

Coleção
SCHAUM

ARQUITETURA DE COMPUTADORES

Nicholas Carter

● Cobertura completa do projeto de hardware e
de software para sistemas de computadores

● 192 problemas resolvidos

● Opções de projeto reais explicados
passo a passo



Bookman

MAIS DE
30 MILHÕES DE
EXEMPLARES VENDIDOS
NO MUNDO

Material com direitos autorais

Obra originalmente publicada sob o título

Schaum's Outline of Theory and Problems of Computer Architecture

© 2002, McGraw-Hill Companies, Inc.

All rights reserved.

ISBN 0-07-136207-X

Capa: Rogério Grilho

Leitura final: Luciano Gomes

Supervisão editorial: Denise Weber Nowaczyk

Editoração eletrônica: Laser House

NICHOLAS P. CARTER é professor assistente no Departamento de Engenharia Elétrica e de Computadores na Universidade de Illinois em Urbana-Champaign. Ele é Ph.D. em Engenharia Elétrica e Ciência dos Computadores pelo Instituto de Tecnologia de Massachusetts. Seus graus de bacharel e de mestre também são daquela instituição. Os interesses de pesquisa do Dr. Carter são em arquitetura de computadores, em especial a interação da tecnologia de fabricação e a arquitetura dos computadores, bem como o projeto de sistemas de computadores utilizando tecnologias não tradicionais de fabricação. Ele recebeu vários prêmios, incluindo o AASERT *fellowship*, e foi nomeado Collins Scholar pela Universidade de Illinois.

Os produtos e as marcas utilizadas neste livro podem ser marcas registradas ou patentes. Onde acreditamos ser esse o caso, foi utilizada a primeira letra em maiúsculo ou no estilo adotado pelo proprietário da marca. Não houve intenção, por parte dos editores, de endosso ou vínculo com os proprietários das patentes. Nem o autor nem os editores tiveram a intenção de expressar qualquer juízo de valor no que diz respeito à validade ou ao status legal das marcas em questão.

Reservados todos os direitos de publicação, em língua portuguesa, à

ARTMED® EDITORA S.A.

(BOOKMAN® COMPANHIA EDITORA é uma divisão da ARTMED® EDITORA S.A.)

Av. Jerônimo de Ornelas, 670 – Santana

90040-340 – Porto Alegre – RS

Fone: (51) 3330-3444 Fax: (51) 3330-2378

É proibida a duplicação ou reprodução deste volume, no todo ou em parte, sob quaisquer formas ou por quaisquer meios (eletrônico, mecânico, gravação, fotocópia, distribuição na Web e outros), sem permissão expressa da Editora.

SÃO PAULO

Av. Rebouças, 1.073 – Jardins

05401-150 – São Paulo – SP

Fone: (11) 3062-3757* Fax: (11) 3062-2487

SAC 0800 703-3444

IMPRESSO NO BRASIL

PRINTED IN BRAZIL

Sumário

CAPÍTULO 1	Introdução	11
1.1	Objetivo deste Livro	11
1.2	Conhecimentos Presumidos	11
1.3	Escopo	11
1.4	Objetivos do Capítulo	12
1.5	Tendências Tecnológicas	12
1.6	Medindo o Desempenho	12
1.7	Aceleração	14
1.8	A Lei de Amdahl	15
1.9	Resumo	15
CAPÍTULO 2	Representações de Dados e Aritmética de Computadores	22
2.1	Objetivos	22
2.2	De Elétrons a Bits	22
2.3	Representação Binária de Inteiros Positivos	23
2.4	Operações Aritméticas com Inteiros Positivos	24
2.5	Inteiros Negativos	28
2.6	Números em Ponto Flutuante	31
2.7	Resumo	36
CAPÍTULO 3	Organização de Computadores	43
3.1	Objetivos	43
3.2	Introdução	43
3.3	Programas	44
3.4	Sistemas Operacionais	47
3.5	Organização dos Computadores	49
3.6	Resumo	52
CAPÍTULO 4	Modelos de Programação	56
4.1	Objetivos	56
4.2	Introdução	56
4.3	Tipos de Instruções	57

<u>4.4 Arquiteturas Baseadas em Pilha</u>	61
<u>4.5 Arquiteturas Baseadas em Registradores de Uso Geral</u>	68
<u>4.6 Comparando Arquiteturas Baseadas em Pilha e em Registradores de Uso Geral</u>	71
<u>4.7 Utilizando Pilhas para Implementar Chamadas de Procedimentos</u>	72
<u>4.8 Resumo</u>	74
CAPÍTULO 5 Projeto de Processadores	80
<u>5.1 Objetivos</u>	80
<u>5.2 Introdução</u>	80
<u>5.3 Arquitetura do Conjunto de Instruções</u>	81
<u>5.4 Microarquitetura de Processadores</u>	87
<u>5.5 Resumo</u>	90
CAPÍTULO 6 Utilização de <i>Pipelines</i>	96
<u>6.1 Objetivos</u>	96
<u>6.2 Introdução</u>	96
<u>6.3 Pipelining</u>	97
<u>6.4 Riscos de Dependências entre Instruções e o seu Impacto sobre a Taxa de Rendimento</u>	100
<u>6.5 Prevendo o Tempo de Execução em Processadores com <i>Pipelines</i></u>	104
<u>6.6 Transmissão de Resultados (<i>Bypassing</i>)</u>	107
<u>6.7 Resumo</u>	109
CAPÍTULO 7 Paralelismo no Nível da Instrução	118
<u>7.1 Objetivos</u>	118
<u>7.2 Introdução</u>	118
<u>7.3 O que é Paralelismo no Nível da Instrução?</u>	120
<u>7.4 Limitações do Paralelismo no Nível da Instrução</u>	120
<u>7.5 Processadores Superescalares</u>	121
<u>7.6 Execução Em-Ordem <i>versus</i> Fora-de-Ordem</u>	122
<u>7.7 Renomeação de Registradores</u>	124
<u>7.8 Processadores PIML</u>	126
<u>7.9 Técnicas de Compilação para Paralelismo no Nível da Instrução</u>	128
<u>7.10 Resumo</u>	131
CAPÍTULO 8 Sistemas de Memória	141
<u>8.1 Objetivos</u>	141
<u>8.2 Introdução</u>	141
<u>8.3 Latência, Taxa de Transferência e Largura de Banda</u>	141
<u>8.4 Hierarquias de Memória</u>	144
<u>8.5 Tecnologias de Memória</u>	146
<u>8.6 Resumo</u>	156
CAPÍTULO 9 Caches	158
<u>9.1 Objetivos</u>	158
<u>9.2 Introdução</u>	158
<u>9.3 Caches de Dados, de Instruções e Unificadas</u>	159
<u>9.4 Descrevendo Caches</u>	160
<u>9.5 Capacidade</u>	160
<u>9.6 Comprimento de Linha</u>	160

9.7	Associatividade	162
9.8	Política de Substituição	165
9.9	<i>Caches Write-Back versus Write-Through</i>	166
9.10	Implementações de <i>Caches</i>	167
9.11	Matrizes de Etiquetas	167
9.12	Lógica de Acertos/Faltas	169
9.13	Matrizes de Dados	170
9.14	Categorizando Faltas de <i>Cache</i>	170
9.15	<i>Caches</i> em Vários Níveis	171
9.16	Resumo	172
CAPÍTULO 10 Memória Virtual		181
10.1	Objetivos	181
10.2	Introdução	181
10.3	Tradução de Endereços	183
10.4	<i>Paginação por Demanda versus Swapping</i>	184
10.5	Tabelas de Páginas	184
10.6	<i>Translation Lookaside Buffers</i>	190
10.7	Proteção	192
10.8	<i>Caches e Memória Virtual</i>	194
10.9	Resumo	195
CAPÍTULO 11 Entrada e Saída		201
11.1	Objetivos	201
11.2	Introdução	201
11.3	Barramentos de E/S	202
11.4	Interrupções	204
11.5	E/S Mapeada em Memória	206
11.6	Acesso Direto à Memória	208
11.7	Dispositivos de E/S	209
11.8	Discos Magnéticos	210
11.9	Resumo	213
CAPÍTULO 12 Multiprocessadores		220
12.1	Objetivos	220
12.2	Introdução	220
12.3	Aceleração e Desempenho	221
12.4	Multiprocessadores	223
12.5	Sistemas Baseados em Troca de Mensagens	225
12.6	Sistemas de Memória Compartilhada	226
12.7	<i>Comparando Memória Compartilhada e Troca de Mensagens</i>	231
12.8	Resumo	232
ÍNDICE		239

Capítulo 1

Introdução

1.1 OBJETIVO DESTE LIVRO

O objetivo deste livro é ser utilizado como um texto de acompanhamento para cursos introdutórios sobre arquitetura de computadores, em nível de graduação ou em cursos de pós-graduação. O seu público principal são estudantes de cursos com disciplinas de arquitetura de computadores que estejam interessados em explicações adicionais, problemas práticos e exemplos a serem utilizados para melhorar a sua compreensão do material, ou na preparação de exercícios.

1.2 CONHECIMENTOS PRESUMIDOS

Este livro presume que o leitor tenha conhecimentos equivalentes àqueles de alunos de segundo ou terceiro ano de Engenharia Elétrica, ou de programas de Ciência da Computação que ainda não tenham tido uma disciplina sobre organização ou arquitetura de computadores. Presume-se uma familiaridade básica com a operação de computadores e sua terminologia, assim como alguma familiaridade com programação em linguagens de alto nível.

1.3 ESCOPO

Este livro cobre uma gama de tópicos ligeiramente mais ampla que a maioria das disciplinas sobre arquitetura de computadores, com um semestre de duração, a fim de aumentar a sua utilidade. Os leitores descobrirão que o material adicional é útil como uma revisão ou como uma introdução para tópicos mais avançados. O livro começa com uma discussão sobre representação de dados e aritmética de computadores, seguida por capítulos sobre a organização de computadores e os modelos de programação. Três capítulos são dedicados à discussão de projeto de processadores, incluindo *pipelining* e paralelismo ao nível da instrução. Estes são seguidos por três capítulos sobre sistemas de memória, incluindo memória virtual e *caches*. Os dois últimos capítulos discutem E/S e fornecem uma introdução aos multiprocessadores.

1.4 OBJETIVOS DO CAPÍTULO

A meta deste capítulo é preparar o leitor para os conteúdos apresentados em capítulos posteriores, ao discutir as tecnologias básicas que determinam o desempenho de computadores e as técnicas utilizadas para medir e discutir o desempenho. Após ler este capítulo e completar os exercícios, o estudante deverá:

1. Compreender e ser capaz de discutir as taxas históricas do aperfeiçoamento da densidade de transistores, do desempenho de circuitos e do desempenho geral de sistemas.
2. Compreender métodos comuns para avaliar o desempenho de computadores.
3. Ser capaz de calcular como as modificações em uma parte de um sistema de computadores afetam o desempenho geral.

1.5 TENDÊNCIAS TECNOLÓGICAS

Desde o início da década de 1980, o desempenho dos computadores tem sido impulsionado por aperfeiçoamentos nas capacidades dos circuitos integrados utilizados para implementar microprocessadores, nos *chips* de memória e em outros componentes de computadores. Ao longo do tempo, os circuitos integrados foram aperfeiçoados em *densidade* (quantos transistores e ligações podem ser colocados em uma área fixa em um *chip* de silício), *velocidade* (a rapidez com que as portas lógicas básicas e dispositivos de memória operam) e a *área* (o tamanho físico do maior circuito integrado que pode ser fabricado).

O impressionante crescimento do desempenho de computadores ao longo das últimas duas décadas foi impulsionado pelo fato de que a velocidade e a densidade dos *chips* foram aperfeiçoados geometricamente, em vez de linearmente. Isto significa que uma melhoria no desempenho de um ano com relação ao seguinte tem sido uma parcela relativamente constante do desempenho do ano anterior, em vez de um valor absoluto constante. Em média, o número de transistores que podem ser produzidos sobre um *chip* de silício aumentou cerca de 50% ao ano, e a velocidade dos transistores aumentou tanto que o atraso de uma porta lógica básica (E, OU, etc.) diminuiu 13% ao ano. A observação de que o desempenho dos computadores melhora geométrica e não linearmente é freqüentemente citada como a *Lei de Moore*.

Exemplo A quantidade de dados que pode ser armazenada em um *chip* de memória RAM dinâmica (DRAM) quadruplicou a cada três anos desde o final da década de 1970, em uma taxa de crescimento anual de 60%.

Do final da década de 1970 até o final da década de 1980, o desempenho dos microprocessadores foi impulsionado principalmente pelos aperfeiçoamentos na tecnologia de fabricação e foi melhorado a uma taxa de 35% ao ano. Desde então, a taxa de aperfeiçoamento efetivamente cresceu para mais de 50% ao ano, embora a taxa do progresso na fabricação de semicondutores tenha permanecido relativamente constante. O aumento na taxa de desempenho tem sido devido a melhorias na arquitetura e na organização de computadores – os projetistas de computadores têm sido capazes de tirar proveito da crescente densidade dos circuitos integrados para acrescentar recursos aos microprocessadores e aos sistemas de memória, os quais proporcionam um desempenho acima do aumento da velocidade dos transistores que os implementam.

1.6 MEDINDO O DESEMPENHOO

Neste capítulo, já discutimos como o desempenho dos computadores melhorou ao longo do tempo, mas sem dar uma definição formal do que é desempenho. Isso se deve, em parte, ao fato de o termo *desempenho* ser muito vago quando utilizado no contexto de sistemas de computadores. Geralmente, o desempenho descreve a rapidez com a qual um determinado sistema pode executar um programa ou programas. Sistemas que executam programas em menos tempo são ditos de melhor desempenho.

A melhor medida para o desempenho de um computador é o tempo de execução de um programa, ou programas, que o usuário deseja executar. Mas geralmente é impraticável testar todos os programas que serão executados em um dado sistema antes de decidir qual computador comprar, ou quando se toma decisões de projeto. Assim, os projetistas de computadores produziram um certo número de unidades para descrever o desempenho de computadores, algumas das quais serão discutidas neste capítulo. Os projetistas também imaginaram uma certa quantidade de unidades para o desempenho de subsistemas individuais dos computadores, as quais serão discutidas nos capítulos que tratam desse assunto.

Tenha em mente que, além do desempenho, muitos fatores podem influenciar decisões de projeto ou de compra. Facilidade de programação é uma preocupação importante, pois o tempo e os gastos necessários para desenvolver programas que sejam úteis podem ser mais significativos do que a diferença nos tempos de execução dos mesmos, uma vez que eles tenham sido desenvolvidos. A questão da compatibilidade também é importante; a maioria dos programas é vendida como imagens binárias que somente poderão ser executadas em uma família de processadores em particular. Se o programa que você precisa não é executável em um dado sistema, não importa a rapidez com que o sistema execute outros programas.

MIPS

Uma medida antiga do desempenho de computadores é a taxa pela qual a máquina executa instruções. Isto é calculado dividindo-se o número de instruções executadas em um programa pelo tempo necessário para executá-lo, e é tipicamente expresso em *milhões de instruções por segundo* (MIPS). Essa medida caiu em desuso porque não leva em conta o fato de que diferentes sistemas freqüentemente precisam de números diferentes de instruções para implementar um dado programa. A taxa de MIPS de um computador nada diz a respeito de quantas instruções são necessárias para executar uma dada tarefa, tornando-a menos útil do que outras unidades para comparar o desempenho de diferentes sistemas.

CPI/IPC

Outra unidade utilizada para descrever o desempenho de computadores é o número de ciclos de relógio necessário para executar cada instrução, conhecido como *ciclos por instrução*, ou CPI. O CPI de um programa, em um sistema, é calculado dividindo-se o número de ciclos de relógio necessários para executar o programa pelo número de instruções executadas. Para sistemas que podem executar mais de uma instrução por ciclo, o número de *instruções executadas por ciclo*, ou IPC, é freqüentemente utilizada, em vez do CPI. O IPC é calculado dividindo-se o número de instruções executadas, ao se executar um programa, pelo número de ciclos de relógio necessários para executar o programa, e é o recíproco do CPI. Essas duas unidades fornecem a mesma informação e a escolha de qual usar geralmente é baseada em qual dos valores é maior do que o número 1. Quando se utiliza o IPC e o CPI para comparar sistemas, é importante lembrar que valores altos de IPC indicam que o programa de referência demorou menos ciclos para ser executado do que valores baixos de IPC, enquanto que valores altos de CPI indicam que foram necessários mais ciclos do que valores baixos de CPI. Assim, um IPC alto tende a indicar bom desempenho, um CPI alto indica um desempenho fraco.

Exemplo Um dado programa consiste de um laço de 100 instruções que é executado 42 vezes. Se demora 16.000 ciclos para executar o programa em um dado sistema, quais são os valores de CPI e de IPC do sistema para este programa?

Solução

O laço de 100 instruções é executado 42 vezes, de modo que o número total de instruções executadas é $100 \times 42 = 4200$. Demora 16.000 ciclos para executar o programa, de modo que o CPI é $16.000 / 4200 = 3.81$. Para calcular o IPC, dividimos 4200 instruções por 16.000 ciclos, obtendo um IPC de 0,26.

Em geral, o IPC e o CPI são medidas ainda menos úteis do desempenho de sistemas atuais do que o MIPS, porque eles não contêm qualquer informação a respeito da freqüência do relógio do sistema ou de quantas instruções o sistema exige para executar uma tarefa. Se você conhece a taxa de MIPS de um sistema em um programa, você pode multiplicá-la pelo número de instruções executadas nele para determinar quanto tempo demorou para ser completado. Se você conhece o CPI de um sistema em um dado programa, você pode multiplicá-lo pelo número de instruções para obter o número de ciclos que demorou para completar o programa, mas você tem que saber o número de ciclos por segundo (a freqüência de relógio do sistema) para converter isto na quantidade de tempo necessário para executar o programa.

Como resultado, o CPI e o IPC raramente são utilizados para comparar sistemas de computadores atuais. No entanto, são unidades muito comuns na pesquisa de arquitetura de computadores, porque a maior parte desse tipo de pesquisa é feita utilizando programas que simulam uma arquitetura em especial, para estimar quantos ciclos um dado programa irá utilizar para ser executado naquela arquitetura. Esses simuladores geralmente são incapazes de prever o ciclo de tempo dos sistemas que eles simulam, de modo que o CPI/IPC é a melhor estimativa de desempenho disponível.

Conjuntos de *Benchmark*

Como discutimos, tanto o MIPS quanto o CPI/IPC têm limitações significativas como medidas de desempenho de computadores. Conjuntos de *benchmark* (medição de desempenho) são uma terceira medida de desempenho de computadores e foram desenvolvidas para resolver as limitações do MIPS e do CPI/IPC.

Um conjunto de *benchmark* consiste de uma série de programas que acredita-se ser o correspondente típico de programas que serão executados no sistema. A pontuação de um sistema no conjunto de *benchmark* é baseada em quanto tempo o sistema demora para executar todos os programas que o compõem. Existem muitos conjuntos de *benchmark* diferentes, que geram estimativas do desempenho de um sistema com diferentes tipos de aplicações.

Um dos conjuntos de *benchmark* mais conhecidos é a suíte SPEC, produzida pela Standard Performance Evaluation Corporation. Sua versão atual, por ocasião da publicação deste livro, é a "SPEC CPU2000 *benchmark*", a terceira principal revisão desde que o conjunto de *benchmark* SPEC foi publicado em 1989.

Os conjuntos de *benchmark* fornecem várias vantagens sobre MIPS e CPI/IPC. Primeiro, os seus resultados de desempenho são baseados em tempos totais de execução, não na taxa de execução de instruções. Segundo, elas fazem uma média do desempenho do sistema por vários programas, de modo a gerar uma estimativa da sua velocidade de média. Isto jorna a avaliação geral do sistema em um conjunto de *benchmark* um indicador melhor do seu desempenho geral do que é a avaliação MIPS em qualquer programa isolado. Além disto, muitos *benchmarks* exigem que os fabricantes publiquem os resultados dos seus sistemas com programas individuais do *benchmark*, bem como a pontuação geral do sistema no conjunto de *benchmark*, tornando possível fazer uma comparação direta de resultados individuais do *benchmark*, se você sabe que um sistema será utilizado para uma aplicação em especial.

Média Geométrica versus Média Aritmética

Muitos conjuntos de *benchmark* utilizam a média *geométrica*, em vez da média *aritmética*, para fazer a média dos resultados dos programas contidos no conjunto de *benchmark* porque um único valor extremo tem um impacto menor sobre a média geométrica de uma série do que sobre a média aritmética. Utilizar a média geométrica torna mais difícil para um sistema atingir uma pontuação alta no *benchmark*, ao atingir um bom desempenho em apenas um dos programas do conjunto, fazendo com que a pontuação geral do sistema seja um indicador melhor do seu desempenho com a maioria dos programas.

A média geométrica de n valores é calculada multiplicando-se os n valores e tirando-se a raiz enésima do produto. A média aritmética, ou média de um conjunto de valores, é calculada somando-se todos os valores e dividindo-se o resultado pelo número de valores.

Exemplo Quais são as médias aritmética e geométrica dos valores 4, 2, 4, 82?

Solução

A média aritmética desta série é

$$\frac{4 + 2 + 4 + 82}{4} = 23$$

A média geométrica é

$$\sqrt[4]{4 \times 2 \times 4 \times 82} = 7,16$$

Note que a inclusão de um valor extremo na série teve um efeito muito maior sobre a média aritmética do que sobre a média geométrica.

1.7 ACELERAÇÃO

Frequentemente, os projetistas de computadores utilizam o termo *aceleração* para descrever como o desempenho de uma arquitetura muda à medida que diferentes melhoramentos são feitos naquela arquitetura. A aceleração é simplesmente a razão entre os tempos de execução antes e depois que a mudança é feita, de modo que:

$$\text{Aceleração} = \frac{\text{Tempo de execução}_{\text{antes}}}{\text{Tempo de execução}_{\text{depois}}}$$

Por exemplo, se um programa demora 25 segundos para ser executado em uma versão de uma arquitetura e 15 segundos para ser executado em uma nova versão, a aceleração geral é de $25\text{ segundos}/15\text{ segundos} = 1,67$.

1.8 A LEI DE AMDAHL

A regra mais importante para projetar sistemas de computadores de alto desempenho é *faça com que o mais comum seja rápido*. Qualitativamente, isto significa que o impacto de um dado aperfeiçoamento sobre o desempenho geral depende tanto de quanto o aperfeiçoamento melhora o desempenho quando ele é utilizado, como de com que frequência esse aperfeiçoamento é utilizado. Quantitativamente, esta regra foi expressa pela *Lei de Amdahl*, que define

$$\text{Tempo de execução}_{\text{novo}} = \text{Tempo de execução}_{\text{antigo}} \times \left[\text{Parcela}_{\text{não-usada}} + \frac{\text{Parcela}_{\text{usada}}}{\text{Aceleração}_{\text{usada}}} \right]$$

Na equação, $\text{Parcela}_{\text{não-usada}}$ é a parcela de tempo (não instruções) na qual o aperfeiçoamento não está em uso, $\text{Parcela}_{\text{usada}}$ é a parcela de tempo na qual o aperfeiçoamento está em uso e $\text{Aceleração}_{\text{usada}}$ é a aceleração que acontece quando o aperfeiçoamento é usado (isto seria a aceleração geral se o aperfeiçoamento fosse utilizado o tempo todo). Note que $\text{Parcela}_{\text{não-usada}}$ e $\text{Parcela}_{\text{usada}}$ são calculados utilizando o tempo de execução *antes* que as modificações sejam aplicadas. Calcular estes valores utilizando o tempo de execução depois que a modificação fosse aplicada daria resultados incorretos.

A Lei de Amdahl pode ser reescrita utilizando a definição de aceleração para dar

$$\text{Aceleração} = \frac{\text{Tempo de execução}_{\text{antigo}}}{\text{Tempo de execução}_{\text{novo}}} = \frac{1}{\text{Parcela}_{\text{não-usada}} + \frac{\text{Parcela}_{\text{usada}}}{\text{Aceleração}_{\text{usada}}}}$$

Exemplo Suponha que uma dada arquitetura não tenha suporte de *hardware* para multiplicações, de modo que as multiplicações tenham que ser feitas por meio de adições repetidas (este é o caso de alguns dos primeiros microprocessadores). Se demora 200 ciclos para executar uma multiplicação por *software* e quatro ciclos para executar a multiplicação por *hardware*, qual é a aceleração geral produzida pelo *hardware* para suporte de multiplicações, se um programa gasta 10% do seu tempo fazendo multiplicações? E com um programa que gasta 40% do seu tempo fazendo multiplicações?

Solução

Em ambos os casos, quando é utilizado *hardware* para multiplicações a aceleração é $200/4 = 50$ (razão entre o tempo para fazer uma multiplicação sem e com o *hardware*). No caso em que o programa gasta 10% do seu tempo fazendo multiplicações, $\text{Parcela}_{\text{não-usada}} = 0,9$ e $\text{Parcela}_{\text{usada}} = 0,1$. Colocando estes valores na Lei de Amdahl, temos: $\text{Aceleração} = 1/[0,9 + (0,1/50)] = 1,11$. Se o programa gasta 40% do seu tempo fazendo multiplicações, antes que o *hardware* para multiplicações seja acrescentado, então $\text{Parcela}_{\text{não-usada}} = 0,6$ e $\text{Parcela}_{\text{usada}} = 0,4$ e obtemos $\text{Aceleração} = 1/[0,6 + (0,4/50)] = 1,64$.

Este exemplo ilustra o impacto sobre o desempenho geral que tem a parcela de tempo na qual um aperfeiçoamento é utilizado. À medida que a $\text{Aceleração}_{\text{usada}}$ vai para o infinito, a aceleração geral converge para $1/\text{Parcela}_{\text{não-usada}}$, porque o aperfeiçoamento nada pode fazer a respeito do tempo de execução da parcela do programa que não utiliza o aperfeiçoamento.

1.9 RESUMO

Este capítulo teve por objetivo fornecer um contexto para o resto do livro, ao explicar algumas das forças tecnológicas que impulsionam o desempenho de computadores, e fornecer um arcabouço para a discussão e avaliação do desempenho de sistemas, que será utilizado por todo o livro.

Os conceitos importantes que o leitor deve entender, após ter estudado este capítulo, são:

1. A tecnologia de computadores é impulsionada por aperfeiçoamentos na tecnologia de fabricação de semicondutores, e estes aperfeiçoamentos progredem geométrica e não linearmente.
2. Há muitos modos de medir o desempenho de computadores, e as medidas mais efetivas do desempenho geral são baseadas no desempenho de um sistema com uma ampla variedade de aplicações.
3. É importante compreender como uma dada unidade de desempenho é gerada, de modo a entender qual é a sua utilidade para prever o desempenho de um sistema, em uma determinada aplicação.
4. O impacto que uma mudança em uma arquitetura tem sobre o desempenho geral depende não apenas de quanto esta mudança melhora o desempenho quando ela é utilizada, mas com qual freqüência esta modificação é útil. A consequência disto é que o impacto de um aperfeiçoamento sobre o desempenho geral é limitado pela parcela de tempo que o aperfeiçoamento não está em uso, independentemente de quanta aceleração este aperfeiçoamento traz quando ele é utilizado.

Problemas Resolvidos

Tendências Tecnológicas (I)

- 1.1** Para ilustrar a rapidez com que a tecnologia de computadores está sendo aperfeiçoada, vamos considerar o que teria acontecido se tivesse acontecido o mesmo com os automóveis. Assuma que o carro médio, em 1977, tinha uma velocidade máxima de 160 quilômetros por hora e que o consumo médio de combustível era de 6,4 km por litro (km/l). Se tanto a velocidade máxima como a eficiência fossem aperfeiçoadas a uma taxa de 35% ao ano, de 1977 a 1987, e a 50% ao ano, de 1987 a 2000, acompanhando o desempenho dos computadores, qual seria a velocidade máxima e o consumo de combustível de um carro em 1987? E em 2000?

Solução

Em 1987:

O período que vai de 1977 a 1987 é de 10 anos; assim, ambas as características teriam sido aperfeiçoadas por um fator de $(1.35)^{10} = 20.1$, dando uma velocidade máxima de 3216 km/h e um consumo de combustível de 128,6 km/l.

Em 2000:

Um período de mais 13 anos, desta vez a uma taxa de aperfeiçoamento de 50% ao ano, para um fator total de $(1.5)^{13} = 194.6$ sobre os valores de 1987. Isto dá uma velocidade máxima de 625.833,6 km/h e um consumo de combustível de 25.025,6 km/l. Isto é rápido o suficiente para cobrir a distância entre a Terra e a Lua em menos de 40 minutos e fazer a viagem de ida e volta com menos de 40 litros de gasolina.

Tendências Tecnológicas (II)

- 1.2** Desde 1987, o desempenho dos computadores tem aumentado a uma taxa de aproximadamente 50% ao ano, com as melhorias na tecnologia de fabricação respondendo por cerca de 35% ao ano e os aperfeiçoamentos na arquitetura por, aproximadamente, 15% ao ano.
1. Se o desempenho do melhor computador disponível em 1/1/1988 fosse definido como sendo 1, qual seria o desempenho esperado do melhor computador disponível em 1/1/2001?
 2. Suponha que não tivesse havido aperfeiçoamentos na arquitetura dos computadores desde 1987, fazendo com que a tecnologia de fabricação fosse a única fonte de melhorias no desempenho. Qual seria o desempenho esperado do melhor computador disponível em 1/1/2001?
 3. Agora, suponha que não tivesse havido aperfeiçoamentos na tecnologia de fabricação, tornando as melhorias na arquitetura a única fonte de melhorias no desempenho. Qual seria o desempenho esperado do computador mais rápido em 1/1/2001?

Solução

1. O desempenho melhora a 50% ao ano e de 1/1/1988 até 1/1/2000 são 13 anos, de modo que o desempenho esperado da máquina de 1/1/2001 é $1 \times (1.5)^{13} = 194.6$.
2. Aqui, o desempenho melhora apenas 35% ao ano, de modo que o desempenho esperado é 49,5.
3. A melhora no desempenho é de 15% ao ano, dando um desempenho esperado de 6,2.

Aceleração (I)

- 1.3 Se a versão de 1998 de um computador executa um programa em 200 s e a versão fabricada no ano 2000 executa o mesmo programa em 150 s, qual é a aceleração que o fabricante obteve ao longo de um período de dois anos?

Solução

$$\text{Aceleração} = \frac{\text{Tempo de execução}_{\text{antes}}}{\text{Tempo de execução}_{\text{depois}}}$$

Assim, a aceleração é de $200\text{ s}/150\text{ s} = 1.33$. Claramente, este fabricante está bem abaixo da taxa de crescimento de desempenho da indústria em geral.

Aceleração (II)

- 1.4 Para atingir uma aceleração de 3 em um programa que originalmente demorava 78 s para ser executado, para quanto deve ser reduzido o tempo de execução do programa?

Solução

Aqui, temos os valores para a Aceleração e o Tempo de execução_{antes}. Substituindo estes valores na fórmula para a aceleração e resolvendo para o Tempo de execução_{depois}, temos que o tempo de execução precisa ser reduzido para 26 s, de modo a atingir uma aceleração de 3.

Medindo o Desempenho (I)

1. Quais são os programas e os conjuntos de *benchmark* utilizados para medir o desempenho de computadores?
2. Por que existem vários *benchmarks* que são utilizados pelas arquiteturas de computadores, ao invés de um *benchmark* "melhor"?

Solução

1. Sistemas de computadores são utilizados para executar uma ampla variedade de programas, alguns dos quais podem não existir por ocasião da compra ou construção do sistema. Assim, geralmente não é possível medir o desempenho de um sistema sobre um conjunto de programas que será executado na máquina. Ao invés disto, programas e conjuntos de *benchmark* são utilizados para medir o desempenho de um sistema com uma ou mais aplicações que acredita-se serem representativas do conjunto de programas que será executado na máquina.

2. Existem vários programas/conjuntos de *benchmark* porque os computadores são utilizados para uma ampla variedade de aplicações, cujo desempenho pode depender de muitos aspectos diferentes do sistema de computador. Por exemplo, o desempenho de aplicações de banco de dados e processamento de transações tende a depender fortemente do desempenho do subsistema de E/S do computador. Em contraste, aplicações para cálculos científicos dependem principalmente do desempenho do processador e do sistema de memória do sistema. Da mesma forma que as aplicações, os conjuntos de *benchmark* variam em termos da quantidade de esforço que elas colocam sobre cada subsistema do computador e é importante utilizar um *benchmark* que exija esforços dos mesmos subsistemas que as aplicações pretendidas. Utilizar um *benchmark* exigente quanto ao processador a fim de avaliar computadores com relação ao processamento de transações não daria uma boa estimativa da taxa pela qual esses sistemas poderiam processar transações, pois o fator limitante é o subsistema de E/S.

Medindo o Desempenho (II)

- 1.6 Quando está executando um programa em particular, o computador A atinge 100 MIPS e o computador B atinge 75 MIPS. No entanto, o computador A demora 60 segundos para executar o programa, enquanto que o computador B demora apenas 45 segundos. Como isto é possível?

Solução

MIPS medem a taxa pela qual um processador executa instruções, mas arquiteturas diferentes de processadores exigem números diferentes de instruções para executar um determinado cálculo. Se, para completar o programa, o computador A tem que executar muito mais instruções do que o computador B, seria possível que o computador A demorasse mais para executar o programa do que o processador B, independentemente do fato de que o computador A executa mais instruções por segundo.

Medindo o Desempenho (III)

- 1.7 Em um conjunto de *benchmark*, o computador C obtém uma pontuação de 42 e o computador D, 35 (pontuações maiores são melhores). Ao executar o seu programa, você descobre que o computador C demora 20% a mais que o computador D. Como isso é possível?

Solução

A explicação mais provável é que o seu programa seja altamente dependente de algum aspecto do sistema que não está sendo exigido pelo conjunto de *benchmark*. Por exemplo, ele pode executar um grande número de cálculos de ponto flutuante e o conjunto de *benchmark* enfatiza o desempenho com inteiros, ou vice-versa.

CPI

- 1.8 Quando executado em um dado sistema, um programa demora 1.000.000 de ciclos. Se o sistema atinge um CPI de 40, quantas instruções foram executadas no programa?

Solução

$CPI = (\text{número de ciclos}) / (\text{número de instruções})$. Portanto, $(\text{número de instruções}) = (\text{número de ciclos}) / CPI$. $1.000.000 \text{ ciclos} / 40 \text{ CPI} = 25.000$. Assim, ao executar o programa, 25.000 instruções foram executadas.

IPC

- 1.9 Qual é o IPC de um programa que executa 35.000 instruções e exige 17.000 ciclos para ser completado?

Solução

$IPC = (\text{número de instruções}) / (\text{número de ciclos})$, de modo que o IPC deste programa é de $35.000 \text{ instruções} / 17.000 \text{ ciclos} = 2.06$.

Média Geométrica versus Média Aritmética

- 1.10 Dado o seguinte conjunto de pontuações de *benchmarks* individuais para cada um dos programas na parte de inteiros do *benchmark* SPEC2000, calcule as médias aritmética e geométrica de cada conjunto. Observe que estas pontuações não representam um conjunto real de medições feitas sobre uma máquina. Elas foram selecionadas para ilustrar o impacto que a utilização de métodos diferentes de cálculo da média tem sobre pontuações de *benchmark*.

Benchmark	Pontuação antes do aperfeiçoamento	Pontuação depois do aperfeiçoamento
1.64.gzip	10	12
175.vpr	14	16
176.gcc	23	28
181.mcf	36	40
186.crafty	9	12
197.parser	12	120
252.eon	25	28
253.perlbmk	18	21
254.gap	30	28
255.vortex	17	21
256.bzip2	7	10
300.twolf	38	42

Solução

Existem 12 *benchmarks* no conjunto, de modo que a média aritmética é calculada somando-se todos os valores em cada conjunto e dividindo o resultado por 12, enquanto que a média geométrica é calculada tirando-se a raiz 12^o do produto de todos os valores em um conjunto. Isto dá os seguintes os valores:

Antes do aperfeiçoamento: média aritmética = 19,92; média geométrica = 17,39

Depois do aperfeiçoamento: média aritmética = 31,5; média geométrica = 24,42

O que vemos, a partir disto, é que a média aritmética é muito mais sensível a mudanças grandes em um dos valores do conjunto que a média geométrica. A maioria dos *benchmarks* individuais vêem mudanças relativamente pequenas à medida que acrescentamos aperfeiçoamentos à arquitetura, mas o *benchmark* 197.parser mostra um aperfeiçoamento com um fator de 10. Isto faz com que a média aritmética aumente em praticamente 60%, enquanto que a média geométrica aumenta apenas 40%. Esta sensibilidade reduzida a valores individuais é o motivo pelo qual os especialistas preferem a média geométrica para fazer a média dos resultados de vários *benchmarks*, uma vez que um resultado muito bom ou um resultado muito ruim, em um conjunto de *benchmarks*, tem menos impacto sobre a pontuação geral.

Lei de Amdahl (I)

- 1.11 Suponha que, ao executar um dado programa, um computador gaste 90% do seu tempo tratando um tipo especial de cálculo, e que os seus fabricantes façam uma mudança que melhore o seu desempenho, naquele tipo de cálculo, por um fator de 10.
1. Se o programa demorava, originalmente, 100 s para ser executado, qual será o seu tempo de execução depois da modificação?
 2. Qual é a aceleração do sistema novo com relação ao antigo?
 3. Qual parte do seu tempo de execução o novo sistema gasta executando o tipo de cálculo que foi aperfeiçoado?

Solução

1. Isto é uma aplicação direta da Lei de Amdahl:

$$\text{Tempo de execução}_{\text{novo}} = \text{Tempo de execução}_{\text{antigo}} \times \left[\text{Parcela}_{\text{não_usada}} + \frac{\text{Parcela}_{\text{usada}}}{\text{Aceleração}_{\text{usada}}} \right]$$

$\text{Tempo de execução}_{\text{antigo}} = 100 \text{ s}$, $\text{Parcela}_{\text{usada}} = 0,9$, $\text{Parcela}_{\text{não_usada}} = 0,1$ $\text{Aceleração}_{\text{usada}} = 10$
Isto dá um $\text{Tempo de execução}_{\text{novo}}$ de 19 s.

2. Utilizando a definição de aceleração, temos uma aceleração de 5,3. Alternativamente, poderíamos substituir os valores da parte 1 na versão da aceleração da Lei de Amdahl e obter o mesmo resultado.
3. A Lei de Amdahl não nos dá um meio direto para responder a esta pergunta. O sistema original gastava 90% do seu tempo executando o tipo de cálculo que foi aperfeiçoado, de modo que ele gastava 90 segundos, de um programa de 100 segundos, executando aquele tipo de cálculo. Uma vez que o cálculo foi aperfeiçoado por um fator de 10, o sistema aperfeiçoado gasta $90/10 = 9$ segundos executando aquele tipo de cálculo. Como 9 segundos é 47% de 19 segundos, o novo tempo de execução, o novo sistema gasta 47% do seu tempo executando o tipo de cálculo que foi aperfeiçoado.

Poderíamos também ter calculado o tempo que o sistema original gastava executando os cálculos que não tinham sido aperfeiçoados (10 segundos). Uma vez que estes cálculos não foram modificados quando o aperfeiçoamento foi feito, o tempo total gasto neles, pelo novo sistema, é o mesmo que no sistema antigo. Isto poderia ser então utilizado para calcular o percentual de tempo gasto nos cálculos que não foram aperfeiçoados e, subtraindo aquele valor de 100, calcular o percentual de tempo gasto nos cálculos que foram aperfeiçoados.

Lei de Amdahl (II)

- 1.12 Um computador gasta 30% do seu tempo acessando a memória, 20% executando multiplicações e 50% executando outras instruções. Como projetista de computadores, você tem que escolher entre aperfeiçoar a memória, o *hardware* de multiplicação ou a execução das instruções que não são de multiplicação. Só existe espaço no *chip* para um aperfeiçoamento, e cada um dos aperfeiçoamentos irá melhorar o seu componente de computação associado por um fator de 2.
1. Sem fazer cálculos, qual aperfeiçoamento você esperaria que desse o maior aumento de desempenho. Por quê?
 2. Qual seria a aceleração ao se fazer cada uma dessas três mudanças?

Solução

1. Aperfeiçoar a execução das instruções que não são de multiplicação deve produzir o maior benefício. Cada benefício aumenta o desempenho da sua área afetada pelo mesmo fator e o sistema gasta mais tempo executando instruções que não são de multiplicação do que com qualquer uma das outras categorias. Uma vez que a Lei de Amdahl diz que

o impacto geral de um aperfeiçoamento cresce à medida que a parcela de tempo que aquele aperfeiçoamento é utilizado cresce, aperfeiçoar as instruções que não são de multiplicação daria o melhor resultado.

2. Substituindo na fórmula de aceleração da Lei de Amdahl os valores do percentual de tempo utilizado e do aperfeiçoamento quando utilizado, mostra que aperfeiçoar o sistema de memória dá uma aceleração de 1,18, aperfeiçoar a multiplicação dá uma aceleração de 1,11 e aperfeiçoar as instruções que não são de multiplicação dá uma aceleração de 1,33, confirmando a intuição da parte 1.

Comparando Diferentes Modificações em uma Arquitetura

- 1.13 Qual aperfeiçoamento propicia a maior redução no tempo de execução: um que é utilizado 20% do tempo, mas melhora o desempenho por um fator de 2 quando utilizado, ou um que é utilizado 70% do tempo, mas melhora o desempenho apenas por um fator de 1,3 quando é utilizado?

Solução

Aplicando a Lei de Amdahl, obtemos a seguinte equação para o primeiro aperfeiçoamento:

$$\text{Tempo de execução}_{\text{novo}} = \text{Tempo de execução}_{\text{antigo}} \times \left[0,8 + \frac{0,2}{2} \right]$$

Assim, o tempo de execução com o primeiro aperfeiçoamento é 90% do tempo de execução sem o aperfeiçoamento. Inserindo os valores para o segundo aperfeiçoamento na Lei de Amdahl, tem-se que o tempo de execução com o segundo aperfeiçoamento é 84% do tempo de execução sem o aperfeiçoamento. Assim, o segundo aperfeiçoamento terá um impacto maior sobre o tempo geral de execução, independentemente do fato de que ele dá uma melhoria menor quando em uso.

Convertendo Aperfeiçoamentos Individuais para Impacto Geral sobre o Desempenho

- 1.14 Um projetista de computadores está desenvolvendo o sistema de memória para a próxima versão de um processador. Se a versão atual do processador gasta 40% do seu tempo processando referências à memória, de quanto o projetista precisa acelerar o sistema de memória para atingir uma aceleração global de 1,2? E uma aceleração de 1,6?

Solução

Para resolver isto, aplicamos a Lei de Amdahl para acelerações, tendo a Aceleração_{nova} como desconhecida, ao invés de Aceleração geral. Parcelfa_{nova} é 0,4, uma vez que o sistema original gasta 40% do seu tempo tratando referências à memória, de modo que Parcelfa_{nova} é 0,6. Para um aumento de 20% no desempenho geral, isto dá:

$$\text{Aceleração} = 1,2 = \frac{1}{0,6 + \frac{0,4}{\text{Aceleração}_{\text{nova}}}}$$

Resolvendo para Aceleração_{nova}, obtemos

$$\text{Aceleração}_{\text{nova}} = \frac{0,4}{\frac{1}{1,2} - 0,6} = 1,71$$

Para encontrar o valor da Aceleração_{nova} necessária para dar uma aceleração de 1,6, o único valor que muda na equação acima é a Aceleração. Resolvendo novamente, obtemos Aceleração_{nova} = 16. Aqui, novamente, vemos os retornos decrescentes que decorrem de aperfeiçoar repetidamente apenas um aspecto do desempenho do sistema. Para aumentar a aceleração geral de 1,2 para 1,6, temos que aumentar a Aceleração_{nova} por um fator de praticamente 10, porque os 60% do tempo que o sistema de memória não está em uso começam a dominar o desempenho geral, à medida que melhoramos o desempenho do sistema de memória.

Aperfeiçoando Instruções

- 1.15** Considere uma arquitetura que tem quatro tipos de instruções: somas, multiplicações, operações de memória e desvios. A tabela abaixo dá o número de instruções que pertencem a cada tipo, no programa com o qual estamos preocupados, o número de ciclos que demora para executar cada tipo de instrução e a aceleração na execução do tipo de instrução, a partir de um aperfeiçoamento proposto (cada aperfeiçoamento afeta apenas um tipo de instrução). Avalie os aperfeiçoamentos para cada um dos tipos de instrução, em termos do seu impacto sobre o desempenho geral.

Tipo de instrução	Número	Tempo de execução	Aceleração para o tipo
Soma	10 milhões	2 ciclos	2,0
Multiplicação	30 milhões	20 ciclos	1,3
Memória	35 milhões	10 ciclos	3,0
Desvio	15 milhões	4 ciclos	4,0

Solução

Para resolver este problema, primeiro precisamos calcular o número de ciclos gastos ao executar cada tipo de instrução, antes que os aperfeiçoamentos sejam aplicados, e a parcela do total de ciclos gastos executando cada tipo de instrução (P_{instr} , para cada um dos aperfeiçoamentos). Isto permitirá que utilizemos a Lei de Amdahl para calcular a aceleração geral para cada aperfeiçoamento proposto. Multiplicar o número de instruções em cada tipo pelo tempo de execução por instrução resulta no número de ciclos gastos para executar cada tipo de instrução, e somar estes valores fornece o número total de ciclos para executar o programa. Os valores estão apresentados na tabela abaixo (o tempo total de execução é de 1.030 milhões de ciclos):

Tipo de instrução	Número	Tempo de execução	Aceleração para o tipo	Número de ciclos	Parcela dos ciclos
Soma	10 milhões	2 ciclos	2,0	20 milhões	2%
Multiplicação	30 milhões	20 ciclos	1,3	600 milhões	58%
Memória	35 milhões	10 ciclos	3,0	350 milhões	34%
Desvio	15 milhões	4 ciclos	4,0	60 milhões	6%

Então, podemos colocar estes valores na Lei de Amdahl, usando a parcela do ciclos como P_{instr} , para obter a aceleração geral a partir de cada aperfeiçoamento.

Assim, melhorar as operações de memória dá a melhor aceleração geral, seguido por melhorar as multiplicações, os desvios e as somas:

Tipo de instrução	Número	Tempo de execução	Aceleração para o tipo	Número de ciclos	Parcela dos ciclos	Aceleração geral
Soma	10 milhões	2 ciclos	2,0	20 milhões	2%	1,01
Multiplicação	30 milhões	20 ciclos	1,3	600 milhões	58%	1,15
Memória	35 milhões	10 ciclos	3,0	350 milhões	34%	1,29
Desvio	15 milhões	4 ciclos	4,0	60 milhões	6%	1,05

Capítulo 2

Representação de Dados e Aritmética de Computadores

2.1 OBJETIVOS

Este capítulo cobre os métodos mais comuns que os sistemas de computadores utilizam para representar dados e como as operações aritméticas são executadas sobre estas representações. Inicia com uma discussão sobre como os *bits* (dígitos binários) são representados por sinais elétricos e continua com uma discussão sobre como números inteiros e de ponto flutuante são representados como seqüências de *bits*.

Após ler este capítulo, você deverá:

1. Compreender como os computadores representam os dados internamente, tanto ao nível do padrão de *bits*, quanto ao nível do sinal elétrico.
2. Ser capaz de traduzir números inteiros e em ponto flutuante de e para suas representações binárias.
3. Ser capaz de executar operações matemáticas básicas (adição, subtração e multiplicação) com números inteiros e em ponto flutuante.

2.2 DE ELÉTRONS A BITS

Os computadores modernos são sistemas *digitais*, o que significa que eles interpretam os sinais elétricos como possuindo um conjunto de valores discretos, ao invés de quantidades analógicas. Ao mesmo tempo que isto aumenta o número de sinais necessários para transportar uma determinada quantidade de informação, facilita o armazenamento de informações e faz com que os sistemas digitais sejam menos sujeitos a ruídos elétricos do que os analógicos.

A *convenção de sinais* de um sistema digital determina como os sinais elétricos analógicos são interpretados como valores digitais. A Fig. 2-1 ilustra as convenções de sinalização mais comuns em computadores modernos. Cada sinal carrega um de dois valores, dependendo do nível de tensão do sinal. Tensões baixas são interpretadas como 0 e tensões altas são interpretadas como 1.

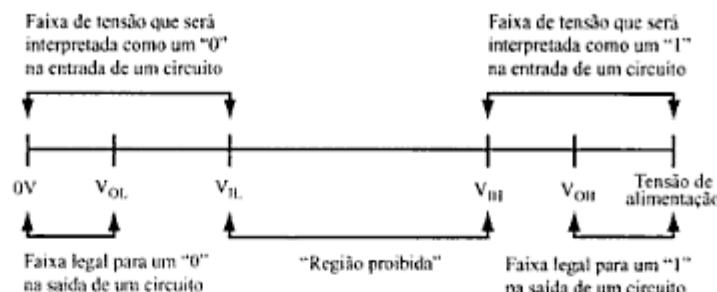


Fig. 2-1 Mapeando tensões para bits.

A convenção de sinalização digital divide a faixa das possíveis tensões em diversas regiões. A região de 0V a V_{IL} é a faixa de tensões que representa o valor lógico zero em um circuito, enquanto a região de V_{IH} até a tensão de alimentação será interpretada como o valor lógico um (1) em um circuito. A região entre V_{IL} e V_{IH} é conhecida como “região proibida” porque não é possível prever se um circuito interpretará a tensão nesta faixa como 0 ou como 1.

A V_{OL} é a tensão mais alta que um circuito pode produzir para gerar um zero lógico e V_{OH} é a tensão mais baixa que um circuito pode produzir para gerar o valor lógico 1. É importante que V_{OH} e V_{OL} estejam mais próximos aos extremos da faixa de tensão do que V_{IL} e V_{IH} , porque os intervalos entre V_{OL} e V_{IL} , e entre V_{IH} e V_{OH} , determinam as *margens de ruído* do sistema digital. A margem de ruído de um sistema digital é a quantidade pela qual o sinal de saída de um circuito pode mudar, antes que seja possível que ele seja interpretado por um outro circuito como o valor oposto. Quanto mais larga a margem de ruído, melhor o sistema será capaz de tolerar os efeitos do acoplamento entre sinais elétricos, perdas resistivas em fios e outros efeitos que podem fazer com que os sinais mudem entre o ponto no qual eles foram gerados e o ponto no qual eles são utilizados.

Sistemas que mapeiam cada sinal elétrico sobre dois valores são conhecidos como *sistemas binários* e a informação que cada sinal carrega é chamada de um *bit* (forma abreviada para *BInary digiT* – dígito binário). Sistemas com mais valores por sinal são possíveis, mas a complexidade adicional de projetar circuitos para interpretar essas convenções de sinais e a redução nas margens de ruído que ocorre quando a faixa de tensão é dividida em mais do que dois valores tornam esses sistemas difíceis de serem construídos. Por este motivo, praticamente todos os sistemas digitais são binários.

2.3 REPRESENTAÇÃO BINÁRIA DE INTEIROS POSITIVOS

Inteiros positivos são representados utilizando o sistema binário de numeração posicional (base 2), semelhante ao sistema de numeração posicional utilizado na aritmética decimal (base 10). Na aritmética base 10, os números são representados como a soma dos múltiplos de cada potência de 10, de modo que o número $1543 = (1 \times 10^3) + (5 \times 10^2) + (4 \times 10^1) + (3 \times 10^0)$. Para números binários, a base é 2; assim, cada posição do número representa uma potência crescente de 2, em vez de uma potência crescente de 10. Por exemplo o número binário $100111 = (1 \times 2^5) + (0 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$ equivale ao número 39 em base decimal. Os valores binários são usualmente precedidos pelo prefixo “0b” para identificá-los como binários, em vez de como números decimais. Assim como um número decimal com n dígitos pode representar valores de 0 a $10^n - 1$, um número binário com n bits, sem sinal, pode representar valores de 0 a $2^n - 1$.

A desvantagem dos números binários, quando comparados aos números decimais, é que eles exigem significativamente mais dígitos para representar um certo inteiro, o que faz com que seja incômodo e maçante trabalhar com eles. Para resolver isto, a notação *hexadecimal*, na qual cada dígito tem 16 valores possíveis, é frequentemente utilizada para representar números binários. Na notação hexadecimal, os números de 0 a 9 tem o mesmo valor que na notação decimal, e as letras A até F (ou a até f – maiúsculas ou minúsculas são irrelevantes na notação hexadecimal) são utilizadas para representar os números de 10 até 15, como indicado na Fig. 2-2.

Número decimal	Representação binária	Representação hexadecimal
0	0b0000	0x0
1	0b0001	0x1
2	0b0010	0x2
3	0b0011	0x3
4	0b0100	0x4
5	0b0101	0x5
6	0b0110	0x6
7	0b0111	0x7
8	0b1000	0x8
9	0b1001	0x9
10	0b1010	0xA
11	0b1011	0xB
12	0b1100	0xC
13	0b1101	0xD
14	0b1110	0xE
15	0b1111	0xF

Fig. 2-2 Notação hexadecimal.

Normalmente, para diferenciar números hexadecimais de números decimais ou números binários, emprega-se a notação “0x” colocado à esquerda do número. Para representar valores maiores do que 15 na notação hexadecimal, é utilizada a notação de numeração posicional com base 16.

Exemplo Quais são as representações binária e hexadecimal do número decimal 47?

Solução

Para converter números decimais em binários, devemos expressá-los como uma soma de valores que são potências de 2:

$$47 = 32 + 8 + 4 + 2 + 1 = 2^5 + 2^3 + 2^2 + 2^1 + 2^0$$

Portanto, a representação binária de 47 é 0b101111.

Para converter números decimais em hexadecimais, podemos expressar o número como uma soma de potências de 16 ou agrupar os *bits*, na representação binária, em conjuntos de 4 *bits* e procurar cada conjunto na Fig. 2-2. Convertendo diretamente, $47 = 2 \times 16 + 15 = 0x2F$. Considerando a representação binária $47 = 0b101111$, ou ainda 0b001111, agrupa-se os *bits* quatro a quatro 0b0010 = 0x2, 0b1111 = 0xF, de modo que $47 = 0x2F$.

2.4 OPERAÇÕES ARITMÉTICAS COM INTEIROS POSITIVOS

A aritmética em base 2 (binária) pode ser feita utilizando-se as mesmas técnicas empregadas na aritmética em base 10 (decimal), exceto pelo conjunto restrito de valores que podem ser representados por cada dígito. Freqüentemente, este é o modo mais fácil para os seres humanos resolverem problemas de matemática envolvendo números binários. Porém, em alguns casos, essas técnicas não são facilmente implementadas em circuitos, fazendo com que os projetistas de computadores escolham outras técnicas. Como veremos nas próximas seções, a adição e a multiplicação são implementadas utilizando circuitos que são análogos às técnicas utilizadas por seres humanos quando fazem operações aritméticas. A divisão é implementada utilizando métodos específicos de computadores e a subtração é implementada de diferentes formas, dependendo da representação utilizada para inteiros negativos.

Exemplo Calcule a soma de 9 e 5 utilizando números binários em um formato binário de 4 *bits*.

Solução

As representações binárias em 4 bits dos números 9 e 5 são 0b1001 e 0b0101, respectivamente. Ao somar os bits menos significativos, obtemos $0b1 + 0b1 = 0b10$, que é um 0 no bit menos significativo do resultado e um transporte de 1 (vai 1) para a próxima posição de bit. Calculando o próximo bit da soma, obtemos $0b1$ (transporte) + $0b0 + 0b0 = 0b1$. Repetindo isto para todos os bits, obtemos o resultado final de 0b1110. A Fig. 2-3 ilustra este processo.

$$\begin{array}{r}
 & & & & \text{Transporte do bit} \\
 & & & 1 & \leftarrow \text{resultante da} \\
 & & & & \text{adição (vai 1)} \\
 0b & 1 & 0 & 0 & 1 \\
 + & 0b & 0 & 1 & 0 \\
 \hline
 0b & 1 & 1 & 1 & 0
 \end{array}$$

Fig. 2-3 Exemplo de soma binária.

Adição/Subtração

O hardware que os computadores utilizam para implementar a adição é muito semelhante ao método delineado acima. Os módulos, conhecidos como *somadores completos*, calculam cada bit da saída baseados nos bits correspondentes às entradas e ao transporte gerado pela soma dos bits anteriores. A Fig. 2-4 mostra um circuito somador de 8 bits.

Para o tipo de somador descrito acima, a velocidade do circuito é determinada pelo tempo que demora para propagar os sinais de transporte (vai 1) por todos os somadores completos. Basicamente, cada somador completo não pode executar a sua parte do cálculo até que todos os somadores completos à sua direita tenham completado seu cálculo, de modo que o tempo de cálculo cresce linearmente com o número de bits nas entradas. Os projetistas desenvolveram circuitos que melhoraram o desempenho desse procedimento ao realizar o máximo possível de cálculo em somador completo antes que cada entrada de transporte esteja disponível, de modo a reduzir o atraso depois que o transporte esteja disponível – ou tomando diversos bits de entrada em conta para a geração dos transportes – mas a técnica básica permanece a mesma.

A subtração pode ser tratada por métodos similares, utilizando módulos que calculam um bit da diferença entre dois números. No entanto, o formato mais comum para inteiros negativos, a notação em complemento de 2, permite que a subtração seja executada ao negar a segunda entrada e fazer uma soma, tornando possível utilizar o mesmo hardware, tanto para a adição quanto para a subtração. A notação em complemento de 2 será discutida posteriormente.

Multiplicação

A multiplicação de inteiros sem sinal é tratada de modo semelhante àquele utilizado pelos seres humanos para multiplicar números decimais com vários dígitos. A primeira entrada da multiplicação é multiplicada por cada bit da segunda entrada, separadamente, e os resultados são somados. Na multiplicação binária, isto é simplificado pelo fato de que o resultado da multiplicação de um número por um bit é, ou o número original, ou 0, fazendo com que o hardware seja menos complexo.

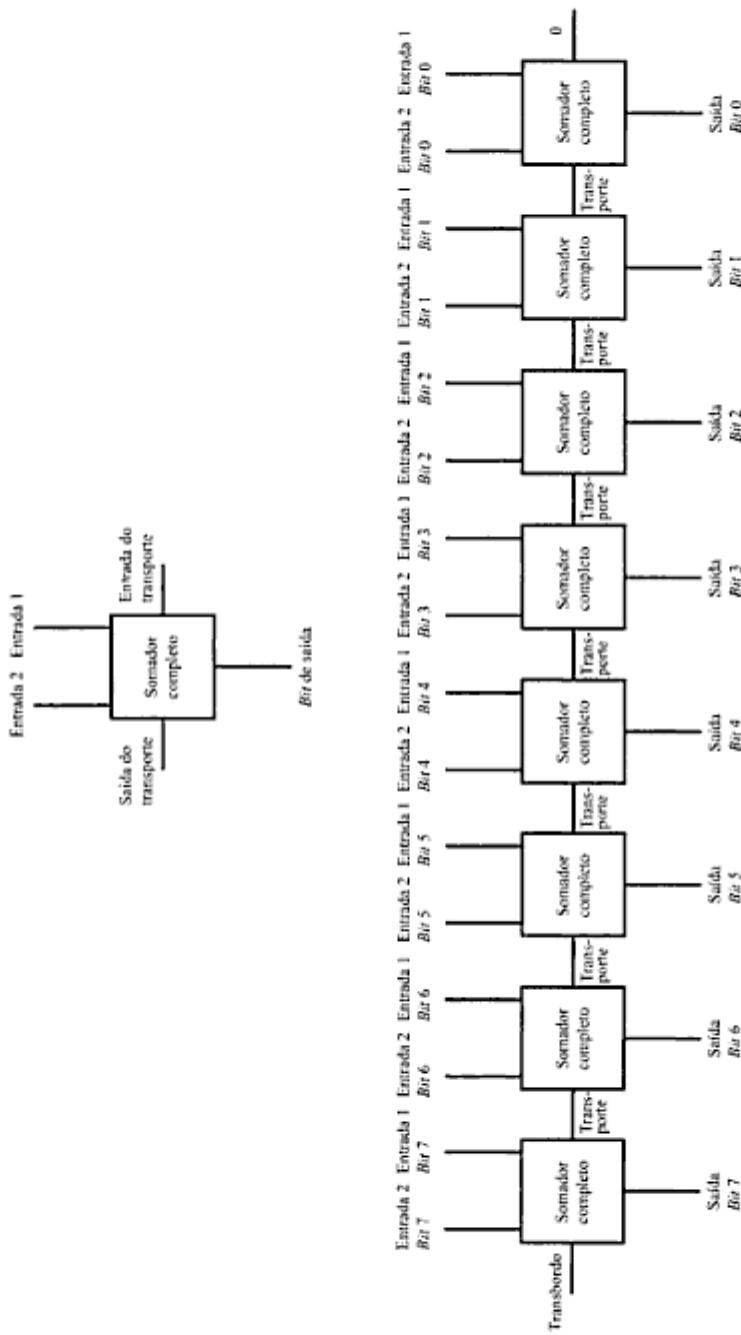


Fig. 2-4 Somador de 8 bits.

$$\begin{array}{r}
 0b1011 \\
 \times 0b0101 \\
 \hline
 1011 \\
 0000 \\
 1011 \\
 + 0000 \\
 \hline
 0b110111
 \end{array}$$

Fig. 2-5 Exemplo de multiplicação.

A Fig. 2-5 mostra um exemplo de multiplicação de 11 (0b1011) por 5 (0b0101). Primeiro, 0b1011 é multiplicado por cada bit de 0b0101, para obter os produtos parciais mostrados na figura. Então, os produtos parciais são somados para obter o resultado final. Note que cada produto parcial sucessivo é deslocado uma posição para a esquerda, para levar em consideração a posição diferente dos valores dos bits na segunda entrada.

Um problema com a multiplicação de inteiros é que o produto de 2 números de n bits pode exigir até $2n$ bits para ser representado. Por exemplo, o produto dos 2 números de 4 bits na Fig. 2-5 exige 6 bits para ser representado. Muitas operações aritméticas podem gerar resultados que não possam ser representados com o mesmo número de bits que as suas entradas. Isto é conhecido como *transbordo (overflow)* ou *transbordo negativo (underflow)* e será discutido a seguir. No caso da multiplicação, o número de bits do transbordo é tão grande que os projetistas de hardware tomam medidas especiais para tratar disto. Em alguns casos, os projetistas fornecem operações diferentes para calcular os n bits mais significativos ou mais significativos do resultado de uma multiplicação de n bits por n bits. Em outros, o sistema descarta os n bits mais significativos, ou os coloca em um registrador de saída especial onde o programador pode acessá-los, se necessário.

Divisão

A divisão pode ser implementada em sistemas de computadores subtraíndo repetidamente o divisor do dividendo e contando o número de vezes que o divisor pode ser subtraído do dividendo, antes que o dividendo torne-se menor do que o divisor. Por exemplo, 15 pode ser dividido por 5, subtraíndo 5 repetidamente de 15, obtendo 10, 5 e 0 como os resultados intermediários. O quociente, 3, é o número de subtrações que tiveram que ser executadas antes que o resultado intermediário se tornasse menor do que o divisor.

Embora seja possível construir um hardware para implementar a divisão por repetidas subtrações, isto seria impraticável por causa do número de subtrações necessárias. Por exemplo, 2^{31} (um número grande em uma representação inteiro sem sinal em 32 bits) dividido por 2 é 2^{30} , o que significa que teriam que ser executadas 2^{30} subtrações para executar esta divisão por meio de subtrações repetidas. Em um sistema operando a 1 GHz, isto demoraria aproximadamente 1 segundo, muito mais tempo do que qualquer outra operação aritmética.

Em vez disso, os projetistas utilizam métodos baseados em pesquisa de tabelas para implementar a divisão. Utilizando tabelas pré-geradas, estas técnicas geram de 2 a 4 bits do quociente em cada ciclo. Isto permite que divisões de inteiros de 32 e de 64 bits sejam feitas em um número razoável de ciclos; apesar disso, a divisão é tipicamente a operação matemática básica mais lenta em um computador.

Transbordo e Transbordo Negativo

A largura de bits de um computador limita o maior e o menor número que pode ser representado como inteiro. Para inteiros sem sinal, um número de n bits pode representar valores de 0 até $2^n - 1$. No entanto, as operações aritméticas, com números que podem ser representados em um dado número de bits, podem gerar resultados que não possam ser representados no mesmo formato. Por exemplo, somar 2 inteiros de n bits pode produzir um resultado de até $2(2^n - 1)$, o que não pode ser representado em n bits e é possível gerar resultados negativos ao subtrair dois inteiros positivos, o que também não pode ser representado por um número de n bits sem sinal.

Quando uma operação gera um resultado que não pode ser expresso no formato dos seus operandos de entrada, diz-se que ocorreu um *transbordo (overflow ou underflow)*. Os transbordos ocorrem quando o resultado de uma operação é grande demais para ser representado no formato das entradas, o que é denominado simplesmente de transbordo (*overflow*), e o transbordo negativo (*underflow*) é quando o resultado é pequeno demais para ser repre-

sentado naquele formato. Os diversos sistemas tratam os transbordos de modos diferentes. Alguns sinalizam um erro quando eles ocorrem, outros substituem o resultado pelo valor mais próximo que pode ser representado naquele formato. Para números em ponto flutuante, o padrão IEEE especifica um conjunto de representações especiais que indicam que um transbordo ocorreu. Essas representações são chamadas de NaNs e serão discutidas mais adiante.

2.5 INTEIROS NEGATIVOS

Para representar inteiros negativos como sequências de *bits*, a notação de numeração posicional utilizada para inteiros precisa ser expandida para indicar se um número é positivo ou negativo. Cobriremos dois esquemas para fazer isto: as representações sinal e magnitude e a notação em complemento de 2.

Representação Sinal e Magnitude

Na representação sinal e magnitude, o *bit* mais significativo (também conhecido como o *bit* de sinal) de um número binário indica se o número é positivo ou negativo. E o resto do número indica o valor absoluto (ou magnitude) do número, utilizando o mesmo formato que a representação binária sem sinal. Números de N bits em sinal e magnitude podem representar quantidades de $-(2^{N-1} - 1)$ até $+(2^{N-1} - 1)$. Note que há duas representações possíveis para 0 na notação com sinal de magnitude: +0 e -0. O +0 tem o valor 0 no campo de magnitude e o *bit* de sinal positivo. O -0 tem um valor igual a 0 no campo de magnitude e o *bit* de sinal negativo.

Exemplo A representação binária sem sinal, em 16 bits, de 152 é 0b0000 0000 1001 1000. Em um sistema de 16 bits em sinal e magnitude, -152 seria representado como 0b1000 0000 1001 1000. Aqui, o *bit* mais à esquerda do número é o *bit* de sinal e o resto do número fornece a magnitude.

As representações em sinal e magnitude têm a vantagem de formar o número negativo de um número de uma forma muito fácil: apenas invertendo o *bit* de sinal. Determinar se um número é positivo ou negativo também é muito fácil, uma vez que só é necessário examinar o *bit* de sinal. A representação em sinal e magnitude faz com que seja fácil executar a multiplicação e a divisão de números com sinal, mas torna difícil executar a soma e a subtração. Para a multiplicação e a divisão, o *hardware* pode simplesmente executar operações sem sinal sobre a parte de magnitude das entradas e examinar os *bits* de sinal das entradas para determinar o *bit* de sinal do resultado.

Exemplo Multiplique os números +7 e -5, utilizando inteiros de 6 bits em sinal e magnitude.

Solução

A representação binária de +7 é 000111 e de -5 é 100101. Para multiplicá-los, multiplicamos as suas porções de magnitude como inteiros sem sinal, gerando 0100011 (35). Então, examinamos os *bits* de sinal dos números sendo multiplicados e estabelecemos que um deles é negativo. Portanto, o resultado da multiplicação tem que ser negativa, fornecendo 1100011 (-35).

A adição e a subtração de números em sinal e magnitude exigem um *hardware* relativamente complexo porque somar (ou subtrair) a representação binária de um número positivo e a representação binária de um número negativo não dá o resultado correto. O *hardware* precisa levar em consideração o valor do sinal de *bit* quando estiver calculando cada *bit* de saída, e é necessário um *hardware* diferente para executar a adição e a subtração. Esta complexidade de *hardware* é o motivo pelo qual pouquíssimos sistemas utilizam a notação sinal e magnitude para os seus inteiros.

Exemplo Qual é o resultado se você tentar somar diretamente as representações em 8 bits em sinal e magnitude de +10 e -4?

Solução

As representações em 8 bits em sinal e magnitude de +10 e -4 são 0b00001010 e 0b10000100. Somar estes dois números binários resulta em 0b10001110, que sistemas em sinal e magnitude interpretam como -14, e não como 6 (o resultado correto do cálculo).

Notação em Complemento de 2

Na notação em complemento de 2, o número negativo é representado invertendo-se cada *bit* da representação sem sinal do número e somando 1 ao resultado (descartando quaisquer *bits* de transbordo que excedam a largura da representação). O nome “complemento de 2” vem do fato de que a soma sem sinal de um número com n bits em complemento de 2 com o seu negativo é 2ⁿ.

Exemplo Qual é a representação em 8 bits, em complemento de 2, de -12, e qual é o resultado sem sinal da soma das representações de +12 e de -12?

Solução

A representação em 8 bits de +12 é 0b00001100, de modo que a representação de -12 em 8 bits, em complemento de 2, é 0b11110100. (Negar cada bit na representação positiva produz 0b11110011, e somar 1 produz o resultado final de 0b11110100.) Este processo é ilustrado na Fig. 2-6.

Valor original:	0b00001100	(12)
Negação de cada bit:	0b11110011	
Somar 1:	0b11110100	(representação em complemento de 2 de -12)

Fig. 2-6 Negação em complemento de 2.

Sumar as representações de +12 e -12 produz 0b00001100 + 0b11110100, o que é 0b100000000. Tratando isto como um número de 9 bits sem sinal, interpretamos este valor como 256 ($2^8 = 256$). Tratando este resultado como um número de 8 bits em complemento de 2, desconsideraremos o 1 de transbordo (o novo bit) para considerarmos o resultado em 8 bits, 0b00000000 = 0, que é o resultado que esperamos da soma de +12 com -12.

Números em complemento de 2 têm algumas propriedades úteis, o que explica porque eles são utilizados em praticamente todos os computadores modernos:

1. O sinal de um número pode ser determinado examinando-se o bit mais significativo da representação. Números negativos tem 1 no seu bit mais significativo; números positivos tem zero (0).
2. Negar um número duas vezes produz o número original, de modo que não é preciso um hardware especial para negar números negativos.
3. A notação em complemento de 2 tem apenas uma representação para 0, eliminando a necessidade de um hardware para detectar +0 e -0.
4. Mais importante, somar as representações em complemento de 2 de um número positivo e de um número negativo (descartando o transbordo) fornece o resultado correto na representação em complemento de 2. Além de eliminar a necessidade de um hardware especial para tratar a adição de números negativos, a subtração pode ser remodelada como uma adição, calculando a negação em complemento de 2 do subtraendo e somando a ele a representação em complemento de 2 do minuendo (por exemplo, 14 - 7 torna-se 14 + (-7)), o que reduz ainda mais os custos com o hardware.

Embora a representação sinal e magnitude compartilhe as duas primeiras vantagens dos números em complemento de 2, as outras duas dão à notação em complemento de 2 uma vantagem significativa sobre a notação em sinal e magnitude. Uma característica razoavelmente incomum da notação em complemento de 2 é que um número de n bits em complemento de 2 pode representar valores de $-(2^{n-1})$ a $+(2^{n-1} - 1)$. Esta assimetria vem do fato de que existe apenas uma representação para zero, o que permite que seja representado um número ímpar de quantidades que não são zero.

Exemplo Qual é o resultado de negar duas vezes a representação de +5, em 4 bits, em complemento de 2?

Solução

+5 = 0b0101. A negação em complemento de 2 é 0b1011 (-5). Negar novamente este valor dá 0b0101, o valor original.

Exemplo Some os valores +3 e -4 em notação de 4 bits em complemento de 2.

Solução

As representações de +3 e -4 com 4 bits em complemento de 2 são 0b0011 e 0b1100. Somar estes dois valores produz 0b1111, que é a representação em complemento de 2 de -1.

Exemplo Calcule $-3 - 4$ em notação de 4 bits, em complemento de 2.

Solução

Para executar a subtração, negamos o segundo operando e somamos. Assim, o cálculo que realmente queremos executar é $-3 + (-4)$. As representações em complemento de 2 de -3 e -4 são 0b1101 e 0b1100. Somando estes 2 valores, obtemos 0b11001 (um resultado de 5 bits, considerando o transbordo). Descartando o quinto bit, que excede a representação, temos 0b1001, a representação em complemento de 2 de -7.

A multiplicação de números em complemento de 2 é mais complicada, porque executar uma multiplicação direta das entradas, sem sinal, das representações em complemento de 2, não fornece o resultado correto. Os multiplicadores poderiam ser projetados para converter ambas as entradas para quantidades positivas e utilizar os bits de sinal das entradas originais para determinar o sinal do resultado, mas isto aumenta o tempo necessário para executar uma multiplicação. Existem métodos, como o método de *codificação de Booth*, que está além do escopo deste livro, para converter rapidamente números em complemento de 2 para um formato que pode ser facilmente multiplicado.

Como vimos, tanto números em sinal e magnitude quanto em complemento de 2 têm seus prós e contras. Números em complemento de 2 permitem implementações simples da adição e da subtração, enquanto números em sinal e magnitude facilitam a multiplicação e a divisão. Como a adição e a subtração são muito mais comuns em programas de computador do que a multiplicação e a divisão, praticamente todos os fabricantes de computadores optaram pela representação dos seus inteiros em complemento de 2, permitindo que eles "façam rapidamente o que é comum".

Extensão de Sinal

Em aritmética de computadores, algumas vezes é necessário converter números representados em um dado número de bits para uma representação que utiliza um número maior de bits. Por exemplo, um programa pode precisar somar uma entrada de 8 bits a um valor de 32 bits. Para obter o resultado correto, a entrada de 8 bits precisa ser convertida para um valor de 32 bits, antes que ela possa ser somada ao inteiro de 32 bits, o que é conhecido como *extensão de sinal*.

Converter números sem sinal para representações mais largas requer simplesmente preencher com zeros os bits à esquerda daqueles da representação original. Por exemplo, o valor sem sinal de 8 bits 0b10110110 torna-se o valor sem sinal de 16 bits 0b0000000010110110. Para fazer a extensão de sinal de um número em sinal e magnitude, move o bit de sinal (o bit mais significativo) da representação antiga para o bit de sinal da nova representação e preencha todos os bits adicionais na nova representação (incluindo a posição do antigo bit de sinal) com zeros.

Exemplo Qual é a representação em 16 bits, em sinal e magnitude, do valor em 8 bits, em sinal e magnitude, 0b10000111 (-7)?

Solução

Para estender o sinal de um número, movemos o antigo bit de sinal para o bit mais significativo da nova representação e preenchemos todas as outras posições de bit com zeros. Isto produz 0b1000000000000111 como sendo a representação de -7 em 16 bits, em sinal e magnitude.

A extensão de sinal de números em complemento de 2 é ligeiramente mais complicada. Para fazer a extensão de sinal de um número em complemento de 2, copie o bit mais significativo da antiga representação para cada bit adicional da nova representação. Assim, números positivos terão zeros em todos os bits acrescentados ao ir para uma representação mais larga, e números negativos terão uns em todas estas posições de bit.

Exemplo Qual é a extensão de sinal de 16 bits do valor 0b10010010 (-110) em 8 bits, em complemento de 2?

Solução

Para fazer a extensão de sinal deste número, copiamos o bit mais significativo para todas as novas posições de bit introduzidas pela extensão da representação. Isto produz 0b11111110010010. Ao negar isto, obtemos 0b000000001101110 (+110), confirmando que a extensão de sinal de números em complemento de 2 dá o resultado correto.

2.6 NÚMEROS EM PONTO FLUTUANTE

Números em ponto flutuante são utilizados para representar quantidades que não podem ser representados por inteiros, ou porque elas contêm valores fracionários, ou porque elas estão além da faixa que pode ser representada dentro da largura de bits do sistema. Praticamente, todos os computadores modernos utilizam a representação de ponto flutuante especificada pelo padrão IEEE 754, no qual os números são representados por uma mantissa e um expoente. De modo semelhante à notação científica, o valor de um número em ponto flutuante é: $mantissa \times 2^{exponte}$.

Esta representação permite que uma ampla gama de valores seja representada em um número relativamente pequeno de *bits*, incluindo valores fracionários e valores cuja magnitude é muito grande para ser representada em um inteiro com o mesmo número de *bits*. No entanto, isto cria o problema de que muitos dos valores na faixa da representação em ponto flutuante não podem ser representados exatamente, do mesmo modo que muitos dos números reais não podem ser representados por um número decimal com um número fixo de dígitos significativos. Quando um cálculo cria um valor que não pode ser representado exatamente pelo formato em ponto flutuante, o hardware precisa arredondar o resultado para um valor que possa ser representado exatamente. No padrão IEEE 754, o modo *default* de fazer o arredondamento é chamado de *modo de arredondamento*. Os valores são arredondados para o número mais próximo que pode ser representado e o resultado que cai exatamente entre os dois números que podem ser representados é arredondado de modo que o dígito menos significativo do seu resultado seja par. O padrão especifica diversos outros modos de arredondamento que podem ser escolhidos pelos programas, incluindo o arredondamento em direção ao zero, em direção a $+\infty$ e em direção a $-\infty$.

Exemplo Os modos de arredondamento no padrão IEEE podem ser aplicados a números decimais, bem como a representações binárias em ponto flutuante. Como os seguintes números decimais seriam arredondados para 2 dígitos significativos, utilizando o modo de arredondamento para o mais próximo?

- a. 1,345
- b. 78,953
- c. 12,5
- d. 13,5

Solução

- a. 1,345 é mais próximo de 1,3 do que de 1,4, de modo que ele será arredondado para 1,3. Uma outra maneira de ver isto é que o terceiro dígito mais significativo é menor do que 5, então ele é arredondado para 0 quando nós arredondamos para 2 dígitos significativos.
- b. Em 78,953, o terceiro dígito mais significativo é 9, que é arredondado para 10, de modo que 78,953 será arredondado para 79.
- c. Em 12,5, o terceiro dígito mais significativo é 5, de modo que nós o arredondamos na direção que torna par o dígito menos significativo do resultado. Neste caso, isto significa arredondar para baixo, para um resultado de 12.
- d. Aqui, temos que arredondar para 14 porque o terceiro dígito mais significativo é 5, e temos que fazer o arredondamento para cima, de modo a fazer com que o resultado seja par.

O padrão IEEE 754 especifica diferentes larguras de *bit* para números em ponto flutuante. As duas larguras mais utilizadas são de precisão simples e de precisão dupla, que estão ilustradas na Fig. 2-7. Números de precisão simples têm 32 *bits* de comprimento e contêm 8 *bits* de expoente, 23 *bits* de fração e 1 *bit* de sinal para o sinal do campo da fração. Os números de precisão dupla têm 11 *bits* de expoente, 52 *bits* de fração e 1 *bit* de sinal.

Sinal	Expoente	Fração	
1	8	23	Precisão simples (32 bits)
1	11	52	Precisão dupla (64 bits)

Fig. 2-7 Formatos de ponto flutuante IEEE 754.

Tanto o campo de expoente quanto o campo de fração de um número em ponto flutuante IEEE 754 são codificados de forma diferente do que as representações de inteiros que discutimos neste capítulo. O campo de fração é um número em sinal e magnitude que representa a parte fracionária de um número binário cuja parte inteira é assu-

midamente 1. Assim, a mantissa de um número em ponto flutuante IEEE 754 é sempre na forma $\pm 1.\text{fração}$, dependendo do valor do bit de sinal. Utilizar um "1 inicial" deste modo aumenta o número de dígitos significativos que podem ser representados por um número em ponto flutuante, em uma dada largura.

Exemplo Qual é o campo de fração da representação de 6.25 em ponto flutuante de precisão simples?

Solução

Números fracionários binários utilizam, com uma base 2, a mesma representação de numeração posicional que os números decimais, de modo que, por exemplo, o número binário $0b11.111 = 2^1 + 2^0 + 2^{-1} + 2^{-2} + 2^{-3} = 3.875$. Ao utilizar este formato, uma fração decimal pode ser convertida diretamente para uma fração binária, de modo que $6.25 = 2^1 + 2^{-2} = 0b110.01$.

Para achar o campo de fração, deslocamos a representação binária do número para baixo, de modo que o valor à esquerda da vírgula binária¹ seja 1; assim, 0b110.01 torna-se $0b1,1001 \times 2^1$. Na representação de fração normalizada utilizada em números de ponto flutuante, o 1 inicial é assumido, e apenas os valores à direita da vírgula binária (neste caso 1001) são representados. Ao estender o valor para o formato de fração de 23 bits em ponto flutuante, obtemos 1001 0000 0000 0000 0000 000 como o campo de fração. Note que, quando estendemos valores fracionários para representações mais largas, acrescentamos zeros à direita do último dígito significativo, em oposição à extensão de sinal de inteiros sem sinal, onde os zeros são acrescentados à esquerda dele.

O campo de expoente de um número em ponto flutuante utiliza uma representação denominada *por excesso* (*biased*), na qual um valor fixo é adicionado ao campo de expoente para determinar a sua representação. Para números em ponto flutuante de precisão simples, o excesso é 127 (o excesso = 1023 para números em precisão dupla), de modo que o valor do campo de expoente pode ser obtido ao subtrair 127 do número binário sem sinal contido no campo.

Exemplo Como seriam representados os números -45 e 123 em 8 bits na notação por excesso utilizada nos expoentes dos números com precisão simples?

Solução

O excesso para este formato é 127, de modo que somamos 127 a cada número para obtermos a representação por excesso.

$$-45 + 127 = 82 = 0b01010010$$

$$123 + 127 = 250 = 0b1111010$$

Exemplo Qual é o valor do expoente representado por um campo de expoente igual a 0b11100010 em um número em ponto flutuante com precisão simples?

Solução

$$0b11100010 = 226 \quad 226 - 127 = 99, \text{ de modo que o campo de expoente é igual a } 99.$$

Representações por excesso são, de certo modo, incomuns, mas elas têm uma vantagem significativa: permitem que comparações em ponto flutuante sejam feitas utilizando o mesmo hardware de comparação que o das comparações entre inteiros sem sinal, uma vez que valores maiores de uma codificação por excesso correspondem a valores maiores do número codificado. Dados os formatos para os campos de fração e de expoente, o valor de um número em ponto flutuante é $(-1, \text{ se o bit de sinal for igual a } 1; 1, \text{ se o bit de sinal for igual a } 0) \times (1.\text{fração}) \times 2^{(\text{expoente} - \text{excesso})}$.

Exemplo Qual é o valor do número em ponto flutuante de precisão simples representado pela cadeia de bits 0b0100 0000 0110 0000 0000 0000 0000?

Solução

Ao dividir este número de acordo com os campos especificados na Fig. 2-7, obtemos um bit de sinal 0, um campo de expoente $0b1000000 = 128$ e um campo de fração $0b11000000000000000000000000$. Subtrair o excesso 127 do campo de expoente gera um expoente de 1. A mantissa é $1,11_2 = 1.75$, uma vez que incluímos o 1 implícito no campo de fração, à esquerda da vírgula binária, de modo que o valor do número em ponto flutuante é $1 \times 1.75 \times 2^1 = 3.5$.

¹ Em uma representação binária, a vírgula binária é equivalente à vírgula decimal.

Números Não Normalizados e NaNs

O padrão IEEE de ponto flutuante especifica diversos padrões de *bits* que representam valores que não são possíveis representar com exatidão no formato de ponto flutuante base: o número zero, os números não normalizados e NaNs*. O 1 assumido na mantissa dos números em ponto flutuante permite um *bit* adicional de precisão na representação, mas evita que o valor zero seja representado com exatidão, uma vez que um campo de fração igual a 0 representa a mantissa 1,0. Uma vez que representar zero de forma exata é muito importante para cálculos numéricos, o padrão IEEE especifica que quando o campo de expoente de um número em ponto flutuante é zero, assume-se que o *bit* inicial da mantissa é zero. Assim, um número em ponto flutuante, com um campo de fração igual a zero e um campo de expoente igual a zero, representa zero de modo exato. Esta convenção também permite que sejam representados os números que estejam mais perto de zero do que $1,0 \times 2^{(1-\text{excesso})}$, se bem que eles tenham menos *bits* de precisão do que números que podem ser representados com um 1 assumido antes do campo de fração.

Números em ponto flutuante (exceto zero) que tenham um campo de expoente igual a zero são conhecidos como números *não normalizados* porque assumem um zero na parte inteira da sua mantissa. Isto contrasta com números que têm outros valores no campo de expoente, os quais tem um 1 assumido na parte inteira da sua mantissa e são conhecidos como números *normalizados*. Assume-se que todos os números não normalizados tenham um campo de expoente igual a $(1 - \text{excesso})$, em vez de $(0 - \text{excesso})$, que seria gerado apenas subtraíndo o excesso do valor dos seus expoentes. Isto fornece um pequeno intervalo entre o número normalizado de menor magnitude e o número não normalizado de maior magnitude que pode ser representado por um formato.

O outro tipo de valor especial no padrão de ponto flutuante são os NaNs, utilizados para sinalizar condições de erro como transbordos, transbordos negativos, divisão por zero e assim por diante. Quando uma destas condições de erro ocorre em uma operação, o hardware produz um NaN como resultado, em vez de sinalizar uma exceção. Operações subsequentes que recebam um NaN como uma das suas entradas copiam-no para suas saídas, em vez de executar os seus cálculos normais. NaNs são indicados pela presença de 1s em todos os *bits* do campo de expoente de um número em ponto flutuante, a menos que o campo de fração do número seja zero, representando um número infinito. A existência de NaNs torna mais fácil escrever programas que possam ser executados em diversos computadores diferentes, porque os programadores podem verificar os resultados de cada cálculo procurando erros dentro do programa, em vez de confiar nas funções de tratamento de exceção do sistema, as quais variam significativamente entre diferentes computadores. A Fig. 2-8 resume a interpretação dos diferentes valores dos campos de expoente e de fração em um número em ponto flutuante.

Campo de expoente	Campo de fração	Representa
0	0	0
0	não 0	$\pm (0, \text{fração}) \times 2^{(1 - \text{excesso})}$ [dependendo do <i>bit</i> de sinal]
Não 0, não todos 1	qualquer	$\pm (1, \text{fração}) \times 2^{(\text{expoente} - \text{excesso})}$ [dependendo do <i>bit</i> de sinal]
Todos 1	0	$\pm \text{infinito}$ [dependendo do <i>bit</i> de sinal]
Todos 1	não 0	NaN

Fig. 2-8 Interpretação dos números IEEE em ponto flutuante.

Aritmética com Números em Ponto Flutuante

Dadas as semelhanças entre a representação IEEE de números em ponto flutuante e a notação científica, não é surpresa que as técnicas utilizadas para a aritmética em ponto flutuante nos computadores sejam muito semelhantes às técnicas usadas na aritmética de números decimais que são expressos em notação científica. Um bom exemplo disso é a multiplicação em ponto flutuante.

Para multiplicar dois números utilizando notação científica, as mantissas dos números são multiplicadas e os expoentes somados. Se o resultado da multiplicação das mantissas for maior ou igual a 10, o produto das mantissas é deslocado de modo que haja exatamente um dígito diferente de zero à esquerda da vírgula decimal, e a soma dos expoentes é aumentada como necessário para manter igual o valor do produto. Por exemplo, para multi-

* N. de R. T. Provém da abreviação de *Not a Number*, que significa não é número.

plicar 5×10^3 por 2×10^6 , multiplicamos as mantissas ($5 \times 2 = 10$) e somamos os expoentes ($3 + 6 = 9$) para obter um resultado inicial de 10×10^9 . Uma vez que a mantissa deste número é maior do que 10, deslocamos a mantissa uma posição e somamos 1 ao expoente, para obter o resultado final de 1×10^{10} .

Os computadores multiplicam números em ponto flutuante utilizando um processo muito semelhante, como ilustrado na Fig. 2-9. O primeiro passo é multiplicar as mantissas dos dois números, utilizando técnicas análogas àquelas utilizadas para multiplicar números decimais, além de somar os seus expoentes. Números em ponto flutuante IEEE utilizam uma representação por excesso para os expoentes, de modo que somar os campos de expoente de dois números em ponto flutuante é ligeiramente mais complicado do que somar dois inteiros. Para calcular a soma dos expoentes, os campos de expoente dos dois números em ponto flutuante são tratados como inteiros e somados, e o valor do excesso é subtraído do resultado. Isto dá a representação por excesso correta para a soma dos dois expoentes.

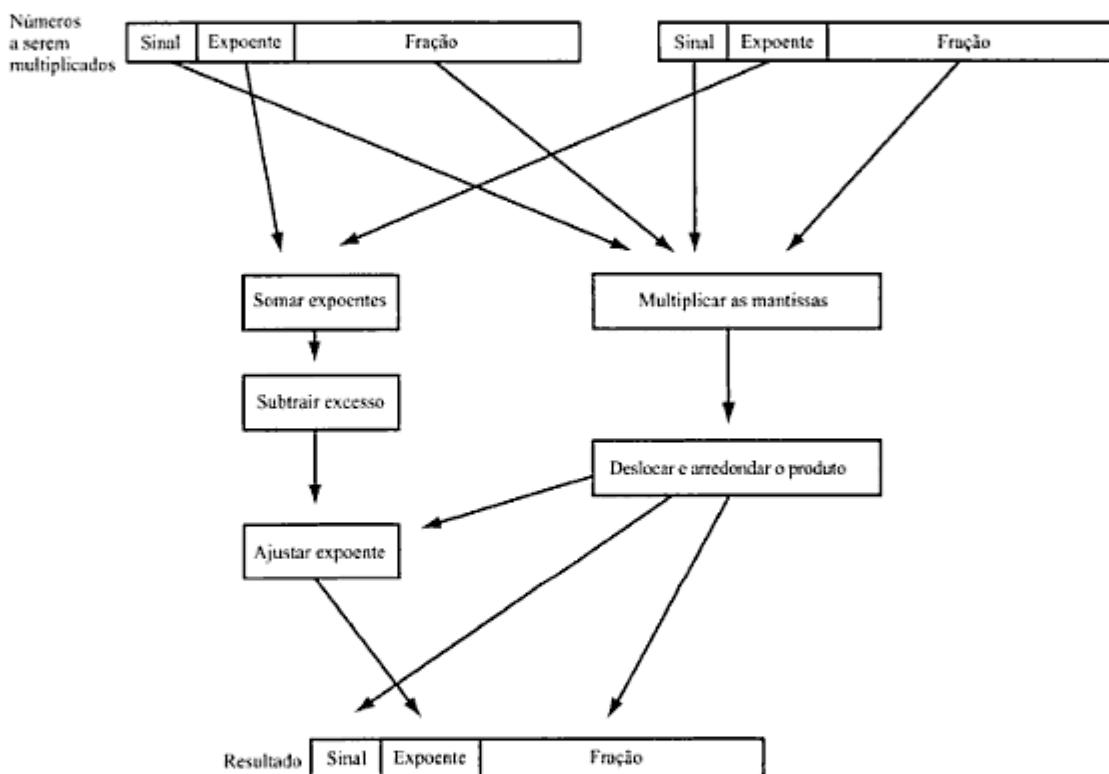


Fig. 2-9 Multiplicação em ponto flutuante.

Uma vez que as mantissas tenham sido multiplicadas, pode ser necessário deslocar o resultado de modo que apenas 1 bit permaneça à esquerda da vírgula binária (isto é, de modo que ele se encaixe no formato $1.xxxxx_2$), e a soma dos expoentes é incrementada de modo que o valor da mantissa $\times 2^{\text{exponente}}$ permaneça o mesmo. O produto das mantissas também pode ter que ser arredondado para caber no número de bits alocados para o campo de fração, uma vez que o produto de duas mantissas com n bits pode exigir até $2n$ bits para ser representado com exatidão. Uma vez que a mantissa tenha sido deslocada e arredondada, o produto final é montado a partir do produto das mantissas e da soma dos expoentes.

Exemplo Utilizando números em ponto flutuante de precisão simples, multiplique 2,5 por 0,75.

Solução

$2,5 = 0b0100\ 0000\ 0010\ 0000\ 0000\ 0000\ 0000$ (campo do expoente igual a $0b1000000$, campo da fração igual a $0b010\ 0000\ 0000\ 0000\ 0000$, mantissa igual a $1.010\ 0000\ 0000\ 0000\ 0000_2$). $0,75 = 0b0011\ 1111\ 0100\ 0000\ 0000\ 0000\ 0000_2$ (campo do expoente igual a $0b0111110$, campo da fração igual a $0b100\ 0000\ 0000\ 0000\ 0000_2$, mantissa igual a $1.100\ 0000\ 0000\ 0000\ 0000_2$). Somar os campos de expoente diretamente e subtrair

o excesso produz o resultado 0b01111111, a representação por excesso de 0. Multiplicar as mantissas dá o resultado de 1,111 0000 0000 0000 0000, que é convertido para um campo de fração igual a 0b111 0000 0000 0000 0000 0000, de modo que o resultado é 0b0011 1111 1111 0000 0000 0000 0000 = $1,111_2 \times 2^9 = 1.875$.

A divisão em ponto flutuante é muito semelhante à multiplicação. O hardware calcula o quociente das mantissas e a diferença entre os expoentes dos números que estão sendo divididos, somando o valor do excesso à diferença entre os campos de expoente dos dois números, de modo a obter a representação por excesso correta do resultado. O quociente das mantissas é, então, deslocado e arredondado para caber dentro do campo de fração do resultado.

A adição em ponto flutuante exige um conjunto diferente de cálculos, que está ilustrado na Fig. 2-10. Do mesmo modo que com a adição dos números em notação científica, o primeiro passo é deslocar uma das entradas até que ambas tenham o mesmo expoente. Ao somar números em ponto flutuante, o número com o menor expoente é deslocado para a direita. Por exemplo, ao somar $1,01_2 \times 2^3$ e $1,001_2 \times 2^0$, o menor valor é deslocado para tornar-se $0,001001_2 \times 2^3$. Deslocar o número com o menor expoente permite o uso de técnicas para executar o arredondamento, as quais retêm apenas as informações necessárias a respeito dos bits menos significativos do menor número, reduzindo o número de bits que efetivamente precisam ser somados.

Uma vez que as entradas tenham sido deslocadas, as suas mantissas são somadas e o resultado é deslocado, se necessário. Finalmente, o resultado é arredondado para caber no campo de fração e o cálculo está completo. A subtração em ponto flutuante utiliza o mesmo processo, exceto que é calculada a diferença entre as mantissas deslocadas, ao invés de somá-las.

Exemplo Utilizando números em ponto flutuante de precisão simples, calcule a soma de 0,25 e 1.5.

Solução

$$0,25 = 0b0011\ 1110\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ (1,0 \times 2^{-2})$$

$$1,5 = 0b0011\ 1111\ 1100\ 0000\ 0000\ 0000\ 0000\ 0000\ (1,5 \times 2^0)$$

Para somar estes números, deslocamos aquele com o menor expoente (0,25) para a direita até que ambos os expoentes sejam os mesmos (neste caso, duas posições). Isto resulta em mantissas iguais a 1,100 0000 0000 0000 0000 e 0,010 0000 0000 0000 0000 para os dois números (incluindo os 1s assumidos nos valores a serem deslocados). Somar estas duas mantissas produz o resultado de 1,110 0000 0000 0000 0000 $\times 2^0$ (o expoente da entrada com o maior expoente) = 1,75. A representação em precisão simples do resultado final é 0b0011 1111 1110 0000 0000 0000 0000.

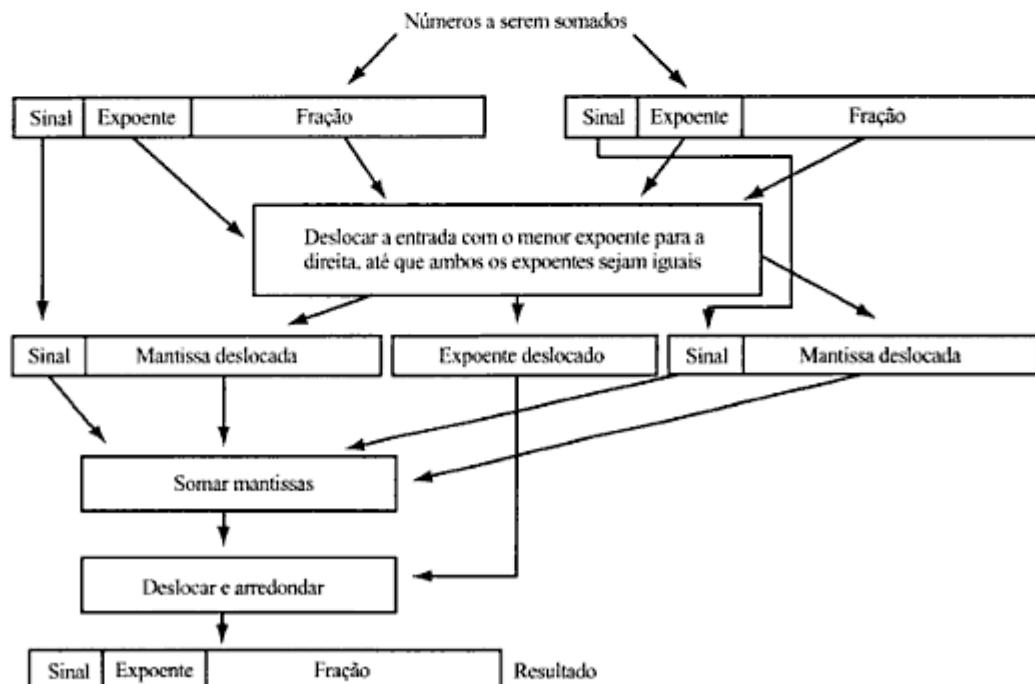


Fig. 2-10 Adição em ponto flutuante.

2.7 RESUMO

Este capítulo descreveu as técnicas que os sistemas de computadores utilizam para representar e manipular dados. Em geral, os computadores utilizam dois níveis de abstração para representar os dados, dispostos como camadas um sobre o outro. O nível mais baixo de abstração é a convenção dos sinais digitais que mapeia os números 0 e 1 sobre sinais elétricos analógicos. São utilizadas várias convenções de sinalização, mas a mais comum delas define o nível lógico 0 como a faixa de tensões próxima à tensão de terra do sistema e o nível lógico 1 como a faixa de tensões próxima à tensão de alimentação do sistema (V_{dd}). Para proteger o circuito contra ruídos elétricos, são definidas faixas para os níveis de tensão que podem ser gerados por um circuito e os valores de entrada para os quais é garantida a interpretação como 0 ou 1 por outro circuito.

O segundo nível de abstração define como os grupos de *bits* são utilizados para representar números inteiros e não inteiros. Números inteiros positivos são representados utilizando um sistema de numeração posicional análogo ao sistema decimal. As representações em sinal e magnitude ou em complemento de 2 são utilizadas para representar inteiros negativos, sendo que o complemento de 2 é a representação mais comum porque permite implementações simples, tanto da adição como da subtração.

Valores que não são inteiros são representados por meio de números em ponto flutuante. Números em ponto flutuante são semelhantes à notação científica, representando os números por meio de uma mantissa e de um expoente. Isto permite que uma faixa muito ampla de valores seja representada com um número pequeno de *bits*, embora nem todos os valores possam ser representados com exatidão. A multiplicação e a divisão podem ser implementadas de forma simples com números em ponto flutuante, enquanto que a adição e a subtração são mais complicadas, uma vez que a mantissa de um dos números precisa ser deslocada para fazer com que os expoentes dos dois números sejam iguais.

Utilizando uma combinação de números inteiros e em ponto flutuante, os programas podem executar uma ampla variedade de operações aritméticas. No entanto, todas estas representações têm suas limitações. A faixa dos inteiros que o computador pode representar é limitada pela sua largura de *bits*, e a tentativa de executar cálculos que gerem resultados que estejam fora desta faixa produzirão resultados incorretos. Os números em ponto flutuante também têm uma faixa limitada, embora a representação mantissa-expoente torne este limite muito maior. A limitação mais significativa dos números em ponto flutuante vem do fato de que eles só podem representar números até uma determinada quantidade de dígitos significativos, devido ao número limitado de *bits* utilizados na representação da mantissa de cada número. Os cálculos que exigem uma precisão maior do que a representação em ponto flutuante não executarão as operações corretamente.

Problemas Resolvidos

Convenções de Sinais (I)

- 2.1** Suponha que um sistema digital tenha $V_{DD} = 3,3\text{ V}$, $V_{IL} = 1,2\text{ V}$, $V_{OL} = 0,7\text{ V}$, $V_{IH} = 2,1\text{ V}$, $V_{OH} = 3,0\text{ V}$. Qual é a margem de ruído para esta convenção de sinais?

Solução

A margem de ruído é a menor das diferenças entre os níveis de uma saída ou entrada válidas para um 0 ou 1. Para esta convenção de sinais $|V_{OL} - V_{IL}| = 0,5\text{ V}$ e $|V_{OH} - V_{IH}| = 0,9\text{ V}$. Portanto, a margem de ruído para esta convenção de sinais é $0,5\text{ V}$, indicando que o valor de qualquer sinal de saída de uma porta lógica pode ser modificado por até $0,5\text{ V}$ devido à ruído no sistema, sem tornar-se um valor inválido.

Convenções de Sinais (II)

- 2.2** Suponha que seja dito que uma dada convenção de sinais tem $V_{DD} = 3,3\text{ V}$, $V_{IL} = 1,0\text{ V}$, $V_{OL} = 1,2\text{ V}$, $V_{IH} = 2,1\text{ V}$, $V_{OH} = 3,0\text{ V}$. Por que esta seria uma convenção de sinais ruim?

Solução

Nesta convenção de sinais, $V_{IH} > V_{OL} > V_{IL}$. Isto significa que uma porta lógica pode gerar um valor de saída que esteja na região proibida, entre V_{IH} e V_{IL} . Não é garantido como tal valor de saída será interpretado por qualquer porta que receba este valor como uma entrada. Um outro modo de expressar isto é dizer que esta convenção de sinais permite que uma porta que esteja tentando dar saída a um 0 produza uma tensão de saída que não será interpretada como 0 pela entrada de uma outra porta, mesmo que não haja ruído no sistema.

Representação Binária de Inteiros Positivos (I)

2.3 Mostre como os seguintes inteiros seriam representados por um sistema que utiliza inteiros de 8 bits sem sinal.

- a. 37
- b. 89
- c. 4
- d. 126
- e. 298

Solução

- a. $37 = 32 + 4 + 1 = 2^5 + 2^2 + 2^0$. Portanto, a representação binária sem sinal, em 8 bits, de 37 é 0b00100101.
- b. $89 = 64 + 16 + 8 + 1 = 2^6 + 2^4 + 2^3 + 2^0 = 0b01011001$.
- c. $4 = 2^2 = 0b00000100$.
- d. $126 = 64 + 32 + 16 + 8 + 4 + 2 = 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 = 0b01111110$.
- e. Esta é uma “pegadinha”. O maior valor que pode ser representado por um número de 8 bits sem sinal é $2^8 - 1 = 255$. O número 298 é maior do que 255, de modo que ele não pode ser representado por um número binário de 8 bits sem sinal.

Representação Binária de Inteiros Positivos (II)

2.4 Qual é o valor decimal dos seguintes inteiros binários sem sinal?

- a. 0b1100
- b. 0b100100
- c. 0b11111111

Solução

- a. $0b1100 = 2^3 + 2^2 = 8 + 4 = 12$
- b. $0b100100 = 2^5 + 2^2 = 32 + 4 = 36$
- c. $0b11111111 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$

Notação Hexadecimal (I)

2.5 Quais são as representações hexadecimais dos seguintes inteiros?

- a. 67
- b. 142
- c. 1348

Solução

- a. $67 = (4 \times 16) + 3 = 0x43$
- b. $142 = (8 \times 16) + 14 = 0x8e$
- c. $1348 = (5 \times 16 \times 16) + (4 \times 16) + 4 = 0x544$

Notação Hexadecimal (II)

2.6 Quais são os valores decimais dos seguintes números hexadecimais?

- a. 0x1b
- b. 0xa7
- c. 0x8ce

Solução

- a. $0x1b = (1 \times 16) + 11 = 27$
- b. $0xa7 = (10 \times 16) + 7 = 167$
- c. $0x8ce = (8 \times 16 \times 16) + (12 \times 16) + 14 = 2254$

Adição de Inteiros sem Sinal

- 2.7 Calcule as somas dos seguintes pares de inteiros sem sinal:
- 0b11000100 + 0b00110110
 - 0b00001110 + 0b10101010
 - 0b11001100 + 0b00110011
 - 0b01111111 + 0b00000001

Solução

(Note que todos estes problemas podem ter sua solução verificada, ao converter as saídas e entradas para decimal.)

- 0b11111010
- 0b10111000
- 0b11111111
- 0b10000000

Multiplicação de Inteiros sem Sinal

- 2.8 Calcule o produto dos seguintes pares de inteiros sem sinal. Produza o resultado em 8 bits.
- 0b1001 × 0b0110
 - 0b1111 × 0b1111
 - 0b0101 × 0b1010

Solução

- $0b1001 \times 0b0110 = (0b1001 \times 0b100) + (0b1001 \times 0b10) = 0b100100 + 0b10010 = 0b00110110$
- 0b11100001
- 0b00110010

Número de Bits Necessários

- 2.9 Quantos bits são necessários para representar os seguintes números decimais como inteiros binários sem sinal?
- 12
 - 147
 - 384
 - 1497

Solução

- 12 é maior que $2^3 - 1$ e menor que $2^4 - 1$, de modo que 12 não pode ser representado em um inteiro de 3 bits sem sinal, mas pode ser representado em um inteiro binário de 4 bits. Portanto, são necessários 4 bits.
- $2^4 - 1 < 147 < 2^5 - 1$, de modo que são necessários 8 bits.
- $2^8 - 1 < 384 < 2^9 - 1$, de modo que são necessários 9 bits.
- $2^{10} - 1 < 1497 < 2^{11} - 1$, de modo que são necessários 11 bits.

Faixas de Representações Binárias

- 2.10 Quais são os maiores e os menores inteiros que podem ser representados com valores de 4, 8 e 16 bits, utilizando:
- Representação binária sem sinal
 - Representação binária em sinal e magnitude
 - Representação em complemento de 2

E ainda, por que as respostas das letras b e c são diferentes?

Solução

- Na representação binária sem sinal, 0 é o menor valor que pode ser representado. O maior valor que pode ser representado em um inteiro binário de n bits sem sinal é $2^n - 1$, dando uma representação máxima de valores iguais a 15, 255 e 65.535 para inteiros sem sinal de 4, 8 e 16 bits.

b. Representações em sinal e magnitude utilizam um *bit* para registrar o sinal de um número, permitindo que eles representem valores de $-(2^{n-1} - 1)$ a $2^{n-1} - 1$. Isto propicia uma faixa de -7 a $+7$, para valores de $4\ bits$; -127 a $+127$, para valores de $8\ bits$; e -32.767 a $+32.767$, para valores de $16\ bits$.

c. Inteiros de $n\ bits$ em complemento de 2 podem representar valores de $-(2^{n-1})$ a $2^{n-1} - 1$. Portanto, inteiros de $4\ bits$ em complemento de 2 podem representar valores de -8 a $+7$, números de $8\ bits$ podem representar valores de -128 a $+127$ e números de $16\ bits$ podem representar valores de -32.768 a $+32.767$.

Representações em sinal e magnitude têm duas representações para 0, enquanto que representações em complemento de 2 têm apenas uma. Isto dá às representações em complemento de 2 a capacidade de representar um valor a mais do que as representações em sinal e magnitude, com o mesmo número de *bits*.

Representação com Sinal de Magnitude

2.11 Converta os seguintes números decimais para a representação em $8\ bits$ em sinal e magnitude:

- a. 23
- b. -23
- c. -48
- d. -65

Solução

a. Na representação em sinal e magnitude, inteiros positivos são representados do mesmo modo que eles o são em representação binária sem sinal, exceto que o *bit* mais significativo da representação é reservado para o *bit* de sinal. Portanto, a representação de 23, em $8\ bits$ em sinal e magnitude, é 0b00010111.

b. Para obter a representação de -23 em sinal e magnitude, simplesmente ajustamos o *bit* de sinal da representação de +23 para 1, produzindo 0b10010111.

- c. 0b10110000
- d. 0b11000001

Notação em Complemento de 2

2.12 Dê a representação com $8\ bits$, em complemento de 2, dos valores do Problema 2.11.

Solução

a. Da mesma forma que a representação em sinal e magnitude, a representação de um número positivo em complemento de 2 é a mesma que a representação sem sinal daquele número, produzindo 0b00010111 como a representação de 23 com $8\ bits$, em complemento de 2.

b. Para negar um número na representação em complemento de 2, invertemos todos os *bits* da sua representação e somamos 1 ao resultado, produzindo 0b11101001 como a representação de -23 com $8\ bits$, em complemento de 2.

- c. 0b11010000
- d. 0b10111111

Extensão de Sinal

2.13 Dê a representação em $8\ bits$ dos números 12 e -18, nas notações em sinal e magnitude e em complemento de 2, e mostre como estas representações têm o sinal estendido para dar representações de $16\ bits$ em cada notação.

Solução

As representações em $8\ bits$ em sinal e magnitude de 12 e de -18 são 0b00001100 e 0b10010010, respectivamente. Para fazer a extensão de sinal de um número em sinal e magnitude, o *bit* de sinal é copiado para o *bit* mais significativo da nova representação e o sinal de *bit* da antiga representação é zerado, dando as representações em sinal e magnitude em $16\ bits$ de 0b0000000000001100 para 12 e 0b1000000000001010 para -18.

As representações de 12 e -18 com $8\ bits$, em complemento de 2, são 0b00001100 e 0b11101110. Números em complemento de 2 têm o sinal estendido copiando-se o *bit* mais significativo do número para os *bits* adicionais da nova representação, dando representações de $16\ bits$ iguais a 0b0000000000001100 e 0b111111110110, respectivamente.

Matemática com Inteiros em Complemento de 2

2.14 Utilizando inteiros de 8 bits em complemento de 2, execute os seguintes cálculos:

- $-34 + (-12)$
- $17 - 15$
- $-22 - 7$
- $18 - (-5)$

Solução

- Na notação em complemento de 2, $-34 = 0b11011110$ e $-12 = 0b11110100$. Somando-os, obtemos $0b11010010$ (lembre-se de que o nono bit é descartado quando são somados dois números de 8 bits em complemento de 2). Isto é igual a -46 , a resposta correta.
- Aqui, podemos tirar proveito do fato de que estamos utilizando notação em complemento de 2 para transformar $17 - 15$ em $17 + (-15)$, ou $0b00010001 + 0b11110001 = 0b00000010 = 2$.
- Novamente, transformamos a expressão para $-22 + (-7)$ para obter o resultado $0b11100011 = -2$.
- Como no item c, transformamos a expressão em $18 + 5 = 0b00010111 = 23$.

Comparando Representações de Inteiros

2.15 Qual das duas representações de inteiros descritas neste capítulo (sinal e magnitude e complemento de 2) seria a mais adequada para as seguintes situações:

- Quando for fundamental que o *hardware* de negação de um número seja o mais simples possível.
- Quando a maioria das operações matemáticas executadas será de adições e subtrações.
- Quando a maioria das operações matemáticas executadas será de multiplicações e divisões.
- Quando for essencial que seja tão fácil quanto possível detectar quando o número é positivo ou negativo.

Solução

- Neste caso, a representação em sinal e magnitude seria a melhor, porque negar um número exige apenas a inversão do bit de sinal.
- Números em complemento de 2 permitem um *hardware* mais simples para adições e subtrações do que os números em sinal e magnitude. Com números em complemento de 2, não é necessário *hardware* adicional para somar números positivos e negativos – tratar estes números como valores sem sinal e somá-los fornece o resultado correto em complemento de 2. A subtração pode ser implementada negando-se o segundo operando efetuando, então, a adição.
- Em contraste, as representações em sinal e magnitude exigem um *hardware* diferente para executar subtrações ou para somar números positivos e negativos, fazendo com que esta representação seja mais onerosa se a maioria dos cálculos a serem executados forem adições e/ou subtrações.
- A representação em sinal e magnitude é a melhor neste caso, porque a multiplicação e a divisão podem ser implementadas tratando-se as partes de magnitude dos números como inteiros sem sinal e, então, determinando o sinal do resultado examinando-se os bits de sinal dos dois valores.
- Neste caso, as duas representações estão muito próximas. Em geral, o sinal de um número em qualquer uma das representações pode ser determinado examinando-se o bit mais significativo do número – se o bit mais significativo for 1, o número é negativo. A exceção é quando o número é 0. As representações em sinal e magnitude têm duas representações para zero, uma com o bit de sinal igual a 1 e uma com o bit de sinal igual a 0, enquanto existe apenas uma representação do zero na notação em complemento de 2.

Resumindo, as duas representações são equivalentes, se não for importante que os valores zero sejam detectados como positivos e negativos. Se for importante determinar se um número é positivo, negativo ou zero, os números em complemento de 2 são ligeiramente melhores.

Arredondamento

2.16 Utilizando o arredondamento para o mais próximo, faça o arredondamento dos seguintes valores decimais para três dígitos significativos:

- 1,234
- 8.940,999
- 179,5
- 178,5

Solução

- a. 1,23
- b. 8.940 (dígitos significativos não precisam estar à direita da vírgula decimal)
- c. 180 (arredondado para um número par)
- d. 180 (arredondado para um número par)

Representações em Ponto Flutuante (I)

2.17 Converta os seguintes valores para ponto flutuante IEEE de precisão simples:

- a. 128
- b. -32,75
- c. 18,125
- d. 0,0625

Solução

- a. $128 = 2^7$, produzindo um campo de expoente igual a 134, um bit de sinal igual a 0 e um campo de fração igual a 0 (por causa do 1 assumido). Portanto, $128 = 0b0100\ 0011\ 0000\ 0000\ 0000\ 0000\ 0000$, em formato de ponto flutuante com precisão simples.
- b. $-32,75 = -100000,11_2$ ou $-1,0000011_2 \times 2^5$. A representação de -32,75 em ponto flutuante com precisão simples é $0b1100\ 0010\ 0000\ 0011\ 0000\ 0000\ 0000$.
- c. $18,125 = 10010,001_2$ ou $1,0010001_2 \times 2^4$, produzindo a representação em ponto flutuante com precisão simples igual a $0b0100\ 0001\ 1001\ 0001\ 0000\ 0000\ 0000$.
- d. $0,0625 = 0,0001_2$ ou 1×2^{-4} , produzindo a representação em ponto flutuante com precisão simples igual a $0b0011\ 1101\ 1000\ 0000\ 0000\ 0000\ 0000$.

Representações em Ponto Flutuante (II)

2.18 Quais valores estão representados pelos seguintes números em ponto flutuante IEEE de precisão simples?

- a. $0b1011\ 1101\ 0100\ 0000\ 0000\ 0000\ 0000$
- b. $0b0101\ 0101\ 0110\ 0000\ 0000\ 0000\ 0000$
- c. $0b1100\ 0001\ 1111\ 0000\ 0000\ 0000\ 0000$
- d. $0b0011\ 1010\ 1000\ 0000\ 0000\ 0000\ 0000$

Solução

- a. Para este número, o bit de sinal = 1, o campo de expoente = 122, de modo que o expoente = -5. O campo da fração = 100 0000 0000 0000 0000, então, esta sequência binária representa $-1,1_2 \times 2^{-5} = -0,046875$.
- b. Aqui, temos um bit de sinal = 0, o campo de expoente = 170, de modo que o expoente é 43 e o campo da fração = 110 0000 0000 0000 0000, então, o valor deste número é $1,11_2 \times 2^{43} = 1,539 \times 10^{13}$ (para quatro dígitos significativos).
- c. $-1,111_2 \times 2^4 = -30$
- d. $1,0_2 \times 2^{-10} = 0,0009766$ (para quatro dígitos significativos)

NaNs e Números Não Normalizados

2.19 Para cada valor IEEE de precisão simples abaixo, explique que tipo de número (normalizado, não normalizado, infinito, zero ou NaN) eles representam. Se a quantidade tiver um valor, cite-o.

- a. $0b0111\ 1111\ 1000\ 1111\ 0000\ 1111\ 0000\ 0000$
- b. $0b0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000$
- c. $0b0100\ 0010\ 0100\ 0000\ 0000\ 0000\ 0000\ 0000$
- d. $0b1000\ 0000\ 0100\ 0000\ 0000\ 0000\ 0000\ 0000$
- e. $0b1111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000$

Solução

- O campo de expoente deste número é composto por 1s e o seu campo de fração não é 0, de modo que ele é um NaN.
- Este número tem um campo de expoente igual a 0, um bit de sinal igual a 0 e um campo de fração igual a 0, que é a representação em ponto flutuante IEEE para +0.
- Este número têm um campo de expoente igual a 132 e um campo de fração igual a 100 0000 0000 0000 0000 0000. Uma vez que o seu campo de expoente não contém apenas 0s nem 1s, ele representa um número normalizado, com o valor de $1,1_2 \times 2^{\text{132}} = 48$.
- Este número tem um campo de expoente todo em zero e o seu campo de fração é não zero, de modo que ele é um número não normalizado. O seu valor é $-0,1_2 \times 2^{-126} = -2^{-127} = -5,877 \times 10^{-39}$ (para quatro dígitos significativos).
- O campo de expoente deste número é todo de 1s, o seu campo de fração é zero e o seu bit de sinal é 1, de modo que ele representa $-\infty$.

Aritmética com Números em Ponto Flutuante

2.20 Utilize números em ponto flutuante IEEE de precisão simples para calcular os seguintes valores:

- 32×16
- $147,5 \times 0,25$
- $0,125 \times 8$
- $13,25 \times 4,5$

Solução

- $32 = 2^5$ e $16 = 2^4$, de modo que as representações em ponto flutuante destes números são 0b0100 0010 0000 0000 0000 0000 0000 0000 e 0b0100 0001 1000 0000 0000 0000 0000. Para multiplicá-los, convertemos os seus campos de fração em uma mantissa e multiplicamos, somamos os campos de expoente e subtraímos o excesso da soma. Isto dá como resultado um campo de expoente igual a 10001000 e um campo de fração igual a 000 0000 0000 0000, uma vez que removemos o 1 assumido do produto das mantissas. O número em ponto flutuante resultante é 0b0100 0100 0000 0000 0000 0000 = $2^9 = 512$.
- $147,5 = 1,00100111_2 \times 2^7 = 0b0100 0011 0001 0011 1000 0000 0000 0000$. $0,25 = 1,0_2 \times 2^{-2} = 0b0011 1110 1000 0000 0000 0000 0000$. Deslocando o número com o menor expoente (0,25) para a direita para tornar os expoentes de ambos os números iguais, resulta em $0,25 = 0,00000001_2 \times 2^7$. Somando as mantissas, temos a soma $1,00100111_2 \times 2^7 = 0b0100 0011 0001 0011 1100 0000 0000 0000$.
- Convertendo estes números para ponto flutuante, temos as representações 0b0011 1110 0000 0000 0000 0000 0000 para 0,125 e 0b0100 0001 0000 0000 0000 0000 0000 para 8. Multiplicando as mantissas e somando os expoentes temos o resultado de 0b0011 1111 1000 0000 0000 0000 0000 = 1.
- $13,25 = 1,10101_2 \times 2^3 = 0b0100 0001 0101 0100 0000 0000 0000$. $4,5 = 1,001_2 \times 2^{-2} = 0b0100 0000 1001 0000 0000 0000 0000$. Deslocando 4,5 para a direita uma posição para tornar os expoentes de ambos os números iguais, resulta em $4,5 = 0,1001_2 \times 2^3$. Somando as mantissas, temos o resultado $1,000111_2 \times 2^3$, de modo que temos que deslocar isto para baixo uma posição para obtermos $1,000111_2 \times 2^2$. A representação disto em ponto flutuante com precisão simples é 0b0100 0001 1000 1110 0000 0000 0000.

Capítulo 3

Organização de Computadores

3.1 OBJETIVOS

Os dois últimos capítulos estabeleceram as bases para a nossa discussão de arquitetura de computadores, ao explicar como os projetistas de computadores descrevem e analisam o desempenho e como os computadores representam e manipulam valores do mundo real. Neste capítulo, começamos a cobrir a arquitetura de computadores propriamente dita, descrevendo os blocos construtivos básicos que compõem sistemas de computadores convencionais: processadores, memória e E/S. Também descreveremos, brevemente, como os programas são representados internamente pelos sistemas de computadores, e como os sistemas operacionais organizam os programas que controlam os dispositivos físicos que compõem um computador.

Após completar este capítulo, você deverá:

1. Compreender os conceitos básicos sobre processadores, memória e dispositivos de E/S, e ser capaz de descrever as suas funções.
2. Estar familiarizado com arquiteturas de computadores de programas armazenados em memória.
3. Compreender as funções básicas dos sistemas operacionais.

3.2 INTRODUÇÃO

Como indicado na Fig. 3-1, a maioria dos sistemas de computadores podem ser divididos em três subsistemas: o processador, a memória e o subsistema de entrada e saída (E/S). O processador é responsável pela execução dos programas, a memória fornece espaço de armazenamento para os programas e os dados aos quais eles fazem referência e o subsistema de E/S permite que o computador e a memória controlem os dispositivos que interagem com o mundo externo ou que armazenem dados, como o CD-ROM, discos rígidos e a placa de vídeo/monitor, mostrados na figura.

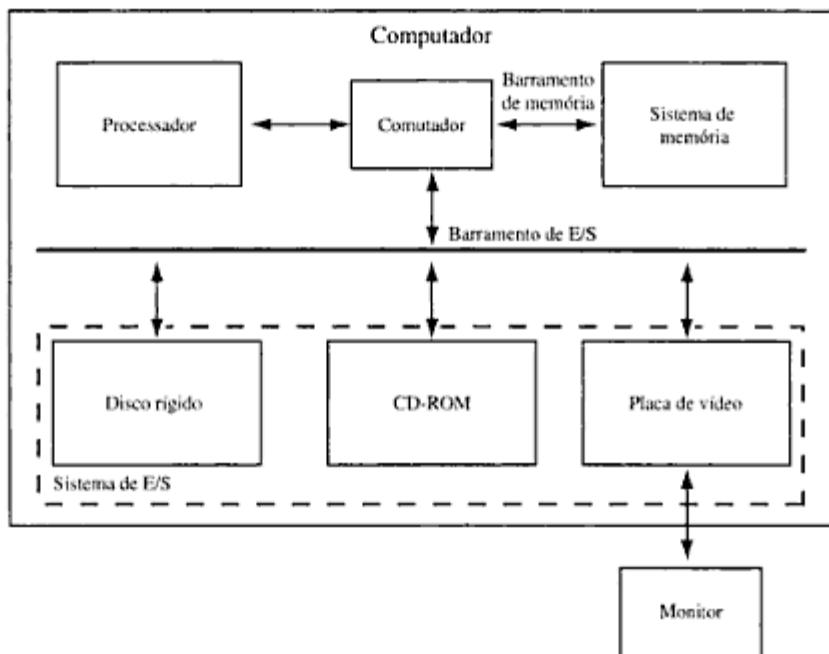


Fig. 3-1 Organização básica de computadores.

Na maioria dos sistemas, o processador tem um único barramento de dados que é conectado ao módulo comutador, tal como a ponte PCI encontrada na maioria dos sistemas PC, embora alguns processadores integrem diretamente o módulo de comutação no mesmo circuito integrado que o processador, de modo a reduzir o número de chips necessários para construir um sistema, bem como o seu custo. O comutador comunica-se com a memória através de um *barramento de memória*, um conjunto dedicado de linhas que transfere dados entre estes dois sistemas. Um *barramento de E/S* distinto conecta o comutador com os dispositivos de E/S. Normalmente são utilizados barramentos separados porque o sistema de E/S geralmente é projetado de forma a ser o mais flexível possível para suportar diversos tipos de dispositivos de E/S e o de memória é projetado para fornecer a maior largura de banda possível entre o processador e o sistema de memória.

3.3 PROGRAMAS

Programas são seqüências de instruções que dizem ao computador o que fazer, embora a visão que um computador tem das instruções que compõem um dado programa seja muito diferente da visão de quem o escreveu. Para o computador, um programa é composto de uma seqüência de números que representam operações individuais. Estas operações são conhecidas como *instruções de máquina*, ou apenas *instruções*, e o conjunto de operações que um dado processador pode executar é conhecido como *conjunto de instruções*.

Praticamente todos os computadores em uso atualmente são computadores com *memória de programa* que representam os programas como números que são armazenados no mesmo espaço de endereçamento que os dados.¹ A abstração de programas armazenados (representando instruções como números armazenados na memória) foi um dos principais avanços na arquitetura dos primeiros computadores. Antes disso, muitos computadores eram programados pelo ajuste de interruptores ou refazendo a conexão de placas de circuitos para definir o novo programa, o que exigia uma grande quantidade de tempo, além de ser muito sujeito a erros.

A abstração de programas armazenados em memória fornece duas vantagens principais sobre as abordagens anteriores. Primeiro, ela permite que os programas sejam armazenados e carregados facilmente na máquina. Uma vez que o programa tenha sido desenvolvido e depurado, os números que representam as suas instruções podem ser

¹ Computadores de programas armazenados em memória também são chamados de computadores von Neumann, em homenagem a John von Neumann, um dos desenvolvedores deste conceito.

escritos em um dispositivo de armazenamento, permitindo que o programa seja carregado novamente para a memória em algum momento no futuro. Nos primeiros sistemas, os dispositivos de armazenamento mais comuns eram cartões perfurados e fitas de papel. Os sistemas modernos geralmente utilizam meio magnético, como discos rígidos. A capacidade de armazenar programas como se fossem dados elimina os erros quando o programa é recarregado (assumindo que o dispositivo no qual o programa está armazenado seja livre de erros), enquanto que solicitar que um usuário introduza o programa novamente, a cada vez que ele for utilizado, geralmente introduz erros que têm que ser corrigidos antes que o programa possa ser executado corretamente – imagine ter que depurar o seu processador de textos a cada vez que você o executasse!

Segundo, e talvez de modo ainda mais significativo, a abstração de programas armazenados em memória permite que os programas tratem a si mesmos ou a quaisquer outros programas como se fossem dados. Programas que tratam a si mesmos como dados são chamados de *programas automodificáveis*, isto é, algumas das instruções em um programa calculam outras instruções do próprio programa. Tais programas eram comuns nos primeiros computadores, pois frequentemente eles eram mais rápidos do que programas que não se modificavam e porque os primeiros computadores implementavam um número pequeno de instruções, fazendo com que fosse difícil criar algumas operações sem o código automodificável. De fato, esse tipo de código foi o único modo para implementar um desvio condicional em, pelo menos, um dos primeiros computadores – o conjunto de instruções não fornecia uma operação de desvio condicional, de modo que os programadores implementaram desvios condicionais escrevendo um código automodificável que calculava os endereços de destino de instruções de desvio incondicional à medida que o programa era executado.

O código automodificável tornou-se menos comum nas máquinas mais modernas, porque mudar o programa durante a execução dificulta a depuração dos programas. À medida que os computadores se tornavam mais rápidos, a facilidade de implementação e depuração de programas tornou-se mais importante do que as melhorias de desempenho que podiam ser obtidas por meio de código automodificável, na maioria dos casos. Além disto, sistemas de memória com *caches* (discutidos no Capítulo 9) tornam o código automodificável menos eficiente, diminuindo as melhorias de desempenho que são obtidas utilizando-se esta técnica.

Ferramentas de Desenvolvimento de Programas

Os programas que tratam outros programas como dados são muito comuns, e a maioria das ferramentas de desenvolvimento de programas caem dentro desta categoria. Estas ferramentas incluem os *compiladores* que convertem programas em linguagens de alto nível, como C e FORTRAN, em linguagem de montagem (*assembly*), os *montadores* que convertem as instruções em linguagem de montagem em representações numéricas utilizadas pelo processador e os *ligadores* (*linkers*) que unem diversos programas em linguagem de máquina em um único arquivo executável. Também estão incluídos nessas categorias os *depuradores* (*debuggers*), programas que apresentam o estado de um outro programa à medida que este é executado, de modo a permitir que programadores acompanhem o progresso de um programa e encontrem erros.

Os primeiros computadores de programas armazenados em memória eram programados diretamente em *linguagem de máquina*, a representação numérica das instruções utilizadas internamente pelo processador. Para escrever um programa, o programador determinava a seqüência de instruções de máquinas necessárias para gerar o resultado correto e dava entrada nos números que representavam no computador estas instruções. Este era um processo que consumia muito tempo e resultava em uma grande quantidade de erros de programação.

O primeiro passo para simplificar o desenvolvimento de programas veio quando foram desenvolvidos os montadores, permitindo que os programadores codificassem em *linguagem de montagem*. Na linguagem de montagem, cada instrução de máquina tem uma representação em texto (como ADD, SUB ou LOAD) que representa o que ela faz e os programas eram escritos utilizando estas instruções. Uma vez que o programa tivesse escrito, o programador executava o montador para converter o programa em linguagem de montagem em um programa equivalente em linguagem de máquina, o qual podia ser executado no computador. A Fig. 3-2 mostra um exemplo de uma instrução em linguagem de montagem e a instrução em linguagem de máquina gerada a partir dela.

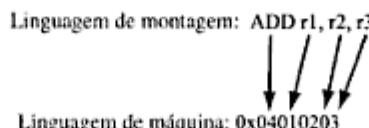


Fig. 3-2 Linguagem de montagem.

Utilizar a linguagem de montagem tornou a tarefa de programação muito mais fácil, ao permitir que os programadores utilizassem um formato de instrução que era mais fácil de ser entendida pelos humanos. A programação ainda era extremamente tediosa porque para executar operações um pouco mais complexas era necessário utilizar várias instruções; além disso, as instruções disponíveis para os programadores diferiam de máquina para máquina. Se um programador quisesse executar um programa em um tipo diferente de computador, o programa tinha que ser completamente reescrito na nova linguagem de montagem desse computador.

As linguagens de alto nível, como FORTRAN, COBOL e C, foram desenvolvidas para resolver estes problemas. Uma instrução em linguagem de alto nível pode especificar muito mais trabalho do que uma instrução em linguagem de montagem. Os estudos têm mostrado que a média do número de instruções escritas e depuradas por dia por um programador é relativamente independente da linguagem utilizada. Uma vez que as linguagens de alto nível permitem que os programas sejam escritos em muito menos instruções do que a linguagem de montagem, o tempo para implementar um programa em linguagem de alto nível é tipicamente muito menor do que o tempo para implementar um programa em linguagem de montagem.

Uma outra vantagem de escrever programas em linguagens de alto nível é que elas são mais portáveis do que programas escritos em linguagens de montagem ou de máquina. Programas escritos em linguagens de alto nível podem ser convertidos para uso em diferentes tipos de computadores pela recompilação do programa, utilizando-se o compilador adequado ao novo computador. Em contraste, programas em linguagem de montagem precisam ser completamente reescritos para novo sistema, o que leva muito mais tempo.

O problema com as linguagens de alto nível é que os computadores não podem executar diretamente instruções em linguagem de alto nível. Assim, um programa chamado *compilador* é utilizado para converter o programa em seu equivalente em linguagem de montagem, que é, então, convertida em linguagem de máquina pelo montador. A Fig. 3-3 ilustra o processo de desenvolvimento e execução de um programa em linguagem de alto nível.

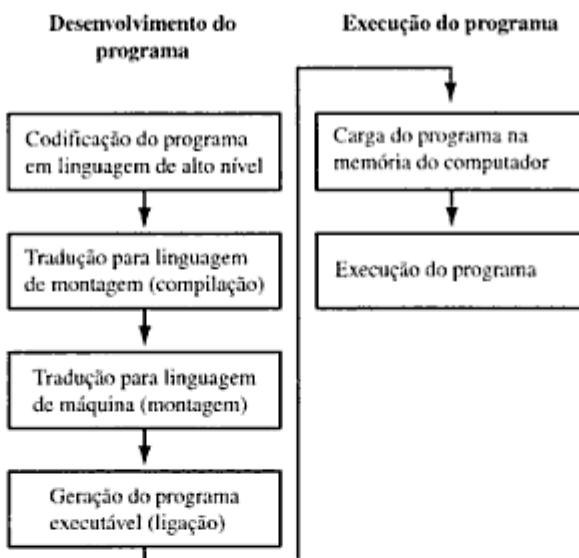


Fig. 3-3 Desenvolvimento de um programa.

Uma alternativa para a compilação de programas é utilizar um *interpretador* para executar a versão do programa em linguagem de alto nível. Os interpretadores são programas que tomam programas em linguagem de alto nível como entradas e executam os passos definidos para cada instrução no programa em linguagem de alto nível, produzindo o mesmo resultado que compilar o programa e então executar a versão compilada. Programas interpretados tendem a ser muito mais lentos do que programas compilados, porque o interpretador tem que examinar cada instrução no programa fonte, à medida que ela ocorre, e então desviar para a rotina que executa a instrução. De muitas maneiras, isto é semelhante à tarefa do compilador de determinar a seqüência de instruções em linguagem de montagem que implementa uma dada instrução em linguagem de alto nível, exceto que o interpretador precisa reinterpretar cada instrução em linguagem de alto nível, cada vez que ela é executada. Se um programa contém um laço que é executado 10.000 vezes, o interpretador precisa interpretar o laço 10.000 vezes, mas o compilador só precisa compilá-lo uma vez.

Dadas as desvantagens de velocidade, os interpretadores são muito menos comuns do que os programas compiladores. Os interpretadores são utilizados principalmente em casos nos quais é importante ser capaz de executar um programa em vários tipos de computadores diferentes, sem a recompilação. Neste caso, utilizar um interpretador permite que cada tipo de computador execute diretamente a versão em linguagem de alto nível do programa.

Compiladores e montadores têm tarefas muito diferentes. Em geral, existe um mapeamento um-para-um entre as instruções em linguagens de montagem e as de máquina, de modo que tudo o que o montador precisa fazer é converter cada instrução de um formato para o outro. Por outro lado, um compilador tem que determinar uma sequência de instruções em linguagem de montagem que implemente as instruções de um programa em linguagem de alto nível, tão eficientemente quanto possível. Por causa disto, o tempo de execução de um programa escrito em uma linguagem de alto nível depende, em grande parte, de quão bom o compilador é, uma vez que o tempo de execução de um programa depende, exclusivamente, da quantidade de instruções em linguagem de máquina executadas.

3.4 SISTEMAS OPERACIONAIS

Em estações de trabalho, PCs e mainframes, o *sistema operacional* é responsável pela administração dos recursos físicos do sistema, pela carga e execução dos programas e pela interface com os usuários. *Sistemas dedicados* – computadores projetados para uma tarefa específica, como controlar um dispositivo – frequentemente não têm um sistema operacional porque eles executam apenas um programa. O sistema operacional é simplesmente um outro programa que conhece tudo sobre o *hardware* no computador, com uma exceção – ele é executado em modo *privilegiado* (ou supervisor), o que permite que ele tenha acesso aos recursos físicos que os programas de usuário não podem controlar, dando a ele a capacidade de iniciar ou interromper a execução de programas do usuário.

Multiprogramação

A maioria dos sistemas de computador suporta a *multiprogramação* (também chamada execução multitarefa), uma técnica que permite que o sistema apresente a ilusão de que vários programas estão sendo executados simultaneamente no computador, mesmo que o sistema possa ter apenas um processador. Em um sistema multiprogramado, os programas de usuário não precisam saber quais outros programas estão sendo executados no sistema ao mesmo tempo em que eles estão sendo executados, ou mesmo quantos outros programas existem. O sistema operacional e o *hardware* fornecem *proteção* para os programas, evitando que um programa tenha acesso aos dados de outro, a menos que eles declarem, explicitamente, a intenção de compartilhar dados. Muitos computadores multiprogramados também são *multusuário* e permitem que mais de uma pessoa esteja utilizando o computador ao mesmo tempo. Sistemas multusuário exigem que o sistema operacional não apenas impeça os programas de acessar os dados um do outro, mas evite que os usuários acessem dados que são privativos a outros usuários.

Um sistema operacional multiprogramado dá a ilusão de que vários programas estão sendo executados simultaneamente, ao comutar muito rapidamente entre programas, como ilustrado na Fig. 3-4. A cada programa é permitida a execução durante uma quantidade fixa de tempo, conhecida como *fatia de tempo*. Quando a fatia de tempo de um programa termina, o sistema operacional interrompe-o ou remove-o do processador, carregando um outro programa no processador. Este processo é conhecido como *comutação de contexto*. Para fazer uma comutação de contexto, o sistema operacional copia o conteúdo dos registradores de máquina do programa que está sendo executado em um dado momento (algumas vezes chamado de *contexto* do programa) para a memória e, então, copia os valores previamente armazenados, associados a outro programa, da memória para esses registradores. Os programas não podem dizer se foi executada uma comutação de contexto – para eles, parece que executam continuamente no processador.

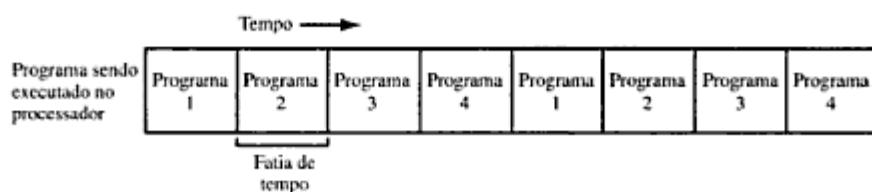


Fig. 3-4 Sistema multiprogramado.

Muitos computadores realizam 60 comutações de contexto por segundo, fazendo com que as fatias de tempo sejam 1/60 de segundo. Alguns sistemas mais modernos executam comutações de contexto com uma frequência maior, o que pode causar problemas a programas que utilizam essa base de tempo (1/60 s) para cronometrar eventos ou estabelecer desempenho.

Ao fazer comutações de contexto 60 ou mais vezes por segundo, um computador pode dar, a cada programa, uma oportunidade de ser executado com frequência suficiente para que o sistema forneça a ilusão que um certo número de programas esteja sendo executado simultaneamente. Evidentemente, à medida que o número de programas no sistema aumenta, esta ilusão diminui – se o sistema estiver executando 120 programas, cada programa pode obter uma fatia de tempo apenas uma vez a cada 2 segundos, o que é um atraso perceptível aos usuários do sistema. A multiprogramação também pode aumentar o tempo de execução de aplicações, porque os recursos do sistema são compartilhados entre todos os programas que estão sendo executados nele.

Proteção

Um dos principais requisitos de um sistema operacional multiprogramado é que ele forneça *proteção* para os programas que estão sendo executados no computador. Essencialmente, isto significa que o resultado de um programa que esteja sendo executado em um computador multiprogramado deve ser o mesmo como se esse programa estivesse sendo o único programa a ser executado no computador. Os programas não devem acessar os dados de outros programas e devem estar seguros de que os seus dados não serão modificados por outros programas. De modo similar, os programas não devem interferir com a utilização que cada um deles faz do subsistema de E/S.

Fornecer proteção em um sistema multiprogramado ou multiusuário exige que o sistema operacional controle os recursos físicos do computador, incluindo o processador, a memória e os dispositivos de E/S. De outro modo, programas de usuário poderiam acessar qualquer parte da memória ou de dispositivos de armazenamento no computador, obtendo acesso a dados que pertencem a outros programas ou usuários. Isto também permite que o sistema operacional evite que mais de um programa acesse um dispositivo de E/S, como uma impressora, ao mesmo tempo.

Uma técnica que os sistemas operacionais utilizam para proteger os dados de um programa é a *memória virtual*, a qual é descrita em detalhes no Capítulo 10. Resumidamente, a memória virtual permite que cada programa opere como se ele fosse o único sendo executado no computador, ao traduzir os endereços de memória aos quais o programa faz referência para endereços físicos realmente utilizados pelo sistema de memória. Desde que o sistema de memória virtual garanta que os endereços de dois programas não sejam traduzidos para o mesmo endereço físico de memória, os programas podem ser escritos como se eles fossem o único programa em execução na máquina, uma vez que nenhuma referência de memória, de nenhum programa, fará acesso aos dados de outro programa.

Modo Privilegiado

Para garantir que o sistema operacional seja o único programa que controle os recursos físicos do sistema, ele é executado em *modo privilegiado*, enquanto que os programas de usuário são executados em *modo de usuário* (algumas vezes chamado de modo não privilegiado). Certas tarefas, como acessar um dispositivo de E/S, executar comutações de contexto ou executar alocação de memória, exigem que o programa esteja em modo privilegiado. Se um programa em modo de usuário tentar executar uma dessas tarefas, o *hardware* evita que ele faça isso e sinaliza a ocorrência de um erro. Quando programas em modo de usuário necessitam executar uma operação que exija modo privilegiado, eles enviam uma solicitação ao sistema operacional, conhecida como *chamada de sistema*, que solicita que faça a operação por eles. Se a operação é algo que o programa de usuário tem permissão para fazer, o sistema operacional executa a operação e retorna o resultado para o usuário. Caso contrário, ele sinaliza a ocorrência de um erro.

Por controlar os recursos físicos do computador, o sistema operacional também é responsável por interfacear o usuário com o sistema. Quando um usuário pressiona uma tecla ou envia algum outro tipo de entrada para o computador, o sistema operacional é o responsável por determinar qual programa deve receber a entrada e por enviar o valor da entrada para o programa. Além disso, quando um programa quer apresentar alguma informação para o usuário, como escrever um caractere no monitor, ele executa uma chamada de sistema para solicitar ao sistema operacional a apresentação dos dados.

3.5 ORGANIZAÇÃO DOS COMPUTADORES

A Fig. 3-1 apresentou um diagrama em blocos em alto nível de um sistema de computador típico. Nesta seção, apresentamos uma breve introdução a cada um dos subsistemas principais: processador, memória e E/S. O objetivo desta discussão é dar ao leitor conhecimento de alto nível suficiente sobre cada subsistema, de modo a prepará-lo para os capítulos seguintes que discutem, detalhadamente, cada um desses subsistemas.

O Processador

O processador é responsável pela execução real das instruções que compõe os programas e o sistema operacional. Como ilustrado na Fig. 3-5, os processadores são compostos de vários blocos: unidades de execução, banco de registradores e lógica de controle. As unidades de execução contêm o *hardware* que executa as instruções. Isto inclui o *hardware* que busca e decodifica as instruções, bem como as unidades lógico-aritméticas (ULAs) que executam os cálculos. Muitos processadores contêm unidades de execução diferentes para cálculos com inteiros e em ponto flutuante, porque é necessário um *hardware* muito diferente para tratar estes dois tipos de dados. Além disto, como veremos no Capítulo 7, os processadores modernos, para melhorar o desempenho, freqüentemente utilizam várias unidades de execução para executar instruções em paralelo.

O *banco de registradores* é uma pequena área de armazenamento para os dados que o processador está usando. Os valores armazenados no banco de registradores podem ser acessados mais rapidamente do que os dados armazenados no sistema de memória, sendo que os bancos de registradores geralmente suportam vários acessos simultâneos. Isto permite que uma operação, como uma adição, leia todas as suas entradas do banco de registradores ao mesmo tempo, ao invés de ter que lê-las uma por vez. Como veremos no Capítulo 4, diferentes processadores acessam e usam o seus bancos de registradores de muitos modos diferentes, mas virtualmente todos processadores têm um banco de registradores de algum tipo.

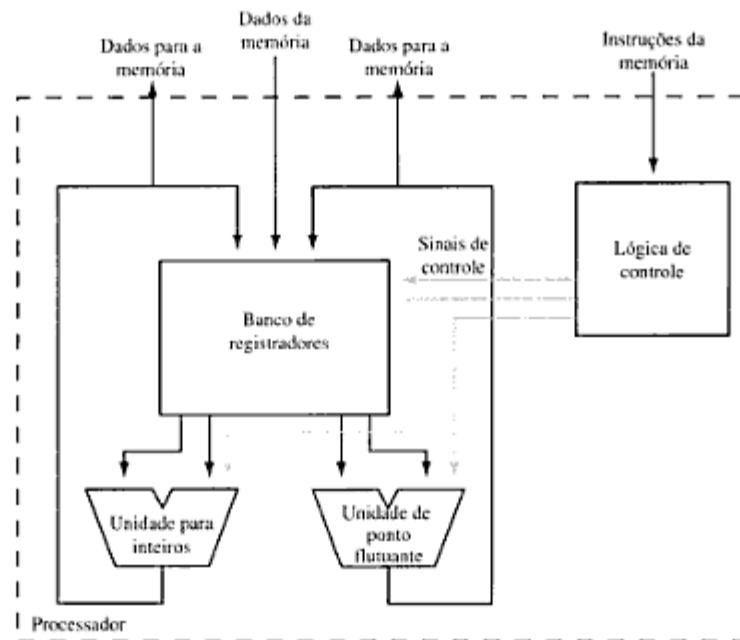


Fig. 3-5 Diagrama em blocos de um processador.

Como pode ser deduzido pelo seu nome, a lógica de controle controla o resto do processador, determinando quando as instruções podem ser executadas e quais operações são necessárias para executar cada instrução. Nos primeiros processadores, ela era uma parte muito pequena do *hardware* do processador, quando comparada com as ULAs e o banco de registradores, mas a quantidade de lógica de controle necessária cresceu significativamente à medida que os processadores tornaram-se mais complexos, fazendo com que seja uma das partes mais difíceis de ser projetada em um processador.

O Sistema de Memória

O sistema de memória age como um receptáculo de armazenamento para os dados e programas utilizados pelo computador. A maioria dos computadores tem dois tipos de memória: *memória apenas de leitura* (*Read Only Memory* – ROM) e *memória de acesso aleatório* (*Random Access Memory* – RAM). Como o seu nome sugere, o conteúdo de uma memória apenas de leitura não pode ser modificado pelo computador, mas pode ser lido. Em geral, a ROM é utilizada para manter um programa que é executado automaticamente pelo computador cada vez que ele é ligado ou reinicializado. Este programa é chamado de *bootstrap* e instrui o computador a carregar o sistema operacional do seu disco rígido ou de outro dispositivo de E/S. O nome deste programa vem da idéia de que o computador está “erguendo-se por sua própria conta”, ao executar um programa que diz a ele como carregar o seu próprio sistema operacional.

Por outro lado, a memória de acesso aleatório tanto pode ser lida como escrita, e é utilizada para manter os programas, o sistema operacional e os dados exigidos pelo computador. A RAM é geralmente volátil, o que significa que ela não mantém os dados armazenados nela quando a alimentação do computador é desligada. Quaisquer dados que necessitem permanecer armazenados, enquanto o computador estiver desligado, precisam ser escritos em um dispositivo de armazenamento permanente, como um disco rígido.

A memória (tanto a RAM quanto a ROM) é dividida em um conjunto de posições de armazenamento, cada uma das quais pode manter 1 *byte* (8 *bits*) de dados. As posições de armazenamento são numeradas, e o número de uma posição de armazenamento (chamada de *endereço*) é utilizado para dizer ao sistema de memória a quais posições o processador quer fazer referência. Uma das características importantes de um sistema de computador é a largura dos endereços que ele utiliza, o que limita a quantidade de memória que o computador pode endereçar. A maioria dos computadores atuais utilizam endereços de 32 ou de 64 *bits*, permitindo que eles façam o acesso a 2^{32} ou 2^{64} *bytes* de memória.

Até o Capítulo 9, estaremos utilizando um modelo simples de memória de acesso aleatório, no qual todas as operações de memória duram o mesmo tempo. O nosso sistema de memória suportará duas operações: carga e armazenamento. As operações de armazenamento ocupam dois operandos: um valor a ser armazenado e o endereço onde o valor deve ser armazenado. Elas colocam o valor especificado na posição de memória especificada pelo endereço. Operações de carga têm um operando que especifica um endereço e retornam o conteúdo dessa posição de memória para o seu destino.

Utilizando este modelo, pode-se imaginar a memória como funcionando de modo semelhante a uma grande folha de papel pautado, onde cada linha na página representa um local de armazenamento para um *byte*. Para escrever (armazenar) um valor na memória, conta-se de cima para baixo na página até que se atinja a linha especificada pelo endereço, apaga-se o valor escrito naquela linha e escreve-se o novo valor. Para ler (carregar) um valor, conta-se de cima para baixo na página até que se atinja a linha especificada pelo endereço e lê-se o valor escrito naquela linha. A maioria dos computadores permite que mais de um *byte* de memória seja armazenado ou carregado por vez. Geralmente, uma operação de carga ou armazenamento opera sobre uma quantidade de dados igual à largura de *bits* do sistema, e o endereço enviado ao sistema de memória especifica a posição do *byte* de dados de endereço mais baixo a ser carregado ou armazenado. Por exemplo, um sistema de 32 *bits* carrega ou armazena 32 *bits* (4 *bytes*) de dados em cada operação, nos 4 *bytes* que começam com o endereço da operação, de modo que uma carga a partir da localização 424 retornaria uma quantidade de 32 *bits* contendo os *bytes* das localizações 424, 425, 426 e 427. Para simplificar o projeto do sistema de memória, alguns computadores exigem que as cargas e armazenamentos sejam “alinhados”, significando que o endereço de uma referência de memória precisa ser um múltiplo do tamanho do dado que está sendo carregado ou armazenado, de modo que uma carga de 4 *bytes* precisa ter um endereço que seja um múltiplo de 4, um armazenamento de 8 *bytes* precisa ter um endereço que seja um múltiplo de 8, e assim por diante. Outros sistemas permitem cargas e armazenamentos desalinhados, mas demoram mais tempo para completar tais operações do que com cargas alinhadas.

Uma questão adicional com cargas e armazenamentos de vários *bytes* é a ordem na qual eles são escritos na memória. Há dois tipos de esquemas de ordenação diferentes que são utilizados nos computadores modernos: *little endian* e *big endian**. No sistema *little endian*, o *byte* menos significativo (o valor menor) de uma palavra é escrito no *byte* de endereço mais baixo, e os outros *bytes* são escritos na ordem crescente de significância. No sistema *big endian*, a ordem é inversa, com o *byte* mais significativo sendo escrito no *byte* de memória com o endereço mais baixo. Os outros *bytes* são escritos em ordem decrescente de significância. A Fig. 3-6 mostra um exemplo de como os sistemas *little endian* e *big endian* escreveriam uma palavra de dados de 32 bits (4 bytes) no endereço 0x1000.

Em geral, os programadores não precisam saber a ordem dos *bytes* no sistema com o qual eles estão trabalhando, exceto quando a mesma posição de memória é acessada utilizando-se cargas e armazenamentos de comprimentos diferentes. Por exemplo, se um armazenamento de um *byte* igual a 0, na localização 0x1000, fosse executado nos sistemas apresentados na Fig. 3-6, uma carga subsequente de 32 bits a partir de 0x1000, retornaria 0x90abcd00 no sistema *little endian* e 0x00abedef no sistema *big endian*. No entanto, a ordem dos *bytes* freqüentemente é um problema quando se transmite dados entre sistemas de computadores diferentes, pois eles interpretarão a mesma seqüência de *bytes* como palavras diferentes de dados nos sistemas *little* ou *big endian*. Para suplantar este problema, os dados precisam ser processados para convertê-los para a ordem dos *bytes* do computador que vai lê-los.

O projeto de sistemas de memória tem um impacto enorme sobre o desempenho de sistemas de computadores e é freqüentemente o fator limitante para a rapidez de execução de uma aplicação. Tanto a largura de banda (quantos dados podem ser carregados ou armazenados em um dado período de tempo) quanto a latência (quanto tempo uma operação de memória em especial demora para ser completada) são importantes para o desempenho da aplicação. Outras questões importantes no projeto de sistemas de memória incluem a proteção (evitar que diferentes programas acessem dados uns dos outros) e como o sistema de memória interage com o sistema de E/S.

	0x1000	0x1001	0x1002	0x1003
<i>Little endian</i>	ef	cd	ab	90
Palavra = 0x90abcdef Endereço = 0x1000				
<i>Big endian</i>	90	ab	cd	ef

Fig. 3-6 Little endian versus big endian.

O Subsistema de E/S

O subsistema de E/S contém os dispositivos que o computador utiliza para comunicar-se com o mundo exterior e para armazenar dados, incluindo discos rígidos, monitores de vídeo, impressoras e acionadores de fita. Os sistemas de E/S são cobertos em detalhes no Capítulo 11. Como indicado na Fig. 3-1, tais dispositivos comunicam-se com o processador por meio de um barramento de E/S, o qual é separado do barramento de memória que o processador utiliza para comunicar-se com o sistema de memória.

Utilizar um barramento de E/S permite que um computador faça a interface com uma ampla gama de dispositivos de E/S, sem ter que implementar uma interface específica para cada um. Esse tipo de barramento também pode suportar um número variável de dispositivos, permitindo que os usuários acrescentem outros posteriormente. Os dispositivos de E/S podem ser projetados para fazer a interface com o barramento, permitindo que eles sejam compatíveis com qualquer computador que utilize o mesmo tipo de barramento de E/S. Por exemplo, a maioria dos PCs e muitas estações de trabalho utilizam o barramento padrão PCI. Todos esses sistemas podem interfacear com dispositivos projetados de acordo com o padrão PCI. Nesse caso, é necessário apenas um *acionador de dispositivo (device driver)* – um programa que permite que o sistema operacional controle o dispositivo de E/S. O lado negativo de utilizar um barramento de E/S para fazer a interface com dispositivos de E/S é que todos esses dispositivos compartilham esse barramento o que o torna mais lento do que se houvesse conexões dedicadas entre o processador e um dispositivo de E/S, pois são projetados para compatibilidade e flexibilidade máximas.

* N de R. T. O termo *endian* tem sua origem no livro *As Viagens de Gulliver* e refere-se à questão de qual lado os ovos devem ser quebrados.

Sistemas de E/S era um dos aspectos menos estudados em arquitetura de computadores, ainda que o seu desempenho fosse fundamental para muitas aplicações. Nos últimos anos, o desempenho desses sistemas tornou-se ainda mais crucial com a escalada de importância dos sistemas de bancos de dados e processamento de transações, pois dependem pesadamente dos subsistemas de E/S dos computadores nos quais são executados. Isto fez com que sistemas de E/S se tornassem uma área de pesquisa ativa, especialmente considerando-se os altos investimentos que as empresas realizam para aperfeiçoar e melhorar o desempenho dos sistemas de bancos de dados e processamento de transações.

3.6 RESUMO

O objetivo deste capítulo foi estabelecer as bases para os próximos capítulos, ao apresentar uma introdução para os principais blocos construtivos do *hardware* de sistemas de computadores e os componentes de *software* que interagem com eles. Discutimos como os sistemas de computadores são divididos em processadores, sistemas de memória e E/S, e fornecemos uma introdução para cada um destes tópicos. Este capítulo também cobriu os diferentes níveis nos quais os programas são implementados, variando de linguagens de máquina, que os processadores executam, às linguagens de alto nível que os usuários tipicamente utilizam para programar os computadores.

Os próximos capítulos estarão concentrados na arquitetura de processadores, a partir dos modelos de programação até as técnicas para melhorar o desempenho, como *pipelining* e paralelismo ao nível da instrução. Após isso, examinaremos o sistema de memória, discutindo memória virtual, hierarquias de memória e memórias *cache*. Finalmente, concluiremos com uma discussão dos sistemas de E/S e uma introdução ao multiprocessamento.

Problemas Resolvidos

Computadores com Memória de Programa

- 3.1** O programa para emacs (um editor de textos UNIX) tem 2.878.448 bytes de tamanho no computador que está sendo utilizado para escrever este livro. Se um ser humano pudesse dar entrada em um byte do programa por segundo, por meio de interruptores utilizados para programar um computador sem programas armazenados em memória (o que parece ser otimista), quanto tempo demoraria para que o programa emacs estivesse pronto para ser executado? Se o humano tivesse uma taxa de erros de 0,001% na entrada de dados, quantos erros seriam feitos quando fosse feita a entrada do programa?

Solução

A 1 byte/s, o programa demandaria 2.878.448 s, o que é, aproximadamente, 47.974 minutos, ou 800 horas, ou 33,3 dias. Obviamente, um editor de textos não seria de muita utilidade se ele demorasse um mês para colocá-lo em execução.

Uma taxa de erros de 0,001% é um erro em cada 100.000 bytes, o que seria extremamente bom para um ser humano. Mesmo assim, ele cometaria, aproximadamente, 29 erros ao dar entrada no programa, cada um dos quais teria que ser depurado e corrigido antes que o programa pudesse ser executado corretamente.

Linguagem de Máquina versus Linguagem de Montagem (Assembly)

- 3.2** a. Qual é diferença entre linguagem de máquina e linguagem de montagem?
b. Por que a linguagem de montagem é considerada mais fácil para seres humanos programarem do que a linguagem de máquina?

Solução

a. Instruções em linguagem de máquina são padrões de bits utilizados para representar as operações dentro do computador. A linguagem de montagem é uma versão da linguagem de máquina, mais legível por seres humanos, na qual cada instrução é representada por uma cadeia de texto que descreve o que a instrução faz.

b. Na programação em linguagem de montagem, o montador, não o ser humano, é o responsável pela conversão das instruções em linguagem de montagem para a linguagem de máquina. Os seres humanos geralmente acham mais fácil entender cadeias de texto que representem instruções de linguagem de montagem do que os números que codificam as instruções em linguagem de máquina. Além disso, confiar no montador para traduzir as instruções em linguagem de montagem para instruções em linguagem de máquina elimina a possibilidade de erros na geração da representação de cada instrução em linguagem de máquina.

Programas Automodificáveis

- 3.3 Por que os programas automodificáveis são menos comuns hoje em dia do que eles eram nos primeiros computadores?

Solução

Há duas razões principais. A primeira é que código automodificável é mais difícil de depurar do que código que não seja automodificável, porque o programa que é executado é diferente daquele que foi escrito. À medida que os computadores tornaram-se mais rápidos, as vantagens de desempenho do código automodificável tornaram-se menos significativas do que a dificuldade crescente de depuração.

Em segundo lugar, os aperfeiçoamentos nos projetos de sistemas de memória reduziram as melhorias de desempenho que podiam ser obtidas através de código automodificável.

Compiladores versus Montadores

- 3.4 Explique brevemente por que a qualidade de um compilador tem mais impacto sobre o tempo de execução de um programa desenvolvido utilizando o compilador do que a qualidade que um montador tem sobre programas desenvolvidos utilizando o montador.

Solução

Em geral, existe um mapeamento um-para-um entre as instruções em linguagem de montagem e as instruções em linguagem de máquina. O trabalho de um montador é traduzir cada instrução em linguagem de montagem para a sua representação em linguagem de máquina. Assumindo que o montador faça esta tradução corretamente, as instruções no programa resultante em linguagem de máquina são exatamente as mesmas que aquelas do programa fonte em linguagem de montagem, apenas com uma codificação diferente. Como o montador não modifica o conjunto de instruções em um programa, ele não tem impacto sobre o tempo de execução sobre o mesmo.

Em contraste, o trabalho de um compilador é determinar uma seqüência de instruções em linguagem de montagem que execute a tarefa especificada por um programa em linguagem de alto nível. Uma vez que o compilador está, ele mesmo, criando a seqüência de instruções em linguagem de montagem para o programa, a qualidade do compilador tem um grande impacto sobre quanto tempo o programa resultante demora para ser executado. Maus compiladores criam programas que fazem muito trabalho desnecessário e, portanto, são executados vagarosamente, enquanto que bons compiladores eliminam este trabalho desnecessário, o que oferece um desempenho melhor.

Multiprogramação (I)

- 3.5 Como um sistema multiprogramado apresenta a ilusão de que vários programas estão sendo executados simultaneamente na máquina? Quais os fatores que fazem com que esta ilusão seja prejudicada?

Solução

Sistemas multiprogramados percorrem freqüentemente todos os programas que estão sendo executados neles – 60 ou mais vezes por segundo. Desde que o número de programas que está sendo executado no sistema seja relativamente pequeno, cada programa terá uma oportunidade de ser executado com freqüência suficiente para que o sistema dê a impressão de que está executando todos os programas ao mesmo tempo, no sentido de que eles parecem estar progredindo simultaneamente.

Se o número de programas que está sendo executado no sistema torna-se muito grande – por exemplo, aproximando-se do número de comutações de contexto executadas por segundo – os usuários serão capazes de perceber os intervalos de tempo em que um dado programa está progredindo e essa ilusão terá sido prejudicada. Mesmo com um pequeno número de programas sendo executado na máquina, freqüentemente é possível saber quando a máquina está compartilhando o seu processador entre os programas, porque a taxa de progresso de cada programa será menor do que se ele tivesse a máquina só para si.

Multiprogramação (II)

- 3.6 Se um computador de 800 MHz faz 60 comutações de contexto por segundo, quantos ciclos existem em cada fatia de tempo?

Solução

$$800 \text{ MHz} = 800.000.000 \text{ de ciclos/s. } 800.000.000 / 60 = 13.333.333 \text{ ciclos/fatia de tempo}$$

Multiprogramação (III)

- 3.7 Suponha que um dado computador faça 60 comutações de contexto por segundo. Se um ser humano interagindo com o computador perceber, em qualquer instante, que uma dada operação demora mais do que 0,5 s para responder a uma entrada, quantos programas podem estar sendo executados no computador sem que o usuário perceba atrasos? (Assuma que o sistema faça comutações entre programas de modo seqüencial e circular, que um programa possa sempre responder a uma entrada durante a primeira fatia de tempo na qual ele é executado após a ocorrência da entrada e que os programas sempre são executados durante uma fatia de tempo completa quando eles são selecionados para execução.) Quantos programas podem estar em execução antes que um usuário perceba o atraso em pelo menos a metade do tempo?

Solução

Se o computador faz 60 comutações de contexto por segundo, há 30 comutações de contexto em 0,5 segundos. Portanto, o computador pode executar até 30 programas e garantir que cada programa obtenha uma fatia de tempo dentro de 0,5 segundos, a partir de qualquer entrada de usuário. Uma vez que está garantido que cada programa pode responder às entradas do usuário durante a primeira fatia de tempo depois que a entrada ocorre, isto garantirá que o usuário nunca notará um atraso.

Para que um usuário sinta um atraso apenas metade do tempo, tem que haver uma probabilidade de 50% de que o programa ao qual uma entrada foi direcionada obtenha uma fatia de tempo dentro dos 0,5 segundos seguintes à ocorrência da entrada. Uma vez que os programas são executados de modo seqüencial e circular, isto significa que metade dos programas tem que ser executados dentro dos 0,5 segundos seguintes à entrada. Uma vez que 30 programas são executados em 0,5 segundos, pode haver até 60 programas sendo executados no computador antes que um usuário note um atraso em mais da metade do tempo.

Sistemas Operacionais (I)

- 3.8 Cite dois exemplos de problemas que poderiam ocorrer se um computador permitisse que programas de usuário fizessem acesso diretamente a dispositivos de E/S, ao invés de exigir que eles passassem pelo sistema operacional.

Solução

Os dois exemplos descritos neste capítulo são:

1. Violações de proteção – se programas de usuário pudessem acessar diretamente dispositivos de armazenamento de dados, então eles poderiam ler ou escrever dados que pertencem a outros programas e aos quais eles não deveriam ter acesso.
2. Violação de acesso seqüencial – um programa pode não ter terminado de utilizar um dispositivo de E/S quando a sua fatia de tempo termina. Se um outro programa tentasse utilizar o mesmo dispositivo antes que o programa tivesse obtido uma outra fatia de tempo, as operações de E/S dos dois programas poderiam ficar intercaladas. Isto poderia resultar em erros incomuns, como a saída dos dois programas ser intercalada num monitor do computador.

Sistemas Operacionais (II)

- 3.9 Por que é necessário que um computador forneça um modo privilegiado e um modo de usuário para os programas?

Solução

O modo privilegiado é o mecanismo que os computadores utilizam para evitar que programas de usuário executem tarefas que são limitadas ao sistema operacional. Sem um modo de execução privilegiado, o *hardware* não seria capaz de saber se um programa que está tentando fazer uma operação é um programa de usuário ou o sistema operacional, impedindo que ele saiba se pode ou não permitir que o programa execute a operação. Alguns sistemas fornecem mais do que dois níveis de execução para permitir que diferentes programas tenham acesso a diferentes recursos, mas dois níveis são suficientes para a maioria dos sistemas operacionais.

Banco de Registradores

- 3.10 Por que aumentar a quantidade de dados que pode ser armazenada no banco de registradores de um processador geralmente melhora o seu desempenho?

Capítulo 4

Modelos de Programação

4.1 OBJETIVOS

Este capítulo descreve os modelos de programação utilizados em dois tipos de processadores: arquiteturas baseadas em pilha e arquiteturas baseadas em registradores de uso geral. Começamos com uma discussão sobre os tipos de operações fornecidas pela maioria dos processadores. Em seguida, uma descrição das arquiteturas baseadas em pilha e em registradores de uso geral. Cada descrição de uma arquitetura inclui um conjunto de instruções de exemplo, para aquele tipo de processador, os quais irão formar a base para os exemplos e os exercícios por todo o resto deste livro. O capítulo conclui com uma comparação entre os dois modelos de programação e uma discussão de como as pilhas são utilizadas para implementar chamadas de procedimento, mesmo nas arquiteturas baseadas em registradores de uso geral.

Após completar este capítulo, você deverá:

1. Estar familiarizado com os tipos diferentes de operações fornecidas na maioria dos processadores.
2. Compreender as arquiteturas baseadas em pilha e ser capaz de escrever pequenos programas em linguagem de montagem, com os conjuntos de instruções descritos neste capítulo.
3. Compreender as arquiteturas baseadas em registradores de uso geral e ser capaz de escrever pequenos programas na linguagem de montagem para estas arquiteturas.
4. Ser capaz de comparar as arquiteturas baseadas em registradores de uso geral e em pilha, e descrever as situações nas quais cada estilo seria mais adequado.
5. Compreender as chamadas de procedimento e como elas são implementadas.

4.2 INTRODUÇÃO

No último capítulo, descrevemos os programas como um conjunto de instruções de máquina, sem fornecer muitos detalhes a respeito do que são instruções e como elas são implementadas. Neste capítulo, cobriremos dois modelos de programação para processadores: as arquiteturas baseadas em pilha e as arquiteturas baseadas em registradores de uso geral (RUG). Um *modelo de programação* de um processador define como as instruções acessam os seus operandos e como as instruções são descritas na linguagem de montagem do processador, mas não o conjunto de operações que são fornecidas pelo processador. Como veremos, processadores com diferentes modelos de programação podem fornecer conjuntos muito semelhantes de operações, mas podem exigir abordagens muito diferentes para a programação.

A Fig. 4-1 dá uma visão de alto nível de como as instruções que serão utilizadas nesse capítulo são executadas. Capítulos posteriores darão explicações mais detalhadas sobre a execução das instruções. Primeiro, o processador busca (lê) a instrução na memória. O endereço da próxima instrução a ser executada é armazenado em um registrador especial conhecido como *contador de programa* (CP), que algumas vezes é chamado de *apontador de instruções*, de modo que o processador possa facilmente determinar onde ele deve procurar a próxima instrução na memória.

Uma vez que o sistema de memória tenha entregue a instrução para o processador, este examina a instrução para verificar o que tem que fazer para realizá-la, executa a operação especificada pela instrução e escreve o resultado da instrução em um registrador ou na memória. Então, o processador atualiza o contador de programa que contém o endereço da próxima instrução a ser executada e repete o procedimento.

O restante deste capítulo começa com uma discussão sobre os diferentes tipos de instruções que são fornecidos pela maioria dos processadores. Então, damos uma introdução às arquiteturas baseadas em pilha e apresentamos um conjunto de instruções exemplo para uma arquitetura baseada em pilha. Segue-se uma discussão de arquiteturas baseadas em registradores de uso geral, incluindo um conjunto exemplo de instruções para estas arquiteturas. O capítulo conclui com uma discussão sobre como as pilhas são utilizadas para implementar chamadas de procedimento, tanto na arquitetura baseada em pilha como na baseada em registradores de uso geral.

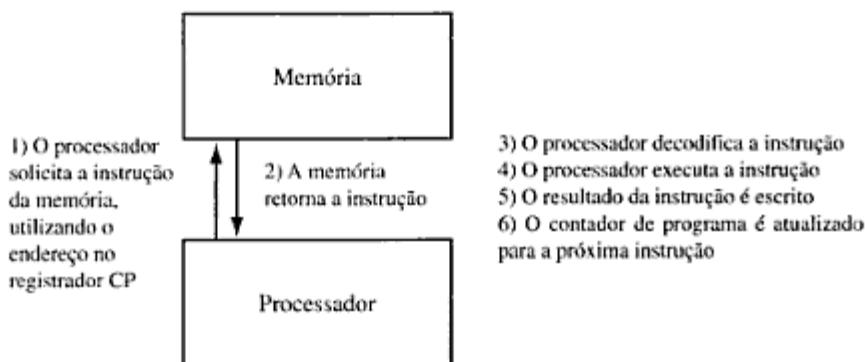


Fig. 4-1 Ciclo básico de execução de instruções.

4.3 TIPOS DE INSTRUÇÕES

Um dos fatores que diferenciam os processadores uns dos outros são os seus conjuntos de instruções – os conjuntos de operações básicas que cada processador fornece. Os primeiros processadores tinham conjuntos de instruções muito diferentes, e o projeto do conjunto de instruções era uma das principais tarefas dos projetistas de computadores. À medida que o campo progrediu, o conjunto de operações fornecido pelos processadores convergiu e agora, praticamente, todos os processadores fornecem conjuntos de instruções muito semelhantes, independentemente de utilizarem modelos de programação baseados em pilha ou em registradores de uso geral. As operações básicas podem ser divididas em quatro categorias: operações de memória, operações aritméticas, comparações e operações de controle (desvios).

Operações Aritméticas

As operações aritméticas executam cálculos básicos, como adições, multiplicações, operações lógicas (E, OU) e cópia de dados. Geralmente, elas utilizam um ou dois dados como entrada e geram uma saída. Em geral, as operações aritméticas leem suas entradas e escrevem suas saídas no banco de registradores, embora algumas arquiteturas CISC (computador com conjunto de instruções complexas) permitam que as operações aritméticas façam referência à memória. As arquiteturas CISC são cobertas com mais detalhes no próximo capítulo.

A Fig. 4-2 mostra o conjunto de operações aritméticas que utilizaremos neste livro e que são representativas das operações aritméticas fornecidas pela maioria dos processadores modernos. A maioria dos processadores fornece um conjunto maior destas operações, freqüentemente tendo diversas instruções que fornecem variações de uma única operação básica. As operações apresentadas aqui foram escolhidas como um compromisso entre a com-

plexidade e ser o mais completo possível, com o objetivo de fornecer um conjunto de instruções rico o suficiente para implementar a maioria dos programas, mas sem sobrecarregar o leitor com complexidade. Note que muitas das instruções tem versões para inteiros e para ponto flutuante. Isto permite que o *hardware* determine se ele deve tratar as entradas das instruções como valores inteiros ou em ponto flutuante e determine qual banco de registradores deve ser utilizado em arquiteturas que tem registradores separados para inteiros e para ponto flutuante.

A maioria das operações apresentadas na Fig. 4-2 é relativamente auto-explicativa, embora duas delas (ASH e LSH) precisem de uma explicação adicional. Estas operações são exemplos de operações de deslocamento (*shift*), operações que mudam a posição dos *bits* de um de seus operandos. As operações de deslocamento tomam os *bits* de seu primeiro operando e os deslocam para a esquerda por um número de posições de *bit* igual ao valor de seu segundo operando (valores negativos no segundo operando indicam que os *bits* são deslocados para a direita). As diferenças entre estas operações repousam sobre quais valores elas inserem nas posições de *bit* que são tornadas vagas pelo deslocamento de um *bit* para fora delas, mas nenhum bit estava disponível para ser deslocado para dentro delas (por exemplo, qual valor vai no *bit* menos significativo de uma palavra que é deslocada uma posição para a esquerda).

As operações de deslocamento lógico (LSH – *logical shift*) são as mais simples das duas operações de deslocamento. Os *bits* que são deslocados para fora da palavra são descartados e são deslocados zeros para dentro das posições de *bit* vazias.

Operação	Função
ADD	Soma os seus dois operandos inteiros
FADD	Soma os seus dois operandos em ponto flutuante
SUB	Subtrai o seu segundo operando do primeiro, em modo inteiro
FSUB	Subtrai o seu segundo operando do primeiro, em ponto flutuante
MUL	Multiplica os seus dois operandos inteiros
FMUL	Multiplica os seus dois operandos em ponto flutuante
DIV	Divide o primeiro operando pelo segundo, em modo inteiro
FDIV	Divide o primeiro operando pelo segundo, em ponto flutuante
MOV	Copia a sua entrada (de qualquer tipo) para a sua saída, as quais podem ser ambas de qualquer tipo
OR	Executa uma operação lógica OU sobre seus dois operandos, os quais podem ser de qualquer tipo
AND	Executa uma operação lógica E sobre seus dois operandos, os quais podem ser de qualquer tipo
NOT	Executa uma negação lógica sobre seu operando, o qual pode ser de qualquer tipo
ASH	Faz o deslocamento (aritmético) do seu primeiro operando pelo número de posições especificado pelo segundo operando
LSH	Faz o deslocamento (lógico) do seu primeiro operando pelo número de posições especificado pelo segundo operando

Fig. 4-2 Operações aritméticas.

Exemplo Em um sistema com palavras de dados de 8 *bits*, qual é o resultado de se fazer uma operação LSH, cuja primeira entrada é 25 e a segunda é 2?

Solução

A representação inteira em 8 *bits* de 25 é 0b 0001 1001. Deslocando para a esquerda duas posições de *bit*, temos 0b0110 01xx, onde "x" indica *bits* vazios. A operação LSH especifica que sejam deslocados zeros para dentro das posições de *bit* vazias, de modo que o resultado final é 0b 0110 0100, que é a representação binária em 8 *bits* de 100 em base 10.

Deve-se observar que deslocar para a esquerda um valor binário inteiro sem sinal, ou positivo, tem o efeito de multiplicá-lo por 2^n , onde n é o número de posições de *bit* pelo qual o valor é deslocado. De modo semelhante, deslocar um valor binário inteiro sem sinal, ou positivo, n posições para a direita, implica em dividi-lo por 2^n , com quaisquer *bits* de resto da divisão sendo descartados. As operações de deslocamento são frequentemente mais rápidas do que multiplicações e divisões, de modo que muitos compiladores e programadores preferem utilizá-las, em vez de operações de multiplicação, quando estão multiplicando ou dividindo por uma potência de 2.

Comparações

Como o seu nome sugere, as operações de comparação comparam dois valores a fim de que o programa possa tomar decisões. Existe uma grande variação, entre os diferentes processadores, no modo como eles tratam o resultado das operações de comparação. Alguns processadores os escrevem em um registrador do banco de registrador. Outros fornecem um registrador especial que mantém o resultado das operações de comparação mais recentes. As arquiteturas que utilizam o registrador especial têm se tornado menos comuns, uma vez que ter um único local para colocar os resultados de comparações torna impossível executar várias comparações em paralelo. A Fig. 4-4 mostra um conjunto comum de operações de comparação. A maioria dos processadores também fornece um conjunto equivalente de operações de comparação para ponto flutuante.

As operações de comparação são utilizadas em conjunto com operações de controle para criar um *desvio condicional* que executa um segmento de um programa ou outro, dependendo do resultado da comparação. Este uso é tão comum que a maioria dos processadores fornece operações de desvio condicional que combinam em uma instrução a comparação e o desvio. Por simplicidade, os conjuntos de instruções apresentados mais adiante neste capítulo para as arquiteturas baseadas em pilha e em registradores de uso geral omitirão as instruções de comparação e fornecerão apenas instruções de desvio, uma vez que os desvios condicionais são o uso mais comum das comparações.

Operação	Função
EQ	Testa dois operandos inteiros para verificar se eles são iguais
NEQ	Testa dois operandos inteiros para verificar se eles são diferentes
GT	Determina se o primeiro operando inteiro é maior do que o segundo
LT	Determina se o primeiro operando inteiro é menor do que o segundo
GEQ	Determina se o primeiro operando inteiro é maior ou igual ao segundo
LEQ	Determina se o primeiro operando inteiro é menor ou igual ao segundo

Fig. 4-4 Operações de comparação.

Operações de Controle

Operações de controle (desvios) afetam o fluxo do programa ao mudar o CP do processador. Quando é executada uma operação que não seja uma operação de controle, o hardware incrementa o contador de programas pelo tamanho da instrução, de modo que ele aponte para a próxima instrução no programa. Por exemplo, em uma arquitetura na qual as instruções têm 32 bits de comprimento, soma-se 4 ao CP após a execução de instrução, pois 32 bits são 4 bytes.

Quando a operação de controle é executada, o contador de programas é ajustado para o valor da entrada¹ da instrução de controle, fazendo com que a execução salte para um ponto diferente do programa. Operações de controle, que são freqüentemente chamadas de desvios, podem ser divididas em duas categorias: *incondicionais* e *condicionais*. Quando são executados, os desvios incondicionais, também chamados de saltos, sempre ajustam o CP para o valor da sua entrada. Desvios condicionais ajustam o CP para que fique igual à sua entrada, se alguma condição, como o resultado de uma comparação, for verdadeira. Dependendo da arquitetura, a comparação pode ser executada como parte da operação de desvio ou pode ter sido executada anteriormente por uma instrução diferente no programa. A Fig. 4-5 mostra um conjunto comum de operações de controle que assumiremos para os nossos processadores.

Normalmente, quando programadores escrevem programas em linguagem de máquina, eles não especificam um valor numérico como endereço de destino da instrução de desvio. Em vez disso, as instruções são rotuladas com valores em texto e as instruções de desvio fazem referência a esses valores. Os compiladores também agem dessa forma, ao traduzirem um programa em uma linguagem de alto nível para o equivalente em linguagem de montagem.

¹ Muitos processadores fornecem diferentes modos de endereçamento para as instruções de desvio, os quais fazem com que a instrução de desvio execute o cálculo e ajuste o CP ao resultado do cálculo, em vez de apenas ajustar o CP ao valor da sua entrada. Os modos de endereçamento são descritos em mais detalhes no próximo capítulo.

Operação	Função
BR ou JMP	Ajusta o CP para o valor operando de entrada, de forma que a próxima instrução a ser executada seja a instrução associada ao endereço correspondente a esse valor
BEQ	Ajusta o CP para o valor do primeiro operando, se os seus dois outros operandos forem iguais
BNE	Ajusta o CP para o valor do primeiro operando, se os seus dois outros operandos forem diferentes
BLT	Ajusta o CP para o valor do primeiro operando, se o segundo operando for menor do que o terceiro
BGT	Ajusta o CP para o valor do primeiro operando, se o segundo operando for maior do que o terceiro
BLE	Ajusta o CP para o valor do primeiro operando, se o segundo operando for menor ou igual ao terceiro
BGE	Ajusta o CP para o valor do primeiro operando, se o segundo operando for maior ou igual ao terceiro

Fig. 4-5 Operações de controle.

Por exemplo, no laço (infinito) mostrado na Fig. 4-6, o rótulo “`início_do_laço`” está associado à instrução imediatamente seguinte a ele. A instrução de desvio no final do laço utiliza o rótulo “`início_do_laço`” como o seu alvo, indicando que o programa deve desviar para a instrução seguinte ao rótulo. Uma das tarefas do montador é calcular os endereços correspondentes a cada rótulo e inserir aquele endereço em qualquer instrução de desvio que faça referência a esse rótulo. Como discutiremos em mais detalhes no próximo capítulo, esses endereços são expressos como deslocamentos do desvio até o seu destino, em vez de ser um endereço de destino na memória.

Utilizar rótulos, em vez de endereços numéricos, proporciona duas vantagens. Primeiro, é muito mais fácil para o programador compreender. Olhando para o código-exemplo na Fig. 4-6, fica claro onde está o alvo da instrução de desvio, mesmo que você nada saiba a respeito da arquitetura do processador. Se o alvo fosse especificado como um endereço, você teria que saber quanto espaço cada instrução ocupa para ser capaz de descobrir o destino de cada desvio, e mesmo isso daria algum trabalho. A segunda vantagem é que o endereço correspondente ao rótulo pode mudar se as instruções antes do rótulo mudarem. Se fossem utilizados endereços fixos, o destino de cada desvio teria que ser modificado cada vez que mudasse o número de instruções antes do desvio. Assim, o programador, ou o compilador, utiliza rótulos para especificar o destino de cada desvio e, quando o programa é montado, o montador calcula o endereço de cada rótulo.

```

início_do_laço:
    (instrução)
    (instrução)
    (instrução)
    BR início_do_laço;

```

Fig. 4-6 Exemplo de rótulo.

4.4 ARQUITETURAS BASEADAS EM PILHA

Em uma arquitetura baseada em pilha, o banco de registradores é invisível para o programa. As instruções leem os seus operandos e escrevem os seus resultados em uma *pilha*, uma estrutura de dados do tipo último-a-entrar-primeiro-a-sair (LIFO – *Last In First Out*).

A Pilha

Como ilustrado na Fig. 4-7, a pilha é uma estrutura de dados do tipo LIFO. O nome *pilha* deve-se ao fato de a estrutura de dados atuar como uma pilha de pratos – um novo prato sempre é colocado em cima de uma pilha de pratos, e é o primeiro a ser removido quando alguém tira um prato da pilha. Uma pilha consiste de um conjunto de posições em memória, cada uma das quais pode reter uma palavra de dados. Quando um valor é acrescentado à pilha, ele é colocado na posição *superior* e todos os dados que atualmente estão na pilha descem uma posição. Os dados só podem ser removidos do topo da pilha. Quando isto é feito, todos os outros dados sobem uma posição. Em geral, os dados não podem ser lidos da pilha sem que isto altere; porém, alguns processadores podem ter operações especiais que permitam que isso ocorra.



Fig. 4-7 Funcionamento de uma pilha.

As pilhas suportam duas operações básicas: PUSH e POP. A operação PUSH tem um operando e o coloca no topo da pilha, empurrando todos os dados anteriores uma posição para baixo. A operação POP remove um valor do topo da pilha e disponibiliza-o como entrada para uma outra instrução. A Fig. 4-8 mostra como um conjunto de operações PUSH e POP afeta uma pilha.

Inicialmente, a pilha está vazia. A primeira operação PUSH coloca o valor 4 no topo da pilha. A segunda operação PUSH coloca o valor 5 no topo da pilha, deslocando o 4 para baixo, para a próxima posição. Então é executada a operação POP, a qual remove o 5 do topo da pilha. Então, o 4 é deslocado para o topo da pilha. Finalmente, uma operação PUSH 7 é executada, deixando 7 no topo da pilha e o 4 na próxima posição abaixo.

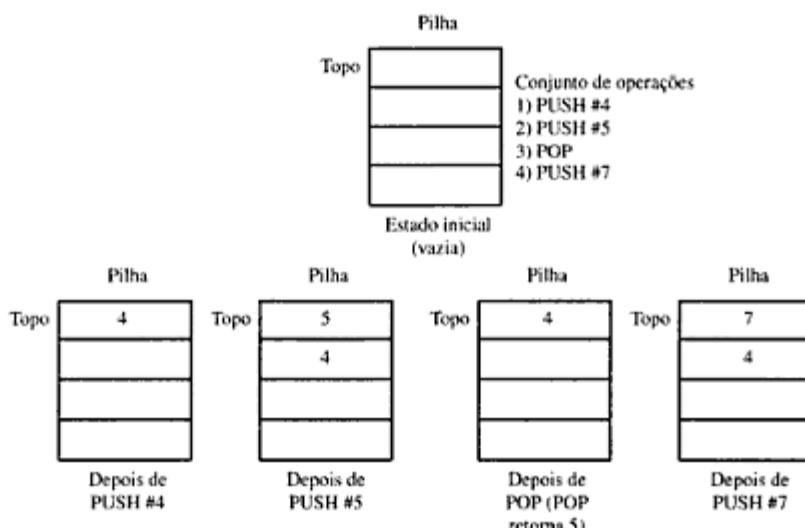


Fig. 4-8 Exemplo de pilha.

Implementando Pilhas

Como as pilhas são uma estrutura de dados abstrata, assume-se que elas tenham profundidade infinita, significando que o programa pode colocar uma quantidade arbitrária de dados na pilha. Na prática, as pilhas são implementadas utilizando-se *buffers* na memória, os quais são de tamanho finito. Se a quantidade de dados da pilha excede o espaço alocado para a pilha, ocorre um erro de *estouro de pilha* (*transbordo*).

PUSH #X	PILHA <- X
POP	a <- PILHA (o valor retirado é descartado)
LD	a <- PILHA PILHA <- (a)
ST	a <- PILHA (a) <- PILHA
ADD	a <- PILHA b <- PILHA PILHA <- a + b (cálculo com inteiros)
FADD	a <- PILHA b <- PILHA PILHA <- a + b (cálculo em ponto flutuante)
SUB	a <- PILHA b <- PILHA PILHA <- b - a (cálculo com inteiros)
FSUB	a <- PILHA b <- PILHA PILHA <- b - a (cálculo em ponto flutuante)
MUL	a <- PILHA b <- PILHA PILHA <- a × b (cálculo com inteiros)
FMUL	a <- PILHA b <- PILHA PILHA <- a × b (cálculo em ponto flutuante)
DIV	a <- PILHA b <- PILHA PILHA <- b / a (cálculo com inteiros)
FDIV	a <- PILHA b <- PILHA PILHA <- b / a (cálculo em ponto flutuante)
AND	a <- PILHA b <- PILHA PILHA <- a & b (cálculo orientado a bits)
OR	a <- PILHA b <- PILHA PILHA <- a b (cálculo orientado a bits)
NOT	a <- PILHA PILHA <- !a (negação orientada a bits)
ASH	a <- PILHA b <- PILHA PILHA <- a deslocado de b posições (deslocamento aritmético)
LSH	a <- PILHA b <- PILHA PILHA <- a deslocado de b posições (deslocamento lógico)
BR	PC <- PILHA
BEQ	a <- PILHA b <- PILHA c <- PILHA PC <- c se b é igual a

BNE	a <- PILHA b <- PILHA c <- PILHA PC <- c se b é diferente de a
BLT	a <- PILHA b <- PILHA c <- PILHA PC <- c se b é menor do que a
BGT	a <- PILHA b <- PILHA c <- PILHA PC <- c se b é maior do que a
BLE	a <- PILHA b <- PILHA c <- PILHA PC <- c se b é menor ou igual a
BGE	a <- PILHA b <- PILHA c <- PILHA PC <- c se b é maior ou igual a

Programação em Arquiteturas Baseadas em Pilha

Programas baseados em pilha são simplesmente sequências de instruções que são executadas uma após a outra. Dado um programa baseado em pilha e uma configuração inicial da pilha, o resultado do programa pode ser calculado aplicando-se a primeira instrução do programa, determinando-se o estado da pilha e da memória depois que a primeira instrução for completada. Repete-se esse procedimento para as instruções subsequentes.

Escrever programas para processadores baseados em pilha pode ser um pouco mais difícil, já que processadores baseados em pilha são mais adequados para a notação pós-fixada (NPR) do que para a notação infixa tradicional. A notação infixa é o modo tradicional de representar expressões matemáticas, na qual a operação é colocada entre os operandos. Na notação pós-fixada, a operação é colocada após os operandos. Por exemplo, a expressão infixa “ $2 + 3$ ” torna-se “ $2\ 3\ +$ ” na notação pós-fixada. Uma vez que uma expressão tenha sido codificada na notação pós-fixada, convertê-la para um programa baseado em pilha é uma operação simples. Começando pela esquerda, cada constante é substituída por uma operação PUSH para colocar a constante na pilha, e os operadores são recolocados junto com a instrução apropriada para a execução da operação.

Exemplo Crie um programa baseado em pilha que execute o seguinte cálculo:

$$2 + (7 \times 3)$$

Solução

Primeiro, precisamos converter a expressão para a notação pós-fixada. Isto é feito convertendo iterativamente cada subexpressão na sua expressão pós-fixada, de modo que $2 + (7 \times 3)$ torna-se $2 + (7\ 3\times)$ e então, $2\ (7\ 3\times)\ +$. Então, convertemos a expressão pós-fixada em uma série de instruções como descrito acima, resultando em

```
PUSH #2
PUSH #7
PUSH #3
MUL
ADD
```

Para verificar se este programa está correto, simulamos a sua execução à mão. Depois de três declarações PUSH, a pilha contém os valores 3, 7, 2 (começando do topo da pilha). A instrução MUL retira o 3 e o 7 da pilha e os multiplica, e coloca o resultado (21) na pilha, fazendo com que a pilha contenha 21 e 2. A instrução ADD retira estes dois valores da pilha e os soma, colocando o resultado (23) na pilha, que é o resultado do cálculo. Isto é igual ao resultado da expressão original, de modo que o programa está correto.

4.5 ARQUITETURAS BASEADAS EM REGISTRADORES DE USO GERAL

Em uma arquitetura baseada em registradores de uso geral (RUG), as instruções lêem os operandos e escrevem os seus resultados em um banco de registradores de acesso aleatório, semelhante àquele ilustrado na Fig. 4-12. O banco de registradores de uso geral permite que uma instrução faça acesso aos registradores em qualquer ordem, ao especificar o número (também chamado de ID) do registrador a ser acessado, de modo muito semelhante ao do sistema de memória que permite que os endereços na memória sejam acessados em qualquer ordem. Uma outra diferença significativa entre um banco de registradores de uso geral e uma pilha é que ler o conteúdo de um registrador de uso geral não o modifica, diferentemente ao retirar um valor de uma pilha. Leituras sucessivas de um registrador de uso geral, sem escritas entre elas, retornarão sempre o mesmo resultado, enquanto que retiradas sucessivas da pilha irão retornar o conteúdo da pilha em uma ordem LIFO.

Para tornar a programação mais fácil, muitas arquiteturas RUG designam significados especiais a alguns dos registradores do banco. Por exemplo, alguns processadores fazem a conexão física do registrador 0 (r_0) com o valor 0, para tornar mais fácil gerar esta constante comum, e outros fazem com que o contador de programas seja disponível como um dos registradores. Estes significados são atribuídos por hardware e não podem ser modificados pelos programas.

Banco de registradores	
Registrador 0	<dado>
Registrador 1	<dado>
Registrador 2	<dado>
Registrador 3	<dado>
Registrador 4	<dado>
Registrador 5	<dado>
Registrador 6	<dado>
Registrador 7	<dado>

Fig. 4-12 Banco de registradores de uso geral.

Instruções em uma Arquitetura RUG

As instruções de um processador RUG precisam especificar os registradores que detêm os seus operandos de entrada e o registrador onde o seu resultado será escrito. O formato mais comum para isto é uma instrução com três operandos, como mostrado na Fig. 4-13. Para a maioria das instruções aritméticas, o argumento mais à esquerda especifica o registrador de destino da instrução, enquanto que os outros argumentos especificam os registradores fonte⁷. Assim, a instrução ADD r_1, r_2, r_3 instrui o processador para ler os conteúdos dos registradores r_2 e r_3 , somá-los e escrever o resultado em r_1 . Os formatos para instruções que têm apenas um operando também são mostrados na figura.

Este formato de instrução é chamado “três operandos” – apesar do fato de que algumas operações têm apenas dois argumentos – para distingui-lo dos formatos de instruções de dois operandos, no qual um dos registradores de operando, como o mais à esquerda, também é o registrador de destino. Por exemplo, a instrução ADD r_1, r_2 em um formato de instrução de dois operandos, diz ao processador para somar os conteúdos de r_1 e r_2 e colocar o resultado em r_1 .

⁷ A codificação das instruções varia de processador para processador. Em especial, alguns processadores utilizam o operando mais à direita como o seu registrador de destino, com os outros operandos sendo as entradas para a instrução. Utilizamos a convenção de que o argumento mais à esquerda é o destino, porque este é o formato mais utilizado em textos sobre arquitetura de computadores.

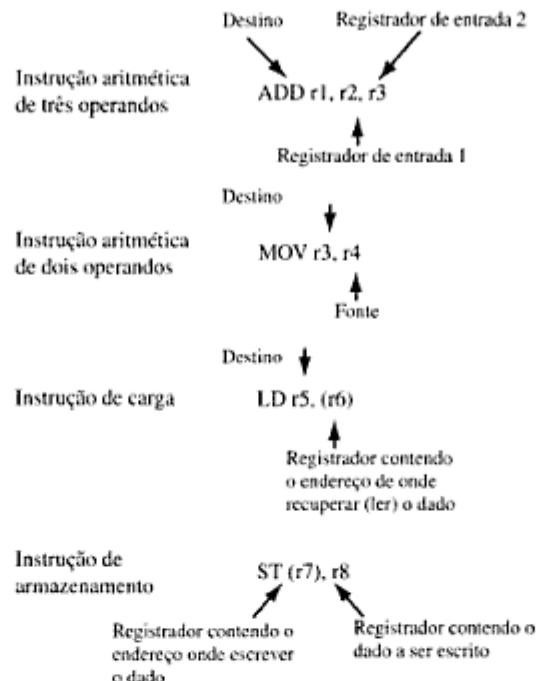


Fig. 4-13 Formatos de instrução de três operandos.

Os formatos de instruções de três operandos são mais flexíveis do que os formatos de dois operandos, na medida em que eles permitem que os registradores de entrada e de saída de uma instrução sejam escolhidos de forma independente, mas exigem mais *bits* para serem codificados. Muitas arquiteturas atuais tem 32 ou mais registradores, exigindo pelo menos 5 *bits* para codificar o ID de cada registrador referido pela instrução. Em arquiteturas de 16 *bits*, ou menores, isto faz com que seja difícil codificar uma instrução de três operandos em uma única palavra de dados, tornando as instruções de dois operandos mais atrativas. Em arquiteturas mais modernas, de 32 e de 64 *bits*, isto não é um grande problema e virtualmente todas estas arquiteturas utilizam codificações de instruções com três operandos. Como elas têm se tornado dominantes, este livro assumirá codificações de três operandos para as instruções RUG, a menos que especificado de forma diferente.

Uma diferença significativa entre as arquiteturas baseadas em pilha e as arquiteturas RUG é o fato de que em uma arquitetura RUG o programa pode escolher, a qualquer tempo, quais valores devem ser armazenados no banco de registradores, permitindo que o programa mantenha no banco de registradores os seus dados mais acessados. Em contraste, as restrições de acesso LIFO em uma pilha limitam a capacidade do programa de escolher quais dados estarão no banco de registradores a qualquer tempo. Nos primeiros computadores, pensava-se que esta era uma vantagem das arquiteturas de pilha, uma vez que elas tenderiam automaticamente a manter os dados mais referenciados no banco de registradores formado pelo topo da pilha. À medida que a tecnologia de compiladores avançou, técnicas de *alocação de registradores*, que selecionam os valores que devem ser mantidos no banco de registradores, foram aperfeiçoadas a um ponto no qual as arquiteturas RUG geralmente podem fazer um uso melhor dos seus banco de registradores do que as arquiteturas baseadas em pilha, fazendo com que as arquiteturas RUG tenham um desempenho melhor do que as arquiteturas baseadas em pilha.

Um Conjunto de Instruções para Arquiteturas RUG

Esta seção apresenta o conjunto-exemplo de instruções para um processador RUG que será utilizado para o resto dos exemplos neste livro. Esta arquitetura será descrita utilizando-se a mesma notação utilizada na seção Conjunto de Instruções para Arquitetura Baseada em Pilha (p. 65), mas estendendo a notação com “rX”, para referenciar o registrador X para armazenamento de valores inteiros, e “FY” para indicar o registrador Y para valores em ponto flutuante. Esta notação é utilizada porque muitos processadores utilizam bancos de registradores separados para dados inteiros ou em ponto flutuante.

Nosso conjunto de instruções RUG implementará as mesmas operações do conjunto de instruções baseadas em pilha apresentado anteriormente, exceto que o conjunto de instruções RUG não contém as operações PUSH e POP, uma vez que estas operações são utilizadas apenas para manipular a pilha. No entanto, o conjunto de instruções RUG contém uma operação MOV para copiar dados de um registrador para outro. Neste livro, utilizaremos o formato de instrução de três operandos e assumiremos que um dos operandos fonte, mas não ambos, em qualquer instrução, pode ser uma constante, em vez de um registrador, utilizando a notação #X para denotar constantes. O nosso conjunto de instruções RUG contém as seguintes operações:

LD ra, (rb) ³	ra <- (rb) (ra pode ser um registro de ponto flutuante)
ST (ra), rb	(ra) <- rb (rb pode ser um registro de ponto flutuante)
MOV ra, rb	ra <- rb (ra e/ou rb podem ser registros de ponto flutuante)
ADD ra, rb, rc	ra <- rb + rc (cálculo inteiro)
FADD fa, fb, fc	fa <- fb + fc (cálculo em ponto flutuante)
SUB ra, rb, rc	ra <- rb - rc (cálculo inteiro)
FSUB fa, fb, fc	fa <- fb - fc (cálculo em ponto flutuante)
MUL ra, rb, rc	ra <- rb rc (cálculo inteiro)
FMUL fa, fb, fc	fa <- fb fc (cálculo em ponto flutuante)
DIV ra, rb, rc	ra <- rb / rc (cálculo inteiro)
FDIV fa, fb, fc	fa <- fb / fc (cálculo em ponto flutuante)
AND ra, rb, rc	ra <- rb & rc (cálculo orientado a bit)
OR ra, rb, rc	ra <- rb rc (cálculo orientado a bit)
NOT ra, rb	ra <- !rb (negação orientada a bit)
ASH ra, rb, rc	ra <- rb deslocado por rc posições (deslocamento aritmético)
LSH ra, rb, rc	ra <- rb deslocado por rc posições (deslocamento lógico)
BR ra	PC <- ra
BR label	PC <- label
BEQ ra, rb, rc	PC <- ra se rb é igual a rc
BEQ label, rb, rc	PC <- label se rb é igual a rc
BNE ra, rb, rc	PC <- ra se rb não é igual a /é diferente de rc
BNE label, rb, rc	PC <- label se rb não é igual a /é diferente de rc
BLT ra, rb, rc	PC <- ra se rb é menor que rc
BLT label, rb, rc	PC <- label se rb é menor que rc
BGT ra, rb, rc	PC <- ra if se rb é maior que rc
BGT label, rb, rc	PC <- label se rb é maior que rc
BLE ra, rb, rc	PC <- ra se rb é menor ou igual a rc
BLE label, rb, rc	PC <- label se rb é menor ou igual a rc
BGE ra, rb, rc	PC <- ra se rb é maior ou igual a rc
BGE label, rb, rc	PC <- label se rb é maior ou igual a rc

³ Note que as operações de memória na nossa arquitetura RUG utilizam parênteses ao redor do nome do registrador que especifica o endereço ao qual elas fazem referência. Esta notação é utilizada para manter a coerência com os modos de endereçamento apresentados no próximo capítulo.

Programação em uma Arquitetura RUG

Assim como os programas para arquiteturas baseadas em pilha, os programas para arquiteturas RUG são simplesmente uma sequência de instruções individuais. Para descobrir o que uma sequência de instruções faz, executa-se uma a uma na ordem fornecida pela sequência, atualizando o conteúdo do banco de registradores após cada instrução. A programação de um processador RUG é menos estruturada do que a programação de uma arquitetura baseada em pilha, uma vez que há menos restrições na ordem pela qual as operações devem ser executadas.

Em um processador baseado em pilha, as operações precisam ser executadas em uma ordem tal que elas deixem os operandos para a próxima instrução no topo da pilha. Em um processador RUG, é válida qualquer ordem que coloque no banco de registradores os operandos para a instrução seguinte, antes que esta instrução seja executada; operações que fazem referência a diferentes registradores podem ser arbitrariamente reordenadas sem fazer com que o programa fique incorreto. Como veremos nos capítulos seguintes, muitos dos processadores modernos tiram proveito disso para reordenar as instruções em tempo de execução, de modo a melhorar o desempenho.

Exemplo Escreva um programa RUG que calcule a função $2 + (7 \times 3)$. Assuma que a arquitetura tem 16 registradores, r0 a r15, e que no início do programa todos os registradores contenham 0. O resultado do cálculo pode ser armazenado em qualquer registrador.

Solução

Um programa que faz isto é

```
MOV r1, #7
MOV r2, #3
MUL r3, r1, r2
MOV r4, #2
ADD r4, r3, r4
```

As primeiras duas instruções MOV colocam os valores 7 e 3 em r1 e r2, respectivamente. A instrução MUL os multiplica e coloca o resultado em r3. A instrução MOV seguinte coloca 2 em r4 e a instrução final ADD soma esta constante com o resultado da MUL, para gerar o resultado final.

Há duas coisas a serem observadas a respeito desta solução. Primeiro, a escolha de registradores foi arbitrária – qualquer escolha que não sobrescrevesse o valor de um registrador antes que ele fosse usado, funcionaria. Segundo, a instrução final sobrescreve um dos seus operandos. Uma vez que não precisamos utilizar novamente o valor em r4, isto está certo e foi feito para ilustrar que isso era possível.

4.6 COMPARANDO ARQUITETURAS BASEADAS EM PILHA E EM REGISTRADORES DE USO GERAL

As arquiteturas baseadas em pilha e em registradores de uso geral diferem principalmente nas suas interfaces com os seus bancos de registradores. Em arquiteturas baseadas em pilha, os dados são armazenados em uma pilha na memória. O banco de registradores do processador pode ser utilizado para implementar a parte superior da pilha, de modo a permitir um acesso mais rápido.

Em arquiteturas RUG, o banco de registradores é um dispositivo de acesso aleatório, onde cada registrador pode ser lido ou escrito de modo independente pelo processador. Nestas arquiteturas, o banco de registradores e a memória são completamente independentes, e os programas são responsáveis por mover os dados entre estes dois tipos de armazenamento, conforme necessário.

As arquiteturas baseadas em pilha foram utilizadas em alguns dos primeiros sistemas de computador por dois motivos. Primeiro, porque os operandos e o destino de uma instrução em uma arquitetura baseada em pilha são implícitos e as instruções utilizam menos bits para serem codificadas do que necessitam em arquiteturas baseadas em registradores de uso geral. Isso reduzia a quantidade de memória ocupada pelos programas, o que era uma questão significativa nas primeiras máquinas. Em segundo, as arquiteturas baseadas em pilha gerenciam automaticamente os registradores, liberando os programadores da necessidade de decidir quais dados devem ser mantidos no banco de registradores.

Uma outra vantagem das arquiteturas baseadas em pilha é que o conjunto de instruções não muda se o tamanho do banco de registradores mudar. Isto significa que programas escritos para um processador baseado em pilha podem ser executados em futuras versões do processador que tenham mais registradores. O impacto desse aumento será no desempenho da execução do programa, pois um número maior de informações da pilha poderá ser armazenada em registradores. Também é muito fácil fazer a compilação em arquiteturas baseadas em pilha – tão fácil que alguns compiladores geram versões de um programa baseadas em pilha como parte do processo de compilação, mesmo quando estão fazendo a compilação para um processador RUG.

Arquiteturas com registradores de uso geral tornaram-se dominantes nos últimos anos devido aos aperfeiçoamentos na tecnologia e à popularização das linguagens de alto nível. À medida que a capacidade das memórias aumentou e o seu preço diminuiu, o espaço ocupado por um programa tornou-se menos importante, tornando a vantagem do tamanho das instruções das arquiteturas baseadas em pilha igualmente menos importante. Outro ponto importante é que compiladores para arquiteturas RUG podem, em relação a arquiteturas baseadas em pilha, explorar a existência de vários registradores para disponibilizar, de forma mais adequada, valores de entrada para instruções. Isso representa uma melhoria de desempenho.

Por causa das suas vantagens de desempenho e da importância decrescente do tamanho do código, praticamente todos processadores das estações de trabalho mais recentes têm arquiteturas RUG. As arquiteturas baseadas em pilha são mais atrativas em sistemas dedicados, nos quais as necessidades de baixo custo e de baixo consumo de energia freqüentemente limitam a quantidade de memória que pode ser incluída em um sistema, fazendo com que o tamanho do código seja uma preocupação.

4.7 UTILIZANDO PILHAS PARA IMPLEMENTAR CHAMADAS DE PROCEDIMENTOS

Chamadas de procedimento são uma parte importante de todas as linguagens de computador. Elas permitem que funções utilizadas comumente sejam escritas uma vez e utilizadas quando necessárias, além de prover uma abstração, o que facilita que várias pessoas colaborem para escrever um programa. No entanto, diversas dificuldades estão envolvidas na implementação de chamadas de procedimentos:

1. Os programas precisam de um modo de passar dados para os procedimentos que eles chamam e receber resultados de volta.
2. Os procedimentos devem ser capazes de alocar espaço na memória para as variáveis locais, sem sobreescriver quaisquer dados utilizados pelo programa que fez a chamada.
3. Uma vez que os procedimentos podem ser chamados a partir de diferentes pontos dentro de um programa e freqüentemente são compilados separadamente do programa que os chama, geralmente é impossível determinar quais registradores podem ser utilizados sem problemas por um procedimento e quais contêm dados que serão necessários depois que o procedimento for completado.
4. Os procedimentos precisam um modo de descobrir de que ponto eles foram chamados, a fim de que a execução possa retornar ao programa que faz a chamada, assim que o procedimento for completado.

Para resolver estes problemas, a maioria dos sistemas utiliza uma estrutura de dados em pilha. Em arquiteturas RUG, a pilha é implementada na memória, como ilustrado na Fig. 4-9, enquanto que as arquiteturas baseadas em pilha podem fazer uso da pilha principal do processador. Quando um procedimento é chamado, um bloco de memória chamado *célula de pilha (stack frame)* é alocado na pilha, incrementando-se o apontador de topo de pilha pelo número de posições na célula de pilha. Como ilustrado na Fig. 4-14, a célula de pilha de um procedimento contém espaço para o conteúdo do banco de registradores do programa que fez a chamada, um valor para a localização para a qual o procedimento deve desviar quando ele for completado (o seu endereço de retorno), os argumentos de entrada para o procedimento e as variáveis locais do procedimento.

Quando um procedimento é chamado, o conteúdo do banco de registradores do programa que fez a chamada é copiado para dentro da célula de pilha, junto com a sua localização de retorno e os dados de entrada para o procedimento. Então, o procedimento utiliza o resto da célula de pilha para manter suas variáveis locais. Uma vez que o número de argumentos de entrada e variáveis locais varia de procedimento para procedimento, diferentes procedimentos terão células de pilha de diferentes tamanhos. A organização dos dados dentro da célula de pilha também varia entre diferentes processadores.

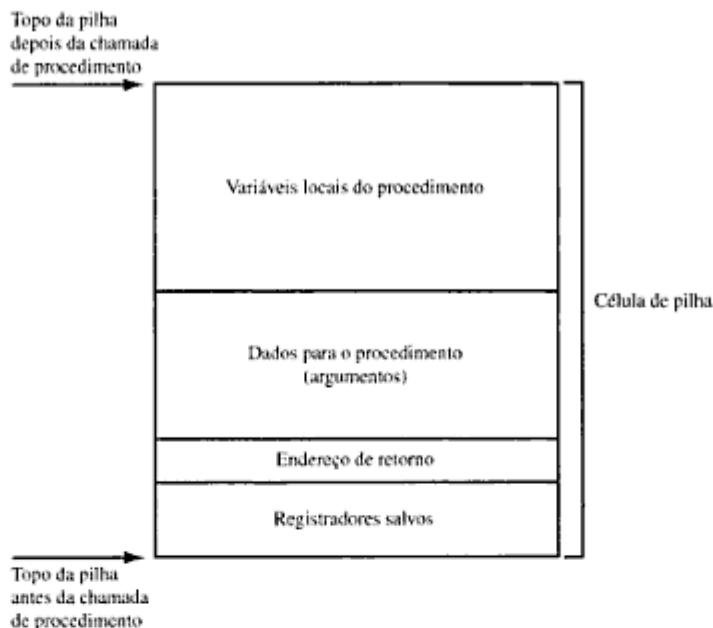


Fig. 4-14 Célula de pilha.

Quando um procedimento termina, ele salta para o endereço de retorno contido na célula de pilha, e a execução do programa que fez a chamada é retomada. O programa que fez a chamada lê o conteúdo do seu banco de registradores, o qual foi salvo na célula de pilha, e trata o resultado do procedimento que pode ser passado, ou por meio de um registrador específico, ou através da pilha. Finalmente, o apontador de topo da pilha é restaurado para a sua posição anterior à chamada do procedimento, retirando, da pilha, a célula de pilha.

Quando um programa faz chamadas de procedimento aninhadas (procedimentos que chamam outros procedimentos), cada procedimento aninhado aloca a sua célula de pilha sobre aquelas já existentes na pilha. Por exemplo, a Fig. 4-15 mostra o conteúdo da pilha durante a execução do procedimento `h()`, que foi chamado de dentro do procedimento `g()`. O procedimento `g()` foi chamado de dentro de `f()`, o qual foi chamado pelo programa principal. Desde que não haja um estouro (transbordo) da pilha, as chamadas de procedimento podem ser aninhadas em tantos níveis quanto necessário, e cada célula de pilha será retirada da pilha quando a execução retornar ao programa que fez a sua chamada.

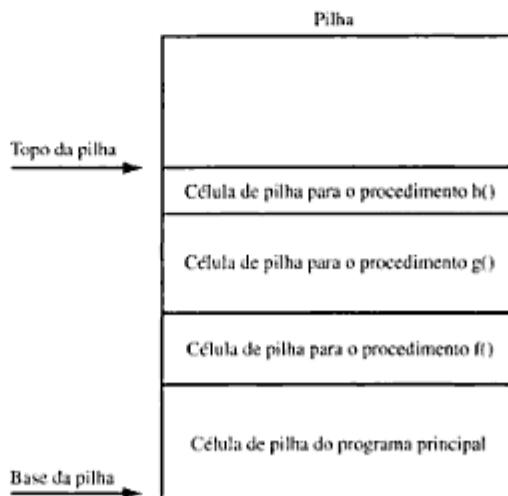


Fig. 4-15 Células de pilha aninhadas.

Convenções de Chamadas

Os vários sistemas de programação podem organizar, de diferentes modos, os dados em uma célula de pilha e podem exigir que os passos envolvidos na chamada de um procedimento sejam executados em ordens diferentes. As exigências de um sistema de programação com relação a como o procedimento é chamado e como os dados são passados entre um programa que faz uma chamada e os seus procedimentos são denominadas *convenções de chamada*.

Muitos sistemas de programação utilizam convenções de chamada para reduzir a quantidade de dados que precisa ser copiada de ou para a pilha durante uma chamada de procedimento. Por exemplo, uma convenção de chamada pode especificar um conjunto de registradores para passar os valores de entrada e de saída entre o programa que faz a chamada e o procedimento. Se esses valores cabem nestes registradores, então não será necessário colocá-los na pilha, reduzindo o número de referências à memória. Sistemas de programação geralmente também tentam reduzir o número de registradores que precisam ser salvos e restaurados durante uma chamada de procedimento ao identificar, no programa que faz a chamada, os registradores cujos valores, ou não serão necessários depois da chamada de procedimento, ou não serão sobreescritos pelo procedimento. Estes registradores não precisam ser salvos e restaurados, reduzindo o ônus da chamada de procedimento.

4.8 RESUMO

Este capítulo cobriu as arquiteturas baseadas em pilha e em registradores de uso geral, dois modelos de programação comuns em processadores. Os processadores baseados em pilha utilizam uma pilha do tipo LIFO, para manter os argumentos e os resultados das suas operações, enquanto que as arquiteturas RUG utilizam um banco de registradores de acesso aleatório. Os dois tipos de processadores geralmente fornecem o mesmo tipo de instrução, com algumas poucas exceções, como as instruções PUSH e POP de uma arquitetura baseada em pilha.

As arquiteturas baseadas em pilhas foram utilizadas em alguns dos primeiros computadores porque elas tinham codificações de instruções muito compactas. A maioria das instruções precisa apenas especificar a operação a ser executada, uma vez que a pilha é a fonte dos seus operandos e o destino dos seus resultados. Poucas operações, como PUSH, utilizam operadores constantes, os quais aumentam o número de bits necessários para codificar a operação. É muito fácil, também, fazer a compilação em arquiteturas baseadas em pilha e elas permitem a compatibilidade entre processadores com diferentes números de registradores, porque o banco de registradores é parte da pilha e é invisível ao programa.

Arquiteturas com registradores de uso geral permitem que o programa escolha quais valores são mantidos no banco de registradores de acesso aleatório. Combinado com o fato de que, diferentemente de retirar um valor de uma pilha, ler um valor de um registrador não remove o seu valor, isto geralmente permite que os programas de arquiteturas RUG atinjam um desempenho melhor do que programas baseados em pilha, porque eles exigem menos instruções para executar um cálculo. À medida que o preço das memórias diminuiu, tornando menos significativa a concisão dos programas baseados em pilha, este desempenho melhorado fez com que as arquiteturas baseadas em registradores de uso geral fossem o modelo de programação dominante.

O restante deste livro assumirá um modelo de processador RUG, uma vez que isto vai ao encontro da maioria dos processadores atuais. No próximo capítulo, discutiremos alguns dos detalhes do projeto de processadores, incluindo o debate RISC versus CISC e o projeto de banco de registradores.

Problemas Resolvidos

Operações de Deslocamento

- 4.1** Qual é o resultado das seguintes operações, quando executadas em um processador de 8 bits que utiliza complemento de 2 para a representação de números inteiros negativos?
- LSH 14, 3
 - ASH 17, 5
 - LSH -23, -2
 - ASH -23, -2

Solução

- A representação de inteiros de 8 bits, em complemento de 2, para 14 é 0b0000 1110. Fazendo o deslocamento à esquerda por três posições produz 0b0111 0000, que é a representação inteira de 112. Constatase-se então que o resultado está correto, já que $112 = 14 \times 2^3$.

Solução

Um dos benefícios de uma pilha é que ela apresenta ao programador a ilusão de um espaço de armazenamento infinitamente grande. Bancos de registradores contêm apenas um pequeno número de posições de armazenamento, de modo que um sistema que utilizasse apenas o banco de registradores para implementar a sua pilha poderia ser capaz de colocar apenas uma pequena quantidade de dados nela. Os programadores que utilizassem tal sistema teriam que fazer um acompanhamento cuidadoso da quantidade de dados presentes na pilha a qualquer tempo, de modo a evitar seu estouro, o que tornaria o sistema muito mais difícil de programar. Sistemas que permitem que a pilha se expanda para a memória, quando ela ultrapassa o tamanho do banco de registradores, são capazes de fornecer uma aproximação muito melhor da pilha ideal de profundidade infinita.

Programação em Arquiteturas Baseadas em Pilha (I)

- 4.6 Qual valor permanece na pilha depois da seguinte sequência de instruções?

```
PUSH #4
PUSH #7
PUSH #8
ADD
PUSH #10
SUB
MUL
```

Solução

Depois das três operações PUSH, a pilha contém 8, 7, 4 (começando no topo). A instrução ADD retira o 8 e o 7, então coloca 15 na pilha, fazendo com que seu conteúdo seja 15, 4. A instrução PUSH 10 faz com que a pilha seja 10, 15, 4. A operação SUB retira 10 e 15 da pilha, subtrai 10 de 15 e coloca 5 de volta na pilha. (Lembre-se que a SUB subtrai o valor de cima na pilha do próximo valor abaixo, de modo que PUSH x, PUSH y, SUB gera x-y.) Finalmente, a MUL retira 5 e 4 da pilha e coloca 20, deixando 20 na pilha.

Programação em Arquiteturas Baseadas em Pilha (II)

- 4.7 Escreva um programa baseado em pilha que calcule a seguinte função: $5 + (3 \times 7) - 8$, assumindo que inicialmente a pilha está vazia.

Solução

Primeiro, convertemos esta expressão para NPR, produzindo $(5 (3 7 \times) +) 8 -$. Note que na NPR os parênteses são completamente desnecessários para gerar o resultado correto. Eles foram incluídos apenas para que o leitor pudesse analisar a expressão NPR com mais facilidade.

Então, a expressão é traduzida em instruções:

```
PUSH #5
PUSH #3
PUSH #7
MUL
ADD
PUSH #8
SUB
```

Programação em Arquiteturas Baseadas em Pilha (III)

- 4.8 Assumindo que a pilha esteja vazia, escreva um programa baseado em pilha que calcule $((10 \times 8) + (4 - 7))^2$.

Solução

Como o nosso processador não fornece uma instrução para calcular o quadrado de um valor, precisamos calcular $(10 \times 8) + (4 - 7)$ duas vezes, de modo que a pilha contenha duas cópias deste resultado, multiplicando-os. (Também seria possível armazenar o resultado na memória e carregá-lo duas vezes na pilha.) Como veremos no Problema 4.13, uma arquitetura RUG pode calcular isto de modo muito mais eficiente, ao utilizar o mesmo registrador como ambas entradas de uma operação MUL.

Transformar o cálculo em formato NPR e depois em instruções, produz o seguinte programa:

```

PUSH 10
PUSH 8
MUL
PUSH 4
PUSH 7
SUB
ADD [Neste ponto, a pilha contém apenas o primeiro resultado de
(10 × 8) + (4 - 7)]
PUSH 10
PUSH 8
MUL
PUSH 4
PUSH 7
SUB
ADD [Neste ponto, a pilha contém duas cópias de
(10 × 8) + (4 - 7)]
MUL

```

Arquiteturas RUG (I)

- 4.9** Explique brevemente como as instruções acessam os seus operandos em uma arquitetura RUG.

Solução

Em uma arquitetura RUG, as instruções leem os seus operandos e escrevem os seus resultados em um banco de registradores de acesso aleatório. Cada instrução especifica tanto os registradores que contêm os operandos como o registrador onde o resultado deve ser escrito.

Arquiteturas RUG (II)

- 4.10** Explique brevemente a diferença entre os formatos de instrução com dois e três operandos.

Solução

Em um formato de instrução com dois operandos, um dos registradores de entrada para a instrução também é o registrador de saída. Em um formato de instrução com três entradas, cada um dos registradores de entrada e saída da instrução são especificados separadamente. Instruções com três operandos são mais flexíveis que as instruções com dois operandos, mas elas exigem mais bits para serem codificadas.

Programação RUG (I)

- 4.11** Assumindo que todos os registradores começam contendo 0, qual é o valor de r7 depois que a seguinte seqüência de instruções for executada?

```

MOV r7, #4
MOV r8, #3
ADD r9, r7, r7
SUB r7, r9, r8
MUL r9, r7, r7

```

Solução

As duas instruções MOV colocam os valores 4 e 3 em r7 e r8, respectivamente. A instrução ADD soma 4 (o valor em r7) a 4 (o valor em r7), obtendo 8, e coloca isto em r9. A instrução SUB subtrai 3 de 8, obtendo 5 e coloca isto em r7. Finalmente, a instrução MUL multiplica 5 e 5 para obter 25 e coloca este valor em r9. Portanto, o valor em r7 ao final da seqüência de instruções é 5.

Programação RUG (II)

- 4.12** Escreva um programa em linguagem de montagem RUG que execute o seguinte cálculo, assumindo que todos registradores começam contendo 0. O resultado final pode ser colocado em qualquer registrador.

$$5 + (3 \times 7) - 8$$

Solução

Aqui está um programa, mas existem muitas variações aceitáveis:

```
MOV r1, #5
MOV r2, #3
MOV r3, #7
MOV r4, #8
MUL r5, r2, r3
ADD r6, r1, r5
SUB r7, r6, r4
```

Programação RUG (III)

- 4.13** Assumindo que todos os registradores comecem contendo 0, escreva um programa RUG que calcule $((10 \times 8) + (4 - 7))^2$. O resultado final pode ser colocado em qualquer registrador.

Solução

Novamente, aqui está um de muitos programas que calculam esta função:

```
MOV r1, #10
MOV r2, #8
MOV r3, #4
MOV r4, #7
MUL r5, r1, r2
SUB r6, r3, r4
ADD r7, r5, r6
MUL r8, r7, r7
```

Comparando Arquiteturas Baseadas em Pilha e RUG (I)

- 4.14** Cite duas vantagens das arquiteturas baseadas em pilha sobre as RUG.

Solução

Este capítulo discutiu três vantagens das arquiteturas baseadas em pilha:

- As instruções em uma arquitetura baseada em pilha ocupam menos memória do que as instruções das arquiteturas RUG, já que instruções de uma arquitetura baseada em pilha não tem que especificar os registradores que contêm as suas fontes ou o registrador onde os seus resultados devem ser escritos.
 - O banco de registradores em uma arquitetura baseada em pilha é invisível para o programador, sendo a parte superior da pilha. Como resultado, futuras implementações de uma arquitetura baseada em pilha podem conter diferentes números de registradores e, ainda assim, executarão os programas escritos para o processador antigo. Em contraste, o número de registradores em uma arquitetura RUG é codificado no conjunto de instruções por meio do número de bits alocados para cada nome de registrador, impedindo que programas escritos para um processador RUG seja executado em um processador RUG com um número diferente de registradores.
 - A pilha fornece a ilusão de uma área de armazenamento infinita, de modo que os programas não tem que se preocupar com um estouro/transbordo do volume de armazenamento no banco de registradores.
- Relacionar quaisquer duas destas vantagens é uma resposta correta para o problema.

Comparando Arquiteturas Baseadas em Pilha e RUG (II)

- 4.15** Cite duas vantagens das arquiteturas RUG sobre as baseadas em pilha.

Solução

As duas principais vantagens das arquiteturas RUG sobre as baseadas em pilha são:

1. Ler um registrador em uma arquitetura RUG não afeta o seu conteúdo, enquanto que ler um valor do topo de uma pilha remove o valor da pilha. Quando um dado valor é utilizado mais de uma vez em um programa, a arquitetura RUG pode alocar aquele valor para um registrador e lê-lo repetidamente quando for necessário. Em contraste, as arquiteturas baseadas em pilha precisam, ou utilizar instruções para duplicar o valor na pilha a cada vez que ele é utilizado como uma entrada para uma instrução, ou armazenar o valor na memória e recarregá-lo cada vez que ele for usado.
2. Programas RUG podem escolher quais valores manter no banco de registradores, enquanto que as arquiteturas baseadas em pilha são limitadas pela natureza LIFO da pilha. Técnicas de alocação de registradores nos compiladores modernos são boas o suficiente para manter, no banco de registradores, os valores aos quais é feita referência com mais frequência no programa, de modo que são necessárias menos referências à memória para completar um dado programa em uma arquitetura RUG do que em pilha, melhorando o desempenho.

Comparando Arquiteturas Baseadas em Pilha e RUG (III)

- 4.16** Por que as arquiteturas RUG tornaram-se dominantes sobre as arquiteturas baseadas em pilha?

Solução

As vantagens-chave das arquiteturas baseadas em pilha são o tamanho menor dos seus programas e a ausência da necessidade de alocação de registradores, enquanto que arquiteturas RUG são capazes de atingir um desempenho melhor quando o programador/compilador faz um bom trabalho na alocação de valores aos registradores. Nos primeiros computadores, a memória era muito cara, de modo que reduzir o tamanho do programa era importante. Também, muitas das técnicas de alocação de registradores utilizadas atualmente não haviam sido desenvolvidas.

À medida que a tecnologia avançou e a memória ficou mais barata, fazer programas de tamanho reduzido nas arquiteturas baseadas em pilha tornou-se menos importante. Além disso, a maior parte da programação agora é feita em linguagens de alto nível, e os compiladores contêm algoritmos sofisticados para a alocação de registradores que fazem bom uso do banco de registradores em uma arquitetura RUG. Por causa disso, as vantagens de desempenho das arquiteturas RUG tornaram-se mais significativas do que a vantagem de tamanho de código das arquiteturas baseadas em pilha, o que tornou as arquiteturas RUG a melhor opção para a maioria dos projetos de processadores.

Células de Pilha

- 4.17** Um programa está sendo executado em uma arquitetura com 32 registradores, cada um deles com 32 bits de largura. Os endereços neste sistema também tem 32 bits. O programa chama um procedimento que ocupa quatro argumentos de 32 bits e aloca oito variáveis internas de 32 bits. Qual é o tamanho da célula de pilha do procedimento? (Assuma que todos os registradores do programa que faz a chamada precisam ser salvos.)

Solução

A célula de pilha tem que ser grande o suficiente para manter o conteúdo do banco de registradores do chamador, o endereço de retorno, as entradas do procedimento e as suas variáveis locais. Isto é $(32 + 1 + 4 + 8) = 45$ valores de 32 bits para este procedimento, ou 180 bytes.

5.3 ARQUITETURA DO CONJUNTO DE INSTRUÇÕES

Quando a maior parte da programação de computadores era feita em linguagem *assembly*, a arquitetura do conjunto de instruções era considerada a parte mais importante da arquitetura dos computadores, porque ela definia quão difícil seria para obter um desempenho ótimo do sistema. Ao longo dos anos, a arquitetura do conjunto de instruções tornou-se menos significativa por diversos motivos. Primeiro, a maior parte da programação é atualmente feita com linguagens de alto nível. Em segundo lugar, e o mais significativo, os consumidores passaram a esperar *compatibilidade* entre as diferentes gerações de um sistema de computadores, o que significa que eles esperam que os programas que eram executados no seu sistema antigo sejam executados sem modificações no seu novo sistema. Como resultado, o conjunto de instruções de um novo processador freqüentemente precisa ter o mesmo conjunto de instruções de um processador anterior, eventualmente com algumas instruções adicionais, o que significa que a maior parte do esforço de projeto de um processador vai para o aperfeiçoamento da microarquitetura, de modo a melhorar o desempenho.

No capítulo anterior, cobrimos uma das mais significativas decisões envolvidas no projeto da arquitetura do conjunto de instruções de um processador (ACI) – a escolha de um modelo de programação. Como foi discutido naquele capítulo, o modelo de programação baseado em registradores de uso genérico tornou-se dominante e será assumido pelo restante deste livro. Neste capítulo, cobriremos quatro questões remanescentes na arquitetura do conjunto de instruções: o debate RISC versus CISC, a escolha dos modos de endereçamento, a utilização de codificações de instruções de comprimento fixo ou variável e instruções vetoriais multimídia.

RISC versus CISC

Antes da década de 1980, havia um grande foco na redução do “intervalo semântico” entre as linguagens utilizadas para programar computadores e as linguagens de máquina. Acreditava-se que tornar as linguagens de máquina mais parecidas às linguagens de programação de alto nível resultaria em melhor desempenho, pela redução do número de instruções exigidas para implementar um programa e tornaria mais fácil compilar programas em linguagens de alto nível para a linguagem de máquina. O resultado final disto foi o projeto de conjuntos de instruções que continham instruções muito complexas.

À medida que a tecnologia de compiladores era aperfeiçoada, os pesquisadores começaram a questionar se estes sistemas com instruções complexas, conhecidos como computadores com conjuntos de instruções complexas (CISC), forneciam um desempenho melhor do que sistemas baseados em conjuntos de instruções mais simples. Esta segunda classe de sistemas tornou-se conhecida como computadores de conjuntos de instruções reduzidas (RISC).

O argumento principal a favor dos computadores CISC é que esses geralmente exigem menos instruções que os computadores RISC para executar uma dada operação, de modo que um computador CISC teria um desempenho melhor que um computador RISC que executasse instruções à mesma taxa. Além disto, programas escritos para arquiteturas CISC tendem a tomar menos espaço na memória que o mesmo programa escrito para a arquitetura RISC. O principal argumento a favor de computadores RISC é que os seus conjuntos de instruções mais simples freqüentemente permitem que eles sejam implementados com freqüências de relógio mais altas, permitindo executar mais instruções na mesma quantidade de tempo. Com uma freqüência de relógio maior, um processador RISC permite que ele execute programas em menos tempo do que um processador CISC levaria para executar os seus programas (os quais exigem menos instruções). O processador RISC terá um desempenho melhor.

Durante a década de 80, e no início dos anos 90, houve muita controvérsia na comunidade de arquitetura de computadores com relação a qual das duas abordagens era a melhor, e, dependendo do ponto de vista, qualquer uma das duas pode ser considerada vencedora. A grande maioria dos conjuntos de instruções introduzidas desde a década de 80 tem sido de arquiteturas RISC, com o argumento de que é superior. Por outro lado, a arquitetura Intel x86 (IA-32), que utiliza um conjunto de instruções CISC, é a arquitetura dominante para PCs/estações de trabalho; em termos de número de processadores vendidos, as arquiteturas CISC foram as vencedoras.

Ao longo dos últimos 20 anos, tem havido uma certa convergência entre as arquiteturas, tornando difícil determinar se uma arquitetura é RISC ou CISC. As arquiteturas RISC incorporaram algumas das instruções complexas mais úteis das arquiteturas CISC, confiando na sua microarquitetura para implementar estas instruções com pouco impacto no ciclo de relógio, e as arquiteturas CISC abandonaram instruções complexas que não eram utilizadas com freqüência suficiente para justificar a sua implementação.

Uma delinearção clara entre as duas é que as arquiteturas RISC são *arquiteturas de carga-armazenamento*, o que significa que apenas instruções de carga e armazenamento podem acessar a memória do sistema. A arquitetura com registradores de uso genérico, descrita no Capítulo 4, é uma arquitetura de carga-armazenamento.

Em muitas arquiteturas CISC, instruções aritméticas e outras podem ler as suas entradas ou escrever as suas saídas diretamente na memória, em vez de fazê-lo sobre registradores. Por exemplo, uma arquitetura CISC pode permitir uma operação ADD, na forma ADD (r1), (r2), (r3), onde os parênteses em volta do nome do registro indicam que o registro contém o endereço de memória onde um operando pode ser encontrado ou o resultado pode ser escrito. Ao utilizar esta notação, a instrução ADD (r1), (r2), (r3) instrui o processador para somar o valor contido na localização de memória, cujo endereço está armazenado em r2; ao valor contido na localização de memória, cujo endereço está armazenado r3, e armazenar o resultado na memória, no endereço contido em r1.

Esta diferença entre as arquiteturas de carga-armazenamento e as arquiteturas que podem unir referências de memória com outras operações é um excelente exemplo das diferenças entre as arquiteturas RISC e CISC. Já que as arquiteturas RISC são implementadas utilizando o modelo carga-armazenamento, um processador RISC exigiria várias instruções para implementar a única operação ADD CISC descrita acima. No entanto, o *hardware* necessário para implementar o processador CISC seria mais complexo, uma vez que ele teria que ser capaz de buscar os operandos da instrução na memória, de modo que o processador CISC provavelmente teria uma duração de ciclo mais longa que o processador RISC (ou exigiria mais ciclos para executar cada instrução).

Exemplo Na nossa arquitetura de carga-armazenamento com registradores de uso genérico, quantas instruções são necessárias para implementar a mesma função que a operação ADD CISC descrita acima?

Assuma que os endereços de memória adequados estão presentes em r1, r2 e r3, no início da execução da instrução.

Solução

São necessárias quatro instruções:

```
LD r4, (r2)
LD r5, (r3)
ADD r6, r4, r5
ST (r1), r6
```

Este exemplo mostra que uma arquitetura RISC pode exigir muito mais operações para implementar uma função do que uma arquitetura CISC, se bem que este é um exemplo extremo. Ele também mostra que arquiteturas RISC geralmente exigem mais registradores para implementar uma função do que as CISC, uma vez que todas as entradas de uma instrução precisam ser carregadas em um banco de registradores antes que a instrução possa ser executada. No entanto, processadores RISC têm a vantagem de "dividir" uma operação CISC complexa em várias operações RISC, permitindo ao compilador organizar as operações RISC para um desempenho melhor. Por exemplo, se as referências à memória ocupam vários ciclos para serem executadas (como geralmente elas fazem), um compilador para uma arquitetura RISC pode colocar outras instruções entre as instruções LD e a ADD do exemplo. Isto dá tempo para que as instruções LD sejam completadas, antes que os seus resultados sejam solicitados pela ADD, evitando que a ADD tenha que esperar por suas entradas. Em contraste, a instrução CISC não tem escolha, a não ser esperar que suas entradas sejam recuperadas da memória do sistema, potencialmente atrasando outras instruções.

Modos de Endereçamento

Como já discutimos, uma das principais diferenças entre as arquiteturas RISC e CISC é o conjunto de instruções que pode acessar a memória. Uma questão relacionada a isto que afeta tanto a arquitetura RISC quanto a CISC é a escolha de quais *modos de endereçamento* a arquitetura suporta. Os modos de endereçamento de uma arquitetura são o conjunto de sintaxes e métodos que as instruções utilizam para especificar um endereço de memória, seja como o endereço-alvo de uma referência de memória ou como o endereço para o qual uma instrução de desvio irá. Dependendo da arquitetura, alguns dos modos de endereçamento podem estar disponíveis apenas para algumas das instruções que fazem referência à memória. Arquiteturas que permitem que qualquer instrução que faça referência à memória utilize qualquer modo de endereçamento são descritas como *ortogonais*, porque a escolha do modo de endereçamento é independente da escolha da instrução.

Até aqui, utilizamos apenas dois modos de endereçamento: endereçamento de registrador para carga de instruções, armazenamento de instruções e instruções CISC que fazem referência à memória; e o endereçamento de rótulos para instruções de desvio. No endereçamento de registrador, uma instrução lê o valor de um registrador e utiliza aquilo como o endereço da referência de memória ou do desvio-alvo. Utilizamos a sintaxe (rx) para indicar que o modo de endereçamento de registrador está sendo utilizado. Um conjunto de instruções que fornecesse apenas endereçamento de registrador poderia ser possível, uma vez que qualquer endereço poderia ser calculado, utilizando-se instruções aritméticas, e ser armazenado em um registrador. Processadores fornecem outros modos de endereçamento

mento porque estes reduzem o número de instruções exigidas para calcular endereços, desta forma melhorando o desempenho. O segundo modo de endereçamento que vimos até agora é o endereçamento de rótulos, no qual uma instrução de desvio especifica o seu destino como um rótulo (*label*) a ser colocado em uma instrução em outro local do programa. Como foi discutido no último capítulo, esses rótulos de texto não aparecem na versão do programa em linguagem de máquina. De fato, a maioria das instruções de desvio não contém explicitamente os seus endereços de destino. É o montador/ligador que traduz o rótulo em um *deslocamento* (que pode ser positivo ou negativo), a partir da localização da instrução de desvio para a localização do seu alvo. Na verdade, a instrução de desvio diz ao processador a distância que ele está da instrução-alvo, em vez de indicar exatamente onde a instrução-alvo está localizada. O processador acrescenta o deslocamento ao apontador da instrução (registrator contador de programa – CP) da instrução de desvio para obter o endereço destino do desvio.

Utilizar um deslocamento em vez de endereços explícitos para o endereçamento de rótulos tem duas vantagens. Primeiro, ele reduz o número de *bits* necessários para codificar a instrução. A maioria dos desvios tem alvos que são relativamente próximos ao desvio, de modo que um número menor de *bits* pode ser utilizado para codificar o deslocamento. Quando um desvio tem um alvo que está longe, o endereço-alvo pode ser calculado por outras instruções, e um modo de endereçamento de registrador, ou similar, pode ser usado. Em segundo lugar, utilizar deslocamentos em vez de endereços explícitos em instruções de desvio permite que o carregador coloque o programa em localizações diferentes na memória, sem ter que modificar o programa. Se fossem usados endereços explícitos, os endereços destino de cada desvio teriam que ser recalculados cada vez que o programa fosse carregado. Esse recurso é particularmente útil para bibliotecas dinâmicas, que são vinculadas ao programa em tempo de execução não possuindo conhecimento prévio dos endereços onde serão carregados.

Exemplo Uma instrução em linguagem *assembly* “BR label1” é ligada e vinculada como parte de um programa maior. O ligador calcula o deslocamento da instrução de desvio para label1 como 0x437 bytes. Se a instrução de desvio for carregada no endereço 0x4000, qual é o endereço destino do desvio? E se a instrução for carregada no endereço 0x4400?

Solução

O endereço-alvo do desvio é a soma do endereço do desvio (o CP quando o desvio é executado) e o deslocamento. Quando o desvio é carregado no endereço 0x4000, o endereço-alvo é 0x4437 (0x4000 + 0x437). Quando o desvio é carregado no endereço 0x4400, o endereço-alvo é 0x4837.

Um outro modo de endereçamento fornecido por muitos processadores é o *endereçamento registrador mais imediato*. Neste modo, que é expresso como imm(rx), o valor do registrador especificado é somado ao valor imediato (constante) especificado na instrução para gerar um endereço de memória.

Exemplo Se o valor de r4 for 0x13000, qual será o endereço ao qual a instrução LD –0x80(r4) faz referência?

Solução

Modo de endereçamento rótulo mais imediato soma o valor imediato ao valor do registrador para obter o endereço destino. Acrescentar –0x080 a 0x13000 dá 0x12f80, o endereço referido pela carga.

O modo registrador mais imediato é extremamente útil para acessar estruturas de dados, que tendem a ter campos que são localizados em deslocamentos fixos a partir do início da estrutura de dados. Ao utilizar este modo de endereçamento, um programa que precise fazer referência a diferentes campos de uma estrutura de dados pode simplesmente carregar o endereço de início da estrutura de dados em um registrador e, então, usar o modo registrador mais imediato para acessar os diferentes campos da estrutura de dados, reduzindo o número de instruções necessárias para executar os cálculos de endereço e o número de registros exigidos para armazenar endereços.

Muitos outros modos de endereçamento foram implementados ao longo dos anos. Em geral, eles são variações do modo registrador mais imediato. Por exemplo, alguns conjuntos de instruções permitem que se some um deslocamento de rótulo a um imediato, ou um deslocamento de rótulo a um valor de registrador mais um imediato.

Um problema com todos os modos de endereçamento que calculam o seu endereço é que eles aumentam o tempo de execução das instruções que os utilizam, uma vez que o processador precisa executar um cálculo antes que o endereço possa ser enviado ao sistema de memória. Para fornecer flexibilidade de endereçamento, sem aumentar a latência de memória, algumas arquiteturas fornecem modos de endereçamento de *pós-incremento*, em vez dos modos de endereçamento no estilo registrador mais imediato. Esses modos de endereçamento, para os quais utilizare-

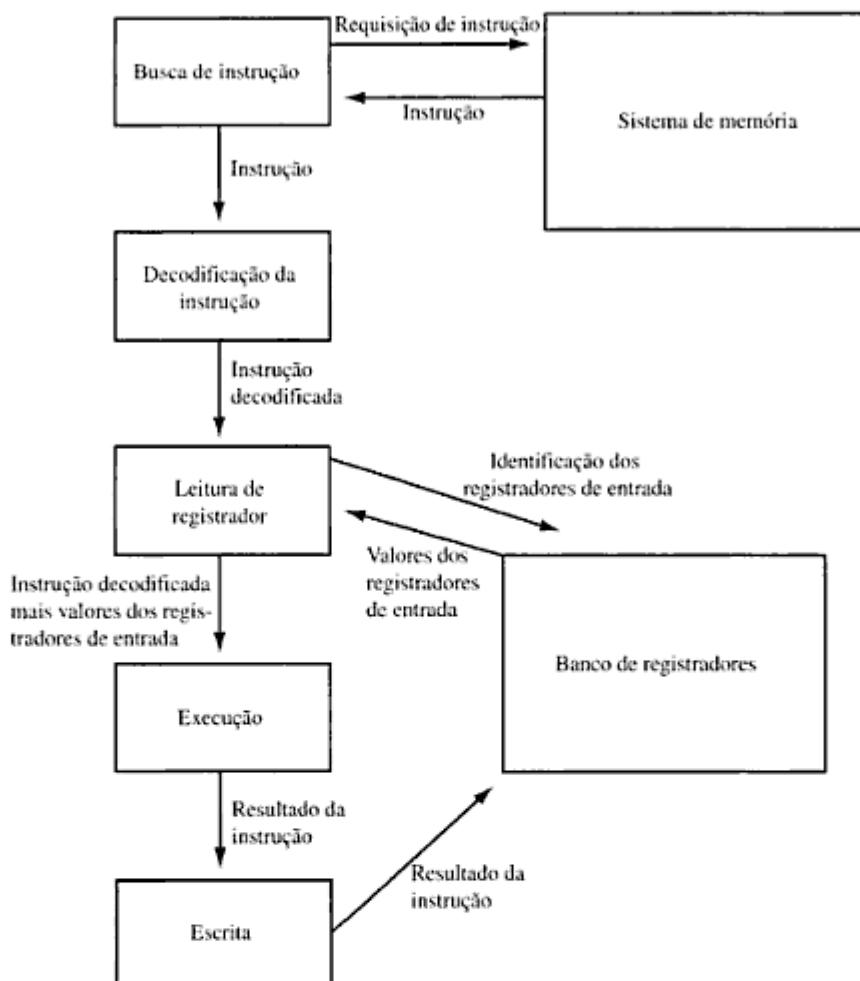


Fig. 5-4 Execução de instruções.

As instruções que acessam o sistema de memória têm um fluxo de execução semelhante, exceto que a saída da unidade de execução é enviada ao sistema de memória por ser, ou o endereço de uma operação de leitura, ou o endereço e o dado de uma operação de escrita. No caso de uma operação de leitura, o valor lido é armazenado no banco de registradores.

Muitas unidades de execução são implementadas utilizando uma estrutura física semelhante àquela mostrada na Figura 5.4. Os módulos que implementam os diferentes passos na execução da instrução são fisicamente dispositos próximos um ao outro interconectados através de linhas de barramento. À medida que a instrução é executada, os dados fluem pelo barramento, de um módulo para o seguinte, com cada módulo executando o seu trabalho em uma seqüência.

Microprogramação

Em um processador *microprogramado*, o *hardware* não precisa executar diretamente as instruções do conjunto de instruções. Ao invés disso, o *hardware* executa microoperações muito simples, e cada instrução determina uma seqüência de microoperações que são utilizadas para implementar a instrução. Essencialmente, cada instrução do conjunto de instruções é traduzida pelo *hardware* em um pequeno programa de microinstruções, de modo semelhante

ao modo como um compilador traduz cada instrução de um programa em linguagem de alto nível para uma sequência de instruções em linguagem *assembly*. Por exemplo, um processador microprogramado pode traduzir a instrução ADD r1, r2, r3 em seis microoperações: uma que lê o valor de r2 e o envia para uma entrada do somador; uma que lê o valor de r3 e o envia para a outra entrada do somador, uma que executa a adição, uma que escreve o resultado da adição em r1, uma que incrementa o valor do contador de programa para que ele aponte para a próxima instrução e uma que busque a próxima instrução da memória. Cada microoperação geralmente demanda um ciclo do processador para ser executada, de modo que, em tal sistema, uma instrução ADD exigiria seis ciclos para ser completada.

Processadores microprogramados contêm uma pequena memória que mantém a seqüência de microinstruções utilizadas para implementar cada instrução do conjunto de instruções. Para executar uma instrução, um processador microprogramado acessa esta memória para localizar o conjunto de microinstruções necessárias para implementar a instrução e, então, executa as microinstruções em seqüência.

A microprogramação tornou-se popular porque as tecnologias utilizadas para implementar os computadores mais antigos (válvulas, transistores discretos e circuitos integrados de pequena escala) limitavam a quantidade de *hardware* que poderia ser construído dentro do processador, e os projetistas de computadores desejavam definir conjuntos de instruções com instruções complexas, de modo a reduzir o número de instruções necessárias para implementar um programa. Ao utilizar a microprogramação, os projetistas podiam construir um *hardware* simples, microprogramando-o para executar as instruções complexas.

Processadores modernos tendem a não utilizar a microprogramação por dois motivos. Primeiramente porque agora tornou-se prático implementar a maior parte das instruções dos processadores diretamente no *hardware*, por causa dos avanços na tecnologia VLSI, o que torna o microcódigo desnecessário. Em segundo lugar, processadores microprogramados tendem a ter um desempenho pior do que os processadores não microprogramados, por causa do tempo adicional envolvido na busca de cada microinstrução na memória de microinstruções.

Projeto do Banco de Registradores

Até aqui, tratamos o banco de registradores como um único dispositivo que contém dados em ponto flutuante e inteiros. A maioria dos processadores não implementam seus registradores desta forma; implementam bancos ou conjuntos de registradores separados para dados inteiros e para ponto flutuante. O banco de registradores para inteiros são referidos utilizando-se a sintaxe "rx", que temos utilizado até aqui para nomes de registradores, e os registradores de ponto flutuante são referidos como "fx". Utilizar esta sintaxe torna mais claro qual é o arquivo de registros que está sendo referido pelas instruções, como cargas e armazenamentos, e que possam precisar fazer referência a um dos bancos de registradores. Instruções aritméticas geralmente são restrinidas ao acesso ao banco de registradores apropriado para o tipo de cálculo que elas executam, se bem que algumas instruções aritméticas possam transferir dados entre bancos de registradores.

Os processadores implementam o banco de registradores separados por dois motivos. Primeiro, isto permite que sejam colocados fisicamente próximos às unidades de execução que os utilizam: o banco de registradores para inteiros pode ser colocado próximo às unidades que executam operações inteiras e o de ponto flutuante, próximo às unidades de execução de ponto flutuante. Isso reduz o comprimento da fiação que liga os bancos de registradores às unidades de execução e, portanto, o tempo necessário para enviar dados de um para outro.

O segundo motivo é que bancos de registradores separados ocupam menos espaço nos processadores que executam mais de uma instrução por ciclo. Os detalhes disto estão além do escopo deste livro, mas o tamanho de um banco de registradores cresce aproximadamente com o quadrado do número de leituras e escritas simultâneas que o banco permite. Para executar uma instrução por ciclo, um banco de registradores precisa permitir duas leituras e uma escrita por ciclo, uma vez que algumas operações aritméticas lêem dois registros e escrevem em um registrador. Cada operação adicional que o processador queira executar em um ciclo aumenta o número de leituras e escritas simultâneas (chamadas portas) por um fator de 3. Essa divisão em banco de registradores para inteiros e para ponto flutuante reduz o número de portas necessárias para cada um. Uma vez que a área de um banco de registradores cresce mais rápida do que linearmente com o número de portas, dois bancos ocupam uma área menor do que um que forneça o mesmo número de portas.

5.5 RESUMO

O objetivo deste capítulo foi fornecer uma introdução ao projeto de processadores, como uma preparação para os próximos dois capítulos, que fornecem discussões mais profundas sobre duas técnicas que são amplamente utilizadas para melhorar o desempenho de processadores: *pipelining* e paralelismo no nível de instruções. O capítulo começou com uma discussão sobre a diferença entre a arquitetura do conjunto de instruções e a microarquitetura do processador. A arquitetura do conjunto de instruções é o projeto das instruções que um processador fornece, incluindo o modelo de programação, o conjunto de operações fornecidas, os modos de endereçamento que o processador suporta e a seleção de quais instruções podem acessar a memória. A microarquitetura de processadores cobre os detalhes de como o processador é implementado. Em geral, a arquitetura do conjunto de instruções refere-se a qualquer aspecto da arquitetura que é visível para um programador em linguagem *assembly*, enquanto que a microarquitetura do processador cobre os detalhes que afetam a rapidez com que um programa é executado. Existe um sobreposição substancial entre essas duas categorias. Por exemplo, uma arquitetura de conjunto de instruções que forneça um certo número de instruções complexas pode exigir uma microarquitetura de processador com um desempenho pior do que a microarquitetura que implemente apenas instruções mais simples.

Dentro da arquitetura do conjunto de instruções, discutimos a diferença entre instruções RISC e CISC, cujo elemento central é a exigência de que arquiteturas RISC sejam arquiteturas de carga-armazenamento, enquanto que a maior parte das arquiteturas CISC permite que outras operações também façam referência à memória. Foi discutido o impacto dos modos de endereçamento e das codificações dos conjuntos de instruções sobre o tamanho dos programas, o desempenho e a complexidade do *hardware* necessário para implementar o processador. Finalmente, foi dada uma introdução às instruções vetoriais multimídia, um acréscimo relativamente novo a muitos conjuntos de instruções.

Nossa introdução à microarquitetura de processadores incluiu um diagrama em blocos do fluxo de dados através de um processador, com a discussão da função de cada elemento no fluxo. Então, discutimos brevemente a microprogramação, uma técnica comumente utilizada para implementar processadores no passado, mas que atualmente é pouco utilizada por causa do seu impacto sobre o desempenho. A nossa discussão sobre a microarquitetura de processadores foi concluída com a cobertura das vantagens e desvantagens envolvidas no projeto do banco de registradores.

No próximo capítulo, discutiremos *pipelining*, uma técnica que melhora o desempenho de processadores ao sobrepor a execução de várias instruções. Isso permite freqüências de relógio mais altas e melhora a taxa pela qual as instruções são executadas. *Pipelining* é freqüentemente combinado com paralelismo no nível das instruções, o objeto do Capítulo 7, para produzir processadores que suprem várias instruções em cada ciclo e sobreponem a execução de instruções para aumentar o número de ciclos de relógio por segundo.

Problemas Resolvidos

Arquitetura do Conjunto de Instruções

- 5.1 O que é uma arquitetura de carga-armazenamento e quais são os prós e os contras de tal arquitetura, quando comparada com outras arquiteturas registradores de uso genérico (RUG)?

Solução

Uma arquitetura de carga-armazenamento é aquela na qual apenas instruções de carga e armazenamento podem acessar o sistema de memória. Em outras arquiteturas RUG, algumas, ou todas as outras instruções, podem ler os seus operandos a partir do sistema de memória ou escrever os seus resultados nele. A vantagem principal de arquiteturas que não são de carga-armazenamento é o número reduzido de instruções necessárias para implementar um programa e a menor utilização do conjunto de registradores. A vantagem de arquiteturas de carga-armazenamento é que limitar o conjunto de instruções que podem acessar o sistema de memória torna a microarquitetura mais simples, o que freqüentemente permite a implementação de um relógio com freqüência maior. Dependendo do que for mais significativo – a freqüência do relógio aumentar ou o número de instruções diminuir –, qualquer uma das abordagens pode resultar em um desempenho melhor.

RISC versus CISC (I)

- 5.2 Reescreva o seguinte fragmento de um programa em estilo CISC de modo que ele seja executado corretamente em um processador RISC (carga-armazenamento) que executa o conjunto de instruções RUG delineada no capítulo an-

Instruções Vetoriais Multimídia (I)

- 5.8 Se um programa opera sobre tipos de dados de 8 bits e as instruções vetoriais multimídia de um processador operam sobre palavras de dados de 64 bits, qual é o aumento máximo de velocidade que pode ser atingido ao se utilizar instruções vetoriais multimídia? Assuma que todas as instruções demoram o mesmo tempo para serem executadas.

Solução

Oito valores de 8 bits podem ser colocados em uma palavra de 64 bits. Portanto, o processador pode executar oito operações de 8 bits em paralelo, utilizando instruções vetoriais multimídia. Se todas as instruções no programa fossem substituídas por instruções vetoriais multimídia, o programa exigiria 1/8 do número de instruções que o programa original, para aumento máximo de velocidade de 8 vezes.

Instruções Vetoriais Multimídia (II)

- 5.9 Se dois registros contêm os valores 0xab0890c2 e 0x4598ee50, qual é o resultado da adição deles, utilizando-se:
- Operações vetoriais multimídia que operam sobre dados de 8 bits?
 - Operações vetoriais multimídia que operam sobre dados de 16 bits?
- Assuma que a aritmética de saturação não está sendo usada.

Solução

Para encontrar o resultado utilizando operações vetoriais multimídia, simplesmente dividimos as palavras de dados de entrada em pedaços de tamanho apropriado e os somamos. Isto resulta nas seguintes adições e resultados finais diferentes (todos os números estão em hexadecimal):

- $(ab + 45), (08 + 98), (90 + ee), (c2 + 50) \rightarrow 0xf0a07e12$
- $(ab08 + 4598), (90c2 + ee50) \rightarrow 0xf0a07f12$

Codificações de Comprimento Fixo versus Comprimento Variável (I)

- 5.10 Quais são os prós e os contras de codificações de instruções de comprimento fixo e variável?

Solução

Codificações de instruções de comprimento variável reduzem a quantidade de memória que os programas ocupam, uma vez que cada instrução ocupa apenas o espaço que ela precisa. Instruções em um esquema de codificação de comprimento fixo ocupam todas o mesmo espaço de armazenamento que a instrução mais longa do conjunto de instruções, o que significa que existe algum número de bits desperdiçados na codificação de instruções que utilizam poucos operandos, não permitem entradas de constantes e assim por diante.

No entanto, conjuntos de instruções de comprimento variável exigem uma lógica de decodificação das instruções mais complexa que os conjuntos de instruções de comprimento fixo, tornando mais difícil calcular o endereço da próxima instrução na memória. Assim, processadores com conjuntos de instruções de comprimento fixo freqüentemente podem ser implementados a freqüências de relógio mais altas do que os processadores com conjuntos de instruções de comprimento variável.

Codificações de Comprimento Fixo versus Comprimento Variável (II)

- 5.11 Um processador possui 32 registradores, utiliza valores imediatos de 16 bits e tem 142 instruções no seu conjunto de instruções. Em um dado programa, 20% das instruções ocupam um registrador de entrada e tem um registrador de saída; 30% têm dois registradores de entrada e um registrador de saída; 25% têm um registrador de entrada, um registrador de saída e também ocupam uma entrada imediata; os 25% restantes têm uma entrada imediata e um registrador de saída.
- Quantos bits são necessários para cada um dos quatro tipos de instruções? Assuma que o conjunto de instruções exige que o comprimento de todas seja um múltiplo de 8 bits.
 - Quanta memória a menos o programa ocupará se for utilizada a codificação de um conjunto de instruções com comprimento variável, ao contrário de uma com comprimento fixo?

Coleção
SCHAUM

A essência do conhecimento

Os livros da Coleção Schaum são estruturados de maneira que o aluno possa aprender a matéria e estudá-la de acordo com o seu ritmo. Além de apresentar o conteúdo essencial, atendo-se a tópicos fundamentais, os textos reúnem uma grande quantidade de exercícios, o que permite testar as habilidades adquiridas. Para o professor, é um material didático completo, com teoria, problemas resolvidos e complementares.

**Teoria e Problemas de
Arquitetura de Computadores
aborda os seguintes tópicos:**

- Introdução
 - Representação de dados e aritmética de computadores
 - Organização de computadores
 - Modelos de programação
 - Projeto de processadores
 - Utilização de *pipelines*
 - Paralelismo no nível da instrução
 - Sistemas de memória
 - *Caches*
 - Memória virtual
 - Entrada e saída
 - Multiprocessadores
-



www.bookman.com.br

ISBN 85-363-0250-X



9 788536 502508

Material com direitos reservados