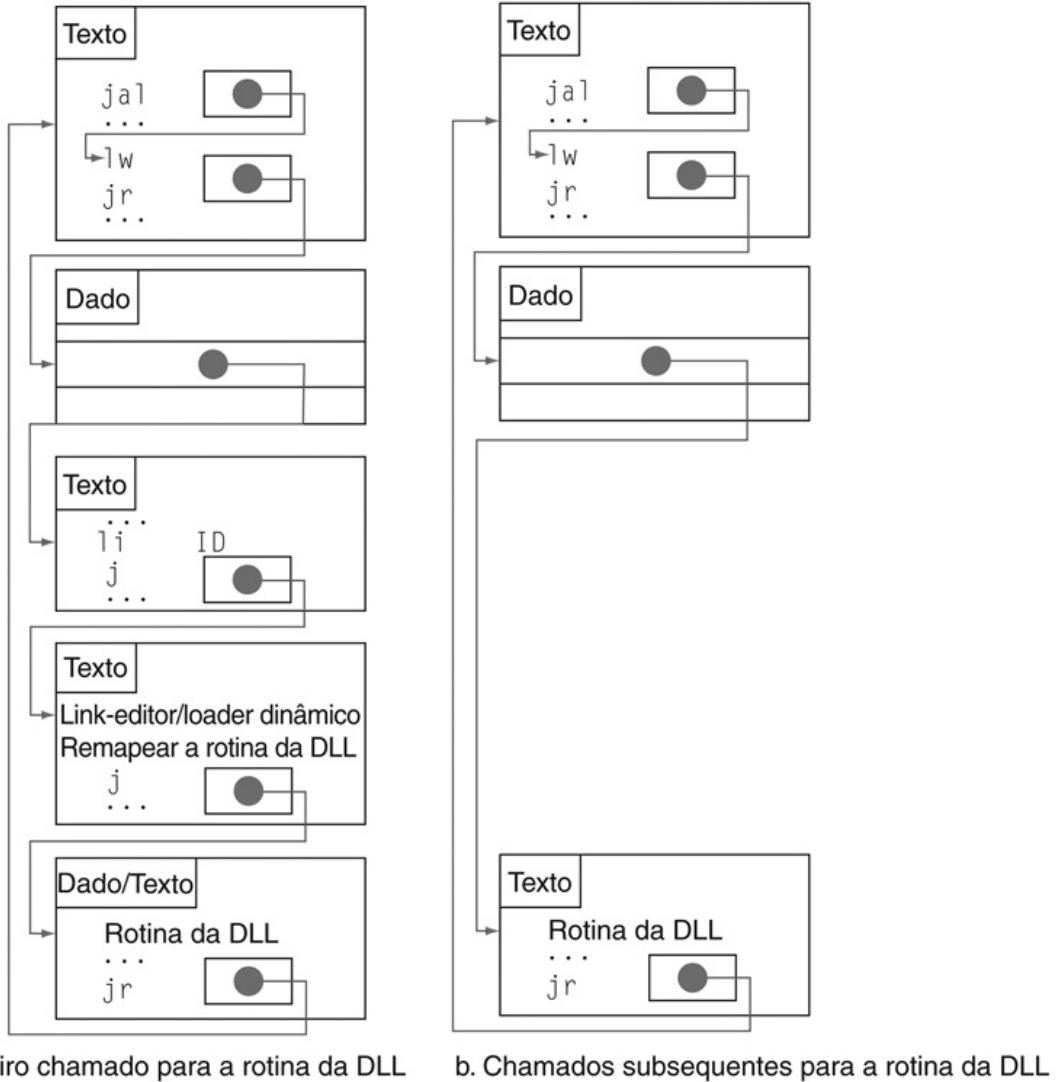


FIGURA 2.22 DLL por meio da link-edição de procedimento tardio.

- (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 subsequentes.
Conforme veremos no [Capítulo 5](#), 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, de modo a apontar para essa rotina. Depois, ele desvia para ela. Quando

a rotina termina, ele retorna ao local de chamada original. Depois disso, a chamada para rotina de biblioteca desvia indiretamente 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 a 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, o Java foi inventado 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 2.23](#) 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.

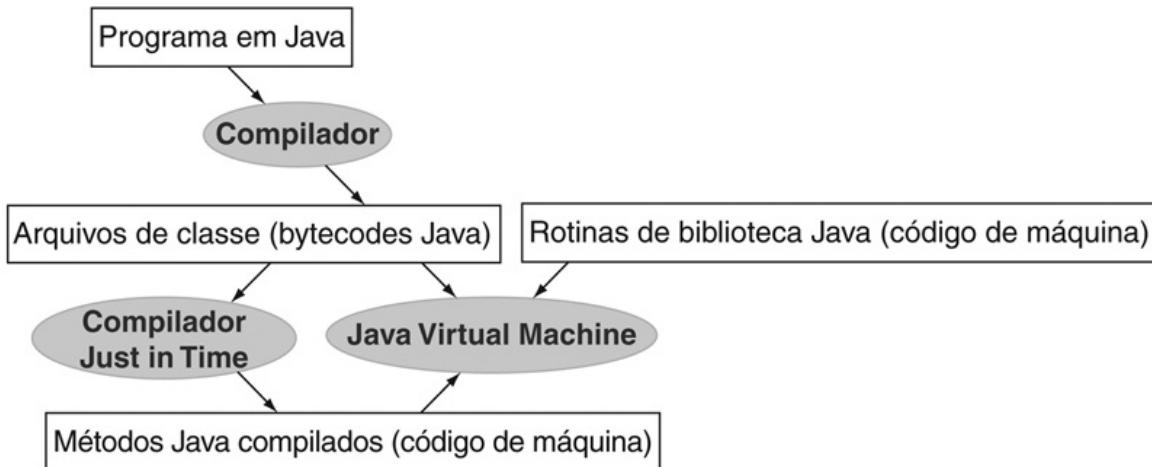


FIGURA 2.23 Uma hierarquia de tradução para Java.

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 Java Virtual Machine (JVM — 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 Just In Time (JIT), que compila os métodos seletivamente para a linguagem nativa da máquina em que está executando.

bytecode Java

Instruções de um conjunto de instruções projetado para interpretar programas em Java.

Um interpretador Java, chamado **Java Virtual Machine** (JVM), pode executar os bytecodes Java. Um interpretador é um programa que simula um conjunto de instruções. 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.

Java Virtual Machine (JVM)

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 centenas de 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.

A fim de 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.

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.

À medida que os computadores ficam mais rápidos, de modo que os compiladores possam fazer mais, e os pesquisadores inventam meios melhores de compilar Java durante a execução, a lacuna de desempenho entre Java e C ou C ++ está se fechando.

Verifique você mesmo

Qual das vantagens de um interpretador em relação a um tradutor você acredita que tenha sido mais importante para os criadores do Java?

1. Facilidade de escrita de um interpretador
2. Melhores mensagens de erro

3. Código-objeto menor
4. Independência de máquina

2.13. Um exemplo de ordenação em C para juntar tudo isso

Um perigo de mostrar o código em assembly em partes é que você não terá ideia 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 outro para ordená-los.

O procedimento swap

Vamos começar com o código para o procedimento swap na [Figura 2.24](#). 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.

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

FIGURA 2.24 Um procedimento em C que troca o conteúdo de duas posições de memória.

Esta subseção utiliza esse procedimento em um exemplo de ordenação.

Esta seção descreve o procedimento swap nessas três partes, concluindo com a junção de todas as partes.

De registradores para swap

Como mencionamos anteriormente na [Seção 2.8](#), 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 (Seção “Procedimentos aninhados”). Essa alocação de registradores corresponde às declarações de variável na primeira parte do procedimento swap da [Figura 2.24](#).

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 palavras sequenciais 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 por meio de um deslocamento à esquerda por 2:

```

sll      $t1, $a1,2          # reg. $t1 = k * 4
add      $t1, $a0,$t1        # reg. $t1 = v + (k * 4)
                                # reg. $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)        # reg. $t0 (temp) = v[k]
lw       $t2, 4($t1)        # reg. $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] = reg. $t2
sw       $t0, 4($t1)        # v[k + 1] = reg. $t0 (temp)

```

Até agora, alocamos registradores e escrevemos o código de modo a 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. A fim de facilitar o acompanhamento, identificamos na [Figura 2.25](#) cada bloco de código com sua finalidade no procedimento.

Corpo do procedimento	
<pre> swap: sll \$t1, \$a1, 2 # reg \$t1 = k * 4 add \$t1, \$a0, \$t1 # reg \$t1 = v + (k * 4) lw \$t0, 0(\$t1) # reg \$t1 tem o endereço de v[k] lw \$t2, 4(\$t1) # reg \$t0 (temp) = v[k] sw \$t2, 0(\$t1) # refere-se ao próximo elemento de v sw \$t0, 4(\$t1) # v[k] = reg \$t2 # v[k+1] = reg \$t0 (temp) </pre>	
Retorno do procedimento	
	<pre> jr \$ra # retorna à rotina que chamou </pre>

FIGURA 2.25 Código assembly do MIPS do procedimento swap na [Figura 2.24](#).

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 bolha ou trocas, que é uma das mais simples, mas não a mais rápida. A [Figura 2.26](#) 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 2.26 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 desvendar 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 necessário 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 pseudoinstrução fornecida pelo montador para a conveniência do programador em assembly; ver seção “Montador”, anteriormente, neste capítulo.) 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` para 1 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:

```
for1tst:slt$t0, $s0, $a1      # reg. $t0=0 se $s0 ≥ $a1 (i≥n)
      beq    $t0, $zero,exit1   # vá para exit1 se $s0≥$a1 (i≥n)
```

O final do loop só retorna 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 ≥ $a1 (i ≥ n)
    beq    $t0, $zero,exit1 # vai para exit1 se $s0 ≥ $a1 (i ≥ n)
        ...
        (corpo do primeiro loop for)
        ...
addi    $s0, $s0, 1      # i += 1
    j forltst            # salta para teste do loop externo
exit1:
```

Voilà! (Os exercícios exploram 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 decremento 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 palavra 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 ≥ $t3
    beq      $t0, $zero, exit2 # vai para exit2 se $t4 ≥ $t3

```

O final do loop retorna para o teste do loop interno:

```

j      for2tst          # salta para teste 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:slti $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 ≥ $t3
beq $t0, $zero, exit2 # vai para exit2 se $t4 ≥ $t3
    ...
    (corpo do segundo loop for)
    ...
addi $s1, $s1, -1      # j -= 1
j  for2tst            # salta para 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 2.27](#), 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 exemplo, nove linhas do procedimento `sort` em C tornaram-se 35 em assembly do MIPS.

Salvando registradores		
	sort:	<pre> addi \$sp,\$sp, -20 # cria espaço na pilha para 5 registradore 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 </pre>
Corpo do procedimento		
Move parâmetros		<pre> move \$\$s2, \$a0 ## copia parâmetro \$a0 para \$\$s2 (salva \$a0) move \$\$s3, \$a1 ## copia parâmetro \$a1 para \$\$s3 (salva \$a1) </pre>
Loop externo	for1tst:	<pre> move \$\$s0, \$zero## i = 0 slt\$t0, \$\$s0, \$\$s3 ## reg. \$t0 = 0 se \$\$s0\$\$s3 (in)≥ beq \$\$t0, \$zero, exit1. ## vai para exit1 se \$\$s0 ≥ \$\$s3 (i ≥ n) </pre>
Loop interno	for2tst:	<pre> addi \$\$s1, \$\$s0, -1## j = i - 1 slti\$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, \$\$s2, \$\$t1## reg. \$t2 = v + (j * 4) lw \$\$t3, 0(\$t2)## reg. \$t3 = v[j] lw \$\$t4, 4(\$t2)## reg. \$t4 = v[j+1] slt \$\$t0, \$\$t4, \$\$t3## reg. \$t0 = 1 se \$t4 ≥ \$t3 beq \$\$t0, \$zero, exit2 ## vai para exit2 se \$t4 ≥ \$t3 </pre>
Passa parâmetros e chama		<pre> move \$a0, \$\$s2 # 1º parâmetro de swap é v (antigo \$a0) move \$a1, \$\$s1 # 1º parâmetro de swap é j jal swap # código de swap mostrado na Figura 2.25 </pre>
Loop interno		<pre> addi \$\$s1, \$\$s1, -1## j -= 1 j for2tst # desvia para teste do loop interno </pre>
Loop externo	exit2:	<pre> addi \$\$s0, \$\$s0, 1 # i += 1 j for1tst # desvia para teste do loop externo </pre>
Restaurando registradores		
	exit1:	<pre> 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 </pre>
Retorno do procedimento		
	jr	\$ra
		# retorna à rotina que chamou jr

FIGURA 2.27 Versão em assembly do MIPS para o procedimento sort da Figura 2.26.

Detalhamento

Uma otimização que funciona com este exemplo é a utilização de *procedimentos inline*. 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 para `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 fosse chamado de vários locais. Essa expansão de código poderia ter um desempenho *inferior* se aumentasse a taxa de falhas

na cache; ver Capítulo 5.

Entendendo o desempenho dos programas

A Figura 2.28 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, porém O3 é a mais rápida, lembrando que o tempo é a única medida precisa do desempenho do programa.

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 2.28 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. Estes 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 2.29 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 JIT 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. 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 no 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 ($0,05 \times 2468$ ou 123 vezes mais rápido que o código C não otimizado *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 2.29 Desempenho de dois algoritmos de ordenação em C e Java usando interpretação e compiladores otimizadores 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 2.28. A JVM é a versão 1.3.1, e o JIT é o Hotspot versão 1.3.1, ambos da Sun.

Detalhamento

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, de modo a 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 recolha, digamos, o terceiro argumento de um procedimento. Quando, o compilador encontra o raro vararg, ele copia os quatro registradores de argumento para os quatro locais reservados na pilha.

2.14. Arrays *versus* ponteiros

Um tópico desafiador para qualquer programador C novo é entender os ponteiros. A comparação entre o código assembly que usa arrays e índices de array para 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 sequência de palavras na memória: uma

usando índices de array e uma usando ponteiros. A [Figura 2.30](#) mostra os dois procedimentos em C.

```
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 2.30 Dois procedimentos em C 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 sequencial 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 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`.

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: sll      $t1,$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 de array[i]
```

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    # se (i < size) vai para 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; o ANSI C requer um teste de tamanho antes do loop, mas pularemos essa conformidade aqui.)

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 seguinte é o corpo do loop *for*, que simplesmente armazena 0 em p:

```
loop2: sw      $zero,0($t0) # Memory[p] = 0
```

Essa instrução implementa o corpo do loop, de modo que o próximo código é o incremento da iteração, que muda p de modo 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 sequencial 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:

```
sll      $t1,$a1,2    # $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

        sll  $t1,$a1,2      # $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 sequências lado a lado ilustra a diferença entre os índices de array e ponteiros (as mudanças introduzidas pela versão de ponteiro estão destacadas):

move \$t0,\$zero	# i = 0	move \$t0,\$a0	# p = & array[0]
loop1: sll \$t1,\$t0,2	# \$t1= i * 4	sll \$t1,\$a1,2	# \$t1= size * 4
add \$t2,\$a0,\$t1	# \$t2 = &array[i]	add \$t2,\$a0,\$t1	# \$t2 = &array[size]
sw \$zero, 0(\$t2)	# array[i]=0	loop2: sw \$zero,0(\$t0)	# Memória[p]=0
addi \$t0,\$t0,1	# i = i + 1	addi \$t0,\$t0,4	# p = p + 4
slt \$t3,\$t0,\$a1	# \$t3=(i < size)	slt \$t3,\$t0,\$t2	# \$t3=(p < &array[size])
bne \$t3,\$zero,loop1	# if () vai para loop1	bne \$t3,\$zero,loop2	# se () vai para loop2

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 move o deslocamento em escala e a adição do limite de array para fora do loop, reduzindo assim as instruções executadas por iteração de 6 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

Como mencionamos, 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*.

Entendendo 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 compiladores com otimizações modernos podem produzir um código usando arrays tão bom quanto. A maioria dos programadores de hoje prefere que o compilador realize o

trabalho pesado.

2.15. Vida real: instruções ARMv7 (32 bits)

ARM é a arquitetura de conjunto de instruções mais comum para dispositivos embutidos, com mais de nove bilhões de dispositivos em 2011 usando ARM, e o crescimento recente tem sido de 2 bilhões por ano. Preparada originalmente para a Acorn RISC Machine, mais tarde modificada para Advanced RISC Machine, ARM surgiu no mesmo ano em que o MIPS e seguiu filosofias semelhantes. A [Figura 2.31](#) lista as semelhanças. A principal diferença é que MIPS tem mais registradores e ARM tem mais modos de endereçamento.

	ARM	MIPS
Data do anúncio	1985	1985
Tamanho da instrução (bits)	32	32
Espaço de endereços (tamanho, modelo)	32 bits, plano	32 bits, plano
Alinhamento de dados	Alinhado	Alinhado
Modos de endereçamento de dados	9	3
Registradores de inteiros (número, modelo, tamanho)	15 GPR × 32 bits	31 GPR × 32 bits
E/S	Mapeado na memória	Mapeado na memória

FIGURA 2.31 Semelhanças nos conjuntos de instruções ARM e MIPS.

Existe um núcleo semelhante dos conjuntos de instruções para instruções aritmética-lógica e de transferência de dados para o MIPS e ARM, como mostra a [Figura 2.32](#).

	Nome da instrução	ARM	MIPS
Registrador-registrador	Add	add	addu, addiu
	Add (trap if overflow)	adds; swivs	add
	Subtract	sub	subu
	Subtract (trap if overflow)	subs; swivs	sub
	Multiply	mul	mult, multu
	Divide	—	div, divu
	And	and	and
	Or	orr	or
	Xor	eor	xor
	Load high part register	—	lui
	Shift left logical	lsl ¹	sllv, sll
	Shift right logical	lsr ¹	srlv, srl
	Shift right arithmetic	asr ¹	sra, sra
	Compare	cmp, cmn, tst, teq	slt/i, slt/iu
Transferência de dados	Load byte signed	ldrsb	lb
	Load byte unsigned	ldrb	lbu
	Load halfword signed	ldrsh	lh
	Load halfword unsigned	ldrh	lhu
	Load word	ldr	lw
	Store byte	strb	sb
	Store halfword	strh	sh
	Store word	str	sw
	Read, write special registers	mrs, msr	move
	Atomic Exchange	swp, swpb	ll;sc

FIGURA 2.32 Instruções ARM registrador-registrador e transferência de dados equivalentes ao núcleo MIPS.

Os traços significam que a operação não está disponível nessa arquitetura ou não é sintetizada em poucas instruções. Se houver várias escolhas de instruções equivalentes ao núcleo MIPS, elas são separadas por vírgulas. ARM inclui deslocamentos como parte de cada instrução de operação de dados, de modo que os shift com sobrescrito 1 são apenas uma variação de uma instrução move, como lsl^1 . Observe que o ARM não possui instrução de divisão.

Modos de endereçamento

A Figura 2.33 mostra os modos de endereçamento de dados admitidos pelo ARM. Diferente do MIPS, o ARM não reserva um registrador para conter 0. Embora o MIPS tenha apenas três modos de endereçamento de dados simples (Figura 2.18), ARM tem nove, incluindo cálculos bastante complexos. Por

exemplo, ARM tem um modo de endereçamento que pode deslocar um registrador por qualquer quantidade, somá-lo aos outros registradores a fim de formar o endereço, e depois atualizar um registrador com esse novo endereço.

Modo de endereçamento	ARM	MIPS
Operando de registrador	X	X
Operando imediato	X	X
Registrador + offset (deslocamento ou base)	X	X
Registrador + registrador (indexado)	X	—
Registrador + registrador escalado (escalado)	X	—
Registrador + offset e registrador de atualização	X	—
Registrador + registrador e registrador de atualização	X	—
Autoincremento, autodecremento	X	—
Dados relativos ao PC	X	—

FIGURA 2.33 Resumo dos modos de endereçamento de dados.

ARM tem modos de endereçamento separados, registrador indireto e registrador + offset, em vez de colocar apenas 0 no deslocamento do segundo modo. Para obter um maior intervalo de endereçamento, o ARM desloca o offset à esquerda, 1 ou 2 bits se o tamanho dos dados for halfword ou uma palavra inteira.

Comparação e desvio condicional

MIPS usa o conteúdo dos registradores para avaliar desvios condicionais. ARM usa os quatro bits de código de condição tradicionais armazenados na palavra de status do programa: *negativo*, *zero*, *carry* e *overflow*. Eles podem ser definidos em qualquer instrução aritmética ou lógica; diferente das arquiteturas anteriores, essa configuração é opcional em cada instrução. Uma opção explícita leva a menos problemas em uma implementação em pipeline. ARM utiliza desvios condicionais para testar os códigos de condição a fim de determinar todas as relações sem sinal e com sinal possíveis.

CMP subtrai um operando do outro, e a diferença define os códigos de condição. CMN *compare negative soma* um operando ao outro, e a soma define os códigos de condição. TST realiza um AND lógico sobre os dois operandos para definir todos os códigos de condição menos overflow, enquanto TEQ utiliza

OR exclusivo a fim de definir os três primeiros códigos de condição.

Um recurso incomum do ARM é que cada instrução tem a opção de executar condicionalmente, dependendo dos códigos de condição. Cada instrução começa com um campo de 4 bits que determina se ele atuará como uma instrução de nenhuma operação (nop) ou como uma instrução real, dependendo dos códigos de condição. Logo, os desvios condicionais são corretamente considerados como executando condicionalmente a instrução de desvio incondicional. A execução condicional permite evitar que um desvio salte sobre uma instrução isolada. É preciso menos espaço de código e tempo para apenas executar uma instrução condicionalmente.

A [Figura 2.34](#) mostra os formatos de instrução para ARM e MIPS. As principais diferenças são o campo de execução condicional de 4 bits em cada instrução e o campo de registrador menor, pois ARM tem metade do número de registradores.

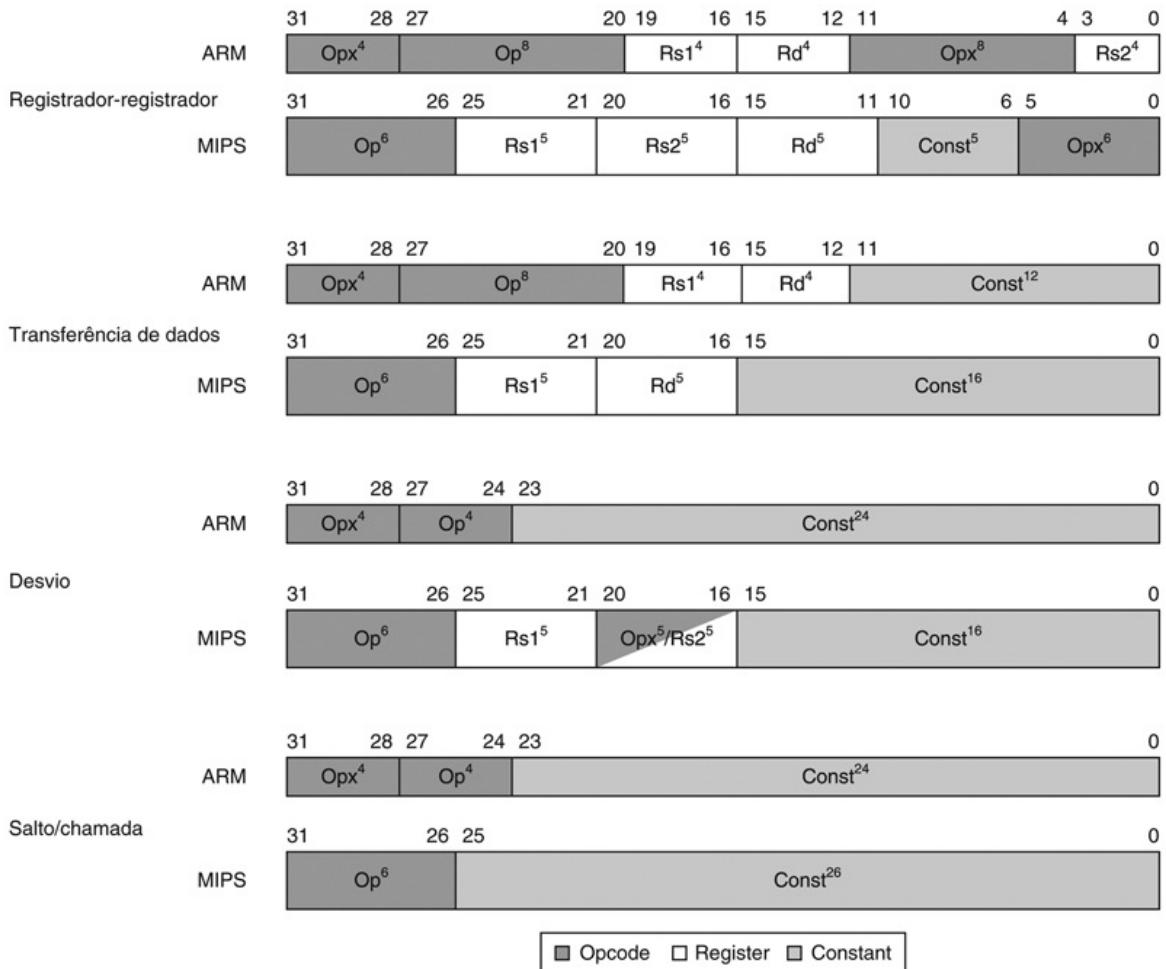


FIGURA 2.34 Formatos de instrução, ARM e MIPS.

As diferenças resultam de arquiteturas com 16 ou 32 registradores.

Recursos exclusivos do ARM

A Figura 2.35 mostra algumas instruções aritmética-lógica não encontradas no MIPS. Por não possuir um registrador dedicado para 0, ARM tem opcodes separados para realizar algumas operações que MIPS pode fazer com \$zero. Além disso, ARM tem suporte para aritmética de múltiplas palavras.

Nome	Definição	ARM	MIPS
Load imediato	Rd = Imm	mov	addi, \$0,
Not	Rd = ~(Rs1)	mvn	nor, \$0,
Move	Rd = Rs1	mov	or, \$0,
Rotação para direita	Rd = Rs $i \gg i$ Rd _{0...i-1} = Rs _{31-i...31}	ror	
And not	Rd = Rs1 & ~(Rs2)	bic	
Subtração Reversa	Rd = Rs2 - Rs1	rsb, rsc	
Apoio à adição de inteiros em múltiplas palavras	CarryOut, Rd = Rd + Rs1 + OldCarryOut	adcs	—
Apoio à subtração de inteiros em múltiplas palavras	CarryOut, Rd = Rd - Rs1 + OldCarryOut	sbc	—

FIGURA 2.35 Instruções aritméticas/lógicas do ARM não encontradas no MIPS.

O campo imediato de 12 bits do ARM tem uma nova interpretação. Os oito bits menos significativos são estendidos em zero a um valor de 32 bits, depois girados para a direita pelo número de bits especificado nos quatro primeiros bits do campo multiplicado por dois. Uma vantagem é que esse esquema pode representar todas as potências de dois em uma palavra de 32 bits. Um estudo interessante seria descobrir se essa divisão realmente recolhe mais imediatos do que um campo simples de 12 bits.

O deslocamento de operandos não é limitado a imediatos. O segundo registrador de todas as operações de processamento aritmético e lógico tem a opção de ser deslocado antes de ser acionado. As opções de deslocamento são *shift left logical*, *shift right logical*, *shift right arithmetic* e *rotate right*.

ARM também possui instruções para salvar grupos de registradores, chamados *loads* e *stores em bloco*. Sob o controle de uma máscara de 16 bits dentro das instruções, qualquer um dos 16 registradores pode ser carregado ou armazenado na memória em uma única instrução. Essas instruções podem salvar e restaurar registradores na entrada e retorno do procedimento. Elas também podem ser usadas para cópia de memória em bloco, sendo este o uso mais importante dessa instrução hoje em dia.

2.16. Vida real: instruções x86

A beleza está toda nos olhos de quem vê.

Margaret Wolfe Hungerford, Molly Bawn, 1877

Os projetistas de conjuntos de instruções às vezes oferecem operações mais poderosas do que aquelas encontradas no ARM e 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 sequência mais simples.

O caminho em direção à complexidade da operação é, portanto, repleto de perigos. A [Seção 2.18](#) demonstra as armadilhas da complexidade.

A evolução do Intel x86

O ARM e o MIPS foram a visão de pequenos grupos individuais, no ano de 1985; as partes dessas arquiteturas se encaixam muito bem e a arquitetura inteira pode ser descrita de forma sucinta. Isso não acontece com o X86; ele é o produto de vários grupos independentes, que evoluíram a arquitetura por 35 anos, acrescentando novos recursos ao conjunto de instruções original, como alguém acrescentando roupas em uma mala pronta. Aqui estão os marcos importantes do X86:

- **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**.

registradores de uso geral (GPR — General-Purpose Register)

Um registrador que pode ser usado para endereços ou para dados, com praticamente qualquer instrução.

- **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 ([Seção 3.7](#)).
- **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 ([Capítulo 5](#)) 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 os mesmos 32 bits de espaço de endereçamento, 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 ([Capítulo 5](#)). Assim como o 80286, o 80386 possui um modo para executar programas do 8086 sem mudanças.
- **1989-1995:** 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 6](#)) 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 *Multi Media Extensions* (MMX). Esse novo conjunto de 57 instruções utiliza a pilha de ponto flutuante de modo a 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) ([Capítulo 6](#)). O Pentium II não introduziu novas instruções.
- **1999:** a Intel acrescentou outras 70 instruções, denominadas *Streaming SIMD Extensions* (SSE), 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 incluem instruções de *prefetch* (pré-busca) da cache, mais instruções de armazenamento de streaming, que contornam as caches e escrevem diretamente na memória.
- **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 x86. 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 na arquitetura vem da inclusão de um novo modo, chamado *modo longo*, que redefine a execução de todas as instruções x86 com endereços e dados de 64 bits. Para enfrentar a quantidade maior de registradores, ela 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 x86 (*modo legado*) e mais um modo que restringe os programas do usuário ao x86, 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 ([Seção 2.11](#)). A AMD ofereceu o SSE3 nos chips subsequentes e incluiu a instrução de troca atômica que estava faltando no ADM64, para manter a compatibilidade binária com a Intel.
- **2006:** a Intel anuncia 54 novas instruções como parte das extensões do conjunto de instruções SSE4. Essas extensões realizam coisas como soma de diferenças absolutas, produtos escalares para arrays de estruturas, extensão de sinal ou zero de dados estreitos para tamanhos mais largos, contagem de

população e assim por diante. Ela também acrescentou suporte para máquinas virtuais ([Capítulo 5](#)).

- **2007:** a AMD anuncia 170 instruções como parte das SSE5, incluindo 46 instruções do conjunto de instruções básico, que acrescenta três instruções de operando, como MIPS.
- **2011:** a Intel lança a Advanced Vector Extension, que expande a largura de registrador das SSE de 128 para 256 bits, redefinindo, assim, cerca de 250 instruções e acrescentando 128 novas instruções.

Essa história ilustra o impacto das “algemas douradas” da compatibilidade com o x86, 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 x86, lembre-se de que esse conjunto de instruções impulsionou a geração de computadores PC e ainda domina a parte da nuvem da era pós-PC. A fabricação de 350 milhões de chips x86 por ano pode parecer pequena em comparação com os 9 bilhões de chips ARMv7, mas muitas empresas adorariam controlar esse mercado. 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 x86; em vez disso, o objetivo é que você tenha alguma familiaridade com os pontos fortes e fracos da arquitetura mais popular do mundo para uso em desktops.*

Em vez de mostrar o conjunto de instruções inteiro de 16, 32 e 64 bits, nesta seção, vamos nos concentrar no subconjunto de 32 bits originado com o 80386. 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 x86

Os registradores do 80386 mostram a evolução do conjunto de instruções ([Figura 2.36](#)). O 80386 estendeu todos os registradores de 16 bits (exceto os registradores de segmento) para 32 bits, inserindo 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 e os ARM, duas vezes.

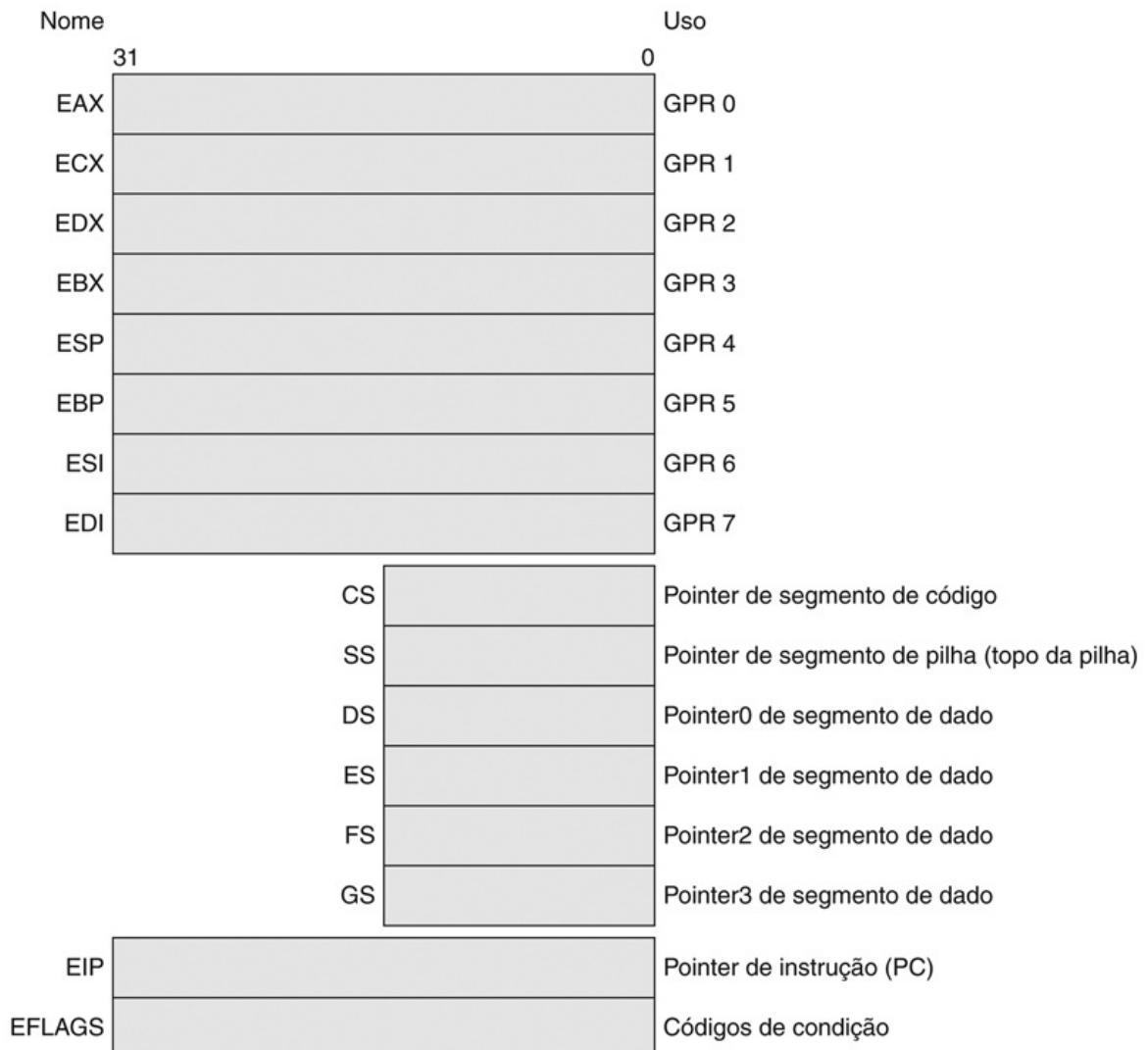


FIGURA 2.36 O conjunto de registradores do 80386.

Começando com o 80386, os oito registradores iniciais foram estendidos para 32 bits e também poderiam ser usados como registradores de uso geral.

A [Figura 2.37](#) mostra que as instruções aritméticas, lógicas e de transferência de dados são instruções de dois operandos. Existem duas diferenças importantes aqui. As instruções aritméticas e lógicas do x86 precisam ter um operando que atue como origem e destino; o ARMv7 e o MIPS admitem 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 ARMv7 e do MIPS.

Tipo de operando de origem/destino	Segundo operando de origem
Registrador	Registrador
Registrador	Imediato
Registrador	Memória
Memória	Registrador
Memória	Imediato

FIGURA 2.37 Tipos de instrução para instruções aritméticas, lógicas e de transferência de dados.

O x86 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 2.36](#) (não EIP ou EFLAGS).

Os 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 2.38](#) mostra os modos de endereçamento do x86 e quais GPRs não podem ser usados com cada modo, além de como obter o mesmo efeito usando instruções MIPS.

Modo	Descrição	Restrições de registradores	MIPS equivalente
Registrador indireto	O endereço está em um registrador.	Não é ESP ou EBP	<code>1w \$s0,0(\$\$1)</code>
Modo baseado com deslocamento de 8 ou 32 bits	O endereço é o conteúdo do registrador-base com o deslocamento.	Não é ESP	<code>1w \$s0,100(\$\$1) # <= 16-bit # deslocamento</code>
Base com o índice escalado	O endereço é a Base + (2^{Escala} x Índice), em que a Escala tem o valor de 0, 1, 2 ou 3.	Base: qualquer GPR Índice: não é ESP	<code>mul \$t0,\$s2,4 add \$t0,\$t0,\$\$1 1w \$\$0,0(\$t0)</code>
Base com índice escalado e deslocamento de 8 ou 32 bits	O endereço é a Base + (2^{Escala} x Índice) + Deslocamento, em que a Escala tem o valor de 0, 1, 2 ou 3.	Base: qualquer GPR Índice: não é ESP	<code>mul \$t0,\$s2,4 add \$t0,\$t0,\$\$1 1w \$\$0,100(\$t0) # <= 16-bit # deslocamento</code>

FIGURA 2.38 Modos de endereçamento de 32 bits do x86 com restrições de registrador e o código MIPS equivalente.

O modo de endereçamento Base mais Índice Escalado, que não

aparece no ARM ou no MIPS, foi incluído para evitar as multiplicações por quatro (fator de escala 2) para transformar um índice de um registrador em um endereço em bytes ([Figuras 2.25 e 2.27](#)). 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 precisa de mais duas instruções: um `lui` 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.)

Operações com inteiros do x86

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 x86. (AMD64 acrescenta endereços e dados de 64 bits, chamados *quad words*; vamos nos ater ao 80386 nesta seção.) 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 os arquitetos do 80386 ofereceram uma forma 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 poder definir um tamanho grande padrão. Esse tamanho de dados padrão é definido por um bit no registrador do segmento de código. Para redefini-lo, um *prefixo* de 8 bits é anexado à instrução a fim de dizer à máquina para usar o outro tamanho grande para essa instrução.

A solução do prefixo foi emprestada do 8086, o que permite que diversos prefixos modifiquem o comportamento da instrução. Os três prefixos originais redefinem o registrador de segmento padrão, bloqueiam o barramento para dar suporte à sincronização ([Seção 2.11](#)) 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 x86 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
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 2.39](#) mostra algumas instruções x86 típicas e suas funções.

Instrução	Função
je nome	se for igual(códigos de condição) {EIP=n} 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 2.39 Algumas instruções x86 típicas e suas funções.

Uma lista de operações frequentes aparece na [Figura 2.40](#). O CALL salva na pilha o EIP da próxima instrução. (EIP é o PC — Program Counter — da Intel.)

Os desvios condicionais no x86 são baseados em *códigos de condição* ou *flags*, assim como no ARMv7. 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. Os endereços de desvio relativos ao PC precisam ser especificados no número de bytes, visto

que, ao contrário do ARMv7 e MIPS, nem todas as instruções do 80386 possuem 4 bytes de extensão.

As instruções para manipulação de strings fazem parte dos antepassados 8080 do x86 e não são comumente executadas na maioria dos programas. Em geral, são mais lentas do que as rotinas de software equivalentes (veja a falácia na [Seção 2.17](#)).

A [Figura 2.40](#) lista algumas das instruções do x86 com inteiros. Muitas das instruções estão disponíveis nos formatos byte e word.

Instrução	Significado
Controle	Desvios condicionais ou incondicionais
jnz, jz	Pula se a condição para o offset EIP + 8-bit; JNE (para JNZ), JE (para JZ) são nomes alternativos
jmp	Pulo incondicional – offset de 8 ou 16 bits
call	Chamada de sub-rotina – offset de 16-bit; o endereço de retorno é empurrado para a pilha
ret	Dispara o endereço de retorno da pilha e pula para ele
loop	Desvio de loop – diminuição do ECX; pula para o deslocamento EIP + 8-but se ECX ≠ 0
Transferência de dados	Movimenta dados entre registradores ou entre registrador e memória
move	Movimento entre dois registradores ou entre um registrador e memória
push, pop	Empurra o operando-fonte na pilha; dispara o operando do topo da pilha para o registrador
les	Carrega o ES a um dos GPRs da memória
Aritmético, lógico	Operações aritméticas e lógicas utilizando dados de registradores e memória
add, sub	Adiciona a fonte ao destino; subtrai a fonte do destino; formato registrador-memória
cmp	Compara fonte e destino; formato registrador-memória
shl, shr, rcr	Desloca à esquerda; desloca à direita lógica; rotacional à direita com o código de condição de carregamento conforme preenchido
cbw	Converte byte em 8 bits direitos de EAX para 16-bit word na direita do EAX
test	Lógico E, fonte e destino configuram os códigos de condição
inc, dec	Incrementa o destino; desincrementa o destino
or, xor	OR lógico; OR exclusive; formato registrador-memória
String	Movimento entre operandos string; comprimento dado por um prefixo repetido
movs	Copia a fonte string para o destino pelo incremento do ESI e EDI; pode ser repetido
lod\$	Carrega um byte, word ou doubleword de uma string em um registrador EAX

FIGURA 2.40 Algumas operações típicas do x86.

Muitas operações utilizam o formato registrador-memória, no qual a origem ou o destino pode ser a memória e o outro pode ser um registrador ou um operando imediato.

Codificação de instruções x86

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 15 bytes.

A [Figura 2.41](#) mostra o formato de instrução para várias instruções de exemplo na [Figura 2.39](#). 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, índice, base”.

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	r/m Pós-byte	Deslocamento

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Imediato

f. TEST EDX, #42

7	1	8	32
TEST	w	Pós-byte	Imediato

FIGURA 2.41 Formatos típicos de instruções x86.

A [Figura 2.42](#) 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 double word. O campo d em mov é usado em instruções que podem mover de/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 15 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 2.42](#) 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	+ disp8	+ disp8	+ disp16	+ disp32	+ disp32	registro
4	AH	SP	ESP	4	end=SI =(sib)	SI+disp8	(sib)+disp8	SI+disp8	(sib)+disp32		"
5	CH	BP	EBP	5	end=DI =disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32		"
6	DH	SI	ESI	6	end=disp16 =ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32		"
7	BH	DI	EDI	7	end=BX =EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32		"

FIGURA 2.42 A codificação do primeiro especificador de endereço do x86: “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: 1) r/m = 6 quando mod = 1 ou mod = 2 no modo de 16 bits seleciona BP mais o deslocamento; 2) r/m = 5 quando mod = 1 ou mod = 2 no modo 16 bits seleciona EBP mais deslocamento; e 3) r/m = 4 no modo de 32 bits quando mod não é igual a 3, em que (sib) significa o uso do modo de índice escalado, mostrado na [Figura 2.38](#). Quando mod = 3, o campo r/m indica um registrador, usando a mesma codificação que o campo reg combinado com o bit w.

Conclusão sobre o x86

A Intel tinha um microprocessador de 16 bits dois anos antes das arquiteturas mais elegantes de seus concorrentes, como o Motorola 68000, e essa dianteira levou à seleção do 8086 como CPU para o IBM PC. Os engenheiros da Intel

geralmente reconhecem que o x86 é mais difícil de ser montado do que máquinas como ARMv7 e MIPS, mas o mercado maior significou, na era do PC, que a AMD e a Intel podiam abrir mão de mais recursos para ajudar a contornar a complexidade adicional. O que o x86 perde no estilo é compensado na fatia do mercado, tornando-o belo, do ponto de vista apropriado.

O que salva é que os componentes arquitetônicos mais usados do x86 não são tão difíceis de implementar, como a AMD e a Intel já demonstraram, 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.

Entretanto, na era pós-PC, apesar das habilidades consideráveis em termos de arquitetura e manufatura, o x86 ainda não se tornou competitivo no dispositivo móvel pessoal.

2.17. Vida real: instruções ARMv8 (64 bits)

Dos muitos problemas em potencial em um conjunto de instruções, aquele que é quase impossível de ser contornado é ter um endereço de memória muito pequeno. Enquanto o x86 foi estendido com sucesso, primeiro para endereços de 32 bits e depois para endereços de 64 bits, muitos de seus irmãos ficaram para trás. Por exemplo, o MOStek 6502 com endereços de 16 bits controlava o Apple II, mas mesmo tendo saído na frente com o primeiro computador pessoal comercialmente bem-sucedido, sua falta de bits de endereço o condenou à caixa de lixo da história.

Os arquitetos do ARM podiam ver o iminente fim de seu computador com 32 bits de endereços, e iniciaram o projeto da versão com endereços de 64 bits do ARM em 2007. Finalmente, ele foi revelado em 2013. Em vez de algumas pequenas mudanças estéticas, para que todos os registradores tivessem 64 bits de largura, basicamente o que aconteceu com o x86, o ARM fez uma reforma completa. A boa notícia é que, se você conhece o MIPS, será muito fácil entender o ARMv8, como é chamada a versão para 64 bits.

Primeiro, em comparação com o MIPS, o ARM descartou praticamente todos os recursos incomuns do v7:

- Não há um campo de execução condicional, como havia em quase toda instrução no v7.
- O campo imediato é simplesmente uma constante de 12 bits, em vez de basicamente uma entrada para uma função que produz uma constante, como

no v7.

- O ARM retirou as instruções Load Multiple e Store Multiple.
- O PC não é mais um dos registradores, o que resultava em desvios inesperados se você escrevesse nele.
- Em segundo lugar, o ARM acrescentou recursos que faltavam e que são úteis no MIPS:
- O V8 possui 32 registradores de uso geral, que os escritores de compilador certamente apreciam muito. Assim como o MIPS, um registrador é fixado em 0, embora em vez disso se refira ao ponteiro de pilha em instruções load e store.
- Seus modos de endereçamento funcionam para todos os tamanhos de word no ARMv8, o que não acontecia no ARMv7.
- Ele inclui uma instrução de divisão, que foi omitida do ARMv7.
- Ele acrescenta o equivalente ao “branch if equal” e o “branch if not equal” do MIPS.

Como a filosofia do conjunto de instruções do v8 é muito mais próxima do MIPS do que do v7, nossa conclusão é que a semelhança principal entre o ARMv7 e o ARMv8 está no nome.

2.18. Falácia e armadilhas

Falácia: instruções mais poderosas significam maior desempenho.

Parte do poder do Intel x86 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 sequência 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 do programa, com o código replicado para reduzir o trabalho extra do loop, copia cerca de 1,5 vez mais rápido. Uma terceira versão, que usa os registradores de ponto flutuante maiores no lugar dos registradores inteiros do x86, copia cerca de 2,0 vezes mais rápido do que a instrução de movimentação 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 perfeitamente os conceitos dos Capítulos 4 e 5 (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-texto 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 geral 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 ou Java. 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.

Falácia: a importância da compatibilidade binária comercial significa que os conjuntos de instruções bem-sucedidos não mudam.

Embora a compatibilidade binária seja sacrossanta, a [Figura 2.43](#) mostra que a arquitetura do x86 cresceu drasticamente. A média é mais de uma instrução por mês no decorrer do seu tempo de vida de 35 anos!

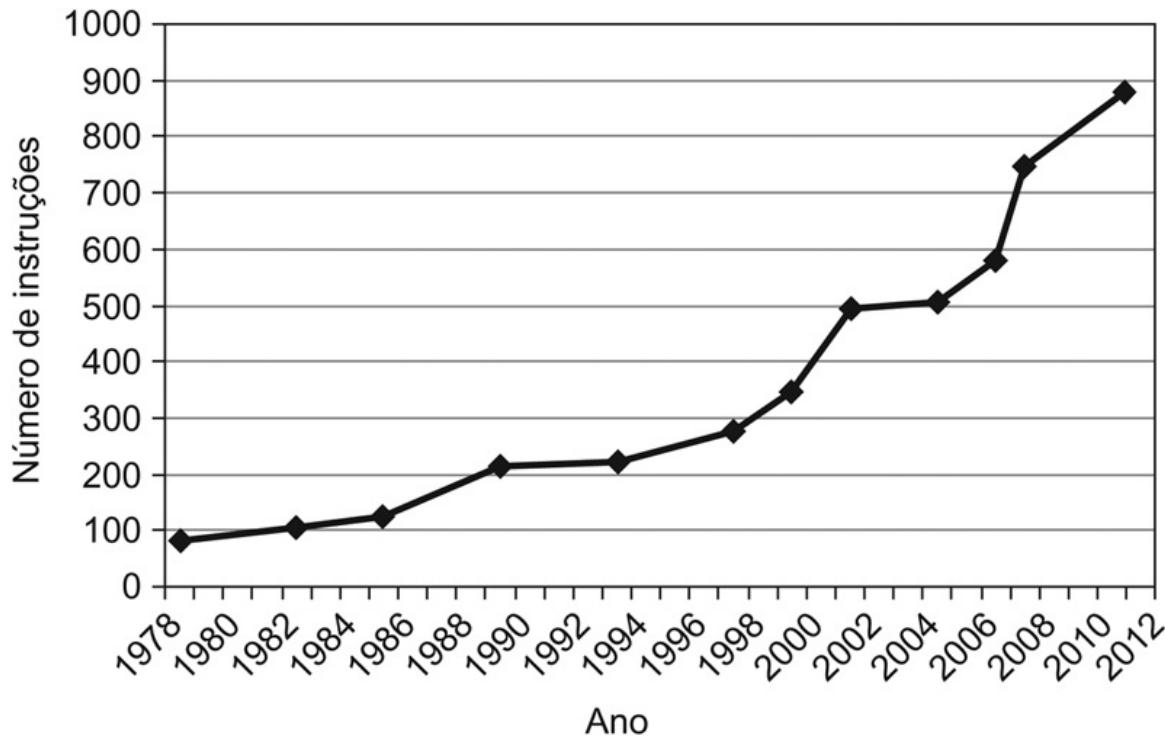


FIGURA 2.43 Crescimento do conjunto de instruções x86 com o tempo.

Embora haja um valor técnico claro em algumas dessas extensões, essa mudança rápida também aumenta a dificuldade para outras empresas tentarem montar processadores compatíveis.

Armadilha: esquecer que os endereços sequenciais de palavras em máquinas com endereçamento em bytes não diferem em um.

Muitos programadores assembly têm lutado contra erros cometidos pela suposição de que o endereço da próxima palavra pode ser encontrado incrementando-se o endereço em um registrador por um, em vez do tamanho da palavra 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. Seguindo a disciplina de pilha da [Figura 2.12](#), a memória que contém o array local será reutilizada assim que o procedimento retornar. Os ponteiros para variáveis automáticas podem levar ao caos.

2.19. Comentários finais

Menos significa mais.

Robert Browning, Andrea del Sarto, 1855

Os dois princípios do computador com *programa armazenado* são o uso de instruções que sejam indiferentes de números e o uso de memória alterável para os programas. Estes 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 quantidade de ciclos de clock necessários por uma instrução e a velocidade do clock. Como ilustramos neste capítulo, três princípios de projeto orientam os autores de conjuntos de instruções a estabelecer 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. *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.

Também vimos a grande ideia de tornar o **caso comum veloz** aplicada a conjuntos de instruções e também à arquitetura do computador. Alguns

exemplos de tornar o caso comum do MIPS veloz são o endereçamento relativo ao PC para desvios condicionais e o endereçamento imediato para operandos constantes maiores.



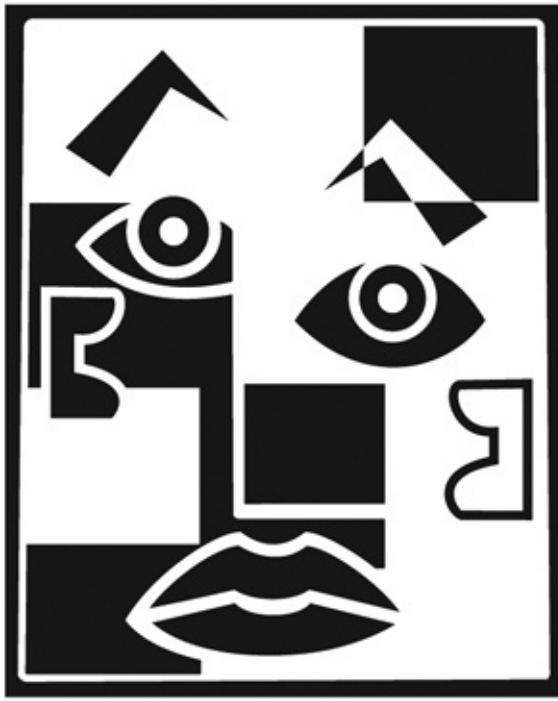
CASO COMUM VELOZ

Acima desse nível de máquina está o assembly, uma linguagem que os humanos podem ler. O montador 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 2.44](#) lista as instruções MIPS que abordamos até aqui, tanto instruções reais quanto pseudoinstruções. Ocultar os detalhes do nível mais alto é outro exemplo da grande ideia da **abstração**.

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
load half unsigned	lhu	I			
store half	sh	I	branch greater than	bgt	I
load byte	lb	I	branch greater than or equal	bge	I
load byte unsigned	lbu	I			
store byte	sb	I			
load linked	ll	I			
store conditional	sc	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			
set less than immediate unsigned	sltiu	I			
jump	j	J			
jump register	jr	R			
jump and link	jal	J			

FIGURA 2.44 O conjunto de instruções do MIPS explicado até aqui, com as instruções MIPS reais à esquerda e as pseudoinstruções à direita.

O Apêndice A ([Seção A.10](#)) descreve a arquitetura MIPS completa. A [Figura 2.1](#) mostra mais detalhes da arquitetura MIPS revelada neste capítulo. As informações que aparecem aqui são encontradas nas colunas 1 e 2 do Guia de Referência do MIPS, no final deste livro.



A B S T R A Ç Ã O

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*.

Essas instruções não nasceram iguais; a popularidade de poucas domina muitas. Por exemplo, a [Figura 2.45](#) mostra a popularidade de cada classe de instruções para o SPE CPUC2006. 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	16%	48%
Transferência de dados	lw, sw, lb, lbu, lh, lhu, sb, lui	referências a estruturas de dados, como arrays	35%	36%
Lógica	and, or, nor, andi, ori, sll, srl	operações em instruções de atribuição	12%	4%
Desvio condicional	beq, bne, slt, slti, sltiu	instruções if e loops	34%	8%
Jump	j, jr, jal	chamadas de procedimento, retornos e instruções case/switch	2%	0%

FIGURA 2.45 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 para os benchmarks médios SPEC CPU2006 de inteiros e ponto flutuante.

A Figura 3.26 no Capítulo 3 mostra a porcentagem das instruções MIPS individuais executadas.

Depois que explicarmos a aritmética do computador no Capítulo 3, revelaremos mais da arquitetura do conjunto de instruções do MIPS.

2.20. Exercícios

O Apêndice A descreve o simulador do MIPS, que é útil para estes exercícios. Embora o simulador aceite pseudoinstruções, tente não as usar 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 pseudoinstruções.

Existem alguns casos em que as pseudoinstruções 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 pseudoinstruções por esses motivos, por favor, acrescente uma sentença ou duas à sua solução, indicando quais pseudoinstruções usou e por quê.

2.1 [5] <§2.2> Para a instrução C a seguir, qual é o código assembly do MIPS correspondente? Suponha que as variáveis f, g, h e i sejam dadas e possam ser consideradas inteiros de 32 bits, conforme declarado em um programa C. Use um número mínimo de instruções assembly do MIPS.

$$F = g + (h - 5);$$

2.2 [5] <§2.2> Para as instruções assembly do MIPS a seguir, qual é a instrução C correspondente?

```
add f, g, h  
add f, i, f
```

2.3 [5] <§§2.2, 2.3> Para a instrução C a seguir, qual é o código assembly do MIPS correspondente? Suponha que as variáveis f, g, h, i e j sejam atribuídas aos registradores \$s0, \$s1, \$s2, \$s3 e \$s4, respectivamente. Suponha que o endereço de base dos arrays A e B estejam nos registradores \$s6 e \$s7, respectivamente.

$$B[8] = A[i - j];$$

2.4 [5] <§§2.2, 2.3> Para as instruções assembly do MIPS a seguir, qual é a instrução C correspondente? Suponha que as variáveis f, g, h, i e j sejam atribuídas aos registradores \$s0, \$s1, \$s2, \$s3 e \$s4, respectivamente. Suponha também que o endereço de base dos arrays A e B estejam nos registradores \$s6 e \$s7, respectivamente.

```
sll      $t0, $s0, 2      # $t0 = f * 4  
add     $t0, $s6, $t0      # $t0 = &A[f]  
sll      $t1, $s1, 2      # $t1 = g * 4  
add     $t1, $s7, $t1      # $t1 = &B[g]  
lw      $s0, $($t0)       # f = A[f]
```

```

addi    $t2, $t0, 4
lw      $t0, 0($t2)
add    $t0 $t0, $s0
sw      $t0, 0($t1)

```

2.5 [5] <§§2.2, 2.3> Para as instruções assembly do MIPS no Exercício 2.4, reescreva o código assembly para diminuir o número de instruções MIPS (se possível) necessárias para executar a mesma função.

2.6 A tabela a seguir mostra valores de 32 bits de um array armazenado na memória.

Endereço	Dados
24	2
38	4
32	3
36	6
40	1

2.6.1 [5] <§§2.2, 2.3> Para os locais de memória na tabela anterior, escreva o código C classificando os dados do mais baixo ao mais alto, colocando o menor valor no menor local de memória mostrado na figura. Suponha que os dados mostrados representem a variável C chamada `Array`, que é um array do tipo `interface`, e que o primeiro número no array mostrado seja o primeiro elemento no array. Suponha que essa máquina em particular seja uma máquina endereçável por byte e uma word consista em 4 bytes.

2.6.2 [5] <§2.2, 2.3> Para os locais de memória na tabela anterior, escreva o código MIPS que classifique os dados do mais baixo ao mais alto, colocando o menor valor no menor local de memória. Use um número mínimo de instruções MIPS. Suponha que o endereço de base de `Array` esteja armazenado no registrador `$s6`.

2.7 [5] <§2.3> Mostre como o valor `0xabcdef12` seria arrumado na memória de uma máquina little-endian e uma máquina big-endian. Suponha que os dados sejam armazenados a partir do endereço 0.

2.8 [5] <§2.4> Traduza `0xabcdef12` para decimal.

2.9 [5] <§§2.2, 2.3> Traduza o código C a seguir para MIPS. Suponha que as variáveis `f`, `g`, `h`, `I` e `j` sejam atribuídas aos registradores `$s0`, `$s1`, `$s2`, `$s3` e `$s4`, respectivamente. Suponha que o endereço de base dos arrays `A` e `B` estejam nos registradores `$s6` e `$s7`, respectivamente. Suponha que os elementos dos arrays `A` e `B` sejam words de 4 bytes:

$$B[8] = A[i] + A[j];$$

2.10 [5] <§§2.2, 2.3> Traduza o código MIPS a seguir para C. Suponha que as variáveis f, g, h, i e j sejam atribuídas aos registradores \$s0, \$s1, \$s2, \$s3 e \$s4, respectivamente. Suponha que o endereço de base dos arrays A e B estejam nos registradores \$s6 e \$s7, respectivamente.

```
addi    $t0, $s6, 4
add    $t1, $s6, $0
sw     $t1, 0($t0)
lw     $t0, 0($t0)
add    $s0, $t1, $t0
```

2.11 [5] <§§2.3, 2.5> Para cada instrução MIPS, mostre o valor dos campos de opcode (OP), registrador fonte (RS) e registrador de destino (RT). Para as instruções tipo I, mostre o valor do campo imediato, e para as instruções tipo R, mostre o valor do campo de registrador de destino (RD).

2.12 Suponha que os registradores \$s0 e \$s1 mantenham os valores 0×80000000 e $0 \times D0000000$, respectivamente.

2.12.1 [5] <§2.4> Qual é o valor de \$t0 para o código assembly a seguir?

```
add $t0, $s0, $s1
```

2.12.2 [5] <§2.4> O resultado em \$t0 é o resultado desejado ou houve overflow?

2.12.3 [5] <§2.4> Para o conteúdo dos registradores \$s0 e \$s1, conforme especificado acima, qual é o valor de \$t0 para o código assembly a seguir?

```
sub $t0, $s0, $s1
```

2.12.4 [5] <§2.4> O resultado em \$t0 é o resultado desejado ou houve overflow?

2.12.5 [5] <§2.4> Para o conteúdo dos registradores $\$s0$ e $\$s1$ especificado acima, qual é o valor de $\$t0$ para o código assembly a seguir?

```
add $t0, $s0, $s1  
add $t0, $t0, $s0
```

2.12.6 [5] <§2.4> O resultado em $\$t0$ é o resultado desejado ou houve overflow?

2.13 Suponha que $\$s0$ contenha o valor 128_{dec} .

2.13.1 [5] <§2.4> Para a instrução $add \$t0, \$s0, \$s1$, qual é ou quais são as faixas de valores para $\$s1$ que resultaria(m) em overflow?

2.13.2 [5] <§2.4> Para a instrução $sub \$t0, \$s0, \$s1$, qual é ou quais são as faixas de valores para $\$s1$ que resultaria(m) em overflow?

2.13.3 [5] <§2.4> Para a instrução $sub \$t0, \$s1, \$s0$, qual é ou quais são as faixas de valores para $\$s1$ que resultaria(m) em overflow?

2.14 [5] <§§2.4, 2.5> Forneça o tipo e a instrução em linguagem assembly para o seguinte valor binário: $0000\ 0010\ 0001\ 0000\ 1000\ 0000\ 0010\ 0000_{bin}$.

2.15 [5] <§§2.4, 2.5> Forneça o tipo e a representação hexadecimal da seguinte instrução: $sw \$t1, 32(\$t2)$

2.16 [5] <§2.5> Forneça o tipo, a instrução em linguagem assembly e a representação binária da instrução descrita pelos seguintes campos MIPS:

$op = 0, rs = 3, rt = 2, rd = 3, shamt = 0, funct = 34$

2.17 [5] <§2.5> Forneça o tipo, a instrução em linguagem assembly e a representação binária da instrução descrita pelos seguintes campos MIPS:

$op = 0 \times 23, rs = 1, rt = 2, const = 0 \times 4$

2.18 [5] <§2.5> Suponha que quiséssemos expandir o arquivo de registradores MIPS para 128 registradores e expandir o conjunto de instruções para conter quatro vezes o número de instruções atuais.

2.18.1 [5] <§2.5> Como isso afetaria o tamanho de cada um dos campos de bit

nas instruções do tipo R?

2.18.2 [5] <§2.5> Como isso afetaria o tamanho de cada um dos campos de bit nas instruções do tipo I?

2.18.3 [5] <§§2.5, 2.10> Como cada uma das duas mudanças propostas diminuiria o tamanho de um programa em assembly MIPS? Por outro lado, como a mudança proposta aumentaria o tamanho de um programa em assembly MIPS?

2.19 Considere o seguinte conteúdo de registradores:

$$\$t0 = 0 \times \text{AAAAAAA}, \$t1 = 0 \times 12345678$$

2.19.1 [5] <§2.6> Para os valores de registradores mostrados acima, qual é o valor de \$t2 para a seguinte sequências de instruções?

sll \$t2, \$t0, 44 ou \$t2, \$t2, \$t1

2.19.2 [5] <§2.6> Para os valores de registradores mostrados acima, qual é o valor de \$t2 para a seguinte sequências de instruções?

```
srl $t2, $t0, 4  
andi $t2, $t2, -1
```

2.19.3 [5] <§2.6> Para os valores de registradores mostrados acima, qual é o valor de \$t2 para a seguinte sequência de instruções?

```
srl $t2, $t0, 3  
andi $t2, $t2, 0xFFE
```

2.20 [5] <§2.6> Ache a sequência mais curta de instruções MIPS que extraia os bits de 16 até 11 do registrador \$t0 e use o valor desse campo para substituir os bits de 31 até 26 no registrador \$t1 sem alterar os outros 26 bits do registrador \$t1.

2.21 [5] <§2.6> Forneça um conjunto mínimo de instruções MIPS que possa ser utilizado para implementar a seguinte pseudoinstrução:

```
not $t1, $t2 // bit-wise invert
```

2.22 [5] <§2.6> Para a instrução C a seguir, escreva uma sequências mínima de instruções assembly do MIPS que faça a operação idêntica. Suponha que $\$t1 = A$, $\$t2 = B$, e $\$s1$ seja o endereço de base de C.

```
A = C[0] << 4;
```

2.23 [5] <§2.7> Suponha que $\$t0$ contenha o valor `0x00101000`. Qual é o valor de $\$t2$ após as instruções a seguir?

```
    slt $t2, $0, $t0
    bne $t2, $0, ELSE
    j DONE
ELSE: addi $t2, $t2, 2
DONE:
```

2.24 [5] <§2.7> Suponha que o contador de programa (PC) seja definido em `0x2000 0000`. É possível usar a instrução assembly do MIPS jump (j) para definir o PC para o endereço como `0x4000 0000`? É possível usar a instrução assembly do MIPS branch-on-equal (beq) para definir o PC como esse mesmo endereço?

2.25 A instrução a seguir não está incluída no conjunto de instruções do MIPS:

```
rpt $t2, loop # se(R[rs] > 0) R[rs] = R[rs]-1, PC = PC + 4 + BranchAddr
```

2.25.1 [5] <§2.7> Se essa instrução tivesse que ser implementada no conjunto de instruções do MIPS, qual seria o formato de instrução mais apropriado?

2.25.2 [5] <§2.7> Qual é a sensibilidade de instruções MIPS mais curta que realiza a mesma operação?

2.26 Considere o seguinte loop em MIPS:

```

LOOP: slt $t2, $0, $t1
      beq $t2, $0, DONE
      subi $t1, $t1, 1
      addi $s2, $s2, 2
      j LOOP

```

DONE:

2.26.1 [5] <§2.7> Suponha que o registrador \$t1 seja inicializado com o valor 10. Qual é o valor no registrador \$s2, supondo que \$s2 seja inicialmente zero?

2.26.2 [5] <§2.7> Para cada um dos loops acima, escreva a rotina equivalente em código C. Suponha que os registradores \$s1, \$s2, \$t1 e \$t2 sejam inteiros A, B, I e temp, respectivamente.

2.26.3 [5] <§2.7> Para os loops escritos em assembly MIPS acima, suponha que o registrador \$t1 seja inicializado com o valor N. Quantas instruções MIPS são executadas?

2.27 [5] <§2.7> Traduza o código C para o código assembly do MIPS. Use um número mínimo de instruções. Suponha que os valores de a, b, i e j estejam nos registradores \$s0, \$s1, \$t0, \$t1, respectivamente. Além disso, suponha que o registrador \$s2 mantenha o endereço de base do array D.

```

for(i = 0; i < a; i++)
    for(j = 0; j < b; j++)
        D[4*j] = i + j;

```

2.28 [5] <§2.7> Quantas instruções MIPS são necessárias para implementar o código C do Exercício 2.27? Se as variáveis a e b forem inicializadas como 10 e 1 e todos os elementos de D forem inicialmente 0, qual é o número total de instruções MIPS que são executadas para completar o loop?

2.29 [5] <§2.7> Traduza esses loops para C. Suponha que o inteiro i em nível de C seja mantido no registrador \$t1, \$s2 mantenha o inteiro em nível de C chamado result, e \$s0 mantenha o endereço de base do inteiro MemArray.

```

        addi $t1, $0, $0
LOOP: lw    $s1, 0($s0)
        add  $s2, $s2, $s1
        addi $s0, $s0, 4
        addi $t1, $t1, 1
        slti $t2, $t1, 100
        bne  $t2, $s0, LOOP

```

2.30 [5] <§2.7> Reescreva o loop do Exercício 2.29 para reduzir o número de instruções MIPS executadas.

2.31 [5] <§2.8> Implemente o seguinte código C em assembly MIPS. Qual é o número total de instruções MIPS necessárias para executar a função?

```

int fib(int n){
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}

```

2.32 [5] <§2.8> As funções normalmente podem ser implementadas pelos compiladores “em linha”. Uma função em linha é quando o corpo da função é copiado para o espaço do programa, permitindo que o overhead da chamada de função seja eliminado. Implemente uma versão “em linha” do código C acima em assembly do MIPS. Qual é a redução no número total de instruções assembly do MIPS necessárias para completar a função? Suponha que a variável C, n, seja inicializada como 5.

2.33 [5] <§2.8> Para cada chamada de função, mostre o conteúdo da pilha após

a chamada de função ser feita. Suponha que o ponteiro de pilha esteja originalmente no endereço 0x7fffffff e siga as convenções de registrador especificadas na [Figura 2.11](#).

2.34 Traduza a função *f* para a linguagem assembly do MIPS. Se você precisar usar os registradores de \$t₀ até \$t₇, use primeiro os registradores de número mais baixo. Suponha que a declaração de função para *func* seja “int *f*(int *a*, int *b*);”. O código para a função *f* é o seguinte:

```
int f(int a, int b, int c, int d){  
    return func(func(a,b),c+d);  
}
```

2.35 [5] <§2.8> Podemos usar a otimização “tail-call” nesta função? Se negativo, explique por que não. Se afirmativo, qual é a diferença no número de instruções executadas em *f* com e sem a otimização?

2.36 [5] <§2.8> Imediatamente antes que a sua função *f* do Exercício 2.34 retorne, o que sabemos sobre o conteúdo dos registradores \$t₅, \$s₃, \$ra e \$sp? Lembre-se de que sabemos o conteúdo da função *f*, mas, para a função *func*, só conhecemos sua declaração.

2.37 [5] <§2.9> Escreva um programa em linguagem assembly do MIPS para converter uma string de números ASCII contendo strings decimais inteiras positivas e negativas para um inteiro. Seu programa deverá esperar que o registrador \$a₀ contenha o endereço de uma string terminada em nulo, contendo alguma combinação dos dígitos de 0 até 9. Seu programa deverá calcular o valor inteiro equivalente a essa string de dígitos, depois colocar o número no registrador \$v₀. Se um caractere que não seja um dígito aparecer em qualquer lugar da string, seu programa deverá parar com um valor -1 no registrador \$v₀. Por exemplo, se o registrador \$a₀ apontar para uma sequência de três bytes 50_{dec}, 52_{dec}, 0_{dec} (a string “24” terminada em nulo), então, quando o programa terminar, o registrador \$v₀ deverá conter o valor 24_{dec}.

2.38 [5] <§2.9> Considere o código a seguir:

```
lbu $t0, 0($t1)
sw $t0, 0($t2)
```

Suponha que o registrador $\$t1$ contenha o endereço $0x1000\ 0000$ e o registrador $\$t2$ contenha o endereço $0x1000\ 0010$. Observe que a arquitetura MIPS utiliza o endereçamento big-endian. Suponha que os dados (em hexadecimal) no endereço $0x1000\ 0000$ sejam: $0x11223344$. Que valor é armazenado no endereço apontado pelo registrador $\$t2$?

2.39 [5] <§2.10> Escreva o código assembly MIPS que cria a constante de 32 bits $0010\ 0000\ 0000\ 0001\ 0100\ 1001\ 0010\ 0100_{bin}$ e armazena esse valor no registrador $\$t1$.

2.40 [5] <§§2.6, 2.10> Se o valor atual do PC é $0x00000000$, você pode usar uma única instrução jump para chegar ao endereço do PC, como mostra o Exercício 2.39?

2.41 [5] <§§2.6, 2.10> Se o valor atual do PC é $0x00000600$, você pode usar uma única instrução branch para chegar ao endereço do PC, como mostra o Exercício 2.39?

2.42 [5] <§§2.6, 2.10> Se o valor atual do PC é $0x1FFFF000$, você pode usar uma única instrução branch para chegar ao endereço do PC, como mostra o Exercício 2.39?

2.43 [5] <§2.11> Escreva o código assembly MIPS para implementar o seguinte código em C:

```
lock(1k);
shvar = max(shvar,x);
unlock(1k);
```

Suponha que o endereço da variável $1k$ esteja em $\$a0$, o endereço da variável $shvar$ esteja em $\$a1$ e o valor da variável x esteja em $\$a2$. Sua seção crítica não deverá conter quaisquer chamadas de função. Use instruções ll/sc a fim de implementar a operação $lock()$, e a operação $unlock()$ é simplesmente uma instrução store comum.

2.44 [5] <§2.11> Repita o Exercício 2.43, mas desta vez use ll/sc para realizar uma atualização atômica da variável $shvar$ diretamente, sem usar $lock()$ e

`unlock()`. Observe que, neste problema, não existe uma variável 1k.

2.45 [5] <§2.11> Usando o seu código do Exercício 2.43 como exemplo, explique o que acontece quando dois processadores começam a executar essa seção crítica ao mesmo tempo, supondo que cada processador executa exatamente uma instrução por ciclo.

2.46 Suponha que, para determinado processador, o CPI das instruções aritméticas seja 1, o CPI das instruções load/store seja 10 e o CPI das instruções branch seja 3. Suponha que um programa tenha os seguintes desmembramentos de instrução: 500 milhões de instruções aritméticas, 300 milhões de instruções load/store e 100 milhões de instruções branch.

2.46.1 [5] <§2.19> Suponha que instruções aritméticas novas, mais poderosas, sejam acrescentadas ao conjunto de instruções. Na média, com o uso dessas instruções aritméticas mais poderosas, podemos reduzir em 25% o número de instruções aritméticas necessárias para executar um programa, e em apenas 10% o custo do aumento do tempo de ciclo de clock. Essa é uma boa escolha de projeto? Por quê?

2.46.2 [5] <§2.19> Suponha que achemos um meio de dobrar o desempenho das instruções aritméticas. Qual é o ganho de velocidade geral de nossa máquina? E se achássemos um modo de melhorar o desempenho das instruções aritméticas em 10 vezes?

2.47 Suponha que, para determinado programa, 70% das instruções executadas sejam aritméticas, 10% sejam load/store e 20% sejam branch.

2.47.1 [5] <§2.19> Dada a mistura de instruções e a suposição de que uma instrução aritmética usa 2 ciclos, uma instrução load/store usa 6 ciclos e uma instrução branch usa 3 ciclos, ache o CPI médio.

2.47.2 [5] <§2.19> Para uma melhoria de 25% no desempenho, quantos ciclos, em média, uma instrução aritmética pode usar se instruções load/store e branch não tiverem qualquer melhoria?

2.47.3 [5] <§2.19> Para uma melhoria de 50% no desempenho, quantos ciclos, em média, uma instrução aritmética pode usar se instruções load/store e branch não tiverem qualquer melhoria?

Respostas das Seções “Verifique você mesmo”

§2.2, página 57: MIPS, C, Java

§2.3, página 62: 2) Muito lento

§2.4, página 68: 2) -8_{dec}

§2.5, página 75: 4) sub \$s2, \$s0, \$s1

§2.6, página 78: Ambos. AND com um padrão de máscara de 1s deixa 0s em todo lugar menos no campo desejado. O deslocamento à esquerda pela quantidade correta remove os bits da esquerda do campo. O deslocamento à direita pela quantidade apropriada coloca o campo nos bits mais à direita da palavra, com 0s no restante da palavra. Observe que AND deixa o campo onde ele estava originalmente e o par deslocado move o campo para a parte mais à direita da palavra.

§2.7, página 83: I. Todos são verdadeiros. II. 1).

§2.8, página 92: Ambos são verdadeiros.

§2.9, página 97: I. 2) II. 3)

§2.10, página 104: I. 4) $+ -128K$. II. 6) um bloco de 256M. III. 4) s11

§2.11, página 107: Ambos são verdadeiros.

§2.12, página 115: 4) Independência de máquina.

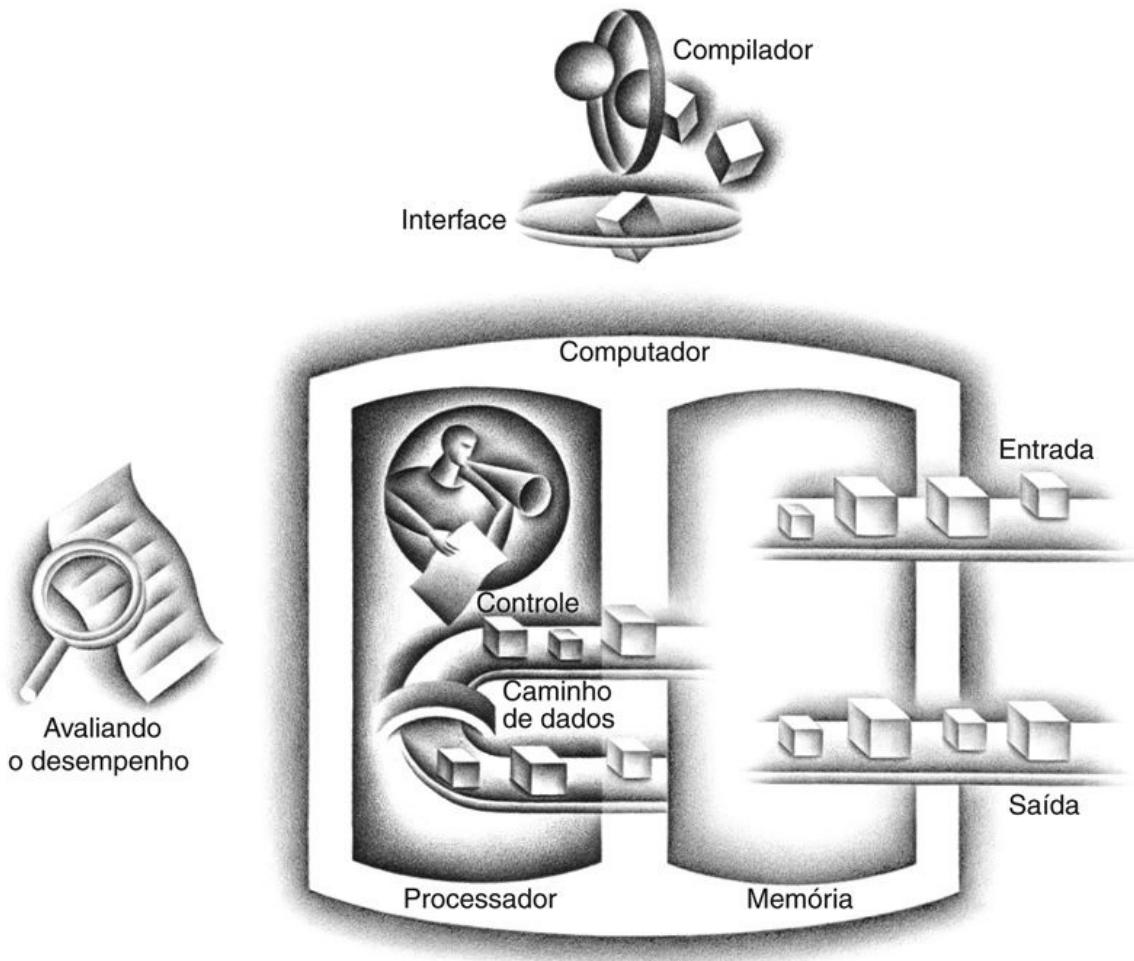
Aritmética Computacional

A precisão numérica é a própria alma da ciência.

*Sir D'arcy Wentworth Thompson
On Growth and Form, 1917*

- 3.1 Introdução
- 3.2 Adição e subtração
- 3.3 Multiplicação
- 3.4 Divisão
- 3.5 Ponto flutuante
- 3.6 Paralelismo e aritmética computacional: paralelismo subword
- 3.7 Vida real: Extensões SIMD streaming e extensões avançadas de vetor no x86
- 3.8 Mais rápido: Paralelismo subword e multiplicação matricial
- 3.9 Falácia e armadilhas
- 3.10 Comentários finais
- 3.11 Exercícios

Os cinco componentes clássicos de um computador



3.1. Introdução

As palavras do computador são compostas de bits; assim, podem ser representadas como números binários. O [Capítulo 2](#) mostra que os inteiros podem ser representados em formato decimal ou binário, mas e quanto aos outros números que ocorrem normalmente? Por exemplo:

- Como são representadas frações e outros números reais?

- O que acontece se uma operação cria um número maior do que poderia ser representado?
- E por trás de todas essas perguntas existe um mistério: como o hardware realmente multiplica ou divide números?

O objetivo deste capítulo é desvendar esses mistérios, incluindo a representação dos números, algoritmos aritméticos, hardware que acompanha esses algoritmos e as implicações de tudo isso para os conjuntos de instruções. Essas ideias podem ainda explicar truques que você já pode ter encontrado nos computadores. Além do mais, mostramos como usar esse conhecimento para tornar muito mais velozes os programas que fazem uso intenso da aritmética.

3.2. Adição e subtração

Subtração: o companheiro esquisito da adição

No 10, Top Ten Courses for Athletes at a Football Factory, David Letterman et al., Book of Top Ten Lists, 1990

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.

$$\begin{array}{r}
 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0111_{\text{bin}} = 7_{\text{dec}} \\
 + \quad 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0110_{\text{bin}} = 6_{\text{dec}} \\
 \hline
 = \quad 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1101_{\text{bin}} = 13_{\text{dec}}
 \end{array}$$

Os 4 bits à direita fazem toda a ação; a Figura 3.1 mostra as somas e os carries. Os carries aparecem entre parênteses, com as setas mostrando como são passados.

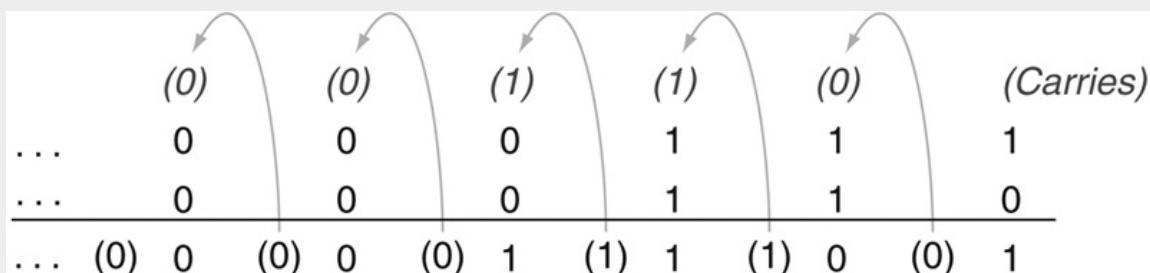


FIGURA 3.1 Adição binária, mostrando *carries* da direita para a esquerda.

O bit mais à direita adiciona 1 a 0, resultando em uma soma de 1 e um *carry out* de 0 para esse bit. Logo, a operação para o segundo dígito da direita é $0 + 1 + 1$. Isso gera uma soma de 0 e um *carry out* de 1 para esse bit. O terceiro dígito é a soma de $1 + 1 + 1$, resultando em um *carry out* de 1 e uma soma de 1 para esse dígito. O quarto bit é $1 + 0 + 0$, tendo uma soma de 1 e nenhum *carry*.

Resposta

A subtração de 6_{dec} de 7_{dec} pode ser feita diretamente:

$$\begin{array}{r}
 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0111_{\text{bin}} = 7_{\text{dec}} \\
 - \quad 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0110_{\text{bin}} = 6_{\text{dec}} \\
 \hline
 = \quad 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0001_{\text{bin}} = 1_{\text{dec}}
 \end{array}$$

ou por meio da soma, usando a representação de complemento de dois de -6 :

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{bin}} = 7_{\text{dec}} \\
 + \quad 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{\text{bin}} = -6_{\text{dec}} \\
 \hline
 = \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{bin}} = 1_{\text{dec}}
 \end{array}$$

Já dissemos que o overflow ocorre quando o resultado de uma operação não pode ser representado com o hardware disponível, nesse caso, uma palavra 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 não 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 não pode ocorrer na adição e na subtração, ainda não respondemos como detectar quando ele *ocorre*. 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 33° 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. Logo, o overflow ocorre quando se somam dois números positivos, e a soma é negativa ou vice-versa. 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 3.2](#) 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 3.2 Condições de overflow para adição e subtração.

Acabamos de ver como detectar o overflow para os números em complemento de dois em um computador. E com relação aos inteiros sem sinal? Os inteiros sem sinal normalmente são usados para endereços de memória em que 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, sem importar o tipo das variáveis. No entanto, os compiladores Fortran do MIPS apanham as instruções aritméticas apropriadas, dependendo do tipo dos operandos.

O **Apêndice B** descreve o hardware que realiza a adição e subtração, que é chamado de **Unidade Lógica e Aritmética** ou **ALU**

Unidade Lógica e Aritmética (ALU)

Hardware que realiza adição, subtração e normalmente operações lógicas como AND e OR.

Detalhamento

Uma fonte de confusão constante para addiu é o seu nome e o que acontece com seu campo imediato. O u significa unsigned (sem sinal), o que significa

que a adição não pode causar uma exceção de overflow. Porém, o campo imediato de 16 bits é estendido por sinal para 32 bits, assim como addi, slti e sltiu. Assim, o campo imediato é sinalizado, mesmo que a operação seja “unsigned”.

Interface hardware/software

O projetista de computador precisa decidir como tratar overflows aritméticos. Embora algumas linguagens como C e Java 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.

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, a fim de invocar a rotina apropriada para essa exceção. O endereço interrompido é salvo de modo que, em algumas situações, o programa possa continuar após o código corretivo ser executado. (A Seção 4.9 abrange as exceções com mais detalhes; o Capítulo 5 descreve 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* (`mfc0`) é usada a fim de 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*.

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.)

Resumo

A questão principal desta seção é que, independentemente da representação, o tamanho finito da palavra dos computadores significa que as operações aritméticas podem criar resultados muito grandes para caber nesse tamanho de palavra 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 de 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.

Algumas linguagens de programação permitem a aritmética de inteiros em complemento de dois com variáveis declaradas com um byte e meio, enquanto MIPS tem apenas operações aritméticas de inteiros sobre palavras completas. Conforme vimos no [Capítulo 2](#), MIPS não possui operações de transferência de dados para bytes e halfwords. Que instruções do MIPS deveriam ser geradas para as operações aritméticas de byte e halfword?

1. Load com 1bu, 1hu; aritmética com add, sub, mult, div; depois, armazenamento usando sb, sh.
2. Load com 1b, 1h; aritmética com add, sub, mult, div; depois, armazenamento usando sb, sh.
3. Load com 1b, 1h; 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

Um recurso que geralmente não é encontrado em microprocessadores de uso geral é a *saturação* de operações. A saturação significa que, quando um cálculo gera overflow, o resultado é definido para o maior número positivo ou o maior número negativo, ao invés de um cálculo de módulo, como na aritmética do complemento de dois. A saturação provavelmente é o que você deseja para operações com mídia. Por exemplo, o botão de volume em um aparelho de rádio seria frustrante se, quando girado, o volume ficasse continuamente mais alto por um tempo e depois, repentinamente, muito baixo. Um botão com saturação pararia no volume mais alto, não importa o quanto você o girasse. As extensões de multimídia para os conjuntos de instruções padrão geralmente oferecem aritmética com saturação.

Detalhamento

MIPS pode interceptar o overflow, porém, diferente de muitos outros computadores, não existe desvio condicional para testar o overflow. Uma sequência de instruções MIPS pode descobrir o overflow. Para a adição com sinal, a sequência é a seguinte (veja o *Detalhamento* no Capítulo 2, para obter uma descrição da instrução xor):

```
addu $t0, $t1, $t2 # $t0 = soma, mas não intercepta
xor $t3, $t1, $t2 # Verifica se os sinais diferem
slt $t3, $t3, $zero # $t3 = 1 se os sinais diferirem
bne $t3, $zero, No_overflow # $t1, $t2 sinais !=,
                           # logo, sem overflow
xor $t3, $t0, $t1 # sinais =; sinal da soma também combina?
                   # $t3 negativo se sinal da soma for diferente
slt $t3, $t3, $zero # $t3 = 1 se sinal da soma for diferente
bne $t3, $zero, Overflow # Todos os 3 sinais !=; vai para overflow
```

Para a adição sem sinal ($$t0 = $t1 + $t2$), o teste é

```
addu $t0, $t1, $t2      # $t0 = soma
nor $t3, $t1, $zero     # $t3 = NOT $t1
                      # (compl. 2 - 1:  $2^{32} - $t1 - 1$ )
slt $t3, $t3, $t2       #  $(2^{32} - $t1 - 1) < $t2$ 
                      #  $\Rightarrow 2^{32} - 1 < $t1 + $t2$ 
bne $t3,$zero,Overflow # se( $2^{32}-1 < $t1 + $t2$ ) vai para overflow
```

Detalhamento

No texto anterior, dissemos que você copia o EPC para um registrador por meio de `mfc0` e depois retorna ao código interrompido por meio de jump register. Isso leva a uma pergunta interessante: já que 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 neste dilema, os programadores MIPS concordaram em reservar os registradores $\$k_0$ e $\$k_1$ 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 (veja a Seção “Interface Hardware/Software”, na Seção 2.10), os compiladores também se abstêm do uso dos registradores $\$k_0$ e $\$k_1$, de modo 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.

Detalhamento

A velocidade da adição é aumentada, determinando-se o carry in para os bits de alta ordem mais cedo. Existem diversos esquemas para antecipar o carry, de modo que o pior que pode acontecer é uma função do \log_2 do número de bits no somador. Esses sinais antecipados são mais rápidos, pois percorrem menos portas na sequência, mas exigem mais portas para antecipar o carry apropriado. O mais comum é o *carry lookahead*, descrito na Seção B.6 do Apêndice B.

3.3. Multiplicação

Multiplicação é vexação, Divisão também é ruim; A regra de três me intriga, e a prática me deixa louco.

Anônimo, manuscrito Elisabetano, 1570

Agora que completamos a explicação de adição e subtração, estamos prontos para montar a operação mais difícil da multiplicação.

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 este exemplo decimal ao uso apenas dos dígitos 0 e 1. Multiplicando 1000_{dec} por 1001_{dec} :

Multiplicando	1000_{dec}
Multiplicador	\times

	1000
	0000
	0000
	1000

Produto	1001000_{dec}

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 sequencial do algoritmo e hardware de multiplicação

Esse projeto imita o algoritmo que aprendemos na escola; o hardware aparece na [Figura 3.3](#). Desenhamos o hardware de modo que os dados fluam de cima para baixo, para que fique mais semelhante à técnica do lápis e papel.

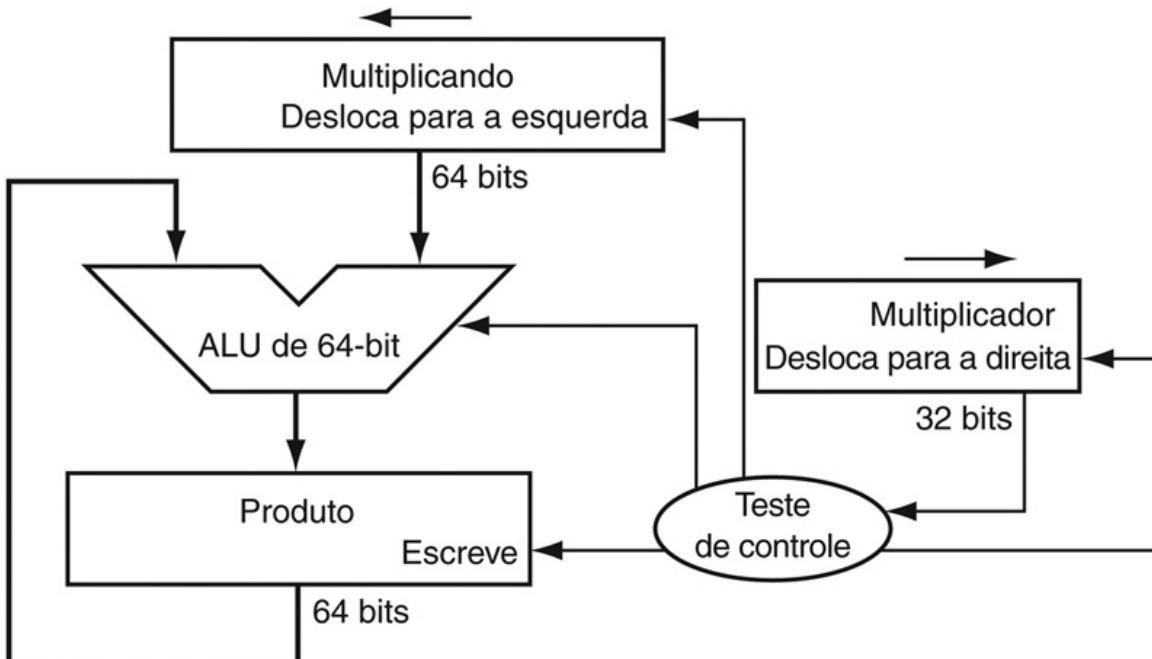


FIGURA 3.3 Primeira versão do hardware de multiplicação.

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 Apêndice B descreve as ALUs.) 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.

Vamos supor que o multiplicador esteja no registrador Multiplicador de 32 bits e que o registrador Produto de 64 bits esteja 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, de modo a alinhar o multiplicando com a soma sendo acumulada no registrador Produto de 64 bits.

A Figura 3.4 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 comuns do que a multiplicação. Como consequência, em muitas aplicações, a multiplicação pode demorar vários ciclos de clock sem afetar o desempenho de forma significativa. Mesmo assim, a lei de Amdahl ([Seção 1.10](#)) nos lembra que até mesmo uma frequência moderada para uma operação lenta pode limitar o desempenho.

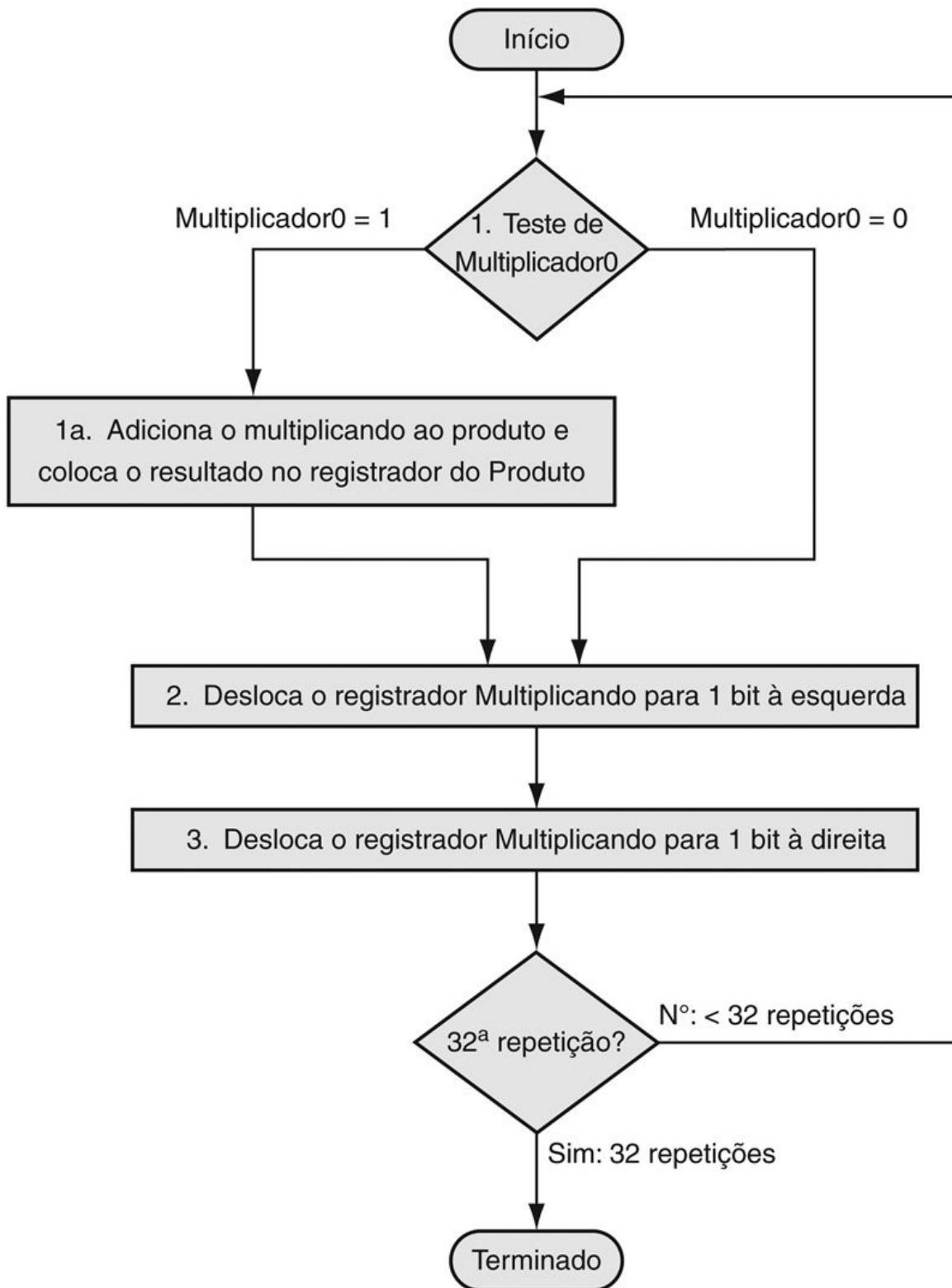


FIGURA 3.4 O primeiro algoritmo de multiplicação, usando o hardware mostrado na [Figura 3.3](#).

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 3.5](#) mostra o hardware revisado.

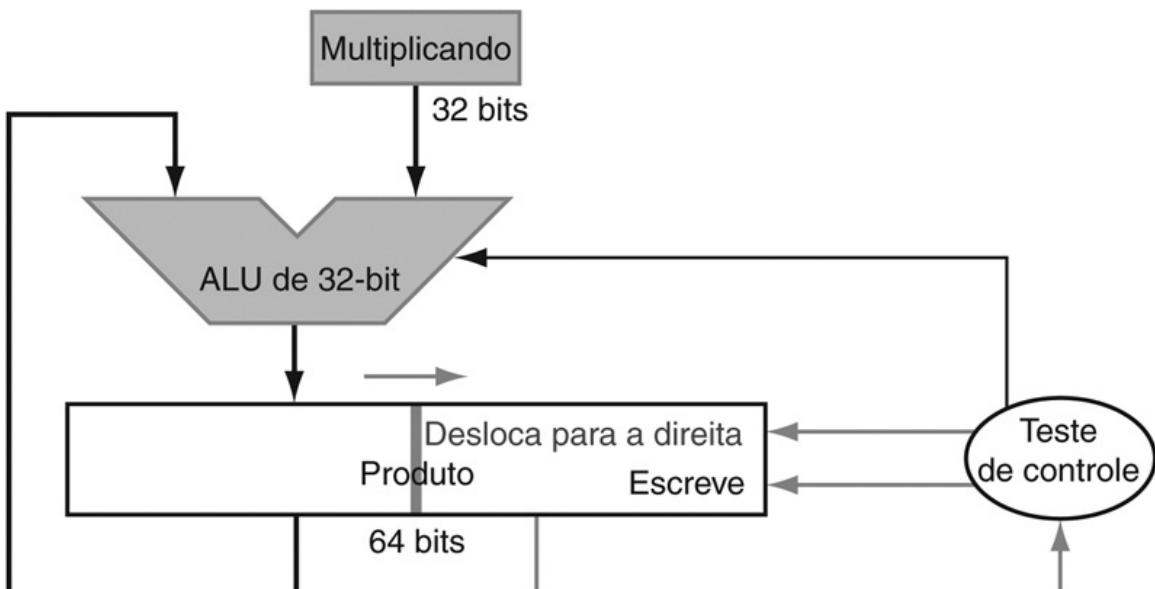


FIGURA 3.5 Versão refinada do hardware de multiplicação.

Compare com a primeira versão na [Figura 3.3](#). 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. (O registrador Produto, na realidade, deverá ter 65 bits, a fim de manter o carry do somador, mas ele aparece aqui como 64 bits para destacar a evolução da [Figura 3.3](#).)

Iteração	Passo	Multiplicador	Multiplicando	Produto
0	Valores iniciais	001 ⁽¹⁾	0000 0010	0000 0000
1	1a: $1 \Rightarrow$ Prod = Prod + Multiplicando	0011	0000 0010	0000 0010
	2: Desloca o Multiplicando à esquerda	0011	0000 0100	0000 0010
	3: Desloca o Multiplicador à direita	0001	0000 0100	0000 0010
2	1a: $1 \Rightarrow$ Prod = Prod + Multiplicando	0001	0000 0100	0000 0110
	2: Desloca o Multiplicando à esquerda	0001	0000 1000	0000 0110
	3: Desloca o Multiplicador à direita	0000	0000 1000	0000 0110
3	1: $0 \Rightarrow$ Nenhuma operação	0000	0000 1000	0000 0110
	2: Desloca o Multiplicando à esquerda	0000	0001 0000	0000 0110
	3: Desloca o Multiplicador à direita	0000	0001 0000	0000 0110
4	1: $0 \Rightarrow$ Nenhuma operação	0000	0001 0000	0000 0110
	2: Desloca o Multiplicando à esquerda	0000	0010 0000	0000 0110
	3: Desloca o Multiplicador à direita	0000	0010 0000	0000 0110

FIGURA 3.6 Exemplo de multiplicação usando o algoritmo da Figura 3.4.

O bit examinado para determinar a próxima etapa está em destaque.

Interface hardware/software

A substituição da 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 na potência de 2 por um deslocamento à esquerda.

Um algoritmo de multiplicação

Exemplo

Usando números de 4 bits para economizar espaço, multiplique $2_{\text{dec}} \times 3_{\text{dec}}$ ou $0010_{\text{bin}} \times 0011_{\text{bin}}$.

Resposta

A Figura 3.6 mostra o valor de cada registrador para cada uma das etapas rotuladas de acordo com a Figura 3.4, com o valor final de $0000\ 0110_{\text{bin}}$ ou 6_{dec} . O negrito é usado para indicar os valores de registrador que mudam nessa etapa e o bit circulado é aquele examinado para determinar a operação da próxima etapa.

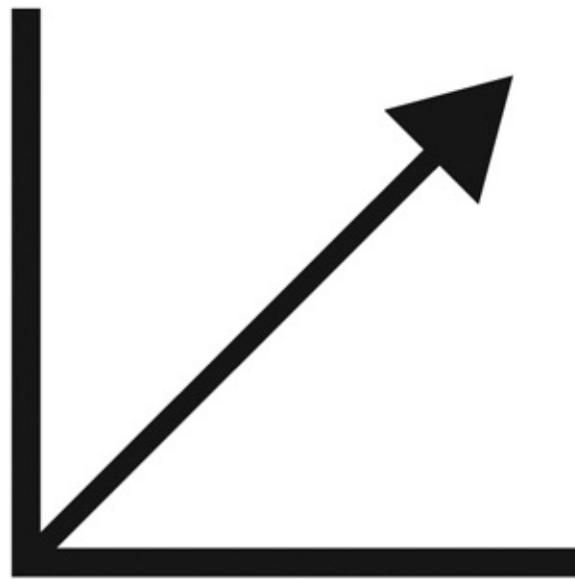
Multiplicação com sinal

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.

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 palavra 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. No início da multiplicação, já se sabe se o multiplicando deve ser somado ou não, 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.



LEI DE MOORE

Uma técnica simples seria conectar as saídas dos somadores à direita das entradas dos somados à esquerda, criando uma pilha de somadores com altura 32. Um modo alternativo de organizar essas 32 adições é em uma árvore paralela, como mostra a [Figura 3.7](#). Em vez de esperar 32 tempos de add, esperamos apenas o $\log_2(32)$ ou cinco tempos de add de 32 bits.

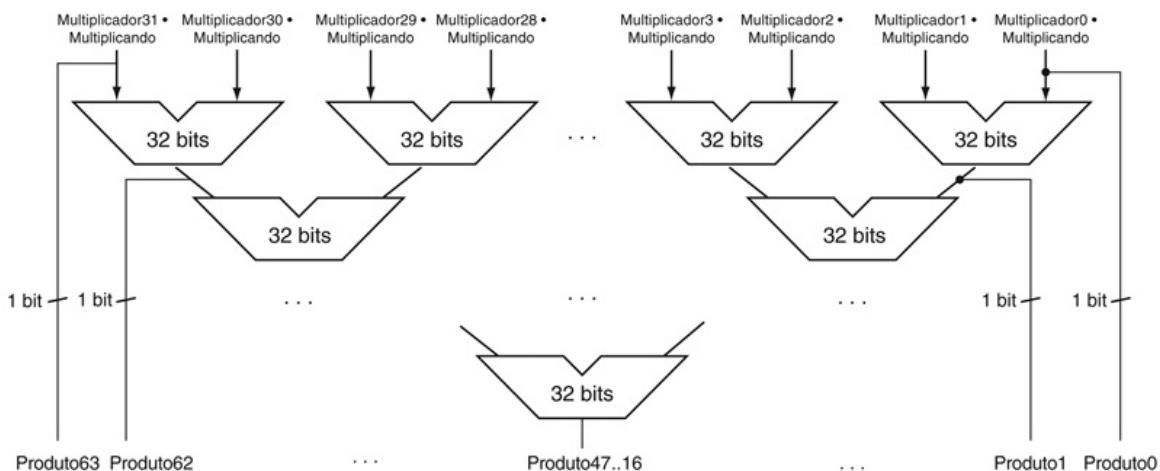
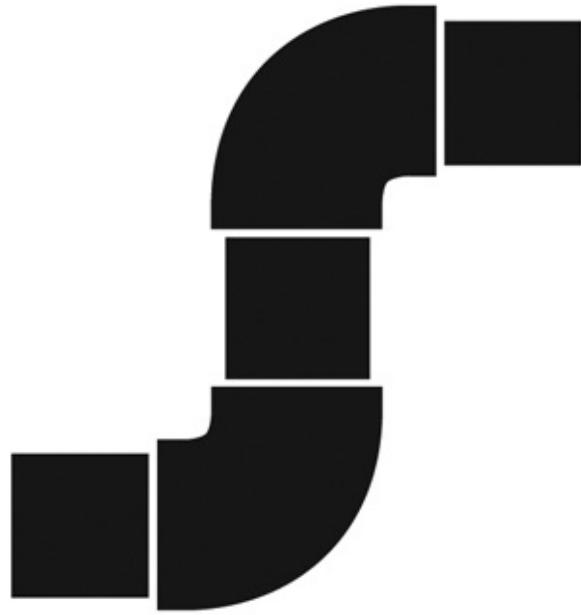


FIGURA 3.7 Hardware da multiplicação rápida.

Em vez de usar um único somador de 32 bits 31 vezes, esse hardware “desenrola o loop” para usar 31 somadores e depois os organiza para minimizar o atraso.

De fato, a multiplicação pode se tornar ainda mais rápida do que cinco tempos de add, veja uso de *somadores para salvar carry* ([Seção B.6 no Apêndice B](#)) e porque é fácil usar um **pipeline** nesse projeto para que possam ser realizadas muitas multiplicações simultaneamente ([Capítulo 4](#)).



PIPELINING

Multiplicação no MIPS

O MIPS oferece um par separado de registradores de 32 bits, a fim de 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 pseudoinstrução para multiplicar, que especifica três registradores de uso geral, criando instruções `mflo` e `mfhi` que colocam 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. Com muito mais hardware, podemos fazer as adições em **paralelo**, tornando-as muito mais rápidas.



PARALELISMO

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 a um registrador de uso geral, a fim de testar o overflow.

3.4. Divisão

Divide et impera.

Tradução do latim para “Dividir e conquistar”, máxima política antiga, citada por Machiavel 1532

A operação recíproca da multiplicação é a divisão, ainda menos frequente 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	1000_{dec}	Quociente
	$\overline{1001010}_{dec}$	Dividendo
	$\underline{-1000}$	
	10	
	101	
	1010	
	$\underline{-1000}$	
	10 _{dec}	Resto

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:

$$\text{Dividendo} = \text{Quociente} \times \text{Divisor} + \text{Resto}$$

em que o resto é menor do que o divisor. Raramente, os programas utilizam a instrução de divisão só para obter o resto, ignorando o quociente.

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.

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 3.8](#) 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, a fim de alinhá-lo com o dividendo. O registrador Resto é inicializado com o dividendo.

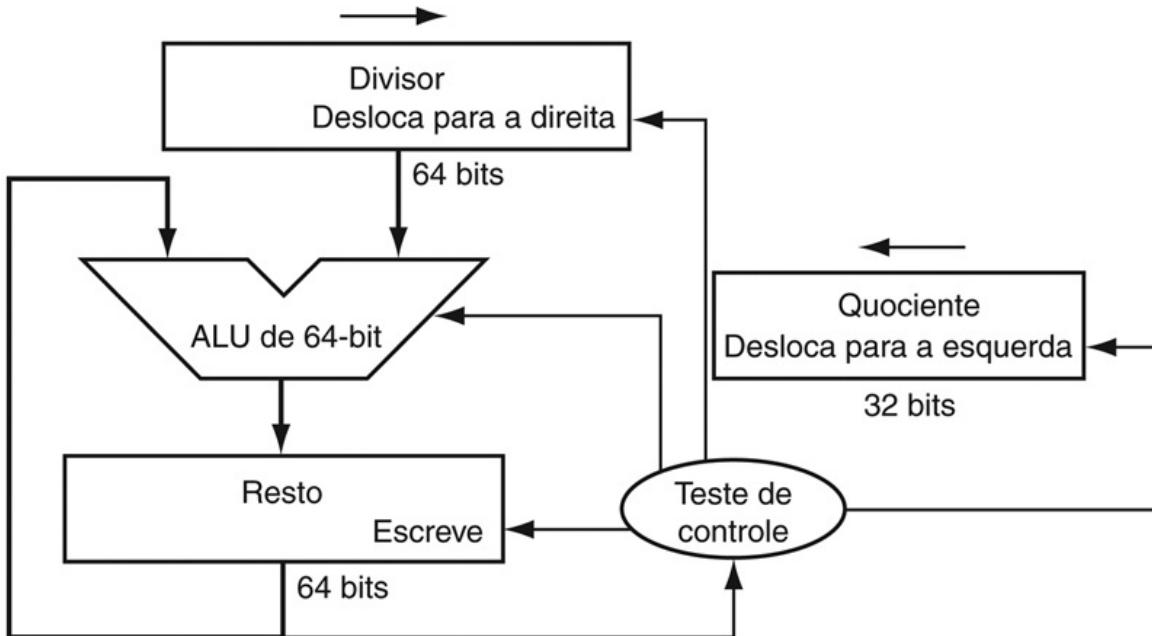


FIGURA 3.8 Primeira versão do hardware de divisão.

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.

A Figura 3.9 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, gerando 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.

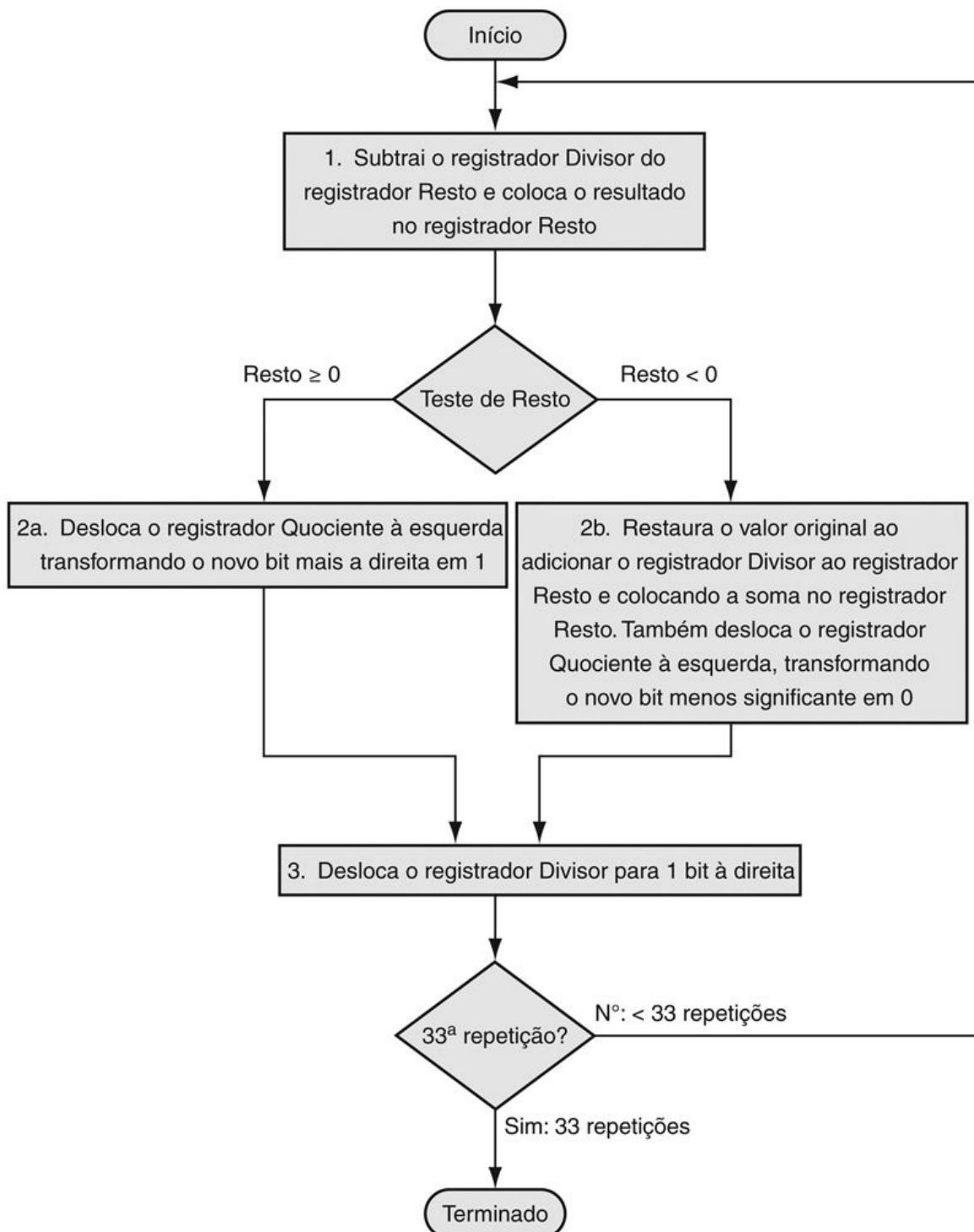


FIGURA 3.9 Um algoritmo de divisão, usando o hardware da [Figura 3.8](#).

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.

Iteração	Etapa	Quociente	Divisor	Resto
0	Valores iniciais	0000	0010 0000	0000 0111
1	1: Resto = Resto – Div	0000	0010 0000	①110 0111
	2b: Resto < 0 \Rightarrow +Div, sll 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	①111 0111
	2b: Resto < 0 \Rightarrow +Div, sll 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	①111 1111
	2b: Resto < 0 \Rightarrow +Div, sll 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	②000 0011
	2a: Resto \geq 0 \Rightarrow sll 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	②000 0001
	2a: Resto \geq 0 \Rightarrow sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Desloca Div direita	0011	0000 0001	0000 0001

FIGURA 3.10 Exemplo de divisão usando o algoritmo da Figura 3.9.

O bit examinado para determinar a próxima etapa está em destaque.

Um algoritmo de divisão

Exemplo

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} .

Resposta

A Figura 3.10 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.

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, ao observar onde existem partes não usadas dos registradores e somadores. A [Figura 3.11](#) mostra o hardware revisado.

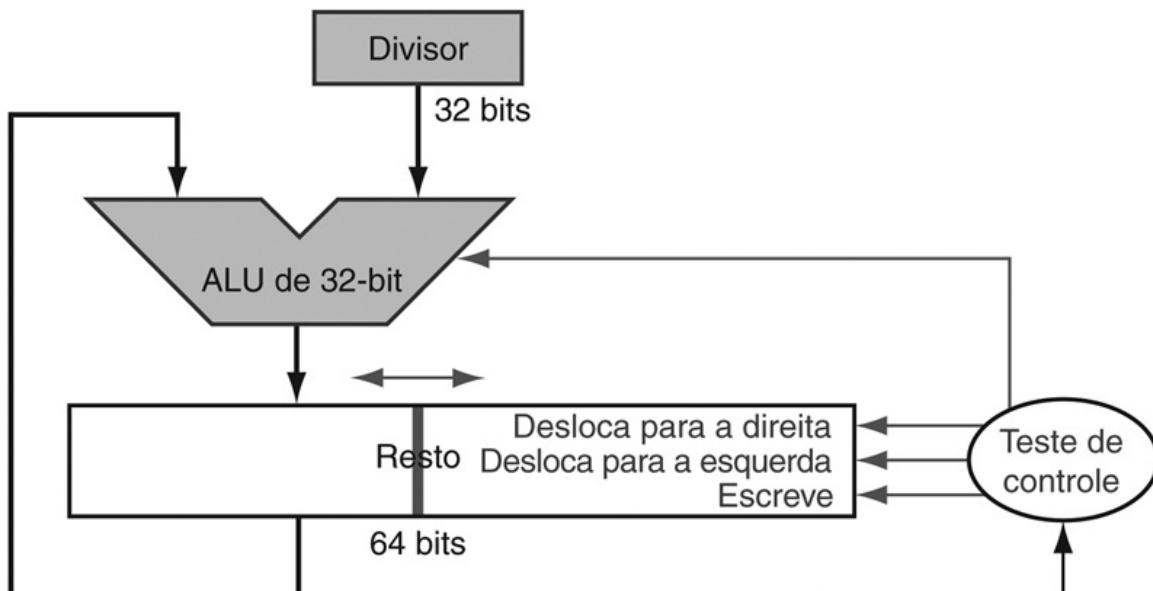


FIGURA 3.11 Uma versão melhorada do hardware de divisão.

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 3.8](#), os registradores ALU e Divisor são divididos ao meio, e o resto é deslocado à esquerda. Esta versão também combina o registrador Quociente com a metade direita do registrador Resto. (Assim como na [Figura 3.5](#), o registrador Resto na realidade deveria ter 65 bits para garantir que o carry do somador não se perca.)

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 complicaçāo da divisão com sinal é que também temos de definir o sinal do resto. Lembre-se de que a seguinte equaçāo 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:

$$-7 = -3 \times 2 + (-1) = -6 - 1$$

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

$$-(x \div y) \neq (-x) \div y$$

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:

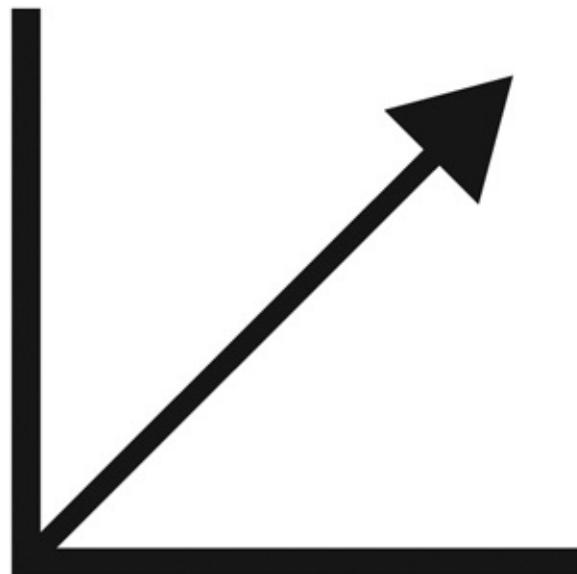
$$+7 \div -2 : \text{Quociente} = -3, \text{Resto} = +1$$

$$-7 \div -2 : \text{Quociente} = +3, \text{Resto} = -1$$

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.

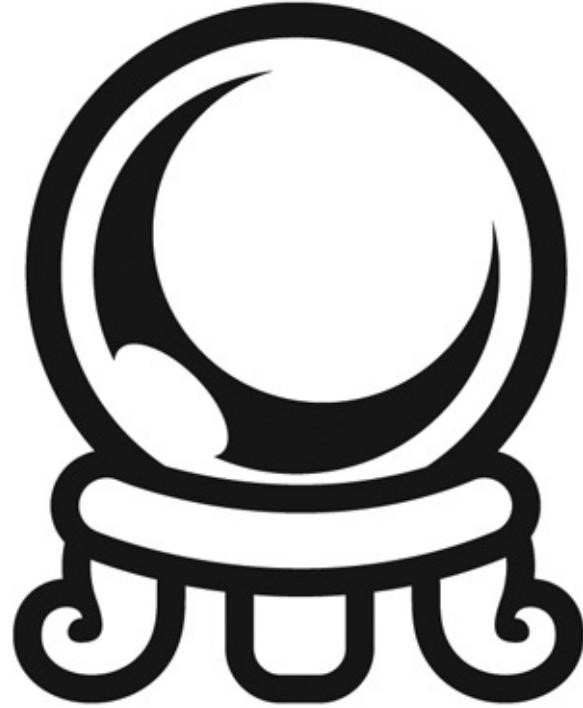
Divisão mais rápida

A **Lei de Moore** se aplica ao hardware de divisão e também à multiplicação, de modo que provavelmente podemos agilizar a divisão jogando hardware nela. Usamos muitos 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.



L E I D E M O O R E

Existem técnicas para produzir mais de um bit do quociente por etapa. A técnica de *divisão SRT* tenta **predizer** 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 previsões erradas. Um valor comum 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.



P R E D I Ç Ã O

A precisão desse método rápido depende da existência de valores apropriados na tabela de pesquisa. A falácia apresentada na [Seção 3.9](#) mostra o que pode acontecer se a tabela estiver incorreta.

Divisão no MIPS

Você já pode ter observado que o mesmo hardware sequencial pode ser usado para multiplicação e divisão nas [Figuras 3.5 e 3.11](#). 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 poderíamos esperar do algoritmo anterior, Hi contém o resto, e Lo 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. Aceleramos a divisão predizendo múltiplos bits do quociente e depois corrigindo erros de predição mais adiante. A Figura 3.12 resume os acréscimos à arquitetura MIPS das duas últimas seções.

Linguagem de assembly 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 detectado
	subtract unsigned	subu \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	Três operandos, overflow detectado
	add immediate unsigned	addiu \$s1,\$s2,100	\$s1 = \$s2 + 100	+ constante, overflow detectado
	move from coprocessor register	mfc0 \$s1,\$epc	\$s1 = \$epc	Copiar Exceção PC + regs especiais
	multiply	mult \$s2,\$s3	Hi, Lo = \$s2 × \$s3	Produto de 64-bit com sinal em Hi, Lo
	multiply unsigned	multu \$s2,\$s3	Hi, Lo = \$s2 × \$s3	Produto de 64-bit sem sinal 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 sem sinal e resto
Transferência de dados	move from Hi	mfhi \$s1	\$s1 = Hi	Costumava copiar de Hi
	move from Lo	mflo \$s1	\$s1 = Lo	Costumava copiar de Lo
	load word	lw \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Word da memória para o registrador
	store word	sw \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Word do registrador para a memória
	load half unsigned	lhu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword da memória para o registrador
	store half	sh \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Halfword do registrador para a memória
	load byte unsigned	lbu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte da memória para o registrador
	store byte	sb \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Byte do registrador para a memória
	load linked word	l1 \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Carrega word como primeira metade da troca atómica
	store conditional word	sc \$s1,20(\$s2)	Memory[\$s2+20]=\$s1:\$s1=0 or 1	Armazena word como primeira metade da troca atómica
Lógica	load upper immediate	lui \$s1,100	\$s1 = 100 * 2 ¹⁶	Carrega constante nos 16 bits altos
	AND	AND \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	Três operandos registradores; AND bit por bit
	OR	OR \$s1,\$s2,\$s3	\$s1 = \$s2 \$s3	Três operandos registradores; OR bit por bit
	NOR	NOR \$s1,\$s2,\$s3	\$s1 = ~ (\$s2 \$s3)	Três operandos registradores; NOR bit por bit
	AND immediate	andi \$s1,\$s2,100	\$s1 = \$s2 & 100	AND bit por bit com constante
	OR immediate	ori \$s1,\$s2,100	\$s1 = \$s2 100	OR bit por bit com constante
	shift left logical	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	Desloca à esquerda pela constante
	shift right logical	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	Desloca à direita pela constante
Desvio condicional	branch on equal	beq \$s1,\$s2,25	if(\$s1 == \$s2) go to PC + 4 + 100	Teste idêntico; desvio relativo ao PC
	branch on not equal	bne \$s1,\$s2,25	if(\$s1 != \$s2) go to PC + 4 + 100	Teste não-idêntico; relativo ao PC
	set on less than	slt \$s1,\$s2,\$s3	if(\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Comparação menos que; complementos de dois
	set less than immediate	slti \$s1,\$s2,100	if(\$s2 < 100) \$s1 = 1; else \$s1=0	Comparação < constante; complementos de dois
	set less than unsigned	sltu \$s1,\$s2,\$s3	if(\$s2 < \$s3) \$s1 = 1; else \$s1=0	Comparação menos que; números naturais
	set less than immediate unsigned	sltiu \$s1,\$s2,100	if(\$s2 < 100) \$s1 = 1; else \$s1 = 0	Comparação < constante; números naturais
	Pulo incondicional	jump j 2500	go to 10000	Pula para o endereço alvo
Pulo incondicional	jump register	jr \$ra	go to \$ra	Para troca, procedimento de retorno
	jump and link	jal 2500	\$ra = PC + 4; go to 10000	Para chamada de procedimento

FIGURA 3.12 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 linguagem de máquina do MIPS aparece no Guia de Referência do MIPS, no final deste livro.

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 um clock por etapa, é explorado ainda mais nos exercícios; o algoritmo aqui apresentado é chamado de divisão com *restauração*. Um terceiro algoritmo, que não salva o resultado da subtração se ele for negativo, é chamado algoritmo de divisão *sem o retorno esperado*. Ele tem em média menos um terço de operações aritméticas.

3.5. Ponto flutuante

A velocidade não o leva a lugar algum se você estiver na direção errada.

Provérbio americano

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:

$3,14159265\dots$ _{dec} (pi)

$2,71828\dots$ _{dec} (e)

$0,000000001$ _{dec} or $1,0$ _{dec} $\times 10^{-9}$ (segundos em um nanossegundo)

$3.155.760.000$ _{dec} or $3,15576$ _{dec} $\times 10^9$ (segundos em um século típico)

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 tem um único dígito à esquerda do ponto decimal. Um número na notação científica que não tem 0s à esquerda do ponto decimal é chamado de número **normalizado**, que é o modo normal como o escrevemos. Por exemplo, $1,0$ _{dec} $\times 10^{-9}$ está em notação científica normalizada, mas $0,1$ _{dec} $\times 10^{-8}$ e $10,0$ _{dec} $\times 10^{-10}$ não estão.

normalizado

Um número na notação de ponto flutuante que não possui 0s à esquerda do ponto decimal.

notação científica

Uma notação que apresenta números com um único dígito à esquerda do ponto decimal.

Assim como podemos mostrar números decimais em notação científica, também podemos mostrar números binários em notação científica:

$$1,0_{\text{bin}} \times 2^{-1}$$

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 à nossa necessidade. Como a base não é 10, também precisamos de um novo nome para o ponto decimal; *ponto*

binário servirá bem.

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

ponto flutuante

Aritmética computacional que representa os números em que o ponto binário não é fixo.

$$1,xxxxxxxx_{\text{bin}} \times 2^{yyyy}$$

(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 palavra, pois os 0s desnecessários são substituídos por dígitos reais à direita do ponto binário.

Representação em ponto flutuante

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 palavra fixo significa que você precisa tirar um bit de um para acrescentar um bit ao outro. Esta decisão 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.

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.

Os números em ponto flutuante normalmente são múltiplos do tamanho de uma palavra. A representação de um número em ponto flutuante MIPS aparece a seguir, em que *s* é o sinal do número de ponto flutuante (1 significa 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	8 bits																										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 grandes demais. 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.

overflow (ponto flutuante)

Uma situação em que um expoente positivo torna-se grande demais para caber 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.

underflow (ponto flutuante)

Uma situação em que um expoente negativo torna-se grande demais 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.

precisão dupla

Um valor de ponto flutuante representado em duas palavras de 32 bits.

precisão simples

Um valor de ponto flutuante representado em uma única palavra de 32 bits.

A representação de um número em ponto flutuante com precisão dupla utiliza duas palavras MIPS, como vemos a seguir, em que *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																																
s	Expoente											Fração																																																			
1 bit	11 bits											20 bits																																																			
fração (continuação)																																																															
32 bits																																																															

A precisão dupla do MIPS permite a representação de 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, em consequência da fração muito 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:

$$(-1)^S \times (1 + \text{Fração}) \times 2^E$$

em que 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 s1, s2, s3, ..., então o valor é

$$(-1)^S \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + \dots) \times 2^E$$

A Figura 3.13 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 a esses símbolos especiais. Quando o programador exibe os resultados, o programa exibirá um símbolo de infinito. (Para os que são matematicamente treinados, a finalidade do infinito é formar o fechamento topológico dos reais.)

Precisão simples		Precisão dupla		Objeto representado
Expoente	Fração	Expoente	Fração	
0	0	0	0	0
0	não zero	0	não zero	\pm número desnormalizado
1–254	qualquer coisa	1–2046	Anything	\pm número ponto flutuante
255	0	2047	0	\pm infinito
255	não zero	2047	não zero	Not a Number (NaN)

FIGURA 3.13 Codificação IEE 754 dos números de ponto flutuante.

Um bit de sinal separado determina o sinal. Os números desnormalizados são descritos no *Detalhamento* na página XX222. Essa informação também é encontrada na coluna 4 do Guia de Referência do MIPS, no final deste livro.

O IEEE 754 até mesmo possui um símbolo para o resultado de operações inválidas, como $0/0$ ou a subtração entre infinito e 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 de dois.)

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 grandes são maiores do que os números com expoentes menores,

desde que os dois expoentes tenham o mesmo sinal.

Exponentes negativos impõem um desafio à ordenação simplificada. Se usarmos o complemento de 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

(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

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 o 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 um expoente -1 é representado pelo padrão de bits do valor $-1 + 127_{dec}$ ou $126_{dec} = 0111\ 1110_{bin}$, e +1 é representado por $1 + 127$ ou $128_{dec} = 1000\ 0000_{bin}$. O bias do expoente para a precisão dupla é 1023. O expoente deslocado significa que o valor representado por um número em ponto flutuante é, na realidade:

$$(-1)^S \times (1 + \text{Fração}) \times 2^{(\text{Expoente} - \text{Bias})}$$

A faixa de números de precisão simples é, então, desde

$$\pm 1.000000000000000000000000 \times 2^{-126}$$

até

$$1.11111111111111111111111_{\text{bin}} \times 2^{+127}.$$

Vamos mostrar a representação.

Representação de ponto flutuante

Exemplo

Mostre a representação binária IEEE 754 para o número $-0,75_{\text{dec}}$ em precisão simples e dupla.

Resposta

O número $-0,75_{\text{dec}}$ também é

$$-3/4_{\text{dec}} \text{ ou } -3/2^2_{\text{dec}}$$

Ele também é representado pela fração binária

$$-11_{\text{bin}} / 2^2_{\text{dec}} \text{ ou } -0,11_{\text{bin}}$$

Em notação científica, o valor é

$$-0,11_{\text{bin}} \times 2^0$$

e, na notação científica normalizada, ele é

$$-1,1_{\text{bin}} \times 2^{-1}$$

A representação geral para um número de precisão simples é

$$(-1)^S \times (1 + \text{Fração}) \times 2^{(\text{Expoente}-127)}$$

Quando subtraímos o bias 127 do expoente de $-1,1_{\text{bin}} \times 2^{-1}$, o resultado é

$$(-1)^1 \times (1 + .1000000000000000000000000)_{\text{bin}} \times 2^{(126-127)}.$$

A representação binária de precisão simples de $-0,75_{\text{dec}}$, é, portanto:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bits 23 bits

A representação em precisão dupla é

$$(-1)^1 \times (1 + .100)_{\text{bin}} \times 2^{(1022-1023)}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	1	1	1	1	1	1	1	0	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	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 bit 11 bits 20 bits 32 bits

Agora, vamos experimentar na outra direção.

Convertendo ponto flutuante binário para decimal

Exemplo

Que número decimal é representado por este float de precisão simples?

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

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,

$$\begin{aligned} (-1)^S \times (1 + \text{Fração}) \times 2^{(\text{Exponente} - \text{Bias})} &= (-1)^1 \times (1 + 0,25) \times 2^{(129 - 127)} \\ &= -1 \times 1,25 \times 2^2 \\ &= -1,25 \times 4 \\ &= -5,0 \end{aligned}$$

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

Seguindo as diretrizes do IEEE, o comitê IEEE 754 foi modificado 20 anos depois do padrão para ver quais mudanças deveriam ser feitas ou se deveriam. O padrão revisado IEEE 754-2008 inclui quase todo o IEEE 754-1985 e acrescenta um formato de 16 bits (“meia precisão”) e um formato de 128 bits (“precisão quádrupla”). Nenhum hardware foi criado para dar suporte à precisão quádrupla, mas ele certamente virá. O padrão revisado também acrescenta aritmética de ponto flutuante, que os mainframes IBM implementaram.

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 0 à esquerda do ponto! 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. Mainframes IBM recentes admitem IEEE 754 além do formato hexa.

Adição em ponto flutuante

Vamos somar os números na notação científica manualmente, para ilustrar os problemas na adição em ponto flutuante: $9,999_{\text{dec}} \times 10^1 + 1,610_{\text{dec}} \times 10^{-1}$. Suponha que só possamos armazenar quatro dígitos decimais do significando e dois dígitos decimais do expoente.

Etapa 1. Para poder somar estes números corretamente, temos de alinhar o ponto decimal do número que possui o menor expoente. Logo, precisamos de uma forma do número menor, $1,610_{\text{dec}} \times 10^{-1}$, 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:

$$0,016 \times 10^1$$

Etapa 2. Em seguida, vem a adição dos significandos:

$$\begin{array}{r} 9,999_{\text{dec}} \\ + 0,016_{\text{dec}} \\ \hline 10,015_{\text{dec}} \end{array}$$

A soma é $10,015_{\text{dec}} \times 10^1$.

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 sequência de 9s, a soma não pode mais ser normalizada, sendo necessário realizar a etapa 3 novamente.

A [Figura 3.14](#) 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 223). Para o exemplo a seguir, lembre-se de que, para a precisão simples, o expoente máximo é 127, e o expoente mínimo é -126.

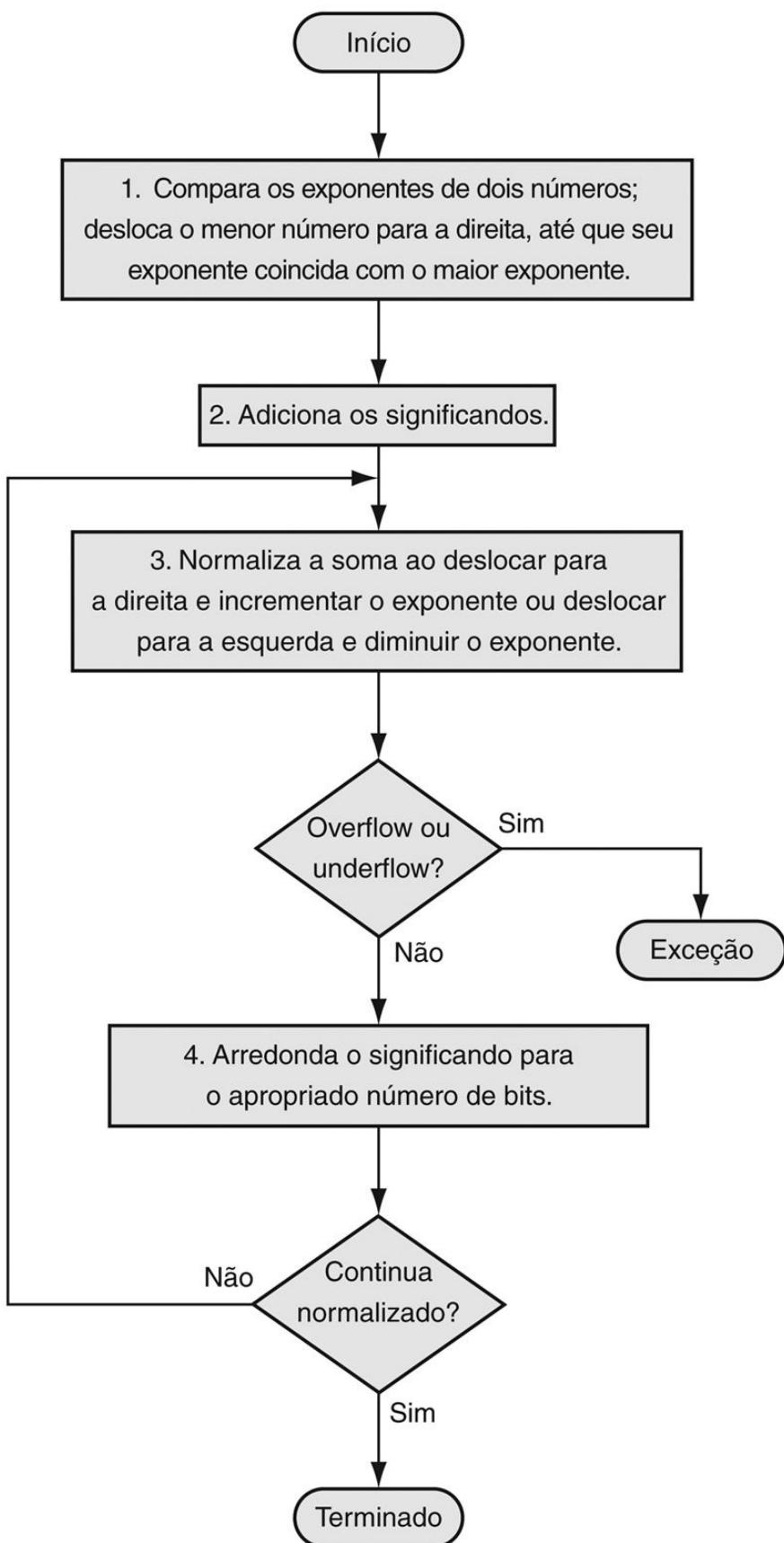


FIGURA 3.14 Adição de ponto flutuante.

O caminho normal é executar as etapas 3 e 4 uma vez, mas se o arredondamento fizer com que a soma não fique normalizada, temos de repetir a etapa 3.

Adição de ponto flutuante em binário

Exemplo

Tente somar os números $0,5_{\text{dec}}$ e $-0,4375_{\text{dec}}$ em binário usando o algoritmo da Figura 3.14.

Resposta

Primeiro, vejamos a versão binária dos dois números na notação científica normalizada, supondo que mantivemos 4 bits de precisão:

$$\begin{aligned} 0,5_{\text{dec}} &= 1/2_{\text{dec}} &= 1/2^1_{\text{dec}} \\ &= 0,1_{\text{bin}} &= 0,1_{\text{bin}} \times 2^0 &= 1,000_{\text{bin}} \times 2^{-1} \\ -0,4375_{\text{dec}} &= -7/16_{\text{dec}} &= -7/2^4_{\text{dec}} \\ &= -0,0111_{\text{bin}} &= -0,0111_{\text{bin}} \times 2^0 &= -1,110_{\text{bin}} \times 2^{-2} \end{aligned}$$

Agora, seguimos o algoritmo:

Etapa 1. O significando do número com o menor expoente ($-1,110_{\text{bin}} \times 2^{-2}$) 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 $127 \geq +4 \geq -126$, 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 em razão do arredondamento.

Essa soma é, então

$$\begin{aligned}
 1,000_{\text{bin}} \times 2^{-4} &= 0,0001000_{\text{bin}} = 0,0001_{\text{bin}} \\
 &= 1 / 2^4 &= 1 / 16_{\text{dec}} &= 0,0625_{\text{dec}}
 \end{aligned}$$

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 3.15](#) esboça a organização básica do hardware para a adição de ponto flutuante.

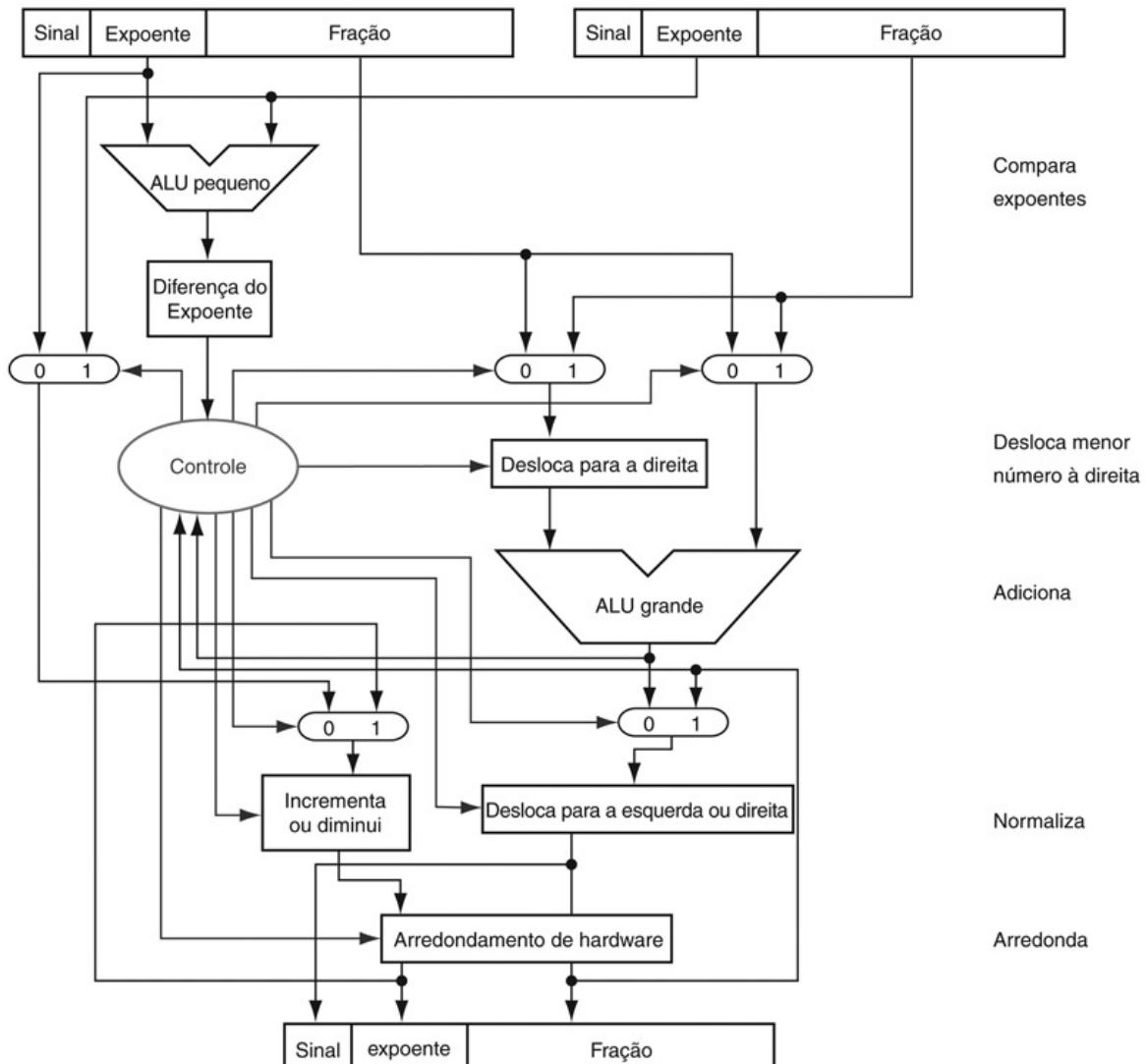


FIGURA 3.15 Diagrama de bloco de uma unidade aritmética dedicada à adição em ponto flutuante.

As etapas da Figura 3.14 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 normalizando novamente para produzir o resultado definitivo.

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 $-5 + 127 = 122$, então:

$$\text{Novo expoente} = 137 + 122 = 259$$

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, a fim de 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. Consideraremos 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 3.16](#), 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, vem 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).

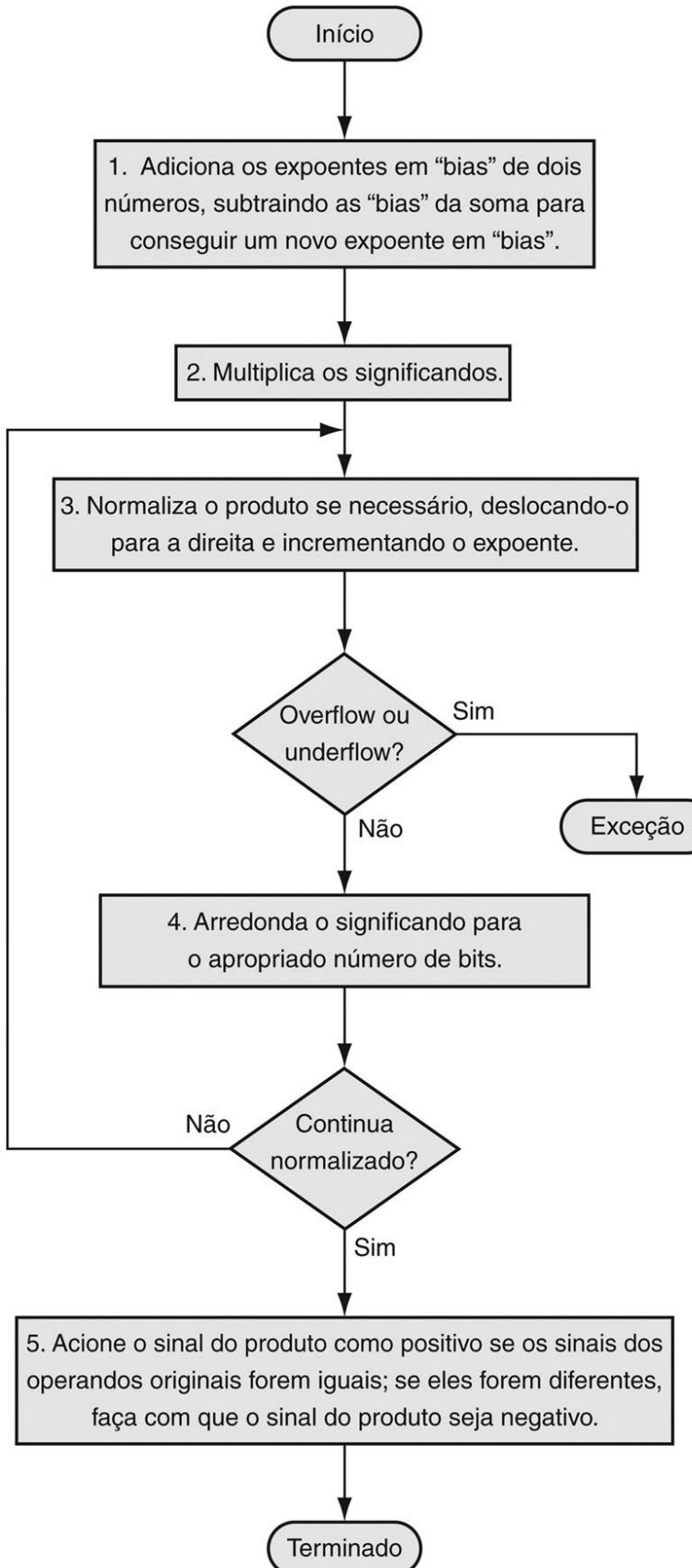


FIGURA 3.16 Multiplicação em ponto flutuante.

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.

Multiplicação em ponto flutuante em decimal

Exemplo

Vamos tentar multiplicar os números $0,5_{\text{dec}}$ e $-0,4375_{\text{dec}}$, usando as etapas na Figura 3.16.

Resposta

Em binário, a tarefa é multiplicar $1,000_{\text{bin}} \times 2^{-1}$ por $-1,110_{\text{bin}} \times 2^{-2}$.

Etapa 1 Somando os expoentes sem bias:

$$-1 + (-2) = -3$$

ou então, usando a representação deslocada:

$$(-1+127)+(-2+127)-127 = (-1-2)+(127+127-127) = -3+127 = 124$$

Etapa 2. Multiplicando os significandos:

$$\begin{array}{r}
 & 1,000_{\text{bin}} \\
 \times & 1,110_{\text{bin}} \\
 \hline
 & 0000 \\
 & 1000 \\
 & 1000 \\
 & \hline
 & 1000 \\
 \hline
 & 1110000_{\text{bin}}
 \end{array}$$

O produto é $1,110000_{\text{bin}} \times 2^{-3}$, mas precisamos mantê-lo com 4 bits, de modo que é $1,110_{\text{bin}} \times 2^{-3}$.

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), em que 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 (bc1t) 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 incluíram loads e stores separados para registradores de ponto flutuante: lwc1 e swc1. Os registradores base para transferências de dados de ponto flutuante, usados para endereços, 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. Assim, o par de registradores \$f2 e \$f3 de precisão simples também forma o registrador de precisão dupla chamado \$f2.

A Figura 3.17 resume a parte de ponto flutuante da arquitetura MIPS revelada neste capítulo, com as adições para dar suporte ao ponto flutuante mostradas em destaque. Semelhante à Figura 2.19 no Capítulo 2, mostramos a codificação dessas instruções na Figura 3.18.

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 sequenciais em palavras 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	bclt 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)
bclt	I	17	8	1	25				bclt 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 3.17 Arquitetura de ponto flutuante do MIPS revelada até aqui.

Ver Apêndice A, [Seção A.10](#), para obter mais detalhes. Essa informação também é encontrada na coluna 2 do Guia de Referência do MIPS, no final deste livro.

op(31:26):								
28-26	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/gez</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	lbu	lhu	lwr		
5(101)	sb	sh	sw				swr	
6(110)	lwcl							
7(111)	swcl							

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)	mfcl		cfc1		mtcl		ctcl	
1(01)	bcl.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.olt.f	c.ul.t.f	c.ole.f	c.ule.f
7(111)	c.sf.f	c.ngle.f	c.seq.f	c.ngl.f	c.lt.f	c.nge.f	c.le.f	c.ngt.f

FIGURA 3.18 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, lw 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 \Rightarrow `bc1.t`) ou falso (bit 16 = 1 \Rightarrow `bc1.f`). As instruções em negrito são descritas nos [Capítulos 2](#) neste capítulo, com o Apêndice A abordando todas as instruções. Essa informação também é encontrada na coluna 2 do Guia de Referência do MIPS, no final deste livro.

Interface hardware/software

Uma questão que os arquitetos de computador enfrentam no suporte à aritmética de ponto flutuante é se devem utilizar 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 interno.

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:
    lwc1 $f16,const5($gp) # $f16 = 5.0 (5.0 na memória)
    lwc1 $f18,const9($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 dividiriam 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`):

```

lwc1 $f18, const32($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                 # retorna
```

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

Exemplo

A maioria dos cálculos de ponto flutuante é realizada com precisão dupla. Vamos realizar uma multiplicação de matrizes $C = C + A * B$. Isso normalmente é chamado de DGEMM, de Double precision, General Matrix Multiply. Veremos versões de DGEMM novamente na Seção 3.8 e, mais adiante, nos Capítulos 4, 5 e 6. Vamos supor que C, A e B sejam matrizes quadradas com 32 elementos em cada dimensão.

```
void mm (double c[][], double a[][], double b[][])
{
    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)
                c[i][j] = c[i][j] + a[i][k] *b[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?

Resposta

Observe que $c[i][j]$ é usado no loop mais interno. Como o índice do loop é k , o índice não afeta $c[i][j]$, de modo que podemos evitar a leitura e o armazenamento de $c[i][j]$ a cada iteração. Em vez disso, o compilador lê $c[i][j]$ em um registrador fora do loop, acumula a soma dos produtos de $a[i][k]$ e $b[k][j]$ nesse mesmo registrador, e depois armazena a soma em $c[i][j]$, ao terminar o loop mais interno.

Mantemos o código mais simples, usando as pseudoinstruções 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; inicialize 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 $c[i][j]$ 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 c)
```

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 de 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]
```

Em seguida, somamos essa soma ao endereço base de c , dando o endereço de $c[i][j]$, e depois carregamos o número de precisão dupla $c[i][j]$ em $\$f4$:

```
addu $t2, $a0, $t2      # $t2 = endereço em bytes de c[i][j]
l.d  $f4, 0($t2)        # $f4 = 8 bytes de c[i][j]
```

As cinco instruções a seguir são praticamente idênticas às cinco últimas: calcular o endereço e depois ler o número de precisão dupla $b[k][j]$.

```
L3: sll $t0, $s2, 5      # $t0 = k * 25 (tamanho de linha de b)
    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 b[k][j]
    l.d  $f16, 0($t0)   # $f16 = 8 bytes de b[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 $a[i][k]$.

```
sll     $t0, $s0, 5      # $t0 = i * 25 (tamanho de linha de a)
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 a[i][k]
l.d    $f18, 0($t0)    # $f18 = 8 bytes de a[i][k]
```

Agora que carregamos todos os dados, finalmente estamos prontos para

realizar algumas operações em ponto flutuante! Multiplicamos os elementos de a e b localizados nos registradores $\$f18$ e $\$f16$, e depois acumulamos a soma em $\$f4$.

```
mul.d \$f16, \$f18, \$f16 # \$f16 = a[i][k] * b[k][j]
add.d \$f4, \$f4, \$f16    # f4 = c[i][j] + a[i][k] * b[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     # if (k != 32) vai para L3
s.d   $f4, 0($t2)      # c[i][j] = $f4
```

De modo semelhante, essas quatro últimas instruções 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     # if (j != 32) vai para L2
addiu $s0, $s0, 1      # $i = i + 1
bne   $s0, $t1, L1     # if (i != 32) vai para L1
...
```

A Figura 3.22, mais adiante, mostra o código em linguagem assembly x86 para uma versão ligeiramente diferente da DGEMM da Figura 3.21.

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.

Detalhamento

Somente 16 dos 32 registradores de ponto flutuante do MIPS puderam ser usados originalmente para operações de precisão simples: \$f0, \$f2, \$f4, ..., \$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, em que 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.

Detalhamento

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 *coprocessadores* e explicam o acrônimo para os loads de ponto flutuante no MIPS: lwc1 significa “load word to coprocessor 1” (“leia uma palavra para o coprocessador 1”), que é a unidade de ponto flutuante. (O coprocessador 0 trata da memória virtual, descrita no Capítulo 5.) 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 *coprocessador* reúne *acumulador* e *memória*.

Detalhamento

Conforme mencionamos na Seção 3.4, acelerar a divisão é mais complicado do que a multiplicação. Além de SRT, outra 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 $1/c$, que é então 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 abrange 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 em Celsius.

O segundo exemplo utiliza múltiplos arrays dimensionais, que não são admitidos explicitamente em Java. O Java permite arrays de arrays, mas cada array pode ter seu próprio tamanho, ao contrário dos arrays multidimensionais em C. Como os exemplos no Capítulo 2, uma versão em 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, em geral, são aproximações para um número que realmente não representam. O motivo é que existe uma variedade infinita de números reais entre, digamos, 0 e 1, porém 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 dois 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 de 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, podemos 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 para baixo em um comparativo com o último dígito da soma anterior.

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 com 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.

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.

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.

O IEEE 754 tem quatro modos de arredondamento: sempre arredondar para cima (para $+\infty$), sempre arredondar para baixo (para $-\infty$), truncar e arredondar para o par mais próximo. 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 beneficiando a 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 for ímpar, some um; se for 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 $0,50 \dots 00_{\text{dec}}$ e $0,50 \dots 01_{\text{dec}}$ 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 $5,01_{\text{dec}} \times 10^{-1}$ a $2,34_{\text{dec}} \times 10^2$ 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.

Detalhamento

As arquiteturas PowerPC, SPARC64, AMD SSE5 e Intel AVX oferecem uma única instrução que realiza multiplicação e adição sobre três registradores: $a = a + (b \times c)$. Obviamente, essa instrução permite um desempenho de ponto flutuante potencialmente mais alto para essa operação comum. Igualmente importante é que, em vez de realizar dois arredondamentos — depois da multiplicação e após a adição — que aconteceria com instruções separadas, a instrução de multiplicação-adição pode realizar um único arredondamento após a adição, o que aumenta a precisão da multiplicação-adição. Essas operações com um único arredondamento são chamadas **multiplicação-adição fundida**. Isso foi acrescentado no padrão IEEE 754-2008 revisado.

multiplicação adição fundida

Uma instrução de ponto flutuante que realiza uma multiplicação e uma adição, mas arredonda apenas depois da adição.

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

$$(-1)^S \times (1 + \text{Fração}) \times 2^{(\text{Expoente-Bias})}$$

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.

Colocando em perspectiva

Padrões de bits não possuem significado inerente. 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 palavra.

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 palavra. Os programadores precisam se lembrar desses limites e escrever programas de acordo.

Tipo C	Tipo Java	Transferências de dados	Operações
interface	int	lw, sw, lui	addu, addiu, subu, mult, div, AND, ANDi, OR, ORi, NOR,slt, slti
unsigned int	—	lw, sw, lui	addu, addiu, subu, mult, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu
char	—	lb, sb, lui	add, addi, sub, mult, div, AND, ANDi, OR, ORi, NOR,slt, slti
—	char	lh, sh, lui	addu, addiu, subu, multu, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu
float	float	lwc1, swc1	add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s
double	double	l.d, s.d	add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d

Interface hardware/software

No capítulo anterior, apresentamos as classes de armazenamento da linguagem de programação C (veja a Seção “Interface Hardware/Software” da Seção 2.7). 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 aqui e no Capítulo 2. Observe que Java omite inteiros sem sinal.

Verifique você mesmo

O padrão IEEE 754-2008 revisado acrescentou o formato de ponto flutuante de 16 bits com 5 bits de expoente. Qual seria o intervalo provável de números que ele poderia representar?

1. $1.0000\ 00 \times 2^0$ 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 *não ordenada* 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.)

Na tentativa de compactar cada bit de precisão de uma operação de ponto flutuante, o padrão permite que alguns números sejam representados de 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 *denormais* ou *subnormais*). Eles têm o mesmo expoente que zero, mas um significando diferente de zero. Eles permitem que um número diminua no significado até se tornar 0, chamado *underflow gradual*. Por exemplo, o menor número normalizado positivo de precisão simples é

$$1.0000\ 0000\ 0000\ 0000\ 0000\ 000_{\text{bin}} \times 2^{-126}$$

mas o menor número não normalizado de precisão simples é

$$0.0000\ 0000\ 0000\ 0000\ 0001_{\text{bin}} \times 2^{-126}, \text{ or } 1.0_{\text{bin}} \times 2^{-149}$$

Para a precisão dupla, a lacuna denormal vai de 1.0×2^{-1022} a 1.0×2^{-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 denormais no software de ponto flutuante portável. Além disso, se os programadores não esperarem os denormais, seus programas poderão surpreendê-los.

3.6. Paralelismo e aritmética computacional: paralelismo subword

Como todo microprocessador desktop, por definição, tem suas próprias telas gráficas, quando a quantidade de transistores aumentou, foi inevitável que seria acrescentado suporte para operações gráficas.

Muitos sistemas gráficos usavam inicialmente 8 bits para representar cada uma das três cores primárias, mais 8 bits para um local de um pixel. O acréscimo de alto-falantes e microfones para teleconferência e videogames sugeriu o suporte também para o som. Amostras de áudio precisam de mais de 8 bits de precisão, mas 16 bits são suficientes.

Todo microprocessador possui suporte especial, de modo que bytes e halfwords ocupam menos espaço quando armazenados na memória ([Seção 2.9](#)), mas, devido à pouca frequência das operações aritméticas sobre esses tamanhos de dados nos programas típicos para inteiros, houve pouco suporte além de transferências de dados. Os arquitetos reconheceram que muitas aplicações gráficas e de áudio realizariam a mesma operação sobre vetores desses dados. Partindo as cadeias de carry dentro de um somador de 128 bits, um processador poderia usar o **paralelismo** para realizar operações simultâneas sobre vetores curtos de dezesseis operandos de 8 bits, oito operandos de 16 bits, quatro operandos de 32 bits ou dois operandos de 64 bits. O custo desses somadores partidos era muito pequeno.



PARALELISMO

Dado que o paralelismo ocorre dentro de uma word larga, as extensões são classificadas como *paralelismo subword*. Isso também é classificado sob o nome mais genérico de *paralelismo em nível de dados*. Eles também foram chamados de vetor ou SIMD, de Single Instruction, Multiple Data (única instrução, múltiplos dados — [Seção 6.6](#)). A popularidade crescente das aplicações multimídia levou a instruções aritméticas que oferecem suporte a operações mais estreitas, que podem facilmente operar em paralelo.

Por exemplo, o ARM acrescentou mais de 100 instruções na extensão da instrução de multimídia NEON, para dar suporte ao paralelismo subword, que pode ser usado ou com ARMv7 ou com ARMv8. Ele acrescentou 256 bytes de novos registradores para o NEON, que podem ser vistos como 32 registradores com 8 bytes de largura ou 16 registradores com 16 bytes de largura. O NEON tem suporte para todos os tipos de dados de subword que você possa imaginar, *exceto* números de ponto flutuante com 64 bits:

- Inteiros com e sem sinal, com 8 bits, 16 bits, 32 bits e 64 bits
- Número de ponto flutuante com 32 bits

A [Figura 3.19](#) oferece um resumo das instruções NEON básicas.

Transferência de dados	Aritméticas	Lógicas/Comparação
VLDR.F32	VADD.F32, VADD{L,W}{S8,U8,S16,U16,S32,U32}	VAND.64, VAND.128
VSTR.F32	VSUB.F32, VSUB{L,W}{S8,U8,S16,U16,S32,U32}	VORR.64, VORR.128
VLD{1,2,3,4}.{I8,I16,I32}	VMUL.F32, VMULL{S8,U8,S16,U16,S32,U32}	VEOR.64, VEOR.128
VST{1,2,3,4}.{I8,I16,I32}	VMLA.F32, VMLAL{S8,U8,S16,U16,S32,U32}	VBIC.64, VBIC.128
VMOV.{I8,I16,I32,F32}, #imm	VMLS.F32, VMLSL{S8,U8,S16,U16,S32,U32}	VORN.64, VORN.128
VMVN.{I8,I16,I32,F32}, #imm	VMAX.{S8,U8,S16,U16,S32,U32,F32}	VCEQ.{I8,I16,I32,F32}
VMOV.{I64,I128}	VMIN.{S8,U8,S16,U16,S32,U32,F32}	VCGE.{S8,U8,S16,U16,S32,U32,F32}
VMVN.{I64,I128}	VABS.{S8,S16,S32,F32}	VCGT.{S8,U8,S16,U16,S32,U32,F32}
	VNEG.{S8,S16,S32,F32}	VCLE.{S8,U8,S16,U16,S32,U32,F32}
	VSHL.{S8,U8,S16,U16,S32,S64,U64}	VCLT.{S8,U8,S16,U16,S32,U32,F32}
	VSHR.{S8,U8,S16,U16,S32,S64,U64}	VTST.{I8,I16,I32}

FIGURA 3.19 Resumo das instruções NEON do ARM para paralelismo subword.

Usamos as chaves {} para mostrar variações opcionais das operações básicas: {S8,U8,8} representam inteiros de 8 bits com e sem sinal ou dados de 8 bits onde o tipo não importa, dos quais 16 cabem em um registrador de 128 bits; {S16,U16,16} representam inteiros de 16 bits com e sem sinal ou dados de 16 bits sem tipo, dos quais 8 cabem em um registrador de 128 bits; {S32,U32,32} representam inteiros de 32 bits com e sem sinal ou dados de 32 bits sem tipo, dos quais 4 cabem em um registrador de 128 bits; {S64,U64,64} representam inteiros de 64 bits com e sem sinal ou dados de 64 bits sem tipo, dos quais 2 cabem em um registrador de 128 bits; {F32} representa números de ponto flutuante de 32 bits com e sem sinal, dos quais 4 cabem em um registrador de 128 bits. Vector Load carrega uma estrutura com n elementos da memória para 1, 2, 3 ou 4 registradores NEON. Ele carrega uma única estrutura de n elementos para uma pista ([Seção 6.6](#)), e os elementos do registrador que não são carregados ficam inalterados. Vector Store escreve uma estrutura de n elementos na memória a partir de 1, 2, 3 ou 4 registradores NEON.

Detalhamento

Além de inteiros com e sem sinal, o ARM inclui o formato de “ponto fixo” de quatro tamanhos, chamados I8, I16, I32 e I64, dos quais 16, 8, 4 e 2 cabem em um registrador de 128 bits, respectivamente. Uma parte do ponto fixo é para a fração (à direita do ponto binário) e o restante dos dados é a parte inteira (à esquerda do ponto binário). O local do ponto decimal fica a critério do software. Muitos processadores ARM não possuem hardware de ponto flutuante e, portanto, as operações de ponto flutuante precisam ser realizadas

por rotinas de biblioteca. A aritmética de ponto fixo pode ser significativamente mais rápida que as rotinas de ponto flutuante no software, mas dão mais trabalho para o programador.

3.7. Vida real: Extensões SIMD streaming e extensões avançadas de vetor no x86

As instruções MMX (*MultiMedia eXtension*) e SSE (*Streaming SIMD Extension*) originais para o x86 incluíam operações semelhantes àquelas encontradas no ARM NEON. 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 de 64 bits que podem ser usados como operandos de ponto flutuante. A AMD expandiu o número para 16 registradores, chamados XMM, como parte do AMD64, que a Intel passou a chamar de EM64T para seu uso. A [Figura 3.20](#) resume as instruções SSE e SSE2.

Transferência de dados	Aritmética	Comparação
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD{SS/PS/SD/PD} xmm, mem/xmm	CMP{SS/PS/SD/PD}
	SUB{SS/PS/SD/PD} xmm, mem/xmm	
MOV{H/L}{PS/PD} xmm, mem/xmm	MUL{SS/PS/SD/PD} xmm, mem/xmm	
	DIV{SS/PS/SD/PD} xmm, mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX{SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD} mem/xmm	

FIGURA 3.20 As instruções de ponto flutuante SSE/SSE2 do x86.

xmm significa que um operando é um registrador SSE2 de 128 bits, e mem/xmm significa que o outro operando está na memória ou é um registrador SSE2. Usamos as chaves {} para mostrar variações opcionais das operações básicas: {SS} representa ponto flutuante de precisão *Scalar Single* ou um operando de 32 bits em um registrador de 128 bits; {PS} representa ponto flutuante de precisão *Packed Single* ou quatro operandos de 32 bits em um registrador de 128 bits; {SD} representa ponto flutuante de precisão *Scalar Double* ou um

operando de 64 bits em um registrador de 128 bits; {PD} representa ponto flutuante de precisão *Packed Double* ou dois operandos de 64 bits em um registrador de 128 bits; {A} significa que o operando de 128 bits é alinhado na memória; {U} significa que o operando de 128 bits é desalinhado na memória; {H} significa mover a metade alta (High) do operando de 128 bits; e {L} significa mover a metade baixa (Low) do operando de 128 bits.

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 colocados em um único registrador SSE2 de 128 bits: quatro de precisão simples e dois de precisão dupla. Assim, os 16 registradores de ponto flutuante para o SSE2 possuem na realidade 128 bits de largura. 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 (PS) ou dois de precisão dupla (PD).

Em 2011, a Intel dobrou a largura dos registradores novamente, agora denominados YMM, com *Advanced Vector Extensions* (AVX). Assim, uma única operação agora pode especificar oito operações de ponto flutuante com 32 bits ou quatro operações de ponto flutuante com 64 bits. As instruções SSE e SSE2 legadas, agora operam sobre os 128 bits mais baixos dos registradores YMM. Assim, para passar de operações de 128 bits para 256 bits, você prefixa a letra “v” (de vetor) na frente das operações em linguagem assembly SSE2 e depois usa os nomes de registrador YMM no lugar do nome do registrador XMM. Por exemplo, a instrução SSE2 para realizar duas multiplicações de ponto flutuante com 64 bits

```
addpd    %xmm0 , %xmm4
```

torna-se

```
vaddpd %ymm0, %ymm4
```

que agora produz quatro multiplicações de ponto flutuante com 64 bits.

Detalhamento

AVX também acrescentou três instruções de endereço ao x86. Por exemplo, vaddpd agora pode especificar

```
vaddpd %ymm0, %ymm1, %ymm4 # %ymm4 = %ymm1 + %ymm2
```

em vez da versão padrão com dois endereços

```
addpd %xmm0, %xmm4 # %xmm4 = %xmm4 + %xmm0
```

(Ao contrário do MIPS, o destino está à direita no x86.) Três endereços podem reduzir o número de registradores e instruções necessárias para um cálculo.

3.8. Mais rápido: Paralelismo subword e multiplicação matricial

Para demonstrar o impacto do paralelismo subword sobre o desempenho, vamos executar o mesmo código no Intel Core i7, primeiro sem AVX e depois com ele. A Figura 3.21 é uma versão não otimizada de uma multiplicação matricial, escrita em C. Como vimos na Seção 3.5, esse programa normalmente é chamado *DGEMM*, que significa Double precision GEneral Matrix Multiply. A partir desta edição, incluímos uma nova seção, intitulada “Mais rápido”, para demonstrar o benefício no desempenho da adaptação do software ao hardware subjacente, neste caso, a versão Sandy Bridge do microprocessador Intel Core i7. Esta nova seção nos Capítulos 3, 4, 5 e 6 melhorará, de forma incremental, o desempenho da DGEMM usando as ideias que cada capítulo introduz.

```

1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.     for (int i = 0; i < n; ++i)
4.         for (int j = 0; j < n; ++j)
5.     {
6.         double cij = C[i+j*n]; /* cij = C[i][j] */
7.         for( int k = 0; k < n; k++ )
8.             cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.         C[i+j*n] = cij; /* C[i][j] = cij */
10.    }
11. }

```

FIGURA 3.21 Versão não otimizada de uma multiplicação matricial com precisão dupla, comumente conhecida como DGEMM, de Double-precision GEneral Matrix Multiply.

Como estamos passando a dimensão da matriz como o parâmetro n , esta versão de DGEMM usa versões unidimensionais das matrizes C , A e B , e a aritmética de endereço para obter um melhor desempenho, em vez de usar os arrays bidimensionais, mais intuitivos, que vimos na [Seção 3.5](#). Os comentários nos lembram dessa notação mais intuitiva.

A [Figura 3.22](#) mostra a saída na linguagem assembly x86 para o loop mais interno da [Figura 3.21](#). As cinco instruções de ponto flutuante começam com um v , como as instruções AVX, mas observe que elas usam os registradores XMM, em vez de YMM, e incluem sd no nome, que significa precisão dupla escalar (scalar double). Veremos em breve as instruções paralelas de subword.

```

1. vmovsd (%r10),%xmm0          # Lê 1 elemento de C para %xmm0
2. mov    %rsi,%rcx             # reg. %rcx = %rsi
3. xor    %eax,%eax            # reg. %eax = 0
4. vmovsd (%rcx),%xmm1          # Lê 1 elemento de B para %xmm1
5. add    %r9,%rcx              # reg. %rcx = %rcx + %r9
6. vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiplica %xmm1, elemento de A
7. add    $0x1,%rax              # reg. %rax = %rax + 1
8. cmp    %eax,%edi             # compara %eax com %edi
9. vaddsd %xmm1,%xmm0,%xmm0      # Soma %xmm1, %xmm0
10. jg     30 <dgemm+0x30>        # desvia se %eax > %edi
11. add    $0x1,%r11d             # reg. %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10)          # Armazena %xmm0 no elemento C

```

FIGURA 3.22 A linguagem assembly x86 para o corpo dos loops aninhados gerados pela compilação do código C não otimizado na Figura 3.21.

Embora esteja lidando com apenas 64 bits de dados, o compilador usa a versão AVX das instruções, em vez do SSE2, provavelmente para que possa usar três endereços por instruções em vez de dois (veja o Detalhamento na [Seção 3.7](#)).

Embora os escritores de compilador eventualmente possam ser capazes de produzir rotineiramente um código de alta qualidade, que usa instruções AVX do x86, por enquanto precisamos “trapacear” usando intrínsecos C, que praticamente informam ao compilador como produzir exatamente um código bom. A [Figura 3.23](#) mostra a versão aperfeiçoada da [Figura 3.21](#), para a qual o compilador Gnu C produz código AVX. A [Figura 3.24](#) mostra o código x86 anotado que é a saída da compilação usando gcc com o nível de otimização –O3.

```

1. #include <x86intrin.h>
2. void dgemm ( int n, double* A, double* B, double* C)
3. {
4.     for ( int i = 0; i < n; i+=4 )
5.         for ( int j = 0; j < n; j++ ) {
6.             __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
7.             for( int k = 0; k < n; k++ )
8.                 c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.                                     _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.                                     _mm256_broadcast_sd(B+k+j*n)));
11.             _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.         }
13. }

```

FIGURA 3.23 Versão C otimizada de DGEMM usando detalhes do C para gerar as instruções paralelas de subword AVX para o x86.

A [Figura 3.24](#) mostra a linguagem assembly produzida pelo compilador para o loop mais interno.

```

1. vmovapd (%r11),%ymm0          # Lê 4 elementos de C para %ymm0
2. mov    %rbx,%rcx              # reg. %rcx = %rbx
3. xor    %eax,%eax              # reg. %eax = 0
4. vbroadcastsd (%rax,%r8,1),%ymml # Faz 4 cópias do elemento B
5. add    $0x8,%rax              # reg. %rax = %rax + 8
6. vmulpd (%rcx),%ymml,%ymml   # Mult. paralela %ymml,4
7. add    %r9,%rcx              # reg. %rcx = %rcx + %r9
8. cmp    %r10,%rax              # Compara %r10 com %rax
9. vaddpd %ymml,%ymm0,%ymm0    # Soma paralela %ymml, %ymm0
10. jne   50 <dgemm+0x50>       # Desvia se não %r10 != %rax
11. add    $0x1,%esi              # reg. %esi = %esi + 1
12. vmovapd %ymm0,(%r11)        # Armazena %ymm0 no elemento 4 C

```

FIGURA 3.24 A linguagem assembly x86 para o corpo dos loops aninhados gerados pela compilação do código C

otimizado da [Figura 3.23](#).

Observe as semelhanças com a [Figura 3.22](#), sendo a principal diferença que as cinco operações de ponto flutuante agora estão usando registradores YMM e usando as versões pd das instruções para precisão dupla paralela, em vez da versão sd para a precisão dupla escalar.

A declaração na linha 6 da [Figura 3.23](#) usa o tipo de dado `_m256d`, que diz ao compilador que a variável manterá 4 valores de ponto flutuante com precisão dupla. O intrínseco `_mm256_load_pd()` também na linha 6 usa instruções AVX para carregar 4 números de ponto flutuante com precisão dupla em paralelo (`_pd`) da matriz C para `c0`. O cálculo de endereço `C+i+j*n` na linha 6 representa o elemento `C[i + j * n]`. Simetricamente, a etapa final na linha 11 usa o intrínseco `_mm256_store_pd()` para armazenar 4 números de ponto flutuante com precisão dupla a partir de `c0` para a matriz C. Ao percorrermos 4 elementos em cada iteração, o loop `for` mais externo, na linha 4, incrementa i de 4, em vez de 1 na linha 3 da [Figura 3.21](#).

Dentro dos loops, na linha 9, primeiro carregamos quatro elementos de A novamente usando `_mm256_load_pd()`. Para multiplicar esses elementos por um elemento de B, na linha 10, primeiro usamos o intrínseco `_mm256_broadcast_sd()`, que faz quatro cópias idênticas do número escalar de precisão dupla — neste caso, um elemento de B — em um dos registradores YMM. Depois, usamos `_mm256_mul_pd()` na linha 9 para multiplicar os quatro resultados de precisão dupla em paralelo. Por fim, `_mm256_add_pd()` na linha 8 soma os quatro produtos às quatro somas em `c0`.

A [Figura 3.24](#) mostra o código x86 resultante, produzido pelo compilador para o corpo dos loops mais internos. Você pode ver as cinco instruções AVX — todas começando com v e quatro das cinco usando pd de precisão dupla paralela — que correspondem aos intrínsecos C mencionados acima. O código é muito semelhante ao da [Figura 3.22](#) anterior: ambos usam 12 instruções, as instruções com inteiros são quase idênticas (mas com registradores diferentes) e as diferenças na instrução de ponto flutuante geralmente são apenas a passagem de *scalar double* (sd) usando registradores XMM para *parallel double* (pd) com registradores YMM. A única exceção é a linha 4 da [Figura 3.24](#). Cada elemento de A precisa ser multiplicado por um elemento de B. Uma solução é colocar quatro cópias idênticas do elemento B de 64 bits lado a lado dentro do registrador YMM de 256 bits, que é exatamente o que a instrução `vbroadcastsd` faz.

Para matrizes com dimensões de 32 por 32, o DGEMM não otimizado na [Figura 3.21](#) roda a 1,7 GigaFLOPS (Floating point Operations Per Second) em um núcleo de um Intel Core i7 (Sandy Bridge) a 2,6 GHz. O código otimizado na [Figura 3.23](#) funciona a 6,4 GigaFLOPS. A versão AVX é 3,85 vezes mais rápida, o que é muito próximo do fator de aumento 4,0 que você poderia esperar com a execução de 4 vezes mais operações de cada vez usando o **paralelismo subword**.



PARALELISMO

Detalhamento

Como dissemos no Detalhamento na Seção 1.6, a Intel oferece o modo Turbo, que roda temporariamente a uma taxa de clock mais alta até que o chip se esquente muito. Esse Intel Core i7 (Sandy Bridge) pode aumentar de 2,6 GHz para 3,3 GHz no modo Turbo. Os resultados acima são com o modo Turbo desativado. Se o ativarmos, melhoramos todos os resultados pelo aumento na taxa de clock de $3,3/2,6 = 1,27$ a 2,1 GFLOPS para DGEMM não otimizado e 8,1 GFLOPS com AVX. O modo Turbo funciona particularmente bem quando se usa apenas um único núcleo de um chip de oito núcleos, como neste caso, pois permite que o único núcleo use muito mais do que sua quota justa de potência, já que os outros núcleos estão ociosos.

3.9. Falácia e armadilhas

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

As falácias e armadilhas aritméticas geralmente advêm da diferença entre a precisão limitada da aritmética computacional e da precisão ilimitada da aritmética natural.

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 , em que 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 x 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 de dois para -5_{dec} é

1111 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 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

11111111111111111111111111111110_{bin}

O resultado é -2_{dec}, em vez de -1_{dec}; próximo, mas não podemos comemorar.

Armadilha: a adição de ponto flutuante não é associativa.

A associatividade se mantém para uma sequência de adições de inteiros em complemento de dois, mesmo que o cálculo exceda. Infelizmente, como os números de ponto flutuante são aproximações dos números reais, e como a aritmética computacional tem precisão limitada, isso não é verdade para os números de ponto flutuante. Devido a grande faixa de números que podem ser representados em ponto flutuante, ocorrem problemas quando se somam dois números grandes de sinais opostos, mas um número pequeno. Por exemplo, vejamos se $c + (a + b) = (c + a) + b$. Suponha que $c = -1,5_{dec} \times 10^{38}$, $a = 1,5_{dec} \times 10^{38}$, e $b = 1,0$, e que estes sejam todos números de precisão simples.

$$\begin{aligned}c + (a + b) &= -1,5_{dec} \times 10^{38} + (1,5_{dec} \times 10^{38} + 1,0) \\&= -1,5_{dec} \times 10^{38} + (1,5_{dec} \times 10^{38}) \\&= 0,0\end{aligned}$$

$$\begin{aligned}
 c + (a + b) &= (-1,5_{\text{dec}} \times 10^{38} + 1,5_{\text{dec}} \times 10^{38}) + 1,0 \\
 &= (0,0_{\text{dec}}) + 1,0 \\
 &= 1,0
 \end{aligned}$$

Como os números de ponto flutuante possuem precisão limitada e resultam em aproximações dos resultados reais, $1,5_{\text{dec}} \times 10^{38}$ é tão maior que $1,0_{\text{dec}}$, que $1,5_{\text{dec}} \times 10^{38} + 1,0$ ainda é $1,5_{\text{dec}} \times 10^{38}$. É por isso que a soma de c , a e b é $0,0$ ou $1,0$, dependendo da ordem das adições de ponto flutuante, de modo que $c + (a + b) \neq (c + a) + b$. Portanto, a adição de ponto flutuante *não* é associativa.

Falácia: as estratégias de execução paralela que funcionam para tipos de dados inteiros também funcionam para tipos de dados de ponto flutuante.

Os programas normalmente têm sido escritos primeiro para executarem sequencialmente antes de simultaneamente, de modo que uma pergunta natural é “as duas versões geram a mesma resposta?”. Se a resposta for não, você pode considerar que existe um defeito na versão paralela, que precisa ser localizado.

Essa técnica considera que a aritmética do computador não afeta os resultados quando passa de sequencial para paralelo. Ou seja, se você tivesse de somar um milhão de números, obteria os mesmos resultados usando 1 processador ou 1.000 processadores. Essa suposição continua para inteiros no complemento de dois, pois a adição de inteiros é associativa. Infelizmente, como a adição de ponto flutuante não é associativa, a suposição não é mantida.

Uma versão mais irritante dessa armadilha ocorre em um computador paralelo, em que o escalonador do sistema operacional pode usar um número diferente de processadores, dependendo do que outros programas estão executando em um computador paralelo. O programador paralelo desavisado pode se confundir com seu programa obtendo respostas ligeiramente diferentes toda vez que for executada exatamente com o mesmo código e entrada idêntica, pois o número variável de processadores em cada execução faria com que as somas de ponto flutuante fossem calculadas em diferentes ordens.

Por causa desse dilema, os programadores que escrevem código paralelo com números de ponto flutuante precisam verificar se os resultados são confiáveis, mesmo que não deem exatamente a mesma resposta que o código sequencial. O campo que lida com essas questões é a análise numérica, abordada em diversos

livros-texto voltados para esse assunto. Esses problemas são bons motivos para a popularidade das bibliotecas numéricas, como LAPACK e SCALAPAK, que foram validadas em suas formas sequencial e paralela.

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 ([Figura 3.25](#)). A seguir, está a história por trás das manchetes.



FIGURA 3.25 Uma amostra dos artigos de jornais e revistas de novembro de 1994, incluindo *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 mais” do *David Letterman Late Show* na televisão. A Intel acabou tendo um custo de US\$300 milhões para substituir os chips com defeito.

O Pentium usava 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 a fim de gerar um novo resto. Assim como a divisão sem restauração, se uma escolha anterior obtiver um resto muito grande, o resto parcial é ajustado em um momento subsequente.

Evidentemente, havia cinco elementos da tabela do 80486 que os engenheiros da Intel pensaram que nunca poderiam ser acessados, e eles otimizaram a lógica 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.

Um professor de matemática no Lynchburg College, na Virgínia, Thomas Nicely, descobriu o *bug* em setembro de 1994. Depois de ligar para o suporte técnico da Intel e não receber uma posição oficial, ele posta sua descoberta na Internet. Essa postagem gerou uma história em uma revista do setor, que por sua vez, fez com que a Intel emitisse um comunicado oficial. Ela chamou o *bug* de um “glitch” que afetaria apenas os matemáticos teóricos, com um usuário normal de planilha vendo um erro somente a cada 27.000 anos. A IBM Research logo contra-argumentou que o usuário comum de planilha veria um erro a cada 24 dias. No fim, a Intel jogou a toalha, lançando o seguinte comunicado no dia 21 de dezembro:

“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 estejam preocupados. Queremos despreocupá-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 engenheiros da Intel não receberam um bônus de Natal naquele ano.

Essa história nos faz refletir sobre alguns pontos. Quão mais econômico seria 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 de que tantos dependem como um microprocessador?

3.10. Comentários finais

Com o passar das décadas, a aritmética computacional tornou-se padronizada, aumentando bastante a portabilidade dos programas. A aritmética de inteiros binários com complemento de dois é encontrada em cada computador vendido hoje e, se incluir suporte para ponto flutuante, oferece aritmética de ponto flutuante binário do padrão IEEE 754.

A aritmética computacional distingue-se 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. Os Capítulos 4 e 5 discutem 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. A recente mudança para o **paralelismo** acenderá novamente os holofotes sobre a análise numérica, à medida que soluções que eram consideradas seguras nos computadores sequenciais precisam ser reconsideradas quando se tenta encontrar o algoritmo mais rápido para computadores paralelos, que ainda alcance um resultado correto.



PARALELISMO

O paralelismo em nível de dados, especificamente o paralelismo subword, oferece um caminho simples para o desempenho mais alto de programas que usam operações aritméticas intensamente para dados inteiros e de ponto flutuante. Mostramos que é possível agilizar a multiplicação matricial tornando-a quase quatro vezes mais rápida, usando instruções que poderiam executar quatro operações de ponto flutuante de uma só vez.

Com a explicação sobre aritmética computacional deste capítulo vem uma descrição detalhada do conjunto de instruções do MIPS. Uma questão que gera 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 3.26](#) 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 3.27](#) estão as instruções que o processador MIPS executa que não se encontram na [Figura 3.26](#). Chamamos o conjunto completo de instrução de hardware de *MIPS-32*. À direita da [Figura 3.27](#) estão as instruções aceitas pelo montador, que não fazem parte do MIPS-32. Chamamos esse conjunto de instruções de *PseudoMIPS*.

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	lbu	I	store word to floating-point double	sdcl	I
store byte	sb	I	branch on floating-point true	bclt	I
load linked (atualização atômica)	ll	I	branch on floating-point false	bclf	I
store cond. (atualização atômica)	sc	I	floating-point compare single	c.x.s	R
branch on equal	beq	I	(x = eq, neq, lt, le, gt, ge)		
branch on not equal	bne	I	floating-point compare double	c.x.d	R
jump	j	J	(x = eq, neq, lt, le, gt, ge)		
jump and link	jal	J			
jump register	jr	R			
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 3.26 O conjunto de instruções MIPS.

Este livro se concentra nas instruções da coluna à esquerda.

Essa informação também se encontra nas colunas 1 e 2 do Guia de Referência do MIPS no final deste livro.

MIPS-32 restantes	Nome	Formato	PseudoMIPS	Nome	Formato
exclusive or ($rs \oplus rt$)	xor	R	absolute value	abs	rd,rs
exclusive or immediate	xori	I	negate (com ou sem sinal)	negs	rd,rs
shift right arithmetic	sra	R	rotate left	rol	rd,rs,rt
shift left logical variable	sllv	R	rotate right	ror	rd,rs,rt
shift right logical variable	srlv	R	multiply and don't check oflw (com ou sem sinal)	mul\$	rd,rs,rt
shift right arithmetic variable	srav	R	multiply and check oflw (com ou sem sinal)	mulos	rd,rs,rt
move to Hi	mthi	R	divide and check overflow	div	rd,rs,rt
move to Lo	mtlo	R	divide and don't check overflow	divu	rd,rs,rt
load halfword	lh	I	remainder (com ou sem sinal)	rem\$	rd,rs,rt
load byte	lb	I	load immediate	li	rd,imm
load word left (não alinhado)	lw1	I	load address	la	rd,addr
load word right (não alinhado)	lwr	I	load double	ld	rd,addr
store word left (não alinhado)	sw1	I	store double	sd	rd,addr
store word right (não alinhado)	swr	I	unaligned load word	ulw	rd,addr
load linked (atualização atômica)	ll	I	unaligned store word	usw	rd,addr
store cond. (atualização atômica)	sc	I	unaligned load halfword (com ou sem sinal)	ulhs	rd,addr
move if zero	movz	R	unaligned store halfword	ush	rd,addr
move if not zero	movn	R	branch	b	Label
multiply and add (com ou sem sinal)	madd\$	R	branch on equal zero	beqz	rs,L
multiply and subtract (com ou sem sinal)	msub\$	I	branch on compare (com ou sem sinal)	bxs	rs,rt,L
branch on \geq zero and link	bgezal	I	($x = lt, le, gt, ge$)		
branch on $<$ zero and link	bltzal	I	set equal	seq	rd,rs,rt
jump and link register	jalr	R	set not equal	sne	rd,rs,rt
branch compare to zero	bxz	I	set on compare (com ou sem sinal)	sx\$	rd,rs,rt
branch compare to zero likely	bxzl	I	($x = lt, le, gt, ge$)		
($x = lt, le, gt, ge$)			load to floating point (s ou d)	l.f	rd,addr
branch compare reg likely	bxl	I	store from floating point (s ou d)	s.f	rd,addr
trap if compare reg	tx	R			
trap if compare immediate	txi	I			
($x = eq, neq, lt, le, gt, ge$)					
return from exception	rfe	R			
system call	syscall	I			
break (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 3.27 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**). O sublinhado representa a letra a ser incluída para representar esse tipo de dados.

A Figura 3.28 indica a popularidade das instruções MIPS para os benchmarks de inteiro e de ponto flutuante SPEC CPU2006. Todas as instruções listadas foram responsáveis por, pelo menos, 0,2% das instruções executadas.

Núcleo MIPS	Nome	Inteiro	PF	Núcleo aritmético + MIPS-32	Nome	Inteiro	PF
add	add	0,0%	0,0%	FP add double	add.d	0,0%	10,6%
add immediate	addi	0,0%	0,0%	FP subtract double	sub.d	0,0%	4,9%
add unsigned	addu	5,2%	3,5%	FP multiply double	mul.d	0,0%	15,0%
add immediate unsigned	addiu	9,0%	7,2%	FP divide double	div.d	0,0%	0,2%
subtract unsigned	subu	2,2%	0,6%	FP add single	add.s	0,0%	1,5%
AND	AND	0,2%	0,1%	FP subtract single	sub.s	0,0%	1,8%
AND immediate	ANDi	0,7%	0,2%	FP multiply single	mul.s	0,0%	2,4%
OR	OR	4,0%	1,2%	FP divide single	div.s	0,0%	0,2%
OR immediate	ORi	1,0%	0,2%	load word to FP double	l.d	0,0%	17,5%
NOR	nor	0,4%	0,2%	store word to FP double	s.d	0,0%	4,9%
shift left logical	sll	4,4%	1,9%	load word to FP single	l.s	0,0%	4,2%
shift right logical	srl	1,1%	0,5%	store word to FP single	s.s	0,0%	1,1%
load upper immediate	lui	3,3%	0,5%	branch on floating-point true	bclt	0,0%	0,2%
load word	lw	18,6%	5,8%	branch on floating-point false	bclf	0,0%	0,2%
store word	sw	7,6%	2,0%	floating-point compare double	c.x.d	0,0%	0,6%
load byte	lb	3,7%	0,1%	multiply	mul	0,0%	0,2%
store byte	sb	0,6%	0,0%	shift right arithmetic	sra	0,5%	0,3%
branch on equal (zero)	beq	8,6%	2,2%	load half	lhu	1,3%	0,0%
branch on not equal (zero)	bne	8,4%	1,4%	store half	sh	0,1%	0,0%
jump and link	jal	0,7%	0,2%				
jump register	jr	1,1%	0,2%				
set less than	slt	9,9%	2,3%				
set less than immediate	slti	3,1%	0,3%				
set less than unsigned	sltu	3,4%	0,8%				
set less than imm. uns.	sltiu	1,1%	0,1%				

FIGURA 3.28 Frequência das instruções MIPS para o benchmark de inteiros e ponto flutuante SPEC CPU2006.

Todas as instruções responsáveis por, pelo menos, 0,2% das instruções estão incluídas na tabela. As pseudoinstruções são convertidas em MIPS-32 antes da execução e, portanto, não aparecem aqui.

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 SCPEC CPU2006 de inteiros, e o núcleo de inteiros mais aritmético domina o ponto flutuante SPEC CPU2006, como mostra a tabela a seguir.

Subconjunto de instruções	Inteiros	Ponto flutuante
Núcleo do MIPS	98%	31%
Núcleo aritmético do MIPS	2%	66%
MIPS-32 restante	0%	3%

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 comuns, 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. Não

importa qual seja o conjunto de instruções ou seu tamanho — MIPS, ARM, x86 —, nunca se esqueça de que os padrões de bits não possuem significado inerente. O mesmo padrão de bits pode representar um inteiro com sinal, um inteiro sem sinal, um número de ponto flutuante, uma string, uma instrução etc. Nos computadores com programa armazenado, é a operação sobre o padrão de bits que determina seu significado.

3.11. Exercícios

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

- 3.1. [5] <§3.2> O que é 5ED4–07A4 quando esses valores representam números hexadecimais de 16 bits sem sinal? O resultado deverá ser escrito em hexadecimal. Mostre o seu trabalho.
- 3.2. [5] <§3.2> O que é 5ED4–07A4 quando esses valores representam números hexadecimais de 16 bits com sinal, armazenados no formato sinal-magnitude? O resultado deverá ser escrito em hexadecimal. Mostre o seu trabalho.
- 3.3. [10] <§3.2> Converta 5ED4 em um número binário. O que torna a base 16 (hexadecimal) um sistema de numeração atraente para representar valores nos computadores?
- 3.4. [5] <§3.2> O que é 4365–3412 quando esses valores representam números octais de 12 bits sem sinal? O resultado deverá ser escrito em octal. Mostre seu trabalho.
- 3.5. [5] <§3.2> O que é 4365–3412 quando esses valores representam números octais de 12 bits com sinal, armazenados no formato sinal-magnitude? O resultado deverá ser escrito em octal. Mostre seu trabalho.
- 3.6. [5] <§3.2> Suponha que 185 e 122 sejam decimais inteiros de 8 bit sem sinal. Calcule 185–122. Existe overflow, underflow ou nenhum destes?
- 3.7. [5] <§3.2> Suponha que 185 e 122 sejam inteiros decimais de 8 bits com sinal, armazenados no formato sinal-magnitude. Calcule 185 + 122. Existe overflow, underflow ou nenhum destes?

- 3.8.** [5] <§3.2> Suponha que 185 e 122 sejam inteiros decimais de 8 bits com sinal, armazenados no formato sinal-magnitude. Calcule 185–122. Existe overflow, underflow ou nenhum destes?
- 3.9.** [10] <§3.2> Suponha que 151 e 214 sejam inteiros decimais de 8 bits com sinal, armazenados no formato de complemento de dois. Calcule 151 + 214 usando a aritmética com saturação. O resultado deverá ser escrito em decimal. Mostre seu trabalho.
- 3.10.** [10] <§3.2> Suponha que 151 e 214 sejam inteiros decimais de 8 bits com sinal, armazenados no formato de complemento de dois. Calcule 151–214 usando a aritmética com saturação. O resultado deverá ser escrito em decimal. Mostre seu trabalho.
- 3.11.** [10] <§3.2> Suponha que 151 e 214 sejam inteiros de 8 bits sem sinal. Calcule 151 + 214 a aritmética com saturação. O resultado deverá ser escrito em decimal. Mostre seu trabalho.
- 3.12.** [20] <§3.3> Usando uma tabela semelhante à que mostramos na [Figura 3.6](#), calcule o produto dos inteiros octais de 6 bits sem sinal 62 e 12 usando o hardware descrito na [Figura 3.3](#). Você deverá mostrar o conteúdo dos registradores em cada etapa.
- 3.13.** [20] <§3.3> Usando uma tabela semelhante à que mostramos na [Figura 3.6](#), calcule o produto dos inteiros hexadecimais de 8 bits sem sinal 62 e 12 usando o hardware descrito na [Figura 3.5](#). Você deverá mostrar o conteúdo de cada registrador em cada etapa.
- 3.14.** [10] <§3.3> Calcule o tempo necessário para realizar uma multiplicação usando a técnica dada nas [Figuras 3.3 e 3.4](#) se um inteiro tiver 8 bits de largura e cada etapa da operação exigir 4 unidades de tempo. Suponha que, na etapa 1a, uma adição sempre é realizada — ou o multiplicando ou o 0 será somado. Suponha também que os registradores já foram inicializados (você está simplesmente contando quanto tempo é necessário para se realizar o próprio loop de multiplicação). Se isso estiver sendo feito no hardware, os deslocamentos do multiplicando e do multiplicador podem ser feitos simultaneamente. Se isso estiver sendo feito no software, eles terão de ser feitos um após o outro. Solucione para cada caso.
- 3.15.** [10] <§3.3> Calcule o tempo necessário para realizar uma multiplicação usando a técnica descrita no texto (31 somadores empilhados verticalmente) se um inteiro tiver 8 bits de largura e um somador exigir 4 unidades de tempo.
- 3.16.** [20] <§3.3> Calcule o tempo necessário para realizar uma multiplicação

usando a técnica dada na [Figura 3.7](#), se um inteiro tiver 8 bits de largura e um somador exigir 4 unidades de tempo.

- 3.17.** [20] <§3.3> Conforme discutimos no texto, uma possível melhoria no desempenho é realizar um deslocamento e soma em vez de uma multiplicação real. Como 9×6 , por exemplo, pode ser escrito como $(2 \times 2 \times 2 + 1) \times 6$, podemos calcular 9×6 deslocando 6 para a esquerda três vezes e depois somando 6 a esse resultado. Mostre a melhor maneira de calcular $0 \times 33 \times 0 \times 55$ usando deslocamentos e adições/subtrações. Suponha que ambas as entradas sejam inteiros de 8 bits sem sinal.
- 3.18.** [20] <§3.4> Usando uma tabelas semelhante à que mostramos na [Figura 3.10](#), calcule 74 dividido por 21 usando o hardware descrito na [Figura 3.8](#). Você deverá mostrar o conteúdo de cada registrador em cada etapa. Suponha que as duas entradas sejam inteiros de 6 bits sem sinal.
- 3.19.** [30] <§3.4> Usando uma tabela semelhante à que mostramos na [Figura 3.10](#), calcule 74 dividido por 21 usando o hardware descrito na [Figura 3.11](#). Você deverá mostrar o conteúdo de cada registrador em cada etapa. Suponha que as duas entradas sejam inteiros de 6 bits sem sinal. Este algoritmo requer uma técnica ligeiramente diferente daquela mostrada na [Figura 3.9](#). Você deverá pensar bem nisso, realizando de um a dois experimentos ou então vá à Web descobrir como fazer isso funcionar corretamente. (Dica: uma solução possível envolve o fato de que a [Figura 3.11](#) possa deslocar o registrador de resto em qualquer direção.)
- 3.20.** [5] <§3.5> Que número decimal o padrão de bits `0x0C000000` representa se ele for um inteiro em complemento de dois? E um inteiro sem sinal?
- 3.21.** [10] <§3.5> Se o padrão de bits `0x0C000000` for colocado no Registrador de Instrução, que instrução MIPS será executada?
- 3.22.** [10] <§3.5> Que número decimal o padrão de bits `0x0C000000` representa se ele for um número de ponto flutuante? Use o padrão IEEE 754.
- 3.23.** [10] <§3.5> Escreva a representação binária do número decimal 63,25, considerando o formato de precisão simples IEEE 754.
- 3.24.** [10] <§3.5> Escreva a representação binária do número decimal 63,25, considerando o formato de precisão dupla IEEE 754.
- 3.25.** [10] <§3.5> Escreva a representação binária do número decimal 63,25 considerando que ele foi armazenado usando o formato IBM de precisão simples (base 16, em vez da base 2, com 7 bits de expoente).
- 3.26.** [20] <§3.5> Escreva o padrão de bits binário para representar $-1,5625 \times 10^{-1}$ considerando um formato semelhante ao empregado pelo

DEC PDP-8 (12 bits da esquerda são o expoente armazenado como um número de complemento de dois, e os 24 bits da direita são a fração armazenada como um número de complemento de dois). Nenhum 1 oculto é utilizado. Compare o intervalo e a precisão desse padrão de 36 bits com os padrões IEEE 754 de precisão simples e dupla.

- 3.27.** [20] <§3.5> IEEE 754-2008 tem um formato de meia precisão, que tem apenas 16 bits de largura. O bit mais à esquerda ainda é o bit de sinal, o expoente tem 5 bits de largura e possui um bias de 15, e a mantissa tem 10 bits de extensão. Assume-se que existe um 1 oculto. Escreva o padrão de bits para representar $-1,5625 \times 10^{-1}$ considerando uma versão modificada desse formato que utiliza um formato com excesso de 16 para armazenar o expoente. Comente sobre o intervalo e a precisão desse formato de ponto flutuante de 16 bits com o padrão IEEE 754 de precisão simples.
- 3.28.** [20] <§3.5> Os Hewlett-Packard 2114, 2115 e 2116 usavam um formato com os 16 bits mais à esquerda sendo a fração armazenada no formato de complemento de dois, seguida por outro campo de 16 bits que tinha nos 8 bits mais à esquerda uma extensão da fração (fazendo com que a fração tenha 24 bits de extensão) e os 8 bits mais à direita representando o expoente. Porém, por um capricho interessante, o expoente era armazenado em formato de magnitude de sinal com o bit de sinal no canto direito! Escreva o padrão de bits para representar $-1,5625 \times 10^{-1}$ considerando esse formato. Nenhum 1 oculto é utilizado. Compare o intervalo e a precisão desse padrão de 32 bits com o padrão IEEE 754 de precisão simples.
- 3.29.** [20] <§3.5> Calcule a soma de $2,6125 \times 10^1$ e $4,150390625 \times 10^{-1}$ à mão, supondo que ambas as entradas sejam armazenadas no formato de meia precisão com 16 bits, descrito no Exercício 3.27. Considere um bit de guarda, um bit de arredondamento e um sticky bit, e arredonde para o par mais próximo. Mostre todas as etapas.
- 3.30.** [30] <§3.5> Calcule o produto de $-8,0546875 \times 10^0$ e $-1,79931640625 \times 10^{-1}$ manualmente, considerando que ambas as entradas sejam armazenadas no formato de meia-precisão com 16 bits descrito no Exercício 3.27. Considere um bit de guarda, um bit de arredondamento e um sticky bit, e arredonde para o par mais próximo. Mostre todas as etapas; porém, como acontece no exemplo do texto, você pode realizar a multiplicação em formato legível para humanos, em vez de usar as técnicas descritas nos Exercícios de 3.12 a 3.14. Indique se existe overflow ou underflow. Escreva sua resposta no formato de 16 bits em ponto flutuante descrito no Exercício

3.27 e também como um número decimal. Qual é a precisão do seu resultado? Compare-o com o número que você obtém se realizar a multiplicação em uma calculadora.

- 3.31. [30] <§3.5>** Calcule $8,625 \times 10^1$ dividido por $-4,875 \times 10^0$ manualmente. Mostre todas as etapas necessárias para se chegar à sua resposta. Suponha que exista um bit de guarda, de arredondamento e um sticky bit, e use-os, se for necessário. Escreva a resposta final em formato de ponto flutuante com 16 bits descrito no Exercício 3.27 e em decimal, comparando o resultado decimal com o que você obtém usando uma calculadora.
- 3.32. [20] <§3.9>** Calcule $(3,984375 \times 10^{-1} + 3,4375 \times 10^{-1}) + 1,771 \times 10^3$ manualmente, considerando que cada um dos valores é armazenado no formato de meia-precisão com 16 bits, descrito no Exercício 3.27 (também descrito no texto). Considere um bit de guarda, um bit de arredondamento e um sticky bit, e arredonde para o par mais próximo. Mostre todas as etapas e escreva sua resposta em formato de ponto flutuante de 16 bits e em decimal.
- 3.33. [20] <§3.9>** Calcule $3,984375 \times 10^{-1} + (3,4375 \times 10^{-1} + 1,771 \times 10^3)$ manualmente, considerando que cada um dos valores é armazenado no formato de meia-precisão com 16 bits, descrito no Exercício 3.27 (também descrito no texto). Considere um bit de guarda, um bit de arredondamento e um sticky bit, e arredonde para o par mais próximo. Mostre todas as etapas, e escreva sua resposta em formato de ponto flutuante de 16 bits e em decimal.
- 3.34. [10] <§3.9>** Com base nas suas respostas dos Exercícios 3.32 e 3.33, $(3,984375 \times 10^{-1} + 3,4375 \times 10^{-1}) + 1,771 \times 10^3 = 3,984375 \times 10^{-1} + (3,4375 \times 10^{-1} + 1,771 \times 10^3)$?
- 3.35. [30] <§3.9>** Calcule $(3,41796875 \times 10^{-3} \times 6,34765625 \times 10^{-3}) \times 1,05625 \times 10^2$ manualmente, considerando que cada um dos valores é armazenado no formato de meia precisão com 16 bits, descrito no Exercício 3.27 (também descrito no texto). Considere um bit de guarda, um bit de arredondamento e um sticky bit, e arredonde para o par mais próximo. Mostre todas as etapas, e escreva sua resposta em formato de ponto flutuante de 16 bits e em decimal.
- 3.36. [30] <§3.9>** Calcule $3,41796875 \times 10^{-3} \times (6,34765625 \times 10^{-3} \times 1,05625 \times 10^2)$ manualmente, considerando que cada um dos valores é armazenado no formato de meia precisão com 16 bits, descrito no Exercício 3.27 (também descrito no texto). Considere um bit de guarda, um bit de arredondamento e um sticky bit, arredondando para o par mais próximo. Mostre todas as etapas, e escreva sua resposta em formato de ponto flutuante de 16 bits e em decimal.

- 3.37.** [10] <§3.9> Com base nas suas respostas dos Exercícios 3.35 e 3.36, $(3,41796875 \times 10^{-3} \times 6,34765625 \times 10^{-3}) \times 1,05625 \times 10^2 = 3,41796875 \times 10^{-3} \times (6,34765625 \times 10^{-3} \times 1,05625 \times 10^2)$?
- 3.38.** [30] <§3.9> Calcule $1,666015625 \times 10^0 \times (1,9760 \times 10^4 + -1,9744 \times 10^4)$ manualmente, considerando que cada um dos valores é armazenado no formato de meia-precisão com 16 bits, descrito no Exercício 3.27 (descrito no texto). Considere um bit de guarda, um bit de arredondamento e um sticky bit, arredondando para o par mais próximo. Mostre todas as etapas, e escreva sua resposta em formato de ponto flutuante de 16 bits e em decimal.
- 3.39.** [30] <§3.9> Calcule $(1,666015625 \times 10^0 \times 1,9760 \times 10^4) + (1,666015625 \times 10^0 \times -1,9744 \times 10^4)$ manualmente, considerando que cada um dos valores é armazenado no formato de meia-precisão com 16 bits, descrito no Exercício 3.27 (também descrito no texto). Considere um bit de guarda, um bit de arredondamento e um sticky bit, arredondando para o par mais próximo. Mostre todas as etapas, e escreva sua resposta em formato de ponto flutuante de 16 bits e em decimal.
- 3.40.** [10] <§3.9> Com base nas suas respostas dos Exercícios 3.38 e 3.39, confirme se $(1,666015625 \times 10^0 \times 1,9760 \times 10^4) + (1,666015625 \times 10^0 \times -1,9744 \times 10^4) = 1,666015625 \times 10^0 \times (1,9760 \times 10^4 + -1,9744 \times 10^4)$?
- 3.41.** [10] <§3.5> Usando o formato de ponto flutuante IEEE 754, escreva o padrão de bits que representaria $-1/4$. Você consegue representar $-1/4$ com exatidão?
- 3.42.** [10] <§3.5> O que você obtém se somar $-1/4$ a si mesmo 4 vezes?
Quanto é $-1/4 \times 4$? Eles são iguais? O que deveriam ser?
- 3.43.** [10] <§3.5> Escreva o padrão de bits na fração de valor $1/3$ considerando um formato de ponto flutuante que usa números binários na fração. Suponha que existam 24 bits e você não precisa normalizar. Essa representação é exata?
- 3.44.** [10] <§3.5> Escreva o padrão de bits na fração de valor $1/3$ considerando um formato de ponto flutuante que usa números Binary Coded Decimal (base 10) na fração, em vez da base 2. Suponha que existam 24 bits e você não precisa normalizar. Essa representação é exata?
- 3.45.** [10] <§3.5> Escreva o padrão de bits supondo que estamos usando números de base 15 na fração de valor $1/3$, em vez da base 2. (Números de base 16 utilizam os símbolos 0-9 e A-F. Números de base 15 usariam 0-9 e A-E.) Suponha que existam 24 bits e você não precisa normalizar. Essa representação é exata?

3.46. [20] <§3.5> Escreva o padrão de bits supondo que estamos usando números de base 30 na fração de valor 1/3, em vez da base 2. (Números de base 16 utilizam os símbolos 0-9 e A-F. Números de base 30 usariam 0-9 e A-T.) Suponha que existam 20 bits e você não precisa normalizar. Essa representação é exata?

3.47. [45] <§§3.6, 3.7> O código C a seguir implementar um filtro FIR de quatro tomadas no array de entrada `sig_in`. Suponha que todos os arrays sejam valores de ponto fixo com 16 bits.

```
for (i = 3;i<128;i++)
    sig_out[i]=sig_in[i-3] * f[0]+sig_in[i-2] * f[1]
    +sig_in[i-1] * f[2]+sig_in[i] * f[3];
```

Suponha que você tenha que escrever uma implementação otimizada deste código com linguagem assembly em um processador que possui instruções SIMD e registradores de 128 bits. Sem conhecer os detalhes do conjunto de instruções, descreva resumidamente como você implementaria esse código, maximizando o uso de operações de subword e minimizando a quantidade de dados que é transferida entre registradores e memória. Escreva todas as suas suposições sobre as instruções que você utiliza.

Respostas das Seções “Verifique você mesmo”

§3.2: página 158: 3.

§3.5: página 195: 3.

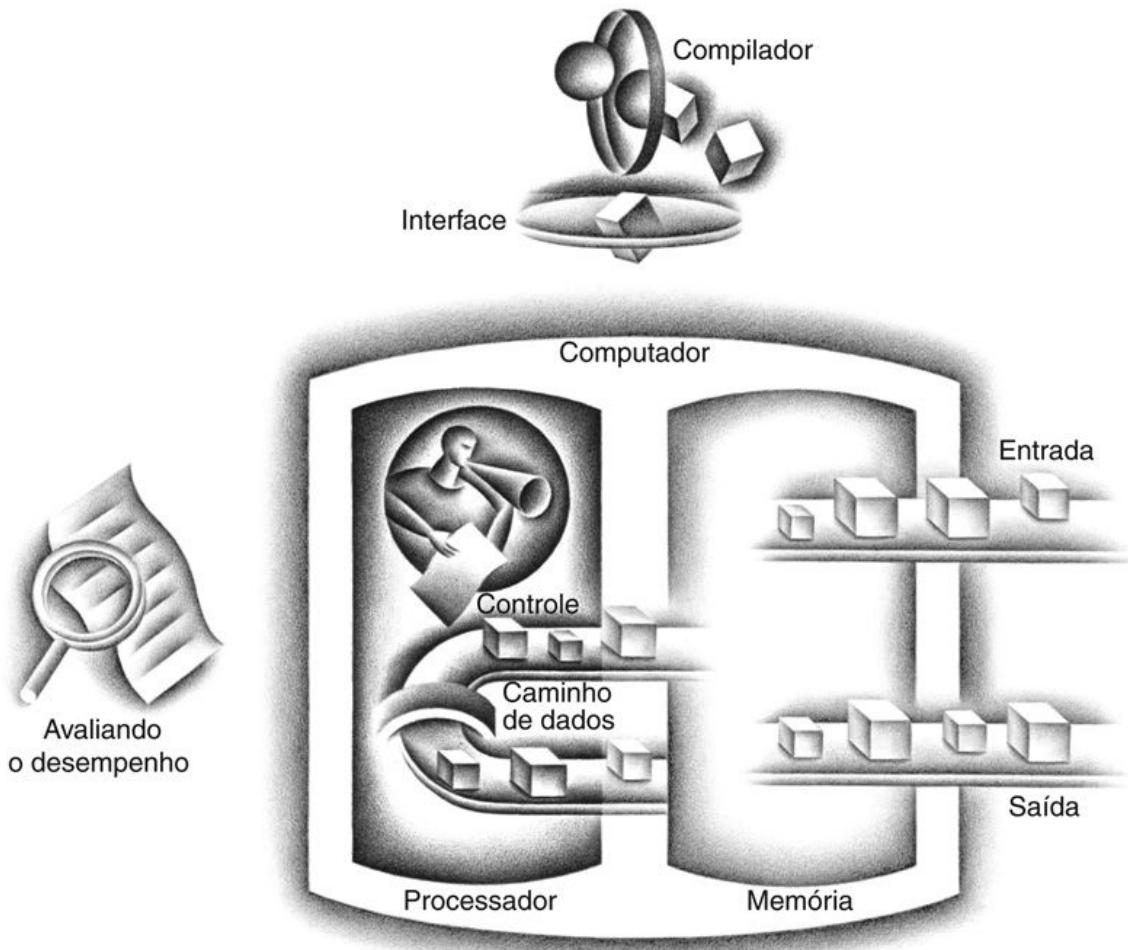
O Processador

Em um assunto importante, nenhum detalhe é pequeno.

Provérbio francês

- 4.1 Introdução
- 4.2 Convenções lógicas de projeto
- 4.3 Construindo um caminho de dados
- 4.4 Um esquema de implementação simples
- 4.5 Visão geral de pipelining
- 4.6 Caminho de dados e controle usando pipeline
- 4.7 Hazards de dados: forwarding *versus* stalls
- 4.8 Hazards de controle
- 4.9 Exceções
- 4.10 Paralelismo e paralelismo avançado em nível de instrução
- 4.11 Vida real: pipelines do ARM Cortex-A8 e Intel Core i7
- 4.12 Mais rápido: Paralelismo em nível de instrução e multiplicação matricial
- 4.13 Faláncias e armadilhas
- 4.14 Comentários finais
- 4.15 Exercícios

Os cinco componentes clássicos de um computador

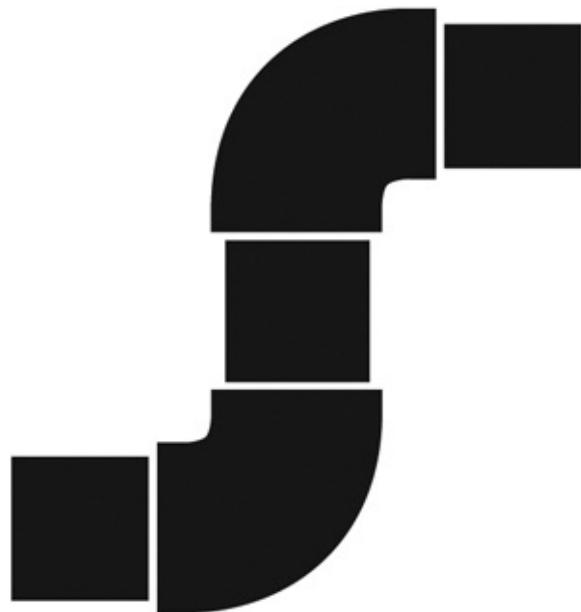


4.1. Introdução

O [Capítulo 1](#) explica que o desempenho de um computador é determinado por três fatores principais: contagem de instruções, tempo de ciclo de clock e *ciclos de clock por instrução* (CPI). O [Capítulo 2](#) explica que o compilador e a arquitetura do conjunto de instruções 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 das técnicas usadas na implementação de um processador, começando com uma sinopse altamente abstrata e simplificada nesta seção. Ela é seguida de uma seção que desenvolve um caminho de dados e constrói uma versão simples de um processador, suficiente para implementar conjuntos de instruções como o MIPS. O corpo do capítulo descreve uma implementação MIPS em **pipelining** mais realista, seguida de uma seção que desenvolve conceitos necessários para implementar conjuntos de instruções mais complexos, como o x86.



PIPELINING

Para o leitor interessado em entender a interpretação de alto nível de instruções e seu impacto sobre o desempenho do programa, esta seção inicial e a [Seção 4.5](#) apresentam os conceitos básicos do pipelining. Tendências recentes são abordadas na [Seção 4.10](#), e a [Seção 4.11](#) descreve as arquiteturas recentes Intel Core i7 e ARM Cortex-A8. A [Seção 4.12](#) mostra como usar o paralelismo

em nível de instrução para mais do que dobrar o desempenho da multiplicação matricial da [Seção 3.8](#). Estas seções oferecem uma base suficiente para entender os conceitos de pipeline em um alto nível.

Para os leitores que desejam um entendimento do processador e seu desempenho com mais profundidade, as [Seções 4.3, 4.4 e 4.6](#) serão úteis. Aqueles interessados em aprender como montar um processador também devem ler as [Seções 4.2, 4.7, 4.8 e 4.9](#).

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 *brench 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 são 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 o computador. Muitos dos princípios básicos de projeto apresentados no [Capítulo 1](#) podem ser ilustrados, considerando-se a implementação, como o princípio *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 ideias básicas usadas para construir um amplo espectro de computadores, desde servidores de alto desempenho a microprocessadores de finalidade geral e processadores embutidos.

Uma sinopse da implementação

No [Capítulo 2](#) vimos instruções MIPS básicas, incluindo as instruções lógicas e aritméticas, as de referência à memória e as de desvio. Muito do que precisa ser feito para implementar essas instruções é igual, independentemente 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 a instrução exata. A simplicidade e a regularidade do conjunto de instruções simplifica a implementação tornando semelhantes as execuções de muitas das classes de instrução.

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. Após usar a ALU, as ações necessárias para completar várias classes de instrução diferem. Uma instrução de referência à memória precisará acessá-la, a fim de escrever dados para um load ou ler dados para um store. Uma instrução lógica e aritmética precisa escrever os dados da ALU de volta a 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, a fim de chegar ao endereço da próxima instrução.

A [Figura 4.1](#) mostra a visão em alto nível de uma implementação MIPS, focando 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.

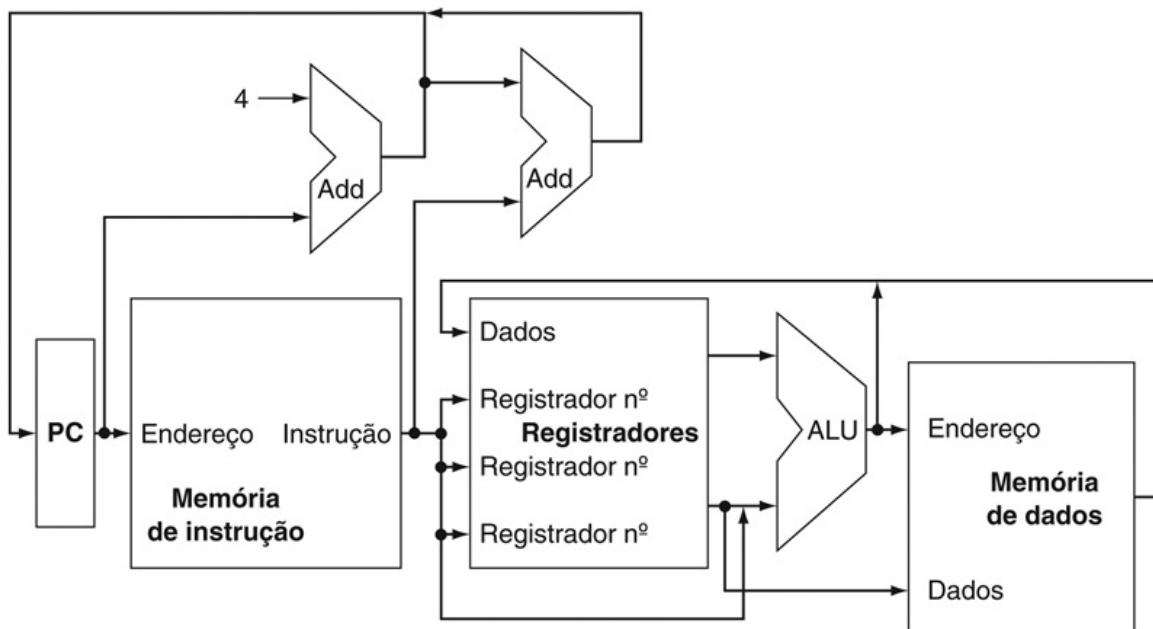


FIGURA 4.1 Uma visão abstrata da implementação do subconjunto MIPS mostrando as principais unidades funcionais e as principais conexões entre elas.

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 trazida, os registradores usados como operandos pela instrução são especificados por campos dessa instrução. Uma vez que os operandos tenham sido trazidos, eles podem ser operados de modo a calcular um endereço de memória (para um load ou store), 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 com a finalidade de 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 endereço da próxima instrução, que vem da ALU (em que o offset do PC e do desvio são somados) ou de um somador que incrementa o PC atual em 4. As linhas grossas interconectando as unidades funcionais representam barramentos, que consistem em múltiplos sinais. As setas 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.

Primeiro, em vários lugares, a [Figura 4.1](#) 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, os dados escritos no banco de registradores podem vir da ALU ou da memória de dados, e a segunda entrada da ALU pode vir de um registrador ou do campo imediato da instrução. 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 a seu destino. Essa seleção normalmente é feita com um dispositivo chamado *multiplexador*, embora uma melhor denominação desse dispositivo seria *seletor de dados*. O Apêndice B descreve o multiplexador, que seleciona entre várias entradas com base na configuração de suas linhas de controle. As linhas de controle são definidas principalmente com base na informação tomada da instrução sendo executada.

A segunda omissão na [Figura 4.1](#) é que 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 apenas em uma instrução load ou em uma instrução lógica ou aritmética. E, é claro, a ALU precisa realizar uma de várias operações. (O Apêndice B descreve o projeto detalhado da ALU.) Assim como os multiplexadores, essas operações são direcionadas por linhas de controle que são definidas com base nos vários campos das instruções.

A [Figura 4.2](#) mostra o caminho de dados da [Figura 4.1](#) 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, 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 no sentido de determinar como definir as linhas de controle.

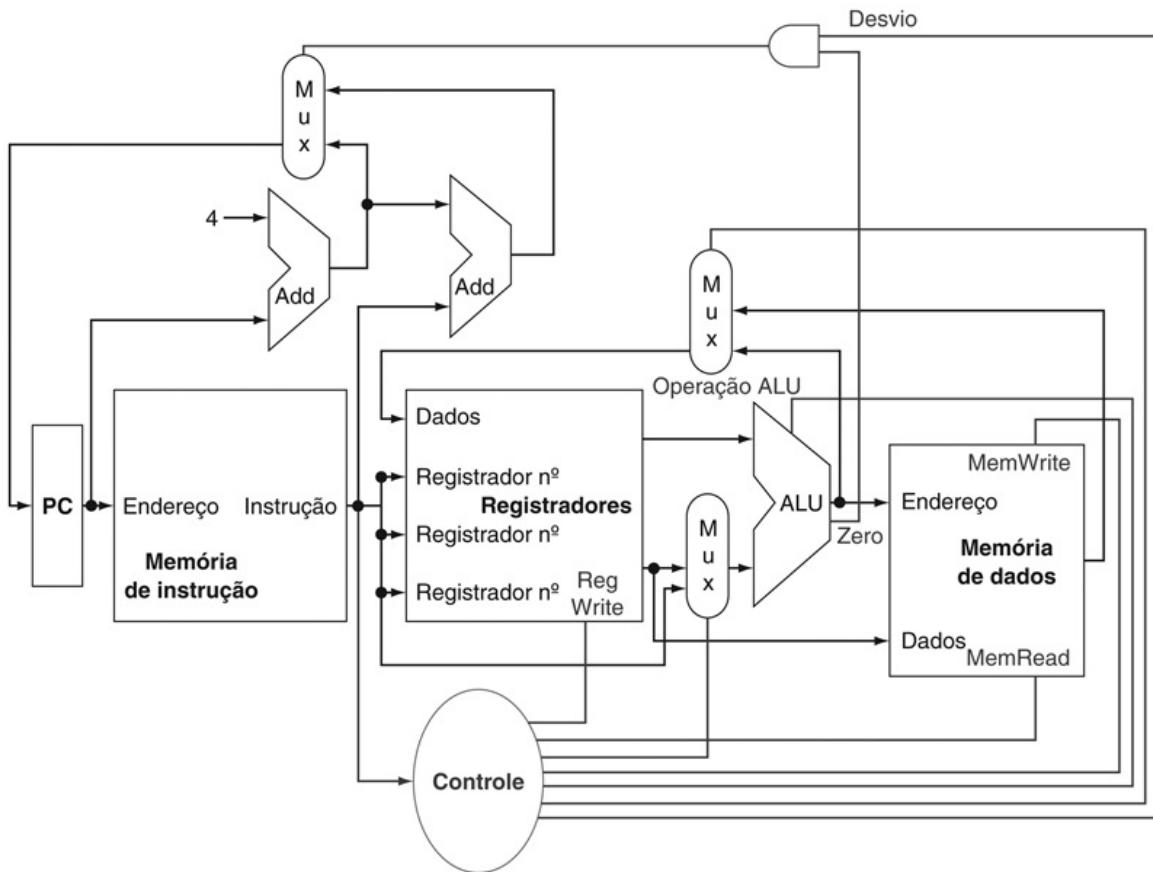


FIGURA 4.2 A implementação básica do subconjunto MIPS incluindo os multiplexadores necessários e as linhas de controle.

O multiplexador superior (“Mux”) 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 do meio, cuja saída retorna para o banco de registradores, é usado para conduzir a saída da ALU (no caso de uma instrução lógica ou aritmética) ou a saída da memória de dados (no caso de um load) a ser escrita no banco de registradores. Finalmente, o multiplexador da parte inferior é usado de modo a determinar se uma segunda entrada da ALU vem dos registradores (para uma instrução lógica-aritmética OU um desvio) ou do campo offset da instrução (para um load ou store). 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.

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, a fim de controlar que ações são realizadas para diferentes classes de instrução. As [Seções 4.3 e 4.4](#) descrevem uma implementação simples que usa um único ciclo de clock longo para cada instrução e segue a forma geral das [Figuras 4.1 e 4.2](#). 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 o ciclo de clock precisa ser bastante esticado para acomodar a instrução mais longa. Após projetar o controle desse computador simples, veremos uma implementação em pipeline com todas as suas complexidades, incluindo as exceções.

Verifique você mesmo

Quantos dos cinco componentes clássicos de um computador — mostrados no início deste capítulo — as Figuras 4.1 e 4.2 contêm?

4.2. Convenções lógicas de projeto

Para tratar do projeto de um computador, precisamos decidir como a implementação lógica do computador irá operar e como esse computador está sincronizado. Esta seção examina algumas ideias 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.

elemento combinacional

Um elemento operacional, como uma porta AND ou uma ALU.

Os elementos do caminho de dados na implementação MIPS consistem em dois tipos diferentes de elementos lógicos: aqueles que operam nos valores dos dados e os 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 4.1](#) e discutida no Apêndice B é um exemplo de 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 combinacionais, mas contêm *estado*. Um elemento contém estado se tiver algum armazenamento interno. Chamamos esses elementos de **elementos de estado**, pois, se desligássemos o computador da tomada, poderíamos reiniciá-lo 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 o computador nunca tivesse sido desligado. Na [Figura 4.1](#), as memórias de instruções e de dados, bem como os registradores, são exemplos de elementos de estado.

elemento de estado

Um elemento da memória, como um registrador ou uma memória.

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 (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 4.1](#). 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 *sequenciais* 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. Tanto a operação dos elementos combinacionais e sequenciais, quanto sua construção são discutidas em mais detalhes no Apêndice B.

Metodologia de clocking

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 garantir a previsibilidade.

metodologia de clocking

O método usado para determinar quando os dados são válidos e estáveis em relação ao clock.

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 sequencial são atualizados apenas em uma transição do clock, que é uma transição rápida de baixo para alto ou vice-versa ([Figura 4.3](#)). 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.

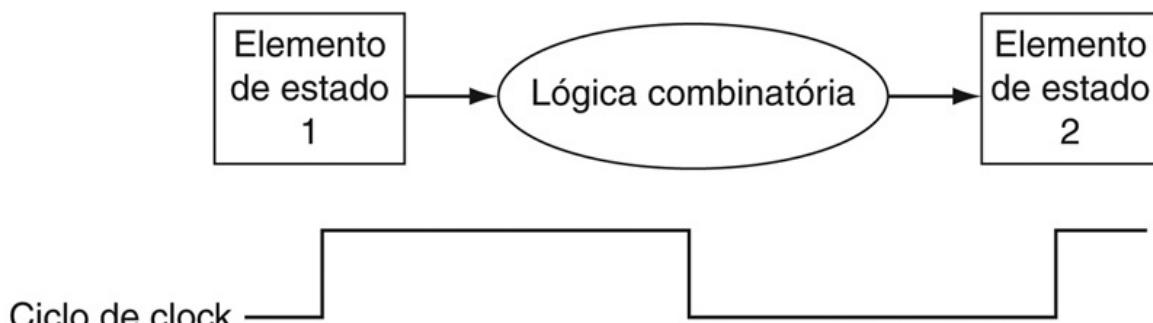


FIGURA 4.3 A lógica combinacional, os elementos de estado e o clock estão intimamente relacionados.

Em um sistema digital síncrono, o clock determina quando os elementos com estado escreverão valores no armazenamento interno. Quaisquer entradas em um elemento de estado precisam atingir 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 neste capítulo, incluindo a memória, são considerados acionados por transição positiva; ou seja, eles mudam na transição de subida do clock.

sincronização acionada por transição

Um esquema de clocking em que todas as mudanças de estado ocorrem em uma transição do clock.

A [Figura 4.3](#) mostra os dois elementos de estado em volta de um bloco de lógica combinacional, que opera em um único ciclo de clock: todos os sinais precisam se propagar desde o elemento de estado 1, passando pela lógica combinacional 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.

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.

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.

Usaremos o termo **ativo** para indicar um sinal que está logicamente alto, o termo *ativar* para especificar que um sinal deve ser conduzido a logicamente alto, e *desativar* ou **inativo** para representar o que é logicamente baixo. Usamos os termos ativar e desativar porque, ao implementarmos o hardware, às vezes 1 representa um sinal lógico alto, mas também pode representar um sinal lógico baixo.

ativo

O sinal está logicamente alto ou verdadeiro.

inativo

O sinal está logicamente baixo ou falso.

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. A [Figura 4.4](#) mostra um exemplo genérico. 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 4.4](#) funciona corretamente. No Apêndice B, discutimos brevemente as outras limitações (como os tempos de setup e hold), bem como outras metodologias de sincronização.

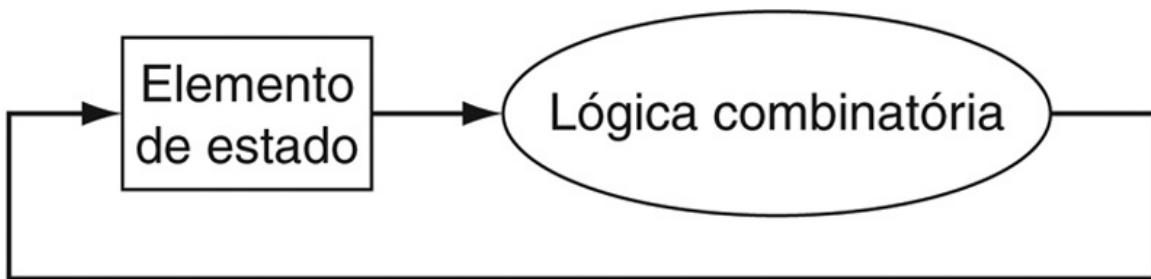


FIGURA 4.4 Uma metodologia acionada por transição permite que um elemento de estado seja lido e escrito no mesmo ciclo de clock sem criar uma disputa que poderia levar a valores de dados indeterminados.

É claro que o ciclo de clock ainda precisa ser longo o suficiente para que os valores de entrada sejam estáveis quando houver transição ativa do clock. O feedback não pode ocorrer dentro de um 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.

Para a arquitetura MIPS de 32 bits, 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 grossas. 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 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.

Verifique você mesmo

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.

Detalhamento

Há também uma versão de 64 bits da arquitetura MIPS e, naturalmente, a maioria dos caminhos em sua implementação teria 64 bits de largura.

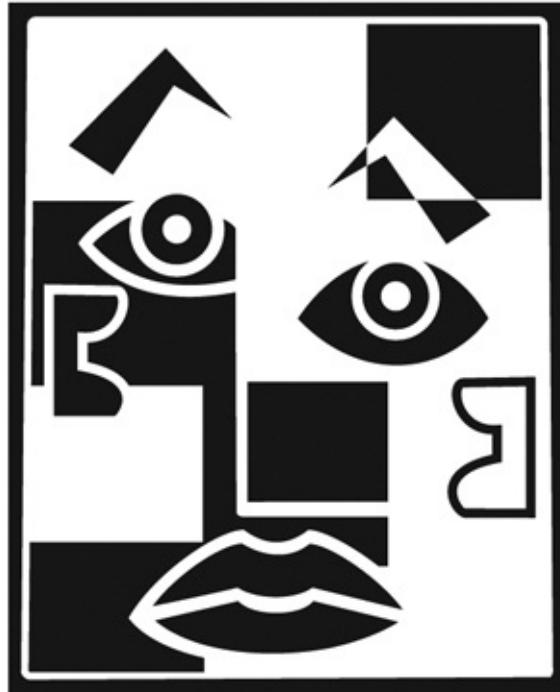
4.3. Construindo um caminho de dados

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, e depois desceremos por todos os níveis de **abstração**. Quando mostramos os elementos do caminho de dados, também mostramos seus sinais de controle. Usamos a abstração nesta explicação, começando de baixo para cima.

elemento do caminho de dados

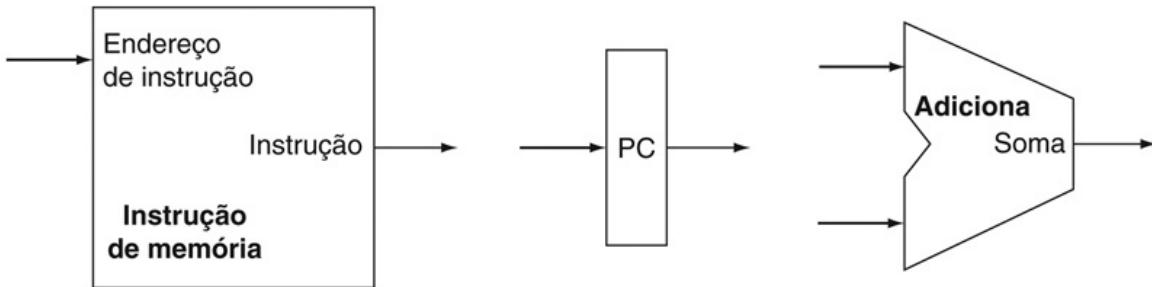
Uma unidade funcional usada para operar sobre os dados ou conter esses

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.



A B S T R A Ç Ã O

A [Figura 4.5a](#) 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 4.5b](#) mostra um registrador, que podemos chamar de **contador de programa (PC)**, que, como vimos no [Capítulo 2](#), é um registrador que contém o endereço da instrução atual. Finalmente, precisaremos de um somador a fim de incrementar o PC para o endereço da próxima instrução. Esse somador, que é combinacional, pode ser construído a partir da ALU que descrevemos 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 *Soma*, como na [Figura 4.5](#), para indicar que ela se tornou permanentemente um somador e não pode realizar as outras funções da ALU.



a. Instrução de memória

b. Contador de programa

c. Adicionador

FIGURA 4.5 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. (Precisaremos escrever na memória de instruções quando carregarmos o programa; isso não é difícil de incluir e o ignoramos em favor da simplicidade.) O contador de programa é um registrador de 32 bits que é 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.

contador de programa (PC)

O registrador que contém o endereço da instrução do programa sendo executado.

Para executar qualquer instrução, precisamos começar buscando a instrução na memória. A fim de 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 4.6 mostra como combinar os três elementos da Figura 4.5 para formar um caminho de dados que busca instruções e incrementa o PC de modo a obter o endereço da próxima instrução sequencial.

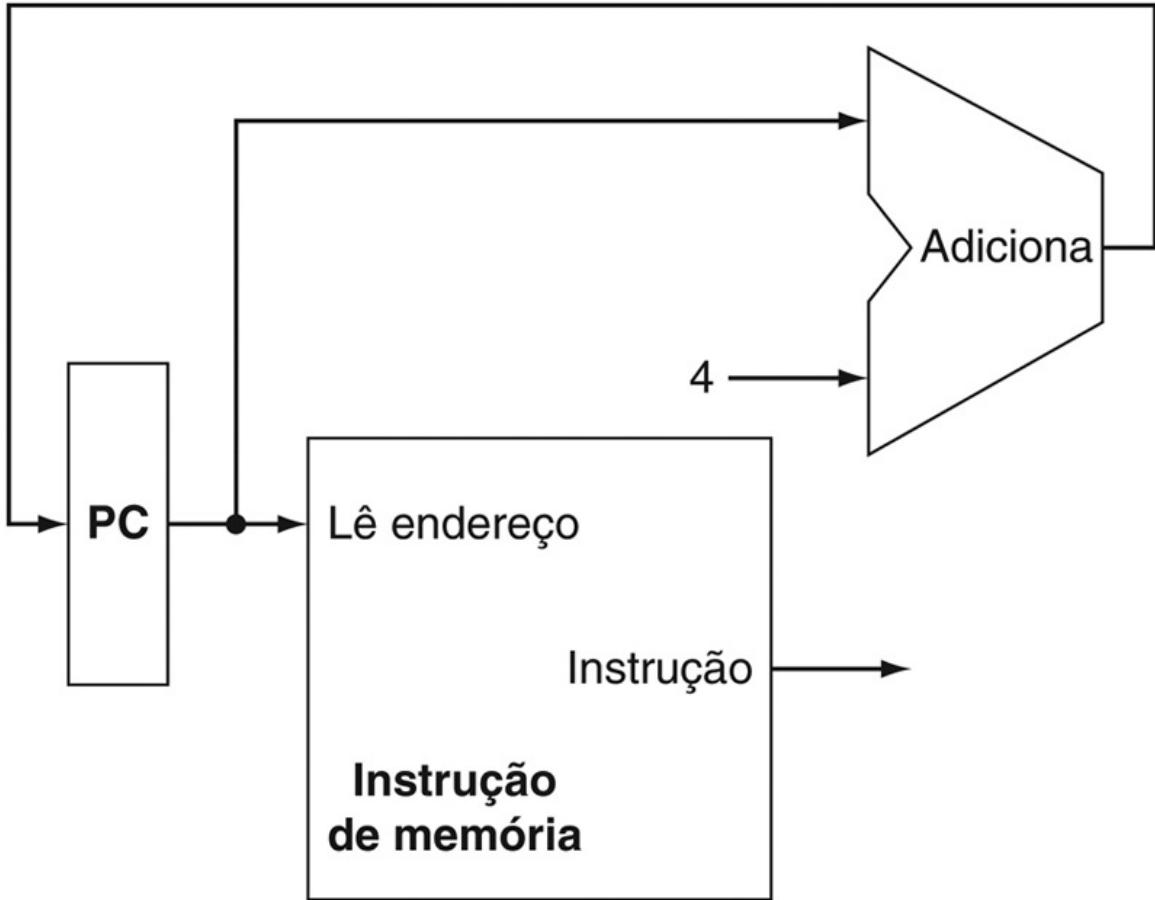


FIGURA 4.6 Uma parte do caminho de dados usada para buscar instruções e incrementar o contador do programa. A instrução buscada é usada por outras partes do caminho de dados.

Agora, vamos considerar as instruções de formato R (Figura 2.20). Todas elas leem dois registradores, realizam uma operação na ALU com o conteúdo dos registradores e escrevem o resultado em um registrador. 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.

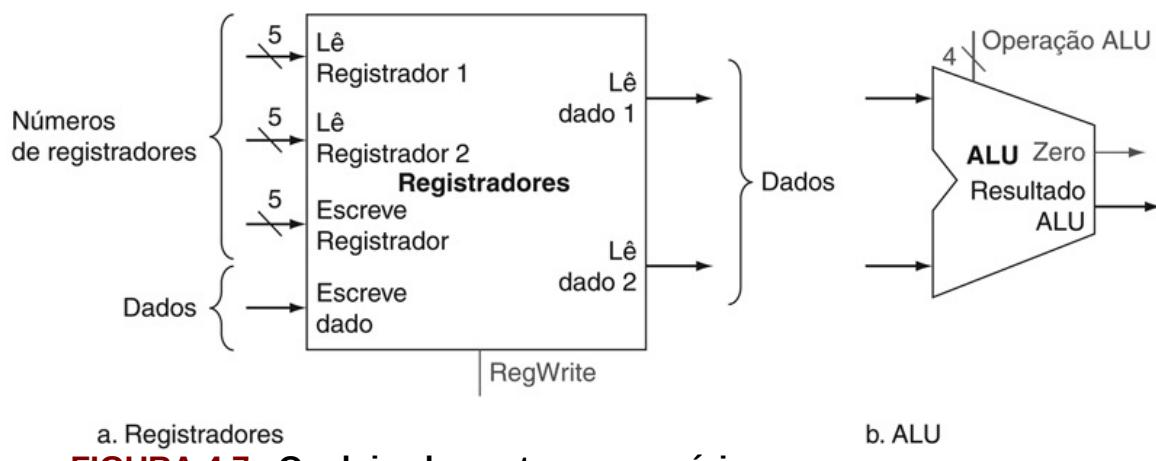
Os registradores de uso geral de 32 bits do processador são armazenados em uma estrutura chamada **banco de registradores**. Um banco de registradores é uma coleção de registradores em que qualquer registrador pode ser lido ou escrito especificando o número do registrador no banco. O banco de registradores contém o estado dos registradores do computador. Além disso,

precisaremos que uma ALU opere nos valores lidos dos registradores.

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.

Devido às instruções de formato R terem três operandos de registrador, precisaremos ler duas palavras de dados do banco de registradores e escrever uma palavra de dados no banco de registradores para cada instrução. A fim de que cada palavra 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 palavra 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 estar ativo para que uma escrita ocorra na transição do clock. A Figura 4.7a mostra o resultado; precisamos de um total de quatro entradas (três para números de registrador e uma para dados) e duas saídas (ambas para dados). 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.



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. Todas as entradas com o número do registrador para o banco de registradores possuem 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 4.9](#), quando discutiremos as exceções; até lá, elas serão omitidas.

A [Figura 4.7b](#) 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.

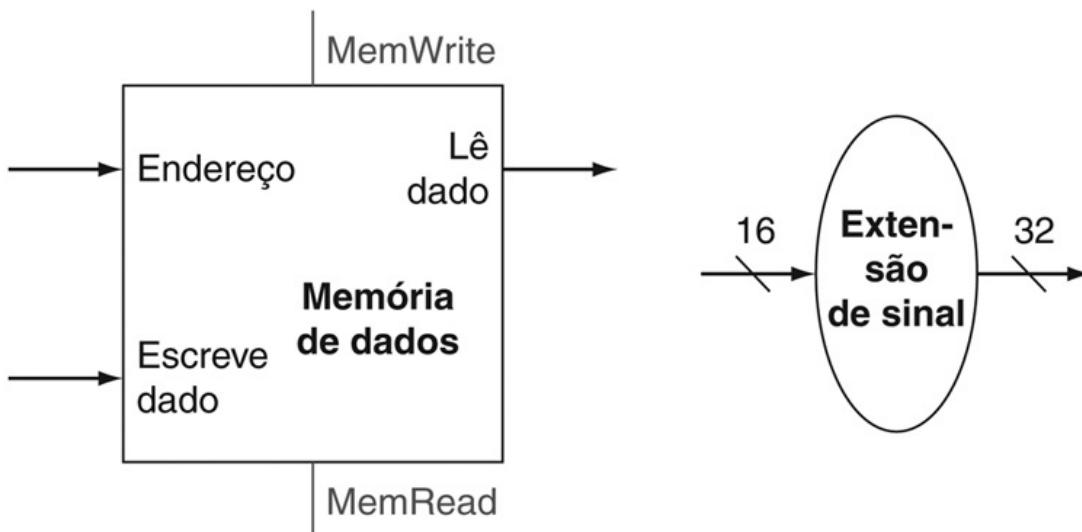
A seguir, considere as instruções MIPS load word e store word, que possuem o formato `lw $t1, offset_value($t2)` ou `sw $t1, offset_value($t2)`. Essas instruções calculam um endereço de memória somando o registrador de base, que é `$t2`, com o campo offset de 16 bits com sinal contido na instrução. Se a instrução for um store, o valor a ser armazenado também precisará ser lido do banco de registradores em que reside, em `$t1`. Se a instrução for um load, o valor lido da memória precisará ser escrito no banco de registradores no

registraror especificado, que é $\$t1$. Consequentemente, precisaremos do banco de registradores e da ALU da Figura 4.7.

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.

Além disso, precisaremos de uma unidade a fim de **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 para ler ou 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 4.8 mostra esses dois elementos.



- a. Unidade de memória de dados b. Unidade de extensão de sinal

FIGURA 4.8 As duas unidades necessárias para implementar loads e stores, além do banco de registradores e da ALU da Figura 4.7, 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 possa estar 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 5](#). 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 ([Capítulo 2](#)).

Consideramos 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.

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 ([Capítulo 2](#)) para os quais precisamos prestar atenção:

- 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 é um offset de uma palavra; esse deslocamento aumenta a faixa efetiva do campo offset por um fator de quatro vezes.

endereço de destino do desvio

O endereço especificado em um desvio, que se torna o novo contador do 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.

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 acompanha sequencialmente 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**.

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 acompanha sequencialmente o desvio.

Portanto, o caminho de dados de desvio precisa realizar 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.) A [Figura 4.9](#) mostra a estrutura do segmento do caminho de dados que lida com os desvios. 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 4.8](#), e um somador. Para realizar a comparação, precisamos usar o banco de registradores mostrado na [Figura 4.7a](#) a fim de 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 B. Como essa ALU fornece um sinal de saída que indica se o resultado era 0, podemos enviar os dois operandos de registrador para a ALU com o conjunto de controle de modo a 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.

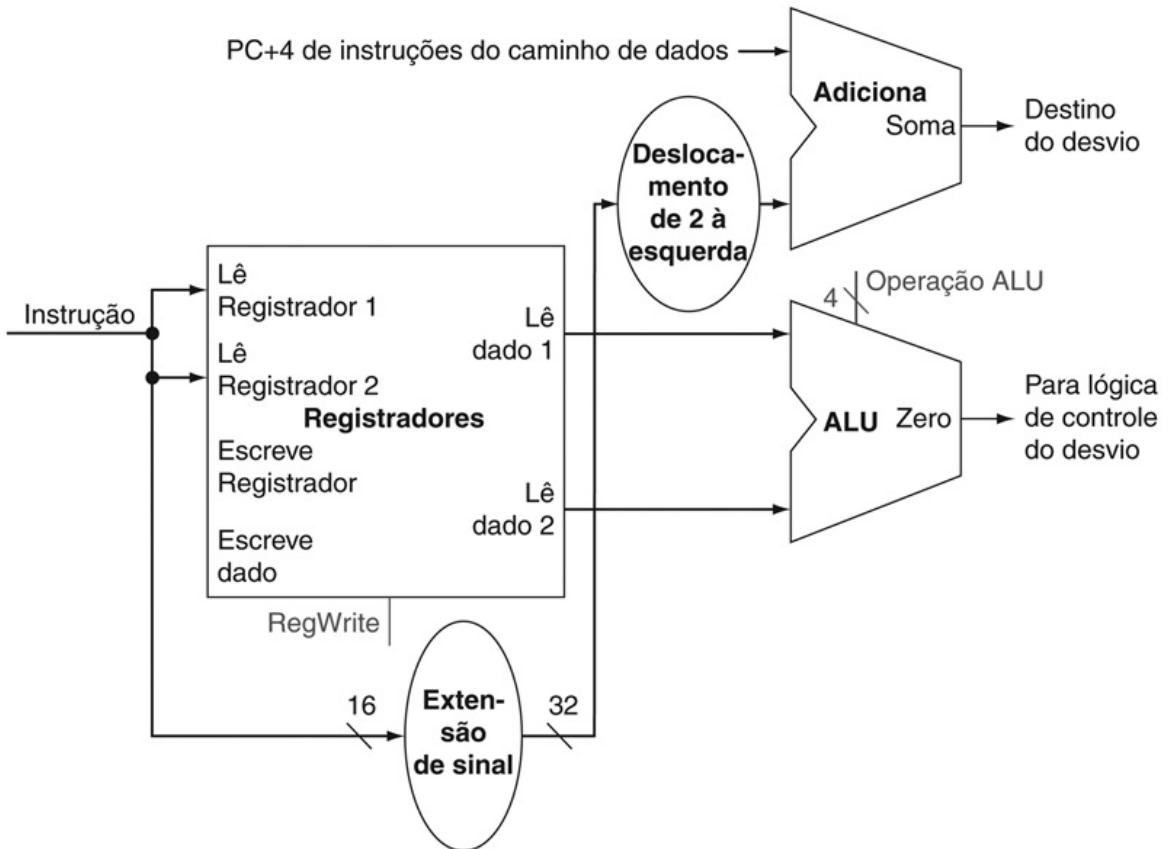


FIGURA 4.9 O caminho de dados para um desvio usa a ALU a fim de avaliar a condição de desvio e um somador separado para calcular o destino do desvio como a soma do PC incrementado e os 16 bits mais baixos da instrução com sinal estendido (o deslocamento do desvio), deslocados de 2 bits para a esquerda.

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 de baixa ordem 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 incrementado ou o destino do desvio deve substituir o PC, com base na saída Zero da ALU.

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 atrasados**, isso significa que a instrução imediatamente posterior ao desvio é sempre executada, *independente* da 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 desvio atrasado primeiro executa a instrução imediatamente posterior ao desvio na ordem sequencial antes de desviar para o endereço de destino do desvio. A motivação para os desvios atrasados surge de como o pipelining afeta os desvios (Seção 4.8). Para simplificar, geralmente ignoramos os desvios atrasados neste capítulo e implementamos uma instrução beq como não sendo atrasado.

desvio atrasado

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 da 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.

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 referência à memória e 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 4.10 mostra a parte operacional do caminho de dados combinado.

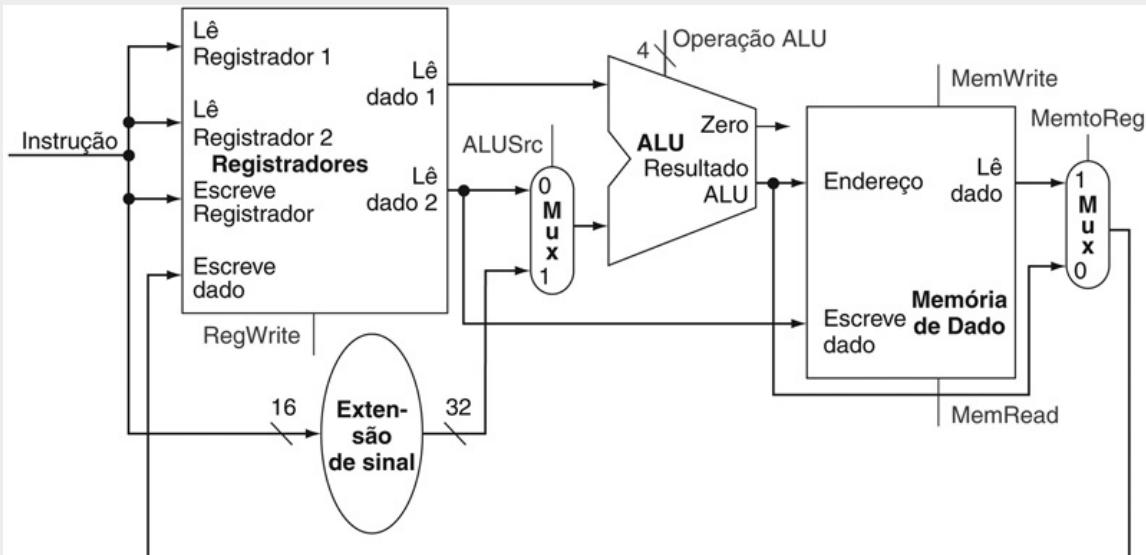


FIGURA 4.10 O caminho de dados para as instruções de acesso à memória e as instruções tipo R.

Este exemplo mostra como um único caminho de dados pode ser montado, a partir das partes nas Figuras 4.7 e 4.8 acrescentando multiplexadores. Dois multiplexadores são necessários, como descrito no exemplo.

Agora, podemos combinar todas as partes de modo a criar um caminho de dados simples para a arquitetura do núcleo MIPS incluindo um caminho de dados para busca de instruções (Figura 4.6), o caminho de dados das instruções tipo R e de acesso à memória (Figura 4.10) e o caminho de dados para desvios (Figura 4.9). A Figura 4.11 mostra o caminho de dados que obtemos compondo as partes separadas. A instrução de desvio usa a ALU principal para comparação dos operandos registradores, de modo que precisamos manter o somador da Figura 4.9, a fim de 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.

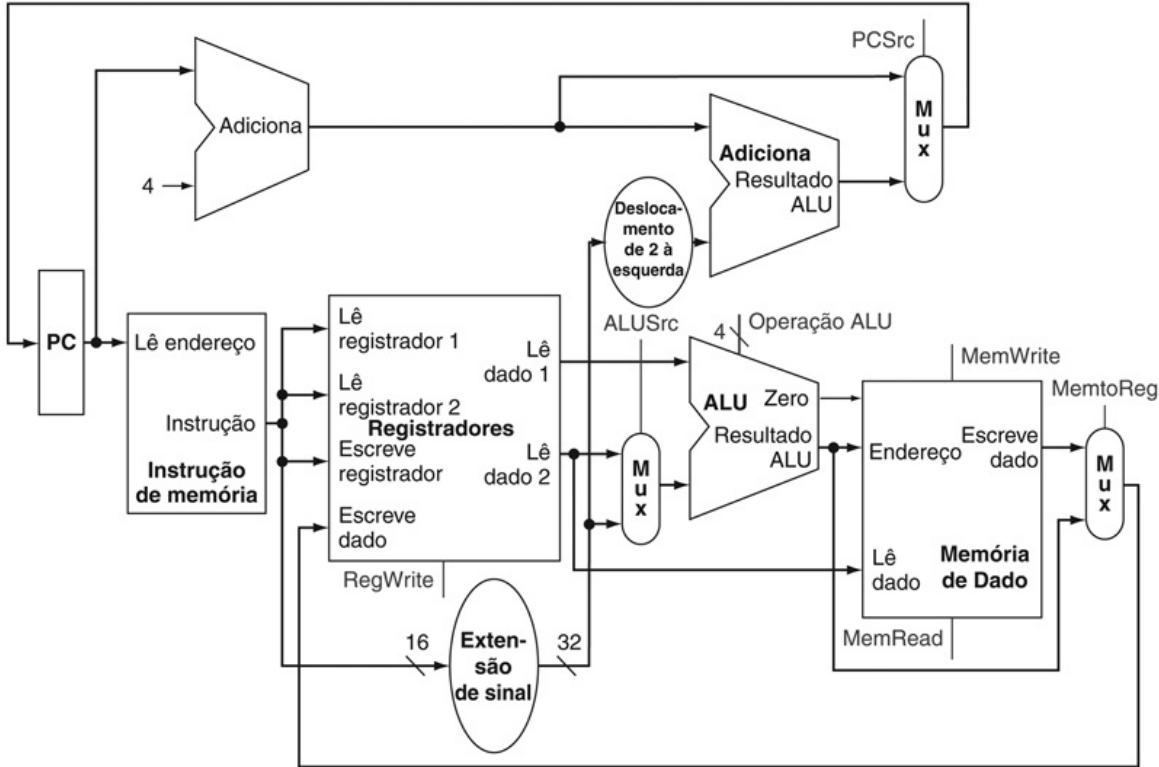


FIGURA 4.11 O caminho de dados simples para a arquitetura MIPS combina os elementos necessários para diferentes classes de instrução.

Os componentes vêm das Figuras 4.6, 4.9 e 4.10. Este caminho de dados pode executar as instruções básicas (load-store word, operações da ALU e desvios) em um único ciclo de clock.

Apenas um multiplexador adicional é necessário para integrar os desvios. O suporte para jumps será incluído mais tarde.

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, o controle seletor de cada multiplexador e 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.

Verifique você mesmo

- Qual das seguintes afirmativas é correta para uma instrução load?
Consulte a Figura 4.10.
 - MemtoReg deve ser definido para fazer com que os dados da memória sejam enviados ao banco de registradores.

- b. MemtoReg deve ser definido para fazer com que o registrador de destino correto seja enviado ao banco de registradores.
- c. Não precisamos nos importar com MemtoReg para loads.

- II. O caminho de dados de ciclo único descrito conceitualmente nesta seção *precisa* ter memórias de instrução e dados separadas, porque:
- a. os formatos dos dados e das instruções são diferentes no MIPS, e, portanto, memórias diferentes são necessárias.
 - b. ter memórias separadas é menos dispendioso.
 - c. o processador opera em um ciclo e não pode usar uma memória de porta simples para dois acessos diferentes dentro desse ciclo.

4.4. 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 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

A ALU MIPS no Apêndice B define as 6 combinações a seguir das quatro entradas de controle:

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 não encontradas no subconjunto que estamos implementando.) 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), dependendo do valor do campo funct (ou function – função) de 6 bits nos bits menos significativos da instrução ([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 ALUOp. ALUOp 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 4.12](#), mostramos como definir as entradas do controle da ALU com base no controle ALUOp de 2 bits e no código de função de 6 bits. Mais adiante neste capítulo, veremos como os bits de ALUOp 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		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 4.12 A forma como os bits de controle da ALU são definidos, dependendo dos bits de controle de ALUOp e dos diferentes códigos de função para as instruções tipo R.

O opcode, que aparece na primeira coluna, determina a definição dos bits de ALUOp. Todas as codificações são mostradas em binário. Observe que quando o código de ALUOp é 00 ou 01, a ação da ALU desejada não depende do campo de código de função; nesse caso, dizemos que “não nos importamos” (don’t care) com o valor do código de função e o campo funct aparece como XXXXXX. Quando o valor de ALUOp é 10, então o código de função é usado para definir a entrada do controle da ALU. Veja o [Apêndice B](#).

Esse estilo de usar vários níveis de decodificação — ou seja, a unidade de controle principal gera os bits de ALUOp, 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, pois a velocidade da unidade de controle normalmente é essencial para o tempo de ciclo de clock.

Há várias maneiras diferentes de implementar o mapeamento do campo ALUOp de 2 bits e do campo funct de 6 bits para os 3 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 ALUOp 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 ALUOp, como fizemos na [Figura 4.13](#); essa **tabela verdade** mostra como o controle da ALU de 4 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 estão zeradas ou que não nos interessam.

ALUOp		Campo funct							Operação
ALUOp1	ALUOp2	F5	F4	F3	F2	F1	F0		
0	0	X	X	X	X	X	X	0010	
X	1	X	X	X	X	X	X	0110	
1	X	X	X	0	0	0	0	0010	
1	X	X	X	0	0	1	0	0110	
1	X	X	X	0	1	0	0	0000	
1	X	X	X	0	1	0	1	0001	
1	X	X	X	1	0	1	0	0111	

FIGURA 4.13 A tabela verdade para os 4 bits de controle da ALU (chamados Operação).

As entradas são ALUOp e o campo de código de função. Apenas

as entradas para as quais o controle da ALU é ativado são mostradas. Algumas entradas don't care foram incluídas. Por exemplo, como ALUOp 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.

tabela verdade

Pela lógica, uma representação de uma operação lógica listando todos os valores das entradas e em seguida, em cada caso, mostrando quais deverão ser as saídas resultantes.

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 ALUOp são 00, como na primeira linha da tabela na [Figura 4.13](#), sempre definimos o controle da ALU em 0010, 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.

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.

Uma vez construída a tabela, ela pode ser otimizada e depois transformada em portas lógicas. Esse processo é completamente mecânico.

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 4.11](#). A fim de 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 load-store. A [Figura 4.14](#) mostra esses formatos.

Campo	0	rs	rt	rd	shamt	funct
Posição de bit	31:26	25:21	20:16	15:11	10:6	5:0
a. Instrução do tipo R						
Campo	35 ou 43	rs	rt		address	
Posição de bit	31:26	25:21	20:16		15:0	
b. Carrega ou armazena instrução						
Campo	4	rs	rt		address	
Posição de bit	31:26	25:21	20:16		15:0	
c. Instrução de desvio						

FIGURA 4.14 As três classes de instrução (tipo R, load/store 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 de modo a formar o endereço de memória. Com os loads, rt é o registrador de destino para o valor lido. Com 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 por igualdade. O campo address de 16 bits tem seu sinal estendido, é deslocado e somado ao PC + 4 para calcular o endereço de

destino do desvio.

Existem várias observações importantes sobre esses formatos de instrução em que nos basearemos:

- Campo op, também chamado **opcode** no [Capítulo 2](#), 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.
- Registrador de base para as instruções load e store está sempre nas posições de bit 25:21 (rs).
- Offset de 16 bits para branch equal, load e store está sempre nas posições 15:0.
- 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 a fim de selecionar que campo da instrução será usado para indicar o número de registrador a ser escrito.

opcode

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

A vantagem do primeiro princípio de projeto do [Capítulo 2](#) — *a simplicidade favorece a regularidade* — aparece aqui na especificação do controle.

Usando essas informações, podemos acrescentar os rótulos de instrução e o multiplexador extra (para a entrada Escreve registrador do banco de registradores) no caminho de dados simples. A [Figura 4.15](#) mostra essas adições, além do bloco de controle da ALU, os sinais de escrita para elementos 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.

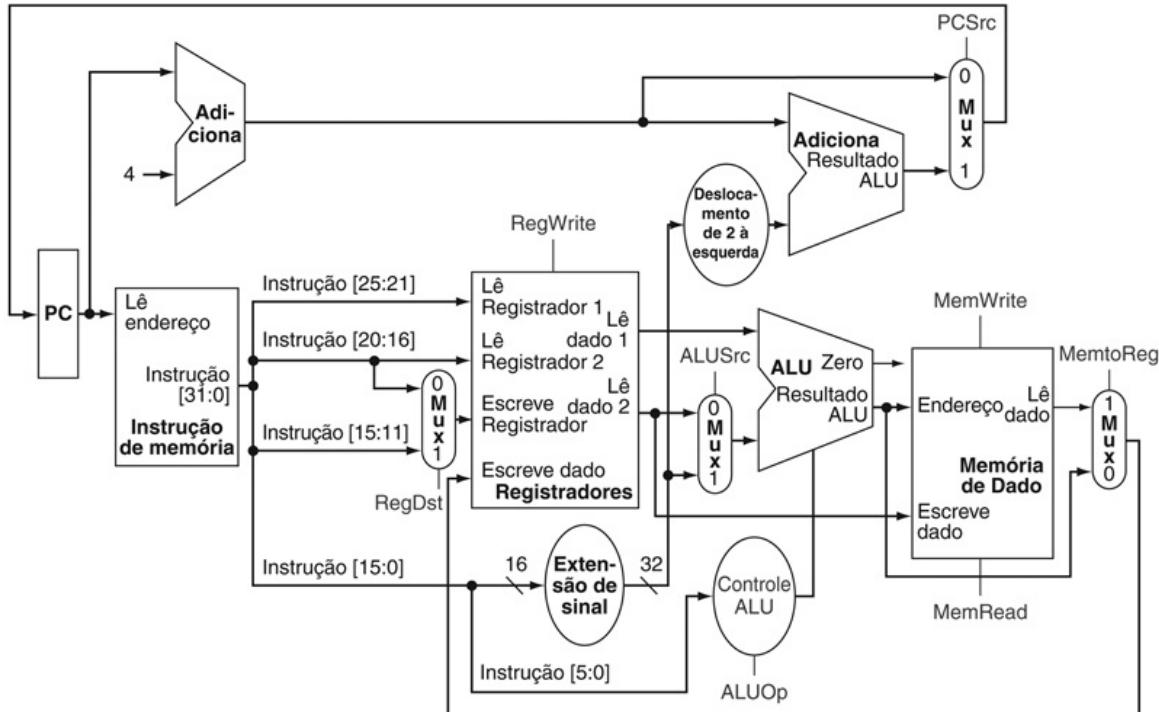


FIGURA 4.15 O caminho de dados da **Figura 4.11** com todos os multiplexadores necessários e todas as linhas de controle identificadas.

As linhas de controle são mostradas em tons de cinza. 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.

A Figura 4.15 mostra sete linhas de controle de um único bit mais o sinal de controle ALUOp de 2 bits. Já definimos como o sinal de controle ALUOp 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 4.16 descreve a função dessas sete linhas de controle.

Nome do sinal	Efeito quando inativo	Efeito quando ativo
RegDst	O número do registrador destino para 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 à entrada Dados para escrita do banco de registradores vem da ALU.	O valor enviado à entrada Dados para escrita do banco de registradores vem da memória de dados.

FIGURA 4.16 O efeito de cada um dos sete sinais de controle.

Quando o controle de um bit 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 PCScrPCScr. 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 PCScrPCScr, 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 4.16 e dois para ALUOp) podem agora ser definidos baseados nos seis sinais de entrada da unidade de controle, que são os bits de opcode 31 a 26. A Figura 4.17 mostra o caminho de dados

com a unidade de controle e os sinais de controle.

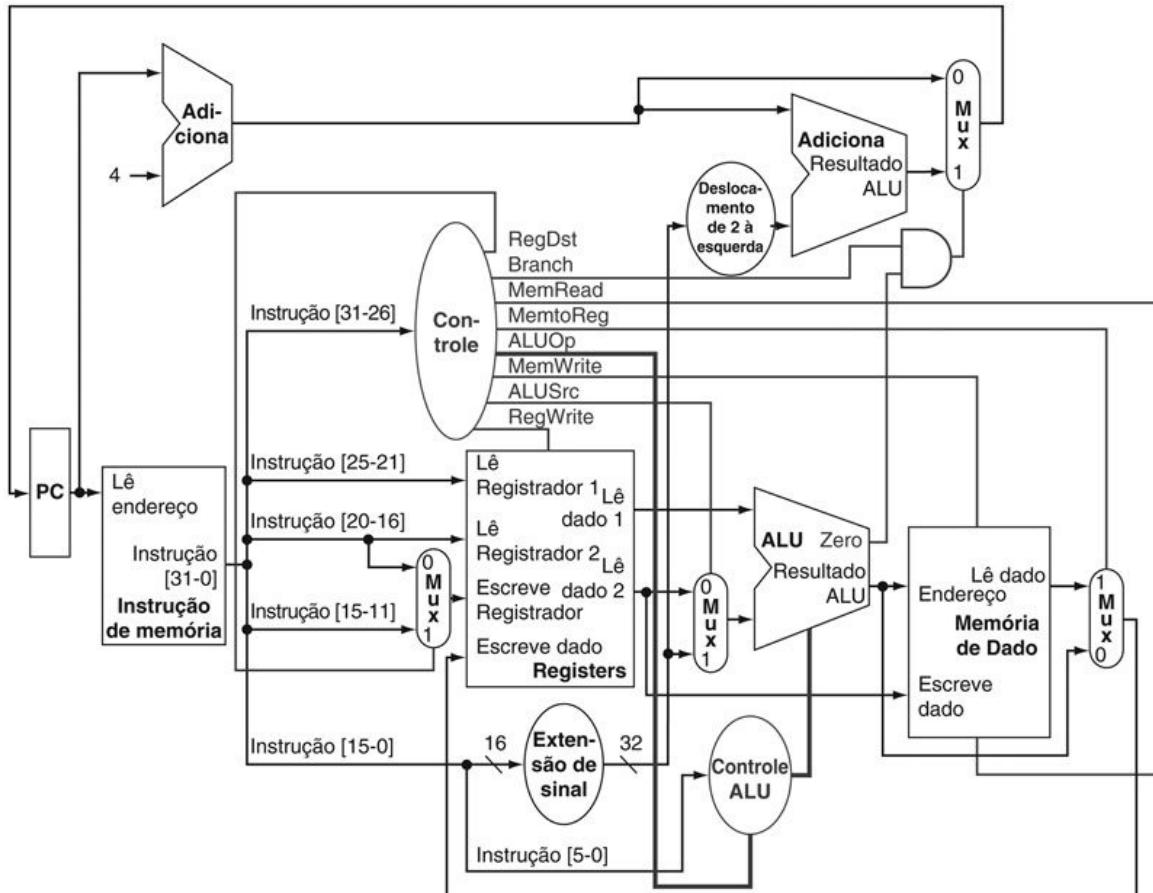


FIGURA 4.17 O caminho de dados simples com a unidade de controle.

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, ALUSrcALUScr e MemtoReg), três sinais para controlar leituras e escritas no banco de registradores e na memória de dados (WriteReg, ReadMem e WriteMem), um sinal de 1 bit usado na determinação de um possível desvio (Branch) e um sinal de controle de 2 bits para a ALU (ALUOp). Uma porta AND é usada de modo a 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 PCScr 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.

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 4.18](#) descreve como os sinais de controle devem ser definidos para cada opcode; essas informações seguem diretamente das [Figuras 4.12, 4.16 e 4.17](#).

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 4.18 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 ALUScr e RegDst são definidos. Além disso, uma instrução tipo R escreve em um registrador (WriteReg = 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 ALUOp para as instruções tipo R é definido como 10 a fim de 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 ALUScr e ALUOp são definidos para realizar o cálculo do endereço. ReadMem e WriteMem são definidos para realizar o acesso à memória. Finalmente, RegDst e WriteReg são definidos 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 ALUOp para um desvio é definido como uma subtração (controle da ALU = 01), usada para testar a igualdade. Repare que o campo MemtoReg é irrelevante quando o sinal WriteReg 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 MemtoReg nas duas últimas linhas da tabela é substituída por X (don't care). Os don't care também podem ser adicionados a RegDst quando WriteReg 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.

Operação do caminho de dados

Com as informações contidas nas [Figuras 4.16 e 4.18](#), 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 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 4.19](#) mostra a operação do caminho de dados para uma instrução tipo R, como `add $t1,$t2,$t3`. Embora tudo ocorra em um 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.

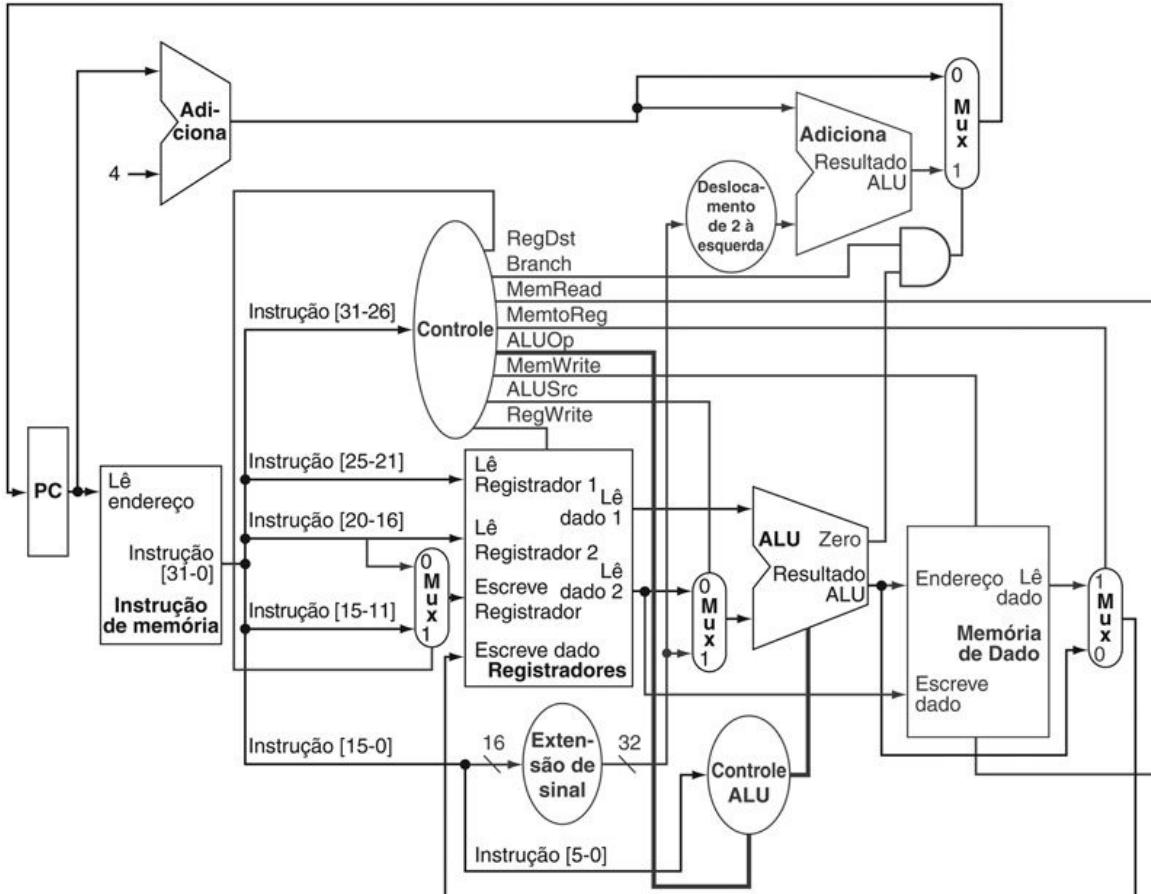


FIGURA 4.19 O caminho de dados em operação para uma instrução tipo R como `add $t1,$t2,$t3`.

As linhas de controle, as unidades do caminho de dados e as conexões que estão ativas aparecem destacadas.

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).
- Da mesma forma, podemos ilustrar a execução de um load word, como

`lw $t1, offset($t2)`

em um estilo semelhante à Figura 4.19. A Figura 4.20 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.

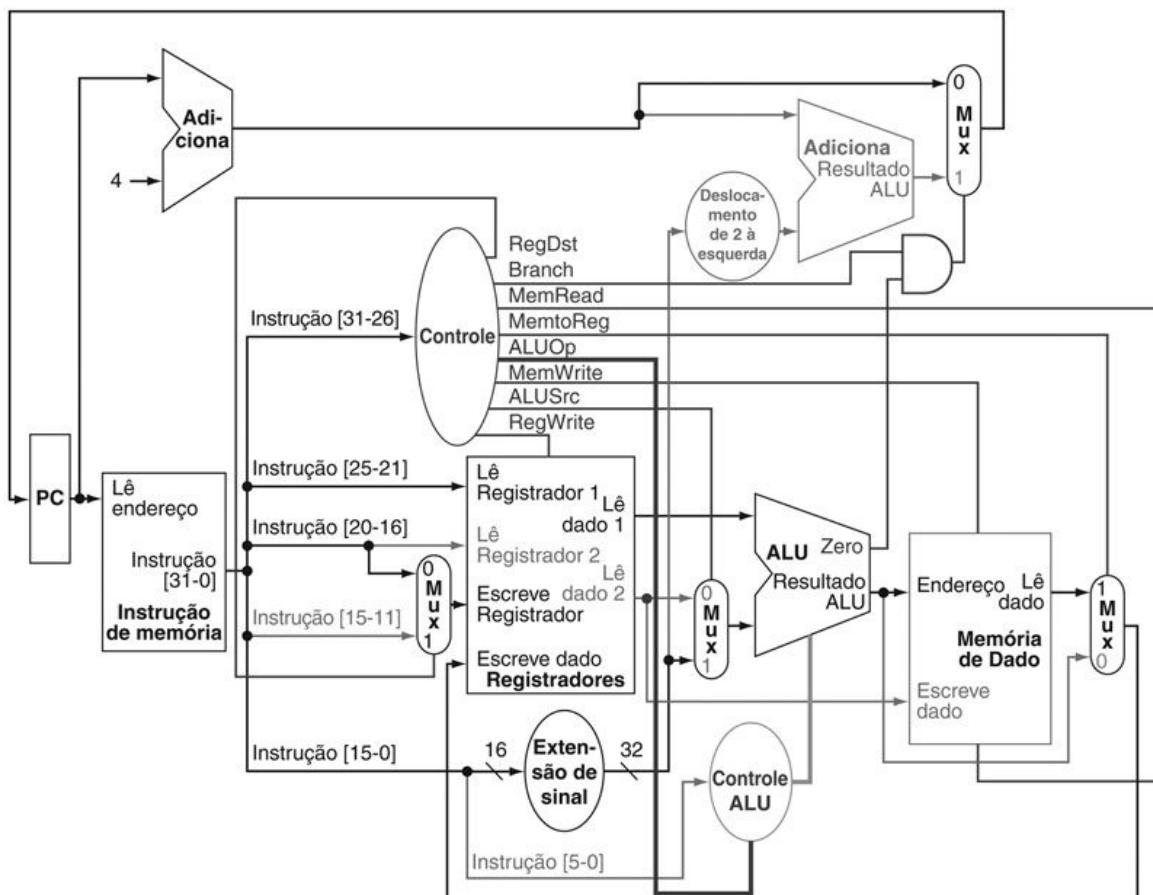


FIGURA 4.20 O caminho de dados em operação para uma instrução load.

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.

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 ($\$t1$).

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 4.21](#) mostra as quatro etapas da execução:

1. Uma instrução é buscada da memória de instruções e o PC é incrementado.

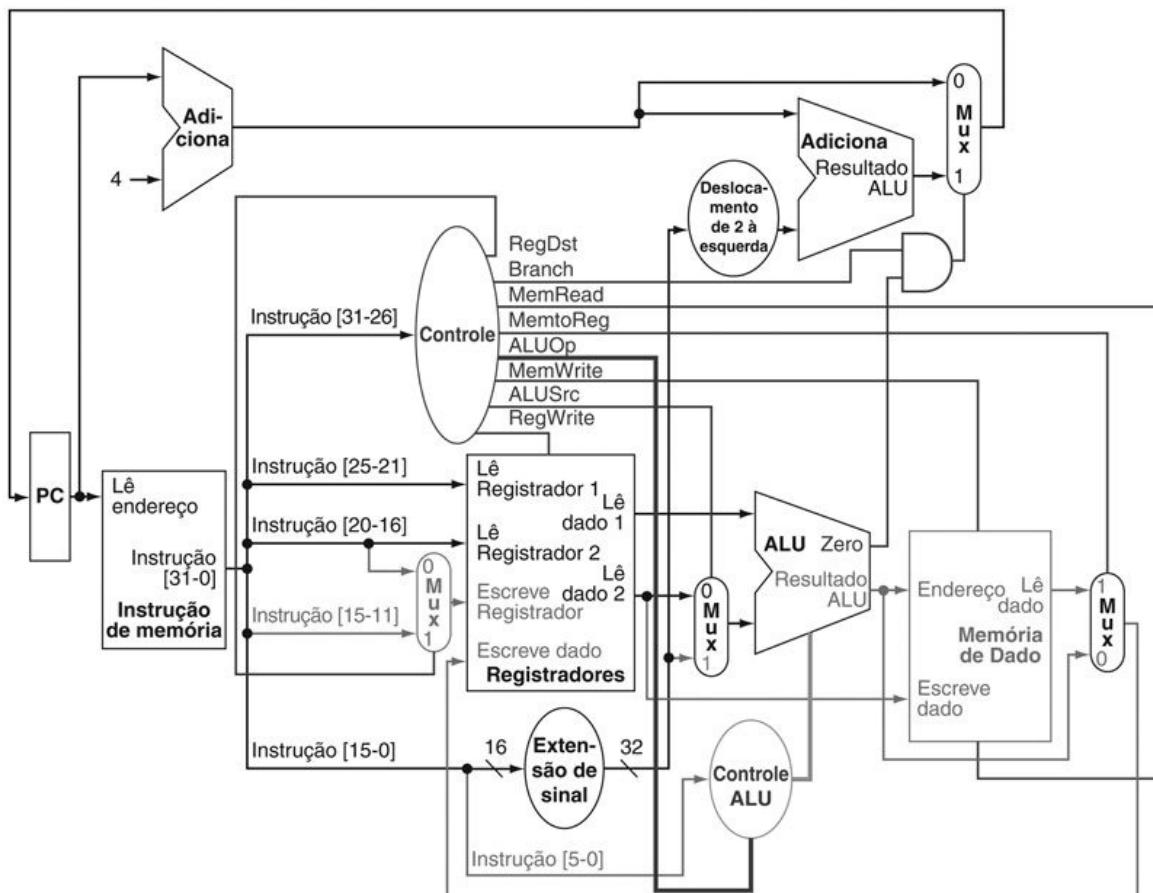


FIGURA 4.21 O caminho de dados em operação para uma instrução branch-on-equal.

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 na seleção do próximo contador de programa dentre os dois candidatos.

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 qual resultado do somador deve ser armazenado no PC.

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 com precisão usando o conteúdo da [Figura 4.18](#). 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 4.22](#) 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.

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 4.22 A função de controle para a implementação de ciclo único simples é completamente especificada por essa tabela verdade.

A parte superior da tabela fornece combinações de sinais de entrada que correspondem aos quatro opcodes, um por coluna, 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 para cada um dos quatro opcodes. Portanto, a saída WriteReg 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 [see original, pg 269], 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.

Agora que temos uma **implementação de ciclo único** da maioria dos conjuntos de instruções MIPS básico, vamos acrescentar a instrução jump para mostrar como o caminho de dados básico e o controle podem ser estendidos ao lidar com outras instruções no conjunto de instruções.

implementação 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.

Implementando jumps

Exemplo

A Figura 4.17 mostra a implementação de várias 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 4.17 para incluir a instrução jump. Descreva como definir quaisquer novas linhas de controle.

Resposta

A instrução jump, mostrada na Figura 4.23, 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 imediato de 26 bits na instrução. 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);
- campo de 26 bits imediato da instrução jump;
- os bits 00_{bin} .

Campo	000010	endereço
Posições dos bits	31:26	25:0

FIGURA 4.23 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 endereço de 26 bits na instrução jump e pela adição de 00 como os dois bits menos significativos.

A Figura 4.24 mostra a adição do componente para jump à Figura 4.17. Um outro multiplexador é usado na seleção da 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.

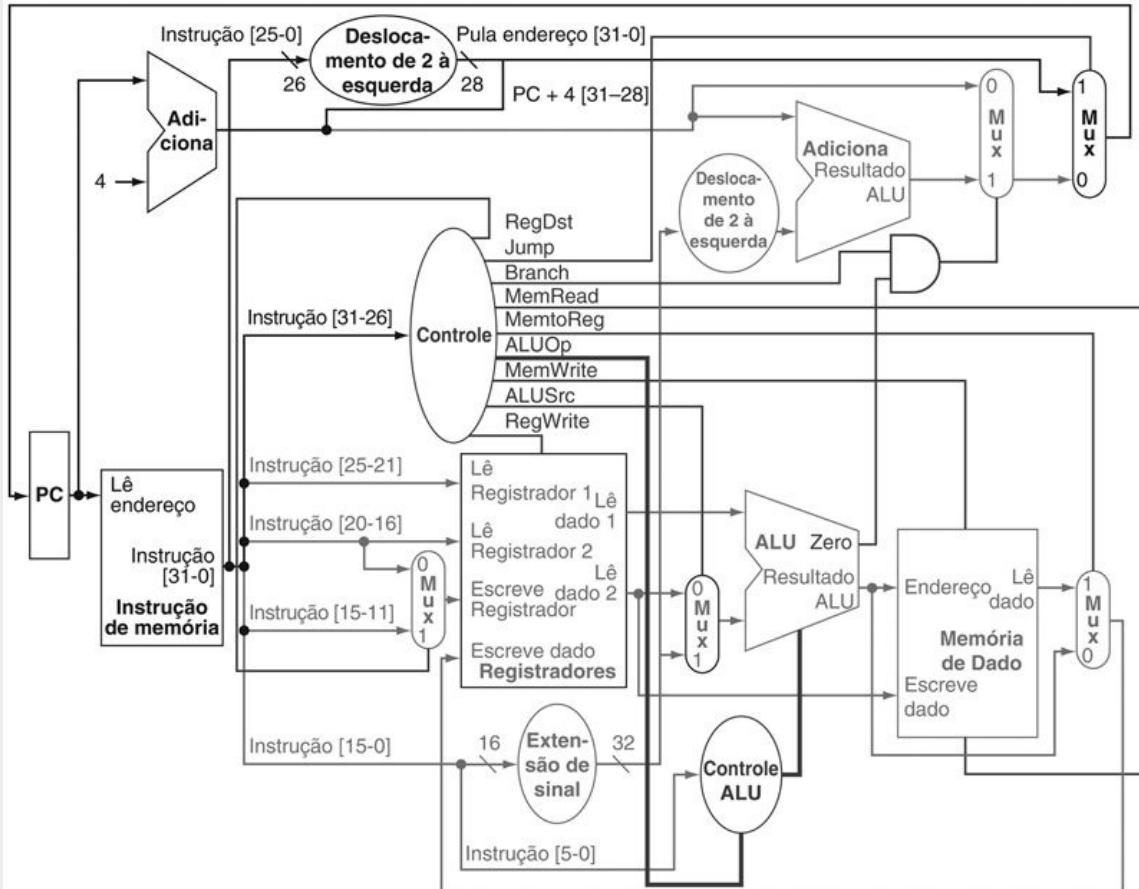


FIGURA 4.24 O controle e o caminho de dados simples são estendidos para lidar com a instrução jump.

Um multiplexador adicional (no canto superior direito) é usado na escolha 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.

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. É claro, o ciclo de clock é determinado pelo caminho mais longo possível no processador. Esse caminho é, quase certamente, uma instrução load, que usa cinco unidades funcionais em 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 ([Capítulo 1](#)), o desempenho geral de uma implementação de ciclo único provavelmente será fraco, já que o ciclo de clock é muito longo.

O ônus de usar o projeto de ciclo único com um ciclo de clock fixo é significativo, mas poderia ser considerado aceitável para este pequeno conjunto de instruções. Historicamente, os primeiros computadores 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 orientações mais complexas, esse projeto de ciclo único decididamente não funcionaria bem.

Como precisamos considerar que o ciclo de clock é igual ao atraso do 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 o nosso princípio básico de projeto do [Capítulo 2](#) de tornar o **caso comum veloz**.



CASO COMUM VELOZ

Na próxima seção, veremos outra técnica de implementação, chamada pipelining, que usa um caminho de dados muito semelhante ao caminho de dados de ciclo único, mas é muito mais eficiente por ter uma vazão muito mais alta. O pipelining melhora a eficiência executando múltiplas instruções simultaneamente.

Verifique você mesmo

Veja os sinais de controle na Figura 4.22. Você consegue combinar alguns deles? 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 trocar um sinal pelo outro sem incluir um inverter?

4.5. Visão geral de pipelining

Nunca perca tempo.

Provérbio americano

Pipelining é uma técnica de implementação em que várias instruções são sobrepostas na execução. Hoje, a técnica de pipelining é praticamente universal.

pipelining

Uma técnica de implementação em que várias instruções são sobrepostas na execução, semelhante a uma linha de montagem.

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 4.10 e 4.11](#), a fim de ver uma introdução às técnicas de pipelining avançadas, utilizadas nos processadores mais recentes, como o Intel Core i7 e o ARM Cortex-A8. Se estiver interessado em explorar a anatomia de um computador com pipeline, esta seção é uma boa introdução às [Seções de 4.6 a 4.9](#).

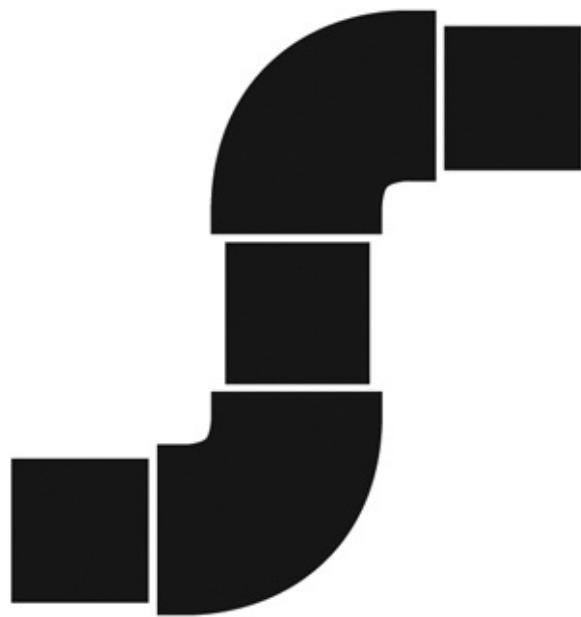
Qualquer um que tenha lavado muitas roupas intuitivamente já usou pipelining. A técnica sem pipeline para lavar roupas seria:

1. Colocar a trouxa de roupa suja na lavadora.
2. Quando a lavadora terminar, colocar a trouxa de roupa molhada na

secadora.

3. Quando a secadora terminar, colocar a trouxa de roupa 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 de roupa suja.



PIPELINING

A técnica com *pipeline* leva muito menos tempo, como mostra a Figura 4.25. Assim que a lavadora terminar com a primeira trouxa e ela for colocada na secadora, você carrega a lavadora com a segunda trouxa de roupa suja. Quando a primeira trouxa estiver seca, você a coloca na tábua para começar a passar e dobrar, move a trouxa de roupa molhada para a secadora e a próxima trouxa de roupa suja para a lavadora. Em seguida, você pede a seu colega para guardar a primeira remessa, começa a passar e dobrar a segunda, a secadora 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.

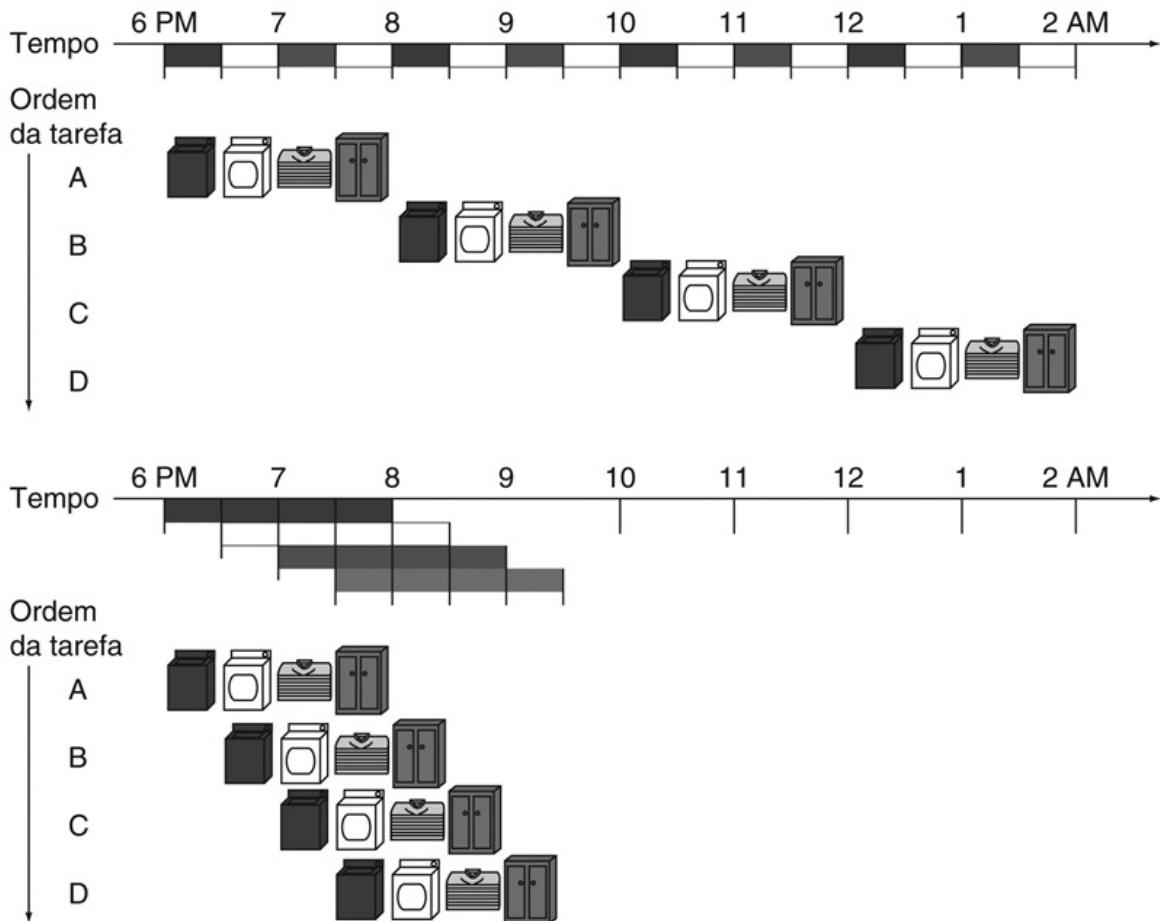


FIGURA 4.25 A analogia da lavagem de roupas para pipelining.

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 oito 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.

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 esteja seca, e seja 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. 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 de conclusão do 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 cinco vezes o tempo de uma trouxa, enquanto 20 trouxas de lavagem sequencial levariam 20 vezes o tempo de uma trouxa. O ganho foi de apenas 2,3 vezes na [Figura 4.25](#) porque mostramos apenas quatro trouxas. Observe que, no início e no final da carga de trabalho na versão com pipeline da [Figura 4.25](#), o pipeline não está completamente cheio. Esse efeito no início 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

Exemplo

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 um ciclo de clock, com uma implementação com pipeline. Os tempos de operação para as principais unidades funcionais neste exemplo são de 200 ps para acesso à memória, 200 ps para operação com ALU e 100 ps para leitura ou escrita de registradores. No modelo de ciclo único, cada instrução leva exatamente um ciclo de clock, de modo que o ciclo precisa ser esticado para acomodar a instrução mais lenta.

Resposta

A Figura 4.26 mostra o tempo exigido para cada uma das oito instruções. O projeto de ciclo único precisa contemplar a instrução mais lenta — na Figura 4.26, ela é lw — de modo que o tempo exigido para cada instrução é 800 ps. Assim como na Figura 4.25, a Figura 4.27 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 3×800 ps ou 2.400 ps.

Classe de instrução	Busca de instrução	Leitura do registrador	Operação ALU	Acesso de dados	Escrita do registrador	Tempo Total
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

FIGURA 4.26 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.

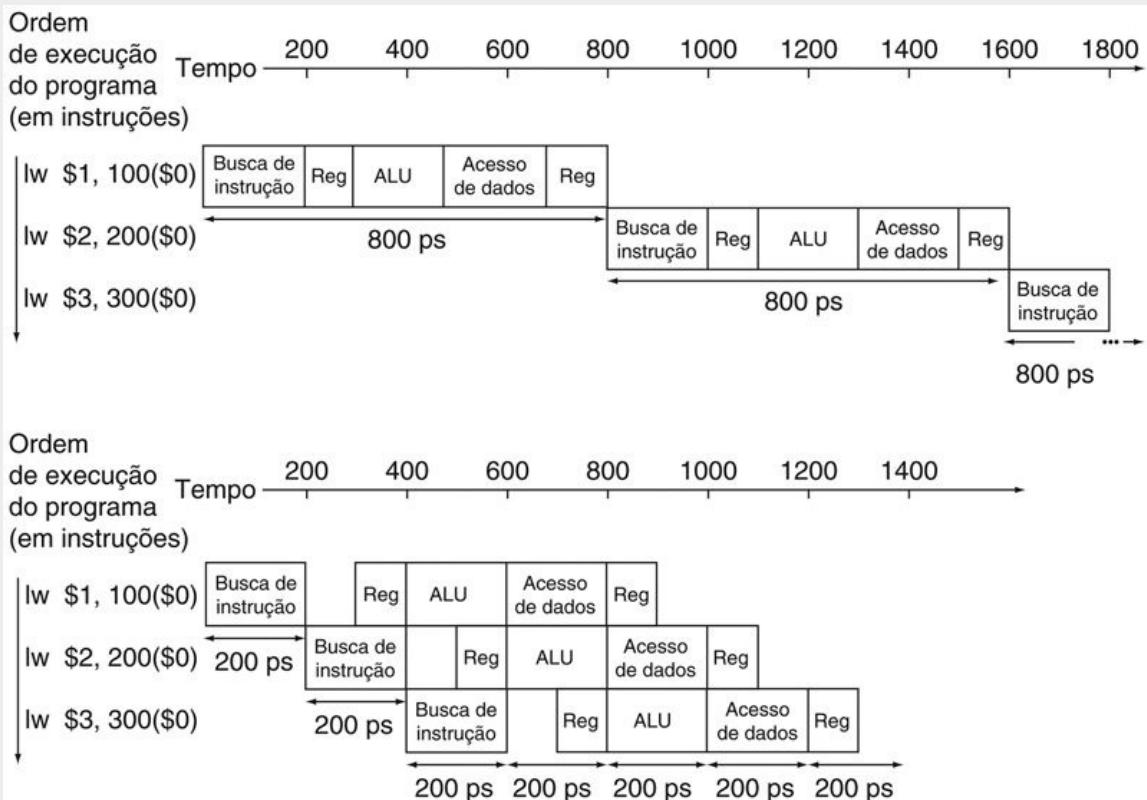


FIGURA 4.27 Em cima, execução em ciclo único, sem pipeline, versus execução com pipeline, embaixo.

Ambas utilizam os mesmos componentes de hardware, cujo tempo está listado na Figura 4.26. Neste caso, vemos um ganho de velocidade de quatro vezes no tempo médio entre as instruções, de 800 ps para 200 ps. Compare com a Figura 4.25. 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.

Todos os estágios do pipeline utilizam um único ciclo de clock, de modo que ele 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 800 ps, embora algumas instruções possam ser tão rápidas quanto 500 ps, o ciclo de clock da execução com pipeline precisa ter o ciclo de clock, no pior caso, de 200 ps, embora alguns estágios levem apenas

100 ps. O uso de pipeline ainda oferece uma melhoria de desempenho de quatro vezes: o tempo entre a primeira e a quarta instruções é de 3×200 ps ou 600 ps.

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:

$$\text{Tempo entre instruções}_{\text{com pipeline}} = \frac{\text{Tempo entre instruções}_{\text{sem pipeline}}}{\text{Número de estágios do pipeline}}$$

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 pipeline; 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 800 ps ou um ciclo de clock de 160 ps. 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 1.400 ps *versus* 2.400 ps. 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 200 ps ao tempo de execução total. O tempo de execução total seria $1.000.000 \times 200$ ps + 1.400 ps, ou 200.001.400 ps. No exemplo sem pipeline, acrescentaríamos 1.000.000 instruções, cada uma exigindo 800 ps de modo que o tempo de execução total seria $1.000.000 \times 800$ ps + 2.400 ps ou 800.002.400 ps. 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:

$$\frac{800.002.400 \text{ ps}}{200.001.400 \text{ ps}} \simeq \frac{800 \text{ ps}}{200 \text{ ps}} \simeq 4,00$$

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 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 x86, no qual as instruções variam de 1 byte a 15 bytes, a técnica de pipelining é muito mais desafiadora. As implementações recentes da arquitetura x86 na realidade traduzem instruções x86 em micro-operações simples, que se parecem com instruções MIPS e depois usam um pipeline de micro-operações no lugar das instruções x86 nativas! ([Seção 4.10.](#))

Em segundo lugar, o MIPS tem apenas alguns poucos formatos de instrução, com os campos do 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 simétricos, 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 x86, 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.

Hazards de pipeline

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

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.

hazard estrutural

Uma ocorrência em que uma instrução planejada não pode ser executada no ciclo de clock apropriado, pois o hardware não admite a combinação de instruções definidas para executar em determinado ciclo de clock.

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 4.27](#) 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, nosso pipeline poderia ter um hazard estrutural.

Hazards de dados

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 mesa de passar para a qual não exista 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 secarem deverão esperar.

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.

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 lavam roupas). Por exemplo, suponha que tenhamos uma instrução add seguida imediatamente por uma instrução subtract que usa a soma (\$s0):

```
add    $s0, $t0, $t1  
sub    $t2, $s0, $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 desperdiçar três ciclos de clock no pipeline.

Embora pudéssemos contar com compiladores para remover todos esses hazards, os resultados não seriam satisfatórios. Essas dependências acontecem com muita frequência, 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 sequência 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

Exemplo

Para as duas instruções anteriores, mostre quais estágios do pipeline estariam conectados pelo forwarding. Use o desenho da Figura 4.28 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 4.25.

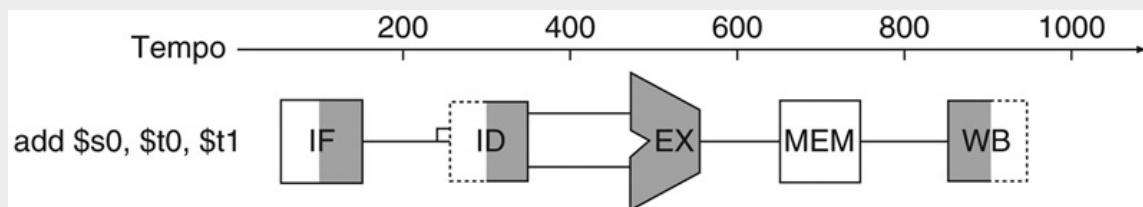


FIGURA 4.28 Representação gráfica do pipeline de instrução, semelhante em essência ao pipeline da lavanderia, na Figura 4.25.

Aqui, usamos símbolos representando os recursos físicos com as abreviações para estágios de pipeline usados no decorrer do capítulo. Os símbolos para os cinco estágios: *IF* para o estágio de busca de instrução, com a caixa representando a memória de instrução; *ID* para o estágio de leitura de decodificação de instrução/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

write-back, com o desenho mostrando o banco de registradores sendo escrito. O sombreamento indica que o elemento é usado pela instrução. Logo, MEM tem um fundo branco porque add não acessa a memória de dados. O sombreamento na metade direita do banco de registradores ou memória significa que o elemento é lido nesse estágio, e o sombreamento da metade esquerda significa que ele é escrito nesse estágio. Logo, a metade direita do ID é sombreada no segundo estágio porque o banco de registradores é lido, e a metade esquerda do WB é sombreada no quinto estágio, pois o banco de registradores é escrito.

Resposta

A Figura 4.29 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.

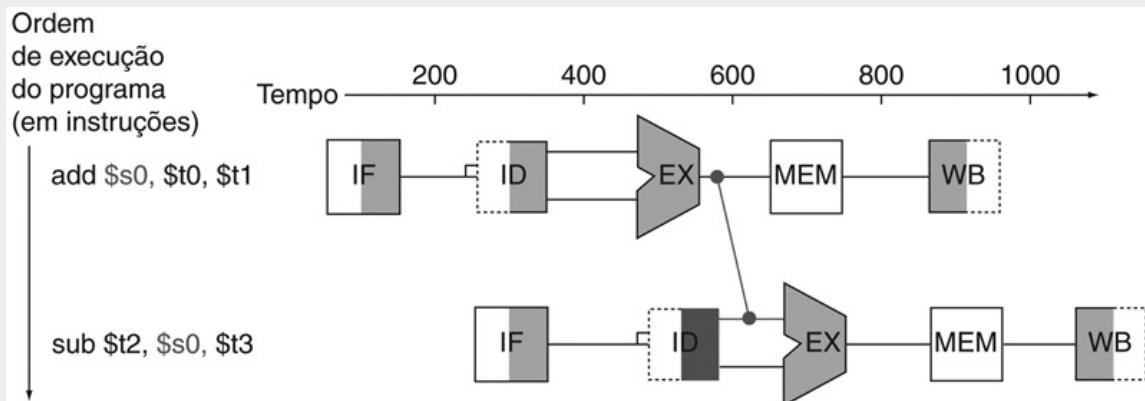


FIGURA 4.29 Representação gráfica do forwarding.

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.

O forwarding funciona muito bem e é descrito com detalhes na [Seção 4.7](#). 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 4.29](#), 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 de 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 4.30](#). 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 4.7](#) mostra como podemos tratar de casos assim, usando a detecção de hardware e stalls ou software que reordena o código para evitar stalls de pipeline no uso de load, como este exemplo ilustra.

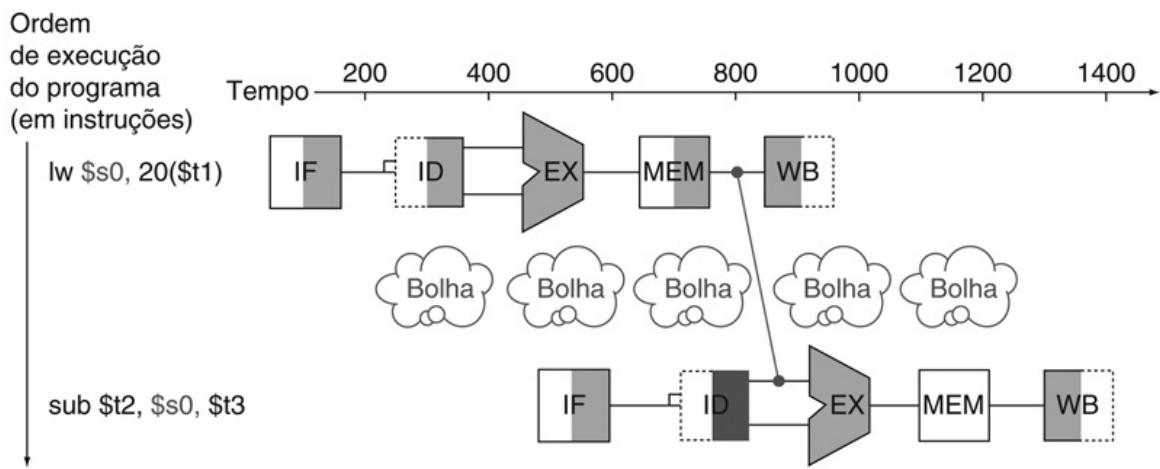


FIGURA 4.30 Precisamos de um stall até mesmo com forwarding, quando uma instrução do formato R após um load tenta usar os dados.

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 4.7](#) mostra os detalhes do que realmente acontece no caso de um hazard.

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 necessários para outra instrução.

pipeline stall

Também chamado **bolha**. Um stall iniciado a fim de resolver 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.

Resposta

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      $t2, 4($t0)
lw      $t4, 8($t0)
add   $t3, $t1,$t2
sw      $t3, 12($t0)
add   $t5, $t1,$t4
sw      $t5, 16($t0)
```

Em um processador com pipeline com forwarding, a sequência reordenada será completada em dois ciclos a menos do que a versão original.

O forwarding leva a outro detalhe da arquitetura MIPS, além dos quatro mencionados anteriormente. Cada instrução MIPS escreve no máximo um resultado e faz isso no último estágio do pipeline. O forwarding é mais difícil se houver vários resultados para encaminhar por instrução, ou se for preciso, escrever um resultado mais cedo na execução da instrução.

Detalhamento

o nome “forwarding” vem da ideia de que o resultado é passado adiante (em inglês *forwarded*) a partir de uma instrução anterior para uma instrução posterior. “Bypassing” vem de contornar (do inglês *bypass*) o resultado pelo banco de registradores até chegar 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.

hazard de controle

Também chamado hazard de desvio Quando a instrução apropriada não pode ser executada no devido ciclo de clock de pipeline porque a instrução buscada não é aquela necessária; ou seja, o fluxo de endereços de instrução não é o que o pipeline esperava.

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 e 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.

Stall: Basta operar sequencialmente 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 *acabou de receber* 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 em qual endereço apanhar a próxima instrução.

Vamos supor que colocamos hardware extra suficiente para que possamos testar registradores, calcular o endereço de desvio e atualizar o PC durante o segundo estágio do pipeline ([Seção 4.8](#)). Até mesmo com esse hardware extra, o pipeline envolvendo desvios condicionais se pareceria com a [Figura 4.31](#). A instrução `lw`, executada se o desvio não for tomado, fica em stall durante um ciclo de clock extra de 200 ps antes de iniciar

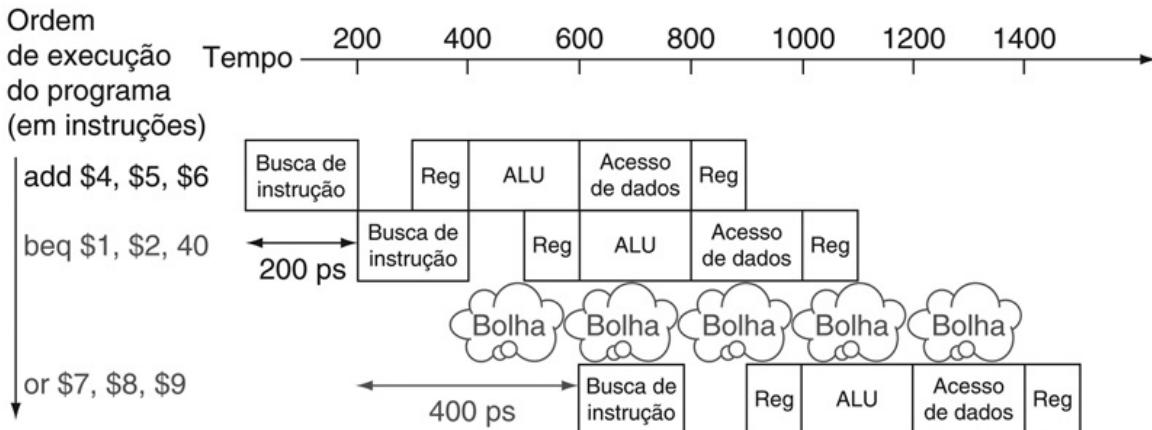


FIGURA 4.31 Pipeline mostrando o stall em cada desvio condicional como solução para controlar os hazards.

Este exemplo considera que o desvio condicional é tomado e a instrução no destino do desvio é a instrução OR. 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 4.8](#). 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 3.27 no Capítulo 3 mostra que os desvios são 17% das instruções executadas no SPECint2006. Como as outras instruções possuem um CPI de 1 e os desvios tomaram um ciclo de clock extra para o stall, então veríamos um CPI de 1,17 e, portanto, um stall de 1,17 em relação ao caso ideal.

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, usando uma de nossas grandes ideias do [Capítulo 1](#):

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.

Os computadores realmente utilizam a **predição** 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 4.32](#) mostra um exemplo assim.

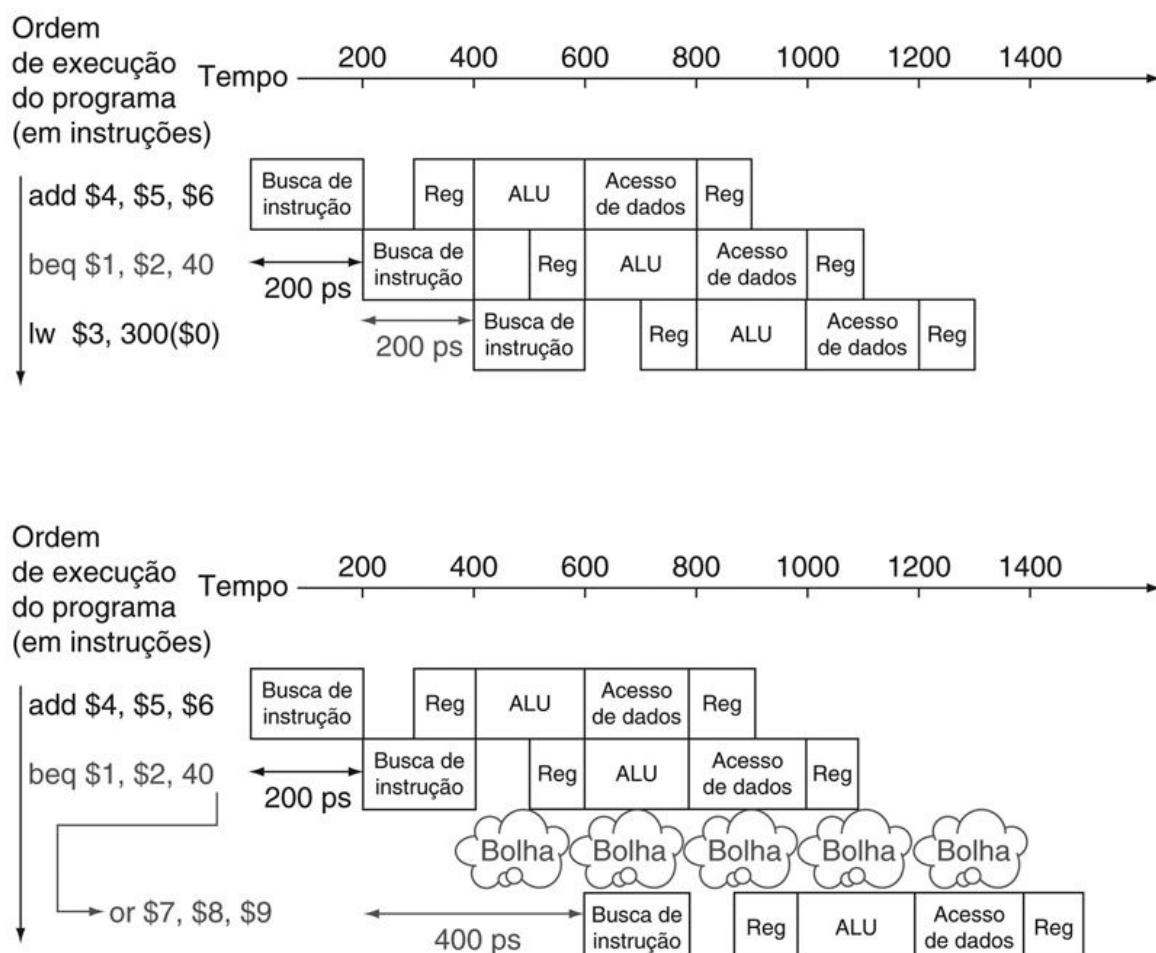
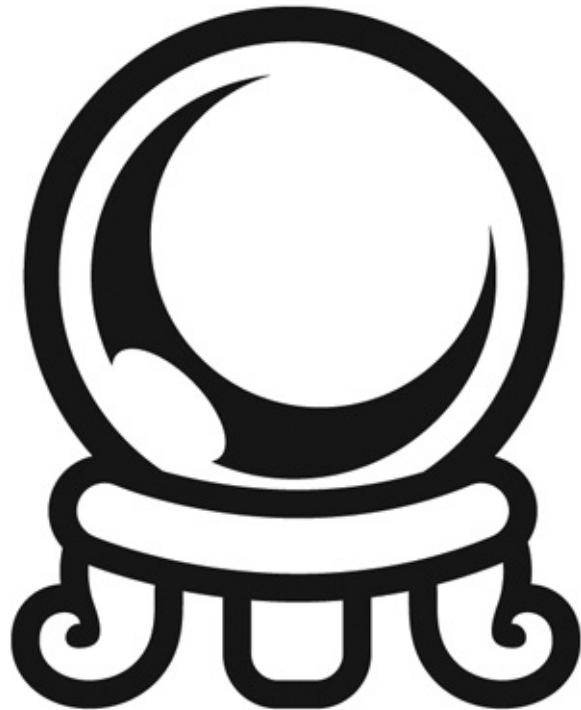


FIGURA 4.32 Prevendo que os desvios não serão tomados como solução para o hazard de controle.

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 4.31](#), 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 4.8](#) esclarecerá os detalhes.

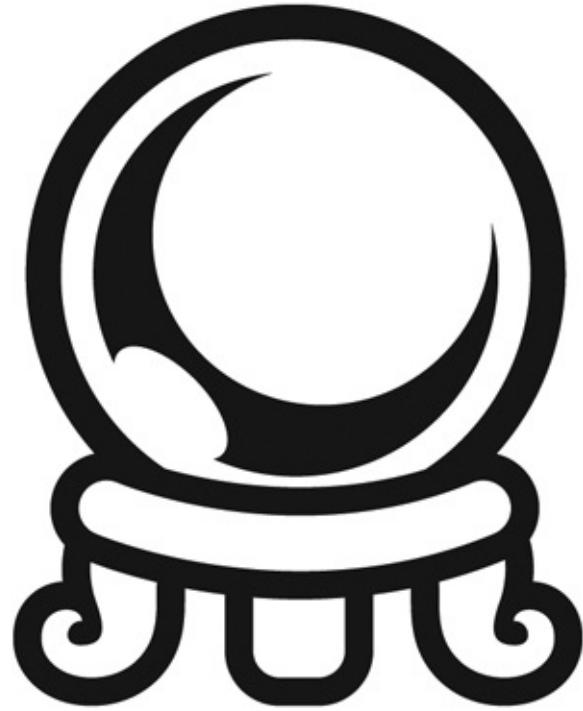
Uma versão mais sofisticada de **previsão de desvio** teria alguns desvios previstos como tomados e alguns como não tomados. Em nossa analogia, os uniformes escuros ou de casa poderiam usar uma fórmula, enquanto os uniformes claros ou de sair poderiam usar outra. No caso da programação, no final dos loops existem desvios que voltam para o início do loop. Como provavelmente serão tomados e desviam para trás, sempre poderíamos prever como tomados os desvios para um endereço anterior.



P R E D I Ç Ã O

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.



P R E D I Ç Ã O

Essas técnicas rígidas para o desvio contam com o comportamento estereotipado e não levam em consideração a 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. Seguindo 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 comum para a previsão 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 se tornado extensos,

resultando em previsores de desvio dinâmicos que podem prever os desvios corretamente, com uma precisão superior a 90% ([Seção 4.8](#)). Quando a escolha estiver errada, o controle do pipeline terá de garantir que as instruções após o desvio errado não tenham efeito, reiniciando 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 4.8](#).

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 sequencial, 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 4.31 não o afeta, e pode ser movida para depois dele, a fim de esconder totalmente seu atraso. Como os delayed branches são úteis quando os desvios são curtos, nenhum processador usa um delayed branch de mais de um ciclo. Para atrasos em desvios maiores, a previsão de desvio baseada em hardware normalmente é usada.

Resumo da visão geral de pipelining

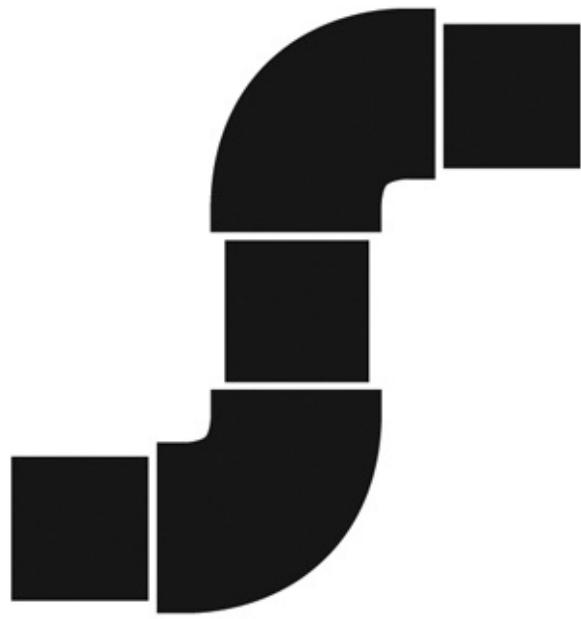
Pipelining é uma técnica que explora o **paralelismo** entre as instruções em um

fluxo de instruções sequenciais. Ela tem a vantagem substancial de que, diferente de programar um multiprocessador, ela é fundamentalmente invisível ao programador.



P A R A L E L I S M O

Nas próximas seções deste capítulo, abordamos o conceito de pipelining usando o subconjunto de instruções MIPS da implementação de ciclo único na [Seção 4.4](#) e mostramos uma versão simplificada de seu pipeline. Depois, examinamos os problemas que a técnica de **pipelining** gera e o desempenho alcançável em situações típicas.



PIPELINING

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 4.10](#). A [Seção 4.10](#) apresenta conceitos avançados de pipelining, como o escalonamento superescalar e dinâmico, e a [Seção 4.11](#) examina os pipelines de microprocessadores recentes.

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 4.6](#). Depois, você poderá usar esse conhecimento para explorar a implementação do forwarding e stalls na [Seção 4.7](#). Você poderá, então, ler a [Seção 4.8](#) e aprender mais sobre soluções para hazards de desvio, e depois ver como as exceções são tratadas, na [Seção 4.9](#).

Verifique você mesmo

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:

Sequência 1	Sequência 2	Sequência 3
1 \leftarrow $\$ + 0$ $0 / (\$ + 0)$		

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
--------------------	--	---

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 4.10, 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 frequências 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 frequência de desvios e os padrões de acesso mais regulares permitem que o compilador tente escalarizar 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 4.10, existem técnicas de compilação e de hardware mais ambiciosas que reduzem as dependências de dados para o escalonamento.



PIPELINING

Colocando em perspectiva

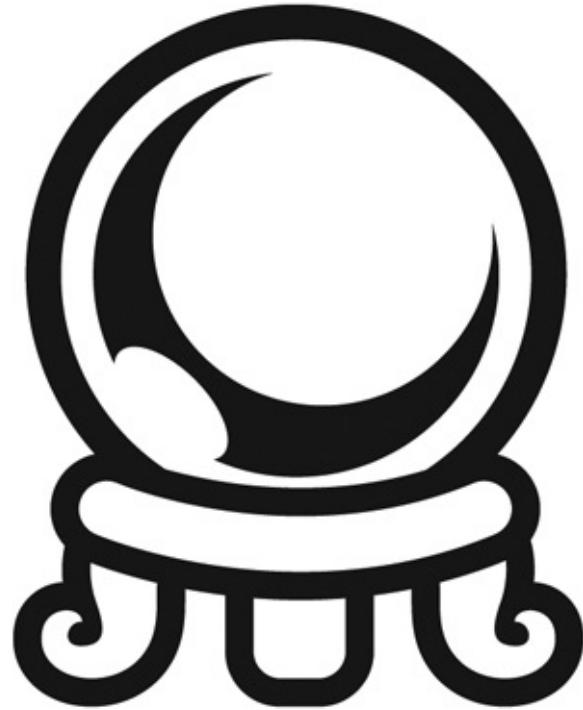
A técnica de pipelining aumenta o número de instruções em execução simultânea e a velocidade em que as instruções são iniciadas e concluídas. A técnica de pipelining não reduz o tempo gasto para completar uma instrução individual, também chamado de **latência**. Por exemplo, o pipeline de cinco estágios ainda usa cinco ciclos de clock para completar a instrução. Nos termos usados no Capítulo 1, a técnica de pipelining melhora a *vazão* de instruções, e não o *tempo de execução* ou *latência* das instruções individualmente.

latência (pipeline)

O número de estágios em um pipeline ou o número de estágios entre duas instruções durante a execução.

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.



P R E D I Ç Ã O

4.6. Caminho de dados e controle usando pipeline

Há menos coisa nisso do que os olhos podem ver.

Tallulah Bankhead, comentário para Alexander Woollcott, 1922

A [Figura 4.33](#) mostra o caminho de dados de ciclo único da [Seção 4.4](#) com os estágios de pipeline identificados. 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.

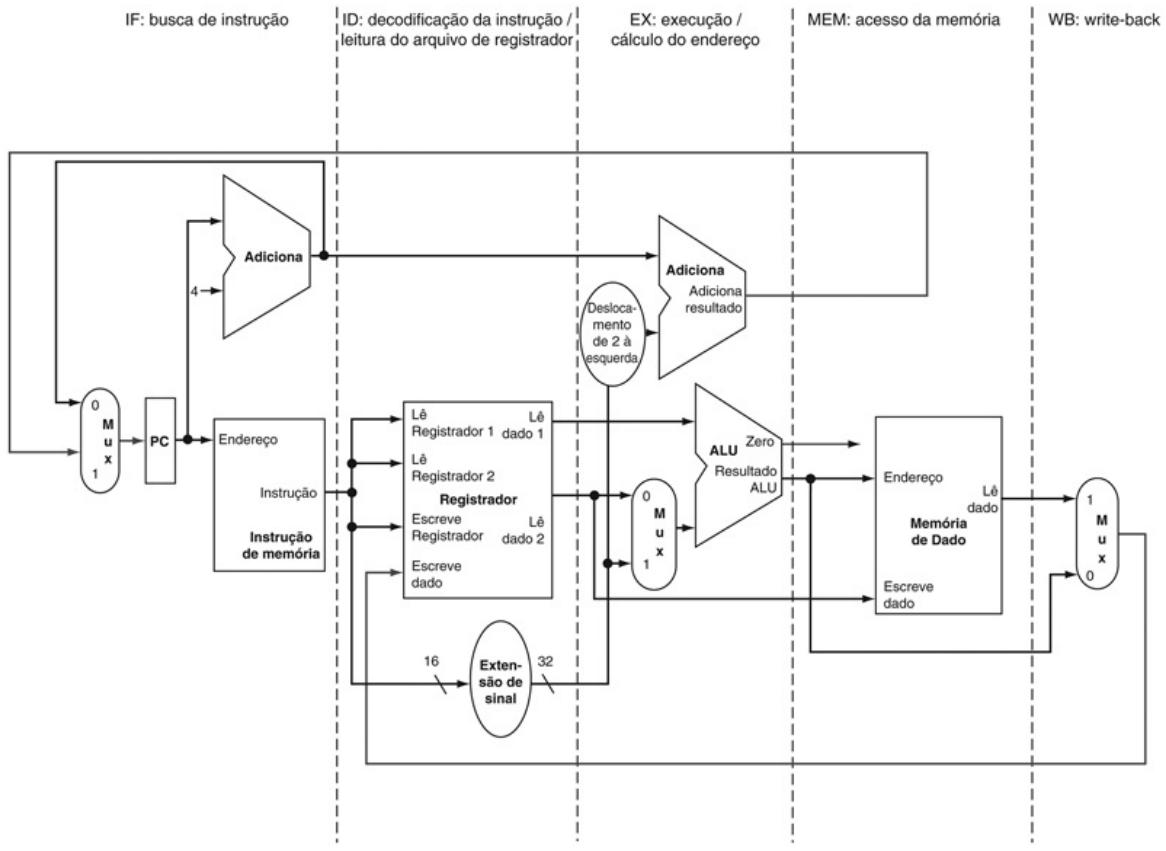


FIGURA 4.33 O caminho de dados da [Seção 4.4](#) (semelhante à [Figura 4.17](#)).

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 destaque, que envia o resultado da ALU ou os dados da memória para a esquerda, a fim de serem escritos no banco de registradores. (Normalmente, usamos linhas coloridas para controle, mas são linhas de dados.)

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.

Na [Figura 4.33](#), 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 4.34](#) mostra a execução das instruções na [Figura 4.27](#), exibindo seus caminhos de dados privados em uma linha de tempo comum. Usamos uma versão estilizada do caminho de dados na [Figura 4.33](#) para mostrar os relacionamentos na [Figura 4.34](#).

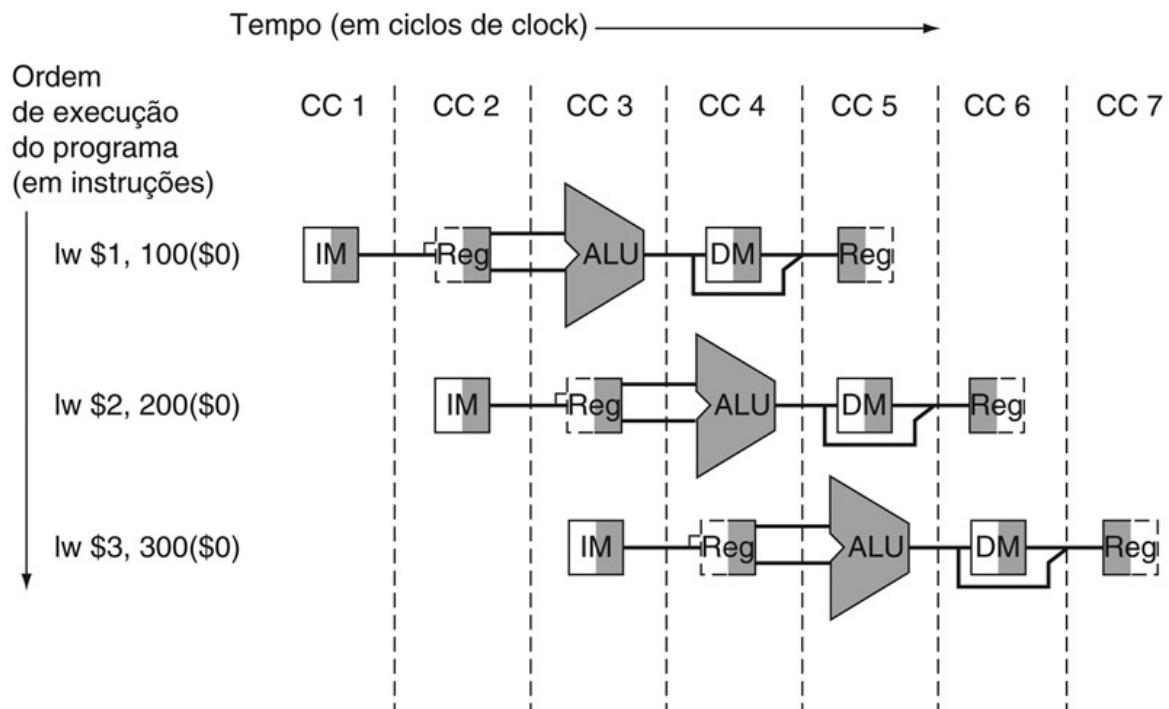


FIGURA 4.34 Instruções executadas usando o caminho de dados de ciclo único na Figura 4.33, assumindo a execução com pipeline.

Semelhante às Figuras de 4.28 a 4.30, 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 4.33.

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, consideramos que o banco de registradores é escrito na primeira metade do ciclo de clock e é lido durante a segunda metade.

A Figura 4.34 parece sugerir que três instruções precisam de três caminhos de dados. Em vez disso, acrescentamos registradores para manter dados de modo que partes do caminho de dados pudessem ser compartilhadas durante a execução da instrução.

Por exemplo, como mostra a Figura 4.34, a memória de instruções é usada durante apenas um dos cinco estágios de uma instrução, permitindo que seja compartilhada por outras instruções durante os outros quatro estágios. A fim de 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 divisórias entre os estágios na Figura 4.33. Retornando à nossa analogia da lavanderia, poderíamos ter um cesto entre cada par de estágios contendo as roupas para a próxima etapa.

A Figura 4.35 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.

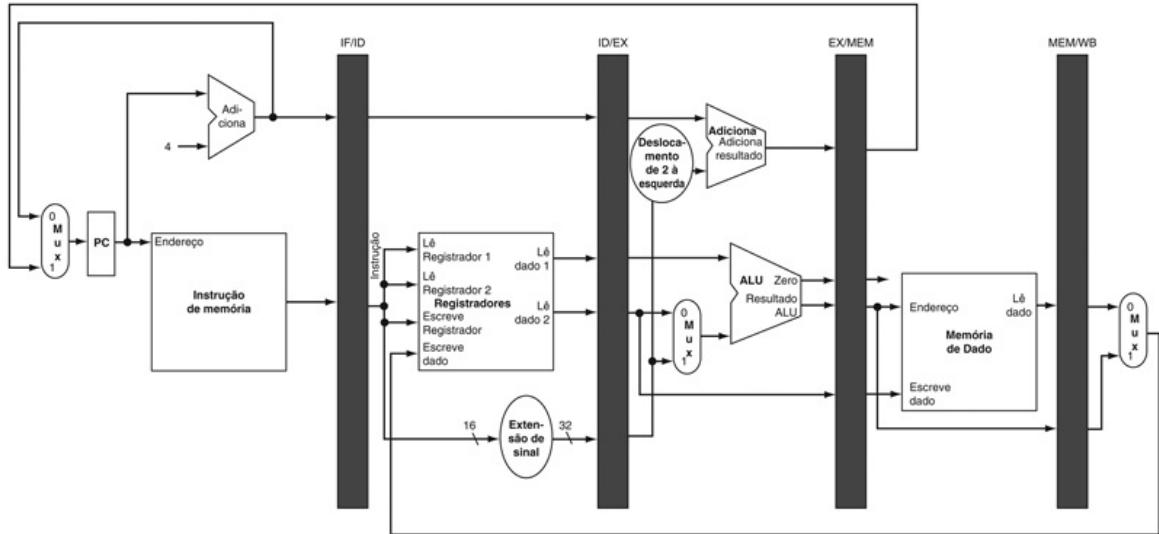


FIGURA 4.35 A versão com pipeline do caminho de dados na [Figura 4.33](#).

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.

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 que é atualizado. Por exemplo, uma instrução load colocará seu resultado em um 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 4.35](#), 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 muito mais tempo para você entender. Mas não tema; as sequências levam muito menos tempo do que parece, pois você pode compará-las e ver que mudanças ocorrem em cada ciclo do clock. A [Seção 4.7](#) descreve o que acontece quando existem hazards de dados entre instruções em um pipeline; ignore-as por enquanto.

As [Figuras 4.36 a 4.38](#), nossa primeira sequência, 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 4.28 a 4.30](#), destacamos a *metade direita* dos registradores ou memória quando estão sendo *lidos* e destacamos a *metade esquerda* quando estão sendo *escritos*.

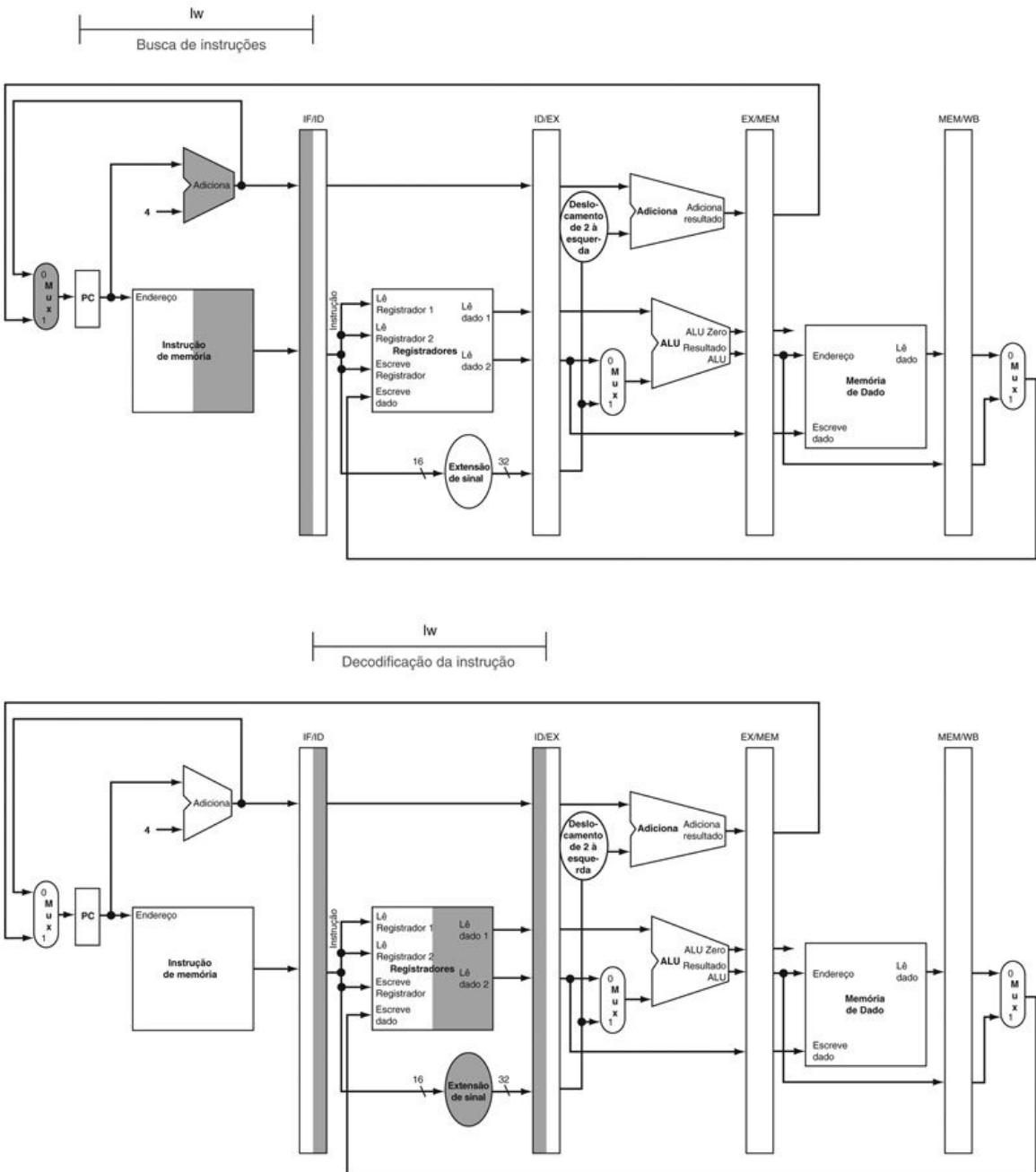


FIGURA 4.36 IF e ID: primeiro e segundo estágios do pipe de uma instrução, com as partes ativas do caminho de dados da Figura 4.35 em destaque.

A convenção de destaque é a mesma utilizada na [Figura 4.28](#). Como na [Seção 4.2](#), 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 qual 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.

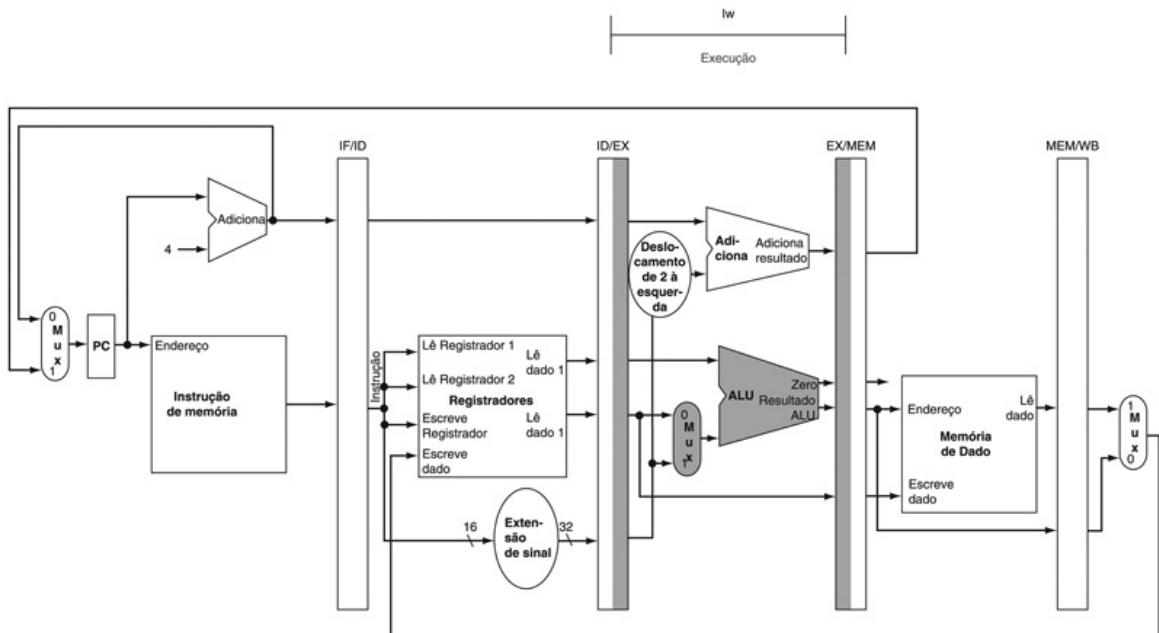


FIGURA 4.37 EX: o terceiro estágio do pipe de uma instrução load, destacando as partes do caminho de dados da [Figura 4.35](#) usadas neste estágio do pipe.

O registrador é acrescentado ao imediato com sinal estendido, e a soma é colocada no registrador de pipeline EX/MEM.

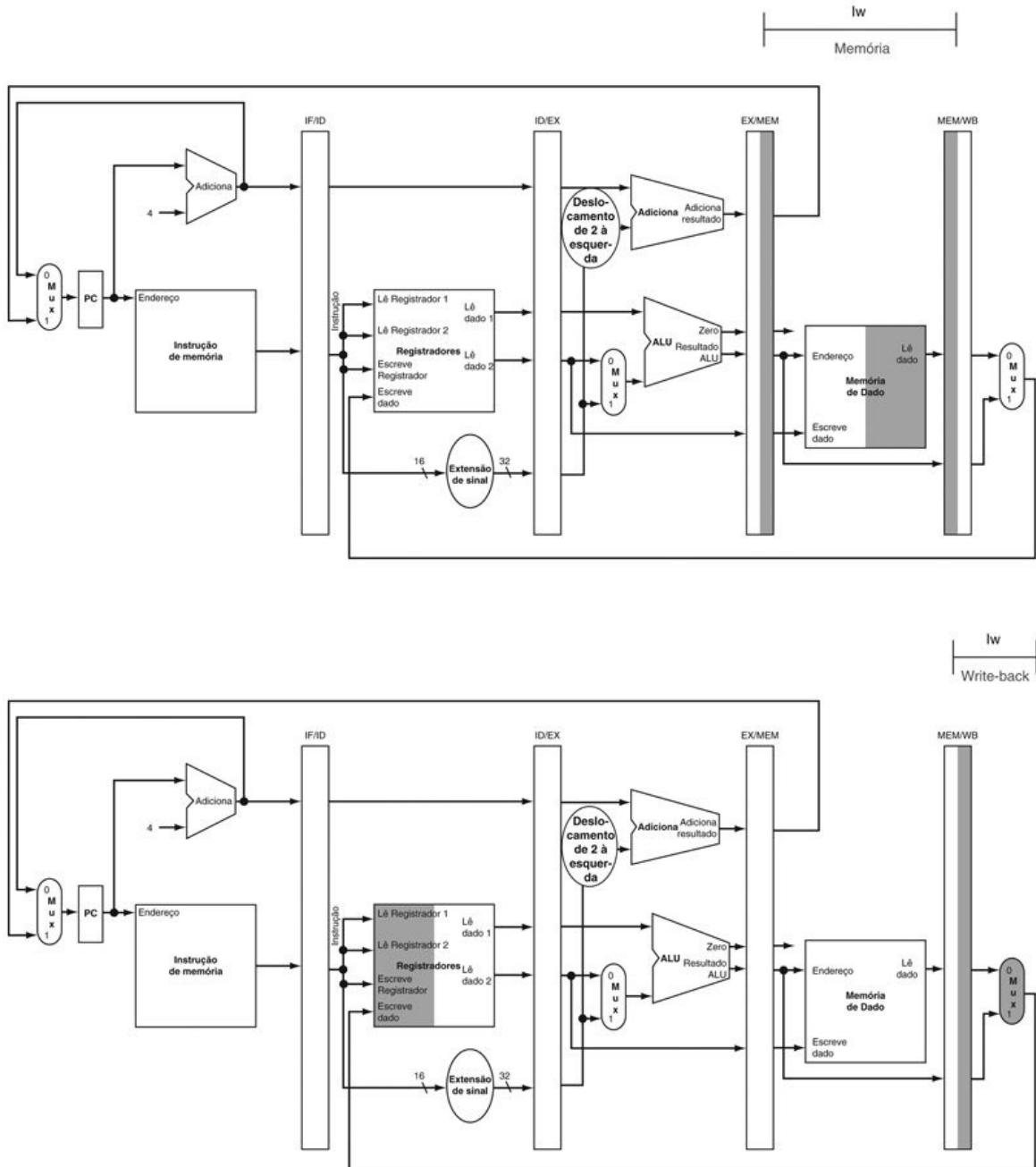


FIGURA 4.38 MEM e WB: o quarto e quinto estágios do pipe de uma instrução load, destacando as partes do caminho de dados da Figura 4.35 usadas nesses estágios do pipe.

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. Nota: existe um bug nesse projeto, que foi consertado na [Figura 4.41](#).

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:

1. *Busca de instruções*: A parte superior da [Figura 4.36](#) mostra a instrução sendo lida da memória usando o endereço no PC e depois colocada no registrador de pipeline IF/ID. 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 pelo pipeline informações potencialmente necessárias.
2. *Decodificação de instruções e leitura do banco de registradores*: A parte inferior da [Figura 4.36](#) 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 o que possa ser necessário por qualquer instrução, durante um ciclo de clock posterior.
3. *Execução ou cálculo de endereço*: A [Figura 4.37](#) 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 4.38](#) 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 4.38](#) 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 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 4.36](#) 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 4.36](#) 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 4.39](#) mostra a terceira etapa; o endereço efetivo é colocado no registrador de pipeline EX/MEM.

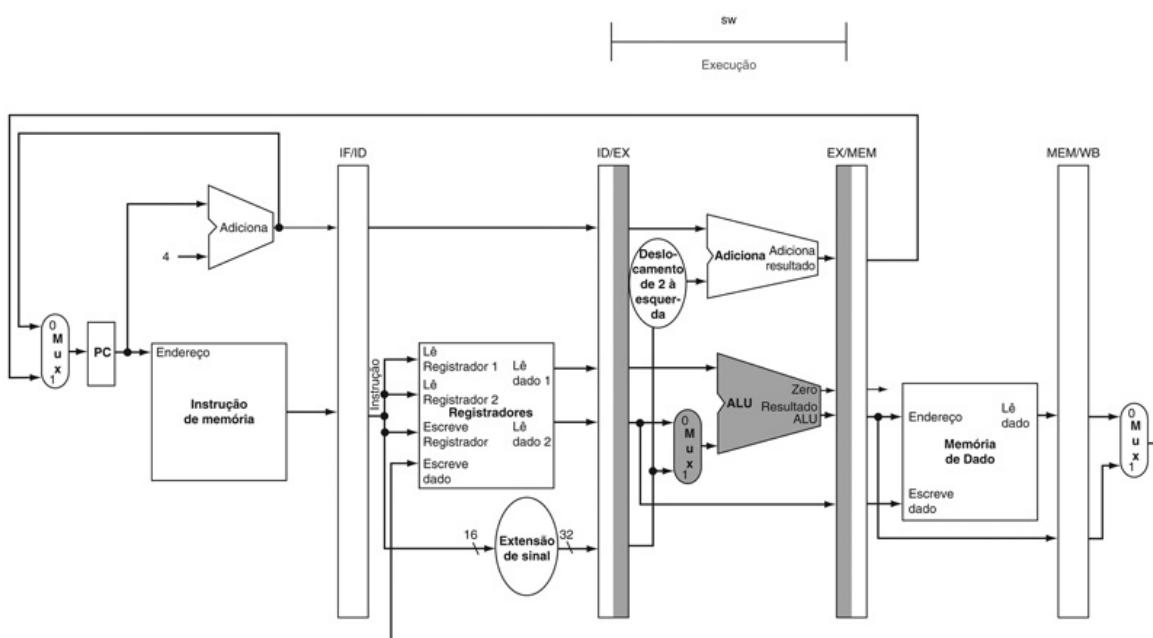


FIGURA 4.39 EX: o terceiro estágio do pipe de uma instrução store.

Ao contrário do terceiro estágio da instrução load na [Figura 4.37](#), 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.

4. Acesso à memória: A parte superior da [Figura 4.40](#) 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.

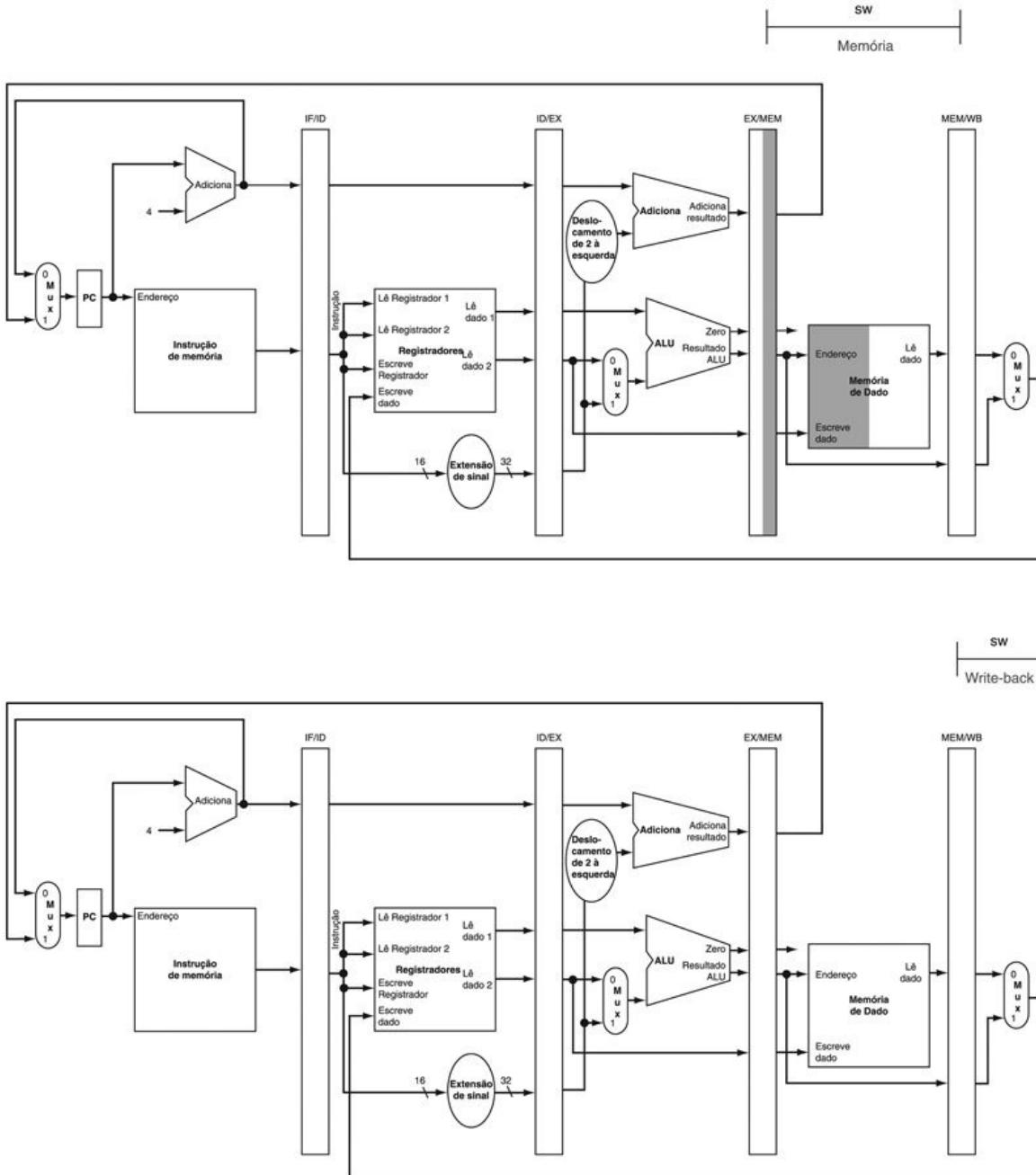


FIGURA 4.40 MEM e WB: o quarto e quinto estágios do pipe de uma instrução store.

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. Uma vez que os dados são escritos na memória, não há nada mais a fazer para a instrução store, de modo que nada acontece no estágio 5.

5. Escrita do resultado: A parte inferior da Figura 4.40 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 a 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.

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 Seção Hazards estruturais”, anteriormente neste capítulo). Logo, esses componentes e seu controle podem ser associados a um único estágio do pipeline.

Agora, podemos expor 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 aos registradores de pipeline EX/MEM para uso no estágio MEM, load precisa passar o número do *registraror* de ID/EX por EX/MEM ao registrador de pipeline MEM/WB, para 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 4.41](#) mostra a versão correta do caminho de dados, passando o número do registrador de escrita primeiro ao registrador ID/EX, depois ao registrador EX/MEM e finalmente ao registrador MEM/WB. O número do

registrador é usado durante o estágio WB de modo a especificar o registrador a ser escrito. A [Figura 4.42](#) é um desenho simples do caminho de dados corrigido, destacando o hardware utilizado em todos os cinco estágios da instrução load word nas [Figuras de 4.36 a 4.38](#). Veja na [Seção 4.8](#) uma explicação de como fazer a instrução branch funcionar como esperado.

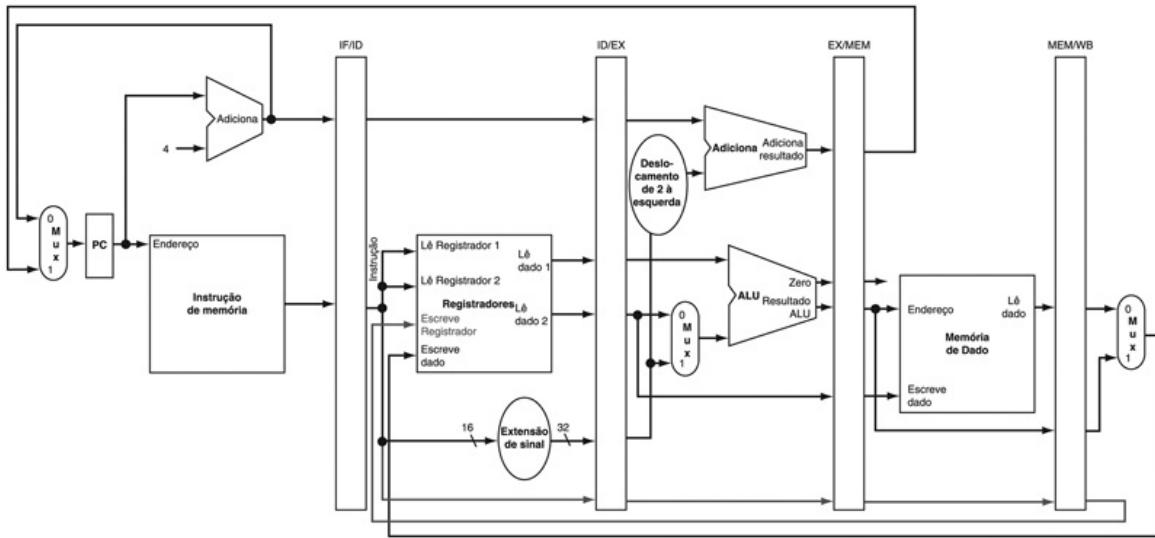


FIGURA 4.41 O caminho de dados em pipeline corrigido para lidar corretamente com a instrução load.

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.

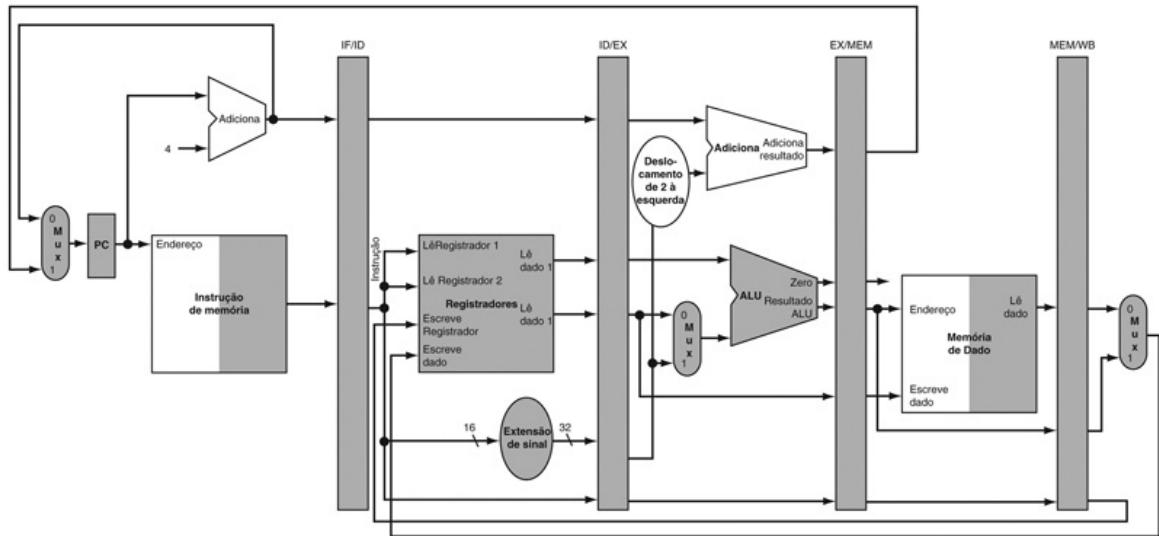


FIGURA 4.42 A parte do caminho de dados na [Figura 4.41](#) usada em todos os cinco estágios de uma instrução load.

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 4.34](#), e *diagramas de pipeline com único ciclo de clock*, como as [Figuras de 4.36 a 4.40](#). Os diagramas com múltiplos ciclos de clock são mais simples, mas não contêm todos os detalhes. Por exemplo, considere esta 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 4.43 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 4.25. 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 funciona bem. A Figura 4.44 mostra a versão mais tradicional do diagrama de pipeline com múltiplos ciclos de clock. Observe que a Figura 4.43 mostra os recursos físicos utilizados em cada estágio, enquanto a Figura 4.44 usa o *nome* de cada estágio.

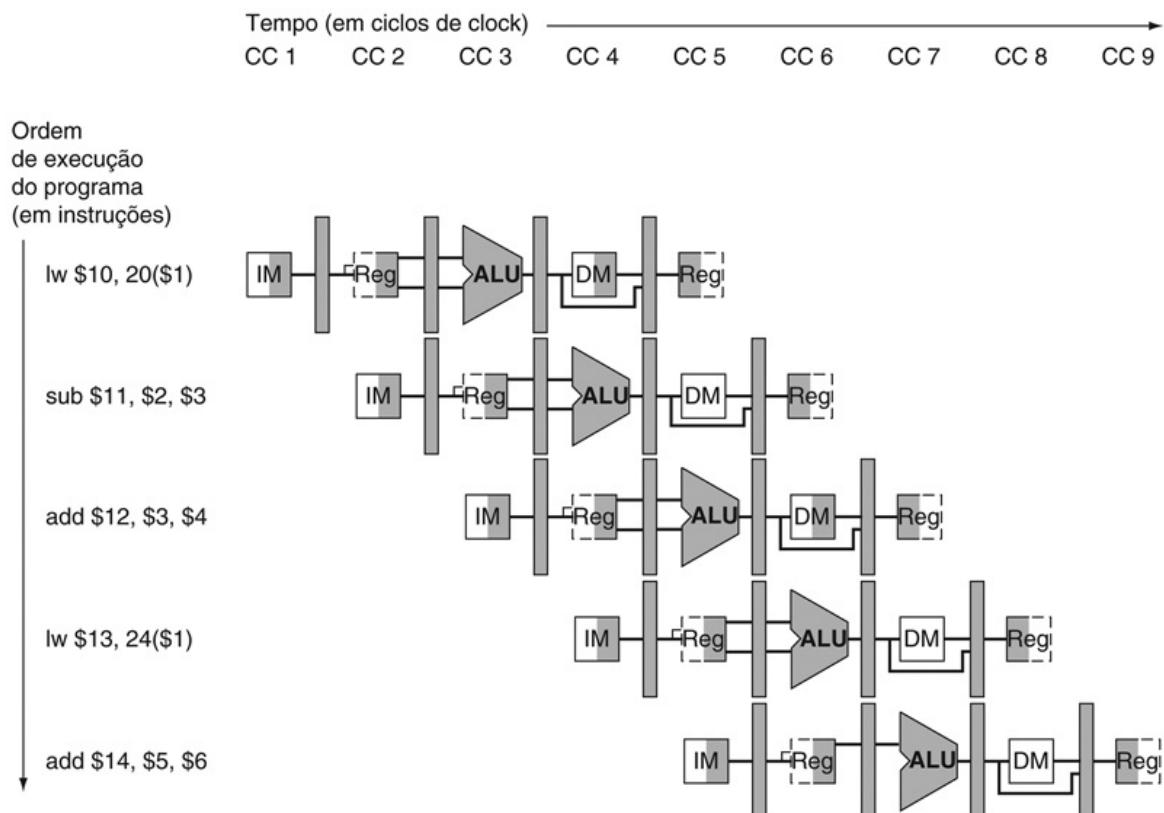


FIGURA 4.43 Diagrama de pipeline com múltiplos ciclos de clock das cinco instruções.

Este 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 4.28, aqui, mostramos os registradores de pipeline entre cada estágio. A Figura 4.44 mostra a maneira tradicional de

desenhar esse diagrama.

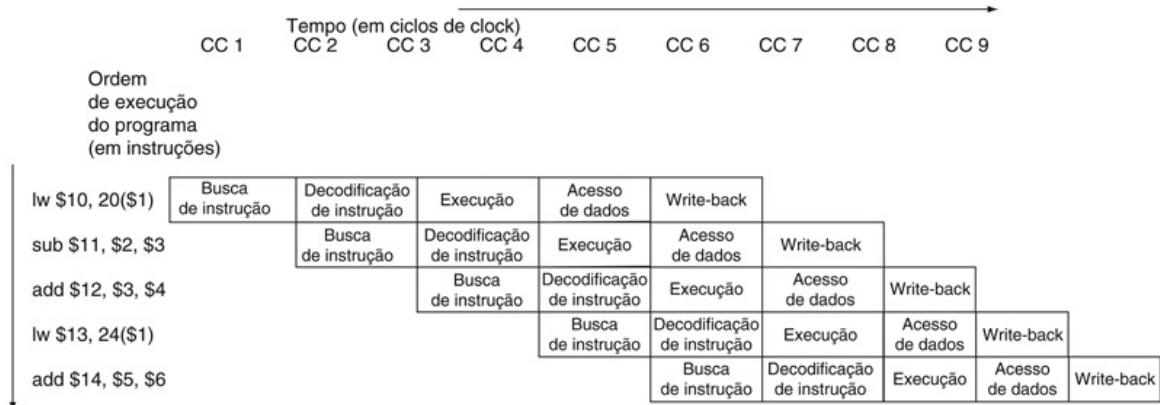


FIGURA 4.44 Diagrama de pipeline tradicional com múltiplos ciclos de clock, com as cinco instruções da [Figura 4.43](#).

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 sequência de ciclos de clock. Usamos diagramas de ciclo múltiplo de clock, a fim de oferecer sinopses de situações de pipelining. 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 4.45](#) mostra o diagrama com ciclo único de clock correspondente ao ciclo de clock 5 das [Figuras 4.43 e 4.44](#). Obviamente, os diagramas com ciclo único de clock possuem mais detalhes e ocupam muito mais espaço para mostrar o mesmo número de ciclos de clock. Os exercícios pedem que você crie esses diagramas para outras sequências de código.

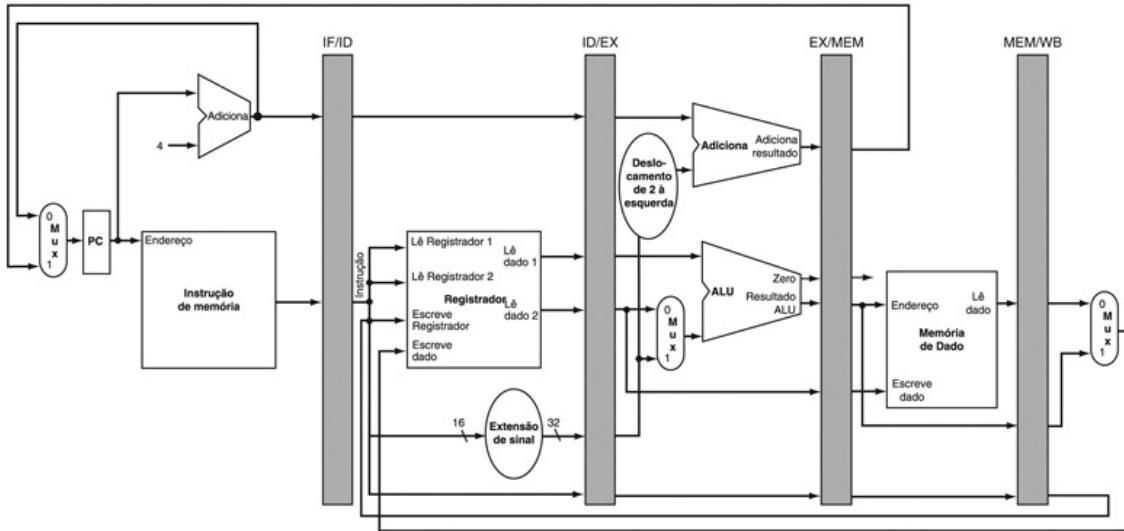
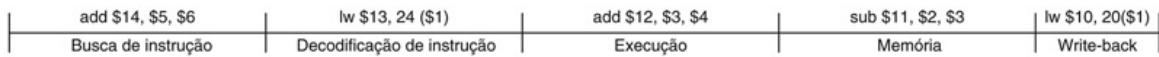


FIGURA 4.45 O diagrama com ciclo único de clock correspondente ao ciclo de clock 5 do pipeline das [Figuras 4.43 e 4.44](#).

Como você pode ver, uma figura com ciclo único de clock é uma fatia vertical de um diagrama com múltiplos ciclos de clock.

Verifique você mesmo

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 quatro afirmações a seguir. Quais delas estão corretas?

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. 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.
4. 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.

Controle em pipeline

No computador 6600, talvez ainda mais do que em qualquer computador anterior, o sistema de controle faz a diferença.

James Thornton, Design of a Computer: The Control Data 6600, 1970

Assim como acrescentamos controle ao caminho de dados simples na [Seção 4.3](#), 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.

O primeiro passo é rotular as linhas de controle no caminho de dados existente. A [Figura 4.46](#) mostra essas linhas. Pegamos o máximo possível emprestado do controle para o caminho de dados simples da [Figura 4.17](#). 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 nas [Figuras 4.12, 4.16 e 4.18](#). Reproduzimos as principais informações nas [Figuras 4.47 a 4.49](#) em uma única página, de modo a facilitar o acompanhamento do restante do texto.

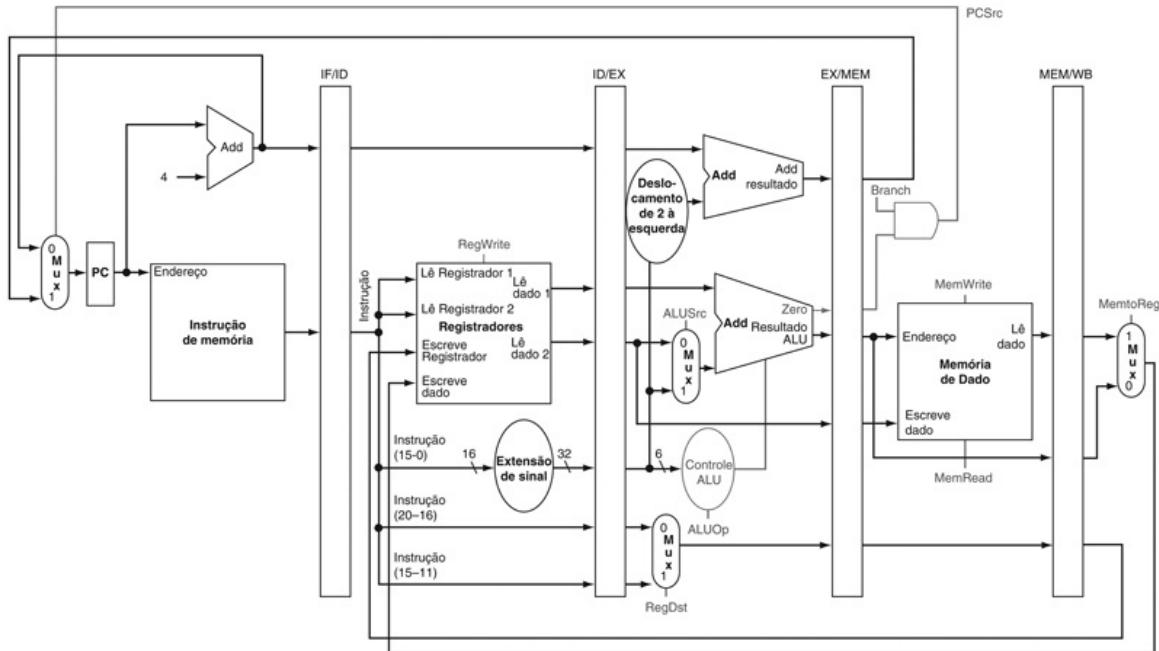


FIGURA 4.46 O caminho de dados em pipeline da Figura 4.41 com sinais de controle identificados.

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, da Seção 4.4. 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 6 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
Desvio igual	01	Desvio igual	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	definir com menos de	101010	definir com menos de	0111

FIGURA 4.47 Uma cópia da **Figura 4.12**.

Essa figura mostra como os bits do controle da ALU são definidos dependendo dos bits de controle ALUOp e dos

diferentes códigos de função para instruções tipo R.

Nome do sinal	Efeito quando inativo (0)	Efeito quando ativo (1)
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 4.48 Uma cópia da Figura 4.16.

A função de cada um dos sete sinais de controle é definida. As linhas de controle da ALU (ALUOp) são definidas na segunda coluna da [Figura 4.47](#). 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 PCScr é controlado por uma porta lógica AND na [Figura 4.46](#). Se o sinal Branch e o sinal Zero da ALU estiverem ativos, então PCScr é 1; caso contrário, ele é 0. O controle define o sinal Branch somente durante uma instrução beq; caso contrário, o PCScr 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	Desvio	Le Mem	Escreve Mem	Escreve Reg	Mem para Reg
Formato R	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

FIGURA 4.49 Os valores das linhas de controle são iguais aos da [Figura 4.18](#), mas foram reorganizados em três grupos, correspondentes aos três últimos estágios do pipeline.

Assim como ocorreu com a implementação com ciclo único, 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.

A fim de 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, o mesmo 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, ALUOp e ALUScr ([Figuras 4.47 e 4.48](#)). Os sinais selecionam o registrador Resultado, 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, ReadMem e WriteMem. Esses sinais são definidos pelas instruções branch equal, load e store, respectivamente. Lembre-se de que o PCScr na [Figura 4.48](#) seleciona o próximo endereço sequencial, a menos que o controle ative Branch e o resultado da ALU seja zero.
5. *Escrita do resultado*: As duas linhas de controle são MemtoReg, que decide entre enviar o resultado da ALU ou o valor da memória para o banco de registradores, e WriteRegWriteReg, que escreve o valor escolhido.

Como a utilização de um pipeline no caminho de dados deixa inalterado o significado das linhas de controle, podemos usar os mesmos valores de controle de antes. A [Figura 4.49](#) tem os mesmos valores da [Seção 4.4](#), mas agora as nove linhas de controle estão agrupadas por estágio do pipeline.

A implementação do controle significa definir as nove linhas de controle desses valores em cada estágio, para cada instrução. A maneira mais simples de fazer isso é estender os registradores do pipeline de modo a 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 4.50](#)

mostra que esses sinais de controle são usados no respectivo estágio do pipeline, à medida que a instrução se move por ele, assim como o número do registrador destino para loads desce pelo pipeline da [Figura 4.41](#). A [Figura 4.51](#) mostra o caminho de dados completo, com os registradores de pipeline estendidos e com as linhas de controle conectadas ao estágio apropriado

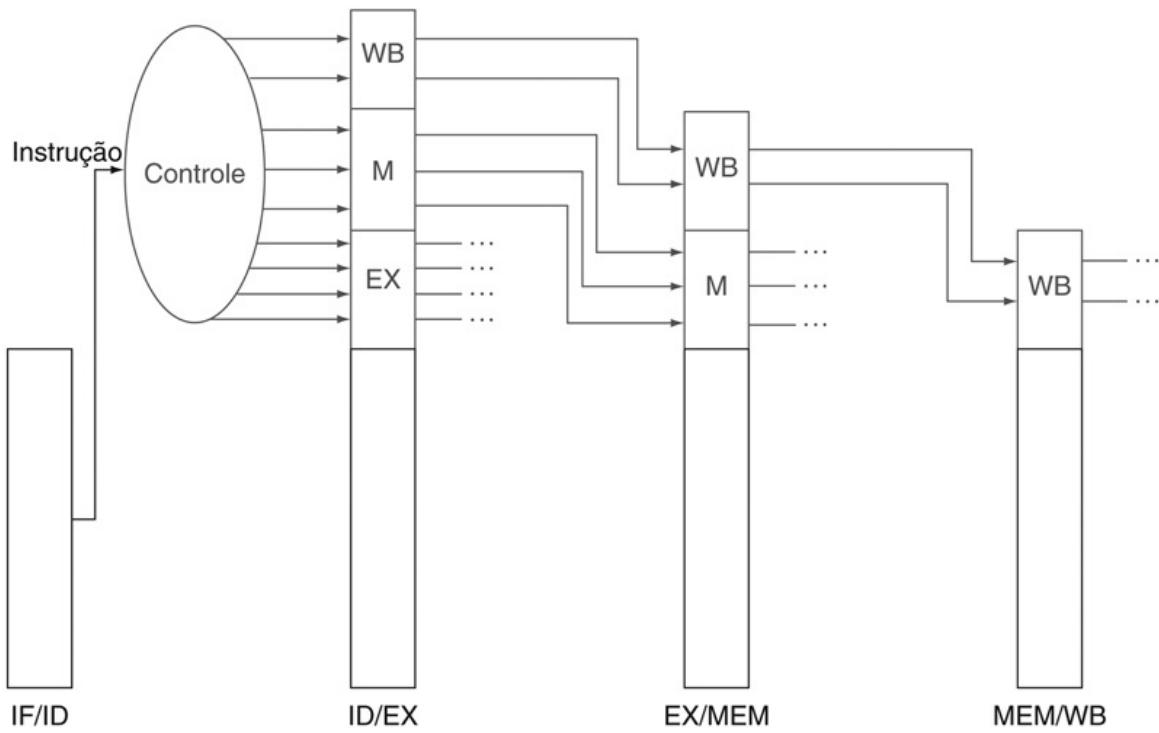


FIGURA 4.50 As linhas de controle para os três estágios finais.

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, a fim de manter as linhas de controle; três são usadas durante o estágio MEM, e as duas últimas são passadas a MEM/WB, para uso no estágio WB.

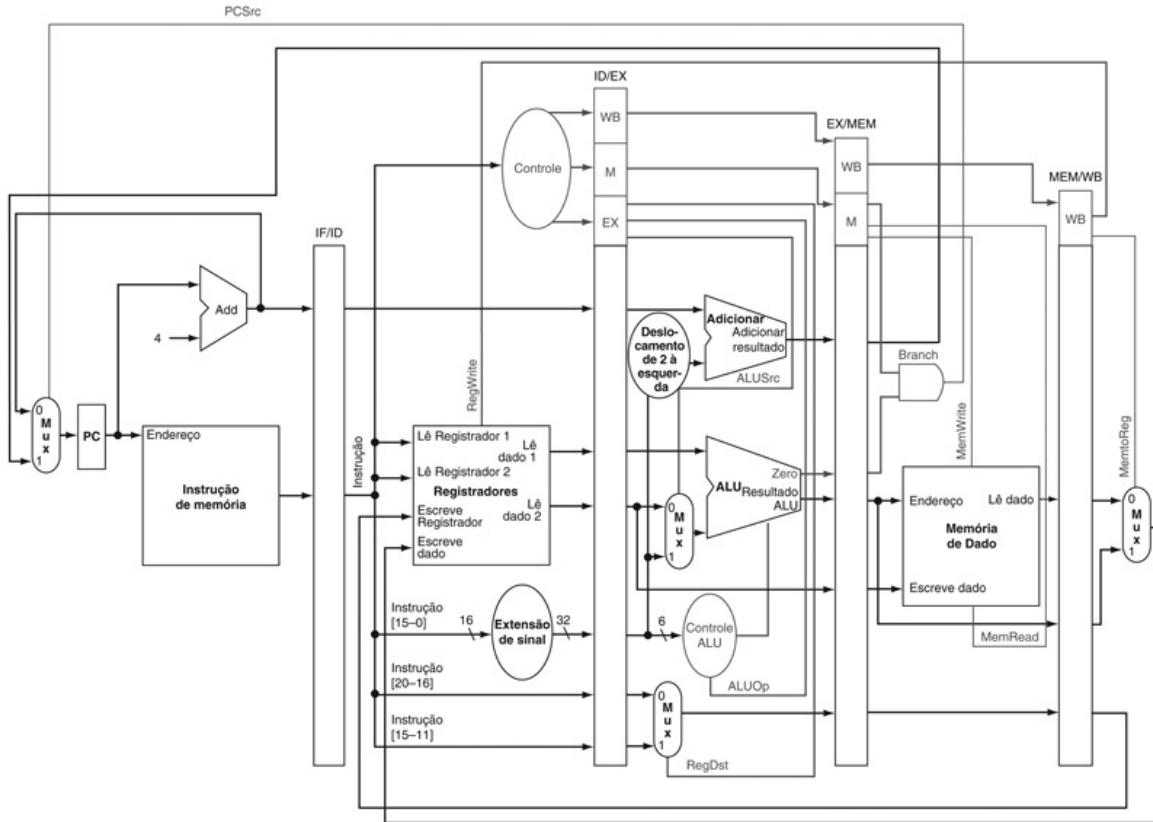


FIGURA 4.51 O caminho de dados em pipeline da Figura 4.46, com os sinais de controle conectados às partes de controle dos registradores de pipeline.

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 ao próximo estágio do pipeline.

4.7. Hazards de dados: forwarding versus stalls

Como assim por que teve de ser criado? É um bypass. Você precisa criar bypasses.

Douglas Adams, The Hitchhiker's Guide to the Galaxy, 1979

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 4.43 a 4.45](#) eram independentes; nenhuma delas usava os resultados calculados por qualquer uma das outras. Mesmo assim, na [Seção 4.5](#), vimos que os hazards de dados são obstáculos para a execução em pipeline.

Vejamos uma sequência com muitas dependências, indicadas com realce:

```
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 e 2º operandos ($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 desejaría que -20 fosse usado nas instruções seguintes, que se referem ao registrador \$2.

Como essa sequência funcionaria com nosso pipeline? A [Figura 4.52](#) ilustra a execução dessas instruções usando uma representação de pipeline com múltiplos ciclos de clock. Para demonstrar a execução dessa sequência de instruções em nosso pipeline atual, o topo da [Figura 4.52](#) mostra o valor do registrador \$2, que muda durante o ciclo de clock 5, quando a instrução sub escreve seu resultado.

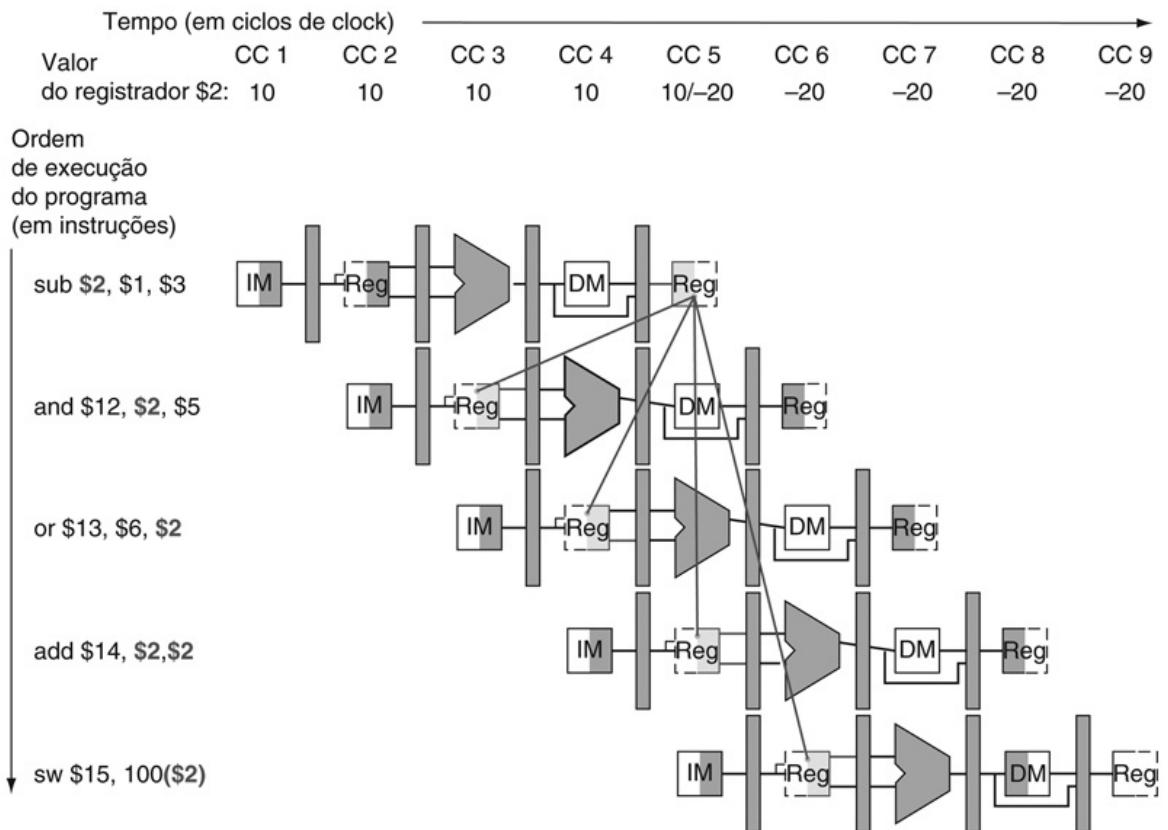


FIGURA 4.52 Dependências em pipeline em uma sequência de cinco instruções usando caminhos de dados simplificados para mostrar as dependências.

Todas as ações dependentes são mostradas em cinza, e “CC 1” no alto da figura significa o ciclo de clock 1. 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*.

O último 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 esta fornece o que foi escrito. Como acontece para muitas implementações dos bancos de registradores, não temos hazard de dados nessa situação.

A Figura 4.52 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 de 10! Usando esse estilo de desenho, esses problemas se tornam aparentes quando uma linha de dependência retorna no tempo.

Conforme dissemos na [Seção 4.5](#), o resultado desejado está disponível no final do estágio EX ou no ciclo de clock 3. Quando os dados são realmente necessários pelas instruções AND e OR? No início 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 sequência da [Seção 4.7](#) 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 sequência da Seção 4.7:

```
sub $2,    $1, $3  # Registrador $2 escrito por sub
and $12,   $2, $5  # 1º operando ($2) escrito por sub
or  $13,   $6, $2  # 2º operando ($2) escrito por sub
add $14,   $2, $2  # 1º e 2º operandos ($2) escrito por sub
sw   $15, 100($2) # Base ($2) escrito por sub
```

Resposta

Conforme já mencionamos, o sub-and é um hazard tipo 1a. Os outros hazards são

- sub-or é um hazard tipo 2b:
MEM/WB.RegistradorRd = 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 indevidamente. Uma solução é simplesmente verificar se o sinal WriteReg estará ativo: examinando o campo de controle WB do registrador de pipeline durante os estágios EX e MEM, é possível determinar se WriteReg está ativo. Lembre-se de que 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 resolvida — mas ainda precisamos fazer o forwarding dos dados corretos.

A Figura 4.53 mostra as dependências entre os registradores de pipeline e as entradas da ALU para a mesma sequência de código da Figura 4.52. 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.

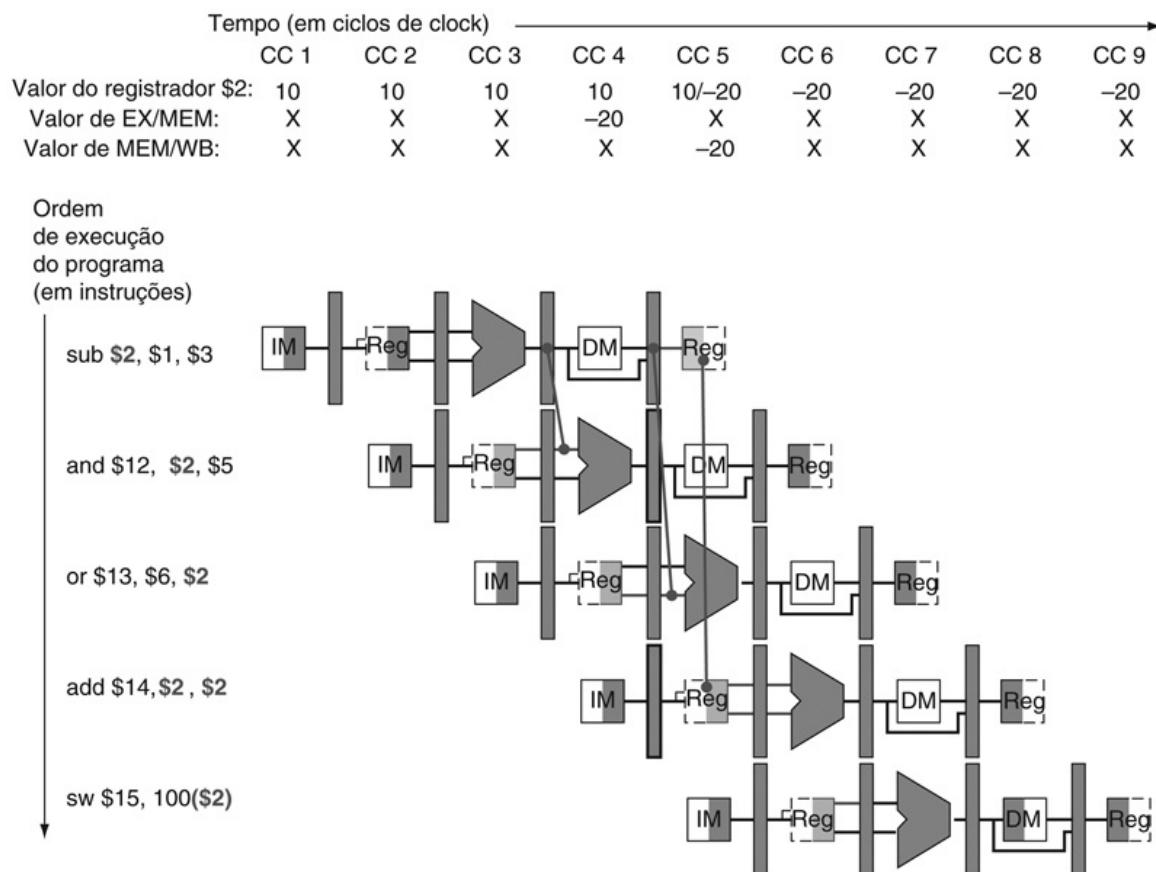


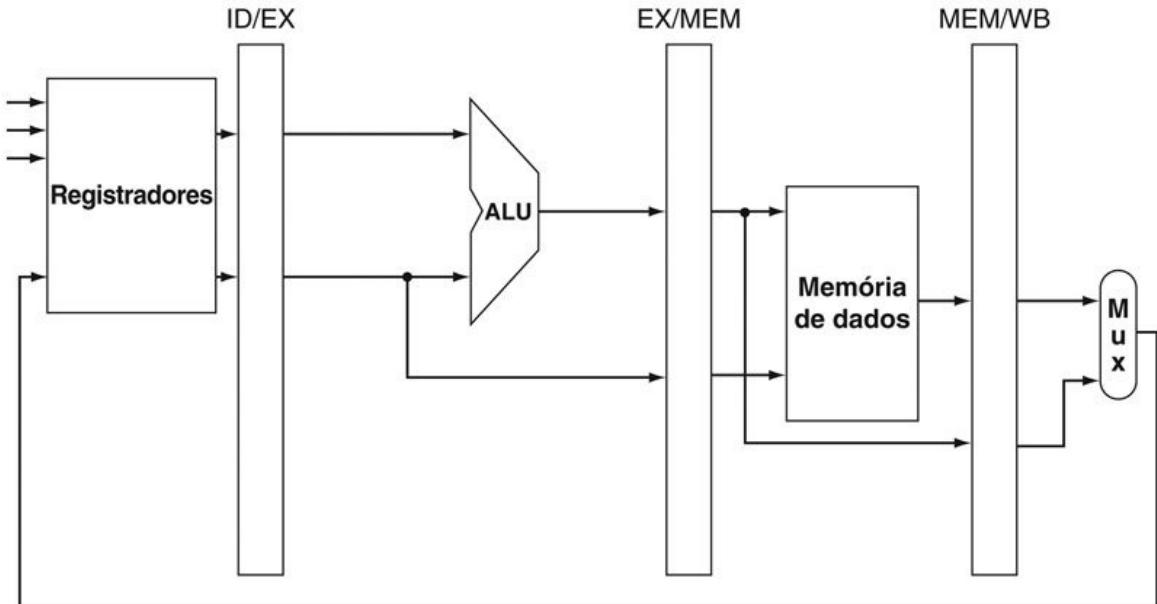
FIGURA 4.53 As dependências entre os registradores de pipeline se movem para a frente no tempo, de modo que é possível fornecer as entradas para a ALU necessárias para a instrução AND e para a instrução OR fazendo forwarding dos resultados encontrados nos registradores de pipeline.

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

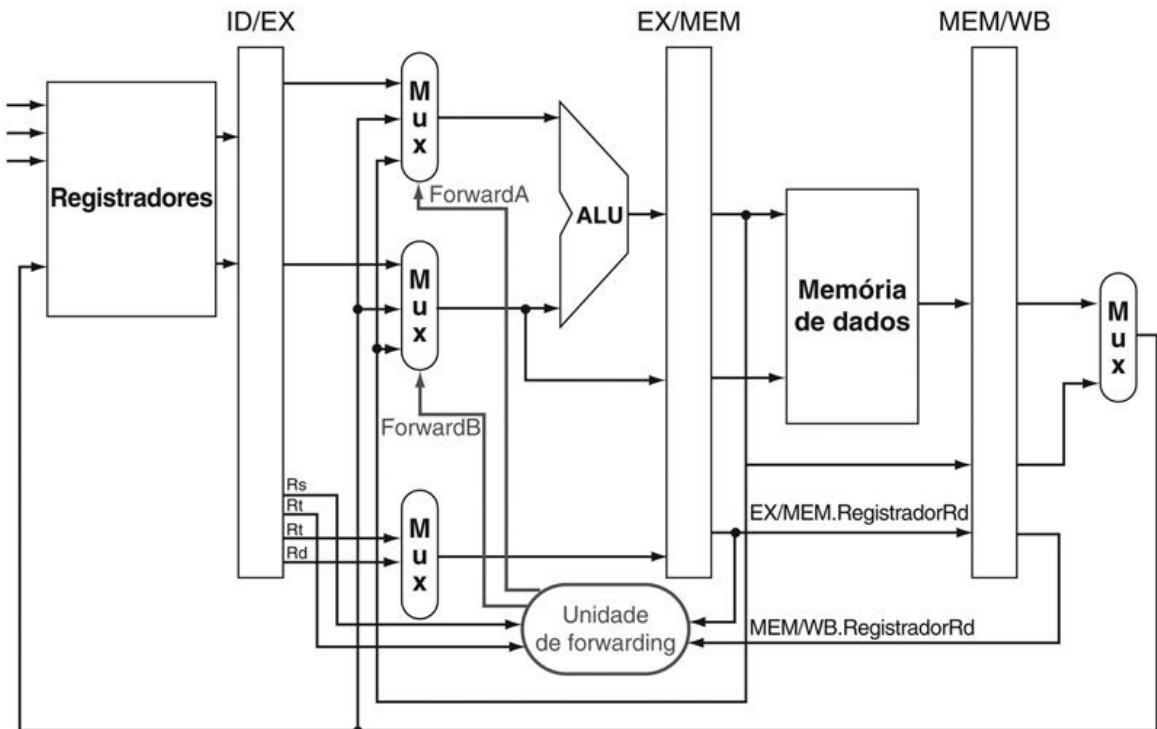
clock, de modo que `add` não causa stall, mas os valores vêm do banco de registradores e não de um registrador de pipeline. O “forwarding” do banco de registradores — ou seja, a leitura apanha o valor da escrita nesse ciclo de clock — é o motivo pelo qual o ciclo de clock 5 mostra o registrador `$2`, tendo o valor 10 no início e -20 no final do ciclo de clock. Como no restante desta seção, tratamos de todo o forwarding, exceto o valor a ser armazenado por uma instrução store.

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. Aumentando 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 4.54](#) mostra um detalhe da ALU e do registrador de pipeline, antes e depois de acrescentar o forwarding. A [Figura 4.55](#) 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.



a. Sem forwarding



b. Com forwarding

FIGURA 4.54 Em cima estão a ALU e os registradores de pipeline antes da inclusão do forwarding.

Embaixo, 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 sinal único. Como na discussão anterior, isso ignora o forwarding de um valor armazenado por uma instrução store. Observe também que esse mecanismo funciona, inclusive, para instruções slt.

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 4.55 Os valores de controle para os multiplexadores de forwarding da Figura 4.54.

O imediato com sinal que é outra entrada da ALU é descrito na Seção “Detalhamento” ao final desta seção.

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 operando 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 a fim de manter o campo rs. Logo, rs (bits 25-21) é acrescentado a ID/EX.

Agora, vamos escrever as duas condições para detectar hazards e os sinais de controle para resolvê-los:

1. *Hazard EX:*

```

if (EX/MEM.WriteReg
and (EX/MEM.RegistradorRd ≠ 0)
and (EX/MEM.RegistradorRd = ID/EX.RegistradorRs)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegistradorRd ≠ 0)
and (EX/MEM.RegistradorRd = ID/EX.RegistradorRt)) ForwardB = 10

```

Observe que o campo EX/MEM.RegistradorRd é o destino de registrador para uma instrução da ALU (que vem do campo Rd da instrução) ou um load (que vem do campo Rt).

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 que pegue o valor do registrador de pipeline EX/MEM.

2. Hazard MEM:

```

if (MEM/WB.WriteReg
and (MEM/WB.RegistradorRd ≠ 0)
and (MEM/WB.RegistradorRd = ID/EX.RegistradorRs)) ForwardA = 01

if (MEM/WB.WriteReg
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 sequência 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 mais recente. Assim, o controle para o hazard em MEM seria (com os acréscimos destacados):

```

if (MEM/WB.WriteReg
and (MEM/WB.RegistradorRd ≠ 0)
and not(EX/MEM.WriteReg and (EX/MEM.RegistradorRd ≠ 0)
        and (EX/MEM.RegistradorRd ≠ ID/EX.RegistradorRs))
and (MEM/WB.RegistradorRd = ID/EX.RegistradorRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegistradorRd ≠ 0))
        and (EX/MEM.RegistradorRd ≠ ID/EX.RegistradorRt)
and (MEM/WB.RegistradorRd = ID/EX.RegistradorRt)) ForwardB = 01

```

A [Figura 4.56](#) mostra o hardware necessário para dar suporte ao forwarding para operações que utilizam resultados durante o estágio EX. Observe que o campo EX/MEM.RegistradorRd é o destino do registrador para uma instrução ALU (que vem do campo Rd da instrução) ou um load (que vem do campo Rt).

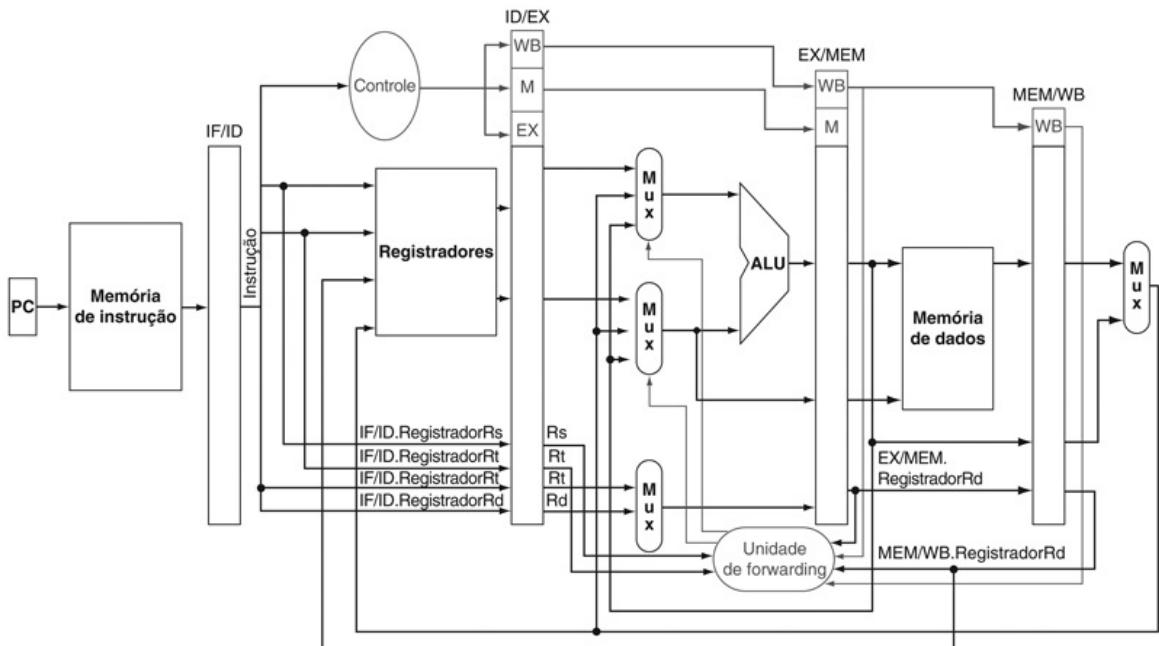


FIGURA 4.56 O caminho de dados modificado para resolver os hazards via forwarding.

Em comparação com o caminho de dados da [Figura 4.51](#), 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.

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, útil quando se realiza cópias da memória para a memória na arquitetura MIPS. Como as cópias são frequentes, precisamos acrescentar mais hardware de forwarding, a fim de fazer com que as cópias de memória para memória se tornem mais rápidas. Se tivéssemos de redesenhar a Figura 4.53, 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 para o leitor.

Além disso, a entrada imediata com sinal para a ALU, necessária para loads e stores, não existe no caminho de dados da Figura 4.56. 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 4.57 mostra esse acréscimo.

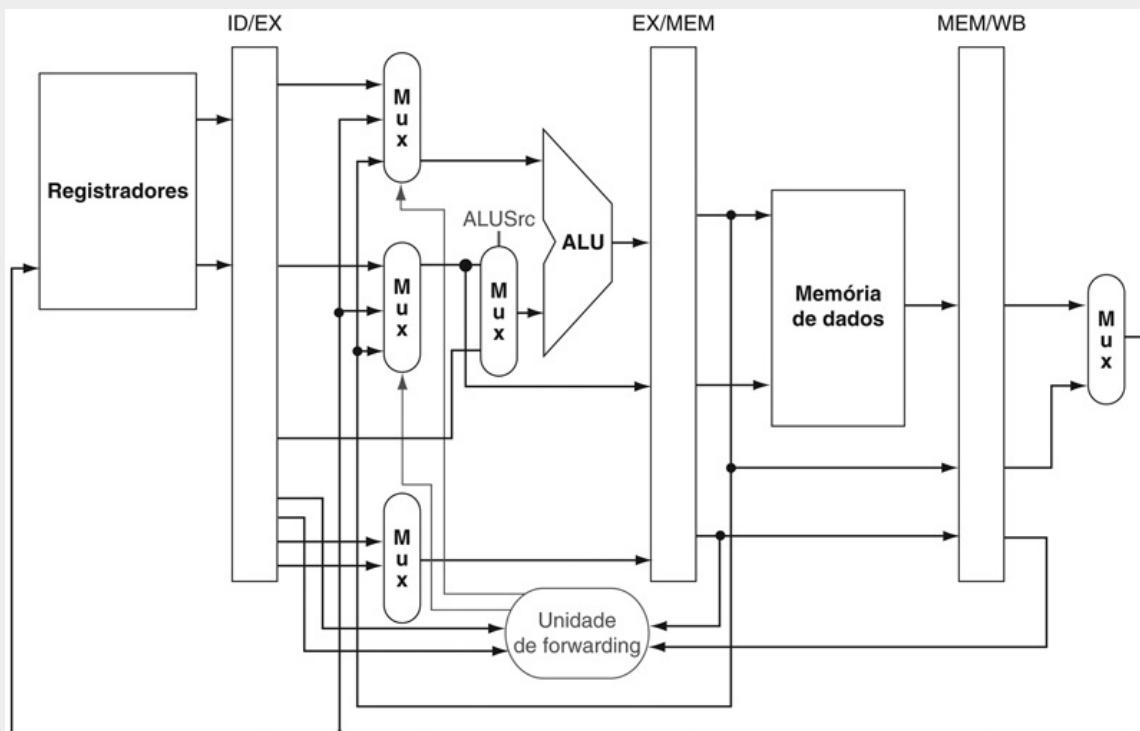


FIGURA 4.57 Uma visão de perto do caminho de dados da Figura 4.54 mostra um multiplexador 2:1, que foi acrescentado para selecionar o imediato com sinal como uma entrada para a ALU

Hazards de dados e stalls

Se a princípio você não obteve sucesso, redefina sucesso.

Anônimo

Conforme dissemos na [Seção 4.5](#), 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 4.58](#) 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 a combinação de load seguida por uma instrução que lê seu resultado.

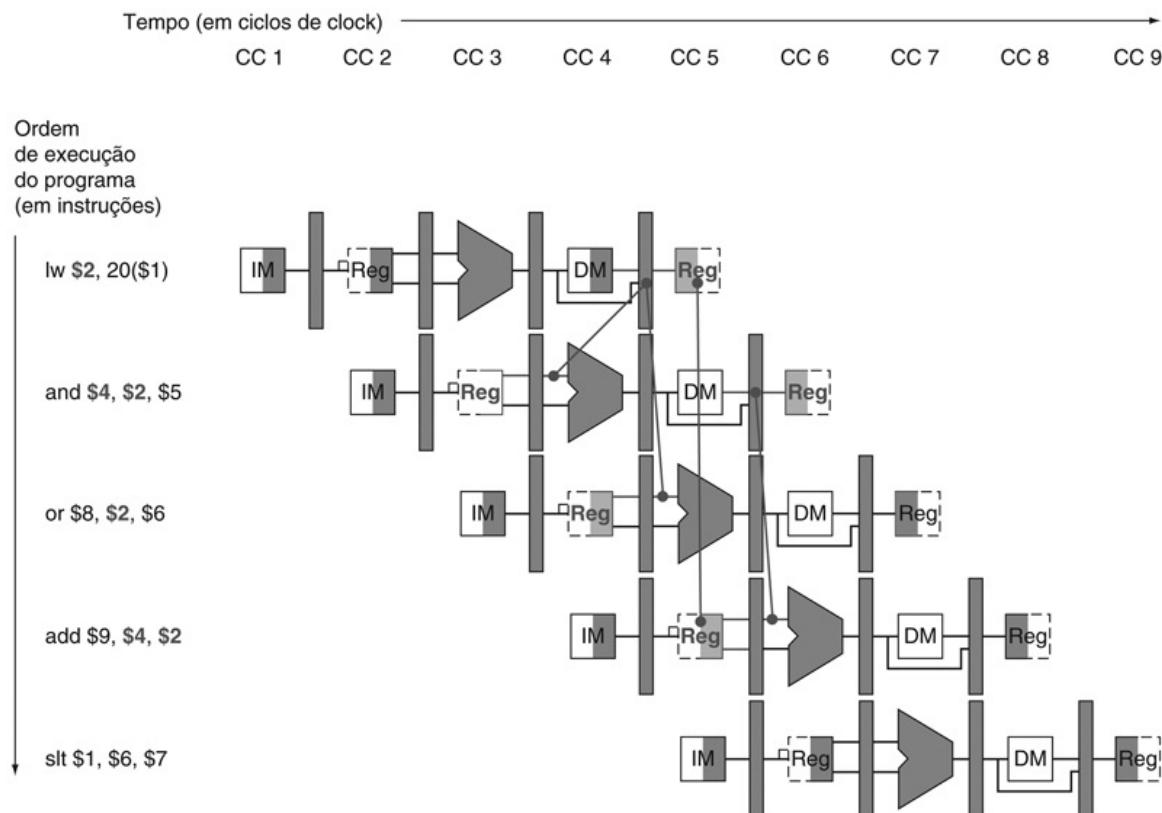


FIGURA 4.58 Uma sequência de instruções em pipeline.

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.

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 o 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.ReadMem 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 um ciclo de clock no pipeline. Depois desse stall de um ciclo, a lógica de forwarding pode lidar com a dependência e a execução prossegue. (Se não houvesse forwarding, então as instruções na [Figura 4.58](#) 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 continuar a trabalhar vazia. Naturalmente, assim como a secadora, a metade do pipeline que começa com o estágio EX precisa estar fazendo algo; o que ela está fazendo é executar instruções que não têm efeito algum: **nops**.

nop

Uma instrução que não realiza operação para mudar de estado.

Como podemos inserir esses nops, que atuam como bolhas, no pipeline? Na [Figura 4.49](#), 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 4.59 mostra o que realmente acontece no hardware: o slot de execução 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. 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. Neste exemplo, o hazard força as instruções AND e OR a repetir no ciclo de clock 4 o que fizeram no ciclo de clock 3: AND lê registradores e decodifica, e OR é apanhado novamente da memória de instruções. Esse trabalho repetido é um stall, mas seu efeito é esticar o tempo das instruções AND e OR e atrasar a busca da instrução add.

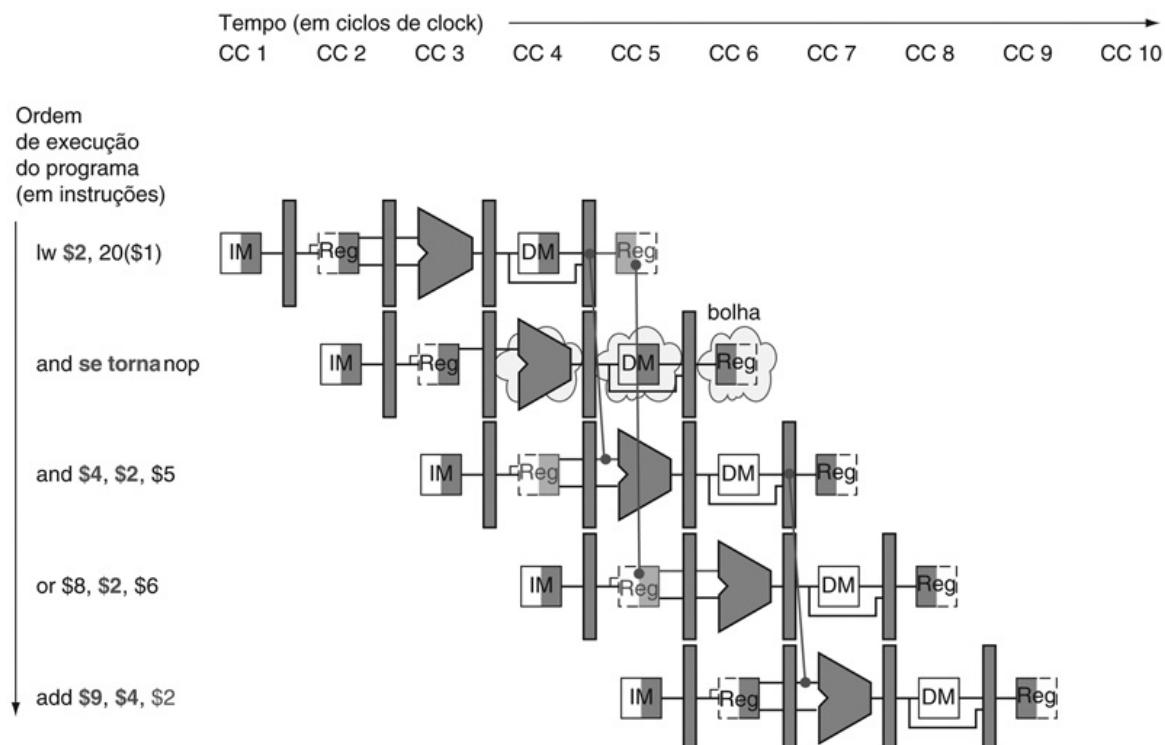


FIGURA 4.59 O modo como os stalls são realmente inseridos no pipeline.

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 ID 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.

A [Figura 4.60](#) 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, a fim de substituir o valor de um registrador de uso geral pelo valor de um 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.

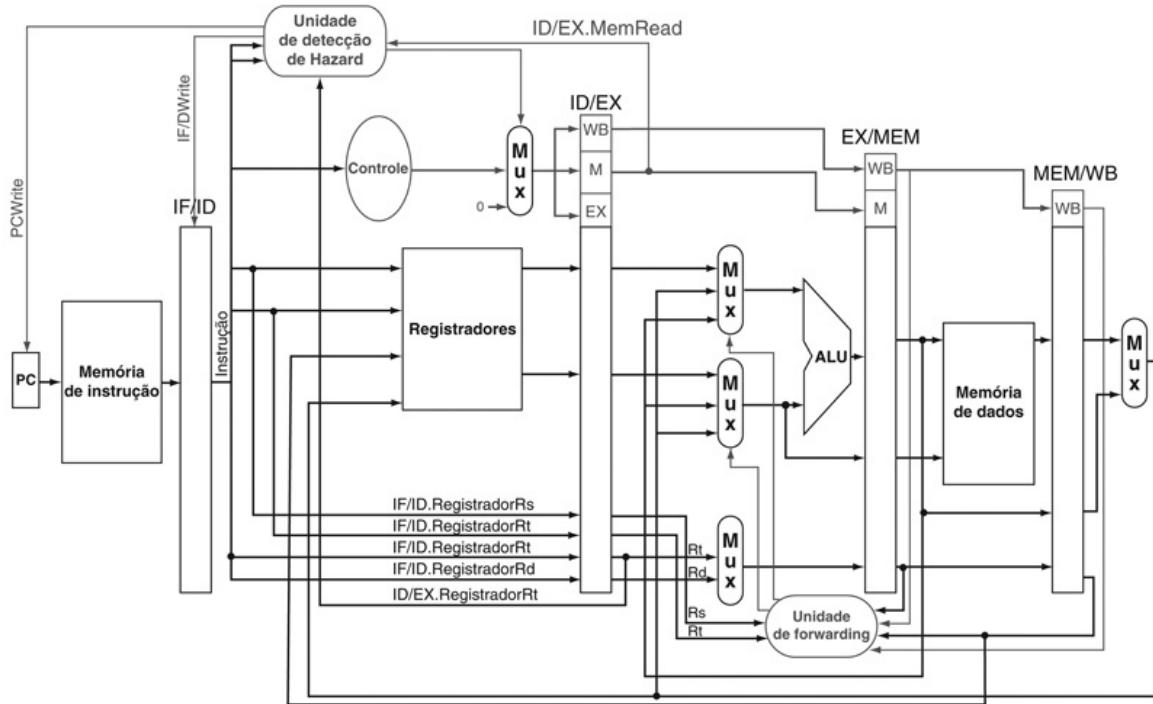


FIGURA 4.60 Visão geral do controle em pipeline, mostrando os dois multiplexadores para forwarding, a unidade de detecção de hazard e a unidade de forwarding.

Embora os estágios ID e EX tenham sido simplificados — a lógica de extensão de sinal imediato e de desvio estão faltando —, este desenho mostra a essência dos requisitos do hardware de forwarding.

Colocando em perspectiva

Embora o compilador geralmente conte com o hardware para resolver dependências de hazard e garantir assim a execução correta, o compilador precisa compreender o pipeline, a fim de alcançar o melhor desempenho. Caso contrário, stalls inesperados reduzirão o desempenho do código compilado.

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 WriteReg e WriteMem precisam ser 0, enquanto os outros sinais de controle podem ser “don’t care”.

4.8. Hazards de controle

Para cada mal que está batendo na raiz há milhares pendurados nos galhos.

Henry David Thoreau, Walden, 1854

Até aqui, preocupamo-nos apenas com os hazards envolvendo operações aritméticas e transferências de dados. Entretanto, como vimos na [Seção 4.5](#), também existem hazards de pipeline envolvendo desvios. A [Figura 4.61](#) 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 4.5](#), 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.

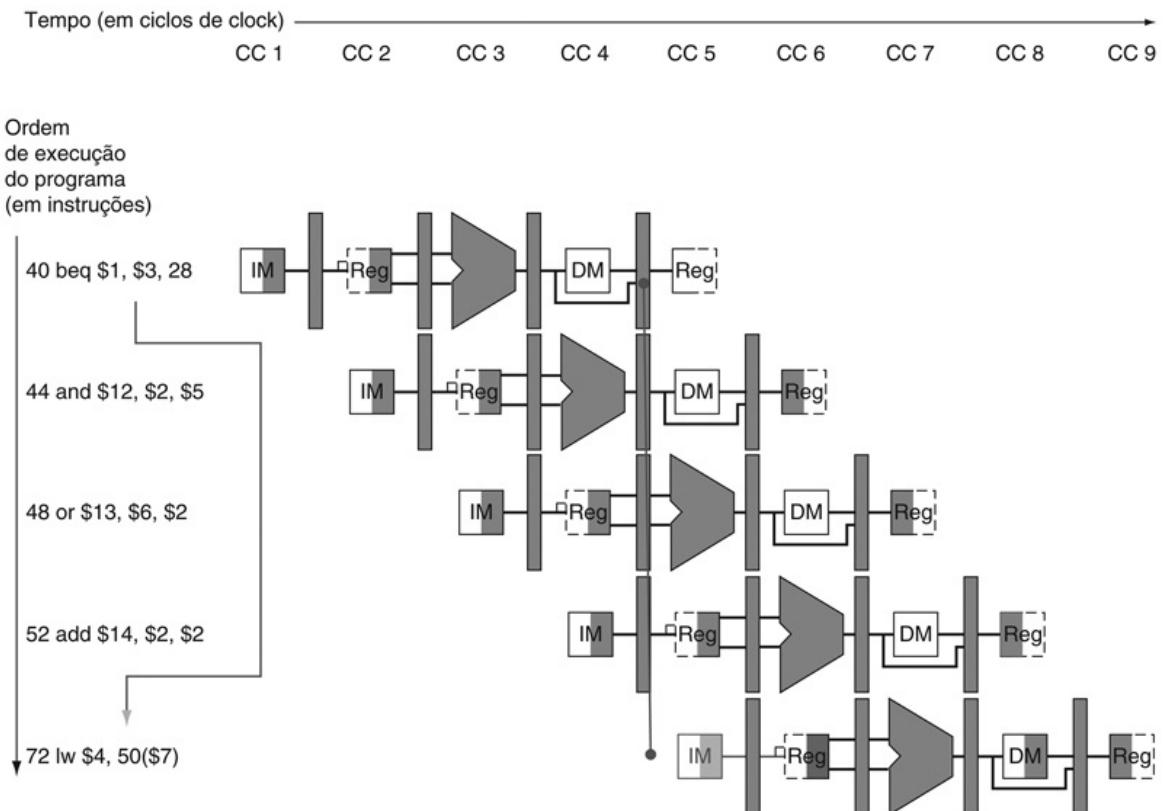
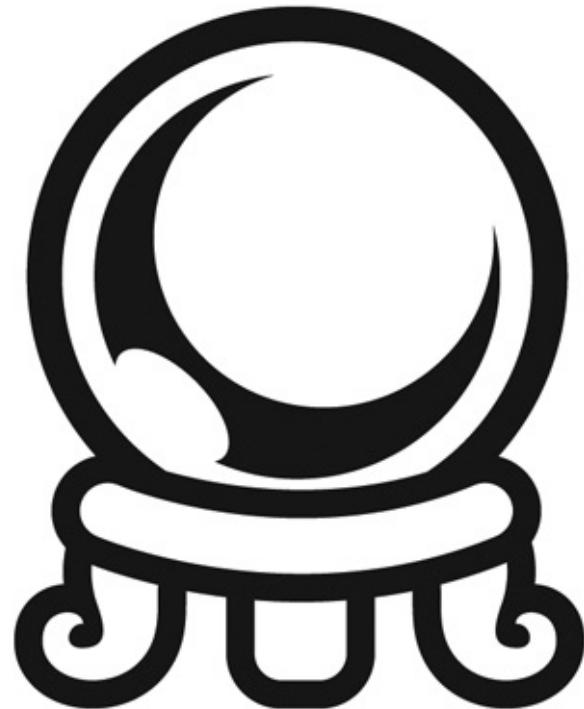


FIGURA 4.61 O impacto do pipeline sobre a instrução branch.

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 sequenciais que seguem o branch serão buscadas e iniciarão sua execução. Sem intervenção, essas três instruções seguintes começarão a executar antes que o beq desvie para lw na posição 72. (A Figura 4.31 considerou um hardware extra para reduzir o hazard de controle a um ciclo de clock; esta figura usa o caminho de dados não otimizado.)

Esta seção sobre hazards de controle é mais curta do que as seções anteriores, sobre hazards de dados, porque os hazards de controle são relativamente simples de entender, e ocorrem com menos frequência que os hazards de dados. Além disso, não há nada tão eficiente contra os hazards de controle como o forwarding é contra 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



P R E D I Ç Ã O

Como vimos na [Seção 4.5](#), fazer um stall até que o desvio termine é muito lento. Uma melhoria comum ao stall do desvio é **prever** 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 no 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

Descarte de instruções em um pipeline, normalmente devido a um evento inesperado.

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, a comparação realizada por uma ALU, através de instrução separada, é requisitada — uma situação semelhante ao uso de códigos de condição para os desvios ([Capítulo 2](#)).

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 precisará funcionar corretamente com essa otimização. Por exemplo, a fim de implementar branch-on-equal (e seu inverso), teremos de fazer um forwarding dos resultados para a lógica do teste de igualdade que opera durante o estágio ID. Existem dois fatores que comprometem o procedimento:

1. Durante o estágio ID, temos de decodificar a instrução, decidir se é necessário um bypass para a unidade de igualdade 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. Por extensão, se um load for imediatamente seguido por um desvio condicional que está no resultado do load, dois ciclos de stall serão necessários, pois o resultado do load aparece no final do ciclo MEM, mas é necessário no início do 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 sequência 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 4.62 mostra o que acontece quando um desvio é tomado. Diferente da Figura 4.61, há somente uma bolha no pipeline para o desvio tomado.

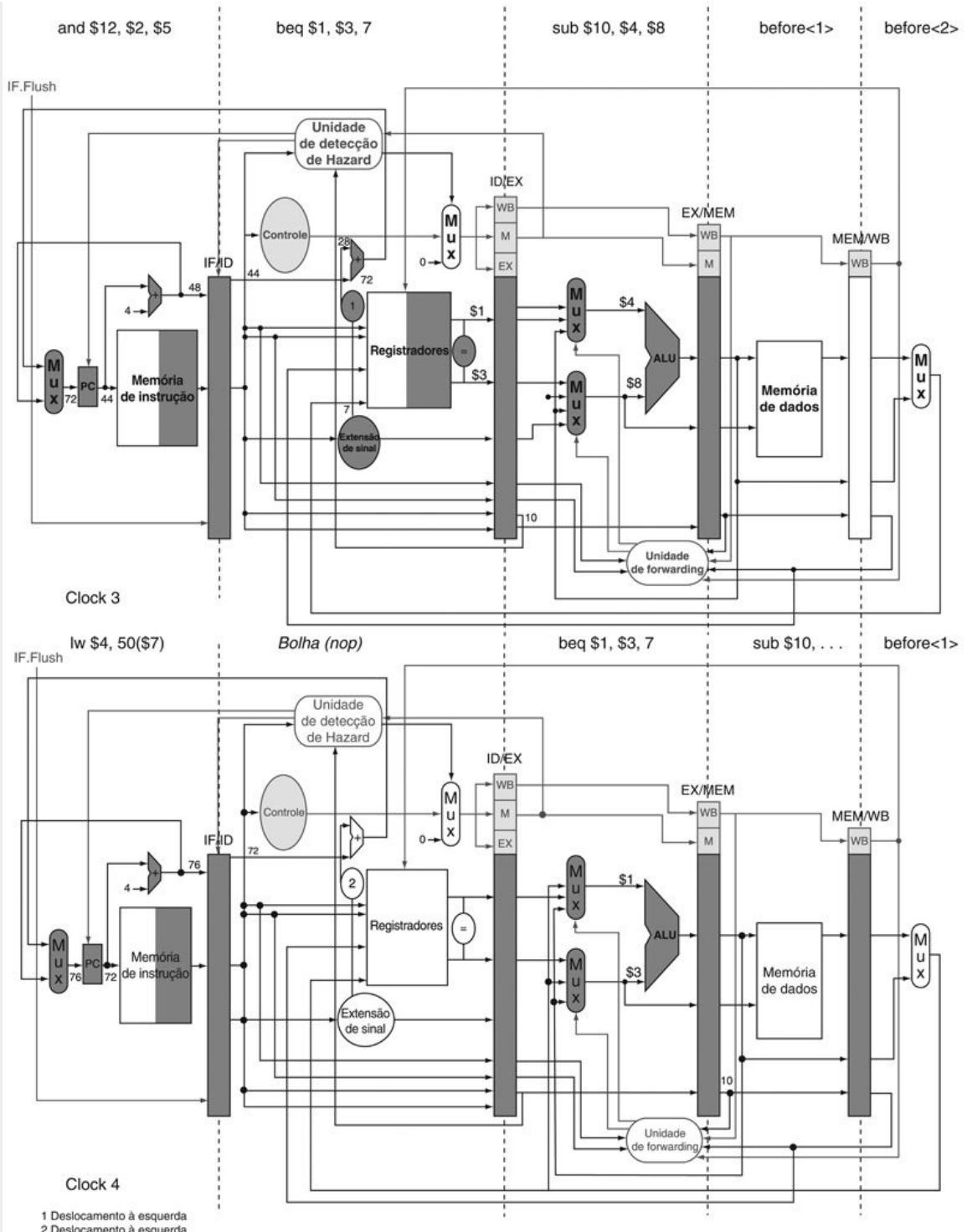


FIGURA 4.62 O estágio ID do ciclo de clock 3 determina que um desvio precisa ser tomado, de modo que seleciona 72 como próximo endereço do PC e zera a instrução buscada para o próximo ciclo de clock.

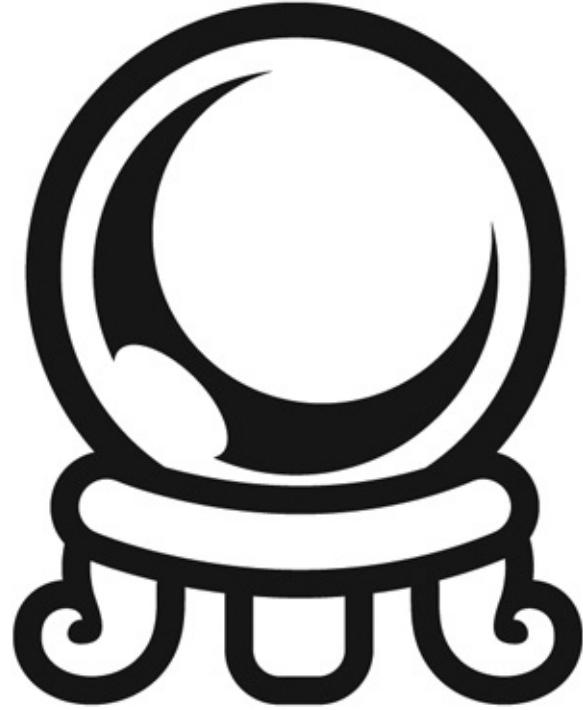
O ciclo de clock 4 mostra a instrução no local 72 sendo buscada e a única bolha ou instrução *nop* no pipeline como

resultado do desvio tomado. (Como o `nop` na realidade é `s11 $0,$0,0`, é discutível se o estágio ID no clock 4 deve ou não ser destacado.)

Previsão dinâmica de desvios

Supor que um desvio não seja tomado é uma forma simples de *previsão de desvios*. Nesse caso, prevemos 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 ([Seção 4.10](#)), a penalidade do desvio aumenta em termos de instruções perdidas. Essa combinação significa que, em um pipeline agressivo, um esquema de previsão estática provavelmente desperdiçará muito desempenho. Como mencionamos na [Seção 4.5](#), 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 **previsão dinâmica de desvios**.



P R E D I Ç Ã O

previsão dinâmica de desvios

Previsão de desvios durante a execução, usando informações em tempo de execução.

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.

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.

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 sequência apropriada é buscada e executada.

Esse esquema simples 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”, já que 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 4.63](#) mostra a máquina de estados finitos para um esquema de previsão de 2 bits.

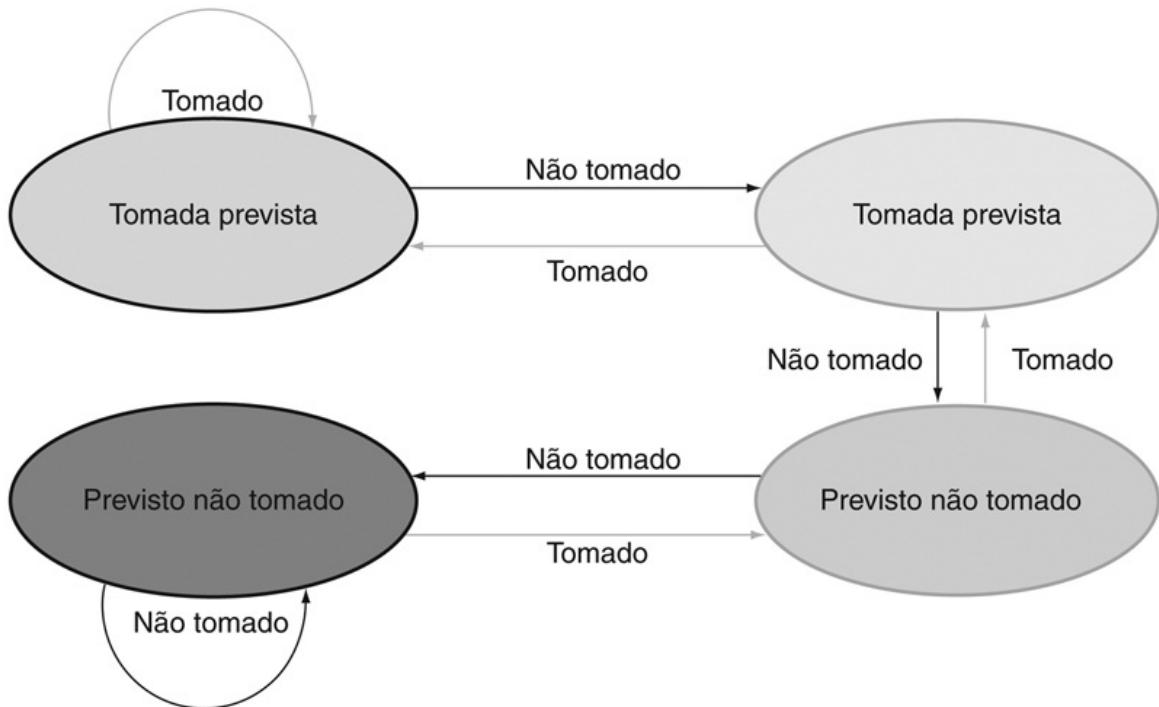


FIGURA 4.63 Os estados em um esquema de previsão de 2 bits.

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 2 bits é um caso geral de uma previsão baseada em contador, incrementado quando a previsão é exata e decrementado em caso contrário, utilizando o ponto intermediário desse intervalo como divisão entre desvio tomado e não tomado.

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; conforme mencionamos anteriormente, isso pode ser até mesmo no estágio ID. Caso contrário, a busca e a execução sequencial continuam. Se a previsão for errada, os bits de previsão são trocados, como mostra a [Figura 4.63](#).

Detalhamento

Conforme descrevemos na Seção 4.5, 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 4.64 mostra as três maneiras como o delay slot do desvio pode ser escalonado.

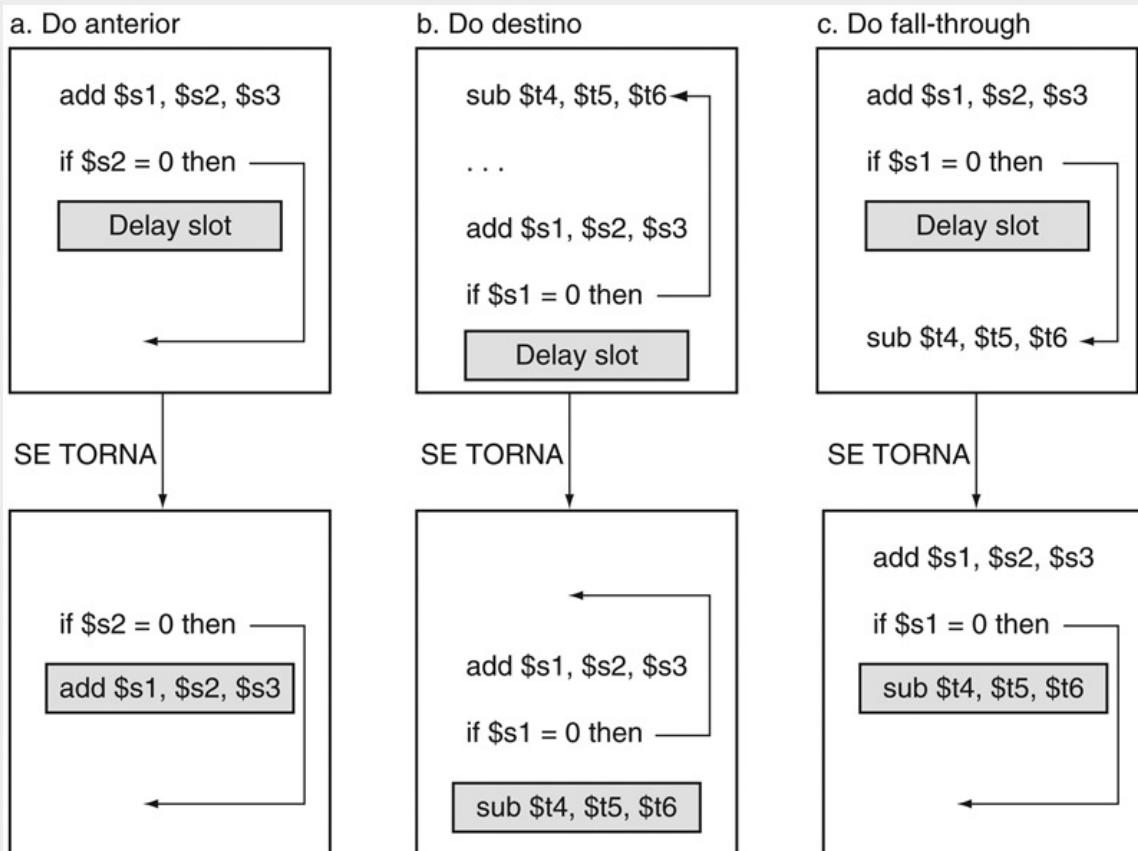


FIGURA 4.64 Escalonando o delay slot do desvio.

Em 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 sequências 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

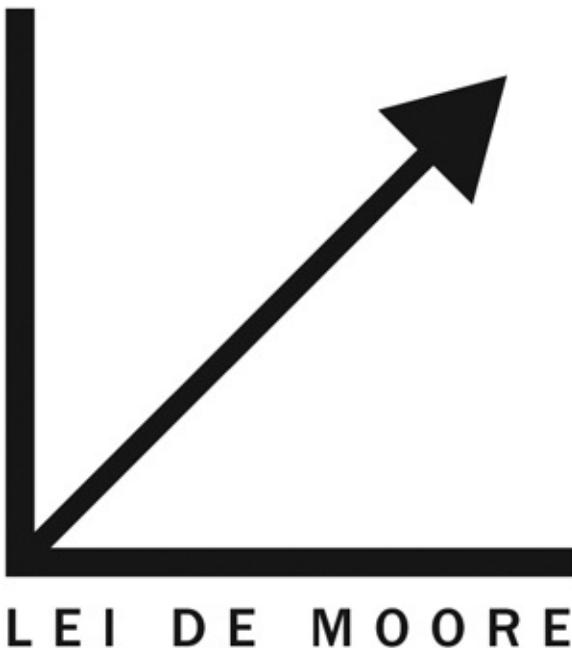
estratégia (b) é preferida quando o desvio é tomado com alta probabilidade, como em um desvio de loop. Finalmente, o desvio pode ser escalonado a partir da sequência não tomada, como em (c). Para tornar essa otimização válida para (b) ou (c), deve ser válido executar a instrução `sub` quando o desvio seguir na direção inesperada. Com “válido”, 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.

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 (Seção 4.10), 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 dinâmicas mais dispendiosas, porém mais flexíveis. Simultaneamente, o crescimento em transistores disponíveis por chip, devido à **Lei de Moore**, tornou a previsão dinâmica relativamente mais barata.

delay slot do desvio

O slot diretamente após a instrução de delayed branch, que na arquitetura MIPS é preenchido por uma instrução que não afeta o desvio.



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 um ciclo, significando que os desvios tomados terão uma penalidade de um ciclo. Os delayed branches são uma técnica para eliminar essa penalidade. Outra técnica é usar uma cache para manter o contador de programa de destino ou instrução de destino, usando um **buffer de destino de desvios**.

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, rastreando, 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 desvio de torneio

Um previsor de desvios com múltiplas previsões para cada desvio e um mecanismo de seleção que escolhe qual previsor deve ser usado para determinado desvio

Detalhamento

Uma maneira de reduzir o número de desvios condicionais é acrescentar instruções de *move condicional*. Em vez de mudar o PC com um desvio condicional, a instrução muda condicionalmente o registrador de destino do move. Se a condição falha, o move atua como um nop. Por exemplo, uma versão da arquitetura do conjunto de instruções MIPS tem duas novas instruções chamadas `movn` (move if not zero) e `movz` (move if zero). Assim, `movn $8,$11,$4` copia o conteúdo do registrador 11 para o registrador 8, desde que o valor no registrador 4 seja diferente de zero; caso contrário, ela não faz nada.

O conjunto de instruções ARMv7 tem um campo de condição na maioria das instruções. Assim, os programas ARM poderiam ter menos desvios condicionais que os programas MIPS.

Resumo sobre pipeline

Começamos na lavanderia, mostrando princípios de pipelining em um ambiente cotidiano. Usando essa analogia como um guia, explicamos o pipelining de instruções passo a passo, começando com um caminho de dados de ciclo único e depois acrescentando registradores de pipeline, caminhos de forwarding, detecção de hazard de dados, previsão de desvio e com flushing de instruções em exceções. A [Figura 4.65](#) mostra o caminho de dados e controle final para este capítulo. Agora, estamos prontos para outro hazard de controle: a questão complicada das exceções.

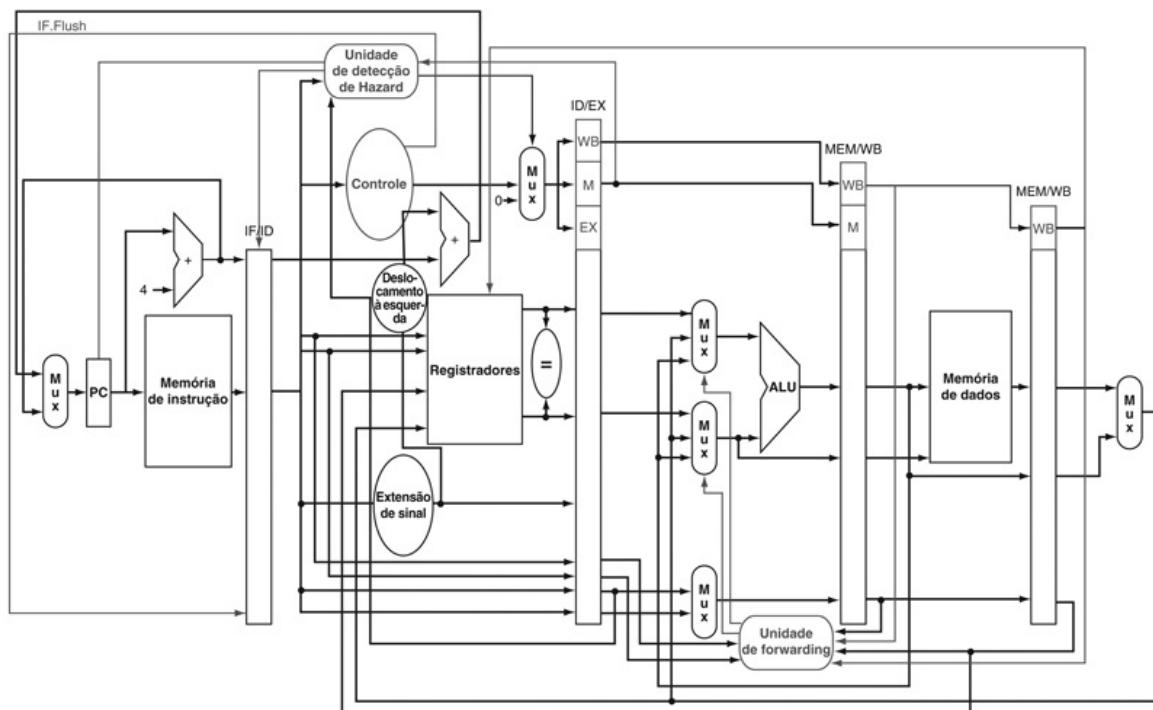


FIGURA 4.65 O caminho de dados e controle final para este capítulo.

Observe que essa é uma figura estilizada, em vez de um caminho de dados detalhado, de modo que não contém o mux ALUScr da [Figura 4.57](#) e os controles multiplexadores da [Figura 4.51](#).

Verifique você mesmo

Considere três esquemas de previsão de desvios: desvio não tomado, previsão tomada e previsão dinâmica. Suponha que todos eles tenham penalidade zero quando preveem 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?

1. Um desvio tomado com frequência de 5%.
2. Um desvio tomado com frequência de 95%.
3. Um desvio tomado com frequência de 70%.

4.9. Exceções

Fazer um computador com facilidades automáticas de interrupção de programa se comportar [sequencialmente] não foi uma tarefa fácil, pois o número de instruções em diversos estágios do processamento, quando ocorre um sinal de interrupção, pode ser muito grande.

Fred Brooks Jr., *Planning a Computer System: Project Stretch*, 1962

Controle é o aspecto mais desafiador do projeto do processador: ele é a parte mais difícil de se acertar e a parte mais difícil de tornar mais rápida. Uma das partes mais difíceis do controle é implementar **exceções** e **interrupções** — eventos diferentes dos desvios ou saltos, que mudam o fluxo normal da execução da instrução. Eles foram criados inicialmente para tratar de eventos inesperados de dentro do processador, como o overflow aritmético. O mesmo mecanismo básico foi estendido para os dispositivos de E/S se comunicarem com o processador, conforme veremos no [Capítulo 5](#).

exceção

Também chamada **interrupção**. Um evento não programado 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.)

Muitas arquiteturas e autores não fazem distinção entre interrupções e exceções, normalmente usando o nome mais antigo *interrupção* para se referirem aos dois tipos de eventos. Por exemplo, o Intel x86 usa interrupção. Seguimos a convenção do MIPS, usando o termo *exceção* para indicar *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. Aqui estão alguns exemplos mostrando se a situação é gerada internamente pelo processador ou se é gerada externamente:

Tipo de evento	De onde?	Terminologia MIPS
Solicitação de dispositivo de E/S	Externa	Interrupção
Chamar o sistema operacional do programa do usuário	Interna	Exceção
Overflow aritmético	Interna	Exceção
Usar uma instrução indefinida	Interna	Exceção
Defeitos do hardware	Ambos	Exceção ou interrupção

Muitos dos requisitos para dar suporte a exceções vêm da situação específica que causa a ocorrência de uma exceção. Consequentemente, retornaremos a esse assunto no [Capítulo 5](#), quando entenderemos melhor a motivação para as capacidades adicionais no mecanismo de exceção. Nesta seção, lidamos com a implementação de controle de modo a detectar dois tipos de exceções que surgem das partes do conjunto de instruções e da implementação que já discutimos.

Detectar condições excepcionais e tomar a ação apropriada normalmente está no percurso de temporização crítico de um processador, que determina o tempo de ciclo de clock e, portanto, o desempenho. Sem a devida atenção às exceções durante o projeto da unidade de controle, as tentativas de acrescentar exceções a uma implementação complicada podem reduzir o desempenho significativamente, bem como complicar a tarefa de corrigir o projeto.

Como as exceções são tratadas em uma arquitetura MIPS

Os dois tipos de exceções que nossa implementação atual pode gerar são: a execução de uma instrução indefinida e um overflow aritmético. Usaremos o

overflow aritmético na instrução add \$1,\$2,\$1 como exemplo de exceção nas próximas páginas. A ação básica que o processador deve realizar quando ocorre uma exceção é salvar o endereço da instrução causadora no *contador de programa de exceção* (Exception Program Counter — EPC) e depois transferir o controle para o sistema operacional em algum endereço especificado.

O sistema operacional pode então tomar a ação apropriada, que pode ser fornecer algum serviço ao programa do usuário, tomar alguma ação predefinida em resposta a um overflow ou terminar a execução do programa e informar um erro. Depois de realizar qualquer ação necessária devido à exceção, o sistema operacional pode terminar o programa ou pode continuar sua execução, usando o EPC para determinar onde reiniciar a execução do programa. No [Capítulo 5](#), veremos mais de perto a questão da retomada da execução.

Para o sistema operacional tratar da 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 da arquitetura MIPS inclui um registrador de status (chamado *registrar Cause*), que mantém um campo que indica o motivo da exceção.

Um segundo método usa **interrupções vetorizadas**. Em uma interrupção vetorizada, o endereço ao qual o controle é transferido é determinado pela causa da exceção. Por exemplo, para acomodar os dois tipos de exceção listados anteriormente, poderíamos definir os dois endereços de vetor de exceção a seguir:

interrupção vetorizada

Uma interrupção para a qual o endereço para onde o controle é transferido é determinado pela causa da exceção.

Tipo de exceção	Endereço do vetor de exceção (em hexa)
Instrução indefinida	8000 0000 _{hexa}
Overflow aritmético	8000 0180 _{hexa}

O sistema operacional sabe o motivo para a exceção pelo endereço em que ela é iniciada. Os endereços são separados por 32 bytes ou oito instruções, e o sistema operacional precisa registrar o motivo para a exceção e pode realizar algum processamento limitado nessa sequência. Quando a exceção não é vetorizada, um único ponto de entrada para todas as exceções pode ser utilizado,

e o sistema operacional decodifica o registrador de status para encontrar a causa.

Podemos realizar o processamento exigido para exceções acrescentando alguns registradores e sinais de controle extras à nossa implementação básica e estendendo o controle ligeiramente. Vamos supor que estejamos implementando o sistema de exceção utilizado na arquitetura MIPS, com o único ponto de entrada sendo o endereço 8000 0180_{hexa}. (A implementação de exceções vetorializadas não é mais difícil.) Precisaremos acrescentar dois registradores adicionais à nossa implementação MIPS atual:

- *EPC*: Um registrador de 32 bits usado para manter 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 tem 32 bits, embora alguns bits atualmente não sejam utilizados. Suponha que haja um campo de cinco bits que codifica as duas fontes de informação possíveis mencionadas anteriormente, com 10 representando uma instrução indefinida e 12 representando o overflow aritmético.

Exceções em uma implementação em pipeline

Uma implementação em pipeline trata exceções como outra forma de hazard de controle. Por exemplo, suponha que haja um overflow aritmético em uma instrução add. 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, desta 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, a fim de dar flush durante o ID. Para dar flush na instrução na fase EX, usamos um novo sinal, chamado EX.Flush, fazendo com que novos multiplexadores zerem as linhas de controle. Para começar a buscar instruções do local 8000 0180_{hexa}, que é o local da exceção para o overflow aritmético, simplesmente acrescentamos uma entrada adicional ao multiplexador do PC, que envia 8000

0180_{hexa} ao PC. A Figura 4.66 mostra essas mudanças.

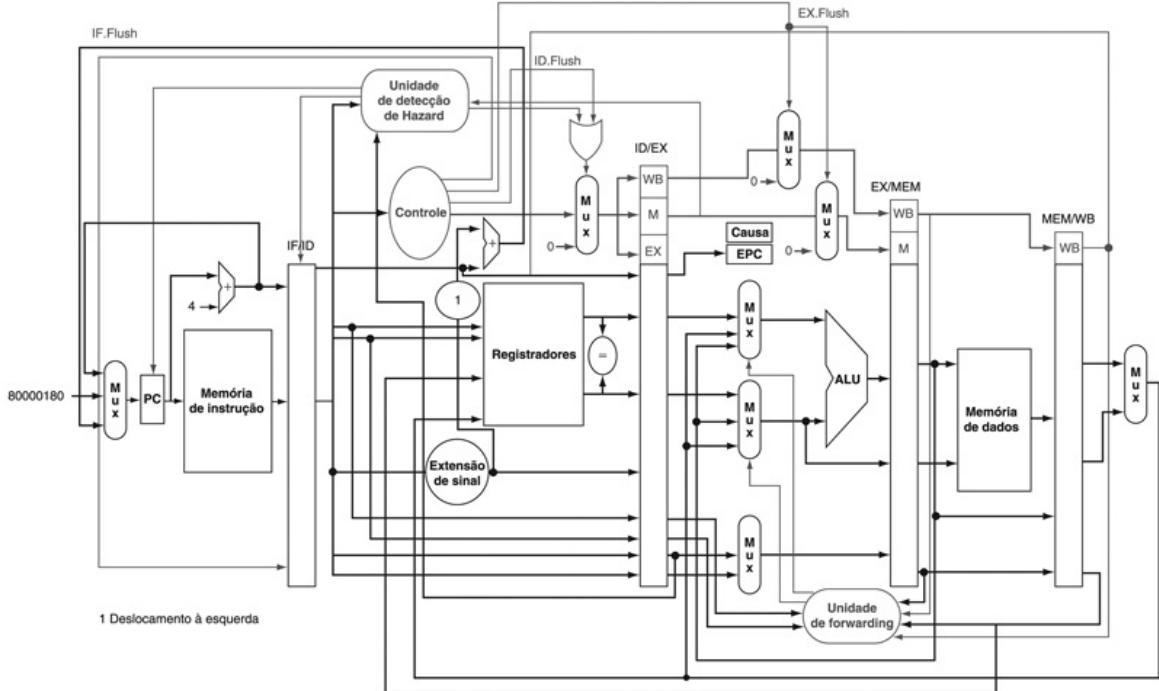


FIGURA 4.66 O caminho de dados com controles para lidar com exceções.

Os principais acréscimos incluem uma nova entrada, com o valor $8000\ 0180_{\text{hexa}}$, 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 $8000\ 0180_{\text{hexa}}$ 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.

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 início após a exceção ser tratada.

A etapa final é salvar o endereço da instrução problemática no *Exception Program Counter* (EPC). Na realidade, salvamos o endereço +4, de modo que a rotina de tratamento da exceção, primeiro deve subtrair 4 do valor salvo. A Figura 4.66 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 sequência de instruções

40 _{hex}	sub	\$11,	\$2,	\$4
44 _{hex}	and	\$12,	\$2,	\$5
48 _{hex}	or	\$13,	\$2,	\$6
4C _{hex}	add	\$1,	\$2,	\$1
50 _{hex}	slt	\$15,	\$6,	\$7
54 _{hex}	lw	\$16,	50(\$7)	
• • •				

considere que as instruções a serem invocadas em uma exceção começem desta forma:

80000180 _{hex}	SW	\$26 , 1000(\$0)
80000184 _{hex}	SW	\$27 , 1004(\$0)
• • •		

Mostre o que acontece no pipeline se houver uma exceção de overflow na instrução add.

Resposta

A Figura 4.67 mostra os eventos, começando com a instrução add no estágio EX. O overflow é detectado durante essa fase, e 8000 0180_{hexa} 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_{hexa} + 4 = 50_{hexa}.

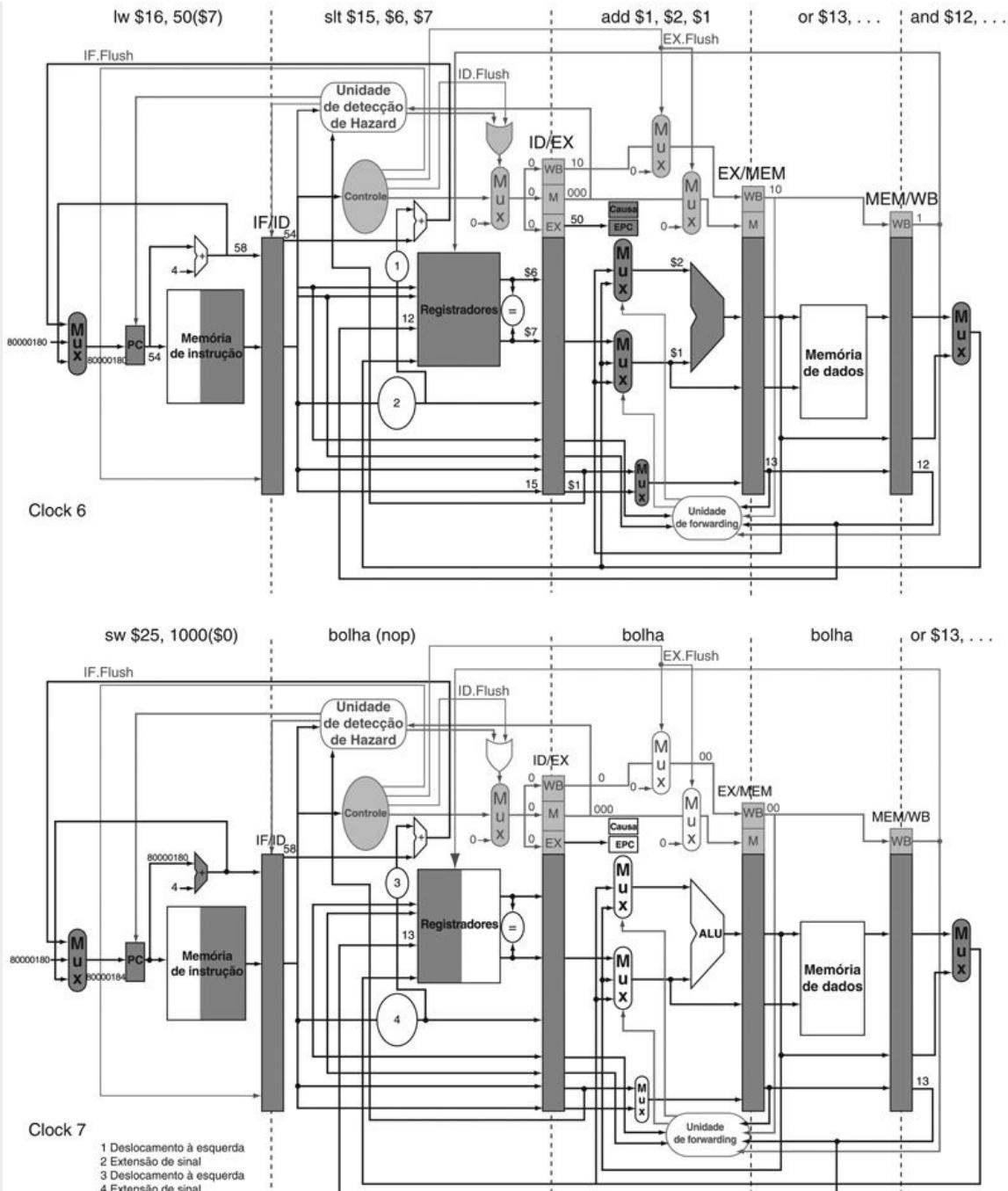


FIGURA 4.67 O resultado de uma exceção devido a um overflow aritmético na instrução add.

O overflow é detectado durante o estágio EX do clock 6, salvando o endereço apóis o add no registrador EPC ($4C + 4 = 50_{\text{hexa}}$). 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, mas a busca da primeira instrução da rotina de exceção — `sw $25,1000($0)` — a partir do local da instrução `8000 0180hexa`. 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.

Mencionamos cinco exemplos de exceções na tabela da [Seção 4.9](#), e veremos outros no [Capítulo 5](#). 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 é priorizar as exceções de modo que seja fácil determinar qual será atendida primeiro. 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 em um campo de exceção pendente, de modo que o hardware possa interromper com base em exceções posteriores, uma vez que a mais antiga tenha sido atendida.

Interface hardware/software

O hardware e o sistema operacional precisam trabalhar em conjunto para que as exceções se comportem conforme o esperado. 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 próprio sistema 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 é fundamental a capacidade de salvar e restaurar o estado de qualquer tarefa. Um dos usos mais importantes e frequentes das exceções é o tratamento de faltas de página e exceções de TLB; o Capítulo 5 descreve essas exceções e seu tratamento com mais detalhes.

Detalhamento

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_{hexa} no início do ciclo de clock, depois que a exceção for detectada, embora a instrução com problema esteja no endereço 4C_{hexa}. Um processador com exceções imprecisas poderia colocar 58_{hexa} 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 5.)

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.

Detalhamento

Embora o MIPS utilize o endereço de entrada de exceção 8000 0180_{hexa} para quase todas as exceções, ele usa o endereço 8000 0000_{hexa} de modo a melhorar o desempenho do tratador de exceção para exceções de falta de TLB (Capítulo 5).

Verifique você mesmo

Qual exceção deverá ser reconhecida primeiro nesta sequência?

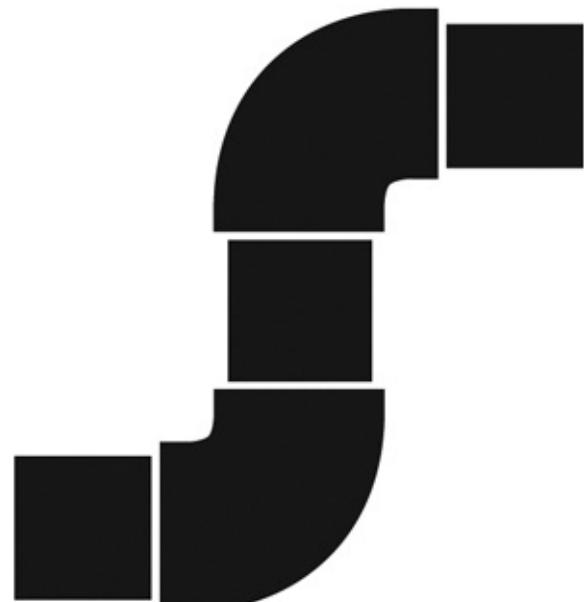
1. add \$1, \$2, \$1 # overflow aritmético
2. XXX \$1, \$2, \$1 # instrução indefinida
3. sub \$1, \$2, \$1 # erro de hardware

4.10. Paralelismo e paralelismo avançado em nível de instrução

Esteja avisado de que esta seção é uma breve introdução de assuntos fascinantes, porém avançados. Se você quiser saber mais detalhes, deverá consultar nosso livro mais avançado, *Computer Architecture: A Quantitative Approach*, 5^a edição (Morgan Kaufmann, 2012), quarta edição, no qual o material explicado nas próximas páginas é expandido para mais de 200 páginas (incluindo Apêndices)!

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, enxaguam 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.



PIPELINING



PARALELISMO

paralelismo em nível de instrução

O paralelismo entre as instruções.

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 de manter todas as máquinas ocupadas e transferir as trouxas de roupa para o próximo estágio do pipeline.

despacho múltiplo

Um esquema pelo qual múltiplas instruções são disparadas em 1 ciclo de clock.

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. Como dissemos no [Capítulo 1](#), às vezes, é útil inverter a

métrica e usar o IPC ou *instruções por ciclo de clock*. Logo, um microprocessador de despacho múltiplo quádruplo de 4 GHz pode executar uma velocidade de pico de 16 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 executando 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, sendo que a principal diferença está na 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 restritivos.

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 pelo processador durante a execução.

Existem duas responsabilidades principais e distintas que precisam ser tratadas em um pipeline de despacho múltiplo:

1. Empacotar as instruções em **slots de despacho**: como o processador determina quantas 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.

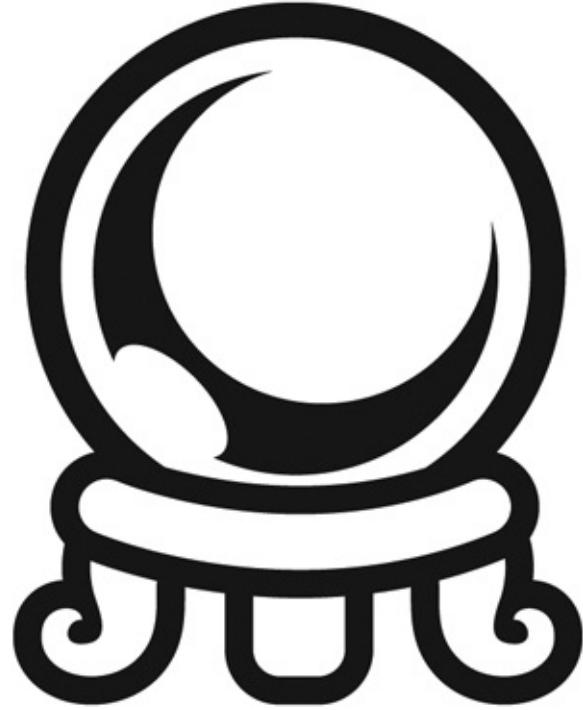
slots de despacho

As posições das quais as instruções poderiam ser despachadas em determinado ciclo de clock; por analogia, correspondem a posições nos blocos iniciais para uma atividade.

Embora as tenhamos descrito como técnicas distintas, na realidade, cada técnica pega algo emprestado da outra e nenhuma pode afirmar ser perfeitamente pura.

O conceito de especulação

Um dos métodos mais importantes para localizar e explorar mais ILP é a especulação. Com base na grande ideia da **predição**, **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. Outro exemplo é que 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.



P R E D I Ç Ã O

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.

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 estiver correta, as instruções são concluídas, permitindo que o conteúdo dos buffers seja escrito nos registradores ou na memória. Se a especulação estiver incorreta, o hardware faz um flush nos buffers e executa novamente, mas na sequência de instruções correta.

A especulação apresenta outro problema possível: especular sobre certas instruções pode gerar 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 causam 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

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)**.

pacote de despacho

O conjunto de instruções despachadas juntas em um ciclo de clock; o pacote pode ser determinado estaticamente, pelo compilador, ou dinamicamente, pelo processador.

VLIW (Very Long Instruction Word)

Um estilo de arquitetura de conjunto de instruções que dispara muitas operações definidas para serem independentes em uma única instrução larga, normalmente com muitos campos de opcode separados.

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.

Um exemplo: despacho múltiplo estático com a ISA do MIPS

Para que você tenha uma ideia 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 4.68](#) 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	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

FIGURA 4.68 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 no-ops 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 (Figura 4.69). Em um 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, a fim de calcular o endereço efetivo para as transferências de dados. Sem esses recursos extras, nosso pipeline com despacho duplo seria atrapalhado pelos hazards estruturais.

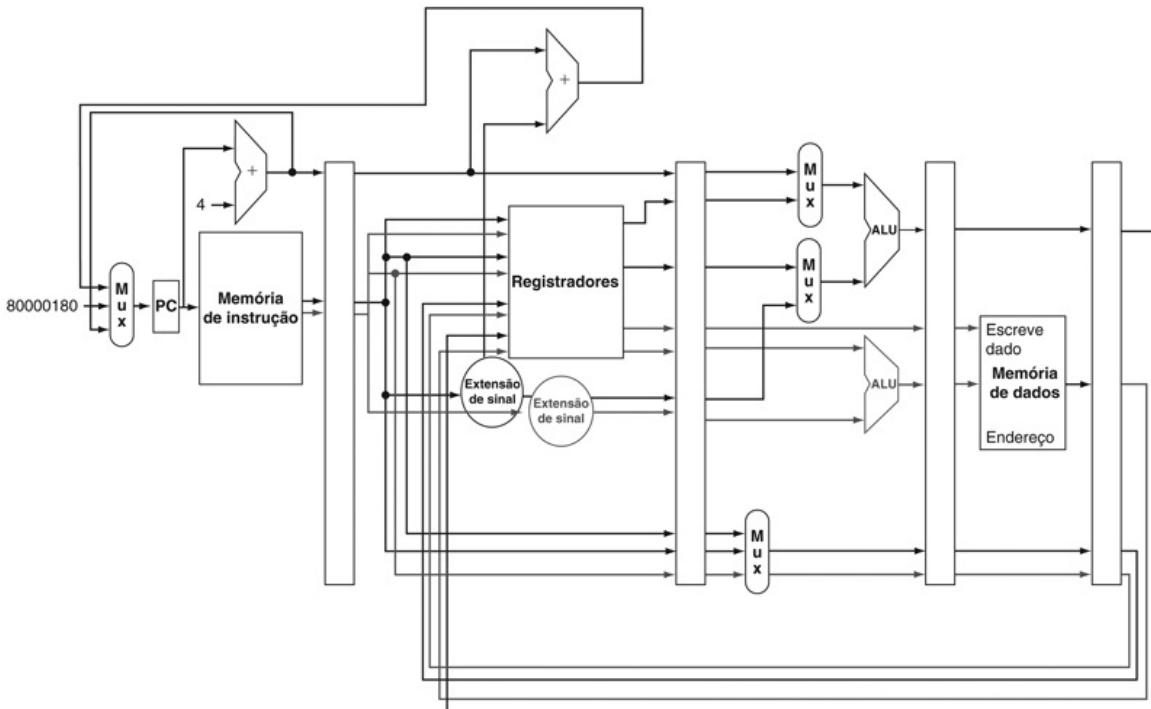


FIGURA 4.69 Um caminho de dados com despacho duplo estático.

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 todo o restante.

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 um 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 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.

latência de uso

Número de ciclos de clock entre uma instrução load e uma instrução que pode usar o resultado do load sem stall do pipeline.

Escalonamento de código simples para despacho múltiplo

Exemplo

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# decremente 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.

Resposta

As três primeiras instruções possuem dependências de dados, bem como as duas últimas. A Figura 4.70 mostra o melhor escalonamento para essas instruções. Observe que apenas um par de instruções possui os dois slots utilizados. São necessários quatro clocks por iteração do loop; com quatro

clocks para executar cinco 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!

	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, 4(\$s1)	4

FIGURA 4.70 O código escalonado conforme apareceria em um pipeline MIPS com despacho duplo.

Os slots vazios são nops.

Uma importante técnica de compilador para conseguir mais desempenho dos loops é o **desdobramento de loop** (loop unrolling), 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.

Desdobramento de loop para pipelines com despacho múltiplo

Exemplo

Veja como o trabalho de desdobramento do loop e escalonamento funciona no exemplo anterior. Para simplificar, suponha que o índice do loop seja um múltiplo de quatro.

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 4.71 mostra o código desdoblado e escalonado.

	Instrução da ALU ou desvio	Instrução de transferência de dados	Ciclo de clock
Loop:	<code>addi \$s1,\$s1,-16</code>	<code>lw \$t0, 0(\$s1)</code>	1
		<code>lw \$t1,12(\$s1)</code>	2
	<code>addu \$t0,\$t0,\$s2</code>	<code>lw \$t2, 8(\$s1)</code>	3
		<code>lw \$t3, 4(\$s1)</code>	4
	<code>addu \$t2,\$t2,\$s2</code>	<code>sw \$t0, 16(\$s1)</code>	5
	<code>addu \$t3,\$t3,\$s2</code>	<code>sw \$t1,12(\$s1)</code>	6
		<code>sw \$t2, 8(\$s1)</code>	7
	<code>bne \$s1,\$zero,Loop</code>	<code>sw \$t3, 4(\$s1)</code>	8

FIGURA 4.71 O código desdoblado e escalonado da Figura 4.70 conforme apareceria no pipeline MIPS com despacho duplo estático.

Os slots vazios são nops. Como a primeira instrução no loop decremente `$s1` em 16, os endereços lidos são o valor original de `$s1`, depois esse endereço menos 4, menos 8 e menos 12.

Durante o processo de desdoblamento, o compilador introduziu registradores adicionais (`$t1`, `$t2`, `$t3`). O objetivo desse processo, chamado **renomeação de registradores**, é eliminar dependências que não são dependências de dados verdadeiras, mas que poderiam levar a hazards em potencial ou impedir que o compilador escalonasse o código de forma flexível. Considere como o código não desdoblado apareceria usando apenas `$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 **antidependência** ou **dependência de nome**, que é uma ordenação forçada puramente pela reutilização de um nome, em vez de uma dependência de dados real, que também é chamada de dependência verdadeira.

Renomear os registradores durante o processo de desdoblamento permite que o compilador move subsequentemente essas instruções independentes, de modo a escalar melhor o código. O processo de renomeação elimina as dependências de nome, enquanto preserva as dependências verdadeiras.

Observe agora que 12 das 14 instruções no loop são executadas como um par. São necessários oito clocks para quatro iterações do loop ou dois clocks por iteração, o que gera um CPI de $8/14 = 0,57$. O desdobramento e o escalonamento do loop com despacho dual nos deram um fator de melhoria de quase 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.

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.

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 escalar 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.

superescalar

Uma técnica de pipelining avançada que permite que o processador execute mais de uma instrução por ciclo de clock selecionando-as durante a execução.

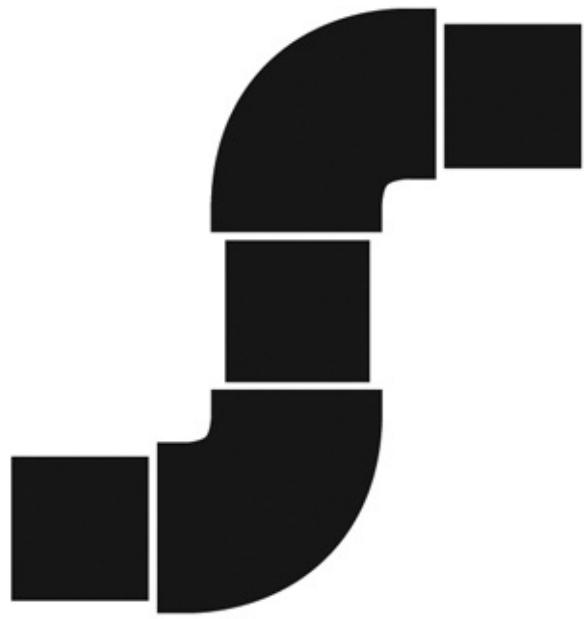
Muitas arquiteturas superescalares estendem a estrutura básica das decisões de despacho dinâmico para incluir **escalonamento dinâmico em pipeline**. O escalonamento dinâmico em pipeline escolhe quais instruções serão executadas em determinado ciclo de clock, enquanto tenta evitar hazards e stalls. Vamos 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

escalonamento dinâmico em pipeline

Suporte do hardware para modificar a ordem de execução das instruções de modo a evitar stalls.

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 5 explica as faltas de cache, 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.



PIPELINING

Escalonamento dinâmico em pipeline

O escalonamento dinâmico em pipeline escolhe quais instruções serão executadas em seguida, possivelmente reordenando-as para evitar stalls. Nestes processadores, o pipeline é dividido em três unidades principais: uma unidade de busca e despacho de instruções, várias unidades funcionais (uma dezena ou mais nos projetos de alto nível em 2013) e uma **unidade de commit**. A [Figura 4.72](#) 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 seção de Detalhamento, 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.

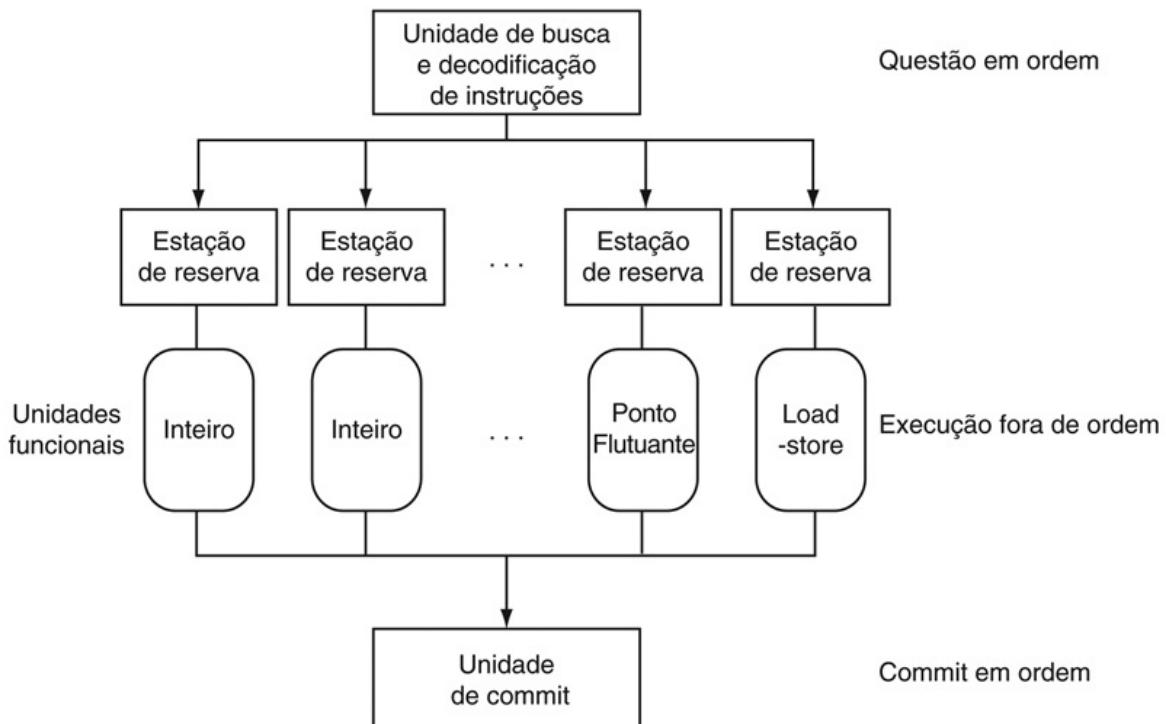


FIGURA 4.72 As três unidades principais de um pipeline escalonado dinamicamente.

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.

A combinação de operandos em buffers nas estações de reserva e os resultados no buffer de reordenação oferecem uma forma de renomeação de registradores, assim como aquela utilizada pelo compilador em nosso exemplo anterior de desdobramento de loop, anteriormente neste capítulo. Para ver como isso funciona conceitualmente, considere as seguintes etapas:

1. Quando uma instrução é despachada, ela é copiada para uma estação de reserva para a unidade funcional apropriada. Quaisquer operandos que estejam disponíveis no banco de registradores ou no buffer de reordenação também serão copiados para a estação de reserva imediatamente. A instrução é mantida em um buffer até que todos os operandos e a unidade funcional 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 esperar para ser produzido por uma unidade funcional. O nome da unidade funcional que produzirá o resultado é rastreado. 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.

Conceitualmente, você pode pensar em um pipeline escalonado de forma dinâmica como uma análise da estrutura de fluxo de dados de um programa. O processador executa as instruções em alguma ordem que preserva a ordem do fluxo de dados do programa. Esse estilo de execução é chamado de **execução fora de ordem**, pois as instruções podem ser executadas em uma ordem diferente daquela em que foram apanhadas.

execução fora de ordem

Uma situação na execução em pipeline quando uma instrução com execução bloqueada não faz com que as instruções seguintes esperem.

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 **commit 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 commit em ordem.

commit em ordem

Um commit em que os resultados da execução em pipeline são escritos no estado visível ao programador na mesma ordem em que as instruções são buscadas.

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 Intel Core i7.

Entendendo o desempenho dos programas

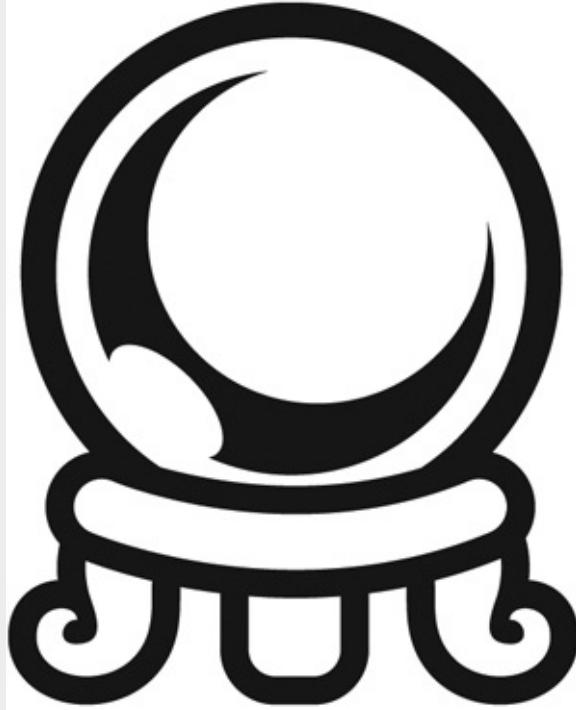
Dado que os compiladores também podem escalonar o código em torno das dependências de dados, você poderia perguntar por que um processador superescalar usaria o escalonamento dinâmico. Existem três motivos principais. Primeiro, nem todos os stalls são previsíveis. Em particular, as

falhas de cache (Capítulo 5) na **hierarquia de memória** causam stalls imprevisíveis. O escalonamento dinâmico permite que o processador oculte alguns desses stalls continuando a executar instruções enquanto esperam que o stall termine.



HIERARQUIA

Segundo, se o processador especula sobre resultados de desvio usando a **predição** 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 o *parallelismo em nível de instrução* (ILP) sem incorporar o escalonamento dinâmico restringiria significativamente os benefícios de tal especulação.



P R E D I Ç Ã O

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, a forma de escalonar uma sequência de instruções dependentes é afetada 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 tentam explorar o **parallelismo** em nível de instrução (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 por meio da **predição**, embora se deva ter cuidado, visto que a especulação incorreta provavelmente reduzirá o desempenho.





P A R A L E L I S M O



P R E D I Ç Ã O

Interface hardware/software

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 na **hierarquia da memória** (o tópico do Capítulo 5) 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.



Eficiência de potência e pipelining avançado

A desvantagem do aumento da exploração do paralelismo em nível de instrução por meio do despacho múltiplo dinâmico e especulação é a eficiência de potência. Cada inovação foi capaz de transformar mais transistores em desempenho, mas geralmente eles faziam isso de modo muito ineficaz. Agora que atingimos o muro da potência, estamos vendo projetos com múltiplos processadores por chip em que os processadores não são tão profundamente dispostos em pipeline ou tão agressivamente especulativos quanto seus predecessores.

A crença é que, embora os processadores mais simples não sejam tão rápidos quanto seus irmãos sofisticados, eles oferecem melhor desempenho por watt, de modo que podem oferecer mais desempenho por chip quando os projetos são restritos mais por potência do que por número de transistores.

A [Figura 4.73](#) mostra o número de estágios de pipeline, largura do despacho, nível de especulação, taxa de clock, núcleos por chip e potência de vários microprocessadores do passado e recentes. Observe a queda nos estágios de pipeline e potência enquanto as empresas passam para projetos multicore.

Microprocesso	Ano	Taxa de clock	Estágios de pipeline	Largura do despacho	Fora de ordem/especulação	Núcleos/chip	Potência	
Intel 486	1989	25 MHz	5	1	Não	1	5	W
Intel Pentium	1993	66 MHz	5	2	Não	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Sim	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Sim	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Sim	1	103	W
Intel Core	2006	2930 MHz	14	4	Sim	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Sim	1	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Sim	8	77	W

FIGURA 4.73 Registro dos microprocessadores Intel em termos de complexidade de pipeline, número de cores (núcleos) e potência.

Os estágios de pipeline do Pentium 4 não incluem os estágios de commit. Se os incluíssemos, os pipelines do Pentium 4 seriam ainda mais profundos.

Detalhamento

Uma unidade de commit controla 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 mais antiga de um registrador até que a instrução atualizando o registrador não seja mais especulativa. Outros processadores mantêm o resultado em buffer, normalmente em uma estrutura chamada buffer de reordenação e a atualização real no banco de registradores ocorre depois, como parte do commit. Stores na memória precisam ser colocados em buffer até o momento do commit, seja em um *buffer de store* (Capítulo 5) ou no buffer de reordenação. A unidade de commit permite que o store escreva na memória, a partir do buffer, quando ele tiver um endereço com dados válidos e quando o store não for mais dependente de desvios previstos.

Detalhamento

Os acessos à memória se beneficiam das *caches sem bloqueio*, que continuam a atender acessos da cache durante uma falta de cache (Capítulo 5). Os processadores com execução fora de ordem precisam do projeto de cache para permitir que as instruções sejam executadas durante uma falha.

Verifique você mesmo

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”.

1. Previsão de desvio
2. Despacho múltiplo
3. VLIW
4. Superescalar
5. Escalonamento dinâmico
6. Execução fora de ordem
7. Especulação
8. Buffer de reordenação
9. Renomeação de registradores

4.11. Vida real: pipelines do ARM Cortex-A8 e Intel Core i7

A Figura 4.74 descreve os dois microprocessadores que examinaremos nesta seção, cujos destinos são os dois exemplos típicos da era pós-PC.

Processador	ARM A8	Intel Core i7 920
Mercado	Dispositivo móvel pessoal	Servidor, Nuvem
Potência do projeto térmico	2 Watts	130 Watts
Taxa de clock	1 GHz	2.66 GHz
Núcleos/chip	1	4
Ponto flutuante?	Não	Sim
Despacho múltiplo?	Dinâmico	Dinâmico
Pico de instruções/ciclo de clock	2	4
Estágios do pipeline	14	14
Escalonamento de pipeline	Estático em ordem	Dinâmico fora de ordem com especulação
Predição de desvio	2-níveis	2-níveis
Caches de 1º nível/núcleo	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
Cache de 2º nível/núcleo	128–1024 KiB	256 KiB
Cache de 3º nível (compartilhado)	–	2–8 MiB

FIGURA 4.74 Especificação do ARM Cortex-A8 e do Intel

O ARM Cortex-A8

O ARM Cortex-A8 roda a 1 GHz com um pipeline de 14 estágios. Ele usa o despacho múltiplo dinâmico, com duas instruções por ciclo de clock. Ele é um pipeline estático em ordem, no qual as instruções são despachadas, executadas e confirmadas ordenadamente. O pipeline consiste em três seções para busca de instruções, decodificação de instruções e execução. A [Figura 4.75](#) mostra o pipeline geral.

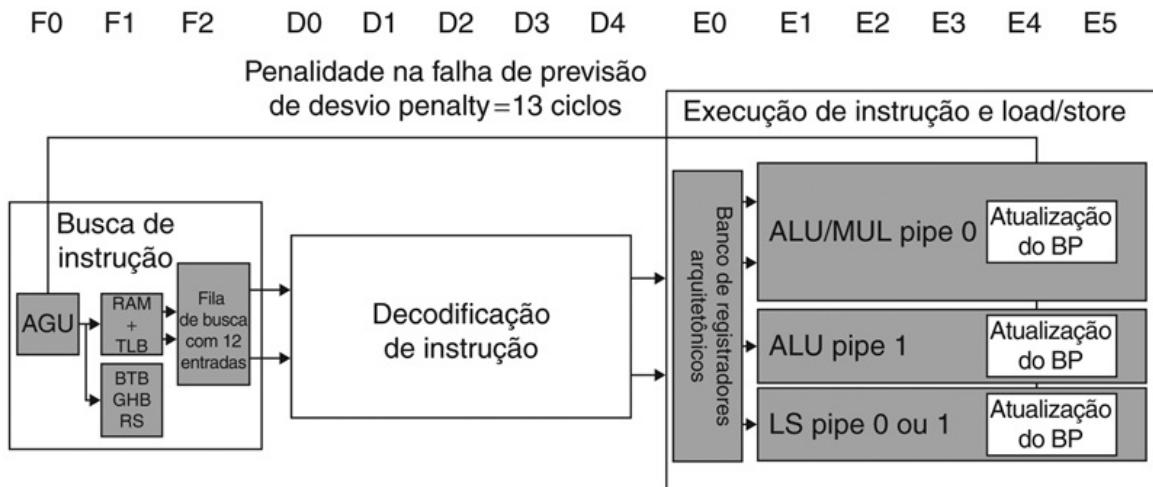


FIGURA 4.75 O pipeline do A8.

Os três primeiros estágios leem instruções para um buffer de busca de instrução com 12 entradas. A *unidade de geração de endereço* (AGU — Address Generation Unit) usa um *buffer de destino de desvio* (BTB — Branch Target Buffer), *buffer de histórico global* (GHB — Global History Buffer) e *pilha de retorno* (RS — Return Stack) para prever desvios a fim de tentar manter cheia a fila de busca. A decodificação de instruções tem cinco estágios e a execução de instruções tem seis estágios.

Os três primeiros estágios buscam duas instruções de uma só vez e tentam manter cheio um buffer de pré-busca com 12 instruções. Ele usa um previsor de desvio de dois níveis usando um buffer de destino de 512 entradas, um buffer de histórico global de 4096 entradas e uma pilha de retorno de 8 entradas. Quando a previsão de desvio é errada, ele esvazia o pipeline, resultando em uma

penalidade na falha de previsão de desvio com 13 ciclos de clock.

Os cinco estágios do pipeline de decodificação determinam se existem dependências entre um par de instruções, o que forçaria a execução sequencial, e em qual pipeline dos estágios de execução as instruções são enviadas.

Os seis estágios da seção de execução de instrução oferecem um pipeline para instruções load e store e dois pipelines para operações aritméticas, embora somente o primeiro do par possa lidar com multiplicações. Qualquer instrução do par pode ser despachada ao pipeline de load-store. Os estágios de execução possuem bypassing pleno entre os três pipelines.

A Figura 4.76 mostra o CPI do A8 usando pequenas versões de programas derivados dos benchmarks SPEC2000. Embora o CPI ideal seja 0,5, o melhor caso é 1,4, o caso mediano é 2,0 e o pior caso é 5,2. Para o caso mediano, 80% dos stalls devem-se aos hazards de pipelining e 20% são stalls devidos à hierarquia de memória. Os stalls do pipeline são causados por falhas de previsão de desvio, hazards estruturais e dependências de dados entre pares de instruções. Devido ao pipeline estático do A8, fica a critério do compilador tentar evitar hazards estruturais e dependências de dados.

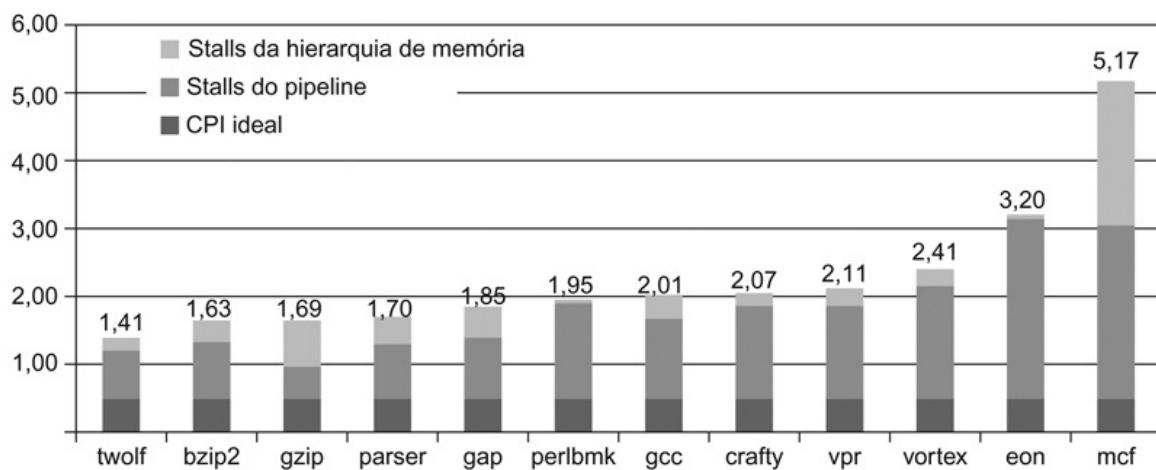


FIGURA 4.76 CPI no ARM Cortex A8 para os benchmarks Minnespec, que são pequenas versões dos benchmarks SPEC2000.

Estes benchmarks usam as entradas muito menores para reduzir o tempo de execução em várias ordens de grandeza. O tamanho menor subestima significativamente o impacto do CPI da hierarquia de memória ([Capítulo 5](#)).

Detalhamento

O Cortex-A8 é um núcleo (core) configurável que tem suporte para a arquitetura do conjunto de instrução ARMv7. Ele é entregue como um *núcleo de propriedade intelectual (IP — Intellectual Property)*. Núcleos IP são a forma dominante de entrega de tecnologia nos mercados de dispositivos móveis embutidos, pessoais e relacionados; bilhões de processadores ARM e MIPS foram criados a partir desses núcleos IP.

Observe que os núcleos IP são diferentes dos núcleos nos computadores Intel i7 multicore. Um núcleo IP (que pode, por si só, ser um multicore) é projetado para ser incorporado com outra lógica (daí ele ser um “núcleo” de um chip), incluindo processadores específicos da aplicação (como um codificador ou decodificador para vídeo), interfaces de E/S e interfaces de memória, e depois fabricado para produzir um processador otimizado para uma aplicação em particular. Embora o núcleo do processador seja quase idêntico, os chips resultantes possuem muitas diferenças. Um parâmetro é o tamanho da cache L2, que pode variar por um fator de oito.

O Intel Core i7 920

Os microprocessadores x86 empregam técnicas sofisticadas de pipelining, usando o despacho múltiplo dinâmico e o escoamento de pipeline dinâmico com execução fora de ordem e especulação para o seu pipeline de 14 estágios. Porém, esses processadores ainda enfrentam o desafio de implementar o complexo conjunto de instruções do x86, descrito no [Capítulo 2](#). O processador Intel busca instruções x86 e as traduz em instruções internas tipo MIPS, que a Intel chama de *micro-operações*. As micro-operações são então executadas por um pipeline sofisticado, especulativo e dinamicamente escalonado, capaz de sustentar a taxa de execução de até seis micro-operações por ciclo de clock. Esta seção é focada nesse pipeline de micro-operações.

Quando consideramos o projeto de processadores sofisticados, escalonados dinamicamente, o projeto das unidades funcionais, da cache e banco de registradores, do despacho de instruções e do controle geral do pipeline tornam-se algo combinado, dificultando a separação entre o caminho de dados e o pipeline. Por causa disso, muitos engenheiros e pesquisadores têm adotado o termo **microarquitetura** para se referirem à arquitetura interna detalhada de um processador.

microarquitetura

A organização do processador, incluindo as principais unidades funcionais, sua interconexão e controle.

O Intel Core i7 utiliza um esquema para resolver as antidependências e a especulação incorreta, que usa um buffer de reordenação junto com a renomeação de registradores. A renomeação de registradores renomeia explicitamente os **registradores arquitetônicos** em um processador (16 no caso da versão de 64 bits da arquitetura x86) para um conjunto maior de registradores físicos. O Core i7 utiliza a renomeação de registradores para remover as antidependências. A renomeação de registradores exige que o processador mantenha um mapa entre os registradores arquitetônicos e os registradores físicos, indicando qual registrador físico é a cópia mais atual de um registrador arquitetônico. Registrando as renomeações que ocorreram, a técnica oferece outra forma de 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 arquitetônicos e físicos.

registradores arquitetônicos

O conjunto de instruções dos registradores visíveis de um processador; por exemplo, no MIPS, existem 32 registradores de inteiros e 32 de ponto flutuante.

A [Figura 4.77](#) mostra a organização geral e o pipeline do Core i7. A seguir estão as oito etapas pelas quais uma instrução x86 passa para a sua execução.

1. Busca de instrução — O processador utiliza um buffer de destino de desvio multinível para obter um equilíbrio entre velocidade e exatidão na previsão. Há também uma pilha de endereços de retorno para agilizar o retorno de função. Falhas de previsão causam uma penalidade de aproximadamente 15 ciclos. Usando o endereço previsto, a unidade de busca de instruções lê 16 bytes da cache de instruções.

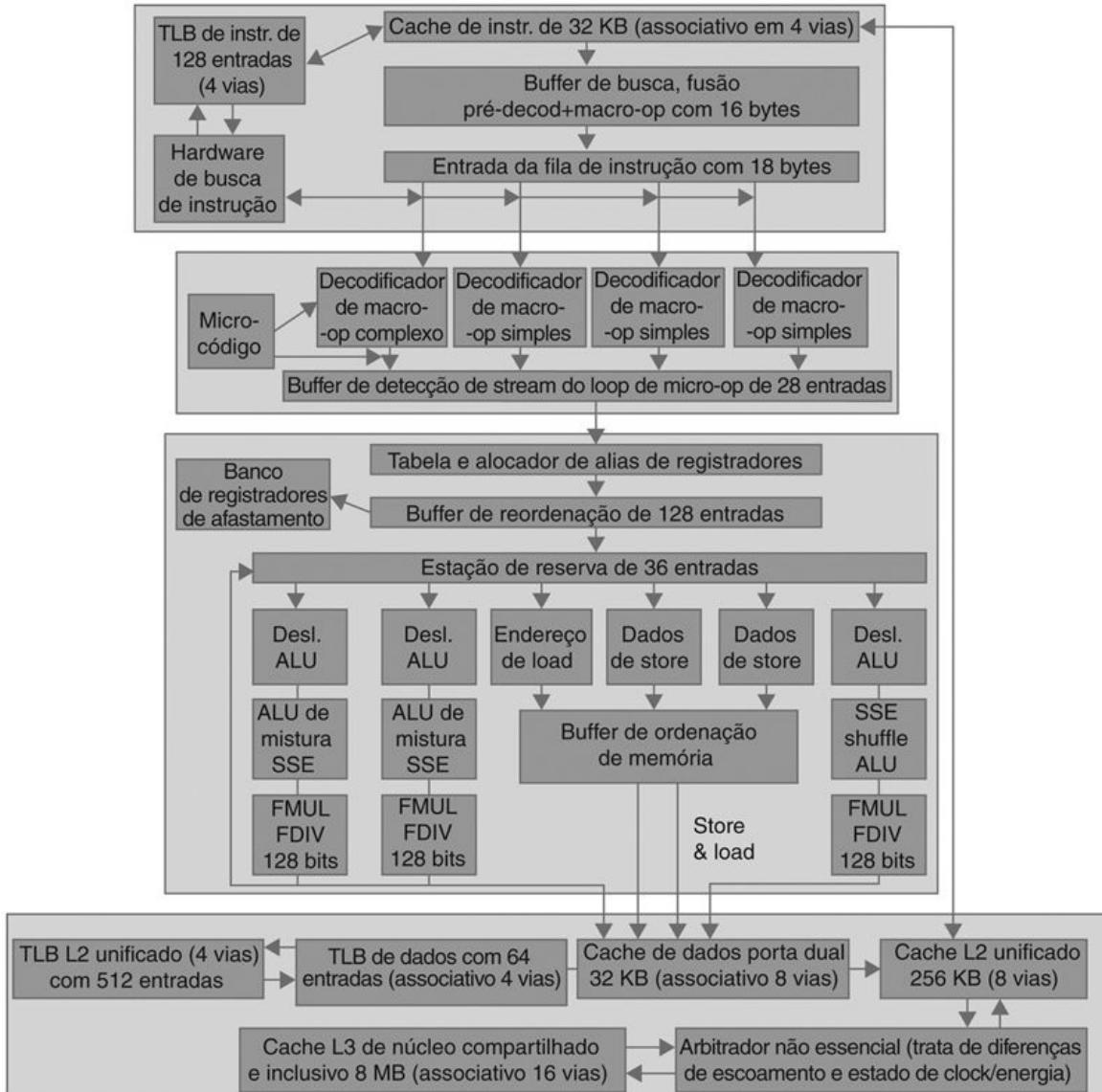


FIGURA 4.77 O pipeline do Core i7 com componentes de memória.

A profundidade total do pipeline é de 14 estágios, com as falhas de previsão de desvio custando 17 ciclos de clock. Esse projeto pode manter em buffer 48 loads e 32 stores. As seis unidades independentes podem iniciar a execução de uma operação RISC pronta a cada ciclo de clock.

2. Os 16 bytes são colocados no buffer de instrução pré-decodificação — O estágio de pré-decodificação transforma os 16 bytes em instruções x86 individuais. Essa pré-decodificação é não trivial, pois o comprimento de uma instrução x86 pode ser de 1 a 15 bytes, e o pré-decodificador precisa percorrer uma série de bytes, antes de descobrir o comprimento da

instrução. As instruções x86 individuais são colocadas na filha de instruções com 18 entradas.

3. Decodificação de micro-operação — As instruções x86 individuais são traduzidas em micro-operações (micro-ops). Três dos decodificadores tratam das instruções x86 que se traduzem diretamente em uma micro-op. Para instruções x86 que possuem semântica mais complexa, existe um mecanismo de microcódigo usado para produzir a sequência de micro-op; ele pode produzir até quatro micro-ops a cada ciclo e continua até que a sequência de micro-ops necessária tenha sido gerada. As micro-ops são posicionadas de acordo com a ordem das instruções x86 no buffer de micro-ops com 28 entradas.
4. O buffer de micro-op realiza a *detecção de stream do loop* — Se houver uma pequena sequência de instruções (menos de 28 instruções ou 256 bytes de extensão) que compreende um loop, o detector de stream do loop encontrará o loop e despachará diretamente as micro-ops a partir do buffer, eliminando a necessidade de ativação dos estágios de busca e decodificação de instruções.
5. Realizar o despacho básico da instrução — Pesquisar o local do registrador nas tabelas de registradores, renomear os registradores, alocar uma entrada no buffer de reordenação e buscar quaisquer resultados dos registradores ou buffer de reordenação antes de enviar as micro-ops para as estações de reserva.
6. O i7 usa uma estação de reserva centralizada com 36 entradas, compartilhada por seis unidades funcionais. Até seis micro-ops podem ser despachadas para as unidades funcionais a cada ciclo de clock.
7. As unidades funcionais individuais executam micro-ops e depois os resultados são enviados de volta a qualquer estação de reserva aguardando, bem como para a unidade de afastamento de registrador, onde atualizarão o estado do registrador, quando se souber que a instrução não é mais especulativa. A entrada correspondente à instrução no buffer de reordenação é marcada como completa.
8. Quando uma ou mais instruções no início do buffer de reordenação forem marcadas como completas, as escritas pendentes na unidade de afastamento de registrador são executadas, e as instruções são removidas do buffer de reordenação.

Detalhamento

O hardware na segunda e quarta etapas pode combinar ou *fundir* operações para reduzir o número de operações que precisam ser realizadas. A *fusão de macro-op* na segunda etapa exige que se combine instruções x86, como uma comparação seguida de um desvio, e que se junte em uma única operação. A *microfusão* na quarta etapa combina pares de micro-operações, como load/operação ALU e operação ALU/store, e os despacha para uma única estação de reserva (onde ainda poderão ser despachados independentemente), aumentando assim a utilização do buffer. Em um estudo da arquitetura Intel Core, que também incorporou a microfusão e a macrofusão, Bird et al. (2007) descobriram que a microfusão tinha pouco impacto sobre o desempenho, enquanto a macrofusão parece ter um impacto positivo moderado sobre o desempenho de inteiros e pouco impacto sobre o desempenho de ponto flutuante.

Desempenho do Intel Core i7 920

A Figura 4.78 mostra o CPI do Intel Core i7 para cada um dos benchmarks SPEC2006. Embora o CPI ideal seja 0,25, o melhor caso é 0,44, o caso mediano é 0,79 e o pior caso é 2,67.

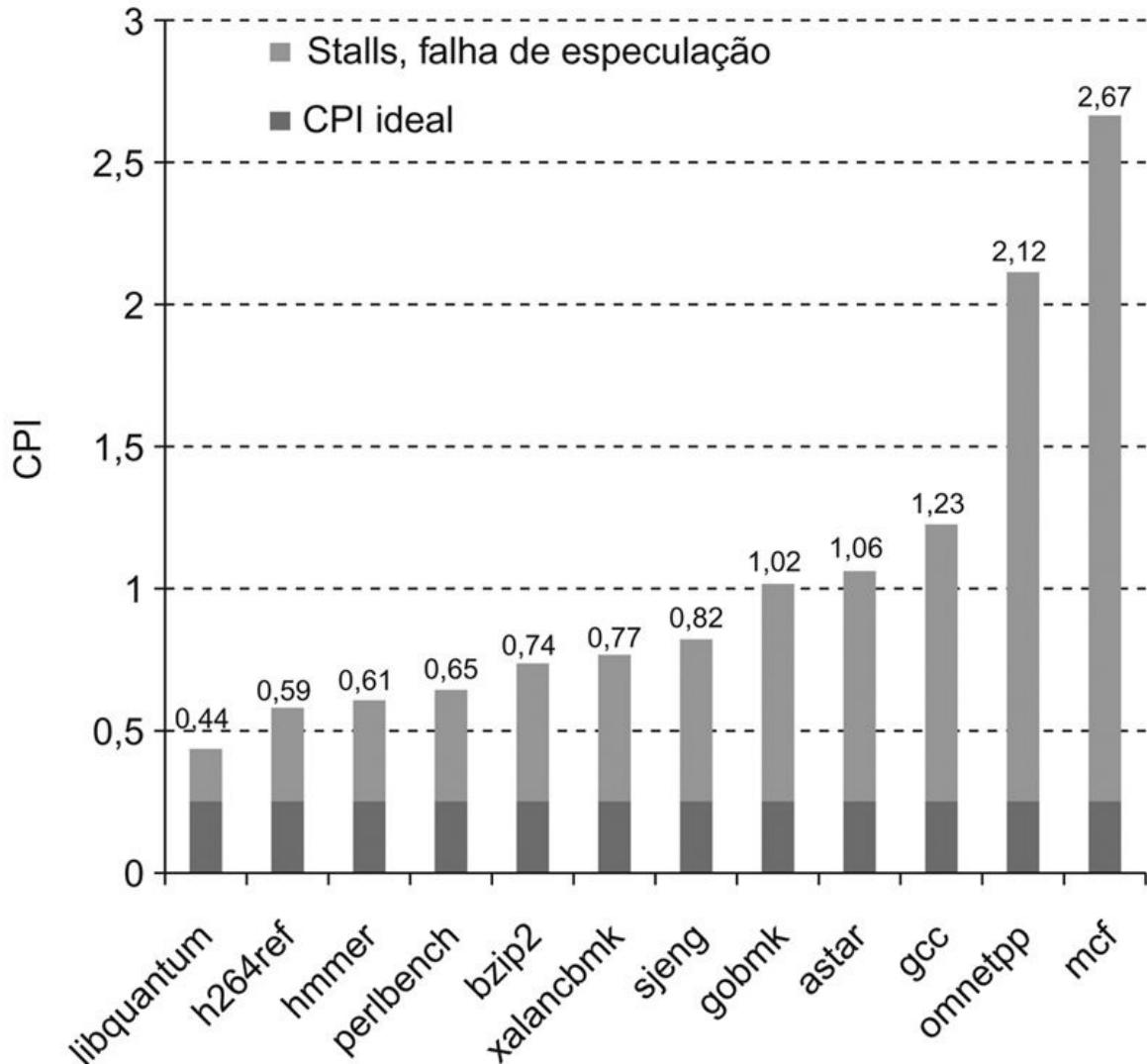


FIGURA 4.78 CPI do Intel Core i7 920 rodando benchmarks de inteiros do SPEC2006.

Embora seja difícil diferenciar entre stalls do pipeline e stalls da memória em um pipeline de execução dinâmica fora de ordem, podemos mostrar a eficácia da previsão de desvio e da especulação. A Figura 4.79 mostra a porcentagem dos desvios mal previstos e a porcentagem do trabalho (medida pelo número de micro-ops despachadas para o pipeline) que não se ausenta (ou seja, seus resultados são anulados) em relação a todos os despachos de micro-op. O mínimo, mediano e máximo das falhas de previsão de desvio são 0%, 2% e 10%. Para o trabalho desperdiçado, eles são 1%, 18% e 39%.

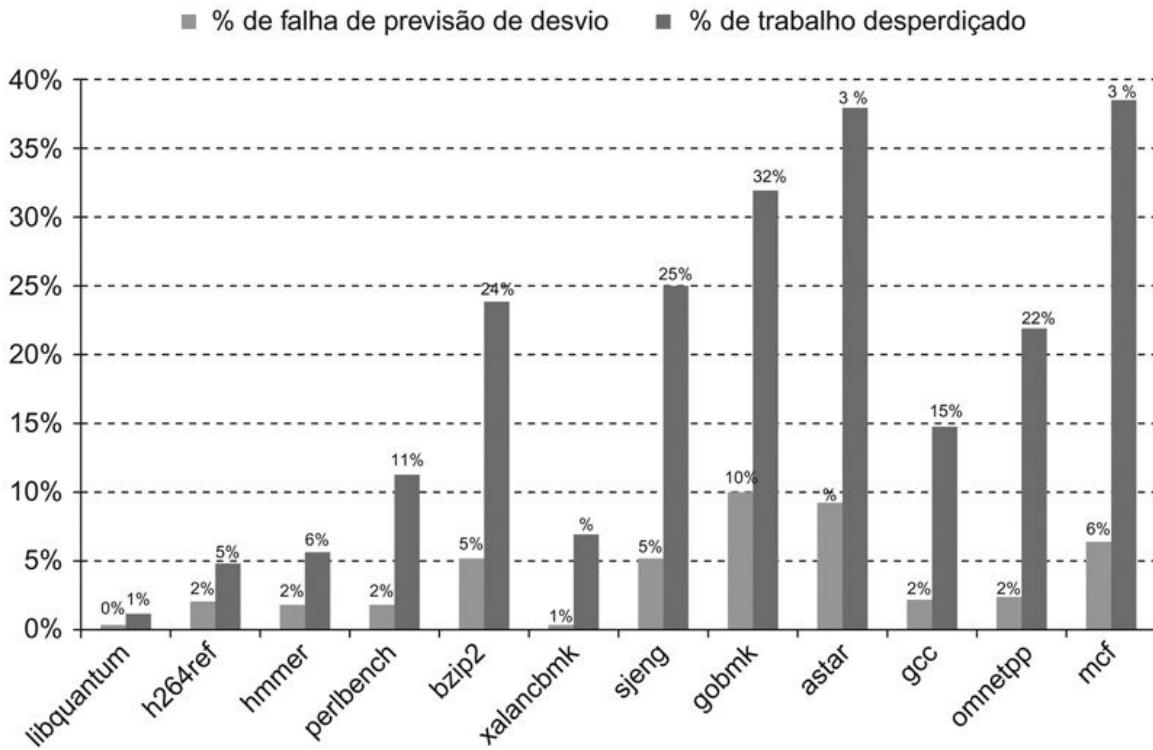


FIGURA 4.79 Porcentagem de erros de previsão de desvio e trabalho desperdiçado devido à especulação improdutiva do Intel Core i7 920 rodando benchmarks de inteiros do SPEC2006.

O trabalho desperdiçado, em alguns casos, corresponde de perto às taxas de falha na previsão de desvio, como para os benchmarks gobmk e astar. Em vários casos, como em mcf, o trabalho desperdiçado parece ser relativamente maior do que a taxa de falha de previsão. Essa divergência provavelmente se deve ao comportamento da memória. Com taxas de falta de cache de dados muito altas, mcf despachará muitas instruções durante uma especulação incorreta, desde que haja estações de reserva suficientes à disposição para as referências de memória adiadas. Quando um desvio entre as muitas instruções especuladas for finalmente mal previsto, as micro-ops correspondentes a todas essas instruções sofrerão flush.

Entendendo o desempenho dos programas

O Intel Core i7 combina um pipeline de 14 estágios e despacho múltiplo agressivo para conseguir alto desempenho. Mantendo baixas as latências para operações back to back, 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 x86 que não são mapeadas para algumas micro-operações simples.
- Desvios que são difíceis de se prever, causando stalls e reinícios mal previstos quando a especulação falha.
- Dependências longas — normalmente causadas por instruções duradouras ou pela **hierarquia de memória** — que causam stalls.
- Atrasos de desempenho que surgem no acesso à memória (Capítulo 5), fazendo com que o processador sofra stall.



4.12. Mais rápido: Paralelismo em nível de instrução e multiplicação matricial

Retornando ao exemplo do DGEMM do Capítulo 3, podemos ver o impacto do paralelismo em nível de instrução desdobrando o loop de modo que o processador com execução de despacho múltiplo, fora de ordem, tenha mais

instruções com que trabalhar. A [Figura 4.80](#) mostra a versão desdobrada da [Figura 3.23](#), que contém os intrínsecos da linguagem C para produzir as instruções AVX.

```
1 #include <x86intrin.h>
2 #define UNROLL (4)
3
4 void dgemm (int n, double* A, double* B, double* C)
5 {
6     for ( int i = 0; i < n; i+=UNROLL*4 )
7         for ( int j = 0; j < n; j++ ) {
8             __m256d c[4];
9             for ( int x = 0; x < UNROLL; x++ )
10                 c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12             for( int k = 0; k < n; k++ )
13             {
14                 __m256d b = _mm256_broadcast_sd(B+k+j*n);
15                 for (int x = 0; x < UNROLL; x++)
16                     c[x] = _mm256_add_pd(c[x],
17                                         _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
18             }
19
20             for ( int x = 0; x < UNROLL; x++ )
21                 _mm256_store_pd(C+i+x*4+j*n, c[x]);
22         }
23 }
```

FIGURA 4.80 Versão C otimizada do DGEMM usando intrínsecos da linguagem C para gerar as instruções paralelas de subword AVX para o x86 ([Figura 3.23](#)) e desdobramento de loop para criar mais oportunidades de paralelismo em nível de instrução.

A [Figura 4.81](#) mostra o código em linguagem assembly produzido pelo compilador para o loop mais interno, que desdobra os três corpos de loop for a fim de expor o paralelismo em nível de instrução.

Assim como o exemplo de desdobramento na [Figura 4.71](#), vamos desdobrar o loop 4 vezes. (Usamos a constante UNROLL no código C para controlar a quantidade de desdobramento caso queiramos experimentar outros valores.) Em vez de desdobrar manualmente o loop em C fazendo 4 cópias de cada um dos intrínsecos da [Figura 3.23](#), podemos contar com o compilador gcc para fazer o desdobramento na otimização -O3. Delimitamos cada intrínseco com um loop

for simples com 4 iterações (linhas 9, 14 e 20) e substituímos o escalar `c0` da [Figura 3.23](#) por um array de 4 elementos `c[]` (linhas 8, 10, 16 e 21).

A [Figura 4.81](#) mostra a saída em linguagem assembly do código desdobrado. Conforme esperado, na [Figura 4.81](#), existem 4 versões de cada uma das instruções AVX da [Figura 3.24](#), com uma exceção. Só precisamos de uma cópia da instrução `vbroadcastsd`, pois podemos usar as quatro cópias do elemento B no registrador `%ymm0` repetidamente por todo o loop. Assim, as 5 instruções AVX da [Figura 3.24](#) tornam-se 17 na [Figura 4.81](#), e as 7 instruções de inteiros aparecem em ambas, embora as constantes e o endereçamento mudem para levar em conta o desdobramento. Portanto, apesar de desdobrar 4 vezes, o número de instruções no corpo do loop só dobra: de 12 para 24.

```

1  vmovapd (%r11),%ymm4          # Lê 4 elementos de C para %ymm4
2  mov    %rbx,%rax              # Registrador %rax = %rbx
3  xor    %ecx,%ecx              # Registrador %ecx = 0
4  vmovapd 0x20(%r11),%ymm3      # Lê 4 elementos de C para %ymm3
5  vmovapd 0x40(%r11),%ymm2      # Lê 4 elementos de C para %ymm2
6  vmovapd 0x60(%r11),%ymm1      # Lê 4 elementos de C para %ymm1
7  vbroadcastsd (%rcx,%r9,1),%ymm0  # Faz 4 cópias do elemento B
8  add    $0x8,%rcx              # Registrador %rcx = %rcx + 8
9  vmulpd (%rax),%ymm0,%ymm5    # Mul paralelo de %ymm1,4 elementos A
10 vaddpd %ymm5,%ymm4,%ymm4     # Add paralelo de %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5 # Mul paralelo de %ymm1,4 elementos A
12 vaddpd %ymm5,%ymm3,%ymm3     # Add paralelo de %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5 # Mul paralelo de %ymm1,4 elementos A
14 vmulpd 0x60(%rax),%ymm0,%ymm0 # Mul paralelo de %ymm1,4 elementos A
15 add    %r8,%rax              # Registrador %rax = %rax + %r8
16 cmp    %r10,%rcx              # Compara %r8 com %rax
17 vaddpd %ymm5,%ymm2,%ymm2     # Add paralelo de %ymm5, %ymm2
18 vaddpd %ymm0,%ymm1,%ymm1     # Add paralelo de %ymm0, %ymm1
19 jne    68 <dgemm+0x68>       # Salta se não %r8 != %rax
20 add    $0x1,%esi              # Registrador %esi = %esi + 1
21 vmovapd %ymm4,(%r11)         # Salva %ymm4 em 4 elementos C
22 vmovapd %ymm3,0x20(%r11)     # Salva %ymm3 em 4 elementos C
23 vmovapd %ymm2,0x40(%r11)     # Salva %ymm2 em 4 elementos C
24 vmovapd %ymm1,0x60(%r11)     # Salva %ymm1 em 4 elementos C

```

FIGURA 4.81 A linguagem assembly x86 para o corpo dos

loops aninhados gerados pela compilação do código C desdobrado na [Figura 4.80](#).

A [Figura 4.82](#) mostra o aumento de desempenho do DGEMM para matrizes 32×32 passando de não otimizado para AVX e depois para AVX com desdobramento. O desdobramento mais do que duplica o desempenho, passando de 6,4 GFLOPS para 14,6 GFLOPS. As otimizações para **paralelismo de subword** e **paralelismo em nível de instrução** resultam em um ganho de velocidade geral de 8,8 *versus* o DGEMM não otimizado da [Figura 3.21](#).

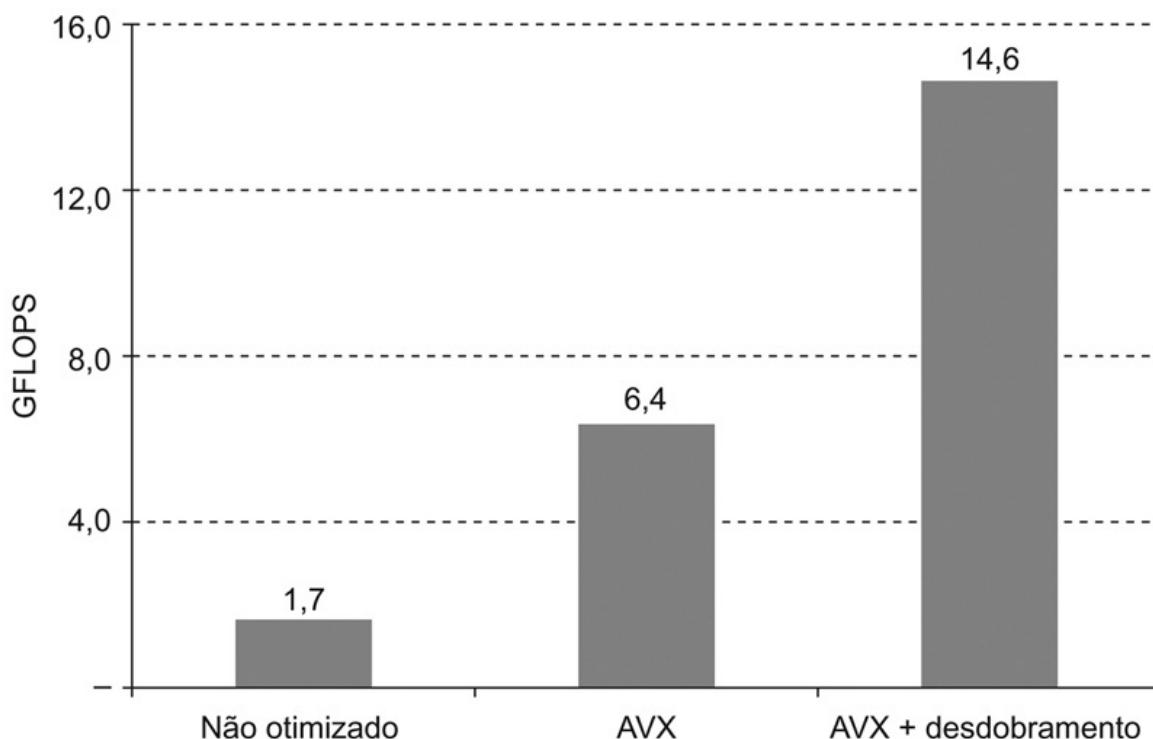


FIGURA 4.82 Desempenho de três versões do DGEMM para matrizes 32×32 .

O paralelismo de subword e o paralelismo em nível de instrução levaram a um ganho de velocidade de quase 9 em relação ao código não otimizado da [Figura 3.21](#).

Detalhamento

Como dissemos no Detalhamento da Seção 3.8, esses resultados são com o modo Turbo desativado. Se o ativarmos, como no Capítulo 3, melhoramos

todos os resultados pelo aumento temporário na taxa de clock de $3,3/2,6 = 1,27$, passando a 2,1 GFLOPS para o DGEMM não otimizado, 8,1 GFLOPS com AVX e 18,6 GFLOPS com desdobramento e AVX. Como dissemos na Seção 3.8, o modo Turbo funciona particularmente bem nesse caso, porque está usando apenas um único núcleo de um chip de oito núcleos.

Detalhamento

Não existem stalls de pipeline apesar da reutilização do registrador `%ymm5` nas linhas de 9 a 17 da Figura 4.81, pois o pipeline do Intel Core i7 renomeia os registradores.

Verifique você mesmo

Indique se cada uma das afirmações a seguir é verdadeira ou falsa.

1. O Intel Core i7 utiliza um pipeline com despacho múltiplo para executar as instruções X86 diretamente.
2. Tanto o A8 quanto o Core i7 utilizam o despacho múltiplo dinâmico.
3. A microarquitetura do Core i7 possui muito mais registradores do que o x86 exige.
4. O Intel Core i7 usa menos de metade dos estágios do pipeline do Intel Pentium 4 Prescott mais antigo (Figura 4.73).

4.13. Falácia e armadilhas

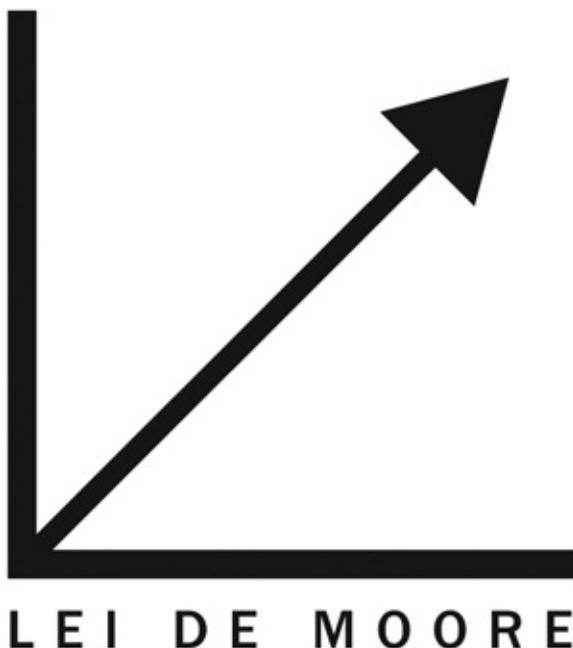
Falácia: Pipelining é fácil.

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 Intel Core i7 terá milhares de linhas é uma indicação da complexidade. Esteja atento!

Falácia: As ideias de pipelining podem ser implementadas independentes da

tecnologia.

Quando o número de transistores no chip e a velocidade dos transistores tornaram um pipeline de cinco estágios a melhor solução, então o delayed branch (veja o primeiro “*Detalhamento*” da Seção “Previsão dinâmica de desvios”) foi uma solução simples para controlar os hazards. Com pipelines maiores, a execução superescalar e a previsão dinâmica de desvios, isso agora 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, devido à **Lei de Moore**, a lógica se tornava muito mais rápida do que a memória, então as múltiplas unidades funcionais e os pipelines dinâmicos fizeram mais sentido. Hoje, a preocupação com a potência está levando a projetos menos agressivos.



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 micro-operações e micropipeline que o Intel Core i7 emprega hoje. Naturalmente, o overhead da tradução e a manutenção da correspondência entre as micro-operações e as instruções permanecem.
- Modos de endereçamento sofisticados podem levar a diferentes tipos de problemas. Os modos de endereçamento que atualizam registradores 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.

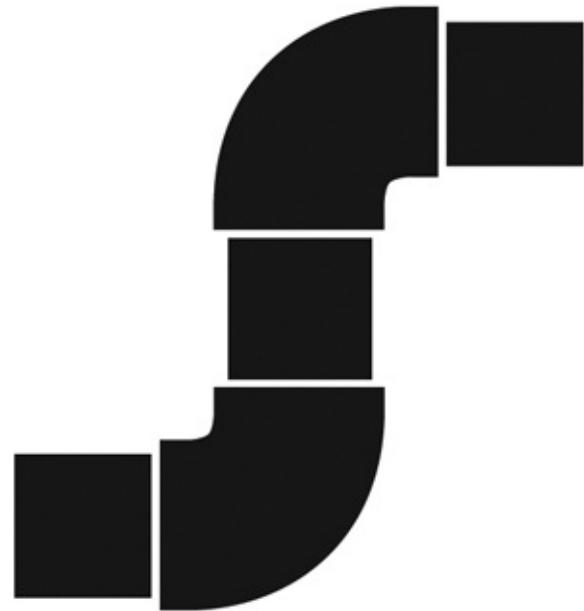
4.14. Comentários finais

Noventa por cento da sabedoria consiste em ser sensato no tempo.

Provérbio americano

Como vimos neste capítulo, tanto o caminho de dados quanto o controle para um processador podem ser projetados começando com a arquitetura do conjunto de instruções e o conhecimento das características básicas da tecnologia. Na [Seção 4.3](#), vimos como o caminho de dados para um processador MIPS poderia ser construído com base na arquitetura e na decisão de criar uma implementação de ciclo único. Naturalmente, a tecnologia básica também afeta muitas decisões de projeto, ditando quais componentes podem ser usados no caminho de dados e também se uma implementação de ciclo único sequer faz sentido.

A técnica de **pipelining** melhora a vazão, mas não o tempo de execução inerente (ou **latência de instrução**) das instruções; para algumas instruções, a latência é semelhante, em duração, à técnica de ciclo único. O despacho de instrução 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. O pipelining foi apresentado como capaz de reduzir o tempo de ciclo de clock do caminho de dados do ciclo único simples. O despacho múltiplo de instruções, em comparação, focaliza claramente na redução dos *ciclos de clock por instrução* (CPI).



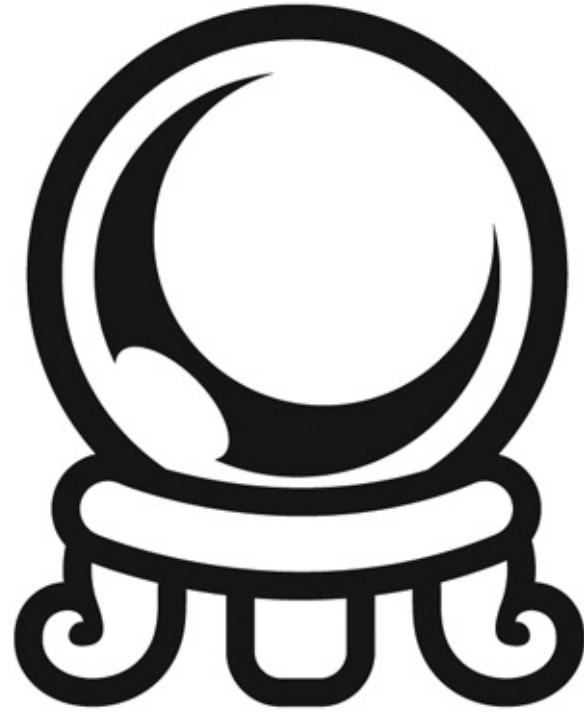
PIPELINING

latência de instrução

O tempo de execução inerente para uma instrução.

A técnica de pipelining e o despacho múltiplo tentam explorar o paralelismo em nível de instrução. A presença de dependências de dados e de controle, que podem se tornar hazards, são as principais limitações para a exploração do paralelismo. Escalonamento e especulação por **predição**, tanto no hardware

quanto no software, são as principais técnicas utilizadas para reduzir o impacto das dependências sobre o desempenho.



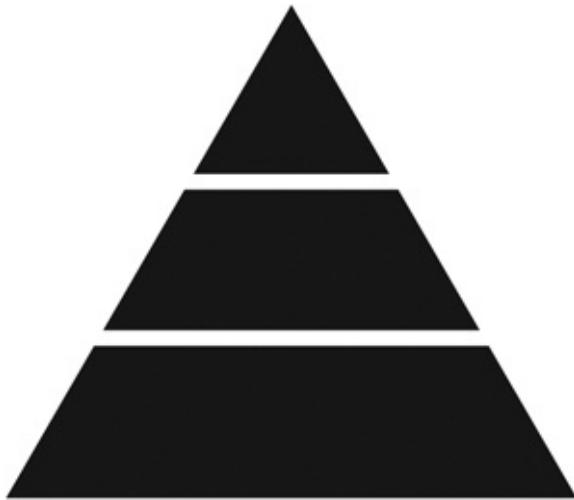
P R E D I Ç Ã O

Mostramos que o desdobramento do loop DGEMM expõe quatro vezese mais instruções que poderiam tirar proveito do mecanismo de execução fora de ordem do Core i7 para mais do que dobrar 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 anual de desempenho dos processadores que começou no início da década de 1980. Como dissemos no [Capítulo 1](#), esses microprocessadores preservaram o modelo de programação sequencial, mas por fim se chocaram com o muro da potência. Assim, a indústria foi forçada a testar multiprocessadores, que exploraram o paralelismo em níveis menos minuciosos (o assunto do [Capítulo 6](#)). Essa tendência também fez com que os projetistas reavaliassem as implicações de desempenho da potência de algumas invenções desde meados da década de 1990, resultando em uma simplificação dos pipelines

em versões mais recentes das microarquiteturas.

Para sustentar os avanços no desempenho de processamento por meio de processadores paralelos, a lei de Amdahl sugere que outra parte do sistema se torne o gargalo. Esse gargalo é o assunto do próximo capítulo: a **hierarquia da memória**.



HIERARQUIA

4.15. Exercícios

4.1 Considere a seguinte instrução:

Instrução: AND Rd, Rs, Rt

Interpretação: $\text{Reg}[\text{Rd}] = \text{Reg}[\text{Rs}] \text{ AND } \text{Reg}[\text{Rt}]$

4.1.1 [5] <§4.1> Quais são os valores dos sinais de controle gerados pelo controle na [Figura 4.2](#) para esta instrução?

4.1.2 [5] <§4.1> Quais recursos (blocos) realizam uma função útil para essa instrução?

4.1.3 [10] <§4.1> Quais recursos (blocos) produzem saídas, mas suas saídas não são usadas para essa instrução? Quais recursos não produzem saídas para ela?

4.2 A implementação básica de ciclo único do MIPS na [Figura 4.2](#) só pode implementar algumas instruções. Novas instruções podem ser acrescentadas

a uma ISA (Instruction Set Architecture) existente, mas a decisão de fazer isso ou não depende, entre outras coisas, do custo e da complexidade que tal acréscimo introduz no caminho de dados e controle do processador. Os três primeiros problemas neste exercício referem-se a esta nova instrução:

Instrução: LWI Rt, Rd(Rs)

Interpretação: $\text{Reg}[\text{Rt}] = \text{Mem}[\text{Reg}[\text{Rd}] + \text{Reg}[\text{Rs}]]$

4.2.1 [10] <§4.1> Quais blocos existentes (se houver) podem ser usados para essa instrução?

4.2.2 [10] <§4.1> De quais novos blocos funcionais (se houver) precisamos para essa instrução?

4.2.3 [10] <§4.1> De quais novos sinais da unidade de controle (se houver) precisamos para dar suporte a essa instrução?

4.3 Quando os projetistas de processador consideram uma melhoria possível no caminho de dados do processador, a decisão normalmente depende da escolha de custo/desempenho. Nos três problemas a seguir, considere que estamos começando com um caminho de dados da [Figura 4.2](#), em que os blocos I-Mem, Add, Mux, ALU, Regs, D-Mem e de Controle têm latências de 400 ps, 100 ps, 30 ps, 120 ps, 200 ps, 350 ps e 100 ps, respectivamente, e custos de 1000, 30, 10, 100, 200, 2000 e 500, respectivamente.

Considere o acréscimo de um multiplicador à ALU. Esse acréscimo somará 300 ps à latência da ALU e acrescentará um custo de 600 à ALU. O resultado será 5% menos instruções executadas, pois não precisaremos mais emular a instrução MUL.

4.3.1 [10] <§4.1> Qual é o tempo de ciclo de clock com e sem essa melhoria?

4.3.2 [10] <§4.1> Qual é o ganho de velocidade obtido acrescentando essa melhoria?

4.3.3 [10] <§4.1> Compare a razão custo/desempenho com e sem essa melhoria.

4.4 Os problemas neste exercício consideram que os blocos lógicos necessários para implementar o caminho de dados de um processador têm as seguintes latências:

I-Mem	Add	Mux	ALU	Regs	D-Mem	Extensão de sinal	Shift-esq-2
200 ps	70 ps	20 ps	90 ps	90 ps	250 ps	15 ps	10 ps

4.4.1 [10] <§4.3> Se a única coisa que precisássemos fazer em um processador fosse buscar instruções consecutivas ([Figura 4.6](#)), qual seria o tempo do ciclo?

4.4.2 [10] <§4.3> Considere um caminho de dados semelhante ao da [Figura 4.11](#), mas para um processador que só tem um tipo de instrução: desvio incondicional relativo ao PC. Qual seria o tempo de ciclo para esse caminho de dados?

4.4.3 [10] <§4.3> Repita o Exercício 4.4.2, mas desta vez precisamos dar suporte apenas a desvios *condicionais* relativos ao PC.

Os três problemas restantes neste exercício referem-se ao elemento Shift-esq-2 no caminho de dados:

4.4.4 [10] <§4.3> Quais tipos de instruções exigem esse recurso?

4.4.5 [20] <§4.3> Para que tipos de instruções (se houver) esse recurso está no caminho crítico?

4.4.6 [10] <§4.3> Supondo que só temos suporte para instruções beq e add, discuta como as mudanças na latência indicada desse recurso afetam o tempo de ciclo do processador. Suponha que as latências de outros recursos não mudem.

4.5 Para os problemas neste exercício, considere que não existem stalls de pipeline e que o desmembramento das instruções executadas seja o seguinte:

add	addi	not	beq	lw	sw
20%	20%	0%	25%	25%	10%

4.5.1 [10] <§4.3> Em que fração de todos os ciclos a memória de dados é utilizada?

4.5.2 [10] <§4.3> Em que fração de todos os ciclos a entrada do circuito por extensão de sinal é necessária? O que esse circuito está fazendo nos ciclos em que sua entrada não é necessária?

4.6 Quando os chips de silício são fabricados, os defeitos nos materiais (por exemplo, o silício) e os erros de manufatura podem resultar em circuitos defeituosos. Um defeito muito comum é quando um fio afeta o sinal em outro. Isso é chamado de falha cross-talk. Uma classe especial de falhas cross-talk é quando um sinal está conectado a um fio que tem um valor lógico constante (por exemplo, um fio da fonte de alimentação). Nesse caso, temos uma falha stuck-at-0 ou stuck-at-1, e o sinal afetado sempre tem um valor lógico 0 ou 1, respectivamente. Os problemas a seguir referem-se ao bit 0 do Registrador Escrita no banco de registradores da [Figura 4.24](#).

4.6.1 [10] <§§4.3, 4.4> Vamos supor que o teste do processador seja feito preenchendo o PC, registradores e memórias de dados e instruções com alguns valores (você pode escolher quais valores), permitindo que uma única

instrução seja executada e depois lendo o PC, memórias e registradores. Esses valores são então examinados para determinar se uma falha em particular está presente. Você conseguiria criar um teste (valores para PC, memórias e registradores) que determinaria se existe uma falha stuck-at-0 nesse sinal?

4.6.2 [10] <§§4.3, 4.4> Repita o Exercício 4.6.1 para uma falha stuck-at-1.

Você conseguiria usar um único teste para stuck-at-0 e stuck-at-1? Caso afirmativo, explique como; se não, explique por que não.

4.6.3 [60] <§§4.3, 4.4> Se soubermos que o processador tem uma falha stuck-at-1 nesse sinal, o processador ainda é utilizável? Para isso, temos de converter qualquer programa que execute em um processador MIPS normal em um programa que funcione nesse processador. Você pode considerar que existe memória de instrução e memória de dados livre suficiente para tornar o programa maior e armazenar dados adicionais. Dica: o processador é utilizável se cada instrução “rompida” por essa falha puder ser substituída por uma sequência de instruções “funcionais” que conseguem o mesmo efeito.

4.6.4 [10] <§§4.3, 4.4> Repita o Exercício 4.6.1, mas agora o teste é se o sinal de controle “MemRead” torna-se 0 se o sinal de controle RegDst for 0; caso contrário, nenhuma falha.

4.6.5 [10] <§§4.3, 4.4> Repita o Exercício 4.6.4, mas agora o teste é se o sinal de controle “Jump” torna-se 0 se o sinal de cont RegDst for 0; caso contrário, nenhuma falha.

4.7 Neste exercício, examinamos detalhadamente como uma instrução é executada em um caminho de dados de ciclo único. Os problemas neste exercício referem-se a um ciclo de clock em que o processador busca a seguinte word de instrução:

10101100011000100000000000010100

Considere que a memória de dados contém apenas zeros e que os registradores do processador possuem os seguintes valores no início do ciclo em que a word de instrução anterior é apanhada:

r0	r1	r2	r3	r4	r5	r6	r8	r12	r31
0	-1	2	-3	-4	10	6	8	2	-16

4.7.1 [5] <§4.4> Quais são as saídas da unidade de extensão de sinal e salto “Shift left 2” (topo da [Figura 4.24](#)) para essa palavra de instrução?

4.7.2 [10] <§4.4> Quais são os valores das entradas da unidade de controle da

ALU para essa instrução?

4.7.3 [10] <§4.4> Qual é o novo endereço do PC após a execução dessa instrução? Destaque o caminho através do qual esse valor é determinado.

4.7.4 [10] <§4.4> Para cada Mux, mostre os valores de sua saída de dados durante a execução dessa instrução e esses valores de registrador.

4.7.5 [10] <§4.4> Para a ALU e as duas unidades de soma, quais são seus valores de entrada de dados?

4.7.6 [10] <§4.4> Quais são os valores de todas as entradas para a unidade de “Registradores”?

4.8 Neste exercício, examinamos como o pipelining afeta o tempo do ciclo de clock do processador. Os problemas neste exercício consideram que os estágios individuais do caminho de dados têm as seguintes latências:

IF	ID	EX	MEM	WB
250 ps	350 ps	150 ps	300 ps	200 ps

Além disso, considere que as instruções executadas pelo processador são desmembradas da seguinte forma:

alu	beq	lw	sw
45%	20%	20%	15%

4.8.1 [5] <§4.5> Qual é o tempo do ciclo de clock em um processador com e sem pipeline?

4.8.2 [10] <§4.5> Qual é a latência total de uma instrução LW em um processador com e sem pipeline?

4.8.3 [10] <§4.5> Se pudessemos dividir um estágio do caminho de dados com pipeline em dois novos estágios, cada um com metade da latência do estágio original, que estágio dividiríamos e qual é o novo tempo do ciclo de clock do processador?

4.8.4 [10] <§4.5> Supondo que não haja stalls ou hazards, qual é a utilização da memória de dados?

4.8.5 [10] <§4.5> Supondo que não haja stalls ou hazards, qual é a utilização da porta de escrita de registrador da unidade “Registradores”?

4.8.6 [30] <§4.5> Em vez de uma organização de ciclo único, podemos usar uma organização multiciclos, em que cada instrução ocupa múltiplos ciclos, mas uma instrução termina antes que outra seja apanhada. Nessa organização, uma instrução só percorre os estágios que ela realmente precisa

(por exemplo, ST só ocupa quatro ciclos, pois não precisa do estágio WB). Compare os tempos do ciclo de clock e os tempos de execução com a organização em ciclo único, multiciclos e em pipeline.

4.9 Neste exercício, examinamos como as dependências de dados afetam a execução no pipeline básico de cinco estágios descrito na [Seção 4.5](#). Os problemas neste exercício referem-se a esta sequência de instruções:

```
or r1,r2,r3
or r2,r1,r4
or r1,r1,r2
```

Além disso, considere os seguintes tempos do ciclo de clock para cada uma das opções relacionadas ao forwarding:

Sem forwarding	Com forwarding completo	Apenas com forwarding ALU-ALU
250 ps	300 ps	290 ps

- 4.9.1** [10] <§4.5> Indique as dependências e seu tipo.
- 4.9.2** [10] <§4.5> Suponha que não haja forwarding nesse processador em pipeline. Indique hazards e acrescente instruções nop para eliminá-los.
- 4.9.3** [10] <§4.5> Suponha que haja forwarding completo. Indique os hazards e acrescente instruções nop para eliminá-los.
- 4.9.4** [10] <§4.5> Qual é o tempo de execução total dessa sequência de instruções sem forwarding e com forwarding completo? Qual é o ganho de velocidade obtido, acrescentando-se forwarding completo a um pipeline que não tinha forwarding?
- 4.9.5** [10] <§4.5> Acrescente instruções nop a esse código para eliminar hazards se houver apenas forwarding ALU-ALU (nenhum forwarding do estágio MEM para EX).
- 4.9.6** [10] <§4.5> Qual é o tempo de execução total dessa sequência de instruções apenas com forwarding ALU-ALU? Qual é o ganho de velocidade em relação a um pipeline sem forwarding?
- 4.10** Neste exercício, examinamos como os hazards de recursos, os hazards de

controle e o projeto da ISA podem afetar a execução em pipeline. Os problemas neste exercício referem-se ao seguinte fragmento de código MIPS:

```
sw r16,12(r6)
lw r16,8(r6)
beq r5,r4,Label # Considere que r5!=r4
add r5,r1,r4
slt r5,r15,r4
```

Considere que os estágios de pipeline individuais possuem as seguintes latências:

IF	ID	EX	MEM	WB
200 ps	120 ps	150 ps	190 ps	100 ps

4.10.1 [10] <§4.5> Para este problema, suponha que todos os desvios sejam perfeitamente previstos (isso elimina todos os hazards de controle) e que nenhum slot de delay seja utilizado. Se tivermos apenas uma memória (para instruções e dados), haverá um hazard estrutural toda vez que precisarmos apanhar uma instrução no mesmo ciclo em que outra instrução acessa dados. Para garantir o processo do forwarding, esse hazard sempre precisa ser resolvido em favor da instrução que acessa dados. Qual é o tempo de execução total dessa sequência de instruções no pipeline de cinco estágios que tem apenas uma memória? Vimos que os hazards de dados podem ser eliminados acrescentando nops ao código. Você conseguiria fazer o mesmo com esse hazard estrutural? Por quê?

4.10.2 [20] <§4.5> Para este problema, suponha que todos os desvios sejam perfeitamente previstos (isso elimina todos os hazards de controle) e que nenhum slot de delay seja utilizado. Se mudarmos as instruções load/store para usar um registrador (sem um offset) como endereço, essas instruções não precisam mais usar a ALU. Como resultado, os estágios MEM e EX podem ser sobrepostos e o pipeline tem apenas quatro estágios. Mude esse código para acomodar essa ISA alterada. Supondo que essa mudança não

afete o tempo do ciclo de clock, que ganho de velocidade é obtido nessa sequência de instruções?

- 4.10.3** [10] <§4.5> Considerando stall-on-branch e nenhum slot de delay, que ganho de velocidade é obtido nesse código se os resultados do desvio forem determinados no estágio ID, em relação à execução em que os resultados do desvio são determinados no estágio EX?
- 4.10.4** [10] <§4.5> Dadas essas latências de estágio de pipeline, repita o cálculo de ganho de velocidade de 4.10.2, mas leve em conta a (possível) mudança no tempo do ciclo de clock. Quando EX e MEM são feitos em um único estágio, a maior parte do trabalho pode ser feita em paralelo. Como resultado, o estágio EX/MEM resultante tem uma latência que é a maior das duas originais, mais 20 ps necessários para o trabalho que poderia ser feito em paralelo.
- 4.10.5** [10] <§4.5> Dadas essas latências de estágio em pipeline, repita o cálculo de ganho de velocidade de 4.10.3, mas leve em conta a (possível) mudança no tempo do ciclo de clock. Suponha que a latência do estágio ID aumente em 50% e a latência do estágio EX diminua em 10 ps quando a resolução do resultado do desvio é passada de EX para ID.
- 4.10.6** [10] <§4.5> Considerando stall-on-branch e nenhum slot de delay, qual é o novo tempo do ciclo de clock e tempo de execução dessa sequência de instruções se o cálculo de endereço de beq for passado para o estágio MEM? Qual é o ganho de velocidade decorrente dessa mudança? Suponha que a latência do estágio EX seja reduzida em 20 ps e a latência do estágio MEM fique inalterada quando a resolução do resultado do desvio for passada de EX para MEM.

4.11 Considere o loop a seguir.

```
loop:lw r1,0(r1)
      and r1,r1,r2
      lw r1,0(r1)
      lw r1,0(r1)
      beq r1,r0,loop
```

Considere que a previsão de desvio perfeita é utilizada (sem stalls devido aos hazards de controle), que não existem slots de delay e que o pipeline possui suporte para forwarding completo. Considere também que muitas iterações desse loop são executadas antes que o loop termine.

- 4.11.1** [10] <§4.6> Mostre um diagrama de execução de pipeline para a terceira iteração desse loop, do ciclo em que apanhamos a primeira instrução dessa iteração até (mas não incluindo) o ciclo em que apanhamos a primeira instrução da iteração seguinte. Mostre todas as instruções que estão no pipeline durante esses ciclos (não apenas aquelas da terceira iteração).
- 4.11.2** [10] <§4.6> Com que frequência (como uma porcentagem de todos os ciclos) temos um ciclo em que todos os cinco estágios do pipeline estão realizando trabalho útil?

- 4.12** Este exercício tem por finalidade ajudá-lo a entender as escolhas entre custo/complexidade/desempenho do forwarding em um processador com pipeline. Os problemas neste exercício referem-se aos caminhos de dados em pipeline da [Figura 4.45](#). Esses problemas consideram que, de todas as instruções executadas em um processador, a fração dessas instruções a seguir tem um tipo particular de dependência de dados RAW. O tipo de dependência de dados RAW é identificado pelo estágio que produz o resultado (EX ou MEM) e a instrução que consome o resultado (1^a instrução que segue aquela que produz o resultado, 2^a instrução que a segue ou ambas). Consideramos que a escrita do registrador é feita na primeira metade do ciclo de clock e que as leituras do registrador são feitas na segunda metade do

ciclo, de modo que dependências “EX para 3^a” e “MEM para 3^a” não são contadas, pois não podem resultar em hazards de dados. Além disso, considere que o CPI do processador é 1 se não houver hazards de dados.

EX para 1 ^a somente	MEM para 1 ^a somente	EX para 2 ^a somente	MEM para 2 ^a somente	EX para 1 ^a e MEM para 2 ^a	Outras dependências RAW
5%	20%	5%	10%	10%	10%

Considere as seguintes latências para estágios individuais do pipeline. No estágio EX, as latências são dadas separadamente para um processador sem forwarding e um processador com diferentes tipos de forwarding.

IF	ID	EX (sem FW)	EX (FW completo)	EX (FW apenas de EX/MEM)	EX (FW apenas de MEM/WB)	MEM	WB
150 ps	100 ps	120 ps	150 ps	140 ps	130 ps	120 ps	100 ps

- 4.12.1 [10] <§4.7>** Se não usarmos forwarding, em que fração dos ciclos estamos realizando stall devido aos hazards de dados?
- 4.12.2 [5] <§4.7>** Se usarmos o forwarding completo (encaminhar todos os resultados que podem ser encaminhados), em que fração dos ciclos estamos realizando stall devido aos hazards de dados?
- 4.12.3 [10] <§4.7>** Vamos supor que não tenhamos recursos para ter Muxes de três entradas que são necessários para o forwarding completo. Temos de decidir se é melhor encaminhar apenas do registrador de pipeline EX/MEM (forwarding do próximo ciclo) ou apenas do registrador de pipeline MEM/WB (forwarding de dois ciclos). Qual das duas opções resulta em menos ciclos de stall de dados?
- 4.12.4 [10] <§4.7>** Para as possibilidades de hazard e latências de estágio de pipeline indicadas, qual é o ganho de velocidade obtido acrescentando-se forwarding completo a um pipeline que não tinha forwarding?
- 4.12.5 [10] <§4.7>** Qual seria o ganho de velocidade adicional (relativo a um processador com forwarding) se acrescentássemos o forwarding de retorno no tempo que elimina todos os hazards de dados? Suponha que o circuito de retorno no tempo ainda a ser inventado acrescente 100 ps à latência do estágio EX de forwarding completo.
- 4.12.6 [20] <§4.7>** Repita o Exercício 4.12.3, mas desta vez determine quais das duas opções resulta em menor tempo por instrução.
- 4.13** Este exercício tem por finalidade ajudá-lo a entender o relacionamento

entre forwarding, detecção de hazard e projeto de ISA. Os problemas neste exercício referem-se a estas sequências de instrução, e considere que ele é executado em um caminho de dados com pipeline em cinco estágios.

```
add r5,r2,r1
lw  r3,4(r5)
lw  r2,0(r2)
or  r3,r5,r3
sw  r3,0(r5)
```

4.13.1 [5] <§4.7> Se não houver forwarding ou detecção de hazard, insira nops para garantir a execução correta.

4.13.2 [10] <§4.7> Repita o Exercício 4.13.1, mas agora use nops somente quando um hazard não puder ser evitado alterando ou rearrumando essas instruções. Você pode considerar que o registrador R7 pode ser usado para manter valores temporários no seu código modificado.

4.13.3 [10] <§4.7> Se o processador tem forwarding, mas nos esquecemos de implementar a unidade de detecção de hazard, o que acontece quando esse código é executado?

4.13.4 [20] <§4.7> Se houver forwarding, para os cinco primeiros ciclos durante a execução desse código, especifique quais sinais são ativados em cada ciclo pelas unidades de detecção de hazard e forwarding na [Figura 4.60](#).

4.13.5 [10] <§4.7> Se não houver forwarding, que novas entradas e sinais de saída precisamos para a unidade de detecção de hazard da [Figura 4.60](#)? Usando essa sequência de instruções como exemplo, explique por que cada sinal é necessário.

4.13.6 [20] <§4.7> Para a unidade de detecção de hazard do Exercício 4.13.5, especifique quais sinais de saída ela ativa em cada um dos cinco primeiros ciclos durante a execução desse código.

4.14 Este exercício tem por finalidade ajudá-lo a entender o relacionamento

entre slots de delay, hazards de controle e execução de desvio em um processador com pipeline. Neste exercício, consideramos que o código MIPS a seguir é executado em um processador com um pipeline em cinco estágios, forwarding completo e um previsor de desvio tomado:

```
lw r2,0(r1)
label1: beq r2,r0,label2 # não tomado uma vez, depois tomado
        lw r3,0(r2)
        beq r3,r0,label1 # tomado
        add r1,r3,r1
label2: sw r1,0(r2)
```

4.14.1 [10] <§4.8> Desenhe um diagrama de execução de pipeline para este código, considerando que não existam slots de delay e que os desvios sejam executados no estágio EX.

4.14.2 [10] <§4.8> Repita o Exercício 4.14.1, mas considere que os slots de delay sejam utilizados. No código apresentado, a instrução que vem após o desvio agora é a instrução do slot de delay para esse desvio.

4.14.3 [20] <§4.8> Uma maneira de mover a resolução do desvio para um estágio anterior é não precisar de uma operação da ALU nos desvios condicionais. As instruções de desvio seriam “bez rd, label” e “bnez rd, label”, e haveria desvio se o registrador tivesse e não tivesse um valor 0, respectivamente. Mude esse código para usar essa instrução de desvio em vez de beq. Você pode considerar que o registrador R8 está disponível como um registrador temporário, e que uma instrução tipo R seq (set if equal) pode ser usada.

A [Seção 4.8](#) descreve como a rigidez dos hazards de controle pode ser reduzida movendo-se a execução do desvio para o estágio ID. Essa técnica envolve um comparador dedicado no estágio ID, como mostra a [Figura 4.62](#). Porém, essa técnica tem o potencial de aumentar a latência do estágio ID, além de requerer lógica adicional de forwarding e detecção de hazard.

4.14.4 [10] <§4.8> Usando como exemplo a primeira instrução de desvio no código apresentado, descreva a lógica de detecção de hazard necessária para dar suporte à execução do desvio no estágio ID como na [Figura 4.62](#). Que tipo de hazard essa nova lógica deveria detectar?

4.14.5 [10] <§4.8> Para o código apresentado, qual é o ganho de velocidade

alcançado movendo-se a execução do desvio para o estágio ID? Explique sua resposta. No seu cálculo de ganho de velocidade, considere que a comparação adicional no estágio ID não afeta o tempo do ciclo de clock.

4.14.6 [10] <§4.8> Usando como exemplo a primeira instrução de desvio no código apresentado, descreva o suporte para forwarding que precisa ser acrescentado para dar suporte à execução do desvio no estágio ID. Compare a complexidade dessa nova unidade de forwarding com a complexidade da unidade de forwarding existente na [Figura 4.62](#).

4.15 A importância de ter um bom previsor de desvio depende da frequência com que os desvios condicionais são executados. Juntamente com a precisão do previsor de desvio, isso determinará quanto tempo será gasto com stall devido a desvios mal previstos. Neste exercício, considere o desmembramento das instruções dinâmicas em diversas categorias de instrução, como a seguir:

Tipo R	BEQ	JMP	LW	SW
40%	25%	5%	25%	5%

Além disso, considere as seguintes precisões do previsor de desvio:

Sempre tomado	Sempre não tomado	2 bits
45%	55%	85%

4.15.1 [10] <§4.8> Os ciclos de stall ocasionados por desvios mal previstos aumentam o CPI. Qual é o CPI extra devido a desvios mal previstos com o previsor sempre tomado? Considere que os resultados do desvio sejam determinados no estágio EX, que não existem hazards de dados e que nenhum slot de delay seja utilizado.

4.15.2 [10] <§4.8> Repita o Exercício 4.15.1 para o previsor “sempre não tomado”.

4.15.3 [10] <§4.8> Repita o Exercício 4.15.1 para o previsor de 2 bits.

4.15.4 [10] <§4.8> Com um previsor de 2 bits, que ganho de velocidade seria alcançado se pudéssemos converter metade das instruções de desvio de um modo que substitua uma instrução de desvio por uma instrução da ALU? Suponha que instruções previstas correta e incorretamente tenham a mesma chance de serem substituídas.

4.15.5 [10] <§4.8> Com um previsor de 2 bits, que ganho de velocidade seria obtido se pudéssemos converter metade das instruções de desvio de um

modo que substituisse cada instrução de desvio por duas instruções da ALU? Suponha que instruções previstas correta e incorretamente tenham a mesma chance de serem substituídas.

- 4.15.6** [10] <§4.8> Algumas instruções de desvio são muito mais previsíveis do que outras. Se soubermos que 80% de todas as instruções de desvio executadas são desvios loop-back fáceis de prever, que sempre são previstos corretamente, qual é a precisão do previsor de 2 bits nos 20% restantes das instruções de desvio?
- 4.16** Este exercício examina a precisão de vários previsores de desvios para o seguinte padrão repetitivo (como em um loop) de resultados do desvio: T, NT, T, T, NT
- 4.16.1** [5] <§4.8> Qual é a precisão dos previsores sempre tomado e sempre não tomado para essa sequência dos resultados do desvio?
- 4.16.2** [5] <§4.8> Qual é a precisão do previsor de dois bits para os quatro primeiros desvios nesse padrão, supondo que o previsor comece no estado inferior esquerdo da [Figura 4.63](#) (previsão não tomada)?
- 4.16.3** [10] <§4.8> Qual é a precisão do previsor de dois bits se esse padrão for repetido indefinidamente?
- 4.16.4** [30] <§4.8> Crie um previsor que alcance uma precisão perfeita se esse padrão for repetido indefinidamente. Seu previsor deverá ser um circuito sequencial com uma saída que oferece uma previsão (1 para tomado, 0 para não tomado) e nenhuma entrada que não seja o clock e o sinal de controle que indica que a instrução é um desvio condicional.
- 4.16.5** [10] <§4.8> Qual é a precisão do seu previsor do Exercício 4.16.4 se ele receber um padrão repetitivo que é o oposto exato deste?
- 4.16.6** [20] <§4.8> Repita o Exercício 4.16.4, mas agora o seu previsor deverá ser capaz de, mais cedo ou mais tarde (após um período de aquecimento durante o qual poderá fazer previsões erradas), começar a prever perfeitamente esse padrão e seu oposto. Seu previsor deverá ter uma entrada que lhe diga qual foi o resultado real. Dica: essa entrada permite que seu previsor determine qual dos dois padrões repetitivos ele recebe.
- 4.17** Este exercício explora como o tratamento de exceção afeta o projeto do pipeline. Os três primeiros problemas neste exercício referem-se às duas instruções a seguir:

Instrução 1	Instrução 2
BNE R1, R2, Label	LW R1, 0(R1)

4.17.1 [5] <§4.9> Quais exceções cada uma dessas instruções pode disparar?

Para cada uma dessas exceções, especifique o estágio do pipeline em que ela é detectada.

4.17.2 [10] <§4.9> Se houver um endereço de handler separado para cada exceção, mostre como a organização do pipeline deve ser mudada para ser capaz de tratar dessa exceção. Você pode considerar que os endereços desses handlers são conhecidos quando o processador é projetado.

4.17.3 [10] <§4.9> Se a segunda instrução dessa tabela for apanhada logo após a instrução da primeira tabela, descreva o que acontece no pipeline quando a primeira instrução causa a primeira exceção que você listou no Exercício 4.17.1. Mostre o diagrama de execução do pipeline do momento em que a primeira instrução é apanhada até o momento em que a primeira instrução do handler de exceção é concluída.

4.17.4 [20] <§4.9> No tratamento de exceção com vetor, a tabela de endereços do handler de exceção está na memória de dados em um endereço conhecido (fixo). Mude o pipeline para implementar esse mecanismo de tratamento de exceção. Repita o Exercício 4.17.3 usando esse pipeline modificado e o tratamento de exceção com vetor.

4.17.5 [15] <§4.9> Queremos simular o tratamento de exceção com vetor (descrito no Exercício 4.17.4) em uma máquina que tem apenas um endereço de handler fixo. Escreva o código que deverá estar nesse endereço fixo. Dica: esse código deverá identificar a exceção, obter o endereço correto da tabela de vetor de exceção e transferir a execução para esse handler.

4.18 Neste exercício, comparamos o desempenho dos processadores de um despacho e processadores de dois despachos, levando em conta as transformações do programa que podem ser feitas para otimizar a execução em dois despachos. Os problemas neste exercício referem-se ao seguinte loop (escrito em C):

```
for( i=0 ; i != j ; i+=2 )
    b[i]=a[i]-a[i+1];
```

Ao escrever código MIPS, considere que as variáveis são mantidas em registradores da seguinte forma, e que todos os registradores, com exceção

daqueles indicados como Livre, são usados para manter diversas variáveis, de modo que não podem mais ser usados.

i	j	a	b	c	Livre
R5	R6	R1	R2	R3	R10, R11, R12

4.18.1 [10] <§4.10> Traduza esse código C para instruções MIPS. Sua tradução deverá ser direta, sem rearrumar as instruções para conseguir melhor desempenho.

4.18.2 [10] <§4.10> Se o loop sair depois de executar apenas duas iterações, desenhe um diagrama de pipeline para o seu código MIPS do Exercício 4.18.1 executado em um processador com dois despachos mostrado na [Figura 4.69](#). Suponha que o processador tenha previsão de desvio perfeita e possa buscar quaisquer duas instruções (não apenas instruções consecutivas) no mesmo ciclo.

4.18.3 [10] <§4.10> Rearrume o seu código do Exercício 4.18.1 para alcançar o melhor desempenho em um processador de dois despachos estático, da [Figura 4.69](#).

4.18.4 [10] <§4.10> Repita o Exercício 4.18.2, mas desta vez use seu código MIPS do Exercício 4.18.3.

4.18.5 [10] <§4.10> Qual é o ganho de velocidade ao passar de um processador de um despacho para o de dois despachos da [Figura 4.69](#)? Use o seu código do Exercício 4.18.1 para um despacho e dois despachos, e considere que 1.000.000 iterações do loop são executadas. Assim como no Exercício 4.18.2, considere que o processador tenha previsões de desvio perfeitas, e que um processador de dois despachos possa buscar duas instruções quaisquer no mesmo ciclo.

4.18.6 [10] <§4.10> Repita o Exercício 4.18.5, mas desta vez considere que, no processador de dois despachos, uma das instruções a serem executadas em um ciclo possa ser de qualquer tipo e a outra uma instrução que não seja de memória.

4.19 Este exercício explora a eficiência de energia e seu relacionamento com o desempenho. Os problemas neste exercício consideram o consumo de energia a seguir para a atividade na Memória de Instrução, Registradores e Memória de Dados. Você pode considerar que os outros componentes do caminho de dados gastam uma quantidade de energia insignificante.

I-Mem	1 Leitura de Registrador	Escrita de Registrador	Leitura de Mem D	Escrita de Mem D
-------	--------------------------	------------------------	------------------	------------------

140 pJ	70 pJ	60 pJ	140 pJ	120 pJ
--------	-------	-------	--------	--------

Suponha que os componentes no caminho de dados tenham as latências a seguir. Você pode considerar que os outros componentes do caminho de dados têm latência insignificante.

I-Mem	Controle	Registrador de leitura ou escrita	ALU	Leitura ou escrita de Mem D
200 ps	150 ps	90 ps	90 ps	250 ps

4.19.1 [10] <§§4.3, 4.6, 4.14> Quanta energia é gasta para executar uma instrução ADD em um projeto de ciclo único e no projeto em pipeline com cinco estágios?

4.19.2 [10] <§§4.6, 4.14> Qual é a instrução MIPS no pior caso em termos do consumo de energia, e qual é a energia gasta para executá-la?

4.19.3 [10] <§§4.6, 4.14> Se a redução de energia é fundamental, como você mudaria o projeto em pipeline? Qual é a redução percentual na energia gasta por uma instrução LW após essa mudança?

4.19.4 [10] <§§4.6, 4.14> Qual é o impacto das suas mudanças do Exercício 4.19.3 sobre o desempenho?

4.19.5 [10] <§§4.6, 4.14> Podemos eliminar o sinal de controle MemRead e fazer com que a memória de dados seja lida em cada ciclo, ou seja, podemos ter MemRead = 1 permanentemente. Explique por que o processador ainda funciona corretamente após essa mudança. Qual é o efeito dessa mudança sobre a frequência de clock e consumo de energia?

4.19.6 [10] <§§4.6, 4.14> Se uma unidade ociosa gasta 10% da potência que gastaria se estivesse ativa, qual é a energia gasta pela memória de instrução em cada ciclo? Que porcentagem da energia geral gasta pela memória de instrução essa energia ociosa representa?

Respostas das Seções “Verifique você mesmo”

§4.1, página 220: 3 de 5: Controle, Caminho de dados, Memória, Entrada e Saída estão faltando.

§4.2, página 222: falso. Elementos de estado disparados na borda tornam a leitura e escrita simultâneas tanto possíveis quanto não ambíguas.

§4.3, página 229: I. a. II. c.

§4.4, página 240: Sim, Desvio e ALUOp0 são idênticos. Além disso, MemtoReg e RegDst são opostos um do outro. Você não precisa de um inversor; basta usar o outro sinal e inverter a ordem das entradas para o

multiplexador!

§4.5, página 251: 1. Stall no resultado 1w. 2. Bypassing do primeiro resultado de add escrito em \$t1. 3. Nenhum stall ou bypassing é necessário.

§4.6, página 261: Afirmações 2 e 4 estão corretas; o restante está incorreto.

§4.8, página 285: 1. Previsão não tomada. 2. Previsão tomada. 3. Previsão dinâmica.

§4.9, página 290: A primeira instrução, pois ela é executada logicamente antes das outras.

§4.10, página 301: 1. Ambos. 2. Ambos. 3. Software. 4. Hardware. 5. Hardware. 6. Hardware. 7. Ambos. 8. Hardware. 9. Ambos.

§4.12, página 308: Duas primeiras são falsas e duas últimas são verdadeiras.

Grande e Rápida: Explorando a Hierarquia de Memória

O ideal seria ter uma capacidade de memória infinitamente grande, a ponto de qualquer palavra 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*

- 5.1 Introdução
- 5.2 Tecnologias de memória
- 5.3 Princípios básicos de cache
- 5.4 Medindo e melhorando o desempenho da cache
- 5.5 Hierarquia de memória estável
- 5.6 Máquinas virtuais
- 5.7 Memória virtual
- 5.8 Uma estrutura comum para hierarquias de memória
- 5.9 Usando uma máquina de estado finito para controlar uma cache simples
- 5.10 Paralelismo e hierarquias de memória: coerência de cache
- 5.11 Vida real: as hierarquias de memória ARM Cortex-A8 e Intel Core i7

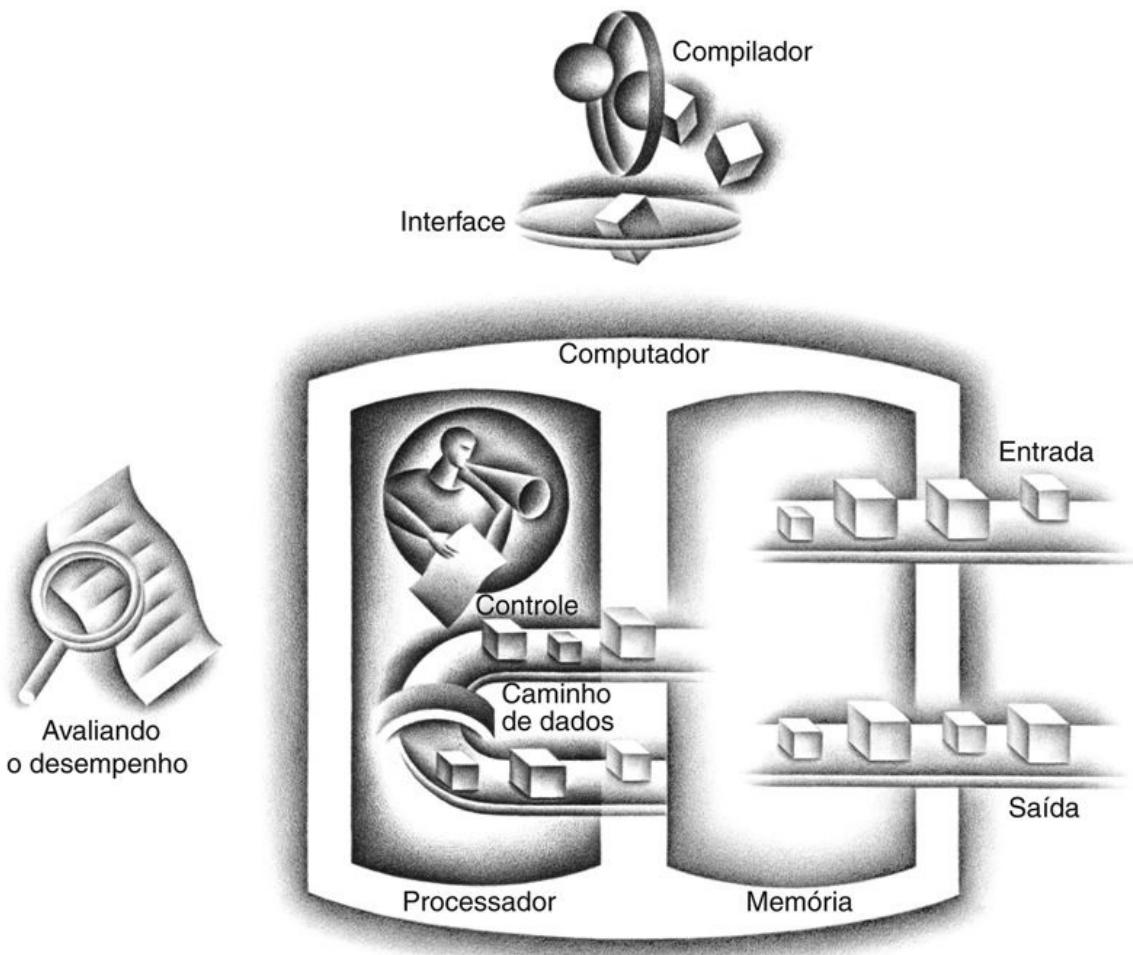
5.12 Mais rápido: Bloqueio de cache e multiplicação matricial

5.13 Falácia e armadilhas

5.14 Comentários finais

5.15 Exercícios

Os cinco componentes clássicos de um computador



5.1. 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 essa ilusão. Antes de vermos como a ilusão é realmente criada, vamos considerar uma analogia simples que ilustra os princípios e mecanismos-chave utilizados.

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 os primeiros 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 rapidamente o que deseja.

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 à mesa para examiná-lo, é provável que precise examiná-lo novamente em breve.

localidade temporal

O princípio em que se um local de dados é referenciado, então, ele tenderá a ser referenciado novamente em breve.

- **Localidade espacial** (localidade no espaço): 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, resolveu 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.

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.

Assim como os acessos aos livros na estante exibem naturalmente a localidade, esta mesma 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 sequencialmente, os programas mostram alta localidade espacial. Os acessos a dados também exibem uma localidade espacial natural. Por exemplo, os acessos sequenciais 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 bit do que as memórias mais lentas e, portanto, são menores.

hierarquia de memória

Uma estrutura que usa múltiplos níveis de memórias; conforme a distância para o processador aumenta, o tamanho das memórias e o tempo de acesso também aumentam.

A [Figura 5.1](#) 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 à velocidade oferecida pela memória mais rápida.

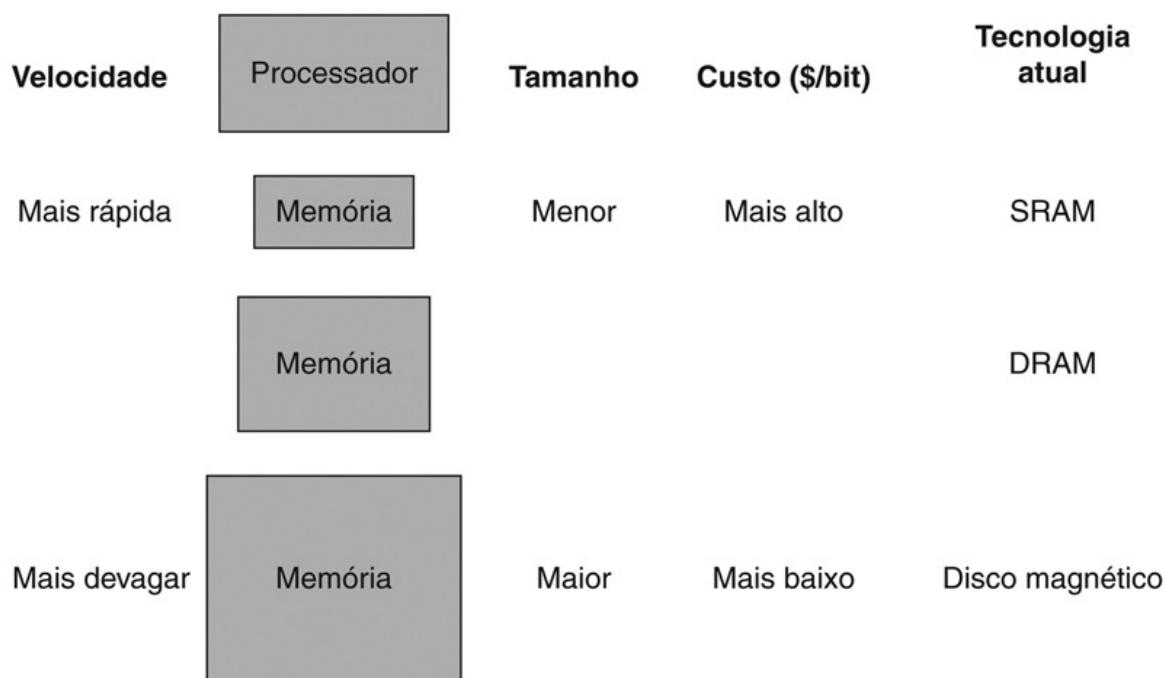


FIGURA 5.1 A estrutura básica de uma hierarquia de memória.

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. A memória flash substituiu os discos em muitos dispositivos embutidos, e pode levar a um novo nível na hierarquia de armazenamento para computadores de desktop e servidor; veja [Seção 5.2](#).

Da mesma forma, os dados são organizados 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 5.2](#) 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.

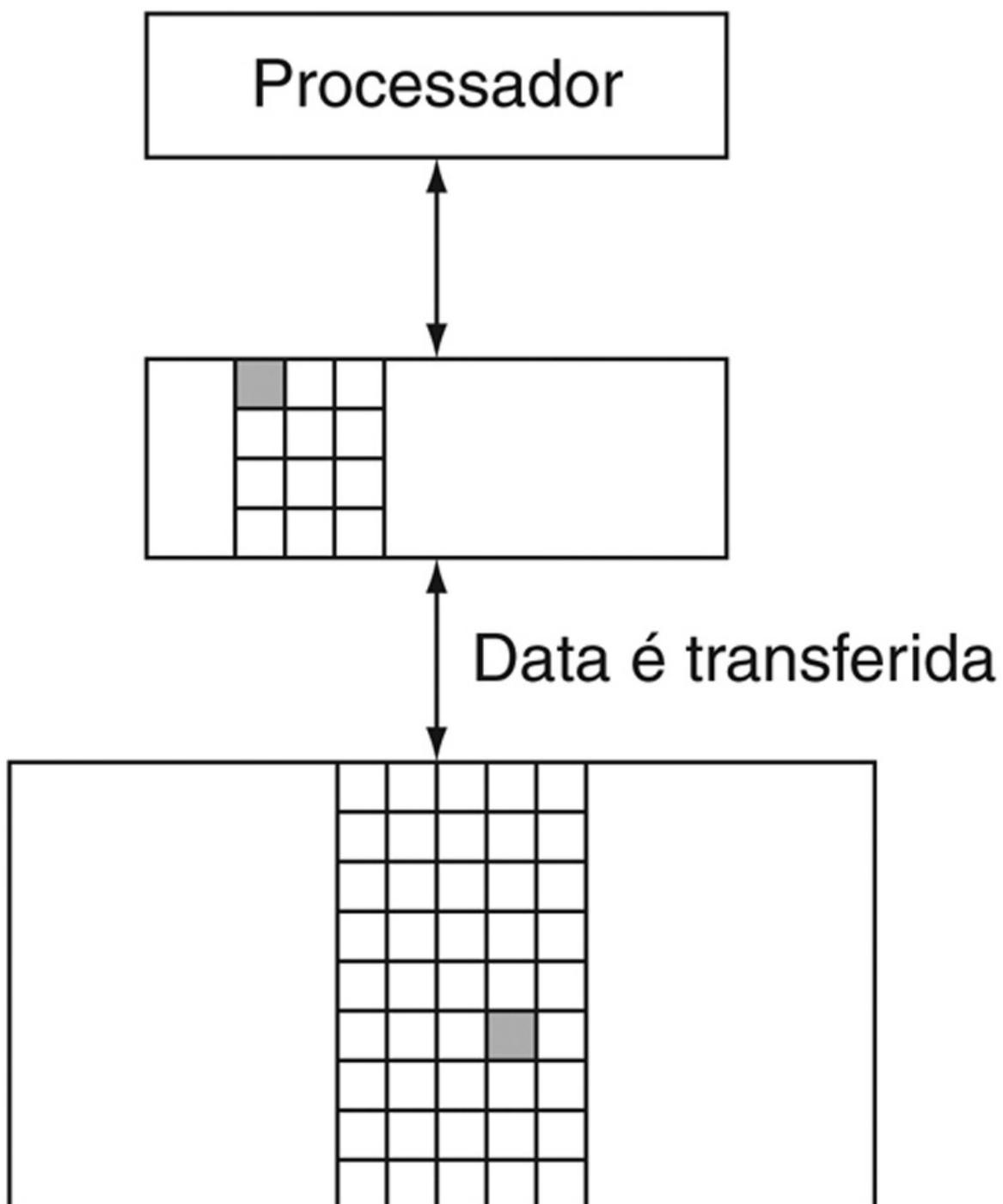


FIGURA 5.2 Cada par de níveis na hierarquia de memória pode ser imaginado como tendo um nível superior e um nível inferior.

Dentro de cada nível, a unidade de informação que está presente ou não é chamada de um *bloco* ou uma *linha*. Em geral, transferimos um bloco inteiro quando copiamos algo entre os níveis.

bloco (ou linha)

A unidade mínima de informação que pode estar presente ou ausente em uma cache.

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.

taxa de acertos

A proporção dos acessos à memória encontrados em um nível da hierarquia de memória.

taxa de falhas

A proporção dos acessos à memória não encontrados em um nível da hierarquia de memória.

Como o desempenho é o principal razão 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 período 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 de substituição de um bloco no nível superior pelo bloco correspondente do nível inferior, mais o tempo para transferir esse bloco ao processador (ou, o tempo de apanhar outro livro das estantes e colocá-lo na mesa). Como o nível superior é menor e construído usando partes 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.)

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 na busca de um bloco de nível inferior para um nível superior da hierarquia de memória, incluindo o tempo para acessar o bloco, transmiti-lo de um nível a outro e inseri-lo no nível que experimentou a falha, e depois passar o bloco a quem o solicitou.

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 para se determinar o desempenho. A confiança nas hierarquias de memória para obter desempenho tem indicado 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 de modo a alcançarem um bom desempenho. Para mostrar como esse entendimento é importante, mais adiante iremos fornecer alguns exemplos, como na [Figura 5.18](#) e na [Seção 5.12](#), que mostram como dobrar o desempenho da multiplicação matricial.

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 voltados a melhorar o desempenho do sistema de memória. Neste capítulo, veremos as principais ideias conceituais, embora muitas simplificações e abstrações tenham sido usadas no sentido de manter o material praticável em tamanho e complexidade.

Colocando em perspectiva

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 espacial movendo blocos consistindo em múltiplas palavras contíguas na memória para níveis superiores na hierarquia.

A Figura 5.3 mostra que uma hierarquia de memória usa tecnologias de memória menores e mais rápidas, perto do processador. Portanto, os acessos com acerto no nível mais alto da hierarquia podem ser processados rapidamente. Os acessos com 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).

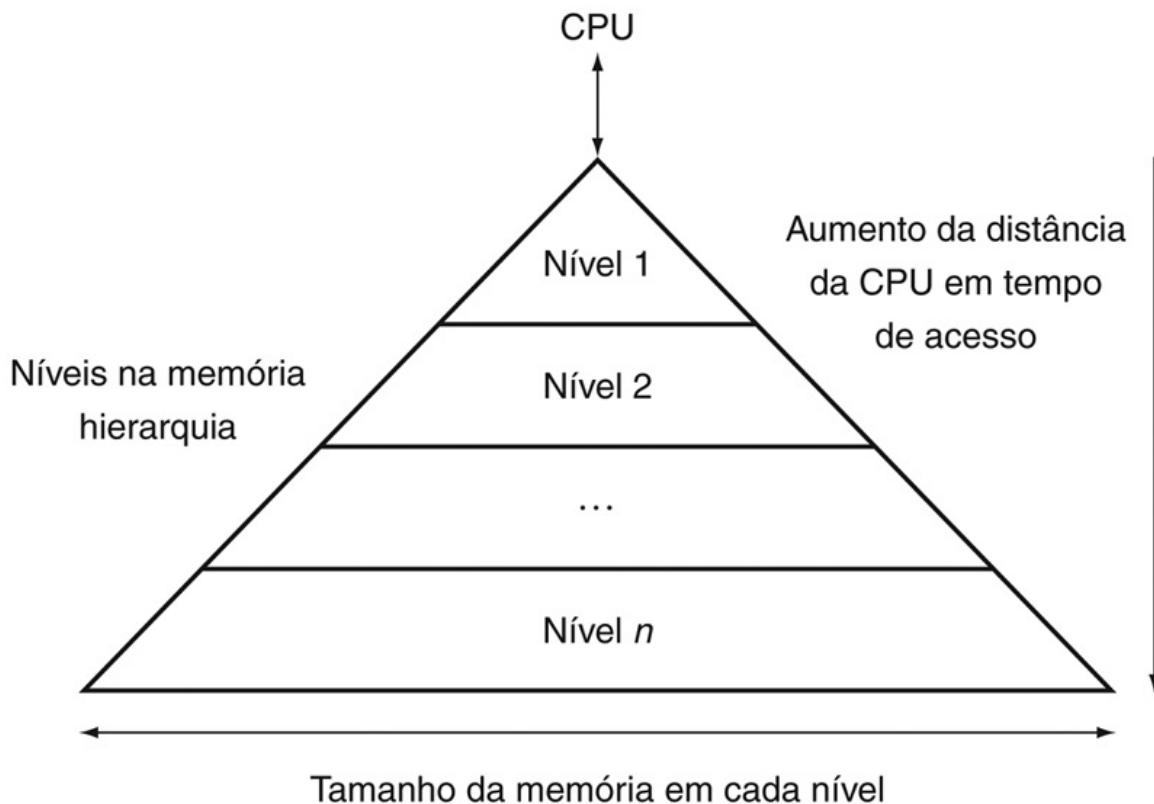


FIGURA 5.3 Este diagrama mostra a estrutura de uma hierarquia de memória: conforme a distância entre ela e o

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.

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 $i + 1$.

Verifique você mesmo

Quais das seguintes afirmações normalmente são verdadeiras?

1. As hierarquias de memória 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.
4. A maioria da capacidade da hierarquia de memória está no nível mais baixo.

5.2. Tecnologias de memória

Existem quatro tecnologias principais usadas atualmente nas hierarquias de memória. A memória principal é implementada a partir da DRAM (Dynamic Random Access Memory), enquanto os níveis mais próximos do processador (caches) usam SRAM (Static Random Access Memory). A DRAM custa menos por bit do que a SRAM, embora seja substancialmente mais lenta. A diferença no preço aumenta porque a DRAM usa muito menos área por bit de memória, e portanto as DRAMs possuem mais capacidade pela mesma quantidade de silício; a diferença na velocidade vem de vários fatores descritos na **Seção B.9** do **Apêndice B**. A terceira tecnologia é a memória flash. Essa memória não volátil é a memória secundária nos Dispositivos Móveis Pessoais. A quarta tecnologia, usada para implementar o nível maior e mais lento na hierarquia nos servidores, é o disco magnético. O tempo de acesso e o preço por bit variam bastante entre essas tecnologias, como mostra a tabela a seguir, usando valores típicos para

2012:

Tecnologia de memória	Tempo de acesso típico	US\$ por GiB em 2012
Memória semicondutora SRAM	0,5–2,5 ns	US\$500–US\$1000
Memória semicondutora DRAM	50–70 ns	US\$10–US\$20
Memória flash semicondutora	5.000–50.000 ns	US\$0,75–US\$1,00
Disco magnético	5.000.000–20.000.000 ns	US\$0,05–US\$0,10

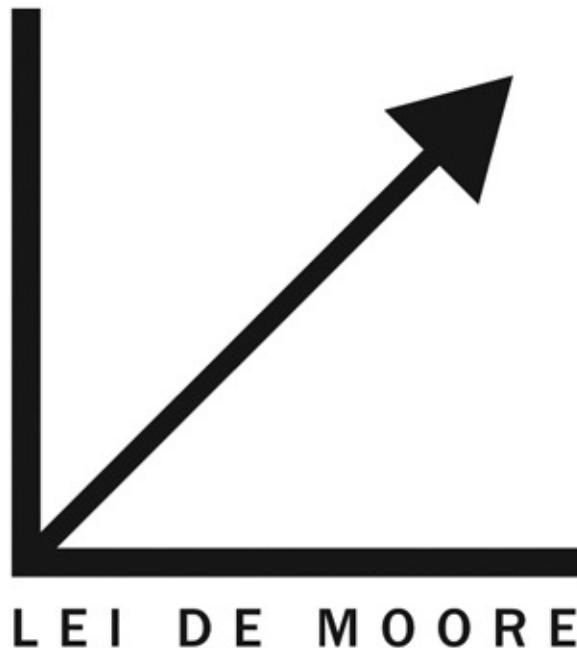
Descrevemos cada tecnologia de memória no restante desta seção.

Tecnologia de SRAM

SRAMs são simplesmente circuitos integrados compostos de arrays de memória com (geralmente) uma única porta de acesso que pode oferecer uma leitura ou uma escrita. SRAMs possuem um tempo de acesso fixo a qualquer dado, embora os tempos de acesso para leitura e escrita possam ser diferentes.

SRAMs não precisam de refresh e, portanto, o tempo de acesso é muito próximo do tempo de ciclo. Elas normalmente utilizam de seis a oito transistores por bit para impedir que a informação seja alterada quando for lida. A SRAM precisa de um mínimo de energia para reter a carga no modo standby.

No passado, a maioria dos sistemas PC e servidor usava chips SRAM separados para suas caches primária, secundária ou mesmo terciária. Hoje, graças à **Lei de Moore**, todos os níveis de caches são integrados no chip do processador, de modo que o mercado para chips SRAM separados quase desapareceu.



Tecnologia DRAM

Em uma SRAM, desde que haja energia aplicada, o valor pode ser mantido indefinidamente. Em uma RAM dinâmica (DRAM), o valor mantido em uma célula é armazenado como uma carga em um capacitor. Um único transistor é então utilizado para acessar essa carga armazenada, seja para ler o valor ou para modificar a carga lá armazenada. Como as DRAMs usam apenas um único transistor por bit de armazenamento, elas são muito mais densas e mais baratas por bit do que a SRAM. Como as DRAMs armazenam a carga em um capacitor, esta não pode ser mantida indefinidamente, e precisa ser renovada periodicamente. É por isso que essa estrutura de memória é denominada dinâmica, ao contrário do armazenamento estático em uma célula de SRAM.

Para renovar uma célula, simplesmente lemos seu conteúdo e o escrevemos de volta. A carga pode ser retida por vários milissegundos. Se cada bit tivesse que ser lido da DRAM e depois escrito de volta individualmente, estaríamos constantemente renovando a DRAM, sem que restasse tempo para acessá-la. Felizmente, as DRAMs utilizam uma estrutura de decodificação de dois níveis, e isso nos permite renovar uma *linha* inteira (que armazena uma linha de words) com um ciclo de leitura, seguido imediatamente por um ciclo de escrita.

A [Figura 5.4](#) mostra a organização interna de uma DRAM e a [Figura 5.5](#) mostra como a densidade, o custo e o tempo de acesso das DRAMs mudaram

com o passar dos anos.

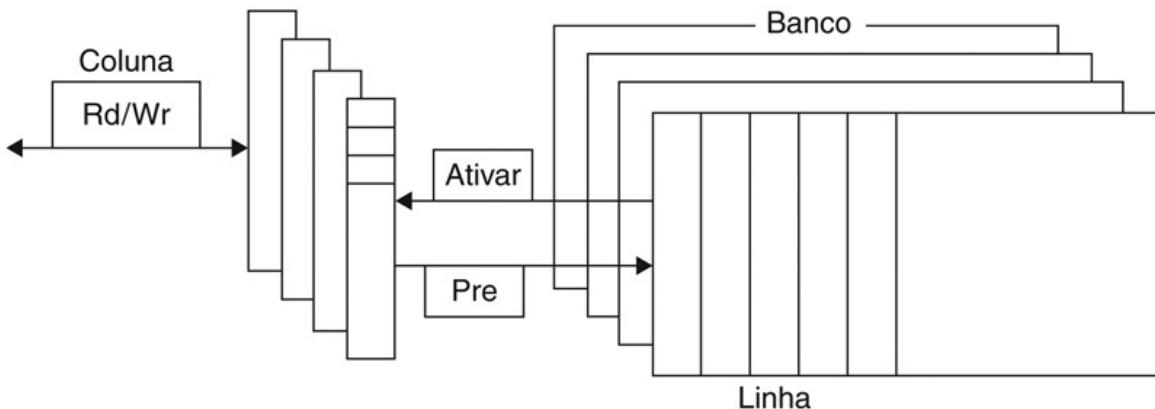


FIGURA 5.4 Organização interna de uma DRAM.

As DRAMs modernas são organizadas em bancos, normalmente quatro para a DDR3. Cada banco consiste em uma série de linhas. O envio de um comando PRE (pré-carga) abre ou fecha um banco. Um endereço de linha é enviado com um comando Ativar, que faz com que a linha seja transferida para um buffer. Quando a linha estiver no buffer, ela poderá ser transferida por endereços de coluna sucessivos por qualquer que seja a largura da DRAM (normalmente 4, 8 ou 16 bits na DDR3), ou especificando uma transferência em bloco e o endereço inicial. Cada comando, bem como as transferências em bloco, é sincronizado com um clock.

Ano de introdução	Tamanho do chip	US\$ por GiB	Tempo de acesso total para uma nova linha/coluna	Tempo de acesso da coluna média para a linha existente
1980	64 Kibibit	\$1.500.000	250 ns	150 ns
1983	256 Kibibit	\$500.000	185 ns	100 ns
1985	1 Mebibit	\$200.000	135 ns	40 ns
1989	4 Mebibit	\$50.000	110 ns	40 ns
1992	16 Mebibit	\$15.000	90 ns	30 ns
1996	64 Mebibit	\$10.000	60 ns	12 ns
1998	128 Mebibit	\$4.000	60 ns	10 ns
2000	256 Mebibit	\$1.000	55 ns	7 ns
2004	512 Mebibit	\$250	50 ns	5 ns
2007	1 Gibibit	\$50	45 ns	1,25 ns
2010	2 Gibibit	\$30	40 ns	1 ns
2012	4 Gibibit	\$1	35 ns	0,8 ns

FIGURA 5.5 Tamanho das DRAM aumentado por múltiplos de quatro, aproximadamente, uma vez a cada três anos até 1996, e depois disso bem mais lentamente.

As melhorias no tempo de acesso têm sido mais lentas, porém contínuas, e o custo acompanha aproximadamente as melhorias na densidade, embora o custo geralmente seja afetado por outras questões, como disponibilidade e demanda. O custo por gibibyte não está ajustado pela inflação.

A organização de linha que ajuda na renovação também ajuda com o desempenho. Para melhorar o desempenho, as DRAMs colocam as linhas em um buffer visando o acesso repetido. O buffer atua como uma SRAM; alterando o endereço, bits aleatórios podem ser acessados no buffer até o acesso da próxima linha. Essa capacidade melhora significativamente o tempo de acesso, pois o tempo de acesso aos bits na linha é muito menor. Tornar o chip mais largo também melhora a largura de banda de memória do chip. Quando a linha está no buffer, ela pode ser transferida para endereços sucessivos qualquer que seja a largura da DRAM (normalmente, 4, 8 ou 16 bits), ou especificando uma transferência em bloco e o endereço inicial dentro do buffer.

Para melhorar ainda mais a interface com os processadores, as DRAMs acrescentaram clocks e são devidamente chamadas de DRAMs síncronas ou SDRAMs. A vantagem das SDRAMs é que o uso de um clock elimina o tempo de sincronização entre a memória e o processador. A vantagem na velocidade das DRAMs síncronas vem da capacidade de transferir os bits na rajada sem ter que especificar bits de endereço adicionais. Em vez disso, o clock transfere os

bits sucessivos de uma só vez. A versão mais veloz é denominada SDRAM DDR (*Double Data Rate* — taxa de dados dupla). O nome indica que as transferências de dados são realizadas nas bordas de subida e descida do clock, obtendo assim o dobro da largura de banda que você poderia esperar com base na taxa de clock e largura dos dados. A versão mais recente dessa tecnologia se chama DDR4. Uma DRAM DDR4-3200 pode realizar 3200 milhões de transferências por segundo, o que significa que tem um clock de 1600 MHz.

Para sustentar tanta largura de banda, é preciso uma organização inteligente *dentro* da DRAM. Em vez de simplesmente um buffer de linha mais rápido, a DRAM pode ser organizada internamente para ler ou escrever, a partir de vários *bancos*, com cada um tendo seu próprio buffer de linha. O envio de um endereço a vários bancos permite que todos eles leiam ou escrevam simultaneamente. Por exemplo, com quatro bancos, há apenas um tempo de acesso e depois os acessos fazem o rodízio entre os quatro bancos, para fornecer quatro vezes a largura de banda. Esse esquema de acesso em forma de rodízio é chamado de *intercalação de endereço*.

Embora os Personal Mobile Devices como o iPad ([Capítulo 1](#)) usem DRAMs individuais, a memória para os servidores normalmente é vendida em pequenas placas chamadas *módulos de memória dual em linha* (DIMMs — Dual Inline Memory Modules). As DIMMs normalmente contêm de 4 a 16 DRAMs, e normalmente são organizadas para que tenham 8 bytes de largura para os sistemas servidores. Uma DIMM usando SDRAMs DDR4-3200 poderia transferir a $8 \times 3200 = 25.600$ megabytes por segundo. Essas DIMMs recebem o nome de sua largura de banda: PC25600. Como uma DIMM pode ter tantos chips de DRAM que somente uma parte deles seja usada para uma transferência em particular, precisamos de um termo para nos referir ao subconjunto de chips em uma DIMM que compartilhe linhas de endereço comuns. Para evitar confusão com os nomes de DRAM internos das linhas e bancos, usamos o termo *fileira de memória* para esse subconjunto de chips em uma DIMM.

Detalhamento

Uma forma de medir o desempenho de um sistema de memória por trás das caches é o benchmark Stream (McCalpin, 1995). Ele mede o desempenho de operações de vetor longas. Elas não possuem localidade temporal e acessam arrays que não são maiores do que a cache do computador sendo testado.

Memória Flash

A memória flash é um tipo de memória somente de leitura programável e apagável eletricamente (EEPROM).

Diferente dos discos e da DRAM, mas como outras tecnologias de EEPROM, as escritas podem desgastar os bits da memória flash. Para lidar com esses limites, a maior parte dos produtos flash inclui um controlador para espalhar as escritas, remapeando blocos que foram escritos muitas vezes, para blocos menos “pisados”. Essa técnica é chamada de *nivelamento do desgaste*. Com o nivelamento do desgaste, os dispositivos móveis pessoais provavelmente não excederão os limites de escrita na memória flash. Esse nivelamento do desgaste reduz o desempenho em potencial da memória flash, mas é necessário para que um software de nível mais alto não tenha que monitorar o desgaste do bloco. Os controladores flash que realizam o nivelamento do desgaste também podem melhorar o aproveitamento, retirando do mapeamento as células que foram manufaturadas com falha.

Memória em disco

Como mostra a [Figura 5.6](#), um disco rígido magnético consiste em um conjunto de placas, que giram em um eixo a 5400 a 15.000 rotações por minuto. As placas de metal são cobertas com um material de gravação magnético nos dois lados, semelhante ao material encontrado em uma fita cassete ou de vídeo. Para ler e gravar informações em um disco rígido, um *braço móvel* contendo uma pequena bobina eletromagnética, chamada *cabeça de leitura/gravação*, é posicionado bem próximo a cada superfície. A unidade inteira fica permanentemente lacrada para controlar o ambiente dentro dela, que, por sua vez, permite que as cabeças do disco fiquem muito mais próximas da superfície do disco.

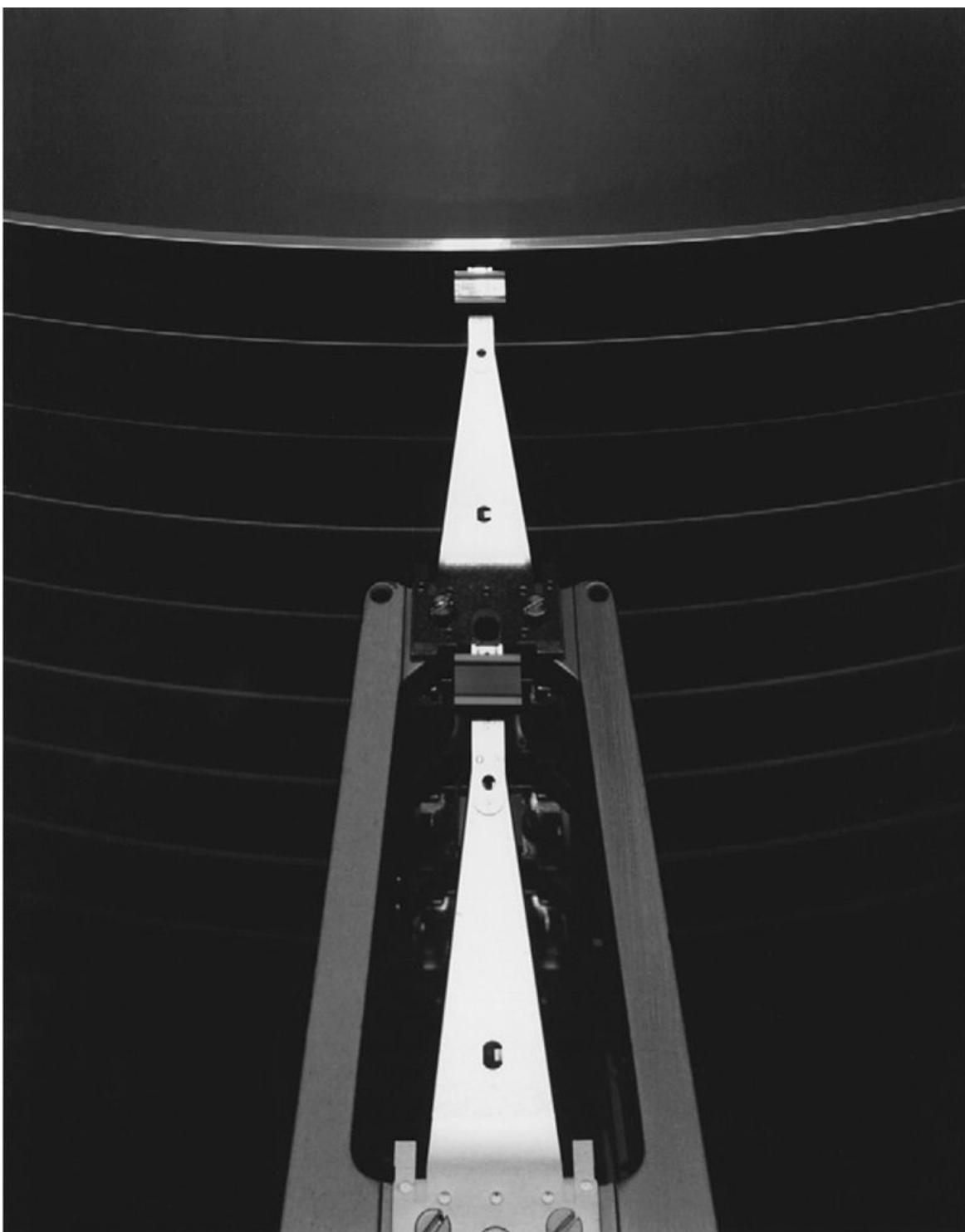


FIGURA 5.6 Um disco mostrando 10 placas e cabeças de leitura/gravação.

O diâmetro dos discos atualmente é de 2,5 ou 3,5 polegadas, e normalmente existem apenas uma ou duas placas por unidade.

Cada superfície do disco é dividida em círculos concêntricos, chamados de

trilhas. Normalmente, existem dezenas de milhares de trilhas por superfície. Cada trilha, por sua vez, é dividida em **setores** que contêm as informações; cada trilha pode ter milhares de setores. Os setores normalmente armazenam 512 a 4096 bytes. A sequência gravada no meio magnético é um número de setor, uma lacuna, a informação sobre esse setor, incluindo o código de correção de erro ([Seção 5.5](#)), uma lacuna, o número do próximo setor, e assim por diante.

trilha

Um de 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 que pode ser lida ou gravada em um disco.

As cabeças do disco para cada superfície são conectadas e se movem em conjunto, de modo que cada cabeça está posicionada 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 os dados, o sistema operacional precisa direcionar o disco através de um processo em três estágios. O primeiro passo é posicionar a cabeça sobre a trilha correta. Essa operação é chamada de **busca**, e o tempo para mover a cabeça até a trilha desejada é chamado de *tempo de busca*.

busca

O processo de posicionar uma cabeça de leitura/gravação sobre a trilha apropriada em um disco.

Os fabricantes de disco informam os tempos de busca mínimo, máximo e médio em seus manuais. Os dois primeiros são fáceis de medir, mas o tempo médio é aberto a várias interpretações, pois depende da distância da busca. A indústria calcula o tempo de busca médio como a soma do tempo para todas as buscas possíveis, dividido pelo número de buscas possíveis. Os tempos de busca médios normalmente são anunciados como algo entre 3 e 13 ms, mas,

dependendo da aplicação e do escalonamento de solicitações de disco, o tempo de busca médio real pode ser de apenas 25% a 33% do número anunciado, devido à localidade das referências ao disco. Essa localidade surge tanto devido a acessos sucessivos ao mesmo arquivo quanto porque o sistema operacional tenta escalarar esses acessos para que sejam feitos juntos.

Quando a cabeça tiver alcançado a trilha correta, temos que esperar até que o setor desejado gire sob a cabeça de leitura/escrita. Esse tempo é chamado de **latência rotacional** ou **atraso rotacional**. A latência média até a informação desejada é a metade da circunferência da trilha no disco. Os discos giram a 5400 RPM até 15.000 RPM. A latência rotacional média a 5400 RPM é

$$\text{Latência rotacional média} = \frac{0,5 \text{ rotação}}{5400 \text{ RPM}} = \frac{0,5 \text{ rotação}}{5400 \text{ RPM} / \left(60 \frac{\text{segundos}}{\text{minuto}} \right)} \\ = 0,0056 \text{ segundos} = 5,6 \text{ ms}$$

latência rotacional

Também chamado **atraso rotacional**. O tempo exigido para que o setor desejado de um disco gire sob a cabeça de leitura/gravação; normalmente considerado como metade do tempo de rotação.

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 em 2012 estavam entre 100 e 200 MB/segundo.

Uma complicação é que a maioria dos controladores de disco possui uma cache embutida que armazena os setores à medida que passam por eles; as taxas de transferência da cache normalmente são maiores, e eram de até 750 MB/segundo (6 Gbit/segundo) em 2012.

Infelizmente, o local onde os números de bloco estão localizados não é mais algo intuitivo. As suposições do modelo setor-trilha-cilindro, indicadas anteriormente, são que os blocos mais próximos estão na mesma trilha, que os blocos no mesmo cilindro levam menos tempo para serem acessados, pois não

há tempo de busca, e que algumas trilhas estão mais próximas do que outras. O motivo para a mudança foi o aumento do nível das interfaces de disco. Para agilizar as transferências sequenciais, essas interfaces de nível mais alto organizam os discos mais como fitas do que como dispositivos de acesso aleatório. Os blocos lógicos são ordenados em um padrão de serpentina por uma única superfície, tentando capturar todos os setores que são gravados na mesma densidade de bit e obter o máximo de desempenho. Logo, blocos sequenciais podem estar posicionados em diferentes trilhas.

Resumindo, as duas diferenças principais entre os discos magnéticos e as tecnologias de memória de semicondutor são que os discos possuem um tempo de acesso mais lento, pois são dispositivos mecânicos — a memória flash é 1000 vezes mais rápida e a DRAM é 100.000 vezes mais rápida —, embora sejam mais baratos por bit, pois possuem uma capacidade de armazenamento muito alta e um custo moderado — o disco é de 10 a 100 vezes mais barato. Os discos magnéticos são não voláteis, como a memória flash, mas, diferente dela, não há o problema de desgaste pela escrita. Porém, a memória flash é muito mais resistente e, portanto, combina muito mais com os impactos inerentes aos dispositivos móveis pessoais.

5.3. Princípios básicos de cache

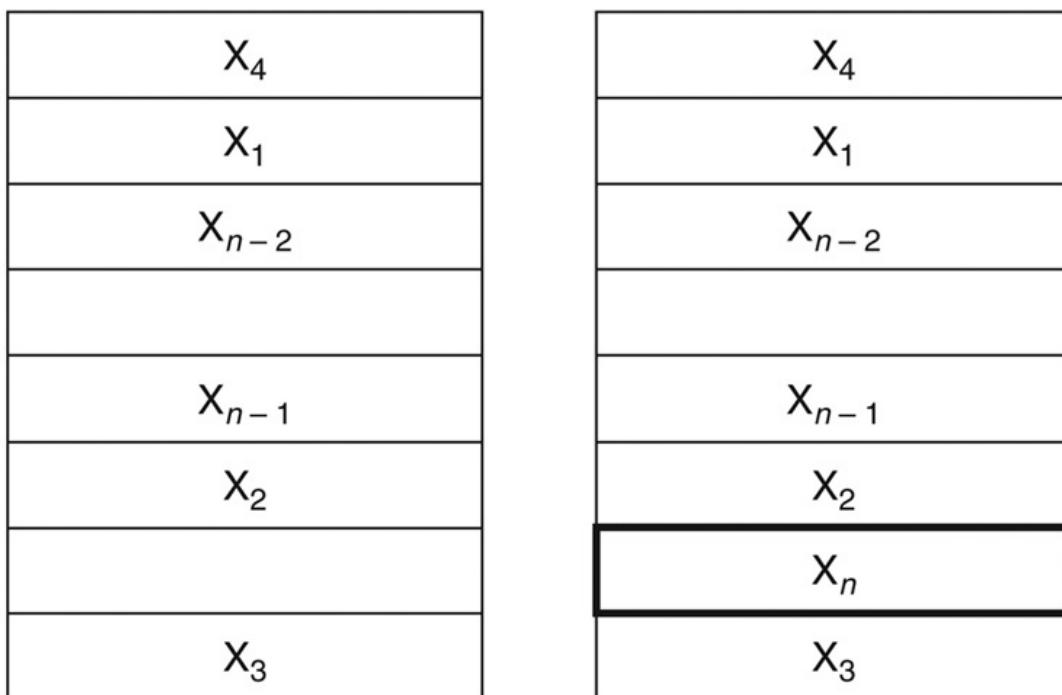
Cache: um lugar seguro para esconder ou guardar coisas.

Webster's New World Dictionary of the American Language, Third College Edition (1988)

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. As memórias no caminho de dados, no [Capítulo 4](#), são simplesmente substituídas por caches. 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 palavra e os blocos também consistem em uma única palavra. (Os leitores que já estão familiarizados com os fundamentos de cache podem pular para a [Seção 5.4](#).) A [Figura 5.7](#) 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, \dots, X_{n-1} , e o processador requisita uma palavra X_n que não está na cache. Essa requisição resulta em uma falha, e a palavra X_n é trazida da memória para a cache.



- a. Antes da referência para X_n b. Depois da referência para X_n

FIGURA 5.7 A cache, imediatamente antes e após uma referência a uma palavra 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.

Olhando o cenário na [Figura 5.7](#), surgem 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 palavra pode ficar exatamente em um lugar na cache, então, é fácil encontrar a palavra se ela estiver na cache. A maneira mais simples de atribuir um local na cache para cada palavra da memória é atribuir um local na cache baseado no *endereço* da palavra 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 a seguir para localizar um bloco:

(Endereço de bloco) módulo (Número de blocos na cache)

cache de mapeamento direto

Uma estrutura de cache em que cada local da memória é mapeado exatamente para um local na cache.

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) do endereço. Assim, a cache de 8 blocos usa os três bits menos significativos ($8 = 2^3$) do endereço do bloco. Por exemplo, a [Figura 5.8](#) 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.

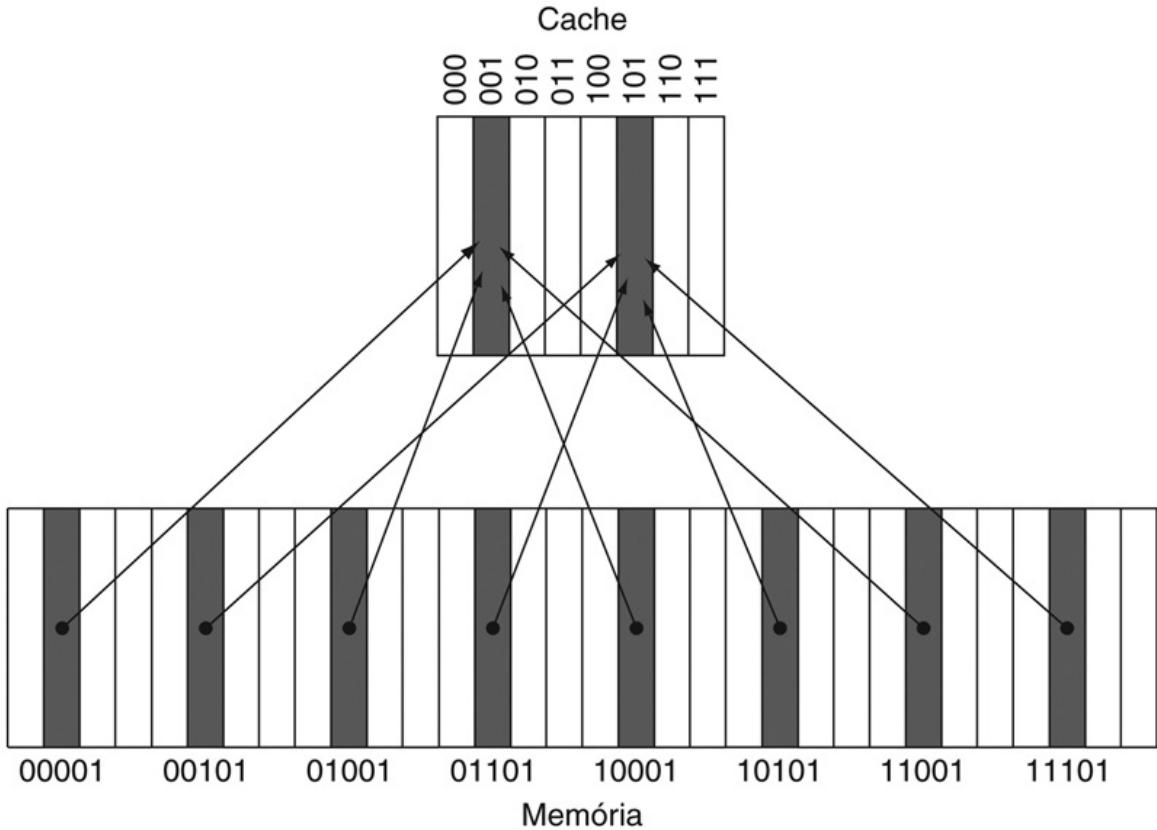


FIGURA 5.8 Uma cache diretamente mapeada com oito entradas mostrando os endereços das palavras de memória entre 0 e 31 que mapeiam para os mesmos locais de cache.

Como há oito palavras na cache, um endereço X é mapeado para a palavra 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.

Como cada local da cache pode armazenar o conteúdo de diversos locais diferentes da memória, como podemos saber se os dados na cache correspondem a uma palavra requisitada? Ou seja, como sabemos se uma palavra 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 palavra na cache corresponde à palavra 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 5.8, precisamos apenas ter os dois bits mais significativos dos cinco bits de endereço

na tag, já que o campo índice com os três bits menos significativos do endereço seleciona o bloco. Os arquitetos omitem os bits de índice porque eles são redundantes, uma vez que, por definição, o campo índice de qualquer endereço de um bloco de cache precisa ser o número daquele bloco.

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 palavra requisitada.

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 5.7](#). Portanto, precisamos saber se a tag deve ser ignorada para essas entradas. O método mais comum é incluir um **bit de validade** indicando se uma entrada contém um endereço válido. Se o bit não estiver marcado, não pode haver uma correspondência para esse bloco.

bit de validade

Um campo nas tabelas de uma hierarquia de memória que indica que o bloco associado na hierarquia contém dados válidos.

No restante desta seção, vamos nos concentrar em explicar como uma cache trata das 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.

Colocando em perspectiva



O caching talvez seja o exemplo mais importante da grande ideia da **predição**. Ele conta com o princípio da localidade para tentar encontrar os dados desejados nos níveis mais altos da hierarquia de memória, e oferece mecanismos para garantir que, quando a predição for errada, ela encontra e usa os dados apropriados dos níveis mais baixos da hierarquia da memória. As taxas de acerto da predição de cache nos computadores modernos, normalmente, são mais altas do que 95% (Figura 5.47).

Acessando uma cache

A seguir, vemos uma sequência de nove referências da memória a uma cache vazia de oito blocos, incluindo a ação para cada referência. A [Figura 5.9](#) mostra como o conteúdo da cache muda em cada falha. Como há oito blocos na cache, os três 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. O estado inicial do cache depois de power-on

Índice	V	Tag	Dados
000	N		
001	N		
010	Y	11_{bin}	Memória (11010_{bin})
011	N		
100	N		
101	N		
110	Y	10 _{bin}	Memória (10110 _{bin})
111	N		

c. Depois de lidar com uma falta de endereço (11010_{bin})

Índice	V	Tag	Dados
000	Y	10 _{bin}	Memória (10000 _{bin})
001	N		
010	Y	11 _{bin}	Memória (11010 _{bin})
011	Y	00_{bin}	Memória (00011_{bin})
100	N		
101	N		
110	Y	10 _{bin}	Memória (10110 _{bin})
111	N		

e. Depois de lidar com uma falta de endereço (00011_{bin})

Índice	V	Tag	Dados
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{bin}	Memória (10110_{bin})
111	N		

b. Depois de lidar com uma falta de endereço (10110_{bin})

Índice	V	Tag	Dados
000	Y	10_{bin}	Memória (10000_{bin})
001	N		
010	Y	11 _{bin}	Memória (11010 _{bin})
011	N		
100	N		
101	N		
110	Y	10 _{bin}	Memória (10110 _{bin})
111	N		

d. Depois de lidar com uma falta de endereço (10000_{bin})

Índice	V	Tag	Dados
000	Y	10 _{bin}	Memória (10000 _{bin})
001	N		
010	Y	10_{bin}	Memória (10010_{bin})
011	Y	00 _{bin}	Memória (00011 _{bin})
100	N		
101	N		
110	Y	10 _{bin}	Memória (10110 _{bin})
111	N		

f. Depois de lidar com uma falta de endereço (10010_{bin})

FIGURA 5.9 O conteúdo da cache é mostrado para cada requisição de referência que falha, com os campos índice e tag mostrados em binário para a sequência de endereços no início desse tópico.

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 sequência 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 palavra 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_{bin} possui tag 10_{bin}

e corresponde ao endereço 10010_{bin} .

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 (5.6b)	$(10110_{\text{bin}} \bmod 8) = 110_{\text{bin}}$
26	11010_{bin}	falha (5.6c)	$(11010_{\text{bin}} \bmod 8) = 010_{\text{bin}}$
22	10110_{bin}	Acerto	$(10110_{\text{bin}} \bmod 8) = 110_{\text{bin}}$
26	11010_{bin}	Acerto	$(11010_{\text{bin}} \bmod 8) = 010_{\text{bin}}$
16	10000_{bin}	falha (5.6d)	$(10000_{\text{bin}} \bmod 8) = 000_{\text{bin}}$
3	00011_{bin}	falha (5.6e)	$(00011_{\text{bin}} \bmod 8) = 011_{\text{bin}}$
16	10000_{bin}	Acerto	$(10000_{\text{bin}} \bmod 8) = 000_{\text{bin}}$
18	10010_{bin}	falha (5.6f)	$(10010_{\text{bin}} \bmod 8) = 010_{\text{bin}}$
16	10000_{bin}	Acerto	$(10000_{\text{bin}} \bmod 8) = 000_{\text{bin}}$

Como a cache está vazia, várias das primeiras referências são falhas; a legenda da [Figura 5.9](#) descreve as ações de cada referência à memória. Na oitava referência, temos demandas em conflito para um bloco. A palavra no endereço 18 (10010_{bin}) deve ser trazida para o bloco de cache 2 (010_{bin}). Logo, ela precisa substituir a palavra no endereço 26 (11010_{bin}), que já está no bloco de cache 2 (010_{bin}). Esse comportamento permite que uma cache tire proveito da localidade temporal: palavras recentemente acessadas substituem palavras 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 que já esteja na sua mesa 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 5.10](#) mostra como um endereço referenciado é dividido em:

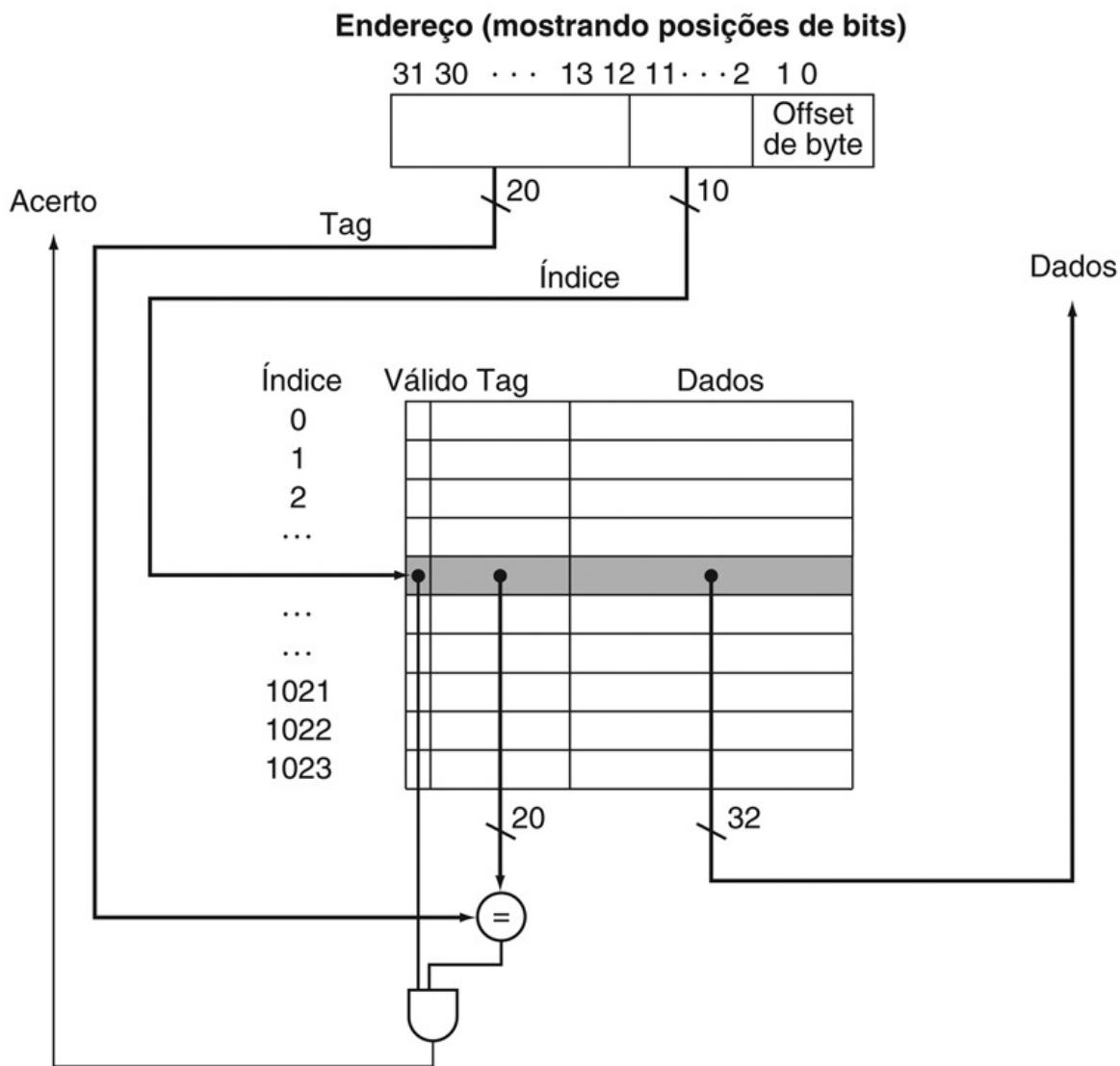


FIGURA 5.10 Para esta cache, a parte inferior do endereço é usada para selecionar uma entrada de cache consistindo em uma palavra de dados e uma tag.

Essa cache mantém 1024 palavras ou 4 KiB. Consideramos endereços de 32 bits neste capítulo. 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) palavras e um tamanho de bloco de 1 palavra, 10 bits são usados para indexar a cache, deixando $32 - 10 = 20$ 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 palavra é fornecida para o processador. Caso contrário, ocorre uma falha.

- Um *campo tag*, usado para ser comparado com o valor do campo tag da cache;
- Um *índice de cache*, usado para selecionar o bloco.

O índice de um bloco de cache, juntamente com o conteúdo da tag desse bloco, especifica de modo único o endereço de memória da palavra 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 palavras são alinhadas como múltiplos de 4 bytes, os dois bits menos significativos de cada endereço especificam um byte dentro de uma palavra e, portanto, são ignorados ao selecionar uma palavra 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 palavra, mas normalmente é de várias palavras. Para as situações a seguir:

- Endereços em bytes de 32 bits.
- Uma cache diretamente mapeada.
- O tamanho da cache é 2^n blocos, de modo que n bits são usados para o índice.
- O tamanho do bloco é 2^m palavras (2^{m+2} bytes), de modo que m bits são usados para a palavra dentro do bloco, e dois bits são usados para a parte de byte do endereço o tamanho do campo de tag é

$$32 - (n + m + 2).$$

O número total de bits em uma cache diretamente mapeada é

$$2^n \times (\text{tamanho do bloco} + \text{tamanho da tag} + \text{tamanho do campo de validade}).$$

Como o tamanho do bloco é 2^m palavras (2^{m+5} bits) e precisamos de 1 bit para o campo de validade, o número de bits nessa cache é

$$2^n \times (2^m \times 32 + (32 - n - m - 2) + 1) = 2^n \times (2^m \times 32 + 31 - n - m).$$

Embora esse seja o tamanho real em bits, a convenção de nomeação é excluir

o tamanho da tag e do campo de validade e contar apenas o tamanho dos dados. Assim, a cache na [Figura 5.10](#) é chamada de cache de 4 KiB.

Bits em uma cache

Exemplo

Quantos bits no total são necessários para uma cache diretamente mapeada com 16 KiB de dados e blocos de 4 palavras, considerando um endereço de 32 bits?

Resposta

Sabemos que 16 KiB são 4096 (2^{12}) palavras. Com um tamanho de bloco de 4 palavras (2^2), há 1024 (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

$$2^{10} \times (4 \times 32 + (32 - 10 - 2 - 2) + 1) = 2^{10} \times 147 = 147 \text{ Kibibits}$$

ou 18,4 KiB para uma cache de 16 KiB.. Para essa cache, o número total de bits na cache é aproximadamente 1,15 vezes o necessário apenas para o armazenamento dos dados.

Mapeando um endereço para um bloco de cache multipalavra

Exemplo

Considere uma cache com 64 blocos e um tamanho de bloco de 16 bytes. Para qual número de bloco o endereço em bytes 1200 é mapeado?

Resposta

A fórmula foi vista no início da Seção 5.2. O bloco é dado por

$$(\text{Endereço do bloco}) \bmod (\text{Número de blocos de cache})$$

Em que 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

$$\left[\frac{\text{Endereço em bytes}}{\text{Bytes por bloco}} \right] \times \text{Bytes por bloco}$$

e

$$\left[\frac{\text{Endereço em bytes}}{\text{Bytes por bloco}} \right] \times \text{Bytes por bloco} + (\text{Bytes por bloco} - 1)$$

Portanto, com 16 bytes por bloco, o endereço em bytes 1200 é o endereço de bloco

$$\left[\frac{1200}{6} \right] = 75$$

que é mapeado para o número de bloco de cache (75 módulo 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 5.11](#), 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 palavras sejam acessadas. Explicando de outra forma: a localidade espacial entre as palavras em um bloco diminui com um bloco muito grande; por conseguinte, os benefícios na taxa de falhas se tornam menores.

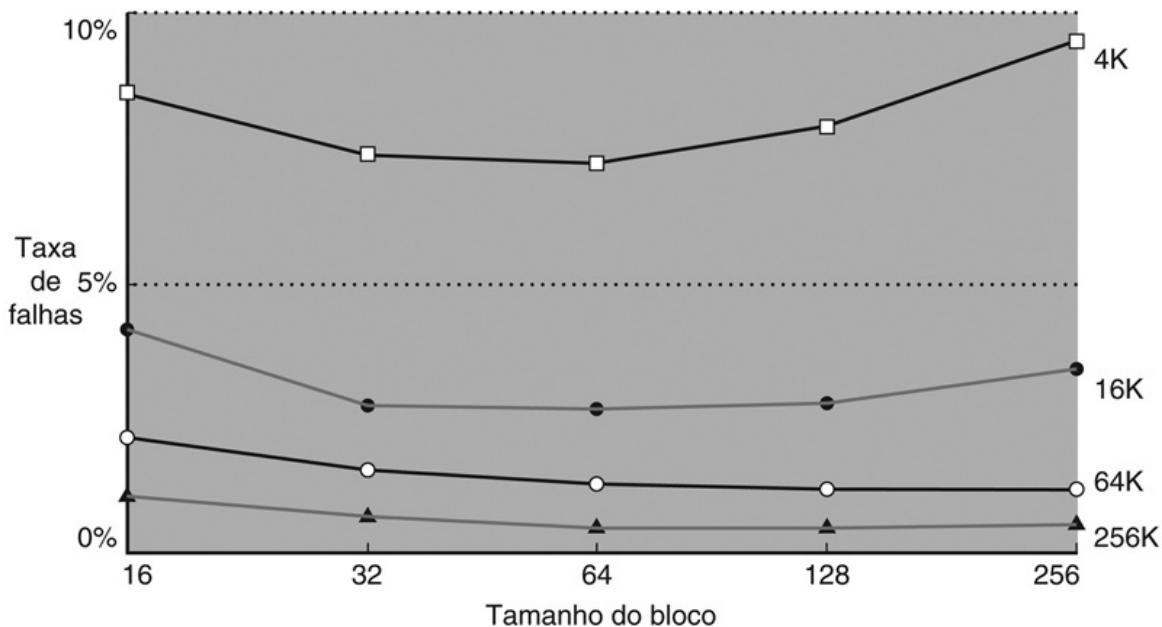


FIGURA 5.11 Taxa de falhas versus tamanho de bloco.

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 trases do SPEC CPU2000 levariam tempo demais se o tamanho de bloco fosse incluído; portanto, esses dados são baseados no SPEC92.

Um problema mais sério associado ao aumento do 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 palavra 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 penalidade de falha — provavelmente aumentará conforme o tamanho do bloco aumenta. Além disso, o aumento na taxa de falhas começa a reduzir 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. Naturalmente, se projetarmos a memória para transferir blocos maiores de forma mais eficiente, poderemos aumentar o tamanho do bloco e obter mais melhorias no desempenho da cache. Discutiremos esse assunto na próxima seção.

Detalhamento

Embora seja difícil fazer algo sobre o componente de latência mais longo da penalidade de falha para blocos grandes, podemos ser capazes de ocultar um pouco do tempo de transferência, de modo que a penalidade de falha seja efetivamente menor. O método mais simples de fazer isso, chamado *reinício precoce*, é simplesmente retomar a execução assim que a palavra requisitada do bloco 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 sequenciais, se o sistema de memória puder entregar uma palavra a cada ciclo de clock, o processador poderá ser capaz de reiniciar sua operação quando a palavra requisitada for retornada, com o sistema de memória entregando novas palavras de instrução em tempo. Essa técnica normalmente é menos eficaz para caches de dados porque é provável que as palavras sejam requisitadas do bloco de uma maneira menos previsível; além disso, a probabilidade de que o processador precise de outra palavra 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 palavra requisitada seja transferida da memória para a cache primeiro. O restante do bloco, então, é transferido, começando com o endereço após a palavra requisitada e retornando para o início do bloco. Essa técnica, chamada *palavra requisitada primeiro*, ou *palavra 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

Antes de olharmos a cache de um sistema real, vamos ver como a unidade de controle lida com as **falhas de cache**. (Descrevemos um controlador de cache na [Seção 5.9](#)). A unidade de controle precisa detectar uma falha de cache 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.

fallha de cache

Uma requisição de dados da cache que não pode ser atendida porque os dados não estão presentes na cache.

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 ([Capítulo 4](#)), ao contrário de 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. Processadores fora de ordem mais sofisticados podem permitir a execução de instruções enquanto se espera por uma falha de cache, mas vamos considerar nesta seção os processadores em ordem, que fazem um stall nas perdas de cache.

Vejamos um pouco mais de perto como as falhas de instrução são tratadas; 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 ao realizar uma leitura. Como o contador do programa é incrementado no primeiro ciclo de clock da execução, o endereço da instrução que gera uma falha de cache de instruções é igual ao valor do contador de programa menos 4. Uma vez que tenhamos 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 palavras com a instrução desejada 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, desta 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**.

write-through

Um esquema em que as escritas sempre atualizam a cache e o próximo nível inferior da hierarquia de memória, garantindo que os dados sejam sempre consistentes entre os dois.

O outro aspecto importante das escritas é o que ocorre em uma falha de dados. Primeiro, buscamos as palavras do bloco da memória. Após o bloco ser buscado e colocado na cache, podemos substituir (sobrescrever) a palavra que causou a falha no bloco de cache. Também escrevemos a palavra 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