

tornariam o processador consideravelmente mais lento. Por exemplo, suponha que 10% das instruções sejam stores. Se o CPI sem falhas de cache fosse 1,0, gastar 100 ciclos extras em cada escrita levaria a um CPI de $1,0 + 100 \times 10\% = 11$, reduzindo o desempenho por um fator maior que 10.

Uma solução para esse problema é usar um **buffer de escrita** (ou write buffer), que armazena os dados enquanto estão esperando para serem escritos na memória. Após escrever os dados na cache e no buffer de dados, o processador pode continuar a execução. Quando uma escrita na memória principal é concluída, a entrada no buffer de escrita é liberada. Se o buffer de escrita estiver cheio quando o processador atingir uma escrita, o processador precisará sofrer stall até que haja uma posição vazia no buffer de escrita. Naturalmente, se a velocidade em que a memória pode completar escritas for menor do que a velocidade em que o processador está gerando escritas, nenhuma quantidade de buffer pode ajudar, pois as escritas estão sendo geradas mais rápido do que o sistema de memória pode aceitá-las.

buffer de escrita

Uma fila que contém os dados enquanto estão esperando para serem escritos na memória.

A velocidade em que as escritas são geradas também pode ser *menor* do que a velocidade com que a memória pode aceitá-las, e stalls ainda podem ocorrer. Isso pode acontecer quando as escritas ocorrem em bursts (ou rajadas). Para reduzir a ocorrência desses stalls, os processadores normalmente aumentam a profundidade do buffer de escrita para além de uma única entrada.

A alternativa para um esquema write-through é um esquema chamado **write-back**, no qual, quando ocorre uma escrita, o novo valor é escrito apenas no bloco da cache. O bloco modificado é escrito no nível inferior da hierarquia quando ele é substituído. Os esquemas write-back podem melhorar o desempenho, especialmente quando os processadores podem gerar escritas tão rápido ou mais rápido do que as escritas podem ser tratadas pela memória principal; entretanto, um esquema write-back é mais complexo de implementar do que um esquema write-through.

write-back

Um esquema que trata das escritas atualizando valores apenas no bloco da

cache e, depois, escrevendo o bloco modificado no nível inferior da hierarquia quando o bloco é substituído.

No restante desta seção, descreveremos as caches de processadores reais e examinaremos como elas tratam leituras e escritas. Na [Seção 5.8](#), descreveremos o tratamento de escritas em mais detalhes.

Detalhamento

As escritas introduzem várias complicações nas caches que não estão presentes para leituras. Discutiremos aqui duas delas: a política nas falhas de escrita e a implementação eficiente das escritas em caches write-back.

Considere uma falha em uma cache write-through. A estratégia mais comum é alocar um bloco na cache, chamado *alocar na escrita*. O bloco é apanhado da memória e depois a parte apropriada do bloco é sobrescrita. Uma estratégia alternativa é atualizar a parte do bloco na memória, mas não colocá-la em cache, o que se chama *não aloca na escrita*. A motivação é que, às vezes, os programas escrevem blocos de dados, como quando o sistema operacional zera uma página de memória. Nesses casos, a busca associada com a falha de escrita inicial pode ser desnecessária. Alguns computadores permitem que a política de alocação de escrita seja alterada com base em cada página.

Implementar stores de modo realmente eficaz em uma cache que usa uma estratégia write-back é mais complexo do que em uma cache write-through. Uma cache write-through pode escrever os dados na cache e ler a tag; se a tag for diferente, então haverá uma falha. Como a cache é write-through, a substituição do bloco na cache não é catastrófica, pois a memória tem o valor correto. Em uma cache write-back, precisamos escrever o bloco novamente na memória se os dados na cache estiverem modificados e tivermos uma falha de cache. Se simplesmente substituíssemos o bloco em uma instrução store antes de sabermos se o store teve acerto na cache (como poderíamos fazer para uma cache write-through), destruiríamos o conteúdo do bloco, que não é copiado, no próximo nível da hierarquia da memória.

Em uma cache write-back, como não podemos substituir o bloco, os stores ou exigem dois ciclos (um ciclo para verificar um acerto seguido de um ciclo para efetivamente realizar a escrita) ou exigem um buffer de escrita para conter esses dados — na prática, permitindo que o store leve apenas um ciclo

por meio de um pipeline de memória. Quando um buffer de store é usado, o processador realiza a consulta de cache e coloca os dados no buffer de store durante o ciclo de acesso de cache normal. Considerando um acerto de cache, os novos dados são escritos do buffer de store para a cache no próximo ciclo de acesso de cache não usado.

Por comparação, em uma cache write-through, as escritas sempre podem ser feitas em um ciclo. Lemos a tag e escrevemos a parte dos dados do bloco selecionado. Se a tag corresponder ao endereço do bloco escrito, o processador pode continuar normalmente, já que o bloco correto foi atualizado. Se a tag não corresponder, o processador gera uma falha de escrita para buscar o resto do bloco correspondente a esse endereço.

Muitas caches write-back também incluem buffers de escrita usados para reduzir a penalidade de falha quando uma falha substitui um bloco modificado. Em casos como esse, o bloco modificado é movido para um buffer write-back associado com a cache enquanto o bloco requisitado é lido da memória. Depois, o buffer write-back é escrito novamente na memória. Considerando que outra falha não ocorra imediatamente, essa técnica reduz à metade a penalidade de falha quando um bloco modificado precisa ser substituído.

Uma cache de exemplo: o processador Intrinsity FastMATH

O Intrinsity FastMATH é um microprocessador embutido que usa a arquitetura MIPS e uma implementação de cache simples. Próximo ao final do capítulo, examinaremos o projeto de cache mais complexo dos microprocessadores ARM e Intel, mas começaremos com este exemplo simples, mas real, por questões didáticas. A [Figura 5.12](#) mostra a organização da cache de dados do Intrinsity FastMATH.

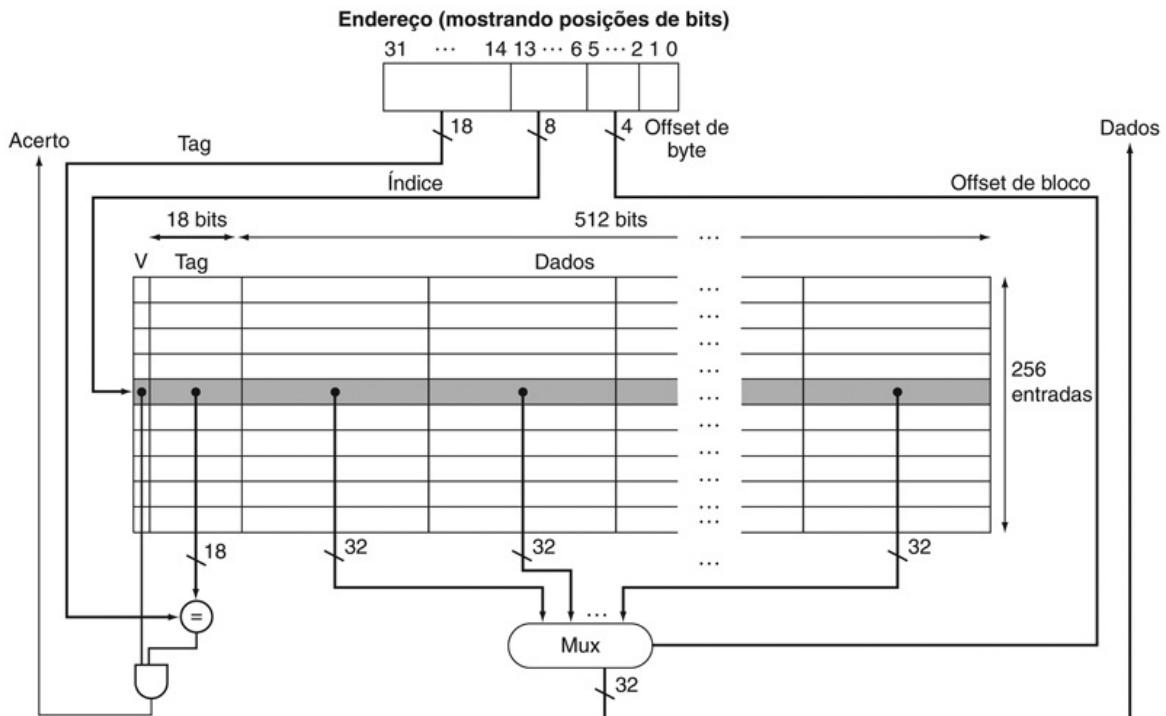


FIGURA 5.12 Cada cache de 16 KiB no Intrinsity FastMATH contém 256 blocos com 16 palavras por bloco.

O campo tag possui 18 bits de largura, e o campo índice possui 8 bits de largura, enquanto um campo de 4 bits (bits 5 a 2) é usado para indexar o bloco e selecionar a palavra do bloco usando um multiplexador de 16 para 1. Na prática, para eliminar o multiplexador, as caches usam uma RAM grande separada para os dados e uma RAM menor para as tags, com o offset de bloco fornecendo os bits de endereço extras para a RAM grande de dados. Nesse caso, a RAM grande possui 32 bits de largura e precisa ter 16 vezes o número de palavras como blocos na cache.

Este processador possui um pipeline de 12 estágios. Quando está operando na velocidade de pico, o processador pode requisitar uma palavra de instrução e uma palavra de dados em cada clock. Para satisfazer às demandas do pipeline sem stalls, são usadas caches de instruções e de dados separadas. Cada cache possui 16 KiB, ou 4096 palavras, com blocos de 16 palavras.

As requisições de leitura para a cache são simples. Como existem caches de dados e de instruções separadas, sinais de controle separados serão necessários para ler e escrever em cada cache. (Lembre-se de que precisamos atualizar a cache de instruções quando ocorre uma falha.) Portanto, as etapas para uma requisição de leitura para qualquer uma das caches são as seguintes:

1. Enviar o endereço à cache apropriada. O endereço vem do PC (para uma instrução) ou da ALU (para dados).
2. Se a cache sinalizar acerto, a palavra requisitada estará disponível nas linhas de dados. Como existem 16 palavras no bloco desejado, precisamos selecionar a palavra correta. Um campo índice de bloco é usado para controlar o multiplexador (mostrado na parte inferior da figura), que seleciona a palavra requisitada das 16 palavras do bloco indexado.
3. Se a cache sinalizar falha, enviaremos o endereço para a memória principal. Quando a memória retorna com os dados, nós os escrevemos na cache e, então, os lemos para atender à requisição.

Para escritas, o Intrinsity FastMATH oferece write-through e write-back, deixando a cargo do sistema operacional decidir qual estratégia usar para cada aplicação. Ele possui um buffer de escrita de uma entrada.

Que taxas de falhas de cache são atingidas com uma estrutura de cache como a usada pelo Intrinsity FastMATH? A [Figura 5.13](#) mostra as taxas de falhas para as caches de instruções e de dados. A taxa de falhas combinada é a taxa de falhas efetiva por referência para cada programa após considerar a frequência diferente dos acessos a instruções e a dados.

Taxa de falhas de instruções	Taxa de falhas de dados	Taxa de falhas combinadas efetivas
0,4%	11,4%	3,2%

FIGURA 5.13 Taxas de falhas de instruções e dados aproximadas para o processador Intrinsity FastMATH para benchmarks SPEC CPU2000.

A taxa de falhas combinada é a taxa de falhas efetiva para a combinação da cache de instruções de 16 KiB. e da cache de dados de 16 KiB.. Ela é obtida ponderando as taxas de falhas individuais de instruções e de dados pela frequência das referências a instruções e dados.

Embora a taxa de falhas seja uma característica importante dos projetos de cache, a medida decisiva será o efeito do sistema de memória sobre o tempo de execução do programa; em breve veremos como a taxa de falhas e o tempo de execução estão relacionados.

Detalhamento

Uma cache combinada com um tamanho total igual à soma das duas **caches divididas** normalmente terá uma taxa de acertos melhor. Essa taxa mais alta ocorre porque a cache combinada não divide rigidamente o número de entradas que podem ser usadas por instruções daquelas que podem ser usadas por dados. Entretanto, muitos processadores usam uma instrução split e uma cache de dados para aumentar a *largura de banda* da cache. (Também pode haver menos falhas de conflito; veja Seção 5.8.)

Aqui estão taxas de falhas para caches do tamanho dos encontrados no processador Intrinsity FastMATH, e para uma cache combinada cujo tamanho é igual ao total das duas caches:

- Tamanho total da cache: 32KB.
- Taxa de falhas efetiva da cache dividida: 3,24%.
- Taxa de falhas da cache combinada: 3,18%.

A taxa de falhas da cache dividida é apenas ligeiramente pior.

A vantagem de dobrar a largura de banda da cache, suportando acessos a instruções e a dados simultaneamente, logo suplanta a desvantagem de uma taxa de falhas um pouco maior. Essa constatação é outro lembrete de que não podemos usar a taxa de falhas como a única medida de desempenho de cache, como mostra a Seção 5.4.

cache dividida

Um esquema em que um nível da hierarquia de memória é composto de duas caches independentes que operam em paralelo uma com a outra, com uma tratando instruções e a outra manipulando dados.

Resumo

Começamos a seção anterior examinando a mais simples das caches: uma cache diretamente mapeada com um bloco de uma palavra. Nesse tipo de cache, tanto os acertos quanto as falhas são simples, já que uma palavra pode estar localizada exatamente em um lugar e existe uma tag separada para cada palavra. A fim de manter a cache e a memória consistentes, um esquema de write-through pode ser usado, de modo que toda escrita na cache também faz com que a memória seja atualizada. A alternativa ao write-through é um esquema write-back, que copia um bloco de volta para a memória quando ele é substituído; discutiremos esse esquema mais detalhadamente em seções futuras.

Para tirar vantagem da localidade espacial, uma cache precisa ter um tamanho de bloco maior do que uma palavra. O uso de um bloco maior diminui a taxa de falhas e melhora a eficiência da cache reduzindo a quantidade de armazenamento de tag em relação à quantidade de armazenamento de dados na cache. Embora um tamanho de bloco maior diminua a taxa de falhas, ele também pode aumentar a penalidade de falha. Se a penalidade de falha aumentasse linearmente com o tamanho de bloco, blocos maiores poderiam facilmente levar a um desempenho menor.

Para evitar a perda de desempenho, a largura de banda da memória principal é aumentada de modo a transferir blocos de cache de maneira mais eficiente. Os dois métodos comuns para aumentar a largura de banda externa à DRAM são tornar a memória mais larga e a intercalação. Os projetistas de DRAM têm constantemente melhorado a interface entre o processador e a memória, a fim de aumentar a largura de banda das transferências no modo rajado e reduzir o custo dos tamanhos de bloco de cache maiores.

Verifique você mesmo

A velocidade do sistema de memória afeta a decisão do projetista sobre o tamanho do bloco de cache. Quais dos seguintes princípios de projeto de cache normalmente são válidos?

1. Quanto mais curta for a latência da memória, menor será o bloco de cache.
2. Quanto mais curta for a latência da memória, maior será o bloco de cache.
3. Quanto maior for a largura de banda da memória, menor será o bloco de cache.
4. Quanto maior for a largura de banda da memória, maior será o bloco de cache.

5.4. Medindo e melhorando o desempenho da cache

Nesta seção, começamos examinando como medir e analisar o desempenho da cache; depois, exploramos duas técnicas diferentes para melhorar o desempenho da cache. Uma delas focaliza o decréscimo da taxa de falhas reduzindo a probabilidade de dois blocos de memória diferentes disputarem o mesmo local da cache. A segunda técnica reduz a penalidade de falha acrescentando um nível adicional na hierarquia. Essa técnica, chamada *caching multinível*, apareceu

inicialmente nos computadores de topo de linha sendo vendidos por mais de US\$100.000 em 1990; desde então, ela se tornou comum nos computadores desktop vendidos por algumas centenas de dólares!

O tempo de CPU pode ser dividido nos ciclos de clock que a CPU gasta executando o programa e os ciclos de clock que gasta esperando o sistema de memória. Normalmente, consideramos que os custos do acesso à cache que são acertos fazem parte dos ciclos de execução normais da CPU. Portanto,

Tempo de CPU

$$= (\text{ciclos de clock de execução da CPU} + \text{Ciclos de clock de stall de memória}) \\ \times \text{Tempo de ciclo de clock}$$

Os ciclos de clock de stall de memória vêm principalmente das falhas de cache, e é isso que iremos considerar aqui. Também limitamos a discussão a um modelo simplificado do sistema de memória. Nos processadores reais, os stalls gerados por leituras e escritas podem ser muito complexos, e a previsão correta do desempenho normalmente exige simulações extremamente detalhadas do processador e do sistema de memória.

Os ciclos de clock de stall de memória podem ser definidos como a soma dos ciclos de stall vindo das leituras, mais os provenientes das escritas:

$$\text{Ciclos de clock de stall de memória} = (\text{Ciclos de stall de leitura} + \text{Ciclos de stall de escrita})$$

Os ciclos de stall de leitura podem ser definidos em função do número de acessos de leitura por programa, a penalidade de falha nos ciclos de clock para uma leitura e a taxa de falhas de leitura:

$$\text{Ciclos de stall de leitura} = \frac{\text{Leituras}}{\text{Programa}} \times \text{Taxa de falhas de leitura} \\ \times \text{Penalidade de falha de leitura}$$

As escritas são mais complicadas. Para um esquema write-through, temos duas origens de stalls: as falhas de escrita, que normalmente exigem que busquemos o bloco antes de continuar a escrita (veja a seção “Detalhamento” na

Seção “Tratando escritas”, anteriormente neste capítulo, para obter mais informações sobre como lidar com escritas), e os stalls do buffer de escrita, que ocorrem quando o buffer de escrita está cheio ao ocorrer uma escrita. Assim, os ciclos de stall para escritas são iguais à soma desses dois fatores:

Ciclos de stall de escrita

$$= \left(\frac{\text{Leituras}}{\text{Programa}} \times \text{Taxa de falhas de escrita} \times \text{Penalidade de falha de escrita} \right) + \text{Stalls do buffer de escrita}$$

Como os stalls do buffer de escrita dependem da proximidade das escritas e não apenas da frequência, não é possível fornecer uma equação simples para calcular esses stalls. Felizmente, nos sistemas com uma profundidade de buffer de escrita razoável (por exemplo, quatro ou mais palavras) e uma memória capaz de aceitar escritas em uma velocidade que excede, significativamente, a frequência de escrita média em programas (por exemplo, por um fator de duas vezes), os stalls do buffer de escrita serão pequenos e podemos ignorá-los. Se um sistema não atendesse a esse critério, ele não seria bem projetado; ao contrário, o projetista deveria ter usado um buffer de escrita mais profundo ou uma organização write-back.

Os esquemas write-back também possuem stalls potenciais extras surgindo da necessidade de escrever um bloco de cache, novamente na memória, quando o bloco é substituído. Discutiremos mais o assunto na [Seção 5.8](#).

Na maioria das organizações de cache write-through, as penalidades de falha de leitura e escrita são iguais (o tempo para buscar o bloco da memória). Se considerarmos que os stalls do buffer de escrita são insignificantes, podemos combinar as leituras e escritas usando uma única taxa de falhas e a penalidade de falha:

$$\text{Ciclos de clock de stall de memória} = \frac{\text{Acessos à memória}}{\text{Programa}} \times \text{Taxa de falhas} \times \text{Penalidade de falha}$$

Também podemos fatorar isso como

$$\text{Ciclos de clock de stall de memória} = \frac{\text{Instruções}}{\text{Programa}} \times \frac{\text{Falhas}}{\text{Instrução}} \times \text{Penalidade de falha}$$

Vamos considerar um exemplo simples para ajudar a entender o impacto no desempenho da cache sobre o desempenho do processador.

Calculando o desempenho da cache

Exemplo

Suponha que uma taxa de falhas de cache de instruções para um programa seja de 2% e que uma taxa de falhas de cache de dados seja de 4%. Se um processador possui um CPI de 2 sem qualquer stall de memória e a penalidade de falha é de 100 ciclos para todas as falhas, determine o quanto mais rápido um processador executaria com uma cache perfeita que nunca falhasse. Suponha que a frequência de todos os loads e stores seja 36%.

Resposta

O número de ciclos de falha da memória para instruções em termos da contagem de instruções (I) é

$$\text{Ciclos de falha de instrução} = I \times 2\% \times 100 = 2,00 \times I$$

A frequência de todos os loads e stores é de 36%. Logo, podemos encontrar o número de ciclos de falha da memória para referências de dados:

$$\text{Ciclos de falha de dados} = I \times 36\% \times 4\% \times 100 = 1,44 \times I$$

O número total de ciclos de stall da memória é $2,00 I + 1,44 I = 3,44 I$. Isso é mais do que três ciclos de stall da memória por instrução. Portanto, o CPI com stalls da memória é $2 + 3,44 = 5,44$. Como não há mudança alguma na contagem de instruções ou na velocidade de clock, a taxa dos tempos de execução da CPU é

$$\frac{\text{Tempo de CPU com stalls}}{\text{Tempo de CPU com cache perfeita}} = \frac{I \times CPI_{\text{stall}} \times \text{Ciclo de clock}}{I \times CPI_{\text{perfeita}} \times \text{Ciclo de clock}}$$

$$= \frac{CPI_{\text{stall}}}{CPI_{\text{perfeita}}} = \frac{5,44}{2}$$

O desempenho com a cache perfeita é melhor por um fator de $\frac{5,44}{2} = 2,72$.

O que acontece se o processador se tornar mais rápido, mas o sistema de memória não? A quantidade de tempo gasto nos stalls da memória tomará uma fração cada vez maior do tempo de execução; a Lei de Amdahl, que examinamos no [Capítulo 1](#), nos lembra desse fato. Alguns exemplos simples mostram como esse problema pode ser sério. Suponha que aceleremos o computador do exemplo anterior reduzindo seu CPI de 2 para 1 sem mudar a velocidade de clock, o que pode ser feito com um pipeline melhorado. O sistema com falhas de cache, então, teria um CPI de $1 + 3,44 = 4,44$, e o sistema com a cache perfeita seria

$$\frac{4,44}{1} = 4,44 \text{ vezes mais rápido}$$

A quantidade de tempo de execução gasto em stalls da memória teria subido de

$$\frac{3,44}{5,44} = 63\%$$

para

$$\frac{3,44}{4,44} = 77\%$$

Da mesma forma, aumentar a velocidade de clock sem mudar o sistema de memória também aumenta a perda de desempenho devido às falhas de cache.

Os exemplos e equações anteriores consideram que o tempo de acerto não é um fator na determinação do desempenho da cache. Claramente, se o tempo de acerto aumentar, o tempo total para acessar uma palavra do sistema de memória crescerá, possivelmente causando um aumento no tempo de ciclo do processador. Embora vejamos em breve outros exemplos do que pode aumentar o tempo de acerto, um exemplo é aumentar o tamanho da cache. Uma cache maior pode ter um tempo de acesso maior, exatamente como se sua mesa na biblioteca fosse muito grande (digamos, 3 m²): você levaria mais tempo para localizar um livro sobre a mesa. Um aumento no tempo de acerto provavelmente acrescenta outro estágio ao pipeline, já que podem ser necessários vários ciclos para um acerto de cache. Embora seja mais complexo calcular o impacto de desempenho de um pipeline mais profundo, em algum ponto, o aumento no tempo de acerto para uma cache maior pode dominar a melhoria na taxa de acertos, levando a uma redução no desempenho do processador.

A fim de capturar o fato de que o tempo de acesso a dados para acertos e falhas afeta o desempenho, os projetistas, às vezes, usam *tempo médio de acesso à memória* (TMAM) como um modo de examinar os projetos de cache alternativos. O tempo médio de acesso à memória é o tempo médio para acessar a memória, considerando acertos, falhas e a frequência dos diferentes acessos; ele é igual ao seguinte:

$$\text{TMAM} = \text{Tempo para um acerto} + \text{Taxa de falhas} \times \text{Penalidade de falha}$$

Calculando o tempo médio de acesso à memória

Exemplo

Ache o TMAM para um processador com tempo de clock de 1 ns, uma penalidade de falha de 20 ciclos de clock, uma taxa de falhas de 0,05 falhas por instrução e um tempo de acesso a cache (incluindo detecção de acerto) de 1 ciclo de clock. Suponha que as penalidades de perda de leitura e escrita

sejam iguais e ignore outros stalls de escrita.

Resposta

O tempo médio de acesso à memória por instrução é

$$\begin{aligned} \text{TMAM} &= \text{Tempo para um acerto} + \text{Taxa de falhas} \times \text{Penalidade de falha} \\ &= 1 + 0,05 \times 20 \\ &= 2 \text{ ciclos de clock} \end{aligned}$$

ou 2 ns.

A próxima subseção discute organizações de cache alternativas que diminuem a taxa de falhas mas podem, algumas vezes, aumentar o tempo de acerto; outros exemplos aparecem em Falácia e armadilhas ([Seção 5.13](#)).

Reduzindo as falhas de cache com um posicionamento de blocos mais flexível

Até agora, quando colocamos um bloco na cache, usamos um esquema de posicionamento simples: um bloco só pode entrar exatamente em um local na cache. Como já dissemos, esse esquema é chamado de *mapeamento direto* porque qualquer endereço de bloco na memória é diretamente mapeado para um único local, no nível superior da hierarquia. Existe, na verdade, toda uma faixa de esquemas para posicionamento de blocos. Em um extremo está o mapeamento direto, em que um bloco só pode ser posicionado exatamente em um local.

No outro extremo está um esquema em que um bloco pode ser posicionado em *qualquer* local na cache. Esse esquema é chamado de **totalmente associativo** porque um bloco na memória pode ser associado com qualquer entrada da cache. Para encontrar um determinado bloco em uma cache totalmente associativa, todas as entradas da cache precisam ser pesquisadas, pois um bloco pode estar posicionado em qualquer uma delas. Para tornar a pesquisa possível, ela é feita em paralelo com um comparador associado a cada entrada da cache. Esses comparadores aumentam muito o custo do hardware, na prática, tornando o posicionamento totalmente associativo, viável apenas para caches com pequenos números de blocos.

cache totalmente associativa

Uma estrutura de cache em que um bloco pode ser posicionado em qualquer local da cache.

A faixa intermediária de projetos entre a cache diretamente mapeada e a cache totalmente associativa é chamada de **associativa por conjunto**. Em uma cache associativa por conjunto, existe um número fixo de locais (pelo menos dois) onde cada bloco pode ser colocado; uma cache associativa por conjunto com n locais para um bloco é chamado de cache associativa por conjunto de n vias. Uma cache associativa por conjunto de n vias consiste em diversos conjuntos, cada um consistindo em n blocos. Cada bloco na memória é mapeado para um *conjunto* único na cache, determinado pelo campo índice, e um bloco pode ser colocado em *qualquer* elemento desse conjunto. Portanto, um posicionamento associativo por conjunto combina o posicionamento diretamente mapeado e o posicionamento totalmente associativo: um bloco é diretamente mapeado para um conjunto e, então, uma correspondência é pesquisada em todos os blocos no conjunto. Por exemplo, a [Figura 5.14](#) mostra onde o bloco 12 pode ser posicionado em uma cache com oito blocos no total, conforme as três políticas de posicionamento de bloco.



FIGURA 5.14 O local de um bloco de memória cujo endereço é 12 em uma cache com 8 blocos varia para posicionamento diretamente mapeado, associativo por conjunto e totalmente associativo.

No posicionamento diretamente mapeado, há apenas um bloco de cache em que o bloco de memória 12 pode ser encontrado, e

esse bloco é dado por $(12 \bmod 8) = 4$. Em uma cache associativa por conjunto de duas vias, haveria quatro conjuntos e o bloco de memória 12 precisa estar no conjunto $(12 \bmod 4) = 0$; o bloco de memória pode estar em qualquer elemento do conjunto. Em um posicionamento totalmente associativo, o bloco de memória para o endereço de bloco 12 pode aparecer em qualquer um dos oito blocos de cache.

cache associativa por conjunto

Uma cache que possui um número fixo de locais (no mínimo dois) onde cada bloco pode ser colocado.

Lembre-se de que, em uma cache diretamente mapeada, a posição de um bloco de memória é determinada por

$(\text{Número do bloco}) \bmod (\text{Número de blocos na cache})$

Em uma cache associativa por conjunto, o conjunto contendo um bloco de memória é determinado por

$(\text{Número do bloco}) \bmod (\text{Número de conjuntos na cache})$

Como o bloco pode ser colocado em qualquer elemento do conjunto, *todas as tags de todos os elementos do conjunto* precisam ser pesquisadas. Em uma cache totalmente associativa, o bloco pode entrar em qualquer lugar e todas as tags de todos os blocos na cache precisam ser pesquisadas.

Podemos pensar em cada estratégia de posicionamento de bloco como uma variação da associatividade por conjunto. A [Figura 5.15](#) mostra as possíveis estruturas de associatividade para uma cache de oito blocos. Uma cache diretamente mapeada é simplesmente uma cache associativa por conjunto de uma via: cada entrada de cache contém um bloco, e cada conjunto possui um elemento. Uma cache totalmente associativa com m entradas é simplesmente uma cache associativa por conjunto de m vias; ele tem um conjunto com m blocos, e uma entrada pode residir em qualquer bloco dentro desse conjunto.

Associativo por conjunto de uma via (diretamente mapeada)

Bloco	Tag	Dados
0		
1		
2		
3		
4		
5		
6		
7		

Associativo por conjunto de duas vias

Conjunto	Tag	Dados	Tag	Dados
0				
1				
2				
3				

Associativo por conjunto de quatro vias

Conjunto	Tag	Dados	Tag	Dados	Tag	Dados	Tag	Dados
0								
1								

Associativo por conjunto de oito vias

Tag	Dados														

FIGURA 5.15 Uma cache de oito blocos configurada como diretamente mapeada, associativa por conjunto de duas vias, associativa por conjunto de quatro vias e totalmente associativa.

O tamanho total da cache em blocos é igual ao número de conjuntos multiplicado pela associatividade. Portanto, para uma cache de tamanho fixo, aumentar a associatividade diminui o número de conjuntos enquanto aumenta o número de elementos por conjunto. Com oito blocos, uma cache associativa por conjunto de oito vias é igual a uma cache totalmente associativa.

A vantagem de aumentar o grau da associatividade é que ela normalmente diminui a taxa de falhas, como mostra o próximo exemplo. A principal desvantagem, que veremos com mais detalhes em breve, é um potencial aumento no tempo de acerto.

Falhas e associatividade nas caches

Exemplo

Considere três caches pequenas, cada uma consistindo em quatro blocos de uma palavra cada. Uma cache é totalmente associativa, uma segunda cache é associativa por conjunto de duas vias, e a terceira cache é diretamente mapeada. Encontre o número de falhas para cada organização de cache, dada a seguinte sequência de endereços de bloco: 0, 8, 0, 6 e 8.

Resposta

O caso diretamente mapeado é mais fácil. Primeiro, vamos determinar para qual bloco de cache cada endereço de bloco é mapeado:

Endereço do bloco	Bloco de cache
0	(0 módulo 4) = 0
6	(6 módulo 4) = 2
8	(8 módulo 4) = 0

Agora podemos preencher o conteúdo da cache após cada referência, usando uma entrada em branco para indicar que o bloco é inválido, texto em negrito para mostrar uma nova entrada incluída na cache para a referência associada e um texto normal para mostrar uma entrada existente na cache:

Endereço do bloco de memória associado	Acerto ou falha	Conteúdo dos blocos de cache após referência			
		0	1	2	3
0	falha	Memória[0]			
8	falha	Memória[8]			
0	falha	Memória[0]			
6	falha	Memória[0]		Memória[6]	
8	falha	Memória[8]		Memória[6]	

A cache diretamente mapeada gera cinco falhas para os cinco acessos.

A cache associativa por conjunto possui dois conjuntos (com índices 0 e 1) com dois elementos por conjunto. Primeiro, vamos determinar para qual conjunto cada endereço de bloco é mapeado:

Endereço do bloco	Bloco de cache
0	(0 módulo 2) = 0
6	(6 módulo 2) = 0
8	(8 módulo 2) = 0

Já que temos uma escolha de qual entrada em um conjunto substituir em uma falha, precisamos de uma regra de substituição. As caches associativas por conjunto normalmente substituem o bloco usado menos recentemente dentro de um conjunto; ou seja, o bloco usado há mais tempo é substituído. (Discutiremos outras regras de substituição mais detalhadamente em breve.) Usando esta regra de substituição, o conteúdo da cache associativa por conjunto após cada referência se parece com o seguinte:

Endereço do bloco de memória associado	Acerto ou falha	Conteúdo dos blocos de cache após referência			
		Conjunto 0	Conjunto 0	Conjunto 1	Conjunto 1
0	falha	Memória[0] 1			
8	falha	Memória[0]	Memória[8] 1		
0	acerto	Memória[0]	Memória[8]		
6	falha	Memória[0]	Memória[6] 1		
8	falha	Memória[8] 1	Memória[6]		

Observe que, quando o bloco 6 é referenciado, ele substitui o bloco 8, já que o bloco 8 foi referenciado menos recentemente do que o bloco 0. A cache associativa por conjunto de duas vias possui quatro falhas, uma a menos do que a cache diretamente mapeada.

A cache totalmente associativa possui quatro blocos de cache (em um único conjunto); qualquer bloco de memória pode ser armazenado em qualquer bloco de cache. A cache totalmente associativa possui o melhor desempenho, com apenas três falhas:

Endereço do bloco de memória associado	Acerto ou falha	Conteúdo dos blocos de cache após referência			
		Bloco 0	Bloco 1	Bloco 2	Bloco 3
0	falha	Memória[0]			
8	falha	Memória[0]	Memória[8]		
0	acerto	Memória[0]	Memória[8]		
6	falha	Memória[0]	Memória[8]	Memória[6]	
8	acerto	Memória[0]	Memória[8]	Memória[6]	

Para essa série de referências, três falhas é o melhor que podemos fazer porque três endereços de bloco únicos são acessados. Repare que se

tivéssemos oito blocos na cache, não haveria qualquer substituição na cache associativa por conjunto de duas vias (confira isso você mesmo), e ele teria o mesmo número de falhas da cache totalmente associativa. Da mesma forma, se tivéssemos 16 blocos, todas as três caches teriam o mesmo número de falhas. Até mesmo esse exemplo trivial mostra que o tamanho da cache e a associatividade não são independentes para a determinação do desempenho da cache.

Quanta redução na taxa de falhas é obtida pela associatividade? A [Figura 5.16](#) mostra a melhoria para uma cache de dados de 64 KiB com um bloco de 16 palavras e mostra a associatividade mudando do mapeamento direto para oito vias. Passar da associatividade de uma via para duas vias diminui a taxa de falhas em, aproximadamente 15%, mas há pouca melhora adicional em passar para uma associatividade mais alta.

Associatividade	Taxa de falhas de dados
1	10,3%
2	8,6%
4	8,3%
8	8,1%

FIGURA 5.16 As taxas de falhas da cache de dados para uma organização como o processador Intrinsity FastMATH para benchmarks SPEC CPU2000 com associatividade variando de uma via a oito vias.

Esses resultados para dez programas SPEC CPU2000 são de Hennessy et al. (2003).

Localizando um bloco na cache

Agora, vamos considerar a tarefa de encontrar um bloco em uma cache que é associativa por conjunto. Assim como em uma cache diretamente mapeada, cada bloco em uma cache associativa por conjunto inclui uma tag de endereço que fornece o endereço do bloco. A tag de cada bloco de cache dentro do conjunto apropriado é verificada para ver se corresponde ao endereço de bloco vindo do processador. A [Figura 5.17](#) mostra como o endereço é decomposto. O valor de índice é usado para selecionar o conjunto contendo o endereço de interesse, e as

tags de todos os blocos no conjunto precisam ser pesquisadas. Como a velocidade é a essência da pesquisa, todas as tags no conjunto selecionado são pesquisadas em paralelo. Assim como em uma cache totalmente associativa, uma pesquisa sequencial tornaria o tempo de acerto de uma cache associativa por conjunto muito lento.

Tag	Índice	Offset de bloco
-----	--------	-----------------

FIGURA 5.17 As três partes de um endereço em uma cache associativa por conjunto ou diretamente mapeada.

O índice é usado para selecionar o conjunto e, depois, a tag é usada para escolher o bloco por comparação com os blocos no conjunto selecionado. O offset do bloco é o endereço dos dados desejados dentro do bloco.

Se o tamanho de cache total for mantido igual, aumentar a associatividade aumenta o número de blocos por conjunto, que é o número de comparações simultâneas necessárias para realizar a pesquisa em paralelo: cada aumento por um fator de dois na associatividade dobra o número de blocos por conjunto e divide por dois o número de conjuntos. Assim, cada aumento pelo dobro na associatividade diminui o tamanho do índice em 1 bit e aumenta o tamanho da tag em 1 bit. Em uma cache totalmente associativa, existe apenas um conjunto, e todos os blocos precisam ser verificados em paralelo. Portanto, não há qualquer índice, e o endereço inteiro, excluindo o offset do bloco, é comparado com a tag de cada bloco. Em outras palavras, a cache inteira é pesquisada sem qualquer indexação.

Em uma cache diretamente mapeada, apenas um único comparador é necessário, pois a entrada pode estar apenas em um bloco, e acessamos a cache por meio da indexação. A [Figura 5.18](#) mostra que em uma cache associativa por conjunto de quatro vias, quatro comparadores são necessários, juntamente com um multiplexador de 4 para 1, a fim de escolher entre os quatro números possíveis do conjunto selecionado. O acesso de cache consiste em indexar o conjunto apropriado e, depois, pesquisar as tags do conjunto. Os custos de uma cache associativa são os comparadores extras e qualquer atraso gerado pela necessidade de comparar e selecionar entre os elementos do conjunto.

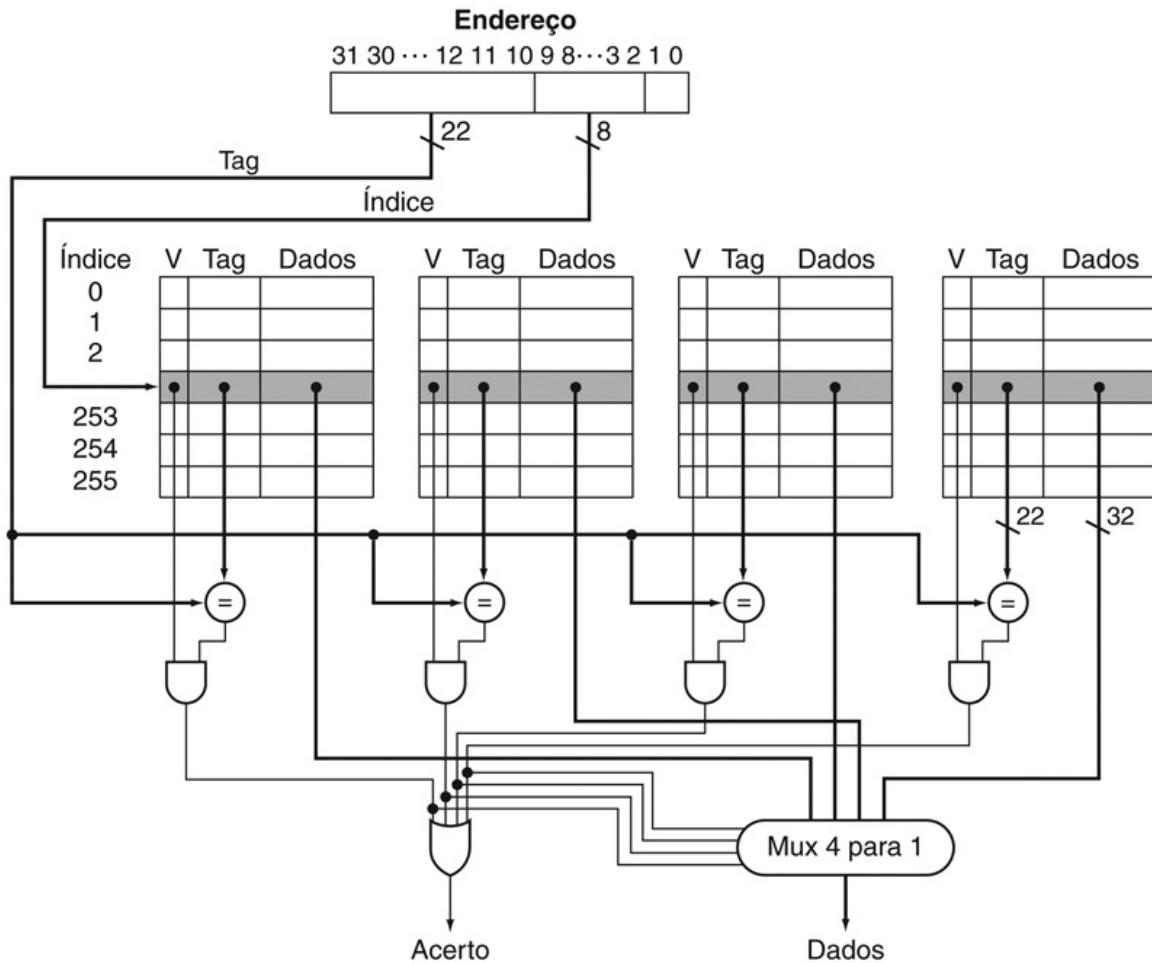


FIGURA 5.18 A implementação de uma cache associativa por conjunto de quatro vias exige quatro comparadores e um multiplexador (mux) de 4 para 1.

Os comparadores determinam qual elemento do conjunto selecionado (se houver) corresponde à tag. A saída dos comparadores é usada para selecionar os dados de um dos quatro blocos do conjunto indexado, usando um multiplexador com um sinal de seleção decodificado. Em algumas implementações, a saída permite que sinais nas partes de dados das RAMs de cache possam ser usados para selecionar a entrada no conjunto que controla a saída. A saída permite que o sinal venha dos comparadores, fazendo com que o elemento correspondente controle as saídas de dados. Essa organização elimina a necessidade do multiplexador.

A escolha entre mapeamento direto, associativo por conjunto ou totalmente associativo em qualquer hierarquia de memória dependerá do custo de uma falha em comparação com o custo da implementação da associatividade, ambos em

tempo e em hardware extra.

Detalhamento

Uma *Content Addressable Memory* (CAM) é um circuito que combina comparação e armazenamento em um único dispositivo. Em vez de fornecer um endereço e ler uma palavra como uma RAM, você fornece os dados e a CAM verifica se tem uma cópia e retorna o índice da linha correspondente. As CAMs significam que os projetistas de cache podem proporcionar a implementação de uma associatividade por conjunto muito mais alta do que se tivessem de construir o hardware a partir de SRAMs e comparadores. Em 2013, o maior tamanho e potência da CAM geralmente levam a uma associatividade por conjunto em duas vias e quatro vias sendo construída a partir de SRAMs padrão e comparadores, com oito vias em diante sendo construídas usando CAMs.

Escolhendo que bloco substituir

Quando ocorre uma falha em uma cache diretamente mapeada, o bloco requisitado só pode entrar em exatamente uma posição, e o bloco ocupando essa posição precisa ser substituído. Em uma cache associativa, temos uma escolha de onde colocar o bloco requisitado e, portanto, uma escolha de qual bloco substituir. Em uma cache totalmente associativa, todos os blocos são candidatos à substituição. Em uma cache associativa por conjunto, precisamos escolher entre os blocos do conjunto selecionado.

O esquema mais comum é o **LRU (Least Recently Used — usado menos recentemente)**, que usamos no exemplo anterior. Em um esquema LRU, o bloco substituído é aquele que não foi usado há mais tempo. O exemplo associativo por conjunto demonstrado anteriormente neste capítulo utiliza LRU, que é o motivo pelo qual substituímos Memória(0) ao invés de Memória(6).

LRU (Least Recently Used — usado menos recentemente)

Um esquema de substituição em que o bloco substituído é aquele que não foi usado há mais tempo.

A substituição LRU é implementada monitorando quando cada elemento em um conjunto foi usado em relação aos outros elementos no conjunto. Para uma cache associativa por conjunto de duas vias, o controle de quando os dois elementos foram usados pode ser implementado mantendo um único bit em cada conjunto e definindo o bit para indicar um elemento sempre que este é referenciado. Conforme a associatividade aumenta, a implementação do LRU se torna mais difícil; na [Seção 5.8](#), veremos um esquema alternativo para substituição.

Tamanho das tags versus associatividade do conjunto

Exemplo

O acréscimo da associatividade requer mais comparadores e mais bits de tag por bloco de cache. Considerando uma cache de 4096 blocos, um tamanho de bloco de quatro palavras e um endereço de 32 bits, encontre o número total de conjuntos e o número total de bits de tag para caches que são diretamente mapeadas, associativas por conjunto de duas e quatro vias, totalmente associativas.

Resposta

Como existem 16 ($=2^4$) bytes por bloco, um endereço de 32 bits produz $32 - 4 = 28$ bits para serem usados para índice e tag. A cache diretamente mapeada possui um mesmo número de conjuntos e blocos e, portanto, 12 bits de índice, já que $\log_2(4096) = 12$; logo, o número total é $(28 - 12) \times 4096 = 16 \times 4096 = 66$ K bits de tag.

Cada grau de associatividade diminui o número de conjuntos por um fator de dois e, portanto, diminui o número de bits usados para indexar a cache por um e aumenta o número de bits na tag por um. Consequentemente, para uma cache associativa por conjunto de duas vias, existem 2048 conjuntos, e o número total é $(28 - 11) \times 2 \times 2048 = 34 \times 2048 = 70$ K bits de tag. Para uma cache associativa por conjunto de quatro vias, o número total de conjuntos é 1024, e o número total é $(28 - 10) \times 4 \times 1024 = 72 \times 1024 = 74$ K bits de tag.

Para uma cache totalmente associativa, há apenas um conjunto com 4096 blocos, e a tag possui 28 bits, produzindo um total de $28 \times 4096 \times 1 = 115$ K bits de tag.

Reduzindo a penalidade de falha usando caches multiníveis

Todos os computadores modernos fazem uso de caches. Para diminuir a diferença entre as rápidas velocidades de clock dos processadores modernos e o tempo relativamente longo necessário para acessar as DRAMs, muitos microprocessadores suportam um nível adicional de cache. Essa cache de segundo nível normalmente está no mesmo chip e é acessada sempre que ocorre uma falha na cache primária. Se a cache de segundo nível contiver os dados desejados, a penalidade de falha para a cache de primeiro nível será o tempo de acesso à cache de segundo nível, que será muito menor do que o tempo de acesso à memória principal. Se nem a cache primária nem a secundária contiverem os dados, um acesso à memória principal será necessário, e uma penalidade de falha maior será observada.

Em que grau é significante a melhora de desempenho pelo uso de uma cache secundária? O próximo exemplo nos mostra.

Desempenho das caches multinível

Exemplo

Suponha que tenhamos um processador com um CPI básico de 1,0, considerando que todas as referências acertem na cache primária e uma velocidade de clock de 4GHz. Considere um tempo de acesso à memória principal de 100 ns, incluindo todo o tratamento de falhas. Suponha que a taxa de falhas por instrução na cache primária seja de 2%. O quanto mais rápido será o processador se acrescentarmos uma cache secundária que tenha um tempo de acesso de 5 ns para um acerto ou uma falha e que seja grande o suficiente de modo a reduzir a taxa de falhas para a memória principal para 0,5%?

Resposta

A penalidade de falha para a memória principal é

$$\frac{100 \text{ ns}}{0,25 \frac{\text{ns}}{\text{ciclo de clock}}} = 400 \text{ ciclos de clock}$$

O CPI efetivo com um nível de cache é dado por

$$\text{CPI total} = \text{CPI básico} + \text{Ciclos de stall de memória por instrução}$$

Para o processador com um nível de caching,

$$\text{CPI total} = 1,0 + \text{Ciclos de stall de memória por instrução} = 1,0 + 2\% \times 400 = 9$$

Com dois níveis de cache, uma falha na cache primária (ou de primeiro nível) pode ser preenchida pela cache secundária ou pela memória principal. A penalidade da falha para um acesso à cache de segundo nível é

$$\frac{5 \text{ ns}}{0,25 \frac{\text{ns}}{\text{ciclo de clock}}} = 20 \text{ ciclos de clock}$$

Se a falha for preenchida na cache secundária, essa será toda a penalidade de falha. Se a falha precisar ir à memória principal, então, a penalidade de falha total será a soma do tempo de acesso à cache secundária e do tempo de acesso à memória principal.

Logo, para uma cache de dois níveis, o CPI total é a soma dos ciclos de stall dos dois níveis de cache e o CPI básico:

$$\begin{aligned} \text{CPI total} &= 1 + \text{Stalls primários por instrução} + \text{Stalls secundários por instrução} \\ &= 1 + 2\% \times 20 + 0,5\% \times 400 = 1 + 0,4 + 2,0 = 3,4 \end{aligned}$$

Portanto, o processador com a cache secundária é mais rápido por um fator

de

$$\frac{9,0}{3,4} = 2,6$$

Como alternativa, poderíamos ter calculado os ciclos de stall somando os ciclos de stall das referências que acertam na cache secundária ($(2\% - 0,5\%) \times 20 = 0,3$) e as referências que vão à memória principal, que precisam incluir o custo para acessar a cache secundária, bem como o tempo de acesso à memória principal ($0,5\% \times (20 + 400) = 2,1$). A soma, $1,0 + 0,3 + 2,1$, é novamente 3,4.

As considerações de projeto para uma cache primária e secundária são significativamente diferentes porque a presença da outra cache muda a melhor escolha em comparação com uma cache de nível único. Em especial, uma estrutura de cache de dois níveis permite que a cache primária se concentre em minimizar o tempo de acerto para produzir um ciclo de clock mais curto, enquanto permite que a cache secundária focalize a taxa de falhas no sentido de reduzir a penalidade dos longos tempos de acesso à memória.

O efeito dessas mudanças nas duas caches pode ser visto comparando cada cache com o projeto otimizado para um nível único de cache. Em comparação com uma cache de nível único, a cache primária de uma **cache multinível** normalmente é menor. Além disso, a cache primária frequentemente usa um tamanho de bloco menor, para se adequar ao tamanho de cache menor e à penalidade de falha reduzida. Em comparação, a cache secundária normalmente será maior do que em uma cache de nível único, já que o tempo de acesso da cache secundária é menos importante. Com um tamanho total maior, a cache secundária pode usar um tamanho de bloco maior do que o apropriado com uma cache de nível único. Ela constantemente utiliza uma associatividade maior que a cache primária, dado o foco da redução de taxas de falha.

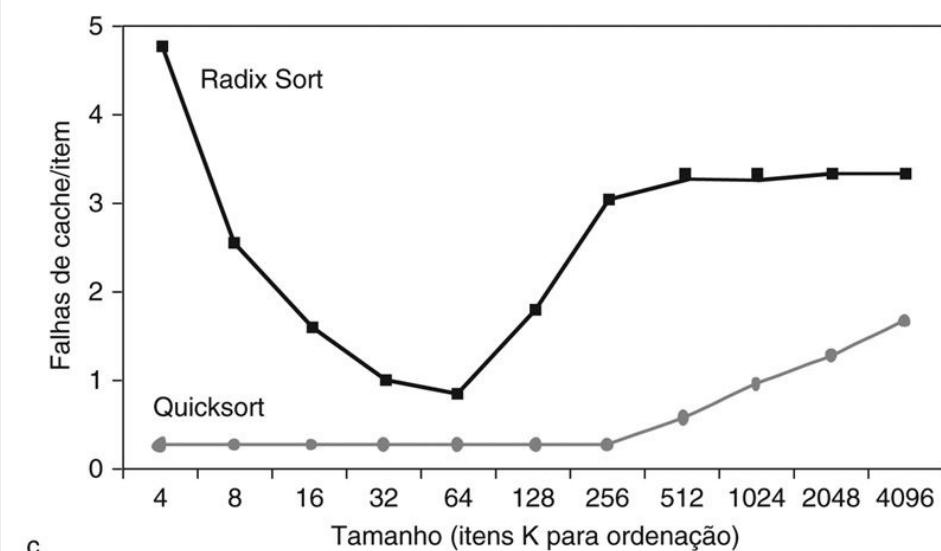
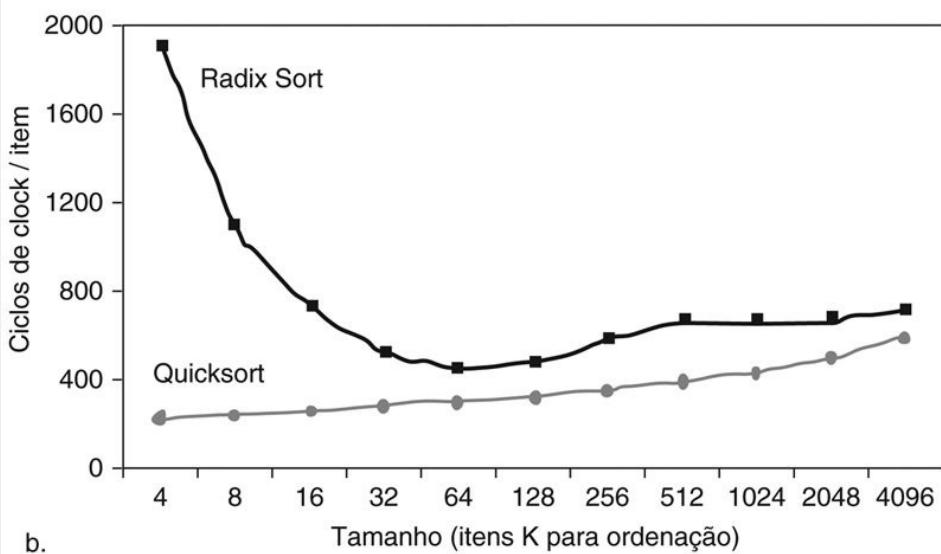
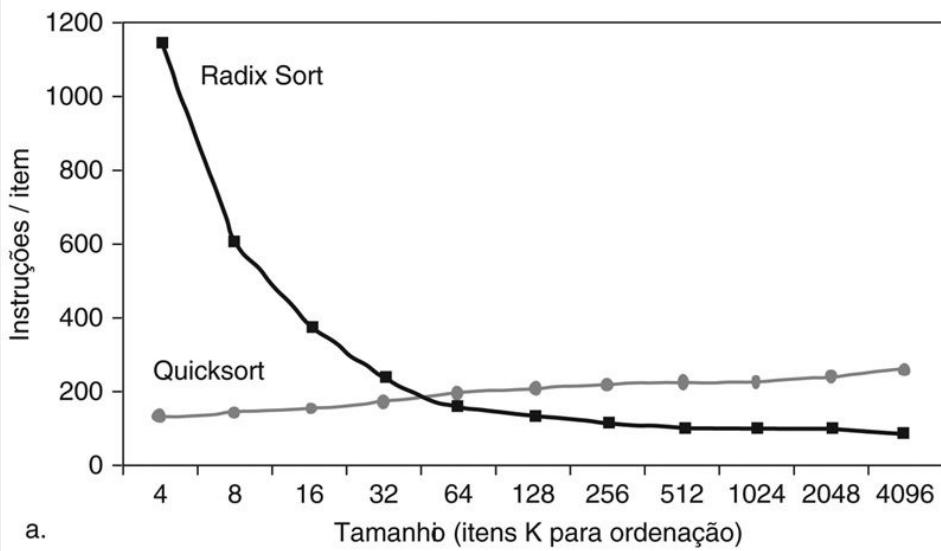
cache multinível

Uma hierarquia de memória com múltiplos níveis de cache, em vez de apenas uma cache e a memória principal.

Entendendo o desempenho dos programas

A classificação tem sido exaustivamente analisada para se encontrar algoritmos melhores: Bubble Sort, Quicksort e assim por diante. A Figura 5.19(a) mostra as instruções executadas por item pesquisado pelo Radix Sort em comparação com o Quicksort. Decididamente, para arrays grandes, o Radix Sort possui uma vantagem algorítmica sobre o Quicksort em termos do número de operações. A Figura 5.19(b) mostra o tempo por chave, em vez das instruções executadas. Podemos ver que as linhas começam na mesma trajetória da Figura 5.19(a), mas, então, a linha do Radix Sort diverge conforme os dados a serem ordenados aumentam. O que está ocorrendo? A Figura 5.19(c) responde olhando as falhas de cache por item ordenado: o Quicksort possui muito menos falhas por item a ser ordenado.

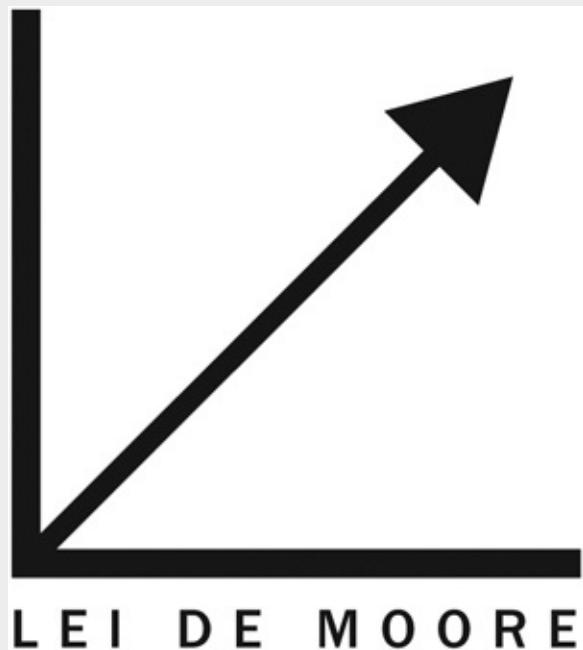
Infelizmente, a análise algorítmica padrão ignora o impacto da hierarquia de memória. À medida que velocidades de clock mais altas e a **Lei de Moore** permitem aos arquitetos compactarem todo o desempenho de um fluxo de instruções, um uso correto da hierarquia de memória é fundamental para que se obtenha um alto desempenho. Como dissemos na introdução, entender o comportamento da hierarquia de memória é vital para compreender o desempenho dos programas nos computadores atuais.



c.

FIGURA 5.19 Comparando o Quicksort e o Radix Sort por (a) instruções executadas por item ordenado, (b) tempo por item ordenado e (c) falhas de cache por item ordenado.

Esses dados são de um artigo de LaMarca e Ladner [1996]. Devido a esses resultados, foram criadas novas versões do Radix Sort que levam a hierarquia de memória em consideração, para readquirir suas vantagens logarítmicas (Seção 5.13). A ideia básica das otimizações de cache é usar todos os dados em um bloco repetidamente antes de serem substituídos em uma falha.



Otimização de software por bloqueio

Dada a importância da hierarquia de memória para o desempenho do programa, não é surpresa o fato de que foram criadas muitas otimizações de software que podem melhorar drasticamente o desempenho reutilizando dados dentro da cache e, portanto, ocasionar menos taxas de perda, devido à melhor localidade temporal.

Ao lidarmos com arrays, podemos obter um bom desempenho do sistema de memória se armazenarmos o array na memória de modo que os acessos ao array

sejam sequenciais na memória. Mas suponha que estejamos lidando com múltiplos arrays, com alguns arrays acessados por linhas e alguns por colunas. Armazenar os arrays linha por linha (*ordem principal de linha*) ou coluna por coluna (*ordem principal de coluna*) não resolve o problema, pois linhas e colunas são usadas em cada iteração do loop.

Ao invés de operar sobre linhas ou colunas inteiras de um array, os algoritmos *bloqueados* operam sobre submatrizes ou *blocos*. O objetivo é maximizar os acessos aos dados carregados na cache antes que esses dados sejam substituídos; ou seja, melhorar a localidade temporal para reduzir as falhas de cache.

Por exemplo, os loops mais internos do DGEMM (linhas de 4 a 9 da [Figura 3.21](#), no [Capítulo 3](#)) são

```
for (int j = 0; j < n; ++j)
{
    double cij = C[i+j*n]; /* cij = C[i][j] */
    for( int k = 0; k < n; k++ )
        cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
    C[i+j*n] = cij; /* C[i][j] = cij */
}
```

Os dois loops lêem todos os elementos N por N de B , lêem os mesmos N elementos em uma linha de A repetidamente, e escrevem o que corresponde a uma linha de N elementos de C . Os comentários facilitam a identificação das linhas e colunas das matrizes. A [Figura 5.20](#) contém um instantâneo dos acessos aos três arrays. Um tom cinza escuro indica um acesso recente, um tom cinza claro indica um acesso mais antigo, e branco significa ainda não acessado.

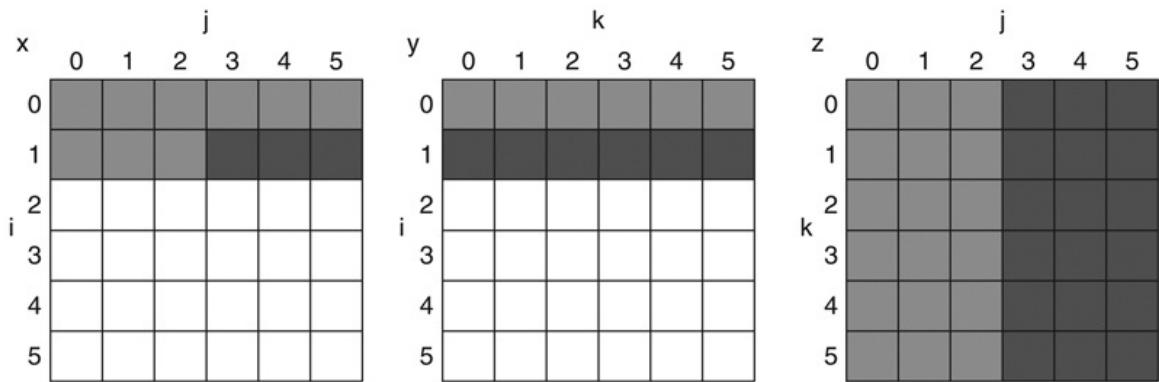


FIGURA 5.20 Um instantâneo dos três arrays c , A e B quando $n = 6$ e $i = 1$.

A idade dos acessos aos elementos do array é indicada pelo tom: branco significa ainda não tocado, cinza claro significa acessos mais antigos, e cinza escuro significa acessos mais recentes. Em comparação com a [Figura 5.22](#), os elementos de A e B são lidos repetidamente para calcular novos elementos de x . As variáveis i , j e k aparecem ao longo das linhas ou colunas usadas para acessar os arrays.

O número de perdas de capacidade depende claramente de N e do tamanho da cache. Se ele puder manter todas as três matrizes N por N , então tudo está bem, desde que não existam conflitos de cache. Propositalmente, escolhemos o tamanho de matriz como 32 por 32 no DGEMM para os Capítulos 3 e 4, de modo que esse seria o caso. Cada matriz tem $32 \times 32 = 1024$ elementos e cada elemento tem 8 bytes, de modo que as três matrizes ocupam 24 KiB, que são suficientes para caber na cache de dados de 32 KiB do Intel Core i7 (Sandy Bridge).

Se a cache puder manter uma matriz N por N e uma linha de N , então pelo menos a “iésima” linha de A e o array B podem permanecer na cache. Menos do que isso e perdas poderão ocorrer para B e C . No pior dos casos, haveria $2N^3 + N^2$ palavras de memória acessadas para N^3 operações.

Para garantir que os elementos sendo acessados podem caber na cache, o código original é mudado para calcular em uma submatriz. Logo, basicamente chamamos a versão de DGEMM da [Figura 4.80](#), no [Capítulo 4](#), repetidamente nas matrizes de tamanho `BLOCKSIZE` por `BLOCKSIZE`. `BLOCKSIZE` é chamado de *fator de bloqueio*.

A [Figura 5.21](#) mostra a versão de bloqueio do DGEMM. A função `do_block` é o DGEMM da [Figura 3.21](#) com três novos parâmetros, `si`, `sj` e `sk` para especificar a posição inicial de cada submatriz de A , B e C . O otimizador `gcc`

remove qualquer overhead de chamada de função, colocando a função “em linha”; ou seja, ele insere o código diretamente, para evitar as instruções convencionais de passagem de parâmetros e manutenção do endereço de retorno.

```
1 #define BLOCKSIZE 32
2 void do_block (int n, int si, int sj, int sk, double *A, double
3 *B, double *C)
4 {
5     for (int i = si; i < si+BLOCKSIZE; ++i)
6         for (int j = sj; j < sj+BLOCKSIZE; ++j)
7         {
8             double cij = C[i+j*n];/* cij = C[i][j] */
9             for( int k = sk; k < sk+BLOCKSIZE; k++ )
10                 cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
11             C[i+j*n] = cij; /* C[i][j] = cij */
12         }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16     for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17         for ( int si = 0; si < n; si += BLOCKSIZE )
18             for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19                 do_block(n, si, sj, sk, A, B, C);
20 }
```

FIGURA 5.21 Versão bloqueada por cache do DGEMM da Figura 3.21.

Suponha que *c* seja inicializado em zero. A função *do_block* é basicamente o DGEMM do [Capítulo 3](#) com alguns novos parâmetros para especificar as posições iniciais das submatrizes de *BLOCKSIZE*. O otimizador *gcc* pode remover as instruções de overhead de função colocando a função *do_block* em linha.

A [Figura 5.22](#) ilustra os acessos aos três arrays usando o bloqueio. Vendo apenas as perdas de capacidade, o número total de palavras da memória acessadas é $2N^3/BLOCKSIZE + N^2$. Esse total é uma melhoria por um fator de *BLOCKSIZE*. Logo, o bloqueio explora uma combinação de localidade espacial e temporal, pois *A* se beneficia com a localidade espacial e *B* se beneficia com a localidade temporal.

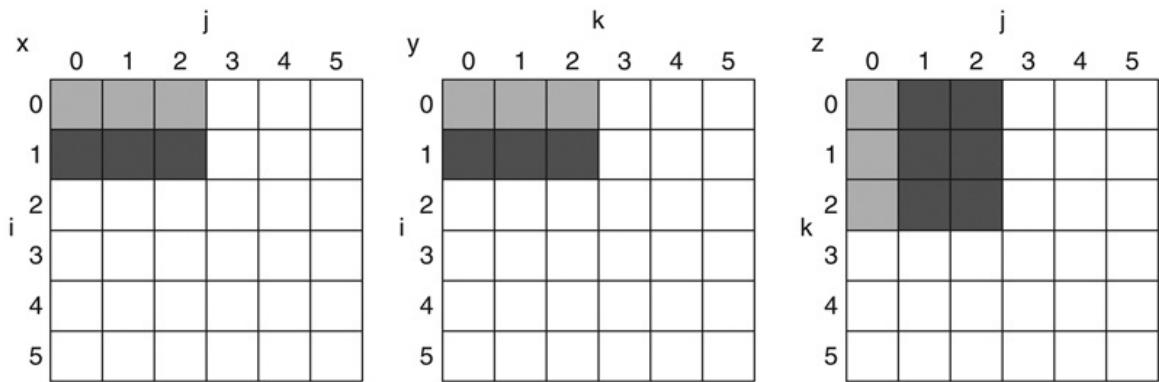


FIGURA 5.22 A idade dos acessos aos arrays c , A e B

quando $BLOCKSIZE = 3$.

Observe, em comparação com a [Figura 5.20](#), o número menor de elementos acessados.

Embora tenhamos visado reduzir as perdas de cache, o bloqueio também pode ser usado para ajudar na alocação de registrador. Apanhando um pequeno tamanho de bloqueio, de modo que o bloco possa ser mantido nos registradores, podemos minimizar o número de loads e stores no programa, o que também melhora o desempenho.

A [Figura 5.23](#) mostra o impacto do bloqueio de cache sobre o desempenho do DGEMM não otimizado à medida que aumentamos o tamanho da matriz além do ponto em que todas as três matrizes cabem na cache. O desempenho não otimizado é dividido ao meio para a matriz maior. A versão com bloqueio de cache é menos de 10% mais lenta, mesmo em matrizes de 960×960 , ou 900 vezes maiores que as matrizes de 32×32 dos Capítulos 3 e 4.

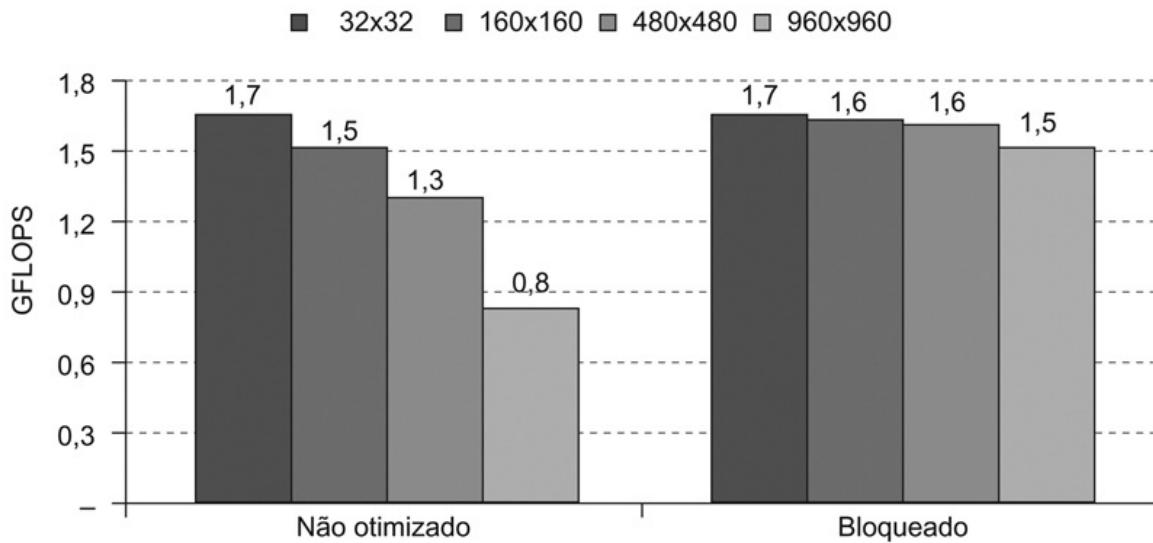


FIGURA 5.23 Desempenho do DGEMM não otimizado (Figura 3.21) contra o DGEMM bloqueado por cache (Figura 5.21) enquanto a dimensão da matriz varia de 32×32 (onde todas as matrizes cabem na cache) a 960×960 .

Detalhamento

Caches multiníveis envolvem diversas complicações. Primeiramente, agora existem vários tipos diferentes de falhas e taxas de falhas correspondentes. No exemplo “Falhas e associatividade nas caches”, anteriormente neste capítulo, vimos a taxa de falhas da cache primária e a **taxa de falhas global** — a fração das referências que falharam em todos os níveis de cache. Há também uma taxa de falhas para a cache secundária, que é a taxa de todas as falhas na cache secundária dividida pelo número de acessos. Essa taxa de falhas é chamada de **taxa de falhas local** da cache secundária. Como a cache primária filtra os acessos, especialmente aqueles com boa localidade espacial e temporal, a taxa de falhas local da cache secundária é muito mais alta do que a taxa de falhas global. No exemplo anterior citado, podemos calcular a taxa de falhas local da cache secundária como: $0,5\%/2\% = 25\%$! Felizmente, a taxa de falhas global determina a frequência com que precisamos acessar a memória principal.

taxa de falhas global

A fração das referências que falham em todos os níveis de uma cache multinível.

taxa de falhas local

A fração das referências a um nível de uma cache que falham; usada em hierarquias multiníveis.

Detalhamento

Com processadores que usam execução fora de ordem (ver Capítulo 4), o desempenho é mais complexo, já que executam instruções durante a penalidade de falha. Em vez da taxa de falhas de instruções e da taxa de falhas de dados, usamos falhas por instrução e esta fórmula:

$$\frac{\text{Ciclos de stall da memória}}{\text{Instrução}} = \frac{\text{Falhas}}{\text{Instrução}} \times (\text{Latência de falha total} - \text{Latência de falha sobreposta})$$

Não há uma maneira geral de calcular a latência de falha sobreposta; portanto, as avaliações das hierarquias de memória para processadores com execução fora de ordem, inevitavelmente, exigem simulações do processador e da hierarquia de memória. Somente vendo a execução do processador durante cada falha é que podemos ver se o processador sofre stall esperando os dados ou simplesmente encontra outro trabalho para fazer. Uma regra é que o processador muitas vezes oculta a penalidade de falha para uma falha de cache L1 que acerta na cache L2, mas raramente oculta uma falha para a cache L2.

Detalhamento

O desafio do desempenho para algoritmos é que a hierarquia de memória varia entre diferentes implementações da mesma arquitetura no tamanho de cache, na associatividade, no tamanho de bloco e no número de caches. Para fazer frente a essa variabilidade, algumas bibliotecas numéricas recentes parametrizam os seus algoritmos e, então, pesquisam o espaço de parâmetros em tempo de execução, de modo a encontrar a melhor combinação para um determinado computador. Essa técnica é chamada de *autotuning*.

Verifique você mesmo

Qual das afirmações a seguir geralmente é verdadeira sobre um projeto com

múltiplos níveis de cache?

1. As caches de primeiro nível são mais focalizadas no tempo de acerto, e as caches de segundo nível se preocupam mais com a taxa de falhas.
2. As caches de primeiro nível são mais focalizadas na taxa de falhas, e as caches de segundo nível se preocupam mais com o tempo de acerto.

Resumo

Nesta seção, nos concentramos em três tópicos: o desempenho da cache, o uso da associatividade para reduzir as taxas de falhas, o uso das hierarquias de cache multinível para reduzir as penalidades de falha e as otimizações de software para melhorar a eficácia das caches.

O sistema de memória tem um efeito significativo sobre o tempo de execução do programa. O número de ciclos de stall de memória depende da taxa de falhas e da penalidade de falha. O desafio, como veremos na [Seção 5.8](#), é reduzir um desses fatores sem afetar significativamente os outros fatores críticos na hierarquia de memória.

Para reduzir a taxa de falhas, examinamos o uso dos esquemas de posicionamento associativos. Esses esquemas podem reduzir a taxa de falhas de uma cache permitindo um posicionamento mais flexível dos blocos dentro dela. Os esquemas totalmente associativos permitem que os blocos sejam posicionados em qualquer lugar, mas também exigem que todos os blocos da cache sejam pesquisados para atender a uma requisição. Os custos mais altos tornam as caches totalmente associativas inviáveis. As caches associativas por conjunto são uma alternativa prática, já que precisamos pesquisar apenas entre os elementos de um único conjunto, escolhido por indexação. As caches associativas por conjunto apresentam taxas de falhas mais altas, porém são mais rápidas de serem acessadas. O grau de associatividade que produz o melhor desempenho depende da tecnologia e dos detalhes da implementação.

Examinamos as caches multiníveis como uma técnica para reduzir a penalidade de falha permitindo que uma cache secundária maior trate das falhas na cache primária. As caches de segundo nível se tornaram comuns quando os projetistas descobriram que o silício limitado e as metas de altas velocidades de clock impedem que as caches primárias se tornem grandes. A cache secundária, que normalmente é 10 ou mais vezes maior do que a cache primária, trata muitos acessos que falham na cache primária. Nesses casos, a penalidade de falha é aquela do tempo de acesso à cache secundária (em geral, menos de dez ciclos de

processador) contra o tempo de acesso à memória (normalmente mais de 100 ciclos de processador). Assim como na associatividade, as negociações de projeto entre o tamanho da cache secundária e seu tempo de acesso dependem de vários aspectos de implementação.

Finalmente, dada a importância da hierarquia de memória no desempenho, vimos como alterar os algoritmos para melhorar o comportamento da cache, sendo que o bloqueio é uma técnica importante quando se lida com grandes arrays.

5.5. Hierarquia de memória estável

Em toda a discussão anterior está implícito que a hierarquia de memória não se esquece. Se fosse rápida mas não confiável, ela não seria muito atraente. Como vimos no [Capítulo 1](#), a grande ideia para a **estabilidade** é a redundância. Nesta seção, primeiro daremos uma passada pelos termos para defini-los assim como as medidas associadas à falha, e depois mostraremos como a redundância pode criar memórias quase inesquecíveis.



E S T A B I L I D A D E

Definição de falha

Começamos com uma suposição de que você possui uma especificação de serviço adequado. Os usuários podem então ver um sistema alternando entre dois estados de serviço entregue com relação à especificação do serviço:

1. *Realização de serviço*, onde o serviço é entregue conforme especificado
2. *Interrupção de serviço*, onde o serviço entregue é diferente do serviço especificado

As transições do estado 1 para o estado 2 são causadas por *fallas* e as

transições do estado 2 para o estado 1 são chamadas de *restaurações*. As falhas podem ser permanentes ou intermitentes. Este último é o caso mais difícil; é mais difícil diagnosticar o problema quando o sistema oscila entre os dois estados. As falhas permanentes são muito mais fáceis de diagnosticar.

Essa definição leva a dois termos relacionados: confiabilidade e disponibilidade.

Confiabilidade é uma medida da realização contínua do serviço — ou, de modo equivalente, do tempo para a falha — a partir de um ponto de referência. Logo, *tempo médio para a falha* (MTTF — Mean Time To Failure) é uma medida de confiabilidade. Um termo relacionado é a *taxa de falhas anual* (AFR — Annual Failure Rate), que é simplesmente a porcentagem de dispositivos que se espera falhar em um ano para determinado MTTF. Quando o MTTF fica grande, ele pode ser enganoso, enquanto a AFR leva a uma intuição melhor.

MTTF versus AFR de discos

Exemplo

Alguns discos hoje são classificados como tendo um MTTF de 1.000.000 horas. Como $1.000.000 \text{ horas} = 1.000.000 / (365 \times 24) = 114 \text{ anos}$, pode parecer que eles praticamente nunca falham. Computadores em escala gigantesca, que rodam serviços para a Internet, como os de busca, podem ter 50.000 servidores. Suponha que cada servidor tenha 2 discos. Use a AFR para calcular quantos discos esperaríamos que falhem por ano.

Resposta

Um ano é $365 \times 24 = 8760$ horas. Um MTTF de 1.000.000 horas significa uma AFR de $8760 / 1.000.000 = 0,876\%$. Com 100.000 discos, esperaríamos que 876 discos falhem por ano, ou uma média de mais de 2 discos falhando por dia!

A interrupção de serviço é medida como o *tempo médio para o reparo* (MTTR — Mean Time To Repair). *Tempo médio entre falhas* (MTBF — Mean Time Between Failures) é simplesmente a soma de MTTF + MTTR. Embora o MTBF seja bastante usado, o MTTF geralmente é o termo mais apropriado. A *disponibilidade* é então uma medida da realização de serviço com relação à alternância entre os dois estados de realização e interrupção. A disponibilidade é

estatisticamente quantificada como

$$\text{Disponibilidade} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}$$

Observe que a confiabilidade e a disponibilidade são realmente medidas quantificáveis, em vez de apenas sinônimos de estabilidade. Encurtar o MTTR pode ajudar a disponibilidade e também aumentar o MTTF. Por exemplo, ferramentas para detecção, diagnóstico e reparo de falhas podem ajudar a reduzir o tempo para o reparo de falhas e, portanto, melhorar a disponibilidade.

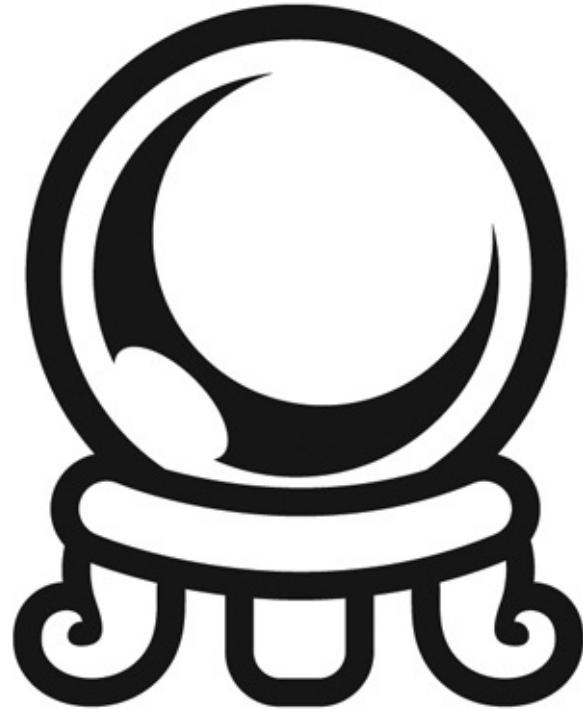
Queremos que a disponibilidade seja muito alta. Uma abreviação é indicar o número de “noves de disponibilidade” por ano. Por exemplo, um serviço de Internet muito bom hoje oferece 4 ou 5 noves de disponibilidade. Com 365 por ano, que significa $365 \times 24 \times 60 = 526.000$ minutos, então a abreviação é decodificada da seguinte forma:

Um nove:	90%	=> 36,5 dias de reparo/ano
Dois noves:	99%	=> 3,65 dias de reparo/ano
Três noves:	99,9%	=> 526 minutos de reparo por ano
Quatro noves:	99,99%	=> 52,6 minutos de reparo por ano
Cinco noves:	99,999%	=> 5,26 minutos de reparo por ano

e assim por diante.

Para aumentar o MTTF, você pode melhorar a qualidade dos componentes ou sistemas de projeto para continuar a operação na presença de componentes que falharam. Logo, a falha precisa ser definida em relação a um contexto, pois a falha de um componente pode não levar a uma falha do sistema. Para tornar essa distinção clara, o termo *falha* é usado para significar a falha de um componente. Aqui estão três maneiras de melhorar o MTTF:

1. *Impedimento de falha*: Impedir a ocorrência de falha pela construção.
2. *Tolerância a falha*: Usar redundância para permitir que o serviço cumpra a especificação do serviço apesar da ocorrência de falhas.
3. *Previsão de falha*: **Prever** a presença e a criação de falhas, permitindo que o componente seja substituído *antes* de falhar.



P R E D I Ç Ã O

O código Single Error Correcting, Double Error Detecting (SEC/DED) de Hamming

Richard Hamming inventou um esquema popular de redundância para a memória, para o qual recebeu o Turing Award em 1968. Para inventar códigos redundantes, é útil falar sobre a “proximidade” dos padrões de bit corretos. Chamamos de *distância de Hamming* simplesmente o número mínimo de bits que são diferentes entre dois padrões de bit corretos quaisquer. Por exemplo, a distância entre 011011 e 001111 é dois. O que acontece se a distância mínima entre os membros de um código for dois e obtivermos um erro de um bit? Isso tornará um padrão válido em um código para um padrão inválido. Assim, se pudermos detectar se os membros de um código são válidos ou não, poderemos detectar erros de único bit, e podemos dizer que temos um **código de detecção de erro** de único bit.

código de detecção de erro

Um código que permite a detecção de um erro nos dados, mas não o local exato e, portanto, a correção do erro.

Hamming usou um *código de paridade* para a detecção de erro. Em um código de paridade, o número de 1s em uma palavra é contado; a palavra tem paridade ímpar se o número de 1s for ímpar e par, em caso contrário. Quando uma palavra é escrita na memória, o bit de paridade também é escrito (1 para ímpar, 0 para par). Ou seja, a paridade da palavra de $N + 1$ bits sempre deverá ser par. Então, quando a palavra é lida, o bit de paridade é lido e verificado. Se a paridade da palavra de memória e o bit de paridade armazenado não combinarem, isso significa que houve um erro.

Exemplo

Calcule a paridade de um byte com o valor 31_{dec} e mostre o padrão armazenado na memória. Suponha que o bit de paridade esteja à direita. Suponha que o bit mais significativo fosse invertido na memória, e então você o lê de volta. Você detectou o erro? O que acontece se os dois bits mais significativos forem invertidos?

Resposta

31_{dec} é 00011111_{bin} , que tem cinco 1s. Para tornar a paridade par, precisamos escrever um 1 no bit de paridade, ou 00011111_{bin} . Se o bit mais significativo fosse invertido quando o lêssemos de volta, veríamos $\underline{1}0011111_{bin}$, que tem sete 1s. Como esperamos a paridade par e calculamos paridade ímpar, sinalizariamos um erro. Se os *dois* bits mais significativos fossem invertidos, veríamos $\underline{1}1011111_{bin}$, que tem oito 1s e paridade par; neste caso, *não* sinalizariamos um erro.

Se houver 2 bits com erro, então um esquema de paridade de 1 bit não detectará erro algum, pois a paridade será a mesma nos dados contendo dois bits errados. (Na verdade, um esquema de paridade de 1 bit pode detectar qualquer número ímpar de erros; porém, a probabilidade de ter 3 erros é muito menor do que a probabilidade de ter dois, de modo que, na prática, um código de paridade de 1 bit é limitado a detectar um único bit errado.)

Naturalmente, um código de paridade não pode corrigir erros, o que Hamming

queria fazer além de detectá-los. Se usássemos um código contendo uma distância mínima de 3, então qualquer erro de único bit estaria mais próximo do padrão correto do que de qualquer outro padrão válido. Ele sugeriu um mapeamento de dados fácil de entender para um código com distância 3, que chamamos *Código de Correção de Erro (ECC) de Hamming*, em sua homenagem. Usamos os bits de paridade extras para permitir a identificação da posição de um erro isolado. Aqui estão as etapas para se calcular o ECC de Hamming:

1. Comece numerando os bits a partir de 1 na esquerda, ao contrário da numeração tradicional, onde o bit mais à direita é o 0.
2. Marque todas as posições de bit que são potências de 2 como bits de paridade (posições 1, 2, 4, 8, 16, ...).
3. Todas as outras posições de bit são usadas para bits de dados (posições 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, ...).
4. A posição do bit de paridade, que determina a sequência de bits de dados que ele verifica (a [Figura 5.24](#) mostra essa explicação graficamente), é:
 - Bit 1 (0001_{bin}) verifica os bits (1,3,5,7,9,11,...), que são os bits onde o bit mais à direita do endereço é 1 (0001_{bin} , 0011_{bin} , 0101_{bin} , 0111_{bin} , 1001_{bin} , 1011_{bin} ,...).
 - Bit 2 (0010_{bin}) verifica os bits (2,3,6,7,10,11,14,15,...), que são os bits onde o segundo bit à direita no endereço é 1.
 - Bit 4 (0100_{bin}) verifica os bits (4–7, 12–15, 20–23,...), que são os bits onde o terceiro bit à direita no endereço é 1.
 - Bit 8 (1000_{bin}) verifica os bits (8–15, 24–31, 40–47,...), que são os bits onde o quarto bit à direita no endereço é 1.

Observe que cada bit de dados é coberto por dois ou mais bits de paridade.

5. Defina os bits de paridade para criar paridade par para cada grupo.

Posição de bit	1	2	3	4	5	6	7	8	9	10	11	12
Bits de dados codificados	p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Cobertura do bit de paridade	p1	X		X		X		X		X		X
	p2		X	X			X	X			X	X
	p4				X	X	X	X				X
	p8								X	X	X	X

FIGURA 5.24 Bits de paridade, bits de dados e cobertura de campo em um código ECC de Hamming para oito bits de dados.

No que parece ser um truque de mágica, você pode então determinar se os bits estão incorretos examinando os bits de paridade. Usando o código de 12 bits da Figura 5.24, se o valor dos quatro cálculos de paridade (p8,p4,p2,p1) fosse 0000, então não haveria erro. Porém, se o padrão fosse, digamos, 1010, que é 10_{dec}, então o ECC de Hamming nos diz que o bit 10 (d6) tem um erro. Como o número é binário, podemos corrigir o erro simplesmente invertendo o valor do bit 10.

Exemplo

Suponha que o valor de um byte de dados seja 10011010_{bin}. Primeiro, mostre o código ECC de Hamming para esse byte e depois inverta o bit 10 e mostre que o código ECC encontra e corrige o erro no único bit.

Resposta

Deixando espaços para os bits de paridade, o padrão de 12 bits é 1 0 0 1 1 0 1 0.

A posição 1 verifica os bits 1,3,5,7,9 e 11, destacados a seguir: 1 0 0 1 1 0 1 0. Para tornar a paridade do grupo par, devemos definir o bit 1 como 0.

A posição 2 verifica os bits 2,3,6,7,10,11, que é 0 1 0 0 1 1 0 1 0, ou paridade ímpar, de modo que definimos a posição 2 como 1.

A posição 4 verifica os bits 4,5,6,7,12, que é 0 1 1 0 0 1 1 0 1, de modo que o definimos como 1.

A posição 8 verifica os bits 8,9,10,11,12, que é 0 1 1 1 0 0 1 1 0 1 0, de modo que o definimos como 0.

A palavra de código final é 011100101010. A inversão do bit 10 muda a

palavra para 011100101110.

O bit de paridade 1 é 0 (011100101110 tem quatro 1s, paridade par; este grupo está OK).

O bit de paridade 2 é 1 (011100101110 tem cinco 1s, paridade ímpar; há um erro em algum lugar).

O bit de paridade 4 é 1 (011100101110 tem dois 1s, paridade par; este grupo está OK).

O bit de paridade 8 é 1 (011100101110 tem três 1s, paridade ímpar; há um erro em algum lugar).

Os bits de paridade 2 e 8 estão incorretos. Como $2 + 8 = 10$, o bit 10 deverá estar errado. Logo, podemos corrigir o erro invertendo o bit 10: 011100101010. Pronto!

Hamming não parou no código de correção de erro de único bit. Com o custo de mais um bit, podemos fazer com que a distância mínima de Hamming em um código seja 4. Isso significa que podemos corrigir erros de único bit *e detectar erros de duplo bit*. A ideia é acrescentar um bit de paridade que seja calculado sobre toda a palavra. Vamos usar uma palavra de dados de quatro bits como um exemplo, que só precisaria de 7 bits para uma detecção de erro de único bit. Os bits de paridade de Hamming H ($p_1 p_2 p_3$) são calculados (paridade par, como sempre), mas a paridade par sobre a palavra inteira, p_4 :

1 2 3 4 5 6 7 **8**
 $p_1 \quad p_2 \quad d_1 \quad p_3 \quad d_2 \quad d_3 \quad d_4 \quad \mathbf{p}_4$

Então, o algoritmo para corrigir um erro e detectar dois é simplesmente calcular a paridade sobre os grupos do ECC (H) como antes e mais um sobre o grupo inteiro (p_4). Existem quatro casos:

1. H é par e p_4 é par; portanto, não houve erro.
2. H é ímpar e p_4 é ímpar, de modo que houve um único erro corrigível. (p_4 deveria calcular paridade ímpar se houvesse um erro).
3. H é par e p_4 é ímpar; houve um único erro em p_4 , e não no restante da palavra, de modo que corrigimos o bit p_4 .
4. H é ímpar e p_4 é par; houve um erro duplo. (p_4 deveria calcular paridade par se houvesse dois erros.)

Single Error Correcting / Double Error Detecting (SEC/DED) é comum na memória de servidores hoje em dia. De modo conveniente, blocos de dados com oito bytes podem obter SEC/DED com apenas um byte a mais, motivo pelo qual

muitas DIMMs possuem 72 bits de largura.

Detalhamento

Para calcular quantos bits são necessários para SEC, imagine que p seja o número total de bits de paridade e d o número de bits de dados na palavra de $p + d$ bits. Se p bits de correção de erro tiverem que apontar para o bit de erro ($p + d$ casos) mais um caso para indicar que não existe erro, precisamos de:

$$2^p \geq p + d + 1 \text{ bits, e portanto } p \geq \log(p + d + 1).$$

Por exemplo, para dados de 8 bits isso significa $d = 8$ e $2^p \geq p + 8 + 1$, de modo que $p = 4$. De modo semelhante, $p = 5$ para 16 bits de dados, 6 para 32 bits, 7 para 64 bits, e assim por diante.

Detalhamento

Em sistemas muito grandes, a possibilidade de erros múltiplos, bem como a falha completa de um único chip de memória largo, torna-se significativa. A IBM introduziu o *Chipkill* para resolver esse problema, e muitos sistemas grandes utilizam essa tecnologia. (A Intel chama sua versão de SDDC.) Semelhante por natureza à técnica de RAID usada para discos, chipkill distribui os dados e informações de ECC, de modo que a falha completa de um único chip de memória pode ser tratada com o suporte à reconstrução dos dados que faltam a partir dos chips de memória restantes. Considerando um cluster de 10.000 processadores com 4 GiB por processador, a IBM calculou as seguintes taxas de erro de memória *irrecuperável* em três anos de operação:

- Paridade apenas — cerca de 90.000, ou uma falha irrecuperável (ou não detectada) a cada 17 minutos.
- SEC/DED apenas — cerca de 3500, ou aproximadamente uma falha não detectada ou irrecuperável a cada 7,5 horas.
- Chipkill — 6, ou cerca de uma falha não detectada ou irrecuperável a cada 2 meses.

Logo, Chipkill é um requisito para computadores em escala gigantesca.

Detalhamento

Detalhamento

Embora erros de um ou dois bits sejam comuns para sistemas de memória, as redes podem ter rajadas de erros de bit. Uma solução é denominada *Cyclic Redundancy Check* (verificação de redundância cíclica). Para um bloco de k bits, um transmissor gera uma sequência de verificação de quadro de $n-k$ bits. Ele transmite n bits exatamente divisíveis por algum número. O receptor divide o quadro por esse número. Se não houver resto, ele considera que não há erro. Se houver, o receptor rejeita a mensagem e pede ao transmissor para enviar novamente. Como você pode imaginar pelo Capítulo 3, é fácil calcular a divisão por alguns números binários com um registrador de deslocamento, o que torna os códigos CRC populares até mesmo quando o hardware era mais precioso. Seguindo mais à frente, os códigos Reed-Solomon utilizam campos de Galois para *corrigir* erros de transmissão em múltiplos bits, mas agora os dados são considerados coeficientes de um polinômio e o espaço de código são os valores de um polinômio. O cálculo Reed-Solomon é, muitas vezes, mais complicado do que a divisão binária!

5.6. Máquinas virtuais

Máquinas Virtuais (VM — Virtual Machines) foram desenvolvidas inicialmente em meados da década de 1960, e continuaram sendo uma parte importante da computação de grande porte com o passar dos anos. Embora bastante ignoradas na era do PC monousuário, durante as décadas de 1980 e 1990, elas obtiveram popularidade recentemente, devido aos seguintes fatores:

- O aumento de importância do isolamento e da segurança nos sistemas modernos
- As falhas na segurança e na confiabilidade dos sistemas operacionais padrão
- O compartilhamento de um único computador entre muitos usuários não relacionados, particularmente para a computação em nuvem
- Os aumentos fantásticos na velocidade bruta dos processadores no decorrer das décadas, tornando mais aceitável o overhead das VMs

A definição mais geral das VMs inclui basicamente todos os métodos de emulação que oferecem uma interface de software padrão, como a Java VM. Nesta seção, estamos interessados nas VMs que oferecem um ambiente completo em nível de sistema, no nível da arquitetura de conjunto de instruções (ISA) binária. Embora algumas VMs excutem diferentes ISAs na VM do hardware nativo, consideramos que elas sempre correspondem ao hardware.

Essas VMs são chamadas de (Operating) *System Virtual Machines* — máquinas virtuais do sistema (operacional). Alguns exemplos são IBM VM/370, VirtualBox, VMware ESX Server e Xen.

As máquinas virtuais do sistema apresentam a ilusão de que os usuários têm um computador inteiro para si, incluindo uma cópia do sistema operacional. Um único computador executa várias VMs e pode aceitar diversos sistemas operacionais (OSs) diferentes. Em uma plataforma convencional, um único OS “possui” todos os recursos do hardware, mas, com uma VM, vários OSs compartilham os recursos do hardware.

O software que dá suporte às VMs é chamado de *monitor de máquina virtual* (VMM — Virtual Machine Monitor), ou *hipervisor*; o VMM é o centro da tecnologia de máquina virtual. A plataforma de hardware básica é chamada de *host* e seus recursos são compartilhados entre as VMs *guest*. O VMM determina como mapear recursos virtuais a recursos físicos: um recurso físico pode ser de tempo compartilhado, particionado ou ainda simulado no software. O VMM é muito menor que um OS tradicional; a parte de isolamento de um VMM possui talvez apenas 10.000 linhas de código.

Embora nosso interesse aqui seja as VMs para melhorar a proteção, elas oferecem dois outros benefícios que são comercialmente significativos:

1. *Gerenciar o software*. As VMs oferecem uma abstração que pode executar uma pilha de software completa, incluindo até mesmo sistemas operacionais antigos, como o DOS. Uma implantação típica poderia ser algumas VMs executando OSs legados, muitas executando a versão atual estável do OS, e algumas testando a próxima versão do OS.
2. *Gerenciar o hardware*. Um motivo para servidores múltiplos é ter cada aplicação executando com a versão compatível do sistema operacional em computadores separados, pois essa separação pode melhorar a confiabilidade. As VMs permitem que essas pilhas de software separadas sejam executadas independentemente enquanto compartilham o hardware, consolidando assim o número de servidores. Outro exemplo é que alguns VMMs admitem a migração de uma VM atual para um computador diferente, seja no sentido de balancear a carga ou sair do hardware com falha.

Interface Hardware/Software

Amazon Web Services (AWS) utiliza as máquinas virtuais em sua plataforma

de computação em nuvem, oferecendo EC2 por cinco razões:

1. Isso permite que a AWS proteja os usuários um do outro, enquanto compartilham o mesmo servidor.
2. Isso simplifica a distribuição de software dentro de um computador em escala gigantesca. Um cliente instala uma imagem de máquina virtual configurada com o software apropriado, e a AWS a distribui para todas as instâncias que um cliente quiser usar.
3. Os clientes (e a AWS) podem “matar” uma VM de modo confiável, para controlar o uso de recursos quando os clientes terminarem seu trabalho.
4. As máquinas virtuais ocultam a identidade do hardware em que o cliente está executando, o que significa que a AWS pode continuar usando servidores antigos e introduzir novos servidores, mais eficientes. O cliente espera que o desempenho para as instâncias corresponda às suas avaliações em “Unidades de Computação EC2”, que a AWS define como: “fornecer a capacidade de CPU equivalente de um processador AMD Opteron 2007 de 1,0 a 1,2 GHz ou Intel Xeon 2007”. Graças à **Lei de Moore**, servidores mais novos claramente oferecem mais Unidades de Computação EC2 do que os mais antigos, mas a AWS pode continuar alugando servidores antigos, desde que sejam econômicos.
5. Os Monitores de Máquina Virtual podem controlar a velocidade com que a VM usa o processador, a rede e o espaço em disco, permitindo que a AWS ofereça muitos pontos com preço de instâncias de diferentes tipos executando nos mesmos servidores subjacentes. Por exemplo, em 2012, a AWS oferecia 14 tipos de instância, desde pequenas instâncias padrão a US\$ 0,08 por hora a instâncias quádruplas extra grandes com alta E/S a US\$ 3,10 por hora.



Em geral, o custo da virtualização do processador depende da carga de trabalho. Os programas ligados ao processador em nível de usuário possuem overhead de virtualização zero, pois o OS raramente é chamado, de modo que tudo é executado nas velocidades nativas. As cargas de trabalhos com uso intenso de E/S em geral usam intensamente o OS, executando muitas chamadas do sistema e instruções privilegiadas, o que pode resultar em um alto overhead de virtualização. Por outro lado, se a carga de trabalho com uso intenso de E/S também for *voltada para E/S*, o custo da virtualização do processador pode ser completamente ocultado, pois o processador geralmente está ocioso, esperando pela E/S.

O overhead é determinado pelo número de instruções que devem ser simuladas pelo VMM e por quanto tempo cada uma precisa simular. Logo, quando as VMs guest executam a mesma ISA que o *host*, como consideramos aqui, o objetivo da arquitetura e do VMM é executar quase todas as instruções diretamente no hardware nativo.

Requisitos de um monitor de máquina virtual

O que um monitor de VM precisa fazer? Ele apresenta uma interface de software ao software guest, precisa isolar o estado dos guests um do outro e precisa proteger-se contra o software guest (incluindo os OSs guest). Os requisitos

qualitativos são:

- O software guest deverá se comportar em uma VM exatamente como se estivesse sendo executado no hardware nativo, exceto pelo comportamento relacionado ao desempenho ou limitações de recursos fixos compartilhados por múltiplas VMs.
- O software guest não deverá alterar diretamente a alocação de recursos reais do sistema.

Para “virtualizar” o processador, o VMM precisa controlar praticamente tudo — acesso ao estado privilegiado, tradução de endereços, E/S, exceções e interrupções — embora a VM guest e o OS atualmente em execução estejam temporariamente utilizando-os.

Por exemplo, no caso de uma interrupção de um temporizador, a VMM suspenderia a VM guest atualmente em execução, salvaria seu estado, trataria da interrupção, determinaria qual VM guest será executada em seguida e, depois, carregaria seu estado. As VMs guest que contam com uma interrupção de temporizador recebem um temporizador virtual e uma interrupção de temporizador simulada pelo VMM.

Para estar no controle, o VMM precisa estar em um nível de privilégio mais alto que a VM guest, que geralmente é executada no modo usuário; isso também garante que a execução de qualquer instrução privilegiada será tratada pelo VMM. Os requisitos básicos do sistema de máquina virtual:

- Pelo menos dois modos de processador, sistema e usuário.
- Um subconjunto de instruções privilegiado, que está disponível apenas no modo do sistema, resultado em um trap se executado no modo usuário; todos os recursos do sistema precisam ser controláveis apenas por meio dessas instruções.

(Falta de) Suporte da arquitetura do conjunto de instruções para máquinas virtuais

Se as VMs forem planejadas durante o projeto da ISA, será relativamente fácil reduzir o número de instruções que devem ser executadas por um VMM e sua velocidade de simulação. Uma arquitetura que permite que a VM seja executada diretamente no hardware recebe o título de *virtualizável*, e a arquitetura IBM 370 orgulhosamente ostenta esse rótulo.

Contudo, como as VMs foram consideradas para aplicações de servidor e PC apenas recentemente, a maioria dos conjuntos de instruções foi criada sem a

virtualização em mente. Esses culpados incluem a x86 e a maioria das arquiteturas RISC, incluindo ARMv7 e MIPS.

Como o VMM precisa garantir que o sistema guest só interaja com recursos virtuais, um OS guest convencional é executado como um programa no modo usuário em cima do VMM. Então, se um OS guest tentar acessar ou modificar informações relacionadas aos recursos do hardware por meio de uma instrução privilegiada (por exemplo, lendo ou escrevendo um bit de status que habilita interrupções), isso será interceptado pelo VMM. O VMM poderá então efetuar as mudanças apropriadas nos recursos reais correspondentes.

Portanto, se qualquer instrução que tenta ler ou escrever essas informações sensíveis for interceptada quando executada no modo usuário, o VMM poderá interceptá-la e dar suporte a uma versão virtual da informação sensível, conforme o OS guest espera.

Na ausência desse suporte, outras medidas deverão ser tomadas. Um VMM precisa tomar precauções especiais para localizar todas as instruções problemáticas e garantir que elas se comportem corretamente quando executadas por um OS guest, aumentando assim a complexidade do VMM e reduzindo o desempenho da execução da VM.

Proteção e arquitetura do conjunto de instruções

Proteção é um esforço conjunto da arquitetura e dos sistemas operacionais, mas os arquitetos tiveram de modificar alguns detalhes desajeitados das arquiteturas de conjunto de instruções existentes quando a memória virtual se tornou popular.

Por exemplo, a instrução POPF do x86 carrega os registradores de flag do topo da pilha para a memória. Um dos flags é o flag *Interrupt Enable* (IE). Se você executar a instrução POPF no modo usuário, em vez de interceptá-la, ela simplesmente muda todos os flags exceto IE. No modo do sistema, ela muda o IE. Como um OS guest é executado no modo usuário dentro de uma VM, isso é um problema, pois espera ver um flag IE alterado.

Historicamente, o hardware mainframe IBM e o VMM exigiam três etapas para melhorar o desempenho das máquinas virtuais:

1. Reduzir o custo da virtualização do processador.
2. Reduzir o custo de overhead da interrupção devido à virtualização.
3. Reduzir o custo da interrupção direcionando as interrupções para a VM apropriada sem chamar o VMM.

Em 2006, novas propostas da AMD e Intel tentaram resolver o primeiro

ponto, reduzindo o custo da virtualização do processador. Será interessante ver quantas gerações de arquitetura e modificações do VMM serão necessárias para resolver todos os três pontos, e quanto tempo passará antes que as máquinas virtuais do século XXI sejam tão eficientes quanto os mainframes IBM e VMMs da década de 1970.

5.7. Memória virtual

...foi inventado um sistema para fazer a combinação entre os sistemas centrais de memória e os tambores de discos aparecer para o programador como um depósito de nível único, com as transferências necessárias ocorrendo automaticamente.

Kilburn et al., One-level storage system, 1962

Nas seções anteriores, vimos como as caches fornecem acesso rápido às partes recentemente usadas do código e dos dados de um programa. Da mesma forma, a memória principal pode agir como uma “cache” para o armazenamento secundário, normalmente implementado com discos magnéticos. Essa técnica é chamada de **memória virtual**. Historicamente, houve duas motivações principais para a memória virtual: permitir o compartilhamento seguro e eficiente da memória entre vários programas, removendo os transtornos de programação de uma quantidade pequena e limitada de memória principal. Cinco décadas após sua invenção, o primeiro motivo é o que ainda predomina.

memória virtual

Uma técnica que usa a memória principal como uma “cache” para armazenamento secundário.

É claro que, para permitir que várias máquinas virtuais compartilhem a mesma memória, precisamos ser capazes de proteger essas VMsumas das outras, garantindo que um programa só possa ler e escrever as partes da memória principal atribuídas a ele. A memória principal precisa conter apenas as partes ativas das muitas máquinas virtuais, exatamente como uma cache contém apenas a parte ativa de um programa. Portanto, o princípio da localidade possibilita a memória virtual e as caches, e a memória virtual nos permite compartilhar eficientemente o processador e a memória principal.

Não podemos saber quais máquinas virtuais irão compartilhar a memória com outras máquinas virtuais quando compilamos seus programas. Na verdade, as máquinas virtuais que compartilham a memória mudam dinamicamente enquanto estão sendo executadas. Devido a essa interação dinâmica, gostaríamos

de compilar cada programa para o seu próprio *espaço de endereçamento* — faixa distinta dos locais de memória acessível apenas a esse programa. A memória virtual implementa a tradução do espaço de endereçamento de um programa para os **endereços físicos**. Esse processo de tradução impõe a **proteção** do espaço de endereçamento de um programa contra outras máquinas virtuais.

endereço físico

Um endereço na memória principal.

proteção

Um conjunto de mecanismos para garantir que múltiplos processos compartilhando processador, memória ou dispositivos de E/S não possam interferir, intencionalmente ou não, um com o outro, lendo ou escrevendo dados um do outro. Esses mecanismos também isolam o sistema operacional de um processo de usuário.

A segunda motivação para a memória virtual é permitir que um único programa do usuário exceda o tamanho da memória principal. Antigamente, se um programa se tornasse muito grande para a memória, cabia ao programador fazê-lo se adequar. Os programadores dividiam os programas em partes e, então, identificavam aquelas mutuamente exclusivas. Esses *overlays* eram carregados ou descarregados sob o controle do programa do usuário durante a execução, com o programador garantindo que o programa nunca tentaria acessar um overlay que não estivesse carregado e que os overlays carregados nunca excederiam o tamanho total da memória. Os overlays eram tradicionalmente organizados como módulos, cada um contendo código e dados. As chamadas entre procedimentos em módulos diferentes levavam um módulo a se sobrepor a outro.

Como você pode bem imaginar, essa responsabilidade era uma carga substancial para os programadores. A memória virtual, criada para aliviar os programadores dessa dificuldade, gerencia automaticamente os dois níveis da hierarquia de memória representados pela memória principal (às vezes, chamada de *memória física* para distingui-la da memória virtual) e pelo armazenamento secundário.

Embora os conceitos aplicados na memória virtual e nas caches sejam os mesmos, suas diferentes raízes históricas levaram ao uso de uma terminologia

diferente. Um bloco de memória virtual é chamado de *página*, e uma falha da memória virtual é chamada de **falta de página**. Com a memória virtual, o processador produz um **endereço virtual**, traduzido por uma combinação de hardware e software para um *endereço físico*, que, por sua vez, pode ser usado de modo a acessar a memória principal. A Figura 5.25 mostra a memória endereçada virtualmente com páginas mapeadas na memória principal. Esse processo é chamado de *mapeamento de endereço* ou **tradução de endereço**. Hoje, os dois níveis de hierarquia de memória controlados pela memória virtual são as DRAMs e a memória flash nos dispositivos móveis pessoais, e as DRAMs e os discos magnéticos nos servidores (Seção 5.2). Se voltarmos à nossa analogia da biblioteca, podemos pensar no endereço virtual como o título de um livro e no endereço físico como seu local na biblioteca.

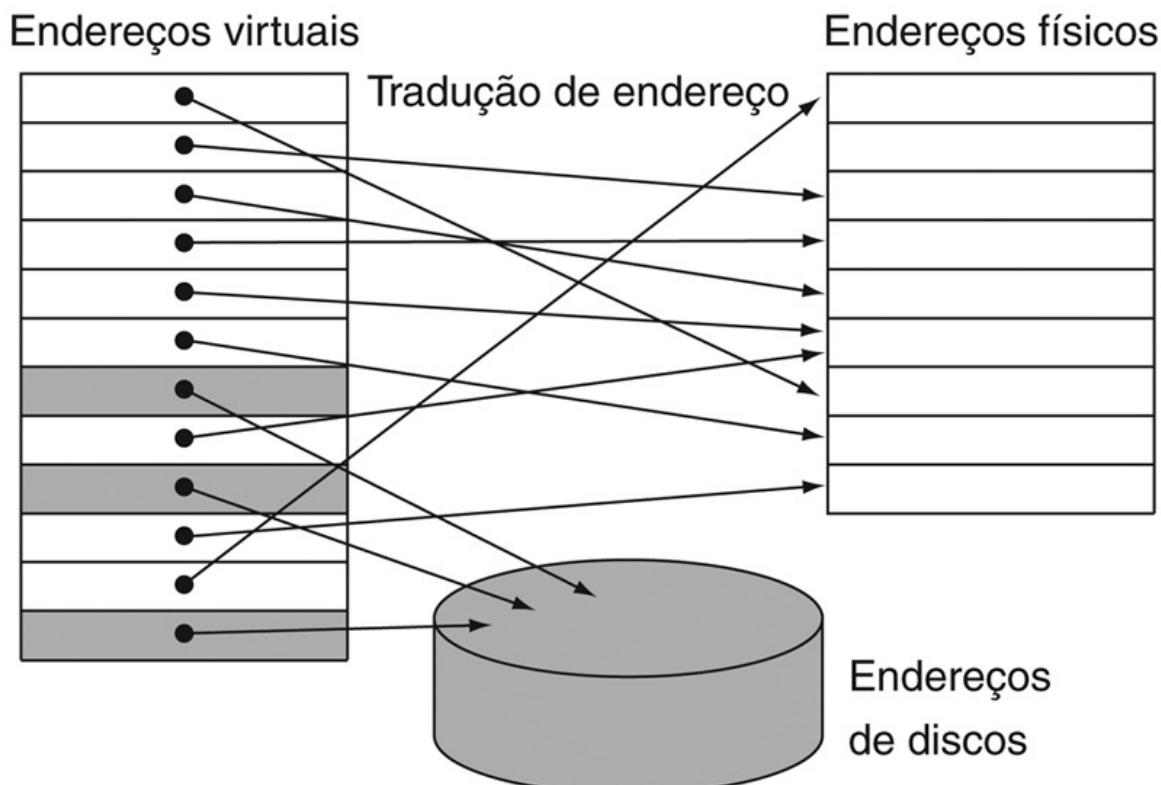


FIGURA 5.25 Na memória virtual, os blocos de memória (chamados de *páginas*) são mapeados de um conjunto de endereços (chamados de *endereços virtuais*) para outro conjunto (chamado de *endereços físicos*).

O processador gera endereços virtuais enquanto a memória é acessada usando endereços físicos. Tanto a memória virtual quanto a memória física são desmembradas em páginas, de

modo que uma página virtual é realmente mapeada em uma página física. Naturalmente, também é possível que uma página virtual esteja ausente da memória principal e não seja mapeada para um endereço físico, residindo no disco em vez disso. As páginas físicas podem ser compartilhadas fazendo dois endereços virtuais apontarem para o mesmo endereço físico. Essa capacidade é usada para permitir que dois programas diferentes compartilhem dados ou código.

falta de página

Um evento que ocorre quando uma página acessada não está presente na memória principal.

endereço virtual

Um endereço que corresponde a um local no espaço virtual e é traduzido pelo mapeamento de endereço para um endereço físico quando a memória é acessada.

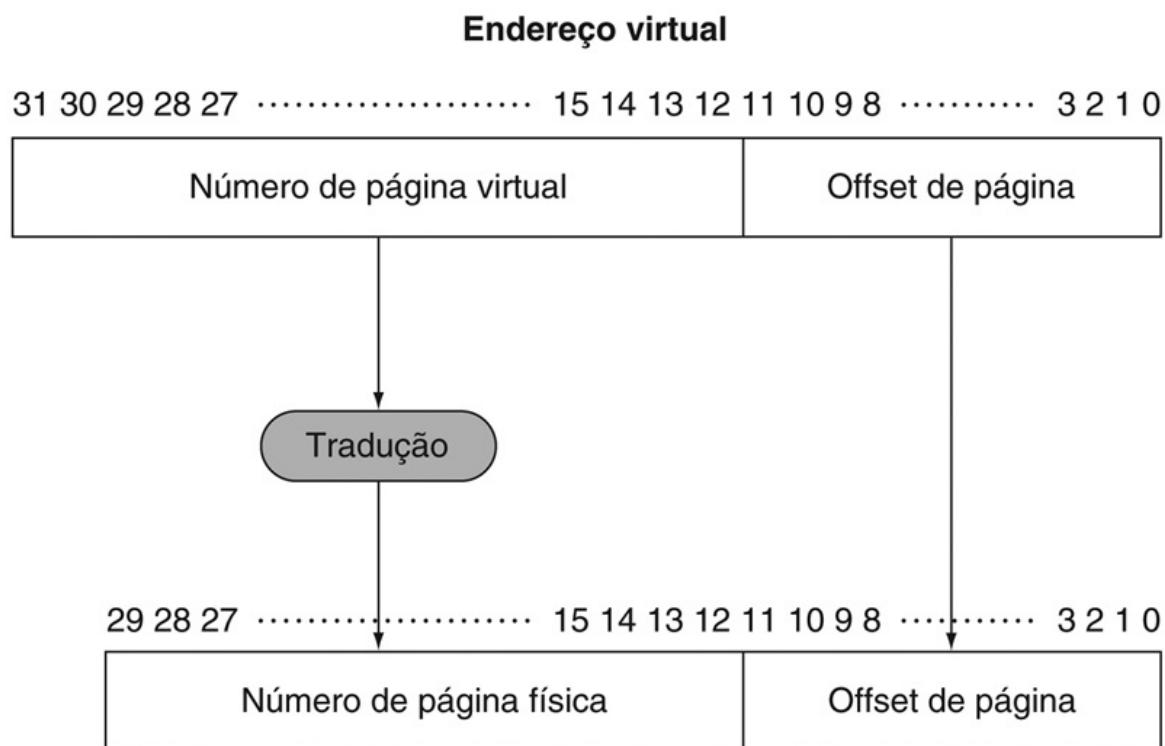
tradução de endereço

Também chamada de **mapeamento de endereço**. O processo pelo qual um endereço virtual é mapeado a um endereço usado para acessar a memória.

A memória virtual também simplifica o carregamento do programa para execução fornecendo *relocação*. A relocação mapeia os endereços virtuais usados por um programa para diferentes endereços físicos antes que os endereços sejam usados no acesso à memória. Essa relocação nos permite carregar o programa em qualquer lugar na memória principal. Além disso, todos os sistemas de memória virtual em uso atualmente relocam o programa como um conjunto de blocos (páginas) de tamanho fixo, eliminando, assim, a necessidade de encontrar um bloco contíguo de memória para alojar um programa; em vez disso, o sistema operacional só precisa encontrar um número suficiente de páginas na memória principal.

Na memória virtual, o endereço é desmembrado em um *número de página virtual* e um *offset de página*. A [Figura 5.26](#) mostra a tradução do número de página virtual para um *número de página física*. O número de página física

constitui a parte mais significativa do endereço físico, enquanto o offset de página, que não é alterado, constitui a parte menos significativa. O número de bits no campo offset de página determina o tamanho da página. O número de páginas endereçáveis com o endereço virtual não precisa corresponder ao número de páginas endereçáveis com o endereço físico. Ter um número de páginas virtuais maior do que as páginas físicas é a base para a ilusão de uma quantidade de memória virtual essencialmente ilimitada.



Endereço físico

FIGURA 5.26 Mapeamento de um endereço virtual em um endereço físico.

O tamanho de página é $2^{12} = 4\text{KB}$. O número de páginas físicas permitido na memória é 2^{18} , já que o número de página física contém 18 bits. Portanto, a memória principal pode ter, no máximo, 1GB, enquanto o espaço de endereço virtual possui 4GB.

Muitas escolhas de projeto nos sistemas de memória virtual são motivadas pelo alto custo de uma falta de página. Uma falta de página em disco levará milhões de ciclos de clock para ser processada. (A tabela na [Seção 5.1](#) mostra

que a latência da memória principal é aproximadamente 100.000 vezes mais rápida do que o disco.) Essa enorme penalidade de falha, dominada pelo tempo para obter a primeira palavra para tamanhos de página típicos, leva a várias decisões importantes nos sistemas de memória virtual:

- As páginas devem ser grandes o suficiente para tentar amortizar o longo tempo de acesso. Tamanhos de 4 KiB a 16 KiB são comuns atualmente. Novos sistemas de desktop e servidor estão sendo desenvolvidos para suportar páginas de 32 KiB e 64 KiB, embora novos sistemas embutidos estejam indo na outra direção, para páginas de 1 KiB.
- Organizações que reduzem a taxa de faltas de página são atraentes. A principal técnica usada aqui é permitir o posicionamento totalmente associativo das páginas na memória.
- As faltas de página podem ser tratadas em nível de software porque o overhead será pequeno se comparado com o tempo de acesso ao disco. Além disso, o software pode se dar ao luxo de usar algoritmos inteligentes para escolher como posicionar as páginas, já que mesmo pequenas reduções na taxa de falhas compensarão o custo desses algoritmos.
- O write-through não funcionará para a memória virtual, visto que as escritas levam muito tempo. Em vez disso, os sistemas de memória virtual usam write-back.

As próximas subseções tratam desses fatores no projeto de memória virtual.

Detalhamento

Apresentamos a motivação para a memória virtual como muitas máquinas virtuais compartilhando a mesma memória, mas a memória virtual foi inventada originalmente de modo que muitos programas pudessem compartilhar um computador como parte de um sistema de tempo compartilhado. Como muitos leitores hoje não possuem experiência com sistemas de tempo compartilhado, usamos as máquinas virtuais para motivar esta seção.

Detalhamento

Para computadores servidores e desktops, processadores de 32 bits já são problemáticos. Embora normalmente imaginemos os endereços virtuais como muito maiores do que os endereços físicos, o contrário pode ocorrer quando o

tamanho de endereço do processador é pequeno em relação ao estado da tecnologia de memória. Nenhum programa ou máquina virtual isolada pode se beneficiar, mas um grupo de programas executados ao mesmo tempo pode se beneficiar de não precisar ser trocado para a memória ou de ser executado em processadores paralelos.

Detalhamento

A discussão da memória virtual neste livro focaliza na paginação, que usa blocos de tamanho fixo. Há também um esquema de blocos de tamanho variável chamado **segmentação**. Na segmentação, um endereço consiste em duas partes: um número de segmento e um offset de segmento. O registrador de segmento é mapeado a um endereço físico e o offset é *somado* para encontrar o endereço físico real. Como o segmento pode variar em tamanho, uma verificação de limites é necessária para garantir que o offset esteja dentro do segmento. O principal uso da segmentação é suportar métodos de proteção mais avançados e compartilhar um espaço de endereçamento. A maioria dos livros de sistemas operacionais contém extensas discussões sobre a segmentação comparada com a paginação e sobre o uso da segmentação para compartilhar logicamente o espaço de endereçamento. A principal desvantagem da segmentação é que ela divide o espaço de endereçamento em partes logicamente separadas que precisam ser manipuladas como um endereço de duas partes: o número de segmento e o offset. A paginação, por outro lado, torna o limite entre o número de página e o offset invisível aos programadores e compiladores.

segmentação

Um esquema de mapeamento de endereço de tamanho variável em que um endereço consiste em duas partes: um número de segmento, que é mapeado para um endereço físico, e um offset de segmento.

Os segmentos também têm sido usados como um método para estender o espaço de endereçamento sem mudar o tamanho da palavra do computador. Essas tentativas têm sido malsucedidas devido à dificuldade e ao ônus de desempenho inerentes a um endereço de duas partes, dos quais os programadores e compiladores precisam estar cientes.

Muitas arquiteturas dividem o espaço de endereçamento em grandes blocos de tamanho fixo que simplificam a proteção entre o sistema operacional e os programas de usuário e aumentam a eficiência da implementação da paginação. Embora essas divisões normalmente sejam chamadas de “segmentos”, esse mecanismo é muito mais simples do que a segmentação de tamanho de bloco variável e não é visível aos programas do usuário; discutiremos o assunto em mais detalhes em breve.

Posicionando uma página e a encontrando novamente

Em razão da penalidade incrivelmente alta decorrente de uma falta de página, os projetistas reduzem a frequência das faltas de página otimizando seu posicionamento. Se permitirmos que uma página virtual seja mapeada em qualquer página física, o sistema operacional, então, pode escolher substituir qualquer página que desejar quando ocorrer uma falta de página. Por exemplo, o sistema operacional pode usar um sofisticado algoritmo e complexas estruturas de dados, que monitoram o uso de páginas, para tentar escolher uma página que não será necessária por um longo tempo. A capacidade de usar um esquema de substituição inteligente e flexível reduz a taxa de faltas de página e simplifica o uso do posicionamento de páginas totalmente associativo.

Como mencionamos na [Seção 5.4](#), a dificuldade em usar posicionamento totalmente associativo está em localizar uma entrada, já que ela pode estar em qualquer lugar no nível superior da hierarquia. Uma pesquisa completa é impraticável. Nos sistemas de memória virtual, localizamos páginas usando uma tabela que indexa a memória; essa estrutura é chamada de **tabela de páginas** e reside na memória. Uma tabela de páginas é indexada pelo número de página do endereço virtual para descobrir o número da página física correspondente. Cada programa possui sua própria tabela de páginas, que mapeia o espaço de endereçamento virtual desse programa para a memória principal. Em nossa analogia da biblioteca, a tabela de páginas corresponde a um mapeamento entre os títulos dos livros e os locais da biblioteca. Exatamente como o catálogo de cartões pode conter entradas para livros em outra biblioteca ou campus em vez da biblioteca local, veremos que a tabela de páginas pode conter entradas para páginas não presentes na memória. A fim de indicar o local da tabela de páginas na memória, o hardware inclui um registrador que aponta para o início da tabela de páginas; esse registrador é chamado de *registrador de tabela de páginas*. Por

enquanto, considere que a tabela de páginas esteja em uma área fixa e contígua da memória.

tabela de páginas

A tabela com as traduções de endereço virtual para físico em um sistema de memória virtual. A tabela, armazenada na memória, normalmente é indexada pelo número de página virtual; cada entrada na tabela contém o número da página física para essa página virtual se a página estiver atualmente na memória.

Interface hardware/software

A tabela de páginas, juntamente com o contador de programa e os registradores, especifica o *estado* de uma máquina virtual. Se quisermos permitir que outra máquina virtual use o processador, precisamos salvar esse estado. Mais tarde, após restaurar esse estado, a máquina virtual pode continuar a execução. Frequentemente nos referimos a esse estado como um *processo*. O processo é considerado *ativo* quando está de posse do processador; caso contrário, ele é considerado *inativo*. O sistema operacional pode ativar um processo carregando o estado do processo, incluindo o contador de programa, o que irá iniciar a execução no valor salvo do contador de programa.

O espaço de endereçamento do processo e, consequentemente, todos os dados que ele pode acessar na memória, é definido pela sua tabela de páginas, que reside na memória. Em vez de salvar a tabela de páginas inteira, o sistema operacional simplesmente carrega o registrador de tabela de páginas de modo a apontar para a tabela de páginas do processo que ele quer tornar ativo. Cada processo possui sua própria tabela de páginas, já que diferentes processos usam os mesmos endereços virtuais. O sistema operacional é responsável por alocar a memória física e atualizar as tabelas de páginas, de modo que os espaços de endereço virtuais dos diferentes processos não colidam. Como veremos em breve, o uso de tabelas de páginas separadas também fornece proteção de um processo contra outro.

A [Figura 5.27](#) usa o registrador de tabela de páginas, o endereço virtual e a tabela de páginas indicada para mostrar como o hardware pode formar um

endereço físico. Um bit de validade é usado em cada entrada de tabela de páginas, exatamente como faríamos em uma cache. Se o bit estiver desligado, a página não está presente na memória principal e ocorre uma falta de página. Se o bit estiver ligado, a página está na memória e a entrada contém o número de página física.

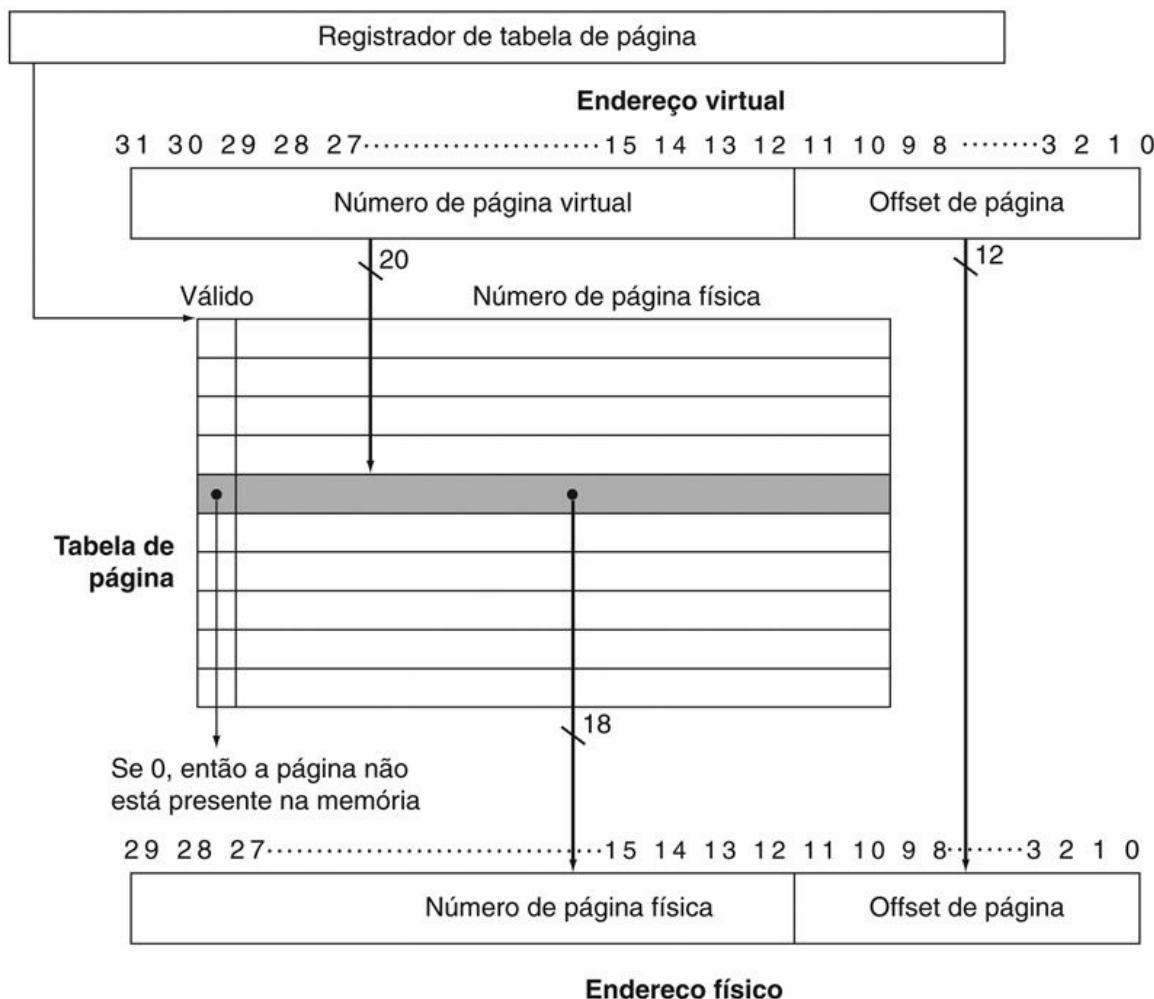


FIGURA 5.27 A tabela de páginas é indexada pelo número de página virtual para obter a parte correspondente do endereço físico.

Consideraremos um endereço de 32 bits. O endereço inicial da tabela de páginas é dado pelo ponteiro da tabela de páginas.

Nessa figura, o tamanho de página é 2^{12} bytes, ou 4 KiB. O espaço de endereço virtual é 2^{32} bytes, ou 4 GiB, e o espaço de endereçamento físico é 2^{30} bytes, que permite uma memória principal de até 1 GiB. O número de entradas na tabela de

páginas é 2^{20} , ou um milhão de entradas. O bit de validade para cada entrada indica se o mapeamento é válido. Se ele estiver desligado, a página não está presente na memória. Embora a entrada de tabela de páginas mostrada aqui só precise ter 19 bits de largura, ela normalmente seria arredondada para 32 bits a fim de facilitar a indexação. Os bits extras seriam usados para armazenar informações adicionais que precisam ser mantidas página a página, como a proteção.

Como a tabela de páginas contém um mapeamento para toda página virtual possível, nenhuma tag é necessária. Em terminologia de cache, o índice usado para acessar a tabela de páginas consiste no endereço de bloco inteiro, que é o número de página virtual.

Faltas de página

Se o bit de validade para uma página virtual estiver desligado, ocorre uma falta de página. O sistema operacional precisa receber o controle. Essa transferência é feita pelo mecanismo de exceção, que vimos no [Capítulo 4](#) e abordaremos mais uma vez posteriormente nesta seção. Quando o sistema operacional obtém o controle, ele precisa encontrar a página no próximo nível da hierarquia (geralmente a memória flash ou o disco magnético) e decidir onde colocar a página requisitada na memória principal.

O endereço virtual por si só não diz imediatamente onde está a página no disco. Voltando à nossa analogia da biblioteca, não podemos encontrar o local de um livro nas estantes apenas sabendo seu título. Em vez disso, precisamos ir ao catálogo e consultar o livro, obter um endereço para o local nas estantes. Da mesma forma, em um sistema de memória virtual, é necessário monitorar o local no disco de cada página em um espaço de endereçamento virtual.

Como não sabemos antecipadamente quando uma página na memória será escolhida para ser substituída, o sistema operacional normalmente cria o espaço na memória flash ou no disco para todas as páginas de um processo no momento em que ele cria o processo. Esse espaço é chamado de **área de swap**. Nesse momento, o sistema operacional também cria uma estrutura para registrar onde cada página virtual está armazenada no disco. Essa estrutura de dados pode ser parte da tabela de páginas ou pode ser uma estrutura de dados auxiliar indexada da mesma maneira que a tabela de páginas. A [Figura 5.28](#) mostra a organização quando uma única tabela contém o número de página física ou o endereço de disco.

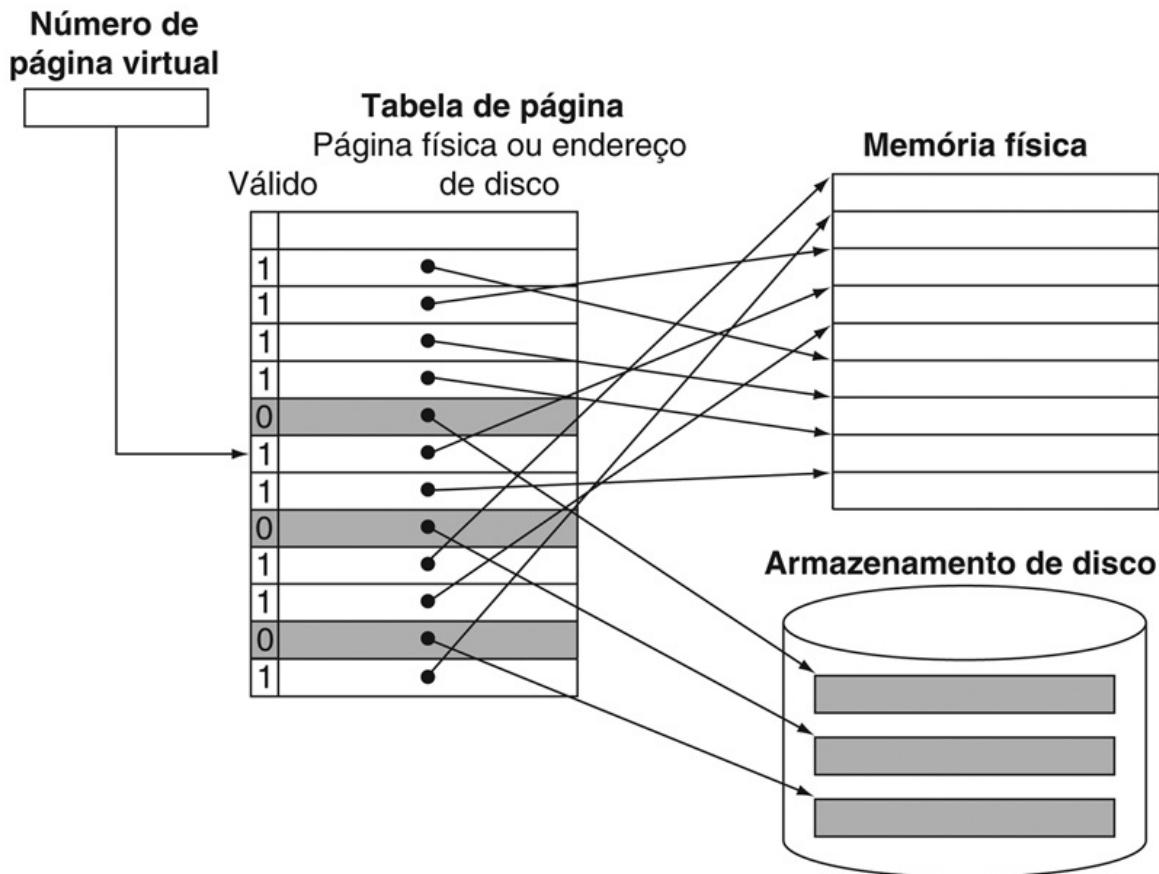


FIGURA 5.28 A tabela de páginas mapeia cada página na memória virtual em uma página na memória principal ou em uma página armazenada em disco, que é o próximo nível na hierarquia.

O número de página virtual é usado para indexar a tabela de páginas. Se o bit de validade estiver ligado, a tabela de páginas fornece o número de página física (ou seja, o endereço inicial da página na memória) correspondente à página virtual. Se o bit de validade estiver desligado, a página reside atualmente apenas no disco, em um endereço de disco especificado. Em muitos sistemas, a tabela de endereços de página física e endereços de página de disco, embora sendo logicamente uma única tabela, é armazenada em duas estruturas de dados separadas. As tabelas duplas se justificam, em parte, porque precisamos manter os endereços de disco de todas as páginas, mesmo que elas estejam atualmente na memória principal. Lembre-se de que as páginas na memória principal e as páginas no disco são idênticas em tamanho.

área de swap

O espaço no disco reservado para o espaço de memória virtual completo de um processo.

O sistema operacional também cria uma estrutura de dados que controla quais processos e quais endereços virtuais usam cada página física. Quando ocorre uma falta de página, se todas as páginas na memória principal estiverem em uso, o sistema operacional precisa escolher uma página para substituir. Como queremos minimizar o número de faltas de página, a maioria dos sistemas operacionais tenta escolher uma página que supostamente não será necessária no futuro próximo. Usando o passado para prever o futuro, os sistemas operacionais seguem o esquema de substituição LRU (Least Recently Used — usado menos recentemente), que mencionamos na [Seção 5.4](#). O sistema operacional procura a página usada menos recentemente, fazendo a suposição de que uma página que não foi usada por um longo período é menos provável de ser usada do que uma página acessada mais recentemente. As páginas substituídas são escritas na área de swap do disco. Caso você esteja curioso, o sistema operacional é apenas outro processo, e essas tabelas controlando a memória estão na memória; os detalhes dessa aparente contradição serão explicados em breve.

Interface hardware/software

Implementar um esquema de LRU completamente preciso é muito dispendioso, pois requer atualizar uma estrutura de dados a *cada* referência à memória. Como alternativa, a maioria dos sistemas operacionais aproxima a LRU monitorando que páginas foram e que páginas não foram usadas recentemente. Para ajudar o sistema operacional a estimar as páginas LRU, alguns computadores fornecem um **bit de referência** ou **bit de uso**, que é ligado sempre que uma página é acessada. O sistema operacional limpa periodicamente os bits de referência e, depois, os registra para que ele possa determinar quais páginas foram tocadas durante um determinado período. Com essas informações de uso, o sistema operacional pode selecionar uma página que está entre as referenciadas menos recentemente (detectadas tendo seu bit de referência desligado). Se esse bit não for fornecido pelo hardware, o sistema operacional precisará encontrar outra maneira de calcular quantas páginas foram acessadas.

bit de referência

Também chamado de **bit de uso**. Um campo que é ligado sempre quando uma página é acessada e usado para implementar LRU ou outros esquemas de substituição.

Detalhamento

Com um endereço virtual de 32 bits, páginas de 4 KiB e 4 bytes por entrada da tabela de páginas, podemos calcular o tamanho total da tabela de páginas:

$$\text{Número de entradas da tabela de páginas} = \frac{2^{32}}{2^{12}} = 2^{20}$$

$$\begin{aligned}\text{Tamanho da tabela de páginas} &= 2^{20} \text{ entradas da tabela de páginas} \\ &\times 2^2 \frac{\text{bytes}}{\text{entrada da tabela de páginas}} = 4 \text{ MiB}\end{aligned}$$

Ou seja, precisaríamos usar 4 MiB da memória para cada programa em execução em um dado momento. Essa quantidade não é ruim para um único processo. Mas, e se houver centenas de processos rodando, cada um com sua própria tabela de página? E como devemos tratar endereços de 64 bits, que por esse cálculo precisariam de 2^{52} palavras?

Diversas técnicas são usadas no sentido de reduzir a quantidade de armazenamento necessária para a tabela de páginas. As cinco técnicas a seguir visam reduzir o armazenamento máximo total necessário, bem como minimizar a memória principal dedicada às tabelas de páginas:

1. A técnica mais simples é manter um registrador de limite que restrinja o tamanho da tabela de páginas para um determinado processo. Se o número de página virtual se tornar maior do que o conteúdo do registrador de limite, entradas precisarão ser incluídas na tabela de páginas. Essa técnica permite que a tabela de páginas cresça à medida que um processo consome mais espaço. Assim, a tabela de páginas só será maior se o processo estiver usando muitas páginas do espaço de endereçamento virtual. Essa técnica exige que o espaço de

endereçamento se expanda apenas em uma direção.

2. Permitir o crescimento apenas em uma direção não é o bastante, já que a maioria das linguagens exige duas áreas cujo tamanho seja expansível: uma área contém a pilha e a outra contém o heap. Devido a essa dualidade, é conveniente dividir a tabela de páginas e deixá-la crescer do endereço mais alto para baixo, assim como do endereço mais baixo para cima. Isso significa que haverá duas tabelas de páginas separadas e dois limites separados. O uso de duas tabelas de páginas divide o espaço de endereçamento em dois segmentos. O bit mais significativo de um endereço normalmente determina que segmento — e, portanto, que tabela de páginas — deve ser usado para esse endereço. Como o segmento é especificado pelo bit de endereço mais significativo, cada segmento pode ter a metade do tamanho do espaço de endereçamento. Um registrador de limite para cada segmento especifica o tamanho atual do segmento, que cresce em unidades de páginas. Esse tipo de segmentação é usado por muitas arquiteturas, inclusive MIPS. Diferente do tipo de segmentação abordado na seção “Detalhamento” anterior, essa forma de segmentação é invisível ao programa de aplicação, embora não para o sistema operacional. A principal desvantagem desse esquema é que ele não funciona bem quando o espaço de endereçamento é usado de uma maneira esparsa e não como um conjunto contíguo de endereços virtuais.
3. Outro método para reduzir o tamanho da tabela de páginas é aplicar uma função de hashing no endereço virtual de modo que a estrutura de dados da tabela de páginas precise ser apenas do tamanho do número de páginas físicas na memória principal. Essa estrutura é chamada de *tabela de páginas invertida*. É claro que o processo de consulta é um pouco mais complexo com uma tabela de páginas invertida, já que não podemos mais simplesmente indexar a tabela de páginas.
4. Múltiplos níveis de tabelas de páginas também podem ser usados no sentido de reduzir a quantidade total de armazenamento para a tabela de páginas. O primeiro nível mapeia grandes blocos de tamanho fixo do espaço de endereçamento virtual, talvez de 64 a 256 páginas no total. Esses grandes blocos são, às vezes, chamados de segmentos, e essa tabela de mapeamento de primeiro nível é chamada de tabela de segmentos, embora os segmentos sejam invisíveis ao usuário. Cada entrada na tabela de segmentos indica se alguma página neste segmento

está alocada e, se estiver, aponta para uma tabela de páginas desse segmento. A tradução de endereços ocorre primeiramente olhando na tabela de segmentos, usando os bits de mais alta ordem do endereço. Se o endereço do segmento for válido, o próximo conjunto de bits mais significativos é usado para indexar a tabela de páginas indicada pela entrada da tabela de segmentos. Esse esquema permite que o espaço de endereçamento seja usado de uma maneira esparsa (vários segmentos não contíguos podem estar ativos), sem precisar alocar a tabela de páginas inteira. Esses esquemas são particularmente úteis com espaços de endereçamento muito grandes e em sistemas de software que exigem alocação não contígua. A principal desvantagem desse mapeamento de dois níveis é o processo mais complexo para a tradução de endereços.

5. A fim de reduzir a memória principal real consumida pelas tabelas de páginas, a maioria dos sistemas modernos também permite que as tabelas de páginas sejam paginadas. Embora isso pareça complicado, esse esquema funciona usando os mesmos conceitos básicos da memória virtual e simplesmente permite que as tabelas de páginas residam no espaço de endereçamento virtual. Entretanto, há alguns problemas pequenos mas cruciais, como uma série interminável de faltas de página, que precisam ser evitadas. A forma como esses problemas são resolvidos é um tema muito detalhado e, em geral, altamente específico ao processador. Em poucas palavras, esses problemas são evitados colocando todas as tabelas de páginas no espaço de endereçamento do sistema operacional e colocando pelo menos algumas das tabelas de páginas para o sistema operacional em uma parte da memória principal que é fisicamente endereçada e está sempre presente — e, portanto, nunca no disco.

E quanto às escritas?

A diferença entre o tempo de acesso à cache e à memória principal é de dezenas a centenas de ciclos, e os esquemas write-through podem ser usados, embora precisemos de um buffer de escrita para ocultar do processador a latência da escrita. Em um sistema de memória virtual, as escritas no próximo nível de hierarquia (disco) levam milhões de ciclos de clock de processador; portanto, construir um buffer de escrita para permitir que o sistema escreva diretamente no disco seria impraticável. Em vez disso, os sistemas de memória virtual precisam

usar write-back, realizando as escritas individuais para a página na memória e copiando a página novamente para o disco quando ela é substituída na memória.

Interface hardware/software

Um esquema write-back possui outra importante vantagem em um sistema de memória virtual. Como o tempo de transferência do disco é pequeno comparado com seu tempo de acesso, copiar de volta uma página inteira é muito mais eficiente do que escrever palavras individuais novamente no disco. Uma operação write-back, embora mais eficiente do que transferir páginas individuais, ainda é onerosa. Portanto, gostaríamos de saber se uma página *precisa* ser copiada de volta quando escolhemos substituí-la. Para monitorar se uma página foi escrita desde que foi lida para a memória, um *bit de modificação* (dirty bit) é acrescentado à tabela de páginas. O bit de modificação é ligado quando qualquer palavra em uma página é escrita. Se o sistema operacional escolher substituir a página, o bit de modificação indica se a página precisa ser escrita no disco antes que seu local na memória possa ser cedido a outra página. Logo, uma página modificada normalmente é chamada de “dirty page”.

Tornando a tradução de endereços rápida: a TLB

Como as tabelas de páginas são armazenadas na memória principal, cada acesso à memória por um programa pode levar, no mínimo, o dobro do tempo: um acesso à memória para obter o endereço físico e um segundo acesso para obter os dados. O segredo para melhorar o desempenho de acesso é basear-se na localidade da referência à tabela de páginas. Quando uma tradução para um número de página virtual é usada, ela provavelmente será necessária novamente no futuro próximo, pois as referências às palavras nessa página possuem localidade temporal e também espacial.

Assim, os processadores modernos incluem uma cache especial que controla as traduções usadas recentemente. Essa cache especial de tradução de endereços é tradicionalmente chamada de **TLB (translation-lookaside buffer)**, embora seria mais correto chamá-la de cache de tradução. A TLB corresponde àquele pequeno pedaço de papel que normalmente usamos para registrar o local de um conjunto de livros que consultamos no catálogo; em vez de pesquisar continuamente o catálogo inteiro, registramos o local de vários livros e usamos o

pedaço de papel como uma cache da biblioteca.

TLB (Translation-Lookaside Buffer)

Uma cache que monitora os mapeamentos de endereços recentemente usados para evitar um acesso à tabela de páginas.

A Figura 5.29 mostra que cada entrada de tag na TLB contém uma parte do número de página virtual, e cada entrada de dados da TLB contém um número de página física. Como não iremos mais acessar a tabela de páginas a cada referência, em vez disso acessaremos a TLB, que precisará incluir outros bits de status, como o bit de modificação e o bit de referência.

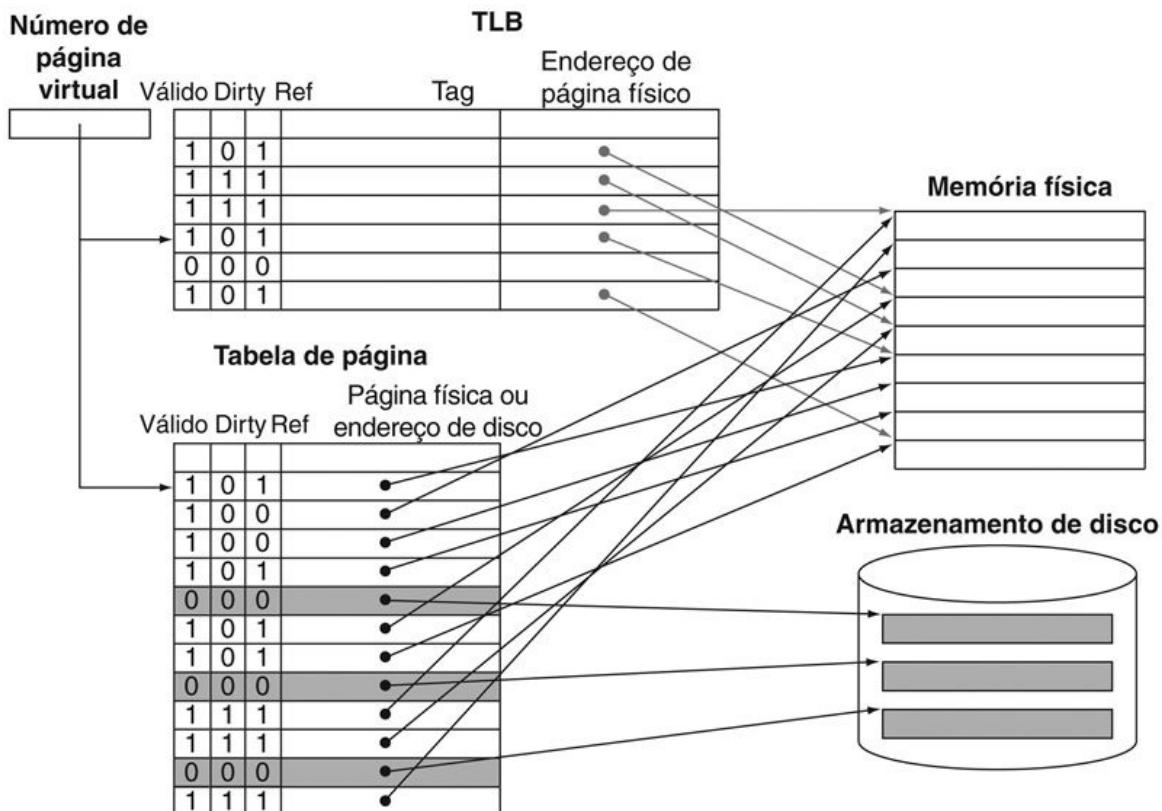


FIGURA 5.29 A TLB age como uma cache da tabela de páginas apenas para as entradas que mapeiam as páginas físicas.

A TLB contém um subconjunto dos mapeamentos de página virtual para física que estão na tabela de páginas. Os mapeamentos da TLB são mostrados em destaque. Como a TLB

é uma cache, ela precisa ter um campo tag. Se não houver uma entrada correspondente na TLB para uma página, a tabela de páginas precisa ser examinada. A tabela de páginas fornece um número de página física para a página (que pode, então, ser usado na construção de uma entrada da TLB) ou indica que a página reside em disco, caso em que ocorre uma falta de página. Como a tabela de páginas possui uma entrada para cada página virtual, nenhum campo tag é necessário; ou seja, ela *não* é uma cache.

Em cada referência, consultamos o número de página virtual na TLB. Se tivermos um acerto, o número de página física é usado para formar o endereço e o bit de referência correspondente é ligado. Se o processador estiver realizando uma escrita, o bit de modificação também é ligado. Se ocorrer uma falha na TLB, precisamos determinar se ela é uma falta de página ou simplesmente uma falha de TLB. Se a página existir na memória, então a falha de TLB indica apenas que a tradução está faltando. Nesse caso, o processador pode tratar a falha de TLB lendo a tradução da tabela de páginas para a TLB e, depois, tentando a referência novamente. Se a página não estiver presente na memória, então a falha de TLB indica uma falta de página verdadeira. Nesse caso, o processador chama o sistema operacional usando uma exceção. Como a TLB possui muito menos entradas do que o número de páginas na memória principal, as falhas de TLB serão muito mais frequentes do que as faltas de página verdadeiras.

As falhas de TLB podem ser tratadas no hardware ou no software. Na prática, com cuidado, pode haver pouca diferença de desempenho entre os dois métodos, uma vez que as operações básicas são iguais nos dois casos.

Depois que uma falha de TLB tiver ocorrido e a tradução faltando tiver sido recuperada da tabela de páginas, precisaremos selecionar uma entrada da TLB para substituir. Como os bits de referência e de modificação estão contidos na entrada da TLB, precisamos copiar esses bits de volta para a entrada da tabela de páginas quando substituirmos uma entrada. Esses bits são a única parte da entrada da TLB que pode ser modificada. O uso de write-back — ou seja, copiar de volta essas entradas no momento da falha e não quando são escritas — é muito eficiente, já que esperamos que a taxa de falhas da TLB seja pequena. Alguns sistemas usam outras técnicas para aproximar os bits de referência e de modificação, eliminando a necessidade de escrever na TLB, exceto para carregar uma nova entrada da tabela em caso de falha.

Alguns valores comuns para uma TLB poderiam ser:

- Tamanho da TLB: 16 a 512 entradas.
- Tamanho do bloco: uma a duas entradas da tabela de páginas (geralmente 4 a 8 bytes cada uma).
- Tempo de acerto: 0,5 a 1 ciclo de clock
- Penalidade de falha: 10 a 100 ciclos de clock
- Taxa de falhas: 0,01% a 1%

Os projetistas têm usado uma ampla gama de associatividades em TLBs. Alguns sistemas usam TLBs pequenas e totalmente associativas porque um mapeamento totalmente associativo possui uma taxa de falhas mais baixa; além disso, como a TLB é pequena, o custo de um mapeamento totalmente associativo não é tão alto. Outros sistemas usam TLBs grandes, normalmente com pequena associatividade. Com um mapeamento totalmente associativo, escolher a entrada a ser substituída se torna difícil, pois é muito caro implementar um esquema de LRU de hardware. Além do mais, como as falhas de TLB são muito mais frequentes do que as faltas de página e, portanto, precisam ser tratadas de modo mais econômico, não podemos utilizar um algoritmo de software caro, como para as falhas. Como resultado, muitos sistemas fornecem algum suporte para escolher aleatoriamente uma entrada a ser substituída. Veremos os esquemas de substituição mais detalhadamente na [Seção 5.8](#).

A TLB do Intrinsity FastMATH

Para ver essas ideias em um processador real, vamos dar uma olhada mais de perto na TLB do Intrinsity FastMATH. O sistema de memória usa páginas de 4 KiB e um espaço de endereçamento de 32 bits; portanto, o número de página virtual tem 20 bits de extensão, como no topo da [Figura 5.30](#). O endereço físico é do mesmo tamanho do endereço virtual. A TLB contém 16 entradas, é totalmente associativa e é compartilhada entre as referências de instruções e de dados. Cada entrada possui 64 bits de largura e contém uma tag de 20 bits (que é o número de página virtual para essa entrada de TLB), o número de página física correspondente (também 20 bits), um bit de validade, um bit de modificação e outros bits de contabilidade. Como na maioria dos sistemas MIPS, ela utiliza software para tratar das falhas de TLB.

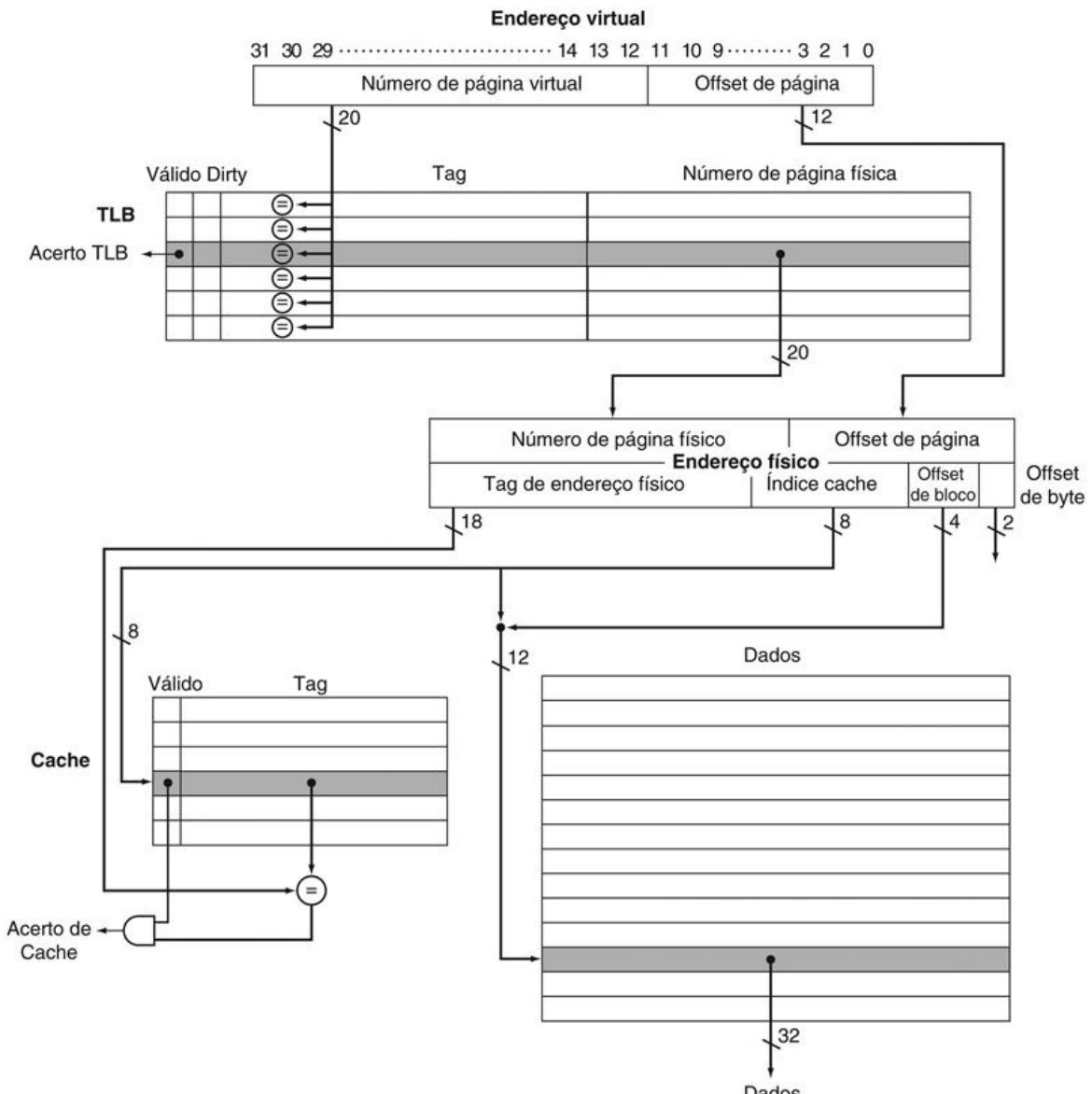


FIGURA 5.30 A TLB e a cache implementam o processo de ir de um endereço virtual para um item de dados no Intrinsity FastMATH.

Esta figura mostra a organização da TLB e a cache de dados considerando um tamanho de página de 4 KiB. Este diagrama focaliza uma leitura; a [Figura 5.31](#) descreve como tratar escritas. Repare que, diferente da [Figura 5.12](#), as RAMs de tag e de dados são divididas. Endereçando a longa, mas estreita, RAM de dados com o índice de cache concatenado com o offset de bloco, selecionamos a palavra desejada no bloco sem um multiplexador 16:1. Embora a cache seja diretamente mapeada, a TLB é totalmente associativa. A implementação de uma TLB totalmente associativa exige que toda tag TLB seja comparada com o número de página virtual, já que a entrada desejada pode

estar em qualquer lugar na TLB. (Ver memórias endereçáveis por conteúdo na seção “Detalhamento” da seção “Localizando um bloco na cache”.) Se o bit de validade da entrada correspondente estiver ligado, o acesso será um acerto de TLB e os bits do número de página física acrescidos aos bits do offset da página formarão o índice usado para acessar a cache.

A [Figura 5.30](#) mostra a TLB e uma das caches, enquanto a [Figura 5.31](#) mostra as etapas no processamento de uma requisição de leitura ou escrita. Quando ocorre uma falha de TLB, o hardware MIPS salva o número de página da referência em um registrador especial e gera uma exceção. A exceção chama o sistema operacional, que trata a falha no software. Para encontrar o endereço físico da página ausente, a rotina de falha de TLB indexa a tabela de páginas usando o número de página do endereço virtual e o registrador de tabela de páginas, que indica o endereço inicial da tabela de páginas do processo ativo. Usando um conjunto especial de instruções de sistema que podem atualizar a TLB, o sistema operacional coloca o endereço físico da tabela de páginas na TLB. Uma falha de TLB leva cerca de 13 ciclos de clock, considerando que o código e a entrada da tabela de páginas estejam na cache de instruções e na cache de dados, respectivamente. (Veremos o código TLB MIPS mais adiante). Uma falta de página verdadeira ocorre se a entrada da tabela de páginas não possuir um endereço físico válido. O hardware mantém um índice que indica a entrada recomendada a ser substituída, escolhida aleatoriamente.

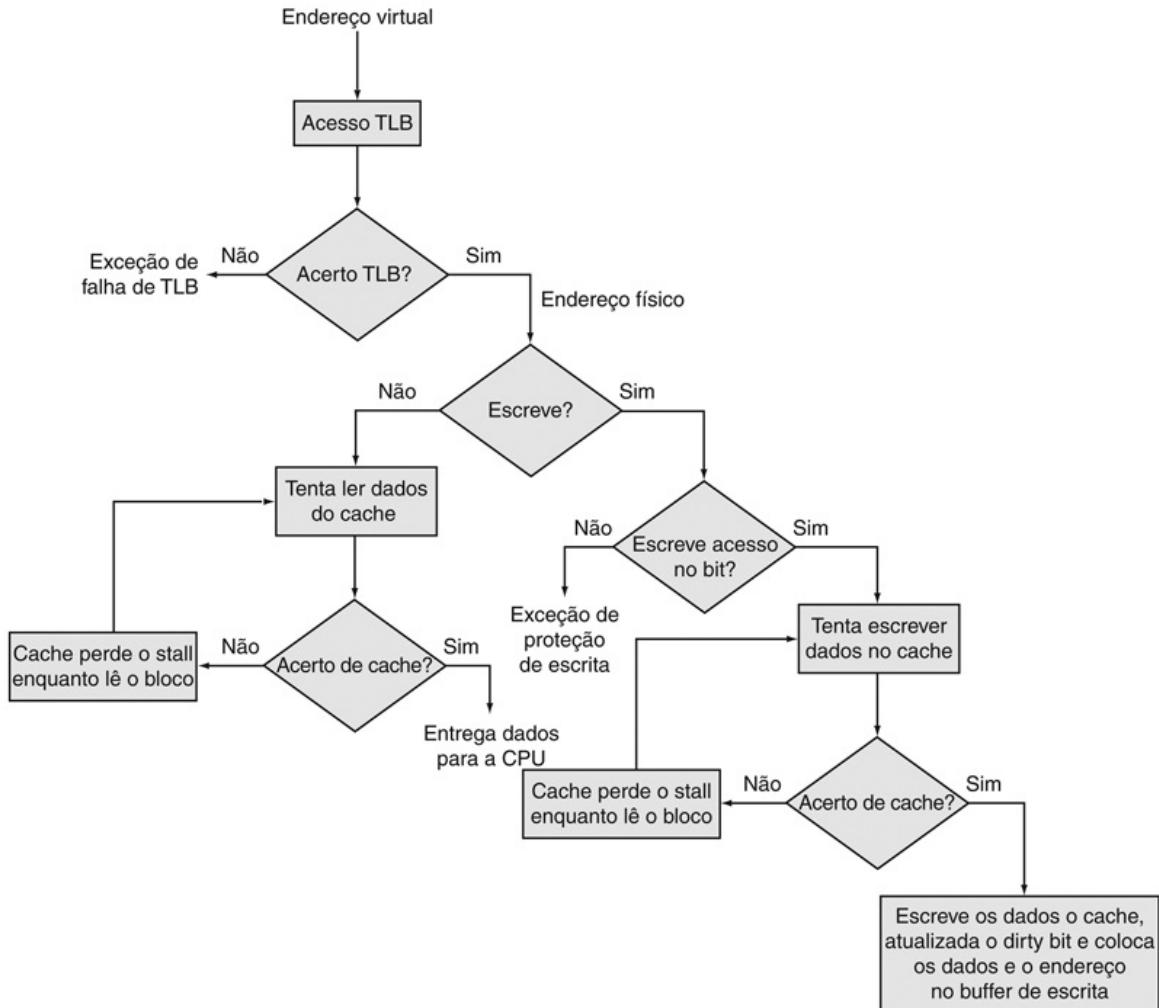


FIGURA 5.31 Processando uma leitura ou uma escrita direta na TLB e na cache do Intrinsity FastMATH.

Se a TLB gerar um acerto, a cache pode ser acessada com o endereço físico resultante. Para uma leitura, a cache gera um acerto ou uma falha e fornece os dados ou causa um stall enquanto os dados são trazidos da memória. Se a operação for uma escrita, uma parte da entrada de cache é substituída por um acerto e os dados são enviados ao buffer de escrita se considerarmos uma cache write-through. Uma falha de escrita é exatamente como uma falha de leitura exceto que o bloco é modificado após ser lido da memória. Uma cache write-back requer que as escritas liguem um dirty bit para o bloco de cache; além disso, um buffer de escrita é carregado com o bloco inteiro apenas em uma falha de leitura ou falha de escrita se o bloco a ser substituído estiver com o dirty bit ligado. Observe que um acerto de TLB e um acerto de cache são eventos independentes, mas um acerto de cache só pode ocorrer após um acerto de TLB, o que significa que os dados precisam estar presentes na

memória. A relação entre as falhas de TLB e as falhas de cache é examinada mais a fundo no exemplo a seguir e nos exercícios no final do capítulo.

Existe uma complicação extra para requisições de escrita: o bit de acesso de escrita na TLB precisa ser verificado. Esse bit impede que o programa escreva em páginas para as quais tenha apenas acesso de leitura. Se o programa tentar uma escrita e o bit de acesso de escrita estiver desligado, uma exceção é gerada. O bit de acesso de escrita faz parte do mecanismo de proteção, que abordaremos em breve.

Integrando memória virtual, TLBs e caches

Nossos sistemas de memória virtual e de cache funcionam em conjunto como uma hierarquia, de modo que os dados não podem estar na cache a menos que estejam presentes na memória principal. O sistema operacional desempenha um importante papel na manutenção dessa hierarquia removendo o conteúdo de qualquer página da cache quando decide migrar essa página para o disco. Ao mesmo tempo, o sistema operacional modifica as tabelas de páginas e a TLB de modo que uma tentativa de acessar quaisquer dados na página migrada gere uma falta de página.

Sob as circunstâncias ideais, um endereço virtual é traduzido pela TLB e enviado para a cache em que os dados apropriados são encontrados, recuperados e devolvidos ao processador. No pior caso, uma referência pode falhar em todos os três componentes da hierarquia de memória: a TLB, a tabela de páginas e a cache. O exemplo a seguir ilustra essas interações em mais detalhes.

Operação geral de uma hierarquia de memória

Exemplo

Em uma hierarquia de memória como a da Figura 5.30, que inclui uma TLB e uma cache organizada como mostrado, uma referência de memória pode encontrar três tipos de falhas diferentes: uma falha de TLB, uma falta de página e uma falha de cache. Considere todas as combinações desses três eventos com uma ou mais ocorrendo (sete possibilidades). Para cada possibilidade, diga se esse evento realmente pode ocorrer e sob que circunstâncias.

Resposta

A Figura 5.32 mostra as circunstâncias possíveis e se elas podem ou não surgir na prática.

TLB	Tabela de página	Cache	Possível? Se sim, sob quais circunstâncias?
Acerta	Acerta	Erra	Possível, apesar da tabela de página nunca ser conferida se a TLB acertar.
Erra	Acerta	Acerta	TLB erra, mas entrada é encontrada na tabela de página; depois de nova tentativa, dado é encontrado no cache.
Erra	Acerta	Erra	TLB erra, mas entrada é encontrada na tabela de página; depois de nova tentativa, dado erra o cache.
Erra	Erra	Erra	TLB erra e é seguido de erro de página; depois de nova tentativa, dado deve errar o cache.
Acerta	Erra	Erra	Impossível: não pode haver tradução na TLB se a página não está presente na memória.
Acerta	Erra	Acerta	Impossível: não pode haver tradução na TLB se a página não está presente na memória.
Erra	Erra	Acerta	Impossível: dados não podem ser permitidos no cache se a página não está presente na memória.

FIGURA 5.32 As possíveis combinações de eventos na TLB, no sistema de memória virtual e na cache.

Três dessas combinações são impossíveis e uma é possível (acerto de TLB, acerto de memória virtual, falha de cache), mas nunca detectada.



Detalhamento

A Figura 5.32 considera que todos os endereços de memória são traduzidos para endereços físicos antes que a cache seja acessada. Nessa organização, a cache é *fisicamente indexada* e *fisicamente rotulada* (tanto o índice quanto a tag de cache são endereços físicos em vez de virtuais). Nesse sistema, a quantidade de tempo para acessar a memória, considerando um acerto de cache, precisa acomodar um acesso de TLB e um acesso de cache; naturalmente, esses acessos podem ser **em pipeline**.

Como alternativa, o processador pode indexar a cache com um endereço que seja completa ou parcialmente virtual. Isso é chamado de **cache virtualmente endereçada** e usa tags que são endereços virtuais; portanto, esse tipo de cache é *virtualmente indexado* e *virtualmente rotulado*. Nestas caches, o hardware de tradução de endereço (TLB) não é usado durante o acesso de cache normal, já que a cache é acessada com um endereço virtual que não foi traduzido para um endereço físico. Isso tira a TLB do caminho crítico, reduzindo a latência da cache. Quando ocorre uma falha de cache, no entanto, o processador precisa traduzir o endereço para um endereço físico de modo que ele possa buscar o bloco de cache da memória principal.

cache virtualmente endereçada

Uma cache acessada com um endereço virtual em vez de um endereço físico.

Quando a cache é acessada com um endereço virtual e páginas são compartilhadas entre processos (que podem acessá-las com diferentes endereços virtuais), há a possibilidade de **aliasing**. O aliasing ocorre quando o mesmo objeto possui dois nomes — nesse caso, dois endereços virtuais para a mesma página. Esta ambiguidade cria um problema porque uma palavra nessa página pode ser colocada na cache em dois locais diferentes, cada um correspondendo a diferentes endereços virtuais. Essa ambiguidade permitiria que um programa escrevesse os dados sem que o outro programa soubesse que eles foram mudados. As caches endereçadas completamente por endereços virtuais apresentam limitações de projeto na cache e na TLB para reduzir o aliasing ou exigem que o sistema operacional (e possivelmente o usuário) tome ações para garantir que o aliasing não ocorra.

aliasing

Uma situação em que o mesmo objeto é acessado por dois endereços; pode ocorrer na memória virtual quando existem dois endereços virtuais para a mesma página física.

Uma conciliação comum entre esses dois pontos de projeto são as caches virtualmente indexadas (algumas vezes, usando apenas a parte do offset de página do endereço, que é um endereço físico, já que não é traduzida), mas usam tags físicas. Esses projetos, que são *virtualmente indexados, mas fisicamente rotulados*, tentam unir as vantagens de desempenho das caches virtualmente indexadas às vantagens da arquitetura mais simples de uma **cache fisicamente endereçada**. Por exemplo, não existe qualquer problema de aliasing nesse caso. A Figura 5.30 considerou um tamanho de página de 4 KiB, mas na realidade ela tem 16 KiB, de modo que o Intrinsity FastMATH pode usar esse truque. Para isso, é preciso haver uma cuidadosa coordenação entre o tamanho de página mínimo, o tamanho da cache e a associatividade.

cache fisicamente endereçada

Uma cache endereçada por um endereço físico.

Implementando proteção com memória virtual

Talvez a função mais importante da memória virtual atualmente seja permitir o compartilhamento de uma única memória principal por diversos processos, enquanto fornece proteção de memória entre esses processos e o sistema operacional. O mecanismo de proteção precisa garantir que, embora vários processos estejam compartilhando a mesma memória principal, um processo rebelde não pode escrever no espaço de endereçamento de outro processo do usuário ou no sistema operacional, intencionalmente ou não. O bit de acesso de escrita na TLB pode proteger uma página de ser escrita. Sem esse nível de proteção, os vírus de computador seriam ainda mais comuns.

Interface hardware/software

Para permitir que o sistema operacional implemente proteção no sistema de memória virtual, o hardware precisa fornecer pelo menos três capacidades básicas, resumidas a seguir. Observe que as duas primeiras são os mesmos requisitos necessários para as máquinas virtuais (Seção 5.6).

1. Suportar pelo menos dois modos que indicam se o processo em execução é de usuário ou de sistema operacional, normalmente chamado de processo **supervisor**, processo de **kernel** ou processo *executivo*.
2. Fornecer uma parte do estado do processador que um processo de usuário pode ler, mas não escrever. Isso inclui o bit de modo usuário/supervisor, que determina se o processador está no modo usuário ou supervisor, o ponteiro para a tabela de páginas e a TLB. Para escrever esses elementos, o sistema operacional usa instruções especiais que só estão disponíveis no modo supervisor.
3. Fornecer mecanismos pelos quais o processador pode passar do modo usuário para o modo supervisor e vice-versa. A primeira direção normalmente é conseguida por uma exceção de **chamada ao sistema**, implementada como uma instrução especial (*syscall* no conjunto de instruções MIPS) que transfere o controle para um local dedicado no espaço de código supervisor. Como em qualquer outra exceção, o contador de programa do ponto da chamada de sistema é salvo no PC de

exceção (EPC), e o processador é colocado no modo supervisor. Para retornar ao modo usuário da exceção, use a instrução *return from exception* (ERET), que retorna ao modo usuário e desvia para o endereço no EPC.

modo supervisor

Também chamado de **modo de kernel**. Um modo que indica que um processo executado é um processo do sistema operacional.

chamada ao sistema

Uma instrução especial que transfere o controle do modo usuário para um local dedicado no espaço de código supervisor, chamando o mecanismo de exceção no processo.

Usando esses mecanismos e armazenando as tabelas de páginas no espaço de endereçamento do sistema operacional, este pode mudar as tabelas de páginas enquanto impede que um processo do usuário as modifique, garantindo que um processo do usuário só possa acessar o armazenamento fornecido pelo sistema operacional.

Também queremos evitar que um processo leia os dados de outro processo. Por exemplo, não desejamos que o programa de um aluno leia as notas enquanto elas estiverem na memória do processador. Uma vez que começamos a compartilhar a memória principal, precisamos fornecer a capacidade de um processo proteger seus dados de serem lidos e escritos por outro processo; caso contrário, o compartilhamento da memória principal será um poço de permissividade!

Lembre-se de que cada processo possui seu próprio espaço de endereçamento virtual. Portanto, se o sistema operacional mantiver as tabelas de páginas organizadas de modo que as páginas virtuais independentes mapeiem as páginas físicas separadas, um processo não será capaz de acessar os dados de outro. É claro que isso exige que um processo de usuário seja incapaz de mudar o mapeamento da tabela de páginas. O sistema operacional pode garantir segurança se ele impedir que o processo do usuário modifique suas próprias tabelas de páginas. No entanto, o sistema operacional precisa ser capaz de

modificar as tabelas de páginas. Colocar as tabelas de páginas no espaço de endereçamento protegido do sistema operacional satisfaz a ambos os requisitos.

Quando os processos querem compartilhar informações de uma maneira limitada, o sistema operacional precisa assisti-los, já que o acesso às informações de outro processo exige mudar a tabela de páginas do processo que está acessando. O bit de acesso de escrita pode ser usado para restringir o compartilhamento apenas à leitura e, como o restante da tabela de páginas, esse bit pode ser mudado apenas pelo sistema operacional. Para permitir que outro processo, digamos, P1, leia uma página pertencente ao processo P2, P2 pediria ao sistema operacional para criar uma entrada na tabela de páginas para uma página virtual no espaço de endereço de P1 que aponte para a mesma página física que P2 deseja compartilhar. O sistema operacional poderia usar o bit de proteção de escrita a fim de impedir que P1 escrevesse os dados, se esse fosse o desejo de P2. Quaisquer bits que determinam os direitos de acesso a uma página precisam ser incluídos na tabela de páginas e na TLB, pois a tabela de páginas é acessada apenas em uma *falha* de TLB.

Detalhamento

Quando o sistema operacional decide deixar de executar o processo P1 para executar o processo P2 (o que chamamos de **troca de contexto** ou *troca de processo*), ele precisa garantir que P2 não possa ter acesso às tabelas de páginas de P1, porque isso comprometeria a proteção. Se não houver uma TLB, basta mudar o registrador de tabela de páginas de modo que aponte para a tabela de páginas de P2 (em vez da de P1); com uma TLB, precisamos limpar as entradas de TLB que pertencem a P1 — tanto para proteger os dados de P1 quanto para forçar a TLB a carregar as entradas para P2. Se a taxa de troca de processos fosse alta, isso poderia ser bastante ineficiente. Por exemplo, P2 poderia carregar apenas algumas entradas de TLB antes que o sistema operacional trocasse novamente para P1. Infelizmente, P1, então, descobriria que todas as suas entradas de TLB desapareceram e seria penalizado com falhas de TLB para recarregá-las. Esse problema ocorre porque os endereços virtuais usados por P1 e P2 são iguais e precisamos limpar a TLB a fim de evitar confundir esses endereços.

troca de contexto

Uma mudança no estado interno do processador para permitir que um

processo diferente use o processador, o que inclui salvar o estado necessário e retornar ao processo sendo atualmente executado.

Uma alternativa comum é estender o espaço de endereçamento virtual acrescentando um *identificador de processo* ou *identificador de tarefa*. O Intrinsity FastMATH possui um campo ID do espaço de endereçamento (ASID) de 8 bits para essa finalidade. Esse pequeno campo identifica o processo que está atualmente sendo executado; ele é mantido em um registrador carregado pelo sistema operacional quando muda de processo. O identificador de processo é concatenado com a parte da tag da TLB, de modo que um acerto de TLB ocorra apenas se o número de página *e* o identificador de processo corresponderem. Essa combinação elimina a necessidade de limpar a TLB, exceto em raras ocasiões.

Problemas semelhantes podem ocorrer para uma cache, já que, em uma troca de processo, a cache conterá dados do processo em execução. Esses problemas surgem de diferentes maneiras para caches física e virtualmente endereçadas; além disso, diversas soluções diferentes, como os identificadores de processo, são usadas para garantir que um processo obtenha seus próprios dados.

Tratando falhas de TLB e faltas de página

Embora a tradução de endereços físicos para virtuais com uma TLB seja simples quando temos um acerto de TLB, como já vimos, o tratamento de falhas de TLB e de faltas de página é mais complexo. Uma falha de TLB ocorre quando nenhuma entrada na TLB corresponde a um endereço virtual. Lembre-se de que uma falha de TLB pode indicar uma de duas possibilidades:

1. A página está presente na memória e precisamos apenas criar a entrada de TLB ausente.
2. A página não está presente na memória e precisamos transferir o controle para o sistema operacional a fim de lidar com uma falta de página.

O MIPS tradicionalmente trata uma falha de TLB por software. Ele traz a entrada da tabela de páginas da memória e, depois, executa novamente a instrução que causou a falha de TLB. Na reexecução, ele terá um acerto de TLB. Se a entrada da tabela de páginas indicar que a página não está na memória, dessa vez ele terá uma exceção de falta de página.

Tratar uma falha de TLB ou uma falta de página requer o uso do mecanismo

de exceção para interromper o processo ativo, transferir o controle ao sistema operacional e, depois, retomar a execução do processo interrompido. Uma falta de página será reconhecida em algum momento durante o ciclo de clock usado para acessar a memória. A fim de reiniciar a instrução após a falta de página ser tratada, o contador de programa da instrução que causou a falta de página precisa ser salvo. Assim como no [Capítulo 4](#), o *contador de programa de exceção* (EPC) é usado para conter esse valor.

Além disso, uma falha de TLB ou uma exceção de falta de página precisa ser sinalizada no final do mesmo ciclo de clock em que ocorre o acesso à memória, de modo que o próximo ciclo de clock começará o processamento da exceção, em vez de continuar a execução normal das instruções. Se a falta de página não fosse reconhecida nesse ciclo de clock, uma instrução load poderia substituir um registrador, e isso poderia ser desastroso quando tentássemos reiniciar a instrução. Por exemplo, considere a instrução `lw $1, 0($1)`: o computador precisa ser capaz de impedir que o estágio de escrita do resultado do pipeline ocorra; caso contrário, ele não poderia reiniciar corretamente a instrução, já que o conteúdo de `$1` teria sido destruído. Uma complicação parecida surge nos stores. Precisamos impedir que a escrita na memória realmente seja concluída quando há uma falta de página; isso normalmente é feito desativando a linha de controle de escrita para a memória.

Interface hardware/software

Entre o momento em que começamos a executar o tratamento de exceção no sistema operacional e o momento em que o sistema operacional salvou todo o estado do processo, o sistema operacional se torna particularmente vulnerável. Por exemplo, se outra exceção ocorresse quando estivéssemos processando a primeira exceção no sistema operacional, a unidade de controle substituiria o contador de programa de exceção, tornando impossível voltar para a instrução que causou a falta de página! Podemos evitar este desastre fornecendo a capacidade de **desabilitar** e **habilitar exceções**. Assim que uma exceção ocorre, o processador liga um bit que desabilita todas as outras exceções; isso poderia acontecer ao mesmo tempo em que o processador liga o bit de modo supervisor. O sistema operacional, então, salva o estado apenas suficiente para lhe permitir se recuperar se outra exceção ocorrer — a saber, os registradores do *contador de programa de exceção* (EPC) e Cause. EPC e Cause são dois dos registradores de controle especiais que ajudam com exceções, falhas de

TLB e faltas de página; a Figura 5.33 mostra o restante. O sistema operacional, então, pode habilitar novamente as exceções. Essas etapas asseguram que as exceções não façam com que o processador perca qualquer estado e, portanto, sejam incapazes de reiniciar a execução da instrução interruptora.

Registrador	Número de registradores CPO	Descrição
EPC	14	Onde reiniciar depois de exceção
Cause	13	Causa da exceção
BadVAddr	8	Endereço que causou exceção
Index	0	Local na TLB a ser lido ou escrito
Random	1	Local pseudorrandômico no TLB
EntryLo	2	Endereço de página física e flags
EntryHi	10	Endereço de página virtual
Context	4	Endereço de tabela de página e número de página

FIGURA 5.33 Registradores de controle MIPS.

Considera-se que estes estejam no coprocessador 0, e por isso são lidos com `mfc0` e escritos com `mtc0`.

habilitar exceção

Também chamado de “habilitar interrupção”. Uma ação ou sinal que controla se o processo responde ou não a uma exceção; necessário para evitar a ocorrência de exceções durante intervalos antes que o processador tenha seguramente salvado o estado necessário para a reinicialização.

Uma vez que o sistema operacional conhece o endereço virtual que causou a falta de página, ele precisa completar três etapas:

1. Consultar a entrada de tabela de páginas usando o endereço virtual e encontrar o local em disco da página referenciada.
2. Escolher uma página física a ser substituída; se a página escolhida estiver com o bit de modificação ligado, ela precisará ser escrita no disco antes que possamos definir uma nova página virtual para essa página física.
3. Iniciar uma leitura de modo a trazer a página referenciada do disco para a página física escolhida.

É claro que essa última etapa levará milhões de ciclos de clock de processador

(assim como a segunda, se a página substituída estiver com o bit de modificação ligado); portanto, o sistema operacional normalmente selecionará outro processo para executar no processador até que o acesso ao disco seja concluído. Como o sistema operacional salvou o estado do processo, ele pode passar o controle do processador à vontade para outro processo.

Quando a leitura da página do disco está completa, o sistema operacional pode restaurar o estado do processo que causou originalmente a falta de página e executar a instrução que retorna da exceção. Essa instrução passará o processador do modo kernel para o modo usuário, bem como restaurará o contador de programa. O processo do usuário, então, reexecuta a instrução que causou a falta de página, acessa a página requisitada com sucesso e continua a execução.

As exceções de falta de página para acessos a dados são difíceis de implementar corretamente em um processador, devido a uma combinação de três fatores:

1. Elas ocorrem no meio das instruções, diferente das faltas de página de instruções.
2. A instrução não pode ser completada, antes que a exceção seja tratada.
3. Após tratar a exceção, a instrução precisa ser reinicializada como se nada tivesse ocorrido.

Tornar **instruções reinicializáveis**, de modo que a exceção possa ser tratada e a instrução possa ser continuada, é relativamente fácil em uma arquitetura como o MIPS. Como cada instrução escreve apenas um item de dados e essa escrita ocorre no final do ciclo da instrução, podemos simplesmente impedir que a instrução seja concluída (não escrevendo) e reiniciar a instrução no começo.

instrução reinicializável

Uma instrução que pode retomar a execução, após uma exceção ser resolvida, sem que a exceção afete o resultado da instrução.

Vejamos o MIPS mais de perto. Quando uma falha de TLB ocorre, o hardware do MIPS salva o número de página da referência em um registrador especial chamado BadVAddr e gera uma exceção.

A exceção chama o sistema operacional, que trata a falha por software. O controle é transferido para o endereço 8000 0000_{hexa} (o local do **handler** da falha de TLB). A fim de encontrar o endereço físico para a página ausente, a rotina de

falha de TLB indexa a tabela de páginas usando o número de página do endereço virtual e o registrador de tabela de páginas, que indica o endereço inicial da tabela de páginas do processo ativo. Para tornar essa indexação rápida, o hardware do MIPS coloca tudo que você precisa no registrador especial Context: os 12 bits mais significativos têm o endereço da base da tabela de páginas e os próximos 18 bits têm o endereço virtual da página ausente. Como cada entrada de tabela de páginas possui uma palavra, os últimos dois bits são 0. Portanto, as duas primeiras instruções copiam o registrador Context para o registrador temporário do kernel \$k1 e, depois, carregam a entrada de tabela de páginas desse endereço em \$k1. Lembre-se de que \$k0 e \$k1 são reservados para uso do sistema operacional sem salvamento; um motivo importante para essa convenção é tornar rápido o handler de falha de TLB. A seguir está o código MIPS para um handler de falha de TLB típico:

handler

Nome de uma rotina de software chamada para “tratar” uma exceção ou interrupção.

```
TLBmiss:  
    mfc0  $k1,Context      # copia o endereço de PTE para temp $k1  
    lw    $k1,0($k1)        # coloca PTE em temp $k1  
    mtc0  $k1,EntryLo      # coloca PTE no registrador especial EntryLo  
    tlbwr                    # coloca EntryLo na entrada de TLB em Random  
    eret                     # retorna da exceção de falha de TLB
```

Como mostrado anteriormente, o MIPS possui um conjunto especial de instruções de sistema que atualiza a TLB. A instrução `tlbwr` copia o registrador de controle `EntryLo` para a entrada de TLB selecionada pelo registrador de controle `Random`. `Random` implementa uma substituição aleatória e, portanto, é basicamente um contador de execução livre. Uma falha de TLB leva cerca de 12 ciclos de clock.

Observe que o handler de falha de TLB não verifica se a entrada de tabela de páginas é válida. Como a exceção para a entrada de TLB ausente é muito mais frequente do que uma falta de página, o sistema operacional carrega a TLB da tabela de páginas sem examinar a entrada e reinicializa a instrução. Se a entrada for inválida, ocorre outra exceção diferente, e o sistema operacional reconhece a

falta de página. Esse método torna rápido o caso frequente de uma falha de TLB, com uma pequena penalidade de desempenho para o raro caso de uma falta de página.

Uma vez que o processo que gerou a falta de página tenha sido interrompido, ele transfere o controle para $8000\ 0180_{\text{hexa}}$, um endereço diferente do handler de falha de TLB. Esse é o endereço geral para exceção; a falha de TLB possui um ponto de entrada especial que reduz a penalidade para uma falha de TLB. O sistema operacional usa o registrador Cause de exceção a fim de diagnosticar a causa da exceção. Como a exceção é uma falta de página, o sistema operacional sabe que será necessário um processamento extenso. Portanto, diferente de uma falha de TLB, ele salva todo o estado do processo ativo. Esse estado inclui todos os registradores de uso geral e de ponto flutuante, o registrador de endereço de tabela de páginas, o EPC e o registrador Cause de exceção. Como os handlers de exceção normalmente não usam os registradores de ponto flutuante, o ponto de entrada geral não os salva, deixando isso para os poucos handlers que precisam deles.

A [Figura 5.34](#) esboça o código MIPS de um handler de exceção. Note que salvamos e restauramos o estado no código MIPS, tomando cuidado quando habilitamos e desabilitamos exceções, mas chamamos o código C para tratar da exceção em particular.

Salva Estado			
Salva GPR	addi \$k1,\$sp, -XCPSIZE sw \$sp, XCT_SP(\$k1) sw \$v0, XCT_VO(\$k1) ... sw \$ra, XCT_RA(\$k1)	# reserva espaço na pilha para o estado # salva \$sp na pilha # salva \$v0 na pilha # salva \$v1, \$ai, \$si, \$ti, ...na pilha # salva \$ra na pilha	
Salva Hi, Lo	mfhi \$v0 mflo \$v1 sw \$v0, XCT_HI(\$k1) sw \$v1, XCT_LO(\$k1)	# copia Hi # copia Lo # salva valor de Hi na pilha # salva valor de Lo na pilha	
Salva registradores de exceção	mfc0 \$a0, \$cr sw \$a0, XCT_CR(\$k1)	# copia registrador Cause # salva valor de \$cr na pilha	
Atribui novo valor a sp	mfco \$a3, \$sr sw \$a3, XCT_SR(\$k1)	# salva \$v1... # copia registrador de status # salva \$sr na pilha	
	move \$sp, \$k1	# sp = sp - XCPSIZE	
Habilita exceções aninhadas			
	andi \$v0, \$a3, MASK1 mtc0 \$v0, \$sr	# \$v0 = \$sr & MASK1, habilita exceções # \$sr = valor que habilita exceções	
Chama handler de exceção em C			
Chama código em C	move \$a0, \$sp	# argl = ponteiro para pilha de exceção	
Atribui novo valor a \$gp	jal xcpt_deliver move \$gp, GPINIT	# chama código em C para tratar exceção # \$gp aponta para área do heap	
Restaura estado			
Restaura a maioria dos registradores. Hi, Lo	move \$at, \$sp lw \$ra, XCT_RA(\$at) ... lw \$a0, XCT_A0(\$k1)	# valor temporário de \$sp # restaura \$ra da pilha # restaura \$t0, ..., \$a1 # restaura \$a0 da pilha	
Restaura registrador do status	lw \$v0, XCT_SR(\$at) li \$v1, MASK2 and \$v0, \$v0, \$v1 mtc0 \$v0, \$sr	# carrega \$sr antigo da pilha # máscara para inabilitar exceções # \$v0 = \$sr & MASK2, inabilita exceções # restaura o valor do registrador de status	
Retorna da exceção			
Restaura \$sp e o restante dos registradores usados como registradores temporários	lw \$sp, XCT_SP(\$at) lw \$v0, XCT_VO(\$at) lw \$v1, XCT_V1(\$at) lw \$k1, XCT_EPC(\$at) lw \$at, XCT_AT(\$at)	# restaura \$sp da pilha # restaura \$v0 da pilha # restaura \$v1 da pilha # copia \$epc antigo da pilha # restaura \$at da pilha	
Restaura EPC e retorna	mtc0 \$k1, \$epc eret \$ra	# restaura \$epc # retorna para a instrução interrompida	

FIGURA 5.34 Código MIPS para salvar e restaurar o estado em uma exceção.

O endereço virtual que causou a falta de página depende dessa exceção ter sido uma falta de instruções ou de dados. O endereço da instrução que gerou a falta está no EPC. Se ela fosse uma falta de página de instruções, o EPC manteria o endereço virtual da página que gerou a falta; caso contrário, o endereço virtual que gerou a falta pode ser calculado examinando a instrução (cujo endereço está no EPC) para encontrar o registrador base e o campo offset.

Detalhamento

Essa versão simplificada considera que o *stack pointer* (sp) é válido. Para evitar o problema de uma falta de página durante esse código de exceção de baixo nível, o MIPS separa uma parte do seu espaço de endereçamento que não pode ter faltas de página, chamada **não mapeada** (unmapped). O sistema operacional insere o código para o ponto de entrada do tratamento de exceções e a pilha de exceção na memória não mapeada. O hardware MIPS traduz os endereços virtuais de $8000\ 0000_{\text{hexa}}$ a $BFFF\ FFFF_{\text{hexa}}$ para endereços físicos simplesmente ignorando os bits superiores do endereço virtual, colocando, assim, esses endereços na parte inferior da memória física. Portanto, o sistema operacional coloca os pontos de entrada dos tratamentos de exceções e as pilhas de exceção na memória não mapeada.

não mapeada

Uma parte do espaço de endereçamento que não pode ter faltas de página.

Detalhamento

O código na Figura 5.34 mostra a sequência de retorno da exceção do MIPS-32. A arquitetura MIPS-I mais antiga usa rfe e jr em vez de eret.

Detalhamento

Para processadores com instruções mais complexas, que podem alcançar muitos locais de memória e escrever muitos itens de dados, tornar as instruções reiniciáveis é muito mais difícil. Processar uma instrução pode gerar uma série de faltas de página no meio da instrução. Por exemplo, os processadores x86 possuem instruções de movimento em bloco que alcançam milhares de palavras de dados. Nesses processadores, as instruções normalmente não podem ser reiniciadas desde o início, como fazemos para instruções MIPS. Em vez disso, a instrução precisa ser interrompida e mais tarde continuada no meio de sua execução. Retomar uma instrução no meio de sua execução normalmente exige salvar algum estado especial, processar a exceção e restaurar esse estado especial. Para que isso seja feito corretamente, é preciso haver uma coordenação cuidadosa e detalhada entre o código de tratamento de exceção no sistema operacional e o hardware.

Detalhamento

Em vez de pagar um nível extra de indireção em cada acesso à memória, o VMM mantém uma *tabela de página sombra* que mapeia diretamente o espaço de endereçamento virtual do guest ao espaço de endereços físicos do hardware. Detectando todas as modificações na tabela de página do guest, o VMM pode garantir que as entradas da tabela de páginas sombra, usadas pelo hardware para as traduções, correspondam àquelas do ambiente de OS do guest, com a exceção das páginas físicas corretas substituídas pelas páginas reais nas tabelas do guest. Logo, o VMM precisa interceptar qualquer tentativa pelo OS guest de mudar sua tabela de páginas ou acessar o ponteiro da tabela de páginas. Isso normalmente é feito protegendo-se a escrita das tabelas de páginas do guest e interceptando qualquer acesso ao ponteiro da tabela de páginas por um OS guest. Como já observamos, essa última ação acontece naturalmente quando o acesso ao ponteiro da tabela de páginas é uma operação privilegiada.

Detalhamento

A parte final da arquitetura a virtualizar é a E/S. Esta é, de longe, a parte mais difícil da virtualização do sistema, devido ao crescente número de dispositivos de E/S conectados ao computador e ao aumento da diversidade de tipos de dispositivo de E/S. Outra dificuldade é o compartilhamento de um dispositivo real entre diversas VMs, e ainda outra vem do suporte aos milhares de drivers de dispositivo que são exigidos, especialmente se diferentes OSs guest forem admitidos no mesmo sistema de VM. A ilusão da VM pode ser mantida dando-se a cada VM versões genéricas de cada tipo de driver de dispositivo de E/S, e depois, deixando que o VMM cuide da E/S real.

Detalhamento

Além de virtualizar o conjunto de instruções para uma máquina virtual, outro desafio é a virtualização da memória virtual, pois cada OS guest em cada máquina virtual gerencia seu próprio conjunto de tabelas de página. Para que isso funcione, o VMM separa as noções de *memória real* e *memória física* (que geralmente são tratadas como sinônimas), e torna a memória real um nível intermediário, separado, entre a memória virtual e a memória física. Alguns usam os termos *memória virtual*, *memória física* e *memória de*

máquina para indicar os mesmos três níveis. O OS guest mapeia a memória virtual à memória real por meio de suas tabelas de páginas, e as tabelas de páginas do VMM mapeiam a memória real do guest à memória física. A arquitetura da memória virtual é especificada ou por meio de tabelas de páginas, como no IBM VM/370 e no x86, ou por meio da estrutura de TLB, como no MIPS.

Resumo

Memória virtual é o nome para o nível da hierarquia de memória que controla a caching entre a memória principal e a memória secundária. A memória virtual permite que um único programa expanda seu espaço de endereçamento para além dos limites da memória principal. Mais importante, a memória virtual suporta o compartilhamento da memória principal entre vários processos simultaneamente ativos, de uma maneira protegida.

Gerenciar a hierarquia de memória entre a memória principal e o disco é uma tarefa difícil devido ao alto custo das faltas de página. Várias técnicas são usadas para reduzir a taxa de falhas:

1. As páginas são ampliadas para tirar proveito da localidade espacial e para reduzir a taxa de falhas.
2. O mapeamento entre endereços virtuais e endereços físicos, que é implementado com uma tabela de páginas, é feito totalmente associativo para que uma página virtual possa ser colocada em qualquer lugar na memória principal.
3. O sistema operacional usa técnicas, como LRU e um bit de referência, para escolher que páginas substituir.

Como as gravações no disco são dispendiosas, a memória virtual usa um esquema write-back e também monitora se uma página foi modificada (usando um bit de modificação) para evitar gravar páginas não alteradas novamente no disco.

O mecanismo de memória virtual fornece tradução de endereços de um endereço virtual usado pelo programa para o espaço de endereçamento físico usado no acesso à memória. Esta tradução de endereços permite compartilhamento protegido da memória principal e oferece várias vantagens adicionais, como a simplificação da alocação de memória. Para garantir que os processos sejam protegidos uns dos outros, é necessário que apenas o sistema operacional possa mudar as traduções de endereços, o que é implementado

impedindo que programas de usuário alterem as tabelas de páginas. O compartilhamento controlado das páginas entre processos pode ser implementado com a ajuda do sistema operacional e dos bits de acesso na tabela de páginas, que indicam se o programa do usuário possui acesso de leitura ou escrita a uma página.

Se um processador precisasse acessar uma tabela de páginas residente na memória para traduzir cada acesso, a memória virtual seria muito dispendiosa e as caches não teriam sentido! Em vez disso, uma TLB age como uma cache para traduções da tabela de páginas. Os endereços são, então, traduzidos do virtual para o físico usando as traduções na TLB.

As caches, a memória virtual e as TLBs se baseiam em um conjunto comum de princípios e políticas. A próxima seção aborda essa estrutura comum.

Entendendo o desempenho dos programas

Embora a memória virtual tenha sido criada para permitir que uma memória pequena aja como uma grande, a diferença de desempenho entre o disco e a memória significa que se um programa acessa rotineiramente mais memória virtual do que a memória física que possui, sua execução será muito lenta. Esse programa estaria continuamente trocando páginas entre a memória e o disco, o que chamamos de *thrashing*. O thrashing, embora raro, é um desastre quando ocorre. Se seu programa realiza thrashing, a solução mais fácil é executá-lo em um computador com mais memória ou comprar mais memória para o computador. Uma opção mais complexa é reexaminar suas estruturas de dados e algoritmo para ver se você pode mudar a localidade e, portanto, reduzir o número de páginas que seu programa usa simultaneamente. Este conjunto de páginas é informalmente chamado de *working set*.

Um problema de desempenho mais comum são as falhas de TLB. Como uma TLB pode tratar apenas de 32 a 64 entradas de página ao mesmo tempo, um programa poderia facilmente ver uma alta taxa de falhas de TLB, já que o processador pode acessar menos de um quarto de megabyte diretamente: $64 \times 4 \text{ KiB} = 0,25 \text{ MiB}$. Por exemplo, as falhas de TLB normalmente são um problema para o Radix Sort. A fim de tentar amenizar esse problema, a maioria das arquiteturas de computadores, agora suporta tamanhos de página variáveis. Por exemplo, além da página de 4 KiB padrão, o hardware do MIPS suporta páginas de 16 KiB, 64 KiB, 256 KiB, 1 KiB, 4 KiB, 16 KiB, 64 MiB e 256 MiB. Consequentemente, se um programa usa grandes tamanhos de

página, ele pode acessar mais memória diretamente sem falhas de TLB.

Na prática, o problema é fazer o sistema operacional permitir que os programas selecionem esses tamanhos de página maiores. Mais uma vez, a solução mais complexa para reduzir as falhas de TLB é reexaminar as estruturas de dados e os algoritmos no sentido de reduzir o working set de páginas; dada a importância dos acessos à memória para o desempenho e a frequência de falhas de TLB, alguns programas com grandes working sets foram recriados com esse objetivo.

Verifique você mesmo

Associe os termos à esquerda com as definições correspondentes na coluna da direita.

1. Cache L1	a. Uma cache para uma cache.
2. Cache L2	b. Uma cache para discos.
3. Memória principal	c. Uma cache para uma memória principal.
4. TLB	d. Uma cache para entradas de tabela de páginas.

5.8. Uma estrutura comum para hierarquias de memória

Agora você reconhece que os diferentes tipos de hierarquias de memória compartilham muita coisa. Embora muitos aspectos das hierarquias de memória difiram quantitativamente, muitas das políticas e recursos que determinam como uma hierarquia funciona são semelhantes em qualidade. A Figura 5.35 mostra como algumas características quantitativas das hierarquias de memória podem diferir. No restante desta seção, discutiremos os aspectos operacionais comuns das hierarquias de memória e como determinar seu comportamento. Examinaremos essas políticas como uma série de questões que se aplicam entre quaisquer dos níveis de uma hierarquia de memória, embora usemos principalmente terminologia de caches por motivo de simplicidade.

Recurso	Valores típicos para caches L1	Valores típicos para caches L2	Valores típicos para memória paginada	Valores típicos para uma TLB
Tamanho total em blocos	250–2000	2.500–25.000	16.000–250.000	40–1024
Tamanho total em kilobytes	16–64	125–2000	1.000.000–1.000.000.000	0,25–16
Tamanho do bloco em bytes	16–64	64–128	4000–64.000	4–32
Penalidade da falha em clocks	10–25	100–1000	10.000.000–100.000.000	10–1000
Taxas de falha (global para L2)	2%–5%	0,1%–2%	0,00001%–0,0001%	0,01%–2%

FIGURA 5.35 Os principais parâmetros quantitativos do projeto que caracterizam os principais elementos da hierarquia de memória em um computador.

Estes são valores típicos para esses níveis em 2012. Embora o intervalo de valores seja grande, isso ocorre parcialmente porque muitos dos valores que mudaram com o tempo estão relacionados; por exemplo, à medida que as caches se tornam maiores para contornar maiores penalidades de falha, os tamanhos de bloco também crescem. Embora não seja mostrado, os microprocessadores de servidor de hoje possuem caches L3, que podem ter de 2 a 8 MiB e conter muito mais blocos do que as caches L2. Caches L3 reduzem a penalidade por falta de cache L2 para 30 a 40 ciclos de clock.

Questão 1: onde um bloco pode ser colocado?

Vimos que o posicionamento de bloco no nível superior da hierarquia pode utilizar diversos esquemas, do diretamente mapeado ao associativo por conjunto e ao totalmente associativo. Como já dissemos, toda essa faixa de esquemas pode ser imaginada como variações em um esquema associativo por conjunto, no qual o número de conjuntos e o número de blocos por conjunto variam:

Nome do esquema	Número de conjuntos	Blocos por conjunto
Mapeamento Direto	Número de blocos na cache	1
Associativo por conjunto	Número de blocos na cache Associatividade	Associatividade (normalmente 2 a 16)
Totalmente associativo	1	Número de blocos na cache

A vantagem de aumentar o grau de associatividade é que normalmente isso diminui a taxa de falhas. A melhoria da taxa de falhas deriva da redução das falhas que disputam o mesmo local. Examinaremos essas falhas mais detalhadamente em breve. Antes, vejamos quanta melhoria é obtida. A Figura 5.36 mostra as taxas de falhas para diversos tamanhos de cache enquanto a associatividade varia de mapeamento direto para a associatividade por conjunto

de oito vias, o que produz uma redução de 20% a 30% na taxa de falhas. Conforme crescem os tamanhos de cache, a melhoria relativa da associatividade aumenta apenas ligeiramente; como a perda geral de uma cache maior é menor, a oportunidade de melhorar a taxa de falhas diminui e a melhoria absoluta na taxa de falhas da associatividade é reduzida significativamente. As possíveis desvantagens da associatividade, como já mencionado, são o custo mais alto e o tempo de acesso mais longo.

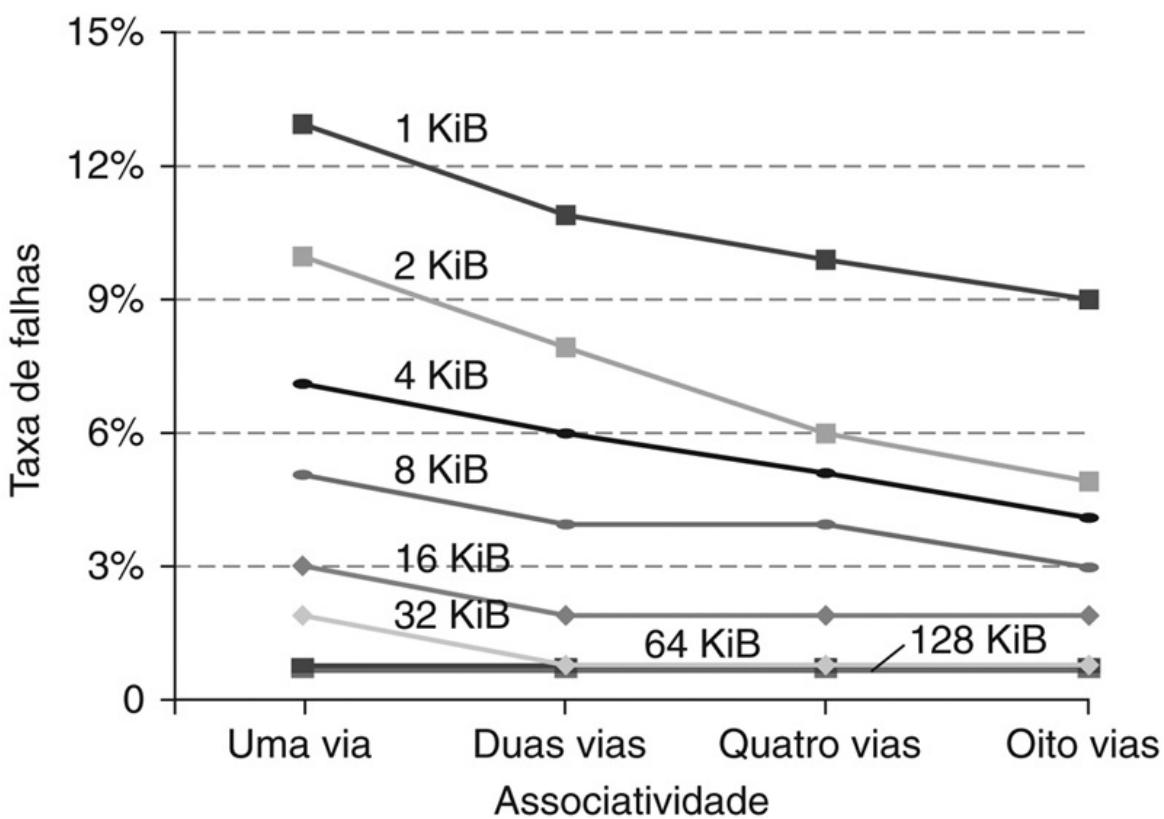


FIGURA 5.36 As taxas de falhas da cache de dados para cada um dos oito tamanhos melhora à medida que a associatividade aumenta.

Embora a vantagem em passar de associação por conjunto de uma via (mapeamento direto) para duas vias seja significativo, os benefícios de maior associatividade são menores (por exemplo, 1%-10% de melhoria passando de duas vias para quatro vias contra 20%-30% de melhoria passando de uma via para duas vias). Há ainda menos melhoria ao passar de quatro vias para oito vias, que, por sua vez, é muito próximo das taxas de falhas de uma cache totalmente associativa. As caches menores obtêm um benefício absoluto muito maior com a associatividade, pois a

taxa de falhas básica de uma cache pequena é maior. A [Figura 5.16](#) explica como esses dados foram coletados.

Questão 2: como um bloco é encontrado?

A escolha de como localizamos um bloco depende do esquema de posicionamento do bloco, já que isso determina o número de locais possíveis. Poderíamos resumir os esquemas da seguinte maneira:

Associatividade	Método de localização	Comparações necessárias
Mapeamento Direto	Indexação	1
Associativo por conjunto	Indexação do conjunto, pesquisa entre os elementos	Grau de associatividade
Total	Pesquisa de todas as entradas de cache	Tamanho da cache
	Tabela de consulta separada	0

A escolha entre os métodos mapeamento direto, associativo por conjunto ou totalmente associativo em qualquer hierarquia de memória dependerá do custo de uma falha comparado com o custo de implementar a associatividade, ambos em termos de tempo e de hardware extra. Incluir a cache L2 no chip permite uma associatividade muito mais alta, pois os tempos de acerto não são tão importantes, e o projetista não precisa se basear nos chips SRAM padrão como blocos de construção. As caches totalmente associativas são proibitivas exceto para pequenos tamanhos, nos quais o custo dos comparadores não é grande e as melhorias da taxa de falhas absoluta são as maiores.

Nos sistemas de memória virtual, uma tabela de mapeamento separada (a tabela de páginas) é mantida para indexar a memória. Além do armazenamento necessário para a tabela, usar um índice exige um acesso extra à memória. A escolha da associatividade total para o posicionamento de página e da tabela extra é motivada pelos seguintes fatos:

1. A associatividade total é benéfica, já que as falhas são muito dispendiosas.
2. A associatividade total permite que softwares usem esquemas sofisticados de substituição projetados para reduzir a taxa de falhas.
3. O mapa completo pode ser facilmente indexado sem a necessidade de pesquisa e de qualquer hardware extra.

Portanto, os sistemas de memória virtual quase sempre usam posicionamento totalmente associativo.

O posicionamento associativo por conjunto é muitas vezes usado para caches

e TLBs, no qual o acesso combina indexação e a pesquisa de um conjunto pequeno. Alguns sistemas têm usado caches com mapeamento direto devido às suas vantagens no tempo de acesso e da simplicidade. A vantagem no tempo de acesso ocorre porque a localização do bloco requisitado não depende de uma comparação. Essas escolhas de projeto dependem de muitos detalhes da implementação, como: se a cache é on-chip, a tecnologia usada para implementar a cache e o papel vital do tempo de acesso na determinação do tempo de ciclo do processador.

Questão 3: que bloco deve ser substituído em uma falha de cache?

Quando uma falha ocorre em uma cache associativa, precisamos decidir qual bloco substituir. Em uma cache totalmente associativa, todos os blocos são candidatos à substituição. Se a cache for associativa por conjunto, precisamos escolher entre os blocos do conjunto. É claro que a substituição é fácil em uma cache diretamente mapeada, porque existe apenas um candidato.

Existem duas principais estratégias para substituição nas caches associativas por conjunto ou totalmente associativas:

- *Substituição aleatória*: os blocos candidatos são selecionados aleatoriamente, talvez usando alguma assistência do hardware. Por exemplo, o MIPS suporta substituição aleatória para falhas de TLB.
- *Substituição LRU (Least Recently Used)*: o bloco substituído é o que não foi usado há mais tempo.

Na prática, o LRU é muito oneroso de ser implementado para hierarquias com mais do que um pequeno grau de associatividade (geralmente, de dois a quatro), já que é oneroso controlar o uso das informações. Mesmo para a associatividade por conjunto de quatro vias, o LRU normalmente é aproximado — por exemplo, monitorando qual par de blocos é o LRU (o que requer 1 bit) e, depois, monitorando qual bloco em cada par é o LRU (o que requer 1 bit por par).

Para maior associatividade, o LRU é aproximado ou a substituição aleatória é usada. Nas caches, o algoritmo de substituição está no hardware, o que significa que o esquema deve ser fácil de implementar. A substituição aleatória é simples de construir em hardware e, para uma cache associativa por conjunto de duas vias, a substituição aleatória possui uma taxa de falhas cerca de 1,1 vez mais alta do que a substituição LRU. Conforme as caches se tornam maiores, a taxa de falhas para as duas estratégias de substituição cai e a diferença absoluta se torna

pequena. Na verdade, a substituição aleatória, algumas vezes, pode ser melhor do que as aproximações simples de LRU que são facilmente implementadas em hardware.

Na memória virtual, alguma forma de LRU é sempre aproximada, já que mesmo uma pequena redução na taxa de falhas pode ser importante quando o custo de uma falha é enorme. Os bits de referência ou funcionalidade equivalente costumam ser fornecidos para facilitar que o sistema operacional monitore um conjunto de páginas usadas menos recentemente. Como as falhas são muito caras e relativamente raras, é aceitável aproximar essa informação, especialmente, em nível de software.

Questão 4: o que acontece em uma escrita?

Uma importante característica de qualquer hierarquia de memória é como ela lida com as escritas. Já vimos as duas opções básicas:

- *Write-through*: as informações são escritas no bloco da cache e no bloco do nível inferior da hierarquia de memória (memória principal para uma cache). As caches na [Seção 5.3](#) usaram esse esquema.
- *Write-back*: as informações são escritas apenas no bloco da cache. O bloco modificado é escrito no nível inferior da hierarquia apenas quando ele é substituído. Os sistemas de memória virtual sempre usam write-back, pelas razões explicadas na [Seção 5.7](#).

Tanto write-back quanto write-through têm suas vantagens. As principais vantagens do write-back são as seguintes:

- As palavras individuais podem ser escritas pelo processador na velocidade em que a cache, não a memória, pode aceitar.
- Diversas escritas dentro de um bloco exigem apenas uma escrita no nível inferior da hierarquia.
- Quando blocos são escritos com write-back, o sistema pode fazer uso efetivo de uma transferência de alta largura de banda, já que o bloco inteiro é escrito. O write-through possui estas vantagens:
- As falhas são mais simples e baratas porque nunca exigem que um bloco seja escrito de volta no nível inferior.
- O write-through é mais fácil de ser implementado do que o write-back, embora, para ser prática, uma cache write-through precisaria usar um buffer de escrita.

Colocando em perspectiva

Embora as caches, as TLBs e a memória virtual inicialmente possam parecer muito diferentes, elas se baseiam nos mesmos dois princípios de localidade e podem ser entendidos examinando como lidam com quatro questões:

Questão 1:	Onde um bloco pode ser colocado?
Resposta:	Em um local (mapeamento direto), em alguns locais (associatividade por conjunto) ou em qualquer local (associatividade total).
Questão 2:	Como um bloco é encontrado?
Resposta:	Existem quatro métodos: indexação (como em uma cache diretamente mapeada), pesquisa limitada (como em uma cache associativa por conjunto), pesquisa completa (como em uma cache totalmente associativa) e tabela de consulta separada (como em uma tabela de páginas).
Questão 3:	Que bloco é substituído em uma falha?
Resposta:	Em geral, o bloco usado menos recentemente ou um bloco aleatório.
Questão 4:	Como as escritas são tratadas?
Resposta:	Cada nível na hierarquia pode usar write-through ou write-back.

Em sistemas de memória virtual, apenas uma política write-back é viável devido à longa latência de uma escrita no nível inferior da hierarquia. A taxa em que as escritas são geradas por um processador excederá a taxa em que o sistema de memória pode processá-las, até mesmo permitindo memórias física e logicamente mais largas, e modos *burst* para a DRAM. Como consequência, hoje em dia as caches de nível mais baixo geralmente usam uma estratégia write-back.

Os três Cs: um modelo intuitivo para entender o comportamento das hierarquias de memória

Nesta subseção, vamos examinar um modelo que esclarece as origens das falhas em uma hierarquia de memória e como as falhas serão afetadas por mudanças na hierarquia. Explicaremos as ideias em termos de caches, embora elas se apliquem diretamente a qualquer outro nível na hierarquia. Neste modelo, todas as falhas são classificadas em uma de três categorias (os **três Cs**):

modelo dos três Cs

Um modelo de cache em que todas as falhas são classificadas em uma de três categorias: falhas compulsórias, falhas de capacidade e falhas de conflito.

- **Falhas compulsórias:** são falhas de cache causadas pelo primeiro acesso a um bloco que nunca esteve na cache. Também são chamadas de **falhas de partida a frio**.
 - **Falhas de capacidade:** são falhas de cache causadas quando a cache não pode conter todos os blocos necessários durante a execução de um programa. As falhas de capacidade ocorrem quando os blocos são substituídos e, depois, recuperados.
 - **Falhas de conflito:** são falhas de cache que ocorrem em caches associativas por conjunto ou diretamente mapeadas quando vários blocos disputam o mesmo conjunto. As falhas de conflito são aquelas falhas em uma cache diretamente mapeada ou associativa por conjunto que são eliminadas em uma cache totalmente associativa do mesmo tamanho. Essas falhas de cache também são chamadas de **falhas de colisão**.

falha compulsória

Também chamada de **falha de partida a frio**. Uma falha de cache causada pelo primeiro acesso a um bloco que nunca esteve na cache.

falha de capacidade

Uma falha de cache que ocorre porque a cache, mesmo com associatividade total, não pode conter todos os blocos necessários para satisfazer à requisição.

falha de conflito

Também chamada de **falha de colisão**. Uma falha de cache que ocorre em uma cache associativa por conjunto ou diretamente mapeada quando vários blocos competem pelo mesmo conjunto e que são eliminados em uma cache totalmente associativa do mesmo tamanho.

A [Figura 5.37](#) mostra como a taxa de falhas se divide nas três origens. Essas origens de falhas podem ser diretamente atacadas mudando algum aspecto do projeto da cache. Como as falhas de conflito surgem diretamente da disputa pelo mesmo bloco de cache, aumentar a associatividade reduz as falhas de conflito. Entretanto, a associatividade pode aumentar o tempo de acesso, levando a um menor desempenho geral.

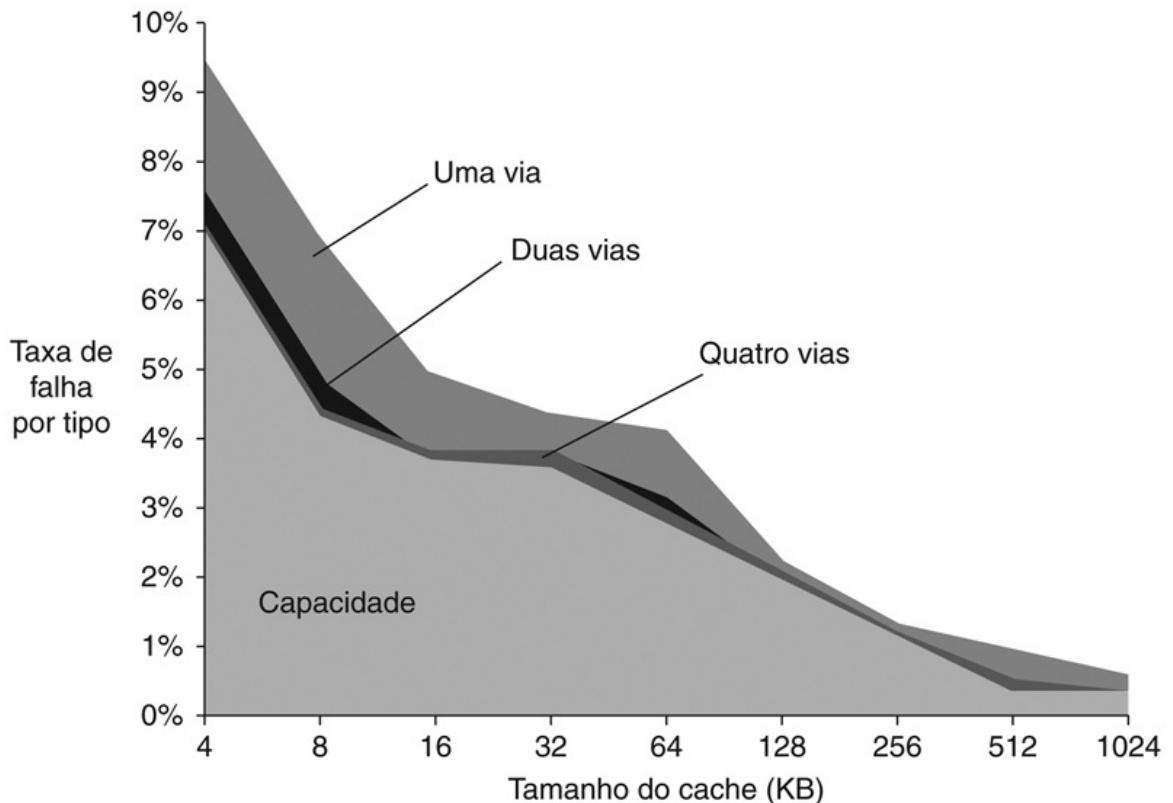


FIGURA 5.37 A taxa de falhas pode ser dividida em três origens de falha.

Este gráfico mostra a taxa de falhas total e seus componentes para uma faixa de tamanhos de cache. Esses dados são para os benchmarks de inteiro e ponto flutuante do SPEC CPU2000 e são da mesma fonte dos dados na [Figura 5.36](#). O componente da falha compulsória é de 0,006% e não pode ser visto nesse gráfico. O próximo componente é a taxa de falhas de capacidade, que depende do tamanho da cache. A parte do conflito, que depende da associatividade e do tamanho da cache, é mostrada para uma faixa de associatividades, de uma via a oito vias. Em cada caso, a seção rotulada corresponde ao aumento na taxa de falhas que ocorre quando a associatividade é alterada do próximo grau mais alto para o grau de associatividade rotulado. Por exemplo, a seção rotulada como *duas vias* indica as falhas adicionais surgindo quando a cache possui associatividade de dois em vez de quatro. Portanto, a diferença na taxa de falhas incorrida por uma cache diretamente mapeada em relação a uma cache totalmente associativa do mesmo tamanho é dada pela soma das seções rotuladas como *quatro vias*, *duas vias* e *uma via*. A diferença entre oito vias e quatro vias é tão pequena que mal pode ser vista nesse gráfico.

As falhas de capacidade podem facilmente ser reduzidas aumentando a cache; na verdade, as caches de segundo nível têm se tornado constantemente maiores durante muitos anos. É claro que, quando tornamos a cache maior, também precisamos ser cautelosos quanto ao aumento no tempo de acesso, que pode levar a um desempenho geral mais baixo. Por isso, as caches de primeiro nível cresceram lentamente ou nem isso.

Como as falhas compulsórias são geradas pela primeira referência a um bloco, a principal maneira de um sistema de cache reduzir o número de falhas compulsórias é aumentando o tamanho do bloco. Isso irá reduzir o número de referências necessárias para tocar cada bloco do programa uma vez, porque o programa consistirá em menos blocos de cache. Como já dissemos, aumentar demais o tamanho do bloco pode ter um efeito negativo sobre o desempenho devido ao aumento na penalidade de falha.

A decomposição das falhas nos três Cs é um modelo qualitativo útil. Nos projetos de cache reais, muitas das escolhas de projeto interagem, e mudar uma característica de cache frequentemente afetará vários componentes da taxa de falhas. Apesar dessas deficiências, esse modelo é uma maneira útil de adquirir conhecimento sobre o desempenho dos projetos de cache.

Colocando em perspectiva

A dificuldade de projetar hierarquias de memória é que toda mudança que melhore potencialmente a taxa de falhas, também pode afetar negativamente o desempenho geral, como mostra a Figura 5.38. Essa combinação de efeitos positivos e negativos é o que torna o projeto de uma hierarquia de memória interessante.

Mudança de projeto	Efeito sobre a taxa de falhas	Possível efeito negativo no desempenho
Aumentar o tamanho da cache	Diminui as falhas de capacidade	Pode aumentar o tempo de acesso
Aumentar a associatividade	Diminui a taxa de falhas devido a falhas de conflito	Pode aumentar o tempo de acesso
Aumentar o tamanho de bloco	Diminui a taxa de falhas para uma ampla faixa de tamanhos de bloco devido à localidade espacial	Aumenta a penalidade de falha. Blocos muito grandes podem aumentar a taxa de falhas

FIGURA 5.38 Dificuldades do projeto de hierarquias de memória.

Verifique você mesmo

Quais das seguintes afirmativas (se houver) normalmente são verdadeiras?

1. Não há um meio de reduzir as falhas compulsórias.
2. As caches totalmente associativas não possuem falhas de conflito.
3. Na redução de falhas, a associatividade é mais importante do que a capacidade.

5.9. Usando uma máquina de estado finito para controlar uma cache simples

Agora, podemos implementar o controle para uma cache, assim como implementamos o controle para os caminhos de dados de único ciclo e em pipeline, no [Capítulo 4](#). Esta seção começa com uma definição de uma cache simples e depois uma descrição das *máquinas de estado finito* (MEF). Ela termina com a MEF de um controlador para essa cache simples.

Uma cache simples

Vamos projetar um controlador para uma cache simples. Aqui estão as principais características da cache:

- Cache mapeada diretamente.
- Write-back usando alocação de escrita.
- O tamanho do bloco é de 4 palavras (16 bytes ou 128 bits).
- O tamanho da cache é de 16 KiB, de modo que ela mantém 1024 blocos.
- Endereços de 32 bits.
- A cache inclui um bit de validade e um bit de modificação por bloco.

Pela [Seção 5.3](#), podemos agora calcular os campos de um endereço para a cache:

- O índice da cache tem 10 bits.
- O offset do bloco tem 4 bits.
- O tamanho da tag tem $32 - (10 + 4)$ ou 18 bits.

Os sinais entre o processador e a cache são:

- 1 bit de sinal Read ou Write.
- 1 bit de sinal Valid, dizendo se existe uma operação de cache ou não.
- 32 bits de endereço.

- 32 bits de dados do processador à cache.
- 32 bits de dados da cache ao processador.
- 1 bit de sinal Ready, dizendo que a operação da cache está completa.

A interface entre a memória e a cache tem os mesmos campos que entre o processador e a cache, exceto que os campos de dados agora têm 128 bits de largura. A largura de memória extra geralmente é encontrada nos microprocessadores de hoje, que lida com palavras de 32 bits ou 64 bits no processador, enquanto o controlador da DRAM normalmente tem 128 bits. Fazer com que o bloco de cache combine com a largura da DRAM simplificou o projeto. Aqui estão os sinais:

- 1 bit de sinal Read ou Write.
- 1 bit de sinal Valid, dizendo se existe uma operação de memória ou não.
- 32 bits de endereço.
- 128 bits de dados da cache à memória.
- 128 bits de dados da memória à cache.
- 1 bit de sinal Ready, dizendo que a operação de memória está completa.

Observe que a interface para a memória não é um número fixo de ciclos. Consideramos um controlador de memória que notificará a cache por meio do sinal Ready quando a leitura ou escrita na memória terminar.

Antes de descrever o controlador de cache, precisamos revisar as máquinas de estados finitos, que nos permitem controlar uma operação que pode utilizar múltiplos ciclos de clock.

Máquinas de estados finitos

A fim de projetar a unidade de controle para o caminho de dados de único ciclo, usamos um conjunto de tabelas verdade que especificava a configuração dos sinais de controle com base na classe de instrução. Para uma cache, o controle é mais complexo porque a operação pode ser uma série de etapas. O controle para uma cache precisa especificar os sinais a serem definidos em qualquer etapa e a próxima etapa na sequência.

máquina de estados finitos

Uma função lógica sequencial consistindo em um conjunto de entradas e saídas, uma função de próximo estado que mapeia o estado atual e as entradas para um novo estado, e uma função de saída que mapeia o estado atual e, possivelmente, as entradas para um conjunto de saídas ativas.

O método de controle multietapas mais comum é baseado em **máquinas de estados finitos**, que normalmente são representadas graficamente. Uma máquina de estado finito consiste em um conjunto de estados e instruções sobre como alterar os estados. As instruções são definidas por uma **função de próximo estado**, que mapeia o estado atual e as entradas de um novo estado. Quando usamos uma máquina de estado finito para controle, cada estado também especifica um conjunto de saídas que são declaradas quando a máquina está nesse estado. A implementação de uma máquina de estados finitos normalmente considera que todas as saídas que não estão declaradas explicitamente têm as declarações retiradas. De modo semelhante, a operação correta do caminho de dados depende do fato de que um sinal que não é declarado explicitamente tem a declaração retirada, em vez de atuar como um don't care.

função de próximo estado

Uma função combinacional que, dadas as entradas e o estado atual, determina o próximo estado de uma máquina de estados finitos.

Os controles multiplexadores são ligeiramente diferentes, pois selecionam uma das entradas, seja ela 0 ou 1. Assim, na máquina de estado finito, sempre especificamos a definição de todos os controles multiplexadores com que nos importamos. Quando implementamos a máquina de estado finito com lógica, a definição de um controle como 0 pode ser o default e, portanto, pode não exigir quaisquer portas lógicas. Um exemplo simples de uma máquina de estados finitos aparece no Apêndice B, e se você não estiver familiarizado com o conceito de uma máquina de estado finito, pode querer examinar o Apêndice B antes de prosseguir.

Uma máquina de estados finitos pode ser implementada com um registrador temporário que mantém o estado atual e um bloco de lógica combinatória que determina os sinais do caminho de dados a serem ativados e o próximo estado. A [Figura 5.39](#) mostra como essa implementação poderia se parecer. Na Seção B.3, a lógica de controle combinacional para uma máquina de estados finitos é implementada com uma ROM (Read-Only Memory) e uma PLA (Programmable Logic Array). Veja também no Apêndice B uma descrição desses elementos lógicos.

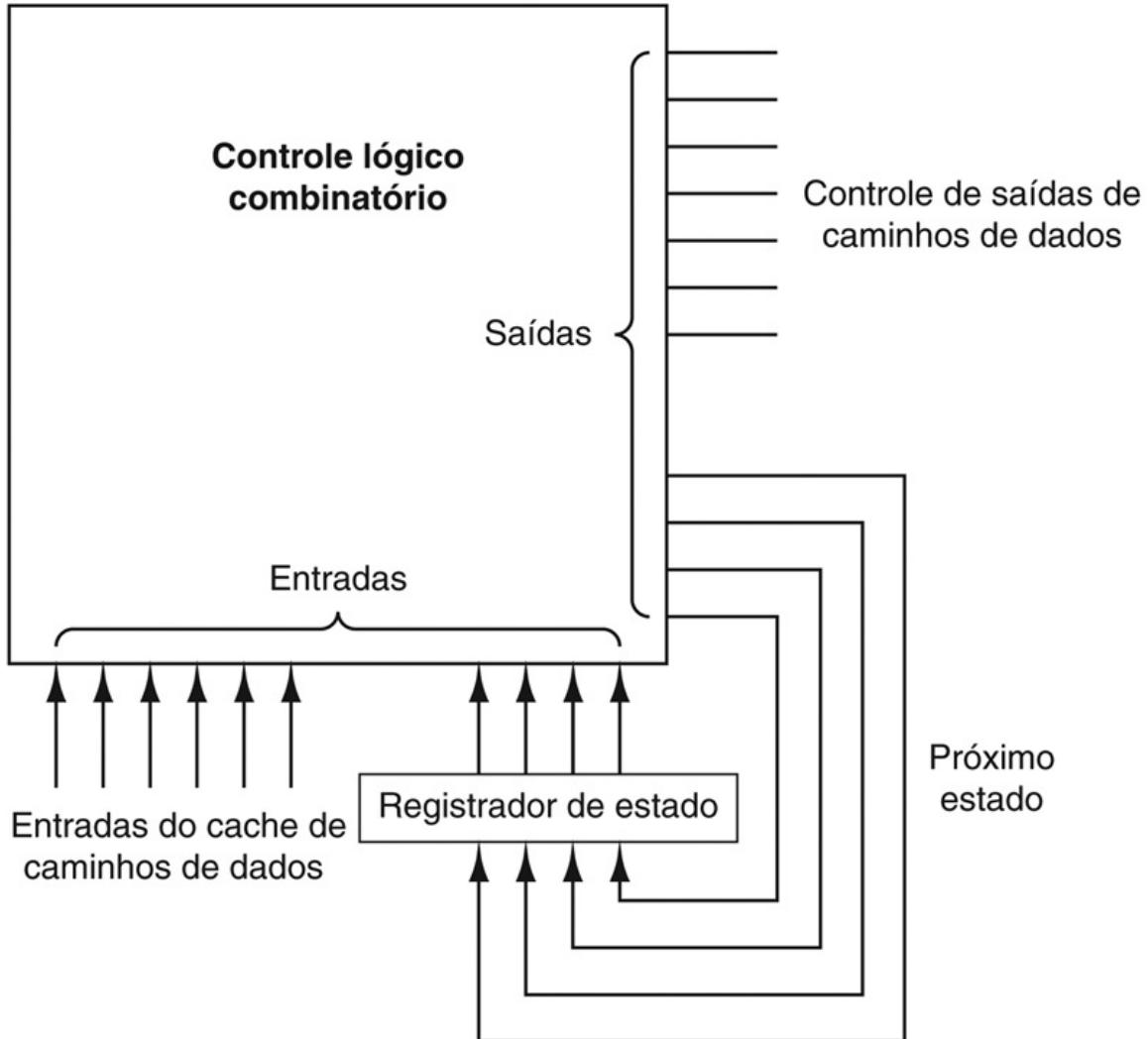


FIGURA 5.39 Controladores da máquina de estados finitos normalmente são implementados com um bloco de lógica combinacional e um registrador para manter o estado atual.

As saídas da lógica combinacional são o número do próximo estado e os sinais de controle a serem ativados para o estado atual. As entradas da lógica combinacional são o estado atual e quaisquer entradas usadas para determinar o próximo estado. Observe que, na máquina de estados finitos utilizada neste capítulo, as saídas dependem apenas do estado atual e não das entradas. A seção *Detalhamento* explica isso em minúcias.

Detalhamento

Observe que este projeto simples é chamado de cache *com bloqueio*, pois o processador precisa esperar até que a cache tenha concluído a solicitação.

Detalhamento

O estilo da máquina de estados finitos neste livro é chamado de máquina de Moore, em homenagem a Edward Moore. Sua característica identificadora é que a saída depende apenas do estado atual. Com uma máquina de Moore, a caixa rotulada como lógica de controle combinacional pode ser dividida em duas partes. Uma parte tem a saída de controle e apenas a entrada de estado, enquanto a outra tem apenas a saída do próximo estado.

Um estilo alternativo de máquina é uma máquina de Mealy, em homenagem a George Mealy. A máquina de Mealy permite que a entrada e o estado atual sejam usados para determinar a saída. As máquinas de Moore possuem vantagens de implementação em potencial na velocidade e no tamanho da unidade de controle. As vantagens na velocidade ocorrem porque as saídas de controle, que são necessárias logo no começo no ciclo de clock, não dependem das entradas, mas somente do estado atual. No Apêndice B, quando a implementação dessa máquina de estado finito é levada às portas lógicas, a vantagem do tamanho pode ser vista com clareza. A desvantagem em potencial de uma máquina de Moore é que ela pode exigir estados adicionais. Por exemplo, em situações em que existe uma diferença de um estado entre duas sequências de estados, a máquina de Mealy pode unificar os estados, fazendo com que as saídas dependam das entradas.

MEF para um controlador de cache simples

A [Figura 5.40](#) mostra os quatro estados do nosso controlador de cache simples:

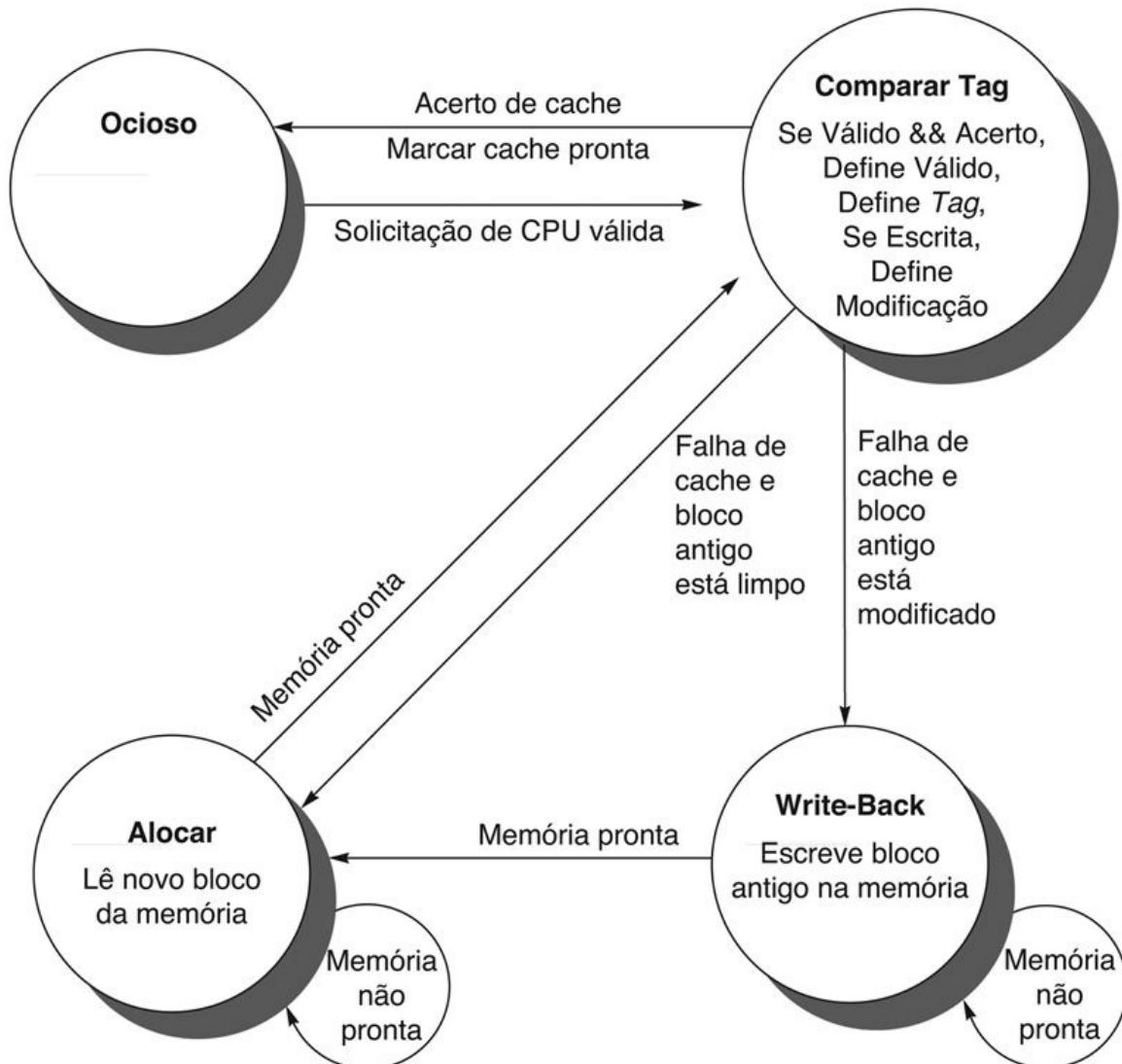


FIGURA 5.40 Quatro estados do controlador simples.

- **Ocioso**: Esse estado espera uma solicitação de leitura ou escrita válida do processador, que move a MEF para o estado **Comparar Tag**.
- **Comparar Tag**: Como o nome sugere, este estado testa se a leitura ou escrita solicitada é um acerto ou uma falha. A parte de índice do endereço seleciona a tag a ser comparada. Se ela for válida e a parte de tag do endereço combinar com a tag, é um acerto. Os dados são lidos da palavra selecionada, se for um load ou são escritos na palavra selecionada, se for um store. O sinal Cache Ready é definido em seguida. Se for uma escrita, o bit de modificação é definido como 1. Observe que um acerto de escrita também define o bit de validade e o campo de tag; embora pareça desnecessário, ele é incluído porque a tag é uma única memória, de modo que, para mudar o bit

de modificação, também precisamos mudar os campos de validade e tag. Se for um acerto e o bloco for válido, a MEF retorna ao estado ocioso. Uma falha primeiro atualiza a tag de cache e depois vai para o estado Write-Back, se o bloco nesse local tiver um valor de bit de modificação igual a 1, ou para o estado Alocar, se for 0.

- **Write-Back:** Esse estado escreve o bloco de 128 bits na memória usando o endereço composto da tag e do índice de cache. Continuamos nesse estado esperando pelo sinal Ready da memória. Quando a escrita na memória termina, a MEF vai para o estado Alocar.
- **Alocar:** O novo bloco é apanhado da memória. Permanecemos nesse estado aguardando pelo sinal Ready da memória. Quando a leitura da memória termina, a MEF vai para o estado Comparar Tag. Embora pudéssemos ter ido para um novo estado para completar a operação em vez de reutilizar o estado Comparar Tag, existe muita sobreposição, incluindo a atualização da palavra apropriada no bloco se o acesso foi uma escrita.

Esse modelo simples facilmente poderia ser estendido com mais estados, para tentar melhorar o desempenho. Por exemplo, o estado Comparar Tag realiza a comparação e a leitura ou escrita dos dados de cache em um único ciclo de clock. Normalmente, a comparação e acesso à cache são feitos em estados separados, no sentido de tentar melhorar o tempo do ciclo de clock. Outra otimização seria acrescentar um buffer de escrita de modo que pudéssemos salvar o bloco de modificação e depois ler o novo bloco primeiro, de modo que o processador não tenha de esperar por dois acessos à memória em uma falha de modificação. A cache então escreveria o bloco modificado do buffer de escrita enquanto o processador está operando sobre os dados solicitados.

5.10. Paralelismo e hierarquias de memória: coerência de cache

Dado que um multiprocessador multicore significa múltiplos processadores em um único chip, esses processadores provavelmente compartilham um espaço de endereçamento físico comum. O caching de dados compartilhados gera um novo problema, pois a visão da memória mantida por dois processadores diferentes é através de suas caches individuais, que, sem quaisquer precauções adicionais, poderiam acabar vendo dois valores diferentes. A [Figura 5.35](#) ilustra o problema e mostra como dois processadores diferentes podem ter dois valores diferentes para o mesmo local. Essa dificuldade geralmente é referenciada como o *problema de coerência de cache*.

Informalmente, poderíamos dizer que um sistema de memória é coerente se qualquer leitura de um item de dados retornar o valor escrito mais recentemente desse item de dados. Essa definição, embora intuitivamente atraente, é vaga e simples; a realidade é muito mais complexa. Essa definição simples contém dois aspectos diferentes do comportamento do sistema de memória, ambos críticos para escrever programas corretos de memória compartilhada. O primeiro aspecto, chamado de *coerência*, define *que valores* podem ser retornados por uma leitura. O segundo aspecto, chamado *consistência*, determina *quando* um valor escrito será retornado por uma leitura.

Vejamos primeiro a coerência. Um sistema de memória é coerente se:

1. Uma leitura por um processador P para um local X que segue uma escrita por P a X, sem escritas de X por outro processador ocorrendo entre a escrita e a leitura por P, sempre retorna o valor escrito por P. Assim, na [Figura 5.41](#), se a CPU A tivesse de ler X após a etapa de tempo 3, ela deverá ver o valor 1.
2. Uma leitura por um processador ao local X que segue uma escrita por outro processador a X retorna o valor escrito se a leitura e escrita forem suficientemente separadas no tempo e nenhuma outra escrita em X ocorrer entre os dois acessos. Assim, na [Figura 5.41](#), precisamos de um mecanismo de modo que o valor 0 na cache da CPU B seja substituído pelo valor 1, após a CPU A armazenar 1 na memória do endereço X, na etapa de tempo 3.
3. As escritas no mesmo local são *serializadas*; ou seja, duas escritas no mesmo local por dois processadores quaisquer são vistas na mesma ordem

por todos os processadores. Por exemplo, se a CPU B armazena 2 na memória do endereço X após a etapa de tempo 3, os processadores nunca podem ler o valor no local X como 2 e mais tarde lê-lo como 1.

Etapa de tempo	Evento	Conteúdo de cache para CPU A	Conteúdo de cache para CPU B	Conteúdo de memória para local X
0				0
1	CPU A lê X	0		0
2	CPU B lê X	0	0	0
3	CPU A armazena 1 em X	1	0	1

FIGURA 5.41 O problema de coerência de cache para um único local da memória (X), lido e escrito por dois processadores (A e B).

Assumimos inicialmente que nenhuma cache contém a variável e que X tem o valor 0. Também consideramos uma cache write-through; uma cache write-back acrescenta algumas complicações adicionais, porém semelhantes. Depois que o valor de X foi escrito por A, a cache de A e a memória contêm o novo valor, mas a cache de B não, e se B ler o valor de X, ele receberá 0!

A primeira propriedade simplesmente preserva a ordem do programa — certamente esperamos que essa propriedade seja verdadeira nos processadores de 1 core, por exemplo. A segunda propriedade define a noção do que significa ter uma visão coerente da memória: se um processador pudesse ler continuamente um valor de dados antigo, claramente diríamos que a memória estava incoerente.

A necessidade de *serialização de escrita* é mais sutil, mas igualmente importante. Suponha que não serializássemos as escritas, e o processador P1 escreve no local X seguido por P2 escrevendo no local X. Serializar as escritas garante que cada processador verá a escrita feita por P2 em algum ponto. Se não serializássemos as escritas, pode ser que algum processador veja a escrita de P2 primeiro e depois veja a escrita de P1, mantendo o valor escrito por P1 indefinidamente. O modo mais simples de evitar essas dificuldades é garantir que todas as escritas no mesmo local sejam vistas na mesma ordem; essa propriedade é chamada *serialização de escrita*.

Ferramentas básicas para impor a coerência

Protocolos para suportar a coerença

Em um multiprocessador coerente com a cache, as caches oferecem *migração* e *replicação* de itens de dados compartilhados:

- *Migração*: Um item de dados pode ser movido para uma cache local e usado lá de uma forma transparente. A migração reduz a latência para acessar um item de dados compartilhado que está alocado remotamente e a demanda de largura de banda sobre a memória compartilhada.
- *Replicação*: Quando os dados compartilhados estão sendo simultaneamente lidos, as caches fazem uma cópia do item de dados na cache local. A replicação reduz a latência de acesso e a disputa por um item lido de dados compartilhado.

É essencial, para o desempenho no acesso aos dados compartilhados, oferecer suporte a essa migração e replicação, de modo que muitos multiprocessadores introduzem um protocolo de hardware que mantém caches coerentes. Os protocolos para manter coerência a múltiplos processadores são chamados de *protocolos de coerência de cache*. Acompanhar o estado de qualquer compartilhamento de um bloco de dados é essencial para implementar um protocolo coerente com a cache.

O protocolo de coerência de cache mais comum é o *snooping*. Cada cache que tem uma cópia dos dados de um bloco da memória física também tem uma cópia do status de compartilhamento do bloco, mas nenhum estado centralizado é mantido. As caches são todas acessíveis por algum meio de broadcast (um barramento ou rede), e todos os controladores monitoram ou *vasculham* o meio, a fim de determinar se eles têm ou não uma cópia de um bloco que é solicitado em um acesso ao barramento ou switch.

Na próxima seção, explicamos a coerência de cache baseada em snooping conforme implementada com um barramento compartilhado, mas qualquer meio de comunicação que envia falhas de cache por broadcast a todos os processadores pode ser usado para implementar um esquema de coerência baseado em snooping. Esse broadcasting de todas as caches torna os protocolos de snooping simples de implementar, mas também limita sua escalabilidade.

Protocolos de snooping

Um método para impor a coerência é garantir que um processador tenha acesso exclusivo a um item de dados antes de escrevê-lo. Esse estilo de protocolo é chamado *protocolo de invalidação de escrita*, pois invalida as cópias em outras caches em uma escrita. O acesso exclusivo garante que não existe qualquer outra

cópia de um item passível de leitura ou escrita quando ocorre a escrita: todas as outras cópias do item em cache são invalidadas.

A Figura 5.42 mostra um exemplo de um protocolo de invalidação para um barramento de snooping com caches write-back em ação. Para ver como esse protocolo garante a coerência, considere uma escrita seguida por uma leitura por outro processador: como a escrita requer acesso exclusivo, qualquer cópia mantida pelo processador de leitura precisa ser invalidada (daí o nome do protocolo). Sendo assim, quando ocorre a leitura, ela falha na cache, e esta é forçada a buscar uma nova cópia dos dados. Para uma escrita, exigimos que o processador escrevendo tenha acesso exclusivo, impedindo que qualquer outro processador seja capaz de escrever simultaneamente. Se dois processadores tentarem escrever os mesmos dados simultaneamente, um deles vence a corrida, fazendo com que a cópia do outro processador seja invalidada. Para que o outro processador complete sua escrita, ele precisa obter uma nova cópia dos dados, que agora precisa conter o valor atualizado. Portanto, esse protocolo também impõe a serialização da escrita.

Atividade do processador	Atividade do barramento	Conteúdo da cache da CPU A	Conteúdo da cache da CPU B	Conteúdo do local de memória X
				0
CPU A lê X	Falha de cache para X	0		0
CPU B lê X	Falha de cache para X	0	0	0
CPU A escreve 1 em X	Invalidação para X	1		0
CPU B lê X	Falha de cache para X	1	1	1

FIGURA 5.42 Um exemplo de um protocolo de invalidação atuando sobre um barramento de snooping para um único bloco de cache (X) com caches write-back.

Consideramos que nenhuma cache mantém X inicialmente e que o valor de X na memória é 0. O conteúdo da CPU e da memória mostra o valor após o processador e a atividade do barramento terem sido completados. Um espaço em branco indica nenhuma atividade ou nenhuma cópia em cache. Quando ocorre a segunda falha por B, a CPU A responde com o valor cancelando a resposta da memória. Além disso, tanto o conteúdo da cache de B quanto o conteúdo de memória de X são atualizados. Essa atualização de memória, que ocorre quando um bloco se torna compartilhado, simplifica o protocolo, mas é possível

acompanhar a posse e forçar o write-back somente se o bloco for substituído. Isso requer a introdução de um estado adicional, chamado “owner” (proprietário), que indica que um bloco pode ser compartilhado, mas o processador que o possui é responsável por atualizar quaisquer outros processadores e memória quando muda o bloco ou o substitui.

Interface hardware/software

Uma ideia interessante é que o tamanho do bloco desempenha um papel importante na coerência da cache. Por exemplo, considere o caso do snooping em uma cache com um tamanho de bloco de oito palavras, com uma única palavra alternativamente escrita e lida por dois processadores. A maioria dos protocolos troca blocos inteiros entre os processadores, aumentando assim as demandas da largura de banda de coerência.

Blocos grandes também podem causar o que é chamado **compartilhamento falso**: quando duas variáveis compartilhadas não relacionadas estão localizadas no mesmo bloco de cache, o bloco inteiro é trocado entre os processadores, embora os processadores estejam acessando variáveis diferentes. Os programadores e compiladores deverão dispor os dados cuidadosamente para evitar o compartilhamento falso.

compartilhamento falso

Quando duas variáveis compartilhadas não relacionadas estão localizadas no mesmo bloco de cache e o bloco inteiro é trocado entre os processadores, embora os processadores estejam acessando variáveis diferentes.

Detalhamento

Embora as três propriedades listadas no início desta seção sejam suficientes para garantir a coerência, a questão de quando um valor escrito será visto também é importante. Para ver por que, observe que não podemos exigir que uma leitura de X na Figura 5.41 veja instantaneamente o valor escrito para X por algum outro processador. Se, por exemplo, uma escrita de X em um processador preceder uma leitura de X em outro processador pouco antes, pode ser impossível garantir que a leitura retorne o valor dos dados escritos, pois estes podem nem sequer ter saído do processador, nesse ponto. A questão

de exatamente *quando* um valor escrito deverá ser visto por um leitor é definido por um *modelo de consistência de memória*.

Fazemos as duas suposições a seguir. Primeiro, uma escrita não termina (e permite que ocorra a próxima escrita) até que todos os processadores tenham visto o efeito dessa escrita. Em segundo lugar, o processador não muda a ordem de qualquer escrita com relação a qualquer outro acesso à memória. Essas duas condições significam que, se um processador escreve no local X seguido pelo local Y, qualquer processador que vê o novo valor de Y também deve ver o novo valor de X. Essas restrições permitem que o processador reordene as leituras, mas força o processador a terminar uma escrita na ordem do programa.

Detalhamento

Como a entrada pode mudar a memória por trás das caches e como a saída poderia precisar do valor mais recente em uma cache write-back, também há um problema de coerência de cache para E/S com as caches de um único processador, bem como entre as caches de multiprocessadores. O problema da coerência de cache para multiprocessadores e E/S (ver Capítulo 6), embora semelhante em origem, tem diferentes características que afetam a solução apropriada. Diferente da E/S, em que múltiplas cópias de dados são um evento raro — a ser evitado sempre que possível —, um programa sendo executado em múltiplos processadores normalmente terá cópias dos mesmos dados em várias caches.

Detalhamento

Além do protocolo de coerência de cache baseado em snooping, em que o status dos blocos compartilhados é distribuído, um protocolo de coerência de cache *baseado em diretório* mantém o status de compartilhamento de um bloco de memória física em apenas um local, chamado *diretório*. A coerência baseada em diretório tem um overhead de implementação ligeiramente mais alto que o snooping, mas pode reduzir o tráfego entre as caches e, portanto, se expandir para quantidades maiores de processadores.

5.11. Vida real: as hierarquias de memória ARM

Cortex-A8 e Intel Core i7

Nesta seção, veremos a hierarquia de memória dos mesmos dois microprocessadores descritos no [Capítulo 4](#): o ARM Cortex-A8 e o Intel Core i7. Esta seção é baseada na [Seção 2.6 de Arquitetura de Computadores: Uma abordagem quantitativa](#), 5^a edição.

A [Figura 5.43](#) resume os tamanhos de endereço e as TLBs dos dois processadores. Observe que o A8 possui duas TLBs com um espaço de endereçamento virtual de 32 bits e um espaço de endereços físicos de 32 bits. O Core i7 possui três TLBs com um endereço virtual de 48 bits e um endereço físico de 44 bits. Embora os registradores de 64 bits do Core i7 possam manter um endereço virtual maior, não houve necessidade de um espaço tão grande pelo software, e os endereços virtuais de 48 bits encurtam tanto o uso de memória da tabela de páginas quanto o hardware da TLB.

Característica	ARM Cortex-A8	Intel Core i7
Endereço virtual	32 bits	48 bits
Endereço físico	32 bits	44 bits
Tamanho de página	Variável: 4, 16, 64 KiB, 1, 16 MiB	Variável: 4 KiB, 2/4 MiB
Organização da TLB	1 TLB para instruções e 1 TLB para dados Ambas as TLBs L1 são totalmente associativas, com 32 entradas, substituição round robin Falhas de TLB tratadas no hardware	1 TLB para instruções e 1 TLB para dados por núcleo Ambas as TLBs L1 são associativas em conjunto com quatro vias, substituição LRU I-TLB L1 tem 128 entradas para páginas pequenas, 7 por thread para páginas grandes D-TLB L1 tem 64 entradas para páginas pequenas, 32 para páginas grandes A TLB L2 é associativa em conjunto com quatro vias, substituição LRU A TLB L2 tem 512 entradas Falhas da TLB tratadas no hardware

FIGURA 5.43 Tradução de endereços e hardware TLB para o ARM Cortex-A8 e Intel Core i7 920.

Os dois processadores fornecem suporte a páginas grandes, que são usadas para coisas como o sistema operacional ou no mapeamento de um buffer de quadro. O esquema de página

grande evita o uso de um grande número de entradas para mapear um único objeto que está sempre presente.

A [Figura 5.44](#) mostra suas caches. Lembre-se de que o A8 tem apenas um processador ou núcleo, enquanto o Core i7 tem quatro. Ambos são organizados com caches de instrução L1 de 32 KiB, associativos em conjunto com quatro vias (por núcleo) com blocos de 64 bytes. O A8 usa o mesmo projeto para cache de dados, enquanto o Core i7 mantém tudo igual, exceto pela associatividade, que aumenta para oito vias. Ambos utilizam uma cache L2 unificada associativa em conjunto com oito vias (por núcleo), com blocos de 64 bytes, embora o A8 varie em tamanho de 128 KiB até 1 MiB, enquanto o Core i7 é fixo em 256 KiB. Como o Core i7 é usado para servidores, ele também oferece uma cache L3 compartilhada por todos os núcleos no chip. Seu tamanho varia, dependendo do número de núcleos. Com quatro núcleos, como neste caso, o tamanho é de 8 MiB.

Característica	ARM Cortex-A8	Intel Nehalem
Organização da cache L1	Caches divididas para instrução e dados	Caches divididas para instrução e dados
Tamanho da cache L1	32KB cada para instruções/dados	32KB cada para instruções/dados por núcleo
Associatividade da cache L1	Associativo por conjunto em 4 vias (I), 4 vias (D)	Associativo por conjunto em 4 vias (I), 8 vias (D)
Substituição L1	Aleatória	Substituição LRU aproximada
Tamanho de bloco da L1	64 bytes	64 bytes
Política de escrita da L1	Write-back, Write-allocate (?)	Write-back, No-write-allocate
Tempo de acerto da L1 (uso de load)	1 ciclo de clock	4 ciclos de clock, em pipeline
Organização da cache L2	Unificada (instrução e dados)	Unificada (instruções e dados) por núcleo
Tamanho da cache L2	128 KiB to 1 MiB	256 KiB (0,25 MiB)
Associatividade da cache L2	Associativo em conjunto com 8 vias	Associativo em conjunto com 8 vias
Substituição L2	Aleatória	Substituição LRU aproximada
Tamanho de bloco da L2	64 bytes	64 bytes
Política de escrita da L2	Write-back, Write-allocate (?)	Write-back, Write-allocate
Tempo de acerto da L2	11 ciclos de clock	10 ciclos de clock
Organização da cache L3	--	Unificada (instrução e dados)
Tamanho da cache L3	--	8 MiB, compartilhado
Associatividade da cache L3	--	Associativo em conjunto com 16 vias
Substituição L3	--	Substituição LRU aproximada
Tamanho de bloco da L3	--	64 bytes
Política de escrita da L3	--	Write-back, Write-allocate
Tempo de acerto da L3	--	35 ciclos de clock

FIGURA 5.44 Caches do ARM Cortex-A8 e do Intel Core i7 920.

Um desafio significativo enfrentado pelos projetistas de cache é dar suporte a processadores como o A8 e o Core i7, que podem executar mais de uma instrução de memória por ciclo de clock. Uma técnica popular é quebrar a cache em bancos e permitir vários acessos **paralelos**, independentes, desde que os acessos sejam para bancos diferentes. A técnica é semelhante aos bancos de DRAM intercalados ([Seção 5.2](#)).



P A R A L L E L I S M

O Core i7 possui otimizações adicionais que permitem reduzir a penalidade de falha. A primeira delas é o retorno da palavra requisitada primeiro em uma falha. Ele também continua executando instruções que acessam a cache de dados durante uma falha de cache. Os projetistas que estão tentando ocultar a latência da falha de cache normalmente usam essa técnica, chamada **cache não bloqueante**, quando montam processadores com execução fora de ordem. Eles implementam dois tipos de não bloqueio. *Acerto sob falha* permite acertos de cache adicionais durante uma falha, enquanto *falha sob acerto* permite múltiplas falhas de cache pendentes. O objetivo do primeiro deles é ocultar alguma latência de falha com outro trabalho, enquanto o objetivo do segundo é sobrepor a latência de duas falhas diferentes.

cache não bloqueante

Uma cache que permite que o processador faça referências a ela enquanto a cache está tratando uma falha anterior.

A sobreposição de uma grande fração dos tempos de falha para múltiplas falhas pendentes requer um sistema de memória de alta largura de banda, capaz de tratar múltiplas falhas em paralelo. Em um dispositivo móvel pessoal, a memória pode apenas ser capaz de tirar proveito limitado dessa capacidade, mas grandes servidores e multiprocessadores frequentemente possuem sistemas de memória capazes de tratar mais de uma falha pendente em paralelo.

O Core i7 possui um mecanismo de pré-busca para acessos a dados. Ele olha um padrão de falhas de dados e usa essas informações para tentar prever o próximo endereço a fim de começar a buscar os dados antes que a falha ocorra. Essas técnicas geralmente funcionam melhor ao acessar arrays em loops.

As sofisticadas hierarquias de memória desses chips e a grande fração dos dies dedicada às caches e às TLBs mostram o significativo esforço de projeto despendido para tentar diminuir a lacuna entre tempos de ciclo de processador e latência de memória.

Desempenho das hierarquias de memória do A8 e do Core i7

A hierarquia de memória do Cortex-A8 foi simulada com uma cache L2 associativa em conjunto com oito vias, usando os benchmarks de inteiros Minnespec. Como dissemos no [Capítulo 4](#), Minnespec é um conjunto de benchmarks que consiste nos benchmarks SPEC2000, mas com diferentes entradas, que reduzem os tempos de execução por várias ordens de grandeza. Embora o uso de entradas menores não mude a mistura de instruções, isso afeta o comportamento da cache. Por exemplo, no mcf, o benchmark de inteiros SPEC2000 com uso mais intensivo de memória, o Minnespec tem uma taxa de falhas para uma cache de 32 KiB que é de apenas 65% da taxa de falhas para a versão SPEC2000 completa. Para uma cache de 1 MiB, a diferença é um fator de seis! Por esse motivo, não se pode comparar os benchmarks Minnespec com os benchmarks SPEC2000, muito menos os benchmarks SPEC2006 ainda maiores, usados para o Core i7 na [Figura 5.47](#). Em vez disso, os dados são úteis para que se veja o impacto relativo das falhas de cache L1 e L2 e sobre o CPI geral, que

usamos no [Capítulo 4](#).

As taxas de falha de cache de instruções do A8 para esses benchmarks (e também para as versões SPEC2000 completas nas quais o Minnespec é baseado) são muito pequenas, mesmo que apenas para a cache L1: perto de zero para a maioria e abaixo de 1% para todos eles. Essa taxa tão baixa provavelmente é resultante da natureza computacionalmente intensa dos programas SPEC e da cache associativa em conjunto com quatro vias, que elimina a maioria das falhas por conflito. A [Figura 5.45](#) mostra os resultados da cache de dados para o A8, que possui taxas de falha de cache L1 e L2 significativas. A penalidade de falha L1 para um Cortex-A8 a 1 GHz é de 11 ciclos de clock, enquanto a penalidade de falha L2 é considerada como sendo 60 ciclos de clock. Usando essas penalidades de falha, a [Figura 5.46](#) mostra a penalidade de falha média por acesso aos dados.

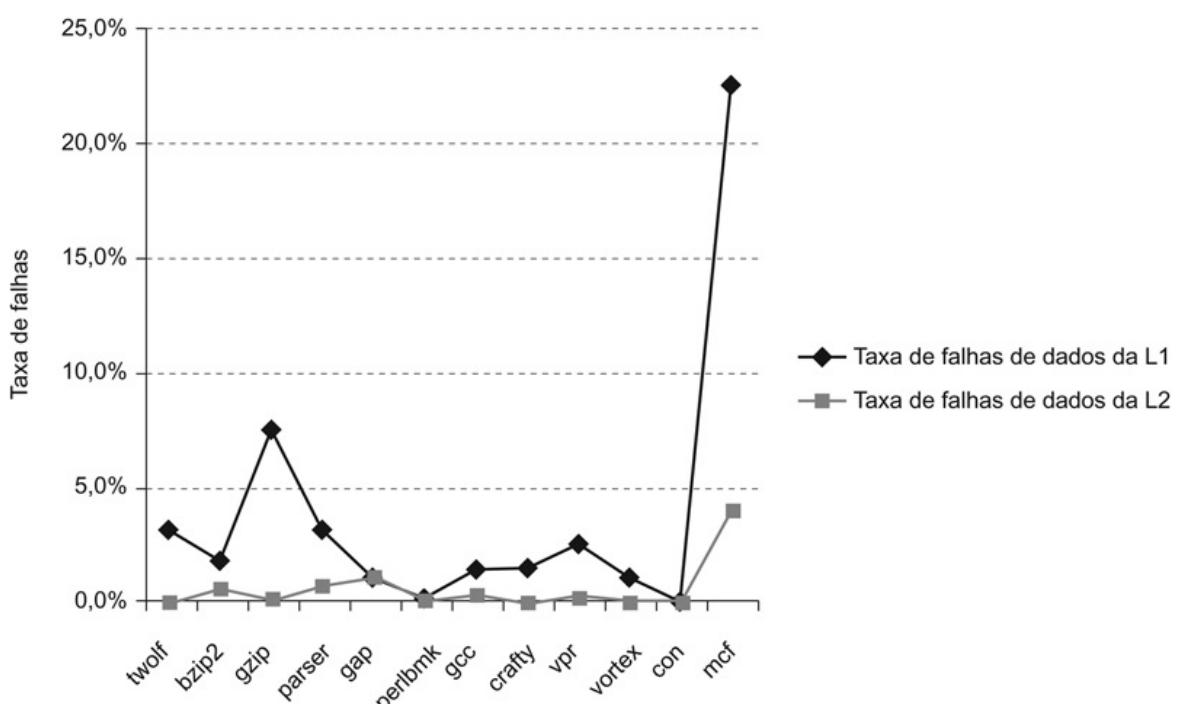


FIGURA 5.45 Taxas de falha de cache de dados para o ARM Cortex-A8 ao executar o Minnespec, uma pequena versão do SPEC2000.

Aplicações com maiores pegadas de memória costumam ter maiores taxas de falha em caches L1 e L2. Observe que a taxa L2 é a taxa de falhas global; ou seja, contando todas as referências, incluindo aquelas que acertam na L1 (veja o Detalhamento da [Seção 5.4](#)). Mcf é conhecido como um “cache

buster". Observe que essa figura é para os mesmos sistemas e benchmarks da [Figura 4.76](#), no [Capítulo 4](#).

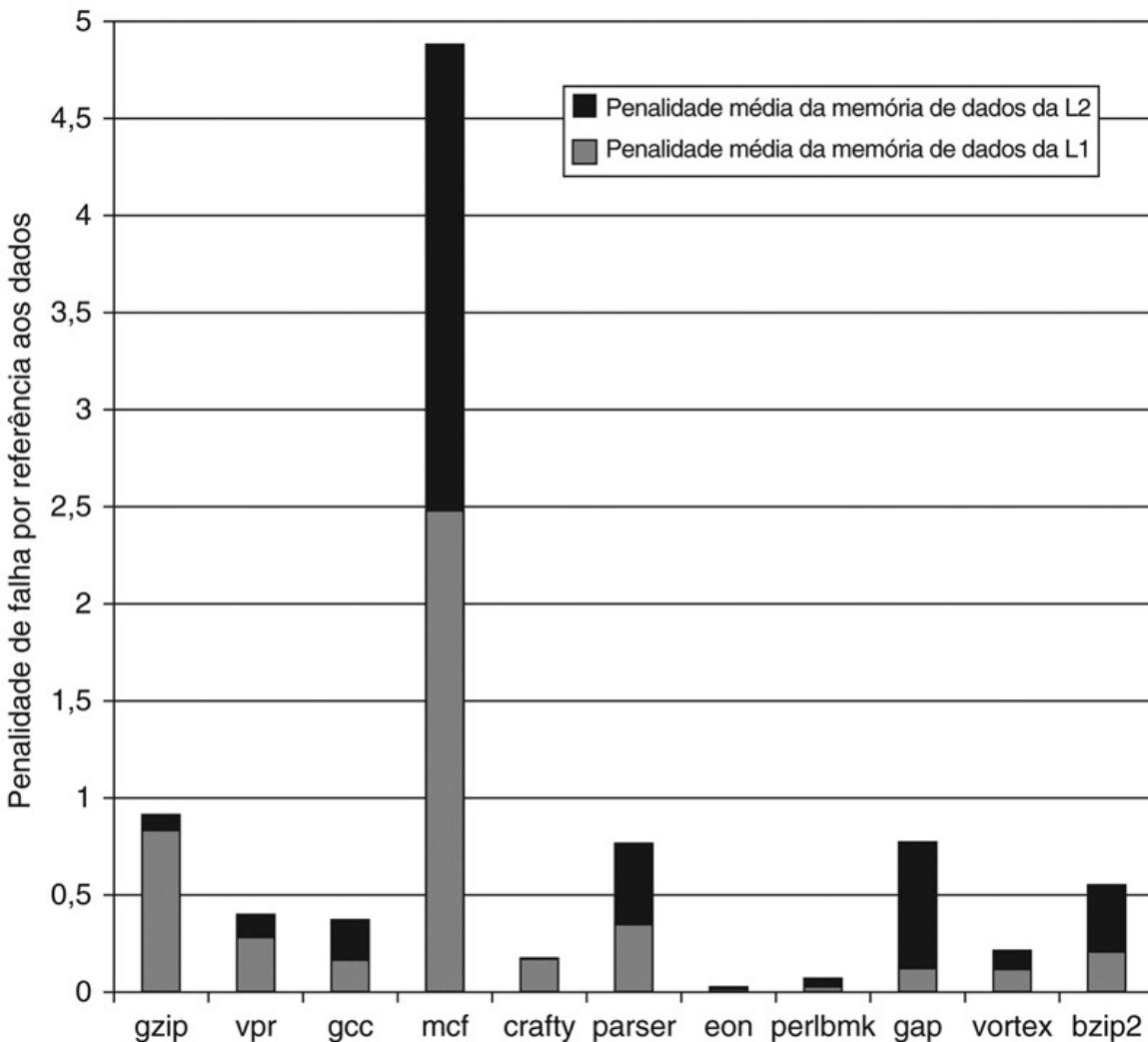


FIGURA 5.46 Penalidade média de acesso à memória em ciclos de clock por referência à memória de dados vindo de caches L1 e L2 para o processador ARM executando o Minnespec.

Embora as taxas de falhas para L1 sejam significativamente mais altas, a penalidade de falha L2, que é mais de cinco vezes mais alta, significa que as falhas L2 podem contribuir significativamente.

A [Figura 5.47](#) mostra as taxas de falha para as caches do Core i7 usando os benchmarks SPEC2006. A taxa de falhas da cache de instruções L1 varia de

0,1% a 1,8%, com uma média um pouco acima de 0,4%. Essa taxa está de acordo com outros estudos do comportamento da cache de instruções para os benchmarks SPEC CPU2006, que mostram baixas taxas de falhas da cache de instruções. Com taxas de falha da cache de dados L1 variando de 5% a 10%, e às vezes maiores, a importância das caches L2 e L3 deverá ser óbvia. Como o custo para uma falha da memória é de mais de 100 ciclos, e a taxa média de falha de dados na L2 é 4%, a L3 certamente é crítica. Considerando que cerca de metade das instruções são loads ou stores, sem a L3, as falhas da cache L2 poderiam acrescentar dois ciclos por instrução à CPI! Em comparação, a taxa média de falha de dados da L3 de 1% ainda é significativa, mas quatro vezes menor que a taxa de falhas L2 e seis vezes menor que a taxa de falhas L1.

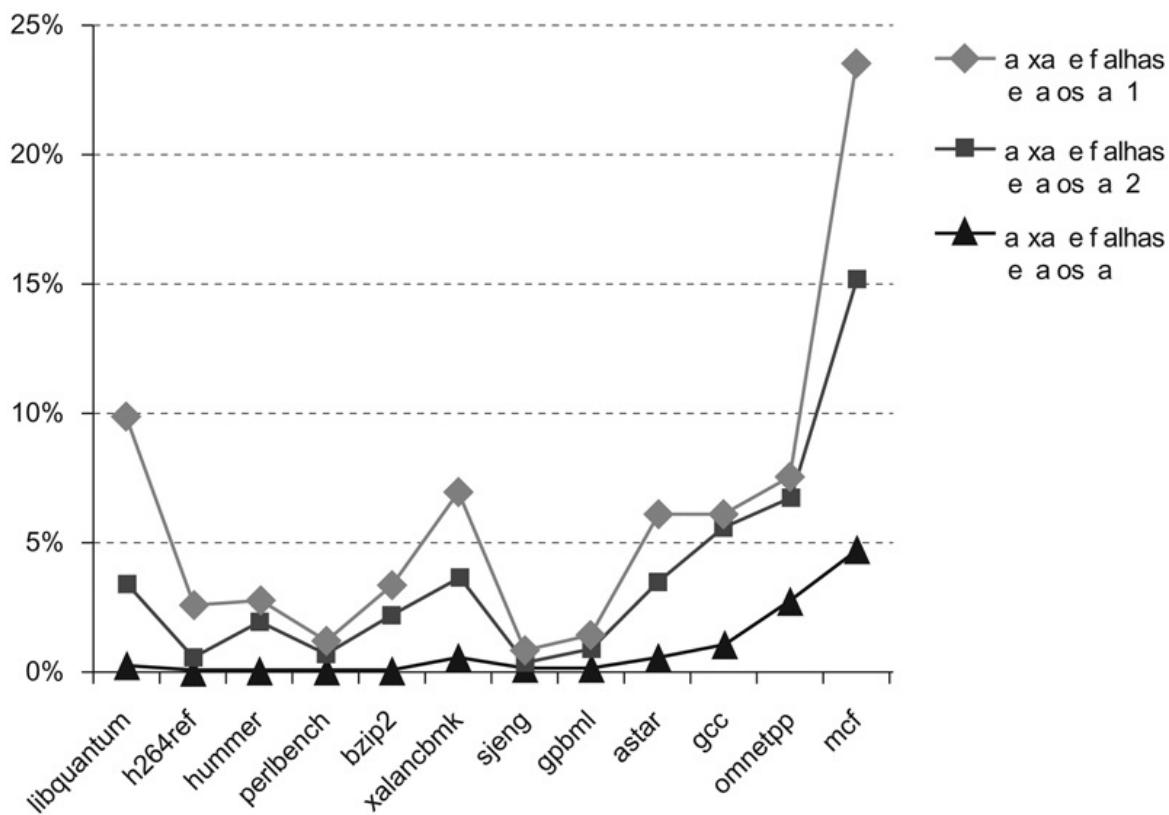


FIGURA 5.47 As taxas de falhas da cache de dados L1, L2 e L3 para o Intel Core i7 executando os benchmarks SPEC CPU2006 completo para inteiros.

Detalhamento

Como a especulação às vezes pode ser errada (Capítulo 4), existem referências à cache de dados L1 que não correspondem a loads ou stores que eventualmente completam a execução. Os dados na Figura 5.45 são medidos contra todas as solicitações, incluindo algumas que são canceladas. A taxa de falhas, quando medida contra apenas os acessos a dados concluídos, é 1,6 vezes maior (uma média de 9,5% contra 5,9% para as falhas de cache de dados L1).

5.12. Mais rápido: Bloqueio de cache e multiplicação matricial

Nosso próximo passo, continuando a saga de melhoria de desempenho do DGEMM, ajustando-o ao hardware subjacente, é acrescentar o bloqueio de cache às otimizações de paralelismo de subword e paralelismo em nível de instrução, dos Capítulos 3 e 4. A [Figura 5.48](#) mostra a versão em bloco do DGEMM da [Figura 4.80](#). As mudanças são as mesmas que foram feitas anteriormente, ao passar do DGEMM não otimizado da [Figura 3.21](#) para o DGEMM em bloco da [Figura 5.21](#), anteriormente neste capítulo. Desta vez, estamos apanhando a versão desdobrada do DGEMM, do [Capítulo 4](#), e chamando-a muitas vezes sobre as submatrizes de A, B e C. De fato, as linhas 28–34 e as linhas 7–8 da [Figura 5.48](#) são idênticas às linhas 14–20 e as linhas 5–6 da [Figura 5.21](#), com exceção do incremento do loop for na linha 7 pela quantidade desdobrada.

```

1 #include <x86intrin.h>
2 #define UNROLL (4)
3 #define BLOCKSIZE 32
4 void do_block (int n, int si, int sj, int sk,
5                 double *A, double *B, double *C)
6 {
7     for ( int i = si; i < si+BLOCKSIZE; i+=UNROLL*4 )
8         for ( int j = sj; j < sj+BLOCKSIZE; j++ ) {
9             __m256d c[4];
10            for ( int x = 0; x < UNROLL; x++ )
11                c[x] = _mm256_load_pd(C+i+x*4+j*n);
12 /* c[x] = C[i][j] */
13            for( int k = sk; k < sk+BLOCKSIZE; k++ )
14            {
15                __m256d b = _mm256_broadcast_sd(B+k+j*n);
16 /* b = B[k][j] */
17                for (int x = 0; x < UNROLL; x++)
18                    c[x] = _mm256_add_pd(c[x], /* c[x]+=A[i][k]*b */
19                                         _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
20            }
21
22
23            for ( int x = 0; x < UNROLL; x++ )
24                _mm256_store_pd(C+i+x*4+j*n, c[x]);
25 /* C[i][j] = c[x] */
26        }
27
28 void dgemm (int n, double* A, double* B, double* C)
29 {
30     for ( int sj = 0; sj < n; sj += BLOCKSIZE )
31         for ( int si = 0; si < n; si += BLOCKSIZE )
32             for ( int sk = 0; sk < n; sk += BLOCKSIZE )
33                 do_block(n, si, sj, sk, A, B, C);
34 }

```

FIGURA 5.48 Versão C otimizada do DGEMM da Figura 4.80 usando o bloqueio de cache.

Essas mudanças são as mesmas encontradas na [Figura 5.21](#). A linguagem assembly produzida pelo compilador para a função `do_block` é quase idêntica à [Figura 4.81](#). Mais uma vez, não existe overhead para chamar o `do_block`, pois o compilador insere a função em linha.

Ao contrário dos capítulos anteriores, não mostramos o código x86 resultante, pois o código do loop interno é quase idêntico à [Figura 4.81](#), já que o bloqueio não afeta a computação, mas apenas a ordem que ela acessa os dados na

memória. O que muda é a contabilidade das instruções de inteiros para implementar os loops for. Ela se expande de 14 instruções antes do loop interno a 8, após o loop da [Figura 4.80](#) para 40 e 28 instruções, respectivamente, para o código de contabilidade gerado para a [Figura 5.48](#). Apesar disso, as instruções extras executadas são ínfimas em comparação com a melhoria no desempenho da redução das falhas de cache. A [Figura 5.49](#) compara o desempenho não otimizado com as otimizações para o paralelismo de subword, paralelismo em nível de instrução e caches. O bloqueio melhora o desempenho sobre o código AVX desdobrado por fatores de 2 a 2,5 para as maiores matrizes. Quando comparamos o código não otimizado com o código contendo todas as três otimizações, a melhoria no desempenho tem fatores de 8 a 15, com o maior aumento para a maior matriz.

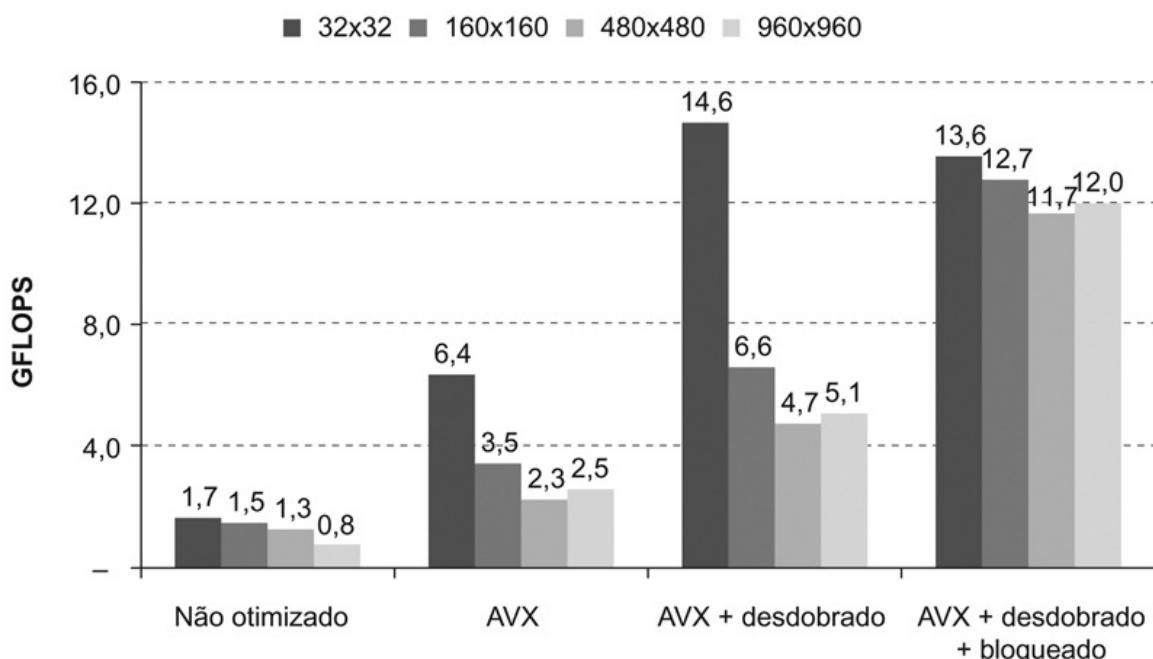


FIGURA 5.49 Desempenho de quatro versões do DGEMM de dimensões de matriz 32×32 a 960×960 .

O código totalmente otimizado para a maior matriz é quase 15 vezes mais rápido que a versão não otimizada na [Figura 3.21](#), no [Capítulo 3](#).

Detalhamento

Como dissemos no Detalhamento da Seção 3.8, esses resultados são para o

modo Turbo desligado. Assim como nos Capítulos 3 e 4, quando o ativamos, melhoramos todos os resultados pelo aumento temporário na taxa de clock, de $3,3/2,6 = 1,27$. O modo Turbo funciona particularmente bem neste caso, pois está usando apenas um único núcleo de um chip com oito núcleos. Entretanto, se quisermos velocidade, devemos usar todos os núcleos, o que será visto no Capítulo 6.

5.13. Falácia e armadilhas

Como um dos aspectos mais naturalmente quantitativos da arquitetura de um computador, a hierarquia de memória pareceria ser menos vulnerável às falácia e armadilhas. Não só houve muitas falácia divulgadas e armadilhas encontradas, mas algumas levaram a grandes resultados negativos. Começamos com uma armadilha que frequentemente pega estudantes em exercícios e exames.

Armadilha: ignorar o comportamento do sistema de memória ao escrever programas ou gerar código em um compilador.

Isso poderia facilmente ser escrito como uma falácia: “Os programadores podem ignorar as hierarquias de memória ao escrever código.” A avaliação da ordenação na [Figura 5.19](#) e do bloqueio de cache na [Seção 5.12](#) demonstra que os programadores podem facilmente dobrar o desempenho se levarem em conta o comportamento do sistema de memória no projeto de seus algoritmos.

Armadilha: esquecer-se de considerar o endereçamento em bytes ou o tamanho de bloco de cache ao simular uma cache.

Quando estamos simulando uma cache (manualmente ou por computador), precisamos levar em conta o efeito de um endereçamento em bytes e blocos multipalavra ao determinar para qual bloco de cache um certo endereço é mapeado. Por exemplo, se tivermos uma cache diretamente mapeada de 32 bytes com um tamanho de bloco de 4 bytes, o endereço em bytes 36 é mapeado no bloco 1 da cache, já que o endereço em bytes 36 é o endereço de bloco 9 e $(9 \bmod 8) = 1$. Por outro lado, se o endereço 36 for um endereço em palavras, então, ele é mapeado no bloco $(36 \bmod 8) = 4$. O problema deve informar claramente a base do endereço.

De modo semelhante, precisamos considerar o tamanho do bloco. Suponha que tenhamos uma cache com 256 bytes e um tamanho de bloco de 32 bytes. Em que bloco o endereço em bytes 300 se encontra? Se dividirmos o endereço 300 em campos, poderemos ver a resposta:

31	30	29	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	1	0	1	1	0	0
Número do bloco na cache										Offset de bloco							

Endereço em blocos

O endereço em bytes 300 é o endereço de bloco

$$\left[\frac{300}{32} \right] = 9$$

O número de blocos na cache é

$$\left[\frac{256}{32} \right] = 8$$

O bloco número 9 cai no bloco de cache número ($9 \bmod 8$) = 1.

Esse erro pega muitas pessoas, incluindo os autores (nos rascunhos anteriores) e instrutores que esquecem se pretendiam que os endereços estivessem em palavras, bytes ou números de bloco. Lembre-se dessa armadilha ao realizar os exercícios.

Armadilha: ter menos associatividade em conjunto para uma cache compartilhada que o número de cores ou threads compartilhando essa cache.

Sem cuidados adicionais, um programa **paralelo** sendo executado em 2^n

processadores ou threads pode facilmente alocar estruturas de dados a endereços que seriam mapeados para o mesmo conjunto de uma cache L2 compartilhada. Se a cache for associativa pelo menos em 2^n vias, então esses conflitos acidentais ficam ocultos pelo hardware do programa. Se não, os programadores poderiam enfrentar bugs de desempenho aparentemente misteriosos — na realidade, devido a falhas de conflito L2 — ao migrar de, digamos, um projeto de 16 núcleos para 32 cores, se ambos utilizarem caches L2 associativas com 16 vias.



PARALLELISM

Armadilha: usar tempo médio de acesso à memória para avaliar a hierarquia de memória de um processador com execução fora de ordem.

Se um processador é suspenso durante uma falha de cache, você pode calcular separadamente o tempo de stall de memória e o tempo de execução do processador, e, portanto, avaliar a hierarquia de memória de forma independente usando o tempo médio de acesso à memória ([Seção 5.4](#)).

Se o processador continuar executando instruções e puder até sustentar mais falhas de cache durante uma falha de cache, então, a única avaliação precisa da hierarquia de memória é simular o processador com execução fora de ordem, juntamente com a hierarquia de memória.

Armadilha: estender um espaço de endereçamento acrescentando segmentos sobre um espaço de endereçamento não segmentado.

Durante a década de 1970, muitos programas ficaram tão grandes que nem todo o código e dados podiam ser endereçados apenas com um endereço de 16 bits. Os computadores, então, foram revisados para oferecer endereços de 32 bits, quer por meio de um espaço de endereçamento de 32 bits não segmentado (também chamado de *espaço de endereçamento plano*), quer acrescentando 16 bits de segmento ao endereço de 16 bits existente. Do ponto de vista do marketing, acrescentar segmentos que fossem visíveis ao programador e que forçassem o programador e o compilador a decomponerem programas em segmentos podia resolver o problema de endereçamento. Infelizmente, existe problema toda vez que uma linguagem de programação quer um endereço que seja maior do que um segmento, como índices para grandes arrays, ponteiros irrestritos ou parâmetros por referência. Além disso, acrescentar segmentos pode transformar todos os endereços em duas palavras — uma para o número do segmento e outra para o offset do segmento —, causando problemas no uso dos endereços em registradores.

Falácia: as taxas de falha de disco na prática correspondem às suas especificações.

Dois estudos recentes avaliaram grandes coleções de discos para verificar a relação entre os resultados na prática e as especificações. Um desses estudos foi de quase 100.000 discos que haviam especificado um MTTF de 1.000.000 a 1.500.000 horas, ou AFR de 0,6% a 0,8%. Eles descobriram que AFRs de 2% a 4% eram comuns, normalmente de três a cinco vezes maiores do que as taxas especificadas (Schroeder et al., 2007). Um segundo estudo de mais de 100.000 discos na Google, que tinha especificado um AFR de cerca de 1,5%, descobriu que as taxas de falha de 1,7% para as unidades em seu primeiro ano subiam para 8,6% para as unidades em seu terceiro ano, ou cerca de cinco a seis vezes a taxa especificada (Pinheiro et al., 2007).

Falácia: os sistemas operacionais são o melhor lugar para escalarizar os acessos ao disco.

Como dissemos na [Seção 5.2](#), interfaces de disco de nível mais alto oferecem endereços de blocos lógicos ao sistema operacional host. Com essa abstração de alto nível, o melhor que um OS pode fazer para tentar ajudar o desempenho é classificar os endereços de blocos lógicos em ordem crescente. Porém, como o disco conhece o mapeamento real entre os endereços lógicos e a geometria física dos setores, trilhas e superfícies, isso pode reduzir as latências rotacional e de busca por meio do reescalonamento.

Por exemplo, suponha que a carga de trabalho seja de quatro leituras (Anderson, 2003):

Operação	LBA inicial	Comprimento
Leitura	724	8
Leitura	100	16
Leitura	9987	1
Leitura	26	128

O host poderia reordenar as quatro leituras para a ordem de bloco lógico:

Operação	LBA inicial	Comprimento
Leitura	26	128
Leitura	100	16
Leitura	724	8
Leitura	9987	1

Dependendo da localização relativa dos dados no disco, a reordenação poderia tornar isso pior, como demonstra a [Figura 5.50](#). As leituras escalonadas pelo disco são concluídas em três quartos de uma rotação do disco, mas as leituras escalonadas pelo OS exigem três rotações.

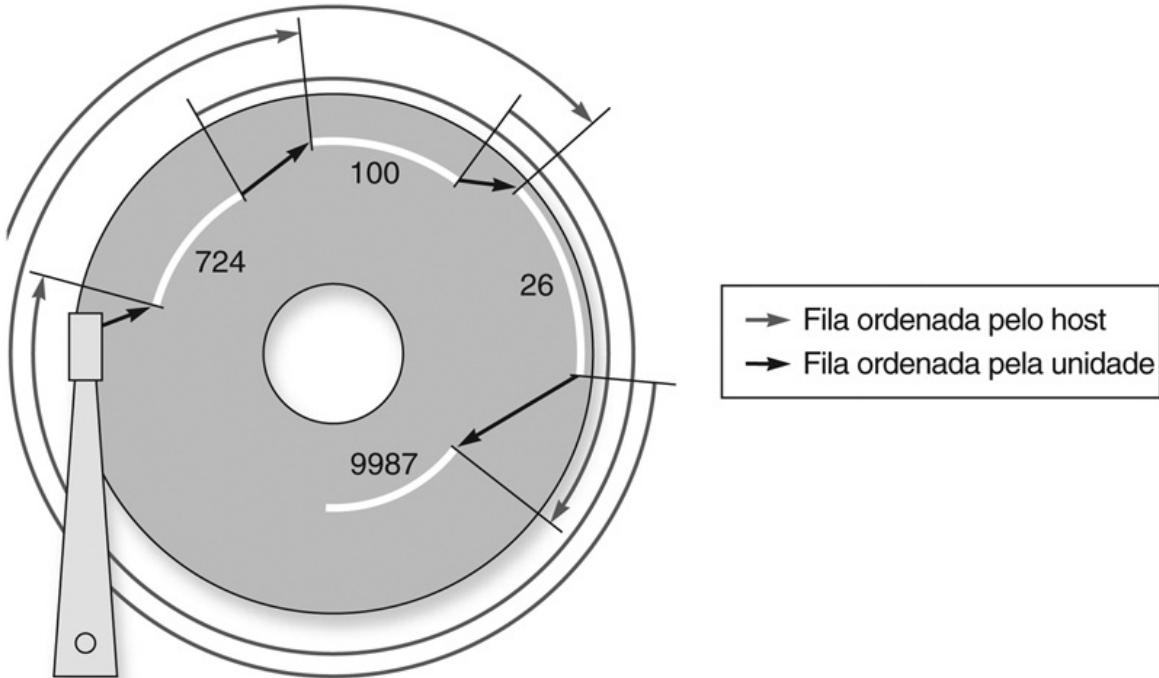


FIGURA 5.50 Exemplo mostrando os acessos escalonados pelo OS e pelo disco, rotulados como Fila ordenada pelo host e Fila ordenada pela unidade.

O primeiro acesso requer três rotações para concluir as quatro leituras, enquanto o segundo as completa em apenas três quartos de uma rotação (Anderson, 2003).

Armadilha: implementar um monitor de máquina virtual em uma arquitetura de conjunto de instruções que não foi projetada para ser virtualizável.

Muitos arquitetos nas décadas de 1970 e 1980 não tiveram o cuidado de garantir que todas as instruções lendo ou escrevendo informações relacionadas a informações de recurso de hardware fossem privilegiadas. Essa atitude *laissez-faire* causa problemas para os VMMs em todas essas arquiteturas, incluindo o x86, que usamos aqui como um exemplo.

A [Figura 5.51](#) descreve as 18 instruções que causam problemas para a virtualização (Robin et al., 2000). As duas classes gerais são instruções que:

- Leem os registradores de controle no modo usuário, o que revela que o sistema operacional guest está sendo executado em uma máquina virtual (como POPF, mencionada anteriormente).
- Verificam a proteção conforme requisitado pela arquitetura segmentada, mas

consideram que o sistema operacional está sendo executado no nível de privilégio mais alto.

Categoría de problema	Instruções do x86 com problema
Acessar registradores sensíveis sem interceptação ao executar no modo usuário	Armazenar registro da tabela de descritor global (SGDT) Armazenar registro da tabela de descritor local (SLDT) Armazenar registrador da tabela de descritor de interrupção (SIDT) Armazenar palavra de status da máquina (SMSW) Push de flags (PUSHF, PUSHFD) Pop de flags (POPF, POPFD)
Ao acessar mecanismos de memória virtual no modo usuário, as instruções falham nas verificações de proteção do x86	Carregar direitos de acesso do descritor de segmento (LAR) Carregar limite de segmento do descritor de segmento (LSL) Verificar se o descritor de segmento pode ser lido (VERR) Verificar se o descritor de segmento pode ser escrito (VERW) Pop para registrador de segmento (POP CS, POP SS, ...) Push para registrador de segmento (PUSH CS, PUSH SS, ...) Chamada distante para nível de privilégio diferente (CALL) Retorno distante para nível de privilégio diferente (RET) Desvio distante para nível de privilégio diferente (JMP) Interrupção de software (INT) Armazenar registrador de seletor de segmento (STR) Mover para/de registradores de segmento (MOVE)

FIGURA 5.51 Resumo de 18 instruções x86 que causam problemas para a virtualização (Robin e Irvine, 2000).

As cinco primeiras instruções no grupo de cima permitem que um programa no modo usuário leia um registrador de controle, como um registrador da tabela de descritores, sem causar uma interrupção. A instrução de “pop de flags” modifica um registrador de controle com informações sensíveis, mas falha silenciosamente quando está no modo usuário. A verificação de proteção da arquitetura segmentada do x86 é a ruína do grupo inferior, pois cada uma dessas instruções verifica o nível de privilégio implicitamente como parte da execução da instrução ao ler um registrador de controle. A verificação considera que o OS precisa estar no nível de privilégio mais alto, o que não acontece para as VMs guest. Somente “Mover para/de registradores de segmento” (MOVE) tenta modificar o estado de controle, e a verificação de proteção também falha.

Para simplificar as implementações dos VMMs no x86, tanto AMD quanto Intel propuseram extensões à arquitetura de um novo modo. O VT-x da Intel oferece um novo modo de execução para rodar VMs, uma definição projetada do

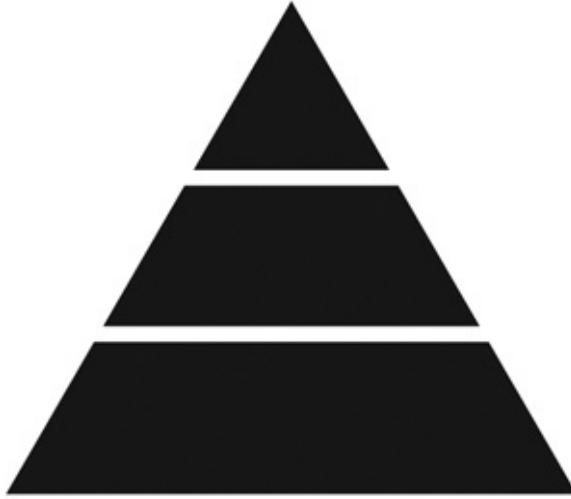
estado da VM, instruções para trocar de VMs rapidamente e um grande conjunto de parâmetros para selecionar as circunstâncias em que um VMM precisa ser chamado. Ao todo, o VT-x acrescenta 11 novas instruções para o x86. O Pacifica da AMD tem propostas semelhantes.

Uma alternativa para modificar o hardware é fazer pequenas modificações no sistema operacional de modo a evitar o uso de partes problemáticas da arquitetura. Essa técnica é chamada de *paravirtualização*, e o VMM Xen de fonte aberta é um bom exemplo. O VMM Xen oferece um OS guest com uma abstração de máquina virtual que utiliza apenas as partes fáceis de virtualizar o hardware físico do x86, em que o VMM é executado.

5.14. Comentários finais

A dificuldade de construir um sistema de memória para fazer frente aos processadores mais rápidos é acentuada pelo fato de que a matéria-prima para a memória principal, DRAMs, ser essencialmente a mesma nos computadores mais rápidos que nos computadores mais lentos e baratos.

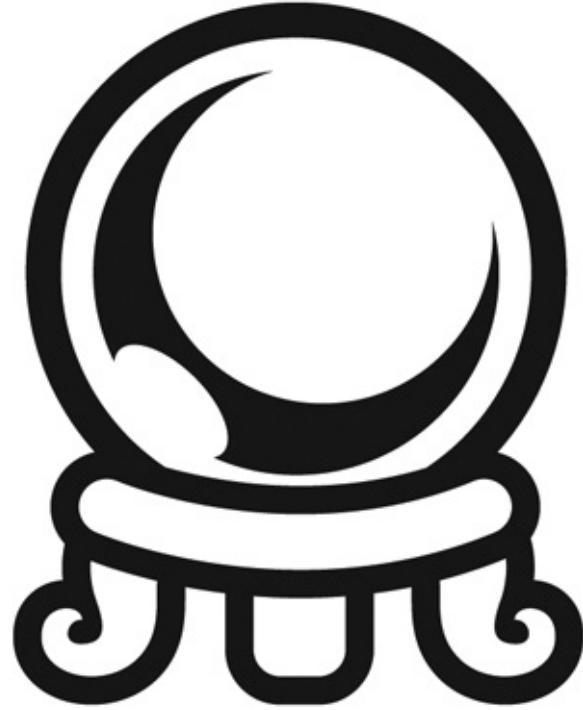
É o princípio da localidade que nos dá uma chance de superar a longa latência do acesso à memória — e a confiabilidade dessa estratégia é demonstrada em todos os níveis da **hierarquia de memória**. Embora esses níveis da hierarquia pareçam muito diferentes em termos quantitativos, eles seguem estratégias semelhantes em sua operação e exploram as mesmas propriedades da localidade.



HIERARQUIA

As caches multiníveis possibilitam o uso mais fácil de outras otimizações por dois motivos. Primeiro, os parâmetros de projeto de uma cache de nível inferior são diferentes dos de uma cache de primeiro nível. Por exemplo, como uma cache de nível inferior será muito maior, é possível usar tamanhos de bloco maiores. Segundo, uma cache de nível inferior não está constantemente sendo usada pelo processador, como em uma cache de primeiro nível. Isso nos permite considerar fazer com que, quando estiver ociosa, uma cache de nível inferior realize alguma tarefa que possa ser útil para evitar futuras falhas.

Outra direção possível é recorrer à ajuda de software. Controlar eficientemente a hierarquia de memória usando uma variedade de transformações de programa e recursos de hardware é um importante foco dos avanços dos compiladores. Duas ideias diferentes estão sendo exploradas. Uma é reorganizar o programa para melhorar sua localidade espacial e temporal. Esse método focaliza os programas orientados para loops que usam grandes arrays como a principal estrutura de dados; grandes problemas de álgebra linear são um exemplo típico, como o DGEMM. Reestruturando os loops que acessam os arrays, podemos obter uma localidade — e, portanto, um desempenho de cache — substancialmente melhor.



P R E D I Ç Ã O

Outra solução é o **prefetching**. Em prefetching, um bloco de dados é trazido para a cache antes de ser realmente referenciado. Muitos microprocessadores utilizam o prefetching de hardware para tentar *prever* os acessos, o que pode ser difícil para o software observar.

prefetching

Uma técnica em que os blocos de dados necessários no futuro são colocados na cache antecipadamente pelo uso de instruções especiais que especificam o endereço do bloco.

Uma terceira técnica utiliza instruções especiais cientes da cache, que otimizam a transferência da memória. Por exemplo, os microprocessadores na [Seção 6.9](#) do [Capítulo 6](#) utilizam uma otimização que não apanha o conteúdo de um bloco da memória em uma falha de escrita, pois o programa irá escrever o bloco inteiro. Essa otimização reduz significativamente o tráfego da memória para um kernel.

Como veremos no [Capítulo 6](#), os sistemas de memória também são um importante tópico de projeto para processadores paralelos. A crescente importância da hierarquia de memória na determinação do desempenho do sistema significa que essa relevante área continuará a ser o foco de projetistas e pesquisadores ainda por vários anos.

5.15. Exercícios

5.1 Neste exercício, veremos as propriedades de localidade de memória do cálculo de matriz. O código a seguir é escrito em C, em que os elementos dentro da mesma linha são armazenados de forma contígua. Suponha que cada palavra seja um inteiro de 32 bits.

```
for (I=0; I<8; I++)
    for (J=0; J<8000; J++)
        A[I][J]=B[I][0]+A[J][I];
```

5.1.1 [5] <§5.1> Quantos inteiros de 32 bits podem ser armazenados em uma linha de cache de 16 bytes?

5.1.2 [5] <§5.1> Referências a quais variáveis exibem localidade temporal?

5.1.3 [5] <§5.1> Referências a quais variáveis exibem localidade espacial?

A localidade é afetada pela ordem de referência e pelo layout dos dados. O mesmo cálculo também pode ser escrito a seguir em Matlab, que difere da linguagem C armazenando elementos da matriz de forma contígua dentro da mesma coluna.

```

for I=1:8
    for J=1:8000
        A(I,J)=B(I,0)+A(J,I);
    end
end

```

5.1.4 [10] <§5.1> Quantos blocos de cache de 16 bytes são necessários para armazenar todos os elementos de matriz de 32 bits sendo referenciados?

5.1.5 [5] <§5.1> Referências a quais variáveis exibem localidade temporal?

5.1.6 [5] <§5.1> Referências a quais variáveis exibem localidade espacial?

5.2 As caches são importantes para fornecer uma hierarquia de memória de alto desempenho aos processadores. A seguir se encontra uma lista de referências a endereços de memória de 32 bits, dadas como endereços de palavra.

3, 180, 43, 2, 191, 88, 190, 14, 181, 44, 186, 253

5.2.1 [10] <§5.3> Para cada uma dessas referências, identifique o endereço binário, a tag e o índice dado uma cache de mapeamento direto com 16 blocos de uma palavra. Além disso, indique se cada referência é um acerto ou uma falha, supondo que a cache esteja inicialmente vazia.

5.2.2 [10] <§5.3> Para cada uma dessas referências, identifique o endereço binário, a tag e o índice dado uma cache de mapeamento direto com blocos de duas palavras e um tamanho total de oito blocos. Liste também se cada referência é um acerto ou uma falha, supondo que a cache esteja inicialmente vazia.

5.2.3 [20] <§§5.3, 5.4> Você está encarregado de otimizar um projeto de cache para as referências indicadas. Existem três projetos de cache de mapeamento direto possíveis, todos com um total de oito palavras de dados: C1 tem blocos de uma palavra, C2 tem blocos de duas palavras e C3 tem blocos de quatro palavras. Em termos de taxa de falhas, qual projeto de cache é o melhor? Se o tempo de stall de falha é de 25 ciclos, e C1 tem um tempo de acesso de 2 ciclos, C2 utiliza 3 ciclos e C3 utiliza 5 ciclos, qual é o melhor

projeto de cache?

Existem muitos parâmetros de projeto diferentes que são importantes para o desempenho geral de uma cache. A lista a seguir indica os parâmetros para diferentes projetos de cache com mapeamento direto.

Tamanho de dados da cache: 32 KiB

Tamanho de bloco da cache: 2 palavras

Tempo de acesso da cache: 1 ciclo

5.2.4 [15] <§5.3> Calcule o número total de bits necessários para a cache listada acima, considerando um endereço de 32 bits. Dado esse tamanho total, ache o tamanho total da cache de mapeamento direto mais próxima com blocos de 16 palavras do mesmo tamanho ou maior. Explique por que a segunda cache, apesar de seu tamanho de dados maior, poderia oferecer desempenho mais lento do que a primeira cache.

5.2.5 [20] <§§5.3, 5.4> Gere uma série de solicitações de leitura que possuem uma taxa de falhas em uma cache associativa em conjunto com duas vias de 2 KB inferior à cache listada na tabela. Identifique uma solução possível que faria com que a cache listada na tabela tivesse uma taxa de falhas igual ou inferior à cache de 2 KiB. Discuta as vantagens e desvantagens de uma solução desse tipo.

5.2.6 [15] <§5.3> A fórmula apresentada na [Seção 5.3](#) mostra o método típico para indexar uma cache mapeada diretamente, especificamente, (Endereço do bloco) módulo (Número de blocos na cache). Supondo um endereço de 32 bits e 1024 blocos na cache, considere uma função de indexação diferente, especificamente, (Endereço de bloco[31:27] XOR Endereço de bloco[26:22]). É possível usar isso para indexar uma cache mapeada diretamente? Se for, explique por que e discuta quaisquer mudanças que poderiam ser necessárias na cache. Se não for possível, explique o motivo.

5.3 Para um projeto de cache mapeada diretamente com endereço de 32 bits, os bits de endereço a seguir são usados para acessar a cache.

Tag	Índice	Offset
31-10	9-5	4-0

5.3.1 [5] <§5.3> Qual é o tamanho do bloco de cache (em palavras)?

5.3.2 [5] <§5.3> Quantas entradas a cache possui?

5.3.3 [5] <§5.3> Qual é a razão entre o total de bits exigido para essa implementação de cache e os bits de armazenamento de dados?

Desde que a alimentação foi ligada, as seguintes referências de cache

endereçadas por byte são registradas.

Endereço											
0	4	16	132	232	160	1024	30	140	3100	180	2180

5.3.4 [10] <§5.3> Quantos blocos são substituídos?

5.3.5 [10] <§5.3> Qual é a razão de acerto?

5.3.6 [20] <§5.3> Indique o estado final da cache, com cada entrada válida representada como um registro de <índice, tag, dados>.

5.4 Lembre-se de que temos duas políticas de escrita e políticas de alocação de escrita; suas combinações podem ser implementadas na cache L1 ou L2.

Considere as seguintes escolhas para as caches L1 e L2:

L1	L2
Write-through, sem alocação de escrita	Write-back, alocação de escrita

5.4.1 [5] <§§5.3, 5.8> Os buffers são empregados entre diferentes níveis de hierarquia da memória para reduzir a latência de acesso. Para essa configuração dada, liste os possíveis buffers necessários entre as caches L1 e L2, bem como entre a cache L2 e a memória.

5.4.2 [20] <§§5.3, 5.8> Descreva o procedimento de tratamento de uma falha de escrita em L1, considerando o componente envolvido e a possibilidade de substituir um bloco modificado.

5.4.3 [20] <§§5.3, 5.8> Para uma configuração de cache exclusiva multinível (um bloco só pode residir em uma das caches L1 e L2), descreva o procedimento de tratamento de uma falha de escrita em L1, considerando o componente envolvido e a possibilidade de substituir um bloco modificado. Considere os seguintes comportamentos do programa e da cache.

Leituras de dados por 1000 instruções	Escritas de dados por 1000 instruções	Taxa de perdas da cache de instruções	Taxa de perdas da cache de dados	Tamanho do bloco (byte)
250	100	0,30%	2%	64

5.4.4 [5] <§§5.3, 5.8> Para uma cache write-through, com alocação de escrita, quais são as larguras de banda mínimas de leitura e escrita (medidas em bytes-por-ciclo) necessárias para alcançar um CPI de 2?

5.4.5 [5] <§§5.3, 5.8> Para uma cache write-back, com alocação de escrita, considerando que 30% dos blocos de cache de dados substituídos são modificados, quais são as larguras de banda mínimas de leitura e escrita

necessárias para um CPI de 2?

5.4.6 [5] <§§5.3, 5.8> Quais são as larguras mínimas de banda necessárias para alcançar o desempenho de CPI = 1,5?

5.5 Aplicações de mídia que tocam arquivos de áudio ou vídeo fazem parte de uma classe de carga de trabalho chamada “streaming”; ou seja, elas trazem grandes quantidades de dados, mas não reutilizam grande parte dele.

Considere uma carga de trabalho de streaming de vídeo que acessa um conjunto de trabalho de 512 KiB sequencialmente com o fluxo de endereço a seguir:

0, 2, 4, 6, 8, 10, 12, 14, 16, ...

5.5.1 [5] <§§5.4, 5.8> Considere um cache com mapeamento direto de 64 KiB com um bloco de 32 bytes. Qual é a taxa de falhas para esse fluxo de endereços? De que modo essa taxa de falhas é sensível ao tamanho da cache ou ao conjunto de trabalho? Como você categorizaria as falhas que essa carga de trabalho está experimentando, com base no modelo 3C?

5.5.2 [5] <§§5.1, 5.8> Recalcule a taxa de falhas quando o tamanho do bloco de cache é de 16 bytes, 64 bytes e 128 bytes. Que tipo de localidade essa carga de trabalho está explorando?

5.5.3 [10] <§5.13> “Prefetching” é uma técnica que aproveita padrões de endereço previsíveis para trazer blocos de cache adicionais quando determinado bloco de cache é acessado. Um exemplo de prefetching é um buffer de fluxo que pré-busca blocos de cache sequencialmente adjacentes em um buffer separado quando determinado bloco de cache é trazido. Se os dados forem encontrados no buffer de prefetch, eles são considerados um acerto e movidos para a cache, e o próximo bloco de cache é pré-buscado. Considere um buffer de stream de duas entradas e suponha que a latência da cache seja tal que um bloco de cache possa ser carregado antes que o cálculo no bloco de cache anterior seja concluído. Qual é a taxa de falhas para esse stream de endereços?

O tamanho do bloco de cache (B) pode afetar a taxa de falhas e a latência de falha. Considerando uma máquina de 1 CPI com uma média de 1,35 referências (a instruções e dados) por instrução, ajude a encontrar o tamanho de bloco ideal dadas as seguintes taxas de falha para diversos tamanhos de bloco.

8: 4%	16: 3%	32: 2%	64: 1,5%	128: 1%
-------	--------	--------	----------	---------

5.5.4 [10] <§5.3> Qual é o tamanho de bloco ideal para uma latência de falha

de $20 \times B$ ciclos?

5.5.5 [10] <§5.3> Qual é o tamanho de bloco ideal para uma latência de falha de $24 + B$ ciclos?

5.5.6 [10] <§5.3> Para uma latência de falha constante, qual é o tamanho de bloco ideal?

5.6 Neste exercício, veremos as diferentes maneiras como a capacidade afeta o desempenho geral. Normalmente, o tempo de acesso da cache é proporcional à capacidade. Suponha que os acessos à memória principal utilizem 70 ns e que os acessos à memória sejam 36% de todas as instruções. A tabela a seguir mostra dados para caches L1 relacionados a cada um dos dois processadores, P1 e P2.

	Tamanho L1	Taxa de falhas L1	Tempo de acerto L1
P1	2 KiB	8,0%	0,66 ns
P2	4 KiB	6,0%	0,90 ns

5.6.1 [5] <§5.4> Considerando que o tempo de acerto de L1 determina os tempos de ciclo para P1 e P2, quais são suas respectivas taxas de clock?

5.6.2 [5] <§5.4> Qual é o TMAM para cada um de P1 e P2?

5.6.3 [5] <§5.4> Considerando um CPI base de 1,0 sem stalls de memória, qual é o CPI total para cada um de P1 e P2? Que processador é mais rápido?

Para os três problemas a seguir, vamos considerar o acréscimo de uma cache L2 para P1, a fim de, possivelmente, compor sua capacidade limitada de cache L1. Use as capacidades e tempos de acerto da cache L1 da tabela anterior ao resolver esses problemas. A taxa de falhas L2 indicada é a sua taxa de falhas local.

Tamanho L2	Taxa de falhas L2	Tempo de acerto L2
1 MB	95%	5,62 ns

5.6.4 [10] <§5.4> Qual é o TMAM para P1 com o acréscimo de uma cache L2? O TMAM é melhor ou pior com a cache L2?

5.6.5 [5] <§5.4> Considerando um CPI base de 1,0 sem stalls de memória, qual é o CPI total para P1 com a adição de um cache L2?

5.6.6 [10] <§5.4> Que processador é mais rápido, agora que P1 tem uma cache L2? Se P1 é mais rápido, que taxa de falhas P2 precisaria em sua cache L1 para corresponder ao desempenho de P1? Se P2 é mais rápido, que taxa de falhas P1 precisaria em seu cache L1 para corresponder ao desempenho de

P2?

5.7 Este exercício examina o impacto de diferentes projetos de cache, especificamente comparando caches associativas com as caches mapeadas diretamente, da [Seção 5.4](#). Para estes exercícios, consulte a tabela de streams de endereço mostrada no Exercício 5.2.

5.7.1 [10] <§5.4> Usando as referências do Exercício 5.2, mostre o conteúdo final da cache para uma cache associativa em conjunto com três vias, com blocos de duas palavras e um tamanho total de 24 palavras. Use a substituição LRU. Em cada referência, identifique os bits de índice, os bits de tag, os bits de offset de bloco e se é um acerto ou uma perda.

5.7.2 [10] <§5.4> Usando as referências do Exercício 5.2, mostre o conteúdo final da cache para uma cache totalmente associativa com blocos de uma palavra e um tamanho total de oito palavras. Use a substituição LRU. Para cada referência, identifique os bits de índice, os bits de tag, e se é um acerto ou uma perda.

5.7.3 [15] <§5.4> Usando as referências do Exercício 5.2, qual é a taxa de perdas para uma cache totalmente associativa com blocos de duas palavras e um tamanho total de oito palavras, usando a substituição LRU? Qual é a taxa de perdas usando a substituição MRU (usado mais recentemente)? Finalmente, qual é a melhor taxa de perdas possível para essa cache, dada qualquer política de substituição?

O caching multinível é uma técnica importante para contornar a quantidade limitada do espaço que uma cache de primeiro nível pode oferecer enquanto mantém sua velocidade. Considere um processador com os seguintes parâmetros:

CPI base, sem stalls da memória	Velocidade do processador	Tempo de acesso à memória principal	Taxa de perdas da cache de 1º nível por instrução	Cache de segundo nível, velocidade mapeada diretamente	Taxa de perda global com cache de 2º nível, mapeada diretamente	Cache de segundo nível, velocidade associativa em conjunto com oito vias	Taxa de perda global com cache de 2º nível, associativo em conjunto com oito vias
1,5	2 GHz	100 ns	7%	12 ciclos	3,5%	28 ciclos	1,5%

5.7.4 [10] <§5.4> Calcule o CPI para o processador na tabela usando: 1) apenas uma cache de primeiro nível, 2) uma cache de mapeamento direto de

segundo nível, e 3) uma cache associativa em conjunto com oito vias de segundo nível. Como esses números mudam se o tempo de acesso da memória principal for dobrado? E se for cortado ao meio?

5.7.5 [10] <§5.4> É possível ter uma hierarquia de cache ainda maior que dois níveis. Dado o processador anterior com uma cache de segundo nível mapeada diretamente, um projetista deseja acrescentar uma cache de terceiro nível que leve 50 ciclos para acessar e que reduzirá a taxa de falhas global para 1,3%. Isso ofereceria melhor desempenho? Em geral, quais são as vantagens e desvantagens de acrescentar uma cache de terceiro nível?

5.7.6 [20] <§5.4> Em processadores mais antigos, como o Intel Pentium e o Alpha 21264, o segundo nível de cache era externo (localizado em um chip diferente) ao processador principal e à cache de primeiro nível. Embora isso permitisse grandes caches de segundo nível, a latência para acessar a cache era muito mais alta, e a largura de banda normalmente era menor, pois a cache de segundo nível trabalhava em uma frequência inferior. Suponha que uma cache de segundo nível de 512 KiB fora do chip tenha uma taxa de perdas global de 4%. Se cada 512 KiB adicionais de cache reduzisse as taxas de perdas globais em 0,7% e a cache tivesse um tempo de acesso total de 50 ciclos, que tamanho a cache deveria ter para corresponder ao desempenho da cache de segundo nível mapeada diretamente, listada na tabela? E ao desempenho da cache associativa em conjunto com oito vias?

5.8 Tempo médio entre falhas (MTBF), tempo médio para o reparo (MTTR) e tempo médio para falhas (MTTF) são métricas úteis para avaliar a confiabilidade e a disponibilidade de um recurso de armazenamento. Explore esses conceitos respondendo às perguntas sobre dispositivos com as métricas a seguir.

MTTF	MTTR
3 anos	1 dia

5.8.1 [5] <§5.5> Calcule o MTBF para os dispositivos com as métricas da tabela.

5.8.2 [5] <§5.5> Calcule a disponibilidade para os dispositivos com as métricas na tabela.

5.8.3 [5] <§5.5> O que acontece com a disponibilidade quando o MTTR aproxima-se de 0? Essa é uma situação realista?

5.8.4 [5] <§5.5> O que acontece com a disponibilidade quando o MTTR fica muito alto, ou seja, um dispositivo é difícil de ser reparado? Isso implica que

o dispositivo possui baixa disponibilidade?

5.9 Este exercício examina o código de Hamming de correção de erro único e detecção de erro duplo (SEC/DED).

5.9.1 [5] <§5.5> Qual é o número mínimo de bits de paridade exigidos para proteger uma palavra de 128 bits usando o código SEC/DED?

5.9.2 [5] <§5.5> A [Seção 5.5](#) indica que os módulos de memória de servidor moderno (DIMMs) empregam o ECC SEC/DED para proteger cada 64 bits com 8 bits de paridade. Calcule a razão custo/desempenho desse código com o código do Exercício 5.9.1. Neste caso, o custo é o número relativo de bits de paridade necessários, enquanto o desempenho é o número relativo de erros que podem ser corrigidos. Qual é o melhor?

5.9.3 Considere um código SEC que protege palavras de 8 bits com 4 bits de paridade. Se lêssemos o valor 0x375, haveria um erro? Se sim, corrija o erro.

5.10 Para um sistema de alto desempenho, como um índice B-tree para banco de dados, o tamanho de página é determinado principalmente pelo tamanho dos dados e pelo desempenho do disco. Suponha que, na média, uma página de índice B-tree esteja 70% cheia com entradas de tamanho fixo. A utilidade de uma página é sua profundidade de B-tree, calculada como $\log_2(\text{entradas})$. A tabela a seguir mostra que, para entradas de 16 bytes, um disco com dez anos de uso, uma latência de 10 ms e uma taxa de transferência de 10 MB/s, o tamanho de página ideal é de 16K.

Tamanho de página (KiB)	Utilidade da página ou profundidade da B-tree (número de acessos ao disco salvos)	Custo do acesso à página de índice (ms)	Utilidade/custo
2	6,49 (ou $\log_2(2048/16 \times 0,7)$)	10,2	0,64
4	7,49	10,4	0,72
8	8,49	10,8	0,79
16	9,49	11,6	0,82
32	10,49	13,2	0,79
64	11,49	16,4	0,70
128	12,49	22,8	0,55
256	13,49	35,6	0,38

5.10.1 [10] <§5.7> Qual é o melhor tamanho de página se as entradas agora tiverem 128 bytes?

5.10.2 [10] <§5.7> Com base no Exercício 5.10.1, qual é o melhor tamanho de página se as páginas estiverem completas até a metade?

5.10.3 [20] <§5.7> Com base no Exercício 5.10.2, qual é o melhor tamanho de

página se for usado um disco moderno com latência de 3 ms e uma taxa de transferência de 100 MB/s? Explique por que os servidores futuros provavelmente terão páginas maiores.

Manter páginas “frequentemente utilizadas” (ou “quentes”) na DRAM pode economizar acessos ao disco, mas como determinamos o significado exato de “frequentemente utilizadas” para determinado sistema? Os engenheiros de dados utilizam a razão de custo entre o acesso à DRAM e ao disco para quantificar o patamar de tempo de reuso para as páginas quentes. O custo de um acesso ao disco é $\$/\text{Disco}/\text{acessos_por_segundo}$, enquanto o custo de manter uma página na DRAM é $\$/\text{DRAM_MiB}/\text{tamanho_pag}$. Os custos típicos de DRAM e disco, e os tamanhos típicos de página de banco de dados em diversos pontos no tempo, são listados a seguir:

Ano	Custo da DRAM (\$/MiB)	Tamanho da página (KiB)	Custo do disco (\$/disco)	Taxa de acesso ao disco (acesso/seg)
1987	5000	1	15000	15
1997	15	8	2000	64
2007	0,05	64	80	83

5.10.4 [10] <§§5.1, 5.7> Quais são os patamares do tempo de reutilização para essas três gerações de tecnologia?

5.10.5 [10] <§5.7> Quais são os patamares do tempo de reutilização se continuarmos usando o mesmo tamanho de página de 4K? Qual é a tendência aqui?

5.10.6 [20] <§5.7> Que outros fatores podem ser alterados para continuar usando o mesmo tamanho de página (evitando assim a reescrita de software)? Discuta sua probabilidade com as tendências atuais de tecnologia e custo.

5.11 Conforme descrevemos na [Seção 5.7](#), a memória virtual utiliza uma tabela de página para rastrear o mapeamento entre endereços virtuais e endereços físicos. Este exercício mostra como essa tabela precisa ser atualizada enquanto os endereços são acessados. A tabela a seguir é um stream de endereços virtuais vistos em um sistema. Considere páginas de 4 KiB, uma TLB totalmente associativa com quatro entradas, e substituição LRU verdadeira. Se as páginas tiverem de ser trazidas do disco, incremente o próximo número de página maior.

4669, 2227, 13916, 34587, 48870, 12608, 49225

TLB

Válido	Tag	Número da página física
1	11	12
1	7	4
1	3	6
0	4	9

Tabela de página

Válido	Página física ou no disco
1	5
0	Disco
0	Disco
1	6
1	9
1	11
0	Disco
1	4
0	Disco
0	Disco
1	3
1	12

5.11.1 [10] <§5.7> Dado o stream de endereços na tabela, e o estado inicial mostrado da TLB e da tabela de página, mostre o estado final do sistema. Indique também, para cada referência, se ela é um acerto na TLB, um acerto na tabela de página ou uma falta de página.

5.11.2 [15] <§5.7> Repita o Exercício 5.11.1, mas desta vez use páginas de 16 KiB em vez de páginas de 4 KiB. Quais seriam algumas das vantagens de ter um tamanho de página maior? Quais são algumas das desvantagens?

5.11.3 [15] <§§5.4, 5.7> Mostre o conteúdo final da TLB se ela for associativa em conjunto com duas vias. Mostre também o conteúdo da TLB se ela for mapeada diretamente. Discuta a importância de se ter uma TLB para o desempenho mais alto. Como seriam tratados os acessos à memória virtual se não houvesse TLB?

Existem vários parâmetros que afetam o tamanho geral da tabela de página.

A seguir estão listados diversos parâmetros importantes da tabela de página.

Tamanho do endereço virtual	Tamanho da página	Tamanho da entrada da tabela de página
32 bits	8 KiB	4 bytes

5.11.4 [5] <§5.7> Dados os parâmetros nessa tabela, calcule o tamanho total da tabela de página para um sistema, executando cinco aplicações que utilizam metade da memória disponível.

5.11.5 [10] <§5.7> Dados os parâmetros na tabela anterior, calcule o tamanho total da tabela de página para um sistema executando cinco aplicações que utilizam metade da memória disponível, dada uma técnica de tabela de página de dois níveis com 256 entradas. Suponha que cada entrada da tabela de página principal seja de 6 bytes. Calcule a quantidade mínima e máxima de memória exigida.

5.11.6 [10] <§5.7> Um projetista de cache deseja aumentar o tamanho de uma cache de 4 KiB indexada virtualmente e marcada fisicamente com tags. Dado o tamanho de página listado na tabela anterior, é possível criar uma cache de 16 KiB com mapeamento direto, considerando duas palavras por bloco? Como o projetista aumentaria o tamanho dos dados da cache?

5.12 Neste exercício, examinaremos as otimizações de espaço/tempo para as tabelas de página. A tabela a seguir mostra parâmetros de um sistema de memória virtual.

Endereço virtual (bits)	DRAM física instalada	Tamanho da página	Tamanho da PTE (bytes)
43	16 GiB	4 KiB	4

5.12.1 [10] <§5.7> Para uma tabela de página de único nível, quantas entradas da tabela de página (PTE) são necessárias? O quanto de memória física é necessário para armazenar a tabela de página?

5.12.2 [10] <§5.7> O uso de uma tabela de página multinível pode reduzir o consumo de memória física das tabelas de página, apenas mantendo as PTEs ativas na memória física. Quantos níveis de tabelas de página serão necessários nesse caso? E quantas referências de memória são necessárias para a tradução de endereço se estiverem faltando na TLB?

5.12.3 [15] <§5.7> Uma tabela de página invertida pode ser usada para otimizar ainda mais o espaço e o tempo. Quantas PTEs são necessárias para armazenar a tabela de página? Considerando uma implementação de tabela de hash, quais são os números do caso comum e do pior caso das referências à memória necessárias para atender a uma falta de TLB?

A tabela a seguir mostra o conteúdo de uma TLB com quatro entradas.

ID entrada	Válido	Página VA	Modificado	Proteção	Página PA
1	1	140	1	RW	30
2	0	40	0	RX	34
3	1	200	1	RO	32
4	1	280	0	RW	31

5.12.4 [5] <§5.7> Sob que cenários o bit de validade da entrada 2 seria definido como 0?

5.12.5 [5] <§5.7> O que acontece quando uma instrução escreve na página VA 30? Quando uma TLB controlada por software seria mais rápido que uma TLB controlada por hardware?

5.12.6 [5] <§5.7> O que acontece quando uma instrução escreve na página VA 200?

5.13 Neste exercício, examinaremos como as políticas de substituição afetam a taxa de falhas. Considere uma cache associativa em conjunto com duas vias e quatro blocos. Você poderá achar útil desenhar uma tabela para solucionar os problemas neste exercício, conforme demonstramos nesta sequência de endereços: 0, 1, 2, 3, 4.

Endereço do bloco de memória acessado	Acerto ou falha	Bloco expulso	Conteúdo dos blocos de cache após referência			
			Conjunto 0	Conjunto 0	Conjunto 1	Conjunto 1
0	Falha		Mem[0]			
1	Falha		Mem[0]		Mem[1]	
2	Falha		Mem[0]	Mem[2]	Mem[1]	
3	Falha		Mem[0]	Mem[2]	Mem[1]	Mem[3]
4	Falha	0	Mem[4]	Mem[2]	Mem[1]	Mem[3]
...						

Considere a seguinte sequência de endereços: 0, 2, 4, 8, 10, 12, 14, 16, 0

5.13.1 [5] <§§5.4, 5.8> Considerando uma política de substituição LRU, quantos acertos essa sequência de endereços exibe?

5.13.2 [5] <§§5.4, 5.8> Considerando uma política de substituição MRU (usado mais recentemente), quantos acertos essa sequência de endereços exibe?

5.13.3 [5] <§§5.4, 5.8> Simule uma política de substituição aleatória lançando uma moeda. Por exemplo, “cara” significa expulsar o primeiro bloco em um conjunto e “coroa” significa expulsar o segundo bloco em um conjunto. Quantos acertos essa sequência de endereços exibe?

5.13.4 [10] <§§5.4, 5.8> Que endereço deve ser expulso em cada substituição para maximizar o número de acertos? Quantos acertos essa sequência de endereços exibe se você seguir essa política “ideal”?

5.13.5 [10] <§§5.4, 5.8> Descreva por que é difícil implementar uma política de substituição de cache que seja ideal para todas as sequências de endereço.

5.13.6 [10] <§§5.4, 5.8> Considere que você poderia tomar uma decisão em cada referência de memória se deseja ou não que o endereço requisitado seja mantido em cache. Que impacto poderia ter sobre a taxa de falhas?

5.14 Para dar suporte às máquinas virtuais, dois níveis de virtualização de memória são necessários. Cada máquina virtual ainda controla o mapeamento entre o endereço virtual (VA) e o endereço físico (PA), enquanto o hipervisor mapeia o endereço físico (PA) de cada máquina virtual e o endereço de máquina (MA) real. Para acelerar esses mapeamentos, uma técnica de software chamada “paginação de shadow” duplica as tabelas de página de cada máquina virtual no hipervisor, e intercepta as mudanças de mapeamento entre VA e PA para manter as duas cópias coerentes. A fim de remover a complexidade das tabelas de página de shadow, uma técnica de hardware chamada tabela de página aninhada (ou tabela de página estendida) oferece suporte explícito a duas classes de tabelas de página (VA → PA e PA → MA) e pode percorrer essas tabelas apenas no hardware.

Considere esta sequência de operações: (1) Criar processo; (2) Falha de TLB; (3) Falta de página; (4) Troca de contexto;

5.14.1 [10] <§§5.6, 5.7> O que aconteceria à sequência de operação indicada, para a tabela de página de shadow e a tabela de página aninhada, respectivamente?

5.14.2 [10] <§§5.6, 5.7> Considerando uma tabela de página de quatro níveis baseada em x86 tanto na tabela de página guest quanto na aninhada, quantas referências à memória são necessárias para atender a uma falha de TLB para a tabela de página nativa *versus* aninhada?

5.14.3 [15] <§§5.6, 5.7> Entre a taxa de falhas de TLB, latência de falha de TLB, taxa de falta de página e latência do tratador de falta de página, quais métricas são mais importantes para a tabela de página de shadow? Quais são importantes para a tabela de página aninhada?

A tabela a seguir mostra parâmetros para um sistema de página por sombra.

Falhas de TLB por 1000 instruções	Latência de falha de TLB NPT	Faltas de página por 1000 instruções	Overhead de shadowing por falta de página
0,2	200 ciclos	0,001	30.000 ciclos

- 5.14.4** [10] <§5.6> Para um benchmark com CPI de execução nativo de 1, quais são os números de CPI se estiver usando tabelas de página de shadow versus NPT (considerando apenas o overhead de virtualização da tabela de página)?
- 5.14.5** [10] <§5.6> Que técnicas podem ser usadas para reduzir o overhead induzido pelo shadowing da tabela de página?
- 5.14.6** [10] <§5.6> Que técnicas podem ser usadas para reduzir o overhead induzido pelo NPT?
- 5.15** Um dos maiores impedimentos para o uso generalizado das máquinas virtuais é o overhead de desempenho ocasionado pela execução de uma máquina virtual. A tabela a seguir lista diversos parâmetros de desempenho e comportamento de aplicação.

CPI base	Acessos privilegiados do O/S por 10.000 instruções	Impacto no desempenho de interceptar o O/S guest	Impacto no desempenho de interceptar a VMM	Acessos de E/S por 10.000 instruções	Tempo de acesso de E/S (inclui tempo para interceptar o O/S guest)
1,5	120	15 ciclos	175 ciclos	30	1100 ciclos

- 5.15.1** [10] <§5.6> Calcule o CPI para o sistema listado, supondo que não existem acessos à E/S. Qual é o CPI se o impacto do desempenho da VMM dobrar? E se for cortado ao meio? Se uma empresa de software da máquina virtual deseja obter uma degradação de desempenho de 10%, qual é a maior penalidade possível para interceptar a VMM?
- 5.15.2** [10] <§5.6> Os acessos de E/S normalmente possuem um grande impacto sobre o desempenho geral do sistema. Calcule o CPI de uma máquina usando as características de desempenho anteriores, considerando um sistema não virtualizado. Calcule o CPI novamente, desta vez usando um sistema virtualizado. Como esses CPIs mudam se o sistema tiver metade dos acessos de E/S? Explique por que as aplicações voltadas para E/S possuem um impacto semelhante da virtualização.
- 5.15.3** [30] <§§5.6, 5.7> Compare as ideias da memória virtual e das máquinas virtuais. Como os objetivos de cada um se comparam? Quais são os prós e contras de cada um? Liste alguns casos em que a memória virtual é desejada, e alguns casos em que as máquinas virtuais são desejadas.
- 5.15.4** [20] <§5.6> A [Seção 5.6](#) discute a virtualização sob a hipótese de que o sistema virtualizado esteja executando a mesma ISA do hardware subjacente. Porém, um uso possível da virtualização é simular ISAs não nativas. Um

exemplo disso é QEMU, que simula uma série de ISAs, como o MIPS, SPARC e PowerPC. Quais são algumas das dificuldades envolvidas nesse tipo de virtualização? É possível que um sistema simulado rode mais rápido do que em sua ISA nativa?

5.16 Neste exercício, exploraremos a unidade de controle de um controlador de cache para um processador com um buffer de escrita. Use a máquina de estados finitos encontrada na [Figura 5.40](#) como ponto de partida para projetar suas próprias máquinas de estados finitos. Suponha que o controlador de cache seja para a cache de mapeamento direto descrita na [Seção 5.9](#), mas você acrescentará um buffer de escrita com uma capacidade de um bloco. Lembre-se de que a finalidade de um buffer de escrita é servir como armazenamento temporário, de modo que o processador não precisa esperar por dois acessos à memória em uma falha modificada. Em vez de escrever de volta o bloco modificado antes de ler o novo bloco, ele coloca o bloco modificado no buffer e começa imediatamente a ler o novo bloco. O bloco modificado pode então ser escrito na memória principal enquanto o processador está trabalhando.

5.16.1 [10] <§§5.8, 5.9> O que deve acontecer se o processador emitir uma solicitação que *acerta* na cache enquanto um bloco está sendo escrito de volta na memória principal a partir do buffer de escrita?

5.16.2 [10] <§§5.8, 5.9> O que deve acontecer se o processador emitir uma solicitação que *falha* na cache enquanto um bloco está sendo escrito de volta à memória principal a partir do buffer de escrita?

5.16.3 [30] <§§5.8, 5.9> Crie uma máquina de estado finito para permitir o uso de um buffer de escrita.

5.17 A coerência da cache refere-se às visões de múltiplos processadores em determinado bloco de cache. A tabela a seguir mostra dois processadores e suas operações de leitura/escrita em duas palavras diferentes de um bloco de cache X (inicialmente, $X[0] = X[1] = 0$). Considere que o tamanho dos inteiros seja de 32 bits.

P1	P2
$X[0] \text{ } ++;$ $X[1] = 3;$	$X[0] = 5;$ $X[1] += 2;$

5.17.1 [15] <§5.10> Liste os valores possíveis do bloco de cache indicado para uma implementação correta do protocolo de coerência de cache. Liste pelo menos um valor possível do bloco se o protocolo não garantir coerência de cache.

5.17.2 [15] <§5.10> Para um protocolo de snooping, liste uma sequência de operação válida em cada processador/cache para terminar as operações de leitura/escrita listadas anteriormente.

5.17.3 [10] <§5.10> Quais são os números, no melhor caso e no pior caso das falhas de cache, necessários para terminar as instruções de leitura/escrita listadas?

A coerência da memória refere-se às visões de múltiplos itens de dados. A tabela a seguir mostra dois processadores e suas operações de leitura/escrita em diferentes blocos de cache (A e B inicialmente 0).

P1	P2
$A = 1; B = 2; A += 2; B ++;$	$C = B; D = A;$

5.17.4 [15] <§5.10> Liste os valores possíveis de C e D para uma implementação que garante as suposições de consistência no início da [Seção 5.10](#).

5.17.5 [15] <§5.10> Liste pelo menos um par possível de valores para C e D se essas suposições não forem mantidas.

5.17.6 [15] <§§5.3, 5.10> Para diversas combinações de políticas de escrita e políticas de alocação de escrita, quais combinações tornam a implementação do protocolo mais simples?

5.18 Multiprocessadores em um chip (CMPs) possuem diversos cores e suas caches em um único chip. O projeto da cache L2 no chip CMP possui opções interessantes. A tabela a seguir mostra as taxas de falhas e as latências de acerto para dois benchmarks com projetos de cache L2 privada *versus* compartilhada. Considere falhas de cache L1 uma vez a cada 32 instruções.

	Privada	Compartilhada
Falhas por instrução no benchmark A	0,30%	0,12%
Falhas por instrução no benchmark B	0,06%	0,03%

Considere as seguintes latências de acerto:

Cache privada	Cache compartilhada	Memória
5	20	180

5.18.1 [15] <§5.13> Qual projeto de cache é melhor para cada um desses benchmarks? Use dados para apoiar sua conclusão.

5.18.2 [15] <§5.13> A latência da cache compartilhada aumenta com o

tamanho do CMP. Escolha o melhor projeto se a latência da cache compartilhada dobrar. Como a largura de banda fora do chip torna-se o gargalo à medida que o número de cores CMP aumenta, escolha o melhor projeto se a latência da memória fora do chip dobrar.

5.18.3 [10] <§5.13> Discuta os prós e os contras das caches L2 compartilhada *versus* privada para cargas de trabalho de único thread, multithreaded e multiprogramadas, e reconsidere-as se houver caches L3 no chip.

5.18.4 [15] <§5.13> Considere que ambos os benchmarks têm um CPI base de 1 (cache L2 ideal). Se ter uma cache sem bloqueio melhora o número médio de falhas L2 concorrentes de 1 para 2, quanta melhoria de desempenho isso oferece sobre uma cache L2 compartilhada? Quanta melhoria pode ser obtida sobre a L2 privada?

5.18.5 [10] <§5.13> Supondo que novas gerações de processadores dobrem o número de núcleos a cada 18 meses, para manter o mesmo nível de desempenho por núcleo, quanta largura de banda fora do chip a mais é necessária para um processador lançado em três anos?

5.18.6 [15] <§5.13> Considerando a hierarquia de memória inteira, que tipos de otimizações podem melhorar o número de falhas simultâneas?

5.19 Neste exercício, mostramos a definição de um log de servidor Web e examinamos otimizações de código para melhorar a velocidade de processamento do log. A estrutura de dados para o log é definida da seguinte forma:

```
struct entry {  
    int srcIP; // endereço IP remoto  
    char URL[128]; // URL solicitado (p.e., "GET index.html")  
    long long refTime; // tempo de referência  
    int status; // status da conexão  
    char browser[64]; // nome do navegador cliente  
} log [NUM_ENTRIES];
```

Considere a seguinte função de processamento em um log:

```
topK_sourceIP (int hour);
```

5.19.1 [5] <§5.15> Quais campos em uma entrada de log serão acessados para a função de processamento de log indicada? Considerando blocos de cache de 64 bytes e nenhum prefetching, quantas falhas de cache por entrada determinada função, contrai na média?

5.19.2 [10] <§5.15> Como você pode reconhecer a estrutura de dados para melhorar a utilização da cache e a localidade do acesso? Mostre seu código de definição da estrutura.

5.19.3 [10] <§5.15> Dê um exemplo de outra função de processamento de log que preferiria um layout de estrutura de dados diferente. Se ambas as funções são importantes, como você reescreveria o programa para melhorar o desempenho geral? Suplemente a discussão com um trecho de código e dados.

Para os problemas a seguir, use os dados de “Cache Performance for SPEC CPU2000 Benchmarks”

(www.cs.wisc.edu/multifacet/misc/spec2000cache-data/) para os pares de benchmarks mostrados na tabela a seguir.

a.	Mesa / gcc
b.	mcf / swim

5.19.4 [10] <§5.15> Para caches de dados de 64 KiB com associatividades de conjunto variadas, quais são as taxas de falhas desmembradas por tipos de falha (falhas frias, de capacidade e de conflito) para cada benchmark?

5.19.5 [10] <§5.15> Selecione a associatividade de conjunto a ser usada por uma cache de dados L1 de 64 KiB compartilhada por ambos os benchmarks. Se a cache L1 tiver de ser mapeada diretamente, selecione a associatividade de conjunto para a cache L2 de 1 MiB.

5.19.6 [20] <§5.15> Dê um exemplo na tabela de taxa de falhas em que a associatividade de conjunto mais alta aumenta a taxa de falhas. Construa uma configuração de cache e fluxo de referência para demonstrar isso.

Respostas das Seções “Verifique você mesmo”

§5.1, página 330: 1 e 4 (3 é falso porque o custo da hierarquia de memória varia por computador, mas em 2013 o custo mais alto normalmente é a DRAM.)

§5.3, página 348: 1 e 4: Uma penalidade de falha menor pode levar a blocos menores, pois você não tem tanta latência para amortizar, embora uma ~~latência da banda da memória mais alta normalmente leve a blocos maiores~~

~~largaia ue vamia ue memoria mais alta informaçao teve a vicos maiores,~~

já que a penalidade de falha é apenas ligeiramente maior.

§5.4, página 365: 1.

§5.7, página 397: 1-a, 2-c, 3-b, 4-d.

§5.8, página 403: 2. (Tanto os tamanhos de bloco maiores quanto o prefetching podem reduzir as falhas compulsórias, de modo que 1 é falso.)

Processadores paralelos do cliente à nuvem

“Balanço bastante, com tudo o que tenho.

Rebato com força ou perco com força.

Gosto de viver ao máximo que eu posso.”

Babe Ruth Jogador americano de beisebol

6.1 Introdução

6.2 A dificuldade de criar programas com processamento paralelo

6.3 SISD, MIMD, SIMD, SPMD e vetor

6.4 Multithreading do hardware

6.5 Multicore e outros multiprocessadores de memória compartilhada

6.6 Introdução às unidades de processamento de gráficos

6.7 Clusters, computadores em escala warehouse e outros multiprocessadores de passagem de mensagens

6.8 Introdução às topologias de rede multiprocessador

6.9 Benchmarks de multiprocessador e modelos de desempenho

6.10 Vida real: benchmarking e rooflines do Intel Core i7 e GPU NVIDIA Tesla

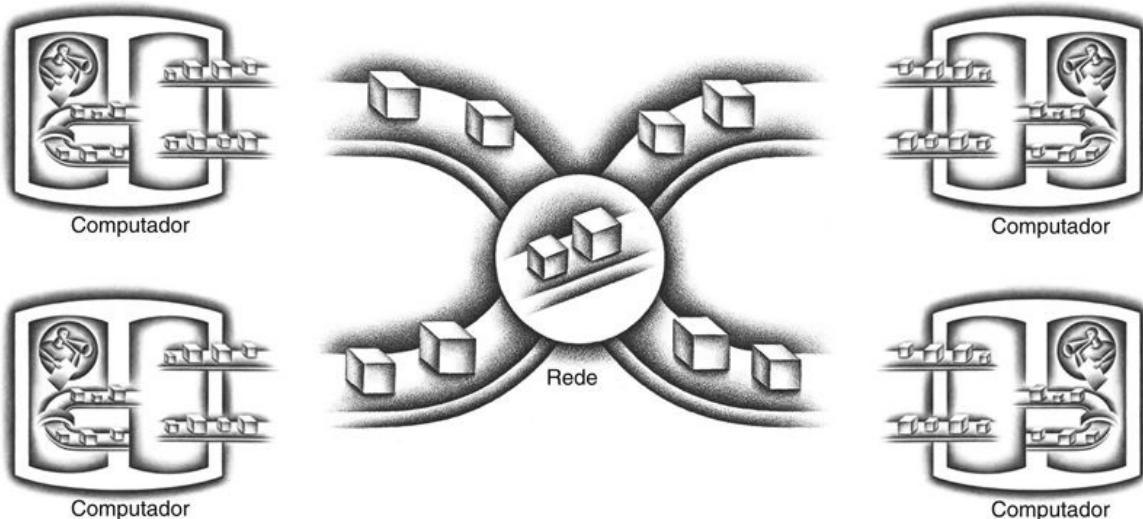
6.11 Mais rápido: processadores múltiplos e multiplicação matricial

6.12 Falácia e armadilhas

6.13 Comentários finais

6.14 Exercícios

Organização de multiprocessador ou cluster



6.1. Introdução

“Sobre as montanhas da lua, pelo vale das sombras, cavague, cavague corajosamente”, respondeu a sombra — “Se você procura o Eldorado!”

Edgar Allan Poe, “Eldorado”, quarta estrofe, 1849

Há muito tempo, os arquitetos de computadores têm buscado o Eldorado do projeto de computadores: criar computadores poderosos simplesmente conectando muitos computadores menores existentes. Esta visão dourada é a origem dos **multiprocessadores**. Idealmente, o cliente pede tantos processadores quanto seu orçamento permitir e recebe uma quantidade correspondente de desempenho. Portanto, o software para multiprocessadores precisa ser projetado para trabalhar com um número variável de processadores. Como dissemos no [Capítulo 1](#), a potência tornou-se o fator limitante para centros de dados e microprocessadores. Substituir grandes processadores ineficazes por muitos processadores eficazes e menores pode oferecer melhor desempenho por watt ou

por joule tanto no grande quanto no pequeno, se o software puder utilizá-los com eficiência. Assim, a melhor eficiência de energia se junta ao desempenho escalável no caso para os multiprocessadores.

multiprocessador

Um sistema de computador com, pelo menos, dois processadores. Isso é o contrário de um **uniprocessador**, que tem apenas um, e é cada vez mais difícil de encontrar hoje.

Como o software multiprocessador é escalável, alguns projetos podem suportar operar mesmo com a ocorrência de quebras no hardware; ou seja, se um único processador falhar em um multiprocessador com n processadores, o sistema fornece serviço continuado com $n - 1$ processadores. Portanto, os multiprocessadores também podem melhorar a disponibilidade ([Capítulo 5](#)).

Alto desempenho pode significar alta vazão para tarefas independentes, chamado **paralelismo em nível de tarefa**, ou **paralelismo em nível de processo**. Essas tarefas paralelas são aplicações independentes de única thread, e são um uso importante e comum dos processadores múltiplos. Essa técnica é contrária à execução de uma única tarefa em processadores múltiplos. Usamos o termo **programa de processamento paralelo** para indicar um único programa que é executado em vários processadores simultaneamente.



PARALELISMO

paralelismo em nível de tarefa ou paralelismo em nível de processo

Utilizar vários processadores executando programas independentes simultaneamente.

programa de processamento paralelo

Um único programa que é executado em vários processadores simultaneamente.

Há muito tempo existem problemas científicos que precisam de computadores muito mais rápidos, e essa classe de problemas tem sido usada para justificar muitos computadores paralelos novos, no decorrer das últimas décadas. Hoje, alguns desses problemas podem ser tratados de forma simples, usando um **cluster** composto de microprocessadores abrigados em muitos servidores independentes ([Seção 6.7](#)). Além disso, os clusters podem servir a aplicações igualmente exigentes fora das ciências, como mecanismos de busca, servidores Web e bancos de dados.

cluster

Um conjunto de computadores conectados por uma rede local (LAN) que funciona como um único e grande multiprocessador.

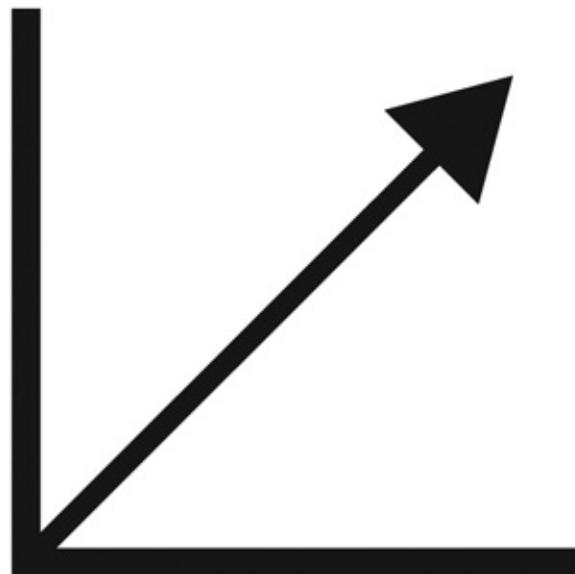
microprocessador multicore

Um microprocessador contendo vários processadores (“núcleos”) em um único circuito integrado. Praticamente todos os microprocessadores hoje nos computadores desktop e servidores são multicore.

Como dissemos no [Capítulo 1](#), os multiprocessadores ganharam destaque porque o problema da potência significa que aumentos futuros no desempenho aparentemente virão de mais processadores por chip, em vez de taxas de clock mais altas e CPI melhorado. Como vimos naquele capítulo, eles são chamados **microprocessadores multicore** e não microprocessadores multiprocessadores, provavelmente para evitar redundância de nomeação. Logo, os processadores normalmente são chamados núcleos (*cores*) em um chip multicore. Espera-se que o número de núcleos aumente conforme a [Lei de Moore](#). Esses multicores quase sempre são **processadores de memória compartilhada (SMPs — Shared Memory Processors)**, pois normalmente compartilham um único espaço de endereços físico. Veremos os SMPs com mais detalhes na [Seção 6.5](#).

processador de memória compartilhada (SMP)

Um processador paralelo com um único espaço de endereços.



LEI DE MOORE

O estado da tecnologia hoje significa que os programadores que se preocupam com o desempenho precisam se tornar programadores paralelos, pois programas sequenciais implicam em programas lentos.

O grande desafio enfrentado pela indústria é criar hardware e software que facilite a escrita de programas de processamento paralelo, que sejam eficientes no desempenho e potência à medida que o número de núcleos por chip aumenta geometricamente.

Essa mudança repentina no projeto do microprocessador apanhou muitos de surpresa, de modo que ainda existe muita confusão sobre a terminologia e o que ela significa. A [Figura 6.1](#) tenta esclarecer os termos serial, paralelo, sequencial e concorrente. As colunas dessa figura representam o software, que é inherentemente sequencial ou concorrente. As linhas da figura representam o hardware, que é serial ou paralelo. Por exemplo, os programadores de compiladores pensam neles como programas sequenciais: as etapas são análise léxica, geração de código, otimização e assim por diante. Ao contrário, os programadores de sistemas operacionais normalmente pensam neles como programas concorrentes: processos em cooperação tratando de eventos de E/S devido as tarefas independentes executando em um computador.

		Software	
		Sequencial	Concorrente
Hardware	Serial	Multiplicação de matriz escrita em MatLab executando em um Intel Pentium 4	Sistema operacional Windows Vista executando em um Intel Pentium 4
	Parallel	Multiplicação de matriz escrita em MatLab executando em um Intel Core 7	Sistema operacional Windows Vista executando em um Intel Core 7

FIGURA 6.1 Categorização e exemplos de hardware/software do ponto de vista da concorrência e do paralelismo.

O motivo desses dois eixos da [Figura 6.1](#) é que o software concorrente pode ser executado no hardware serial, como os sistemas operacionais para o uniprocessador Intel Pentium 4, ou no hardware paralelo, como um OS no mais recente Intel Core i7. O mesmo acontece para o software sequencial. Por exemplo, o programador MatLab escreve uma multiplicação de matriz pensando nela sequencialmente, mas poderia executá-la serialmente no hardware do Pentium 4 ou em paralelo no hardware do Core i7.

Você poderia supor que o único desafio da revolução paralela é descobrir como fazer com que o software naturalmente sequencial tenha alto desempenho no hardware paralelo, mas também fazer com que os programas concorrentes tenham alto desempenho nos multiprocessadores, à medida que o número de processadores aumenta. Com essa distinção, no restante deste capítulo usaremos *programa de processamento paralelo* ou *software paralelo* para indicar o software sequencial ou concorrente executando em hardware paralelo. A próxima seção descreve por que é difícil criar programas eficientes para processamento paralelo.

Antes de prosseguirmos ainda mais até o paralelismo, não se esqueça das nossas incursões iniciais dos capítulos anteriores:

- [Capítulo 2, Seção 2.11](#): Paralelismo e instruções: Sincronização
- [Capítulo 3, Seção 3.6](#): Paralelismo e aritmética computacional: paralelismo subword
- [Capítulo 4, Seção 4.10](#): Paralelismo em nível de instrução
- [Capítulo 5, Seção 5.10](#): Paralelismo e hierarquias de memória: coerência de cache

Verifique você mesmo

Verdadeiro ou falso: para que se beneficie de um multiprocessador, uma

aplicação precisa ser concorrente.

6.2. A dificuldade de criar programas com processamento paralelo

A dificuldade com o paralelismo não está no hardware; é que muito poucos programas de aplicação importantes foram escritos para completar as tarefas mais cedo nos multiprocessadores. É difícil escrever software que usa processadores múltiplos para completar uma tarefa mais rápido, e o problema fica pior à medida que o número de processadores aumenta.

Mas por que isso acontece? Por que os programas de processamento paralelo devem ser tão mais difíceis de desenvolver do que os programas sequenciais?

A primeira razão é que você *precisa* obter um bom desempenho e eficiência do programa paralelo em um multiprocessador; caso contrário, você usaria um programa sequencial em um processador, já que a programação sequencial é mais simples. Na verdade, as técnicas de projeto de processadores, como execução superescalar e fora de ordem, tiram vantagem do paralelismo em nível de instrução ([Capítulo 4](#)), normalmente sem envolvimento do programador. Tais inovações reduzem a necessidade de reescrever programas para multiprocessadores, já que os programadores poderiam não fazer nada e ainda assim seus programas sequenciais seriam executados mais rapidamente nos novos computadores.

Por que é difícil escrever programas de multiprocessador que sejam rápidos, especialmente quando o número de processadores aumenta? No [Capítulo 1](#), usamos a analogia de oito repórteres tentando escrever um único artigo na esperança de realizar o trabalho oito vezes mais rápido. Para ter sucesso, a tarefa precisa ser dividida em oito partes de mesmo tamanho, pois senão alguns repórteres estariam ociosos enquanto esperam que aqueles com partes maiores terminem. Outro obstáculo ao desempenho seria que os repórteres gastariam muito tempo se comunicando entre si, em vez de escrever suas partes do artigo. Para essa analogia e para a programação paralela, os desafios incluem escalonamento, particionamento do trabalho em partes paralelas, balanceamento da carga de modo uniforme entre os trabalhadores, tempo para sincronização e overhead para a comunicação entre as partes. O desafio é ainda maior quando aumenta o número de repórteres para um artigo do jornal e quanto aumenta o número de processadores para a programação paralela. Nossa discussão no [Capítulo 1](#) revela outro obstáculo, conhecido como a Lei de Amdahl. Ela nos

lembra que mesmo as pequenas partes de um programa precisam estar em paralelo para que o programa faça bom proveito dos muitos núcleos.

Desafio do speed-up

Exemplo

Suponha que você queira alcançar um speed-up 90 vezes mais rápido com 100 processadores. Que fração da computação original pode ser sequencial?

Resposta

A Lei de Amdahl (Capítulo 1) diz que:

$$\text{Tempo de execução após melhoria} = \frac{\text{Tempo de execução afetado pela melhoria}}{\text{Quantidade de melhoria} + \text{Tempo de execução não afetado}}$$

Podemos reformular a lei de Amdahl em termos de speed-up *versus* o tempo de execução original:

$$\text{Speed-up} = \frac{\text{Tempo de execução antes}}{(\text{Tempo de execução antes} - \text{Tempo de execução afetado}) + \frac{\text{Tempo de execução afetado}}{\text{Quantidade de melhoria}}}$$

Essa fórmula normalmente é reescrita considerando-se que o tempo de execução anterior é 1 para alguma unidade de tempo, e o tempo de execução afetado pela melhoria é considerado a fração do tempo de execução original:

$$\text{Speed-up} = \frac{1}{(1 - \text{Fração de tempo afetada}) + \frac{\text{Fração de tempo afetada}}{\text{Quantidade de melhoria}}}$$

Substituindo a meta de speed-up por 90 e a quantidade de melhoria por 100 na fórmula anterior:

$$90 = \frac{1}{(1 - \text{Fração de tempo afetada}) + \frac{\text{Fração de tempo afetada}}{100}}$$

Então, simplificando a fórmula e resolvendo para a fração de tempo afetada:

$$90 \times (1 - 0,99 \times \text{Fração de tempo afetada}) = 1$$

$$90 - (90 \times 0,99 \times \text{Fração de tempo afetada}) = 1$$

$$90 - 1 = 90 \times 0,99 \times \text{Fração de tempo afetada}$$

$$\text{Fração de tempo afetada} = 89 / 89,1 = 0,999$$

Portanto, para obter um speed-up de 90 com 100 processadores, a porcentagem sequencial só poderá ser 0,1%.

Entretanto, existem aplicações com um substancial paralelismo, como veremos em seguida.

Desafio do speed-up: ainda maior

Exemplo

Suponha que você queira realizar duas somas: uma é a soma de 10 variáveis escalares e outra é uma soma matricial de um par de arrays bidimensionais, com dimensões 10×10 . Por enquanto, vamos considerar que apenas a soma matricial possa se tornar paralela; logo veremos como tornar as somas escalares paralelas. Que speed-up você obtém com 10 versus 40 processadores? Em seguida, calcule os speed-ups supondo que as matrizes crescem para 20 por 20.

Resposta

Se considerarmos que o desempenho é uma função do tempo para uma adição, t , então há 10 adições que não se beneficiam dos processadores paralelos e 100 adições que se beneficiam. Se o tempo para um único processador é $110t$, o tempo de execução para 10 processadores é

Tempo de execução após melhoria =

$$\frac{\text{Tempo de execução afetado pela melhoria}}{\text{Quantidade de melhoria}} + \text{Tempo de execução não afetado}$$

$$\text{Tempo de execução após melhoria} = \frac{100t}{10} + 10t = 20t$$

Então, o speed-up com 10 processadores é $110t/20t = 5,5$. O tempo de execução para 40 processadores é

$$\text{Tempo de execução após melhoria} = \frac{100t}{40} + 10t = 12,5t$$

de modo que o speed-up com 40 processadores é $110t/12,5t = 8,8$. Assim, para o tamanho deste problema, obtemos cerca de 55% do speed-up em potencial com 10 processadores, mas somente 22% com 40.

Veja o que acontece quando aumentamos a matriz. O programa sequencial agora utiliza $10t + 400t = 410t$. O tempo de execução para 10 processadores é

$$\text{Tempo de execução após melhoria} = \frac{400t}{10} + 10t = 50t$$

de modo que o speed-up com 10 processadores é $410t/50t = 8,2$. O tempo de execução para 40 processadores é

$$\text{Tempo de execução após melhoria} = \frac{400t}{40} + 10t = 20t$$

de modo que o speed-up com 40 processadores é $410t/20t = 20.5$. Assim, para esse grande problema, obtemos cerca de 82% do speed-up em potencial

com 10 processadores e 51% com 40.

Esses exemplos mostram que obter um bom speed-up em um multiprocessador enquanto se mantém o tamanho do problema fixo é mais difícil do que conseguir um bom speed-up aumentando o tamanho do problema. Isso nos permite apresentar dois termos que descrevem maneiras de expandir.

Expansão forte significa medir o speed-up enquanto se mantém o tamanho do problema fixo. **Expansão fraca** significa que o tamanho do problema cresce proporcionalmente com o aumento no número de processadores. Vamos supor que o tamanho do problema, M , seja o conjunto de trabalho na memória principal, e que temos P processadores. Então, a memória por processador para a expansão forte é aproximadamente M/P , e para a expansão fraca ela é aproximadamente M .

expansão forte

Speed-up alcançado em um multiprocessador sem aumentar o tamanho do problema.

expansão fraca

Speed-up alcançado em um multiprocessador enquanto se aumenta o tamanho do problema proporcionalmente ao aumento no número de processadores.

Observe que a **hierarquia de memória** pode interferir com a sabedoria convencional, de que a expansão fraca é mais fácil que a expansão forte. Por exemplo, se um conjunto de dados com expansão fraca não couber mais na cache de último nível de um microprocessador multicore, o desempenho resultante poderia ser muito pior do que o uso da expansão forte.



HIERARQUIA

Dependendo da aplicação, você pode argumentar em favor de qualquer uma dessas técnicas de expansão. Por exemplo, o benchmark de banco de dados débito-crédito TPC-C requer que você aumente o número de contas de cliente para conseguir um maior número de transações por minuto. O argumento é que não faz sentido pensar que determinada base de clientes de repente começará a usar caixas eletrônicos 100 vezes por dia só porque o banco adquiriu um computador mais rápido. Em vez disso, se você for demonstrar um sistema que pode funcionar 100 vezes o número de transações por minuto, deverá fazer uma experiência com 100 vezes a quantidade de clientes. Problemas maiores geralmente precisam de mais dados, o que é um argumento em favor da expansão fraca.

Este exemplo final mostra a importância do balanceamento de carga.

Desafio do speed-up: balanceamento de carga

Exemplo

Para conseguir o speed-up de 20,5 no problema maior, mostrado anteriormente, com 40 processadores, consideramos que a carga foi balanceada perfeitamente. Ou seja, cada um dos 40 processadores teve 2,5% do trabalho a realizar. Em vez disso, mostre o impacto sobre o speed-up se a carga de um processador for maior que todo o restante. Calcule com o dobro da carga (5%) e cinco vezes a carga (12,5%) para o processador com mais

trabalho. Qual é a utilização do restante dos processadores?

Resposta

Se um processador tem 5% da carga paralela, então ele precisa realizar $5\% \times 400$ ou 20 adições, e os outros 39 compartilharão as 380 restantes. Como eles estão operando simultaneamente, podemos simplesmente calcular o tempo de execução como um máximo

$$\text{Tempo de execução após melhoria} = \text{Max}\left(\frac{380t}{39}, \frac{20t}{1}\right) + 10t = 30t$$

O speed-up cai de 20,5 para $410t/30t = 14$. Os 39 processadores restantes são utilizados em menos da metade do tempo: enquanto aguardam $20t$ para que o processador com mais trabalho termine, eles só realizam uma computação por $380t/39 = 9,7t$.

Se um processador tem 12,5% da carga, ele precisa realizar 50 adições. A fórmula é

$$\text{Tempo de execução após melhoria} = \text{Max}\left(\frac{350t}{39}, \frac{50t}{1}\right) + 10t = 60t$$

O speed-up cai ainda mais para $410t/60t = 7$. O restante dos processadores é utilizado em menos de 20% do tempo ($9t/50t$). Esse exemplo demonstra o valor do balanceamento de carga, pois apenas um único processador com o dobro da carga dos outros reduz o speed-up em um terço, e cinco vezes a carga em um processador reduz o speed-up por quase um fator de três.

Agora que compreendemos melhor os objetivos e os desafios do processamento paralelo, apresentamos uma sinopse do restante do capítulo. A próxima seção (6.3) descreve um esquema de classificação muito mais antigo do que a [Figura 6.1](#). Além disso, ela descreve dois estilos de arquiteturas do conjunto de instruções, que dão suporte à execução de aplicações sequenciais em hardware paralelo, a saber, *SIMD* e *vetor*. A [Seção 6.4](#), depois, descreve o *multithreading*, um termo frequentemente confundido com multiprocessamento,

em parte porque conta com a concorrência semelhante nos programas. A [Seção 6.5](#) descreve a primeira das duas alternativas de uma característica fundamental do hardware paralelo: se todos os processadores nos sistemas contam com um único espaço de endereços físico ou não. Como já dissemos, as duas versões populares dessas alternativas são chamadas *multiprocessadores de memória compartilhada* (SMPs) e *clusters*, e essa seção explica a primeira. A [Seção 6.6](#) descreve um estilo de computador relativamente novo, da comunidade de hardware gráfico, chamado *unidade de processamento de gráficos* (GPU), que também considera um endereço físico único. A [Seção 6.7](#) descreve os clusters, um exemplo popular de um computador com espaços de endereços físicos. A [Seção 6.8](#) mostra as topologias comuns usadas para conectar muitos processadores, sejam nós de servidores e um cluster ou núcleos em um microprocessador. Em seguida, discutimos a dificuldade de localizar benchmarks paralelos, na [Seção 6.9](#). Essa seção também inclui um modelo de desempenho simples, porém esclarecedor, que ajuda no projeto de aplicações e também de arquiteturas. Usamos esse modelo, além de benchmarks paralelos, na [Seção 6.10](#), a fim de comparar um computador multicore com uma GPU. A [Seção 6.11](#) divulga a última e maior etapa em nossa jornada de aceleração da multiplicação matricial. Para matrizes que não cabem na cache, o processamento paralelo usa 16 cores para melhorar o desempenho por um fator de 14. Fechamos com as falárias e armadilhas e nossas conclusões para o paralelismo.

Na próxima seção, apresentamos os acrônimos que você provavelmente já viu para identificar diferentes tipos de computadores paralelos.

Verifique você mesmo

Verdadeiro ou falso: a expansão forte não está ligada à lei de Amdahl.

6.3. SISD, MIMD, SIMD, SPMD e vetor

Uma categorização do hardware paralelo proposta na década de 1960 ainda está em uso atualmente. Ela foi baseada no número de fluxos de instruções e no número de fluxos de dados. A [Figura 6.2](#) mostra as categorias. Portanto, um processador convencional tem um único fluxo de instruções e um único fluxo de dados, e um multiprocessador convencional possui fluxos de instruções e dados múltiplos. Essas duas categorias são abreviadas como **SISD** e **MIMD**, respectivamente.

		Fluxos de Dados	
		Único	Múltiplo
Fluxos de instrução	Único	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Múltiplo	MISD: não há exemplos atuais	MIMD: Intel Core i7

FIGURA 6.2 Categorização de hardware e exemplos baseados no número de fluxos de instruções e fluxos de dados: SISD, SIMD, MISD e MIMD.

SISD

ou Single Instruction stream, Single Data stream. Um processador único.

MIMD

ou Multiple Instruction streams, Multiple Data streams. Um multiprocessador.

Embora seja possível escrever programas separados que são executados em diferentes processadores em um computador MIMD e ainda trabalharem juntos para um objetivo grandioso e coordenado, os programadores normalmente escrevem um único programa que executa em todos os processadores de um computador **MIMD**, contando com instruções condicionais quando diferentes processadores deveriam executar diferentes seções de código. Esse estilo é chamado **Single Program Multiple Data (SPMD)**, mas é apenas o modo normal de programar um computador MIMD.

SPMD

Single Program, Multiple Data streams. O modelo de programação MIMD convencional, em que um único programa é executado em todos os processadores.

O mais próximo que podemos chegar de um processador com múltiplos fluxos de instruções e fluxo de dados único (MISD) poderia ser um “processador de fluxo”, que realizaria uma série de cálculos sobre um único fluxo de dados em um padrão em pipeline: analisar a entrada da rede, decodificar os dados, descompactá-los, procurar uma combinação e assim por diante. O inverso do

MISD faz muito mais sentido. Computadores SIMD operam sobre vetores de dados. Por exemplo, uma única instrução SIMD poderia somar 64 números enviando 64 fluxos de dados a 64 ALUs, para formar 64 somas dentro de um único ciclo de clock. As instruções paralelas de subword, que vimos nas Seções 3.6 e 3.7, são outro exemplo de SIMD; na verdade, a letra do meio do acrônimo SSE da Intel significa SIMD.

SIMD

ou Single Instruction stream, Multiple Data streams. A mesma instrução é aplicada a muitos fluxos de dados, assim como em um processador de vetor.

As virtudes do SIMD são que todas as unidades de execução paralelas são sincronizadas e todas elas respondem a uma única instrução que emana de um único *contador de programa* (PC). Do ponto de vista de um programador, isso é próximo do já conhecido SISD. Embora cada unidade esteja executando a mesma instrução, cada unidade de execução tem seus próprios registradores de endereço e, portanto, cada unidade pode ter diferentes endereços de dados. Assim, nos termos da [Figura 6.1](#), uma aplicação sequencial poderia ser compilada para executar em hardware serial organizado como um SISD ou em hardware paralelo que foi organizado como um SIMD.

A motivação original por trás do SIMD foi amortizar o custo da unidade de controle por dezenas de unidades de execução. Outra vantagem é o tamanho reduzido da memória do programa — SIMD só precisa de uma cópia do código que está sendo executado simultaneamente, enquanto os MIMDs com passagem de mensagem podem precisar de uma cópia em cada processador, e o MIMD com memória compartilhada precisará de múltiplas caches de instrução.

SIMD funciona melhor quando lida com arrays em loops `for`. Logo, para o paralelismo funcionar no SIMD, é preciso haver muitos dados estruturados de forma idêntica, o que é chamado de **paralelismo em nível de dados**. SIMD é mais fraco em instruções `case` ou `switch`, em que cada unidade de execução precisa realizar uma operação diferente sobre seus dados, dependendo de quais dados ela tenha. As unidades de execução com os dados errados devem ser desativadas, de modo que as unidades com dados corretos possam continuar. Se houver n casos, nessas situações, os processadores SIMD basicamente executam em $1/n$ do desempenho de pico.

parallelismo em nível de dados

Paralelismo obtido realizando-se a mesma operação sobre dados independentes.

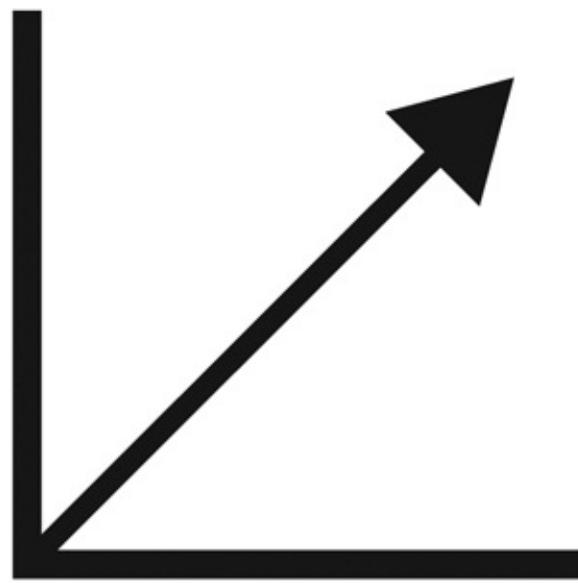
Os chamados processadores de array que inspiraram a categoria SIMD desapareceram na história, mas duas interpretações do SIMD permanecem ativas hoje.

SIMD no x86: extensões de multimídia

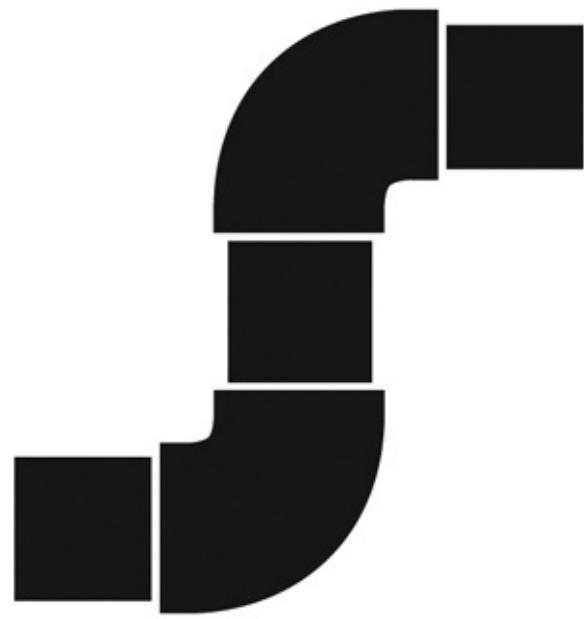
Conforme descrito no [Capítulo 3](#), o paralelismo de subword para dados inteiros estreitos foi a inspiração original das instruções MMX (Multimedia Extension) do x86, em 1996. À medida que a **Lei de Moore** continuava, mais instruções eram acrescentadas, levando primeiro às *Streaming SIMD Extensions* (SSE) e agora às *Advanced Vector Extensions* (AVX). AVX tem suporte para a execução simultânea de número de ponto flutuante de 64 bits. A largura da operação e dos registradores é codificada no opcode dessas instruções de multimídia. Enquanto a largura de dados de registradores e operações crescia, o número de opcodes para instruções de multimídia explodia, e agora existem centenas de instruções SSE e AVX ([Capítulo 3](#)).

Vetor

Uma interpretação mais antiga e mais elegante do SIMD é a chamada *arquitetura de vetor*, que possui uma identidade muito próxima dos computadores projetados por Seymour Cray, a partir da década de 1970. Esta, também é uma grande combinação de problemas com muito paralelismo em nível de dados. Em vez de ter 64 ALUs realizando 64 adições simultaneamente, como os antigos processadores de array, as arquiteturas de vetor colocaram a ALU em pipeline para obter bom desempenho com custo reduzido. A filosofia básica da arquitetura de vetor é coletar elementos de dados da memória, ordená-los em um grande conjunto de registradores, operar sobre eles sequencialmente nos registradores usando **unidades de execução em pipeline** e depois escrever os resultados de volta para a memória. Um recurso importante das arquiteturas de vetor é um conjunto de registradores de vetor. Assim, uma arquitetura de vetor poderia ter 32 registradores de vetor, cada um com 64 elementos de 64 bits.



LEI DE MOORE



PIPELINING

Comparando código de vetor com código

convencional

Exemplo

Suponha que estendamos a arquitetura do conjunto de instruções MIPS com instruções e registradores de vetor. As operações de vetor utilizam os mesmos nomes das operações MIPS, mas com a letra “V” acrescentada. Por exemplo, addv.d soma dois vetores de precisão dupla. As instruções de vetor recolhem como entrada um par de registradores de vetor (addv.d) ou um registrador de vetor e um registrador escalar (addvs.d). No segundo caso, o valor no registrador escalar é usado como a entrada para todas as operações — a operação addvs.d somará o conteúdo de um registrador escalar a cada elemento em um registrador de vetor. Os nomes lv e sv indicam load de vetor e store de vetor, e carregam ou armazenam um vetor inteiro de dados de precisão dupla. Um operando é o registrador de vetor a ser carregado ou armazenado; o outro operando, que é um registrador MIPS de uso geral, é o endereço inicial do vetor na memória. Dada essa descrição curta, mostre o código MIPS convencional *versus* o código MIPS de vetor para

$$Y = a \times X + Y$$

onde X e Y são vetores de 64 números de ponto flutuante com precisão dupla, inicialmente residentes na memória, e a é uma variável escalar de precisão dupla. (Esse exemplo é o chamado loop DAXPY, que forma o loop interno do benchmark Linpack; DAXPY significa Double precision a \times X Plus Y.) Suponha que os endereços iniciais de X e Y estejam em $\$s0$ e $\$s1$, respectivamente.

Resposta

Aqui está o código MIPS convencional para o DAXPY:

l.d	\$f0,a(\$sp)	:carrega escalar a
addiu	\$t0,\$s0,#512	:limite superior do que carregar
loop: l.d	\$f2,0(\$s0)	:carrega x(i)
	\$f2,\$f2,\$f0	: $a \times x(i)$
mul.d	\$f4,0(\$s1)	:carrega y(i)
l.d	\$f4,\$f4,\$f2	: $a \times x(i) + y(i)$
add.d	\$f4,0(\$s1)	:armazena em y(i)
s.d	\$s0,\$s0,#8	:incrementa índice a x
addiu	\$s1,\$s1,#8	:incrementa índice a y
subu	\$t1,\$t0,\$s0	:calcula limite
bne	\$t1,\$zero,loop	:verifica se terminou

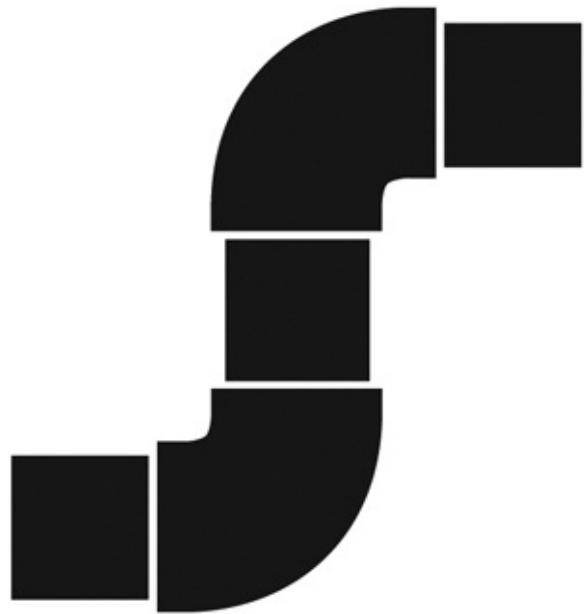
Aqui está o código MIPS de vetor para o DAXPY:

l.d	\$f0,a(\$sp)	:carrega escalar a
lv	\$v1,0(\$s0)	:carrega vetor x
mulvs.d	\$v2,\$v1,\$f0	:multiplica vetor escalar
lv	\$v3,0(\$s1)	:carrega vetor y
addv.d	\$v4,\$v2,\$v3	:soma y ao produto
sv	\$v4,0(\$s1)	:armazena o resultado

Existem algumas comparações interessantes entre os dois segmentos de código neste exemplo. A mais impressionante é que o processador de vetor reduz bastante a largura de banda de instrução dinâmica, executando apenas seis instruções contra quase 600 para a arquitetura MIPS tradicional. Essa redução ocorre tanto porque as operações de vetor trabalham sobre 64 elementos quanto porque as instruções de overhead que constituem quase metade do loop no MIPS não estão presentes no código de vetor. Como você poderia esperar, essa redução nas instruções buscadas e executadas economiza energia.

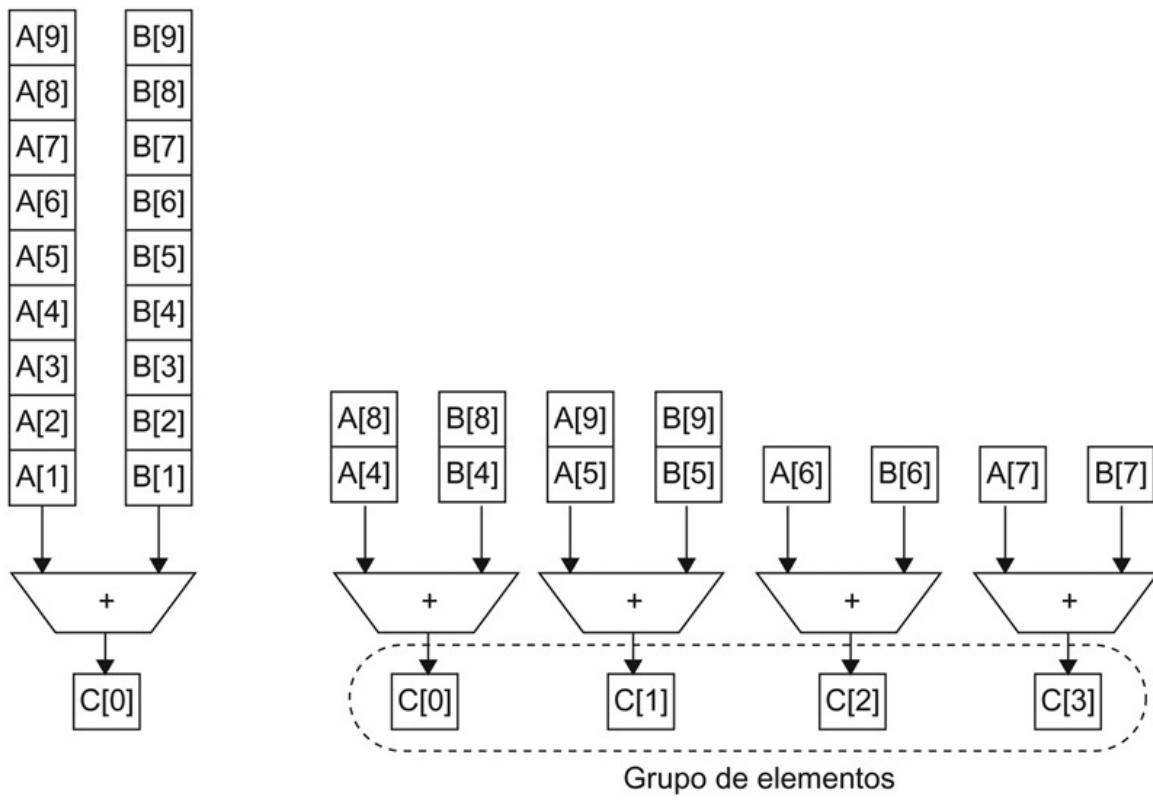
Outra diferença importante é a frequência dos hazards de **pipeline** ([Capítulo 4](#)). No código MIPS direto, cada add.d precisa esperar por um mul.d, cada s.d precisa esperar pelo add.d e cada add.d e mul.d precisa esperar pelo l.d. No processador de vetor, cada instrução de vetor só gerará stall para o primeiro elemento em cada vetor, depois os elementos subsequentes fluirão tranquilamente pelo pipeline. Assim, os stalls do pipeline só são necessários uma vez por *operação* de vetor, ao invés de uma vez por *elemento* de vetor. Neste exemplo, a frequência de stall do pipeline no MIPS será aproximadamente 64

vezes maior do que na versão de vetor do MIPS. Os stalls do pipeline podem ser reduzidos no MIPS usando desdobramento de loop ([Capítulo 4](#)). Porém, a grande diferença na largura de banda de instrução não pode ser reduzida.



PIPELINING

Como os elementos de vetor são independentes, eles podem ser operados em paralelo, semelhante ao paralelismo de subword para as instruções AVX. Todos os computadores de vetor modernos possuem unidades funcionais de vetor com múltiplos pipelines paralelos (chamadas *pistas de vetor*; veja as [Figuras 6.2 e 6.3](#)) que podem produzir dois ou mais resultados por ciclo de clock.



(a)

(b)

FIGURA 6.3 Usando múltiplas unidades funcionais para melhorar o desempenho de uma única instrução de soma vetorial, $C = A + B$.

O processador vetorial (a) à esquerda tem um único pipeline de adição por ciclo. O processador vetorial (b) à direita tem quatro pipelines ou pistas de adição e pode completar quatro adições por ciclo. Os elementos dentro de uma única instrução de soma vetorial são intercalados nas quatro pistas.

Detalhamento

O loop no exemplo anterior combinou exatamente com o tamanho do vetor. Quando os loops são mais curtos, as arquiteturas de vetor utilizam um registrador que reduz o tamanho das operações de vetor. Quando os loops são maiores, acrescentamos código de contabilidade para percorrer operações de vetor de tamanho total e tratar do restante. Esse último processo é conhecido como *strip mining* (ou *garimpagem*).

Vetor versus escalar

As instruções de vetor possuem várias propriedades importantes em comparação com as arquiteturas convencionais de conjunto de instruções, que são chamadas *arquiteturas escalares* nesse contexto:

- Uma única instrução de vetor especifica muito trabalho — isso é equivalente a executar um loop inteiro. A largura de banda de busca e decodificação de instrução necessária é bastante reduzida.
- Usando uma instrução de vetor, o compilador ou programador indica que o cálculo de cada resultado no vetor é independente do cálculo de outros resultados no mesmo vetor, de modo que o hardware não tem de verificar hazards de dados dentro de uma instrução de vetor.
- Arquiteturas e compiladores de vetor têm a reputação de tornar mais fácil do que os multiprocessadores MIMD para escrever aplicações eficientes quando elas contêm paralelismo em nível de dados.
- O hardware só precisa verificar hazards de dados entre duas instruções de vetor uma vez por operando de vetor, e não uma vez para cada elemento dentro dos vetores. A redução de verificações pode economizar energia, além de tempo.
- Instruções de vetor que acessam a memória possuem um padrão de acesso conhecido. Se os elementos do vetor forem todos adjacentes, então buscar o vetor de um conjunto de bancos de memória bastante intervalados funciona muito bem. Assim, o custo da latência para a memória principal é visto apenas uma vez para o vetor inteiro, e não uma vez para cada palavra do vetor.
- Como um loop inteiro é substituído por uma instrução de vetor cujo comportamento é predeterminado, os hazards de controle que, normalmente surgiriam do desvio do loop, são inexistentes.
- A economia na largura de banda de instrução e verificação de hazard mais o uso eficaz da largura de banda da memória dão às arquiteturas de vetor, vantagens em potência e energia contra as arquiteturas escalares.

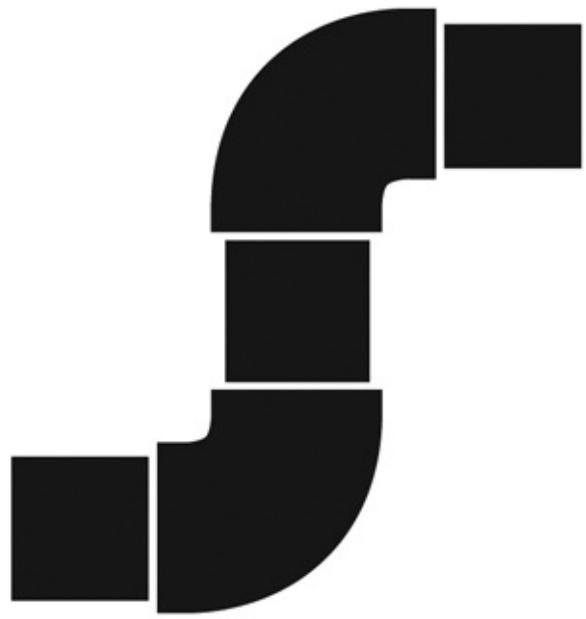
Por esses motivos, as operações de vetor podem se tornar mais rápidas que uma sequência de operações escalares sobre o mesmo número de itens de dados, e os projetistas são motivados a incluir unidades de vetor se o domínio da aplicação puder usá-las com frequência.

Vetor versus extensões de multimídia

Assim como as extensões de multimídia encontradas nas instruções AVX do x86, uma instrução de vetor especifica múltiplas operações. Porém, as extensões de multimídia normalmente especificam algumas poucas operações, enquanto a instrução de vetor especifica dezenas de operações. Diferente das extensões de multimídia, o número de elementos em uma operação de vetor não está no opcode, mas em um registrador separado. Isso significa que diferentes versões da arquitetura de vetor podem ser implementadas com um número diferente de elementos, apenas mudando o conteúdo desse registrador e retendo, portanto, a compatibilidade binária. Ao contrário, um novo grande conjunto de opcodes é acrescentado toda vez que o tamanho do “vetor” muda na arquitetura da extensão de multimídia do x86: MMX, SSE, SSE2, AVX, AVX2,

Também diferente das extensões de multimídia, as transferências de dados não precisam ser contíguas. Os vetores admitem os acessos “strided”, em que o hardware carrega cada n -ésimo elemento de dados na memória, e acessos indexados, em que o hardware encontra os endereços dos itens a serem carregados em um registrador de vetor. Os acessos indexados também são chamados de *gather-scatter*, pois os loads indexados ajuntam (*gather*) elementos da memória principal para elementos de vetor contíguos e os stores indexados espalham (*scatter*) elementos de vetor pela memória principal.

Assim como as extensões de multimídia, as arquiteturas de vetor facilmente capturam a flexibilidade nas larguras de dados, de modo que é fácil fazer uma operação de vetor funcionar em 32 elementos de dados de 64 bits ou em 64 elementos de dados de 32 bits ou em 128 elementos de dados de 16 bits ou em 256 elementos de dados de 8 bits. A semântica paralela de uma instrução de vetor permite que uma implementação execute essas operações usando uma unidade funcional profundamente em **pipeline**, um array de unidades funcionais paralelas ou uma combinação de unidades funcionais paralelas e em pipeline. A [Figura 6.3](#) ilustra como melhorar o desempenho de vetor usando pipelines paralelas para executar uma instrução de soma vetorial.



PIPELINING

As instruções de aritmética vetorial normalmente só permitem que o elemento N de um registrador de vetor tome parte das operações com o elemento N de outros registradores de vetor. Isso simplifica bastante a construção de uma unidade de vetor altamente paralela, que pode ser estruturada como múltiplas **pistas de vetor** paralelas. Como em uma rodovia de trânsito, podemos aumentar a vazão de pico de uma unidade de vetor acrescentando mais pistas. A [Figura 6.4](#) mostra a estrutura de uma unidade de vetor com quatro pistas. Assim, a mudança de uma para quatro pistas reduz o número de clocks por instrução de vetor por um fator aproximado de quatro. Para que as múltiplas pistas sejam vantajosas, tanto as aplicações quanto a arquitetura precisam ter suporte para vetores longos. Caso contrário, elas serão executadas tão rapidamente que você ficará sem instruções, exigindo técnicas **paralelas** em nível de instrução, como aquelas do [Capítulo 4](#), para fornecer instruções de vetor suficientes.

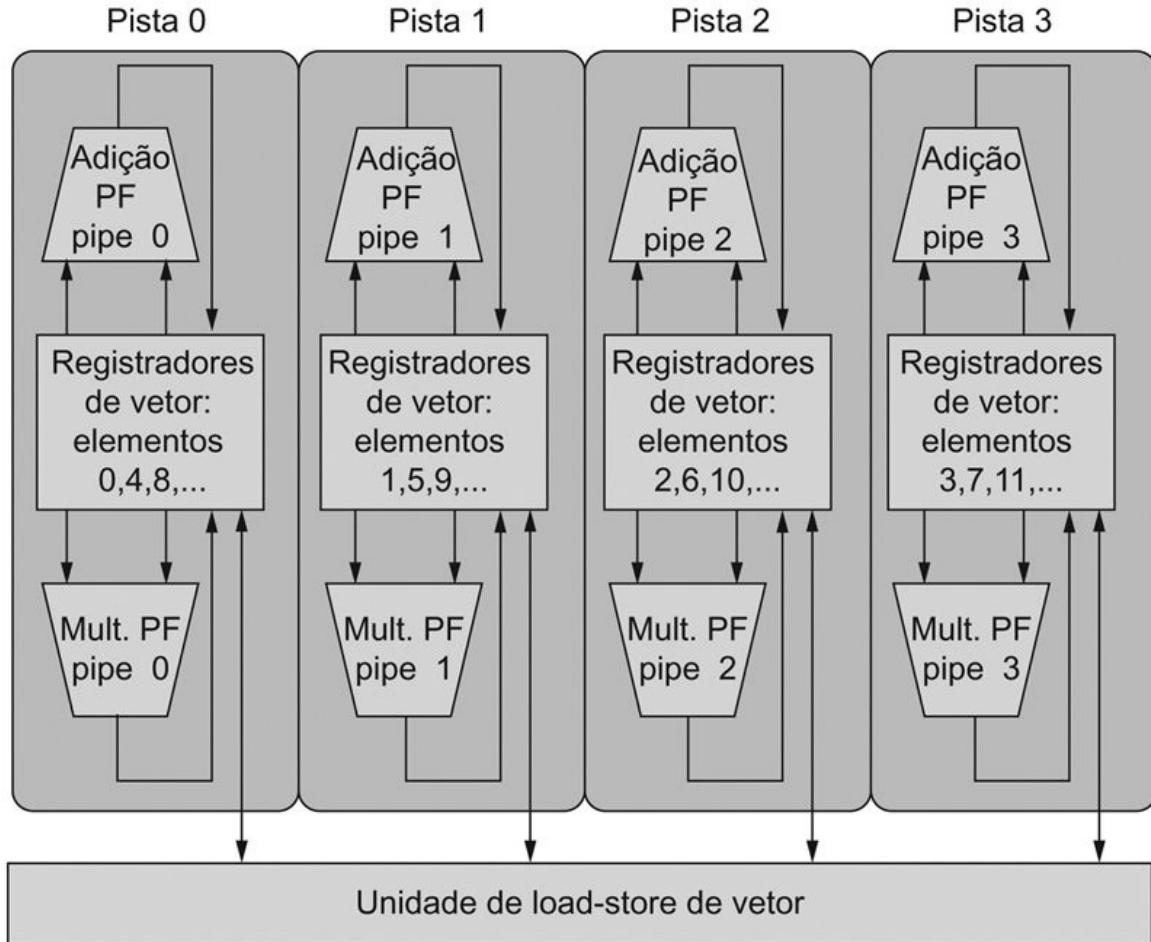


FIGURA 6.4 Estrutura de uma unidade de vetor contendo quatro pistas.

O armazenamento do registrador de vetor é dividido pelas pistas, com cada pista mantendo cada quarto elemento de cada registrador de vetor. A figura mostra três unidades funcionais de vetor: adição de PF, multiplicação de PF e uma unidade de load-store. Cada uma das unidades aritméticas de vetor contém quatro pipelines de execução, uma por pista, que atua em conjunto para completar uma única instrução de vetor. Observe como cada seção do arquivo de registrador de vetor precisa fornecer portas de leitura e escrita suficientes ([Capítulo 4](#)) para as unidades funcionais locais à sua pista.

pista de vetor

Uma ou mais unidades funcionais de vetor e uma parte do arquivo de registradores de vetor. Inspiradas nas pistas das rodovias, que aumentam a velocidade do tráfego, as pistas múltiplas executam operações de vetor

simultaneamente.



PARALELISMO

Geralmente, as arquiteturas de vetor são um meio muito eficaz de executar programas de processamento paralelo de dados; elas combinam melhor com a tecnologia de compilador do que extensões de multimídia; e são mais fáceis de evoluir com o tempo do que as extensões de multimídia na arquitetura x86.

Dadas essas categorias clássicas, em seguida examinamos como explorar os fluxos paralelos das instruções a fim de melhorar o desempenho de um *único* processador, que reutilizaremos com múltiplos processadores.

Verifique você mesmo

Verdadeiro ou falso: conforme exemplificado no x86, as extensões de multimídia podem ser consideradas como uma arquitetura de vetor com vetores curtos que suportam apenas transferências contíguas de dados de vetor.

Detalhamento

Dadas as vantagens do vetor, por que eles não são mais comuns fora da computação de alto desempenho? Havia preocupações sobre o estado maior

para registradores de vetor aumentando o tempo de troca de contexto e a dificuldade de tratar as falhas de página nos loads e stores de vetor, e as instruções SIMD conseguiram alguns dos benefícios das instruções de vetor. Além disso, enquanto os avanços no paralelismo em nível de instrução pudessem oferecer a promessa de desempenho da Lei de Moore, haveria pouco motivo para arriscar na mudança dos estilos de arquitetura.

Detalhamento

Outra vantagem das extensões de vetor e multimídia é que é relativamente fácil estender uma arquitetura de conjunto de instruções escalar com essas instruções para melhorar o desempenho das operações paralelas com dados.

Detalhamento

Os processadores x86 da geração Haswell da Intel têm suporte para AVX2, que possui uma operação “gather”, mas não uma operação “scatter”.

6.4. Multithreading do hardware

Um conceito relacionado ao MIMD, especialmente pelo ponto de vista do programador, é o **multithreading do hardware**. Enquanto MIMD conta com múltiplos **processos** ou **threads** para tentar manter os diversos processadores ocupados, o multithreading do hardware permite que múltiplas threads compartilhem as unidades funcionais de um único processador de um modo sobreposto, para tentar utilizar os recursos do hardware de modo eficaz. Para permitir esse compartilhamento, o processador precisa duplicar o estado independente de cada thread. Por exemplo, cada thread teria uma cópia separada do banco de registradores e do contador de programa (PC). A memória em si pode ser compartilhada por meio de mecanismos de memória virtual, que já suportam multiprogramação. Além disso, o hardware precisa suportar a capacidade de mudar para uma thread diferente com relativa rapidez. Em especial, uma troca de thread deve ser muito mais eficiente do que uma troca de processo, que normalmente exige centenas a milhares de ciclos de processador, enquanto uma troca de thread pode ser instantânea.

multithreading do hardware

Aumentar a utilização de um processador trocando para outra thread quando uma thread é suspensa.

thread

Uma thread inclui o contador de programa, o estado do registrador e a pilha. Ela é um processo simplificado; enquanto as threads normalmente compartilham um único espaço de endereços, o processo não faz isso.

processo

Um processo inclui uma ou mais threads, o espaço de endereços e o estado do sistema operacional. Logo, uma comutação de processos normalmente invoca o sistema operacional, mas uma comutação de threads não.

Existem dois métodos principais de multithreading do hardware. O **multithreading fine-grained** comuta entre threads a cada instrução, resultando em execução intercalada de várias threads. Essa intercalação normalmente é feita de forma circular, saltando quaisquer threads que estejam suspensas nesse ciclo de clock. Para tornar o multithreading fine-grained prático, o processador precisa ser capaz de trocar threads a cada ciclo de clock. Uma importante vantagem do multithreading fine-grained é que ele pode ocultar as perdas de vazão que surgem dos stalls curtos e longos, já que as instruções de outras threads podem ser executadas quando uma thread é suspensa. A principal desvantagem do multithreading fine-grained é que ele torna mais lenta a execução das threads individuais, já que uma thread que está pronta para ser executada sem stalls será atrasada por instruções de outras threads.

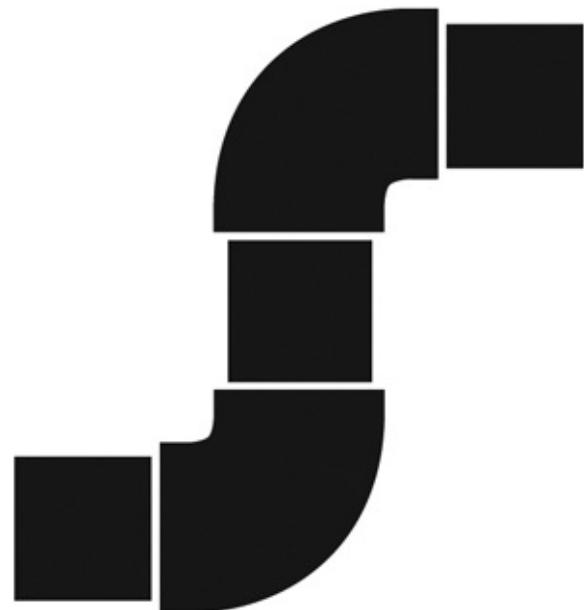
multithreading fine-grained

Uma versão do multithreading do hardware que sugere a comutação entre as threads após cada instrução.

multithreading coarse-grained

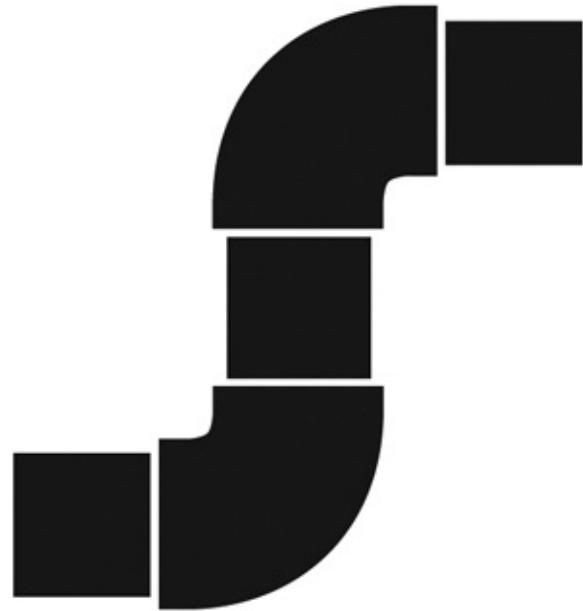
Uma versão do multithreading do hardware que sugere a comutação entre as threads somente após eventos significativos, como uma falha de cache de último nível.

O **multithreading coarse-grained** foi criado como uma alternativa para o multithreading fine-grained. Esse método de multithreading comuta threads apenas em stalls onerosos, como as falhas de cache de último nível. Essa mudança reduz a necessidade de tornar a comutação de thread essencialmente gratuita e tem muito menos chance de tornar mais lenta a execução de uma thread individual, visto que só serão despachadas instruções de outras threads quando uma thread encontrar um stall oneroso. Entretanto, o multithreading coarse-grained sofre de uma grande desvantagem: é limitado em sua capacidade de sanar perdas de vazão, especialmente de stalls mais curtos. Essa limitação surge dos custos de inicialização de **pipeline** do multithreading coarse-grained. Como um processador com multithreading coarse-grained despacha instruções por meio de uma única thread, quando ocorre um stall, o pipeline precisa ser esvaziado ou congelado. A nova thread que começa a ser executada após o stall precisa preencher o pipeline antes que as instruções consigam ser concluídas. Devido a esse overhead de inicialização, o multithreading coarse-grained é muito mais útil para reduzir a penalidade dos stalls de alto custo, em que a reposição de pipeline é insignificante comparada com o tempo de stall.



PIPELINING

O **simultaneous multithreading** (SMT) é uma variação do multithreading do hardware que usa os recursos de um processador em **pipeline**, de despacho múltiplo, escalonado dinamicamente, para explorar paralelismo em nível de thread ao mesmo tempo em que explora o paralelismo em nível de instrução ([Capítulo 4](#)). O princípio mais importante que motiva o SMT é que os processadores de despacho múltiplo normalmente possuem mais paralelismo de unidade funcional do que uma única thread efetivamente pode usar. Além disso, com a renomeação de registradores e o escalonamento dinâmico ([Capítulo 4](#)), diversas instruções de threads independentes podem ser despachadas sem considerar as dependências entre elas; a resolução das dependências pode ser tratada pela capacidade de escalonamento dinâmico.



PIPELINING

simultaneous multithreading (SMT)

Uma versão do multithreading que reduz o custo do multithreading, utilizando os recursos necessários para a microarquitetura de despacho múltiplo, escalonada dinamicamente.

Por contar com os mecanismos dinâmicos existentes, SMT não troca de recurso a cada ciclo, mas *sempre* está executando instruções de múltiplas threads, deixando para o hardware a associação de slots de instrução e registradores renomeados com suas threads apropriadas.

A [Figura 6.5](#) ilustra conceitualmente as diferenças na capacidade de um processador de explorar recursos superescalares para as configurações de processador a seguir. A parte superior mostra como quatro threads seriam executadas de forma independente em um superescalar sem suporte a multithreading. A parte inferior mostra como as quatro threads poderiam ser combinadas para serem executadas no processador de maneira mais eficiente usando três opções de multithreading:

- Um superescalar com multithreading coarse-grained.

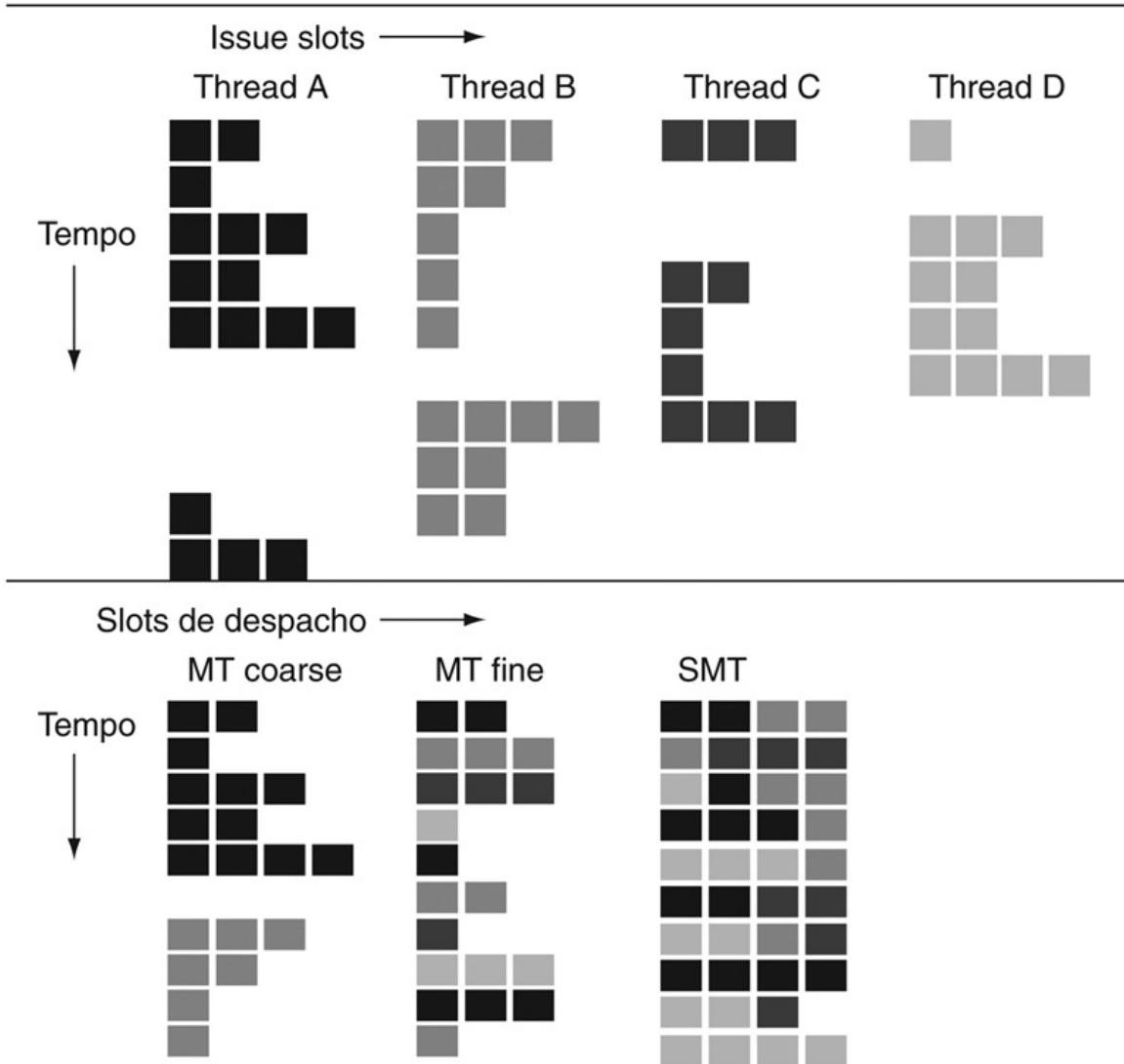


FIGURA 6.5 Como quatro threads usam os slots de despacho de um processador superescalar em diferentes métodos.

As quatro threads no topo mostram como cada uma seria executada em um processador superescalar padrão sem suporte a multithreading. Os três exemplos embaixo mostram como elas seriam executadas juntas em três opções de multithreading. A dimensão horizontal representa a capacidade de despacho de instrução em cada ciclo de clock. A dimensão vertical representa uma sequência dos ciclos de clock. Uma caixa vazia (branca) indica que o slot de despacho correspondente está vago nesse ciclo de clock. Os tons de cinza e preto correspondem a quatro threads diferentes nos processadores multithreading. Os efeitos de inicialização de pipeline adicionais para multithreading coarse, que não estão ilustrados nessa figura, levariam a mais perda na vazão para multithreading coarse.

- Um superescalar com multithreading fine-grained.
- Um superescalar com simultaneous multithreading.

No superescalar sem suporte a multithreading do hardware, o uso dos slots de despacho é limitado por uma falta de **paralelismo em nível de instrução**. Além disso, um importante stall, como uma falha de cache de instruções, pode deixar o processador inteiro ocioso.



PARALELISMO

No superescalar com multithreading coarse-grained, os longos stalls são parcialmente ocultados pela comutação para outra thread que usa os recursos do processador. Embora isso reduza o número de ciclos de clock completamente ociosos, o overhead de inicialização do pipeline ainda produz ciclos ociosos, e as limitações do paralelismo em nível de instrução significam que nem todos os slots de despacho serão utilizados. Em um processador com multithreading coarse-grained, a intercalação de threads elimina quase todos os ciclos de clock ociosos. Porém, como apenas uma thread despacha instruções em um determinado ciclo de clock, as limitações do paralelismo em nível de instrução ainda geram slots ociosos dentro de alguns ciclos de clock.

No caso SMT, o paralelismo em nível de thread e o paralelismo em nível de instrução são explorados simultaneamente, com múltiplas threads usando os slots de despacho em um único ciclo de clock. O ideal é que o uso de slots de

despacho seja limitado por desequilíbrios nas necessidades e na disponibilidade de recursos entre múltiplas threads. Na prática, outros fatores podem restringir o número de slots usados. Embora a [Figura 6.5](#) simplifique bastante a operação real desses processadores, ela ilustra as potenciais vantagens de desempenho em potencial do multithreading em geral e do SMT em particular.

A [Figura 6.6](#) representa graficamente os benefícios do multithreading em termos de desempenho e energia em um processador isolado do Intel Core i7 960, que possui suporte do hardware para duas threads. O speed-up médio é 1,31, que não é ruim, dados os modestos recursos extras para o multithreading do hardware. A melhoria média na eficiência de energia é 1,07, que é excelente. Em geral, você ficaria satisfeito com um speed-up neutro em termos de desempenho da energia.

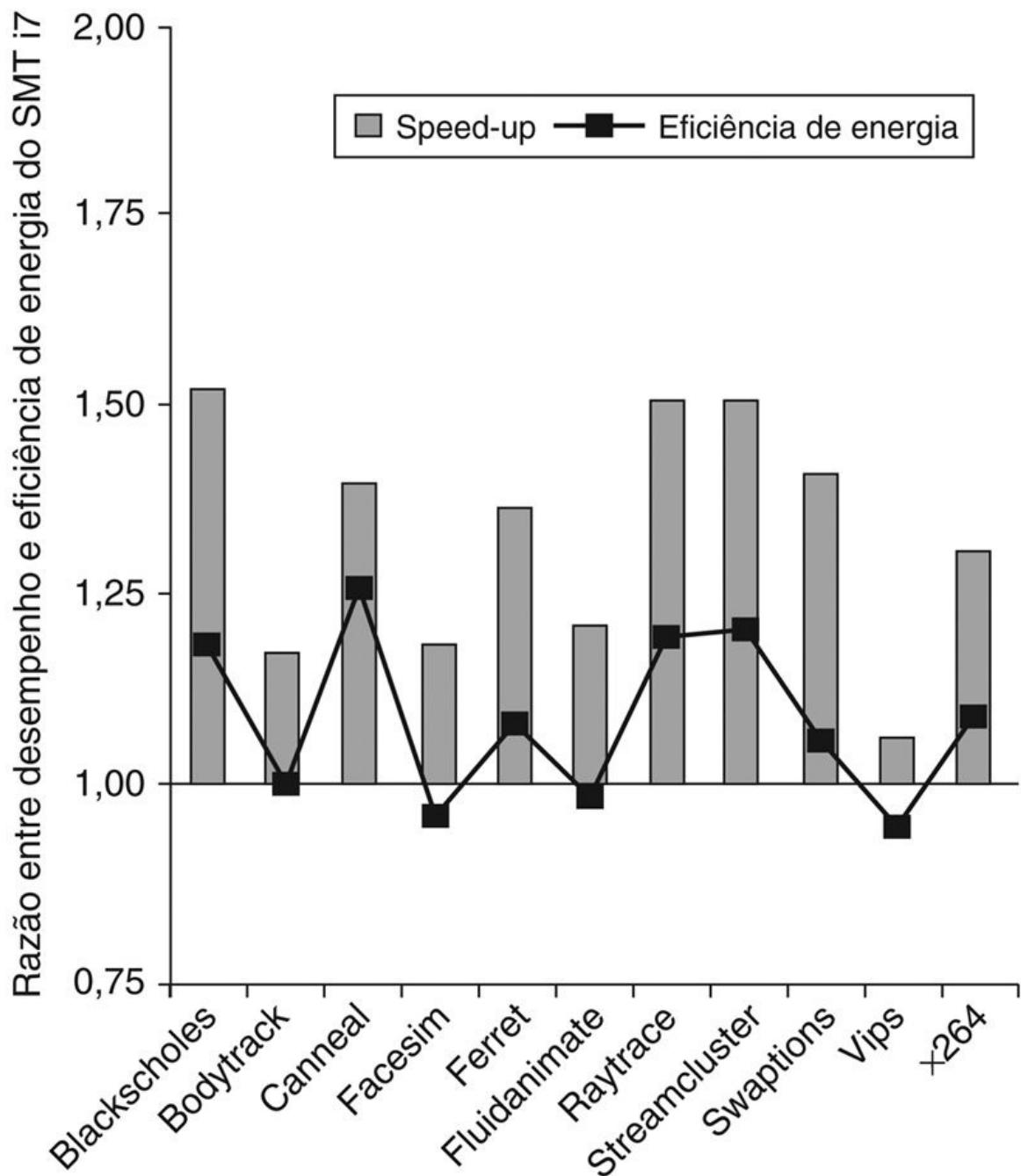


FIGURA 6.6 O speed-up pelo uso do multithreading sobre um core em um processador i7 é 1,31 em média para os benchmarks PARSEC e a melhoria em termos da eficiência de energia é 1,07.

Estes dados foram coletados e analisados por Esmaeilzadeh et. al. [2011].

Agora que já vimos como múltiplas threads podem utilizar os recursos de um único processador com mais eficiência, mostramos em seguida como usá-las

para explorar os processadores múltiplos.

Verifique você mesmo

1. Verdadeiro ou falso: tanto o multithreading quanto o multicore contam com o paralelismo para obter mais eficiência de um chip.
2. Verdadeiro ou falso: o *multithreading simultâneo* (SMT) utiliza threads para melhorar a utilização de recursos de um processador fora de ordem, escalonado dinamicamente.

6.5. Multicore e outros multiprocessadores de memória compartilhada

Embora o multithreading do hardware melhorasse a eficiência dos processadores a um custo razoável, o grande desafio da última década tem sido entregar o potencial de desempenho da Lei de Moore programando de modo eficiente o número cada vez maior de processadores por chip.

Dada a dificuldade de reescrever programas antigos para que funcionem bem em hardware paralelo, uma pergunta natural é o que os projetistas de computador podem fazer para simplificar a tarefa. Uma resposta para isso foi oferecer um único espaço de endereços físico que todos os processadores possam compartilhar, de modo que os programas não precisem se preocupar com o local onde são executados, apenas que podem ser executados em paralelo. Nessa técnica, todas as variáveis de um programa podem ficar disponíveis a qualquer momento para qualquer processador. A alternativa é ter um espaço de endereços separado por processador, o que requer que o compartilhamento seja explícito; vamos descrever essa opção na [Seção 6.7](#). Quando o espaço de endereços físico é comum, então o hardware normalmente oferece coerência de cache para dar uma visão consistente da memória compartilhada ([Seção 5.8](#)).

Como já dissemos, um *multiprocessador de memória compartilhada* (SMP) é aquele que oferece ao programador um *único espaço de endereços físico* para todos os processadores — que é o que quase sempre acontece para os chips multicore —, embora um termo mais preciso teria sido multiprocessador de *endereço compartilhado*. Os processadores se comunicam por meio de variáveis compartilhadas na memória, com todos os processadores capazes de acessar qualquer local da memória por meio de loads e stores. A [Figura 6.7](#) mostra a organização clássica de um SMP. Observe que esses sistemas ainda podem

executar tarefas independentes em seus próprios espaços de endereços virtuais, mesmo que todos compartilhem um espaço de endereços físico.

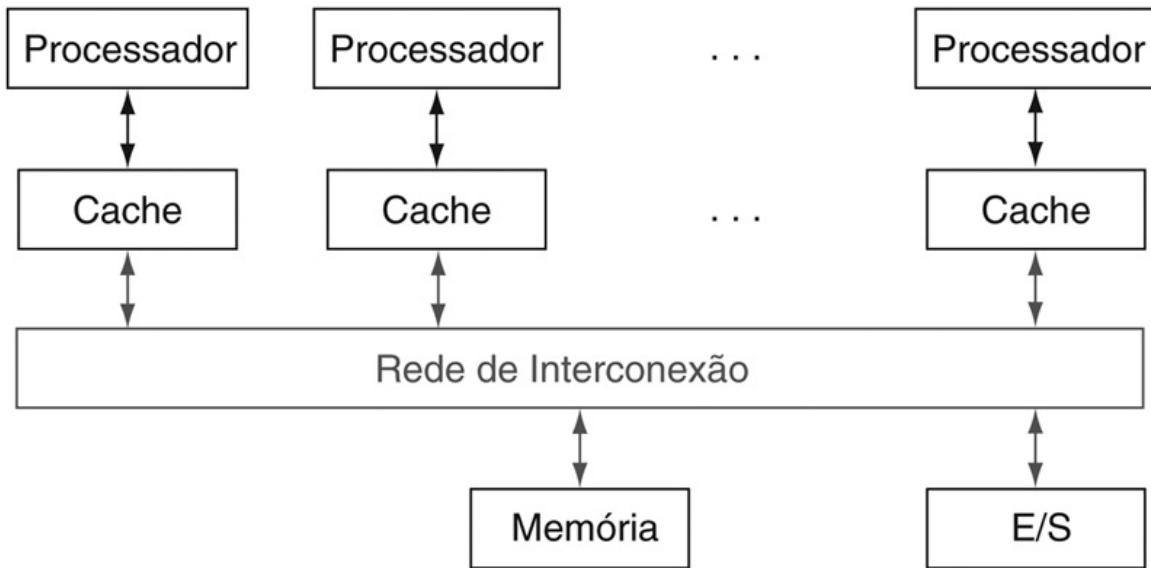


FIGURA 6.7 Organização clássica de um multiprocessador de memória compartilhada.

Microprocessadores com único espaço de endereços podem ser de dois estilos. No primeiro, a latência para uma palavra na memória não depende de qual processador o solicite. Essas máquinas são chamadas multiprocessadores de **acesso uniforme à memória (UMA)**. No segundo estilo, alguns acessos à memória são muito mais rápidos do que outros, dependendo de qual processador pede qual palavra, normalmente porque a memória principal é dividida e conectada a diferentes microprocessadores ou a diferentes controladores de memória no mesmo chip. Essas máquinas são chamadas multiprocessadores de **acesso não uniforme à memória (NUMA)**. Como você poderia esperar, os desafios de programação são mais difíceis para um multiprocessador NUMA do que para um multiprocessador UMA, mas as máquinas NUMA podem expandir para tamanhos maiores, e as NUMAs podem ter latência inferior para a memória próxima.

acesso uniforme à memória (UMA)

Um multiprocessador em que a latência a qualquer palavra na memória é aproximadamente a mesma, não importa qual processador solicita o acesso.

acesso não uniforme à memória (NUMA)

Um tipo de multiprocessador com espaço de endereços único em que alguns acessos à memória são muito mais rápidos do que outros, dependendo de qual processador solicita qual palavra.

Como os processadores operando em paralelo normalmente compartilharão dados, eles também precisam coordenar quando operarão sobre dados compartilhados; caso contrário, um processador poderia começar a trabalhar nos dados antes que outro tenha terminado. Essa coordenação é chamada de **sincronização**. Quando o compartilhamento tem o suporte de um único espaço de endereços, é preciso haver um mecanismo separado para sincronização. Uma técnica utiliza um **lock** para uma variável compartilhada. Somente um processador de cada vez pode adquirir o lock, e outros processadores interessados nos dados compartilhados precisam esperar até que o processador original libere a variável. A [Seção 2.11](#) do [Capítulo 2](#) descreve as instruções para o locking no conjunto de instruções MIPS.

sincronização

O processo de coordenar o comportamento de dois ou mais processos, que podem estar sendo executados em diferentes processadores.

lock

Um dispositivo de sincronização que permite o acesso aos dados somente por um processador de cada vez.

Um programa de processamento paralelo simples para um espaço de endereços compartilhado

Exemplo

Suponha que queremos somar 64.000 números em um computador com multiprocessador de memória compartilhada com tempo de acesso à memória uniforme. Vamos considerar que temos 64 processadores.

Resposta

Resposta

A primeira etapa é garantir uma carga balanceada por processador, de modo que dividimos o conjunto de números em subconjuntos do mesmo tamanho. Não alocamos os subconjuntos a um espaço de memória diferente, já que existe uma única memória para essa máquina; apenas atribuímos endereços iniciais diferentes a cada processador. P_n é o número que identifica o processador, entre 0 e 63. Todos os processadores começam o programa executando um loop que soma seu subconjunto de números:

```
sum[Pn] = 0;  
for (i = 1000*Pn; i < 1000*(Pn+1); i += 1)  
    sum[Pn] += A[i]; /*soma as áreas atribuídas */
```

(Observe que o código C $i += 1$ é apenas uma abreviação de $i = i + 1$.)

A próxima etapa é fazer essas 64 somas parciais. Essa etapa se chama **redução**, onde dividimos para conquistar. A metade dos processadores soma pares de somas parciais, depois, um quarto soma pares das novas somas parciais e assim por diante, até que tenhamos uma única soma final. A Figura 6.8 ilustra a natureza hierárquica dessa redução.

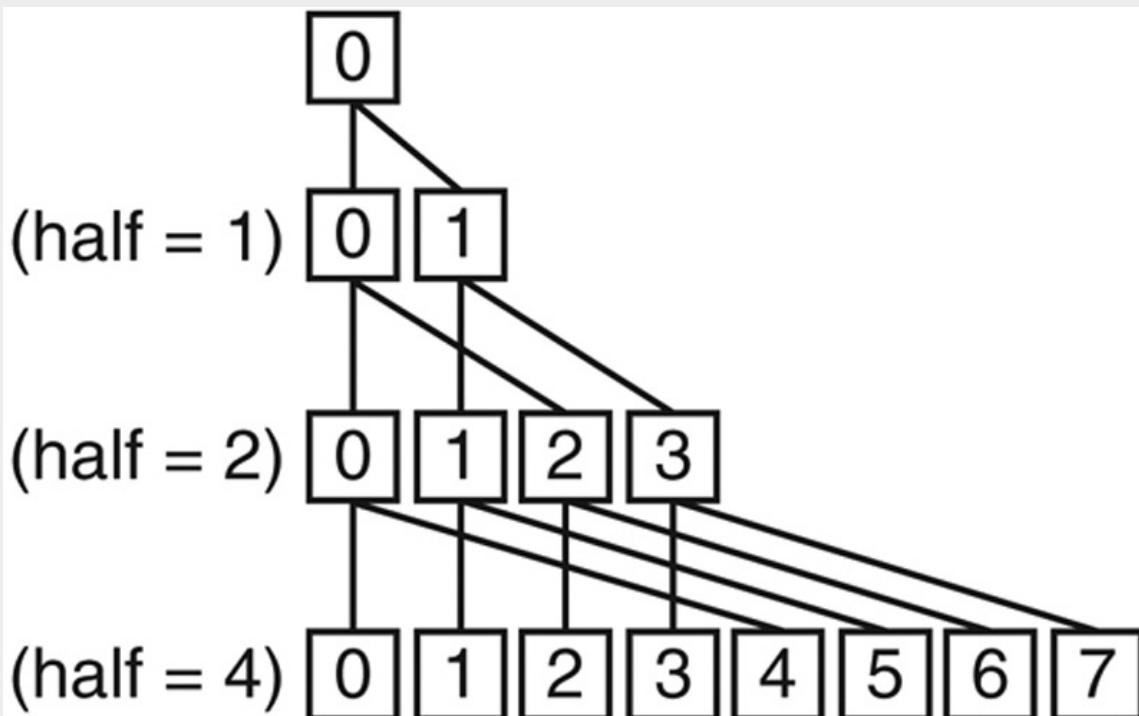


FIGURA 6.8 Os quatro últimos níveis de uma redução que soma os resultados de cada processador, de baixo para cima.

Para todos os processadores cujo número i é menor que half , adicione a soma produzida pelo processador número $(i + \text{half})$ à sua soma.

Neste exemplo, os dois processadores precisam ser sincronizados antes que o processador “consumidor” tente ler o resultado do local da memória escrito pelo processador “produtor”; caso contrário, o consumidor pode ler o valor antigo dos dados. Queremos que cada processador tenha sua própria versão da variável contadora de loop i , de modo que precisamos indicar que ela é uma variável “privada”. Aqui está o código (half também é privada):

```

half = 64; /*64 processadores no multiprocessador*/
do
    synch(); /*espera conclusão da soma parcial*/
    if (half%2 != 0 && Pn == 0)
        sum[0] += sum[half-1];
    /*soma condicional necessária quando half é
     ímpar; Processor0 obtém elemento ausente */
    half = half/2; /*linha divisora sobre quem soma */
    if (Pn < half) sum[Pn] += sum[Pn+half];
while (half > 1); /*sai com soma final em Sum[0] */

```

redução

Uma função que processa uma estrutura de dados e retorna um único valor.

Interface hardware/software

Dado o interesse duradouro pela programação paralela, tem havido centenas de tentativas de se criar sistemas de programação paralelos. Um exemplo limitado, porém popular, é o **OpenMP**. Esta é simplesmente uma *Interface de Programas de Aplicação* (API) juntamente com um conjunto de diretivas de compilador, variáveis de ambiente e rotinas de biblioteca de runtime que podem estender as linguagens de programação padrão. Ela oferece um modelo de programação portátil, expansível e simples para os multiprocessadores de memória compartilhada. Seu principal objetivo é gerar loops paralelos e realizar reduções.

A maioria dos compiladores C já tem suporte para OpenMP. O comando para usar a API OpenMP com o compilador C do UNIX é simplesmente:

```
cc -fopenmp algo.c
```

OpenMP estende a linguagem C usando *pragmas*, que são simplesmente comandos para o pré-processador de macros C, como `#define` e `#include`. Para definir o número de processadores que queremos usar como 64, como no exemplo anterior, simplesmente usamos o comando

```

#define P 64 /* define uma constante que usaremos algumas vezes */
#pragma omp parallel num_threads(P)

```

Ou seja, as bibliotecas de runtime deverão usar 64 threads paralelas.

Para transformar o loop for sequencial em um loop for paralelo que divide o trabalho igualmente entre todas as threads que dissemos que ele usaria, simplesmente escrevemos (considerando que `sum` seja inicializada em 0)

```
#pragma omp parallel for
for (Pn = 0; Pn < P; Pn += 1)
    for (i = 0; 1000*Pn; i < 1000*(Pn+1); i += 1)
        sum[Pn] += A[i]; /*somas as áreas atribuídas */
```

Para realizar a redução, podemos usar outro comando que diz ao OpenMP qual é o operador de redução e que variável você precisa usar para colocar o resultado da redução.

```
#pragma omp parallel for reduction(+: FinalSum)
for (i = 0; i < P; i += 1)
    FinalSum += sum[i]; /* reduz para um número final */
```

Observe que, agora, fica a cargo da biblioteca OpenMP encontrar o código para somar 64 números usando 64 processadores de modo eficiente.

Embora o OpenMP facilite a escrita de um código paralelo simples, ele não é muito útil com a depuração, de modo que muitos programadores de código paralelo utilizam sistemas de programação paralela mais sofisticados do que o OpenMP, assim como muitos programadores hoje utilizam linguagens mais produtivas do que C.

OpenMP

Uma API para multiprocessamento com memória compartilhada em C, C++ ou Fortran, que roda em plataformas UNIX e Microsoft. Ela inclui diretivas de compilador, uma biblioteca e diretivas de runtime.

Dado esse passeio do hardware e software MIMD clássico, nosso próximo caminho será um passeio mais exótico por um tipo de arquitetura MIMD com uma herança diferente e, portanto, com uma perspectiva muito diferente no

desafio da programação paralela.

Verifique você mesmo

Verdadeiro ou falso: multiprocessadores de memória compartilhada não podem tirar proveito do paralelismo em nível de tarefa.

Detalhamento

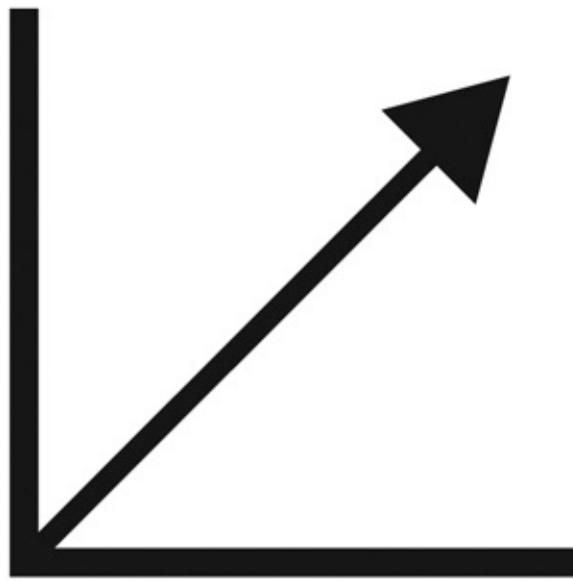
Alguns escritores modificaram o acrônimo SMP para significar *Symmetric Multiprocessor* (multiprocessador simétrico), indicando que a latência do processador até a memória era aproximadamente a mesma para todos os processadores. Essa mudança foi feita para contrastá-los com os processadores NUMA em grande escala, pois ambas as classes usavam um único espaço de endereços. Como os clusters provaram ser muito mais populares do que os multiprocessadores NUMA em grande escala, neste livro, restauramos SMP para o seu significado original e o usamos para contrastar com aquele que usa múltiplos espaços de endereços, como os clusters.

Detalhamento

Uma alternativa ao compartilhamento do espaço de endereços físico seria ter espaços de endereços físicos separados, mas compartilhar um espaço de endereços virtuais comum, deixando para o sistema operacional a tarefa de cuidar da comunicação. Essa técnica tem sido experimentada, mas possui um alto overhead para oferecer uma abstração de memória compartilhada prática ao programador orientado a desempenho.

6.6. Introdução às unidades de processamento de gráficos

A justificativa original para o acréscimo de instruções SIMD às arquiteturas existentes foi que muitos microprocessadores eram conectados a telas gráficas em PCs e estações de trabalho, de modo que uma fração cada vez maior do tempo de processamento era usada para os gráficos. Daí, quando a **Lei de Moore** aumentou o número de transistores disponíveis aos microprocessadores, fez sentido melhorar o processamento gráfico.



LEI DE MOORE

Uma força motriz importante para melhorar o processamento gráfico foi a indústria de jogos de computador, tanto em PCs quanto em consoles de jogos dedicados, como o Play- Station da Sony. O mercado de jogos em rápido crescimento encorajou muitas empresas a fazerem investimentos cada vez maiores no desenvolvimento de hardware gráfico mais rápido, e esse feedback positivo levou o processamento gráfico a melhorar em um ritmo mais rápido do que o processamento de uso geral nos principais microprocessadores.

Dado que a comunidade de gráficos e jogos teve objetivos diferentes da comunidade de desenvolvimento de microprocessador, ela evoluiu seu próprio estilo de processamento e terminologia. Quando os processadores gráficos aumentaram sua potência, eles ganharam o nome *Graphics Processing Units*, ou *GPUs*, para distingui-los das CPUs.

Por algumas centenas de dólares, qualquer um pode comprar uma GPU hoje, com centenas de unidades paralelas de ponto flutuante, tornando mais acessível a computação de alto desempenho. O interesse na computação GPU prosperou quando esse potencial foi combinado com uma linguagem de programação que facilitou a programação das GPUs. Logo, muitos programadores de aplicações científicas e de multimídia atualmente estão pensando se irão usar GPUs ou CPUs.

Aqui estão algumas das principais características de como as GPUs se distinguem das CPUs:

- GPUs são aceleradores que complementam uma CPU, de modo que não

precisam ser capazes de realizar todas as tarefas de uma CPU. Esse papel lhes permite dedicar todos os seus recursos aos gráficos. Não importa se as GPUs realizam algumas tarefas mal ou que não realizem, visto que, em um sistema com uma CPU e uma GPU, a CPU pode realizá-las se for preciso.

- Os tamanhos dos problemas que a GPU resolve normalmente são de centenas de megabytes a gigabytes, mas não centenas de gigabytes a terabytes.
- Estas diferenças levaram a diferentes estilos de arquitetura:
- Talvez a maior diferença seja que as GPUs não contam com caches multinível para contornar a longa latência para a memória, como nas CPUs. Em vez disso, as GPUs contam com o multithreading do hardware ([Seção 6.4](#)) para ocultar a latência para a memória. Ou seja, entre o momento de uma solicitação de memória e o momento em que os dados chegam, a GPU executa centenas ou milhares de threads que são independentes dessa solicitação.
- A memória da GPU é assim orientada para largura de banda, em vez de latência. Existem até mesmo chips de DRAM separados para GPUs que são mais largas e possuem largura de banda mais alta que os chips de DRAM para as CPUs. Além disso, as memórias da GPU tradicionalmente têm tido memória principal menor que os microprocessadores convencionais. Em 2013, as GPUs normalmente tinham de 4 a 6GiB ou menos, enquanto as CPUs tinham de 32 a 256GiB. Finalmente, lembre-se de que, para a computação de uso geral, você precisa incluir o tempo para transferir os dados entre a memória da CPU e a memória da GPU, pois a GPU é um coprocessador.
- Dada a confiança em muitos threads para oferecer boa largura de banda de memória, as GPUs podem acomodar muitos processadores paralelos (MIMD), além de muitas threads. Logo, cada processador de GPU é mais altamente multithreaded do que uma CPU típica, além de terem mais processadores.

Interface hardware/ software

Embora as GPUs fossem projetadas para um conjunto mais estreito de aplicações, alguns programadores questionaram se poderiam especificar suas aplicações em uma forma que lhes permitissem aproveitar o alto desempenho em potencial das GPUs. Depois de cansar de tentar especificar seus problemas usando as APIs e linguagens gráficas, eles desenvolveram linguagens de

programação inspiradas em C para permitir que escrevam programas diretamente às GPUs. Um exemplo é a CUDA (Compute Unified Device Architecture) da NVIDIA, que permite que o programador escreva programas em C para execução nas GPUs, embora com algumas restrições. OpenCL é uma iniciativa multiempresarial para desenvolver uma linguagem de programação portável, que oferece muitos dos benefícios da linguagem CUDA.

A NVIDIA decidiu que o tema unificador de todas essas formas de paralelismo é a *Thread CUDA*. Usando esse nível mais baixo de paralelismo como primitiva de programação, o compilador e o hardware podem reunir milhares de Threads CUDA para utilizar os diversos estilos de paralelismo dentro de uma GPU: multithreading, MIMD, SIMD e paralelismo em nível de instrução. Essas threads são colocadas em blocos e executadas em grupos de 32 de cada vez. Um processador multithreaded dentro de uma GPU executa esses blocos de threads, e uma GPU consiste de 8 a 32 desses processadores multithreaded.

Introdução à arquitetura de GPU NVIDIA

Usamos sistemas NVIDIA como nosso exemplo porque representam bem as arquiteturas de GPU. Especificamente, seguimos a terminologia da linguagem de programação paralela CUDA e usamos a arquitetura Fermi como exemplo.

Assim como as arquiteturas de vetor, as GPUs funcionam bem somente com problemas paralelos em nível de dados. Os dois estilos possuem transferências de dados *gather-scatter*, e os processadores GPU possuem ainda mais registradores do que os processadores de vetor. Ao contrário da maioria das arquiteturas de vetor, as GPUs também contam com o multithreading do hardware dentro de um único processador SIMD multithreaded para ocultar a latência da memória ([Seção 6.4](#)).

Um processador SIMD multithreaded é semelhante a um processador de vetor, mas o primeiro possui muitas unidades funcionais paralelas, em vez de somente algumas com pipelines profundas, como o segundo.

Como já dissemos, uma GPU contém uma coleção de processadores SIMD multithreaded; ou seja, uma GPU é um MIMD composto de processadores SIMD multithreaded. Por exemplo, NVIDIA possui quatro implementações da arquitetura Fermi com diferentes preços, com 7, 11, 14 ou 15 processadores SIMD multithreaded. Para fornecer escalabilidade transparente pelos modelos de

GPUs com diferentes números de processadores SIMD multithreaded, o hardware Thread Block Scheduler atribui blocos de threads aos processadores SIMD multithreaded. A [Figura 6.9](#) mostra um diagrama de blocos simplificado de um processador SIMD multithreaded.

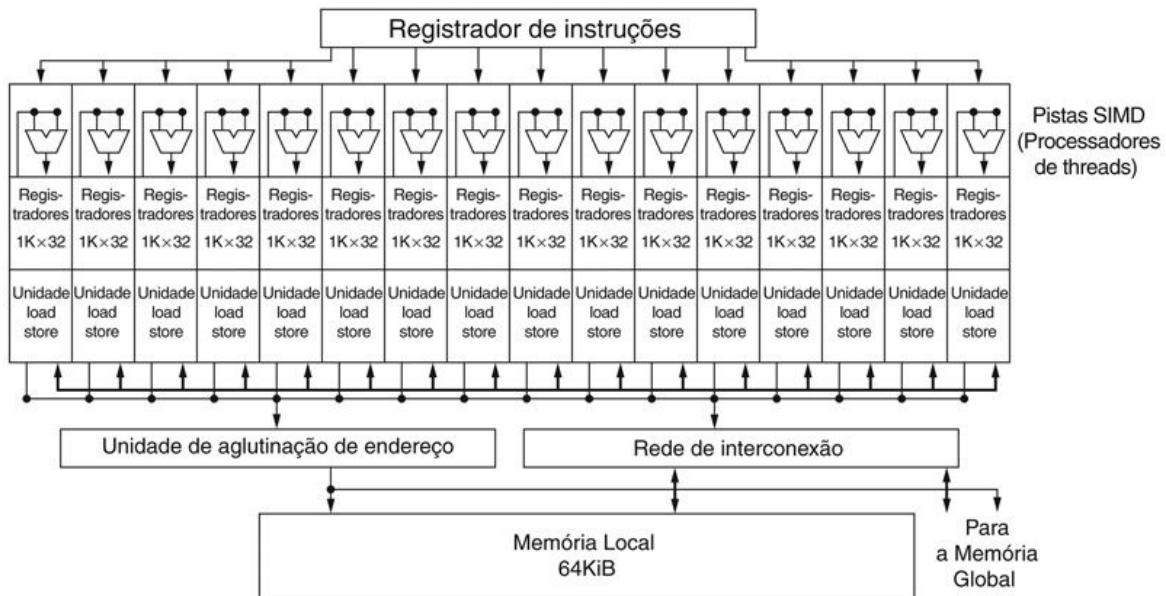


FIGURA 6.9 Diagrama de blocos simplificado do caminho de dados de um processador SIMD multithreaded.

Ele possui 16 pistas SIMD. O SIMD Thread Scheduler possui muitas threads SIMD independentes, das quais ele escolhe para executar nesse processador.

Descendo mais um nível de detalhe, o objeto de máquina que o hardware cria, gerencia, escalona e executa é uma *thread de instruções SIMD*, que também chamaremos de *thread SIMD*. Essa é uma thread tradicional, mas contém instruções SIMD exclusivamente. Essas threads SIMD possuem seus próprios contadores de programa e são executadas em um processador SIMD multithreaded. O *SIMD Thread Scheduler* inclui um controlador que lhe permite saber quais threads de instruções SIMD estão prontas para ser executadas e, depois, as envia para uma unidade de despacho, para serem executadas no processador SIMD multithreaded. Ele é idêntico a um escalonador de threads de hardware de um processador multithreaded tradicional ([Seção 6.4](#)), exceto que está escalonando threads de instruções SIMD. Assim, o hardware de GPU tem dois níveis de escalonadores de hardware:

1. O *Thread Block Scheduler* que atribui blocos de threads aos processadores

SIMD multithreaded, e

2. o SIMD Thread Scheduler *dentro* de um processador SIMD, que escalona quando as threads SIMD deverão ser executadas.

As instruções SIMD dessas threads possuem largura 32, de modo que cada thread de instruções SIMD calcularia 32 dos elementos do cálculo. Como a thread consiste em instruções SIMD, o processador SIMD precisa ter unidades funcionais paralelas para realizar a operação. Nós as chamamos de *Pistas SIMD*, e são muito semelhantes às pistas de vetor que vimos na [Seção 6.3](#).

Detalhamento

O número de pistas por processador SIMD varia no decorrer das gerações de GPU. Com Fermi, cada thread de instruções SIMD com largura 32 é mapeada para 16 pistas SIMD, de modo que cada instrução SIMD em uma thread de instruções SIMD usa dois ciclos de clock para ser concluída. Cada thread de instruções SIMD é executada em lock step. Continuando com a analogia de um processador SIMD como um processador de vetor, você poderia dizer que ele tem 16 pistas, e o comprimento do vetor seria 32. Essa natureza larga, porém superficial, é o motivo para usarmos o termo processador SIMD em vez de processador de vetor, pois é mais intuitivo.

Como, por definição, as threads de instruções SIMD são independentes, o SIMD Thread Scheduler pode escolher qualquer thread de instruções SIMD que estiver pronta, e não precisa ficar preso à próxima instrução SIMD na sequência dentro de uma única thread. Logo, usando a terminologia da Seção 6.4, ele utiliza o multithreading fine-grained.

Para segurar esses elementos da memória, um processador SIMD Fermi possui impressionantes 32.768 registradores de 32 bits. Assim como um processador de vetor, esses registradores são divididos logicamente pelas pistas de vetor ou, neste caso, pistas SIMD. Cada thread SIMD é limitada a não mais do que 64 registradores, de modo que você poderia pensar em uma thread SIMD como tendo até 64 registradores de vetor, com cada registrador de vetor tendo 32 elementos e cada elemento tendo 32 bits de largura.

Como Fermi possui 16 pistas SIMD, cada uma contém 2048 registradores. Cada thread CUDA recebe um elemento de cada um dos registradores de vetor. Observe que uma thread CUDA é simplesmente um corte vertical de uma thread de instruções SIMD, correspondendo a um elemento executado por uma pista SIMD. Saiba que as threads CUDA são muito diferentes das

threads POSIX; não é possível fazer chamadas de sistema arbitrárias ou sincronizar arbitrariamente em uma thread CUDA.

Estruturas de memória da GPU NVIDIA

A Figura 6.10 mostra as estruturas de memória de uma GPU NVIDIA. Chamamos a memória no chip, local a cada processador SIMD multithreaded, de *memória local*. Ela é compartilhada pelas pistas SIMD dentro de um processador SIMD multithreaded, mas essa memória não é compartilhada entre os processadores SIMD multithreaded. Chamamos a DRAM fora do chip, compartilhada pela GPU inteira e por todos os blocos de threads, de *memória da GPU*.

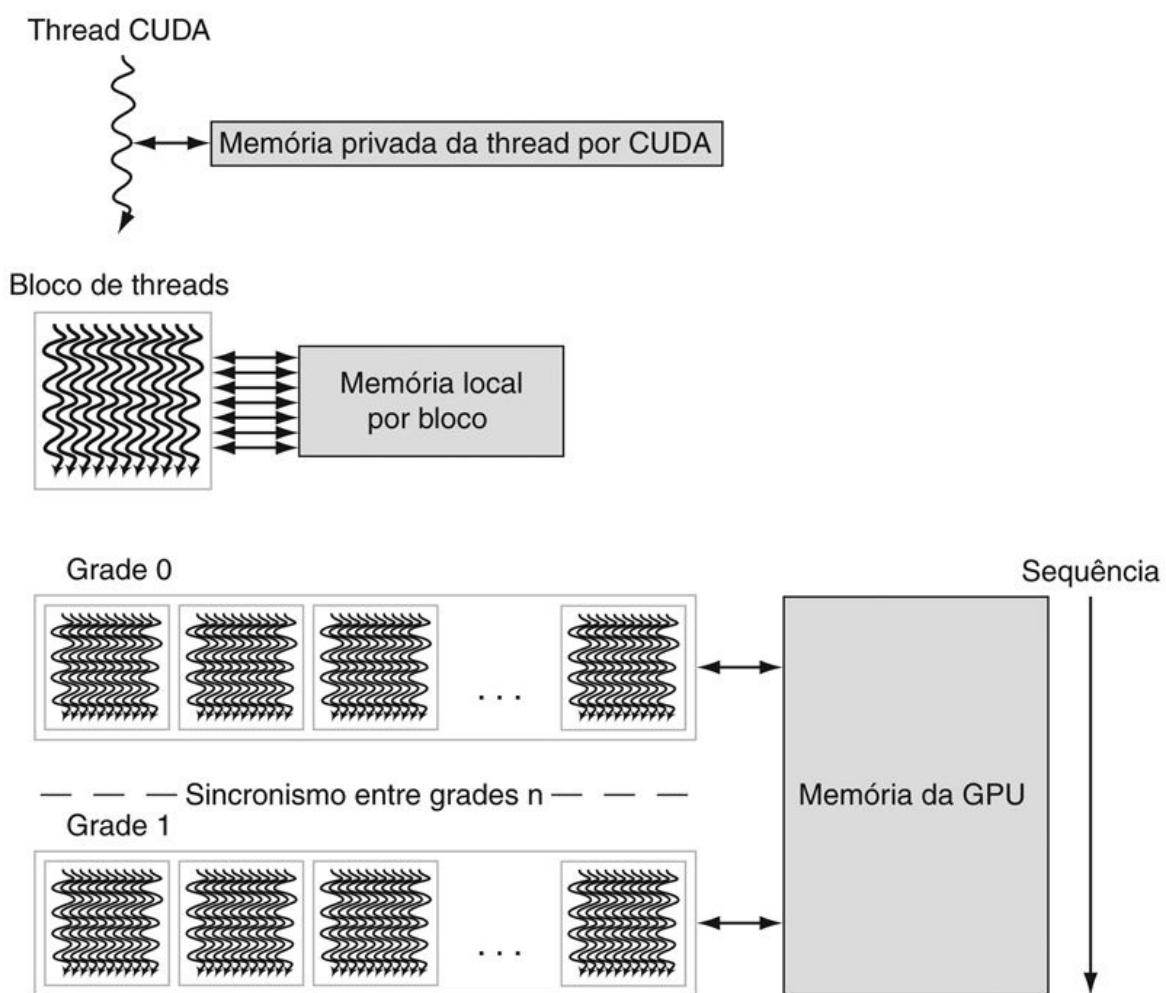


FIGURA 6.10 Estruturas de memória da GPU.

A memória da GPU é compartilhada pelos loops vetorizados.

Todas as threads de instruções SIMD dentro de um bloco de threads compartilham a memória local.

Em vez de contar com caches grandes para conter os conjuntos de trabalho inteiros de uma aplicação, as GPUs tradicionalmente utilizam caches streaming menores, e contam com um extenso multithreading de instruções SIMD para ocultar a longa latência até a DRAM, pois esses conjuntos de trabalho podem ter centenas de megabytes. Logo, eles não caberão na cache de último nível de um microprocessador multicore. Dado o uso do multithreading de hardware para ocultar a latência da DRAM, a área do chip usada para caches nos processadores do sistema é gasta no lugar dos recursos de computação e no grande número de registradores para manter o estado de muitas threads das instruções SIMD.

Detalhamento

Embora ocultar a latência da memória seja a filosofia básica, observe que as GPUs e os processadores de vetor mais recentes adicionaram caches. Por exemplo, a recente arquitetura Fermi adicionou caches, mas eles são considerados como filtros de largura de banda para reduzir as demandas sobre a memória da GPU ou como aceleradores para as poucas variáveis cuja latência não pode ser ocultada pelo multithreading. A memória local para quadros de pilha, chamadas de função e derramamento de registradores é uma boa utilização das caches, pois a latência é importante quando se chama uma função. As caches também economizam energia, pois os acessos à cache no chip gastam muito menos energia do que os acessos a vários chips de DRAM externos.

Colocando as GPUs em perspectiva

Em um nível alto, os computadores multicore com extensões de instrução SIMD compartilham semelhanças com as GPUs. A [Figura 6.11](#) resume as semelhanças e diferenças. Ambos são MIMDs cujos processadores utilizam várias pistas SIMD, embora as GPUs tenham mais processadores e muito mais pistas. Ambos utilizam o multithreading do hardware para melhorar a utilização do processador, embora as GPUs tenham suporte do hardware para muito mais threads. Ambos usam caches, embora as GPUs usem caches de streaming menores e os computadores multicore usem grandes caches multinível para tentar conter totalmente os conjuntos de trabalho inteiros. Ambos utilizam um

espaço de endereços de 64 bits, embora a memória principal física seja muito menor nas GPUs. Embora as GPUs suportem a proteção de memória no nível de página, elas ainda não suportam a paginação por demanda.

Recurso	Multicore with SIMD	GPU
Processadores SIMD	4 a 8	8 a 16
Pistas SIMD/processador	2 a 4	8 a 16
Suporte a hardware multithreading para threads SIMD	2 a 4	16 a 32
Maior tamanho de cache	8 MiB	0,75 MiB
Tamanho do endereço de memória	64-bit	64-bit
Tamanho da memória principal	8 GiB a 256 GiB	4 GiB a 6 GiB
Proteção de memória em nível de página	Sim	Sim
Paginação por demanda	Sim	Não
Coerência de cache	Sim	Não

FIGURA 6.11 Semelhanças e diferenças entre multicore com extensões de multimídia SIMD e GPUs recentes.

Processadores SIMD também são semelhantes aos processadores de vetor. Os múltiplos processadores nas GPUs atuam como núcleos MIMD independentes, assim como muitos computadores de vetor possuem múltiplos processadores de vetor. Essa visão consideraria o Fermi GTX 580 como uma máquina de 16 núcleos com suporte do hardware para multithreading, onde cada núcleo possui 16 pistas. A maior diferença é o multithreading, que é fundamental para GPUs e não existe na maioria dos processadores de vetor.

GPUs e CPUs não retornam, na genealogia da arquitetura de computação, para um ancestral comum; não existe um Elo Perdido que explica ambos. Como resultado dessa herança incomum, GPUs não têm usado os termos comuns na comunidade da arquitetura de computação, o que causou confusão sobre o que são GPUs e como elas funcionam. Para ajudar a acabar com a confusão, a Figura 6.12 (da esquerda para a direita) lista o termo mais descritivo usado nesta seção, o termo mais próximo da computação tradicional, o termo GPU oficial da NVIDIA, caso você esteja interessado, e depois uma pequena descrição do termo. Esta “Pedra de Roseta das GPU” poderá ajudar a relacionar esta seção e suas ideias às descrições mais convencionais da GPU.

Tipo	Nome mais descritivo	Termo antigo mais próximo, fora das GPUs	Termo oficial da GPU CUDA/NVIDIA	Definição do livro
Program abstractions	Loop vtorizável	Loop vtorizável	Grade	Um loop vtorizável, executado sobre a GPU, composto de um ou mais blocos de threads (corpos de loop vtorizado) que podem ser executados em paralelo.
	Corpo do loop vtorizável	Corpo de um loop vtorizado (garimpado)	Bloco de thread	Um loop vtorizado executado em um processador SIMD multithreaded, composto de uma ou mais threads de instruções SIMD. Elas podem se comunicar via memória local.
	Seque de operações de pista SIMD	Uma iteração de um loop escalar	Thread CUDA	Um corte vertical de uma thread de instruções SIMD correspondente a um elemento executado por uma pista SIMD. O resultado é armazenado dependendo da máscara e registrador de predicado.
Objeto de máquina	Uma thread de instruções SIMD	Thread of Vector Instructions	Warp	Uma thread tradicional, mas contém apenas instruções SIMD que são executadas em um processador SIMD multithreaded. Resultados armazenados dependendo de uma máscara por elemento.
	Instrução SIMD	Instrução de vetor	Instrução PTX	Uma única instrução SIMD executada através das pistas SIMD.
Hardware de processamento	Processador SIMD multithreaded	(Multithreaded) Vector Processor	Streaming Multiprocessor	Um processador SIMD multithreaded executa threads de instruções SIMD, independente de outros processadores SIMD.
	Escalonador de bloco de thread	Processador escalar	Giga Thread Engine	Atribui múltiplos blocos de threads (corpos de loop vtorizado) a processadores SIMD multithreaded.
	SIMD Thread Scheduler	Escalonador de threads em uma CPU multithreaded	Escalonador de warp	Unidade de hardware que escalona e emite threads de instruções SIMD quando elas estão prontas para serem executadas; inclui um scoreboard para rastrear a execução da thread SIMD.
	Pista SIMD	Pista de vetor	Processador de threads	Uma pista SIMD executa as operações em uma thread de instruções SIMD sobre um único elemento. Resultados armazenados dependendo da máscara.
Hardware de memória	Memória da GPU	Memória principal	Memória global	Memória DRAM acessível por todos os processadores SIMD multithreaded em uma GPU.
	Memória local	Memória local	Memória compartilhada	SRAM local rápida para um processador SIMD multithreaded, indisponível a outros processadores SIMD.
	Registradores de pista SIMD	Registradores de pista de vetor	Registradores do processador de threads	Registradores em uma única pista SIMD alocada por um bloco de threads completo (corpo de loop vtorizado).

FIGURA 6.12 Guia rápido para os termos referentes à GPU.

Usamos a primeira coluna para os termos do hardware. Quatro grupos dividem esses 12 termos. De cima para baixo:
Abstrações de Programa, Objetos de Máquina, Hardware de Processamento e Hardware de Memória.

Embora as GPUs estejam se movendo para o ramo principal da computação, elas não podem abandonar sua responsabilidade por continuar a se superar nos gráficos. Assim, o projeto de GPUs pode fazer mais sentido quando os arquitetos perguntam, dado o hardware investido para trabalhar bem com gráficos, como podemos complementá-lo a fim de melhorar o desempenho de uma gama maior de aplicações.

Tendo abordado dois estilos diferentes de MIMD que possuem um espaço de

endereços compartilhado, a seguir, apresentamos os processadores paralelos, onde cada processador tem seu próprio espaço de endereços privado, facilitando bastante a criação de sistemas muito maiores. Os serviços de Internet que você usa diariamente dependem desses sistemas em larga escala.

Detalhamento

Embora a GPU tenha sido apresentada como tendo uma memória separada da CPU, tanto a AMD quanto a Intel anunciaram produtos “agregados”, que combinam GPUs e CPUs para compartilhar uma única memória. O desafio será manter a memória com alta largura de banda em uma arquitetura mista, que foi um dos alicerces das GPUs.

Verifique você mesmo

Verdadeiro ou falso: GPUs contam com chips de DRAM gráficos para reduzir a latência da memória e, portanto, aumentar o desempenho em aplicações gráficas.

6.7. Clusters, computadores em escala warehouse e outros multiprocessadores de passagem de mensagens

A técnica alternativa ao compartilhamento de um espaço de endereços é que cada processador tenha seu próprio espaço privado de endereços físicos. A [Figura 6.13](#) mostra a organização clássica de um multiprocessador com múltiplos espaços de endereços privados. Esse multiprocessador alternativo precisa se comunicar por meio da **passagem de mensagens** explícita, que tradicionalmente é o nome desse estilo de computadores. Desde que o sistema tenha rotinas para **enviar e receber mensagens**, a coordenação é embutida na passagem da mensagem, pois um processador sabe quando uma mensagem é enviada, e o processador receptor sabe quando uma mensagem chega. Se o emissor precisar de confirmação de que a mensagem chegou, o processador receptor poderá então enviar uma mensagem de confirmação para o emissor.

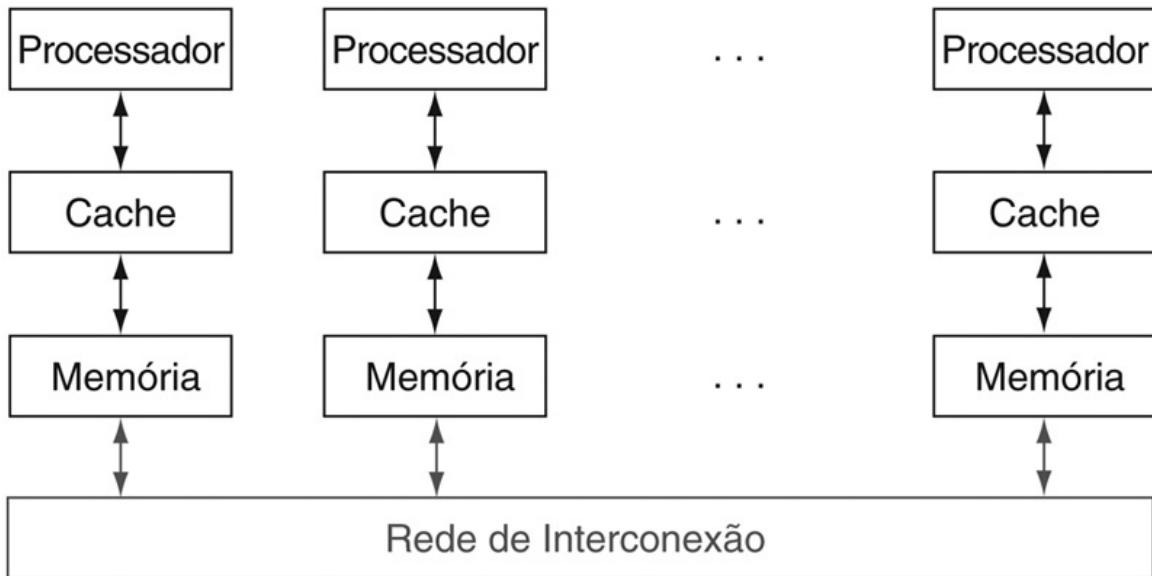


FIGURA 6.13 Organização clássica de um multiprocessador com múltiplos espaços de endereços privados, tradicionalmente chamado de multiprocessador de passagem de mensagens.

Observe que, diferente do SMP da [Figura 6.7](#), a rede de interconexão não está entre as caches e a memória, mas entre os nós processador-memória.

passagem de mensagens

Comunicação entre vários processadores enviando e recebendo informações explicitamente.

rotina para enviar mensagem

Uma rotina usada por um processador em máquinas com memórias privadas para passar uma mensagem a outro processador.

rotina para receber mensagem

Uma rotina usada por um processador em máquinas com memórias privadas para aceitar uma mensagem de outro processador.

Houve várias tentativas de construir computadores em larga escala com base em redes de passagem de mensagens de alto desempenho, e eles ofereceram

melhor desempenho de comunicação absoluta do que os clusters criados por meio de redes locais. Na verdade, muitos supercomputadores hoje utilizam redes customizadas. O problema é que eles são muito mais caros do que redes locais, como Ethernet. Poucas aplicações hoje, fora da computação de alto desempenho, poderiam justificar o desempenho de comunicação mais alto, dados os custos muito mais altos.

Interface hardware/software

Computadores que contam com a passagem de mensagens para a comunicação, em vez da memória compartilhada coerente com a cache, são muito mais fáceis para os projetistas de hardware (Seção 5.8). A vantagem para os programadores é que a comunicação é explícita, o que significa que existem menos surpresas de desempenho do que com a comunicação implícita nos computadores de memória compartilhada coerentes com a cache. A desvantagem para os programadores é que é mais difícil transportar um programa sequencial para um computador com passagem de mensagens, pois cada comunicação precisa ser identificada antecipadamente, ou o programa não funcionará. A memória compartilhada coerente com a cache permite que o hardware descubra quais dados precisam ser comunicados, o que facilita o transporte. Existem diferenças de opinião quanto ao caminho mais curto para o alto desempenho, dados os prós e contras da comunicação implícita, mas não há confusão no mercado hoje. Os microprocessadores multicore utilizam memória física compartilhada e os nós de um cluster se comunicam entre si usando passagem de mensagens.

Algumas aplicações concorrentes são bem executadas em hardware paralelo, independente de se ele oferece endereços compartilhados ou passagem de mensagens. Em particular, o paralelismo em nível de tarefa e aplicações com pouca comunicação — como busca na Web, servidores de correio e servidores de arquivos — não exigem que o endereçamento compartilhado funcione bem. Como resultado, os **clusters** se tornaram o exemplo mais divulgado atualmente do computador paralelo de passagem de mensagens. Dadas as memórias separadas, cada nó de um cluster executa uma cópia distinta do sistema operacional. Ao contrário, os núcleos dentro de um microprocessador são conectados usando uma rede de alta velocidade dentro do chip, e um sistema de memória compartilhada multichip utiliza a interconexão da memória para a

comunicação. A interconexão de memória possui maior largura de banda e menor latência, permitindo um desempenho de comunicação muito melhor para os multiprocessadores de memória compartilhada.

clusters

Coleções de computadores conectados por E/S por switches de rede padrão para formar um multiprocessador de passagem de mensagens.

O ponto fraco das memórias separadas para a memória do usuário de um ponto de vista da programação paralela se transforma em um ponto forte na estabilidade do sistema ([Seção 5.5](#)). Como um cluster consiste em computadores independentes conectados através de uma rede local, é muito mais fácil substituir um computador sem derrubar o sistema em um cluster do que em um multiprocessador de memória compartilhada. Fundamentalmente, o endereço compartilhado significa que é difícil isolar um processador e substituí-lo sem um trabalho heroico por parte do sistema operacional e no projeto físico do servidor. Também é fácil para os clusters reduzirem de tamanho de forma controlada quando um servidor falha, melhorando assim a **estabilidade**. Como o software do cluster é uma camada que roda em cima dos sistemas operacionais locais que rodam em cada computador, é muito mais fácil desconectar e substituir um computador defeituoso.



E S T A B I L I D A D E

Como os clusters são construídos por meio de computadores inteiros e redes independentes e escaláveis, esse isolamento também facilita expandir o sistema sem paralisar a aplicação que executa sobre o cluster.

Menor custo, maior disponibilidade e a rápida e gradual expansibilidade tornam os clusters atraentes para provedores de serviços de Internet, apesar de

seu desempenho de comunicação mais fraco em comparação com multiprocessadores de memória compartilhada em larga escala. Os mecanismos de busca que centenas de milhões de nós que utilizamos todos os dias dependem dessa tecnologia. Amazon, Facebook, Google, Microsoft e outros possuem múltiplos centros de dados, cada um com clusters de dezenas de milhares de processadores. É claro que o uso de múltiplos processadores nas empresas de serviço de Internet tem sido altamente bem-sucedido.

Computadores em escala warehouse

Qualquer um pode construir uma CPU veloz. O truque é construir um sistema veloz.

Seymour Cray, considerado o pai do supercomputador.

Os serviços de Internet, como aqueles que descrevemos acima, necessitavam da construção de novos prédios para abrigar, alimentar e resfriar 100.000 servidores. Embora eles possam ser classificados como apenas grandes clusters, arquitetura e operação são mais sofisticadas. Eles atuam como um computador gigante e custam na ordem de US\$150 milhões para o prédio, a infraestrutura elétrica e de resfriamento, os servidores e o equipamento de rede que conecta e abriga 50.000 a 100.000 servidores. Nós os consideramos uma nova classe de computador, chamada *computadores em escala warehouse* (WSC — Warehouse-Scale Computers).

Interface hardware/ software

A estrutura mais popular para o processamento batch em um WSC é MapReduce [Dean, 2008] e seu gêmeo open-source, Hadoop. Inspirado pelas funções Lisp com o mesmo nome, Map primeiro aplica uma função fornecida pelo programador a cada registro lógico de entrada. Map é executado em milhares de servidores para produzir um resultado intermediário de pares chave-valor. Reduce coleta a saída dessas tarefas distribuídas e as aglutina usando outra função definida pelo programador. Com suporte apropriado do software, ambos são altamente paralelos, embora fáceis de entender e usar. Dentro de 30 minutos, um programador iniciante pode executar uma tarefa MapReduce em milhares de servidores.

Por exemplo, um programa MapReduce calcula o número de ocorrências de

cada palavra em inglês em uma grande coleção de documentos. A seguir encontra-se uma versão simplificada desse programa, que mostra apenas o loop mais interno e considera apenas uma ocorrência de todas as palavras em inglês encontradas em um documento:

```
map(String key, String value):
    // chave: nome do documento
    // valor: conteúdo do documento
    for each word w in value:
        EmitIntermediate(w, "1"); // Produz lista de todas as palavras reduce(String key, Iterator values):
    // chave: uma palavra
    // valores: uma lista de contadores
    int result = 0;
    for each v in values:
        result += ParseInt(v); // obtém inteiro do par chave-valor
    Emit(AsString(result));
```

A função `EmitIntermediate` usada na função Map emite cada palavra no documento e o valor um. Depois, a função Reduce soma todos os valores por palavra para cada documento, usando `ParseInt()` para obter o número de ocorrências por palavra em todos os documentos. O ambiente de runtime MapReduce escalona tarefas map e tarefas reduce aos servidores de um WSC.

Nessa escala extrema, que requer inovação na distribuição de energia, resfriamento, monitoramento e operações, o WSC é um descendente moderno dos supercomputadores da década de 1970 — tornando Seymour Cray o padrinho dos arquitetos do WSC de hoje. Seus computadores extremos lidavam com cálculos que não podiam ser feitos em nenhum outro lugar, mas eram tão caros que apenas algumas poucas empresas poderiam pagar por eles. Atualmente, a meta é fornecer tecnologia de informação para o mundo, ao invés de computação de alto desempenho para cientistas e engenheiros. Logo, os WSCs certamente desempenham um papel mais importante para a sociedade hoje do que os supercomputadores de Cray no passado.

Embora compartilhando algumas metas comuns com os servidores, os WSCs possuem três distinções principais:



PARALELISMO

1. *Paralelismo Amplo e Fácil:* Uma preocupação para um arquiteto de servidor é se as aplicações no mercado alvo possuem paralelismo suficiente para justificar a quantidade de hardware paralelo e se o custo é muito alto para que o hardware de comunicação suficiente explore esse paralelismo. Um arquiteto WSC não tem essa preocupação. Primeiro, aplicações batch como MapReduce beneficiam-se do grande número de conjuntos de dados independentes que precisam de processamento independente, como as bilhões de páginas Web a partir de um Web crawl. Segundo, aplicações de serviço interativo na Internet, também conhecidas como **Software as a Service (SaaS)**, podem se beneficiar dos milhões de usuários independentes dos serviços interativos da Internet. Leituras e escritas raramente são dependentes no SaaS e, portanto, o SaaS raramente precisa de sincronização. Por exemplo, a busca usa um índice somente de leitura, e o e-mail normalmente está lendo e escrevendo informações independentes. Chamamos esse tipo de paralelismo fácil de *Paralelismo em Nível de Solicitação*, já que muitos esforços independentes podem prosseguir em paralelo naturalmente, com pouca necessidade de comunicação ou sincronização.
2. *Custos Operacionais Contam:* Tradicionalmente, arquitetos de servidor projetam seus sistemas para obter desempenho de pico dentro de um orçamento financeiro, e preocupam-se com a potência apenas para garantir que não excederão a capacidade de resfriamento de seus invólucros. Eles

normalmente ignoravam os custos operacionais de um servidor, supondo que são mínimos em comparação com os custos da compra. O WSC possui tempos de vida mais longos — o prédio e a infraestrutura elétrica e de resfriamento geralmente são amortizados por 10 ou mais anos —, de modo que os custos operacionais se acumulam: energia, distribuição de potência e resfriamento representam mais de 30% dos custos de um WSC durante 10 anos.

3. *Escala e as Oportunidades/Problemas Associados com a Escala:* Para construir um único WSC, é preciso comprar 100.000 servidores e também a infraestrutura de suporte, o que significa descontos por volume. Logo, os WSCs são tão grandes internamente que você consegue economia de escala mesmo que não haja muitos WSCs. Essas economias de escala levaram à *computação em nuvem*, já que os menores custos unitários de um WSC significavam que as empresas de nuvem poderiam alugar servidores a uma taxa lucrativa e ainda estar abaixo daquilo que custaria para os que desejam fazer isso externamente. O outro lado da oportunidade econômica da escala é a necessidade de lidar com a frequência de falha da expansão. Mesmo que um servidor tivesse um Tempo Médio Para a Falha de incríveis 25 anos (200.000 horas), o arquiteto WSC precisaria projetar para 5 falhas de servidor a cada ano. A [Seção 5.13](#) mencionou uma taxa de falha de disco anual (AFR), medida no Google, de 2% a 4%. Se houvesse 4 discos por servidor e sua taxa de falha anual fosse 2%, o arquiteto WSC deveria esperar ver um disco falhando a cada *hora*. Assim, a tolerância a falhas é ainda mais importante para o arquiteto WSC do que para o arquiteto de servidor.

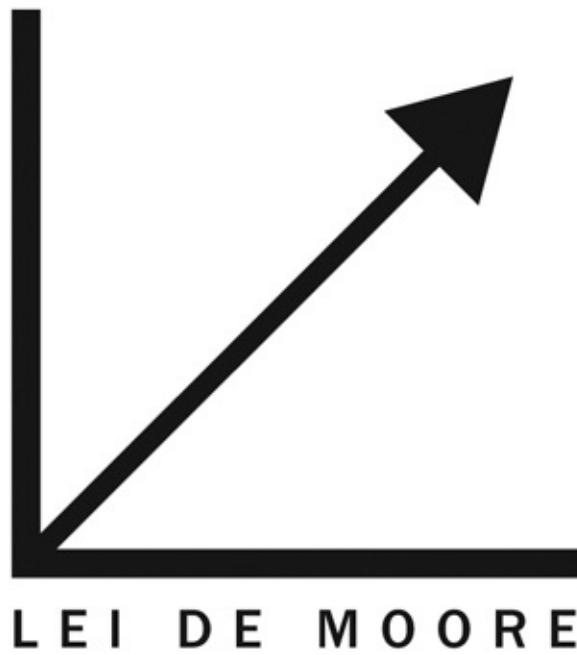
Software as a Service (SaaS)

Em vez de vender software instalado e executar nos próprios computadores dos clientes, o software é executado em um local remoto e fica disponível pela Internet normalmente por meio de uma interface web com os clientes. Clientes SaaS são cobrados com base no uso, ao contrário da posse.

As economias de escala desvendadas pelo WSC observaram o propósito há muito tempo sonhado da computação como um utilitário. A computação em nuvem significa que qualquer um, em qualquer lugar, com boas ideias, um modelo de negócios e um cartão de crédito pode aproveitar dos milhares de

servidores para oferecer sua visão quase instantaneamente para o mundo inteiro. Naturalmente, existem obstáculos importantes que poderiam limitar o crescimento da computação em nuvem — como a segurança, a privacidade, padrões e a taxa de crescimento da largura de banda da Internet —, mas podemos prever que isso está sendo tratado, de modo que os WSCs e a computação em nuvem possam crescer. Para entender melhor a taxa de crescimento da computação em nuvem, em 2012, a Amazon Web Services (AWS) anunciou que acrescenta nova capacidade de servidor *a cada dia* para dar suporte a toda a infraestrutura global da Amazon em 2003, quando a Amazon era uma empresa com receita anual de US\$ 5,2 bilhões, com 6000 empregados.

Agora que compreendemos a importância dos multiprocessadores por passagem de mensagens, especialmente para a computação em nuvem, em seguida veremos as formas de juntar os nós de um WSC. Graças à **Lei de Moore** e o número cada vez maior de núcleos por chip, agora precisamos também de redes dentro de um chip, de modo que essas topologias são importantes na grande escala e também na pequena.



Detalhamento

A estrutura MapReduce embaralha e ordena os pares chave-valor no final da

fase Map, para produzir grupos que compartilhem a mesma chave. Esses grupos são então passados para a fase Reduce.

Detalhamento

Outra forma de computação em grande escala é a *computação em grade*, em que os computadores são espalhados por grandes áreas, e depois os programas que executam neles precisam se comunicar por redes de longa distância. A forma mais comum e exclusiva de computação em grade foi promovida pelo projeto SETI@home. Observou-se que milhões de PCs ficam ociosos em determinado momento, sem realizar nada de útil, e eles poderiam ser apanhados e ter boa utilidade se alguém desenvolvesse software que pudesse rodar nesses computadores e depois dar a cada PC uma parte independente do problema para atuar. O primeiro exemplo foi o Search for ExtraTerrestrial Intelligence (SETI), ou *busca por inteligência extraterrestre*, que foi lançado na UC Berkeley em 1999. Mais de 5 milhões de usuários de computador em mais de 200 países se inscreveram para o SETI@home, com mais de 50% fora dos EUA. Ao final de 2011, o desempenho médio da grade SETI@home era de 3,5 PetaFLOPS.

Verifique você mesmo

1. Verdadeiro ou falso: assim como os SMPs, os computadores com passagem de mensagens contam com locks para a sincronização.
2. Verdadeiro ou falso: os clusters possuem memória separadas e, portanto, precisam de muitas cópias do sistema operacional.

6.8. Introdução às topologias de rede multiprocessador

Os chips multicore exigem que as redes nos chips conectem os núcleos, e os clusters exigem redes locais que conectem os servidores. Esta seção revisa os prós e os contras de diferentes topologias de redes de interconexão.

Os custos de rede incluem o número de switches, o número de links em um switch que se conectam à rede, a largura (número de bits) por link, o tamanho dos links quando a rede é mapeada no chip. Por exemplo, alguns núcleos ou servidores podem ser adjacentes e outros podem estar no outro lado do chip ou no outro lado do centro de dados. O desempenho da rede também tem muitas faces. Ele inclui a latência em uma rede não carregada para enviar e receber uma mensagem, a vazão em termos do número máximo de mensagens que podem ser transmitidas em determinado período de tempo, atrasos causados pela disputa por uma parte da rede, e desempenho variável dependendo do padrão de comunicação. Outra obrigação da rede pode ser tolerância a falhas, pois os sistemas podem ter de operar na presença de componentes defeituosos. Finalmente, nesta era de chips de potência limitada, a eficiência de potência das diferentes organizações pode superar outros aspectos.

As redes normalmente são desenhadas como gráficos, com cada arco do gráfico representando um link da rede de comunicação. Nas figuras desta seção, o nó processador-memória aparece como um quadrado preto, e o switch aparece como um círculo claro. Nesta seção, todos os links são *bidirecionais*; ou seja, a informação pode fluir em qualquer direção. Todas as redes consistem em switches cujos links vão para os nós processador-memória e para outros switches. A primeira rede conecta uma sequência de nós:



Essa topologia é chamada de *anel*. Como alguns nós não são conectados diretamente, algumas mensagens terão um salto por nós intermediários até que cheguem ao destino final.

Diferente de um barramento — um conjunto compartilhado de fios que permitem o broadcasting para todos os dispositivos conectados —, um anel é capaz de realizar muitas transferências simultâneas.

Como existem diversas topologias para escolher, métricas de desempenho são necessárias a fim de distinguir esses projetos. Duas são comuns. A primeira é a **largura de banda de rede total**, que é a largura de banda de cada link multiplicado pelo número de links, e representa a largura de banda máxima. Para a rede de anel apresentada, com P processadores, a largura de banda de rede total seria P vezes a largura de banda do link; a largura de banda de rede total de um barramento é a largura de banda desse barramento.

largura de banda de rede

Informalmente, a taxa de transferência máxima de uma rede; pode se referir à velocidade de um único link ou a taxa de transferência coletiva de todos os links na rede.

Para balancear esse caso com melhor largura de banda, incluímos outra métrica que é mais próxima do pior caso: a **largura de banda da corte**. Esta é calculada dividindo-se a máquina em duas metades. Depois você soma a largura de banda dos links que cruzam essa linha divisória imaginária. A largura de banda de corte de um anel é duas vezes a largura de banda do link, e é uma vez a largura de banda de link para o barramento. Se um único link for tão rápido quanto o barramento, o anel tem apenas o dobro da velocidade de um barramento no pior caso, mas é P vezes mais rápido no melhor caso.

largura de banda de corte

A largura de banda entre duas partes iguais de um multiprocessador. Essa medida é para uma divisão do multiprocessador no pior caso.

Como algumas topologias de rede não são simétricas, surge a questão de onde desenhar a linha imaginária quando fizer o corte da máquina. A largura de banda de corte é uma métrica do pior caso, de modo que a resposta é escolher a divisão que gera o desempenho de rede mais pessimista. Em outras palavras, calcule todas as larguras de banda de corte possíveis e escolha a menor. Tomamos a visão pessimista porque programas paralelos normalmente são limitados pelo elo

mais fraco na cadeia de comunicação.

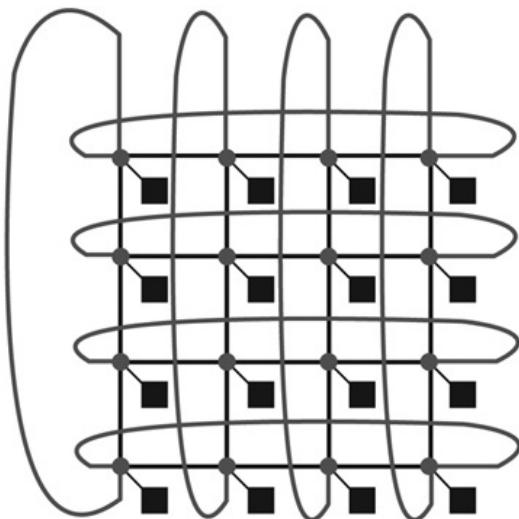
No outro extremo de um anel está a **rede totalmente conectada**, em que cada processador tem um link bidirecional com cada outro processador. Para as redes totalmente conectadas, a largura de banda de rede total é $P \times (P - 1)/2$, e a largura de banda de corte é $(P/2)^2$.

rede totalmente conectada

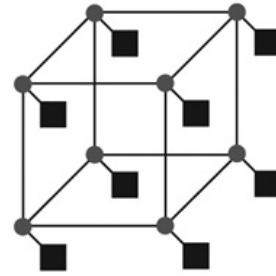
Uma rede que conecta nós processador-memória fornecendo um link de comunicação dedicado entre cada nó.

A tremenda melhoria no desempenho das redes totalmente conectadas é anulada pelo enorme aumento no custo. Essa consequência inspira os engenheiros a inventarem novas topologias que estão entre o custo dos anéis e o desempenho das redes totalmente conectadas. A avaliação do sucesso depende em grande parte da natureza da comunicação na carga de trabalho de programas paralelos executados na máquina.

O número de topologias diferentes que foram discutidas nas diversas publicações seria difícil de contar, mas somente uma minoria foi utilizada em processadores paralelos comerciais. A [Figura 6.14](#) ilustra duas das topologias mais comuns.



a. Grade 2-D ou malha de 16 nós



b. Árvore de n cubos com 8 nós
($8 = 2^3$, de modo que $n = 3$)

FIGURA 6.14 Topologias de rede que apareceram nos processadores paralelos comerciais.

Os círculos claros representam switches, e os quadrados pretos representam nós processador-memória. Embora um switch tenha muitos links, geralmente apenas um vai para o processador. A topologia booliana de n cubos é uma interconexão n -dimensional com 2^n nós, exigindo n links por switch (mais um para o processador) e, portanto, n nós com o vizinho mais próximo. Constantemente, essas topologias básicas têm sido suplementadas com arcos extras para melhorar o desempenho e a confiabilidade.

Uma alternativa a colocar um processador em cada nó de uma rede é deixar apenas o switch em alguns desses nós. Os switches são menores que os nós processador-memória-switch, e assim podem ser mais bem compactados, encurtando assim a distância e aumentando o desempenho. Essas redes normalmente são chamadas **redes multiestágio** para refletir as múltiplas etapas em que uma mensagem pode trafegar. Os tipos de redes multiestágio são tão numerosos quanto as redes de único estágio; a [Figura 6.15](#) ilustra duas das organizações multiestágio mais comuns. Uma **rede crossbar** permite que qualquer nó se comunique com qualquer outro nó em uma passada pela rede. Uma *rede Ômega* usa menos hardware do que a rede crossbar ($2n \log_2 n$ contra n^2 switches), mas pode ocorrer disputa entre as mensagens, dependendo do padrão de comunicação. Por exemplo, a rede Ômega na [Figura 6.15](#) não pode enviar uma mensagem de P_0 a P_6 ao mesmo tempo em que envia uma mensagem de P_1 a P_4 .

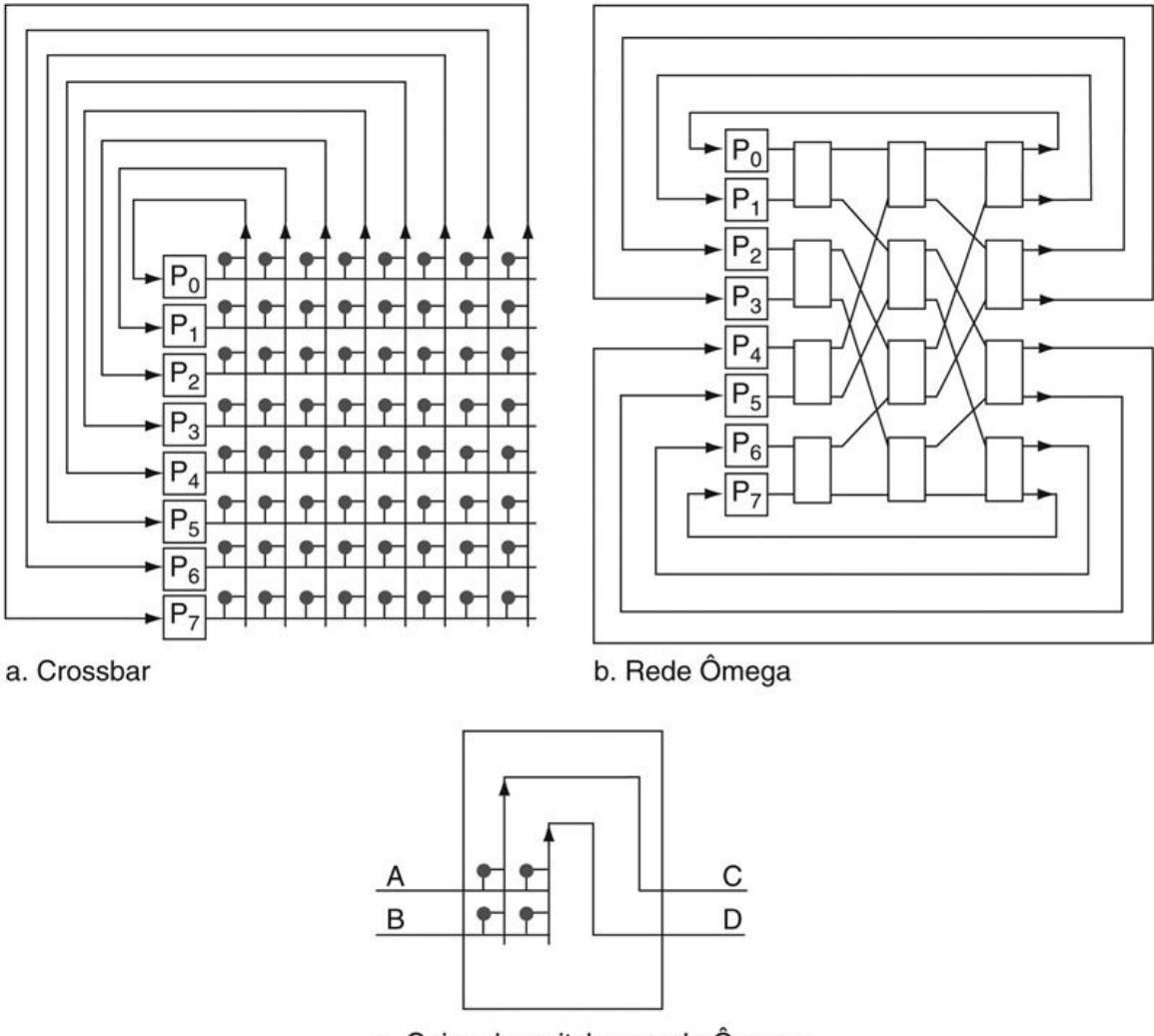


FIGURA 6.15 Topologias comuns de rede multiestágio para oito nós.

Os switches nesses desenhos são mais simples do que nos desenhos anteriores, pois os links são unidirecionais; os dados entram na parte de baixo e saem pelo link da direita. A caixa de switch em c pode passar A para C e B para D ou B para C e A para D. O crossbar usa n^2 switches, em que n é o número de processadores, enquanto a rede Ômega usa $2n \log_2 n$ caixas de switch grandes, cada uma composta logicamente por quatro dos switches menores. Nesse caso, o crossbar utiliza 64 switches contra 12 caixas de switch, ou 48 switches, na rede Ômega. O crossbar, porém, pode aceitar qualquer combinação de mensagens entre os processadores, enquanto a rede Ômega não pode.

rede multiestágio

Uma rede que fornece um pequeno switch em cada nó.

rede crossbar

Uma rede que permite que qualquer nó se comunique com qualquer outro nó em uma passada pela rede.

Implementando topologias de rede

Nesta seção, a análise simples de todas as redes ignora considerações práticas importantes na elaboração de uma rede. A distância de cada link afeta o custo da comunicação com uma taxa de clock muito alta; geralmente, quanto maior a distância, mais cara é a manutenção de uma taxa de clock alta. Distâncias mais curtas também facilitam a atribuição de mais fios ao link, pois a potência necessária para usar muitos fios é menor se os fios forem curtos. Fios mais curtos também custam menos que os mais longos. Outra limitação prática é que os desenhos tridimensionais precisam ser mapeados em chips, cuja mídia é basicamente bidimensional. O último aspecto é o da energia. Questões relacionadas à energia, por exemplo, podem forçar os chips multicore a utilizarem topologias de grade simples. O resultado disso é que topologias que parecem ser elegantes quando desenhadas na prancheta podem ser impraticáveis quando construídas em silício ou em um centro de dados. Agora que compreendemos a importância dos clusters e vimos as topologias que os acompanham para conectá-los, em seguida, veremos o hardware e o software de interface da rede com o processador.

Verifique você mesmo

Verdadeiro ou falso: Para um anel com P nós, a razão entre a largura de banda total da rede e a largura de banda de corte é $P/2$.

6.9. Benchmarks de multiprocessador e modelos de desempenho

Como vimos no [Capítulo 1](#), sistemas de benchmarking sempre é um assunto delicado, pois é uma forma altamente visível de tentar determinar qual sistema é

melhor. Os resultados afetam não apenas as vendas de sistemas comerciais, mas também a reputação dos projetistas desses sistemas. Logo, os participantes querem ganhar a competição, mas eles também querem ter certeza de que, se alguém mais ganhar, eles mereçam ganhar porque possuem um sistema genuinamente melhor. Esse desejo ocasiona regras para garantir que os resultados do benchmark não sejam simplesmente truques de engenharia para esse benchmark, mas, em vez disso, avanços que melhoraram o desempenho de aplicações reais.

Para evitar possíveis truques, uma boa regra é que você não pode mudar o benchmark. O código-fonte e os conjuntos de dados são fixos, e existe uma única resposta apropriada. Qualquer desvio dessas regras torna os resultados inválidos.

Muitos benchmarks de multiprocessador seguem essas tradições. Uma exceção comum é ser capaz de aumentar o tamanho do problema de modo que você possa executar o benchmark em sistemas com um número bem diferente de processadores. Ou seja, muitos benchmarks permitem pouca facilidade de expansão, em vez de exigir muita facilidade, embora você deva ter cuidado ao comparar resultados para programas executando problemas com diferentes tamanhos.

A [Figura 6.16](#) é um resumo de vários benchmarks paralelos, também descritos a seguir:

- *Linpack* é uma coleção de rotinas de álgebra linear, e as rotinas para realizar a eliminação Gaussiana constituem o que é conhecido como benchmark Linpack. A rotina DGEMM no exemplo da [Seção 3.5](#) representa uma pequena fração do código fonte do benchmark Linpack, mas é responsável pela maior parte do tempo de execução do benchmark. Ele permite expansão fraca, deixando que o usuário escolha qualquer tamanho de problema. Além do mais, ele permite que o usuário reescreva o Linpack em qualquer formato e em qualquer linguagem, desde que calcule o resultado apropriado e realize o mesmo número de operações de ponto flutuante para um problema de determinado tamanho. Duas vezes por ano, os 500 computadores com o desempenho Linpack mais rápido são publicados em www.top500.org. O primeiro nessa lista é considerado pela imprensa como o computador mais rápido do mundo.

Benchmark	Expansão?	Reprograma?	Descrição
Linpack	Fraca	Sim	Matriz densidade linear em álgebra [Dongarra, 1979]
SPECrate	Fraca	Não	Trabalho independente de paralelismo [Henning, 2007]
Stanford Parallel Applications for Shared Memory SPLASH 2 [Woo et al., 1995]	Forte (embora ofereça dois tamanhos de problema)	Não	Complexo 1D FFT Decomposição LU bloqueada Fatorização de Cholesky esparsa bloqueada Radix Sort inteiro Barnes-Hut Multipole Rápida e Adaptativa Simulação de oceano Radiosidade hierárquica Traçador de raios Renderizador de Volume Simulação de água com estrutura de dados espaciais Simulação de água sem estrutura de dados espaciais
NAS Parallel Benchmarks [Bailey et al., 1991]	Fraca	Sim (C ou Fortran somente)	EP: embaralhosamente paralelo MG: multigrade simplificada CG: grade não estruturada para um método gradiente conjugado FT: equação diferencial parcial 3-D utilizando FFTs IS: maior sort de inteiro
PARSEC Benchmark Suite [Bienia et al., 2008]	Fraca	Não	Blackscholes – Opção de especificação com o equação diferencial parcial (PDE) Black-Scholes Bodytrack – Rastreamento do corpo de uma pessoa Canneal – Reconhecimento simulado cache-aware para otimizar o roteamento Dedup – Compressão de próxima geração com deduplicação de dados. Facesim – Simulação dos movimentos de um rosto humano Ferret – Servidor de busca de conteúdos similares Fluidanimate – Dinâmicas de fluidos para uma animação com método SPH Freqmine – Mineração frequente de conjunto de itens Streamcluster – Agrupamento on-line de um input stream Swaptions – Precificação de um portfolio de swaptions Vips – Processamento de imagem x264 – Codificação de vídeo H.264
Berkeley Design Patterns [Asanovic et al., 2006]	Forte ou fraca	Sim	Máquina em estado finito Lógica Combinatória Gráfico Transversal Grade estruturada Matriz densidade Matriz esparsa Métodos espetrais (FFT) Programação dinâmica N-Body MapReduce Backtrack/Desvio e Limite Inferência modelo gráfica Grade não estruturada

FIGURA 6.16 Exemplos de benchmarks paralelos.

- *SPECrate* é uma métrica de vazão baseada nos benchmarks SPEC CPU, como SPEC CPU 2006 ([Capítulo 1](#)). Em vez de relatar o desempenho dos programas individuais, *SPECrate* executa muitas cópias do programa simultaneamente. Assim, ele mede o paralelismo em nível de tarefa, pois não há comunicação entre as tarefas. Você pode executar tantas cópias dos programas quantas desejar, de modo que essa novamente é uma forma de expansão fraca.
- *SPLASH* e *SPLASH 2* (Stanford Parallel Applications for Shared Memory)

foram esforços realizados por pesquisadores na Stanford University na década de 1990 para reunir um conjunto de benchmarks paralelo, semelhante em objetivos ao conjunto de benchmarks SPEC CPU. Ele inclui kernels e aplicações, incluindo muitos advindos da comunidade de computação de alto desempenho. Esse benchmark requer expansão forte, embora venha com dois conjuntos de dados.

- Os *benchmarks paralelos NAS* (*NASA Advanced Supercomputing*) foram outra tentativa na década de 1990 de realizar o benchmark em multiprocessadores. Tomados da dinâmica de fluidos computacional, eles consistem em cinco kernels, e permitem expansão fraca, definindo alguns poucos conjuntos de dados. Assim como o Linpack, esses benchmarks podem ser reescritos, mas as regras exigem que a linguagem de programação só possa ser C ou Fortran.
- O recente *conjunto de benchmarks PARSEC* (*Princeton Application Repository for Shared Memory Computers*) consiste em programas multithreaded que usam **Pthreads** (POSIX threads) e OpenMP (Open MultiProcessing; veja a [Seção 6.5](#)). Eles focam em domínios computacionais emergentes e consistem de nove aplicações e três kernels. Oito contam com paralelismo de dados, três contam com paralelismo em pipeline, e um com paralelismo não estruturado.
- Tratando-se da nuvem, o objetivo do *Yahoo! Cloud Serving Benchmark* (YCSB) é comparar o desempenho dos serviços de dados da nuvem. Ele oferece um framework que facilita a execução do benchmark de novos serviços de dados pelos clientes, usando Cassandra e HBase como exemplos representativos. [Cooper, 2010]

Pthreads

Uma API do UNIX para criar e manipular threads. Ela é estruturada como uma biblioteca.

O lado negativo dessas restrições tradicionais dos benchmarks é que a inovação é limitada principalmente à arquitetura e compilador. Melhores estruturas de dados, algoritmos, linguagens de programação e assim por diante, geralmente, não podem ser usados, pois isso geraria um resultado ilusório. O sistema poderia ganhar, digamos, por causa do algoritmo, e não por causa do hardware ou do compilador.

Embora essas orientações sejam compreensíveis quando os alicerces da computação são relativamente estáveis — como eram na década de 1990 e na primeira metade da década seguinte —, elas são indesejáveis durante uma revolução da programação. Para que essa revolução tenha sucesso, precisamos encorajar a inovação em todos os níveis.

Uma técnica foi defendida pelos pesquisadores na Universidade da Califórnia em Berkeley. Eles identificaram 13 padrões de projeto, que afirmam que será parte das aplicações do futuro. Esses padrões de projeto são implementados por frameworks ou kernels. Alguns exemplos são matrizes esparsas, grades estruturadas, máquinas de estados finitos, *map reduce* e travessia de gráfico. Mantendo as definições em um alto nível, elas esperam encorajar inovações em qualquer nível do sistema. Assim, o sistema com o solucionador de matriz esparsa mais rápido, fica livre para usar qualquer estrutura de dados, algoritmo e linguagem de programação, além de novas arquiteturas e compiladores.

Modelos de desempenho

Um tópico relacionado aos benchmarks são os modelos de desempenho. Como temos visto com a crescente diversidade arquitetônica neste capítulo — multithreading, SIMD, GPUs —, seria especialmente útil se tivéssemos um modelo simples que oferecesse critérios para o desempenho de diferentes arquiteturas. Ele não precisa ser perfeito, apenas criterioso.

O modelo 3Cs para o desempenho de cache, do [Capítulo 5](#), é um exemplo de modelo de desempenho. Ele não é um modelo de desempenho perfeito, pois ignora fatores potencialmente importantes, como tamanho de bloco, diretiva de alocação de bloco e diretiva de substituição de bloco. Além do mais, ele possui algumas peculiaridades. Por exemplo, uma falta pode ser atribuída à capacidade em um projeto e a uma falha de conflito em outra cache com o mesmo tamanho. Mesmo assim, o modelo 3Cs já é popular há mais de 25 anos, pois oferece ideias para o comportamento dos programas, ajudando arquitetos e programadores a melhorarem suas criações com base em concepções desse modelo.

Para encontrar tal modelo para computadores paralelos, vamos começar com kernels pequenos, como aqueles dos 13 padrões de projeto Berkeley, da [Figura 6.16](#). Embora existam versões com diferentes tipos de dados para esses kernels, o ponto flutuante é popular em diversas implementações. Portanto, o desempenho de pico com ponto flutuante é um limite para a velocidade desses kernels em determinado computador. Para chips multicore, o desempenho de

pico com ponto flutuante é o desempenho de pico coletivo de todos os núcleos no chip. Se houvesse múltiplos microprocessadores no sistema, você multiplicaria o pico por chip pelo número total de chips.

As demandas no sistema de memória podem ser estimadas dividindo-se esse desempenho de pico em ponto flutuante pelo número médio de operações de ponto flutuante por byte acessado:

$$\frac{\text{Operações de ponto flutuante/seg}}{\text{Operações de ponto flutuante/Byte}} = \text{Bytes/seg}$$

A razão entre operações de ponto flutuante por byte de memória acessada é chamada de **intensidade aritmética**. Ela pode ser calculada apanhando-se o número total de operações de ponto flutuante para um programa dividido pelo número total de bytes de dados transferidos para a memória principal durante a execução do programa. A [Figura 6.17](#) mostra a intensidade aritmética de vários dos padrões de projeto de Berkeley da [Figura 6.16](#).

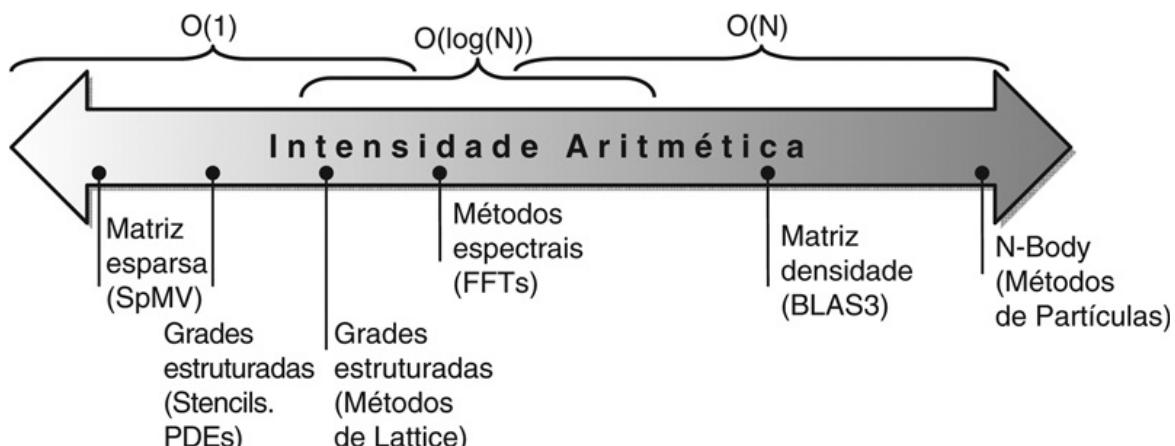


FIGURA 6.17 Intensidade aritmética, especificada como o número de operações de ponto flutuante para executar o programa dividido pelo número de bytes acessados na memória principal (Williams, Waterman e Patterson, 2009).

Alguns kernels possuem uma intensidade aritmética que se expande com o tamanho do problema, como Matrizes Densidade, mas existem muitos kernels com intensidades aritméticas independentes do tamanho do problema. Para os

kernels nesse primeiro caso, a expansão fraca pode levar a diferentes resultados, pois coloca muito menos demanda sobre o sistema de memória.

intensidade aritmética

A razão entre as operações de ponto flutuante em um programa e o número de bytes de dados acessados por um programa a partir da memória principal.

O modelo roofline

Este modelo simples reúne desempenho de ponto flutuante, intensidade aritmética e desempenho da memória em um gráfico bidimensional (Williams, Waterman e Patterson, 2009). O desempenho de pico em ponto flutuante pode ser encontrado usando as especificações de hardware mencionadas anteriormente. Os conjuntos de trabalho dos kernels que consideramos aqui não se encaixam em caches no chip, de modo que o desempenho de pico da memória pode ser definido pelo sistema de memória por trás das caches. Um modo de encontrar o desempenho de pico da memória é o benchmark Stream. (Veja a seção *Detalhamento* na Seção “Tecnologia DRAM”, no [Capítulo 5](#).)

A [Figura 6.18](#) mostra o modelo, que é feito uma vez para um computador, e não para cada kernel. O eixo-Y vertical é o desempenho de ponto flutuante alcançável de 0,5 a 64,0 GFLOPs/seg. O eixo-X horizontal é a intensidade aritmética, variando de 1/8 FLOPs/DRAM por byte acessado a 16 FLOPs/DRAM por byte acessado. Observe que o gráfico é uma escala log-log.

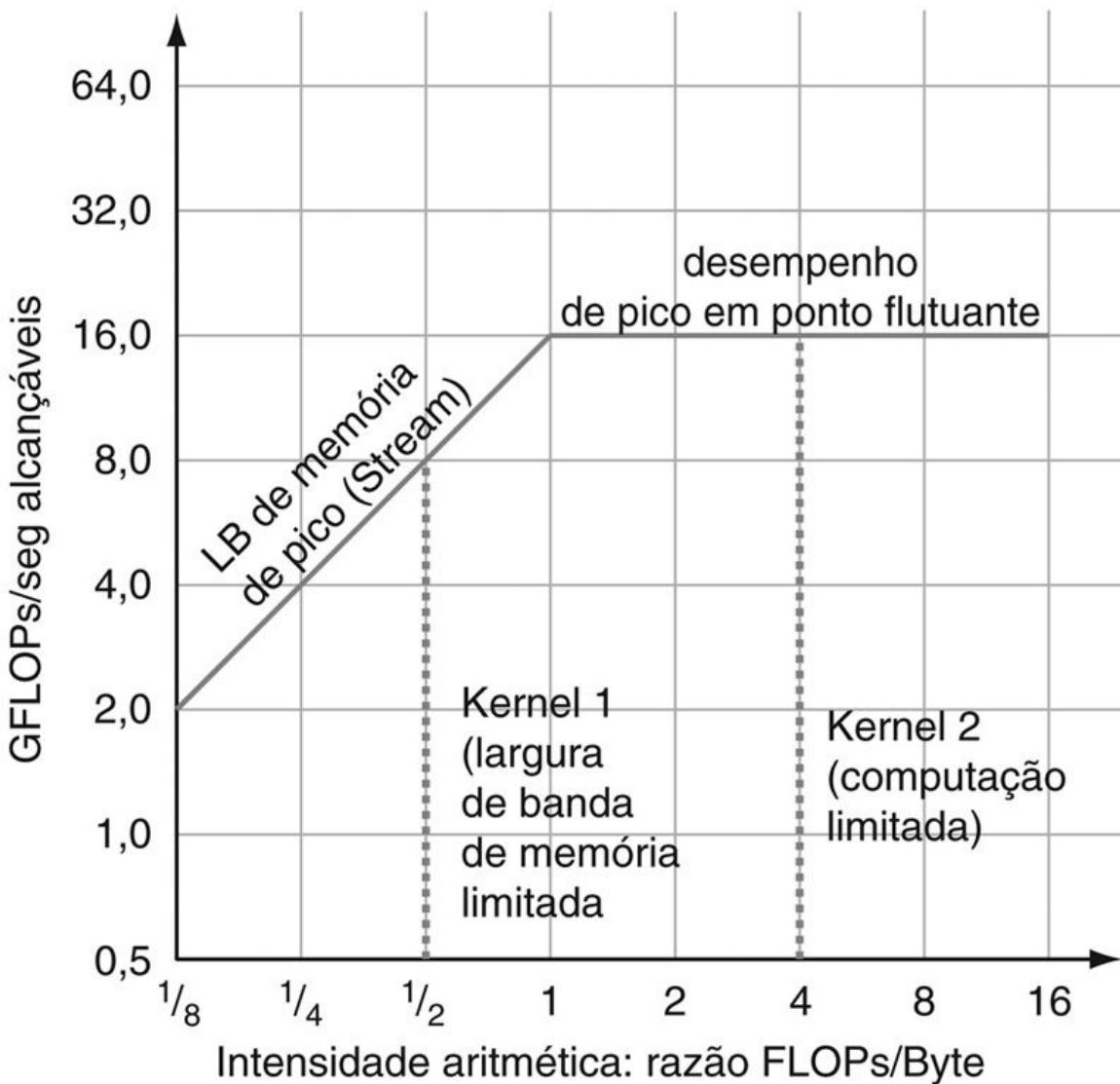


FIGURA 6.18 Modelo Roofline (Williams, Waterman e Patterson, 2009).

Este exemplo tem um desempenho de pico em ponto flutuante de 16 GFLOPs/segundo e uma largura de banda da memória de pico de 16 GB/segundo do benchmark Stream. (Como o Stream na realidade tem quatro medições, essa linha é a média das quatro.) A linha vertical pontilhada à esquerda representa o Kernel 1, que tem uma intensidade aritmética de 0,5 FLOPs/byte. Ela é limitada pela largura de banda de memória a não mais que 8 GFLOPs/segundo nesse Opteron X2. A linha vertical pontilhada à direita representa o Kernel 2, que tem uma intensidade aritmética de 4 FLOPs/byte. Ela é limitada apenas computacionalmente a 16 GFLOPs/segundo. (Esses dados são baseados no AMD Opteron X2 (Revision F) usando dual cores executando a 2GHz em um sistema dual socket.)

Para determinado kernel, podemos encontrar um ponto no eixo X com base em sua intensidade aritmética. Se desenhássemos uma linha vertical passando por esse ponto, o desempenho do kernel nesse computador teria de ficar em algum lugar nessa linha. Podemos desenhar uma linha horizontal mostrando o desempenho de pico em ponto flutuante do computador. Obviamente, o desempenho real em ponto flutuante não pode ser maior que a linha horizontal, pois esse é um limite do hardware.

Como poderíamos desenhar o desempenho de pico da memória, que é medido em bytes/seg? Como o eixo X é FLOPs/byte e o eixo Y é FLOPs/seg, bytes/seg é simplesmente uma linha diagonal em um ângulo de 45 graus nessa figura. Logo, podemos desenhar uma terceira linha que mostre o desempenho máximo em ponto flutuante que o sistema de memória desse computador pode suportar para determinada intensidade aritmética. Podemos expressar os limites como uma fórmula para desenhar a linha no gráfico da [Figura 6.18](#):

$$\text{GFLOPs / seg alcançável} = \text{Min} (\text{LB memória de pico} \times \text{Intensidade aritmética, Desempenho de pico em ponto flutuante})$$

As linhas horizontal e diagonal dão nome a esse modelo simples e indicam seu valor. A “roofline” (linha do telhado) define um limite superior no desempenho de um kernel, dependendo de sua intensidade aritmética. Dada uma roofline de um computador, você pode aplicá-la repetidamente, pois ela não varia por kernel.

Se pensarmos na intensidade aritmética como um poste que atinge o telhado, ou ele atinge a parte inclinada do telhado, o que significa que o desempenho por fim está limitado pela largura de banda da memória, ou atinge a parte plana do telhado, o que significa que o desempenho é computacionalmente limitado. Na [Figura 6.18](#), o kernel 1 é um exemplo do primeiro, e o kernel 2 é um exemplo do segundo.

Observe que o “ponto de cumeeira”, em que os telhados diagonal e horizontal se encontram, oferece uma percepção interessante para o computador. Se for muito longe à direita, então somente os kernels com intensidade aritmética muito alta podem alcançar o desempenho máximo desse computador. Se for muito à esquerda, então quase todo kernel poderá potencialmente atingir o desempenho máximo.

Comparando duas gerações de Opterons

O AMD Opteron X4 (Barcelona) com quatro núcleos é o sucessor do Opteron X2 com dois núcleos. Para simplificar o projeto da placa, eles usam o mesmo soquete. Logo, eles possuem os mesmos canais de DRAM e, portanto, a mesma largura de banda de memória de pico. Além de dobrar o número de núcleos, o Opteron X4 também tem o dobro do desempenho de pico em ponto flutuante por núcleo: os núcleos do Opteron X4 podem emitir duas instruções SSE2 de ponto flutuante por ciclo de clock, enquanto os núcleos do Opteron X2 emitem no máximo uma. Como os dois sistemas que estamos comparando possuem taxas de clock semelhantes — 2,2GHz para o Opteron X2 contra 2,3GHz para o Opteron X4 — o Opteron X4 tem cerca de quatro vezes o desempenho de ponto flutuante de pico do Opteron X2 com a mesma largura de banda de DRAM. O Opteron X4 também tem uma cache L3 de 2MiB, que não é encontrada no Opteron X2.

A [Figura 6.19](#) compara os modelos roofline para ambos os sistemas. Como poderíamos esperar, o ponto de cumeeira move-se para a direita, passando de 1 no Opteron X2 para 5 no Opteron X4. Logo, para ver um ganho de desempenho na próxima geração, os kernels precisam de uma intensidade aritmética maior que 1, ou seus conjuntos de trabalho terão de caber nas caches do Opteron X4.

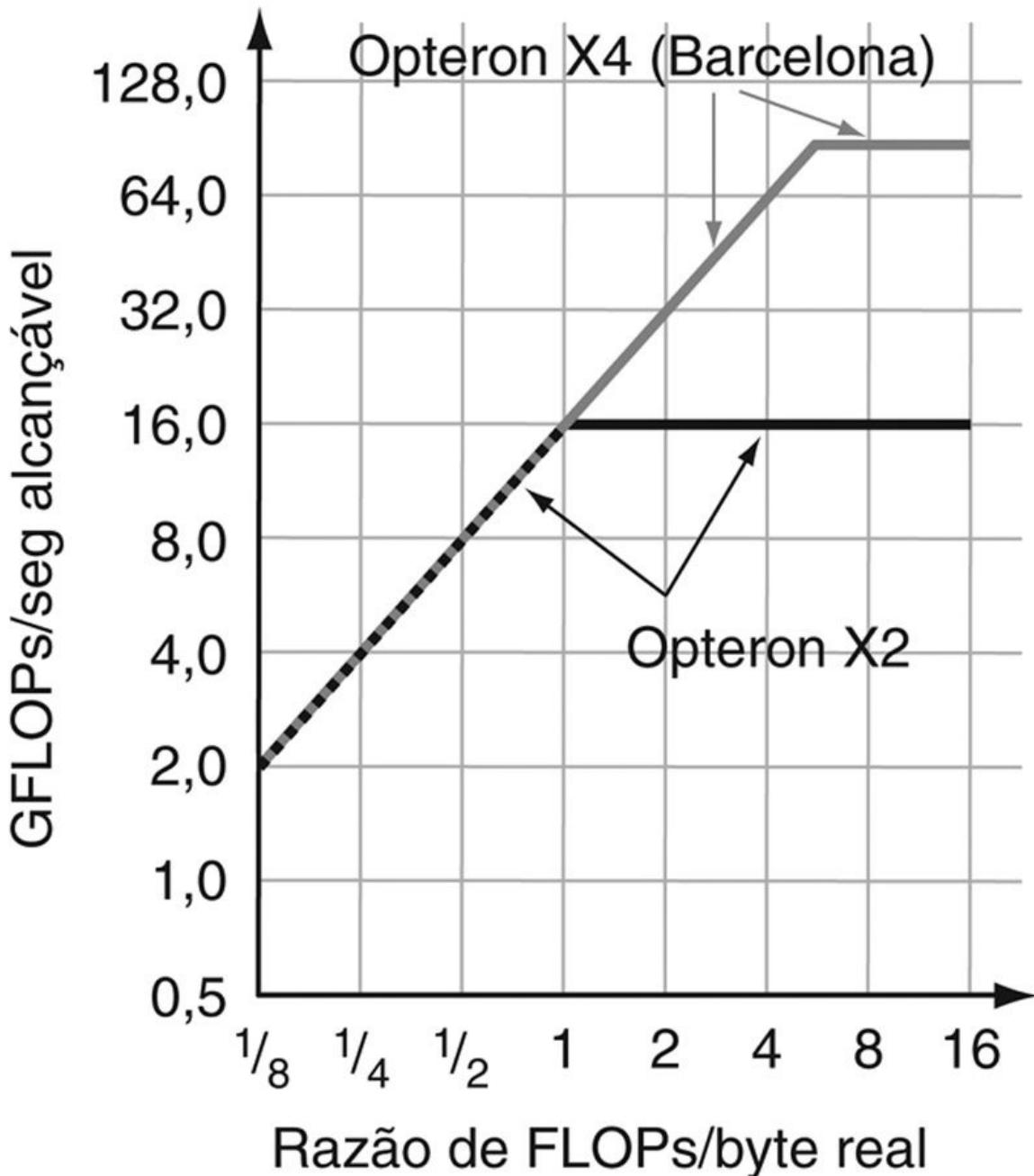


FIGURA 6.19 Modelos roofline de duas gerações de Opterons.

A roofline do Opteron X2, que é a mesma que na [Figura 6.18](#), está em preto, e a roofline do Opteron X4 está em cinza claro. O ponto de cumeeira mais alto do Opteron X4 significa que os kernels que eram computacionalmente limitados no Opteron X2 poderiam ser limitados pelo desempenho da memória no Opteron X4.

O modelo roofline oferece um limite superior para o desempenho. Suponha

que seu programa esteja muito abaixo desse limite. Que otimizações você deverá realizar, e em que ordem?

Para reduzir os gargalos computacionais, as duas otimizações a seguir podem ajudar a quase todo kernel:

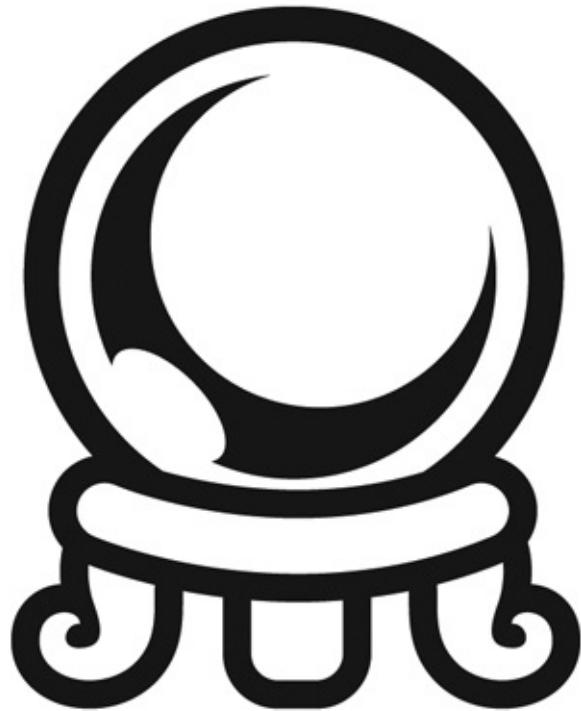


PARALELISMO

1. *Mix de operações de ponto flutuante.* O desempenho de pico em ponto flutuante para um computador normalmente exige um número igual de adições e multiplicações quase simultâneas. Esse equilíbrio é necessário ou porque o computador admite uma instrução multiplicação-adição unificada (veja a seção *Detalhamento* na Seção “Aritmética de precisão”, no [Capítulo 3](#)) ou porque a unidade de ponto flutuante tem um número igual de somadores de ponto flutuante e multiplicadores de ponto flutuante. O melhor desempenho também requer que uma fração significativa do mix de instruções seja de operações de ponto flutuante, e não instruções de inteiros.
2. *Melhore o paralelismo em nível de instrução e aplique SIMD.* Para arquiteturas superescalares, o desempenho mais alto surge com a busca, execução e comprometimento de três a quatro instruções por ciclo de clock ([Seção 4.10](#)). O objetivo aqui é melhorar o código do compilador para aumentar o ILP. Uma forma é desdobrando loops. Para as arquiteturas x86,

uma única instrução AVX pode operar sobre operandos de precisão dupla, de modo que elas devem ser usadas sempre que possível (Seções 3.7 e 3.8).

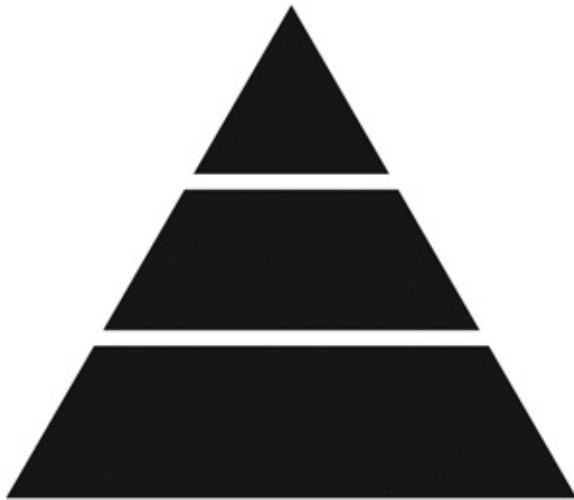
Para reduzir os gargalos da memória, as duas otimizações a seguir podem ajudar:



P R E D I Ç Ã O

1. *Pré-busca do software.* Normalmente, o desempenho mais alto exige manter muitas operações da memória no ato, que é mais fácil de se fazer executando acessos de **predição** via instruções de pré-busca do software, em vez de esperar até que os dados sejam exigidos pela computação.
2. *Afinidade de memória.* A maioria dos microprocessadores de hoje inclui um controlador de memória no mesmo chip com o microprocessador, o que melhora o desempenho da **hierarquia de memória**. Se o sistema tiver múltiplos chips, isso significa que os mesmos endereços vão para a DRAM que é local a um chip, e o restante requer que os acessos pela interconexão do chip acessem a DRAM que é local a outro chip. Essa divisão resulta em

acessos não uniformes à memória, que descrevemos na [Seção 6.5](#). O acesso à memória através de outro chip reduz o desempenho. Essa segunda otimização tenta alocar dados e as threads encarregadas de operar sobre esses dados no mesmo par memória-processador, de modo que os processadores raramente precisam acessar a memória dos outros chips.



HIERARQUIA

O modelo roofline pode ajudar a decidir quais dessas otimizações serão realizadas e em que ordem. Podemos pensar em cada uma dessas otimizações como um “teto” abaixo da roofline apropriada, significando que você não pode ultrapassar um teto sem realizar a otimização associada.

A roofline computacional pode ser encontrada nos manuais, e a roofline de memória pode ser encontrada executando-se o benchmark Stream. Os tetos computacionais, como o equilíbrio de ponto flutuante, também vêm dos manuais desse computador. O teto de memória, como a afinidade de memória, exige a execução de experimentos em cada computador, para determinar a lacuna entre eles. A boa notícia é que esse processo só precisa ser feito uma vez por computador, pois quando alguém caracterizar os tetos de um computador, todos poderão usar os resultados a fim de priorizar suas otimizações para esse computador.

A [Figura 6.20](#) acrescenta tetos ao modelo roofline da [Figura 6.18](#), mostrando os tetos computacionais no gráfico superior e os tetos da largura de banda de

memória no gráfico inferior. Embora os tetos mais altos não sejam rotulados com as duas otimizações, isso está implícito nessa figura; para ultrapassar o teto mais alto, você já deverá ter ultrapassado todos os tetos abaixo.

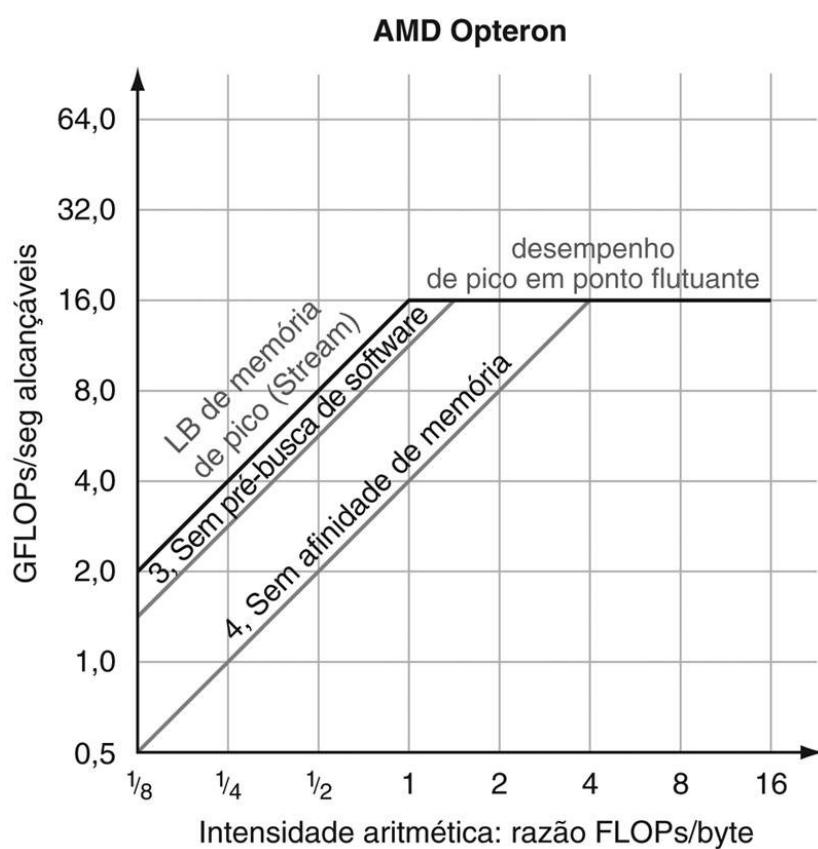
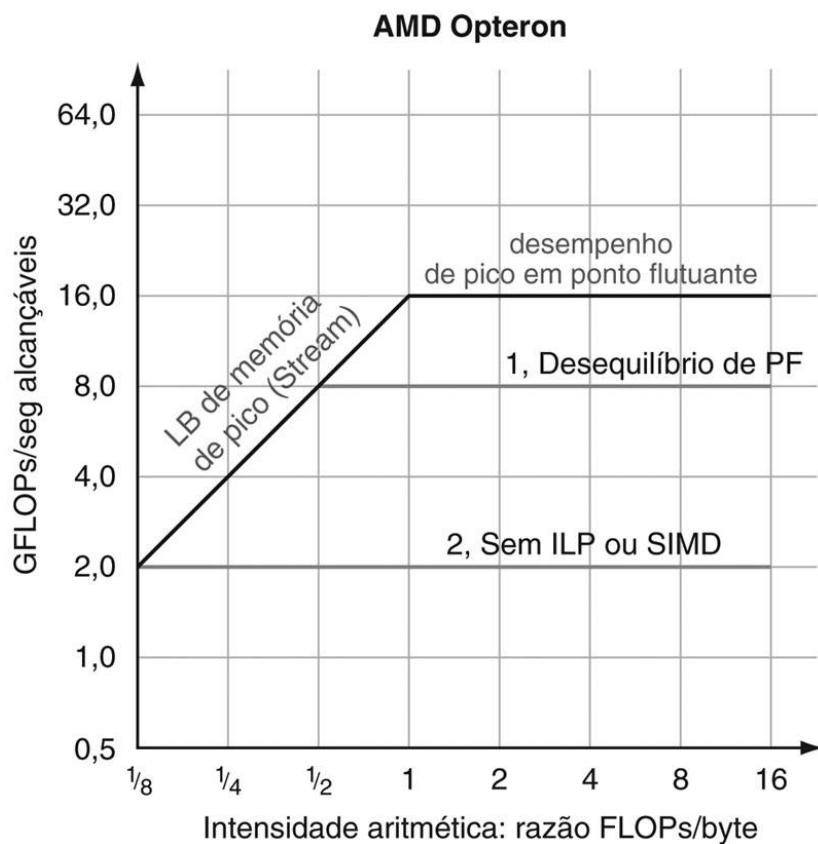
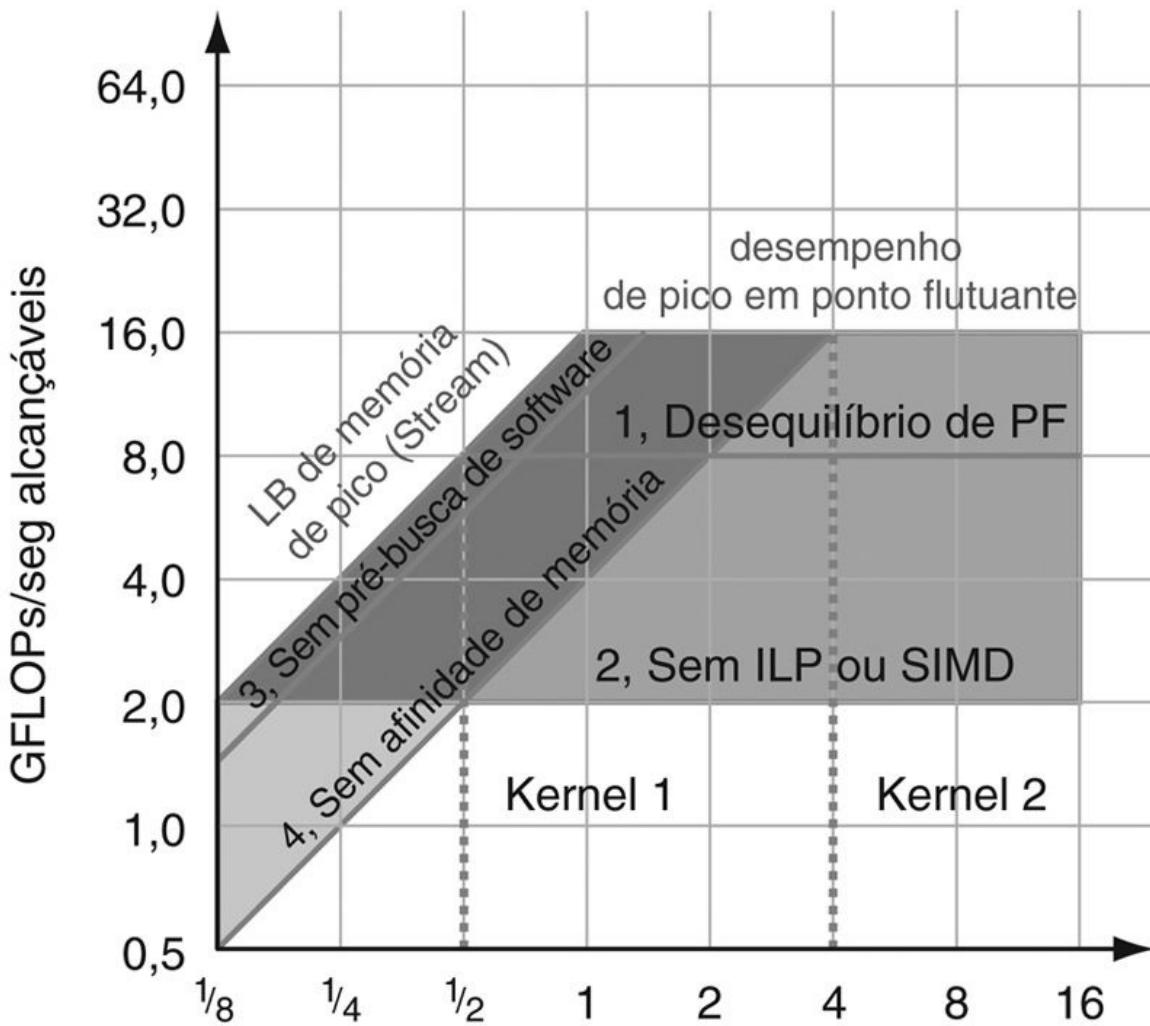


FIGURA 6.20 Modelo roofline com tetos.

O gráfico superior mostra os “tetos” computacionais de 8 GFLOPs/seg, se o mix de operações de ponto flutuante estiver desequilibrado e 2 GFLOPs/seg, se as otimizações para aumentar o ILP e o SIMD também estiverem faltando. O gráfico inferior mostra os tetos de largura de banda da memória de 11 GB/seg sem pré-busca de software e 4,8 GB/seg, se as otimizações de afinidade de memória também estiverem faltando.

A espessura da lacuna entre o teto e o próximo limite mais alto é a recompensa por tentar essa otimização. Assim, a Figura 6.20 sugere que a otimização 2, que melhora o ILP, tem um grande benefício para melhorar a computação nesse computador, e a otimização 4, que melhora a afinidade de memória, tem um grande benefício para melhorar a largura de banda da memória nesse computador.

A Figura 6.21 combina os tetos da Figura 6.20 em um único gráfico. A intensidade aritmética de um kernel determina a região de otimização, que, por sua vez, sugere quais otimizações tentar. Observe que as otimizações computacionais e as otimizações de largura de banda da memória se sobrepõem para grande parte da intensidade aritmética. Três regiões são sombreadas de formas diferentes na Figura 6.21 para indicar as diferentes estratégias de otimização. Por exemplo, o Kernel 2 cai no trapezoide cinza claro à direita, que sugere trabalhar apenas nas otimizações computacionais. O Kernel 1 cai no paralelogramo cinza escuro no meio, que sugere tentar os dois tipos de otimização. Além do mais, ele sugere começar com as otimizações 2 e 4. Observe que as linhas verticais do Kernel 1 caem abaixo da otimização de desequilíbrio de ponto flutuante, de modo que a otimização 1 pode ser desnecessária. Se um kernel caísse no triângulo cinza no canto inferior esquerdo, isso sugeriria tentar apenas otimizações de memória.



Intensidade aritmética: razão FLOPs/byte

FIGURA 6.21 Modelo roofline com tetos, áreas sobrepostas sombreadas e os dois kernels da [Figura 6.18](#).

Os kernels cuja intensidade aritmética se encontra no trapezoide azul claro à direita deverão focalizar otimizações de computação, e os kernels cuja intensidade aritmética se encontra no triângulo cinza no canto inferior esquerdo devem focalizar otimizações de largura de banda de memória. Aqueles que se encontram no paralelogramo cinza escuro no meio precisam se preocupar com ambos. Quando o Kernel 1 cai no paralelogramo do meio, tente otimizar ILP e SIMD, afinidade de memória e pré-busca de software. O Kernel 2 cai no trapezoide à direita; portanto, tente otimizar ILP e SIMD e o equilíbrio das operações de ponto flutuante.

Até aqui, estivemos supondo que a intensidade aritmética é fixa, mas esse não

é realmente o caso. Primeiro, existem kernels cuja intensidade aritmética aumenta com o tamanho do problema, como para os problemas Matriz Densidade e N-body ([Figura 6.17](#)). Na realidade, esse pode ser o motivo para os programadores terem mais sucesso com a expansão fraca do que com a expansão forte. Segundo, a eficácia da **hierarquia de memória** afeta o número de acessos que vão para a memória, de modo que as otimizações que melhoram o desempenho da cache também melhoram a intensidade aritmética. Um exemplo é melhorar a localidade temporal desdobrando loops e depois agrupando instruções com endereços semelhantes. Muitos computadores possuem instruções de cache especiais, que alocam dados em uma cache, mas não preenchem primeiro os dados da memória nesse endereço, pois eles logo serão modificados. Essas duas otimizações reduzem o tráfego da memória, movendo assim o poste da intensidade aritmética para a direita por um fator de, digamos, 1,5. Esse deslocamento para a direita poderia colocar o kernel em uma região de otimização diferente.



HIERARQUIA

Embora os exemplos anteriores mostrem como ajudar os programadores a melhorarem o desempenho, o modelo também pode ser usado por arquitetos para decidir onde eles otimizariam o hardware para melhorar o desempenho dos kernels que acreditam que serão importantes.

A próxima seção usa o modelo roofline para demonstrar a diferença de

desempenho entre um microprocessador multicore e uma GPU, e para ver se essas diferenças refletem o desempenho de programas reais.

Detalhamento

Os tetos são ordenados de modo que os mais baixos são mais fáceis de otimizar. Logicamente, um programador pode otimizar em qualquer ordem, mas ter essa sequência reduz as chances de desperdiçar esforço em uma otimização que não possui benefício devido a outras restrições. Assim como no modelo 3Cs, desde que o modelo roofline possa gerar esclarecimentos, um modelo pode ter suposições que provam ser otimistas. Por exemplo, o modelo supõe que o programa tem balanceamento de carga entre todos os processadores.

Detalhamento

Uma alternativa ao benchmark Stream é usar a largura de banda bruta da DRAM como roofline. Enquanto a largura de banda bruta definitivamente é um limite superior rígido, o desempenho real da memória normalmente está tão distante desse limite que não é tão útil como um limite superior. Ou seja, nenhum programa pode chegar perto desse limite. A desvantagem de usar o Stream é que uma programação muito cuidadosa pode exceder os resultados do Stream, de modo que a roofline da memória pode não ser um limite tão rígido quanto a roofline computacional. Ficamos com o Stream porque menos programadores serão capazes de oferecer mais largura de banda de memória do que o Stream descobre.

Detalhamento

Embora o modelo roofline apresentado seja para processadores multicores, ele certamente também funcionaria para um uniprocessador.

Verifique você mesmo

Verdadeiro ou falso: a principal desvantagem com as técnicas convencionais de benchmarks para computadores paralelos é que as regras que garantem imparcialidade também suprimem a inovação do software.

6.10. Vida real: benchmarking e rooflines do Intel Core i7 960 e GPU NVIDIA Tesla

Um grupo de pesquisadores da Intel publicou um artigo (Lee et al., 2010) comparando um Intel Core i7 960 quad-core com extensões SIMD de multimídia com a GPU da geração anterior, o NVIDIA Tesla GTX 280. A [Figura 6.22](#) lista as características dos dois sistemas. Os dois produtos foram comprados no segundo semestre de 2009. O Core i7 está na tecnologia de semicondutores de 45 nanômetros da Intel, enquanto a GPU está na tecnologia de 65 nanômetros da TSMC. Embora pudesse ter sido mais justo a comparação ser feita por um terceiro agente ou por ambas as partes interessadas, o propósito desta seção *não* é determinar o quanto mais rápido um produto é em relação ao outro, mas tentar entender o valor relativo dos recursos desses dois estilos de arquitetura contrastantes.

	Core i7-960	GTX 280	GTX 480	Ratio 280/i7	Ratio 480/i7
Número de elementos de processamento (núcleos ou SMs)	4	30	15	7,5	3,8
Frequência de clock (GHz)	3,2	1,3	1,4	0,41	0,44
Tamanho do die	263	576	520	2,2	2,0
Tecnologia	Intel 45 nm	TSMC 65 nm	TSMC 40 nm	1,6	1,0
Potência (chip, não módulo)	130	130	167	1,0	1,3
Transistores	700 M	1400 M	3030 M	2,0	4,4
Largura de banda da memória (GBytes/seg)	32	141	177	4,4	5,5
Largura SIMD precisão simples	4	8	32	2,0	8,0
Largura SIMD precisão dupla	2	1	16	0,5	8,0
FLOPs escalar de pico em precisão simples (GFLOP/seg)	26	117	63	4,6	2,5
FLOPs SIMD de pico em precisão simples (GFLOP/seg)	102	311 a 933	515 ou 1344	3,0–9,1	6,6–13,1
(SP 1 adição ou multiplicação)	N.A.	(311)	(515)	(3,0)	(6,6)
(SP 1 instrução combinada de multiplicação)	N.A.	(622)	(1344)	(6,1)	(13,1)
(SP raro despacho dual combinado de multiplicação e multiplicação)	N.A.	(933)	N.A.	(9,1)	–
Peal FLOPS SIMD de precisão dupla (GFLOP/seg)	51	78	515	1,5	10,1

FIGURA 6.22 Especificações do Intel Core i7-960, NVIDIA GTX 280 e GTX 480.

As duas colunas da direita mostram as razões entre o Tesla GTX 280 e do Fermi GTX 480 e o Core i7. Embora o estudo de caso seja entre o Tesla 280 e o i7, incluímos o Fermi 480 para mostrar sua relação com o Tesla 280, pois ele foi descrito neste capítulo. Observe que essas larguras de banda de memória são mais altas do que na [Figura 6.23](#), pois estas são larguras de banda

nos pinos da DRAM, e as da [Figura 6.23](#) são nos processadores, conforme medidas por um programa de benchmark. (Da Tabela 2 em Lee et al. [2010]).

As rooflines do Core i7 960 e do GTX 280 na [Figura 6.23](#) ilustram as diferenças nos computadores. Não apenas o GTX 280 possui muito mais largura de banda de memória e desempenho de ponto flutuante em precisão dupla, mas também seu ponto de cumeeira de precisão dupla se localiza consideravelmente à esquerda. O ponto de cumeeira de precisão dupla é 0,6 para o GTX 280, contra 3,1 para o Core i7. Como já dissemos, quanto mais distante da esquerda estiver o ponto de cumeeira da roofline, mais fácil será atingir o desempenho computacional de pico. Para o desempenho em precisão simples, o ponto de cumeeira se move mais para a direita nos dois computadores, de modo que é muito mais difícil atingir o telhado do desempenho em precisão simples. Observe que, a intensidade aritmética do kernel é baseada nos bytes que vão para a memória principal, e não os bytes que vão para a memória cache. Assim, como já dissemos, o caching pode alterar a intensidade aritmética de um kernel em determinado computador, se a maioria das referências realmente forem para a cache. Observe também que essa largura de banda é para acessos com stride unitário nas duas arquiteturas. Os endereços gather-scatter reais podem ser mais baixos no GTX 280 e no Core i7, conforme veremos.

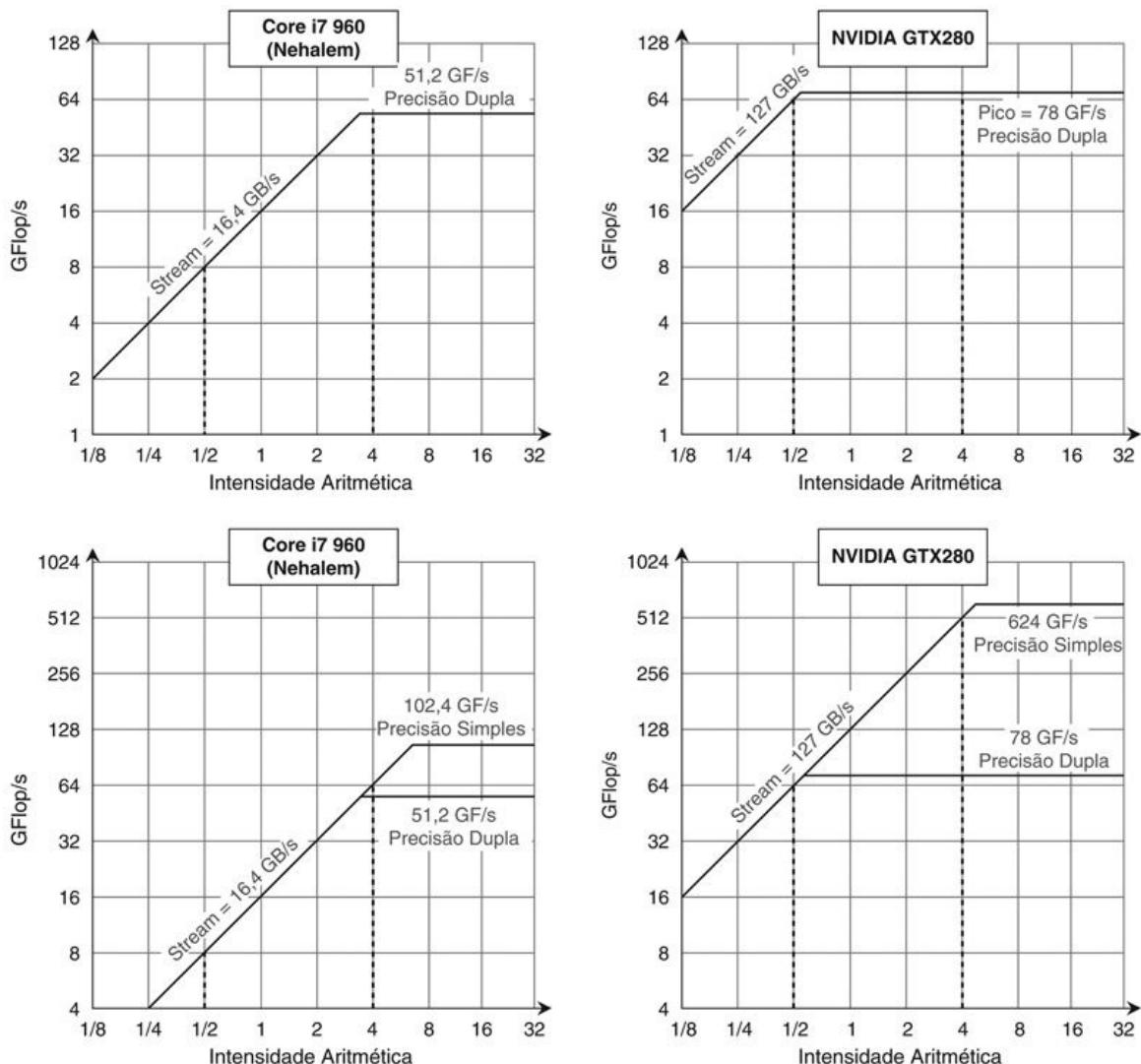


FIGURA 6.23 Modelo roofline (Williams, Waterman e Patterson, 2009).

Essas rooflines mostram o desempenho de ponto flutuante, em precisão dupla na fileira superior e o desempenho em precisão simples na fileira inferior. (O teto do desempenho de PF em PD também está na fileira inferior, por comparação.) O Core i7 960 à esquerda tem um desempenho de PF em PD de 51,2 GFLOP/seg, um pico de PF em OS de 102,4 GFLOP/seg e uma largura de banda de memória de pico de 16,4 GBytes/seg. O NVIDIA GTX 280 tem um pico de PF em PD de 78 GFLOP/seg, pico de PF em PS de 624 GFLOP/seg e 127 GBytes/seg de largura de banda da memória. A linha vertical tracejada à esquerda representa uma intensidade aritmética de 0,5 FLOP/byte. Ela é limitada pela largura de banda da memória a não mais que 8 PD GFLOP/seg ou 8 PS GFLOP/seg no Core i7. A linha vertical tracejada à direita tem uma intensidade

aritmética de 4 FLOP/byte. Ela é limitada apenas computacionalmente a 51,2 PD GFLOP/seg e 102,4 PS GFLOP/seg no Core i7 e 78 PD GFLOP/seg e 512 PS GFLOP/seg no GTX 280. para alcançar a taxa de computação mais alta no Core i7, você precisa usar todos os 4 núcleos e instruções SSE com o mesmo número de multiplicações e adições. Para o GTX 280, você precisa usar instruções combinadas de multiplicação-adição em todos os processadores SIMD multithreaded.

Os pesquisadores selecionaram os programas de benchmark analisando as características computacionais e de memória de quatro pacotes de benchmark propostos recentemente, e então “formularam o conjunto de *kernels de cálculo da vazão*, que capturam essas características”. A [Figura 6.24](#) mostra os resultados do desempenho, com números maiores significando mais rápido. As rooflines ajudam a explicar o desempenho relativo nesse estudo de caso.

Kernel	Unidades	Core i7-960	GTX 280	GTX 280/ i7-960
SGEMM	GFLOP/seg	94	364	3,9
MC	Bilhões de caminhos/seg	0,8	1,4	1,8
Conv	Milhões de pixels/seg	1250	3500	2,8
FFT	GFLOP/seg	71,4	213	3,0
SAXPY	GBytes/seg	16,8	88,8	5,3
LBM	Milhões de pesquisas/seg	85	426	5,0
Solv	Quadros/seg	103	52	0,5
SpMV	GFLOP/seg	4,9	9,1	1,9
GJK	Quadros/seg	67	1020	15,2
Sort	Milhões de elementos/seg	250	198	0,8
RC	Quadros/seg	5	8,1	1,6
Search	Milhões de consultas/seg	50	90	1,8
Hist	Milhões de pixels/seg	1517	2583	1,7
Bilat	Milhões de pixels/seg	83	475	5,7

FIGURA 6.24 Desempenho bruto e relativo medido para as duas plataformas.

Neste estudo, SAXPY é usado simplesmente como uma medida de largura de banda da memória, de modo que a unidade da direita é GBytes/seg e não GFLOP/seg. (Baseado na Tabela 3 em [Lee et al., 2010].)

Dado que as especificações brutas de desempenho do GTX 280 variam de 2,5× mais lentas (taxa de clock) a 7,5× mais rápidas (núcleos por chip), enquanto o desempenho varia de 2,0× mais lento (Solv) até 15,2× mais rápido (GJK), os pesquisadores da Intel decidiram encontrar os motivos para as diferenças:

- *Largura de banda da memória.* A GPU tem 4,4× a largura de banda da memória, o que ajuda a explicar por que LBM e SAXPY são executados 5,0 e 5,3× mais rápido; seus conjuntos de trabalho são centenas de megabytes e, portanto, não cabem na cache do Core i7. (Logo, quanto ao acesso intensivo à memória, eles propositalmente não usaram o bloqueio de cache, como no Capítulo 5.) Logo, a inclinação das rooflines explicam seu desempenho.

SpMV também possui um grande conjunto de trabalho, mas só executa 1,9× mais rápido, pois o ponto flutuante de precisão dupla do GTX 280 é apenas 1,5× mais rápido que o Core i7.

- *Largura de banda de cálculo.* Cinco dos kernels restantes são limitados pelo cálculo: SGEMM, Conv, FFT, MC e Bilat. O GTX é mais rápido por 3,9, 2,8, 3,0, 1,8 e 5,7 × , respectivamente. Os três primeiros utilizam aritmética de ponto flutuante com precisão simples, e a precisão simples do GTX 280 é de 3 a 6× mais rápida. MC usa precisão dupla, o que explica por que ele é apenas 1,8× mais rápido, pois o desempenho em PD é apenas 1,5× mais rápido. Bilat usa funções transcendentais, para as quais o GTX 280 tem suporte direto. O Core i7 gasta dois terços de seu tempo calculando funções transcendentais para o Bilat, de modo que o GTX 280 é 5,7× mais rápido. Essa observação ajuda a explicar o valor do suporte do hardware para operações que ocorrem na sua carga de trabalho: ponto flutuante com precisão dupla e talvez até mesmo transcendentais.
- *Benefícios da cache.* *Ray casting* (RC) é apenas 1,6× mais rápido no GTX, porque o bloqueio da cache com as caches do Core i7 impede que ele se torne limitado pela largura de banda da memória (Seções 5.4 e 5.14), como nas GPUs. O bloqueio de cache também pode ajudar no Search. Se as árvores de índice forem pequenas, de modo que caibam na cache, o Core i7 terá o dobro da velocidade. Árvores de índice maiores os tornam limitados pela largura de banda da memória. Em geral, o GTX 280 executa a busca 1,8× mais rápido. O bloqueio de cache também ajuda no Sort. Embora a maioria dos programadores não executaria o Sort em um processador SIMD, ele pode ser escrito com uma primitiva Sort de 1 bit, chamada *split*. Porém, o algoritmo *split* executa muito mais instruções do que uma ordenação escalar. Como resultado, o Core i7 executa 1,25× mais rápido que o GTX 280. Observe que as caches também ajudam outros kernels no Core i7, pois o bloqueio de cache permite que SGEMM, FFT e SpMV se tornem limitados pelo cálculo. Essa observação enfatiza novamente a importância das otimizações por bloqueio de cache do [Capítulo 5](#).
- *Gather-Scatter.* As extensões SIMD de multimídia ajudam pouco se os dados estiverem espalhados pela memória principal; o desempenho ideal vem somente quando os acessos aos dados são alinhados em limites de 16 bytes. Assim, GJK tem pouco benefício com o SIMD no Core i7. Como já dissemos, as GPUs oferecem endereçamento *gather-scatter*, que é encontrado em uma arquitetura de vetor, mas não aparece na maioria das

extensões SIMD. O controlador de memória até mesmo reúne os acessos na mesma página da DRAM em batch ([Seção 5.2](#)). Essa combinação significa que o GTX 280 roda o GJK com surpreendentes 15,2× mais rápido do que o Core i7, o que é maior do que qualquer parâmetro físico isolado na [Figura 6.22](#). Esta observação reforça a importância do *gather-scatter* para as arquiteturas de vetor e GPU, que não existe nas extensões SIMD.

- *Sincronização*. O desempenho da sincronização é limitado pelas atualizações atômicas, que são responsáveis por 28% do runtime total no Core i7, mesmo tendo uma instrução de busca e incremento no hardware. Assim, *Hist* é apenas 1,7× mais rápido no GTX 280. *Solv* resolve um lote de restrições independentes em uma pequena quantidade de cálculo, seguida por sincronização de barreira. O Core i7 se beneficia das instruções atômicas e um modelo de consistência de memória que garante resultados corretos, mesmo que nem todos os acessos anteriores à hierarquia de memória tenham sido concluídos. Sem o modelo de consistência de memória, a versão do GTX 280 dispara alguns batches a partir do processador do sistema, o que leva ao GTX 280 rodando com a metade da velocidade do Core i7. Essa observação explica como o desempenho da sincronização pode ser importante para alguns problemas de paralelismo de dados.

É surpreendente a frequência com que os pontos fracos no Tesla GTX 280, que foram desvendados pelos kernels selecionados pelos pesquisadores da Intel, já haviam sido tratados na arquitetura sucessora do Tesla: Fermi tem desempenho mais rápido em ponto flutuante com precisão dupla, operações atômicas mais rápidas e caches. Também foi interessante que o suporte para *gather-scatter* das arquiteturas de vetor, décadas antes das instruções SIMD, foi tão importante para a utilidade efetiva dessas extensões SIMD, que alguns já haviam previsto antes da comparação. Os pesquisadores da Intel observaram que 6 dos 14 kernels explorariam o SIMD melhor com o suporte para *gather-scatter* mais eficiente no Core i7. Esse estudo certamente estabelece a importância também do bloqueio de cache.

Agora que vimos uma grande gama de resultados de benchmarking com diferentes multiprocessadores, vamos voltar ao nosso exemplo do DGEMM, para ver com detalhes o quanto temos que mudar o código C para tirar proveito dos múltiplos processadores.

6.11. Mais rápido: processadores múltiplos e

multiplicação matricial

Esta seção é a última e maior etapa em nossa jornada pelo desempenho incremental da adaptação do DGEMM ao hardware subjacente do Intel Core i7 (Sandy Bridge). Cada Core i7 possui 8 núcleos, e o computador que usamos possui 2 Core i7s. Assim, temos 16 núcleos para executar o DGEMM.

A [Figura 6.25](#) mostra a versão OpenMP do DGEMM, que utiliza esses núcleos. Observe que a linha 30 é a *única* linha acrescentada à [Figura 5.48](#) para fazer com que esse código seja executado em múltiplos processadores: uma pragma OpenMP que diz ao compilador para usar várias threads no loop for mais externo. Ela diz ao computador para espalhar o trabalho do loop mais externo por todas as threads.

```

1 #include <x86intrin.h>
2 #define UNROLL (4)
3 #define BLOCKSIZE 32
4 void do_block (int n, int si, int sj, int sk,
5                 double *A, double *B, double *C)
6 {
7     for ( int i = si; i < si+BLOCKSIZE; i+=UNROLL*4 )
8         for ( int j = sj; j < sj+BLOCKSIZE; j++ ) {
9             __m256d c[4];
10            for ( int x = 0; x < UNROLL; x++ )
11                c[x] = _mm256_load_pd(C+i+x*4+j*n);
12                /* c[x] = C[i][j] */
13                for( int k = sk; k < sk+BLOCKSIZE; k++ )
14                {
15                    __m256d b = _mm256_broadcast_sd(B+k+j*n);
16                    /* b = B[k][j] */
17                    for (int x = 0; x < UNROLL; x++)
18                        c[x] = _mm256_add_pd(c[x], /* c[x]+=A[i][k]*b */
19                                              _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
20                }
21
22            for ( int x = 0; x < UNROLL; x++ )
23                _mm256_store_pd(C+i+x*4+j*n, c[x]);
24                /* C[i][j] = c[x] */
25        }
26    }
27
28 void dgemm (int n, double* A, double* B, double* C)
29 {
30 #pragma omp parallel for
31     for ( int sj = 0; sj < n; sj += BLOCKSIZE )
32         for ( int si = 0; si < n; si += BLOCKSIZE )
33             for ( int sk = 0; sk < n; sk += BLOCKSIZE )
34                 do_block(n, si, sj, sk, A, B, C);
35 }

```

FIGURA 6.25 Versão OpenMP do DGEMM da Figura 5.48.

A linha 30 é o único código OpenMP, fazendo com que o loop for mais externo opere em paralelo. Essa linha é a única diferença da [Figura 5.48](#).

A [Figura 6.26](#) representa um gráfico de speed-up de multiprocessador clássico, mostrando a melhoria de desempenho contra uma única thread à medida que o número de threads aumenta. Esse gráfico facilita a visão dos desafios da expansão forte *versus* a expansão fraca. Quando tudo cabe na cache de dados de primeiro nível, como acontece para as matrizes 32×32 , a inclusão de threads, na verdade, prejudica o desempenho. A versão de DGEMM para 16

threads leva quase o dobro do tempo da versão de única thread, neste caso. Ao contrário, as duas maiores matrizes obtêm um ganho de velocidade de $14\times$ usando 16 threads, daí as duas linhas clássicas “acima e à direita” na Figura 6.26.

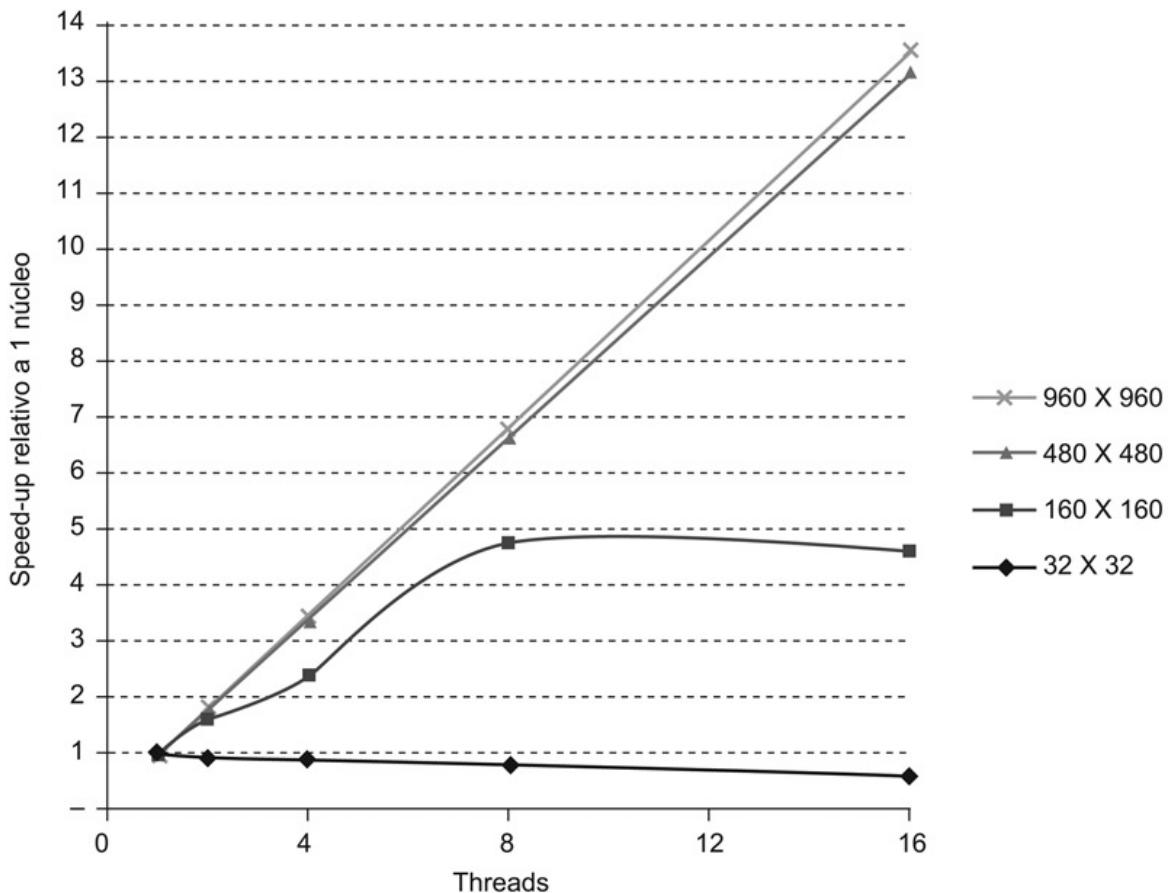


FIGURA 6.26 Melhorias de desempenho relativas a uma única thread à medida que o número de threads aumenta.
A forma mais honesta de apresentar esses gráficos é tornar o desempenho relativo à melhor versão de um programa de único processador, como fizemos aqui. Esse gráfico é relativo ao desempenho do código na Figura 5.48 sem incluir pragmas OpenMP.

A Figura 6.27 mostra o aumento de desempenho absoluto enquanto aumentamos o número de threads de 1 para 16. O DGEMM agora opera a 174 GFLOPs para matrizes de 960×960 . Como a nossa versão C não otimizada do DGEMM da Figura 3.21 executou esse código em apenas 0,8 GFLOPs, as otimizações nos Capítulos de 3 a 6, que ajustam o código ao hardware

subjacente, resultam em um ganho de velocidade de mais de 200 vezes!

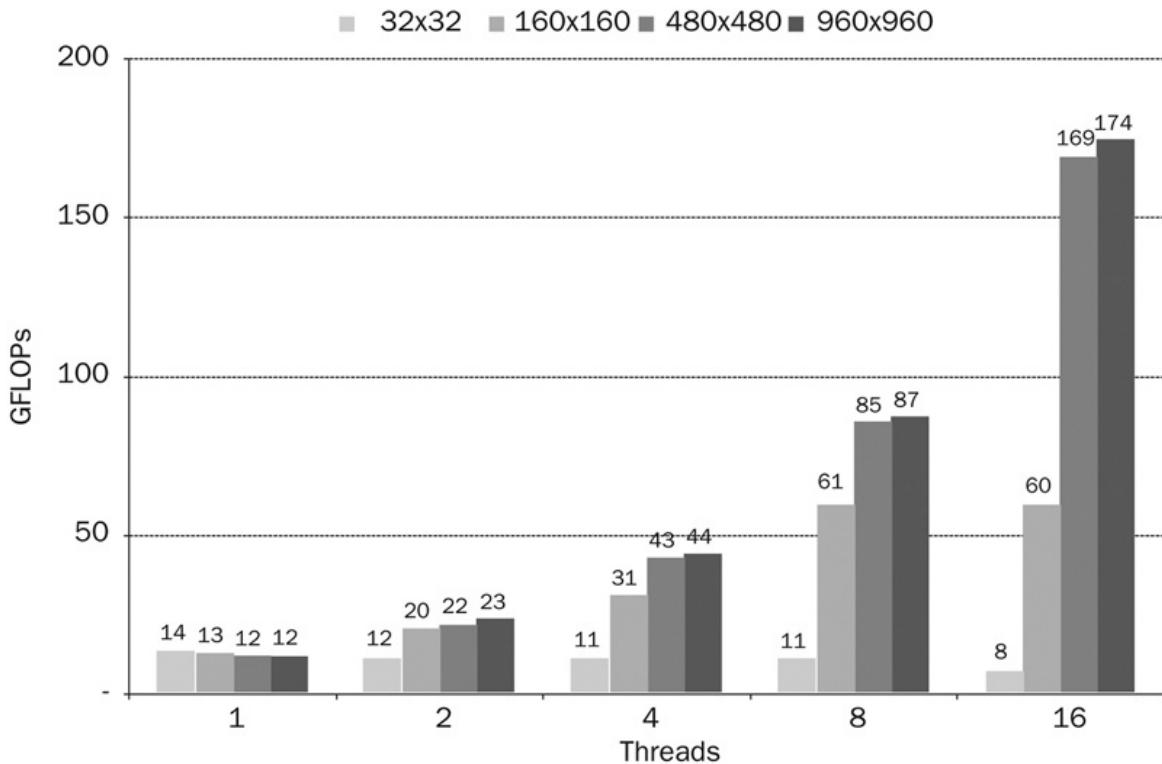


FIGURA 6.27 Desempenho do DGEMM contra o número de threads para quatro tamanhos de matriz.

A melhoria de desempenho em comparação com o código não otimizado da [Figura 3.21](#) para a matriz de 960×960 com 16 threads é surpreendente: 212 vezes mais rápido!

A seguir, veremos nossos avisos das falácia e armadilhas do multiprocessamento. O cemitério da arquitetura de computador está repleto de projetos de processamento paralelo que as ignoraram.

Detalhamento

Esses resultados acontecem com o modo Turbo desligado. Estamos usando um sistema de chip dual neste caso, de modo que não é surpresa que obtenhamos um ganho de velocidade total no Turbo ($3,3/2,6 = 1,27$) com 1 thread (apenas 1 núcleo em um dos chips) ou 2 threads (1 núcleo por chip). Ao aumentarmos o número de threads e, portanto, o número de núcleos ativos, o benefício do modo Turbo diminui, pois há menos potência a ser

gasta sobre os núcleos ativos. Para 4 threads, o ganho de velocidade médio do Turbo é 1,23, para 8 é 1,13 e para 16 é 1,11.

Detalhamento

Embora o Sandy Bridge tenha suporte para duas threads de hardware por núcleo, não conseguimos mais desempenho com 32 threads. O motivo é que um único hardware AVX é compartilhado entre as duas threads multiplexadas para um núcleo, de modo que a atribuição de duas threads por núcleo na verdade prejudica o desempenho, devido ao overhead da multiplexação.

6.12. Falácia e armadilhas

Por mais de uma década os analistas anunciam que a organização de um único computador alcançou seus limites e que avanços verdadeiramente significantes só podem ser feitos pela interconexão de uma multiplicidade de computadores de tal modo que permita solução cooperativa... Demonstrou-se a continuada validade do método de processador único...

Gene Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", Spring Joint Computer Conference, 1967

Os muitos ataques ao processamento paralelo revelaram inúmeras falácias e armadilhas. Veremos quatro delas aqui.

Falácia: a Lei de Amdahl não se aplica aos computadores paralelos.

Em 1987, o diretor de uma organização de pesquisa afirmou que a Lei de Amdahl tinha sido quebrada por uma máquina de multiprocessador. Para tentar entender a base dos relatos da mídia, vejamos a citação que nos trouxe a Lei de Amdahl [1967, p. 483]:

Uma conclusão bastante óbvia que pode ser tirada nesse momento é que o esforço despendido em conseguir altas velocidades de processamento paralelo é desperdiçado se não for acompanhado de conquistas de mesmas proporções nas velocidades de processamento

sequencial.

Essa afirmação ainda deve ser verdadeira; a parte ignorada do programa deve limitar o desempenho. Uma interpretação da lei leva ao seguinte princípio: partes de cada programa precisam ser sequenciais e, portanto, precisa haver um limite superior lucrativo para o número de processadores — digamos, 100. Mostrando speed-up linear com 1.000 processadores, esse princípio se torna falso e, então, a Lei de Amdahl foi quebrada.

O método dos pesquisadores foi simplesmente usar a expansão fraca: em vez de ir 1.000 vezes mais rápido, eles calcularam 1.000 vezes mais trabalho em tempo comparável. Para o algoritmo deles, a parte sequencial do programa era constante, independente do tamanho da entrada, e o restante era totalmente paralelo — daí, speed-up linear com 1.000 processadores.

Obviamente, a Lei de Amdahl se aplica aos processadores paralelos. O que essa pesquisa salienta é que um dos principais usos de computadores mais rápidos é executar problemas maiores. Basta ter certeza de que os usuários realmente se importam com esses problemas, em vez de ser uma justificativa para comprar um computador caro, simplesmente para manter muitos processadores ocupados.

Falácia: o desempenho de pico segue o desempenho observado.

A indústria de supercomputadores usou essa métrica no marketing, e a falácia é enfatizada com as máquinas paralelas. Não apenas os marqueteiros estão usando o desempenho de pico quase inatingível de um nó processador, como eles também o estão multiplicando pelo número total de processadores, considerando speed-up perfeito! A lei de Amdahl sugere como é difícil alcançar qualquer um desses picos; multiplicar os dois só multiplicará os pecados. O modelo roofline ajuda a entender melhor o desempenho de pico.

Armadilha: não desenvolver o software para tirar proveito de (ou otimizar) uma arquitetura de multiprocessador.

Há um longo histórico de software ficando para trás nos processadores paralelos, possivelmente porque os problemas do software são muito mais difíceis. Temos um exemplo para mostrar a sutileza dessas questões, mas existem muitos exemplos que poderíamos escolher!

Um problema encontrado com frequência ocorre quando o software projetado para um processador único é adaptado a um ambiente multiprocessador. Por

exemplo, o sistema operacional da Silicon Graphics protegia originalmente a tabela de página com um único lock, supondo que a alocação de página é pouco frequente. Em um processador único, isso não representa um problema de desempenho, mas em um multiprocessador, pode se tornar um gargalo de desempenho importante para alguns programas. Considere um programa que usa um grande número de páginas que são inicializadas quando começa a ser executado, o que o UNIX faz para as páginas alocadas estaticamente. Suponha que o programa seja colocado em paralelo, de modo que múltiplos processos aloquem as páginas. Como a alocação de página requer o uso da tabela de página, que é bloqueada sempre que está em uso, até mesmo um kernel do SO que permita múltiplas threads no SO será colocado em série se todos os processos tentarem alocar suas páginas ao mesmo tempo (que é exatamente o que poderíamos esperar no momento da inicialização!).

Essa serialização da tabela de página elimina o paralelismo na inicialização e tem um impacto significativo sobre o desempenho paralelo geral. Esse gargalo de desempenho persiste até mesmo para o paralelismo em nível de tarefa. Por exemplo, suponha que dividamos o programa de processamento paralelo em tarefas separadas e as executemos, uma tarefa por processador, de modo que não haja compartilhamento entre elas. É exatamente isso o que um usuário fez, pois acreditava que o problema de desempenho era devido ao compartilhamento não intencional ou interferência em sua aplicação. Infelizmente, o lock ainda coloca todas as tarefas em série — de modo que, até mesmo o desempenho da tarefa independente é fraco.

Essa armadilha indica os tipos de bugs de desempenho sutis, porém significativos, que podem surgir quando o software é executado em multiprocessadores. Assim como muitos outros componentes de software essenciais, os algoritmos do OS e as estruturas de dados precisam ser repensadas em um contexto de multiprocessador. Colocar locks em partes menores da tabela de página efetivamente elimina o problema.

Falácia: você pode obter um bom desempenho de vetor sem fornecer largura de banda de memória.

Como vimos com o modelo roofline, a largura de banda de memória é muito importante para todas as arquiteturas. DAXPY requer 1,5 referências de memória por operação de ponto flutuante, e essa razão é comum para muitos códigos científicos. Mesmo que as operações de ponto flutuante não levavsem

tempo algum, um Cray-1 não poderia aumentar o desempenho do DAXPY da sequência de vetor utilizada, pois ele era limitado pela memória. O desempenho do Cray-1 no Linpack saltou quando o compilador usou o bloqueio para alterar a computação de modo que os valores pudessem ser mantidos nos registradores de vetor. Essa técnica reduziu o número de referências à memória por FLOP e melhorou o desempenho por um fator de quase dois! Assim, a largura de banda de memória no Cray-1 tornou-se suficiente para um loop que anteriormente exigia mais largura de banda, o que é exatamente aquilo que o modelo roofline preveria.

6.13. Comentários finais

Estamos dedicando todo o nosso desenvolvimento de produto futuro aos projetos multicore. Acreditamos que esse seja um ponto de inflexão importante para a indústria. [...] Essa não é uma corrida, é uma mudança de mares na computação.

Paul Otellini, Presidente da Intel, Intel Developers Forum, 2004

O sonho de construir computadores apenas agregando processadores existe desde os primeiros dias da computação. No entanto, o progresso na construção e no uso de processadores paralelos eficientes tem sido lento. Essa velocidade de progresso foi limitada pelos difíceis problemas de software, bem como por um longo processo de evolução da arquitetura dos multiprocessadores para melhorar a usabilidade e a eficiência. Discutimos muitos dos problemas de software neste capítulo, incluindo a dificuldade de escrever programas que obtêm bom speed-up devido à Lei de Amdahl. A grande variedade de métodos arquitetônicos diferentes e o sucesso limitado e a vida curta de muitas arquiteturas até agora se juntam às dificuldades do software. Para obter ainda mais detalhes sobre os assuntos deste capítulo, consulte *Arquitetura de Computadores: Uma abordagem quantitativa, 5ª. Edição*: o Capítulo 4 explica mais sobre GPUs e tem comparações entre GPUs e CPUs, e o Capítulo 6 para mais sobre WSCs.

Como dissemos no Capítulo 1, apesar desse longo e sinuoso passado, a indústria da tecnologia de informação agora tem seu futuro ligado à computação paralela. Embora seja fácil apontar fatos para que esse esforço falhe como muitos no passado, existem motivos para termos esperança:

- Claramente, o *software como um serviço* (SaaS) está ganhando mais

importância, e os clusters provaram ser um modo muito bem-sucedido de oferecer tais serviços. Oferecendo redundância em um nível mais alto, incluindo centros de dados geograficamente distribuídos, esses serviços têm oferecido disponibilidade $24 \times 7 \times 365$ para os clientes no mundo inteiro.

- Acreditamos que os Computadores em Escala Warehouse (WSC) estejam mudando os objetivos e os princípios do projeto de servidor, assim como as necessidades de clientes móveis estão mudando os objetivos e os princípios do projeto de microprocessador. Ambos estão revolucionando também a indústria do software. O desempenho por dólar e o desempenho por joule impulsionam tanto o hardware de clientes móveis quanto o hardware de WSC, e o paralelismo é fundamental para oferecer esses conjuntos de objetivos.
- Operações SIMD e de vetor são uma boa combinação para aplicações multimídia, que estão desempenhando um papel maior na era pós-PC. Elas compartilham a vantagem de serem mais fáceis para o programador do que a programação MIMD paralela clássica e de serem mais eficientes que MIMD, em termos de energia. Para entender melhor a importância de SIMD versus MIMD, a [Figura 6.28](#) representa o número de núcleos para MIMD contra o número de operações de 32 e 64 bits por ciclo de clock no modo SIMD para computadores x86 no decorrer do tempo. Para os computadores x86, esperamos ver dois núcleos adicionais por chip a cada dois anos e a largura do SIMD dobrar aproximadamente a cada quatro anos. Dadas essas suposições, no decorrer da próxima década, o speed-up em potencial vindo do paralelismo SIMD é o dobro daquele do paralelismo MIMD. Dada a eficácia do SIMD para multimídia e sua importância crescente na era pós-PC, essa ênfase pode ser apropriada. Logo, é pelo menos tão importante compreender o paralelismo SIMD quanto o paralelismo MIMD, embora esse último tenha recebido muito mais atenção.

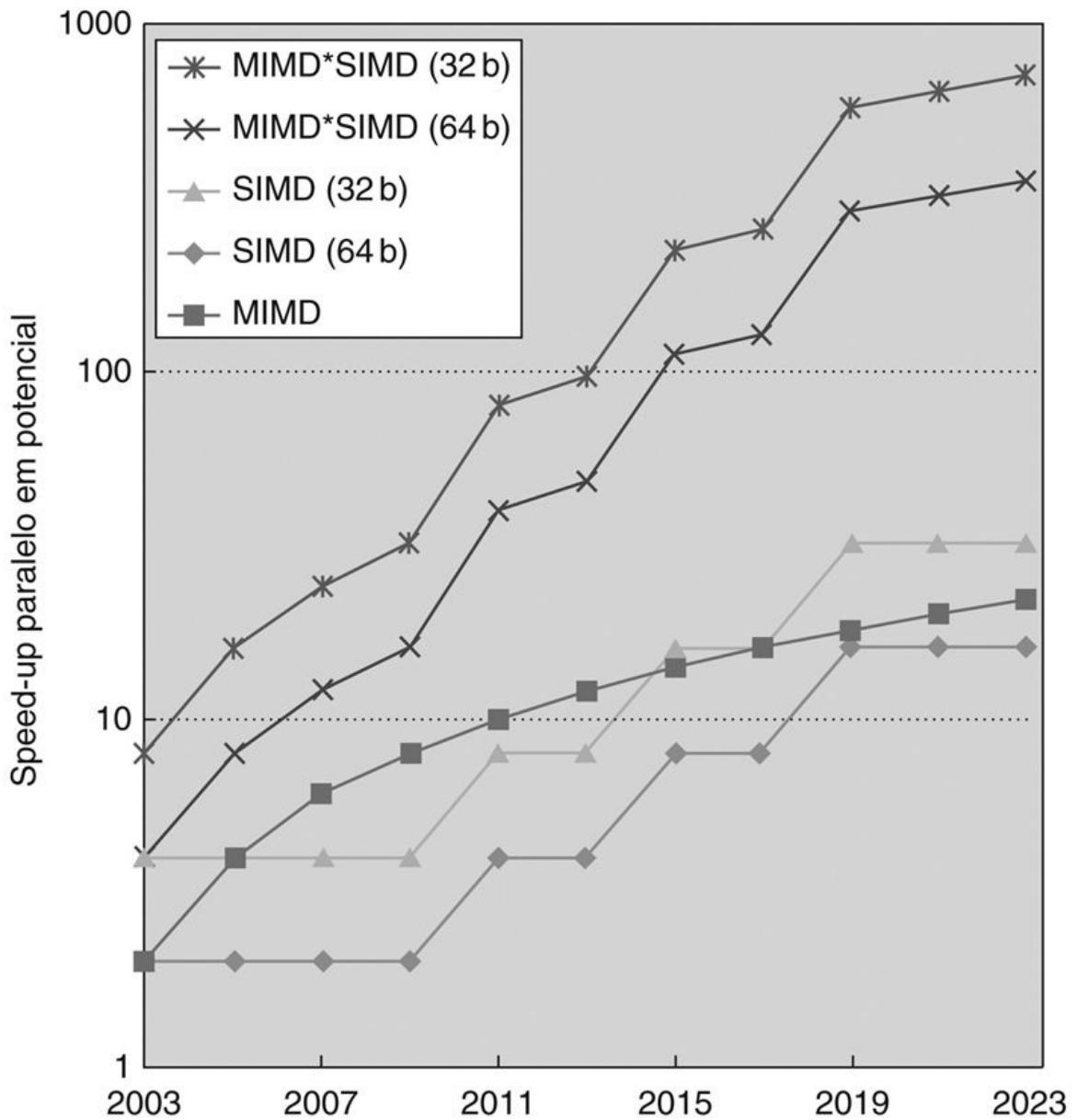


FIGURA 6.28 Speed-up em potencial via paralelismo a partir de MIMD, SIMD e de ambos com o passar do tempo para os computadores x86.

Esta figura considera que dois núcleos por chip para MIMD serão acrescentados a cada dois anos, e que o número de operações para SIMD dobrará a cada quatro anos.

- O uso de processamento paralelo em domínios como a computação científica e de engenharia é comum. Esse domínio de aplicação possui uma necessidade quase ilimitada de mais computação. Ele também possui muitas aplicações com uma grande quantidade de concorrência natural. Mais uma vez, os clusters dominam essa área de aplicação. Por exemplo, usando o

relatório Top 500 de 2012, os clusters representam mais de 80% dos 500 resultados Linpack mais rápidos.

- Todos os fabricantes de microprocessador de desktop e servidor estão construindo multiprocessadores para alcançar o desempenho mais alto, de modo que, diferente do passado, não existe um caminho fácil para o desempenho mais alto para aplicações sequenciais. Como dissemos anteriormente, os programas sequenciais agora são programas lentos. Logo, os programadores que precisam de desempenho mais alto *precisam* colocar seus códigos em paralelo ou escrever novos programas de processamento paralelo.
- No passado, os microprocessadores e os multiprocessadores estavam sujeitos a diferentes definições de sucesso. Ao expandir o desempenho do processador único, os arquitetos de microprocessador ficavam felizes se o desempenho com uma única thread subisse pela raiz quadrada da área de silício aumentada. Assim, eles ficavam felizes com uma melhoria de desempenho abaixo da linear em termos de recursos. O sucesso do multiprocessador era definido como um speed-up *linear* como função do número de processadores, supondo que o custo da compra ou o custo da administração de n processadores era n vezes o custo de um processador. Agora que o paralelismo está acontecendo no chip via multicore, podemos usar a métrica tradicional do microprocessador, para ter sucesso na melhoria do desempenho abaixo da linear.
- O sucesso da compilação em tempo de execução just-in-time e do autoajuste torna viável pensar no software adaptando-se para tirar proveito do número cada vez maior de núcleos por chip, o que oferece uma flexibilidade que não está disponível quando limitado a compiladores estáticos.
- Diferente do passado, o movimento do código-fonte aberto tornou-se uma parte fundamental da indústria de software. Esse movimento é uma meritocracia, em que melhores soluções de engenharia podem ganhar a fatia de desenvolvedores em relação a questões legadas. Ele também alcança a inovação, convidando a mudança no software antigo e recebendo novas linguagens e produtos de software. Essa cultura aberta poderia ser extremamente útil nessa época de mudança rápida.
Para motivar os leitores a abraçar essa revolução, demonstramos o potencial do paralelismo de forma concreta para a multiplicação matricial no Intel Core i7 (Sandy Bridge) nas seções Mais Rápido dos Capítulos de 3 a 6:
- O paralelismo em nível de dados no [Capítulo 3](#) melhorou o desempenho por

um fator de 3,85, executando quatro operações de ponto flutuante com 64 bits em paralelo, usando operandos de 256 bits das instruções AVX, demonstrando o valor do padrão SIMD.

- O paralelismo em nível de instrução no [Capítulo 4](#) empurrou o desempenho por outro fator de 2,3, desdobrando os loops 4 vezes para oferecer, ao hardware de execução fora de ordem, mais instruções para escalar.
- As otimizações de cache no [Capítulo 5](#) melhoraram o desempenho de matrizes que não cabiam na cache de dados L1 por outro fator de 2,0 a 2,5, usando o bloqueio de cache para reduzir as falhas de cache.
- O paralelismo em nível de thread, neste capítulo, melhorou o desempenho de matrizes que não cabiam em uma única cache de dados L1 por outro fator de 4 a 14, utilizando todos os 16 núcleos de nossos chips multicore, demonstrando o valor do padrão MIMD. Fizemos isso acrescentando uma única linha, usando uma pragma OpenMP.

O uso das ideias deste livro e o ajuste do software a esse computador acrescentou 24 linhas de código ao DGEMM. Para os tamanhos de matriz de 32×32 , 160×160 , 480×480 e 960×960 , o speed-up de desempenho geral advindo dessas ideias, colocadas em prática nessas duas dezenas de linhas de código, tem um fator de 8, 39, 129 e 212, respectivamente!

Essa revolução paralela na interface de hardware/software talvez seja o maior desafio que esse campo encarou nos últimos 60 anos. Você também pode pensar nela como a maior das oportunidades, como demonstram nossas seções Mais Rápido. Essa revolução oferecerá muitas novas oportunidades de pesquisa e negócios dentro e fora do campo de TI, e as empresas que dominam a era do multicore podem não ser as mesmas que dominaram a era do processador único. Depois de compreender as tendências fundamentais do hardware e aprender a adaptar o software a elas, talvez você será um dos inovadores que aproveitará as oportunidades que certamente aparecerão nos tempos de incerteza por vir. Esperamos que um dia nos beneficiemos de suas invenções!

6.14. Exercícios

- 6.1** Primeiro, escreva uma lista das atividades diárias que você realiza normalmente em um fim de semana. Por exemplo, você poderia levantar da cama, tomar um banho, se vestir, tomar o café, secar seu cabelo, escovar os dentes etc. Lembre-se de distribuir sua lista de modo que tenha um mínimo de dez atividades.

6.1.1 [5] <§6.2> Agora considere qual dessas atividades já está explorando alguma forma de paralelismo (por exemplo, escovar vários dentes ao mesmo tempo em vez de um de cada vez, carregar um livro de cada vez para a escola em vez de colocar todos eles na sua mochila, e depois carregá-los “em paralelo”). Para cada uma de suas atividades, discuta se elas já estão sendo executadas em paralelo, mas se não, por que não estão.

6.1.2 [5] <§6.2> Em seguida, considere quais das atividades poderiam ser executadas simultaneamente (por exemplo, tomar café e escutar as notícias). Para cada uma das suas atividades, descreva qual outra atividade poderia ser emparelhada com essa atividade.

6.1.3 [5] <§6.2> Para o Exercício 6.1.2, o que poderíamos mudar sobre os sistemas atuais (por exemplo, banhos, roupas, TVs, carros) de modo que pudéssemos realizar mais tarefas em paralelo?

6.1.4 [5] <§6.2> Estime quanto tempo a menos seria necessário para executar essas atividades se você tentasse executar o máximo de tarefas em paralelo possível.

6.2 Você está tentando preparar três tortas de amora. Os ingredientes são os seguintes:

1 xícara de manteiga

1 xícara de açúcar

4 ovos grandes

1 colher de chá de extrato de baunilha

1/2 colher de chá de sal

1/4 colher de chá de noz moscada

1 1/2 xícaras de farinha de trigo

1 xícara de amoras

A receita para uma única torta é a seguinte:

Passo 1: pré-aqueça o forno a 160 °C. Unte e polvilhe farinha na forma.

Passo 2: em uma bacia grande, bata com a batedeira a manteiga e o açúcar em velocidade média até que a massa fique leve e macia. Acrescente ovos, baunilha, sal e noz moscada. Bata até que tudo fique totalmente misturado. Reduza a velocidade da batedeira e acrescente farinha de trigo, 1/2 xícara por vez, batendo até ficar bem misturado.

Passo 3: inclua as amoras aos poucos. Espalhe uniformemente na forma da torta. Leve ao forno 60 minutos.

6.2.1 [5] <§6.2> Sua tarefa é cozinhar três tortas da forma mais eficiente possível. Supondo que você só tenha um forno com tamanho suficiente para

conter uma torta, uma bacia grande, uma forma de torta e uma batedeira, prepare um plano para fazer as três tortas o mais rapidamente possível. Identifique os gargalos para completar essa tarefa.

6.2.2 [5] <§6.2> Suponha agora que você tenha três bacias, três formas de torta e três batedeiras. O quanto o processo fica mais rápido, agora que você tem esses recursos adicionais?

6.2.3 [5] <§6.2> Agora suponha que você tem dois amigos que o ajudarão a cozinar, e que você tem um forno grande, que possa acomodar todas as três tortas. Como isso mudará o plano que você preparou no Exercício 6.2.1?

6.2.4 [5] <§6.2> Compare a tarefa de preparação da torta com o cálculo de três iterações de um loop em um computador paralelo. Identifique o paralelismo em nível de dados e o paralelismo em nível de tarefa no loop de preparação da torta.

6.3 Muitas aplicações de computador envolvem a pesquisa por um conjunto de dados e a classificação dos dados. Diversos algoritmos eficientes de busca e classificação foram criados para reduzir o tempo de execução dessas tarefas tediosas. Neste problema, vamos considerar como é melhor colocar essas tarefas em paralelo.

6.3.1 [10] <§6.2> Considere o seguinte algoritmo de busca binária (um algoritmo clássico do tipo dividir e conquistar) que procura um valor X em um array de N elementos A e retorna o índice da entrada correspondente:

```

BinarySearch(A[0..N-1], X) {
    low = 0
    high = N -1
    while (low <= high) {
        mid = (low + high) / 2
        if (A[mid] > X)
            high = mid -1
        else if (A[mid] < X)
            low = mid + 1
        else
            return mid // found
    }
    return -1 // not found
}

```

Suponha que você tenha Y núcleos em um processador multicore para executar BinarySearch. Supondo que Y seja muito menor que N, expresse o fator de speed-up que você poderia esperar obter para os valores de Y e N. Desenhe isso em um gráfico.

6.3.2 [5] <§6.2> Em seguida, suponha que Y seja igual a N. Como isso afetaria suas conclusões na sua resposta anterior? Se você estivesse encarregado de obter o melhor fator de speed-up possível (ou seja, expansão forte), explique como poderia mudar esse código para obter isso.

6.4 Considere o seguinte trecho de código em C:

```
for (j=2;j<1000;j++)
    D[j] = D[j-1]+D[j-2];
```

O código MIPS correspondente a esse fragmento é:

	addiu	\$s2,\$zero,7992
	addiu	\$s1,\$zero,16
loop:	l.d	\$f0, -16(\$s1)
	l.d	\$f2, -8(\$s1)
	add.d	\$f4, \$f0, \$f2
	s.d	\$f4, 0(\$s1)
	addiu	\$s1, \$s1, 8
	bne	\$s1, \$s2, loop

As instruções têm as seguintes latências associadas (em ciclos):

add.d	l.d	s.d	addiu
4	6	1	2

6.4.1 [10] <§6.2> Quantos ciclos são necessários para que todas as instruções em uma única iteração do loop anterior sejam executadas?

6.4.2 [10] <§6.2> Quando uma instrução em uma iteração posterior de um loop depende do valor de dados produzido em uma iteração anterior do mesmo loop, dizemos que existe uma *dependência carregada pelo loop* entre as iterações do loop. Identifique as dependências carregadas pelo loop no código anterior. Identifique a variável de programa dependente e os registradores em nível de assembly. Você pode ignorar a variável de indução de loop j.

6.4.3 [10] <§6.2> O desdobramento de loop foi descrito no [Capítulo 4](#). Aplique

o desdobramento de loop a esse loop e depois considere a execução desse código em um sistema de passagem de mensagem com memória distribuída com 2 nós. Suponha que usaremos a passagem de mensagem conforme descrito na [Seção 6.7](#), na qual apresentamos uma nova operação $\text{send}(x,y)$ que envia ao nó x o valor y , e uma operação $\text{receive}()$ que espera pelo valor sendo enviado a ele. Suponha que as operações send gastem um ciclo para despachar (ou seja, outras instruções no mesmo nó podem prosseguir para o próximo ciclo), mas gastem 10 ciclos para serem recebidas no nó receptor. Operações receive provocam stall da execução no nó em que são executadas até que recebam uma mensagem. Produza um schedule para os dois nós; considere um fator de desdobramento de 4 para o corpo do loop (ou seja, o corpo do loop aparecerá quatro vezes). Calcule o número de ciclos necessários para que o loop seja executado no sistema de passagem de mensagens.

6.4.4 [10] <§6.2> A latência da rede de interconexão desempenha um papel importante na eficiência dos sistemas de passagem de mensagens. Que velocidade a interconexão precisa ter, a fim de obter qualquer speed-up com o uso do sistema distribuído descrito no Exercício 6.4.3?

6.5 Considere o seguinte algoritmo mergesort recursivo (outro algoritmo clássico para dividir e conquistar). Mergesort foi descrito inicialmente por John von Neumann em 1945. A ideia básica é dividir uma lista não classificada x de m elementos em duas sublistas de aproximadamente metade do tamanho da lista original. Repita essa operação em cada sublista e continue até que tenhamos listas de tamanho 1. Depois, começando com sublistas de tamanho 1, faça o “merge” das duas sublistas em uma única lista classificada.

```
Mergesort(m)
    var list left, right, result
    if length(m) ≤ 1
        return m
    else
        var middle = length(m) / 2
        for each x in m up to middle
            add x to left
        for each x in m after middle
            add x to right
        left = Mergesort(left)
        right = Mergesort(right)
        result = Merge(left, right)
    return result
```

A etapa do merge é executada pelo seguinte código:

```

Merge(left,right)
    var list result
    while length(left) >0 and length(right) > 0
        if first(left) ≤ first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
        if length(left) >0
            append rest(left) to result
        if length(right) >0
            append rest(right) to result
    return result

```

6.5.1 [10] <§6.2> Suponha que você tenha Y núcleos em um processador multicore para executar o MergeSort. Supondo que Y seja muito menor que $\text{length}(m)$, expresse o fator de speed-up que você poderia esperar obter para os valores de Y e $\text{length}(m)$. Desenhe isso em um gráfico.

6.5.2 [10] <§6.2> Em seguida, considere que Y é igual a $\text{length}(m)$. Como isso afetaria suas conclusões na sua resposta anterior? Se você estivesse encarregado de obter o melhor fator de speed-up possível (ou seja, expansão forte), explique como poderia mudar esse código para obtê-lo.

6.6 A multiplicação de matriz desempenha um papel importante em diversas aplicações. Duas matrizes só podem ser multiplicadas se o número de colunas da primeira matriz for igual ao número de linhas na segunda.

Vamos supor que tenhamos uma matriz $m \times n$ (A) e queremos multiplicá-la por uma matriz $n \times p$ (B). Podemos expressar seu produto como uma matriz $m \times p$ indicada por AB (ou $A \cdot B$). Se atribuirmos $C = AB$, e $c_{i,j}$ indicar a entrada em C na posição (i, j) , então para cada elemento i e j com $1 \leq i \leq m$ e $1 \leq j \leq p$. Agora, queremos ver se podemos fazer o cálculo de C em paralelo. Suponha que as matrizes estejam dispostas na memória sequencialmente da seguinte forma: $a_{1,1}, a_{2,1}, a_{3,1}, a_{4,1}, \dots$, etc.

6.6.1 [10] <§6.5> Suponha que iremos calcular C em uma máquina de memória compartilhada de núcleo único e uma máquina com memória compartilhada

de 4 núcleos. Calcule o speed-up que esperaríamos obter em uma máquina de 4 núcleos, ignorando quaisquer problemas de memória.

6.6.2 [10] <§6.5> Repita o Exercício 6.6.1, supondo que as atualizações em C incorrem em uma falha de cache, devido ao compartilhamento falso quando os elementos consecutivos que estão em sequência (ou seja, índice i) são atualizados.

6.6.3 [10] <§6.5> Como você consertaria o problema de compartilhamento falso que pode ocorrer?

6.7 Considere as seguintes partes de dois programas diferentes rodando ao mesmo tempo com quatro processadores em um processador multicore simétrico (SMP). Suponha que, antes que esse código seja executado, tanto x quanto y sejam 0.

Core 1: $x = 2;$

Core 2: $y = 2;$

Core 3: $w = x + y + 1;$

Core 4: $z = x + y;$

6.7.1 [10] <§6.5> Quais são todos os valores resultantes possíveis de w , x , y e z ? Para cada resultado possível, explique como poderíamos chegar a esses resultados. Você precisará examinar todas as intercalações possíveis das instruções.

6.7.2 [5] <§6.5> Como você poderia tornar a execução mais determinística, de modo que somente um conjunto de valores seja possível?

6.8 O problema do jantar dos filósofos é um problema clássico de

sincronização e concorrência. O problema geral é enunciado como filósofos sentados em volta de uma mesa redonda fazendo uma de duas coisas: comendo ou pensando. Quando eles estão comendo, não estão pensando, e quando estão pensando, não estão comendo. Há uma tigela de macarrão no centro. Um garfo é colocado entre cada filósofo. O resultado é que cada filósofo tem um garfo à sua esquerda e um garfo à sua direita. Devido à forma como se come macarrão, o filósofo precisa de dois garfos para comer, e só pode usar os garfos do seu lado esquerdo e direito. Os filósofos não conversam entre si.

6.8.1 [10] <§6.7> Descreva o cenário em que nenhum dos filósofos consegue comer (ou seja, inanição). Qual é a sequência de eventos que leva a esse problema?

6.8.2 [10] <§6.7> Descreva como podemos solucionar esse problema introduzindo o conceito de uma prioridade. Mas podemos garantir que trataremos de todos os filósofos de forma justa? Explique.

Agora, suponha que contrataremos um garçom encarregado de atribuir garfos aos filósofos. Ninguém pode pegar um garfo até que o garçom lhe diga que pode. O garçom tem conhecimento global de todos os garfos. Além disso, se impusermos a diretriz de que os filósofos sempre solicitarão o seu garfo da esquerda antes de apanhar seu garfo da direita, então podemos garantir que o impasse (deadlock) será evitado.

6.8.3 [10] <§6.7> Podemos implementar as solicitações ao garçom como uma fila de solicitações ou como uma retentativa periódica de uma solicitação. Com uma fila, as solicitações são tratadas na ordem em que são recebidas. O problema com o uso da fila é que podemos nem sempre ser capazes de atender ao filósofo cuja solicitação está no início da fila (devido à indisponibilidade de recursos). Descreva um cenário com cinco filósofos, em que uma fila é fornecida, mas o serviço não é concedido mesmo que haja garfos disponíveis para outro filósofo (cuja solicitação está mais profunda na fila) utilizar.

6.8.4 [10] <§6.7> Se implementarmos solicitações ao garçom repetindo periodicamente nossa solicitação até que os recursos estejam disponíveis, isso solucionará o problema descrito no Exercício 6.8.3? Explique.

6.9 Considere as três organizações de CPU a seguir:

CPU SS: Um microprocessador superescalar de dois núcleos que oferece capacidades de despacho fora de ordem em duas unidades funcionais (FUs). Somente uma única thread pode ser executada em cada núcleo de cada vez.

CPU MT: Um processador multithreaded fine-grained que permite que instruções de duas threads sejam executadas simultaneamente (ou seja, existem duas unidades funcionais), embora somente instruções de uma única thread possam ser emitidas em cada ciclo.

CPU SMT: Um processador SMT que permite que instruções de duas threads sejam executadas simultaneamente (ou seja, existem duas unidades funcionais), e as instruções de qualquer uma ou ambas as threads podem ser emitidas para executar em qualquer ciclo.

Suponha que tenhamos duas threads, X e Y, para executar nessas CPUs, o que inclui as seguintes operações:

Thread X	Thread Y
A1 – leva 3 ciclos para executar	B1 – leva 2 ciclos para executar
A2 – sem dependências	B2 – conflitos para uma unidade funcional com B1
A3 – conflitos para uma unidade funcional com A1	B3 – depende do resultado de B2
A4 – depende do resultado de A3	B4 – sem dependências e leva 2 ciclos para executar

Suponha que todas as instruções utilizem um único ciclo para serem executadas, a menos que sejam observados de outra forma ou que encontrem um hazard.

6.9.1 [10] < §6.4> Suponha que você tenha uma CPU SS. Quantos ciclos são necessários para executar essas duas threads? Quantos slots de despacho são desperdiçados devido a hazards?

6.9.2 [10] < §6.4> Agora, suponha que você tenha duas CPUs SS. Quantos ciclos são necessários para executar essas duas threads? Quantos slots de despacho são desperdiçados devido a hazards?

6.9.3 [10] < §6.4> Suponha que você tenha uma CPU MT. Quantos ciclos são necessários para executar essas duas threads? Quantos slots de despacho são desperdiçados devido a hazards?

6.10 O software de virtualização está sendo agressivamente implantado para reduzir os custos de gerenciamento dos servidores de alto desempenho de hoje. Empresas como VMWare, Microsoft e IBM desenvolveram diversos produtos de virtualização. O conceito geral, descrito no [Capítulo 5](#), é que uma camada hipervisora pode ser introduzida entre o hardware e o sistema operacional para permitir que vários sistemas operacionais compartilhem o mesmo hardware físico. A camada hipervisora é então responsável por alocar recursos de CPU e memória, além de tratar os serviços normalmente tratados pelo sistema operacional (por exemplo, E/S).

A virtualização oferece uma visão abstrata do hardware subjacente ao sistema operacional host e ao software de aplicação. Isso exigirá que repensemos como os sistemas multicore e multiprocessador serão projetados no futuro para dar suporte ao compartilhamento das CPUs e memórias por diversos sistemas operacionais simultaneamente.

6.10.1 [30] <§6.4> Selecione dois hipervisores no mercado hoje e compare como eles virtualizam e gerenciam o hardware subjacente (CPUs e memória).

6.10.2 [15] <§6.4> Discuta quais mudanças podem ser necessárias nas plataformas de CPU multicore do futuro, a fim de que sejam mais coerentes com as demandas de recursos impostas sobre esses sistemas. Por exemplo, o multithreading pode desempenhar um papel eficaz para reduzir a competição por recursos de computação?

6.11 Gostaríamos de executar o loop a seguir da forma mais eficiente possível. Temos duas máquinas diferentes, uma máquina MIMD e uma máquina SIMD.

```
for (i=0; i < 2000; i++)
    for (j=0; j<3000; j++)
        X_array[i][j] = Y_array[j][i] + 200;
```

6.11.1 [10] <§6.3> Para uma máquina MIMD com quatro CPUs, mostre a sequência de instruções MIPS que você executaria em cada CPU. Qual é o speed-up para essa máquina MIMD?

6.11.2 [20] <§6.3> Para uma máquina SIMD de largura 8 (ou seja, oito unidades funcionais SIMD paralelas), escreva um programa em assembly usando suas próprias extensões SIMD ao MIPS para executar o loop. Compare o número de instruções executadas na máquina SIMD com a máquina MIMD.

6.12 Um array sistólico é um exemplo de uma máquina MISD. Um array sistólico é uma rede de pipeline ou frontend de elementos de processamento de dados. Cada um desses elementos não precisa de um contador de programa, pois a execução é disparada pela chegada de dados. Os arrays sistólicos com clock são calculados em lock-step, com cada

processador realizando fases alternadas de cálculo e comunicação.

6.12.1 [10] <§6.3> Considere as implementações propostas de um array sistólico (você pode encontrá-las na Internet ou em publicações técnicas). Depois, tente programar o loop fornecido no Exercício 6.11 usando esse modelo MISD. Discuta quaisquer dificuldades que você encontrar.

6.12.2 [10] <§6.3> Discuta as semelhanças e diferenças entre uma máquina MISD e SIMD. Responda a essa pergunta em termos de paralelismo em nível de dados.

6.13 Suponha que queiramos executar o loop DAXPY mostrado na [Seção 6.3](#) em assembly MIPS na GPU NVIDIA 8800 GTX descrita neste capítulo. Nesse problema, vamos supor que todas as operações matemáticas sejam realizadas em números de ponto flutuante com precisão simples (vamos mudar o nome do loop para SAXPY). Suponha que as instruções utilizem o seguinte número de ciclos para serem executadas.

Loads	Stores	Add.S	Mult.S
5	2	3	4

6.13.1 [20] <§6.6> Descreva como você construirá warps para o loop SAXPY explorar os oito núcleos fornecidos em um único multiprocessador.

6.14 Faça o download do CUDA Toolkit e SDK em http://www.nvidia.com/object/cuda_get.html. Lembre-se de usar a versão “emurelease” (Emulation Mode) do código (você não precisará do hardware NVIDIA real para esse trabalho). Crie os programas de exemplo fornecidos no SDK e confirme se eles rodarão no emulador.

6.14.1 [90] <§6.6> Usando o “template” de exemplo de SDK como ponto de partida, escreva um programa CUDA para realizar o seguinte vetor de operações:

- $a - b$ (subtração vetor-vetor)
- $a \cdot b$ (produto pontual de vetor)

O produto pontual de dois vetores $a = [a_1, a_2, \dots, a_n]$ e $b = [b_1, b_2, \dots, b_n]$ é definido como:

$$a \cdot b = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

Submeta o código para cada programa que demonstra cada operação e verifique a exatidão dos resultados.

6.14.2 [90] <§6.6> Se você tiver hardware GPU disponível, complete uma análise de desempenho do seu programa, examinando o tempo de computação para a GPU e uma versão de CPU do seu programa para uma faixa de tamanhos de vetor. Explique quaisquer resultados que você encontrar.

6.15 A AMD recentemente anunciou que integrará uma unidade de processamento gráfica com seus núcleos x86 em um único pacote, embora com diferentes clocks para cada um dos núcleos. Este é um exemplo de um sistema multiprocessador heterogêneo que esperamos ver produzido comercialmente no futuro próximo. Um dos principais pontos de projeto será permitir a comunicação de dados rápida entre a CPU e a GPU. Atualmente a comunicação deve ser realizada entre chips de CPU e GPU discretos. Mas isso está mudando na arquitetura Fusion da AMD. Atualmente, o plano é usar múltiplos canais PCI express (pelo menos, 16) para facilitar a intercomunicação. A Intel também está saltando para essa arena com seu chip Larrabee. A Intel está considerando o uso de sua tecnologia de interconexão QuickPath.

6.15.1 [25] <§6.7> Compare a largura de banda e latência associadas a essas duas tecnologias de interconexão.

6.16 Consulte a [Figura 6.14b](#), que mostra uma topologia de interconexão de cubo n de ordem 3, que interconecta oito nós. Um recurso atraente de uma topologia de rede de interconexão de cubo n é sua capacidade de sustentar links partidos e ainda oferecer conectividade.

6.16.1 [10] <§6.8> Desenvolva uma equação que calcule quantos links no cubo n (onde n é a ordem do cubo) podem falhar e ainda podemos garantir que um link não partido existirá para conectar qualquer nó no cubo n .

6.16.2 [10] <§6.8> Compare a resiliência à falha do cubo n com uma rede de interconexão totalmente conectada. Desenhe uma comparação da confiabilidade como uma função do número de links que podem falhar para as duas topologias.

6.17 O benchmarking é um campo de estudo que envolve identificar cargas de trabalho representativas para rodar em plataformas de computação específicas a fim de poder comparar objetivamente o desempenho de um sistema com outro. Neste exercício, vamos comparar duas classes de benchmarks: o benchmark Whetstone CPU e o pacote de benchmark

PARSEC. Selecione um programa do PARSEC. Todos os programas deverão estar disponíveis gratuitamente na Internet. Considere a execução de múltiplas cópias do Whetstone contra a execução do benchmark PARSEC em qualquer um dos sistemas descritos na [Seção 6.10](#).

6.17.1 [60] <§6.10> O que é inherentemente diferente entre essas duas classes de carga de trabalho quando executadas nesses sistemas multicore?

6.17.2 [60] <§6.10> Em termos do modelo roofline, que dependência terão os resultados que você obtiver ao executar esses benchmarks na quantidade de compartilhamento e sincronização presente na carga de trabalho utilizada?

6.18 Ao realizar cálculos sobre matrizes esparsas, a latência na hierarquia de memória torna-se um fator muito importante. As matrizes esparsas não possuem a localidade espacial no fluxo de dados, normalmente encontrada nas operações de matriz. Como resultado, novas representações de matriz foram propostas.

Uma das representações de matriz esparsa mais antigas é o Yale Sparse Matrix Format. Ele armazena uma matriz esparsa inicial $m \times n$, M , em formato de linha usando três arrays unidimensionais. Suponha que R indique o número de entradas diferentes de zero em M . Construímos um array A de tamanho R que contém todas as entradas diferentes de zero de M (na ordem da esquerda para a direita e de cima para baixo). Também construímos um segundo array IA de tamanho $m + 1$ (ou seja, uma entrada por linha, mais um). $IA(i)$ contém o índice de A do primeiro elemento diferente de zero da linha i . A linha i da matriz original se estende de $A(IA(i))$ até $A(IA(i + 1) - 1)$. O terceiro array, JA , contém o índice de coluna de cada elemento de A , de modo que também tem tamanho R .

6.18.1 [15] <§6.10> Considere a matriz esparsa X a seguir e escreva o código C que armazenaria esse código no Yale Sparse Matrix Format.

Linha 1 [1, 2, 0, 0, 0, 0]
Linha 2 [0, 0, 1, 1, 0, 0]
Linha 3 [0, 0, 0, 0, 9, 0]
Linha 4 [2, 0, 0, 0, 0, 2]
Linha 5 [0, 0, 3, 3, 0, 7]
Linha 6 [1, 3, 0, 0, 0, 1]

6.18.2 [10] <§6.10> Em termos do espaço de armazenamento, supondo que cada elemento na matriz X tenha formato de ponto flutuante com precisão simples, calcule a quantidade de armazenamento usada para armazenar a matriz acima no Yale Sparse Matrix Format.

6.18.3 [15] <§6.10> Realize a multiplicação de matriz da Matriz X pela Matriz Y mostrada a seguir.

$$[2, 4, 1, 99, 7, 2]$$

Coloque esse cálculo em um loop e meça o tempo de sua execução. Não se esqueça de aumentar o número de vezes que esse loop é executado para obter uma boa resolução na sua medição do tempo. Compare o tempo de execução do uso de uma representação simples da matriz e do Yale Sparse Matrix Format.

6.18.4 [15] <§6.10> Você consegue achar uma representação de matriz esparsa mais eficiente (em termos de overhead de espaço e computacional)?

6.19 Nos sistemas do futuro, esperamos ver plataformas de computação heterogêneas construídas a partir de CPUs heterogêneas. Começamos a ver algumas aparecendo no mercado de processamento embutido nos sistemas que contêm DSPs de ponto flutuante e CPUs de microcontrolador em um pacote de módulo multichip.

Suponha que você tenha três classes de CPU:

CPU A — Uma CPU multicore de velocidade moderada (com uma unidade de ponto flutuante) que pode executar múltiplas instruções por ciclo.

CPU B — Uma CPU inteira de único core rápida (ou seja, sem unidade de ponto flutuante) que pode executar uma única instrução por ciclo.

CPU C — Uma CPU de vetor lenta (com capacidade de ponto flutuante) que pode executar múltiplas cópias da mesma instrução por ciclo.

Suponha que nossos processadores executem nas seguintes frequências:

CPU A	CPU B	CPU C
1 GHz	3 GHz	250 MHz

A CPU A pode executar 2 instruções por ciclo, a CPU B pode executar 1 instrução por ciclo, e a CPU C pode executar 8 instruções (embora a mesma instrução) por ciclo. Suponha que todas as operações possam concluir sua execução em um único ciclo de latência sem quaisquer hazards.

Todas as três CPUs possuem a capacidade de realizar aritmética com inteiros, embora a CPU B não possa realizar aritmética de ponto flutuante. As CPUs A e B têm um conjunto de instruções semelhante a um processador MIPS. A CPU C só pode realizar operações de soma e subtração de ponto flutuante, assim como loads e stores de memória. Suponha que todas as CPUs tenham acesso à memória compartilhada e que a sincronização tenha custo zero.

A tarefa em mãos é comparar duas matrizes X e Y, que contêm cada uma 1024×1024 elementos de ponto flutuante. A saída deverá ser uma contagem dos índices numéricos em que o valor em X foi maior que o valor em Y.

6.19.1 [10] <§6.11> Descreva como você particionaria o problema nas três CPUs diferentes para obter o melhor desempenho.

6.19.2 [10] <§6.11> Que tipo de instrução você acrescentaria à CPU de vetor C para obter o melhor desempenho?

6.20 Suponha que um sistema de computador quad-core possa processar consultas de banco de dados em uma taxa de estado constante de solicitações por segundo. Suponha também que cada instrução leve, na média, uma quantidade de tempo fixa para ser processada. A tabela a seguir mostra pares de latência de transação e taxa de processamento.

Latência de transação média	Taxa de processamento de transação máxima
1 ms	5000/seg
2 ms	5000/seg
1 ms	10.000/seg

2 ms	10.000/seg
------	------------

Para cada um dos pares na tabela, responda às seguintes perguntas:

- 6.20.1** [10] <§6.11> Na média, quantas solicitações estão sendo processadas em determinado instante?
- 6.20.2** [10] <§6.11> Se você passasse para um sistema de 8 cores, na forma ideal, o que aconteceria com a vazão do sistema (ou seja, quantas transações/segundo o computador processará)?
- 6.20.3** [10] <§6.11> Discuta por que raramente obtemos esse tipo de speed-up simplesmente aumentando o número de cores.

Respostas das Seções “Verifique você mesmo”

§6.1, página 440: Falso. O paralelismo em nível de tarefa pode ajudar as aplicações sequenciais e as aplicações sequenciais podem ser criadas para executar em hardware paralelo, embora isso seja mais difícil.

§6.2, página 445: Falso. A expansão *fraca* pode compensar uma parte serial do programa que, de outra forma, limitaria a expansão, mas não tanto para a expansão forte.

§6.3, página 450: Verdadeiro, mas não existem recursos de vetor úteis, como gather-scatter e registradores de comprimento de vetor, que melhoram a eficiência das arquiteturas de vetor. (Como um detalhe mencionado nesta seção, as extensões SIMD do AVX2 oferecem loads indexados por meio de uma operação gather, mas *não* scatter para stores indexados. A geração Haswell do microprocessador x86 é a primeira a dar suporte para AVX2.)

§6.4, página 454: 1. Verdadeiro. 2. Verdadeiro.

§6.5, página 457: 1. Falso. Como o endereço compartilhado é um endereço *físico*, tarefas múltiplas, cada em seus próprios espaços de endereços *virtuais*, podem ser executados muito bem em um multiprocessador de memória compartilhada.

§6.6, página 463: Falso. Chips de DRAM gráficos são premiados por sua maior largura de banda.

§6.7, página 468: 1. Falso. Enviar e receber uma mensagem é uma sincronização implícita, além de um meio de compartilhar dados. 2. Verdadeiro.

§6.8, página 471: Verdadeiro

§6.9, página 480: Verdadeiro. Provavelmente precisamos de inovação em todos os níveis da pilha de hardware e software para que a computação paralela tenha sucesso.

APÊNDICE

A

Assemblers, Link-editores e o Simulador SPIM

James R. Larus
Microsoft Research, Microsoft

O receio do insulto sério não pode justificar sozinho a supressão da livre expressão.

*Louis Brandeis,
Whitney v. California, 1927*

- A.1 Introdução
- A.2 Assemblers
- A.3 Link-editores
- A.4 Carregando
- A.5 Uso da memória
- A.6 Convenção para chamadas de procedimento
- A.7 Exceções e interrupções
- A.8 Entrada e saída
- A.9 SPIM
- A.10 Assembly do MIPS R2000
- A.11 Comentários finais
- A.12 Exercícios

A.1. Introdução

Codificar instruções como números binários é algo natural e eficiente para os computadores. Os humanos, porém, têm muita dificuldade para entender e manipular esses números. As pessoas leem e escrevem símbolos (palavras) muito melhor do que longas sequências de dígitos. O [Capítulo 2](#) mostrou que não precisamos escolher entre números e palavras, pois as instruções do computador podem ser representadas de muitas maneiras. Os humanos podem escrever e ler símbolos, enquanto os computadores podem executar os números binários equivalentes. Este apêndice descreve o processo pelo qual um programa legível ao ser humano é traduzido para um formato que um computador pode executar, oferece algumas dicas sobre a escrita de programas em assembly e explica como executar esses programas no SPIM, um simulador que executa programas MIPS.

Linguagem assembly é a representação simbólica da codificação binária — **linguagem de máquina** — de um computador. O assembly é mais legível do que a linguagem de máquina porque utiliza símbolos no lugar de bits. Os símbolos no assembly nomeiam padrões de bits que ocorrem comumente, como opcodes (códigos de operação) e especificadores de registradores, de modo que as pessoas possam ler e lembrar-se deles. Além disso, o assembly permite que os programadores utilizem *rótulos* para identificar e nomear palavras particulares da memória que mantêm instruções ou dados.

linguagem de máquina

A representação binária utilizada para a comunicação dentro de um sistema computacional.

Uma ferramenta chamada **assembler** (*montador*) traduz do assembly para instruções binárias. Os assemblers oferecem uma representação mais amigável do que os 0s e 1s de um computador, o que simplifica a escrita e a leitura de programas. Nomes simbólicos para operações e locais são uma faceta dessa representação. Outra faceta são as facilidades de programação que aumentam a clareza de um programa. Por exemplo, as **macros**, discutidas na [Seção A.2](#), permitem que um programador estenda o assembly, definindo novas operações.

assembler (*montador*)

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

macro

Uma facilidade de combinação e substituição de padrões que oferece um mecanismo simples para nomear uma sequência de instruções utilizada com frequência.

Um assembler lê um único *arquivo-fonte* em assembly e produz um *arquivo-objeto* com instruções de máquina e informações de trabalho que ajudam a combinar vários arquivos-objeto em um programa. A [Figura A.1.1](#) ilustra como um programa é montado. A maioria dos programas consiste em vários arquivos — também chamados de *módulos* — que são escritos, compilados e montados de forma independente. Um programa também pode usar rotinas pré-escritas fornecidas em uma *biblioteca de programa*. Um módulo normalmente contém *referências* a sub-rotinas e dados definidos em outros módulos e em bibliotecas. O código em um módulo não pode ser executado quando contém **referências não resolvidas** para rótulos em outros arquivos-objeto ou bibliotecas. Outra ferramenta, chamada **link-editor**, combina uma coleção de arquivos-objeto e biblioteca em um *arquivo executável*, que um computador pode executar.

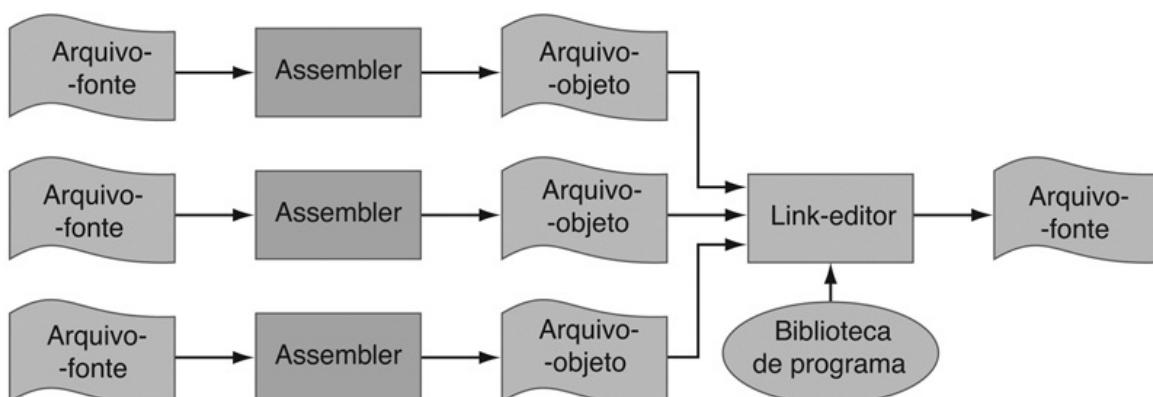


FIGURA A.1.1 O processo que produz um arquivo executável.

Um assembler traduz um arquivo em assembly para um arquivo-objeto, que é link-editado a outros arquivos e bibliotecas para gerar um arquivo executável.

referência não resolvida

Uma referência que exige mais informações de um arquivo externo para estar completa.

link-editor

Também chamado **linker**. Um programa de sistema que combina programas em linguagem de máquina montados independentemente e resolve todos os rótulos indefinidos em um arquivo executável.

Para ver a vantagem do assembly, considere a sequência de figuras mostrada a seguir, todas contendo uma pequena sub-rotina que calcula e exibe a soma dos quadrados dos inteiros de 0 a 100. A [Figura A.1.2](#) mostra a linguagem de máquina que um computador MIPS executa. Com esforço considerável, você poderia usar as tabelas de formato de opcode e instrução do [Capítulo 2](#) para traduzir as instruções em um programa simbólico, semelhante à [Figura A.1.3](#). Essa forma de rotina é muito mais fácil de ler, pois as operações e os operandos estão escritos com símbolos, em vez de padrões de bits. No entanto, esse assembly ainda é difícil de acompanhar, pois os locais da memória são indicados por seus endereços, e não por um rótulo simbólico.

0010011101110111111111111100000
101011110111110000000000010100
1010111101001000000000000100000
1010111101001010000000000100100
101011110100000000000000011000
101011110100000000000000011100
100011110101110000000000011100
100011110111000000000000011000
00000001110011100000000000011001
0010010111001000000000000000001
00101001000000010000000001100101
101011110101000000000000011100
0000000000000000111100000010010
00000011000011111100100000100001
000101000010000011111111110111
1010111101110010000000000011000
00111100000001000001000000000000
1000111101001010000000000011000
00001100000100000000000011101100
001001001000010000000010000110000
1000111101111100000000000010100
00100111011110100000000000100000
000000111100000000000000001000
000000000000000000000000001000001

FIGURA A.1.2 Código em linguagem de máquina MIPS para uma rotina que calcula e exibe a soma dos quadrados dos inteiros, entre 0 a 100.

addiu	\$29, \$29, -32
sw	\$31, 20(\$29)
sw	\$4, 32(\$29)
sw	\$5, 36(\$29)
sw	\$0, 24(\$29)
sw	\$0, 28(\$29)
lw	\$14, 28(\$29)
lw	\$24, 24(\$29)
multu	\$14, \$14
addiu	\$8, \$14, 1
slti	\$1, \$8, 101
sw \$8,	28(\$29)
mflo	\$15
addu	\$25, \$24, \$15
bne	\$1, \$0, -9
sw	\$25, 24(\$29)
lui	\$4, 4096
lw	\$5, 24(\$29)
jal	1048812
addiu	\$4, \$4, 1072
lw	\$31, 20(\$29)
addiu	\$29, \$29, 32
jr	\$31
move	\$2, \$0

FIGURA A.1.3 A mesma rotina da [Figura A.1.2](#) escrita em **linguagem assembly**.

Entretanto, o código para a rotina não rotula registradores ou locais de memória, nem inclui comentários.

A [Figura A.1.4](#) mostra o assembly que rotula endereços de memória com nomes simbólicos ou mnemônicos. A maior parte dos programadores prefere ler e escrever dessa forma. Os nomes que começam com um ponto, por exemplo, .data e .globl, são **diretivas do assembler**, que dizem ao assembler como traduzir um programa, mas não produzem instruções de máquina. Nomes seguidos por um sinal de dois-pontos, como str: ou main:, são rótulos que dão nome ao próximo local da memória. Esse programa é tão legível quanto a maioria dos programas em assembly (exceto por uma óbvia falta de comentários), mas ainda é difícil de acompanhar, pois muitas operações simples são exigidas para realizar tarefas simples e porque a falta de construções de fluxo de controle do assembly oferece poucos palpites sobre a operação do programa.

```

.text
.align 2
.globl main

main:
    subu    $sp, $sp, 32
    sw      $ra, 20($sp)
    sd      $a0, 32($sp)
    sw      $0, 24($sp)
    sw      $0, 28($sp)

loop:
    lw      $t6, 28($sp)
    mul   $t7, $t6, $t6
    lw      $t8, 24($sp)
    addu  $t9, $t8, $t7
    sw      $t9, 24($sp)
    addu  $t0, $t6, 1
    sw      $t0, 28($sp)
    ble   $t0, 100, loop
    la     $a0, str
    lw      $a1, 24($sp)
    jal   printf
    move  $v0, $0
    lw      $ra, 20($sp)
    addu  $sp, $sp, 32
    jr     $ra

.str:
    .data
    .align 0
    .asciiz "The sum from 0 .. 100 is %d\n"

```

FIGURA A.1.4 A mesma rotina da [Figura A.1.2](#) escrita em assembly com rótulos, mas sem comentários.

Os comandos que começam com pontos são diretivas do assembler (ver [páginas A-47 a A-49](#)). `.text` indica que as linhas seguintes contêm instruções. `.data` indica que elas contêm dados. `.align n` indica que os itens nas linhas seguintes devem ser alinhados em um limite de 2^n bytes. Logo, `.align 2` significa que o próximo item deverá estar em um limite de palavra. `.globl main` declara que `main` é um símbolo global, que deverá ser visível

ao código armazenado em outros arquivos. Finalmente, `.asciiz` armazena na memória uma string terminada em nulo.

diretiva do assembler

Uma operação que diz ao assembler como traduzir um programa, mas não produz instruções de máquina; sempre começa com um ponto.

Em contraste, a rotina em C na [Figura A.1.5](#) é mais curta e mais clara, pois as variáveis possuem nomes simbólicos e o loop é explícito, em vez de construído com desvios. Na verdade, a rotina em C é a única que escrevemos. As outras formas do programa foram produzidas por um compilador C e um assembler.

```
#include <stdio.h>

int
main (int argc, char *argv[])
{
    int i;
    int sum = 0;

    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

FIGURA A.1.5 A rotina escrita na linguagem de programação C.

Em geral, o assembly desempenha duas funções (ver [Figura A.1.6](#)). A primeira é como uma linguagem de saída dos compiladores. Um *compilador* traduz um programa escrito em uma *linguagem de alto nível* (como C ou Pascal) para um programa equivalente em linguagem de máquina ou assembly. A linguagem de alto nível é chamada de **linguagem-fonte** (ou **código-fonte**), e a saída do compilador é sua *linguagem-objeto* (ou código-objeto).

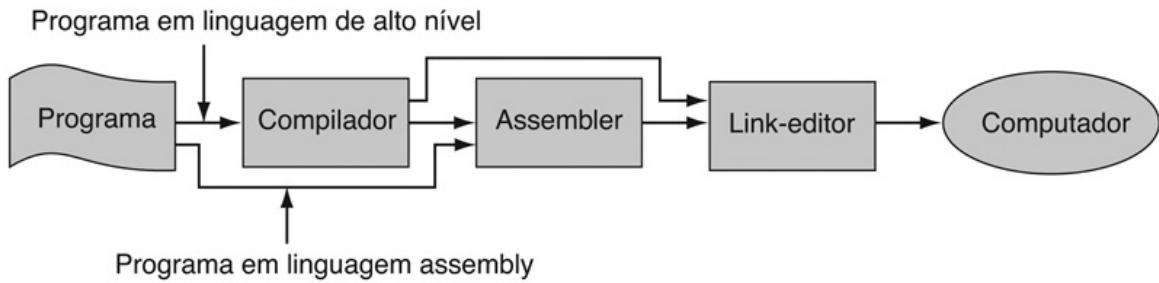


FIGURA A.1.6 O assembly é escrito por um programador ou é a saída de um compilador.

linguagem-fonte (ou código-fonte)

A linguagem de alto nível em que um programa é escrito originalmente.

A outra função do assembly é como uma linguagem para a escrita de programas. Essa função costumava ser a dominante. Hoje, porém, devido a memórias maiores e compiladores melhores, a maioria dos programadores utiliza uma linguagem de alto nível e raramente ou nunca vê as instruções que um computador executa. Apesar disso, o assembly ainda é importante para a escrita de programas em que a velocidade ou o tamanho são fundamentais, ou para explorar recursos do hardware que não possuem correspondentes nas linguagens de alto nível.

Embora este apêndice dê destaque ao assembly do MIPS, a programação assembly na maioria das outras máquinas é muito semelhante. As outras instruções e os modos de endereçamento nas máquinas CISC, como o VAX, podem tornar os programas assembly mais curtos, mas não mudam o processo de montagem de um programa, nem oferecem ao assembly as vantagens das linguagens de alto nível, como verificação de tipos e fluxo de controle estruturado.

Quando usar a linguagem assembly

O motivo principal para programar em assembly, em vez de outra linguagem de alto nível, é que a velocidade ou o tamanho de um programa têm extrema importância. Por exemplo, imagine um computador que controla um mecanismo qualquer, como os freios de um carro. Um computador incorporado em outro dispositivo, como um carro, é chamado de *computador embutido*. Esse tipo de computador precisa responder rápida e previsivelmente aos eventos no mundo

exterior. Como um compilador introduz incerteza sobre o custo de tempo das operações, os programadores podem achar difícil garantir que um programa em linguagem de alto nível responderá dentro de um intervalo de tempo definido — digamos, 1 milissegundo após um sensor detectar que um pneu está derrapando. Um programador assembly, por outro lado, possui mais controle sobre as instruções executadas. Além disso, em aplicações embutidas, reduzir o tamanho de um programa, de modo que caiba em menos chips de memória, reduz o custo do computador embutido.

Uma técnica híbrida, em que a maior parte de um programa é escrita em uma linguagem de alto nível e seções fundamentais são escritas em assembly, aproveita os pontos fortes das duas linguagens. Os programas normalmente gastam a maior parte do seu tempo executando uma pequena fração do código-fonte do programa. Essa observação é exatamente o princípio da localidade em que as caches se baseiam (veja Seção 5.1, no [Capítulo 5](#)).

O perfil do programa mede onde um programa gasta seu tempo e pode localizar as partes de tempo crítico de um programa. Em muitos casos, essa parte do programa pode se tornar mais rápida com melhores estruturas de dados ou algoritmos. No entanto, às vezes, melhorias de desempenho significativas só são obtidas com a recodificação de uma parte crítica de um programa em linguagem assembly.

Essa melhoria não é necessariamente uma indicação de que o compilador da linguagem de alto nível falhou. Os compiladores costumam ser melhores do que os programadores na produção uniforme de código de máquina de alta qualidade por um programa inteiro. Entretanto, os programadores entendem os algoritmos e o comportamento de um programa em um nível mais profundo do que um compilador e podem investir esforço e engenhosidade consideráveis melhorando pequenas seções do programa. Em particular, os programadores em geral consideram vários procedimentos simultaneamente enquanto escrevem seu código. Os compiladores compilam cada procedimento isoladamente e precisam seguir convenções estritas governando o uso dos registradores nos limites de procedimento. Retendo valores comumente usados nos registradores, até mesmo entre os limites dos procedimentos, os programadores podem fazer um programa ser executado com mais rapidez.

Outra vantagem importante do assembly está na capacidade de explorar instruções especializadas, por exemplo, instruções de cópia de string ou combinação de padrões. Os compiladores, na maior parte dos casos, não podem determinar se um loop de programa pode ser substituído por uma única

instrução. Contudo, o programador que escreveu o loop pode substituí-lo facilmente por uma única instrução.

Hoje, a vantagem de um programador sobre um compilador tornou-se difícil de manter, pois as técnicas de compilação melhoraram e os pipelines das máquinas aumentam de complexidade ([Capítulo 4](#)).

O último motivo para usar assembly é que não existe uma linguagem de alto nível disponível em um computador específico. Computadores muito antigos ou especializados não possuem um compilador, de modo que a única alternativa de um programador é o assembly.

Desvantagens da linguagem assembly

O assembly possui muitas desvantagens, que argumentam fortemente contra seu uso generalizado. Talvez sua principal desvantagem seja que os programas escritos em assembly são inherentemente específicos à máquina e precisam ser reescritos para serem executados em outra arquitetura de computador. A rápida evolução dos computadores, discutida no [Capítulo 1](#), significa que as arquiteturas se tornam obsoletas. Um programa em assembly permanece firmemente ligado à sua arquitetura original, mesmo depois que o computador for substituído por máquinas mais novas, mais rápidas e mais econômicas.

Outra desvantagem é que os programas em assembly são maiores do que os programas equivalentes escritos em uma linguagem de alto nível. Por exemplo, o programa em C da [Figura A.1.5](#) possui 11 linhas de extensão, enquanto o programa em assembly da [Figura A.1.4](#) possui 31 linhas. Em programas mais complexos, a razão entre o assembly e a linguagem de alto nível (seu *fator de expansão*) pode ser muito maior do que o fator de três nesse exemplo. Infelizmente, estudos empíricos mostraram que os programadores escrevem quase o mesmo número de linhas de código por dia em assembly e em linguagens de alto nível. Isso significa que os programadores são aproximadamente x vezes mais produtivos em uma linguagem de alto nível, onde x é o fator de expansão do assembly.

Para aumentar o problema, programas maiores são mais difíceis de ler e entender e contêm mais bugs. O assembly realça esse problema, devido à sua completa falta de estrutura. Os idiomas de programação comuns, como instruções *if-then* e loops, precisam ser criados a partir de desvios e jumps. Os programas resultantes são difíceis de ler, pois o leitor precisa recrivar cada construção de nível mais alto a partir de suas partes, e cada ocorrência de uma

instrução pode ser ligeiramente diferente. Por exemplo, veja a [Figura A.1.4](#) e responda a estas perguntas: que tipo de loop é utilizado? Quais são seus limites inferior e superior?

Detalhamento

Os compiladores podem produzir linguagem de máquina diretamente, em vez de contar com um assembler. Esses compiladores executam muito mais rapidamente do que aqueles que invocam um assembler como parte da compilação. Todavia, um compilador que gera linguagem de máquina precisa realizar muitas das tarefas que um assembler normalmente trata, como resolver endereços e codificar instruções como números binários. A escolha é entre velocidade de compilação e simplicidade do compilador.

Detalhamento

Apesar dessas considerações, algumas aplicações embutidas são escritas em uma linguagem de alto nível. Muitas dessas aplicações são programas grandes e complexos, que precisam ser muito confiáveis. Os programas em assembly são maiores e mais difíceis de escrever e ler do que os programas em linguagem de alto nível. Isso aumenta bastante o custo da escrita de um programa em assembly e torna muito difícil verificar a exatidão desse tipo de programa. Na verdade, essas considerações levaram o Departamento de Defesa dos EUA, que paga por muitos sistemas embutidos complexos, a desenvolver a Ada, uma nova linguagem de alto nível para a escrita de sistemas embutidos.

A.2. Assemblers

Um assembler traduz um arquivo de instruções em assembly para um arquivo de instruções de máquinas binárias e dados binários. O processo de tradução possui duas etapas principais. A primeira etapa é encontrar locais de memória com rótulos, de modo que o relacionamento entre os nomes simbólicos e endereços seja conhecido quando as instruções forem traduzidas. A segunda etapa é traduzir cada instrução assembly combinando os equivalentes numéricos dos opcodes (códigos de operação), especificadores de registradores e rótulos em uma instrução válida. Como vemos na [Figura A.1.1](#), o assembler produz um arquivo de saída, chamado de *arquivo-objeto*, que contém as instruções de máquina, dados e informações de manutenção.

Um arquivo-objeto normalmente não pode ser executado porque referencia procedimentos ou dados em outros arquivos. Um **rótulo** é **externo** (também chamado **global**) se o objeto rotulado puder ser referenciado a partir de arquivos diferentes de onde está definido. Um rótulo é *local* se o objeto só puder ser usado dentro do arquivo em que está definido. Na maior parte dos assemblers, os rótulos são locais por padrão e precisam ser declarados como globais explicitamente. As sub-rotinas e variáveis globais exigem rótulos externos, pois são referenciados a partir de muitos arquivos em um programa. **Rótulos locais** ocultam nomes que não devem ser visíveis a outros módulos — por exemplo, funções estáticas em C, que só podem ser chamadas por outras funções no mesmo arquivo. Além disso, nomes gerados pelo compilador — por exemplo, um nome para a instrução no início de um loop — são locais, de modo que o compilador não precisa produzir nomes exclusivos em cada arquivo.

rótulo externo

Também chamado **rótulo global**. Um rótulo que se refere a um objeto que pode ser referenciado a partir de arquivos diferentes daquele em que está definido.

rótulo local

Um rótulo que se refere a um objeto que só pode ser usado dentro do arquivo em que está definido.

Rótulos locais e globais

Exemplo

Considere o programa na Figura A.1.4. A sub-rotina possui um rótulo externo (global) `main`. Ela também contém dois rótulos locais — `loop` e `str` — visíveis apenas dentro do seu arquivo em assembly. Finalmente, a rotina também contém uma referência não resolvida a um rótulo externo `printf`, que é a rotina da biblioteca que exibe valores. Quais rótulos na Figura A.1.4 poderiam ser referenciados a partir de outro arquivo?

Resposta

Somente os rótulos globais são visíveis fora de um arquivo, de modo que o único rótulo que poderia ser referenciado por outro arquivo é `main`.

Como o assembler processa cada arquivo em um programa individual e isoladamente, ele só sabe os endereços dos rótulos locais. O assembler depende de outra ferramenta, o link-editor, para combinar uma coleção de arquivos-objeto e bibliotecas em um arquivo executável, resolvendo os rótulos externos. O assembler auxilia o link-editor, oferecendo listas de rótulos e referências não resolvidas.

No entanto, até mesmo os rótulos locais apresentam um desafio interessante a um assembler. Ao contrário dos nomes na maioria das linguagens de alto nível, os rótulos em assembly podem ser usados antes de serem definidos. No exemplo, na [Figura A.1.4](#), o rótulo `str` é usado pela instrução `la` antes de ser definido. A possibilidade de uma **referência à frente**, como essa, força o assembler a traduzir um programa em duas etapas: primeiro encontre todos os rótulos e depois produza as instruções. No exemplo, quando o assembler vê a instrução `la`, ele não sabe onde a palavra rotulada com `str` está localizada ou mesmo se `str` rotula uma instrução ou um dado.

referência à frente

Um rótulo usado antes de ser definido.

A primeira passada de um assembler lê cada linha de um arquivo em assembly e a divide em seus componentes. Essas partes, chamadas *lexemas*, são palavras,

números e caracteres de pontuação individuais. Por exemplo, a linha

ble \$t0, 100, loop

contém seis lexemas: o opcode `ble`, o especificador de registrador `$t0`, uma vírgula, o número `100`, uma vírgula e o símbolo `loop`.

Se uma linha começa com um rótulo, o assembler registra em sua **tabela de símbolos** o nome do rótulo e o endereço da palavra de memória que a instrução ocupa. O assembler, então, calcula quantas palavras de memória ocupará a instrução na linha atual. Acompanhando os tamanhos das instruções, o assembler pode determinar onde a próxima instrução entrará. Para calcular o tamanho de uma instrução de tamanho variável, como aquelas no VAX, um assembler precisa examiná-la em detalhes. Por outro lado, instruções de tamanho fixo, como aquelas no MIPS, exigem apenas um exame superficial. O assembler realiza um cálculo semelhante para estabelecer o espaço exigido para instruções de dados. Quando o assembler atinge o final de um arquivo assembly, a tabela de símbolos registra o local de cada rótulo definido no arquivo.

tabela de símbolos

Uma tabela que faz a correspondência entre os nomes dos rótulos e os endereços das palavras de memória que as instruções ocupam.

O assembler utiliza as informações na tabela de símbolos durante uma segunda passada pelo arquivo, que, na realidade, produz o código de máquina. O assembler novamente examina cada linha no arquivo. Se a linha contém uma instrução, o assembler combina as representações binárias de seu opcode e operandos (especificadores de registradores ou endereço de memória) em uma instrução válida. O processo é semelhante ao usado na Seção 2.5 do [Capítulo 2](#). As instruções e as palavras de dados que referenciam um símbolo externo definido em outro arquivo não podem ser completamente montadas (elas não estão resolvidas) porque o endereço do símbolo não está na tabela de símbolos. Um assembler não reclama sobre referências não resolvidas, porque o rótulo correspondente provavelmente estará definido em outro arquivo.

Colocando em perspectiva

Assembly é uma linguagem de programação. Sua principal diferença das linguagens de alto nível, como BASIC, Java e C, é que o assembly oferece apenas alguns tipos simples de dados e fluxo de controle. Os programas em assembly não especificam o tipo do valor mantido em uma variável. Em vez disso, um programador precisa aplicar as operações apropriadas (por exemplo, adição de inteiro ou ponto flutuante) a um valor. Além disso, em assembly, os programas precisam implementar todo o fluxo de controle com instruções do tipo “*go to*”. Os dois fatores tornam a programação em assembly para qualquer máquina — MIPS ou x86 — mais difícil e passível de erro do que a escrita em uma linguagem de alto nível.

Detalhamento

Se a velocidade de um assembler for importante, esse processo em duas etapas pode ser feito em uma passada pelo arquivo assembly com uma técnica conhecida como **backpatching**. Em sua passada pelo arquivo, o assembler monta uma representação binária (possivelmente incompleta) de cada instrução. Se a instrução referencia um rótulo ainda não definido, o assembler consulta essa tabela para encontrar todas as instruções que contêm uma referência à frente ao rótulo. O assembler volta e corrige sua representação binária para incorporar o endereço do rótulo. O backpatching agiliza o assembly porque o assembler só lê sua entrada uma vez. Contudo, isso exige que um assembler mantenha uma representação binária inteira de um programa na memória, de modo que as instruções possam sofrer backpatching. Esse requisito pode limitar o tamanho dos programas que podem ser montados. O processo é complicado por máquinas com diversos tipos de desvios que se espalham por diferentes intervalos de instruções. Quando o assembler vê inicialmente um rótulo não resolvido em uma instrução de desvio, ele precisa usar o maior desvio possível ou arriscar ter de voltar e reajustar muitas instruções para criar espaço em um desvio maior.

backpatching

Um método para traduzir do assembly para instruções de máquina, em que o assembler tem uma representação binária (possivelmente incompleta) de cada instrução em uma passada por um programa e depois retorna para preencher

rótulos previamente indefinidos.

Formato do arquivo-objeto

Os assemblers produzem arquivos-objeto. Um arquivo-objeto no UNIX contém seis seções distintas (veja [Figura A.2.1](#)):

- O *cabeçalho do arquivo-objeto* descreve o tamanho e a posição das outras partes do arquivo.
- O **segmento de texto** contém o código em linguagem de máquina para rotinas no arquivo-fonte. Essas rotinas podem ser não executáveis devido a referências não resolvidas.

Cabeçalho do arquivo-objeto	Segmento de texto	Segmento de dados	Informações de relocação	Tabela de símbolos	Informações de depuração
-----------------------------	-------------------	-------------------	--------------------------	--------------------	--------------------------

FIGURA A.2.1 Arquivo-objeto.

Um assembler do UNIX produz um arquivo-objeto com seis seções distintas.

segmento de texto

O segmento de um arquivo-objeto do UNIX que contém o código em linguagem de máquina para as rotinas no arquivo-fonte.

- O **segmento de dados** contém uma representação binária dos dados no arquivo-fonte. Os dados também podem estar incompletos devido a referências não resolvidas a rótulos em outros arquivos.

segmento de dados

O segmento de um objeto ou arquivo executável do UNIX que contém uma representação binária dos dados inicializados, usados pelo programa

- As **informações de relocação** identificam instruções e palavras de dados que dependem de **endereços absolutos**. Essas referências precisam mudar se partes do programa forem movidas na memória.

informações de relocação

O segmento de um arquivo-objeto do UNIX que identifica instruções e palavras de dados que dependem de endereços absolutos.

endereço absoluto

O endereço real na memória de uma variável ou rotina.

- A *tabela de símbolos* associa endereços a rótulos externos no arquivo-fonte e lista referências não resolvidas.
- As *informações de depuração* contêm uma descrição concisa da maneira como o programa foi compilado, de modo que um depurador possa descobrir quais endereços de instrução correspondem às linhas em um arquivo-fonte e exibir as estruturas de dados em formato legível.

O assembler produz um arquivo-objeto que contém uma representação binária do programa e dos dados, além de informações adicionais para ajudar a ligar as partes de um programa. Essas informações de relocação são necessárias porque o assembler não sabe quais locais da memória um procedimento ou parte de dados ocupará depois de ser ligado ao restante do programa. Os procedimentos e dados de um arquivo são armazenados em uma parte contígua da memória, mas o assembler não sabe onde essa memória estará localizada. O assembler também passa algumas entradas da tabela de símbolos para o link-editor. Em particular, o assembler precisa registrar quais símbolos externos são definidos em um arquivo e quais referências não resolvidas ocorrem em um arquivo.

Detalhamento

Por conveniência, os assemblers consideram que cada arquivo começa no mesmo endereço (por exemplo, posição 0) com a expectativa de que o link-editor *reposicione* o código e os dados quando forem designados seus locais na memória. O assembler produz *informações de relocação*, que contêm uma entrada descrevendo cada instrução ou palavra de dados no arquivo que referencia um endereço absoluto. No MIPS, somente as instruções call, load e store da sub-rotina referenciam endereços absolutos. As instruções que usam endereçamento relativo ao PC, como desvios, não precisam ser relocadas.

Facilidades adicionais

Os assemblers oferecem diversos recursos convenientes que ajudam a tornar os programas em assembly mais curtos e mais fáceis de escrever, mas não mudam fundamentalmente o assembly. Por exemplo, as *diretivas de layout de dados* permitem que um programador descreva os dados de uma maneira mais concisa e natural do que sua representação binária.

Na [Figura A.1.4](#), a diretiva

```
.asciiz "The sum from 0 .. 100 is %d\n"
```

armazena caracteres da string na memória. Compare essa linha com a alternativa de escrever cada caractere como seu valor ASCII (a [Figura 2.15](#), no [Capítulo 2](#), descreve a codificação ASCII para os caracteres):

```
.byte 84, 104, 101, 32, 115, 117, 109, 32  
.byte 102, 114, 111, 109, 32, 48, 32, 46  
.byte 46, 32, 49, 48, 48, 32, 105, 115  
.byte 32, 37, 100, 10, 0
```

A diretiva `.asciiz` é mais fácil de ler porque representa caracteres como letras, e não como números binários. Um assembler pode traduzir caracteres para sua representação binária muito mais rapidamente e com mais precisão do que um ser humano. As diretivas de layout de dados especificam os dados em um formato legível aos seres humanos, que um assembler traduz para binário. Outras diretivas de layout são descritas na [Seção A.10](#).

Diretiva de string

Exemplo

Defina a sequência de bytes produzida por esta diretiva:

```
.asciiz "The quick brown fox jumps over the lazy dog"
```

Resposta

```
.byte 84, 104, 101, 32, 113, 117, 105, 99  
.byte 107, 32, 98, 114, 111, 119, 110, 32  
.byte 102, 111, 120, 32, 106, 117, 109, 112  
.byte 115, 32, 111, 118, 101, 114, 32, 116  
.byte 104, 101, 32, 108, 97, 122, 121, 32  
.byte 100, 111, 103, 0
```

As *macros* são uma facilidade de combinação e troca de padrão, que oferece um mecanismo simples para nomear uma sequência de instruções usada com frequência. Em vez de digitar repetidamente as mesmas instruções toda vez que forem usadas, um programador chama a macro e o assembler substitui a chamada da macro pela sequência de instruções correspondente. As macros, como as sub-rotinas, permitem que um programador crie e nomeie uma nova abstração para uma operação comum. No entanto, diferente das sub-rotinas, elas não causam uma chamada e um retorno de sub-rotina quando o programa é executado, pois uma chamada de macro é substituída pelo corpo da macro quando o programa é montado. Depois dessa troca, a montagem resultante é indistinguível do programa equivalente, escrito sem macros.

Macros

Exemplo

Como um exemplo, suponha que um programador precise exibir muitos números. A rotina de biblioteca `printf` aceita uma string de formato e um ou mais valores para exibir como seus argumentos. Um programador poderia exibir o inteiro no registrador \$7 com as seguintes instruções:

```
.data
int_str: .ascii " %d"
.text
    la    $a0, int_str # Carrega endereço da string
                      # no primeiro argumento
```

```
    mov   $a1, $7  # Carrega valor no
                  # segundo argumento
    jal   printf  # Chama a rotina printf
```

A diretiva `.data` diz ao assembler para armazenar a string no segmento de dados do programa, e a diretiva `.text` diz ao assembler para armazenar as instruções em seu segmento de texto.

Entretanto, a exibição de muitos números dessa maneira é tediosa e produz um programa extenso, difícil de ser entendido. Uma alternativa é introduzir uma macro, `print_int`, para exibir um inteiro:

```
.data
int_str: .ascii " %d"
.text
.macro print_int($arg)
    la    $a0, int_str # Carrega endereço da string
                      # no primeiro argumento
    mov   $a1, $arg   # Carrega parâmetro da macro
                      # ($arg) no segundo argumento
    jal   printf    # Chama a rotina printf
.end_macro
print_int($7)
```

A macro possui um **parâmetro formal**, `$arg`, que nomeia o argumento da macro. Quando a macro é expandida, o argumento de uma chamada é substituído pelo parâmetro formal em todo o corpo da macro. Depois, o assembler substitui a chamada pelo corpo recém-expandido da macro. Na

primeira chamada em `print_int`, o argumento é `$7`, de modo que a macro se expande para o código

parâmetro formal

Uma variável que é o argumento de um procedimento ou macro; ela é substituída por esse argumento quando a macro é expandida.

```
la    $a0, int_str
mov   $a1, $7
jal   printf
```

Em uma segunda chamada a `print_int`, digamos, `print_int($t0)`, o argumento é `$t0`, de modo que a macro expande para:

```
la    $a0, int_str
mov   $a1, $t0
jal   printf
```

Para o que a chamada `print_int($a0)` se expande?

Resposta

```
la $a0, int_str
mov $a1, $a0
jal printf
```

Esse exemplo ilustra uma desvantagem das macros. Um programador que utiliza essa macro precisa estar ciente de que `print_int` utiliza o registrador `$a0` e por isso não pode exibir corretamente o valor nesse registrador.

Interface hardware/software

Alguns assemblers também implementam *pseudoinstruções*, que são instruções fornecidas por um assembler mas não implementadas no hardware. O Capítulo 2 contém muitos exemplos de como o assembler MIPS sintetiza pseudoinstruções e modos de endereçamento do conjunto de instruções de hardware do MIPS. Por exemplo, a Seção 2.7, no Capítulo 2, descreve como o assembler sintetiza a instrução `blt`, a partir de duas outras instruções: `slt` e `bne`. Estendendo o conjunto de instruções, o assembler MIPS torna a programação em assembly mais fácil sem complicar o hardware. Muitas pseudoinstruções também poderiam ser simuladas com macros, mas o assembler MIPS pode gerar um código melhor para essas instruções, pois pode usar um registrador dedicado (`$at`) e é capaz de otimizar o código gerado.

Detalhamento

Os assemblers *montam condicionalmente* partes de código, o que permite que um programador inclua ou exclua grupos de instruções quando um programa é montado. Esse recurso é particularmente útil quando várias versões de um programa diferem por um pequeno valor. Em vez de manter esses programas em arquivos separados — o que complica bastante o reparo de bugs no código comum —, os programadores normalmente mesclam as versões em um único arquivo. O código particular a uma versão é montado condicionalmente, de modo que possa ser excluído quando outras versões do programa forem montadas.

Se as macros e a montagem condicional são tão úteis, por que os assemblers para sistemas UNIX nunca ou quase nunca as oferecem? Um motivo é que a maioria dos programadores nesses sistemas escreve programas em linguagens de mais alto nível, como C. A maior parte do código assembly é produzida por compiladores, que acham mais conveniente repetir o código do que definir macros. Outro motivo é que outras ferramentas no UNIX — como `cpp`, o pré-processador C, ou `m4`, um processador de macro de uso geral — podem oferecer macros e montagem condicional para programas em assembly.

A.3. Link-editores

A **compilação separada** permite que um programa seja dividido em partes que são armazenadas em arquivos diferentes. Cada arquivo contém uma coleção logicamente relacionada de sub-rotinas e estruturas de dados que formam um *módulo* de um programa maior. Um arquivo pode ser compilado e montado independente de outros arquivos, de modo que as mudanças em um módulo não exigem a recompilação do programa inteiro. Conforme já discutimos, a compilação separada necessita da etapa adicional de link-edição para combinar os arquivos-objeto de módulos separados e consertar suas referências não resolvidas.

compilação separada

Dividir um programa em muitos arquivos, cada qual podendo ser compilado sem conhecimento do que está nos outros arquivos.

A ferramenta que mescla esses arquivos é o *link-editor* (veja [Figura A.3.1](#)). Ele realiza três tarefas:

- Pesquisa as bibliotecas de programa para encontrar rotinas de biblioteca usadas pelo programa.
- Determina os locais da memória que o código de cada módulo ocupará e realoca suas instruções ajustando referências absolutas.
- Resolve referências entre os arquivos.

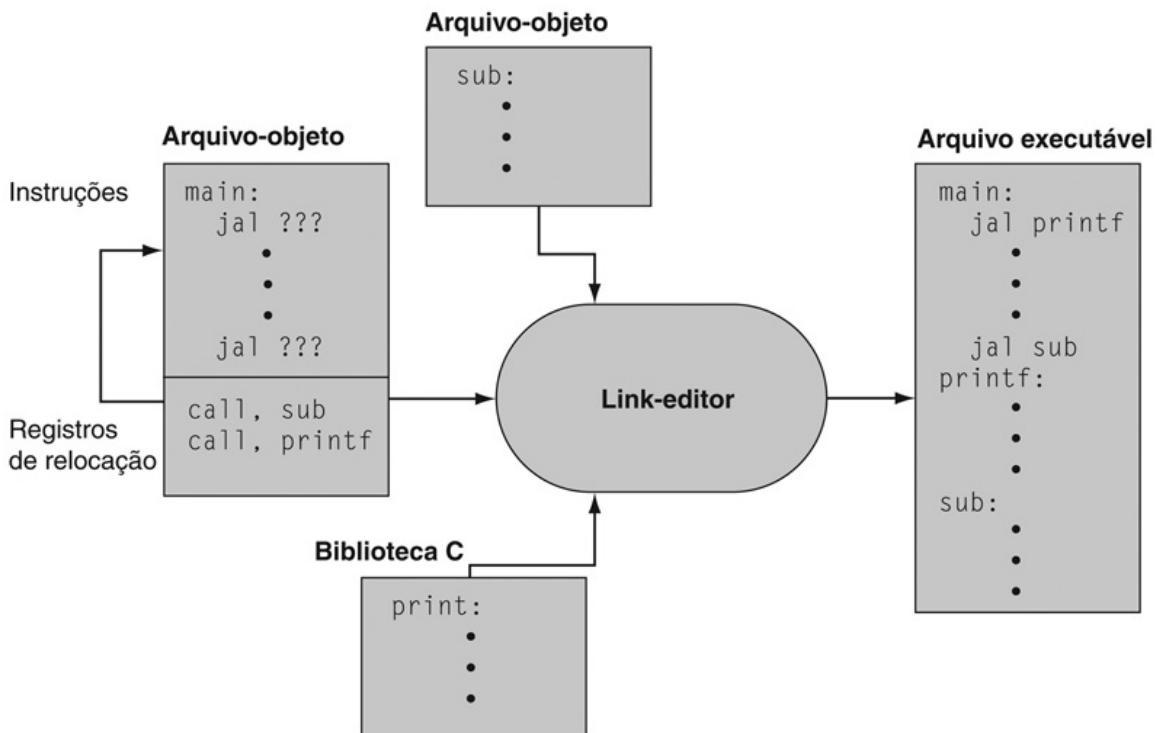


FIGURA A.3.1 O link-editor pesquisa uma coleção de arquivos-objeto e bibliotecas de programa para encontrar rotinas não locais usadas em um programa, combina-as em um único arquivo executável e resolve as referências entre as rotinas em arquivos diferentes.

A primeira tarefa de um link-editor é garantir que um programa não contenha rótulos indefinidos. O link-editor combina os símbolos externos e as referências não resolvidas, a partir dos arquivos de um programa. Um símbolo externo em um arquivo resolve uma referência de outro arquivo se ambos se referirem a um rótulo com o mesmo nome. As referências não combinadas significam que um símbolo foi usado, mas não definido em qualquer lugar do programa.

Referências não resolvidas nesse estágio do processo de link-edição não necessariamente significam que um programador cometeu um erro. O programa poderia ter referenciado uma rotina de biblioteca cujo código não estava nos arquivos-objeto passados ao link-editor. Depois de combinar os símbolos no programa, o link-editor pesquisa as bibliotecas de programa do sistema para encontrar sub-rotinas e estruturas de dados predefinidas que o programa referencia. As bibliotecas básicas contêm rotinas que leem e escrevem dados, alocam e liberam memória e realizam operações numéricas. Outras bibliotecas contêm rotinas para acessar bancos de dados ou manipular janelas de terminal. Um programa que referencia um símbolo não resolvido que não está em

qualquer biblioteca é errôneo e não pode ser link-editado. Quando o programa usa uma rotina de biblioteca, o link-editor extrai o código de rotina da biblioteca e o incorpora ao segmento de texto do programa. Essa nova rotina, por sua vez, pode depender de outras rotinas de biblioteca, de modo que o link-editor continua buscando outras rotinas de biblioteca até que nenhuma referência externa esteja não resolvida ou até que uma rotina não possa ser encontrada.

Se todas as referências externas forem resolvidas, o link-editor em seguida determina os locais da memória que cada módulo ocupará. Como os arquivos foram montados isoladamente, o assembler não poderia saber onde as instruções ou os dados de um módulo seriam colocados em relação a outros módulos. Quando o link-editor coloca um módulo na memória, todas as referências absolutas precisam ser *relocadas* para refletir seu verdadeiro local. Como o link-editor possui informações de relocação que identificam todas as referências relocáveis, ele pode eficientemente localizar e remendar essas referências.

O link-editor produz um arquivo executável que pode ser executado em um computador. Normalmente, esse arquivo tem o mesmo formato de um arquivo-objeto, exceto que não contém referências não resolvidas ou informações de relocação.

A.4. Carregando

Um programa que link-edita sem um erro pode ser executado. Antes de ser executado, o programa reside em um arquivo no armazenamento secundário, como um disco. Em sistemas UNIX, o kernel do sistema operacional traz o programa para a memória e inicia sua execução. Para iniciar um programa, o sistema operacional realiza as seguintes etapas:

1. Lê o cabeçalho do arquivo executável para determinar o tamanho dos segmentos de texto e de dados.
2. Cria um novo espaço de endereçamento para o programa. Esse espaço de endereçamento é grande o suficiente para manter os segmentos de texto e de dados, junto com um segmento de pilha (veja a [Seção A.5](#)).
3. Copia instruções e dados do arquivo executável para o novo espaço de endereçamento.
4. Copia argumentos passados ao programa para a pilha.
5. Inicializa os registradores da máquina. Em geral, a maioria dos registradores é apagada, mas o stack pointer precisa receber o endereço do primeiro local da pilha livre (veja a [Seção A.5](#)).
6. Desvia para a rotina de partida, que copia os argumentos do programa da pilha para os registradores e chama a rotina `main` do programa. Se a rotina `main` retornar, a rotina de partida termina o programa com a chamada do sistema `exit`.

A.5. Uso da memória

As próximas seções elaboram a descrição da arquitetura MIPS apresentada anteriormente no livro. Os capítulos anteriores focaram principalmente no hardware e seu relacionamento com o software de baixo nível. Essas seções tratavam principalmente de como os programadores assembly utilizam o hardware do MIPS. Essas seções descrevem um conjunto de convenções seguido em muitos sistemas MIPS. Em sua maior parte, o hardware não impõe essas convenções. Em vez disso, elas representam um acordo entre os programadores para seguirem o mesmo conjunto de regras, de modo que o software escrito por diferentes pessoas possa atuar junto e fazer uso eficaz do hardware MIPS.

Os sistemas baseados em processadores MIPS, normalmente, dividem a memória em três partes (veja [Figura A.5.1](#)). A primeira parte, próxima do início do espaço de endereçamento (começando no endereço 400000_{hexa}), é o *segmento de texto*, que mantém as instruções do programa.

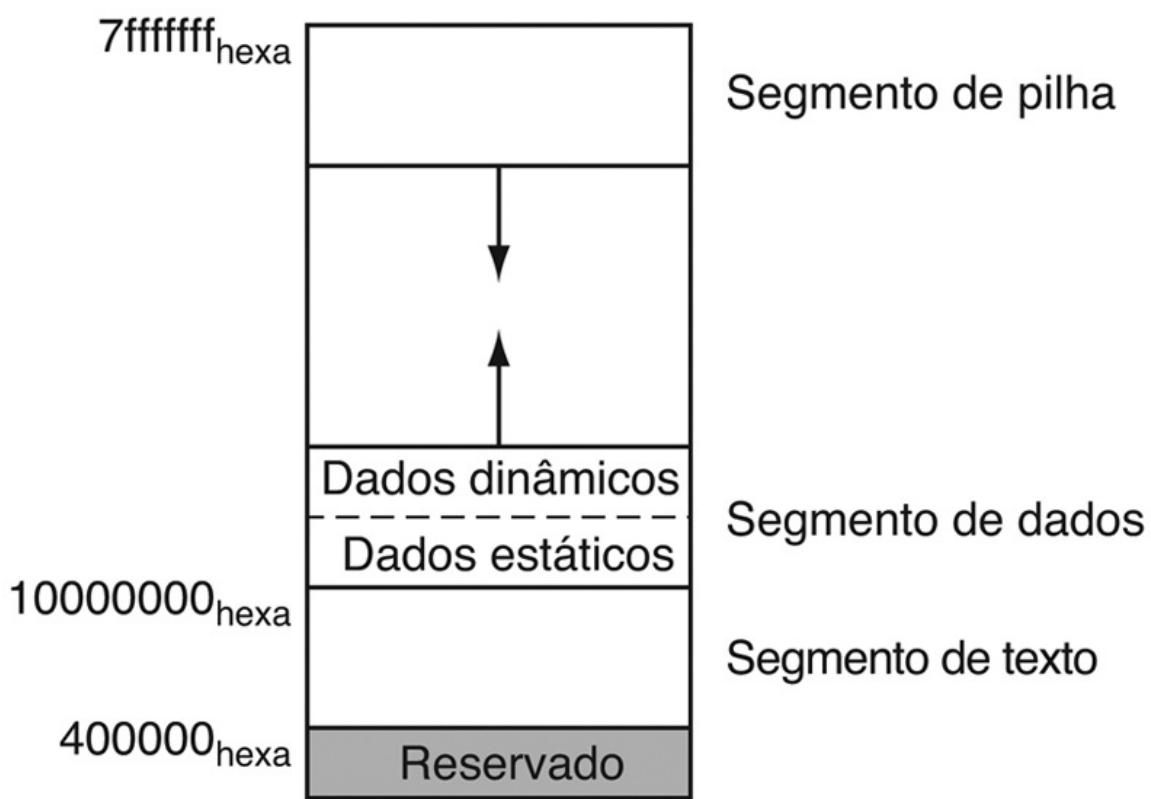


FIGURA A.5.1 Layout da memória.

A segunda parte, acima do segmento de texto, é o *segmento de dados*, dividido em mais duas partes. Os **dados estáticos** (começando no endereço 10000000_{hexa}) contêm objetos cujo tamanho é conhecido pelo compilador e cujo tempo de vida — o intervalo durante o qual um programa pode acessá-los — é a execução inteira do programa. Por exemplo, em C, as variáveis globais são alocadas estaticamente, pois podem ser referenciadas, a qualquer momento, durante a execução de um programa. O link- editor atribui objetos estáticos a locais no segmento de dados e resolve referências a esses objetos.

dados estáticos

A parte da memória que contém dados cujo tamanho é conhecido pelo compilador e cujo tempo de vida é a execução inteira do programa.

Interface hardware/software

Como o segmento de dados começa muito acima do programa, no endereço 10000000_{hexa} , as instruções load e store não podem referenciar diretamente os objetos de dados com seus campos de offset de 16 bits (veja Seção 2.5, no Capítulo 2). Por exemplo, para carregar a palavra no segmento de dados no endereço 10010020_{hexa} para o registrador \$v0, são necessárias duas instruções:

```
lui    $s0, 0x1001 # 0x1001 significa 1001 base 16  
lw     $v0, 0x0020($s0) # 0x10010000 + 0x0020 = 0x10010020
```

(O $0x$ antes de um número significa que ele é um valor hexadecimal. Por exemplo, $0x800$ é 8000_{hexa} ou $32.768_{\text{dec.}}$)

Para evitar repetir a instrução lui em cada load e store, os sistemas MIPS normalmente dedicam um registrador (\$gp) como um *ponteiro global* para o segmento de dados estático. Esse registrador contém o endereço 10008000_{hexa} , de modo que as instruções load e store podem usar seus campos de 16 bits com sinal para acessar os primeiros 64KB do segmento de dados estático. Com esse ponteiro global, podemos reescrever o exemplo como uma única instrução:

```
lw    $v0 , 0x8020($gp)
```

Naturalmente, um ponteiro global torna os locais de endereçamento entre $10000000_{\text{hexa}} - 10010000_{\text{hexa}}$ mais rápidos do que outros locais do heap. O compilador MIPS normalmente armazena *variáveis globais* nessa área, pois essas variáveis possuem locais fixos e se ajustam melhor do que outros dados globais, como arrays.

Imediatamente acima dos dados estáticos estão os *dados dinâmicos*. Esses dados, como seu nome sugere, são alocados pelo programa enquanto ele é executado. Nos programas C, a rotina de biblioteca `malloc` localiza e retorna um novo bloco de memória. Como um compilador não pode prever quanta memória um programa alocará, o sistema operacional expande a área de dados dinâmica para atender à demanda. Conforme indica a seta para cima na figura, `malloc` expande a área dinâmica com a chamada do sistema `sbrk`, que faz com que o sistema operacional acrescente mais páginas ao espaço de endereçamento virtual do programa (veja Seção 5.7, no [Capítulo 5](#)) imediatamente acima do segmento de dados dinâmico.

A terceira parte, o **segmento de pilha** do programa, reside no topo do espaço de endereçamento virtual (começando no endereço $7fffffff_{\text{hexa}}$). Assim como os dados dinâmicos, o tamanho máximo da pilha de um programa não é conhecido antecipadamente. À medida que o programa coloca valores na pilha, o sistema operacional expande o segmento de pilha para baixo, em direção ao segmento de dados.

segmento de pilha

A parte da memória usada por um programa para manter frames de chamada de procedimento.

Essa divisão de três partes da memória não é a única possível. Contudo, ela possui duas características importantes: os dois segmentos dinamicamente expansíveis são bastante distantes um do outro e eles podem crescer para usar o espaço de endereços inteiro de um programa.

A.6. Convenção para chamadas de procedimento

As convenções que controlam o uso dos registradores são necessárias quando os procedimentos em um programa são compilados separadamente. Para compilar um procedimento em particular, um compilador precisa saber quais registradores pode usar e quais são reservados para outros procedimentos. As regras para usar os registradores são chamadas de **convenções para uso dos registradores** ou **convenções para chamadas de procedimento**. Como o nome sugere, essas regras são, em sua maior parte, convenções seguidas pelo software, em vez de regras impostas pelo hardware. No entanto, a maioria dos compiladores e programadores tenta seguir essas convenções estritamente, pois sua violação causa bugs traiçoeiros.

convenção para uso dos registradores

Também chamada **convenção para chamadas de procedimento**. Um protocolo de software que controla o uso dos registradores por procedimentos.

A convenção para chamadas descrita nesta seção é aquela utilizada pelo compilador gcc. O compilador nativo do MIPS utiliza uma convenção mais complexa, que é ligeiramente mais rápida.

A CPU do MIPS contém 32 registradores de uso geral, numerados de 0 a 31. O registrador \$0 contém o valor fixo 0.

- Os registradores \$at (1), \$k0 (26) e \$k1 (27) são reservados para o assembler e o sistema operacional e não devem ser usados por programas do usuário ou compiladores.
- Os registradores \$a0—\$a3 (4-7) são usados para passar os quatro primeiros argumentos às rotinas (os argumentos restantes são passados na pilha). Os registradores \$v0 e \$v1 (2, 3) são usados para retornar valores das funções.
- Os registradores \$t0—\$t9 (8-15, 24, 25) são **registradores salvos pelo caller**, usados para manter quantidades temporárias que não precisam ser preservadas entre as chamadas (veja Seção 2.8, no [Capítulo 2](#)).
- Os registradores \$s0—\$s7 (16-23) são **registradores salvos pelo callee**, que mantêm valores de longa duração os quais devem ser preservados entre as

chamadas.

- O registrador \$gp (28) é um ponteiro global que aponta para o meio de um bloco de 64K de memória no segmento de dados estático.
- O registrador \$sp (29) é o stack pointer, que aponta para o último local na pilha. O registrador \$fp (30) é o frame pointer. A instrução `jal` escreve no registrador \$ra (31) o endereço de retorno de uma chamada de procedimento. Esses dois registradores são explicados na próxima seção.

registrador salvo pelo caller

Um registrador salvo pela rotina que faz uma chamada de procedimento.

registrador salvo pelo callee

Um registrador salvo pela rotina sendo chamada.

As abreviações e os nomes de duas letras para esses registradores — por exemplo, \$sp para o stack pointer — refletem os usos intencionados na convenção de chamada de procedimento. Ao descrever essa convenção, usaremos os nomes em vez de números de registrador. A [Figura A.6.1](#) lista os registradores e descreve seus usos intencionados.

Nome do registrador	Número	Uso
\$zero	0	constante 0
\$at	1	reservado para o montador
\$v0	2	avaliação de expressão e resultados de uma função
\$v1	3	avaliação de expressão e resultados de uma função
\$a0	4	argumento 1
\$a1	5	argumento 2
\$a2	6	argumento 3
\$a3	7	argumento 4
\$t0	8	temporário (não preservado pela chamada)
\$t1	9	temporário (não preservado pela chamada)
\$t2	10	temporário (não preservado pela chamada)
\$t3	11	temporário (não preservado pela chamada)
\$t4	12	temporário (não preservado pela chamada)
\$t5	13	temporário (não preservado pela chamada)
\$t6	14	temporário (não preservado pela chamada)
\$t7	15	temporário (não preservado pela chamada)
\$s0	16	temporário salvo (preservado pela chamada)
\$s1	17	temporário salvo (preservado pela chamada)
\$s2	18	temporário salvo (preservado pela chamada)
\$s3	19	temporário salvo (preservado pela chamada)
\$s4	20	temporário salvo (preservado pela chamada)
\$s5	21	temporário salvo (preservado pela chamada)
\$s6	22	temporário salvo (preservado pela chamada)
\$s7	23	temporário salvo (preservado pela chamada)
\$t8	24	temporário (não preservado pela chamada)
\$t9	25	temporário (não preservado pela chamada)
\$k0	26	reservado para o kernel do sistema operacional
\$k1	27	reservado para o kernel do sistema operacional
\$gp	28	ponteiro para área global
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	endereço de retorno (usado por chamada de função)

FIGURA A.6.1 Registradores do MIPS e convenção de uso.

Chamadas de procedimento

Esta seção descreve as etapas que ocorrem quando um procedimento (*caller*) invoca outro procedimento (*callee*). Os programadores que escrevem em uma linguagem de alto nível (como C ou Pascal) nunca veem os detalhes de como um procedimento chama outro, pois o compilador cuida dessa manutenção de baixo

nível. Contudo, os programadores assembly precisam implementar explicitamente cada chamada e retorno de procedimento.

A maior parte da manutenção associada a uma chamada gira em torno de um bloco de memória chamado **frame de chamada de procedimento**. Essa memória é usada para diversas finalidades:

frame de chamada de procedimento

Um bloco de memória usado para manter valores passados a um procedimento como argumentos, a fim de salvar registradores que um procedimento pode modificar mas que o caller não deseja que sejam alterados, e fornecer espaço para variáveis locais a um procedimento.

- Para manter valores passados a um procedimento como argumentos.
- Para salvar registradores que um procedimento pode modificar, mas que o caller não deseja que sejam alterados.
- Para oferecer espaço para variáveis locais a um procedimento.

Na maioria das linguagens de programação, as chamadas e retornos de procedimento seguem uma ordem estrita do tipo último a entrar, primeiro a sair (LIFO — Last-In, First-Out), de modo que essa memória pode ser alocada e liberada como uma pilha, motivo pelo qual esses blocos de memória às vezes são chamados frames de pilha.

A [Figura A.6.2](#) mostra um frame de pilha típico. O frame consiste na memória entre o frame pointer (\$fp), que aponta para a primeira palavra do frame, e o stack pointer (\$sp), que aponta para a última palavra do frame. A pilha cresce para baixo a partir dos endereços de memória mais altos, de modo que o frame pointer aponta para cima do stack pointer. O procedimento que está executando utiliza o frame pointer para acessar rapidamente os valores em seu frame de pilha. Por exemplo, um argumento no frame de pilha pode ser lido para o registrador \$v0 com a instrução

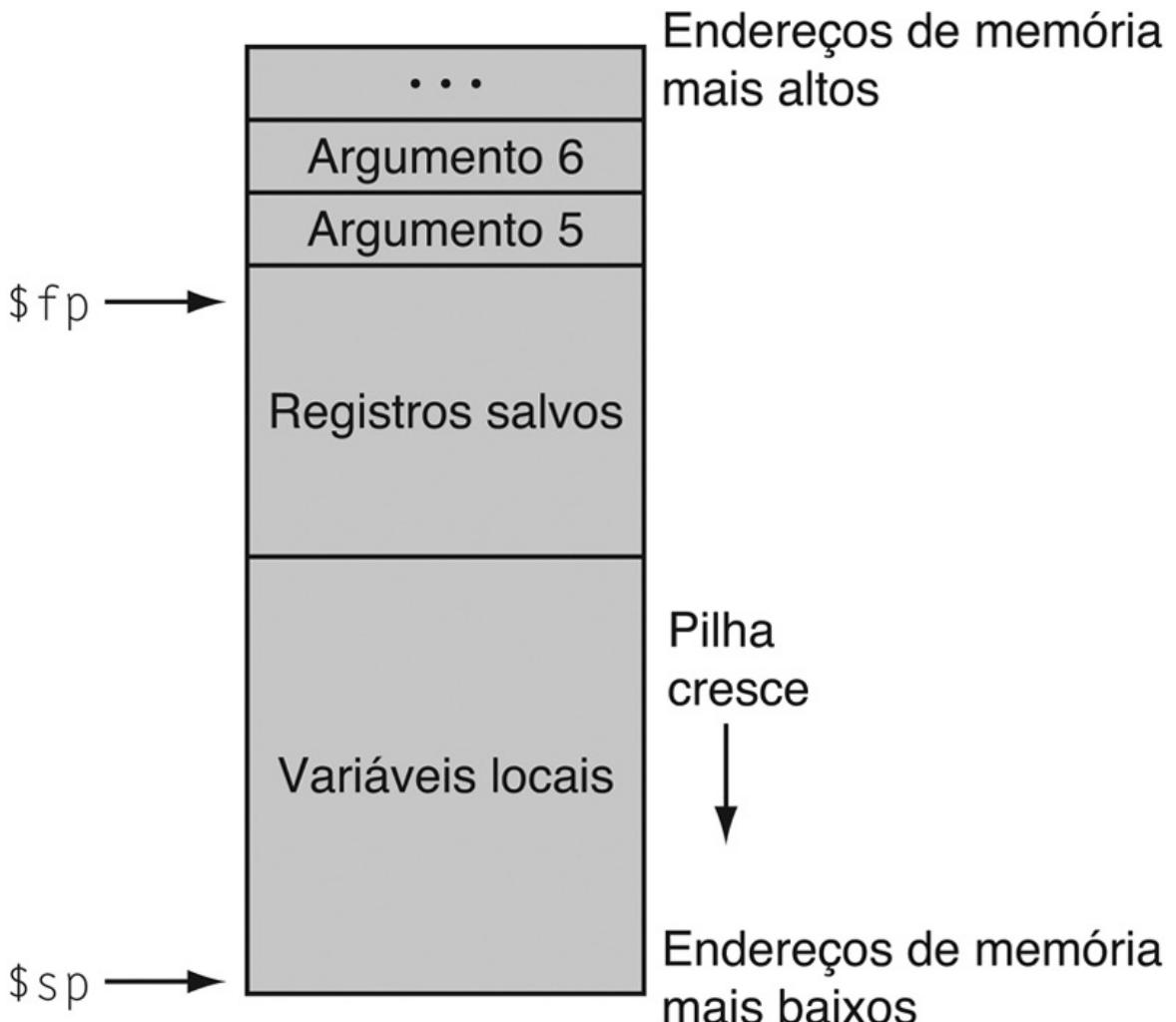


FIGURA A.6.2 Layout de um frame de pilha.

O frame pointer ($\$fp$) aponta para a primeira palavra do frame de pilha do procedimento em execução. O stack pointer ($\$sp$) aponta para a última palavra do frame. Os quatro primeiros argumentos são passados em registradores, de modo que o quinto argumento é o primeiro armazenado na pilha.

`lw $v0, 0($fp)`

Um frame de pilha pode ser construído de muitas maneiras diferentes; porém, o caller e o callee precisam combinar a sequência de etapas. As etapas a seguir descrevem a convenção de chamada utilizada na maioria das máquinas MIPS.

Essa convenção entra em ação em três pontos durante uma chamada de procedimento: imediatamente antes do caller invocar o callee, assim que o callee começa a executar e imediatamente antes do callee retornar ao caller. Na primeira parte, o caller coloca os argumentos da chamada de procedimento em locais padrões e invoca o callee para fazer o seguinte:

1. Passar argumentos. Por convenção, os quatro primeiros argumentos são passados nos registradores \$a0–\$a3. Quaisquer argumentos restantes são colocados na pilha e aparecem no início do frame de pilha do procedimento chamado.
2. Salvar registradores salvos pelo caller. O procedimento chamado pode usar esses registradores (\$a0–\$a3 e \$t0–\$t9) sem primeiro salvar seu valor. Se o caller espera utilizar um desses registradores após uma chamada, ele deverá salvar seu valor antes da chamada.
3. Executar uma instrução `jal` (veja Seção 2.8 do [Capítulo 2](#)), que desvia para a primeira instrução do callee e salva o endereço de retorno no registrador `$ra`.

Antes que uma rotina chamada comece a executar, ela precisa realizar as seguintes etapas para configurar seu frame de pilha:

1. Alocar memória para o frame, subtraindo o tamanho do frame do stack pointer.
2. Salvar os registradores salvos pelo callee no frame. Um callee precisa salvar os valores desses registradores (\$s0–\$s7, \$fp e \$ra) antes de alterá-los, pois o caller espera encontrar esses registradores inalterados após a chamada. O registrador \$fp é salvo para cada procedimento que aloca um novo frame de pilha. No entanto, o registrador \$ra só precisa ser salvo se o callee fizer uma chamada. Os outros registradores salvos pelo callee, que são utilizados, também precisam ser salvos.
3. Estabelecer o frame pointer somando o tamanho do frame de pilha menos 4 a \$sp e armazenando a soma no registrador \$fp.

Interface hardware/software

A convenção de uso dos registradores do MIPS oferece registradores salvos pelo caller e pelo callee, pois os dois tipos de registradores são vantajosos em circunstâncias diferentes. Os registradores salvos pelo caller são usados para manter valores de longa duração, como variáveis de um programa do usuário. Esses registradores só são salvos durante uma chamada de procedimento se o

caller espera utilizar o registrador. Por outro lado, os registradores salvos pelo callee são usados para manter quantidades de curta duração, que não persistem entre as chamadas, como valores imediatos em um cálculo de endereço. Durante uma chamada, o caller não pode usar esses registradores para valores temporários de curta duração.

Finalmente, o callee retorna ao caller executando as seguintes etapas:

1. Se o callee for uma função que retorna um valor, coloque o valor retornado no registrador $\$v0$.
2. Restaure todos os registradores salvos pelo callee que foram salvos na entrada do procedimento.
3. Remova o frame de pilha adicionando o tamanho do frame a $\$sp$.
4. Retorne desviando para o endereço no registrador $\$ra$.

Detalhamento

Uma linguagem de programação que não permite **procedimentos recursivos** — procedimentos que chamam a si mesmos, direta ou indiretamente, por meio de uma cadeia de chamadas — não precisa alocar frames em uma pilha. Em uma linguagem não recursiva, o frame de cada procedimento pode ser alocado estaticamente, pois somente uma invocação de um procedimento pode estar ativa ao mesmo tempo. As versões mais antigas de Fortran proibiam a recursão porque frames alocados estaticamente produziam código mais rápido em algumas máquinas mais antigas. Todavia, em arquiteturas load-store, como MIPS, os frames de pilha podem ser tão rápidos quanto, porque o registrador frame pointer aponta diretamente para o frame de pilha ativo, o que permite que uma única instrução load ou store acesse valores no frame. Além disso, a recursão é uma técnica de programação valiosa.

procedimentos recursivos

Procedimentos que chamam a si mesmos, direta ou indiretamente, por meio de uma cadeia de chamadas.

Exemplo de chamada de procedimento

Como exemplo, considere a rotina em C

```

main ()
{
    printf ("The factorial of 10 is %d\n", fact (10));
}

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

```

que calcula e exibe $10!$ (o fatorial de 10, $10! = 10 \times 9 \times \dots \times 1$). `fact` é uma rotina recursiva que calcula $n!$ multiplicando n vezes $(n - 1)!$. O código assembly para essa rotina ilustra como os programas manipulam frames de pilha.

Na entrada, a rotina `main` cria seu frame de pilha e salva os dois registradores salvos pelo callee que serão modificados: `$fp` e `$ra`. O frame é maior do que o exigido para esses dois registradores, pois a convenção de chamada exige que o tamanho mínimo de um frame de pilha seja 24 bytes. Esse frame mínimo pode manter quatro registradores de argumento (`$a0-$a3`) e o endereço de retorno `$ra`, preenchidos até um limite de dupla palavra (24 bytes). Como `main` também precisa salvar o `$fp`, seu frame de pilha precisa ter duas palavras a mais (lembre-se de que o stack pointer é mantido alinhado em um limite de dupla palavra).

```

.text
.globl main
main:
    subu $sp,$sp,32      # Frame de pilha tem 32 bytes
    sw    $ra,20($sp)    # Salva endereço de retorno
    sw    $fp,16($sp)    # Salva frame pointer antigo
    addiu $fp,$sp,28     # Prepara frame pointer

```

A rotina `main`, então, chama a rotina de fatorial e lhe passa o único argumento 10. Depois que `fact` retorna, `main` chama a rotina de biblioteca `printf` e lhe passa uma string de formato e o resultado retornado de `fact`:

```

li      $a0,10      # Coloca argumento (10) em $a0
jal    fact         # Chama função de fatorial

la      $a0,$LC      # Coloca string de formato em $a0
move   $a1,$v0      # Move resultado de fact para $a1
jal    printf       # Chama a função para exibir

```

Finalmente, depois de exibir o fatorial, `main` retorna. Entretanto, primeiro, ela precisa restaurar os registradores que salvou e remover seu frame de pilha:

```

lw      $ra,20($sp)  # Restaura endereço de retorno
lw      $fp,16($sp)  # Restaura frame pointer
addiu $sp,$sp,32     # Remove frame de pilha
jr      $ra           # Retorna a quem chamou

.rdata
$LC:
.ascii  "The factorial of 10 is %d\n\000"

```

A rotina de fatorial é semelhante em estrutura a `main`. Primeiro, ela cria um frame de pilha e salva os registradores salvos pelo callee que serão usados por ela. Além de salvar `$ra` e `$fp`, `fact` também salva seu argumento (`$a0`), que ela usará para a chamada recursiva:

```

.text
fact:
    subu  $sp,$sp,32    # Frame de pilha tem 32 bytes
    sw    $ra,20($sp)  # Salva endereço de retorno
    sw    $fp,16($sp)  # Salva frame pointer
    addiu $fp,$sp,28    # Prepara frame pointer
    sw    $a0,0($fp)   # Salva argumento (n)

```

O núcleo da rotina `fact` realiza o cálculo do programa em C. Ele testa se o argumento é maior do que 0. Se não for, a rotina retorna o valor 1. Se o argumento for maior do que 0, a rotina é chamada recursivamente para calcular

`fact(n-1)` e multiplica esse valor por n :

```
lw      $v0,0($fp)    # Carrega n
bgtz   $v0,$L2        # Desvia se n > 0
li      $v0,1          # Retorna 1
jr      $L1            # Salta para o código de retorno

$L2:
lw      $v1,0($fp)    # Carrega n
subu   $v0,$v1,1       # Calcula n - 1
move   $a0,$v0          # Move valor para $a0
jal    fact            # Chama função de fatorial

lw      $v1,0($fp)    # Carrega n
mul   $v0,$v0,$v1       # Calcula  $fact(n-1) * n$ 
```

Finalmente, a rotina de fatorial restaura os registradores salvos pelo callee e retorna o valor no registrador `$v0`:

```
$L1:                      # Resultado está em $v0
lw    $ra, 20($sp)  # Restaura $ra
lw    $fp, 16($sp)  # Restaura $fp
addiu $sp, $sp, 32  # Remove conteúdo da pilha
jr    $ra             # Retorna a quem chamou
```

Pilha em procedimentos recursivos

Exemplo

A Figura A.6.3 mostra a pilha na chamada `fact(7)`. `main` executa primeiro, de modo que seu frame está mais abaixo na pilha. `main` chama `fact(10)`, cujo frame de pilha vem em seguida na pilha. Cada invocação chama `fact` recursivamente para calcular o próximo fatorial mais inferior. Os frames de pilha fazem um paralelo com a ordem LIFO dessas chamadas. Qual é a aparência da pilha quando a chamada para o `fact(10)` retorna?

Pilha

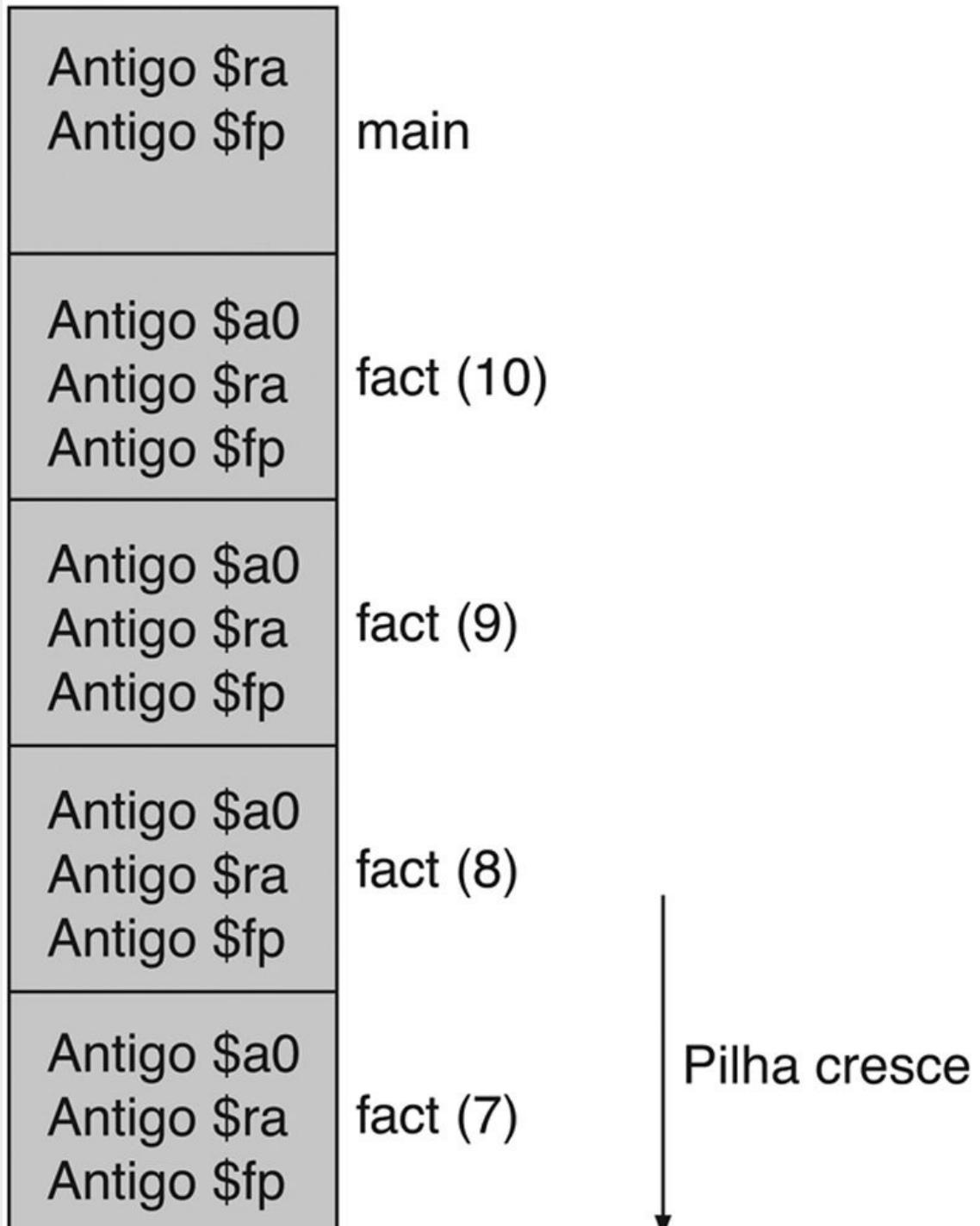
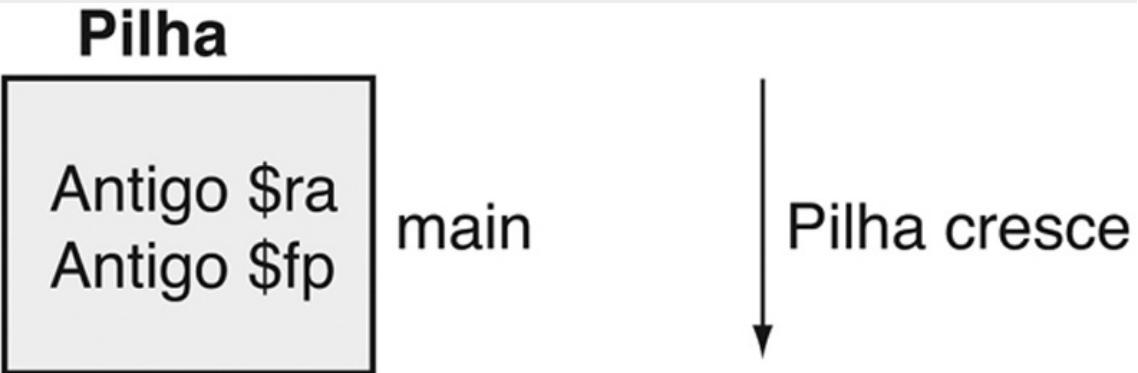


FIGURA A.6.3 Frames da pilha durante a chamada de `fact(7)`.

Resposta



Detalhamento

A diferença entre o compilador MIPS e o compilador gcc é que o compilador MIPS normalmente não usa um frame pointer, de modo que esse registrador está disponível como outro registrador salvo pelo callee, \$s8. Essa mudança salva algumas das instruções na chamada de procedimento e sequência de retorno. Contudo, isso complica a geração do código, porque um procedimento precisa acessar seu frame de pilha com \$sp, cujo valor pode mudar durante a execução de um procedimento se os valores forem colocados na pilha.

Outro exemplo de chamada de procedimento

Como outro exemplo, considere a seguinte rotina que calcula a função tak, que é um benchmark bastante utilizado, criado por Ikuo Takeuchi. Essa função não calcula nada de útil, mas é um programa altamente recursivo que ilustra a convenção de chamada do MIPS.

```
int tak (int x, int y, int z)
{
    if (y < x)
        return 1+ tak (tak (x - 1, y, z),
                      tak (y - 1, z, x),
                      tak (z - 1, x, y));
    else
        return z;
}

int main ()
{
    tak(18, 12, 6);
}
```

O código assembly para esse programa está logo em seguida. A função tak primeiro salva seu endereço de retorno no frame de pilha e seus argumentos nos registradores salvos pelo callee, pois a rotina pode fazer chamadas que precisam usar os registradores \$a0–\$a2 e \$ra. A função utiliza registradores salvos pelo callee, pois eles mantêm valores que persistem por toda a vida da função, o que inclui várias chamadas que potencialmente poderiam modificar registradores.

```

.text
.globl tak

tak:
    subu    $sp, $sp, 40
    sw      $ra, 32($sp)

    sw      $s0, 16($sp)    # x
    move   $s0, $a0
    sw      $s1, 20($sp)    # y
    move   $s1, $a1
    sw      $s2, 24($sp)    # z
    move   $s2, $a2
    sw      $s3, 28($sp)    # temporário

```

A rotina, então, inicia a execução testando se $y < x$. Se não for, ela desvia para o rótulo L1, que aparece a seguir.

```
bge    $s1, $s0, L1    # if (y < x)
```

Se $y < x$, então ela executa o corpo da rotina, que contém quatro chamadas recursivas. A primeira chamada usa quase os mesmos argumentos do seu pai:

```

addiu  $a0, $s0, -1
move   $a1, $s1
move   $a2, $s2
jal    tak           # tak (x - 1, y, z)
move   $s3, $v0

```

Observe que o resultado da primeira chamada recursiva é salvo no registrador

$\$s3$, de modo que possa ser usado mais tarde.

A função agora prepara argumentos para a segunda chamada recursiva.

```
addiu    $a0, $s1, -1
move     $a1, $s2
move     $a2, $s0
jal      tak          # tak (y - 1, z, x)
```

Nas instruções a seguir, o resultado dessa chamada recursiva é salvo no registrador $\$s0$. No entanto, primeiro, precisamos ler, pela última vez, o valor salvo do primeiro argumento a partir desse registrador.

```
addiu    $a0, $s2, -1
move     $a1, $s0
move     $a2, $s1
move     $s0, $v0
jal      tak          # tak (z - 1, x, y)
```

Depois de três chamadas recursivas mais internas, estamos prontos para a chamada recursiva final. Depois da chamada, o resultado da função está em $\$v0$, e o controle desvia para o epílogo da função.

```
move     $a0, $s3
move     $a1, $s0
move     $a2, $v0
jal      tak          # tak (tak(...), tak(...), tak(...))
addiu   $v0, $v0, 1
j       L2
```

Esse código no rótulo $L1$ é a consequência da instrução *if-then-else*. Ele apenas move o valor do argumento z para o registrador de retorno e cai no epílogo da função.

L1:

move \$v0, \$s2

O código a seguir é o epílogo da função, que restaura os registradores salvos e retorna o resultado da função a quem chamou.

L2:

lw	\$ra, 32(\$sp)
lw	\$s0, 16(\$sp)
lw	\$s1, 20(\$sp)
lw	\$s2, 24(\$sp)
lw	\$s3, 28(\$sp)
addiu	\$sp, \$sp, 40
jr	\$ra

A rotina main chama a função tak com seus argumentos iniciais, depois pega o resultado calculado (7) e o apresenta usando a chamada ao sistema do SPIM para exibir inteiros:

```
.globl main
main:
    subu    $sp, $sp, 24
    sw     $ra, 16($sp)

    li      $a0, 18
    li      $a1, 12

    li      $a2, 6
    jal    tak          # tak(18, 12, 6)

    move   $a0, $v0
    li      $v0, 1          # syscall print_int
    syscall

    lw      $ra, 16($sp)
    addiu $sp, $sp, 24
    jr      $ra
```

A.7. Exceções e interrupções

A Seção 4.9 do [Capítulo 4](#) descreve o mecanismo de exceção do MIPS, que responde a exceções causadas por erros durante a execução de uma instrução e a interrupções externas causadas por dispositivos de E/S. Esta seção descreve o **tratamento de interrupção** e exceção com mais detalhes.¹ Nos processadores MIPS, uma parte da CPU, chamada *coprocessador 0*, registra as informações de que o software precisa para lidar com exceções e interrupções. O simulador SPIM do MIPS não implementa todos os registradores do coprocessador 0, pois muitos não são úteis em um simulador ou fazem parte do sistema de memória, que o SPIM não implementa. Contudo, o SPIM oferece os seguintes registradores do coprocessador 0:

tratamento de interrupção

Um trecho de código executado como resultado de uma exceção ou interrupção.

Nome do registrador	Número do registrador	Uso
BadVAddr	8	endereço de memória em que ocorreu uma referência de memória problemática
Count	9	temporizador
Compare	11	valor comparado com o temporizador que causa interrupção quando combinam
Status	12	máscara de interrupções e bits de habilitação
Cause	13	tipo de exceção e bits de interrupções pendentes
EPC	14	endereço da instrução que causou a exceção
Config	16	configuração da máquina

Esses sete registradores fazem parte do conjunto de registradores do coprocessador 0. Eles são acessados pelas instruções `mfc0` e `mtc0`. Após uma exceção, o registrador EPC contém o endereço da instrução executada quando a exceção ocorreu. Se a exceção foi causada por uma interrupção externa, então a instrução não terá iniciado a execução. Todas as outras exceções são causadas pela execução da instrução no EPC, exceto quando a instrução problemática está no delay slot de um desvio ou salto. Nesse caso, o EPC aponta para a instrução

de desvio ou salto e o bit BD é habilitado no registrador Cause. Quando esse bit está habilitado, o handler de exceção precisa procurar a instrução problemática em EPC + 4. No entanto, de qualquer forma, um handler de exceção retoma o programa corretamente, retornando à instrução no EPC.

Se a instrução que causou a exceção fez um acesso à memória, o registrador BadVAddr contém o endereço do local de memória referenciado.

O registrador Count é um timer que incrementa em uma taxa fixa (como padrão, a cada 10 milissegundos) enquanto o SPIM está executando. Quando o valor no registrador Count for igual ao valor no registrador Compare, ocorre uma interrupção de hardware com nível de prioridade 5.

A [Figura A.7.1](#) mostra o subconjunto dos campos do registrador Status implementados pelo simulador SPIM do MIPS. O campo interrupt mask contém um bit para cada um dos seis níveis de interrupção de hardware e dois de software. Um bit de máscara 1 permite que as interrupções nesse nível parem o processador. Um bit de máscara 0 desativa as interrupções nesse nível. Quando uma interrupção chega, ela habilita seu bit de interrupção pendente no registrador Cause, mesmo que o bit de máscara esteja desabilitado. Quando uma interrupção está pendente, ela interromperá o processador quando seu bit de máscara for habilitado mais tarde.

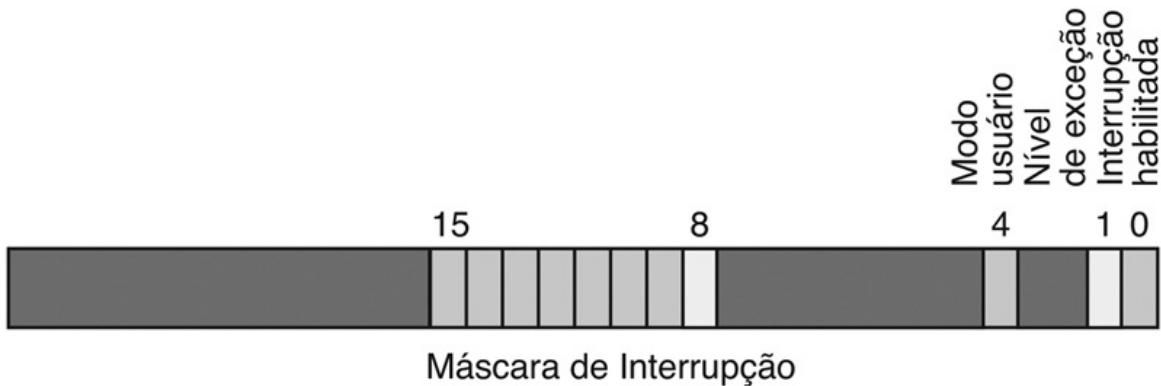


FIGURA A.7.1 O registrador Status.

O bit de modo usuário é 0 se o processador estiver funcionando no modo kernel e 1 se estiver funcionando no modo usuário. No SPIM, esse bit é fixado em 1, pois o processador SPIM não implementa o modo kernel. O bit de nível de exceção normalmente é 0, mas é colocado em 1 depois que ocorre uma exceção. Quando esse bit é 1, as interrupções são desativadas e o EPC não é atualizado se outra exceção ocorrer. Esse bit impede que um handler de exceção seja

incomodado por uma interrupção ou exceção, mas deve ser reiniciado quando o handler termina. Se o bit `interrupt enable` for 1, as interrupções são permitidas. Se for 0, elas serão inibidas.

A [Figura A.7.2](#) mostra o subconjunto dos campos do registrador `Cause` que o SPIM implementa. O bit de branch delay é 1 se a última exceção ocorreu em uma instrução executada no slot de retardo de um desvio. Os bits de interrupções pendentes tornam-se 1 quando uma interrupção é gerada em determinado nível de hardware ou software. O registrador de código de exceção descreve a causa de uma exceção por meio dos seguintes códigos:

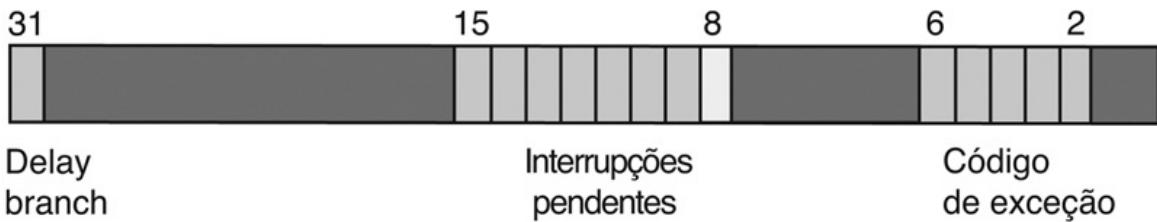


FIGURA A.7.2 O registrador `Cause`.

Número	Nome	Causa da exceção
0	Int	interrupção (hardware)
4	AdEL	exceção de erro de endereço (load ou busca de instrução)
5	AdES	exceção de erro de endereço (store)
6	IBE	erro de barramento na busca da instrução
7	DBE	erro de barramento no load ou store de dados
8	Sys	exceção de chamada do sistema (syscall)
9	Bp	exceção de ponto de interrupção (breakpoint)
10	RI	exceção de instrução reservada
11	CpU	coprocessador não implementado
12	Ov	exceção de overflow aritmético
13	Tr	interceptação (trap)
15	FPE	ponto flutuante

Exceções e interrupções fazem com que um processador MIPS desvie para uma parte do código, no endereço 80000180_{hexa} (no espaço de endereçamento do kernel, não do usuário), chamada *handler de exceção*. Esse código examina a causa da exceção e desvia para um ponto apropriado no sistema operacional. O sistema operacional responde a uma exceção terminando o processo que causou

a exceção ou realizando alguma ação. Um processo que causa um erro, como a execução de uma instrução não implementada, é terminado pelo sistema operacional. Por outro lado, outras exceções, como faltas de página, são solicitações de um processo para o sistema operacional realizar um serviço, como trazer uma página do disco. O sistema operacional processa essas solicitações e retoma o processo. O último tipo de exceções são interrupções de dispositivos externos. Elas em geral fazem com que o sistema operacional mova dados de/para um dispositivo de E/S e retome o processo interrompido.

O código no exemplo a seguir é um handler de exceção simples, que invoca uma rotina para exibir uma mensagem a cada exceção (mas não interrupções). Esse código é semelhante ao handler de exceção (`exceptions.s`) usado pelo simulador SPIM.

Handler de exceções

Exemplo

O handler de exceção primeiro salva o registrador `$at`, que é usado em pseudoinstruções no código do handler, depois salva `$a0` e `$a1`, que mais tarde utiliza para passar argumentos. O handler de exceção não pode armazenar os valores antigos a partir desses registradores na pilha, como faria uma rotina comum, pois a causa da exceção poderia ter sido uma referência de memória que usou um valor incorreto (como 0) no stack pointer. Em vez disso, o handler de exceção armazena esses registradores em um registrador de handler de exceção (`$k1`, pois não pode acessar a memória sem usar `$at`) e dois locais da memória (`save0` e `save1`). Se a própria rotina de exceção pudesse ser interrompida, dois locais não seriam suficientes, pois a segunda exceção gravaria sobre valores salvos durante a primeira exceção. Entretanto, esse handler de exceção simples termina a execução antes de permitir interrupções, de modo que o problema não surge.

```
.ktext 0x80000180
mov $k1, $at    # Salva o registrador $at
sw  $a0, save0  # Handler não é reentrante e não pode
sw  $a1, save1  # usar a pilha para salvar $a0, $a1
                  # Não precisa salvar $k0/$k1
```

O handler de exceção, então, move os registradores Cause e EPC para os registradores da CPU. Os registradores Cause e EPC não fazem parte do conjunto de registradores da CPU. Em vez disso, eles são registradores no coprocessador 0, que é a parte da CPU que trata das exceções. A instrução `mfc0 $k0, $13` move o registrador 13 do coprocessador 0 (o registrador Cause) para o registrador da CPU `$k0`. Observe que o handler de exceção não precisa salvar os registradores `$k0` e `$k1`, pois os programas do usuário não deveriam usar esses registradores. O handler de exceção usa o valor do registrador Cause para testar se a exceção foi causada por uma interrupção (ver a tabela anterior). Se tiver sido, a exceção é ignorada. Se a exceção não foi uma interrupção, o handler chama `print_excp` para apresentar uma mensagem.

```
mfc0    $k0, $13          # Move Cause para $k0
srl     $a0, $k0, 2         # Extrai o campo ExcCode
andi    $a0, $a0, 0xf
bgtz   $a0, done          # Desvia se ExcCode for Int (0)
mov     $a0, $k0          # Move Cause para $a0
mfco   $a1, $14          # Move EPC para $a1
jal    print_excp        # Mostra mensagem de erro de exceção
```

Antes de retornar, o handler de exceção apaga o registrador Cause, reinicia o registrador Status para ativar interrupções e limpar o bit EXL, de modo a permitir que exceções subsequentes mudem o registrador EPC, e restaura os registradores `$a0`, `$a1` e `$at`. Depois, ele executa a instrução `eret` (retorno de exceção), que retorna à instrução apontada pelo EPC. Esse handler de exceção retorna à instrução após aquela que causou a exceção, a fim de não reexecutar a instrução que falhou e causar a mesma exceção novamente.

```

done:    mfc0    $k0, $14      # Muda EPC
        addiu   $k0, $k0, 4       # Não reexecuta
                                # instrução que falhou
        mtc0    $k0, $14      # EPC

        mtc0    $0, $13      # Apaga registrador Cause

        mfc0    $k0, $12      # Repara registrador Status
        andi   $k0, 0xffffd    # Apaga bit EXL
        ori    $k0, 0x1       # Habilita interrupções
        mtc0    $k0, $12

        lw     $a0, save0      # Restaura registradores
        lw     $a1, save1
        mov    $at, $k1

        eret                  # Retorna ao EPC

        .kdata
save0:   .word 0
save1:   .word 0

```

Detalhamento

Em processadores MIPS reais, o retorno de um handler de exceção é mais complexo. O handler de exceção não pode sempre desviar para a instrução após o EPC. Por exemplo, se a instrução que causou a exceção estivesse em um delay slot de uma instrução de desvio (veja Capítulo 4), a próxima instrução a executar pode não ser a instrução seguinte na memória.

A.8. Entrada e saída

O SPIM simula um dispositivo de E/S: um console mapeado em memória em que um programa pode ler e escrever caracteres. Quando um programa está executando, o SPIM conecta seu próprio terminal (ou uma janela de console separada na versão `xspim` do X-Windows ou na versão `PCSpim` do Windows) ao processador. Um programa MIPS executando no SPIM pode ler os caracteres que você digita. Além disso, se o programa MIPS escreve caracteres no terminal, eles aparecem no terminal do SPIM ou na janela de console. Uma exceção a essa regra é Control-C: esse caractere não passa pelo programa, mas, em vez disso, faz com que o SPIM pare e retorne ao modo de comando. Quando o programa para de executar (por exemplo, porque você pressionou Control-C ou porque o programa atingiu um ponto de interrupção), o terminal é novamente conectado ao SPIM para que você possa digitar comandos do SPIM.

Para usar a E/S mapeada em memória (ver a seguir), o `spim` ou o `xspim` precisam ser iniciados com o flag `-mapped_io`. `PCSpim` pode ativar a E/S mapeada em memória por meio de um flag de linha de comando ou pela caixa de diálogo “Settings” (Configurações).

O dispositivo de terminal consiste em duas unidades independentes: um *receptor* e um *transmissor*. O receptor lê caracteres digitados no teclado. O transmissor exibe caracteres no vídeo. As duas unidades são completamente independentes. Isso significa, por exemplo, que os caracteres digitados no teclado não são reproduzidos automaticamente no monitor. Em vez disso, um programa reproduz um caractere lendo-o do receptor e escrevendo-o no transmissor.

Um programa controla o terminal com quatro registradores de dispositivo mapeados em memória, como mostra a [Figura A.8.1](#). “Mapeado em memória” significa que cada registrador aparece como uma posição de memória especial. O *registrador Receiver Control* está na posição $ffff0000_{\text{hexa}}$. Somente dois de seus bits são realmente usados. O bit 0 é chamado “pronto”: se for 1, isso significa que um caractere chegou do teclado, mas ainda não foi lido do registrador Receiver Data. O bit de pronto é apenas de leitura: tentativas de sobrescrevê-lo são ignoradas. O bit de pronto muda de 0 a 1 quando um caractere é digitado no teclado e ele muda de 1 para 0 quando o caractere é lido do registrador Receiver Data.

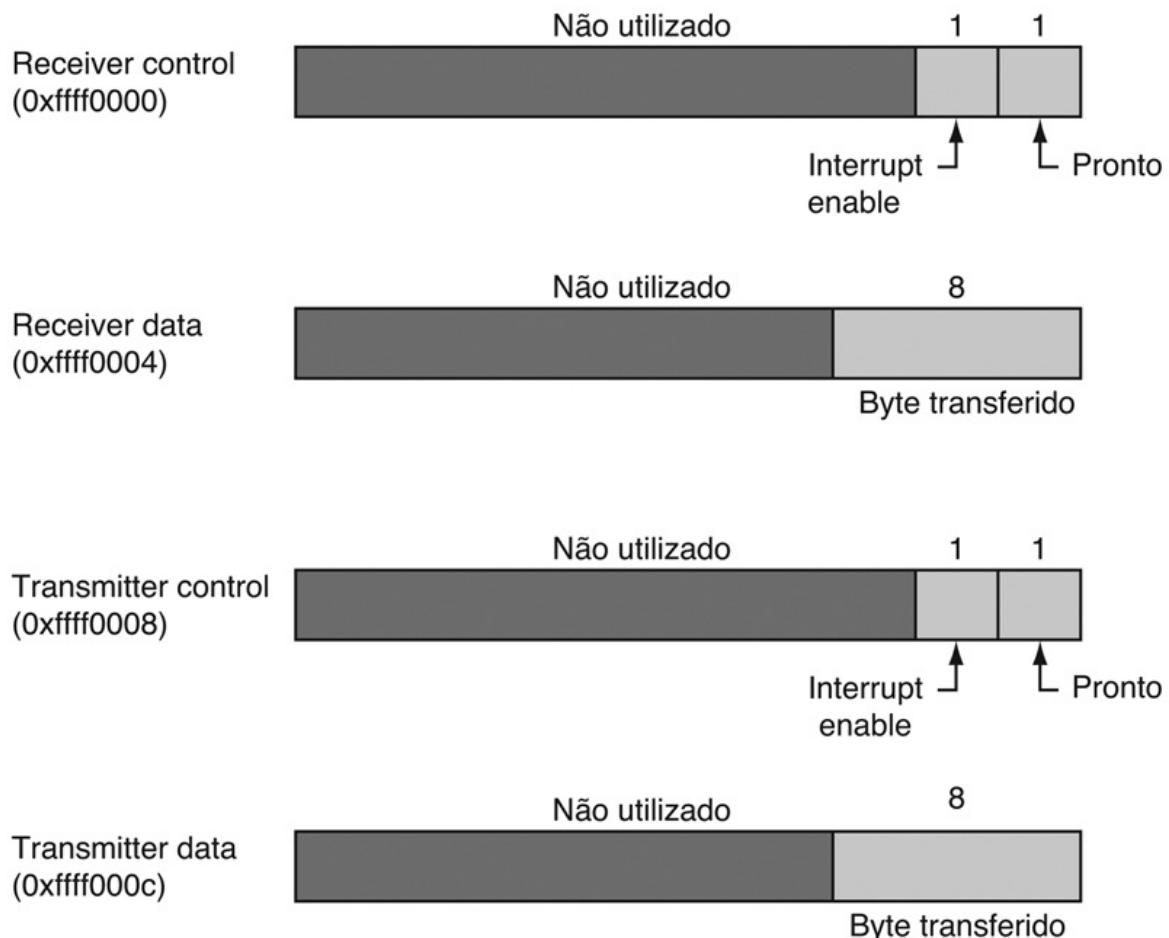


FIGURA A.8.1 O terminal é controlado por quatro registradores de dispositivo, cada um deles parecendo com uma posição de memória no endereço indicado.

Somente alguns bits desses registradores são realmente utilizados. Os outros sempre são lidos como 0s e as escritas são ignoradas.

O bit 1 do registrador Receiver Control é o “interrupt enable” do teclado. Esse bit pode ser lido e escrito por um programa. O interrupt enable inicialmente é 0. Se ele for colocado em 1 por um programa, o terminal solicita uma interrupção no nível de hardware 1 sempre que um caractere é digitado e o bit de pronto se torna 1. Todavia, para que a interrupção afete o processador, as interrupções também precisam estar ativadas no registrador Status ([Seção A.7](#)). Todos os outros bits do registrador Receiver Control não são utilizados.

O segundo registrador de dispositivo do terminal é o *registraror Receiver Data* (no endereço $ffff0004_{\text{hexa}}$). Os oito bits menos significativos desse registrador contêm o último caractere digitado no teclado. Todos os outros bits

contém 0s. Esse registrador é apenas de leitura e muda apenas quando um novo caractere é digitado no teclado. A leitura do registrador Receiver Data reinicia o bit de pronto no registrador Receiver Control para 0. O valor nesse registrador é indefinido se o registrador Receiver Control for 0.

O terceiro registrador de dispositivo do terminal é o *registrator Transmitter Control* (no endereço ffff0008_{hexa}). Somente os dois bits menos significativos desse registrador são usados. Eles se comportam de modo semelhante aos bits correspondentes no registrador Receiver Control. O bit 0 é chamado de “pronto” e é apenas de leitura. Se esse bit for 1, o transmissor estará pronto para aceitar um novo caractere para saída. Se for 0, o transmissor ainda está ocupado escrevendo o caractere anterior. O bit 1 é “interrupt enable” e pode ser lido e escrito. Se esse bit for definido em 1, então o terminal solicita uma interrupção no nível de hardware 0 sempre que o transmissor estiver pronto para um novo caractere e o bit de pronto se torna 1.

O registrador de dispositivo final é o *registrator Transmitter Data* (no endereço ffff000c_{hexa}). Quando um valor é escrito nesse local, seus oito bits menos significativos (ou seja, um caractere ASCII como na [Figura 2.15](#), no [Capítulo 2](#)) são enviados para o console. Quando o registrador Transmitter Data é escrito, o bit de pronto no registrador Transmitter Control é retornado para 0. Esse bit permanece sendo 0 até passar tempo suficiente para transmitir o caractere para o terminal; depois, o bit de pronto se torna 1 novamente. O registrador Transmitter Data só deverá ser escrito quando o bit de pronto do registrador Transmitter Control for 1. Se o transmissor não estiver pronto, as escritas no registrador Transmitter Data são ignoradas (as escritas parecem ter sucesso, mas o caractere não é enviado).

Computadores reais exigem tempo para enviar caracteres a um console ou terminal. Esses atrasos de tempo são simulados pelo SPIM. Por exemplo, depois que o transmissor começa a escrever um caractere, o bit de pronto do transmissor torna-se 0 por um tempo. O SPIM mede o tempo em instruções executadas, e não em tempo de clock real. Isso significa que o transmissor não fica pronto novamente até que o processador execute um número fixo de instruções. Se você interromper a máquina e examinar o bit de pronto, ele não mudará. Contudo, se você deixar a máquina executar, o bit por fim mudará de volta para 1.

A.9. SPIM

SPIM é um simulador de software que executa programas em assembly escritos para processadores que implementam a arquitetura MIPS-32, especificamente o Release 1 dessa arquitetura com um mapeamento de memória fixo, sem caches e apenas os coprocessadores 0 e 1.² O nome do SPIM é simplesmente MIPS ao contrário. O SPIM pode ler e executar imediatamente os arquivos em assembly. O SPIM é um sistema autocontido para executar programas do MIPS. Ele contém um depurador e oferece alguns serviços de forma semelhante ao sistema operacional. SPIM é muito mais lento do que um computador real (100 ou mais vezes). Entretanto, seu baixo custo e grande disponibilidade não têm comparação com o hardware real!

Uma pergunta óbvia é: por que usar um simulador quando a maioria das pessoas possui PCs que contêm processadores executando muito mais rápido do que o SPIM? Um motivo é que o processador nos PCs são 80x86s da Intel, cuja arquitetura é muito menos regular e muito mais complexa de entender e programar do que os processadores MIPS. A arquitetura do MIPS pode ser a síntese de uma máquina RISC simples e limpa.

Além disso, os simuladores podem oferecer um ambiente melhor para a programação em assembly do que uma máquina real, pois podem detectar mais erros e oferecer uma interface melhor do que um computador real.

Finalmente, os simuladores são uma ferramenta útil no estudo de computadores e dos programas executados. Como eles são implementados em software, não em silício, os simuladores podem ser examinados e facilmente modificados para acrescentar novas instruções, criar novos sistemas, como os multiprocessadores ou apenas coletar dados.

Simulação de uma máquina virtual

Uma arquitetura MIPS básica é difícil de programar diretamente, por causa dos delayed branches, delayed loads e modos de endereçamento restritos. Essa dificuldade é tolerável, pois esses computadores foram projetados para serem programados em linguagens de alto nível e apresentam uma interface criada para compiladores, em vez de programadores assembly. Boa parte da complexidade da programação é resultante de instruções delayed. Um *delayed branch* exige dois ciclos para executar (veja as seções “Detalhamentos” nas páginas 250 e 282

do [Capítulo 4](#)). No segundo ciclo, a instrução imediatamente após o desvio é executada. Essa instrução pode realizar um trabalho útil que normalmente teria sido feito antes do desvio. Ela também pode ser um *nop* (nenhuma operação), que não faz nada. De modo semelhante, os *delayed loads* exigem dois ciclos para trazer um valor da memória, de modo que a instrução imediatamente após um *load* não pode usar o valor (veja Seção 4.2 do [Capítulo 4](#)).

O MIPS sabiamente escolheu ocultar essa complexidade fazendo com que seu assembler implemente uma **máquina virtual**. Esse computador virtual parece ter branches e loads não delayed e um conjunto de instruções mais rico do que o hardware real. O assembler *reorganiza* instruções para preencher os delay slots. O computador virtual também oferece *pseudoinstruções*, que aparecem como instruções reais nos programas em assembly. O hardware, porém, não sabe nada a respeito de pseudoinstruções, de modo que o assembler as traduz para sequências equivalentes de instruções de máquina reais. Por exemplo, o hardware do MIPS só oferece instruções para desvio quando um registrador é igual ou diferente de 0. Outros desvios condicionais, como aquele que desvia quando um registrador é maior do que outro, são sintetizados comparando-se os dois registradores e desviando quando o resultado da comparação é verdadeiro (diferente de zero).

máquina virtual

Um computador virtual que parece ter desvios e loads não delayed e um conjunto de instruções mais rico do que o hardware real.

Como padrão, o SPIM simula a máquina virtual mais rica, pois essa é a máquina que a maioria dos programadores achará útil. Todavia, o SPIM também pode simular os desvios e loads delayed no hardware real. A seguir, descrevemos a máquina virtual e só mencionamos rapidamente os recursos que não pertencem ao hardware real. Ao fazer isso, seguimos a convenção dos programadores (e compiladores) assembly do MIPS, que normalmente utilizam a máquina estendida como se estivesse implementada em silício.

Introdução ao SPIM

O restante deste apêndice é uma introdução ao SPIM e à linguagem assembly MIPS R2000. Muitos detalhes nunca irão preocupá-lo; porém, o grande volume de informações, às vezes, poderá obscurecer o fato de que o SPIM é um

programa simples e fácil de usar. Esta seção começa com um tutorial rápido sobre o uso do SPIM, que deverá permitir que você carregue, depure e execute programas MIPS simples.

O SPIM vem em diferentes versões para diferentes tipos de sistemas. A única constante é a versão mais simples, chamada `spim`, que é um programa controlado por linha de comandos, executado em uma janela de console. Ele opera como a maioria dos programas desse tipo: você digita uma linha de texto, pressiona a tecla Enter (ou Return) e o `spim` executa seu comando. Apesar da falta de uma interface sofisticada, o `spim` pode fazer tudo que seus primos mais sofisticados fazem.

Existem dois primos sofisticados do `spim`. A versão que roda no ambiente X-Windows de um sistema UNIX ou Linux é chamada `xspim`. `xspim` é um programa mais fácil de aprender e usar do que o `spim`, pois seus comandos sempre são visíveis na tela e porque ele continuamente apresenta os registradores e a memória da máquina. A outra versão sofisticada se chama `PCspim` e roda no Microsoft Windows.

Recursos surpreendentes

Embora o SPIM fielmente simule o computador MIPS, o SPIM é um simulador, e certas coisas não são idênticas a um computador real. As diferenças mais óbvias são que a temporização da instrução e o sistema de memória não são idênticos. O SPIM não simula caches ou a latência da memória, nem reflete com precisão os atrasos na operação de ponto flutuante ou nas instruções de multiplicação e divisão. Além disso, as instruções de ponto flutuante não detectam muitas condições de erro, o que deveria causar exceções em uma máquina real.

Outra surpresa (que também ocorre na máquina real) é que uma pseudoinstrução se expande para várias instruções de máquina. Quando você examina a memória passo a passo, as instruções que encontra são diferentes daquelas do programa-fonte. A correspondência entre os dois conjuntos de instruções é muito simples, pois o SPIM não reorganiza as instruções para preencher delay slots.

Ordem de bytes

Os processadores podem numerar os bytes dentro de uma palavra de modo que o byte com o número mais baixo seja o mais à esquerda ou à direita. A convenção

usada por uma máquina é considerada sua *ordem de bytes*. Os processadores MIPS podem operar com a ordem de bytes *big-endian* ou *little-endian*. Por exemplo, em uma máquina big-endian, a diretiva `.byte 0, 1, 2, 3` resultaria em uma palavra de memória contendo

Byte #			
0	1	2	3

enquanto, em uma máquina little-endian, a palavra seria

Byte #			
3	2	1	0

O SPIM opera com duas ordens de bytes. A ordem de bytes do SPIM é a mesma ordem de bytes da máquina utilizada para executar o simulador. Por exemplo, em um Intel 80x86, o SPIM é little-endian, enquanto em um Macintosh ou Sun SPARC, o SPIM é big-endian.

Chamadas ao sistema

O SPIM oferece um pequeno conjunto de serviços semelhantes aos oferecidos pelo sistema operacional, por meio da instrução de chamada ao sistema (`syscall`). Para requisitar um serviço, um programa carrega o código da chamada ao sistema (veja [Figura A.9.1](#)) no registrador `$v0` e os argumentos nos registradores `$a0–$a3` (ou `$f12`, para valores de ponto flutuante). As chamadas ao sistema que retornam valores colocam seus resultados no registrador `$v0` (ou `$f0` para resultados de ponto flutuante). Por exemplo, o código a seguir exibe “the answer = 5”:

Serviço	Código de chamada ao sistema	Argumentos	Resultado
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$f0)
read_double	7		double (in \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	endereço (em \$v0)
exit	10		
print_char	11	\$a0 = char	
read_char	12		char (in \$v0)
open	13	\$a0 = nome de arquivo (string), \$a1 = flags, \$a2 = modo	descritor de arquivo (em \$a0)
read	14	\$a0 = descritor de arquivo, \$a1 = buffer, \$a2 = tamanho	número de caracteres lidos (em \$a0)
write	15	\$a0 = descritor de arquivo, \$a1 = buffer, \$a2 = tamanho	número de caracteres escritos (em \$a0)
close	16	\$a0 = descritor de arquivo	
exit2	17	\$a0 = resultado	

FIGURA A.9.1 Serviços do sistema.

```
.data
str:
    .asciiiz "the answer = "
.text
```

```

li      $v0, 4      # código de chamada ao sistema
          # para print_str
la      $a0, str    # endereço da string a exibir
syscall           # exibe a string

li      $v0, 1      # código de chamada ao sistema
          # para print_str
li      $a0, 5      # inteiro a exibir
syscall           # exibe

```

A chamada ao sistema `print_int` recebe um inteiro e o exibe no console. `print_float` exibe um único número de ponto flutuante; `print_double` exibe um número de precisão dupla; e `print_string` recebe um ponteiro para uma string terminada em nulo, que ele escreve no console.

As chamadas ao sistema `read_int`, `read_float` e `read_double` leem uma linha inteira da entrada, até o caractere de newline, inclusive. Os caracteres após o número são ignorados. `read_string` possui a mesma semântica da rotina de biblioteca `fgets` do UNIX. Ela lê até $n - 1$ caracteres para um buffer e termina a string com um byte nulo. Se menos de $n - 1$ caracteres estiverem na linha atual, `read_string` lê até o caractere de newline, inclusive, e novamente termina a string com nulo. *Aviso:* os programas que usam essas syscalls para ler do terminal não deverão usar E/S mapeada em memória (ver [Seção A.8](#)).

`sbrk` retorna um ponteiro para um bloco de memória contendo n bytes adicionais. `exit` interrompe o programa que o SPIM estiver executando. `exit2` termina o programa SPIM, e o argumento de `exit2` torna-se o valor retornado quando o próprio simulador SPIM termina.

`print_char` e `read_char` escrevem e leem um único caractere, respectivamente. `open`, `read`, `write` e `close` são as chamadas da biblioteca padrão do UNIX.

A.10. Assembly do MIPS R2000

Um processador MIPS consiste em uma unidade de processamento de inteiros (a CPU) e uma coleção de coprocessadores que realizam tarefas auxiliares ou operam sobre outros tipos de dados, como números de ponto flutuante (veja [Figura A.10.1](#)). O SPIM simula dois coprocessadores. O coprocessador 0 trata de exceções e interrupções. O coprocessador 1 é a unidade de ponto flutuante. O SPIM simula a maior parte dos aspectos dessa unidade.

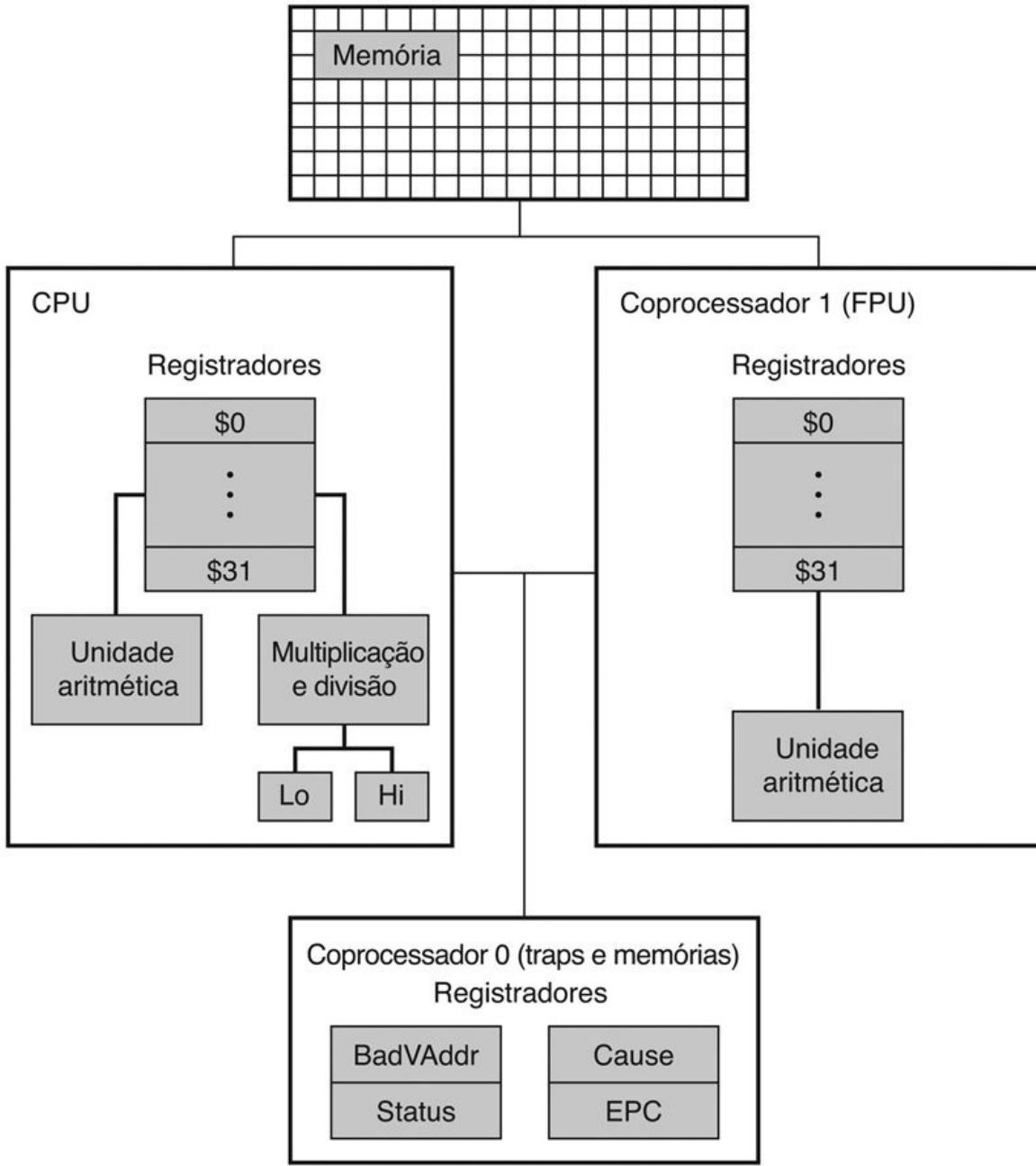


FIGURA A.10.1 CPU e FPU do MIPS R2000.

Modos de endereçamento

O MIPS é uma arquitetura load-store, o que significa que somente instruções load e store acessam a memória. As instruções de cálculo operam apenas sobre os valores nos registradores. A máquina pura oferece apenas um modo de endereçamento de memória: $c(rx)$, que usa a soma do c imediato e do

registraror rx como endereço. A máquina virtual oferece os seguintes modos de endereçamento para instruções load e store:

Formato	Cálculo de endereço
(registraror)	conteúdo do registraror
imm	imediato
imm (registraror)	imediato + conteúdo do registraror
rótulo	endereço do rótulo
rótulo ± imediato	endereço do rótulo + ou - imediato
rótulo ± imediato (registraror)	endereço do rótulo + ou - (imediato + conteúdo do registraror)

A maior parte das instruções load e store opera apenas sobre dados alinhados. Uma quantidade está *alinhada* se seu endereço de memória for um múltiplo do seu tamanho em bytes. Portanto, um objeto halfword precisa ser armazenado em endereços pares e um objeto palavra (word) precisa ser armazenado em endereços que são múltiplos de quatro. No entanto, o MIPS oferece algumas instruções para manipular dados não alinhados (`lw1`, `lwr`, `sw1` e `swr`).

Detalhamento

O assembler MIPS (e SPIM) sintetiza os modos de endereçamento mais complexos, produzindo uma ou mais instruções antes que o load ou o store calculem um endereço complexo. Por exemplo, suponha que o rótulo `table` referenciasse o local de memória `0x10000004` e um programa tivesse a instrução

```
ld $a0, table + 4($a1)
```

O assembler traduziria essa instrução para as instruções

```
lui $at, 4096  
addu $at, $at, $a1  
lw $a0, 8($at)
```

A primeira instrução carrega os bits mais significativos do endereço do rótulo no registrador \$at, que é o registrador que o assembler reserva para seu próprio uso. A segunda instrução acrescenta o conteúdo do registrador \$a1 ao endereço parcial do rótulo. Finalmente, a instrução load utiliza o modo de endereçamento de hardware para adicionar a soma dos bits menos significativos do endereço do rótulo e o offset da instrução original ao valor no registrador \$at.

Sintaxe do assembler

Os comentários nos arquivos do assembler começam com um sinal de sharp (#). Tudo desde esse sinal até o fim da linha é ignorado.

Os identificadores são uma sequência de caracteres alfanuméricos, símbolos de underscore (_) e pontos (.), que não começam com um número. Os opcodes de instrução são palavras reservadas que *não podem* ser usadas como identificadores. Rótulos são declarados por sua colocação no início de uma linha e seguidos por um sinal de dois-pontos, por exemplo:

```
.data  
item: .word 1  
      .text  
      .globl main      # Precisa ser global  
main: lw             $t0, item
```

Os números estão na base 10 por padrão. Se eles forem precedidos por 0x, serão interpretados como hexadecimais. Logo, 256 e 0x100 indicam o mesmo valor.

As strings são delimitadas com aspas (“”). Caracteres especiais nas strings

seguem a convenção da linguagem C:

- nova linha \n
- tabulação \t
- aspas \"

O SPIM admite um subconjunto das diretivas do assembler do MIPS:

.align n	Alinha o próximo dado em um limite de 2^n bytes. Por exemplo, .align 2 alinha o próximo valor em um limite da palavra. .align 0 desativa o alinhamento automático das diretivas .half, .word, .float e .double até a próxima diretiva .data ou .kdata.
.ascii str	Armazena a string str na memória, mas não a termina com nulo.
.asciiz str	Armazena a string str na memória e a termina com nulo.
.byte b1, ..., bn	Armazena os n valores em bytes sucessivos da memória.
.data <end>	Itens subsequentes são armazenados no segmento de dados. Se o argumento opcional end estiver presente, os itens subsequentes são armazenados a partir do endereço end.
.double d1, ..., dn	Armazena os n números de precisão dupla em ponto flutuante em locais de memória sucessivos.
.extern sym tamanho	Declara que o dado armazenado em sym possui tamanho bytes de extensão e é um rótulo global. Essa diretiva permite que o assembler armazene o dado em uma parte do segmento de dados que é acessada eficientemente por meio do registrador \$gp.
.float f1, ..., fn	Armazena os n números de precisão simples em ponto flutuante em locais sucessivos na memória.
.globl sym	Declara que o rótulo sym é global e pode ser referenciado a partir de outros arquivos.
.half h1, ..., hn	Armazena as n quantidades de 16 bits em halfwords sucessivas da memória.
.kdata<addr>	Itens de dados subsequentes são armazenados no segmento de dados do kernel. Se o argumento opcional addr estiver presente, itens subsequentes são armazenados a partir do endereço addr.
.ktext<addr>	Itens subsequentes são colocados no segmento de texto do kernel. No SPIM, esses itens só podem ser instruções ou palavras (ver a diretiva .word, mais adiante). Se o argumento opcional addr estiver presente, os itens subsequentes são armazenados a partir do endereço addr.
.set noat e .sat at	A primeira diretiva impede que o SPIM reclame sobre instruções subsequentes que utilizam o registrador \$at. A segunda diretiva reativa a advertência. Como as pseudoinstruções se expandem para o código que usa o registrador \$at, os programadores precisam ter muito cuidado ao deixar valores nesse registrador.
.space n	Aloca n bytes de espaço no segmento atual (que precisa ser o segmento de dados no SPIM).
.text <addr>	Itens subsequentes são colocados no segmento de texto do usuário. No SPIM, esses itens só podem ser instruções ou palavras (ver a diretiva .word a seguir). Se o argumento opcional addr estiver presente, os itens subsequentes são armazenados a partir do endereço addr.
.word w1, ..., wn	Armazena as n quantidades de 32 bits em palavras de memória sucessivas.

O SPIM não distingue as várias partes do segmento de dados (.data, .rdata e .sdata).

Codificando instruções do MIPS

A Figura A.10.2 explica como uma instrução MIPS é codificada em um número binário. Cada coluna contém codificações de instrução para um campo (um grupo de bits contíguo) a partir de uma instrução. Os números na margem esquerda são valores para um campo. Por exemplo, o opcode j possui um valor 2 no campo opcode. O texto no topo de uma coluna nomeia um campo e especifica quais bits ele ocupa em uma instrução. Por exemplo, o campo op está contido nos bits 26-31 de uma instrução. Esse campo codifica a maioria das instruções. No entanto, alguns grupos de instruções utilizam campos adicionais para distinguir instruções relacionadas. Por exemplo, as diferentes instruções de ponto flutuante são especificadas pelos bits 0-5. As setas a partir da primeira coluna mostram quais opcodes utilizam esses campos adicionais.

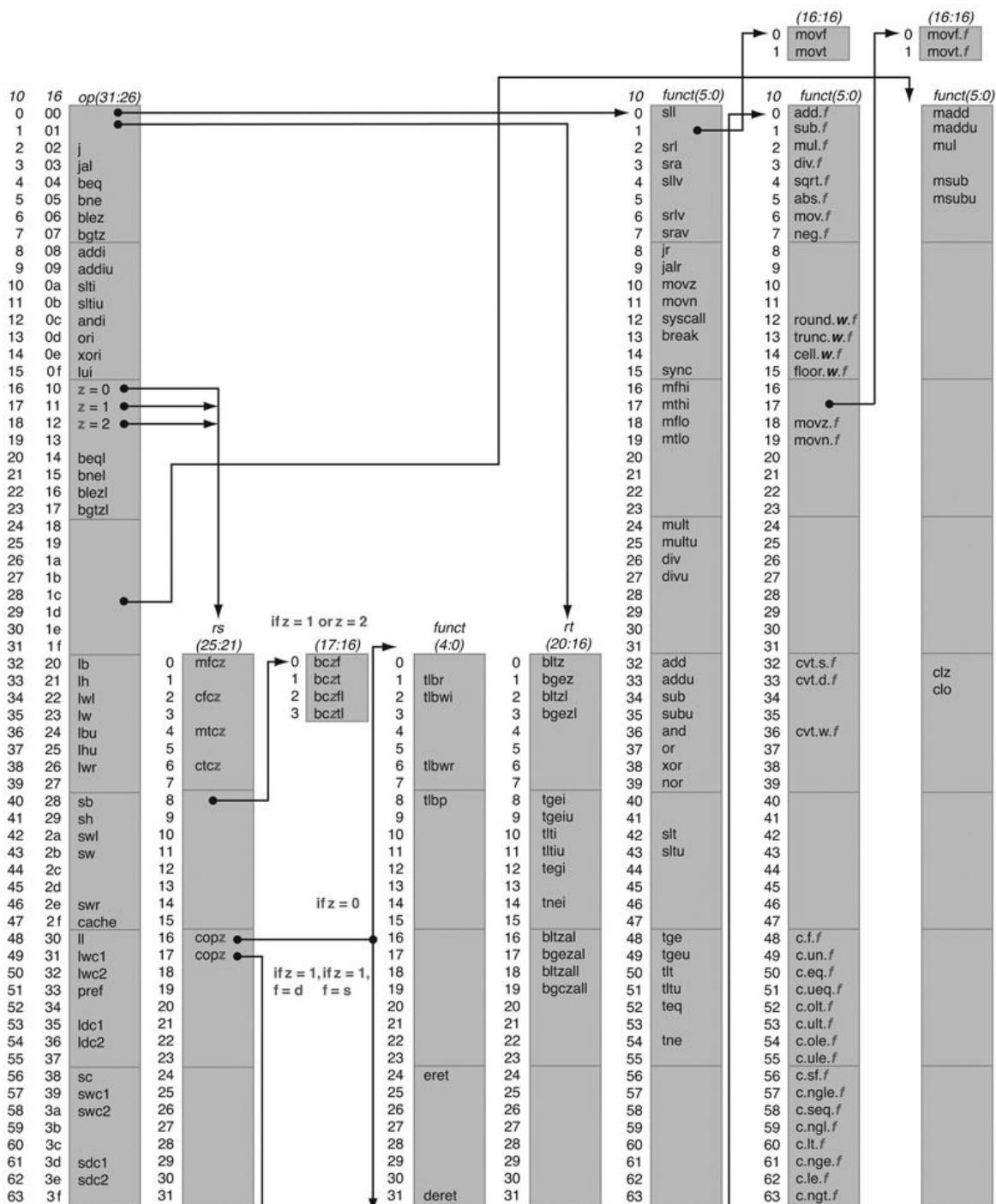


FIGURA A.10.2 Mapa de opcode do MIPS.

Os valores de cada campo aparecem à sua esquerda. A primeira coluna mostra os valores na base 10 e a segunda mostra a base 16 para o campo op (bits 31 a 26) na terceira coluna. Esse campo op especifica completamente a operação do MIPS, exceto para 6 valores de op: 0, 1, 16, 17, 18 e 19. Essas operações são determinadas pelos outros campos, identificados por ponteiros. O último campo (funct) utiliza “f” para indicar “s” se

$rs = 16$ e $op = 17$ ou “d” se $rs = 17$ e $op = 17$. O segundo campo (rs) usa “z” para indicar “0”, “1”, “2” ou “3” se $op = 16, 17, 18$ ou 19 , respectivamente. Se $rs = 16$, a operação é especificada em outro lugar: se $z = 0$, as operações são especificadas no quarto campo (bits 4 a 0); se $z = 1$, então as operações são no último campo com $f = s$. Se $rs = 17$ e $z = 1$, então as operações estão no último campo com $f = d$.

Formato de instrução

O restante deste apêndice descreve as instruções implementadas pelo hardware MIPS real e as pseudoinstruções fornecidas pelo assembler MIPS. Os dois tipos de instruções podem ser distinguidos facilmente. As instruções reais indicam os campos em sua representação binária. Por exemplo, em:

Adição (com overflow)

add rd, rs, rt	<table border="1" style="border-collapse: collapse; width: 100%;"> <tr> <td style="width: 60px; text-align: center;">0</td><td style="width: 50px; text-align: center;">rs</td><td style="width: 50px; text-align: center;">rt</td><td style="width: 50px; text-align: center;">rd</td><td style="width: 50px; text-align: center;">0</td><td style="width: 60px; text-align: center;">0x20</td></tr> </table>	0	rs	rt	rd	0	0x20
0	rs	rt	rd	0	0x20		
	6 5 5 5 5 6						

a instrução add consiste em seis campos. O tamanho de cada campo em bits é o pequeno número abaixo do campo. Essa instrução começa com 6 bits em 0. Os especificadores de registradores começam com um r , de modo que o próximo campo é um especificador de registrador de 5 bits chamado rs . Esse é o mesmo registrador que é o segundo argumento no assembly simbólico à esquerda dessa linha. Outro campo comum é imm_{16} , que é um número imediato de 16 bits.

As pseudoinstruções seguem aproximadamente as mesmas convenções, mas omitem a informação de codificação de instrução. Por exemplo:

Multiplicação (sem overflow)

`mul rdest, rsrc1, src2 pseudoinstrução`

Nas pseudoinstruções, $rdest$ e $rsrc1$ são registradores e $src2$ é um registrador ou um valor imediato. Em geral, o assembler e o SPIM traduzem uma forma mais geral de uma instrução (por exemplo, `add $v1, $a0, 0x55`) para uma forma especializada (por exemplo, `addi $v1, $a0, 0x55`).

Instruções aritméticas e lógicas

Valor absoluto

abs rdest, rsrc *pseudoinstrução*

Coloca o valor absoluto do registrador rsrc no registrador rdest.

Adição (com overflow)

add rd, rs, rt

0	rs	rt	rd	0	0x20
6	5	5	5	5	6

Adição (sem overflow)

addu rd, rs, rt

0	rs	rt	rd	0	0x21
6	5	5	5	5	6

Coloca a soma dos registradores rs e rt no registrador rd.

Adição imediato (com overflow)

addi rt, rs, imm

8	rs	rt	imm
6	5	5	16

Adição imediato (sem overflow)

addiu rt, rs, imm

9	rs	rt	imm
6	5	5	16

Coloca a soma do registrador rs e o imediato com sinal estendido no registrador rt.

AND

and rd, rs, rt	<table border="1"><tr><td>0</td><td>rs</td><td>rt</td><td>rd</td><td>0</td><td>0x24</td></tr><tr><td>6</td><td>5</td><td>5</td><td>5</td><td>5</td><td>6</td></tr></table>	0	rs	rt	rd	0	0x24	6	5	5	5	5	6
0	rs	rt	rd	0	0x24								
6	5	5	5	5	6								

Coloca o AND lógico dos registradores rs e rt no registrador rd.

AND imediato

andi rt, rs, imm	<table border="1"><tr><td>0xc</td><td>rs</td><td>rt</td><td>imm</td></tr><tr><td>6</td><td>5</td><td>5</td><td>16</td></tr></table>	0xc	rs	rt	imm	6	5	5	16
0xc	rs	rt	imm						
6	5	5	16						

Coloca o AND lógico do registrador rs e o imediato estendido com zeros no registrador rt.

Contar uns iniciais

clz rd, rs	<table border="1"><tr><td>0x1c</td><td>rs</td><td>0</td><td>rd</td><td>0</td><td>0x21</td></tr><tr><td>6</td><td>5</td><td>5</td><td>5</td><td>5</td><td>6</td></tr></table>	0x1c	rs	0	rd	0	0x21	6	5	5	5	5	6
0x1c	rs	0	rd	0	0x21								
6	5	5	5	5	6								

Contar zeros iniciais

clz rd, rs	<table border="1"><tr><td>0x1c</td><td>rs</td><td>0</td><td>rd</td><td>0</td><td>0x20</td></tr><tr><td>6</td><td>5</td><td>5</td><td>5</td><td>5</td><td>6</td></tr></table>	0x1c	rs	0	rd	0	0x20	6	5	5	5	5	6
0x1c	rs	0	rd	0	0x20								
6	5	5	5	5	6								

Conta o número de uns (zeros) iniciais da palavra no registrador rs e coloca o resultado no registrador rd. Se uma palavra contém apenas uns (zeros), o

resultado é 32.

Divisão (com overflow)

div rs, rt	<table border="1"><tr><td>0</td><td>rs</td><td>rt</td><td>0</td><td>0x1a</td></tr><tr><td>6</td><td>5</td><td>5</td><td>10</td><td>6</td></tr></table>	0	rs	rt	0	0x1a	6	5	5	10	6
0	rs	rt	0	0x1a							
6	5	5	10	6							

Divisão (sem overflow)

divu rs, rt	<table border="1"><tr><td>0</td><td>rs</td><td>rt</td><td>0</td><td>0x1b</td></tr><tr><td>6</td><td>5</td><td>5</td><td>10</td><td>6</td></tr></table>	0	rs	rt	0	0x1b	6	5	5	10	6
0	rs	rt	0	0x1b							
6	5	5	10	6							

Divide o registrador rs pelo registrador rt. Deixa o quociente no registrador lo e o resto no registrador hi. Observe que, se um operando for negativo, o restante não será especificado pela arquitetura MIPS e dependerá da convenção da máquina em que o SPIM é executado.

Divisão (com overflow)

div rdest, rsrc1, src2 *pseudoinstrução*

Divisão (sem overflow)

divu rdest, rsrc1, src2 *pseudoinstrução*

Coloca o quociente do registrador rsrc1 pelo src2 no registrador rdest.

Multiplicação

mult rs, rt	<table border="1"><tr><td>0</td><td>rs</td><td>rt</td><td>0</td><td>0x18</td></tr><tr><td>6</td><td>5</td><td>5</td><td>10</td><td>6</td></tr></table>	0	rs	rt	0	0x18	6	5	5	10	6
0	rs	rt	0	0x18							
6	5	5	10	6							

Multiplicação sem sinal

multu rs, rt	0	rs	rt	0	0x19
	6	5	5	10	6

Multiplica os registradores rs e rt. Deixa a palavra menos significativa do produto no registrador lo e a palavra mais significativa no registrador hi.

Multiplicação (sem overflow)

mul rd, rs, rt	0x1c	rs	rt	rd	0	2
	6	5	5	5	5	6

Coloca os 32 bits menos significativos do produto de rs e rt no registrador rd.

Multiplicação (com overflow)

mulo rdest, rsrc1, src2 *pseudo instrução*

Multiplicação sem sinal (com overflow)

mulou rdest, rsrc1, src2 *pseudo instrução*

Coloca os 32 bits menos significativos do produto do registrador rsrc1 e src2 no registrador rdest.

Multiplicação adição

madd rs, rt	0x1c	rs	rt	0	0
	6	5	5	10	6

Multiplicação adição sem sinal

maddu rs, rt	0x1c	rs	rt	0	1
	6	5	5	10	6

Multiplica os registradores rs e rt e soma o produto de 64 bits resultante ao valor de 64 bits nos registradores concatenados lo e hi.

Multiplicação subtração

msub rs, rt

0x1c	rs	rt	0	4
6	5	5	10	6

Multiplicação subtração sem sinal

msub rs, rt

0x1c	rs	rt	0	5
6	5	5	10	6

Multiplica os registradores rs e rt e subtrai o produto de 64 bits resultante do valor de 64 bits nos registradores concatenados lo e hi.

Negar valor (com overflow)

neg rdest, rsrc *pseudoinstrução*

Negar valor (sem overflow)

negu rdest, rsrc *pseudoinstrução*

Coloca o negativo do registrador rsrc no registrador rdest.

NOR

nor rd, rs, rt

0	rs	rt	rd	0	0x27
6	5	5	5	5	6

Coloca o NOR lógico dos registradores rs e rt para o registrador rd.

NOT

not

not rdest, rsrc *pseudoinstrução*

Coloca a negação lógica bit a bit do registrador rsrc no registrador rdest.

OR

or rd, rs, rt

0	rs	rt	rd	0	0x25
6	5	5	5	5	6

Coloca o OR lógico dos registradores rs e rt no registrador rd.

OR imediato

ori rt, rs, imm

Oxd	rs	rt	imm
6	5	5	16

Coloca o OR lógico do registrador rs e o imediato estendido com zero no registrador rt.

Resto

rem rdest, rsrc1, rsrc2 *pseudoinstrução*

Resto sem sinal

remu rdest, rsrc1, rsrc2 *pseudoinstrução*

Coloca o resto do registrador rsrc1 dividido pelo registrador rsrc2 no registrador rdest. Observe que se um operando for negativo, o resto não é especificado pela arquitetura MIPS e depende da convenção da máquina em que o SPIM é executado.

Shift lógico à esquerda

sll rd, rt, shamt

0	rs	rt	rd	shamt	0
6	5	5	5	5	6

Shift lógico à esquerda variável

sllv rd, rt, rs

0	rs	rt	rd	0	4
6	5	5	5	5	6

Shift aritmético à direita

sra rd, rt, shamt

0	rs	rt	rd	shamt	3
6	5	5	5	5	6

Shift aritmético à direita variável

sraw rd, rt, rs

0	rs	rt	rd	0	7
6	5	5	5	5	6

Shift lógico à direita

srl rd, rt, shamt

0	rs	rt	rd	shamt	2
6	5	5	5	5	6

Shift lógico à direita variável

srlv rd, rt, rs

0	rs	rt	rd	0	6
6	5	5	5	5	6

Desloca o registrador rt à esquerda (direita) pela distância indicada pelo shamt imediato ou pelo registrador rs e coloca o resultado no registrador rd. Observe que o argumento rs é ignorado para sll, sra e srl.

Rotate à esquerda

rol rdest, rsrc1, rsrc2 *pseudoinstrução*

Rotate à direita

ror rdest, rsrc1, rsrc2 *pseudoinstrução*

Gira o registrador rsrc1 à esquerda (direita) pela distância indicada por rsrc2 e coloca o resultado no registrador rdest.

Subtração (com overflow)

sub rd, rs, rt	0	rs	rt	rd	0	0x22
	6	5	5	5	5	6

Subtração (sem overflow)

subu rd, rs, rt	0	rs	rt	rd	0	0x23
	6	5	5	5	5	6

Coloca a diferença dos registradores rs e rt no registrador rd.

OR exclusivo

xor rd, rs, rt	0	rs	rt	rd	0	0x26
	6	5	5	5	5	6

Coloca o XOR lógico dos registradores rs e rt no registrador rd.

XOR imediato

xori rt, rs, imm	Oxe	rs	rt	Imm
	6	5	5	16

Coloca o XOR lógico do registrador rs e o imediato estendido com zeros no registrador rt.

Instruções para manipulação de constantes

Load superior imediato

lui rt, imm	Oxf	0	rt	imm
	6	5	5	16

Carrega a halfword menos significativa do imediato imm na halfword mais significativa do registrador rt. Os bits menos significativos do registrador são colocados em 0.

Load imediato

li rdest, imm *pseudoinstrução*

Move o imediato imm para o registrador rdest.

Instruções de comparação

Set se menor que

slt rd, rs, rt	0	rs	rt	rd	0	0x2a
	6	5	5	5	5	6

Set se menor que sem sinal

sltu rd, rs, rt	0	rs	rt	rd	0	0x2b
	6	5	5	5	5	6

Coloca o registrador rd em 1 se o registrador rs for menor que rt; caso contrário, coloca-o em 0.

Set se menor que imediato

slt i rt, rs, imm	0xa	rs	rt	imm
	6	5	5	16

Set se menor que imediato sem sinal

sltiu rt, rs, imm	0xb	rs	rt	imm
	6	5	5	16

Coloca o registrador rt em 1 se o registrador rs for menor que o imediato estendido com sinal, e em 0 em caso contrário.

Set se igual

seq rdest, rsrc1, rsrc2 *pseudoinstrução*

Coloca o registrador rdest em 1 se o registrador rsrc1 for igual a rsrc2, e em 0 caso contrário.

Set se maior ou igual

sge rdest, rsrc1, rsrc2 *pseudoinstrução*

Set se maior ou igual sem sinal

sgeu rdest, rsrc1, rsrc2 *pseudoinstrução*

Coloca o registrador rdest em 1 se o registrador rsrc1 for maior ou igual a rsrc2, e em 0 caso contrário.

Set se maior que

sgt rdest, rsrc1, rsrc2 *pseudoinstrução*

Set se maior que sem sinal

sgtu rdest, rsrc1, rsrc2 *pseudoinstrução*

Coloca o registrador rdest em 1 se o registrador rsrc1 for maior que rsrc2, e em 0 caso contrário.

Set se menor ou igual

sle rdest, rsrc1, rsrc2 *pseudoinstrução*

Set se menor ou igual sem sinal

sleu rdest, rsrc1, rsrc2 *pseudoinstrução*

Coloca o registrador rdest em 1 se o registrador rsrc1 for menor ou igual a rsrc2, e em 0 caso contrário.

Set se diferente

sne rdest, rsrc1, rsrc2 *pseudoinstrução*

Coloca o registrador rdest em 1 se o registrador rsrc1 não for igual a rsrc2, e em 0 caso contrário.

Instruções de desvio

As instruções de desvio utilizam um campo *offset* de instrução de 16 bits com sinal; logo, elas podem desviar $2^{15} - 1$ instruções (não bytes) para frente ou 2^{15} instruções para trás. A instrução *jump* contém um campo de endereço de 26 bits. Em processadores MIPS reais, as instruções de desvio são delayed branches, que não transferem o controle até que a instrução após o desvio (seu “delay slot”) tenha sido executada (veja [Capítulo 4](#)). Os delayed branches afetam o cálculo de offset, pois precisam ser calculados em relação ao endereço da instrução do delay slot ($PC + 4$), que é quando o desvio ocorre. O SPIM não simula esse delay slot, a menos que os flags *-bare* ou *-delayed_branch* sejam especificados.

No código assembly, os offsets normalmente não são especificados como números. Em vez disso, uma instrução desvia para um rótulo, e o assembler calcula a distância entre o desvio e a instrução destino.

No MIPS-32, todas as instruções de desvio condicional reais (não pseudo) têm uma variante “provável” (por exemplo, a variável provável de *beq* é *beql*), que não executa a instrução no delay slot do desvio se o desvio não for tomado. Não

use essas instruções; elas poderão ser removidas em versões subsequentes da arquitetura. O SPIM implementa essas instruções, mas elas não são descritas daqui por diante.

Branch

b label *pseudoinstrução*

Desvia incondicionalmente para a instrução no rótulo (label).

Branch coprocessador falso

bclf cc label

0x11	8	cc	0	Offset
6	5	3	2	16

Branch coprocessador verdadeiro

bclt cc label

0x11	8	cc	1	Offset
6	5	3	2	16

Desvia condicionalmente pelo número de instruções especificado pelo offset se o flag de condição de ponto flutuante numerado como cc for falso (verdadeiro). Se cc for omitido da instrução, o flag de código de condição 0 é assumido.

Branch se for igual

beq rs, rt, label

4	rs	rt	Offset
6	5	5	16

Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador rs for igual a rt.

Branch se for maior ou igual a zero

bgez rs, label

1	rs	1	Offset
6	5	5	16

Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador rs for maior ou igual a 0.

Branch se for maior ou igual a zero e link

bgezal rs, label

1	rs	0x11	Offset
6	5	5	16

Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador rs for maior ou igual a 0. Salva o endereço da próxima instrução no registrador 31.

Branch se for maior que zero

bgtz rs, label

7	rs	0	Offset
6	5	5	16

Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador rs for maior que 0.

Branch se for menor ou igual a zero

blez rs, label

6	rs	0	Offset
6	5	5	16

Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador rs for menor ou igual a 0.

Branch se for menor e link

bltzal rs, label

1	rs	0x10	Offset
6	5	5	16

Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador rs for menor que 0. Salva o endereço da próxima instrução no registrador 31.

Branch se for menor que zero

bltz rs, label

1	rs	0	Offset
6	5	5	16

Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador rs for menor que 0.

Branch se for diferente

bne rs, rt, label

5	rs	rt	Offset
6	5	5	16

Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador rs não for igual a rt.

Branch se for igual a zero

beqz rsrc, label *pseudoinstrução*

Desvia condicionalmente para a instrução no rótulo se rsrc for igual a 0.

Branch se for maior ou igual

bge rsrc1, rsrc2, label *pseudoinstrução*

Branch se for maior ou igual sem sinal

bgeu rsrc1, rsrc2, label *pseudoinstrução*

Desvia condicionalmente até a instrução no rótulo se o registrador rsrc1 for maior ou igual a rsrc2.

Branch se for maior

bgt rsrc1, src2, label *pseudoinstrução*

Branch se for maior sem sinal

bgtu rsrc1, src2, label *pseudoinstrução*

Desvia condicionalmente para a instrução no rótulo se o registrador rsrc1 for maior do que src2.

Branch se for menor ou igual

ble rsrc1, src2, label *pseudoinstrução*

Branch se for menor ou igual sem sinal

bleu rsrc1, src2, label *pseudoinstrução*

Desvia condicionalmente para a instrução no rótulo se o registrador rsrc1 for menor ou igual a src2.

Branch se for menor

blt rsrc1, rsrc2, label *pseudoinstrução*

Branch se for menor sem sinal

bltu rsrc1, rsrc2, label *pseudoinstrução*

Desvia condicionalmente para a instrução no rótulo se o registrador rsrc1 for menor do que src2.

Branch se não for igual a zero

bnez rsrc, label *pseudoinstrução*

Desvia condicionalmente para a instrução no rótulo se o registrador rsrc não for igual a 0.

Instruções de jump

Jump

j target

2	target
---	--------

Desvia incondicionalmente para a instrução no destino.

Jump-and-link

jal target

3	target
6	26

Desvia incondicionalmente para a instrução no destino. Salva o endereço da próxima instrução no registrador \$ra.

Jump-and-link registrador

jalr rs, rd

0	rs	0	rd	0	9
6	5	5	5	5	6

Desvia incondicionalmente para a instrução cujo endereço está no registrador rs. Salva o endereço da próxima instrução no registrador rd (cujo default é 31).

Jump registrador

jr rs

0	rs	0	8
6	5	15	6

Desvia incondicionalmente para a instrução cujo endereço está no registrador rs.

Instruções de trap

Trap se for igual

teq rs, rt	0	rs	rt	0	0x34
	6	5	5	10	6

Se o registrador rs for igual ao registrador rt, gera uma exceção de Trap.

Trap se for igual imediato

teqi rs, imm	1	rs	0xc	imm
	6	5	5	16

Se o registrador rs for igual ao valor de imm com sinal estendido, gera uma exceção de Trap.

Trap se não for igual

teq rs, rt	0	rs	rt	0	0x36
	6	5	5	10	6

Se o registrador rs não for igual ao registrador rt, gera uma exceção de Trap.

Trap se não for igual imediato

teqi rs, imm	1	rs	0xe	imm
	6	5	5	16

Se o registrador rs não for igual ao valor de imm com sinal estendido, gera uma exceção de Trap.

Trap se for maior ou igual

tge rs, rt

0	rs	rt	0	0x30
6	5	5	10	6

Trap sem sinal se for maior ou igual

tgeu rs, rt

0	rs	rt	0	0x31
6	5	5	10	6

Se o registrador rs for maior ou igual ao registrador rt, gera uma exceção de Trap.

Trap se for maior ou igual imediato

tgei rs, imm

1	rs	8	imm
6	5	5	16

Trap sem sinal se for maior ou igual imediato

tgeiu rs, imm

1	rs	9	imm
6	5	5	16

Se o registrador rs for maior ou igual ao valor de imm com sinal estendido, gera uma exceção de Trap.

Trap se for menor

tlt rs, rt

0	rs	rt	0	0x32
6	5	5	10	6

Trap sem sinal se rs for menor

tltu rs, rt

0	rs	rt	0	0x33
6	5	5	10	6

Se o registrador rs for menor que o registrador rt, gera uma exceção de Trap.

Trap se rs for menor imediato

tlti rs, imm

1	rs	a	imm
6	5	5	16

Trap sem sinal se rs for menor imediato

tltiu rs, imm

1	rs	b	imm
6	5	5	16

Se o registrador rs for menor do que o valor de imm com sinal estendido, gera uma exceção de Trap.

Instruções load

Load endereço

la rdest, endereço *pseudoinstrução*

Carrega o *endereço* calculado — não o conteúdo do local — para o

registrador rdest.

Load byte

1b rt, endereço	0x20	rs	rt	Offset
	6	5	5	16

Load byte sem sinal

1bu rt, endereço	0x24	rs	rt	Offset
	6	5	5	16

Carrega o byte no *endereço* para o registrador *rt*. O byte tem sinal estendido por 1b, mas não por 1bu.

Load halfword

1h rt, endereço	0x21	rs	rt	Offset
	6	5	5	16

Load halfword sem sinal

1hu rt, endereço	0x25	rs	rt	Offset
	6	5	5	16

Carrega a quantidade de 16 bits (halfword) no *endereço* para o registrador *rt*. A halfword tem sinal estendido por 1h, mas não por 1hu.

Load word

lw rt, endereço	0x23	rs	rt	Offset
	6	5	5	16

Carrega a quantidade de 32 bits (palavra) no *endereço* para o registrador *rt*.

Load word coprocessador 1

lwcl ft, endereço	0x31	rs	rt	Offset
	6	5	5	16

Carrega a palavra no *endereço* para o registrador *ft* da unidade de ponto flutuante.

Load word à esquerda

lw1 rt, endereço	0x22	rs	rt	Offset
	6	5	5	16

Load word à direita

lwr rt, endereço	0x26	rs	rt	Offset
	6	5	5	16

Carrega os bytes da esquerda (direita) da palavra do *endereço* possivelmente não alinhado para o registrador *rt*.

Load doubleword

ld rdest, endereço *pseudoinstrução*

Carrega a quantidade de 64 bits no *endereço* para os registradores *rdest* e *rdest + 1*.

Load halfword não alinhada

ulh rdest, endereço pseudoinstrução

Load halfword sem sinal não alinhada

ulhu rdest, endereço pseudoinstrução

Carrega a quantidade de 16 bits (halfword) no *endereço* possivelmente não alinhado para o registrador *rdest*. A halfword tem extensão de sinal por *ulh*, mas não *ulhu*.

Load word não alinhada

ulw rdest, endereço pseudoinstrução

Carrega a quantidade de 32 bits (palavra) no *endereço* possivelmente não alinhado para o registrador *rdest*.

Load Linked

11	rt, endereço	0x30	rs	rt	Offset
		6	5	5	16

Carrega a quantidade de 32 bits (palavra) no *endereço* para o registrador *rt* e inicia uma operação ler-modificar-escrever indivisível. Essa operação é concluída por uma instrução de store condicional (sc), que falhará se outro processador escrever no bloco que contém a palavra carregada. Como o SPIM não simula processadores múltiplos, a operação de store condicional sempre tem sucesso.

Instruções store

Store byte

sb rt, endereço	0x28	rs	rt	Offset
	6	5	5	16

Armazena o byte baixo do registrador rt no *endereço*.

Store halfword

sh rt, endereço	0x29	rs	rt	Offset
	6	5	5	16

Armazena a halfword baixa do registrador rt no *endereço*.

Store word

sw rt, endereço	0x2b	rs	rt	Offset
	6	5	5	16

Armazena a palavra do registrador rt no *endereço*.

Store word coprocessador 1

swcl ft, address	0x31	rs	ft	Offset
	6	5	5	16

Armazena o valor de ponto flutuante no registrador ft do coprocessador de ponto flutuante no *endereço*.

Store double coprocessador 1

sdcl ft, address	0x3d	rs	ft	Offset
	6	5	5	16

Armazena o valor de ponto flutuante da dupla palavra nos registradores ft e ft + 1 do coprocessador de ponto flutuante em *endereço*. O registrador ft precisa ser um número par.

Store word à esquerda

swl rt, address	0x2a	rs	rt	Offset
	6	5	5	16

Store word à direita

swr rt, address	0x2e	rs	rt	Offset
	6	5	5	16

Armazena os bytes da esquerda (direita) do registrador rt no *endereço* possivelmente não alinhado.

Store doubleword

sd rsrc, endereço *pseudoinstrução*

Armazena a quantidade de 64 bits nos registradores rsrc e rsrc + 1 no *endereço*.

Store halfword não alinhada

ush rsrc, endereço *pseudoinstrução*

Armazena a halfword baixa do registrador rsrc no *endereço* possivelmente não alinhado.

Store word não alinhada

usw rsrc, endereço *pseudoinstrução*

Armazena a palavra do registrador rsrc no *endereço* possivelmente não alinhado.

Store condicional

sc rt, address	0x38	rs	rt	Offset
	6	5	5	16

Armazena a quantidade de 32 bits (palavra) no endereço rt para a memória no *endereço* e completa uma operação ler-modificar-escrever indivisível. Se essa operação indivisível tiver sucesso, a palavra da memória será modificada e o registrador rt será colocado em 1. Se a operação indivisível falhar porque outro processador escreveu em um local no bloco contendo a palavra endereçada, essa instrução não modifica a memória e escreve 0 no registrador rt. Como o SPIM não simula diversos processadores, a instrução sempre tem sucesso.

Instruções para movimentação de dados

Move

move rdest, rsrc *pseudoinstrução*

Move o registrador rsrc para rdest.

Move de hi

mfhi rd	0	0	rd	0	0x10
	6	10	5	5	6

Move de lo

`mflo rd`

0	0	rd	0	0x12
6	10	5	5	6

A unidade de multiplicação e divisão produz seu resultado em dois registradores adicionais, `hi` e `lo`. Essas instruções movem os valores de, e para, esses registradores. As pseudoinstruções de multiplicação, divisão e resto que fazem com que essa unidade pareça operar sobre os registradores gerais movem o resultado depois que o cálculo terminar.

Move o registrador `hi` (`lo`) para o registrador `rd`.

Move para `hi`

`mthi rs`

0	rs	0	0x11
6	5	15	6

Move para `lo`

`mtlo rs`

0	rs	0	0x13
6	5	15	6

Move o registrador `rs` para o registrador `hi` (`lo`).

Move do coprocessador 0

`mfc0 rt, rd`

0x10	0	rt	rd	0
6	5	5	5	11

Move do coprocessador 1

mfcl rt, fs	0x11	0	rt	fs	0
	6	5	5	5	11

Os coprocessadores têm seus próprios conjuntos de registradores. Essas instruções movem valores entre esses registradores e os registradores da CPU.

Move o registrador rd em um coprocessador (registrador fs na FPU) para o registrador rt da CPU. A unidade de ponto flutuante é o coprocessador 1.

Move double do coprocessador 1

`mfc1.d rdest, frsrc1` *pseudoinstrução*

Move os registradores de ponto flutuante frsc1 e frsrc1 + 1 para os registradores da CPU rdest e rdest + 1.

Move para coprocessador 0

mtc0 rd, rt	0x10	4	rt	rd	0
	6	5	5	5	11

Move para coprocessador 1

mtc1 rd, fs	0x11	4	rt	fs	0
	6	5	5	5	11

Move o registrador da CPU rt para o registrador rd em um coprocessador (registrador fs na FPU).

Move condicional diferente de zero

movn rd, rs, rt	0	rs	rt	rd	0xb
	6	5	5	5	11

Move o registrador *rs* para o registrador *rd* se o registrador *rt* não for 0.

Move condicional zero

movz rd, rs, rt	0	rs	rt	rd	Oxa
	6	5	5	5	11

Move o registrador *rs* para o registrador *rd* se o registrador *rt* for 0.

Move condicional em caso de FP falso

movf rd, rs, cc	0	rs	cc	0	rd	0	1
	6	5	3	2	5	5	6

Move o registrador da CPU *rs* para o registrador *rd* se o flag de código de condição da FPU número *cc* for 0. Se *cc* for omitido da instrução, o flag de código de condição 0 será assumido.

Move condicional em caso de FP verdadeiro

movt rd, rs, cc	0	rs	cc	1	rd	0	1
	6	5	3	2	5	5	6

Move o registrador da CPU *rs* para o registrador *rd* se o flag de código de condição da FPU número *cc* for 1. Se *cc* for omitido da instrução, o bit de código de condição 0 é assumido.

Instruções de ponto flutuante

O MIPS possui um coprocessador de ponto flutuante (número 1) que opera sobre números de ponto flutuante de precisão simples (32 bits) e precisão dupla (64 bits). Esse coprocessador tem seus próprios registradores, que são numerados de

\$f0 a \$f31. Como esses registradores possuem apenas 32 bits, dois deles são necessários para manter doubles, de modo que somente registradores de ponto flutuante com números pares podem manter valores de precisão dupla. O coprocessador de ponto flutuante também possui 8 flags de código de condição (cc), numerados de 0 a 7, que são alterados por instruções de comparação e testados por instruções de desvio (bc1f ou bc1t) e instruções move condicionais.

Os valores são movidos para dentro e para fora desses registradores uma palavra (32 bits) de cada vez pelas instruções lwc1, swc1, mtc1 e mfc1 ou um double (64 bits) de cada vez por ldc1 e sdc1, descritos anteriormente, ou pela pseudoinstruções l.s, l.d, s.s e s.d, descritas a seguir.

Nas instruções reais a seguir, os bits 21-26 são 0 para precisão simples e 1 para precisão double. Nas pseudoinstruções a seguir, fdest é um registrador de ponto flutuante (por exemplo, \$f2).

Valor absoluto de ponto flutuante double

abs.d	fd	,	fs	0x11	1	0	fs	fd	5
				6	5	5	5	5	6

Valor absoluto de ponto flutuante single

abs.s	fd	,	fs	0x11	0	0	fs	fd	5
				6	5	5	5	5	6

Calcula o valor absoluto do double (single) de ponto flutuante no registrador fs e o coloca no registrador fd.

Adição de ponto flutuante double

add.d	fd	,	fs	,	ft	0x11	0x11	ft	fs	fd	0
						6	5	5	5	5	6

Adição de ponto flutuante single

add.s	fd,	fs,	ft	0x11	0x10	ft	fs	fd	0
				6	5	5	5	5	6

Calcula a soma dos doubles (singles) de ponto flutuante nos registradores fs e ft e a coloca no registrador fd.

Teto de ponto flutuante para word

ceil.w.d	fd,	fs	0x11	0x11	0	fs	fd	0xe
			6	5	5	5	5	6
ceil.w.s	fd,	fs	0x11	0x10	0	fs	fd	0xe

Calcula o teto do double (single) de ponto flutuante no registrador fs, converte para um valor de ponto fixo de 32 bits e coloca a palavra resultante no registrador fd.

Comparação igual double

c.eq.d	cc	fs,	ft	0x11	0x11	ft	fs	cc	0	FC	2
				6	5	5	5	3	2	2	4

Comparação igual single

c.eq.s	cc	fs,	ft	0x11	0x10	ft	fs	cc	0	FC	2
				6	5	5	5	3	2	2	4

Compara o double (single) de ponto flutuante no registrador fs com aquele em

`ft` e coloca o flag de condição de ponto flutuante `cc` em 1 se forem iguais. Se `cc` for omitido, o flag de código de condição 0 é assumido.

Comparação menor ou igual double

c.le.d cc fs, ft	0x11	0x11	ft	fs	cc	0	FC	0xe
	6	5	5	5	3	2	2	4

Comparação menor ou igual single

c.le.s cc fs, ft	0x11	0x10	ft	fs	cc	0	FC	0xe
	6	5	5	5	3	2	2	4

Compara o double (single) de ponto flutuante no registrador `fs` com aquele no `ft` e coloca o flag de condição de ponto flutuante `cc` em 1 se o primeiro for menor ou igual ao segundo. Se o `cc` for omitido, o flag de código de condição 0 é assumido.

Comparação menor que double

c.lt.d cc fs, ft	0x11	0x11	ft	fs	cc	0	FC	0xc
	6	5	5	5	3	2	2	4

Comparação menor que single

c.lt.s cc fs, ft	0x11	0x10	ft	fs	cc	0	FC	0xc
	6	5	5	5	3	2	2	4

Compara o double (single) de ponto flutuante no registrador `fs` com aquele no `ft` e coloca o flag de condição de ponto flutuante `cc` em 1 se o primeiro for

menor que o segundo. Se o *cc* for omitido, o flag de código de condição 0 é assumido.

Converte single para double

cvt.d.s fd, fs

0x11	0x10	0	fs	fd	0x21
6	5	5	5	5	6

Converte integer para double

cvt.d.w fd, fs

0x11	0x14	0	fs	fd	0x21
6	5	5	5	5	6

Converte o número de ponto flutuante de precisão simples ou inteiro no registrador fs para um número de precisão dupla (simples) e o coloca no registrador fd.

Converte double para single

cvt.s.d fd, fs

0x11	0x11	0	fs	fd	0x20
6	5	5	5	5	6

Converte integer para single

cvt.s.w fd, fs

0x11	0x14	0	fs	fd	0x20
6	5	5	5	5	6

Converte o número de ponto flutuante de precisão dupla ou inteiro no registrador fs para um número de precisão simples e o coloca no registrador fd.

Converte double para integer

cvt.w.d fd, fs

0x11	0x11	0	fs	fd	0x24
6	5	5	5	5	6

Converte single para integer

cvt.w.s fd, fs

0x11	0x10	0	fs	fd	0x24
6	5	5	5	5	6

Converte o número de ponto flutuante de precisão dupla ou simples no registrador fs para um inteiro e o coloca no registrador fd.

Divisão de ponto flutuante double

div.d fd, fs, ft

0x11	0x11	ft	fs	fd	3
6	5	5	5	5	6

Divisão de ponto flutuante single

div.s fd, fs, ft

0x11	0x10	ft	fs	fd	3
6	5	5	5	5	6

Calcula o quociente dos números de ponto flutuante de precisão dupla (simples) nos registradores fs e ft e o coloca no registrador fd.

Piso de ponto flutuante para palavra

floor.w.d	fd,	fs	0x11	0x11	0	fs	fd	0xf
			6	5	5	5	5	6

floor.w.s	fd,	fs	0x11	0x10	0	fs	fd	0xf
			6	5	5	5	5	6

Calcula o piso do número de ponto flutuante de precisão dupla (simples) no registrador fs e coloca a palavra resultante no registrador fd.

Carrega double de ponto flutuante

l.d fdest, endereço *pseudoinstrução*

Carrega single de ponto flutuante

l.s fdest, endereço *pseudoinstrução*

Carrega o número de ponto flutuante de precisão dupla (simples) em *endereço* para o registrador fdest.

Move ponto flutuante double

mov.d	fd,	fs	0x11	0x11	0	fs	fd	6
			6	5	5	5	5	6

Move ponto flutuante single

mov.s	fd,	fs	0x11	0x10	0	fs	fd	6
			6	5	5	5	5	6

Move o número de ponto flutuante de precisão dupla (simples) do registrador fs para o registrador fd.

Move condicional de ponto flutuante double se falso

movf.d	fd,	fs,	cc	0x11	0x11	cc	0	fs	fd	0x11
				6	5	3	2	5	5	6

Move condicional de ponto flutuante single se falso

movf.s	fd,	fs,	cc	0x11	0x10	cc	0	fs	fd	0x11
				6	5	3	2	5	5	6

Move o número de ponto flutuante de precisão dupla (simples) do registrador fs para o registrador fd se o flag do código de condição cc for 0. Se o cc for omitido, o flag de código de condição 0 é assumido.

Move condicional de ponto flutuante double se verdadeiro

movt.d	fd,	fs,	cc	0x11	0x11	cc	1	fs	fd	0x11
				6	5	3	2	5	5	6

Move condicional de ponto flutuante single se verdadeiro

movt.s	fd,	fs,	cc	0x11	0x10	cc	1	fs	fd	0x11
				6	5	3	2	5	5	6

Move o double (single) de ponto flutuante do registrador fs para o registrador fd se o flag do código de condição cc for 1. Se o cc for omitido, o flag do código de condição 0 será assumido.

Move ponto flutuante double condicional se não for zero

movn.d	fd,	fs,	rt	0x11	0x11	rt	fs	fd	0x13
				6	5	5	5	5	6

Move ponto flutuante single condicional se não for zero

movn.s fd, fs, rt	0x11	0x10	rt	fs	fd	0x13
	6	5	5	5	5	6

Move o número de ponto flutuante double (single) do registrador fs para o registrador fd se o registrador rt do processador não for 0.

Move ponto flutuante double condicional se for zero

movz.d fd, fs, rt	0x11	0x11	rt	fs	fd	0x12
	6	5	5	5	5	6

Move ponto flutuante single condicional se for zero

movz.s fd, fs, rt	0x11	0x10	rt	fs	fd	0x12
	6	5	5	5	5	6

Move o número de ponto flutuante double (single) do registrador fs para o registrador fd se o registrador rt do processador for 0.

Multiplicação de ponto flutuante double

mul.d fd, fs, ft	0x11	0x11	ft	fs	fd	2
	6	5	5	5	5	6

Multiplicação de ponto flutuante single

mul.s fd, fs, ft	0x11	0x10	ft	fs	fd	2
	6	5	5	5	5	6

Calcula o produto dos números de ponto flutuante double (single) nos registradores fs e ft e o coloca no registrador fd.

Negação double

neg.d fd, fs	0x11	0x11	0	fs	fd	7
	6	5	5	5	5	6

Negação single

neg.s fd, fs	0x11	0x10	0	fs	fd	7
	6	5	5	5	5	6

Nega o número de ponto flutuante double (single) no registrador fs e o coloca no registrador fd.

Arredondamento de ponto flutuante para palavra

round.w.d fd, fs	0x11	0x11	0	fs	fd	0xc
	6	5	5	5	5	6

round.w.s fd, fs	0x11	0x10	0	fs	fd	0xc
	6	5	5	5	5	6

Arredonda o valor de ponto flutuante double (single) no registrador fs, converte para um valor de ponto fixo de 32 bits e coloca a palavra resultante no registrador fd.

Raiz quadrada de double

sqrt.d fd, fs

0x11	0x11	0	fs	fd	4
6	5	5	5	5	6

Raiz quadrada de single

sqrt.s fd, fs

0x11	0x10	0	fs	fd	4
6	5	5	5	5	6

Calcula a raiz quadrada do número de ponto flutuante double (single) no registrador fs e a coloca no registrador fd.

Store de ponto flutuante double

s.d fdest, endereço *pseudoinstrução*

Store de ponto flutuante single

s.s fdest, endereço *pseudoinstrução*

Armazena o número de ponto flutuante double (single) no registrador fdest em *endereço*.

Subtração de ponto flutuante double

sub.d fd, fs, ft

0x11	0x11	ft	fs	fd	1
6	5	5	5	5	6

Subtração de ponto flutuante single

sub.s	fd,	fs,	ft	0x11	0x10	ft	fs	fd	1
				6	5	5	5	5	6

Calcula a diferença dos números de ponto flutuante double (single) nos registradores fs e ft e a coloca no registrador fd.

Truncamento de ponto flutuante para palavra

trunc.w.d	fd,	fs	0x11	0x11	0	fs	fd	0xd
			6	5	5	5	5	6

trunc.w.s	fd,	fs	0x11	0x10	0	fs	fd	0xd
			6	5	5	5	5	6

Trunca o valor de ponto flutuante double (single) no registrador fs, converte para um valor de ponto fixo de 32 bits e coloca a palavra resultante no registrador fd.

Instruções de exceção e interrupção

Retorno de exceção

eret	0x10	1	0	0x18
	6	1	19	6

Coloca em 0 o bit EXL no registrador Status do coprocessador 0 e retorna à instrução apontada pelo registrador EPC do coprocessador 0.

Chamada ao sistema

syscall

0	0	0xc
6	20	6

O registrador \$v0 contém o número da chamada ao sistema (veja [Figura A.9.1](#)) fornecido pelo SPIM.

Break

break code

0	code	0xd
6	20	6

Causa a exceção *código*. A Exceção 1 é reservada para o depurador.

Nop

nop

0	0	0	0	0	0
6	5	5	5	5	6

Não faz nada.

A.11. Comentários finais

A programação em assembly exige que um programador escolha entre os recursos úteis das linguagens de alto nível — como estruturas de dados, verificação de tipo e construções de controle — e o controle completo sobre as instruções que um computador executa. Restrições externas sobre algumas aplicações, como o tempo de resposta ou o tamanho do programa, exigem que um programador preste muita atenção a cada instrução. No entanto, o custo desse nível de atenção são programas em assembly maiores, mais demorados para escrever e mais difíceis de manter do que os programas em linguagem de alto nível.

Além do mais, três tendências estão reduzindo a necessidade de escrever programas em linguagem assembly. A primeira tendência é em direção à melhoria dos compiladores. Os compiladores modernos produzem código comparável ao melhor código escrito manualmente — e, às vezes, melhor ainda. A segunda tendência é a introdução de novos processadores, que não apenas são mais rápidos, mas, no caso de processadores que executam várias instruções ao mesmo tempo, também são mais difíceis de programar manualmente. Além disso, a rápida evolução dos computadores modernos favorece os programas em linguagem de alto nível que não estejam presos a uma única arquitetura. Finalmente, temos testemunhado uma tendência em direção a aplicações cada vez mais complexas, caracterizadas por interfaces gráficas complexas e muito mais recursos do que seus predecessores. Grandes aplicações são escritas por equipes de programadores e exigem recursos de modularidade e verificação semântica fornecidos pelas linguagens de alto nível.

Leitura adicional

Aho, A., R. Sethi e J. Ullman [1985]. *Compilers: Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley.

Ligeiramente desatualizado e faltando a cobertura das arquiteturas modernas, mas ainda é a referência padrão sobre compiladores.

Sweetman, D. [1999]. *See MIPS Run*, San Francisco CA: Morgan Kaufmann Publishers.

Uma introdução completa, detalhada e envolvente sobre o conjunto de instruções do MIPS e a programação em assembly nessas máquinas.

A documentação detalhada sobre a arquitetura MIPS-32 está disponível na Web:

MIPS32™ Architecture for Programmers Volume I: Introduction to the MIPS-32™ Architecture

(<http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArch2B-MIPS32INT-AFP-02.00.pdf/getDownload>)

MIPS32™ Architecture for Programmers Volume II: The MIPS-32 Instruction Set

(<http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArch2B-MIPS32BIS-AFP-02.00.pdf/getDownload>)

MIPS32™ Architecture for Programmers Volume III: The MIPS-32 Privileged Resource Architecture

(<http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArch2B-MIPS32PRA-AFP-02.00.pdf/getDownload>)

A.12. Exercícios

- A.1 [5] <§A.5>** A [Seção A.5](#) descreveu como a memória é dividida na maioria dos sistemas MIPS. Proponha outra maneira de dividir a memória, que cumpra os mesmos objetivos.
- A.2 [20] <§A.6>** Reescreva o código para `fact` utilizando menos instruções.
- A.3 [5] <§A.7>** É seguro que um programa do usuário utilize os registradores `$k0` ou `$k1`?
- A.4 [25] <§A.7>** A [Seção A.7](#) contém código para um handler de exceção muito simples. Um problema sério com esse handler é que ele desativa as interrupções por um longo tempo. Isso significa que as interrupções de um dispositivo de E/S rápido podem ser perdidas. Escreva um handler de exceção melhor, que não possa ser interrompido mas que ative as interrupções o mais rápido possível.
- A.5 [15] <§A.7>** O handler de exceção simples sempre desvia para a instrução após a exceção. Isso funciona bem, a menos que a instrução que causou a exceção esteja no delay slot de um desvio. Nesse caso, a próxima instrução será o alvo do desvio. Escreva um handler melhor, que use o registrador EPC para determinar qual instrução deverá ser executada após a exceção.
- A.6 [5] <§A.9>** Usando o SPIM, escreva e teste um programa de calculadora que leia inteiros repetidamente e os adicione a um acumulador. O programa deverá parar quando receber uma entrada 0, exibindo a soma nesse ponto. Use as chamadas do sistema do SPIM descritas na [Seção A.9](#).
- A.7 [5] <§A.9>** Usando o SPIM, escreva e teste um programa que leia três inteiros e exiba a soma dos dois maiores desses três. Use as chamadas do sistema do SPIM descritas na [Seção A.9](#). Você pode usar o critério de desempate que desejar.
- A.8 [5] <§A.9>** Usando o SPIM, escreva e teste um programa que leia um inteiro positivo usando as chamadas do sistema do SPIM. Se o inteiro não for positivo, o programa deverá terminar com a mensagem “Entrada inválida”; caso contrário, o programa deverá exibir os nomes dos dígitos dos inteiros por extenso, delimitados por, exatamente, um espaço. Por exemplo, se o usuário informou “728”, a saída deverá ser “Sete Dois Oito”.
- A.9 [25] <§A.9>** Escreva e teste um programa em assembly do MIPS para calcular e exibir os 100 primeiros números primos. Um número n é primo se ele só puder ser dividido exatamente por ele mesmo e por 1. Duas rotinas

deverão ser implementadas:

- `testa_primo(n)` retorna 1 se n for primo e 0 se n não for primo.
- `main()` percorre os inteiros, testando se cada um deles é primo. Exibe os 100 primeiros números primos.

Teste seus programas executando-os no SPIM.

A.10 [10] <§§A.6, A.9> Usando o SPIM, escreva e teste um programa recursivo para solucionar um problema matemático clássico, denominado Torres de Hanói. (Isso exigirá o uso de frames de pilha para admitir a recursão.) O problema consiste em três pinos (1, 2 e 3) e n discos (o número n pode variar; os valores típicos poderiam estar no intervalo de 1 a 8). O disco 1 é menor que o disco 2, que, por sua vez, é menor que o disco 3, e assim por diante, com o disco n sendo o maior. Inicialmente, todos os discos estão no pino 1, começando com o disco n na parte inferior, o disco $n - 1$ acima dele, e assim por diante, até o disco 1 no topo. O objetivo é mover todos os discos para o pino 2. Você só pode mover um disco de cada vez, ou seja, o disco superior de qualquer um dos três pinos para o topo de qualquer um dos outros dois pinos. Além do mais, existe uma restrição: não é possível colocar um disco maior em cima de um disco menor.

O programa em C a seguir pode ser usado como uma base para a escrita do seu programa em assembly:

```

/* move n discos menores de start para finish usando
extra */

void hanoi(int n, int start, int finish, int extra){
    if(n != 0){
        hanoi(n-1, start, extra, finish);
        print_string("Move disk");
        print_int(n);
        print_string("from peg");
        print_int(start);
        print_string("to peg");
        print_int(finish);
        print_string(".\n");
        hanoi(n-1, extra, finish, start);
    }
}
main(){
    int n;
    print_string("Enter number of disks>");
    n = read_int();
    hanoi(n, 1, 2, 3);
    return 0;
}

```

¹ Esta seção discute as exceções na arquitetura MIPS-32, que é o que o SPIM implementa na Versão 7.0 em diante. As versões anteriores do SPIM implementaram a arquitetura MIPS-I, que tratava exceções de forma ligeiramente diferente. A conversão de programas a partir dessas versões para execução no MIPS-32 não deverá ser difícil, pois as mudanças são limitadas aos campos dos registradores Status e Cause e à substituição da instrução rfe pela instrução eret.

² As primeiras versões do SPIM (antes da 7.0) implementaram a arquitetura MIPS-1 utilizada nos processadores MIPS R2000 originais. Essa arquitetura é quase um subconjunto apropriado da arquitetura MIPS-32, sendo que a diferença é a maneira como as exceções são tratadas. O MIPS-32 também introduziu aproximadamente 60 novas instruções, que são aceitas pelo SPIM. Os programas executados nas versões anteriores do SPIM e que não usavam exceções deverão ser executados sem modificação nas versões mais recentes do SPIM. Os programas que usavam exceções exigirão pequenas mudanças.

APÊNDICE

B

Fundamentos do Projeto Lógico

Eu sempre gostei dessa palavra: Boolean.

Claude Shannon

IEEE Spectrum, abril de 1992

(A tese do professor Shannon mostrou que a álgebra inventada por George Boole no século XIX poderia representar o funcionamento das chaves elétricas.)

B.1 Introdução

B.2 Portas, tabelas verdade e equações lógicas

B.3 Lógica combinacional

B.4 Usando uma linguagem de descrição de hardware

B.5 Construindo uma unidade lógica e aritmética básica

B.6 Adição mais rápida: Carry Lookahead

B.7 Clocks

B.8 Elementos de memória: Flip-flops, latches e registradores

B.9 Elementos de memória: SRAMs e DRAMs

B.10 Máquinas de estados finitos

B.11 Metodologias de temporização

B.12 Dispositivos programáveis em campo

B.13 Comentários finais

B.14 Exercícios

B.1. Introdução

Este apêndice oferece uma rápida discussão sobre os fundamentos do projeto lógico. Ele não substitui um curso sobre projeto lógico nem permitirá projetar sistemas lógicos funcionais significativos. Contudo, se você tiver pouca ou nenhuma experiência com projeto lógico, este apêndice oferecerá uma base suficiente para entender todo o material deste livro. Além disso, se você quiser entender parte da motivação por trás da forma como os computadores são implementados, este material servirá como uma introdução útil. Se a sua curiosidade for aumentada, mas não saciada por este apêndice, as referências ao final oferecem fontes de informação adicionais.

A [Seção B.2](#) introduz os blocos de montagem básicos da lógica, a saber, *portas lógicas*. A [Seção B.3](#) utiliza esses blocos de montagem para construir sistemas lógicos *combinacionais* simples, que não contêm memória. Se você já teve alguma experiência com sistemas lógicos ou digitais, provavelmente estará acostumado com o material dessas duas primeiras seções. A [Seção B.5](#) mostra como usar os conceitos das [Seções B.2](#) e [B.3](#) para projetar uma ALU para o processador MIPS. A [Seção B.6](#) mostra como criar um somador rápido e pode ser pulada sem problemas se você não estiver interessado neste assunto. A [Seção B.7](#) é uma introdução rápida ao assunto de clocking, necessário para discutirmos como funcionam os elementos de memória. A [Seção B.8](#) introduz os elementos de memória, e a [Seção B.9](#) a estende para focalizar em memórias de acesso aleatório; ele descreve as características importantes para entender como são usadas, conforme discutimos no [Capítulo 4](#) e a base que motiva muitos dos aspectos do projeto de hierarquia de memória no [Capítulo 5](#). A [Seção B.10](#) descreve o projeto e o uso das máquinas de estados finitos, que são blocos lógicos sequenciais. Se você pretende ler apenas o material sobre controle, no [Capítulo 4](#), poderá passar superficialmente pelos apêndices, mas deverá ter alguma familiaridade com todo o material, exceto a [Seção B.11](#), pois ela serve para aqueles que desejam ter um conhecimento mais profundo das metodologias de temporização. Ela explica os fundamentos de como funciona o clock acionado por transição, introduz outro esquema de temporização e descreve rapidamente o problema de sincronizar entradas assíncronas.

Ao longo do apêndice, onde for apropriado, também incluímos segmentos em Verilog para demonstrar como a lógica pode ser representada em Verilog, que apresentaremos na [Seção B.4](#).

B.2. Portas, tabelas verdade e equações lógicas

A eletrônica dentro de um computador moderno é *digital* e opera apenas com dois níveis de tensão: alta e baixa. Todos os outros valores de tensão são temporários e ocorrem na transição entre os valores. (Conforme discutimos mais adiante nesta seção, uma armadilha possível no projeto digital é a amostragem de um sinal quando ele não é nitidamente alto ou baixo.) O fato de que os computadores são digitais também é um motivo fundamental para eles usarem números binários, pois um sistema binário corresponde à abstração básica inerente à eletrônica. Em diversas famílias lógicas, os valores e os relacionamentos entre os dois valores de tensão diferem. Assim, em vez de se referir aos níveis de tensão, falamos sobre sinais que são (logicamente) verdadeiros, 1, **ativos**; ou sinais que são (logicamente) falsos, 0, **inativos**. Os valores 0 e 1 são chamados *complementos* ou *inversos* um do outro.

sinal ativo

Um sinal que é (logicamente) verdadeiro ou 1.

sinal inativo

Um sinal que é (logicamente) falso ou 0.

Os blocos lógicos são categorizados como um dentre dois tipos, dependendo se contêm memória ou não. Os blocos sem memória são chamados *combinacionais*; a saída de um bloco combinacional depende apenas da entrada atual. Nos blocos com memória, as saídas podem depender das entradas e do valor armazenado na memória, chamado *estado* do bloco lógico. Nesta seção e na seguinte, focalizaremos apenas a **lógica combinacional**. Depois de introduzir diferentes elementos de memória na [Seção B.8](#), descreveremos como é projetada a **lógica sequencial**, que é a lógica que inclui estado.

lógica combinacional

Um sistema lógico cujos blocos não contêm memória e, portanto, calculam a

mesma saída dada à mesma entrada.

lógica sequencial

Um grupo de elementos lógicos que contêm memória e cujo valor, portanto, depende das entradas e também do conteúdo atual da memória.

Tabelas verdade

Como um bloco de lógica combinacional não contém memória, ele pode ser especificado completamente definindo os valores das saídas para cada conjunto de valores de entrada possíveis. Essa descrição normalmente é dada como uma *tabela verdade*. Para um bloco lógico com n entradas, existem 2^n entradas na tabela verdade, pois existem todas essas combinações possíveis de valores de entrada. Cada entrada especifica o valor de todas as saídas para essa combinação de entrada em particular.

Tabelas verdade

Exemplo

Considere uma função lógica com três entradas, A , B e C , e três saídas, D , E e F . A função é definida da seguinte maneira: D é verdadeiro se pelo menos uma entrada for verdadeira, E é verdadeiro se exatamente duas entradas forem verdadeiras, e F é verdadeiro somente se todas as três entradas forem verdadeiras. Mostre a tabela verdade para essa função.

Resposta

A tabela verdade terá $2^3 = 8$ entradas. Aqui está ela:

Entradas			Saídas		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0

1	1	0	1	1	0
1	1	1	1	0	1

As tabelas verdade podem descrever qualquer função lógica combinacional; porém, elas aumentam de tamanho rapidamente e podem não ser fáceis de entender. Às vezes, queremos construir uma função lógica que será 0 para muitas combinações de entrada e usamos um atalho para especificar apenas as entradas da tabela verdade para as saídas diferentes de zero. Esta técnica é usada no [Capítulo 4](#).

Álgebra Booleana

Outra técnica é expressar a função lógica com equações lógicas. Isso é feito com o uso da *álgebra Booleana* (que tem o nome de Boole, um matemático do século XIX). Em álgebra Booleana, todas as variáveis possuem os valores 0 ou 1 e, nas formulações típicas, existem três operadores:

- O operador OR é escrito como $+$, como em $A + B$. O resultado de um operador OR é 1 se uma das variáveis for 1. A operação OR também é chamada *soma lógica*, pois seu resultado é 1 se um dos operandos for 1.
- O operador AND é escrito como \cdot , como em $A \cdot B$. O resultado de um operador AND é 1 somente se as duas entradas forem 1. A operação AND também é chamada de *produto lógico*, pois seu resultado é 1 apenas se os dois operandos forem 1.
- O operador unário NOT é escrito como A^- . O resultado de um operador NOT é 1 somente se a entrada for 0. A aplicação do operador NOT a um valor lógico resulta em uma inversão ou negação do valor (ou seja, se a entrada for 0, a saída será 1 e vice-versa).

Existem sete leis da álgebra Booleana úteis na manipulação de equações lógicas:

- Lei da identidade: $A + 0 = A$ e $A \cdot 1 = A$.
- Leis de zero e um: $A + 1 = 1$ e $A \cdot 0 = 0$.
- Leis inversas: $A + A^- = 1$ e $A \cdot A^- = 0$.
- Leis comutativas: $A + B = B + A$ e $A \cdot B = B \cdot A$.
- Leis associativas: $A + (B + C) = (A + B) + C$ e $A \cdot (B \cdot C) = (A \cdot B) \cdot C$.
- Leis distributivas: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ e $A + (B \cdot C) = (A + B) \cdot (A + C)$.

Além disso, existem dois outros teoremas úteis, chamados leis de DeMorgan,

que são discutidos com mais profundidade nos exercícios.

Qualquer conjunto de funções lógicas pode ser escrito como uma série de equações com uma saída no lado esquerdo de cada equação e, no lado direito, uma fórmula consistindo em variáveis e os três operadores anteriores.

Equações lógicas

Exemplo

Mostre as equações lógicas para as funções lógicas, D , E e F , descritas no exemplo anterior.

Resposta

Aqui está a equação para D :

$$D = A + B + C$$

F é igualmente simples:

$$F = A \cdot B \cdot C$$

E é um pouco complicada. Pense nela em duas partes: o que precisa ser verdadeiro para E ser verdadeiro (duas das três entradas precisam ser verdadeiras) e o que não pode ser verdadeiro (todas as três não podem ser verdadeiras). Assim, podemos escrever E como

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot (\overline{A \cdot B \cdot C})$$

Também podemos derivar E observando que E é verdadeiro apenas se exatamente duas das entradas forem verdadeiras. Então, podemos escrever E como um OR dos três termos possíveis que possuem duas entradas verdadeiras e uma entrada falsa:

$$E = (A \cdot B \cdot \bar{C}) + (A \cdot C \cdot \bar{B}) + (B \cdot C \cdot \bar{A})$$

A prova de que essas duas expressões são equivalentes é explorada nos exercícios.

Em Verilog, descrevemos a lógica combinacional, sempre que possível, usando a instrução assign, descrita, a partir da página B-23. Podemos escrever uma definição para E usando o operador OR exclusivo da Verilog, como em assign $E = (A \wedge B \wedge C) * (A + B + C) * (A * B * C)$, que é outra maneira de descrever essa função. D e F possuem representações ainda mais simples, que são exatamente como o código C correspondente: $D = A | B | C$ e $F = A \& B \& C$.

Portas lógicas

Blocos lógicos são criados a partir de **portas lógicas** que implementam as funções lógicas básicas. Por exemplo, uma porta AND implementa a função AND e uma porta OR implementa a função OR. Como AND e OR são comutativos e associativos, uma porta AND ou OR pode ter várias entradas, com a saída igual ao AND ou OR de todas as entradas. A função lógica NOT é implementada com um inversor que sempre possui uma única entrada. A representação padrão desses três blocos de montagem lógicos aparece na [Figura B.2.1](#).



FIGURA B.2.1 Desenho padrão para uma porta AND, porta OR e um inversor, mostrados da esquerda para a direita.

Os sinais à esquerda de cada símbolo são as entradas, enquanto a saída aparece à direita. As portas AND e OR possuem duas entradas. Os inversores possuem uma única entrada.

porta lógica

Um dispositivo que implementa funções lógicas básicas, como AND ou OR.

Em vez de desenhar inversores explicitamente, uma prática comum é acrescentar “bolhas” às entradas ou saída de uma porta lógica para fazer com que o valor lógico nessa linha de entrada ou de saída seja invertida. Por exemplo, a Figura B.2.2 mostra o diagrama lógico para a função $\overline{A} + B$, usando inversores explícitos à esquerda e usando as entradas e a saída em bolha à direita.

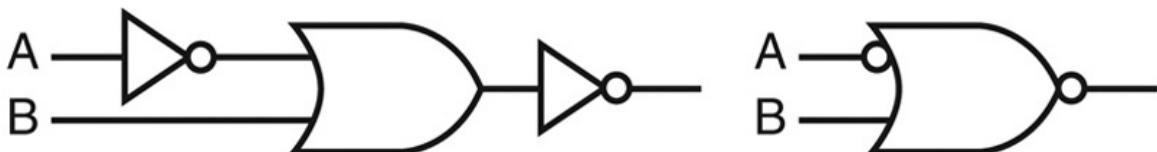


FIGURA B.2.2 Implementação de porta lógica de $\overline{A} + B$ usando inversões explícitas à esquerda e entradas e saídas em bolha à direita. A função lógica pode ser simplificada para $A \cdot B^{\sim}$ ou, em Verilog, $A \& \sim B$.

Qualquer função lógica pode ser construída usando portas AND, portas OR e inversão; vários dos exercícios dão a oportunidade de tentar implementar algumas funções lógicas comuns com portas. Na próxima seção, veremos como uma implementação de qualquer função lógica pode ser construída usando esse conhecimento.

De fato, todas as funções lógicas podem ser construídas com apenas um único tipo de porta lógica, se essa porta for inversora. As duas portas inversoras são chamadas **NOR** e **NAND** e correspondem às portas OR e AND invertidas, respectivamente. Portas NOR e NAND são chamadas *universais*, pois qualquer função lógica pode ser construída por meio desse tipo de porta. Os exercícios exploram melhor esse conceito.

Porta NOR

Uma porta OR invertida.

Porta NAND

Uma porta AND invertida.

Verifique você mesmo

Você consegue?

As duas expressões lógicas, a seguir, são equivalentes? Se não, encontre valores para as variáveis, mostrando que não são:

- $(A \cdot B \cdot \bar{C}) + (A \cdot C \cdot \bar{B}) + (B \cdot C \cdot \bar{A})$
- $B \cdot (A \cdot \bar{C} + C \cdot \bar{A})$

B.3. Lógica combinacional

Nesta seção, examinamos alguns dos maiores blocos de montagem lógicos mais utilizados e discutimos o projeto da lógica estruturada que pode ser implementado automaticamente, a partir de uma equação lógica ou tabela verdade por um programa de tradução. Por fim, discutimos a noção de um array de blocos lógicos.

Decodificadores

Um bloco lógico que usaremos na montagem de componentes maiores é um **decodificador**. O tipo mais comum de decodificador possui uma entrada de n bits e 2^n saídas, onde somente uma saída é ativada para cada combinação de entradas. Esse decodificador traduz a entrada de n bits para um sinal que corresponde ao valor binário da entrada de n bits. As saídas, portanto, são numeradas como, digamos, Out0, Out1, ..., Out $2^n - 1$. Se o valor da entrada for i , então Out*i* será verdadeiro e todas as outras saídas serão falsas. A Figura B.3.1 mostra um decodificador de 3 bits e a tabela verdade. Esse decodificador é chamado *decodificador 3 para 8*, pois existem 3 entradas e 8 (2^3) saídas. Há também um elemento lógico chamado de *codificador*, que realiza a função inversa de um decodificador, exigindo 2^n entradas e produzindo uma saída de n bits.

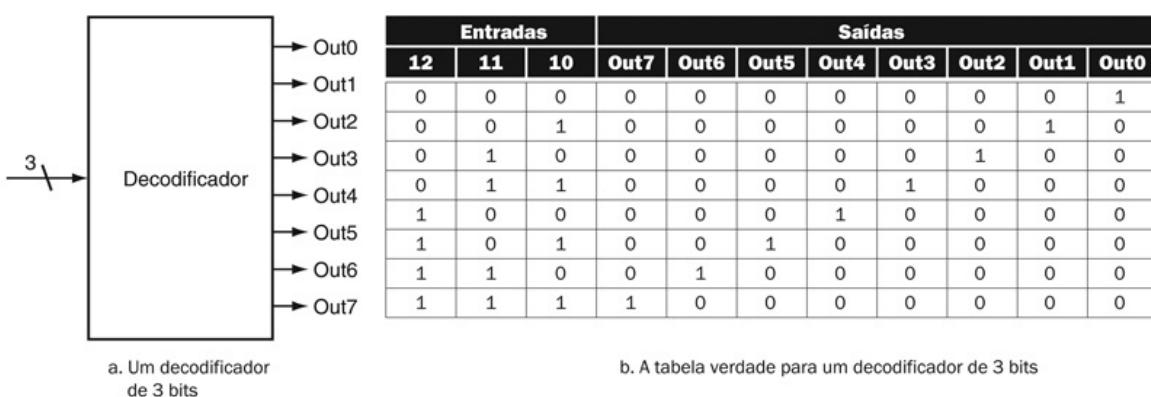


FIGURA B.3.1 Decodificador de 3 bits possui 3 entradas, chamadas 12, 11 e 10, e $(2^3) = 8$ saídas, chamadas de Out0 a Out7.

Somente a saída correspondente ao valor binário da entrada é

verdadeira, como mostra a tabela verdade. O rótulo 3, na entrada do decodificador, diz que o sinal de entrada possui 3 bits de largura.

decodificador

Um bloco lógico que possui uma entrada de n bits e 2^n saídas, onde somente uma saída é ativada para cada combinação de entradas.

Multiplexadores

Uma função lógica básica que usamos com muita frequência no [Capítulo 4](#) é o *multiplexador*. Um multiplexador poderia ser denominado de *seletor*, pois sua saída é uma das entradas selecionada por um controle. Considere o multiplexador de duas entradas. O lado esquerdo da [Figura B.3.2](#) mostra que esse multiplexador tem três entradas: dois valores de dados e um **valor seletor** (ou de **controle**). O valor seletor determina qual das entradas se torna a saída. Podemos representar a função lógica calculada por um multiplexador de duas entradas, mostrado em forma de portas lógicas no lado direito da [Figura B.3.2](#), como $C = (A \cdot S) + (B \cdot \bar{S})$.

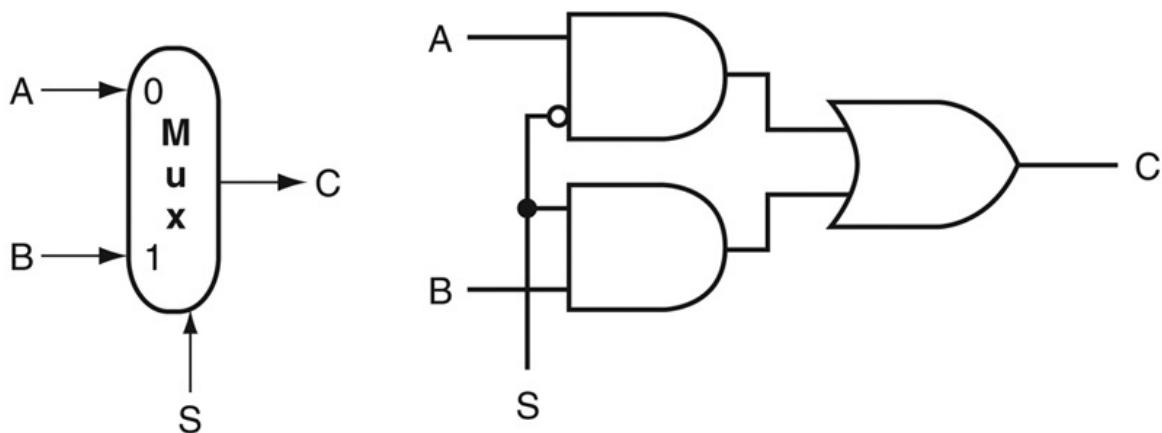


FIGURA B.3.2 Um multiplexador de duas entradas, à esquerda, e sua implementação com portas lógicas, à direita.

O multiplexador tem duas entradas de dados (A e B), que são rotuladas com 0 e 1, e uma entrada seletora (S), além de uma saída C. A implementação de multiplexadores em Verilog exige um pouco mais de trabalho, especialmente quando eles

possuem mais de duas entradas. Mostramos como fazer isso a partir da página B-23.

valor seletor

Também chamado **valor de controle**. O sinal de controle usado para selecionar um dos valores de entrada de um multiplexador como a saída do multiplexador.

Os multiplexadores podem ser criados com qualquer quantidade de entradas de dados. Quando existem apenas duas entradas, o seletor é um único sinal que seleciona uma das entradas se ela for verdadeira (1) e a outra se ela for falsa (0). Se houver n entradas de dados, terá de haver $\lfloor \log_2 n \rfloor$ entradas seletoras. Nesse caso, o multiplexador basicamente consiste em três partes:

1. Um decodificador que gera n sinais, cada um indicando um valor de entrada diferente
2. Um array de n portas AND, cada uma combinando com uma das entradas com um sinal do decodificador
3. Uma única porta OR grande, que incorpora as saídas das portas AND

Para associar as entradas com valores do seletor, rotulamos as entradas de dados numericamente (ou seja, 0, 1, 2, 3, ..., $n - 1$) e interpretamos as entradas do seletor de dados como um número binário. Às vezes, utilizamos um multiplexador com sinais de seletor não decodificados.

Os multiplexadores são representados combinacionalmente em Verilog usando expressões *if*. Para multiplexadores maiores, instruções *case* são mais convenientes, mas deve-se ter cuidado ao sintetizar a lógica combinacional.

Lógica de dois níveis e PLAs

Conforme indicado na seção anterior, qualquer função lógica pode ser implementada apenas com as funções AND, OR e NOT. Na verdade, um resultado muito mais forte é verdadeiro. Qualquer função lógica pode ser escrita em um formato canônico, no qual cada entrada é verdadeira ou uma variável complementada e existem apenas dois níveis de portas — um sendo AND e o outro OR — com uma possível inversão na saída final. Essa representação é chamada de *representação de dois níveis* e existem duas formas, chamadas **soma de produtos** e *produto de somas*. Uma representação da soma de produtos é uma

soma lógica (OR) de produtos (termos usando o operador AND); um produto de somas é exatamente o oposto. Em nosso exemplo anterior, tínhamos duas equações para a saída E :

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot (\overline{A} \cdot \overline{B} \cdot \overline{C})$$

$$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$$

Esta segunda equação está na forma de soma de produtos: ela possui dois níveis de lógica e as únicas inversões estão em variáveis individuais. A primeira equação possui três níveis de lógica.

soma de produtos

Uma forma de representação lógica que emprega uma soma lógica (OR) de produtos (termos unidos usando o operador AND).

Detalhamento

também podemos escrever E como um produto de somas:

$$E = \overline{(\overline{A} + \overline{B} + C)} \cdot \overline{(\overline{A} + \overline{C} + B)} \cdot \overline{(\overline{B} + C + A)}$$

Para derivar esse formato, você precisa usar os *teoremas de DeMorgan*, discutidos nos exercícios.

Neste texto, usamos o formato da soma de produtos. É fácil ver que qualquer função lógica pode ser representada como uma soma de produtos, construindo tal representação a partir da tabela verdade para a função. Cada entrada da tabela verdade para a qual a função é verdadeira, corresponde a um termo do produto. O termo do produto consiste em um produto lógico de todas as entradas ou os complementos das entradas, dependendo se a entrada na tabela verdade possui um 0 ou 1 correspondente a essa variável. A função lógica é a soma lógica dos

termos do produto onde a função é verdadeira. Isso pode ser visto mais facilmente com um exemplo.

Soma de produtos

Exemplo

Mostre a representação da soma dos produtos para a seguinte tabela verdade para D .

Entradas		Saídas	
A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Resposta

Existem quatro termos no produto, pois a função é verdadeira (1) para quatro combinações de entrada diferentes. São estes

$$\begin{aligned} & \bar{A} \cdot \bar{B} \cdot C \\ & \bar{A} \cdot B \cdot C \\ & A \cdot \bar{B} \cdot \bar{C} \\ & A \cdot B \cdot C \end{aligned}$$

Assim, podemos escrever a função para D como a soma destes termos:

$$D = (\bar{A} \cdot \bar{B} \cdot C)(\bar{A} \cdot B \cdot \bar{C})(A \cdot \bar{B} \cdot \bar{C})(A \cdot B \cdot C)$$

Observe que somente as entradas da tabela verdade para as quais a função é verdadeira geram os termos na equação.

Podemos usar esse relacionamento entre uma tabela verdade e uma representação bidimensional para gerar uma implementação no nível de portas lógicas de qualquer conjunto de funções lógicas. Um conjunto de funções lógicas corresponde a uma tabela verdade com várias colunas de saída, como vimos no exemplo da página B.5. Cada coluna de saída representa uma função lógica diferente, que pode ser construída diretamente a partir da tabela verdade.

A representação da soma dos produtos corresponde a uma implementação lógica estruturada comum, chamada **array lógico programável (PLA – Programmable Logic Array)**. Uma PLA possui um conjunto de entradas e complementos de entrada correspondentes (que podem ser implementados com um conjunto de inversores) e dois estágios de lógica. O primeiro estágio é um array de portas AND que formam um conjunto de **termos do produto** (às vezes chamados **mintermos**); cada termo do produto pode consistir em qualquer uma das entradas ou seus complementos. O segundo estágio é um array de portas OR, cada uma das quais formando uma soma lógica de qualquer quantidade de termos do produto. A [Figura B.3.3](#) mostra a forma básica de uma PLA.

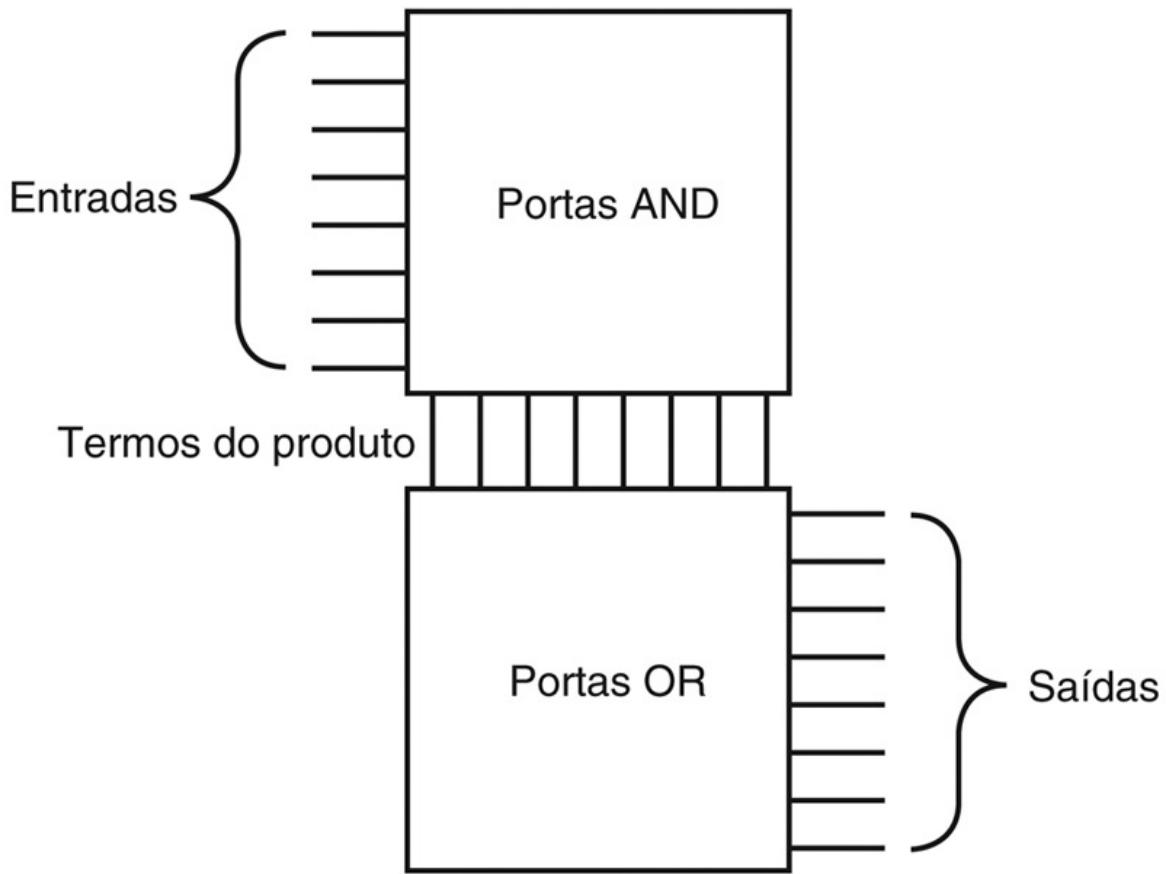


FIGURA B.3.3 O formato básico de uma PLA consiste em um array de portas AND seguido por um array de portas OR.

Cada entrada no array de portas AND é um termo do produto consistindo em qualquer quantidade de entradas ou entradas invertidas. Cada entrada no array de portas OR é um termo da soma consistindo em qualquer quantidade desses termos do produto.

array lógico programável (PLA)

Um elemento lógico estruturado composto de um conjunto de entradas e complementos de entrada correspondentes e dois estágios de lógica: o primeiro gerando termos do produto das entradas e complementos da entrada e o segundo gerando termos da soma dos termos do produto. Logo, PLAs implementam funções lógicas como uma soma de produtos.

mintermos

Também chamados **termos do produto**. Um conjunto de entradas lógicas

unidas por conjunção (operações AND); os termos do produto formam o primeiro estágio lógico do *array lógico programável* (PLA).

Uma PLA pode implementar diretamente a tabela verdade de um conjunto de funções lógicas com várias entradas e saídas. Como cada entrada onde a tabela verdade é verdadeira exige um termo do produto, haverá uma linha correspondente na PLA. Cada saída corresponde a uma linha em potencial das portas OR no segundo estágio. O número de portas OR corresponde ao número de entradas da tabela verdade para as quais a saída é verdadeira. O tamanho total de uma PLA, como aquela mostrada na [Figura B.3.3](#), é igual à soma do tamanho do array de portas AND (chamado *plano AND*) e o tamanho do array de portas OR (chamado *plano OR*). Examinando a [Figura B.3.3](#), podemos ver que o tamanho do array de portas AND é igual ao número de entradas vezes o número de termos do produto diferentes, e o tamanho do array de portas OR é o número de saídas vezes o número de termos do produto.

Uma PLA possui duas características que a ajudam a se tornar um meio eficiente de implementar um conjunto de funções lógicas. Primeiro, somente as entradas da tabela verdade que produzem um valor verdadeiro para pelo menos uma saída possuem quaisquer portas lógicas associadas a elas. Segundo, cada termo do produto diferente terá apenas uma entrada na PLA, mesmo que o termo do produto seja usado em várias saídas. Vamos examinar um exemplo.

PLAs

Exemplo

Considere o conjunto de funções lógicas definido no exemplo da página B-5. Mostre uma implementação em PLA desse exemplo para D , E e F .

Resposta

Aqui está a tabela verdade construída anteriormente:

Entradas			Saídas		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0

0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Como existem sete termos do produto exclusivos com pelo menos um valor verdadeiro na seção de saída, haverá sete colunas no plano AND. O número de linhas no plano AND é três (pois existem três entradas) e também haverá três linhas no plano OR (pois existem três saídas). A Figura B.3.4 mostra a PLA resultante, com os termos do produto correspondendo às entradas da tabela verdade, de cima para baixo.

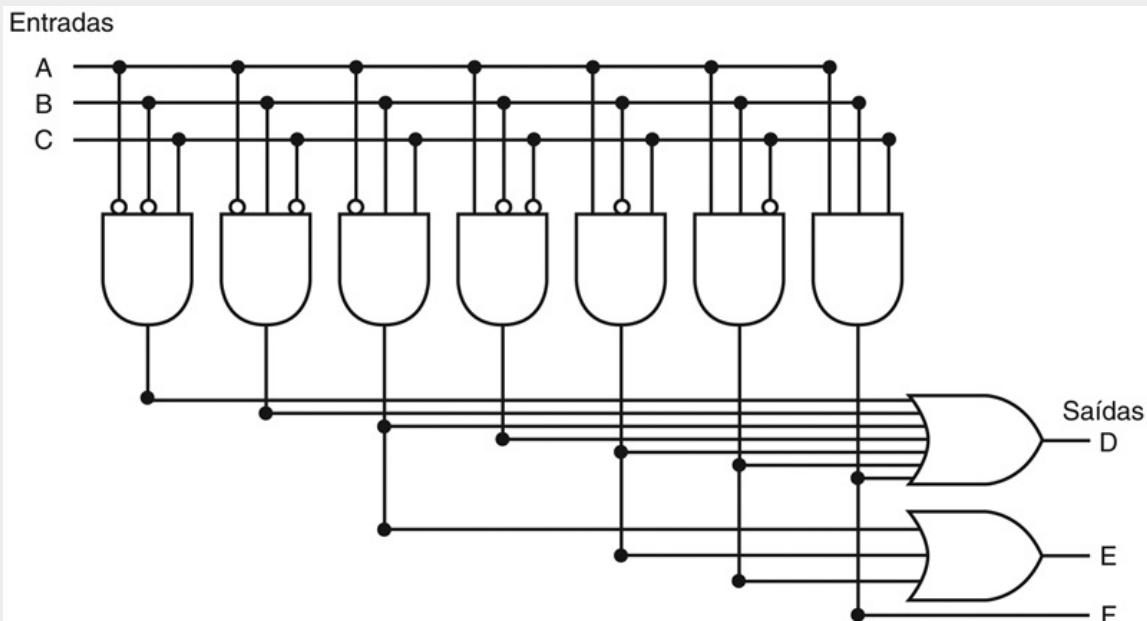


FIGURA B.3.4 A PLA para implementar a função lógica descrita anteriormente.

Em vez de desenhar todas as portas, como fizemos na [Figura B.3.4](#), os projetistas normalmente mostram apenas a posição das portas AND ou das portas OR. Os pontos são usados na interseção de uma linha de sinal do termo do produto e uma linha de entrada ou uma linha de saída quando uma porta AND ou OR correspondente é exigida. A [Figura B.3.5](#) mostra como a PLA da [Figura B.3.4](#) ficaria quando desenhada dessa maneira. O conteúdo de uma PLA é fixo

quando a PLA é criada, embora também existam formas de estruturas tipo PLA, chamadas *PALs*, que podem ser programadas eletronicamente quando um projetista está pronto para usá-las.

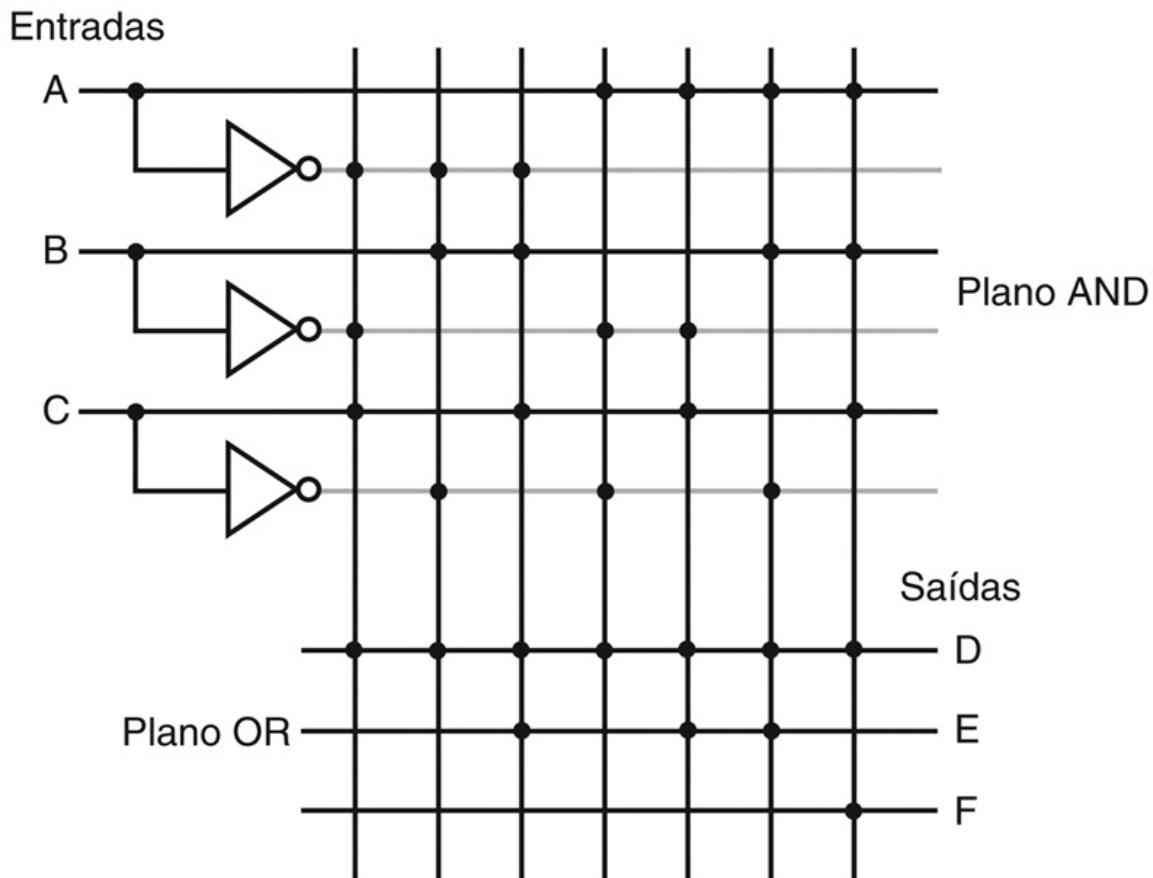


FIGURA B.3.5 Uma PLA desenhada usando pontos para indicar os componentes dos termos do produto e os termos da soma no array.

Em vez de usar inversores nas portas lógicas, normalmente todas as entradas percorrem a largura do plano AND nas formas original e complemento. Um ponto no plano AND indica que a entrada, ou seu inverso, ocorre no termo do produto. Um ponto no plano OR indica que o termo do produto correspondente aparece na saída correspondente.

ROMs

Outra forma de lógica estruturada que pode ser usada para implementar um

conjunto de funções lógicas é uma **memória somente de leitura (ROM – Read-Only Memory)**. Uma ROM é chamada de memória porque possui um conjunto de locais que podem ser lidos; porém, o conteúdo desses locais é fixo, normalmente no momento em que a ROM é fabricada. Há também **ROMs programáveis (PROMs – Programmable ROMs)**, que podem ser programadas eletronicamente, quando um projetista conhece seu conteúdo. Há também PROMs apagáveis; esses dispositivos exigem um processo de apagamento lento usando luz ultravioleta, e assim são usados como memórias somente de leitura, exceto durante o processo de projeto e depuração.

memória somente de leitura (ROM)

Uma memória cujo conteúdo é definido no momento da criação, após o qual o conteúdo só pode ser lido. A ROM é usada como lógica estruturada para implementar um conjunto de funções lógicas usando os termos das funções lógicas como entradas de endereço e as saídas como bits em cada word da memória.

ROM programável (PROM)

Uma forma de memória somente de leitura que pode ser programada quando um projetista conhece seu conteúdo.

Uma ROM possui um conjunto de linhas de endereço de entrada e um conjunto de saídas. O número de entradas endereçáveis na ROM determina o número de linhas de endereço: se a ROM contém 2^m entradas endereçáveis, chamadas de *altura*, então existem m linhas de entrada. O número de bits em cada entrada endereçável é igual ao número de bits de saída e, às vezes, é chamado de *largura* da ROM. O número total de bits na ROM é igual à altura vezes a largura. A altura e a largura, às vezes, são chamadas coletivamente como o *formato* da ROM.

Uma ROM pode codificar uma coleção de funções lógicas diretamente a partir da tabela verdade. Por exemplo, se houver n funções com m entradas, precisamos de uma ROM com m linhas de endereço (e 2^m entradas), com cada entrada sendo de n bits de largura. O conteúdo na parte de entrada da tabela verdade representa os endereços do conteúdo na ROM, enquanto o conteúdo da parte de saída da tabela verdade constitui o conteúdo da ROM. Se a tabela verdade for organizada de modo que a sequência de entradas na parte da entrada

constitua uma sequência de números binários (como todas as tabelas verdade mostradas até aqui), então a parte de saída também indica o conteúdo da ROM em ordem. No exemplo anterior, havia três entradas e três saídas. Isso equivale a uma ROM com $2^3 = 8$ entradas, cada uma com 3 bits de largura. O conteúdo dessas entradas em ordem crescente, por endereço, é dado diretamente pela parte de saída da tabela verdade que aparece no exemplo anterior.

ROMs e PLAs estão bastante relacionadas. Uma ROM é totalmente decodificada: ela contém uma word de saída completa para cada combinação de entrada possível. Uma PLA é decodificada apenas parcialmente. Isso significa que uma ROM sempre terá mais entradas. Para a tabela verdade anterior, na página B-14, a ROM contém itens para todas as oito entradas possíveis, enquanto a PLA contém apenas os sete termos de produto ativos. À medida que o número de entradas cresce, o número de entradas na ROM cresce exponencialmente. Ao contrário, para a maioria das funções lógicas reais, o número de termos de produto cresce muito mais lentamente. Essa diferença torna as PLAs geralmente mais eficientes para implementar as funções lógicas combinacionais. As ROMs possuem a vantagem de serem capazes de implementar qualquer função lógica com o seu número de entradas e saídas. Essa vantagem facilita mudar o conteúdo da ROM se a função lógica mudar, pois o tamanho da ROM não precisa mudar.

Além de ROMs e PLAs, os sistemas de síntese de lógica modernos também traduzirão pequenos blocos de lógica combinacional em uma coleção de portas que podem ser colocadas e ligadas automaticamente. Embora algumas pequenas coleções de portas não façam uso eficiente da área, para pequenas funções lógicas elas possuem menos overhead do que a estrutura rígida de uma ROM ou PLA e, por isso, são preferidas.

Para projetar a lógica fora de um circuito integrado personalizado ou semipersonalizado, uma opção comum é um dispositivo programável em campo; descrevemos esses dispositivos na [Seção B.12](#).

Don't Cares

Normalmente, na implementação de alguma lógica combinacional, existem situações em que não nos importamos com o valor de alguma saída, seja porque outra saída é verdadeira ou porque um subconjunto de combinações de entrada determina os valores das saídas. Essas situações são conhecidas como *don't cares*. Don't cares são importantes porque facilitam a otimização da

implementação de uma função lógica.

Existem dois tipos de don't cares: don't cares de saída e don't cares de entrada, ambos podendo ser representados em uma tabela verdade. Os *don't cares de saída* surgem quando não nos importamos com o valor de uma saída para alguma combinação de entrada. Eles aparecem como X na parte de saída de uma tabela verdade. Quando uma saída é um don't care para alguma combinação de entrada, o projetista ou o programa de otimização da lógica é livre para tornar a saída verdadeira ou falsa para essa combinação de entrada. *Don't cares de entrada* surgem quando uma saída depende apenas de algumas das entradas, e também são mostradas como X, embora na parte de entrada da tabela verdade.

Don't Cares

Exemplo

Considere uma função lógica com entradas A , B e C definidas da seguinte maneira:

- Se A ou C é verdadeira, então a saída D é verdadeira, qualquer que seja o valor de B .
- Se A ou B é verdadeira, então a saída E é verdadeira, qualquer que seja o valor de B .
- A saída F é verdadeira se exatamente uma das entradas for verdadeira, embora não nos importemos com o valor de F , sempre que D e E são verdadeiras.

Mostre a tabela verdade completa para essa função e a tabela verdade usando don't cares. Quantos termos do produto são exigidos em uma PLA para cada uma delas?

Resposta

Aqui está a tabela verdade completa, sem os don't cares:

Entradas			Saídas		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	0

1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	0

Isso exige sete termos do produto sem otimização. A tabela verdade escrita com don't cares de saída se parece com esta:

Entradas			Saídas		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	X
1	0	0	1	1	X
1	0	1	1	1	X
1	1	0	1	1	X
1	1	1	1	1	X

Se também usarmos os don't cares de entrada, essa tabela verdade pode ser simplificada ainda mais, para mostrar:

Entradas			Saídas		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
X	1	1	1	1	X
1	X	X	1	1	X

Essa tabela verdade simplificada exige uma PLA com quatro mintermos ou pode ser implementada em portas discretas com uma porta AND de duas entradas e três portas OR (duas com três entradas e uma com duas entradas). Isso confronta a tabela verdade original, que tinha sete mintermos e exigiria quatro portas AND.

A minimização lógica é crítica para conseguir implementações eficientes. Uma ferramenta útil para a minimização manual da lógica aleatória são os

mapas de Karnaugh. Os mapas de Karnaugh representam a tabela verdade graficamente, de modo que os termos do produto que podem ser combinados são facilmente vistos. Apesar disso, a otimização manual das funções lógicas significativas usando os mapas de Karnaugh não é prática, tanto devido ao tamanho dos mapas quanto pela sua complexidade. Felizmente, o processo de minimização lógica é muito mecânico e pode ser realizado por ferramentas de projeto. No processo de minimização, as ferramentas tiram proveito dos don't cares, de modo que sua especificação é importante. As referências do livro-texto ao final deste apêndice oferecem uma discussão mais profunda sobre minimização lógica, mapas de Karnaugh e a teoria por trás de tais algoritmos de minimização.

Arrays de elementos lógicos

Muitas das operações combinacionais a serem realizadas sobre os dados precisam ser feitas em uma word inteira (32 bits) de dados. Assim, normalmente queremos montar um array de elementos lógicos, que podemos representar apenas mostrando que determinada operação acontecerá a uma coleção inteira de entradas. Dentro de uma máquina, quase sempre queremos selecionar entre um par de *barramentos*. Um **barramento** é uma coleção de linhas de dados tratada em conjunto como um único sinal lógico. (O termo *barramento* também é usado para indicar uma coleção compartilhada de linhas com várias fontes e usos.)

barramento

No projeto lógico, uma coleção de linhas de dados que é tratada em conjunto como um único sinal lógico; também é uma coleção compartilhada de linhas com várias fontes e usos.

Por exemplo, no conjunto de instruções MIPS, o resultado de uma instrução escrita em um registrador pode vir de uma dentre duas origens. Um multiplexador é usado para escolher qual dos dois barramentos (cada um com 32 bits de largura) será escrito no registrador Result. O multiplexador de 1 bit, que mostramos anteriormente, precisará ser replicado 32 vezes.

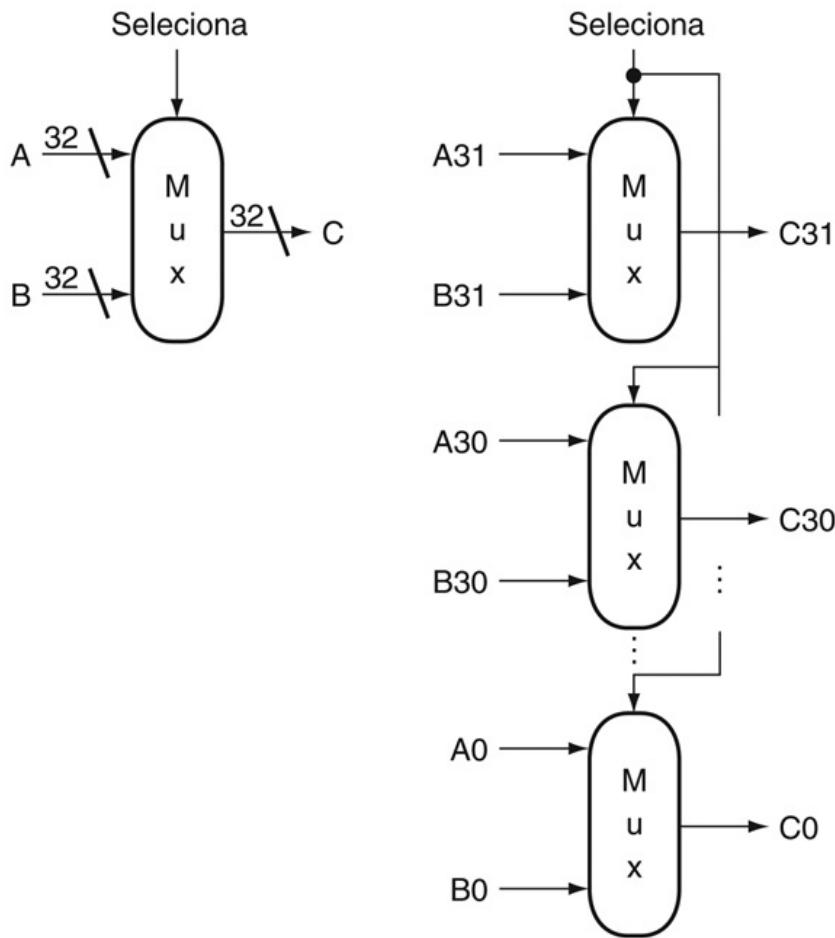
Indicamos que um sinal é um barramento em vez de uma única linha de 1 bit representando-o com uma linha mais grossa em uma figura. A maioria dos barramentos possui 32 bits de largura; os que não possuem são rotulados explicitamente com sua largura. Quando mostramos uma unidade lógica cujas

entradas e saídas são barramentos, isso significa que a unidade precisa ser replicada por um número de vezes suficiente para acomodar a largura da entrada. A Figura B.3.6 mostra como desenhamos um multiplexador que seleciona entre um par de barramentos de 32 bits e como isso se expande em termos de multiplexadores de 1 bit de largura. Às vezes, precisamos construir um array de elementos lógicos onde as entradas para alguns elementos no array são saídas de elementos anteriores. Por exemplo, é assim que é construída uma ALU de múltiplos bits de largura. Nesses casos, temos de mostrar explicitamente como criar arrays mais largos, pois os elementos individuais do array não são mais independentes, como acontece no caso de um multiplexador de 32 bits de largura.

Verifique você mesmo

A paridade é uma função na qual a saída depende do número de 1s na entrada. Para uma função de paridade par, a saída é 1 se a entrada tiver um número par de uns. Suponha que uma ROM seja usada para implementar uma função de paridade par com uma entrada de 4 bits. Qual dentre A, B, C ou D representa o conteúdo da ROM?

Endereço	A	B	C	D
0	0	1	0	1
1	0	1	1	0
2	0	1	0	1
3	0	1	1	0
4	0	1	0	1
5	0	1	1	0
6	0	1	0	1
7	0	1	1	0
8	1	0	0	1
9	1	0	1	0
10	1	0	0	1
11	1	0	1	0
12	1	0	0	1
13	1	0	1	0
14	1	0	0	1
15	1	0	1	0



- a. Um multiplexador de 2 para 1 com 32 bits de largura

- b. O multiplexador de 32 bits é, na realidade, um array de 32 multiplexadores de 1 bit

FIGURA B.3.6 Um multiplexador é duplicado 32 vezes para realizar uma seleção entre duas entradas de 32 bits.

Observe que ainda existe apenas um sinal de seleção de dados usado para todos os 32 multiplexadores de 1 bit.

B.4. Usando uma linguagem de descrição de hardware

Hoje, a maior parte do projeto digital dos processadores e sistemas de hardware relacionados é feita por meio de uma **linguagem de descrição de hardware**. Essa linguagem tem duas finalidades. Primeiro, ela oferece uma descrição abstrata do hardware para simular e depurar o projeto. Segundo, com o uso da síntese lógica e ferramentas de compilação de hardware, essa descrição pode ser compilada para a implementação do hardware.

linguagem de descrição de hardware

Uma linguagem de programação para descrever o hardware utilizado para gerar simulações de um projeto de hardware e também como entrada para ferramentas de síntese que podem gerar hardware real.

Nesta seção, introduzimos a linguagem de descrição de hardware Verilog e mostramos como ela pode ser usada para o projeto combinacional. No restante do apêndice, expandimos o uso da Verilog para incluir o projeto da lógica sequencial.. A Verilog do sistema acrescenta estruturas e alguns recursos úteis a ela.

Verilog é uma das duas principais linguagens de descrição de hardware; a outra é **VHDL**. A primeira é um pouco mais utilizada no setor e é baseada em C, ao contrário de VHDL, que é baseada em Ada. O leitor um pouco mais familiarizado com C achará mais fácil acompanhar os fundamentos da Verilog, que utilizamos neste apêndice. Os leitores já acostumados com VHDL deverão achar os conceitos simples, desde que já conheçam um pouco da sintaxe da linguagem C.

Verilog

Uma das duas linguagens de descrição de hardware mais comuns.

VHDL

Uma das duas linguagens de descrição de hardware mais comuns.

Verilog pode especificar uma definição comportamental e uma estrutural de um sistema digital. Uma **especificação comportamental** descreve como um sistema digital opera funcionalmente. Uma **especificação estrutural** descreve a organização detalhada de um sistema digital normalmente utilizando uma descrição hierárquica. Uma especificação estrutural pode ser usada para descrever um sistema de hardware em termos de uma hierarquia de elementos básicos, como portas lógicas e chaves. Assim, poderíamos usar a Verilog para descrever o conteúdo exato das tabelas verdade e o caminho de dados da seção anterior.

especificação comportamental

Descreve como um sistema digital opera funcionalmente.

especificação estrutural

Descreve como um sistema digital é organizado em termos de uma conexão hierárquica de elementos.

Com o surgimento das ferramentas de **síntese de hardware**, a maioria dos projetistas agora utiliza Verilog ou VHDL para descrever estruturalmente apenas o caminho de dados, contando com a síntese lógica para gerar o controle, a partir da descrição comportamental. Além disso, a maioria dos sistemas de CAD oferece grandes bibliotecas de partes padronizadas, como ALUs, multiplexadores, bancos de registradores, memórias, blocos lógicos programáveis, além de portas básicas.

ferramentas de síntese de hardware

Software de projeto auxiliado por computador que pode gerar um projeto no nível de portas lógicas, baseado em descrições comportamentais de um sistema digital.

A obtenção de um resultado aceitável usando bibliotecas e síntese de lógica exige que a especificação seja escrita vigiando a síntese eventual e o resultado desejado. Para nossos projetos simples, isso significa principalmente deixar claro o que esperamos que seja implementado na lógica combinacional e o que esperamos exigir da lógica sequencial. Na maior parte dos exemplos que usamos

nesta seção, e no restante deste apêndice, escrevemos em Verilog visando à síntese eventual.

Tipos de dados e operadores em Verilog

Existem dois tipos de dados principais em Verilog:

1. Um **wire** especifica um sinal combinacional.
2. Um **reg** (registrador) mantém um valor, que pode variar com o tempo. Um reg não precisa corresponder necessariamente a um registrador real em uma implementação, embora isso normalmente aconteça.

wire

Em Verilog, especifica um sinal combinacional.

reg

Em Verilog, um registrador.

Um registrador ou wire, chamado X, que possui 32 bits de largura, é declarado como um array: `reg [31:0] X` ou `wire [31:0] X`, que também define o índice de 0 para designar o bit menos significativo do registrador. Como normalmente queremos acessar um subcampo de um registrador ou wire, podemos nos referir ao conjunto contíguo de bits de um registrador ou wire com a notação `[bit inicial: bit final]`, onde os dois índices devem ser valores constantes.

Um array de registradores é usado para uma estrutura como um banco de registradores ou memória. Assim, a declaração

```
reg [31:0] registerfile[0:31]
```

especifica uma variável registerfile que é equivalente a um banco de registradores MIPS, onde o registrador 0 é o primeiro. Ao acessar um array, podemos nos referir a um único elemento, como em C, usando a notação `registerfile[numreg]`.

Os valores possíveis para um registrador ou wire em Verilog são

- 0 ou 1, representando o falso ou verdadeiro lógico

- X, representando desconhecido, o valor inicial dado a todos os registradores e a qualquer wire não conectado a algo
- Z, representando o estado de impedância alta para portas tristate, que não discutiremos neste apêndice

Valores constantes podem ser especificados como números decimais e também como binário, octal ou hexadecimal. Normalmente, queremos dizer o tamanho de um campo constante em bits. Isso é feito prefixando o valor com um número decimal que especifica seu tamanho em bits. Por exemplo:

- 4'b0100 especifica uma constante binária de 4 bits com o valor 4, assim como 4'd4
- -8'h4 especifica uma constante de 8 bits com o valor -4 (na representação, complemento a dois)

Os valores também podem ser concatenados colocando-os dentro de {} separados por vírgulas. A notação {x {bit field}} replica bit field x vezes. Por exemplo:

- {16{2'b01}} cria um valor de 32 bits com o padrão 0101 ... 01.
- {A[31:16], B[15:0]} cria um valor cujos 16 bits mais significativos vêm de A e cujos 16 bits menos significativos vêm de B.

Verilog oferece o conjunto completo de operadores unários e binários de C, incluindo os operadores aritméticos (+, -, *, /), os operadores lógicos (&, |, ~), os operadores de comparação (==, !=, >, <, <=, >=), os operadores de deslocamento (<<, >>) e o operador condicional de C (? , que é usado na forma condição ? expr1 :expr2 e retorna expr1 se a condição for verdadeira e expr2 se ela for falsa). Verilog acrescenta um conjunto de operadores unários de redução lógica (&, |, ^) que geram um único bit aplicando o operador lógico a todos os bits de um operando. Por exemplo, &A retorna o valor obtido pelo AND de todos os bits de A, e ^A retorna a redução obtida pelo uso do OR exclusivo em todos os bits de A.

Verifique você mesmo

Quais dos seguintes itens definem exatamente o mesmo valor?

1. 8'bimoooo
2. 8'hF0
3. 8'd240
4. {{4{1'b1}}, {4{1'b0}}} }
5. {4'b1, 4'b0}

Estrutura de um programa em Verilog

Um programa em Verilog é estruturado como um conjunto de módulos, que podem representar qualquer coisa desde uma coleção de portas lógicas até um sistema completo. Os módulos são semelhantes às classes em C++, embora não tão poderosas. Um módulo especifica suas portas de entrada e saída, que descrevem as conexões de entrada e saída de um módulo. Um módulo também pode declarar variáveis adicionais. O corpo de um módulo consiste em:

- Construções `initial`, que podem inicializar variáveis `reg`
- Atribuições contínuas, que definem apenas lógica combinacional
- Construções `always`, que podem definir a lógica sequencial ou combinacional
- Instâncias de outros módulos, usadas para implementar o módulo sendo definido

Representando lógica combinacional complexa em Verilog

Uma atribuição contínua, indicada com a palavra-chave `assign`, atua como uma função lógica combinacional: a saída é atribuída continuamente ao valor e uma mudança nos valores de entrada é refletida imediatamente no valor da saída. Os wires só podem receber valores com atribuições contínuas. Usando a atribuição contínua, podemos definir um módulo que implementa um meio-somador, como mostra a [Figura B.4.1](#).

```

module half_adder (A,B,Sum,Carry);
    input A,B; //duas entradas de 1 bit
    output Sum, Carry; //duas saídas de 1 bit
    assign Sum = A ^ B; //sum é A xor B
    assign Carry = A & B; //Carry é A e B
endmodule

```

FIGURA B.4.1 Um módulo em Verilog que define um meio-somador usando atribuições contínuas.

As instruções de atribuição são um modo garantido de escrever Verilog, que gera lógica combinacional. Entretanto, para estruturas mais complexas, as instruções de atribuição podem ser esquisitas ou tediosas de usar. Também é possível usar o bloco `always` de um módulo para descrever um elemento lógico combinacional, embora com muito cuidado. O uso de um bloco `always` permite a inclusão de construções de controle da Verilog, como *if-then-else*, instruções *case*, instruções *for* e instruções *repeat*. Essas instruções são semelhantes às que existem em C, com pequenas mudanças.

Um bloco `always` especifica uma lista opcional de sinais aos quais o bloco é sensitivo (em uma lista começando com `@`). O bloco `always` é reavaliado se qualquer um dos sinais listados mudar de valor; se a lista for omitida, o bloco `always` é constantemente reavaliado. Quando um bloco `always` está especificando a lógica combinacional, a **lista de sensitividade** deverá incluir todos os sinais de entrada. Se houver várias instruções Verilog a serem executadas em um bloco `always`, elas estão cercadas pelas palavras-chave `begin` e `end`, que tomam o lugar de `{ }` em C. Um bloco `always`, portanto, se parece com

```

always @(lista de sinais que causam reavaliação) begin
    Instruções Verilog incluindo atribuições e outras
    instruções de controle end

```

lista de sensitividade

A lista de sinais que especifica quando um bloco `always` deve ser reavaliado.

Variáveis reg só podem ser atribuídas dentro de um bloco always, usando uma instrução de atribuição procedural (distinguida da atribuição contínua vista anteriormente). Contudo, existem dois tipos diferentes de atribuições procedurais. O operador de atribuição = é executado como em C; o lado direito é avaliado e o lado esquerdo recebe o valor. Além do mais, ele é executado como uma instrução de atribuição C normal: ou seja, é completado antes que a próxima instrução seja executada. Logo, o operador de atribuição = tem o nome **atribuição bloqueante**. Esse bloqueio pode ser útil na geração da lógica sequencial, e voltaremos a esse assunto em breve. A outra forma de atribuição (**não bloqueante**) é indicada por <=. Na atribuição não bloqueante, todo o lado direito das atribuições em um grupo always é avaliado, e as atribuições são feitas simultaneamente. Como um primeiro exemplo da lógica combinacional implementada usando um bloco always, a [Figura B.4.2](#) mostra a implementação de um multiplexador 4-para-1, que usa uma construção case para facilitar a escrita. A construção case se parece com uma instrução switch da linguagem C. A [Figura B.4.3](#) mostra uma definição de uma ALU MIPS, que também usa uma instrução case.

```
module Mult4to1 (In1,In2,In3,In4,Sel,Out);
    input [31:0] In1, In2, In3, In4; /four 32-bit inputs
    input [1:0] Sel; //selector signal
    output reg [31:0] Out;// 32-bit output
    always @(In1, In2, In3, In4, Sel)
        case (Sel) //a 4->1 multiplexor
            0: Out <= In1;
            1: Out <= In2;
            2: Out <= In3;
            default: Out <= In4;
        endcase
    endmodule
```

FIGURA B.4.2 Uma definição Verilog de um multiplexador 4-para-1 com entradas de 32 bits, usando uma instrução case.

Uma instrução case atua como uma instrução switch da linguagem C, exceto que, em Verilog, somente o código associado ao case selecionado é executado (como se cada estado de case tivesse um break no final) e não existe passagem direta para a instrução seguinte.

```

module MIPSALU (ALUctl, A, B, ALUOut, Zero);
    input [3:0] ALUctl;
    input [31:0] A,B;
    output reg [31:0] ALUOut;
    output Zero;
    assign Zero = (ALUOut==0); //Zero é verdadeiro se ALUOut
                           // for 0; vai a qualquer lugar
    always @(ALUctl, A, B) //reavalia se estes mudarem
        case (ALUctl)
            0: ALUOut <= A & B;
            1: ALUOut <= A | B;
            2: ALUOut <= A + B;
            6: ALUOut <= A - B;
            7: ALUOut <= A < B ? 1:0;
            12: ALUOut <= ~(A | B); // resultado é nor
        default: ALUOut <= 0; //default 0, não deve acontecer;
        endcase
    endmodule

```

FIGURA B.4.3 Uma definição comportamental em Verilog de uma ALU MIPS.

Isso poderia ser sintetizado por meio de uma biblioteca de módulos contendo operações aritméticas e lógicas básicas.

atribuição bloqueante

Em Verilog, uma atribuição que completa antes da execução da próxima instrução.

atribuição não bloqueante

Uma atribuição que continua após a avaliação do lado direito, atribuindo o valor ao lado esquerdo somente depois que todo o lado direito for avaliado.

Como apenas variáveis reg podem ser atribuídas dentro de blocos always, quando queremos descrever a lógica combinacional usando um bloco always,

devemos ter o cuidado de garantir que o reg não seja sintetizado como um registrador. Diversas armadilhas são descritas na Seção “Detalhamento” a seguir.

Detalhamento

Instruções de atribuição contínua sempre geram lógica combinacional, mas outras estruturas Verilog, mesmo quando em blocos always, podem gerar resultados inesperados durante a síntese lógica. O problema mais comum é a criação de lógica sequencial implicando a existência de um latch ou registrador, o que resulta em uma implementação mais lenta e mais dispendiosa do que talvez pretendido. Para garantir que a lógica que deverá ser combinacional seja sintetizada dessa maneira, faça o seguinte:

1. Coloque toda a lógica combinacional em uma atribuição contínua ou em um bloco always.
2. Verifique se todos os sinais usados como entradas aparecem na lista de sensitividade de um bloco always.
3. Garanta que cada caminho dentro de um bloco always atribui um valor ao mesmo conjunto exato de bits.

O último deles é o mais fácil de se deixar de lado; examine o exemplo da Figura B.5.15 para convencer-se de que essa propriedade foi respeitada.

Verifique você mesmo

Supondo que todos os valores sejam inicialmente zero, quais são os valores de A e B depois de executar este código Verilog dentro de um bloco always?

```
C=1;  
A <= C;  
B = C;
```

B.5. Construindo uma unidade lógica e aritmética

ALU n. [Arthritic Logic Unit ou (raro) Arithmetic Logic Unit] Um gerador de números aleatórios fornecido por padrão com todos os sistemas computacionais.

Stan Kelly-Bootle, The Devil's DP Dictionary, 1981

A **unidade lógica e aritmética (ALU – Arithmetic Logic Unit)** é o músculo do computador, o dispositivo que realiza as operações aritméticas, como adição e subtração, ou as operações lógicas, como AND e OR. Esta seção constrói uma ALU a partir de quatro blocos de montagem do hardware (portas AND e OR, inversores e multiplexadores) e ilustra como funciona a lógica combinacional. Na próxima seção, veremos como a adição pode ser agilizada por meio de projetos mais inteligentes.

Como a word do MIP tem 32 bits de largura, precisamos de uma ALU de 32 bits. Vamos supor que iremos conectar 32 ALUs de 1 bit para criar a ALU desejada. Portanto, vamos começar construindo uma ALU de 1 bit.

Uma ALU de 1 bit

As operações lógicas são as mais fáceis, pois são mapeadas diretamente nos componentes de hardware da [Figura B.2.1](#).

A unidade lógica de 1 bit para AND e OR se parece com a [Figura B.5.1](#). O multiplexador à direita, então, seleciona a AND b ou a OR b , dependendo se o valor de *Operação* é 0 ou 1. A linha que controla o multiplexador aparece em destaque para distingui-la das linhas com dados. Observe que renomeamos as linhas de controle e a saída do multiplexador para lhes dar nomes que refletem a função da ALU.

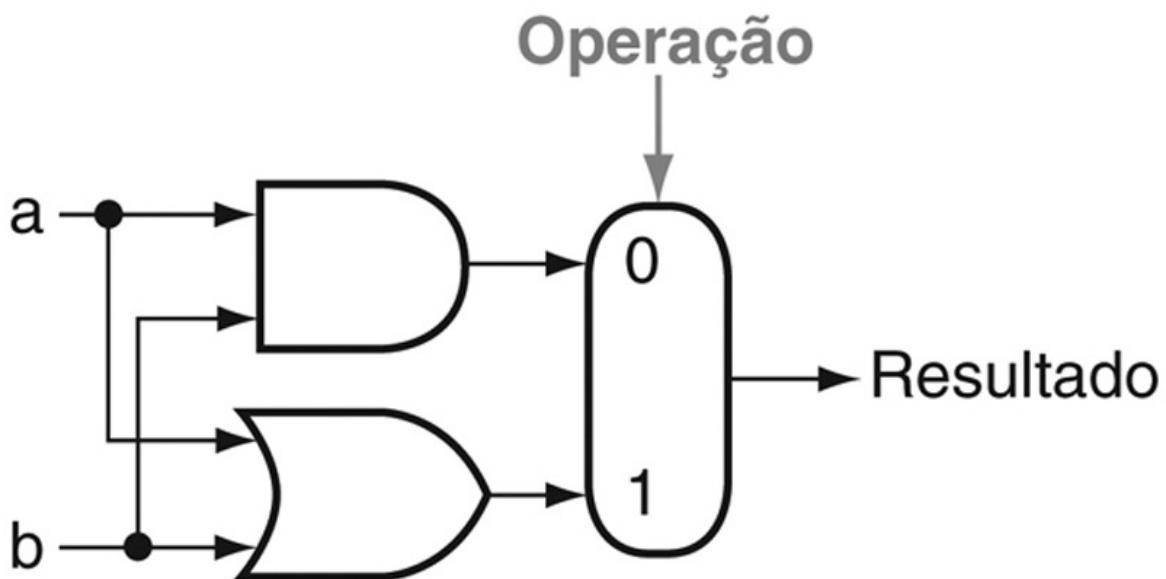


FIGURA B.5.1 A unidade lógica de 1 bit para AND e OR.

A próxima função a incluir é a adição. Um somador precisa ter duas entradas para os operandos e uma saída de único bit para a soma. É preciso haver uma segunda saída para o carry, chamada *CarryOut*. Como o *CarryOut* do somador vizinho precisa ser incluído como uma entrada, precisamos de uma terceira entrada. Essa entrada é chamada *CarryIn*. A [Figura B.5.2](#) mostra as entradas e as saídas de um somador de 1 bit. Como sabemos o que a adição precisa fazer, podemos especificar as saídas dessa “caixa preta” com base em suas entradas, como a [Figura B.5.3](#) demonstra.

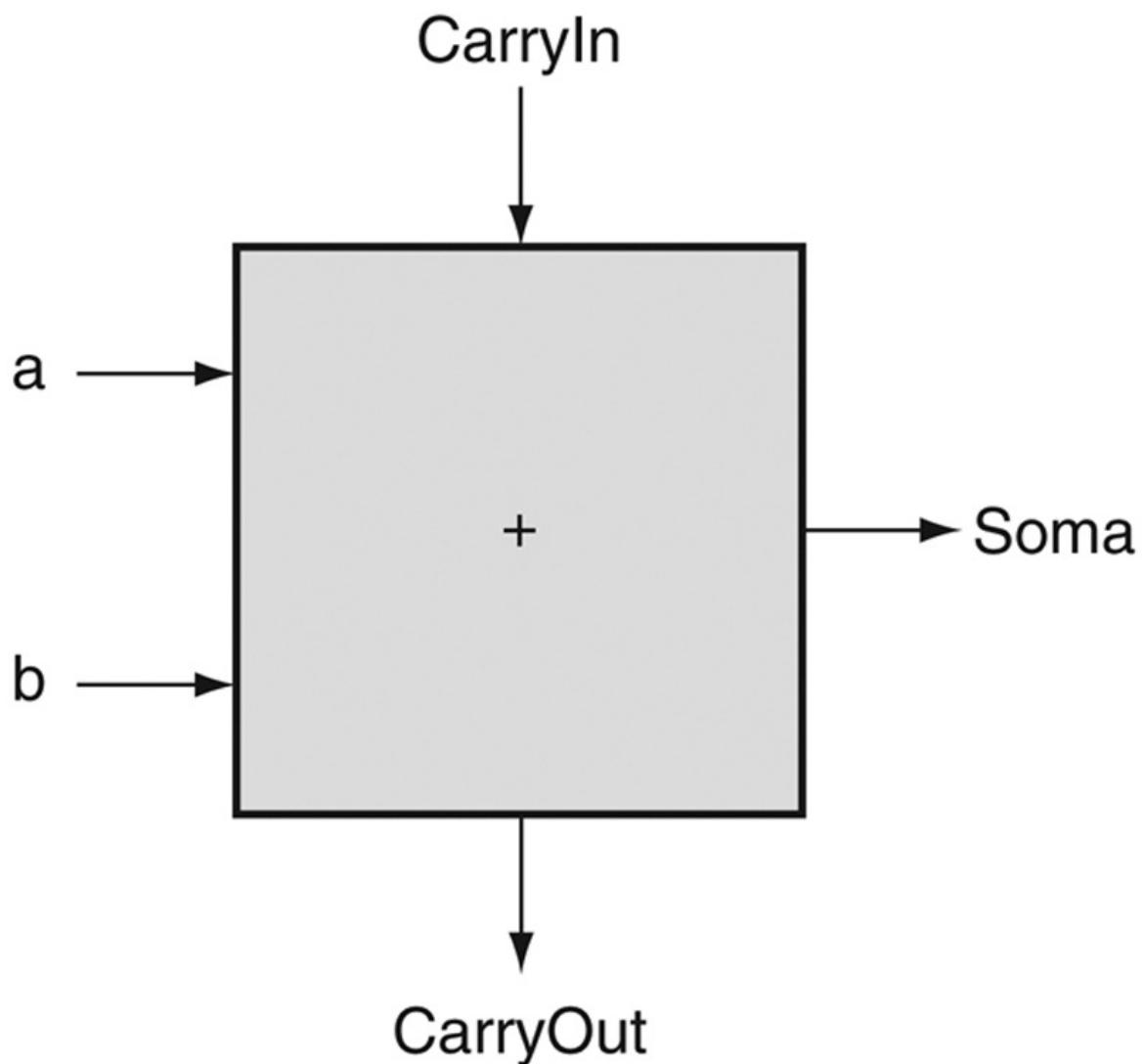


FIGURA B.5.2 Um somador de 1 bit.

Esse somador é chamado de somador completo; ele também é chamado somador (3,2), pois tem 3 entradas e 2 saídas. Um somador com apenas as entradas a e b é chamado somador (2,2) ou meio somador.

Entradas			Saídas		Comentários
a	b	CarryIn	CarryOut	Soma	
0	0	0	0	0	$0 + 0 + 0 = 00_{\text{bin}}$
0	0	1	0	1	$0 + 0 + 1 = 01_{\text{bin}}$
0	1	0	0	1	$0 + 1 + 0 = 01_{\text{bin}}$
0	1	1	1	0	$0 + 1 + 1 = 10_{\text{bin}}$
1	0	0	0	1	$1 + 0 + 0 = 01_{\text{bin}}$
1	0	1	1	0	$1 + 0 + 1 = 10_{\text{bin}}$
1	1	0	1	0	$1 + 1 + 0 = 10_{\text{bin}}$
1	1	1	1	1	$1 + 1 + 1 = 11_{\text{bin}}$

FIGURA B.5.3 Especificação de entrada e saída para um somador de 1 bit.

Podemos expressar as funções de saída CarryOut e Soma como equações lógicas e essas equações, por sua vez, podem ser implementadas com portas lógicas. Vamos realizar um CarryOut. A [Figura B.5.4](#) mostra os valores das entradas quando CarryOut é 1.

Entradas		
a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1

FIGURA B.5.4 Valores das entradas quando CarryOut é 1.

Podemos transformar essa tabela verdade em uma equação lógica:

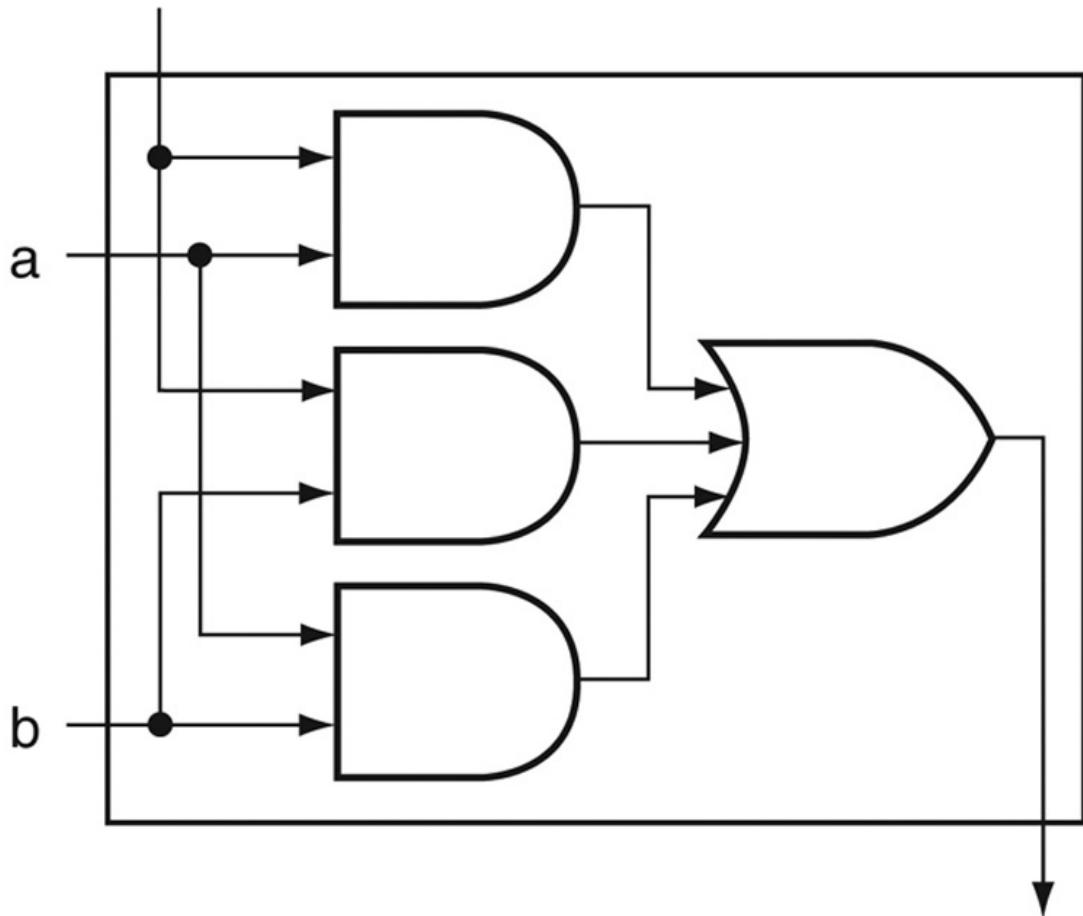
$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn})$$

Se $a \cdot b \cdot \text{CarryIn}$ for verdadeiro, então todos os outros três termos também precisam ser verdadeiros, de modo que podemos omitir esse último termo correspondente à quarta linha da tabela. Assim, podemos simplificar a equação para:

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

A [Figura B.5.5](#) mostra que o hardware dentro da caixa preta do somador para CarryOut consiste em três portas AND e uma porta OR. As três portas AND correspondem exatamente aos três termos entre parênteses da fórmula anterior para CarryOut, e a porta OR soma os três termos.

CarryIn



CarryOut

FIGURA B.5.5 Hardware do somador para o sinal CarryOut.

O restante do hardware do somador é a lógica para a saída de Soma dada na equação seguinte.

O bit Soma é ligado quando exatamente uma entrada é 1 ou quando todas as três entradas são 1. A Soma resulta em uma equação Booleana complexa (lembre-se de que a^- significa NOT a):

$$\text{Soma} = (a \cdot \bar{b} \cdot \bar{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \bar{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

O desenho da lógica para o bit Soma, na caixa preta do somador, fica como um exercício para o leitor.

A [Figura B.5.6](#) mostra uma ALU de 1 bit derivada da combinação do somador

com os componentes anteriores. Às vezes, os projetistas também querem que a ALU realize mais algumas operações simples, como gerar 0. O modo mais fácil de somar uma operação é expandir o multiplexador controlado pela linha Operação e, para este exemplo, conectar 0 diretamente à nova entrada desse multiplexador expandido.

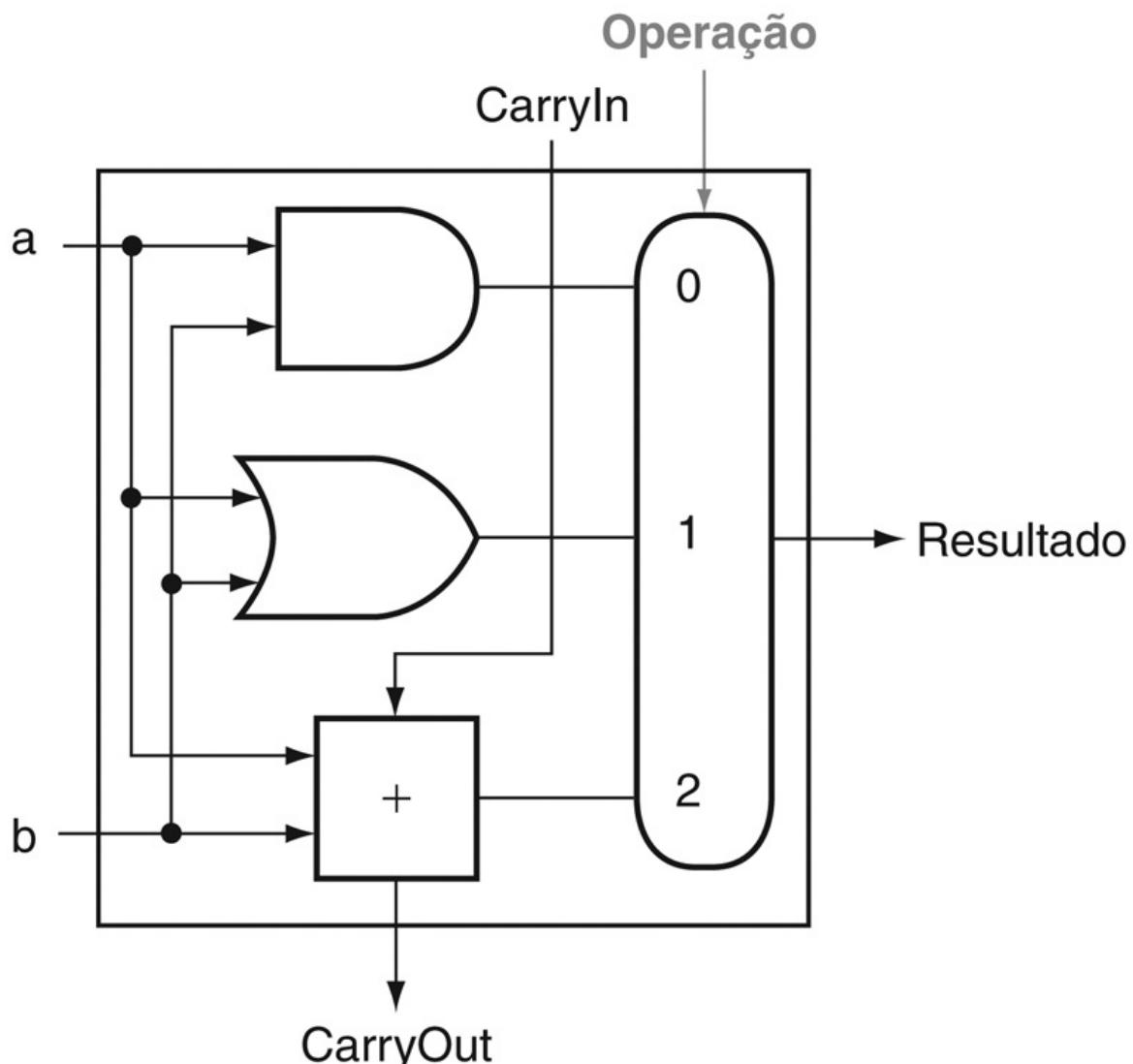


FIGURA B.5.6 Uma ALU de 1 bit realiza AND, OR e adição
([Figura B.5.5](#)).

Uma ALU de 32 bits

Agora que completamos a ALU de 1 bit, a ALU completa de 32 bits é criada conectando “caixas pretas” adjacentes. Usando x_i para indicar o i -ésimo bit de x , a [Figura B.5.7](#) mostra uma ALU de 32 bits. Assim como uma única pedra pode causar ondulações partindo da costa de um lago tranquilo, um único carry do bit menos significativo (Result0) pode causar ondulações por todo o somador, levando a uma carry do bit mais significativo (Result31). Logo, o somador criado ligando diretamente os carries de somadores de 1 bit é chamado de somador de *carry por ondulação*. Veremos um modo rápido de conectar os somadores de 1 bit a partir da página B-38.

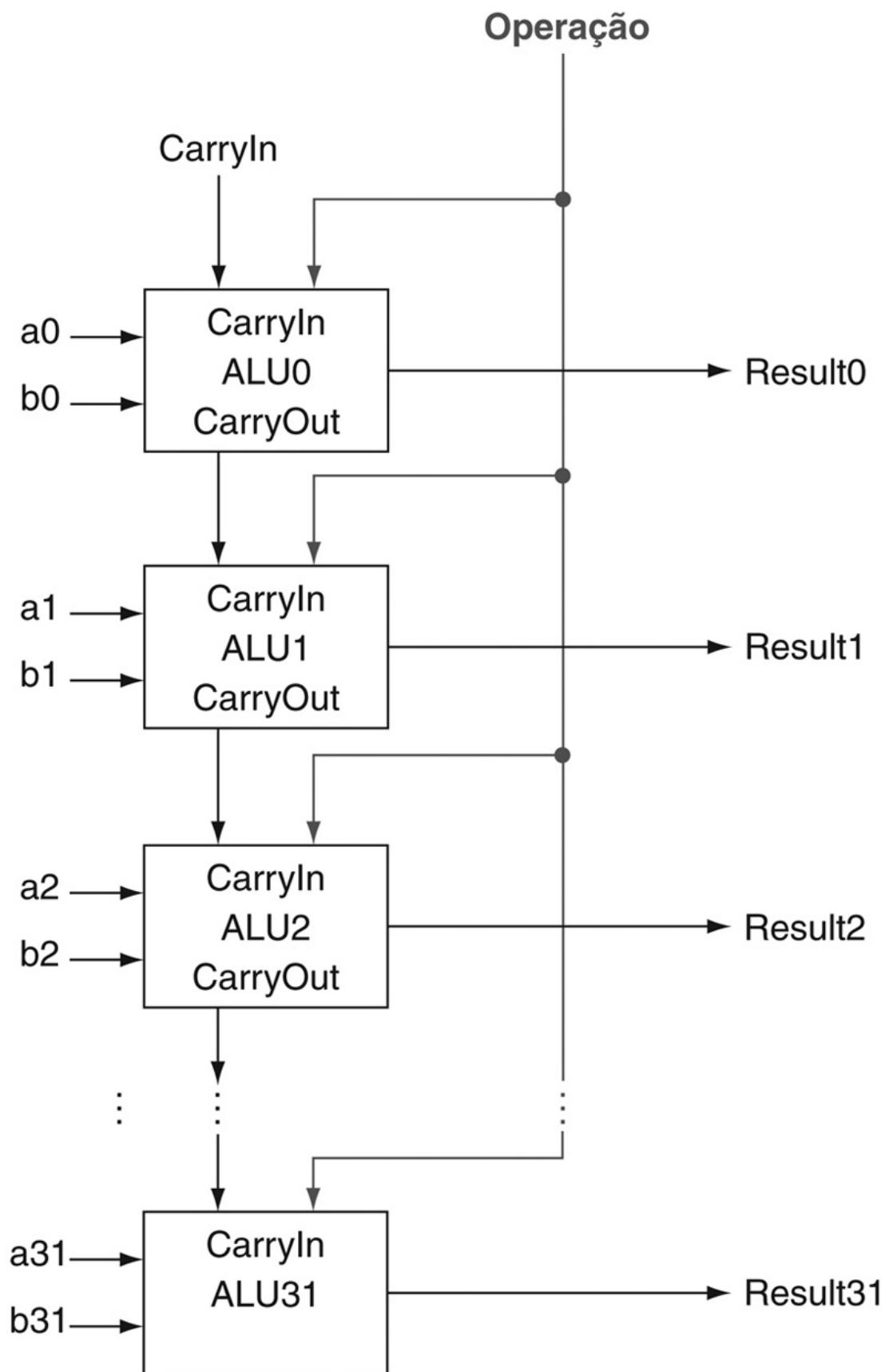


FIGURA B.5.7 Uma ALU de 32 bits construída a partir de 32 ALUs de 1 bit.

O CarryOut de um bit é conectado ao CarryIn do próximo bit mais significativo. Essa organização é chamada de carry por ondulação.

A subtração é o mesmo que a adição da versão negativa de um operando, e é assim que os somadores realizam a subtração. Lembre-se de que o atalho para negar um número em complemento a dois é inverter cada bit (às vezes chamado de *complemento a um*) e depois somar 1. Para inverter cada bit, simplesmente acrescentamos um multiplexador 2:1 que escolhe entre b e b^- , como mostra a Figura B.5.8.

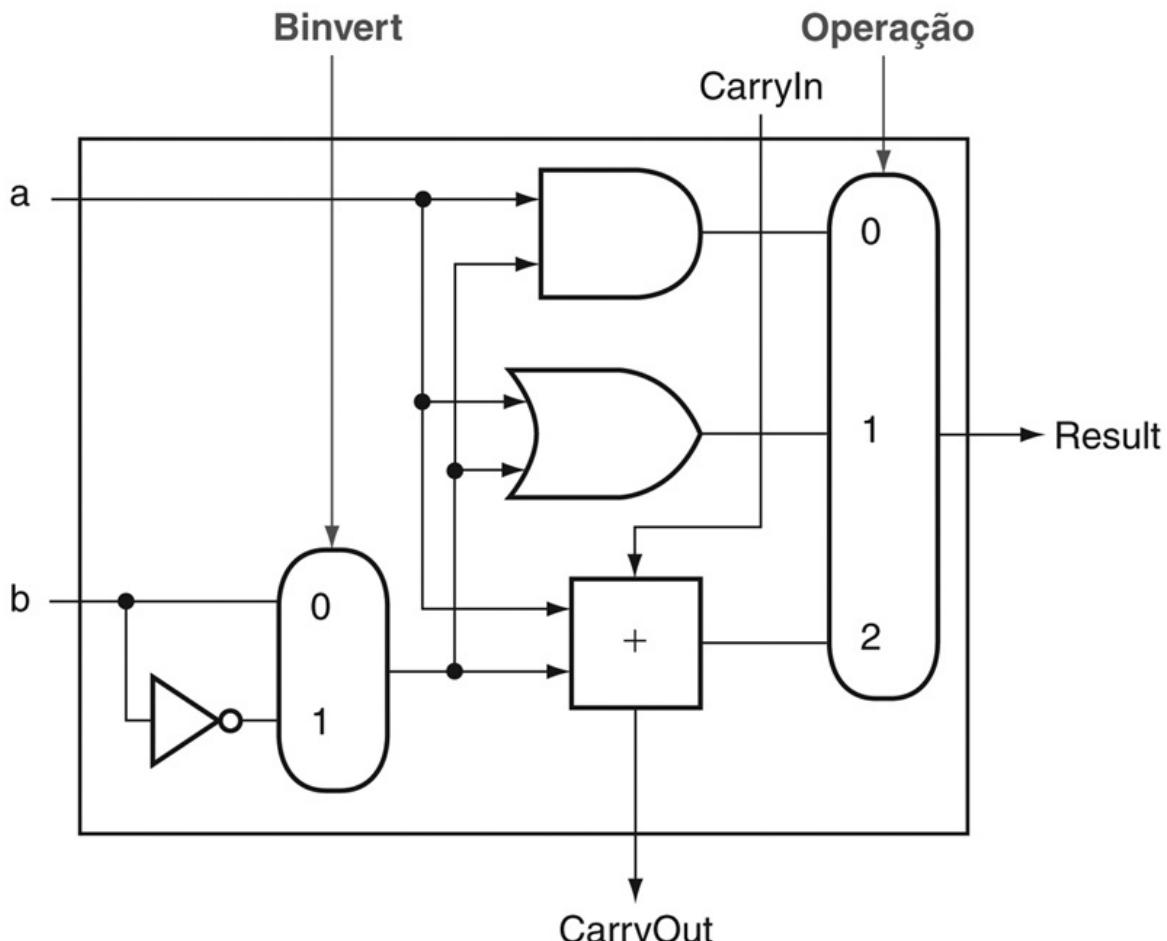


FIGURA B.5.8 Uma ALU de 1 bit que realiza AND, OR e adição entre a e b ou a e b^- .

Selecionando b^- (Binvert = 1) e definindo CarryIn como 1 no bit

menos significativo da ALU, obtemos a subtração em complemento a dois de b a partir de a, em vez da adição de b e a.

Suponha que conectemos 32 dessas ALUs de 1 bit, como fizemos na [Figura B.5.7](#). O multiplexador adicionado dá a opção de b ou seu valor invertido, dependendo de Binvert, mas essa é apenas uma etapa na negação de um número em complemento a dois. Observe que o bit menos significativo ainda possui um sinal CarryIn, embora seja desnecessário para a adição. O que acontece se definirmos esse CarryIn como 1 em vez de 0? O somador, então, calculará $a + b + 1$. Selecionando a versão invertida de b, obtemos exatamente o que queremos:

$$a + \bar{b} + 1 = a + (\bar{b} + 1) = a + (-b) = a - b$$

A simplicidade do projeto de hardware de um somador em complemento a dois ajuda a explicar por que a representação em complemento a dois tornou-se um padrão universal para a aritmética computacional com inteiros.

Uma ALU no MIPS também precisa de uma função NOR. Em vez de acrescentar uma porta separada para NOR, podemos reutilizar grande parte do hardware já existente na ALU, como fizemos para a subtração. A ideia vem da seguinte tabela verdade sobre NOR:

$$\overline{(a + b)} = \bar{a} \cdot \bar{b}$$

Ou seja, NOT (a OR b) é equivalente a NOT a AND NOT b. Esse fato é chamado de teorema de DeMorgan e é explorado nos exercícios com mais profundidade.

Como temos AND e NOT b, só precisamos acrescentar NOT a à ALU. A [Figura B.5.9](#) mostra essa mudança.

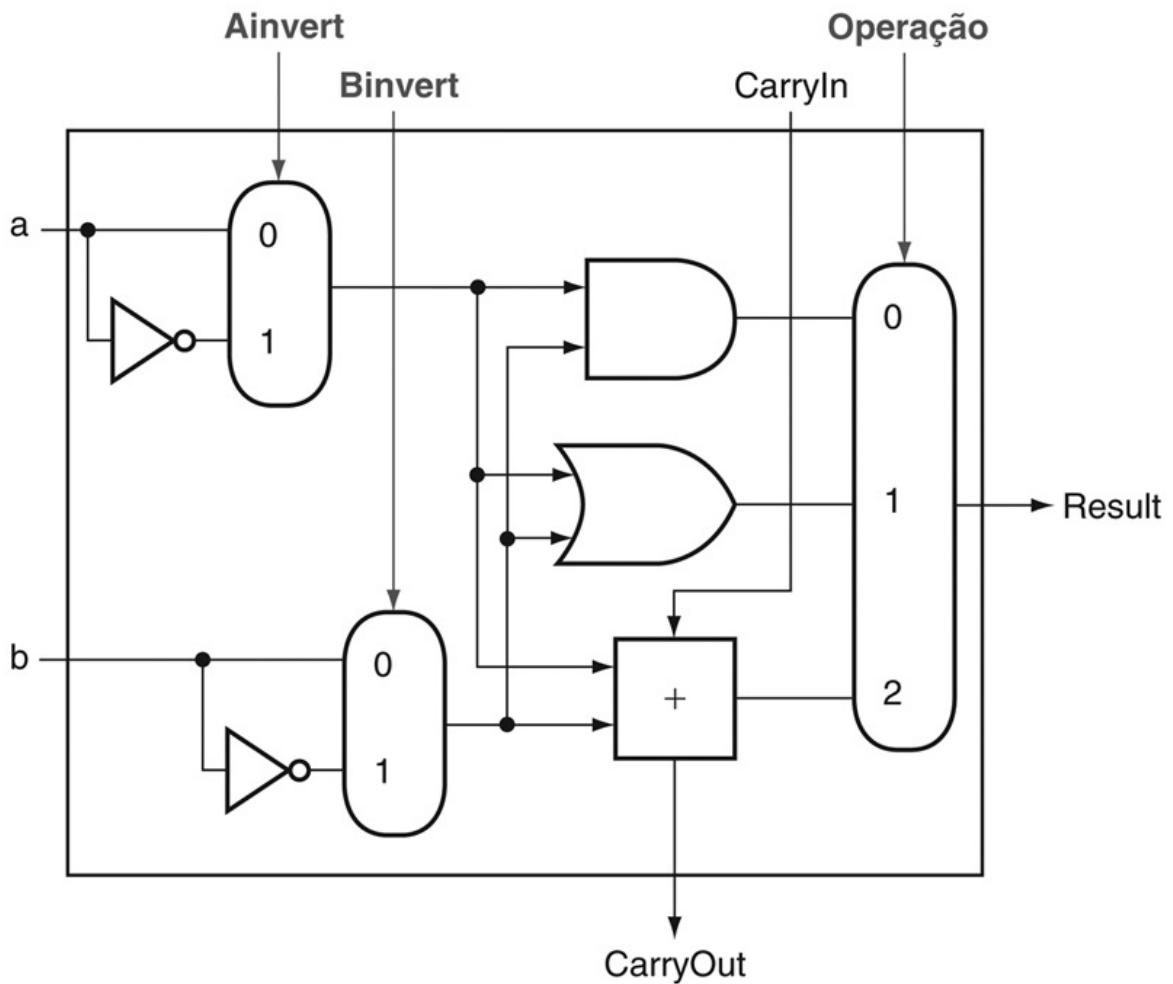


FIGURA B.5.9 Uma ALU de 1 bit que realiza AND, OR e adição entre a e b ou a- e b-.

Selecionando a (**Ainvert** = 1) e b (**Binvert** = 1), obtemos a NOR b, em vez de a AND b.

Ajustando a ALU de 32 bits ao MIPS

Essas quatro operações – adição, subtração, AND, OR – são encontradas na ALU de quase todo computador e as operações da maioria das instruções MIPS podem ser realizadas por essa ALU. Mas o projeto da ALU está incompleto.

Uma instrução que ainda precisa de suporte é a instrução “set on less than” (**slt**). Lembre-se de que a operação produz 1 se $rs < rt$, ou 0 em caso contrário. Consequentemente, **slt** definirá como 0 todos os bits, menos o bit menos significativo, com o valor do bit menos significativo definido de acordo com a comparação. Para a ALU realizar **slt**, primeiro precisamos expandir o

multiplexador de três entradas da [Figura B.5.8](#) para acrescentar uma entrada para o resultado de `slt`. Chamamos essa nova entrada de *Less* e a usamos apenas para `slt`.

O desenho superior da [Figura B.5.10](#) mostra a nova ALU de 1 bit com o multiplexador expandido. A partir da descrição de `slt` anterior, temos de conectar 0 à entrada *Less* para os 31 bits superiores da ALU, pois esses bits sempre serão 0. O que falta considerar é como comparar e definir o valor do *bit menos significativo* para instruções “set on less than”.

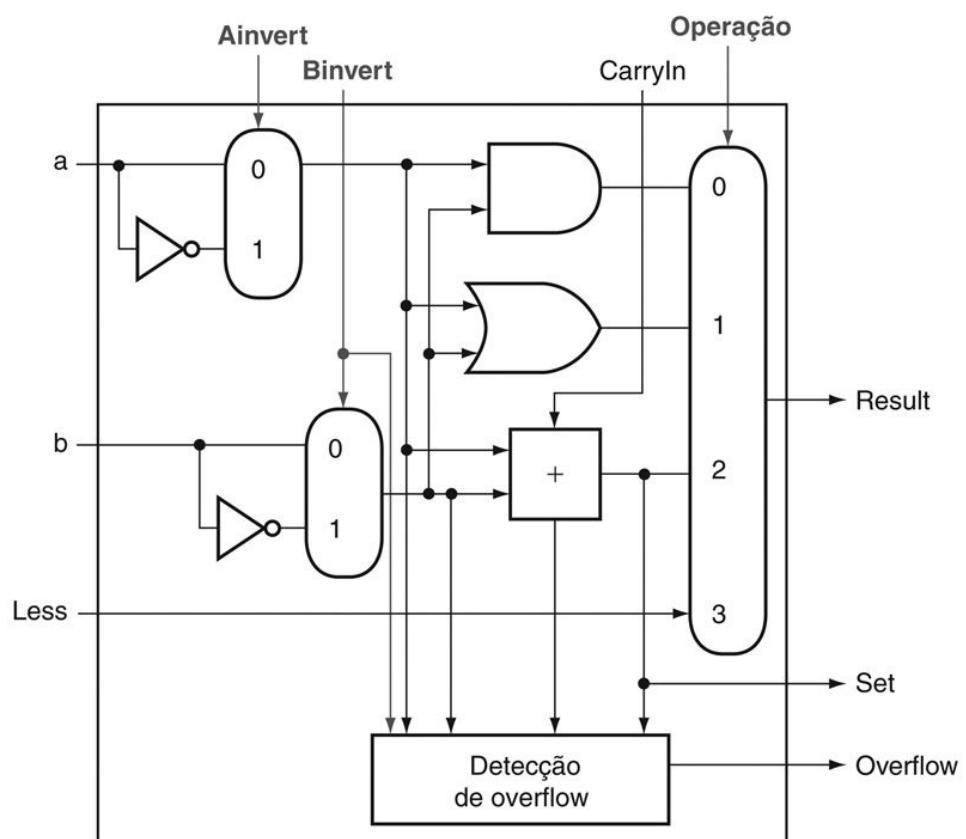
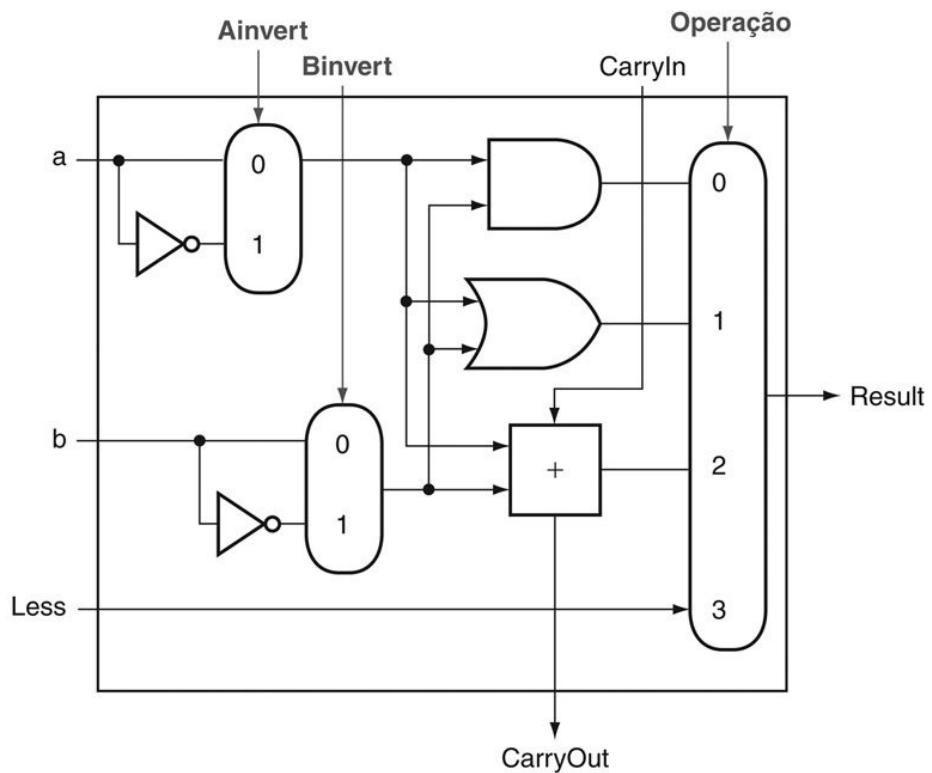


FIGURA B.5.10 (Superior) Uma ALU de 1 bit que realiza AND, OR e adição entre a e b ou b^- , e (inferior) uma ALU de 1 bit para o bit mais significativo.

O desenho superior inclui uma entrada direta que está conectada para realizar a operação “set on less than” ([Figura B.5.11](#)); o desenho inferior possui uma saída direta do somador para a comparação “less than”, chamada Set. (Veja, no Exercício B.24, no final deste apêndice, como calcular o overflow com menos entradas.)

O que acontece se subtrairmos b de a? Se a diferença for negativa, então $a < b$, pois

$$\begin{aligned}(a - b) < 0 &\Rightarrow ((a - b) + b) < (0 + b) \\ &\Rightarrow a < b\end{aligned}$$

Queremos que o bit menos significativo de uma operação “set on less than” seja 1 se $a < b$; ou seja, 1 se $a - b$ for negativo e 0 se for positivo. Esse resultado desejado corresponde exatamente aos valores do bit de sinal: 1 significa negativo e 0 significa positivo. Seguindo essa linha de argumento, só precisamos conectar o bit de sinal da saída do somador ao bit menos significativo para obter “set on less than”.

Infelizmente, a saída Result do bit da ALU mais significativo no topo da [Figura B.5.10](#) para a operação `slt` *não* é a saída do somador; a saída da ALU para a operação `slt` é obviamente o valor de entrada `Less`.

Assim, precisamos de uma nova ALU de 1 bit, para o bit mais significativo, a qual possui um bit de saída extra: a saída do somador. O desenho inferior da [Figura B.5.10](#) mostra o projeto, com essa nova linha de saída do somador chamada `Set` e usada apenas para `slt`. Como precisamos de uma ALU especial para o bit mais significativo, acrescentamos a lógica de detecção de overflow, pois também está associada a esse bit.

Infelizmente, o teste de “less than” é um pouco mais complicado do que acabamos de descrever, devido ao overflow, conforme exploramos nos exercícios. A [Figura B.5.11](#) mostra a ALU de 32 bits.

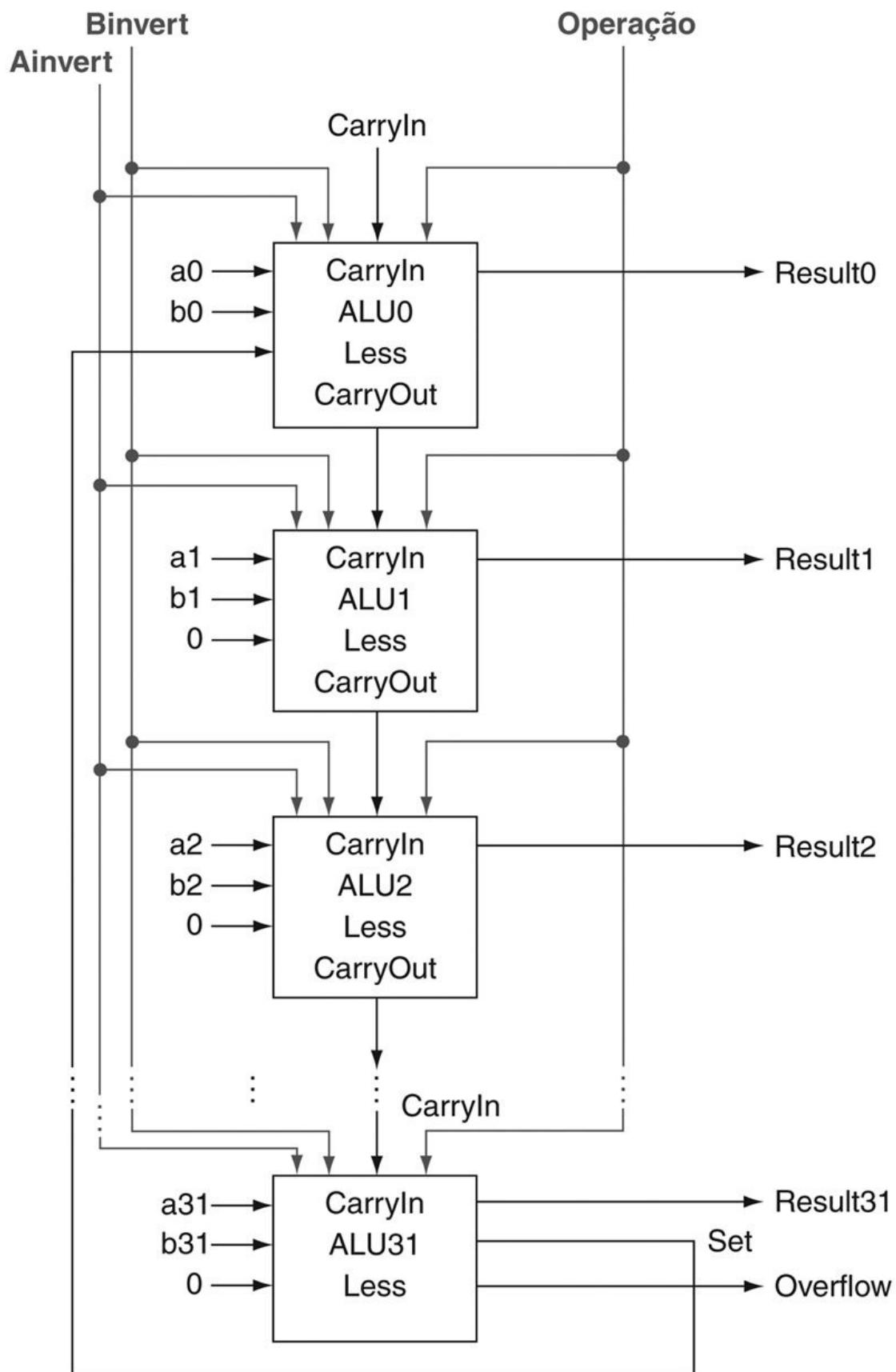


FIGURA B.5.11 Uma ALU de 32 bits construída a partir de 31 cópias da ALU de 1 bit na parte superior da [Figura B.5.10](#) e uma ALU de 1 bit na parte inferior dessa figura.

As entradas Less são conectadas a 0, exceto para o bit menos significativo, que está conectado à saída Set do bit mais significativo. Se a ALU realizar $a - b$ e selecionarmos a entrada 3 no multiplexador da [Figura B.5.10](#), então Result = 0 ... 001 se $a < b$, e Result = 0 ... 000 caso contrário.

Observe que toda vez que quisermos que a ALU subtraia, colocamos CarryIn e Binvert em 1. Para adições ou operações lógicas, queremos que as duas linhas de controle sejam 0. Portanto, podemos simplificar o controle da ALU combinando CarryIn e Binvert a uma única linha de controle, chamada *Bnigate*.

Para ajustar ainda mais a ALU ao conjunto de instruções do MIPS, temos de dar suporte a instruções de desvio condicional. Essas instruções desviam se dois registradores forem iguais ou se forem diferentes. O modo mais fácil de testar a igualdade com a ALU é subtrair b de a e depois testar se o resultado é zero, pois

$$(a - b = 0) \Rightarrow a = b$$

Assim, se acrescentarmos hardware para testar se o resultado é 0, podemos testar a igualdade. O modo mais simples é realizar um OR de todas as saídas juntas e depois enviar esse sinal por um inverter:

$$\text{Zero} = \overline{(\text{Result31} + \text{Result30} + \dots + \text{Result2} + \text{Result1} + \text{Result0})}$$

A [Figura B.5.12](#) mostra a ALU de 32 bits revisada. Podemos pensar na combinação da linha Ainvert de 1 bit, a linha Binvert de 1 bit, e as linhas de Operação de 2 bits como linhas de controle de 4 bits para a ALU, pedindo que realize soma, subtração, AND, OR ou “set on less than”. A [Figura B.5.13](#) mostra as linhas de controle da ALU e a operação ALU correspondente.

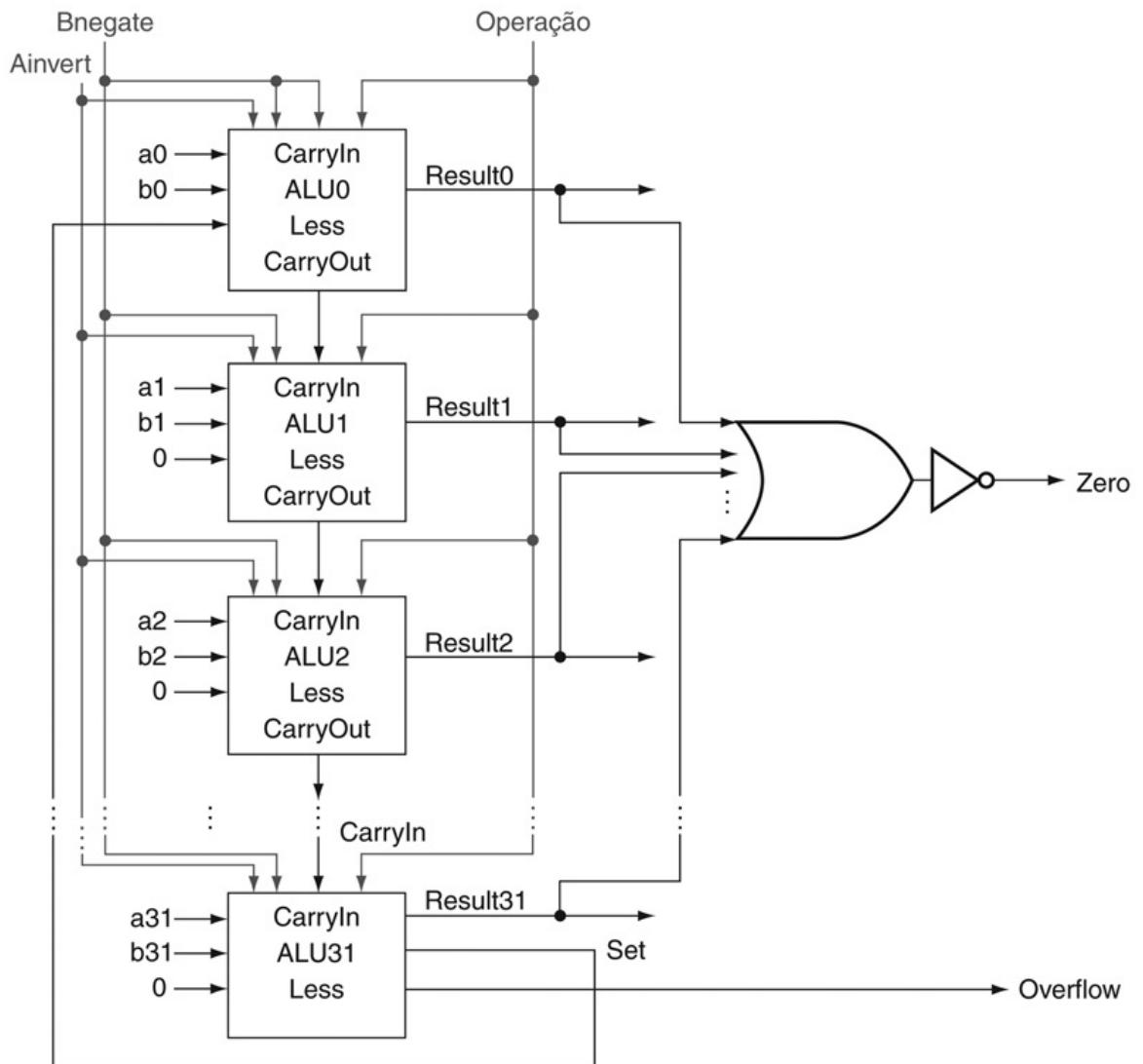


FIGURA B.5.12 A ALU final de 32 bits.

Isso acrescenta um detetor de zero à [Figura B.5.11](#).

Linhas de controle ALU	Função
0000	AND
0001	OR
0010	adição
0110	subtração
0111	set on less than
1100	NOR

FIGURA B.5.13 Os valores das três linhas de controle ALU, Bnegate e Operação, juntamente com as operações ALU correspondentes.

Finalmente, agora que vimos o que há dentro de uma ALU de 32 bits, usaremos o símbolo universal para uma ALU completa, como mostra a [Figura B.5.14](#).

Operação ALU

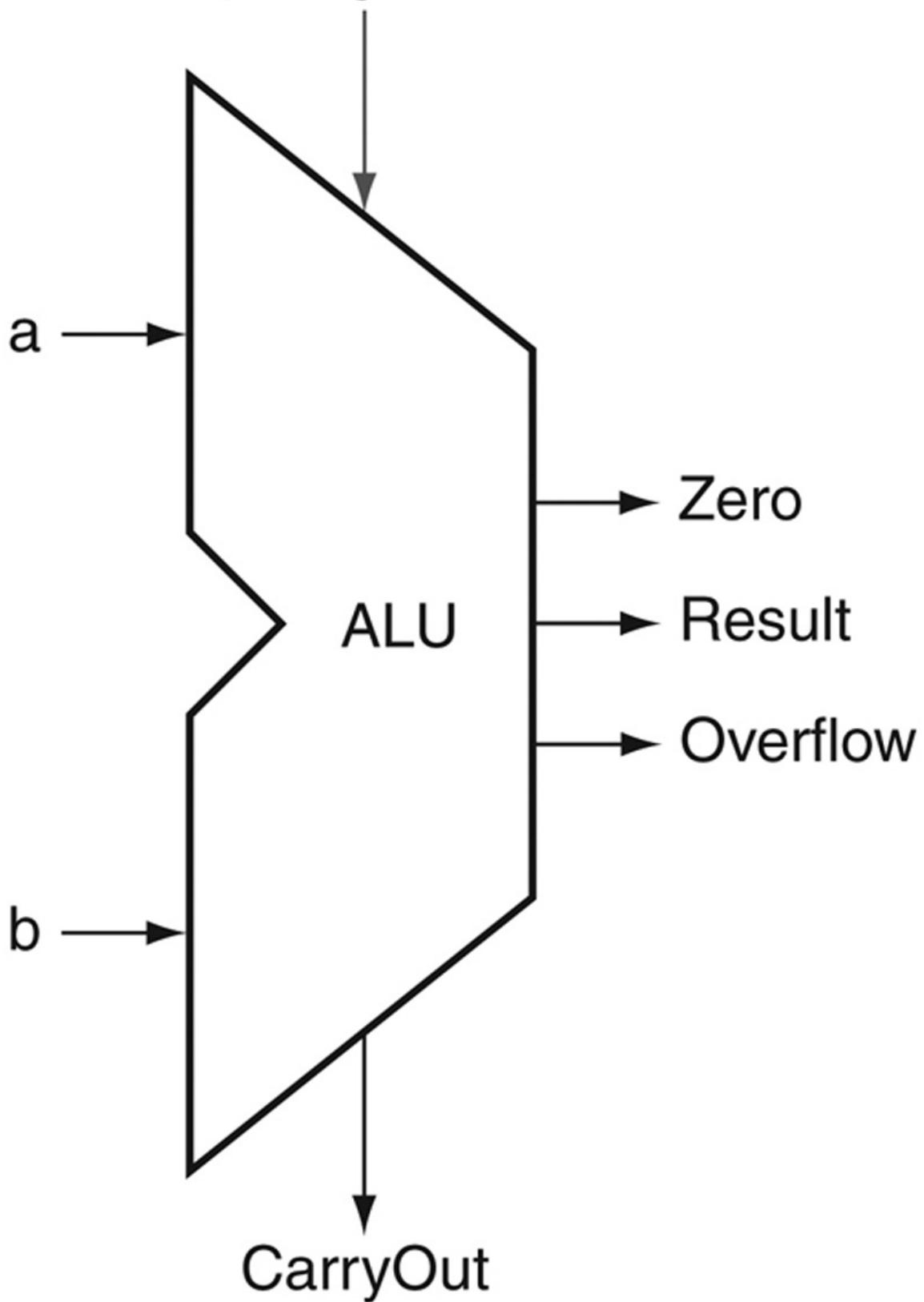


FIGURA B.5.14 O símbolo normalmente usado para representar uma ALU, como mostra a [Figura B.5.12](#).

Esse símbolo também é usado para representar um somador, de modo que normalmente é rotulado com ALU ou Adder (somador).

Definindo a ALU MIPS em Verilog

A [Figura B.5.15](#) mostra como uma ALU combinacional do MIPS poderia ser especificada em Verilog; essa especificação provavelmente seria compilada com uma biblioteca de partes padrão, que oferecesse um somador, que poderia ser instanciado. Para completar, mostramos o controle da ALU para o MIPS na [Figura B.5.16](#), que usamos no [Capítulo 4](#), onde montamos uma versão Verilog do caminho de dados do MIPS.

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
    input [3:0] ALUctl;
    input [31:0] A,B;
    output reg [31:0] ALUOut;
    output Zero;
    assign Zero = (ALUOut==0); //Zero é true se ALUOut é 0
    always @(ALUctl, A, B) begin //reavalia se isso mudar
        case (ALUctl)
            0: ALUOut <= A & B;
            1: ALUOut <= A | B;
            2: ALUOut <= A + B;
            6: ALUOut <= A - B;
            7: ALUOut <= A < B ? 1 : 0;
            12: ALUOut <= ~(A | B); // resultado é nor
        default: ALUOut <= 0;
        endcase
    end
endmodule
```

FIGURA B.5.15 Definição comportamental em Verilog de uma ALU MIPS.

```

module ALUControl (ALUOp, FuncCode, ALUCl);
    input [1:0] ALUOp;
    input [5:0] FuncCode;
    output [3:0] reg ALUCl;
    always case (FuncCode)
        32: ALUOp<=2; // soma
        34: ALUOp<=6; //subtrai
        36: ALUOP<=0; // and
        37: ALUOp<=1; // or
        39: ALUOp<=12; // nor
        42: ALUOp<=7; // slt
        default: ALUOp<=15; // não deverá acontecer
    endcase
endmodule

```

FIGURA B.5.16 O controle ALU do MIPS: um peça simples da lógica de controle combinacional.

A próxima pergunta é: com que rapidez essa ALU pode somar dois operandos de 32 bits? Podemos determinar as entradas a e b, mas a entrada CarryIn depende da operação no somador de 1 bit adjacente. Se traçarmos todo o caminho pela cadeia de dependências, conectamos o bit mais significativo ao bit menos significativo, de modo que o bit mais significativo da soma precisa esperar pela avaliação *sequencial* de todos os 32 somadores de 1 bit. Essa reação em cadeia sequencial é muito lenta para ser usada no hardware de tempo crítico. A próxima seção explora como agilizar a adição. Esse assunto não é fundamental para a compreensão do restante do apêndice e pode ser pulado.

Verifique você mesmo

Suponha que você queira acrescentar a operação NOT (a AND b), chamada NAND. Como a ALU poderia mudar para dar suporte a ela?

1. Nenhuma mudança. Você pode calcular NAND rapidamente usando a ALU atual, pois $(a \cdot b) = \overline{a} + \overline{b}$ e já temos NOT a, NOT b, e OR.
2. Você precisa expandir o multiplexador grande para acrescentar outra entrada e depois acrescentar nova lógica para calcular NAND.

B.6. Adição mais rápida: Carry Lookahead

A chave para agilizar a adição é determinar o carry in para os bits mais significativos mais cedo. Existem diversos esquemas para antecipar o carry, de modo que o cenário de pior caso é uma função do \log_2 do número de bits do somador. Esses sinais antecipatórios são mais rápidos, porque passam por menos portas em sequência, mas são necessárias muito mais portas para antecipar o carry apropriado.

Uma chave para entender os esquemas de carry rápido é lembrar que, diferente do software, o hardware executa em paralelo sempre que as entradas mudam.

Carry rápido usando hardware “infinito”

Conforme mencionamos anteriormente, qualquer equação pode ser representada em dois níveis de lógica. Como as únicas entradas externas são os dois operandos e o CarryIn para o bit menos significativo do somador, em teoria, poderíamos calcular os valores de CarryIn para todos os bits restantes do somador com apenas dois níveis de lógica.

Por exemplo, o CarryIn para o bit 2 do somador é exatamente o CarryOut do bit 1, de modo que a fórmula é:

$$\text{CarryIn2} = (b_1 \cdot \text{CarryIn1}) + (a_1 \cdot \text{CarryIn1}) + (a_1 \cdot b_1)$$

De modo semelhante, CarryIn1 é definido como:

$$\text{CarryIn1} = (b_0 \cdot \text{CarryIn0}) + (a_0 \cdot \text{CarryIn0}) + (a_0 \cdot b_0)$$

Usando a abreviação mais curta e mais tradicional de ci para CarryIn, podemos reescrever as fórmulas como:

$$c_2 = (b_1 \cdot c_1) + (a_1 \cdot c_1) + (a_1 \cdot b_1)$$

$$c_1 = (b_0 \cdot c_0) + (a_0 \cdot c_0) + (a_0 \cdot b_0)$$

Substituindo a definição de c_1 para a primeira equação, o resultado é esta fórmula:

$$c_2 = (a_1 \cdot a_0 \cdot b_0) + (a_1 \cdot a_0 \cdot c_0) + (a_1 \cdot b_0 \cdot c_0)$$

$$+ (b_1 \cdot a_0 \cdot b_0) + (b_1 \cdot a_0 \cdot c_0) + (b_1 \cdot b_0 \cdot c_0) + (a_1 \cdot b_1)$$

Você pode imaginar como a equação se expande à medida que chegamos a bits mais significativos do somador; ela cresce rapidamente com o número de bits. Essa complexidade é refletida no custo do hardware para o carry rápido, tornando esse esquema simples tremendamente dispendioso para somadores largos.

Carry rápido usando o primeiro nível de abstração: propagar e gerar

A maior parte dos esquemas de carry rápido limita a complexidade das equações para simplificar o hardware, enquanto ainda causa melhorias de velocidade substanciais em relação ao carry por ondulação. Um esquema desse tipo é um *somador carry lookahead*. No [Capítulo 1](#), dissemos que os sistemas computacionais enfrentam a complexidade usando níveis de abstração. Um somador carry lookahead conta com níveis de abstração em sua implementação.

Vamos fatorar a equação original como uma primeira etapa:

$$c_{i+1} = (b_i \cdot c_i) + (a_i \cdot c_i) + (a_i \cdot b_i)$$

$$= (a_i \cdot b_i) + (a_i + b_i) \cdot c_i$$

Se tivéssemos de reescrever a equação para c_2 usando essa fórmula, veríamos alguns padrões repetidos:

$$c_2 = (a_1 \cdot b_1) + (a_1 + b_1) \cdot ((a_0 \cdot b_0) + (a_0 + b_0) \cdot c_0)$$

Observe o surgimento repetido de $(a_i \cdot b_i)$ e $(a_i + b_i)$ na fórmula anterior. Eses dois fatores importantes são tradicionalmente chamados *gerar* (g_i) e *propagar* (p_i):

$$\begin{aligned} g_i &= a_i \cdot b_i \\ p_i &= a_i + b_i \end{aligned}$$

Usando-os para definir c_{i+1} , obtemos

$$c_{i+1} = g_i + p_i \cdot c_i$$

Para ver de onde os sinais recebem seus nomes, suponha que g_i seja 1. Então

$$c_{i+1} = g_i + p_i \cdot c_i = 1 + p_i \cdot c_i = 1$$

Ou seja, o somador *gera* um CarryOut (c_{i+1}) independente do valor de CarryIn (c_i). Agora, suponha que g_i seja 0 e p_i seja 1. Então

$$c_{i+1} = g_i + p_i \cdot c_i = 0 + 1 \cdot c_i = c_i$$

Ou seja, o somador *propaga* CarryIn para um CarryOut. Juntando os dois, $CarryIni+1$ é 1 se g_i for 1 ou se p_i for 1 e $CarryIni$ for 1.

Como uma analogia, imagine uma fileira de dominós encostados um no outro. O dominó da outra ponta pode ser empurrado para mais longe desde que não existam intervalos entre eles. De modo semelhante, um carry out pode se tornar verdadeiro por uma geração distante desde que todas as propagações entre eles sejam verdadeiras.

Contando com as definições de propagar e gerar como nosso primeiro nível de abstração, podemos expressar o sinal CarryIn de forma mais econômica. Vamos mostrá-lo para 4 bits:

$$c_1 = g_0 + (p_0 \cdot c_0)$$

$$c_2 = g_1 + (p_1 \cdot g_0) + (p_1 \cdot p_0 \cdot c_0)$$

$$c_3 = g_2 + (p_2 \cdot g_1) + (p_2 \cdot p_1 \cdot g_0) + (p_2 \cdot p_1 \cdot p_0 \cdot c_0)$$

$$c_4 = g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0) \\ + (p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0)$$

Essas equações representam apenas o bom senso: `CarryIni` é 1 se algum somador anterior gerar um carry e todos os somadores intermediários propagarem um carry. A [Figura B.6.1](#) usa um encanamento para tentar explicar o carry lookahead.

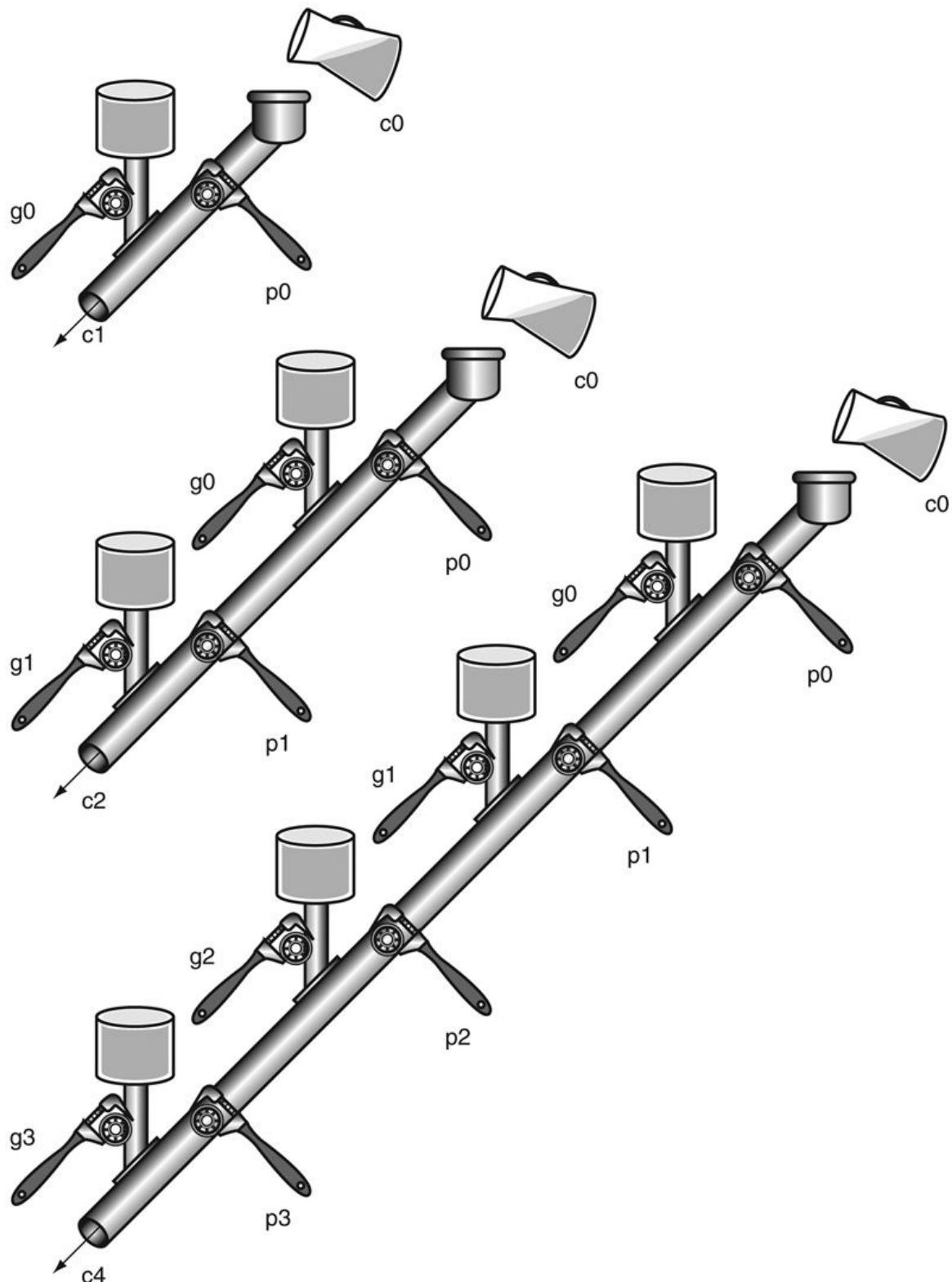


FIGURA B.6.1 Uma analogia de encanamento para o carry lookahead para 1 bit, 2 bits e 4 bits, usando canos d'água e registros.

As chaves são viradas para abrir e fechar os registros. A água aparece em destaque. A saída do encanamento (c_{i+1}) estará completa se o valor gerado mais próximo (g_i) estiver aberto ou se o valor de propagação i (p_i) estiver aberto e houver fluxo de água acima, seja de um gerador anterior, ou propagado com água por trás dele. O CarryIn (c_0) pode resultar em um carry out sem a ajuda de quaisquer gerações, mas com a ajuda de *todas* as propagações.

Até mesmo essa forma simplificada leva a grandes equações e, portanto, uma lógica considerável, mesmo para um somador de 16 bits. Vamos tentar prosseguir para dois níveis de abstração.

Carry rápido usando o segundo nível de abstração

Primeiro, consideramos esse somador de 4 bits com sua lógica de carry lookahead como um único bloco de montagem. Se os conectarmos no padrão de carry por ondulação para formar um somador de 16 bits, a soma será mais rápida do que a original, com um pouco mais de hardware. Para ir mais rápido, precisaremos do carry lookahead em um nível mais alto. Para realizar o carry lookahead para somadores de 4 bits, precisamos propagar e gerar sinais nesse nível mais alto. Aqui, eles são para os quatro blocos somadores de 4 bits:

$$\begin{aligned}P_0 &= p_3 \cdot p_2 \cdot p_1 \cdot p_0 \\P_1 &= p_7 \cdot p_6 \cdot p_5 \cdot p_4 \\P_2 &= p_{11} \cdot p_{10} \cdot p_9 \cdot p_8 \\P_3 &= p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12}\end{aligned}$$

Ou seja, o sinal de “super” propagação para a abstração de 4 bits (P_i) é verdadeiro somente se cada um desses bits no grupo propagar um carry.

Para o sinal de “super” geração (G_i), nos importamos apenas se houver um carry out do bit mais significativo do grupo de 4 bits. Isso obviamente ocorre se a geração for verdadeira para esse bit mais significativo; ela também ocorre se uma geração anterior for verdadeira e todas as propagações intermediárias, incluindo aquela do bit mais significativo, também forem verdadeiras:

$$G_0 = g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0)$$

$$G_1 = g_7 + (p_7 \cdot g_6) + (p_7 \cdot p_6 \cdot g_5) + (p_7 \cdot p_6 \cdot p_5 \cdot g_4)$$

$$G_2 = g_{11} + (p_{11} \cdot g_{10}) + (p_{11} \cdot p_{10} \cdot g_9) + (p_{11} \cdot p_{10} \cdot p_9 \cdot g_8)$$

$$G_3 = g_{15} + (p_{15} \cdot g_{14}) + (p_{15} \cdot p_{14} \cdot g_{13}) + (p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12})$$

A [Figura B.6.2](#) atualiza nossa analogia de encanamento para mostrar P0 e G0.

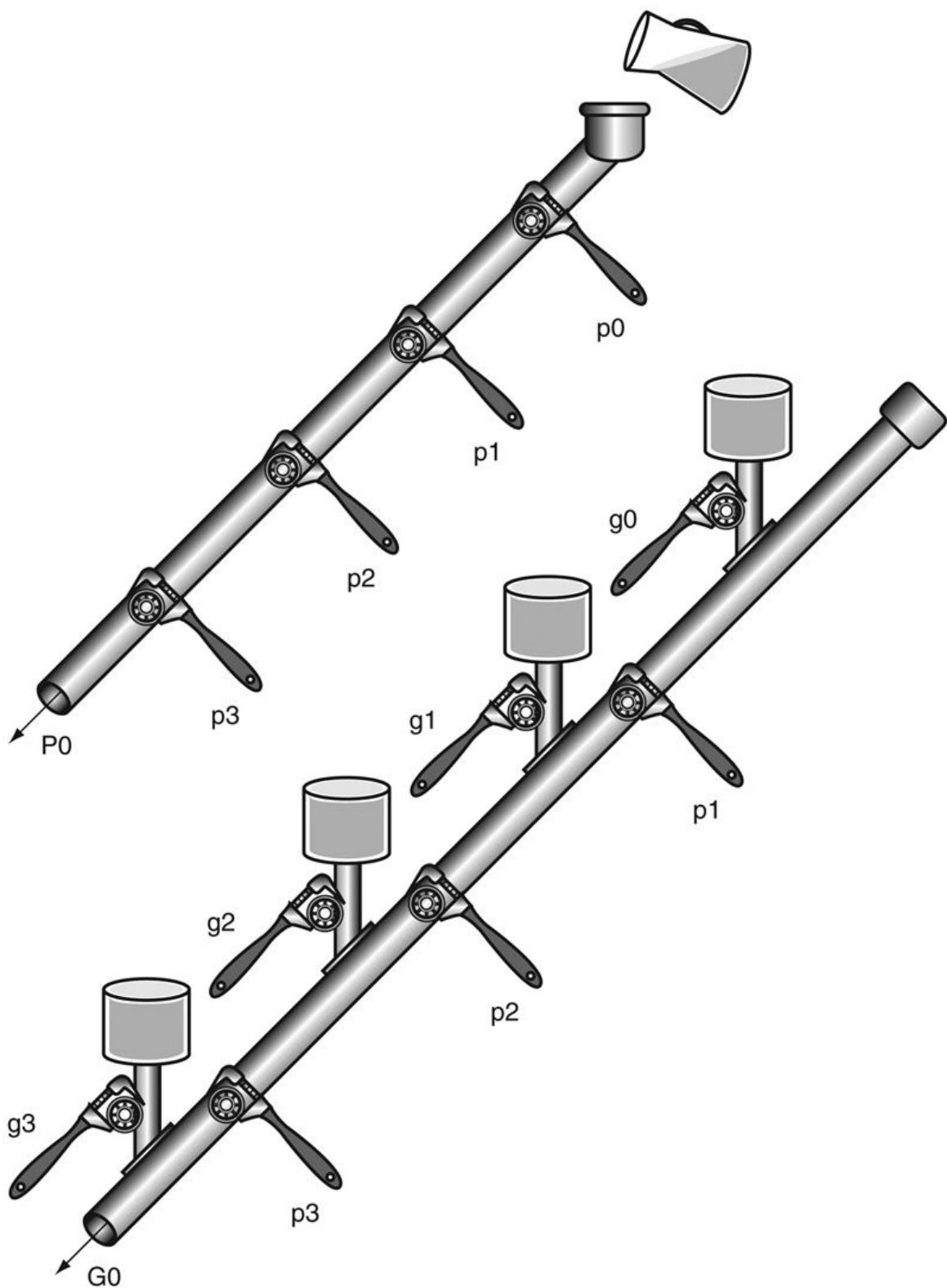


FIGURA B.6.2 Uma analogia de encanamento para os sinais de carry lookahead de próximo nível P0 e G0.

P0 é aberto apenas se todas as quatro propagações (p_i)

estiverem abertas, enquanto a água flui em G0 somente se, pelo menos, uma geração (g_i) estiver aberta e todas as propagações de fluxo abaixo, a partir dessa geração, estiverem abertas.

Então, as equações nesse nível de abstração mais alto para o carry in para cada grupo de 4 bits do somador de 16 bits (C1, C2, C3, C4 na [Figura B.6.3](#)) são muito semelhantes às equações de carry out para cada bit do somador de 4 bits (c1, c2, c3, c4) na página B-40:

$$C1 = G0 + (P0 \cdot c0)$$

$$C2 = G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0)$$

$$C3 = G2 + (P2 \cdot G1) + (P2 \cdot P1 \cdot G0) + (P2 \cdot P1 \cdot P0 \cdot c0)$$

$$C4 = G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) \\ + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)$$

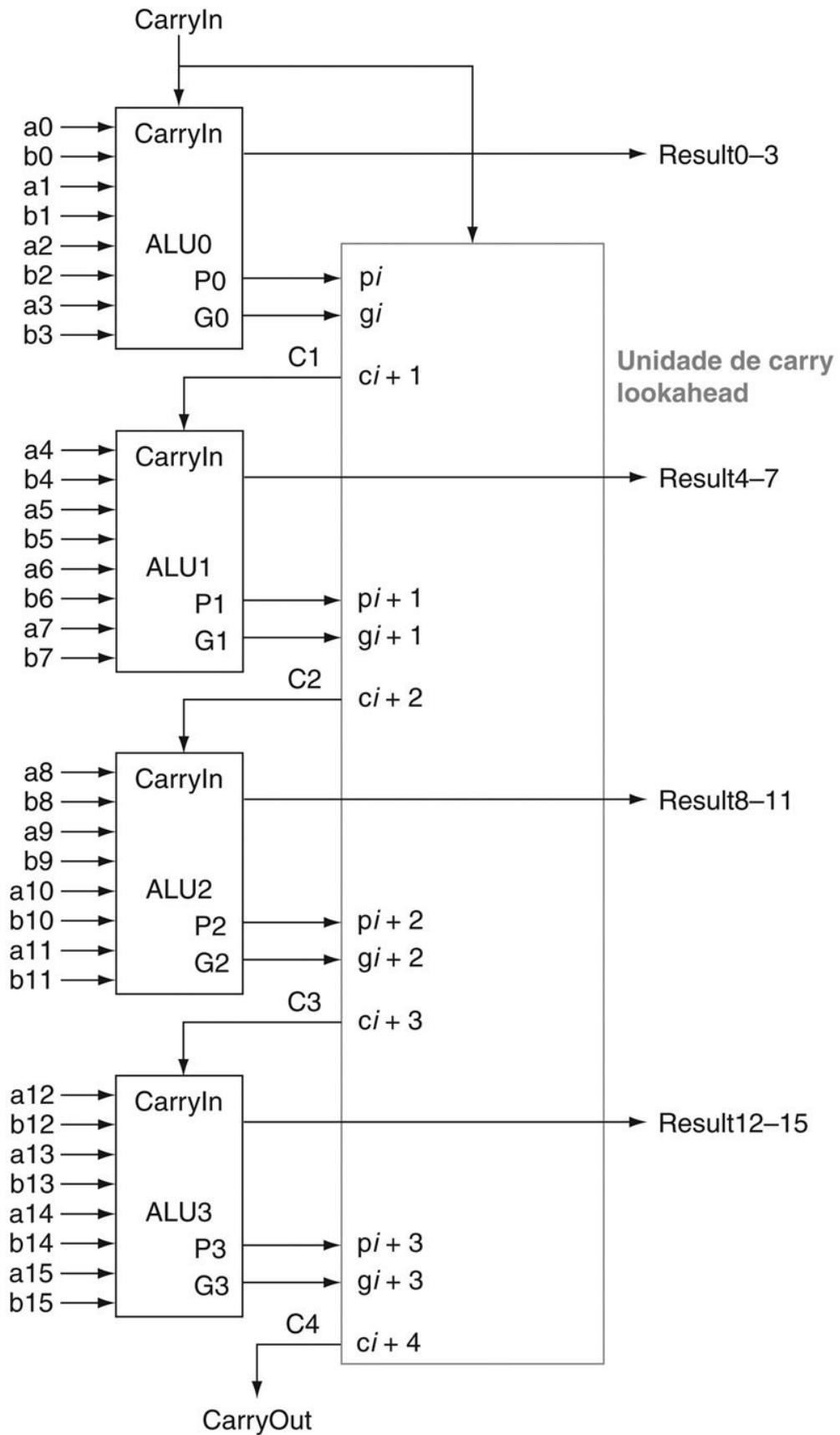


FIGURA B.6.3 Quatro ALUs de 4 bits usando carry lookahead para formar um somador de 16 bits.
Observe que os carries vêm da unidade de carry lookahead, e não das ALUs de 4 bits.

A Figura B.6.3 mostra somadores de 4 bits conectados com tal unidade de carry lookahead. Os exercícios exploram as diferenças de velocidade entre esses esquemas de carry, diferentes notações para a propagação e geração de sinais em múltiplos bits, e o projeto de um somador de 64 bits.

Ambos os níveis de propagação e geração

Exemplo

Determine os valores de gi , pi , Pi e Gi destes dois números de 16 bits:

a :	0001	1010	0011	0011	_{bin}
b :	1110	0101	1110	1011	_{bin}

Além disso, qual é o CarryOut15 (C4)?

Resposta

O alinhamento dos bits facilita ver os valores de geração gi ($ai \cdot bi$) e propagação pi ($ai + bi$):

a :	0001	1010	0011	0011
b :	1110	0101	1110	1011
gi :	0000	0000	0010	0011
pi :	1111	1111	1111	1011

onde os bits são numerados de 15 a 0, da esquerda para a direita. Em

seguida, as “super” propagações (P_3 , P_2 , P_1 , P_0) são simplesmente o AND das propagações de nível inferior:

$$\begin{aligned}P_3 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\P_2 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\P_1 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\P_0 &= 1 \cdot 0 \cdot 1 \cdot 1 = 0\end{aligned}$$

Os “super” geradores são mais complexos; portanto, use as seguintes equações:

$$\begin{aligned}G_0 &= g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0) \\&= 0 + (1 \cdot 0) + (1 \cdot 0 \cdot 1) + (1 \cdot 0 \cdot 1 \cdot 1) = 0 + 0 + 0 + 0 = 0 \\G_1 &= g_7 + (p_7 \cdot g_6) + (p_7 \cdot p_6 \cdot g_5) + (p_7 \cdot p_6 \cdot p_5 \cdot g_4) \\&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 1 + 0 = 1 \\G_2 &= g_{11} + (p_{11} \cdot g_{10}) + (p_{11} \cdot p_{10} \cdot g_9) + (p_{11} \cdot p_{10} \cdot p_9 \cdot g_8) \\&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0 \\G_3 &= g_{15} + (p_{15} \cdot g_{14}) + (p_{15} \cdot p_{14} \cdot g_{13}) + (p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12}) \\&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0\end{aligned}$$

Finalmente, CarryOut15 é

$$\begin{aligned}C_4 &= G_3 + (P_3 \cdot G_2) + (P_3 \cdot P_2 \cdot G_1) + (P_3 \cdot P_2 \cdot P_1 \cdot G_0) \\&\quad + (P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0) \\&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0 \cdot 0) \\&= 0 + 0 + 1 + 0 + 0 = 1\end{aligned}$$

Logo, existe um carry out quando se somam esses dois números de 16 bits.

O motivo pelo qual o carry lookahead pode tornar os carries mais rápidos é que toda a lógica começa a avaliação no momento em que o ciclo de clock começa, e o resultado não mudará quando a saída de cada porta deixar de mudar. Tomando um atalho de passar por menos portas para enviar o carry no sinal, a saída das portas deixará de mudar mais cedo, e, por isso, o tempo para o somador pode ser menor.

Para apreciar a importância do carry lookahead, precisamos calcular o desempenho relativo entre ele e os somadores de carry por ondulação.

Velocidade do carry por ondulação versus carry lookahead

Exemplo

Um modo simples de modelar o tempo para a lógica é considerar que cada porta AND ou OR leva o mesmo tempo para um sinal passar por ela. O tempo é estimado simplesmente contando-se o número de portas ao longo do caminho por uma parte da lógica. Compare o número de *atrasos de porta* para os caminhos de dois somadores de 16 bits, um usando o carry por ondulação e um usando o carry lookahead em dois níveis.

Resposta

A Figura B.5.5 mostra que o sinal carry out utiliza dois atrasos de porta por bit. Depois, o número de atrasos de porta entre um carry in para o bit menos significativo e o carry out do mais significativo é $16 \times 2 = 32$.

Para o carry lookahead, o carry out do bit mais significativo é apenas C4, definido no exemplo. São necessários dois níveis de lógica para especificar C4 em termos de P_i e G_i (o OR de vários termos AND). P_i é especificado em um nível de lógica (AND) usando p_i , e G_i é especificado em dois níveis usando p_i e G_i , de modo que o pior caso para esse próximo nível de abstração são dois níveis de lógica. p_i e G_i são um nível de lógica cada um, definido em termos de a_i e b_i . Se assumirmos que um atraso de porta para cada nível de lógica nessas equações, o pior caso é $2 + 2 + 1 = 5$ atrasos de porta.

Logo, para o caminho de carry in para carry out, a adição de 16 bits por um somador carry lookahead é seis vezes mais rápida, usando essa estimativa muito simples da velocidade do hardware.

Resumo

O carry lookahead oferece um caminho mais rápido do que esperar que os carries ondulem por todos os 32 somadores de 1 bit. Esse caminho mais rápido é pavimentado por dois sinais, gerar e propagar. O primeiro cria um carry independente da entrada do carry, e o outro passa um carry adiante. O carry lookahead também oferece outro exemplo de como a abstração é importante no projeto de computadores para lidar com a complexidade.

Verifique você mesmo

Usando a estimativa simples da velocidade de hardware com atrasos de porta, qual é o desempenho relativo de uma adição de 8 bits com carry por ondulação *versus* uma adição de 64 bits usando a lógica de carry lookahead?

1. Um somador de 64 bits com carry lookahead é três vezes mais rápido: adições de 8 bits possuem atrasos de 16 portas, e adições de 64 bits possuem atrasos de 7 portas.
2. Elas têm praticamente a mesma velocidade, pois adições de 64 bits precisam de mais níveis de lógica no somador de 16 bits.
3. Adições de 8 bits são mais rápidas do que 64 bits, mesmo com carry lookahead.

Detalhamento

Agora, consideramos todas, menos uma das operações lógicas e aritméticas para o conjunto de instruções principal do MIPS: a ALU na Figura B.5.14 omite o suporte às instruções de deslocamento. Seria possível ampliar o multiplexador da ALU para incluir um deslocamento à esquerda de 1 bit ou um deslocamento à direita de 1 bit. Mas os projetistas de hardware criaram um circuito chamado *barrel shifter*, que pode deslocar de 1 a 32 bits não em mais tempo do que é preciso para somar dois números de 32 bits, de modo que o deslocamento normalmente é feito fora da ALU.

Detalhamento

A equação lógica para a saída de Sum do somador completo na página B-28 pode ser expressa de forma mais simples usando uma porta mais poderosa do que AND e OR. Uma porta *OR exclusiva* é verdadeira se os dois operandos

divergirem, ou seja,

$$x \neq y \Rightarrow 1 \text{ and } x = y \Rightarrow 0$$

Em algumas tecnologias, o OR exclusivo é mais eficiente do que dois níveis de portas AND e OR. Usando o símbolo \oplus para representar o OR exclusivo, aqui está a nova equação:

$$\text{Sum} = a \oplus b \oplus \text{CarryIn}$$

Além disso, desenhamos a ALU da maneira tradicional, usando portas. Os computadores são projetados hoje em transistores CMOS, que são basicamente chaves. A ALU CMOS e os barrel shifters tiram proveito dessas chaves e podem ter menos multiplexadores do que aparece em nossos projetos, mas os princípios de projeto são semelhantes.

Detalhamento

Usar minúsculas e maiúsculas para distinguir a hierarquia de símbolos de geração e propagação não funciona quando você tem mais de dois níveis. Uma notação alternativa que funciona é $g_{i..j}$ e $p_{i..j}$ para os sinais de geração e propagação para os bits de i a j . Assim, $g_{1..1}$ é a geração para o bit 1, $g_{4..1}$ é a geração para os bits de 4 a 1, e $g_{16..1}$ é para os bits de 16 a 1.

B.7. Clocks

Antes de discutirmos sobre elementos de memória e lógica sequencial, é útil discutir brevemente o tópico de clocks. Esta seção curta introduz o assunto e é semelhante à discussão encontrada na Seção 4.2. Outros detalhes sobre metodologias de clock e temporização são apresentados na [Seção B.11](#).

Clocks são necessários na lógica sequencial para decidir quando um elemento que contém estado deve ser atualizado. Um clock é simplesmente um sinal de execução livre com um *tempo de ciclo* fixo; a *frequência de clock* é simplesmente o inverso do tempo de ciclo. Como vemos na [Figura B.7.1](#), o *tempo de ciclo de clock* ou *período de clock* é dividido em duas partes: quando o clock é alto e quando o clock é baixo. Neste texto, usamos apenas o **clocking acionado por transição**. Isso significa que todas as mudanças de estado ocorrem em uma transição do clock. Usamos uma metodologia acionada por transição porque é mais simples de explicar. Dependendo da tecnologia, essa pode ou não ser a melhor escolha para uma **metodologia de clocking**.

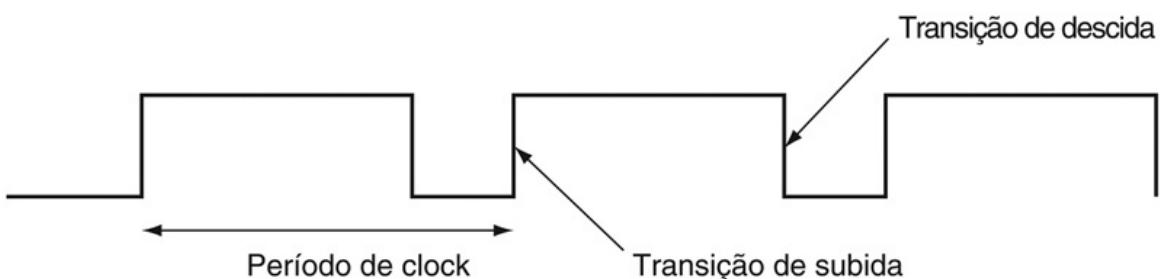


FIGURA B.7.1 Um sinal de clock oscila entre valores alto e baixo.

O período de clock é o tempo para um ciclo completo. Em um projeto acionado por transição, a transição de subida ou descida do clock é ativa e faz com que o estado seja alterado.

clocking acionado por transição

Um esquema de clocking em que todas as mudanças de estado ocorrem em uma transição do clock.

metodologia de clocking

A técnica usada para determinar quando os dados são válidos e estáveis em relação ao clock.

Em uma metodologia acionada por transição, a transição de subida ou a transição de descida do clock é *ativa* e faz com que haja mudanças de estado. Como veremos na próxima seção, os **elementos de estado** em um projeto acionado por transição são implementados de modo que o conteúdo dos elementos de estado só mudem na transição de clock ativa. A escolha da transição que será ativa é influenciada pela tecnologia de implementação e não afeta os conceitos envolvidos no projeto da lógica.

elementos de estado

Um elemento de memória.

A transição do clock atua como um sinal de amostragem, fazendo com que o valor da entrada de dados para um elemento de estado seja amostrado e armazenado no elemento de estado. O uso de um acionador por transição significa que o processo de amostragem é essencialmente instantâneo, eliminando problemas que poderiam ocorrer se os sinais fossem amostrados em momentos ligeiramente diferentes.

A principal restrição em um sistema com clock, também chamado de **sistema síncrono**, é que os sinais escritos nos elementos de estado precisam ser *válidos* quando ocorre a transição de clock ativa. Um sinal é válido se ele for estável (ou seja, não muda), e o valor não mudará novamente até as entradas mudarem. Como os circuitos combinacionais não podem ter feedback, se as entradas de uma unidade lógica combinacional não forem alteradas, as saídas por fim se tornarão válidas.

sistema síncrono

Um sistema de memória que emprega clocks e onde os sinais de dados são lidos apenas quando o clock indicar que os valores de sinal são estáveis.

A [Figura B.7.2](#) mostra a relação entre os elementos de estado e os blocos lógicos combinacionais em um projeto lógico síncrono, sequencial. Os

elementos de estado, cujas saídas mudam apenas depois da transição do clock, oferecem entradas válidas ao bloco de lógica combinacional. Para garantir que os valores escritos nos elementos de estado na transição de clock ativa sejam válidos, o clock precisa ter um período longo o suficiente para que todos os sinais no bloco de lógica combinacional se estabilizem, depois a transição do clock pega esses valores para armazenar nos elementos de estado. Essa restrição define um limite inferior sobre o tamanho do período de clock, que precisa ser longo o suficiente para que todas as entradas de elementos de estado sejam válidas.

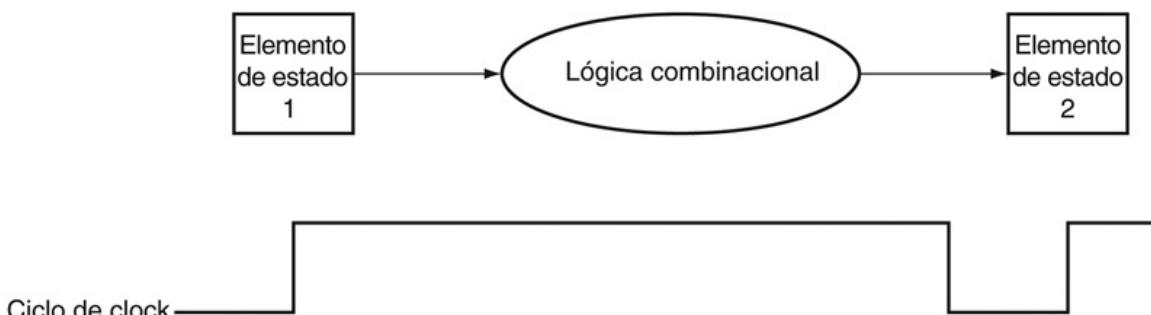


FIGURA B.7.2 As entradas de um bloco de lógica combinacional vêm de um elemento de estado, e as saídas são escritas em um elemento de estado.

A transição do clock determina quando o conteúdo dos elementos de estado são atualizados.

No restante desse apêndice, bem como no [Capítulo 4](#), normalmente omitimos o sinal de clock, pois estamos supondo que todos os elementos de estado são atualizados na mesma transição de clock. Alguns elementos de estado serão escritos em cada transição de clock, enquanto outros serão escritos apenas sob certas condições (como um registrador sendo atualizado). Nesses casos, teremos um sinal de escrita explícito para esse elemento de estado. O sinal de escrita ainda precisa ser disparado com o clock para que a atualização ocorra apenas na transição do clock se o sinal de escrita estiver ativo. Veremos como isso é feito e utilizado na próxima seção.

Outra vantagem de uma metodologia acionada por transição é a possibilidade de ter um elemento de estado utilizado como entrada e saída para o mesmo bloco de lógica combinacional, como vemos na [Figura B.7.3](#). Na prática, deve-se ter o cuidado de impedir corridas (“races”) em tais situações e garantir que o período de clock seja longo o suficiente; esse assunto é discutido com maior

profundidade na Seção B.11.

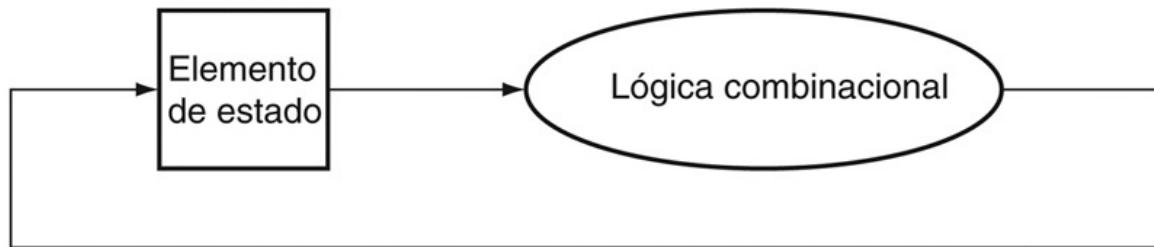


FIGURA B.7.3 Uma metodologia acionada por transição permite que um elemento de estado seja lido e escrito no mesmo ciclo de clock sem criar uma corrida que pudesse levar a valores indeterminados nos dados.

Naturalmente, o ciclo de clock ainda precisa ser grande o suficiente para que os valores de entrada sejam estáveis quando ocorrer a transição de clock ativa.

Agora que discutimos como o clocking é utilizado para atualizar elementos de estado, podemos discutir como construir os elementos de estado.

Detalhamento

Ocasionalmente, os projetistas acham útil ter uma pequena quantidade de elementos de estado que mudam na transição de clock oposta a partir da maioria dos elementos de estado. Isso exige cuidado extremo, pois tal técnica tem efeitos sobre as entradas e as saídas do elemento de estado. Por que, então, os projetistas fariam isso? Considere o caso em que a quantidade de lógica combinacional antes e depois de um elemento de estado é pequena o suficiente, de modo que cada uma poderia operar em meio ciclo de clock, em vez de um ciclo de clock completo, que é mais comum. Então, o elemento de estado pode ser escrito na transição de clock correspondente a meio ciclo de clock, pois as entradas e saídas serão utilizáveis após meio ciclo de clock. Um lugar comum onde essa técnica é usada é em **bancos de registradores**, onde a simples leitura ou escrita do banco de registradores normalmente pode ser feita em metade do ciclo de clock normal. O Capítulo 4 utiliza essa ideia para reduzir o overhead da técnica de pipelining.

banco de registradores

Um elemento de estado que consiste em um conjunto de registradores que podem ser lidos e escritos fornecendo um número de registrador a ser acessado.

B.8. Elementos de memória: flip-flops, latches e registradores

Nesta seção e na seguinte, discutimos os princípios básicos por trás dos elementos de memória, começando com flip-flops e latches, passando para bancos de registradores e finalmente para as memórias. Todos os elementos de memória armazenam estado: a saída de qualquer elemento da memória depende das entradas e do valor armazenado dentro do elemento da memória. Assim, todos os blocos lógicos com um elemento da memória contêm estado e são sequenciais.

Os tipos mais simples de elementos de memória são *sem clock*; ou seja, eles não possuem qualquer entrada de clock. Embora só usemos elementos de memória com clock neste texto, um latch sem clock é o elemento de memória mais simples, de modo que veremos esse circuito primeiro. A [Figura B.8.1](#) mostra um *latch S-R* (latch set-reset), construído a partir de um par de portas NOR (portas OR com saídas invertidas). As saídas Q e Q^- representam o valor do estado armazenado e seu complemento. Quando nem S nem R estão ativos, as portas NOR cruzadas, atuam como inversores e armazem os valores anteriores de Q e Q^- .

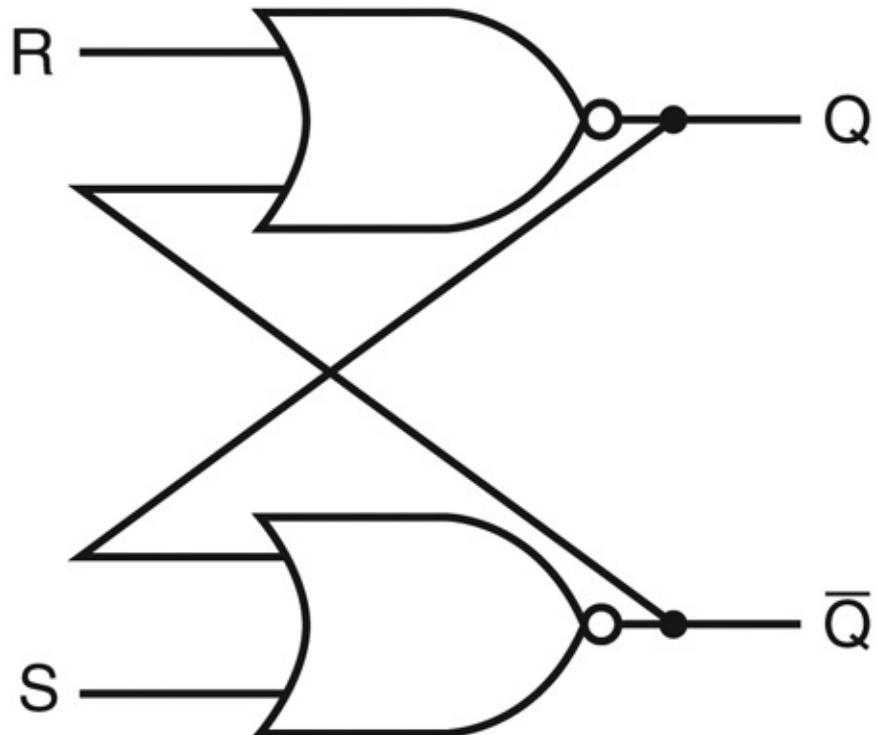


FIGURA B.8.1 Um par de portas NOR cruzadas pode armazenar um valor interno.

O valor armazenado na saída Q é reciclado invertendo-o para obter Q^- e depois invertendo Q^- para obter Q . Se R ou Q^- estiverem ativos, Q será desativado e vice-versa.

Por exemplo, se a saída, Q , for verdadeira, então o inversor de baixo produz uma saída falsa (que é Q^-), que se torna a entrada para o inversor de cima, que produz uma saída verdadeira, que é Q , e assim por diante. Se S estiver ativa, então a saída Q estará ativa e Q^- estará inativa. Quando S e R estiverem inativas, os últimos valores de Q e Q^- continuarão a ser armazenados na estrutura cruzada. A ativação de S e R simultaneamente pode levar a uma operação incorreta: dependendo de como S e R são desativadas, o latch pode oscilar ou tornar-se metaestável (isso é descrito com mais detalhes na [Seção B.11](#)).

Essa estrutura cruzada é a base para elementos de memória mais complexos, que nos permitem armazenar sinais de dados. Esses elementos contêm portas adicionais, usadas para armazenar valores de sinal e fazem com que o estado seja atualizado apenas em conjunto com um clock. A próxima seção mostra como esses elementos são criados.

Flip-flops e latches

Flip-flops e latches são os elementos de memória mais simples. Em flip-flops e latches, a saída é igual ao valor do estado armazenado dentro do elemento. Além do mais, diferente do latch S-R descrito anteriormente, todos os latches e flip-flops que usaremos neste ponto em diante são com clock, o que significa que eles têm uma entrada de clock e a mudança de estado é acionada por esse clock. A diferença entre um flip-flop e um latch é o ponto em que o clock faz com que o estado realmente mude. Em um latch com clock, o estado é alterado sempre que as entradas apropriadas mudam e o clock está ativo, enquanto, em um flip-flop, o estado é trocado somente em uma transição de clock. Como em todo este texto usamos uma metodologia de temporização acionada por transição, onde o estado é atualizado apenas em transições de clock, só precisamos usar flip-flops. Os flip-flops normalmente são criados a partir de latches, de modo que começamos descrevendo a operação de um latch com clock simples e depois discutimos a operação de um flip-flop construído a partir desse latch.

flip-flop

Um elemento da memória para o qual a saída é igual ao valor do estado armazenado dentro do elemento e para o qual o estado interno é alterado apenas em uma transição do clock.

latch

Um elemento da memória em que a saída é igual ao valor do estado armazenado dentro do elemento e o estado é alterado sempre que as entradas apropriadas mudarem e o clock estiver ativo.

Para aplicações computacionais, a função de flip-flops e latches é armazenar um sinal. Um *latch D* ou **flip-flop D** armazena o valor de seu sinal de entrada de dados na memória interna. Embora existam muitos outros tipos de latches e flip-flops, o tipo D é o único bloco de montagem básico de que precisaremos. Um latch D possui duas entradas e duas saídas. As entradas são o valor de dados a ser armazenado (chamado *D*) e um sinal de clock (chamado *C*) que indica quando o latch deve ler o valor na entrada *D* e armazená-lo. As saídas são simplesmente o valor do estado interno (*Q*) e seu complemento (*Q̄*). Quando a entrada de clock *C* está ativa, o latch é considerado *aberto*, e o valor da saída (*Q*) torna-se o valor da entrada *D*. Quando a entrada do clock *C* está inativa, o latch é considerado *fechado*, e o valor da saída (*Q*) é qualquer valor que tenha sido

armazenado pela última vez em que o latch foi aberto.

flip-flop D

Um flip-flop com uma entrada de dados que armazena o valor desse sinal de entrada na memória interna quando ocorre transição do clock.

A [Figura B.8.2](#) mostra como um latch D pode ser implementado com duas portas adicionais acrescentadas às portas NOR cruzadas. Visto que, quando o latch é aberto, o valor de Q muda quando D muda, essa estrutura, às vezes, é chamada de *latch transparente*. A [Figura B.8.3](#) mostra como esse latch D funciona, supondo que a saída Q seja inicialmente falsa e que D mude primeiro.

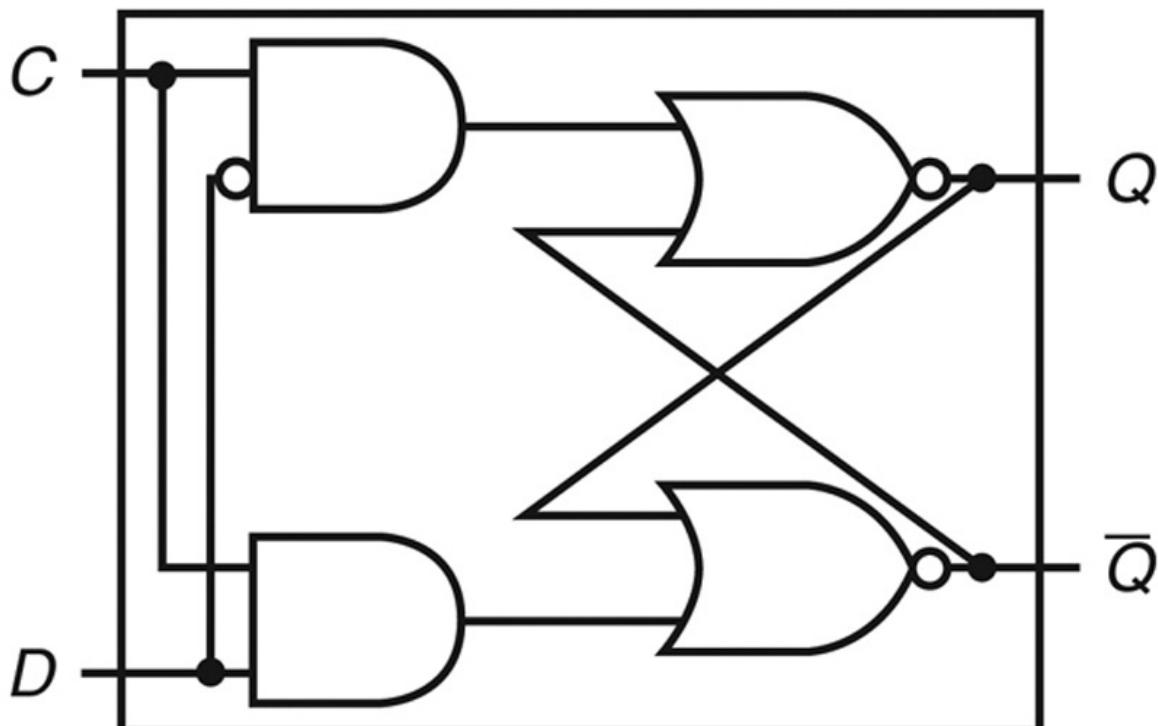


FIGURA B.8.2 Um latch D implementado com portas NOR.

Uma porta NOR atua como um inversor se a outra entrada for 0. Assim, o par cruzado de portas NOR atua para armazenar o valor de estado, a menos que a entrada do clock C , esteja ativa, quando o valor da entrada D substitui o valor de Q e é armazenado. O valor da entrada D precisa ser estável quando o sinal de clock C mudar de ativo para inativo.

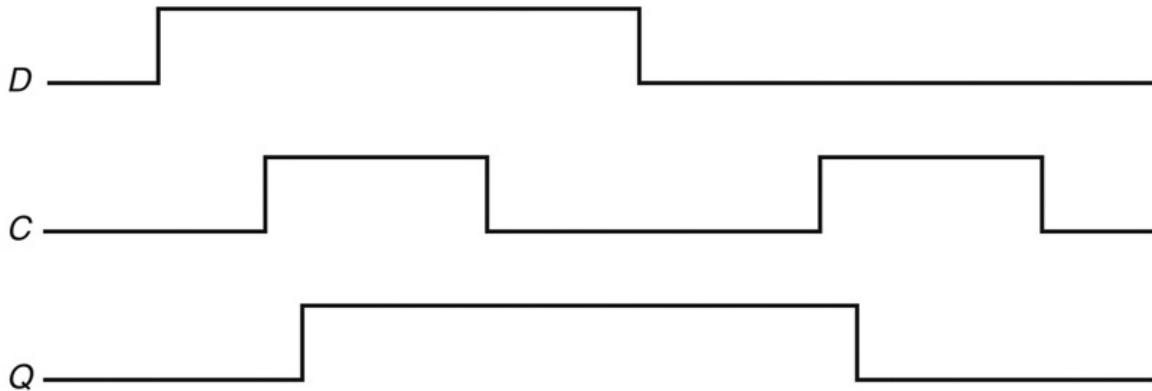


FIGURA B.8.3 Operação de um latch D, assumindo que a saída está inicialmente inativa.

Quando o clock C, está ativo, o latch é aberto e a saída Q imediatamente assume o valor da entrada D.

Como já dissemos, usamos flip-flops como bloco de montagem básico, no lugar de latches. Os flip-flops não são transparentes: suas saídas mudam *apenas* na transição do clock. Um flip-flop pode ser construído de modo que seja acionado na transição de subida (positivo) ou descida (negativo) do clock; para nossos projetos, podemos usar qualquer tipo. A [Figura B.8.4](#) mostra como um flip-flop D de transição de descida é construído, a partir de um par de latches D. Em um flip-flop D, a saída é armazenada quando ocorre a transição do clock. A [Figura B.8.5](#) mostra como opera esse flip-flop.

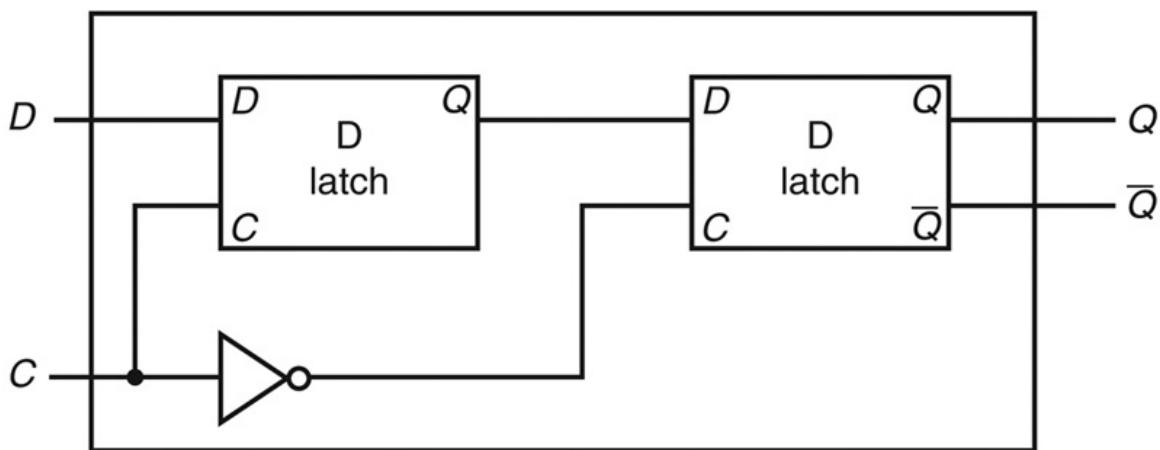


FIGURA B.8.4 Um flip-flop D acionado na transição de descida.

O primeiro latch, chamado de mestre, é aberto e acompanha a entrada D quando a entrada de clock C, é ativada. Quando a entrada de clock C cai, o primeiro latch é fechado, mas o

segundo latch, chamado de escravo, é aberto e recebe sua entrada da saída do latch mestre.

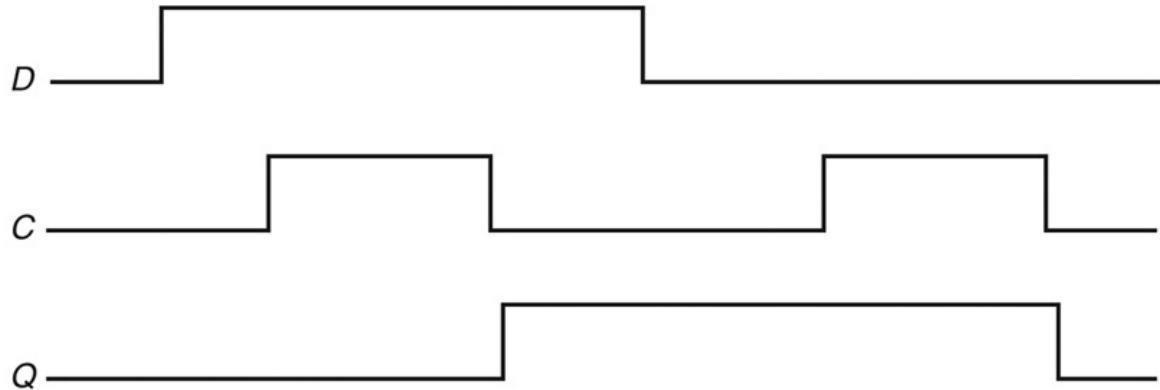


FIGURA B.8.5 Operação de um flip-flop D acionado na transição de descida, assumindo que a saída está inicialmente inativa.

Quando a entrada de clock C muda de ativa para inativa, a saída Q armazena o valor da entrada D. Compare esse comportamento com o do latch D com clock, mostrado na [Figura B.8.3](#). Em um latch com clock, o valor armazenado e a saída, Q, mudam sempre que C está alto, ao contrário de somente quando C realiza transições.

Aqui está uma descrição em Verilog de um módulo para um flip-flop D na transição de subida, assumindo que C é a entrada de clock e D é a entrada de dados:

```

module DFF(clock,D,Q,Qbar);
    input clock, D;
    output reg Q; // Q é um reg, pois é atribuído em um
bloco always
    output Qbar;
    assign Qbar = ~ Q; // Qbar é sempre exatamente o
inverso de Q
    always @(posedge clock) // realiza ações sempre que
o clock levanta
        Q = D;
endmodule

```

Como a entrada D é pega na transição do clock, ela precisa ser válida por um período imediatamente antes e depois da transição do clock. O tempo mínimo que a entrada precisa ser válida, antes da transição do clock, é chamado **tempo de preparação**; o tempo mínimo durante o qual ela precisa ser válida, após a transição do clock, é chamado **tempo de suspensão**. Assim, as entradas de qualquer flip-flop (ou qualquer coisa construída usando flip-flops) precisam ser válidas durante uma janela que começa no tempo de preparação, antes da transição do clock e termina no tempo de suspensão, após a transição do clock, como mostra a [Figura B.8.6](#). A [Seção B.11](#) fala sobre as restrições de clocking e temporização, incluindo o atraso de propagação por um flip-flop, com mais detalhes.

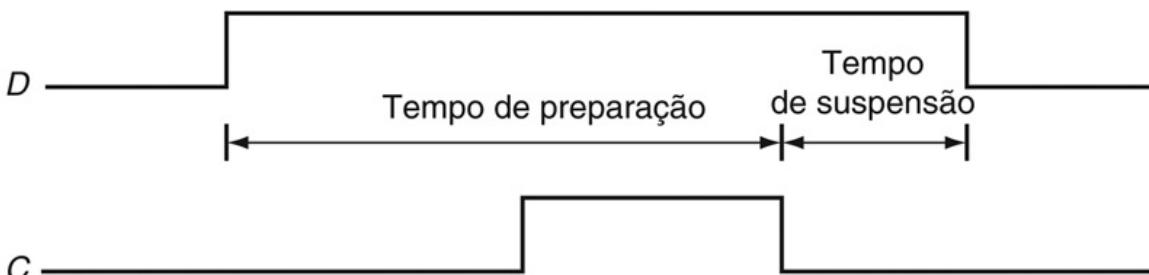


FIGURA B.8.6 Requisitos de tempo de preparação e suspensão para um flip-flop D acionado na transição de descida.

A entrada precisa ser estável por um período antes da transição do clock, bem como após a transição do clock. O tempo mínimo que o sinal precisa estar estável antes da transição de clock é chamado de tempo de preparação, enquanto o tempo mínimo

que o sinal precisa estar estável após o clock é chamado de tempo de suspensão. Deixar de atender a esses requisitos mínimos pode resultar em uma situação em que a saída do flip-flop pode nem sequer ser previsível, conforme descrevemos na [Seção B.11](#). Os tempos de suspensão normalmente são 0 ou muito pequenos e, portanto, não causam preocupação.

tempo de preparação

O tempo mínimo que a entrada para um dispositivo de memória precisa ser válida, antes da transição do clock.

tempo de suspensão

O tempo mínimo durante o qual a entrada precisa ser válida, após a transição do clock.

Podemos usar um array de flip-flops D para construir um registrador que possa manter um dado de múltiplos bits, como um byte ou uma word. Usamos registradores por todos os nossos caminhos de dados no [Capítulo 4](#).

Bancos de registradores

Uma estrutura central no nosso caminho de dados é um *banco de registradores*. Ele consiste em um conjunto de registradores que podem ser lidos e escritos fornecendo um número de registrador a ser acessado. Um banco de registradores pode ser implementado com um decodificador para cada porta de leitura ou escrita e um array de registradores construídos a partir de flip-flops D. Como a leitura de um registrador não muda qualquer estado, só precisamos fornecer um número de registrador como entrada, e a única saída será os dados contidos nesse registrador. Para escrever em um registrador, precisaremos de três entradas: um número de registrador, os dados a escrever e um clock que controla a escrita no registrador. No [Capítulo 4](#), usamos um banco de registradores com duas portas de leitura e uma porta de escrita. Esse banco de registradores é desenhado como mostra a [Figura B.8.7](#). As portas de leitura podem ser implementadas com um par de multiplexadores, cada um tão largo quanto o número de bits em cada registrador do banco de registradores. A [Figura B.8.8](#) mostra a implementação de duas portas de leitura de registrador para um banco de registradores de 32

bits.

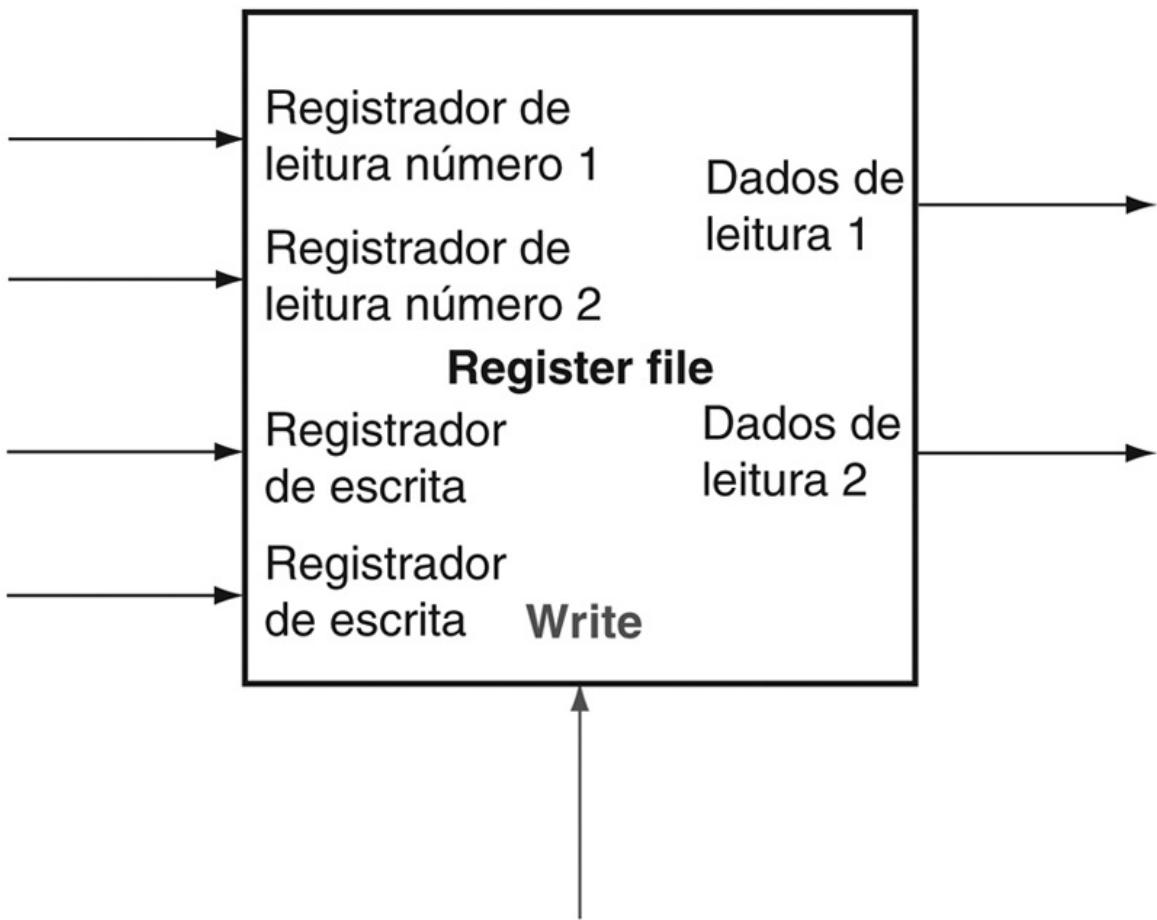


FIGURA B.8.7 Um banco de registradores com duas portas de leitura e uma porta de escrita possui cinco entradas e duas saídas.

A entrada de controle Write aparece em destaque na parte inferior.

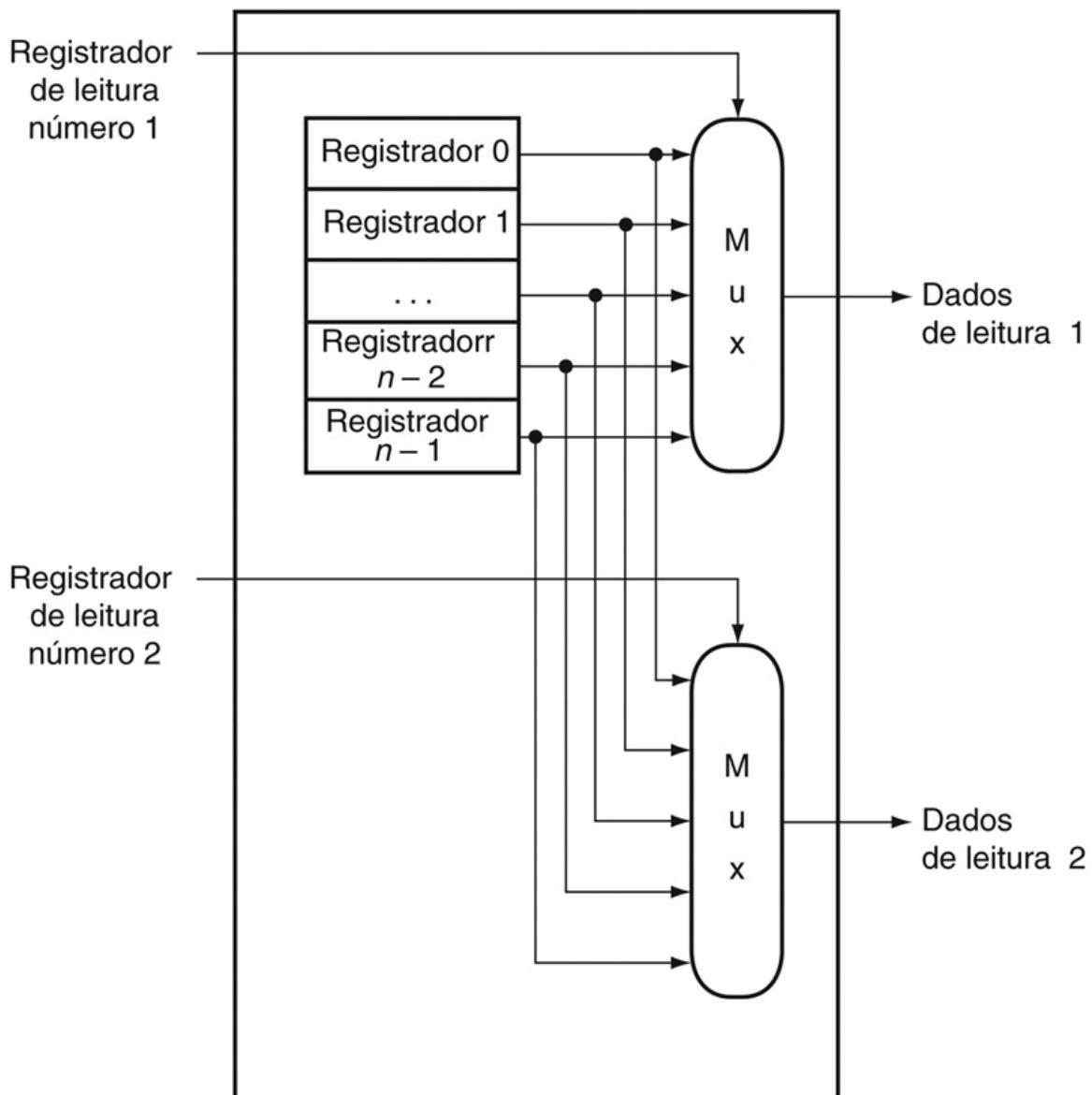


FIGURA B.8.8 A implementação de duas portas de leitura para um banco de registradores com n registradores pode ser feita com um par de multiplexadores n-para-1, cada um com 32 bits de largura.

O sinal “Registrador de leitura número” é usado como sinal seletor do multiplexador. A [Figura B.8.9](#) mostra como a porta de escrita é implementada.

A implementação da porta de escrita é ligeiramente mais complexa, pois só podemos mudar o conteúdo do registrador designado. Podemos fazer isso usando um decodificador para gerar um sinal que possa ser usado para determinar qual registrador escrever. A [Figura B.8.9](#) mostra como implementar a

porta de escrita para um banco de registradores. É importante lembrar que o flip-flop só muda de estado na transição do clock. No [Capítulo 4](#), conectamos sinais de escrita para o banco de registradores explicitamente e assumimos que o clock mostrado na [Figura B.8.9](#) está anexado implicitamente.

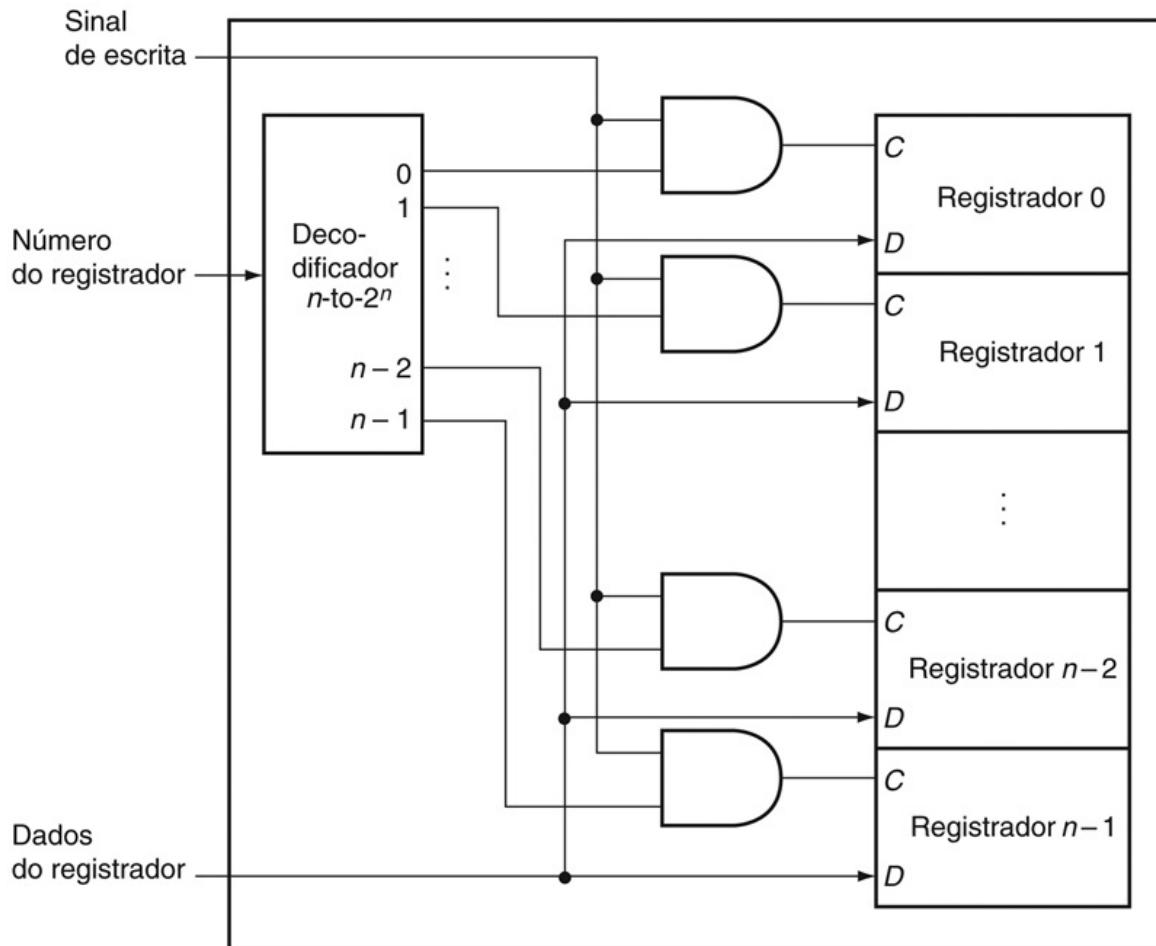


FIGURA B.8.9 A porta de escrita para um banco de registradores é implementada com um decodificador que é usado com o sinal de escrita (Write) para gerar a entrada C dos registradores.

Todas as três entradas (o número do registrador, os dados e o sinal de escrita) terão restrições de tempo de preparação e suspensão que garantem que os dados corretos são escritos no banco de registradores.

O que acontece se o mesmo registrador for lido e escrito durante um ciclo de clock? Como a escrita no banco de registradores ocorre na transição do clock, o

registrador será válido durante o tempo em que for lido, como vimos anteriormente na [Figura B.7.2](#). O valor retornado será o valor escrito em um ciclo de clock anterior. Se quisermos que uma leitura retorne o valor atualmente sendo escrito, uma lógica adicional no banco de registradores ou fora dele será necessária. O [Capítulo 4](#) utiliza muito dessa lógica.

Especificando a lógica sequencial em Verilog

Para especificar a lógica sequencial em Verilog, temos de entender como gerar um clock, como descrever quando um valor é escrito em um registrador e como especificar o controle sequencial. Vamos começar especificando um clock. Um clock não é um objeto predefinido em Verilog; em vez disso, geramos um clock usando a notação `#n` da Verilog antes de uma instrução; isso causa um atraso de n etapas de tempo de simulação antes da execução da instrução. Na maioria dos simuladores Verilog, também é possível gerar um clock como uma entrada externa, permitindo que o usuário especifique em tempo de simulação o número de ciclos de clock para executar uma simulação.

O código na [Figura B.8.10](#) implementa um clock simples que é alto ou baixo para uma unidade de simulação e depois muda de estado. Usamos a capacidade de atraso e atribuição bloqueante para implementar o clock.

```
reg clock; // clock é um registrador
always
#1 clock = 1; #1 clock = 0;
```

FIGURA B.8.10 Uma especificação de um clock.

Em seguida, precisamos ser capazes de especificar a operação de um registrador acionado por transição. Em Verilog, isso é feito usando a lista de sensitividade em um bloco `always` e especificando como acionador a transição positiva ou negativa de uma variável binária com a notação `posedge` ou `negedge`, respectivamente. Logo, o código Verilog a seguir, faz com que o registrador A seja escrito com o valor b na transição positiva do clock:

No decorrer deste capítulo e nas seções Verilog do [Capítulo 4](#), consideraremos um projeto acionado por transição positiva. A [Figura B.8.11](#) mostra uma especificação Verilog de um banco de registradores MIPS que considera duas

leituras e uma escrita, com apenas a escrita possuindo clock.

```
reg [31:0] A;
wire [31:0] b;

always @(posedge clock) A <= b;

module registerfile (Read1,Read2,WriteReg,WriteData,RegWrite,
Data1,Data2,clock);
    input [5:0] Read1,Read2,WriteReg; // os números dos registradores
    para leitura ou escrita
    input [31:0] WriteData; // dados para escrever
    input RegWrite, // O controle de escrita
    clock; // o clock para acionar a escrita
    output [31:0] Data1, Data2; // os valores de registradores lidos
    reg [31:0] RF [31:0]; // 32 registradores cada um com 32 bits

    assign Data1 = RF[Read1];
    assign Data2 = RF[Read2];

    always begin
        // escreve no registrador o novo valor se Regwrite
        for alto
            @(posedge clock) if (RegWrite) RF[WriteReg] <=
        WriteData;
        end
    endmodule
```

FIGURA B.8.11 Um banco de registradores MIPS escrito em Verilog comportamental.

Esse banco de registradores escreve na transição de subida do clock.

Verifique você mesmo

No código Verilog para o banco de registradores da Figura B.8.11, as portas de saída correspondentes aos registradores lidos são atribuídas por meio de uma atribuição contínua, mas o registrador sendo escrito é atribuído em um bloco always. Qual dos seguintes é o motivo?

- a. Não existe um motivo especial. Isso simplesmente foi conveniente.
- b. Porque Data1 e Data2 são portas de saída e WriteData é uma porta de entrada.
- c. Porque a leitura é um evento combinacional, enquanto a escrita é um evento sequencial.

B.9. Elementos de memória: SRAMs e DRAMs

Registradores e bancos de registradores oferecem os blocos de montagem básicos para pequenas memórias, mas quantidades maiores de memória são construídas usando **SRAMs (Static Random Access Memories)** ou **DRAMs** (Dynamic Random Access Memories). Primeiro, discutimos sobre SRAMs, que são mais simples, e depois passamos para DRAMs.

SRAM (Static Random Access Memory)

Uma memória na qual os dados são armazenados estaticamente (como nos flip-flops), e não dinamicamente (como na DRAM). SRAMs são mais rápidas do que as DRAMs, mas menos densas e mais caras por bit.

SRAMs

SRAMs são simplesmente circuitos integrados que são arrays de memória com (normalmente) uma única porta de acesso que pode ser de leitura ou escrita. SRAMs possuem um tempo de acesso fixo a qualquer dado, embora as características de leitura e escrita geralmente sejam diferentes. Um chip de SRAM possui uma configuração específica em termos do número de locais endereçáveis, bem como a largura de cada local endereçável. Por exemplo, uma SRAM de $4M \times 8$ oferece 4M entradas, cada uma com 8 bits. Assim, ela terá 22 linhas de entrada (pois $4M = 2^{22}$), uma linha de saída de dados de 8 bits, e uma única linha de entrada de dados de 8 bits. Assim como as ROMs, o número de locais endereçáveis é chamado de *altura*, com o número de bits por unidade chamado de *largura*. Por diversos motivos técnicos, as SRAMs mais novas e mais rápidas normalmente estão disponíveis em configurações estreitas: $\times 1$ e $\times 4$. A [Figura B.9.1](#) mostra os sinais de entrada e saída para uma SRAM de $2M \times 16$.

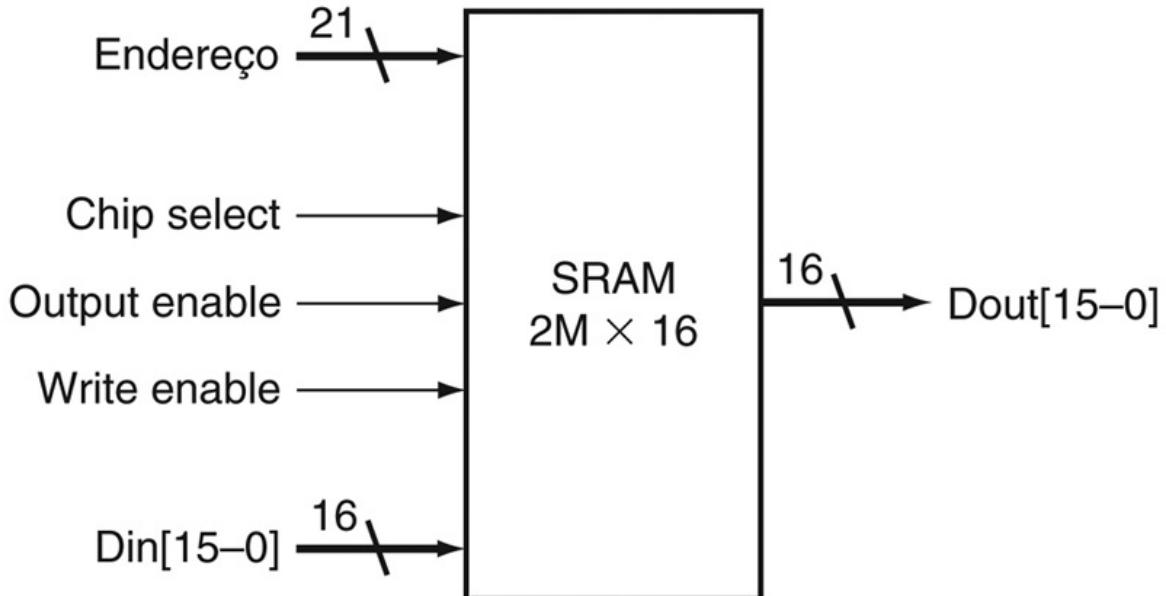


FIGURA B.9.1 Uma SRAM de $2M \times 16$ mostrando as 21 linhas de endereço ($2M = 2^{21}$) e 16 entradas de dados, as três linhas de controle e as 16 saídas de dados.

Para iniciar um acesso de leitura ou escrita, o sinal Chip select precisa ser ativado. Para leituras, também temos de ativar o sinal Output enable que controla se o dado selecionado pelo endereço foi levado para os pinos. O Output enable é útil para conectar várias memórias a um barramento de saída único e usar Output enable para determinar qual memória é levada ao barramento. O tempo de acesso de leitura da SRAM costuma ser especificado como o espaço de tempo entre o Output enable ser verdadeiro e as linhas de endereço estarem válidas até o momento em que os dados estão nas linhas de saída. Os tempos de acesso de leitura típicos para SRAMs em 2004 variavam desde 2-4ns para as partes mais rápidas da CMOS, que costumam ser um pouco menores e mais estreitas, até 8-20ns para as partes maiores típicas, que em 2004 tinham mais de 32 milhões de bits de dados. A demanda por SRAMs de menor potência, para produtos eletrônicos e aparelhos digitais, cresceu bastante nos últimos cinco anos; essas SRAMs possuem potências de stand-by e acesso muito menores, mas normalmente são de 5-10 vezes mais lentas. Mais recentemente, as SRAMs síncronas — semelhantes às DRAMs síncronas, que discutimos na próxima seção — também têm sido desenvolvidas.

Para a escrita, temos de fornecer os dados a serem escritos e o endereço, além dos próprios sinais que causarão a escrita. Quando Write enable e Chip select são verdadeiros, os dados nas linhas de entrada de dados são escritos na célula

especificada pelo endereço. Existem requisitos de tempo de preparação e tempo de suspensão para as linhas de endereço e dados, assim como existiam para flip-flops D e latches. Além disso, o sinal Write enable não é uma transição de clock, mas um pulso com um requisito de largura mínima. O tempo para concluir uma escrita é especificado pela combinação entre tempos de preparação, tempos de suspensão e a largura do pulso Write enable.

SRAMs grandes não podem ser construídas da mesma maneira como construímos um banco de registradores porque, diferente de um banco de registradores, no qual um multiplexador 32 para 1 poderia ser prático, o multiplexador de 64K para 1 que seria necessário para uma SRAM de $64K \times 1$ seria totalmente inviável. Em vez de usar um multiplexador gigante, memórias grandes são implementadas com uma linha de saída compartilhada, chamada *linha de bit*, que múltiplas células de memória no array de memória podem ativar. Para permitir que múltiplas origens possam colocar um sinal em uma única linha, é utilizado um *buffer tristate* (ou *buffer de três estados*). Um buffer tristate possui duas entradas — um sinal de dados e um Output enable — e uma única saída que está em um dos três estados: ativa, inativa ou alta impedância. A saída de um buffer tristate é igual ao sinal de entrada de dados, ativo ou inativo, se o Output enable estiver ativo; caso contrário, está em um *estado de alta impedância*, permitindo que outro buffer tristate cujo Output enable esteja ativo determine o valor de uma saída compartilhada.

A [Figura B.9.2](#) mostra um conjunto de buffers tristate ligados para formar um multiplexador com uma entrada decodificada. É fundamental que o Output enable de, no máximo, um dos buffers tristate esteja ativo; caso contrário, os buffers tristate podem tentar colocar valores diferentes na linha de saída. Usando buffers tristate nas células individuais da SRAM, cada célula que corresponde a uma saída em particular pode compartilhar a mesma linha de saída. O uso de um conjunto de buffers tristate é incorporado nos flip-flops que formam as células básicas da SRAM. A [Figura B.9.3](#) mostra como uma SRAM pequena de 4×2 poderia ser criada, usando latches D com uma entrada chamada Enable, que controla a saída tristate.

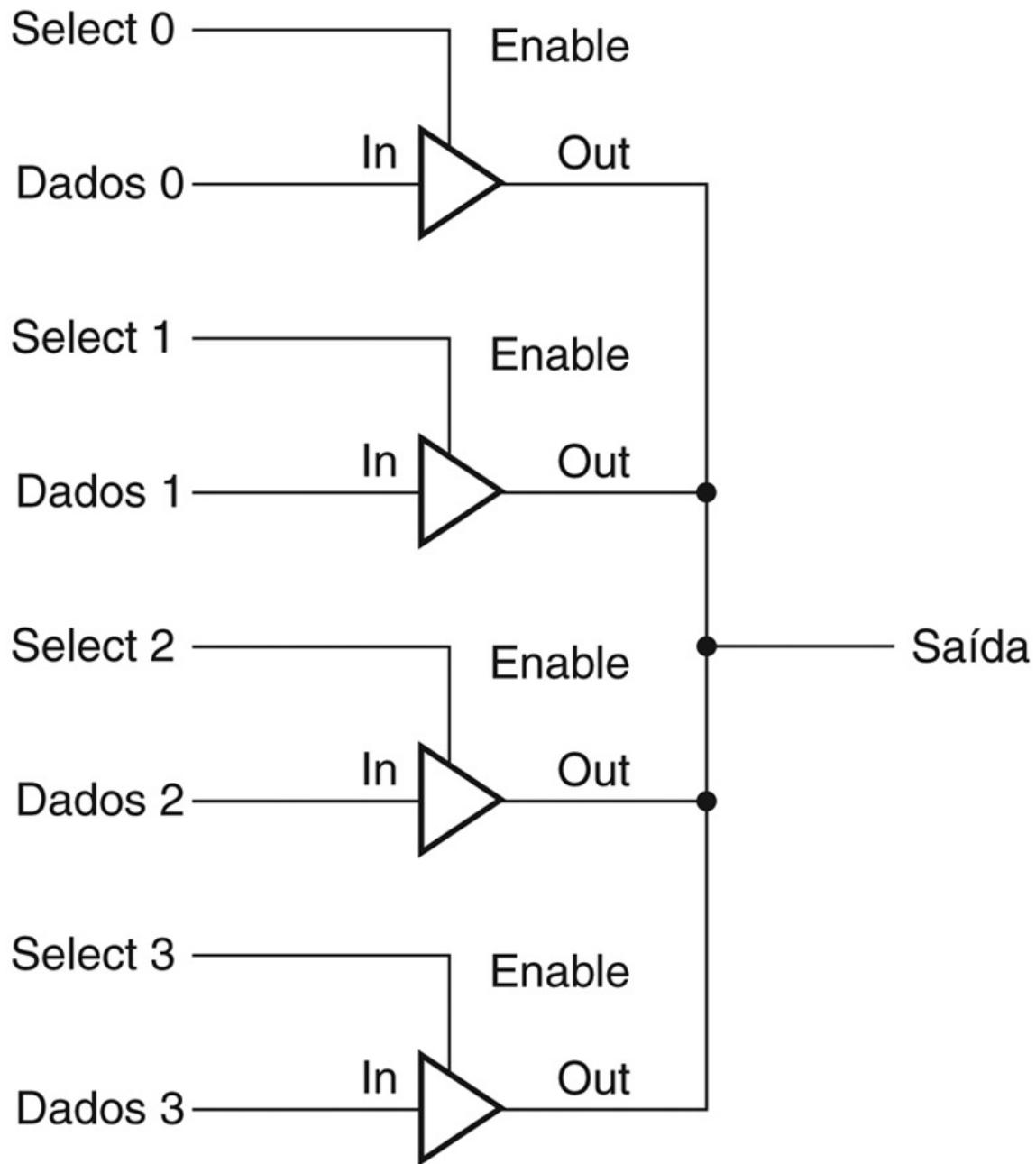


FIGURA B.9.2 Quatro buffers tristate são usados para formar um multiplexador.

Somente uma das quatro entradas Select pode estar ativa. Um buffer tristate com um Output enable inativo possui uma saída de alta impedância que permite um buffer tristate, cujo Output enable está ativo, ativar a linha de saída compartilhada.

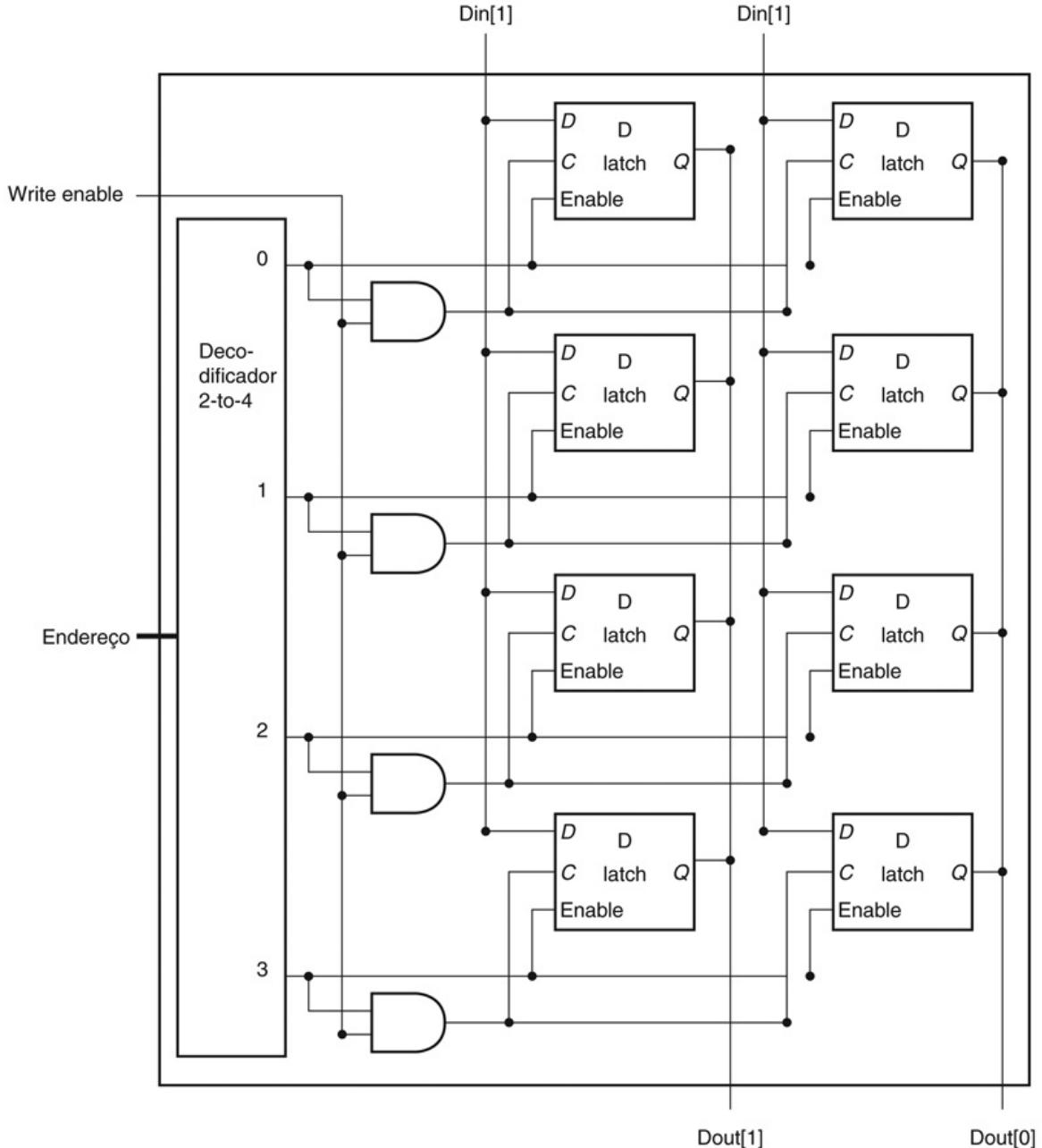


FIGURA B.9.3 A estrutura básica de uma SRAM 4×2 consiste em um decodificador que seleciona qual par de células ativar.

As células ativadas utilizam uma saída tristate conectada às linhas de bit verticais que fornecem os dados requisitados. O endereço que seleciona a célula é enviado em um de um conjunto de linhas de endereço horizontais, chamadas linhas de words. Para simplificar, os sinais Output enable e Chip select foram omitidos, mas facilmente poderiam ser incluídos com algumas portas AND.

O projeto na [Figura B.9.3](#) elimina a necessidade de um multiplexador enorme; porém, ainda exige um decodificador muito grande e um número igualmente grande de linhas de words. Por exemplo, em uma SRAM $4M \times 8$, precisaríamos de um decodificador de 2 para $4M$ e $4M$ linhas de words (que são as linhas usadas para habilitar os flip-flops individuais)! Para contornar esse problema, as memórias grandes são organizadas como arrays retangulares e utilizam um processo de decodificação em duas etapas. A [Figura B.9.4](#) mostra como uma SRAM de $4M \times 8$ poderia ser organizada internamente usando uma decodificação em duas etapas. Conforme veremos, o processo de decodificação em dois níveis é muito importante para entender a operação das DRAMs.

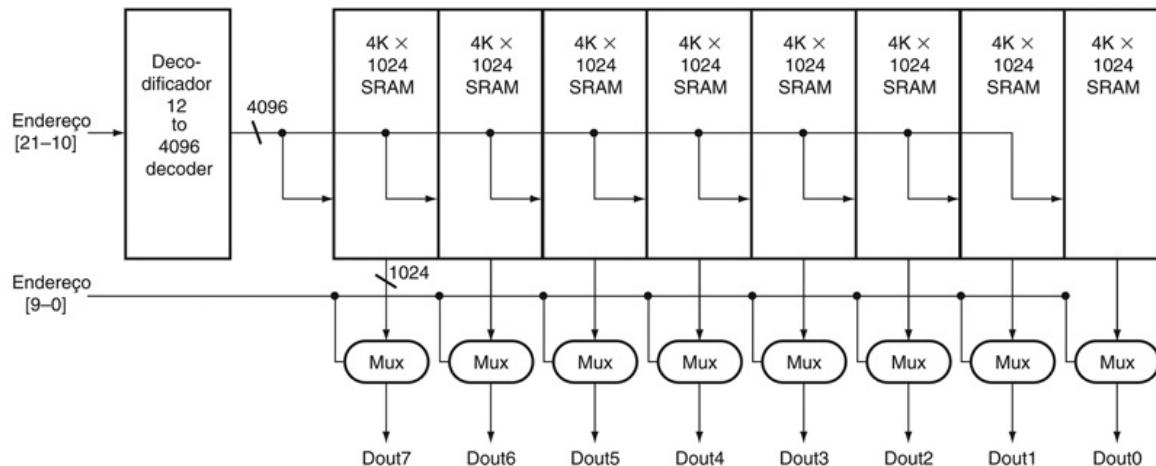


FIGURA B.9.4 Organização típica de uma SRAM $4M \times 8$ como um array de arrays $4K \times 1024$.

O primeiro decodificador gera os endereços para oito arrays $4K \times 1024$; depois, um conjunto de multiplexadores é utilizado para selecionar 1 bit de cada array de 1024 bits de largura. Esse é um projeto muito mais fácil do que um decodificador de único nível, que precisaria de um decodificador enorme ou de um multiplexador gigantesco. Na prática, uma SRAM moderna desse tamanho provavelmente usaria um número ainda maior de blocos, cada um deles um pouco menor.

Recentemente, vimos o desenvolvimento de SRAMs síncronas (SSRAMs) e DRAMs síncronas (SDRAMs). A principal capacidade oferecida pelas RAMs síncronas é a capacidade de transferir uma *rajada* de dados a partir de uma série de endereços sequenciais dentro de um array ou linha. A rajada é definida por um endereço inicial, fornecido no padrão normal, e um tamanho de rajada. A

vantagem de velocidade das RAMs síncronas vem da capacidade de transferir os bits na rajada sem ter de especificar bits de endereço adicionais. Em vez disso, um clock é usado para transferir os bits sucessivos na rajada. A eliminação da necessidade de especificar o endereço para as transferências dentro da rajada melhora bastante a taxa para transferir o bloco de dados. Devido a essa capacidade, as SRAMs e DRAMs síncronas estão rapidamente se tornando as RAMs preferidas para a criação de sistemas de memória nos computadores. Discutimos o uso de DRAMs síncronas em um sistema de memória com mais detalhes na próxima seção e no [Capítulo 5](#).

DRAMs

Em uma RAM estática (SRAM), o valor armazenado em uma célula é mantido em um par de portas inverteras e, desde que haja energia sendo aplicada, o valor pode ser mantido indefinidamente. Em uma RAM dinâmica (DRAM), o valor mantido em uma célula é armazenado como uma carga em um capacitor. Um único transistor é utilizado para acessar essa carga armazenada ou para ler o valor ou para escrever sobre a carga armazenada lá. Como as DRAMs utilizam apenas um único transistor por bit de armazenamento, eles são muito mais densos e mais baratos por bit. Em comparação, as SRAMs exigem quatro a seis transistores por bit. Nas DRAMs, a carga é armazenada em um capacitor, de modo que não pode ser mantida indefinidamente e precisa sofrer *refresh* periodicamente. É por isso que essa estrutura de memória é chamada de *dinâmica*, ao contrário do armazenamento estático em uma célula de SRAM.

Para renovar a célula, lemos seu conteúdo e o escrevemos de volta. A carga pode ser mantida por vários milissegundos, o que poderia corresponder a algo perto de um milhão de ciclos de clock. Hoje, controladores de memória de único chip normalmente tratam da função de refresh de modo independente do processador. Se cada bit tivesse de ser lido da DRAM e depois escrito de volta individualmente, com as DRAMs grandes contendo vários megabytes, estaríamos sempre realizando refresh na DRAM, sem deixar tempo para acessá-la. Felizmente, as DRAMs também usam uma estrutura de decodificação de dois níveis, e isso nos permite realizar refresh em uma linha inteira (que compartilha uma linha de words) com um ciclo de leitura seguido por um ciclo de escrita. Em geral, as operações de refresh consomem 1% a 2% dos ciclos ativos da DRAM, deixando os 98% a 99% restantes dos ciclos disponíveis para leitura e escrita de dados.

Detalhamento

Como uma DRAM lê e escreve o sinal armazenado em uma célula? O transistor dentro da célula é uma chave, chamada *transistor de passagem*, que permite que o valor armazenado no capacitor seja acessado para leitura ou escrita. A Figura B.9.5 mostra como é a célula de único transistor. O transistor de passagem atua como uma chave: quando o sinal na linha de words está ativo, a chave é fechada, conectando o capacitor à linha de bits. Se a operação for de escrita, então o valor a ser escrito é colocado na linha de bits. Se o valor for 1, o capacitor será carregado. Se o valor for 0, então o capacitor será descarregado. A leitura é um pouco mais complexa, pois a DRAM precisa detectar uma carga muito pequena armazenada no capacitor. Antes de ativar a linha de words para uma leitura, a linha de bits é carregada com uma tensão que se encontra no meio, entre a tensão baixa e alta. Depois, ativando a linha de words, a carga no capacitor é lida na linha de bits. Isso faz com que a linha de bits mude ligeiramente para a direção alta ou baixa, e essa mudança é detectada com um amplificador de sensibilidade, que pode detectar pequenas mudanças na tensão.

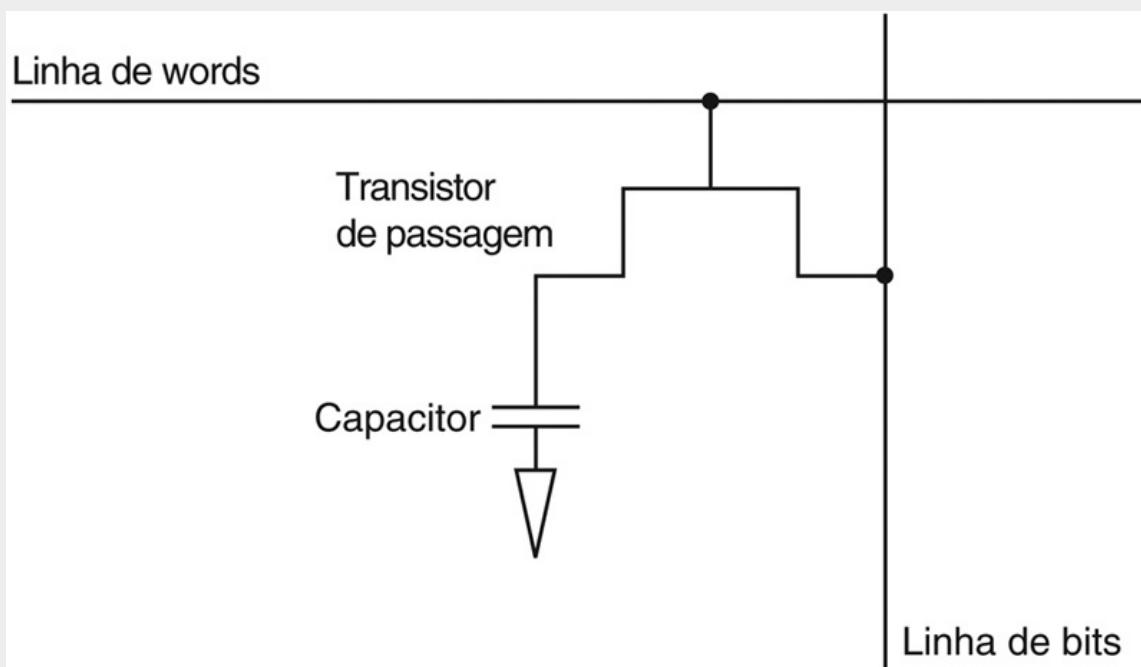


FIGURA B.9.5 Uma célula de DRAM com único transistor contém um capacitor que armazena o conteúdo da célula e um transistor usado para acessar a célula.

As DRAMs utilizam um decodificador de dois níveis consistindo em um *acesso de linha* seguido por um *acesso de coluna*, como mostra a [Figura B.9.6](#). O acesso de linha escolhe uma dentre diversas linhas e ativa a linha de words correspondente. O conteúdo de todas as colunas na linha ativa é armazenado em um conjunto de latches. O acesso à coluna, então, seleciona os dados dos latches de coluna. Para economizar pinos e reduzir o custo do pacote, as mesmas linhas de endereço são usadas para o endereço de linha e coluna; um par de sinais, chamados RAS (Row Access Strobe) e CAS (Column Access Strobe), é utilizado para sinalizar a DRAM que um endereço de linha ou coluna está sendo fornecido. O refresh é realizado simplesmente lendo as colunas nos latches de coluna e depois escrevendo os mesmos valores de volta. Assim, uma linha inteira sofre refresh em um ciclo. O esquema de endereçamento de dois níveis, combinado com o circuito interno, torna os tempos de acesso típicos da DRAM muito maiores (por um fator de 5-10) do que os tempos de acesso da SRAM. Em 2004, os tempos de acesso da DRAM típicos variavam de 45 a 65ns; DRAMs de 256 Mbits estão em plena produção, e as primeiras amostras ao cliente das DRAMs de 1GB estavam disponíveis no primeiro trimestre de 2004. Um custo muito menor por bit torna a DRAM a melhor escolha para a memória principal, enquanto o tempo de acesso mais rápido torna a SRAM a melhor escolha para as caches.

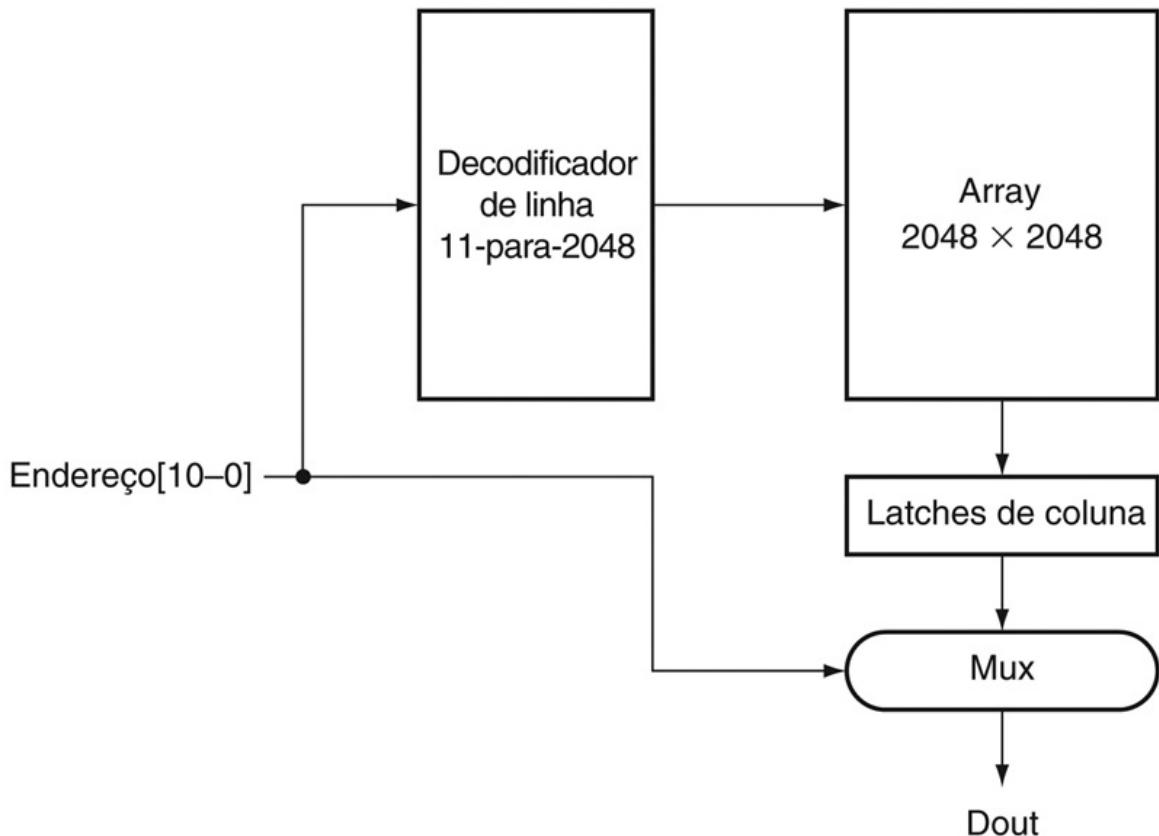


FIGURA B.9.6 Uma DRAM de $4M \times 1$ é criada com um array de 2048×2048 .

O acesso à linha usa 11 bits para selecionar uma linha, que é então guardada em 2048 latches de 1 bit. Um multiplexador escolhe o bit de saída a partir desses 2048 latches. Os sinais RAS e CAS controlam se as linhas de endereço são enviadas ao decodificador de linhas ou multiplexador de colunas.

Você poderia observar que uma DRAM de $64M \times 4$ na realidade acessa 8 kbits em cada acesso de linha e depois joga fora tudo, menos 4 deles durante um acesso de coluna. Os projetistas da DRAM utilizaram uma estrutura interna da DRAM como um meio de oferecer maior largura de banda a partir de uma DRAM. Isso é feito permitindo que o endereço de coluna mude sem mudar o endereço de linha, resultando em um acesso a outros bits nos latches de coluna. Para tornar esse processo mais rápido e mais preciso, as entradas de endereço utilizam clocks, levando à forma dominante de DRAM em uso atualmente: DRAM ou SDRAM síncrona.

Desde 1999, mais ou menos, as SDRAMs são o chip de memória preferido da maioria dos sistemas de memória principais baseados em cache. As SDRAMs oferecem acesso rápido a uma série de bits dentro de uma linha, transferindo

sequencialmente todos os bits em uma rajada sob o controle de um sinal de clock. Em 2004, as DDRRAMs (Double Data Rate RAMs), chamadas “double data rate” porque transferem dados nas transições de subida e descida de um clock fornecido externamente, eram a forma mais utilizada das SDRAMs. Conforme discutimos no [Capítulo 5](#), essas transferências de alta velocidade podem ser usadas para aumentar a largura de banda disponível a partir da memória principal para corresponder às necessidades do processador e das caches.

Correção de erros

Devido ao potencial para adulteração de dados em memórias grandes, a maioria dos sistemas computacionais utiliza algum tipo de código de verificação de erro para detectar a possível adulteração de dados. Um código simples bastante utilizado é o *código de paridade*. Em um código de paridade, o número de 1s em uma word é contado; a word tem paridade ímpar se o número de 1s for ímpar; caso contrário, a paridade é par. Quando uma word é escrita na memória, o bit de paridade também é escrito (1 para ímpar, 0 para par). Depois, quando a word é lida, o bit de paridade é lido e verificado. Se a paridade da word da memória e o bit de paridade armazenado não combinarem, então houve um erro.

Um esquema de paridade de 1 bit pode detectar, no máximo, 1 bit de erro em um item de dados; se houve 2 bits de erro, então um esquema de paridade de 1 bit não detectará qualquer erro, pois a paridade não combinará os dados com dois erros. (Na realidade, um esquema de paridade de 1 bit pode detectar qualquer número ímpar de erros; porém, a probabilidade de ter três erros é muito menor do que a probabilidade de ter dois; portanto, na prática, um código de paridade de 1 bit é limitado a detectar um único bit de erro.) Naturalmente, um código de paridade não pode dizer qual bit em um item de dados está errado.

Um esquema de paridade de 1 bit é um **código de detecção de erro**; há também *códigos de correção de erro* (ECC) que detectarão e permitirão a correção de um erro. Para grandes memórias principais, muitos sistemas utilizam um código que permite a detecção de até 2 bits de erro e a correção de um único bit de erro. Esses códigos funcionam usando mais bits para codificar os dados; por exemplo, os códigos típicos utilizados para as memórias principais exigem 7 ou 8 bits para cada 128 bits de dados.

código de detecção de erro

Um código que permite a detecção de um erro nos dados, mas não o local exato, e portanto nem a correção do erro.

Detalhamento

Um código de paridade de 1 bit é um *código de distância 2*, o que significa que, se olharmos para os dados mais o bit de paridade, nenhuma mudança de 1 bit é suficiente para gerar outra combinação válida dos dados mais a paridade. Por exemplo, se mudarmos um bit nos dados, a paridade estará errada e vice-versa. Naturalmente, se mudarmos 2 bits (ou 2 bits de dados ou 1 bit de dados e o bit de paridade), a paridade corresponderá aos dados e o erro não poderá ser detectado. Logo, existe uma distância de dois entre as combinações válidas da paridade e dos dados.

Para detectar mais de um erro ou corrigir um erro, precisamos de um *código de distância 3*, que tem a propriedade de que qualquer combinação válida dos bits no código de correção de erro e os dados ter pelo menos 3 bits diferindo de qualquer outra combinação. Suponha que tenhamos tal código e que tenhamos um erro nos dados. Nesse caso, o código mais os dados ficará a 1 bit de distância de uma combinação válida e podemos corrigir os dados para essa combinação válida. Se tivermos dois erros, podemos reconhecer que existe um erro, mas não podemos corrigir os erros. Vejamos um exemplo. Aqui estão as words de dados e um código de correção de erros de distância 3, para um item de dados de 4 bits.

Dados	Bits de código	Dados	Bits de código
0000	000	1000	111
0001	011	1001	100
0010	101	1010	010
0011	110	1011	001
0100	110	1100	001
0101	101	1101	010
0110	011	1110	100
0111	000	1111	111

Para entender como isso funciona, vamos escolher uma word de dados, digamos, 0110, cujo código de correção de erro é 011. Aqui estão as quatro possibilidades de erro de 1 bit para esses dados: 1110, 0010, 0100 e 0111. Agora veja o item de dados com o mesmo código (011), que é a entrada com

o valor 0001. Se o decodificador de correção de erro recebesse uma das quatro words de dados possíveis com erro, ele teria de escolher entre corrigir para 0110 ou 0001. Embora essas quatro words com erro tenham apenas 1 bit trocado do padrão correto, 0110, elas têm 2 bits diferentes da correção alternativa, 0001. Logo, o mecanismo de correção de erro pode facilmente escolher a correção para 0110, pois um único erro é uma probabilidade muito maior. Para ver se dois erros podem ser detectados, basta observar que todas as combinações com 2 bits trocados possuem um código diferente. A única reutilização do mesmo código é com 3 bits diferentes, mas, se corrigirmos um erro de 2 bits, corrigiremos para o valor errado, pois o decodificador irá supor que ocorreu apenas um único erro. Se quisermos corrigir erros de 1 bit e detectar, mas não corrigir erroneamente os erros de 2 bits, precisamos de um código com distância 4.

Embora distinguíssemos entre o código e os dados em nossa explicação, na verdade, um código de correção de erro trata a combinação de código e dados como uma única word em um código maior (7 bits neste exemplo). Assim, ele lida com erros nos bits de código da mesma forma que os erros nos bits de dados.

Embora o exemplo anterior exija $n - 1$ bits para n bits de dados, o número de bits exigido cresce lentamente, de modo que, para um código de distância 3, uma word de 64 bits precisa de 7 bits e uma word de 128 bits precisa de 8. Esse tipo de código é chamado de *código de Hamming*, devido a R. Hamming, que descreveu um método para a criação de tal código.

B.10. Máquinas de estados finitos

Como vimos anteriormente, os sistemas lógicos digitais podem ser classificados como combinacionais ou sequenciais. Os sistemas sequenciais contêm estado armazenado em elementos da memória internos ao sistema. Seu comportamento depende do conjunto de entradas fornecidas e do conteúdo da memória interna ou do estado do sistema. Assim, um sistema sequencial não pode ser descrito com uma tabela verdade. Em vez disso, um sistema sequencial é descrito como uma **máquina de estados finitos** (ou, normalmente, apenas *máquina de estados*). Uma máquina de estados finitos possui um conjunto de estados e duas funções, chamadas **função de próximo estado** e a **função de saída**. O conjunto de estados corresponde a todos os valores possíveis do armazenamento interno. Assim, se houver n bits de armazenamento, haverá 2^n estados. A função de próximo estado é uma função combinacional que, dadas as entradas e o estado atual, determina o próximo estado do sistema. A função de saída produz um conjunto de saídas a partir do estado atual e suas entradas. A [Figura B.10.1](#) mostra isso em forma de diagrama.

máquina de estados finitos

Uma função lógica sequencial consistindo em um conjunto de entradas e saídas, uma função de próximo estado que mapeia o estado atual e as entradas para um novo estado, e uma função de saída que mapeia o estado atual e, possivelmente, as entradas para um conjunto de saídas ativas.

função de próximo estado

Uma função combinacional que, dadas as entradas e o estado atual, determina o próximo estado de uma máquina de estados finitos.

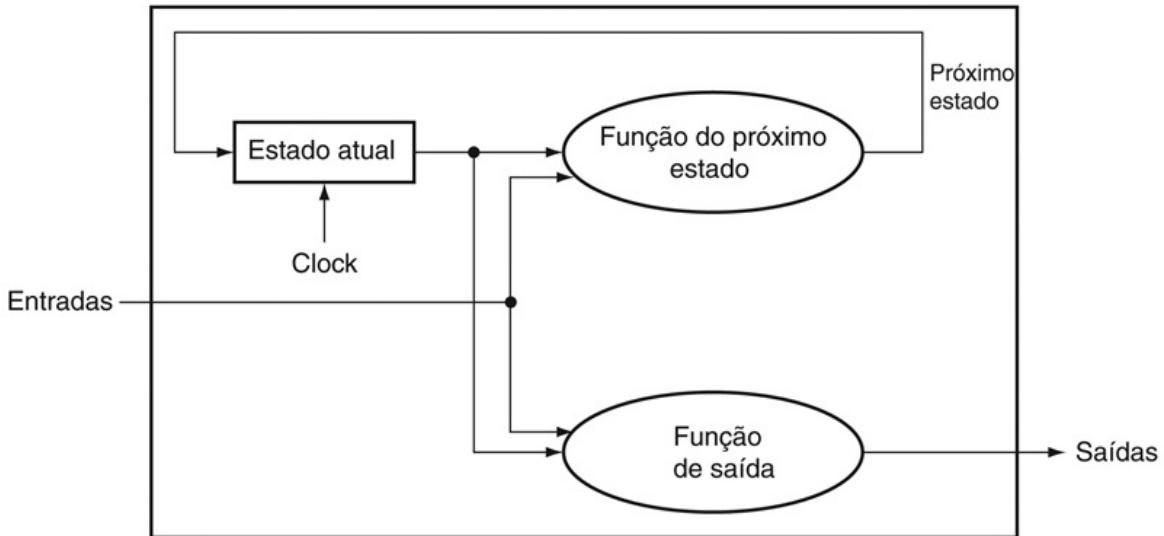


FIGURA B.10.1 Uma máquina de estados consiste em armazenamento interno que contém o estado e duas funções combinacionais: a função de próximo estado e a função de saída.

Normalmente, a função de saída é restrita a usar apenas o estado atual como sua entrada; isso não muda a capacidade de uma máquina sequencial, mas afeta seu funcionamento interno.

As máquinas de estados que discutimos aqui e no [Capítulo 4](#) são *síncronas*. Isso significa que o estado muda junto com o ciclo de clock, e um novo estado é calculado uma vez a cada clock. Assim, os elementos de estado são atualizados apenas na transição do clock. Usamos essa metodologia nesta seção e no [Capítulo 4](#), e em geral não mostramos o clock explicitamente. Usamos máquinas de estados no [Capítulo 4](#) para controlar a execução do processador e as ações do caminho de dados.

Para ilustrar como uma máquina de estados finitos opera e é projetada, vejamos um exemplo simples e clássico: controlar um semáforo de trânsito. (Os [Capítulos 4](#) e [5](#) contêm exemplos mais detalhados do uso de máquinas de estados finitos para controlar a execução do processador.) Quando uma máquina de estados finitos é usada como controlador, a função de saída normalmente é restrita a depender apenas do estado atual. Essa máquina de estados finitos é chamada de *máquina de Moore*. Esse é o tipo de máquina de estados finitos usado em todo este livro. Se a função de saída puder depender do estado atual e da entrada atual, a máquina é chamada de *máquina de Mealy*. Essas duas máquinas são equivalentes em suas capacidades, e uma pode ser transformada na outra mecanicamente. A vantagem básica de uma máquina de Moore é que ela

pode ser mais rápida, enquanto uma máquina de Mealy pode ser menor, pois pode precisar de menos estados do que uma máquina de Moore. No [Capítulo 5](#), discutimos as diferenças com mais detalhes e mostramos uma versão Verilog do controle de estados finitos usando uma máquina de Mealy.

Nosso exemplo trata do controle de um semáforo em um cruzamento de uma rua norte-sul e uma rua leste-oeste. Para simplificar, vamos considerar apenas os sinais verde e vermelho; a inclusão de um sinal amarelo fica como um exercício. Queremos que os sinais alternem por não menos do que 30 segundos em cada direção, de modo que usaremos um clock de 0,033Hz para que a máquina alterne entre os estados com um tempo não inferior a 30 segundos. Existem dois sinais de saída:

- *NSlite*: quando este sinal está ativo, a luz na rua norte-sul está verde; quando esse sinal está inativo, a luz na rua norte-sul está vermelha.
 - *EWlite*: quando este sinal está ativo, a luz na rua leste-oeste está verde; quando esse sinal está inativo, a luz na rua leste-oeste está vermelha.
- Além disso, existem duas entradas:
- *NScar*: indica que um carro está sobre o detector colocado na rua, na frente do semáforo da rua norte-sul (indo para o norte ou para o sul).
 - *EWcar*: indica que um carro está sobre o detector colocado na rua, na frente do semáforo da rua leste-oeste (indo para o leste ou para o oeste).

O semáforo deverá mudar de uma direção para a outra somente se um carro estiver esperando para seguir na outra direção; caso contrário, o semáforo deverá continuar a mostrar verde na mesma direção do último carro que cruzou a interseção.

Para implementar esse semáforo simples, precisamos de dois estados:

- *NSgreen*: o semáforo é verde na direção norte-sul.
- *EWgreen*: o semáforo é verde na direção leste-oeste.

Também precisamos criar a função de próximo estado, que pode ser especificada com uma tabela:

	Entradas		
	NScar	EWcar	Estado seguinte
Nsgreen	0	0	NSgreen
Nsgreen	0	1	EWgreen
Nsgreen	1	0	NSgreen
Nsgreen	1	1	EWgreen
Ewgreen	0	0	EWgreen

Ewgreen	0	1	EWgreen
Ewgreen	1	0	NSgreen
Ewgreen	1	1	NSgreen

Observe que não especificamos no algoritmo o que acontece quando carros se aproximam vindo das duas direções. Nesse caso, a função do estado seguinte que apresentamos muda o estado para garantir que um fluxo constante de carros de uma direção não possa bloquear um carro na outra direção.

A máquina de estados finitos é concluída com a especificação da função de saída:

Saídas		
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

Antes de examinarmos como implementar esta máquina de estados finitos, vejamos uma representação gráfica, muito utilizada para máquinas de estados finitos. Nessa representação, os nós são usados para indicar estados. Dentro do nó, colocamos uma lista das saídas ativadas para esse estado. Os arcos direcionados são usados para mostrar a função de próximo estado, com os rótulos nos arcos especificando a condição de entrada como funções lógicas. A Figura B.10.2 mostra a representação gráfica para essa máquina de estados finitos.

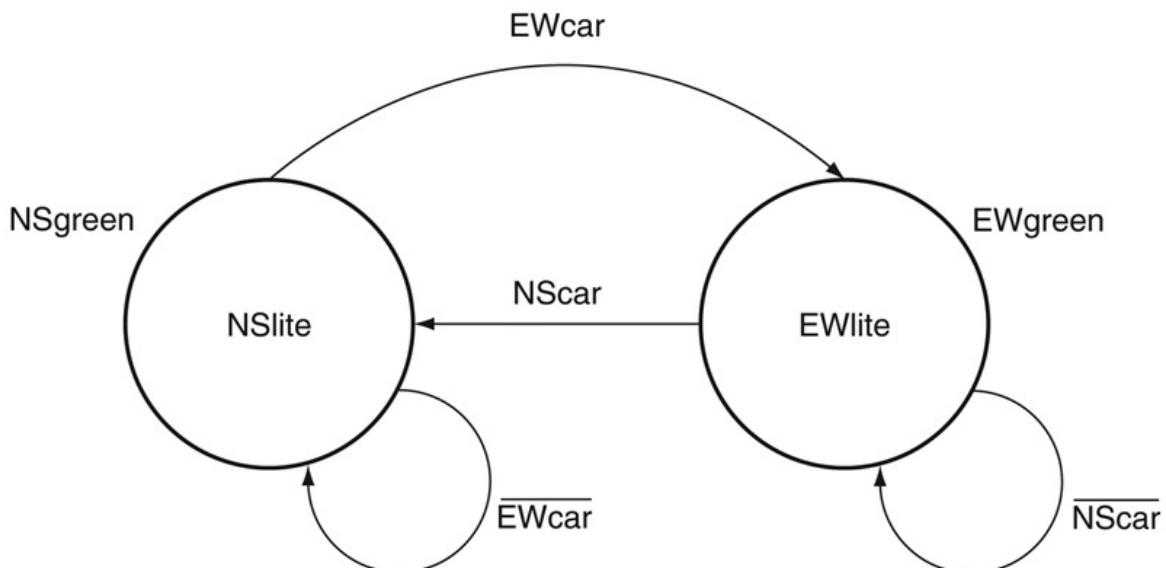


FIGURA B.10.2 A representação gráfica do controle de semáforo de dois estados.

Simplificamos as funções lógicas nas transições de estado. Por exemplo, a transição de NSgreen para EWgreen na tabela de próximo estado é $(NScar \cdot EWcar) + (NScar \cdot \overline{EWcar})$, que é equivalente a $EWcar$.

Uma máquina de estados finitos pode ser implementada com um registrador para manter o estado atual e um bloco de lógica combinacional que calcula a função de próximo estado e a função de saída. A Figura B.10.3 mostra uma máquina de estados finitos com 4 bits de estado e, portanto, até 16 estados. Para implementar a máquina de estados finitos dessa maneira, primeiro temos de atribuir números de estado aos estados. Este processo é chamado de *atribuição de estado*. Por exemplo, poderíamos atribuir NSgreen ao estado 0 e WEgreen ao estado 1. O registrador de estado teria um único bit. A função de próximo estado seria dada como

$$\text{PróximoEstado} = (\overline{\text{EstadoAtual}} \cdot EWcar) + (\text{EstadoAtual} \cdot \overline{NScar})$$

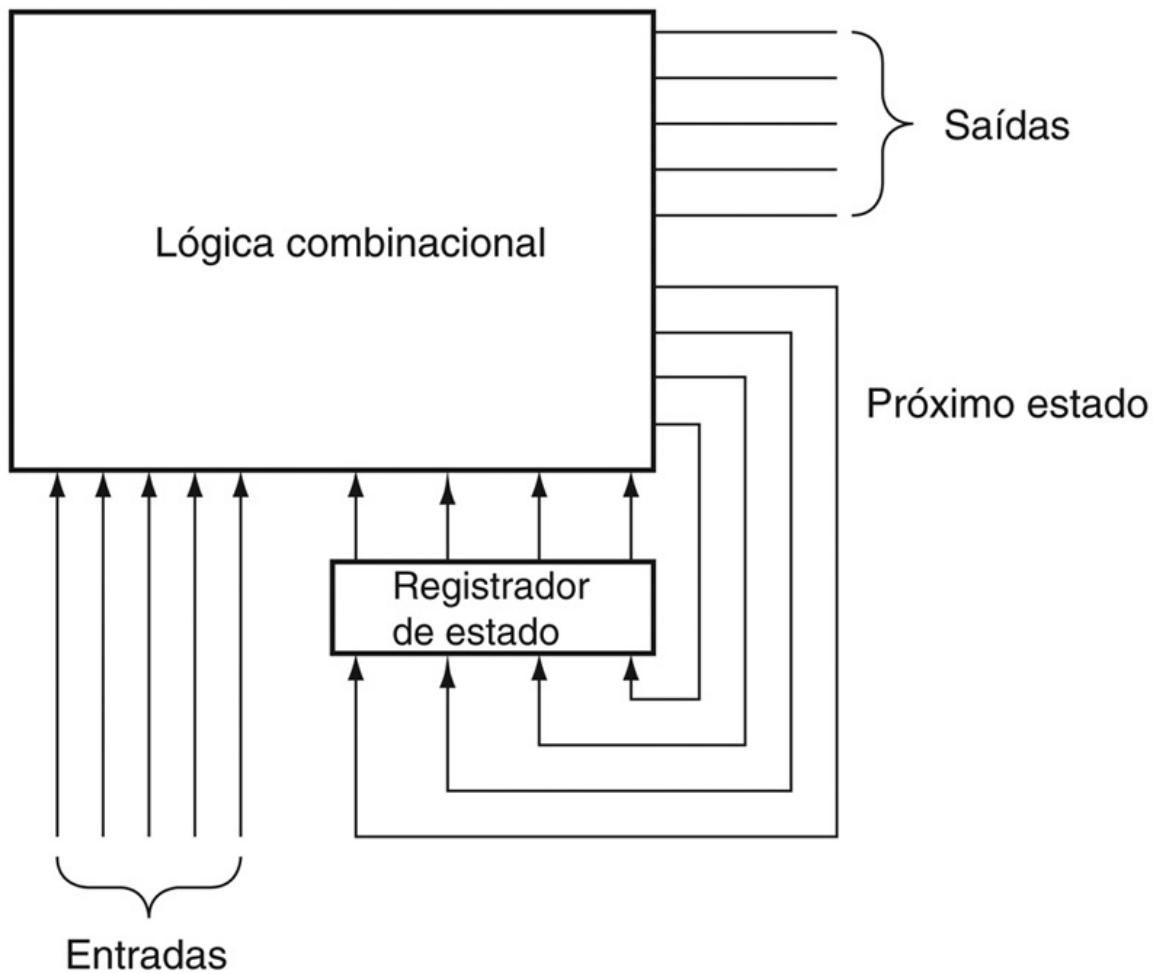


FIGURA B.10.3 Uma máquina de estados finitos é implementada com um registrador de estado que mantém o estado atual e um bloco de lógica combinacional para calcular as funções de próximo estado e de saída.

As duas últimas funções normalmente são separadas e implementadas com dois blocos de lógica separados, o que pode exigir menos portas.

onde EstadoAtual é o conteúdo do registrador de estado (0 ou 1) e EstadoSeguinte é a saída da função de próximo estado, que será escrita no registrador de estado ao final do ciclo de clock. A função de saída também é simples:

$$\begin{aligned} \text{Ewlite} &= \text{EstadoAtual} \\ \text{Nslite} &= \text{EstadoAtual} \end{aligned}$$

O bloco de lógica combinacional normalmente é implementado por meio de lógica estruturada, como PLA. Uma PLA pode ser construída, automaticamente, a partir das tabelas de função de próximo estado e saída. Na verdade, existem programas de CAD (Computer-Aided Design) que transformam uma representação gráfica ou textual de uma máquina de estados finitos e produzem automaticamente uma implementação otimizada.

Para mostrar como poderíamos escrever o controle em Verilog, a [Figura B.10.4](#) mostra uma versão Verilog projetada para a síntese. Observe que, para essa função de controle simples, uma máquina de Mealy não é útil, mas esse estilo de especificação é utilizado no [Capítulo 5](#) para implementar uma função de controle que é uma máquina de Mealy e possui menos estados do que o controlador da máquina de Moore.

```
module TrafficLite (EWCAR, NSCAR, EWLITE, NSLITE, clock);  
    input EWCAR, NSCAR, clock;  
    output EWLITE, NSLITE;  
  
    reg state;  
  
    initial state=0; // define estado inicial  
    // após duas atribuições, define a saída, baseada apenas  
    // na variável de estado  
    assign NSLITE = ~state; //NSLITE ativo se estado = 0;  
    assign EWLITE = state; //EWLITE ativo se estado = 1;  
  
    always @(posedge clock) // todas as atualizações de estado  
    // em uma transição positiva do clock  
        case (state)  
            0: state = EWCAR; //muda estado apenas se EWCAR  
            1: state = NSCAR; //muda estado apenas se NSCAR  
        endcase  
endmodule
```

FIGURA B.10.4 Uma versão Verilog do controlador de semáforo.

Verifique você mesmo

Qual é o menor número de estados em uma máquina de Moore para os quais

uma máquina de Mealy poderia ter menos estados?

- a. Dois, pois poderia haver uma máquina de Mealy de um estado que poderia fazer a mesma coisa.
- b. Três, pois poderia ser uma máquina de Moore simples, que fosse para um dentre dois estados diferentes e sempre retornasse para o estado original depois disso. Para uma máquina tão simples, uma máquina de Mealy de dois estados é possível.
- c. Você precisa de pelo menos quatro estados para explorar as vantagens de uma máquina de Mealy em relação a uma máquina de Moore.

B.11. Metodologias de temporização

No decorrer deste apêndice e no restante do texto, usamos uma metodologia de temporização acionada por transição. Essa metodologia de temporização tem a vantagem de ser mais simples de explicar e entender do que uma metodologia acionada por nível. Nesta seção, explicamos essa metodologia de temporização com um pouco mais de detalhe e também apresentamos o clocking sensível ao nível. Concluímos esta seção discutindo rapidamente a questão dos sinais assíncronos e sincronizadores, um problema importante para os projetistas digitais.

A finalidade desta seção é apresentar os principais conceitos da metodologia de clocking. A seção faz algumas suposições importantes para simplificar; se você estiver interessado em entender a metodologia de temporização com mais detalhes, consulte uma das referências listadas ao final deste apêndice.

Usamos a metodologia de temporização acionada por transição porque ela é mais simples de explicar e tem menos regras exigidas para exatidão. Em particular, se considerarmos que todos os clocks chegam ao mesmo tempo, temos garantias de que um sistema com registradores acionados por transição entre os blocos de lógica combinacional pode operar corretamente, sem condições de corrida, se tornarmos o clock longo o suficiente. Uma condição de *corrida* ocorre quando o conteúdo de um elemento de estado depende da velocidade relativa de elementos lógicos diferentes. Em um projeto acionado por transição, o ciclo de clock precisa ser grande o suficiente para acomodar o caminho de um flip-flop, passando pela lógica combinacional, até chegar a outro flip-flop, onde precisa satisfazer ao requisito do tempo de preparação. A [Figura B.11.1](#) mostra esse requisito para um sistema que usa flip-flops acionados por transição de subida. Em tal sistema, o período de clock (ou tempo de ciclo) precisa ser, pelo menos, tão grande quanto

$$t_{\text{prop}} + t_{\text{combinacional}} + t_{\text{preparação}}$$

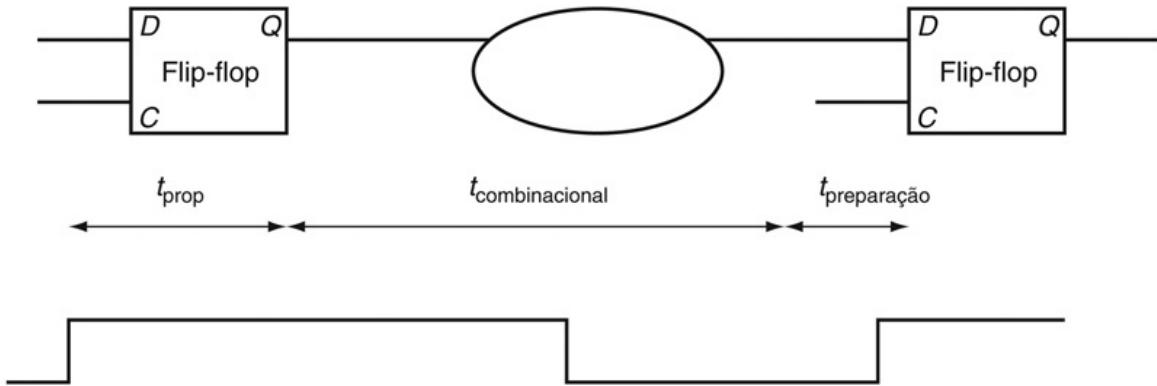


FIGURA B.11.1 No projeto acionado por transição, o clock precisa ser grande o suficiente para permitir que os sinais sejam válidos para o tempo de preparação exigido antes da próxima transição de clock.

O tempo para uma entrada de flip-flop se propagar até as saídas do flip-flop é t_{prop} ; o sinal leva então $t_{\text{combinacional}}$ para atravessar a lógica combinacional e precisa ser válido por $t_{\text{preparação}}$ antes da próxima transição de clock.

para os valores de pior caso desses três atrasos, que são definidos da seguinte maneira:

- t_{prop} é o tempo para um sinal se propagar por um flip-flop; às vezes, ele também é chamado de clock-para-Q.
- $t_{\text{combinacional}}$ é o maior atraso para qualquer lógica combinacional (que, por definição, é cercada por dois flip-flops).
- $t_{\text{preparação}}$ é o tempo antes da transição do clock de subida que a entrada para um flip-flop precisa ser válida.

Fazemos uma suposição para simplificar: os requisitos do tempo de suspensão são satisfeitos, o que quase nunca é um problema com a lógica moderna.

Uma complicação adicional que precisa ser considerada nos projetos acionados por transição é a **inclinação do clock**. Inclinação do clock é a diferença em tempo absoluto entre os instantes em que dois elementos de estado vêem uma transição de clock. A inclinação do clock aumenta porque o sinal de clock normalmente usará dois caminhos diferentes, com atrasos um pouco diferentes, para alcançar dois elementos de estado diferentes. Se a inclinação do clock for grande o suficiente, pode ser possível que um elemento de estado mude e faça com que a entrada para outro flip-flop mude antes que a transição do clock seja vista pelo segundo flip-flop.

inclinação do clock

A diferença em tempo absoluto entre os tempos em que dois elementos de estado vêem uma transição de clock.

A Figura B.11.2 ilustra esse problema, ignorando o tempo de preparação e o atraso de propagação do flip-flop. Para evitar operação incorreta, o período de clock é aumentado para permitir a inclinação máxima do clock. Assim, o período do clock precisa ser maior do que

$$t_{\text{prop}} + t_{\text{combinacional}} + t_{\text{preparação}} + t_{\text{inclinação}}$$

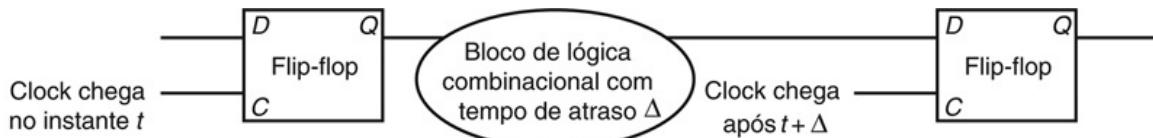


FIGURA B.11.2 Ilustração de como a inclinação do clock pode causar uma condição de corrida, levando à operação incorreta.

Devido à diferença entre quando os dois flip-flops veem o clock, o sinal armazenado no primeiro flip-flop pode se adiantar e mudar a entrada para o segundo flip-flop antes que o clock chegue no segundo flip-flop.

Com essa restrição sobre o período de clock, os dois clocks também podem chegar na ordem contrária, com o segundo clock chegando $t_{\text{inclinação}}$ mais cedo, e o circuito funcionará corretamente. Os projetistas reduzem os problemas com inclinação do clock roteando o sinal do clock para minimizar a diferença nos tempos de chegada. Além disso, projetistas inteligentes também oferecem alguma margem, tornando o clock um pouco maior do que o mínimo; isso permite a variação nos componentes e também na fonte de alimentação. Como a inclinação do clock também pode afetar os requisitos do tempo de suspensão, é importante minimizar o tamanho da inclinação do clock.

Os projetos acionados por transição possuem duas desvantagens: eles exigem uma lógica extra e, às vezes, podem ser mais lentos. Olhar apenas para o flip-flop D *versus* o latch sensível ao nível que usamos para construir o flip-flop

mostra que o projeto acionado por transição requer mais lógica. Uma alternativa é usar o **clocking sensível ao nível**. Como as mudanças de estado em uma metodologia sensível ao nível não são instantâneas, um esquema sensível ao nível é ligeiramente mais complexo e exige cuidado adicional para fazer com que opere corretamente.

clocking sensível ao nível

Uma metodologia de temporização em que as mudanças de estado ocorrem em níveis de clock altos ou baixos, mas não são instantâneas como são em projetos acionados por transição.

Temporização sensível ao nível

Em uma metodologia de temporização sensível ao nível, as mudanças de estado ocorrem nos níveis alto ou baixo, mas não são instantâneas, como acontece em uma metodologia acionada por transição. Devido à mudança não instantânea no estado, condições de corrida podem ocorrer com facilidade. Para garantir que um projeto sensível ao nível também funcione corretamente se o clock for lento o suficiente, os projetistas utilizam o *clocking em duas fases*. O clocking em duas fases é um esquema que utiliza dois sinais de clock não superpostos. Como os dois clocks, normalmente chamados de Φ_1 e Φ_2 , não são superpostos, no máximo um dos sinais de clock é alto em um momento determinado, como mostra a Figura B.11.3. Podemos usar esses clocks para criar um sistema que contenha latches sensíveis ao nível, mas que seja livre de quaisquer condições de corrida, como eram os projetos acionados por transição.

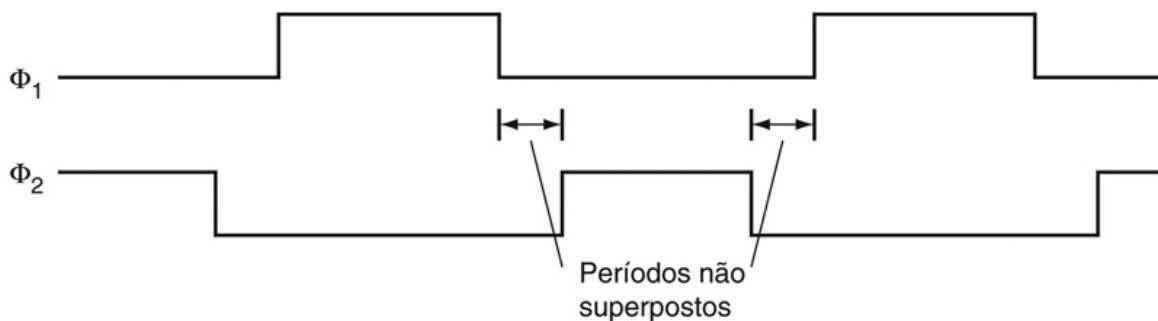


FIGURA B.11.3 Um esquema de clocking de duas fases mostrando o ciclo de cada clock e os períodos não superpostos.

Um modo simples de projetar tal sistema é alternar o uso de latches abertos em φ_1 com os latches abertos em φ_2 . Como os dois clocks não estão ativos ao mesmo tempo, uma condição de corrida não pode ocorrer. Se a entrada para um bloco combinacional for um clock φ_1 , então sua saída é acionada por um clock φ_2 , e só é aberta durante φ_2 , quando o latch de entrada estiver fechado e, portanto, tiver uma saída válida. A [Figura B.11.4](#) mostra como opera um sistema com temporização de duas fases e latches alternados. Assim como em um projeto acionado por transição, temos de prestar atenção à inclinação do clock, particularmente entre as duas fases do clock. Aumentando o intervalo de não superposição entre as duas fases, podemos reduzir a margem de erro em potencial. Assim, o sistema tem garantias de operar corretamente se cada fase for grande o suficiente e houver não sobreposição grande o suficiente entre as fases.

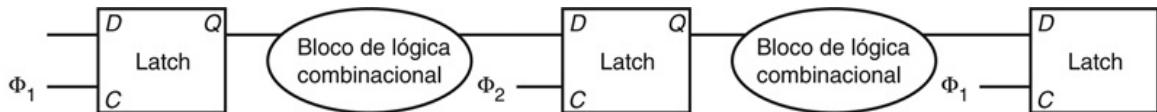


FIGURA B.11.4 Um esquema de temporização de duas fases com latches alternados, mostrando como o sistema opera nas duas fases de clock.

A saída de um latch é estável na fase oposta, a partir de sua entrada C. Assim, o primeiro bloco de entradas combinacionais possui uma entrada estável durante φ_2 , e sua saída é guardada durante φ_2 . O segundo bloco combinacional (mais à direita) opera exatamente na forma oposta, com entradas estáveis durante φ_1 . Assim, os atrasos pelos blocos combinacionais determinam o tempo mínimo que os respectivos clocks precisam estar ativos. O tamanho do período não superposto é determinado pela inclinação de clock máxima e pelo atraso mínimo de qualquer bloco lógico.

Entradas assíncronas e sincronizadores

Usando um único clock ou um clock de duas fases, podemos eliminar condições de corrida se os problemas de inclinação de clock forem evitados. Infelizmente, não é prático fazer um sistema inteiro funcionar com um único clock e ainda reduzir a inclinação do clock. Embora a CPU possa usar um único clock, os

dispositivos de E/S provavelmente terão seu próprio clock. Um dispositivo assíncrono pode se comunicar com a CPU por uma série de etapas de handshaking. Para traduzir a entrada assíncrona para um sinal síncrono que pode ser usado para mudar o estado de um sistema, precisamos usar um *sincronizador*, cujas entradas são o sinal assíncrono e um clock, cuja saída é um sinal síncrono com o clock de entrada.

Nossa primeira tentativa de criar um sincronizador usa um flip-flop D acionado por transição, cuja entrada *D* é o sinal assíncrono, como mostra a [Figura B.11.5](#). Como nos comunicamos com um protocolo de handshaking, não importa se detectamos o estado ativo do sinal assíncrono em um clock ou no seguinte, pois o sinal será mantido ativo até que seja confirmado. Assim, você poderia pensar que essa estrutura simples é suficiente para examinar o sinal com precisão, o que aconteceria, exceto por um pequeno problema.

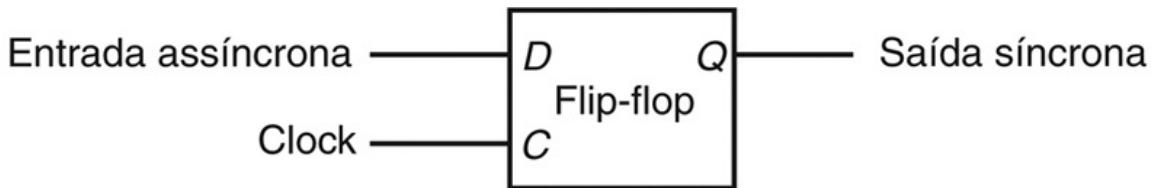


FIGURA B.11.5 Um sincronizador construído a partir de um flip-flop D é usado para examinar um sinal assíncrono para produzir uma saída que é síncrona com o clock.

Esse “sincronizador” não funcionará corretamente!

O problema é uma situação chamada **metaestabilidade**. Suponha que o sinal assíncrono esteja intercalando entre alto e baixo quando chega a transição do clock. Certamente, não é possível saber se o sinal será mantido como alto ou baixo. Poderíamos viver com esse problema. Infelizmente, a situação é pior: quando o sinal examinado não estiver estável pelos tempos de preparação e suspensão exigidos, o flip-flop poderá entrar em um estado *metaestável*. Nesse estado, a saída não terá um valor alto ou baixo legítimo, mas estará na região indeterminada entre eles. Além do mais, o flip-flop não tem garantias de sair desse estado em qualquer período determinado. Alguns blocos lógicos que examinam a saída do flip-flop podem ver sua saída como 0, enquanto outros podem vê-la como 1. Essa situação é chamada de **falha do sincronizador**.

metaestabilidade

Uma situação que ocorre se um sinal for examinado quando não estiver estável pelos tempos de preparação e suspensão exigidos, possivelmente fazendo com que o valor examinado caia na região indeterminada entre um valor alto e baixo.

falla do sincronizador

Uma situação em que um flip-flop entra em um estado metaestável e onde alguns blocos lógicos lendo a saída do flip-flop veem 0 enquanto outros veem 1.

Em um sistema puramente síncrono, a falha do sincronizador pode ser evitada garantindo-se que os tempos de preparação e suspensão para um flip-flop ou latch sempre sejam atendidos, mas isso é impossível quando a entrada é assíncrona. Em vez disso, a única solução possível é esperar por um tempo suficiente antes de examinar a saída do flip-flop para garantir que sua saída seja estável, e que tenha saído do estado metaestável, se tiver entrado nele. Qual é o tempo suficiente? Bem, a probabilidade de que o flip-flop permaneça no estado metaestável diminui exponencialmente, de modo que, depois de um período muito curto, a probabilidade de que o flip-flop esteja no estado metaestável é muito baixa; porém, a probabilidade nunca chega a 0! Assim, os projetistas esperam por tempo suficiente para que a probabilidade de uma falha do sincronizador seja muito baixa, e o tempo entre essas falhas será de anos ou até mesmo milhares de anos.

Para a maioria dos projetos de flip-flop, esperar por um período muitas vezes maior do que o tempo de preparação torna a probabilidade de falha de sincronismo muito baixa. Se a taxa de clock for maior do que o período de metaestabilidade em potencial (o que é provável), então um sincronizador seguro poderá ser criado com dois flip-flops D, como mostra a [Figura B.11.6](#). Se você estiver interessado em ler mais sobre esses problemas, consulte as referências.

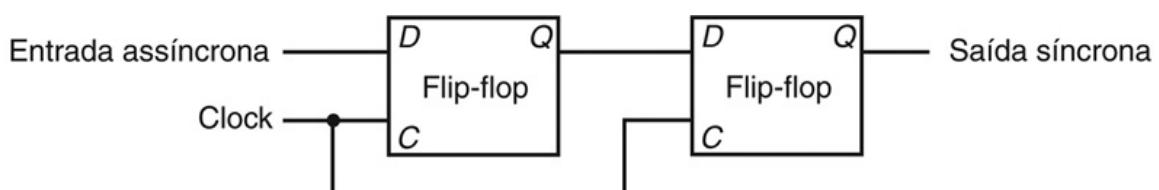


FIGURA B.11.6 Este sincronizador funcionará corretamente

se o período de metaestabilidade que desejamos evitar for menor que o período de clock.

Embora a saída do primeiro flip-flop possa ser metaestável, ela não será vista por qualquer outro elemento lógico até o segundo clock, quando o segundo flip-flop D examina o sinal, que nesse momento não deverá mais estar em um estado metaestável.

Verifique você mesmo

Suponha que tenhamos um projeto com uma inclinação de clock muito grande – maior do que o **tempo de propagação** do registrador. Sempre será possível que esse projeto atraso o clock por tempo suficiente para garantir que a lógica opere corretamente?

- a. Sim, se o clock for lento o suficiente, os sinais sempre podem se propagar e o projeto funcionará, mesmo que a inclinação seja muito grande.
- b. Não, pois é possível que dois registradores vejam a mesma transição de clock afastada o suficiente para que um registrador seja acionado, e suas saídas propagadas e vistas por um segundo registrador com a mesma transição de clock.

tempo de propagação

O tempo exigido para que uma entrada de um flip-flop se propague para as saídas do flip-flop.

B.12. Dispositivos programáveis em campo

Dentro de um chip personalizado ou semipersonalizado, os projetistas podem utilizar a flexibilidade da estrutura básica para facilmente implementar a lógica combinacional ou sequencial. Como um projetista que não quer usar um CI personalizado ou semipersonalizado implementa uma lógica complexa tirando proveito dos níveis de integração muito altos à sua disposição? Os componentes mais populares utilizados para o projeto lógico sequencial e combinacional fora de um CI personalizado ou semipersonalizado é um **dispositivo programável em campo (FPD – Field Programmable Device)**. Um FPD é um circuito integrado contendo lógica combinacional, e possivelmente dispositivos de memória, configuráveis pelo usuário final.

dispositivo programável em campo (FPD)

Um circuito integrado contendo lógica combinacional, e possivelmente dispositivos de memória, configurável pelo usuário final

Os FPDs em geral se encontram em dois campos: **dispositivos lógicos programáveis** (PLDs – Programmable Logic Devices), que são puramente combinacionais, e **gate arrays programáveis em campo** (FPGAs – Field Programmable Gate Arrays), que oferecem lógica combinacional e flip-flops. Os PLDs consistem em duas formas: **PLDs simples** (SPLDs), que normalmente são uma PLA ou uma **lógica de array programável** (PAL – Programmable Array Logic), e PLDs complexos, que permitem mais de um bloco lógico e também interconexões configuráveis entre os blocos. Quando falamos de uma PLA em um PLD, queremos dizer uma PLA com plano and e or programável pelo usuário. Uma **PAL** é como uma PLA, exceto que o plano or é fixo.

dispositivo lógico programável (PLD)

Um circuito integrado com lógica combinacional, cuja função é configurada pelo usuário final.

gate array programável em campo (FPGA)

Um circuito integrado configurável contendo blocos lógicos combinacionais e flip-flops.

dispositivo lógico programável simples (SPLD)

Dispositivo lógico programável normalmente contendo uma única PAL ou PLA.

array lógico programável (PAL)

Contém um plano and programável seguido por um plano or fixo.

Antes de discutirmos sobre os FPGAs, é útil falarmos sobre como os FPDs são configurados. A configuração é basicamente uma questão de onde fazer ou romper conexões. Estruturas de porta e de registrador são estáticas, mas as conexões podem ser configuradas. Observe que, configurando as conexões, um usuário determina quais funções lógicas são implementadas. Considere uma PLA configurável: determinando onde estão as conexões no plano and e no plano or, o usuário dita quais funções lógicas são computadas pela PLA. As conexões nos FPDs são permanentes ou reconfiguráveis. As conexões permanentes envolvem a criação ou a destruição de uma conexão entre dois fios. FPLDs atuais utilizam uma tecnologia **antifuse**, que permite que uma conexão seja criada no momento da programação, que será então permanente. A outra maneira de configurar FPLDs CMOS é por meio de uma SRAM. A SRAM é baixada durante a inicialização, e o conteúdo controla a configuração das chaves, que, por sua vez, determinam quais linhas de metal estão conectadas. O uso do controle da SRAM tem a vantagem de que o FPD pode ser reconfigurado alterando-se o conteúdo da SRAM. As desvantagens do controle baseado em SRAM são duas: a configuração é volátil, sendo recarregada a cada inicialização, e o uso de transistores ativos para as chaves aumenta um pouco a resistência de tais conexões.

antifuse

Uma estrutura de um circuito integrado que, quando programada, faz uma conexão permanente entre dois fios.

FPGAs incluem dispositivos lógicos e de memória, normalmente estruturados em um array bidimensional, com os corredores dividindo as linhas e colunas usadas para a interconexão global entre as células do array. Cada célula é uma combinação de portas e flip-flops que pode ser programada para realizar alguma função específica. Por serem basicamente RAMs pequenas e programáveis, elas também são chamadas de **Look-Up Tables (LUTs)**. As FPGAs mais novas contêm blocos de montagem mais sofisticados, como partes de somadores e blocos de RAM que podem ser usados para criar bancos de registradores. Algumas FPGAs grandes contêm até mesmo núcleos RISC de 32 bits!

Look-Up Tables (LUTs)

Em um dispositivo programável em campo, o nome dado às células porque consistem em uma pequena quantidade de lógica e RAM.

Além de programar cada célula para realizar uma função específica, as interconexões entre as células também são programáveis, permitindo que as FPGAs modernas, com centenas de blocos e centenas de milhares de portas, sejam utilizadas para funções lógicas complexas. A interconexão é um desafio importante nos chips personalizados, e isso é ainda mais verdadeiro para FPGAs, pois as células não representam unidades naturais de decomposição para o projeto estruturado. Em muitas FPGAs, 90% da área é reservada para a interconexão e apenas 10% são blocos lógicos e de memória.

Assim como você não pode projetar um chip personalizado ou semipersonalizado sem ferramentas CAD, também precisa delas para FPDs. Foram desenvolvidas ferramentas de síntese de lógica que visam às FPGAs, permitindo a geração de um sistema usando FPGAs a partir da Verilog estrutural e comportamental.

B.13. Comentários finais

Esse apêndice introduz os fundamentos do projeto lógico. Se você tiver compreendido o material deste apêndice, estará pronto para pegar o material dos Capítulos 4 e 5, ambos usando os conceitos discutidos extensivamente neste apêndice.

Leitura adicional

Existem muitos textos bons sobre projeto lógico. Aqui estão alguns que você poderia examinar.

Ciletti, M. D. [2002] *Advanced Digital Design with the Verilog HDL*,
Englewood Cliffs, NJ: Prentice-Hall.

Um livro profundo sobre projeto lógico usando Verilog.

Katz, R. H. [2004] *Modern Logic Design*, segunda edição, Reading, MA:
Addison Wesley.

Um texto geral sobre projeto lógico.

Wakerly, J. F. [2000] *Digital Design: Principles and Practices*, terceira edição,
Englewood Cliffs, NJ: Prentice-Hall.

Um texto geral sobre projeto lógico.

B.14. Exercícios

B.1 [10] <§B.2> Além das leis básicas que discutimos nesta seção, existem dois teoremas importantes, chamados teoremas de DeMorgan:

$$\overline{A + B} = \overline{A} \cdot \overline{B} \text{ and } \overline{A \cdot B} = \overline{A} + \overline{B}$$

Prove os teoremas de DeMorgan com uma tabela verdade no formato

A	B	\overline{A}	\overline{B}	$A + B$	$\overline{A} \cdot \overline{B}$	$A \cdot B$	$\overline{A + B}$
0	0	1	1	1	1	1	1
0	1	1	0	0	0	1	1
1	0	0	1	0	0	1	1
1	1	0	0	0	0	0	0

B.2 [15] <§B.2> Prove que as duas equações para E no exemplo a partir da página B-7 são equivalentes, usando os teoremas de DeMorgan e os axiomas mostrados na página B-7.

B.3 [10] <§B.2> Mostre que existem 2^n entradas em uma tabela verdade para uma função com n entradas.

B.4 [10] <§B.2> Uma função lógica que é usada para diversas finalidades (incluindo dentro de somadores e para calcular paridade) é *OR exclusiva*. A saída de uma função OR exclusiva de duas entradas é verdadeira somente se exatamente uma das entradas for verdadeira. Mostre a tabela verdade para uma função OR exclusiva de duas entradas e implemente essa função usando portas AND, portas OR e inversores.

B.5 [15] <§B.2> Prove que a porta NOR é universal mostrando como montar as funções AND, OR e NOT usando uma porta NOR de duas entradas.

B.6 [15] <§B.2> Prove que a porta NAND é universal, mostrando como montar as funções AND, OR e NOT usando uma porta NAND de duas entradas.

B.7 [10] <§§B.2, B.3> Construa a tabela verdade para uma função de paridade ímpar com quatro entradas (veja na página B-53 uma descrição sobre paridade).

B.8 [10] <§§B.2, B.3> Implemente a função de paridade ímpar com quatro entradas e portas AND e OR usando entradas e saídas em bolha.

B.9 [10] <§§B.2, B.3> Implemente a função de paridade ímpar com quatro entradas com uma PLA.

B.10 [15] <§§B.2, B.3> Prove que um multiplexador de duas entradas também é universal, mostrando como montar uma porta NAND (ou NOR) usando um multiplexador.

B.11 [5] <§§4.2, B.2, B.3> Suponha que X consista em 3 bits, $x_2 \ x_1 \ x_0$.

Escreva quatro funções lógicas que sejam verdadeiras se e somente se:

- X contém apenas 0
- X contém um número par de 0s
- X, quando interpretado como um número binário sem sinal, é menor que 4
- X, quando interpretado como um número com sinal (complemento a dois) é negativo

B.12 [5] <§§4.2, B.2, B.3> Implemente as quatro funções descritas no Exercício B.11 usando uma PLA

B.13 [5] <§§4.2, B.2, B.3> Suponha que X consista em 3 bits, $x_2 \ x_1 \ x_0$, e Y consista em 3 bits, $y_2 \ y_1 \ y_0$. Escreva funções lógicas que sejam verdadeiras se e somente se:

- $X < Y$, onde X e Y são considerados números binários sem sinal
- $X < Y$, onde X e Y são considerados números binários sem sinal (complemento a dois)
- $X = Y$

Use uma técnica hierárquica que possa ser estendida para números maiores de bits. Mostre como você pode estendê-la para a comparação em 6 bits.

B.14 [5] <§§B.2, B.3> Implemente uma rede de computação que possui duas entradas de dados (A e B), duas saídas de dados (C e D), e uma entrada de controle (S). Se S é igual a 1, a rede está no modo pass-through, e C deve ser igual a A , e D deve ser igual a B . Se S é igual a 0, a rede está no modo crossing, e C deve ser igual a B , e D deve ser igual a A .

B.15 [15] <§§B.2, B.3> Derive a representação do produto das somas para E , mostrada na página B-11, começando com a representação da soma dos produtos. Você precisará usar os teoremas de DeMorgan.

B.16 [30] <§§B.2, B.3> Dê um algoritmo para construir a representação da soma dos produtos para uma equação de lógica arbitrária consistindo em AND, OR e NOT. O algoritmo deverá ser recursivo e não deverá construir a tabela verdade no processo.

B.17 [5] <§§B.2, B.3> Mostre uma tabela verdade para um multiplexador (entradas A , B e S ; saída C), usando don't cares para simplificar a tabela,

onde for possível.

B.18 [5] <§B.3> Qual é a função implementada pelos seguintes módulos da Verilog:

```

module FUNC1 (I0, I1, S, out);
    input I0, I1;
    input S;
    output out;
    out = S? I1: I0;
endmodule

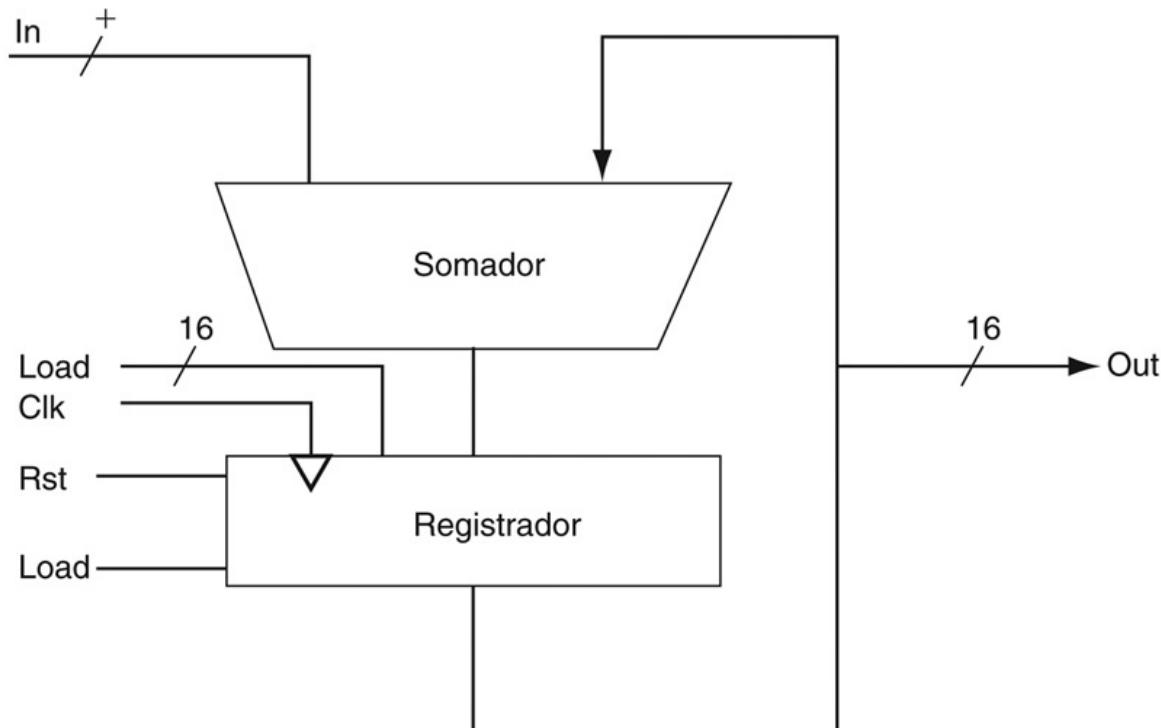
module FUNC2 (out,ctl,clk,reset);
    output [7:0] out;
    input ctl, clk, reset;
    reg [7:0] out;
    always @(posedge clk)
        if (reset) begin
            out <= 8'b0 ;
        end
        else if (ctl) begin
            out <= out + 1;
        end
        else begin
            out <= out - 1;
        end
    endmodule

```

B.19 [5] <§B.4> O código em Verilog na página B-42 é para um flip-flop D.
 Mostre o código em Verilog para um latch D.

B.20 [10] <§§ B.3, B.4> Escreva uma implementação de módulo em Verilog de um decodificador (e/ou codificador) 2-para-4.

B.21 [10] <§§B.3, B.4> Dado o diagrama lógico a seguir para um acumulador, escreva sua implementação do módulo em Verilog. Considere um registrador acionado por transição e Rst assíncrono.



B.22 [20] <§§B.3, B.4, B.5> A Seção 3.3 apresenta a operação básica e possíveis implementações dos multiplicados. Uma unidade básica dessas implementações é uma unidade de shift e soma. Mostre uma implementação Verilog para essa unidade. Mostre como você pode usar essa unidade para montar um multiplicador de 32 bits.

B.23 [20] <§§B.3, B.4, B.5> Repita o Exercício B.22, mas para um divisor sem sinal, ao invés de um multiplicador.

B.24 [15] <§B.5> A ALU admite set on less than (s1t) usando apenas o bit de sinal do somador. Vamos experimentar uma operação set on less than usando os valores -7_{dec} e 6_{dec} . Para tornar mais simples acompanhar o exemplo, vamos limitar as representações binárias a 4 bits: 1001_{bin} e 0110_{bin} .

$$1001_{\text{bin}} - 0110_{\text{bin}} = 1001_{\text{bin}} + 1010_{\text{bin}} = 0011_{\text{bin}}$$

Esse resultado sugeriria que $-7 > 6$, que certamente é errado. Logo, temos que considerar o overflow na decisão. Modifique a ALU de 1 bit na [Figura B.5.10](#), na página B-26, para lidar com s1t corretamente. Faça suas mudanças em uma fotocópia dessa figura, para ganhar tempo.

B.25 [20] <§B.6> Uma verificação simples do overflow durante a adição é ver se o CarryIn para o bit mais significativo não é igual ao CarryOut do bit mais significativo. Prove que essa verificação é a mesma da [Figura B.3.2](#).

B.26 [5] <§B.6> Reescreva as equações da página B-35 para uma lógica carry lookahead para um somador de 16 bits usando uma nova notação. Primeiro, use os nomes para os sinais CarryIn dos bits individuais do somador. Ou seja, use c_4, c_8, c_{12}, \dots ao invés de C_1, C_2, C_3, \dots . Além disso, considere que $P_{i,j}$ signifique um sinal de propagação para os bits de i a j , e $G_{i,j}$ signifique um sinal de geração para os bits i a j . Por exemplo, a equação

$$C_2 = G_1 + (P_1 \cdot G_0) + (P_1 \cdot P_0 \cdot c_0)$$

pode ser escrita como

$$c_8 = G_{7,4} + (P_{7,4} \cdot G_{3,0}) + (P_{7,4} \cdot P_{3,0} \cdot c_0)$$

Essa notação mais geral é útil na criação de somadores mais largos.

B.27 [15] <§B.6> Escreva as equações para a lógica carry-lookahead para um somador de 64 bits usando a nova notação do Exercício B.26 e usando somadores de 16 bits como blocos de montagem. Inclua um desenho semelhante à [Figura B.6.3](#) na sua solução.

B.28 [10] <§B.6> Agora, calcule o desempenho relativo dos somadores. Suponha que o hardware correspondente a qualquer equação contendo apenas termos OR ou AND, como as equações para p_i e g_i na página B-40, utilize uma unidade de tempo T. As equações que consistem no OR de vários termos AND, como as equações para c_1, c_2, c_3 e c_4 na página B-31, usariam assim duas unidades de tempo, 2T. O motivo é que é necessário T para produzir os termos AND e depois um T adicional para produzir o resultado

do OR. Calcule os números e a razão de desempenho para somadores de 4 bits para o carry por ondulação e carry lookahead. Se os termos nas equações forem ainda mais definidos por outras equações, então some os atrasos apropriados para essas equações intermediárias, e continue recursivamente até que os bits de entrada reais do somador sejam usados em uma equação. Inclua um desenho de cada somador rotulado com os atrasos calculados e o caminho do atraso no pior caso destacado.

B.29 [15] <§B.6> Este exercício é semelhante ao Exercício B.28, mas desta vez calcule as velocidades relativas de um somador de 16 bits usando apenas o carry por ondulação, carry por ondulação de grupos de 4 bits que usam carry lookahead, e o esquema de carry lookahead na página B-30.

B.30 [15] <§B.6> Este exercício é semelhante aos Exercícios B.28 e B.29, mas desta vez calcule as velocidades relativas de um somador de 64 bits usando apenas o carry por ondulação, carry por ondulação de grupos de 4 bits que usam carry lookahead, carry por ondulação de grupos de 16 bits que usam carry lookahead, e o esquema de carry lookahead do Exercício B.27.

B.31 [10] <§B.6> Ao invés de pensar em um somador como um dispositivo que soma dois números e depois une os carries, podemos pensar no somador como um dispositivo de hardware que pode somar três entradas (a_i, b_i, c_i) e produzir duas saídas ($s, c_i + 1$). Ao somar dois números, há pouco que podemos fazer com essa observação. Quando estamos somando mais de dois operandos, é possível reduzir o custo do carry. A ideia é formar duas somas independentes, chamadas S' (bits de soma) e C' (bits de carry). Ao final do processo, precisamos somar C' e S' usando um somador normal. Essa técnica de atrasar a propagação de carry até o final de uma soma de números é chamada *adição carry save*. O desenho em bloco no canto inferior direito da [Figura B.14.1](#) mostra a organização, com dois níveis de somadores carry save, conectados por um único somador normal. Calcule os atrasos para somar quatro números de 16 bits usando somadores carry lookahead completos contra o carry save com um somador carry lookahead formando a soma final. (A unidade de tempo T no Exercício B.28 é a mesma.)

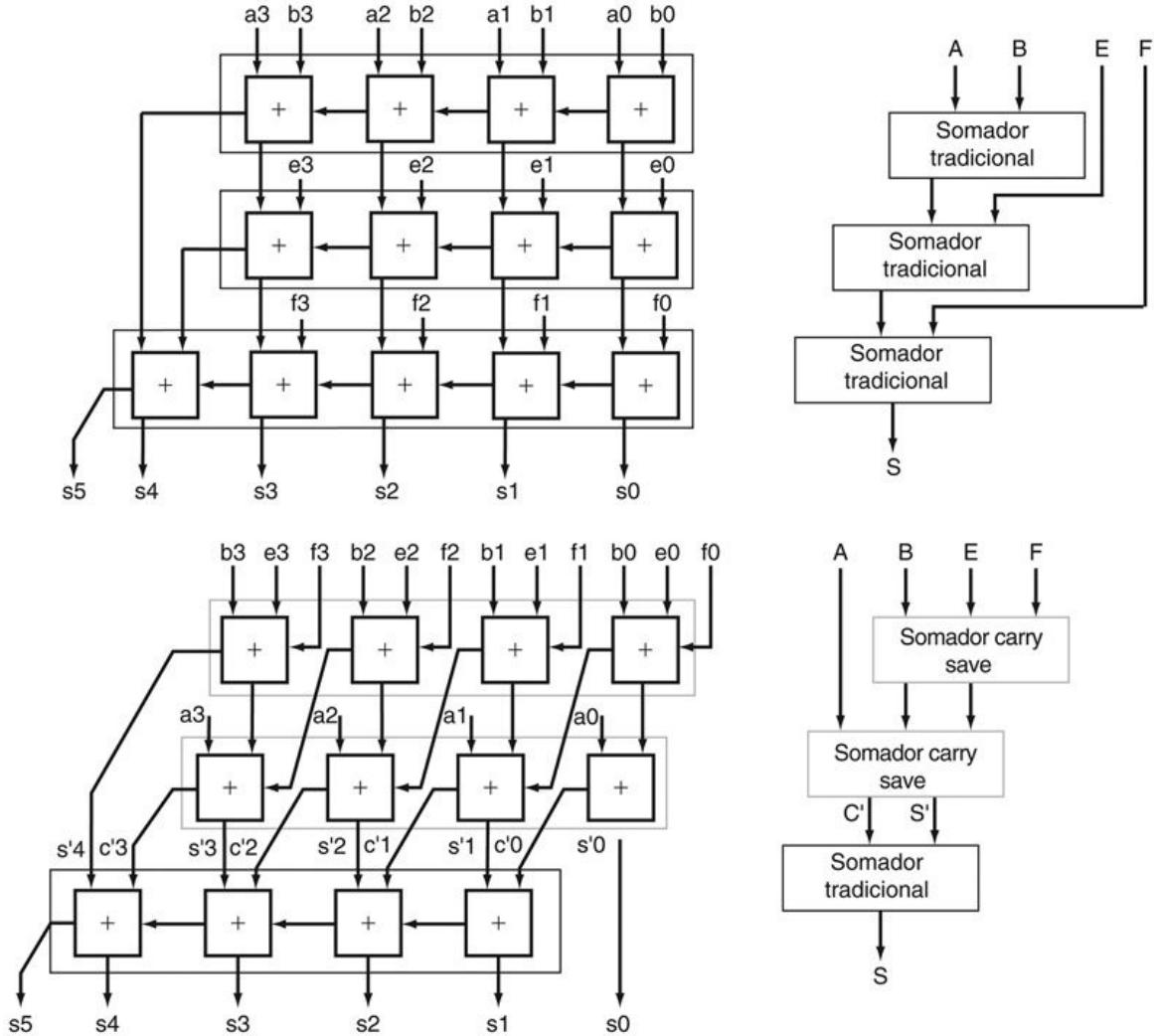


FIGURA B.14.1 Carry por ondulação tradicional e adição carry save de quatro números de 4 bits.

Os detalhes aparecem à esquerda, com os sinais individuais em minúsculas, e os blocos correspondentes de nível superior estão à direita, com sinais coletivos em maiúsculas. Observe que a soma de quatro números de n bits pode gerar $n + 2$ bits.

B.32 [20] <§B.6> Talvez o caso mais provável de somar muitos números ao mesmo tempo em um computador, seria quando se tenta multiplicar mais rapidamente usando muitos somadores para somar muitos números em um único ciclo de clock. Comparado com o algoritmo de multiplicação do Capítulo 3, um esquema carry save com muitos somadores poderia multiplicar mais de 10 vezes mais rápido. Esse exercício estima o custo e a velocidade para multiplicar dois números de 16 bits positivos. Suponha que você tenha 16 termos intermediários $M_{15}, M_{14}, \dots, M_0$, chamados *produtos*

parciais, que contêm o resultado dos bits do multiplicando AND bits multiplicadores m15, m14, ..., m0. A ideia é usar somadores carry save para reduzir os n operandos em $2n/3$ em grupos paralelos de três, e fazer isso repetidamente até que você obtenha dois números grandes para somar com um somador tradicional. Primeiro, mostre a organização em blocos dos somadores carry save de 16 bits para somar esses 16 termos, mostrados à direita na [Figura B.14.1](#). Depois, calcule os atrasos para somar esses 16 números. Compare esse tempo com o esquema de multiplicação iterativo no [Capítulo 3](#), mas considere apenas 16 iterações usando o somador de 16 bits que tem carry lookahead completo, cuja velocidade foi calculada no Exercício B.29.

B.33 [10] <§B.6> Existem ocasiões em que queremos somar uma coleção de números. Suponha que queiramos somar quatro números de 4 bits (A, B, E, F) usando somadores completos de 1 bit. Vamos ignorar o carry lookahead por enquanto. Você provavelmente conectararia os somadores de 1 bit na organização no topo da [Figura B.14.1](#). Abaixo da organização tradicional está uma nova organização dos somadores completos. Tente somar quatro números usando as duas organizações para se convencer que você obtém a mesma resposta.

B.34 [5] <§B.6> Primeiro, mostre a organização em bloco dos somadores carry save de 16 bits para somar esses 16 termos, como mostra a [Figura B.14.1](#). Suponha que o atraso de tempo por cada somador de 1 bit seja 2T. Calcule o tempo da soma de quatro números de 4 bits para a organização no tipo *versus* a organização na parte inferior da [Figura B.14.1](#).

B.35 [5] <§B.8> Normalmente, você esperaria que, dado um diagrama de tempo contendo uma descrição das mudanças que ocorrem em uma entrada de dados D e uma entrada de clock C (como nas Figuras B.8.3 e B.8.6), haveria diferenças entre as formas de onda de saída (Q) para um latch D e flip-flop D. Em uma sentença ou duas, descreva as circunstâncias (por exemplo, a natureza das entradas) para as quais não haveria qualquer diferença entre as duas formas de onda de saída.

B.36 [5] <§B.8> A [Figura B.8.8](#) ilustra a implementação do banco de registradores para o caminho de dados do MIPS. Imagine que um novo banco de registradores deva ser montado, mas que haja somente dois registradores e apenas uma porta de leitura, e que cada registrador tenha apenas 2 bits de dados. Redesenhe a [Figura B.8.8](#) de modo que cada fio no seu diagrama corresponda a somente 1 bit de dados (diferente do diagrama

na [Figura B.8.8](#), em que alguns fios são de 5 bits e alguns fios são de 32 bits). Redesenhe os registradores usando flip-flops D. Você não precisa mostrar como implementar um flip-flop D ou um multiplexador.

- B.37** [10] <§B.10> Um amigo gostaria que você montasse um “olho eletrônico” para usar como um dispositivo de segurança falso. O dispositivo consiste em três luzes alinhadas em sequência, controladas pelas saídas Esquerda, Meio e Direita, que, se ativadas, indicam que uma luz deve ser acesa. Somente uma luz está acesa de cada vez, e a luz se “move” da esquerda para a direita e depois da direita para a esquerda, o que afugenta os ladrões, que acreditam que o dispositivo está monitorando sua atividade. Desenhe uma representação gráfica para a máquina de estados finitos usada para especificar o olho eletrônico. Observe que a velocidade do movimento do olho será controlada pela velocidade do clock (que não deverá ser muito grande) e que basicamente não existem entradas.
- B.38** [10] <§B.10> Atribua números de estado aos estados da máquina de estados finitos que você construiu para o Exercício B.37 e escreva um conjunto de equações lógicas para cada uma das saídas, incluindo os bits do estado seguinte.

- B.39** [15] <§§B.2, B.8, B.10> Construa um contador de 3 bits usando três flip-flops D e uma seleção de portas. As entradas deverão consistir em um sinal que retorna o contador a 0, chamado *reset*, e um sinal para incrementar o contador, chamado *inc*. As saídas deverão ser o valor do contador. Quando o contador tiver valor 7 e for incrementado, ele deverá retornar e se tornar 0.
- B.40** [20] <§B.10> Um *código cinza* é uma sequência de números binários com a propriedade de que não mais de 1 bit muda ao passar de um elemento da sequência para o outro. Por exemplo, aqui está um código cinza binário de 3 bits: 000, 001, 011, 010, 110, 111, 101 e 100. Usando três flip-flops D e uma PLA, construa um contador de código cinza de 3 bits que tem duas entradas: *reset*, que define o contador como 000, e *inc*, que faz o contador seguir para o próximo valor na sequência. Observe que o código é cíclico, de modo que o valor após 100 na sequência é 000.

- B.41** [25] <§B.10> Queremos acrescentar uma luz amarela ao nosso exemplo de semáforo da página B-55. Faremos isso alterando o clock para 0,25Hz (um tempo de ciclo de clock de 4 segundos), que é a duração de uma luz amarela. Para impedir que as luzes verde e vermelha se repitam muito rapidamente, acrescentamos um temporizador de 30 segundos. O temporizador tem uma única entrada, chamada *TimerReset*, que reinicia o

timer, e uma única saída, chamada *TimerSignal*, que indica que o período de 30 segundos expirou. Além disso, temos que redefinir os sinais de trânsito para incluir o amarelo. Fazemos isso definindo dois sinais de saída para cada luz: verde e amarelo. Se a saída NSgreen estiver ativada, a luz verde está acesa; se a saída NSyellow estiver ativada, a luz amarela está acesa. Se os dois sinais estiverem desativados, a luz vermelha está acesa. *Não ative os sinais verde e amarelo ao mesmo tempo*, pois os motoristas americanos certamente ficarão confusos, mesmo que os motoristas europeus entendam o que isso significa! Desenhe a representação gráfica para a máquina de estados finitos para esse controlador melhorado. Escolha nomes para os estados que sejam *diferentes* dos nomes das saídas.

B.42 [15] <§B.10> Escreva as tabelas de próximo estado e função de saída para o controlador de semáforo descrito no Exercício B.41.

B.43 [15] <§§B.2, B.10> Atribua os números de estado aos estados no exemplo de semáforo do Exercício B.41 e use as tabelas do Exercício B.42 para escrever um conjunto de equações lógicas para cada uma das saídas, incluindo as saídas do estado seguinte.

B.44 [15] <§§B.3, B.10> Implemente as equações lógicas do Exercício B.43 como uma PLA.

Respostas das Seções “Verifique você mesmo”

§B.2, página B-6: Não. Se $A = 1$, $C = 1$, $B = 0$, o primeiro é verdadeiro, mas o segundo é falso.

§B.3, página B-14: C.

§B.4, página B-17: Todos são exatamente iguais.

§B.4, página B-20: $A = 0$, $B = 1$.

§B.5, página B-30: 2.

§B.6, página B-37: 1.

§B.8, página B-47: c.

§B.10, página B-58: b.

§B.11, página B-62: b.

Índice

Nota: As referências de página precedidas por uma única letra com hífen referem-se aos apêndices.

A

abstrações

interface hardware/software, [18](#)

para simplificar o projeto, [9](#)

princípio, [17](#)

acertos de cache, [388](#)

acerto sob perda, [412](#)

acrônimos, [7](#)

add (Add), [55](#)

add.d (FP Add Double), [A-55](#)

addiu (Add Imm.Unsigned), [103](#)

add sem sinal, instrução, [157](#)

addi (Add Immediate), [55](#)

add.s (FP Add Single), [A-55](#)

addu (Add Unsigned), [55](#)

adição, [155–159](#), *Ver também Aritmética*

binária, [155–156](#)

instruções, [A-39](#)

operандos, 156
ponto flutuante, 177–180, 185, A-55–56
significandos, 177
velocidade, 159
adição de ponto flutuante, 177–180
binária, 178
diagrama de blocos de unidade aritmética, 181
etapas, 177–178
ilustração, 179
instruções, 185, A-55–56
Advanced Vector Extensions (AVX), 198, 200
álgebra Booleana, B-4
algoritmo de divisão, 166
algoritmo de multiplicação, 162
algoritmos de classificação, 123
aliasing, 389
 alocação de espaço
 na heap, 90–92
 na pilha, 90
ALU, bloco de controle, 232
ALU, controle, 229–231, *Ver também* Unidade Lógica e Aritmética (ALU)
 bits, 230
ALU de 1 bit, B-20–23, *Ver também* Unidade lógica e aritmética (ALU)
 CarryOut, B-21
 ilustração, B-24
 para bit mais significativo, B-26

realizando AND, OR e adição, [B-24](#), [B-26](#)
somador, [B-21](#)
unidade lógica para AND/OR, [B-20](#)

ALU de 32 bits, [B-23–30](#), *Ver também* Unidade lógica e aritmética (ALU)
ajustando ao MIPS, [B-24–28](#)
com 32 ALUs de 1 bit, [B-23](#)
de 31 cópias de ALU de 1 bit, [B-27](#)
definição em Verilog, [B-28–30](#)
ilustração, [B-28](#)
somador de carry por ondulação, [B-23](#)

ALUOp, [230](#)
bits, [230](#), [231](#)
sinal de controle, [232](#)

Amazon Web Services (AWS), [372](#)

AMD64, [132](#), [197](#)

AMD Opteron X4 (Barcelona), [475](#), [476](#)

andi (And Immediate), [55](#)

AND, operação, [76](#), [A-40](#), [B-4](#)

AND, portas, [B-9](#)

Annual Failure Rate (AFR), [366](#)
versus. MTTF de discos, [367](#)

antidependência, [295](#)

antifuse, [B-63](#)

Apple iPad 2 A1395, [16](#)
circuito integrado do processador, [17](#)
placa lógica, [17](#)

Application Binary Interface (ABI), [18](#)
aritmética, [153–208](#)
 adição, [155–159](#)
 adição e subtração, [155–159](#)
 divisão, [164–170](#)
 faláncias e armadilhas, [202–204](#)
 multiplicação, [159–164](#)
 paralelismo de subword, [195–196](#)
 paralelismo de subword e multiplicação matricial, [198–201](#)
 paralelismo e, [195–196](#)
 ponto flutuante, [171–195](#)
 Streaming SIMD Extensions e extensões avançadas de vetor no x86, [197–198](#)
 subtração, [155–159](#)
armadilhas, *Ver também* faláncias
 aritmética, [202–204](#)
 associatividade, [418](#)
 avaliação de processador fora da ordem, [418](#)
 definição, [41](#)
 desenvolvimento de software com multiprocessadores, [488](#)
 endereços de word sequenciais, [140](#)
 extensão do espaço de endereços, [418](#)
 hierarquias de memória, [417–421](#)
 ignorando o comportamento do sistema de memória, [417](#)
 implementação de VMM, [420–421](#)
 pipelining, [310–311](#)
 ponteiro para variáveis automáticas, [140](#)

simulando cache, 417
subconjunto de equação de desempenho, 42–43
ARM Cortex-A8, 217, 302–303
 caches no, 412
 desempenho, 413–415
 especificação, 302
 hardware de TLB para, 411
 hierarquias de memória, 411–415
 taxas de falha de cache de dados para, 413
 tradução de endereço para, 411
ARM, instruções, 127
 cálculos, 127–128
 campo de condição, 284
 campo imediato de 12 bits, 130
 características, 130
 comparação e desvio condicional, 127–130
 formatos, 130
 loads e stores de bloco, 130
 lógicas, 130
 modos de endereçamento, 127
 registrador-registrador, 128
 semelhanças com o MIPS, 128
 transferência de dados, 128
ARMv7, 53
ARMv8, 138–139
arquitetos de computador, 9–10

abstração para simplificar projeto, 9
caso comum rápido, 9
confiança via redundância, 10
hierarquia de memórias, 10
lei de Moore, 9
paralelismo, 9
pipelining, 9
predição, 10

arquitetura do conjunto de instruções
ARM, 127
cálculo de endereço de desvio, 225
definição, 18, 44
mantendo, 44
proteção, 374
suporte para máquina virtual, 373–374

arquivos executáveis, A-3
definição, 110
produção do link-editor, A-15

arquivos-fonte, A-2

arquivos objeto, 109, A-2
cabeçalho, 109, A-9
definição, A-7
formato, A-9–10
informação de depuração, 108
informação de relocação, 109
link-edição, 109–112

segmento de dados estático, 109
segmento de texto, 109
tabela de símbolos, 109, 110
arrays, 364
elementos lógicos, B-14
múltiplas dimensões, 192
ponteiros *versus*, 123–127
procedimentos para definir como zero, 124
arredondamento, 192
bits, 194
com dígitos de guarda, 192
modos IEEE 754, 192
preciso, 192
ASCII
definição, 92
números binários *versus*, 93
representação de caracteres, 92
símbolos, 95
assembler, diretivas, A-5
assemblers, 108–109, A-7–13
aceitação de números, 109
arquivo-objeto, 109
código assembly condicional, A-13
definição, 12, A-2
função, 109, A-7
informações de relocação, A-9, A-10

- macros, [A-2](#), [A-11–13](#)
 - pseudoinstruções, [A-13](#)
 - tabela de símbolos, [A-8](#)
 - velocidade, [A-9](#)
 - associatividade
 - aumentando, [358](#)
 - aumentando o grau, [353](#), [398](#)
 - conjunto, tamanho de tag *versus*, [358](#)
 - em caches, [354](#)
 - atalho de negação, [66](#)
 - atalho de verificação de limites, [82](#)
 - atraso no pior caso, [240](#)
 - atraso rotacional, *Ver latência rotacional*
 - atribuição em bloco, [B-18](#)
 - atribuição sem bloqueio, [B-18](#)
 - automóveis, aplicação de computador em, [3](#)
 - Average Memory Access Time (AMAT), [352](#)
 - calculando, [352](#)
- ## B
- backpatching, [A-9](#)
 - balanceamento de carga, [441–442](#)
 - bancos de registradores, [224](#), [228](#), [B-40](#), [B-43–45](#)
 - com duas portas de leitura/uma porta de escrita, [B-44](#)
 - definição, [224](#), [B-40](#), [B-43](#)
 - implementação de duas portas de leitura, [B-44](#)

implementação de porta de escrita, [B-45](#)
no Verilog comportamental, [B-46](#)
únicos, [228](#)

barramentos, [B-14](#)

benchmarks, [471–473](#)

- definição, [39](#)
- Linpack, [471](#)
- multicores, [458–464](#)
- multiprocessador, [471–473](#)
- NAS paralelos, [472](#)
- pacote PARSEC, [473](#)
- paralelos, [472](#)
- SPEC CPU, [39–40](#)
- SPEC power, [40–41](#)
- SPECCrate, [471–472](#)
- Stream, [479](#)

beq (Branch On Equal), [55](#)

bge (Branch Greater Than or Equal), [109](#)

bgt (Branch Greater Than), [109](#)

bibliotecas de programas, [A-2](#)

big-endian, ordem de byte, [60](#), [A-33](#)

bit de modificação, [382](#)

bit de referência, [381](#)

bit de sinal, [65](#)

bit de validade, [338](#)

bit mais significativo

ALU de 1 bit para, [B-26](#)
bits
ALUOp, [230](#), [231](#)
arredondamento, [194](#)
definição, [11](#)
guarda, [194](#)
modificados, [382](#)
padrões, [194–195](#)
referência, [381](#)
sinal, [64](#)
sticky, [194](#)
válidos, [335](#)
bits de dados, [369](#)
bits menos significativos, [B-25](#)
definição, [64](#)
bits sticky, [194](#)
bloco básico, [81](#)
Blocos
combinacionais, [B-3](#)
dados válidos, [338](#)
definição, [329](#)
encontrando, [399](#)
estado, [B-3](#)
estratégias de posicionamento, [353](#)
estratégias de substituição, [400](#)
exploração da localidade espacial, [342](#)

loads/stores, [130](#)
locais de posicionamento, [398–399](#)
localizando na cache, [356–357](#)
palavras múltiplas, mapeando endereços a, [342](#)
posicionamento flexível, [352–353](#)
seleção de substituição, [358](#)
taxa de perda, [342](#)
usados menos recentemente (LRU), [358](#)

blocos de threads, [463](#)

bloqueio de cache e multiplicação matricial, [415–416](#)

blt (Branch Less Than), [108](#)

bne (Branch On Not Equal), [55](#)

bolhas, [275](#)

branch delay, slots

- definição, [282](#)
- escalonamento, [283](#)

branch equal, [278](#)

branch-on-equal, instrução, [237](#)

Bubble Sort, [122](#)

buffer de frame, [14](#)

buffer de renovação de rastreio, [14](#)

buffer em três estados, [B-48](#)

buffers de armazenamento, [300](#)

buffers de escrita

- cache write-back, [346](#)
- definição, [345](#)

stalls, 349
buffers de reordenação, 300
busca, 382
busca e incremento indivisíveis, 107
bytes
 endereçamento, 60
 ordem, 60, A-33

C

cabeçalho de leitura-escrita, 333
caches, 336–349, *Ver também* blocos
 acessando, 338–341
 associativas em conjunto, 352
 associatividade, 354–356
 bits, 341
 bits necessários, 341
 campo de tag, 340
 definição, 17, 336
 divisão, 348
 escritas, 344–346
 fisicamente endereçadas, 388
 fisicamente indexadas, 388
 fisicamente marcadas, 388
 FSM para controlar, 403–404
 ilustração do conteúdo, 339
 inconsistentes, 344

índice, 340

Intrinsics FastMATH, exemplo, 346–349

locais, 337

localizando blocos, 356–357

mapeadas diretamente, 336, 337, 342, 352

memória virtual e integração de TLB, 385–386

multinível, 349, 359

não bloqueantes, 412

no ARM Cortex-A8, 412

no Intel Core i7, 412

primárias, 359, 365

resumo, 348

secundárias, 359, 365

simulando, 417

tamanho, 341

totalmente associativas, 352

vazias, 338–339

virtualmente endereçadas, 388

virtualmente indexadas, 388

virtualmente marcadas, 388

write-back, 345, 346, 401

write-through, 344, 346, 400

caches associativos em conjunto, 352, *Ver também Caches*

duas vias, 353

escolha, 399

estratégias de substituição de bloco, 400

falhas, 354–356
local de bloco da memória, 352
 n vias, 352
partes de endereço, 356
quatro vias, 353, 356
caches de mapeamento direto, *Ver também caches*
comparador único, 356
definição, 336, 352
escolha, 399
falhas, 354
ilustração, 337
local de bloco de memória, 352
número total de bits, 341
partes de endereço, 356
caches endereçadas fisicamente, 388
caches multinível, *Ver também caches*
complicações, 364
definição, 349, 364
desempenho, 359
penalidade de falta, reduzindo, 359
resumo, 365–366
caches repartidas, 348
caches sem bloqueio, 301, 412
caches totalmente associativas, *Ver também Caches*
definição, 352
escolha, 399

estratégias de substituição de bloco, 400
local de bloco da memória, 352
perdas, 356

caches virtualmente endereçadas, 388

caches write-back, *Ver também caches*

- buffers de write, 346
- complexidade, 346
- definição, 345, 401
- stalls, 349
- vantagens, 401

cálculo de execução/endereço

- instrução load, 258
- instrução store, 258
- linha de controle, 263

callee, 85, 86

caller, 85

caminho de dados de desvio

- ALU, 225
- operações, 225

caminhos de dados

- definição, 17
- desvio, 225
- dois despachos estáticos, 294
- em operação para instrução branch-on-equal, 237
- em operação para instrução load, 236
- em operação para instrução tipo R, 235

montagem, 223–229
operação, 234–238
para arquitetura MIPS, 228
para instrução jump, 239
para instruções de busca, 224
para instruções de memória, 226
para instruções tipo R, 226, 234–235
para resolução de hazard via forwarding, 272
pipeline, 252–265
projeto, 223
caminhos de dados de ciclo único, *Ver também* caminhos de dados
execução de instrução, 254
ilustração, 253
caminhos de dados em pipeline, 252–265
com sinais de controle, 263–265
com sinais de controle conectados, 267
corrigidos, 260
ilustração, 254
nos estágios da instrução load, 260
campo de condição, 284
campos
definição, 71
MIPS, 71–72
nomes, 71
registrar Cause, A-26, A-27
registrar de Status, A-26, A-27

caractéres

Caracteres

- em Java, [95–97](#)
- representação ASCII, [92](#)
- carga, [A-15](#)
- carregadores, [112](#)
- carry lookahead, [B-30–37](#)
 - ALUs de 4 bits usando, [B-36](#)
 - analogia de encanamento, [B-33, B-34](#)
 - rápido com hardware “infinito”, [B-30–31](#)
 - rápido com primeiro nível de abstração, [B-31–32](#)
 - rápido com segundo nível de abstração, [B-32–37](#)
 - resumo, [B-37](#)
 - somador, [B-31](#)
 - velocidade de ripple carry *versus*, [B-36](#)
- carry rápido
 - com hardware “infinito”, [B-30–31](#)
 - com primeiro nível de abstração, [B-31–32](#)
 - com segundo nível de abstração, [B-32–37](#)
- caso comum rápido, [9](#)
- Central Processing Unit (CPU), *Ver também* processadores
- coprocessador 0, [A-25–26](#)
- definição, [16](#)
- desempenho, [28–29](#)
- equação de desempenho clássica, [30–33](#)
- medidas de tempo, [28–29](#)
- tempo, [349](#)
- tempo de execução, [27, 28–29](#)

tempo do sistema, [27](#)
tempo do usuário, [27](#)
centros de dados, [5](#)
chamadas ao sistema, [A-33–34](#)
carga, [A-33](#)
código, [A-33–34](#)
definição, [389](#)
chamadas de procedimento
 convenção, [A-17–25](#)
 exemplos, [A-20–25](#)
 frame, [A-18](#)
 preservação por, [89](#)
chips, [16, 21, 22](#)
 processo de fabricação, [22](#)
ciclos de clock
 atraso no pior caso, [240](#)
 definição, [28](#)
 número de registradores, [57](#)
 stall de memória, [349](#)
ciclos de clock de stall da memória, [349](#)
ciclos de clock por instrução (CPI), [29, 249](#)
 dois níveis de caching, [359](#)
 um nível de caching, [359](#)
ciclos de stall de escrita, [350](#)
ciclos de stall de leitura, [349](#)
cilindro, [333](#)

circuitos integrados (ICs), [16](#) Ver também *chips específicos*

custo, [22](#)

definição, [21](#)

processo de manufatura, [22](#)

Very Large-Scale (VLSIs), [21](#)

classificação de memória, [333](#)

C, linguagem

algoritmos de classificação, [123](#)

atribuição, compilando para MIPS, [56–57](#)

compilando, [127](#)

compilando atribuição com registradores, [58](#)

compilando loops while, [80](#)

hierarquia de tradução, [108](#)

tradução para linguagem assembly do MIPS, [56](#)

variáveis, [89](#)

clocking em duas fases, [B-61](#)

clocking sensível ao nível, [B-60–61](#)

definição, [B-60](#)

duas fases, [B-60](#)

clocks, [B-38–40](#)

especificação, [B-46](#)

inclinação, [B-60](#)

no projeto disparado por transição, [B-59](#)

sistema síncrono, [B-38–39](#)

transição, [B-38, B-40](#)

clusters

definição, 25, 437
isolamento, 465
organização, 499

CMOS (Complementary Metal Oxide Semiconductor), 34

codificação

- função lógica da ROM, B-12
- instrução de ponto flutuante, 187
- instrução MIPS, 72, 103, A-37
- instrução x86, 136–137

código de detecção de erro, 368

código de função, 71

código de máquina, 70

coerência de cache, 407–410

- coerência, 407
- consistência, 407
- esquemas de imposição, 408–409
- migração, 408
- problema, 407, 408, 410
- protocolo de snooping, 409–410
- protocolos, 409
- replicação, 409

commit na ordem, 298

comparação e troca indivisível, 107

comparações, 81

- com sinal *versus* sem sinal, 82
- operandos constantes, 81

compartilhamento falso, 410

compilação

- C, instruções de atribuição, 56–57
- C, linguagem, 80–81, 127
- if-then-else, 79
- loops while, 80–81
- procedimentos, 85, 88–89
- procedimentos recursivos, 88–89
- programas de ponto flutuante, 188–191

compiladores, 107–108

- criação de desvio, 80
- definição, 11
- especulação, 292
- função, 11, 107–108, A-5
- Just In Time (JIT), 115
- otimização, 123
- produção em linguagem de máquina, A-6, A-7

complemento a um, 69, B-24

computação em grade, 468

computação em nuvem, 468

- definição, 5

computadores

- aplicações, 3
- aritmética, 153–208
- classes de aplicação, 3–5
- componentes, 14, 154

desktop, 3
embutidos, 4, A-6
era pós-PC, 5–6
medida de projeto, 45
organização de componente, 14
princípios, 74
representação da instrução, 69–75
revolução da informação, 3
servidores, 3
computadores desktop, definição, 3
computadores embutidos, 4
definição, A-6
projeto, 4
requisitos da aplicação, 4
computadores pessoais (PCs), 5
definição, 3
comunicação, 19–20
reduzindo o overhead, 37–38
conceito de programa armazenado, 54
como princípio do computador, 74
ilustração, 75
princípios, 141
confiabilidade, 366
conjunto de trabalho, 397
conjuntos de instruções, 207
ARM, 284

MIPS-32, [207](#)
MIPS, [53](#), [141](#), [206](#)
projeto para pipelining, [244](#)
Pseudo MIPS, [205](#)
x86, crescimento, [141](#)
contador de instruções, [30](#), [32](#)
contadores de programa (PCs), [223](#)
atualizações de instrução, [253](#)
definição, [85](#), [223](#)
exceção, [390](#), [392](#)
incrementando, [223](#), [225](#)
mudando com desvio condicional, [284](#)
Content Addressable Memory (CAM), [357](#)
controladores de cache, [410](#)
controle
 ALU, [229–231](#)
 desafio, [285](#)
 forwarding, [269](#)
 para instrução jump, [239](#)
 pipeline, [263–265](#)
 terminando, [238](#), [239](#)
controle em pipeline, [263–265](#), *Ver também* controle
 especificando, [263](#)
 ilustração do esboço, [277](#)
 linhas de controle, [263](#), [265](#)
coprocessadores, [A-25–26](#)

definição, 192
instruções move, A-53–54
cores (núcleos)
definição, 36
número por chip, 36
correção de erro, B-53–55
corrida, B-59
CPU, 7
CUDA, ambiente de programação, 459

D

dados estáticos
como dados dinâmicos, A-16
definição, A-15
segmento, 91
decisão adiada, 249
decodificadores, B-6
dois níveis, B-53
decodificando linguagem de máquina, 102–105
defeito, 22
DeMorgan, teoremas, B-8
dependência de nome, 295
dependências
detecção, 269–270
entre registradores de pipeline, 270
entre registradores de pipeline e entradas da ALU, 270

inserção de bolha, 275
nome, 295
sequência, 267
dependências em pipeline, 267
desafio de speed-up, 440–441
 balanceando a carga, 441–442
 maior problema, 441
desdobramento de loop
 definição, 295
 para pipelines de despacho múltiplo, 295
 renomeação de registrador, 295
desempenho, 24–30
 avaliando, 24
 componentes, 32
 CPU, 28–29
 definição, 24–27
 equação clássica da CPU, 30–33
 instrução, 29–30
 medição de tempo, 27
 medindo, 28–29
 melhorando, 29
 programa, 33
 razão, 25
 relativo, 25–27
 tempo de resposta, 25
 throughput, 25

usando equação, 30

desempenho da cache, 349–365

- calculando, 350
- impacto sobre desempenho do processador, 350
- tempo de acerto, 351–352

desempenho do programa

- compreendendo, 7
- elementos que afetam, 33

desempenho máximo de ponto flutuante, 474

desempenho relativo, 26–27

despacho múltiplo, 290–296

- desdobramento de loop, 295
- dinâmico, 291, 296–298
- escalonamento de código, 295
- estático, 291, 292–296
- pacotes de despacho, 292
- processadores, 290, 291
- vazão, 299

destino do desvio

- buffers, 284
- endereços, 225

desvio não tomado

- definição, 225
- suposição, 278

desvios adiados, 83, *Ver também desvios*

- definição, 226

limitações de escalonamento, 283
pipelines de cinco estágios, 22, 283–284
reduzindo, 278–279
solução do hazard de controle, 249

desvios condicionais
alterando o contador de programa com, 284
ARM, 127–130
compilando if-then-else em, 79
definição, 78
em loops, 100
endereçamento relativo ao PC, 99
implementação, 83

desvios em pipeline, 279

desvios incondicionais, 79

desvios, *Ver também* desvios condicionais
adiados, 83, 226, 249, 278–279, 282, 284
condição, 226
criação de compilador, 79
decisão, subindo, 278
endereçamento, 98–101
endereço de destino, 278
execução no estágio ID, 279
incondicionais, 79
pipeline, 278
terminando, 81

desvio tomado

definição, [225](#)
redução de custo, [278](#)
detecção de erro, [B-53](#)
DGEMM (Double precision General Matrix Multiply), [198](#), [308](#), [362](#), [484](#)
desempenho, [364](#)
versão C otimizada, [199](#), [200](#), [416](#)
versão de cache em bloco, [364](#)
diagramas de pipeline de múltiplo ciclo de clock, [260–261](#)
cinco instruções, [261](#)
ilustração, [262](#)
diagramas de pipeline de único ciclo de clock, [260–261](#)
ilustração, [262](#)
dicing, [22](#)
dies, [22](#), [22](#)
dígitos de guarda
arredondamento com, [192](#)
definição, [192](#)
diretivas de leiaute de dados, [A-10](#)
discos rígidos
definição, [19](#)
tempos de acesso, [19](#)
dispositivos de entrada, [14](#)
dispositivos de saída, [14](#)
distância de Hamming, [368](#)
div.d (divisão dupla em ponto flutuante), [A-57](#)
div (divisão), [A-40](#)

dividendo, [165](#)
divisão, [164–170](#)
 algoritmo, [167](#)
 dividendo, [165](#)
 divisor, [165](#)
divisão com sinal, [166–169](#)
divisor, [165](#)
div.s (divisão simples em ponto flutuante), [A-57](#)
divu (divisão sem sinal), [A-40](#), *Ver também* aritmética
 com sinal, [166–169](#)
 hardware, [164–166](#)
 hardware, versão melhorada, [168](#)
 instruções, [A-40](#)
 mais rápida, [169](#)
 no MIPS, [169](#)
 operandos, [165](#)
 ponto flutuante, [185](#), [A-57](#)
 quociente, [165](#)
 resto, [165](#)
 SRT, [169](#)
don't cares, [B-12–14](#)
exemplo, [B-12–14](#)
termo, [231](#)
Double Data Rate (DDR), [331](#)
Double Data Rate RAMs (DDRRAMs), [331–333](#), [B-53](#)
Dual Inline Memory Modules (DIMMs), [333](#)

Dynamically Linked Libraries (DLLs), [112–114](#)

definição, [112](#)

versão da link-edição de procedimento tardio, [113](#)

Dynamic Random Access Memory (DRAM), [331](#), [331–333](#), [B-51–53](#)

crescimento de capacidade, [21](#)

custo, [19](#)

decodificador de dois níveis, [B-53](#)

definição, [17](#), [B-51](#)

Double Date Rate (DDR), [331–333](#)

largura de banda externa, [349](#)

organização interna, [333](#)

síncrona (SDRAM), [331–333](#), [B-51](#), [B-53](#)

tamanho, [349](#)

transistor de passagem, [B-51](#)

único transistor, [B-52](#)

velocidade, [19](#)

E

Electrically Erasable Programmable Read-Only Memory (EEPROM), [333](#)

elementos

caminho de dados, [223](#), [226](#)

combinacionais, [220](#)

estado, [220](#), [222](#), [224](#), [B-38](#), [B-40](#)

memória, [B-40–47](#)

elementos combinacionais, [220](#)

elementos de estado

banco de registradores, [B-40](#)
clock e, [222](#)
definição, [220](#), [B-38](#)
em instruções de armazenamento/acesso, [224](#)
entradas, [221](#)
lógica combinacional e, [222](#)
elementos de memória, [B-40–47](#)
 clock, [B-40](#)
 desbloqueados, [B-41](#)
 DRAMs, [B-51–54](#)
 flip-flop, [B-41](#)
 flip-flop D, [B-41](#), [B-43](#)
 latch, [B-41](#)
 latch D, [B-41](#)
 SRAMs, [B-47–51](#)
 tempo de preparação, [B-43](#)
 tempo de suspensão, [B-43](#)
elementos do caminho de dados
 compartilhamento, [226](#)
 definição, [223](#)
endereçamento
 base, [101](#)
 deslocamento, [101](#)
 em jumps e desvios, [98–101](#)
 imediato, [101](#)
 immediatos de 32 bits, [98–101](#)

modos do MIPS, [101–102](#)
modos x86, [133](#)
pseudodireto, [101](#)
registrador, [101](#)
relativo ao PC, [99](#), [101](#)
endereço de retorno, [84](#)
endereços
 base, [59](#)
 byte, [59](#)
 definição, [58](#)
 immediatos de 32 bit, [98–101](#)
 memória, [66](#)
 virtuais, [375–377](#), [394](#)
endereços físicos, [375](#)
 espaço, [452](#), [455](#)
 mapeamento, [375–376](#)
endereços virtuais
 causando falhas de página, [393](#)
 definição, [375](#)
 mapeamento, [375–376](#)
 tamanho, [376](#)
entradas, [231](#)
E/S, [A-29–31](#)
 mapeada na memória, [A-29](#)
E/S, benchmarks, *Ver benchmarks*
escalonamento de pipeline dinâmico, [296–298](#)

buffer de reordenação, 300
conceito, 296–297
especulação baseada em hardware, 298
estação de reserva, 296–297
unidade de commit, 296–297
unidades primárias, 297

escritas

- cache write-back, 345, 346
- cache write-through, 345, 346
- complicações, 345
- custo, 397
- esquemas, 345
- memória virtual, 382
- tratamento, 344–346
- tratamento da hierarquia de memória, 400–401

espaço de endereços, 375, 377

- compartilhado, 454
- estendendo, 418
- físico isolado, 452
- ID (ASID), 391
- não mapeado, 394
- plano, 418
- virtual, 391

espaço de endereços plano, 418

especulação, 292–292

- baseada em hardware, 298

desempenho e, 292
implementação, 292
mecanismo de recuperação, 292
problemas, 292
estabilidade pela redundância, 10
estações de reserva
 colocando operandos em buffer, 297–298
 definição, 296–297
estado
 atribuição, B-57
 componentes lógicos, 221
 especificação, 378
 exceção, salvando/restaurando, 394
 no esquema de previsão de 2 bits, 282
estágio de acesso à memória
 instrução load, 258
 instrução store, 258
 linha de controle, 265
estágio de decodificação de instrução/leitura de banco de registradores
 instrução load, 253
 instrução store, 258
 linha de controle, 263
estágio do cálculo de execução ou endereço, 258
estágio e busca de instrução
 instrução load, 253
 instrução store, 258

linha de controle, 263
estágio EX
detecção de exceção de overflow, 287
instruções load, 258
instruções store, 258
Ethernet, 19
exabyte, 4
exceções, 285–290, A-25–29
associação, 290
caminho de dados com controles para tratamento, 288
definição, 157, 285
detectando, 285
estágio para salvar/restaurar, 394
estouro, 288
exemplo de computador em pipeline, 287
imprecisas, 290
instruções, A-60
interrupções *versus*, 285
motivos, 285–286
na arquitetura MIPS, 285–286
na implementação em pipeline, 286–290
PC, 390, 391–392
precisas, 290
resultado devido a estouro na instrução add, 289
tipos de evento, 285
exception enable, 392

Exception Program Counters (EPCs), [285](#)

captura de endereço, [290](#)

copiando, [158](#)

definição, [158, 286](#)

determinando o reinício, [285–286](#)

transferindo, [159](#)

exclusão mútua, [105](#)

exclusive OR (XOR), instruções, [A-43](#)

execução fora da ordem

complexidade do desempenho, [364](#)

definição, [298](#)

processadores, [301](#)

execução paralela, [105](#)

expansão

forte, [441, 443](#)

fraca, [441](#)

expoentes, [172–173](#)

extensão de sinal, [225](#)

atalho, [67](#)

definição, [66](#)

extensões de multimídia

vetor *versus*, [446–448](#)

F

facilidades, [A-10–13](#)

faláncias, *Ver também* armadilhas

adição imediata sem sinal, 200
aritmética, 202–204
baixa utilização usa pouca energia, 42
definição, 41
desempenho de pico, 488
deslocamento à direita, 202
importância da compatibilidade binária comercial, 140
instruções poderosas significam maior desempenho, 139
lei de Amdahl, 487
linguagem assembly para desempenho, 139–140
pipelining, 310–311
falhas do sincronizador, B-62
faltas de cache
cache associativa em conjunto, 354
cache de mapeamento direto, 353
cache totalmente associativa, 356
capacidade, 401
ciclos de clock de stall da memória, 349
compulsórias, 401
conflito, 402
definição, 343
etapas, 344
na cache write-through, 344
reduzindo com posicionamento de bloco flexível, 352–353
substituição de bloco, 400
tratamento, 344

faltas de página, 380, *Ver também* memória virtual
definição, 375
endereço virtual causando, 393, 394
para acesso a dados, 394
tratamento, 376, 391–396
falta sob falta, 412
fator de bloqueio, 363
Fermi, arquitetura, 459, 484
Field Programmable Devices (FPDs), B-63
Field Programmable Gate Arrays (FPGAs), B-63
flip-flops
definição, B-41
flip-flops D, B-41, B-43
flip-flops D, B-41, B-43
fluxo de instruções da esquerda para a direita, 252
formato de instrução tipo J, 98
formato I, 72
formato R, 232
definição, 72
operações da ALU, 225
formatos de instrução, 137
ARM, 130
definição, 70
instrução jump, 239
MIPS, 130
tipo I, 72

tipo J, 98
tipo R, 72, 231
x86, 137
forwarding, 266–277
ALU antes, 271
caminho de dados para resolução de hazard, 272
com duas instruções, 245
controle, 269
definição, 245
funcionamento, 268
multiplexadores, 272
múltiplos resultados, 248
registradores de pipeline antes, 271
representação gráfica, 246
frações, 172, 173
função do próximo estado, 405, B-55
definição, 405
funções de controle
definindo, 234
para implementação de único ciclo, 238
Fused-Multiply-Add (FMA), operação, 194

G

gather-scatter, 483
geração
definição, B-31

exemplo, [B-34](#)
super, [B-32](#)
gigabytes, [4](#)
Graphics Processing Units (GPUs), [458–464](#), *Ver também* computação GPU
arquitetura NVIDIA, [459–462](#)
caches multinível, [458](#)
como aceleradoras, [458](#)
definição, [39](#), [442](#)
memória, [459](#)
paralelismo, [459](#)
perspectiva, [462–464](#)
GTX, [247](#), [480–484](#)

H

halfwords, [96](#)
Hamming, Error Correction Code (ECC), [368–369](#)
calculando, [368–369](#)
Hamming, Richard, [368](#)
handler de exceção, [A-28–30](#)
definição, [A-27](#)
retorno dos, [A-29](#)
handlers
definição, [393](#)
perda de TLB, [392](#)
hardware
como camada hierárquica, [11](#)

linguagem, 11–13
operações, 54–57
procedimentos de suporte, 84–92
síntese, B-16
virtualizável, 373

hardware virtualizável, 373

hazard de dados de uso de load, 246, 278

hazards, 244–245, *Ver também* pipelining
 controle, 248–249, 277–285
 dados, 245, 266–277
 estruturais, 245, 259
 forwarding, 273

hazards de controle, 248–249, 277–285
 previsão de desvio como solução, 249
 previsão de desvio dinâmico, 280–283

processadores estáticos de despacho múltiplo, 293–294

redução branch delay, 278–279

resumo de pipeline, 284

simplicidade, 277

soluções, 249

stalls de pipeline como solução, 249

suposição de desvio não tomado, 278

técnica de decisão adiada, 249

hazards de dados, 245, 266–277, *Ver também* hazards
 forwarding, 245, 266–277
 stalls, 274–277

uso de load, 247, 278
hazards estruturais, 244, 259
heap
 alocando espaço, 90–92
 definição, 90
hierarquia de memória confiável, 366–371
 falha, definição, 366
hierarquia de memórias, 10, 477
 armadilhas, 417–421
 ARM Cortex-A8, 411–415
 bloco (ou linha), 329
 caches, 336–365
 definição, 328
 dependência de, 329
 desafios de projeto, 403
 desempenho de cache, 349–365
 diagrama de estrutura, 331
 estrutura, 328
 estrutura comum, 397–403
 explorando, 325–498
 Intel Core i7, 411–415
 memória virtual, 374–397
 múltiplos níveis, 328
 operação geral, 388–389
 paralelismo, 407–410
 parâmetros quantitativos de projeto, 397

pares de nível, 329
Redundant Arrays of Inexpensive Disks (RAID), 410
tempo de execução de programa, 365
variância, 365

I

identificadores de processo, 391
identificadores de tarefa, 391
ID, estágio
 execução de desvio, 279
 instruções load, 258
 instruções store, 255
IEEE 754, padrão de ponto flutuante, 173, 174, *Ver também* ponto flutuante
 modos de arredondamento, 192
If, instruções, 99
If-then-else, 79
impedimento de falha, 367
implementação de único ciclo
 definição, 239
 desempenho com pipeline *versus*, 242
 execução sem pipeline *versus* execução com pipeline, 243
 função de controle, 238
 não uso da, 240
 penalidade, 240
índice fora dos limites, verificação, 82
informação de depuração, A-9

informação de relocação, [A-9](#), [A-10](#)
instruções, [51–143](#), *Ver também* instruções aritméticas, *Ver também* MIPS, *Ver também* operandos
add imediato, [62](#)
adição, [157](#), [A-39](#)
ARM, [127](#)
assembly, [57](#)
básicas, [205](#)
bloco básico, [81](#)
busca, [224](#)
campos, [69](#)
cientes da cache, [421](#)
codificação, [72](#)
como words, [53](#)
comparação, [A-44–45](#)
conversão, [A-56–57](#)
definição, [11](#), [53](#)
desempenho, [29–30](#)
deslocamento, [A-42–43](#)
desvio, [A-45–48](#)
desvio condicional, [78](#)
divisão, [A-40](#)
exceção e interrupção, [A-60](#)
flushing, [278](#), [279](#), [290](#)
fluxo da esquerda para a direita, [252](#)
immediatas, [62](#)
introdução, [53–54](#)

jump, 82, 84, A-48–49
load, 58, A-50–50
load ligado, 106
lógicas aritméticas, 223, A-39–43
manipulação de constantes, A-44
move condicional, 284
movimentação de dados, A-53–55
multiplicação, 164, A-40–41
negação, A-42
nop, 275
operações lógicas, 75–77
OR exclusivo, A-43
ponto flutuante, 185–187, A-55–60
ponto flutuante (x86), 197
referência à memória, 218
reiniciáveis, 394
representação no computador, 69–75
resto, A-42
retomando, 394
sequência de pipeline, 274
sinais eletrônicos, 69
store, 61, A-51–53
store condicional, 106
subtração, 157, A-43
tipo R, 224
tomada de decisão, 78–83

transferência de dados, 58
trap, A-49–50
vetor, 446
x86, 130–136

instruções aritméticas, *Ver também* Instruções lógicas, 223
MIPS, A-39–43
operandos, 57–63

instruções cientes da cache, 421

instruções de comparação, A-44–45
lista, A-44–45
ponto flutuante, A-56

instruções de conjunto, 81

instruções de conversão, A-56–57

instruções de deslocamento, 75, A-42–43

instruções de desvio, A-45–48
impacto do pipeline, 277
instrução jump *versus*, 239
lista de, A-46–48

instruções de flushing, 278, 279
definição, 279
exceções, 290

instruções de manipulação de constante, A-44

instruções de máquina, 70

instruções de movimentação de dados, A-53–55

instruções de movimento condicional, 284

instruções de negação, [A-42](#), [A-59](#)
instruções de ponto flutuante, [A-55–60](#)

- adição, [A-55–56](#)
- comparação, [A-56](#)
- conversão, [A-56–57](#)
- divisão, [A-57](#)
- load, [A-57–58](#)
- move, [A-58–59](#)
- multiplicação, [A-59](#)
- negação, [A-59](#)
- raiz quadrada, [A-59](#)
- store, [A-59](#)
- subtração, [A-60](#)
- truncamento, [A-60](#)
- valor absoluto, [A-55](#)

instruções de tipo R, [224](#)

- caminho de dados em operação para, [235](#)
- caminho de dados para, [234–235](#)

instruções de tomada de decisão, [78–83](#)

instruções de transferência de dados, *Ver também* instruções

- definição, [58](#)
- load, [58](#)
- offset, [59](#)
- store, [61](#)

instruções de trap, [A-49–50](#)

instruções imediatas, [62](#)

instruções jump, 225
 controle e caminho de dados, 239
 formato de instrução, 239
 implementando, 239
 instrução de desvio *versus*, 239
 lista de, A-48–49

instruções load, *Ver também* instruções store
 bloco, 130
 caminho de dados em pipeline, 260
 caminho de dados na operação, 236
 compilando, 61
 com sinal, 66
 definição, 58
 detalhes, A-50–51
 estágio EX, 258
 estágio ID, 255
 estágio IF, 255
 estágio MEM, 257
 estágio WB, 257
 halfword sem sinal, 96
 interligadas, 106, 107
 lista de, A-50–51
 load byte sem sinal, 66
 load half, 96
 load upper immediate, 97, 98
 ponto flutuante, A-57–58

registrador de base, [232](#)
sem sinal, [66](#)
unidade para implementação, [226](#)

instruções por ciclo de clock (IPC), [291](#)

instruções reiniciáveis, [392](#)

instruções store, *Ver também* instruções load
bloco, [130](#)
compilando com, [61](#)
condicionais, [106](#)
definição, [61](#)
dependência de instrução, [273](#)
detalhes, [A-52–53](#)
estágio EX, [259](#)
estágio ID, [255](#)
estágio IF, [255](#)
estágio MEM, [259](#)
estágio WB, [259](#)
lista de, [A-52–53](#)
ponto flutuante, [A-60](#)
registrador de base, [232](#)
unidade para implementar, [226](#)

Intel Core i7, [39–41](#), [217](#), [438](#), [480–484](#)
caches, [412](#)
desempenho, [413](#)
hardware de TLB para, [411](#)
hierarquias de memória, [411–415](#)

microarquitetura, 295
registradores arquiteturais, 304
SPEC CPU, benchmark, 39–40
SPEC power, benchmark, 40–41
tradução de endereço, 411

Intel Core i7, 920, 303–305
 microarquitetura, 304

Intel Core i7, 960
 benchmarking e rooflines, 480–484

Intel Core i7, pipelines, 301, 303–305
 componentes de memória, 305
 desempenho, 305–307
 desempenho do programa, 307
 especificação, 302

Intel x86, microprocessadores
 taxa de clock e potência, 34

intensidade aritmética, 474

intercalação, 349

intercalação de endereço, 333

interrupção de serviço, 366

interrupções
 definição, 157, 285
 exceções *versus*, 285
 imprecisas, 290
 instruções, A-60
 precisas, 290

tipos de evento, 285
vetorizadas, 286
interrupt enable, 392
Intrinsicsy FastMATH, processador, 346–349
caches, 346
processamento de leitura, 387
processamento write-through, 387
taxas de perda de dados, 348, 356
TLB, 385

J

jal (Jump And Link), [55](#)

Java

algoritmos de classificação, [123](#)

bytecode, [114](#)

caracteres, [95–97](#)

hierarquia de tradução, [114](#)

interpretando, [114, 127](#)

objetivos, [114](#)

programas, iniciando, [114–115](#)

strings, [95–97](#)

j (Jump), [55](#)

jr (Jump Register), [55](#)

Just In Time (JIT), compiladores, [115](#)

K

Karnaugh, mapas, [B-14](#)

kilobyte, [4](#)

L

LAPACK, [203](#)

largura de banda, [25–26](#)

bisseção, [467](#)

externa à DRAM, [349](#)

memória, [333, 349](#)

rede, 469

largura de banda da memória, 480, 488

largura de banda de bisseção, 467

latches

- definição, B-41
- latch D, B-41, B-41
- latches D, B-41, B-41

latência

- instrução, 311
- pipeline, 252
- uso, 294–295

latência de instrução, 311

latência de uso

- definição, 294–295
- uma instrução, 294–295

latência rotacional, 335

lbu (Load Byte Unsigned), 55

lei de Amdahl, 351, 440

- corolário, 41
- definição, 41
- e (AND), 55
- falácia, 487

Lei de Moore, 9, 331, 458

lhu (Load Halfword Unsigned), 55

lingote de cristal de silício, 22

linguagem Assembly, 12

definição, 12, 107
desvantagens, A-6–7
ilustração, 13
linguagens de alto nível *versus*, A-9
MIPS, 55, 73, A-34–60
ponto flutuante, 186
produção de, A-6
programas, 107
quando usar, A-5–6
traduzindo para linguagem de máquina, 73
linguagem de máquina, 12
decodificação, 102–105
definição, 12, 70, A-2
ilustração, 13
MIPS, 74
offset de desvio, 100
ponto flutuante, 186
SRAM, 17
traduzindo linguagem assembly MIPS para, 73
linguagem-fonte, A-5
linguagens de alto nível, 11–13, A-5
benefícios, 13
importância, 13
linguagens de descrição de hardware, *Ver também* Verilog
definição, B-15
usando, B-15–20

VHDL, [B-16](#)

linguagens de programação *Ver também linguagens específicas*

orientadas a objeto, [127](#)

variáveis, [57](#)

linguagens orientadas a objeto, *Ver também Java*

definição, [127](#)

linhas de controle

acesso à memória, [265](#)

ativadas, [234](#)

busca de instrução, [263](#)

configuração, [234](#)

decodificação de instrução/leitura de banco de registradores, [263](#)

execução/cálculo de endereço, [263](#)

no caminho de dados, [232](#)

três estágios finais, [265](#)

valores, [263](#)

write-back, escrita do resultado, [265](#)

linkagem de arquivos-objeto, [109–112](#)

link-editores, [109–112](#), [A-13–14](#)

arquivos executáveis, [110](#), [A-15](#)

definição, [109](#), [A-3](#)

etapas, [110](#)

ilustração da função, [A-14](#)

usando, [109–112](#)

Linpack, [471](#)

lista de sensitividade, [B-18](#)

little-endian, ordem de bytes, [A-33](#)

ll (Load Linked), [55](#)

load word, [58, 61](#)

localidade

espacial, [327, 330](#)

princípio, [327](#)

temporal, [327, 330](#)

localidade espacial, [327](#)

exploração de bloco grande, [342](#)

tendência, [331](#)

localidade temporal, [327](#)

tendência, [331](#)

locks, [453](#)

lógica

array programável (PAL), [B-63](#)

combinacional, [222, B-3, B-6–15](#)

componentes, [221](#)

dois níveis, [B-7–11](#)

equações, [B-5](#)

minimização, [B-14](#)

projeto, [220–223, B-1–64](#)

sequencial, [B-3, B-45–47](#)

lógica combinacional, [221, B-2, B-6–15](#)

arrays, [B-14–14](#)

decodificadores, [B-6](#)

definição, [B-3](#)

dois níveis, [B-7–11](#)
don't cares, [B-12–14](#)
multiplexadores, [B-7](#)
ROMs, [B-11–12](#)
Verilog, [B-18–20](#)
lógica de dois níveis, [B-7–11](#)
lógica sequencial, [B-3](#)
Lookup Tables (LUTs), [B-63](#)
loops, [80–81](#)
 desvios condicionais, [99](#)
 for, [123](#)
 previsão, [280–283](#)
 teste, [124, 125](#)
 while, compilando, [80–81](#)
loops for, [123](#)
lui (Load Upper Imm.), [55](#)
lwc1 (Load FP Single), [A-55](#)
lw (Load Word), [55](#)

M

macros
 definição, [A-2](#)
 exemplo, [A-11–13](#)
 uso, [A-11](#)
mapas de bits
armazenamento, [14](#)

definição, 14, 63
objetivo, 14
máquinas de estados finitos (FSMs), 395–407, B-54–58
atribuição de estado, 405, B-57
controladores, 406
estilo, 405
exemplo do semáforo, B-55–57
função de saída, B-55, B-56
função do próximo estado, 405, B-55
implementação de registrador de estado, B-58
Mealy, 405
Moore, 405
para controlador de cache simples, 406–407
síncronas, B-55
máquinas virtuais (VMs), 371–374
benefícios, 371
definição, A-32
ilusão, 396
melhoria do desempenho, 374
para melhoria da proteção, 371
simulação, A-32
suporte à arquitetura do conjunto de instruções, 373–374
matriz ativa, 14
Mealy, máquina, 404–406, B-55, B-58
Mean Time To Failure(MTTF), 366
melhorando, 367

versus AFR de discos, 367–368

megabyte, 4

memória

afinidade, 477

cache, 17, 336–348, 349–365

CAM, 357

definição, 17

DRAM, 17, 331–333, B-51–53

endereços, 66

flash, 19

GPU, 459

instruções, caminho de dados para, 226

largura de banda, 333, 348

leiaute, A-16

não volátil, 19

operandos, 58–59

principal, 19

SDRAM, 331–333

secundária, 19

somente leitura (ROM), B-11–12

SRAM, B-47–51

stalls, 350

tecnologias para criação, 20–24

uso, A-15–17

virtual, 374–397

volátil, 19

memória em disco, 333–335
memória flash, 333
 características, 19
 definição, 19
memória não volátil, 19
memória principal, 375, *Ver também* memória
 definição, 19
 endereços físicos, 375
 tabelas de página, 382
memória secundária, 19
memória virtual, 374–397, *Ver também* páginas
 escritas, 382
 faltas de página, 375, 380
 implementação da proteção, 389–391
 integração, 385–386
 mecanismo, 396
 motivações, 374–375
 resumo, 396
 segmentação, 377
 tradução de endereço, 375, 383–384
 virtualização, 396
memória volátil, 19
metaestabilidade, B-61
metodologia de clocking, 221–223, B-38
 disparada por transição, 221, B-38, B-59
 por previsibilidade, 221

sensível ao nível, [B-60–61](#)

metodologia de clocking disparada por transição, [221](#), [222](#), [B-38](#), [B-59](#)

aresta de subida/aresta de descida, [B-38](#)

clocks, [B-59](#)

desvantagens, [B-60](#)

ilustração, [B-40](#)

vantagem, [B-39](#)

métodos

estáticos, [A-15](#)

mfc0 (Move From Control), [A-54](#)

mfhi (Move From Hi), [A-54](#)

mflo (Move From Lo), [A-54](#)

microarquiteturas, [304](#)

Intel Core i7, [920](#), [304](#)

microprocessadores

mudança de projeto, [438](#)

multicore, [7](#), [36](#), [437–438](#)

migração, [408](#)

milhões de instruções por segundo (MIPS), [43](#)

mintermos

definição, [B-9](#)

MIPS-32, conjunto de instruções, [207](#)

MIPS, [55](#), [73](#), [A-34–60](#)

alocação de memória para programa e dados, [91](#)

campos, [71–72](#)

classes de instrução, [143](#)

codificação de instrução, 72, 103, A-37
compilando atribuição C complexa para, 56–57
compilando instruções de atribuição C para, 56
conjunto de instruções, 53, 142, 206
convenções de registrador, 92
CPU, A-35
despacho múltiplo estático, 293–295
divisão, 169
endereçamento para imediatos de 32 bits, 101–102
endereços de memória, 60
exceções, 285–286
formatos de instrução, 105, 130, A-38–39
FPU, A-35
instruções aritméticas, 54, A-39–43
instruções de comparação, A-44–45
instruções de desvio, A-45–48
instruções de manipulação de constante, A-44
instruções de ponto flutuante, 185–187
instruções jump, A-48–50
instruções lógicas, A-39–43
linguagem de máquina, 74
mapa de opcode, A-38
mapeamento de instrução assembly, 69–70
modos de endereçamento, A-34–36
multiplicação, 164
núcleo aritmético, 205

operандos, 55
pseudo, 205, 207
registradores de controle, 392
semelhanças do ARM, 128
sintaxe do assembler, A-36–38
suporte a diretiva do assembler, A-36–38
MIPS, conjunto de instruções básico, 208, *Ver também MIPS*
 ilustração da implementação, 220
 implementação, 217–220
 subconjunto, 217
 visão abstrada, 219
 visão geral, 218
MIPS, core (núcleo)
 arquitetura, 170
 conjunto de instruções, 205, 217–220
MMX (MultiMedia eXtension), 197
modelo de consistência de memória, 410
modelo dos três Cs, 401–403
modo kernel, 389
modos de endereçamento, A-34–36
módulos, A-3
monitores de cristal líquido (LCDs), 14
Moore, máquinas, 404–406, B-55, B-58
 motores de busca, 3
move, instruções, A-53–55
coprocessadores, A-54

detalhes, [A-53–55](#)
ponto flutuante, [A-58–59](#)
move (Move), [121](#)

MS-DOS

- mul.d (FP Multiply Double), [A-59](#)
- mul.s (FP Multiply Single), [A-59](#)
- mult (Multiply), [A-40-41](#)

multicore, [452–455](#)

Multiple Instruction Multiple Data (MIMD), [489](#)
definição, [443](#), [444](#)

Multiple Instruction Single Data (MISD), [443](#)

multiplexadores, [B-7](#)

- controle de seletor, [227](#)
- controles, [405](#)
- definição, [218](#)
- duas entradas, [B-7](#)
- forwarding, valores de controle, [272](#)
- no caminho de dados, [232](#)

multiplicação, [159–164](#), *Ver também* aritmética

- assinada, [163](#)
- hardware, [160–162](#)
- instruções, [164](#), [A-40–41](#)
- mais rápida, [163–164](#)
- multiplicador, [159](#)
- multiplicando, [159](#)
- no MIPS, [164](#)

operандos, 159
ponto flutuante, 180–182, A-59
primeiro algoritmo, 161
produto, 159
rápida, hardware, 164
versão sequencial, 160–162
multiplicação com sinal, 163
multiplicação de ponto flutuante, 180–184
 binária, 184–185
 etapas, 180–184
 ilustração, 183
 instruções, 185
 significandos, 180
multiplicação matricial, 198–201, 484–487
multiplicador, 159
multiplicando, 159
multiprocessadores
 benchmarks, 471–473
 definição, 437
 desempenho, 490
 memória compartilhada, 438, 452–455
 organização, 464
 passagem de mensagens, 464–468
 perspectiva histórica, 561
 software, 437
 UMA, 453

multiprocessadores de memória compartilhada (SMP), [452–455](#)

definição, [438, 452](#)

espaço único de endereços físicos, [452](#)

sincronização, [453](#)

multiprocessadores multicore, [7, 36](#)

definição, [7, 437–438](#)

multithreading

coarse-grained, [450](#)

definição, [442](#)

fine-grained, [450](#)

hardware, [450–452](#)

simultâneo (SMT), [451–452](#)

multithreading coarse-grained, [450](#)

multithreading do hardware, [450–452](#)

coarse-grained, [450](#)

opções, [451](#)

simultâneo, [451–452](#)

multithreading fine-grained, [450](#)

multithreading simultâneo (SMT), [451–452](#)

parallelismo em nível de thread, [452](#)

slots de despacho não usados, [451](#)

suporte, [451](#)

multu (Multiply Unsigned), [A-41](#)

N

NAND, portas, [B-6](#)

NAS (NASA Advanced Supercomputing), [472](#)
Newton, iteração, [192](#)
nivelamento por desgaste, [333](#)
Nonuniform Memory Access (NUMA), [454](#)
Nops, [275](#)
nor (NOR), [55](#)
NOR, operação, [77](#), [A-42](#)
NOR, portas, [B-6](#)
cruzadas, [B-40](#)
latch D implementado com, [B-41](#)
notação científica
definição, [171](#)
para reais, [172](#)
somando números, [177](#)
notação deslocada, [69](#), [174](#)
NOT, operação, [77](#), [A-42](#), [B-4](#)
números
binários, [63](#)
computador *versus* mundo real, [195](#)
com sinal, [63–67](#)
decimais, [63](#)
desnormalizados, [195](#)
hexadecimais, [70–71](#)
sem sinal, [63–67](#)
números binários, [70–71](#)
ASCII *versus*, [93](#)

conversão para números decimais, 66
definição, 63
números com sinal, 63–67
 sinal e magnitude, 64
 tratando como sem sinal, 82
números decimais
 conversão de números binários para, 66
 definição, 63
números desnormalizados, 195
números hexadecimais, 70–71
 conversão de número binário, 70–71
números não sinalizados, 63–67
NVIDIA, arquitetura de GPU, 459–462
NVIDIA GTX, 247, 480–484
NVIDIA Tesla GPU, 480–484

O

opcodes
 definição, 71, 232
 definição de linha de controle, 234
OpenMP (Open MultiProcessing), 456, 473
operações
 atômicas, implementando, 105
 hardware, 54–57
 inteiros do x86, 133, 135–136
 lógicas, 75–77

operações lógicas, [75–77](#)

AND, [76](#), [A-40](#)

ARM, [130](#)

deslocamentos, [75](#)

MIPS, [A-39–43](#)

NOR, [77](#), [A-42](#)

NOT, [77](#), [A-42](#)

OR, [77](#), [A-42](#)

operandos, [57–63](#), *Ver também* instruções

compilando atribuição quando na memória, [59](#)

constante, [62–63](#)

deslocamento, [130](#)

divisão, [165](#)

immediatos de 32 bits, [97–98](#)

instruções aritméticas, [57](#)

memória, [58–59](#)

MIPS, [55](#)

multiplicação, [159](#)

ponto flutuante, [186](#)

somando, [156](#)

operandos constantes, [62–63](#)

em comparações, [81](#)

ocorrência frequente, [62](#)

operandos imediatos de 32 bits, [97–98](#)

ordem principal de coluna, [362](#)

ordem principal de linha, [191](#), [362](#)

ori (Or Immediate), [55](#)
OR, operação, [77](#), [A-42](#), [B-4](#)
or (OR), [55](#)
otimização
 compilador, [123](#)
 manual, [126](#)
otimização de software
 via bloqueio, [362–366](#)
overflow
 definição, [64](#), [173](#)
 detecção, [157](#)
 exceções, [288](#)
 ocorrência, [65](#)
 ponto flutuante, [173](#)
 saturação, [158](#)
 subtração, [156](#)

P

packed, formato de ponto flutuante, [197](#)
pacotes de despacho, [292](#)
páginas, *Ver também* memória virtual
 definição, [375](#)
 localizando, [378–379](#)
 LRU, [380](#)
 modificadas, [382](#)
 número físico, [376](#)

número virtual, 376
offset, 376
posicionando, 378–379
tamanho, 376
páginas modificadas, 382
paralelismo, 9, 36, 290–301
 aritmética de computadores, 195–196
 benefícios de desempenho, 37–38
 despacho múltiplo, 290–296
 GPUs e, 459
 hierarquias de memória e, 407–410
 multicore, 452
 multithreading, 452
 nível de dados, 205, 444
 nível de instrução, 36, 290, 300
 nível de processo, 437
 nível de tarefa, 437
 Redundant Arrays of Inexpensive Disks (RAID), 410
paralelismo de subword, 195–196, 308
 e multiplicação matricial, 198–201
paralelismo em nível de dados, 444
paralelismo em nível de instrução (ILP), *Ver também* paralelismo
 aumentando a exploração, 300
 definição, 36, 291
 multiplicação matricial, 307–309
paralelismo em nível de processo, 437

parallelismo em nível de requisição, 467
parallelismo em nível de tarefa, 437
parâmetros formais, A-12
paravirtualização, 421
paridade
 bits, 369
 código, 368, B-52
PARSEC (Princeton Application Repository for Shared Memory Computers),
 473
passagem de mensagens
 definição, 464
 multiprocessadores, 464–468
PCI-Express (PCIe), 471
penalidade de falha
 caches multinível, reduzindo, 359
 definição, 329
 determinação, 342–343
Pentium, jogada de moralidade do bug, 203–204
perda cold-start, 401
perdas compulsórias, 401
perdas de capacidade, 401
perdas de conflito, 401
perdas por colisão, 402
Personal Mobile Device (PMD)
 definição, 5
petabyte, 4
pilhas

alocando espaço, 90
definição, 85
para argumentos, 122
pop, 85
procedimentos recursivos, A-22–23
push, 85, 87

pipelines

- cinco estágios, 242, 256, 263
- despacho duplo estático, 293
- diagramas de múltiplos ciclos de clock, 260–261
- diagramas de único ciclo de clock, 260–261
- estágio de acesso à memória, 256, 258
- estágio de busca de instrução, 256, 258
- estágio de decodificação de instrução e leitura de banco de registradores, 253, 258
- estágio de execução e cálculo de endereço, 256, 258
- estágio de write-back, 256, 258
- estágios, 242
- gargalos de desempenho, 300
- impacto da instrução de desvio, 277
- latência, 252
- representação gráfica, 246, 260–263
- sequência de instruções, 274

pipelining, 9, 240–252

- analogia da lavanderia, 241
- armadilha, 310–311
- avançado, 300–301

benefícios, 240
exceções, 286–290
falácia, 310–311
fórmula de speed-up, 241
hazards, 244–245
hazards de controle, 248–249
hazards de dados, 245
hazards estruturais, 245, 259
instruções de execução simultânea, 252
melhoria de desempenho, 244
paradoxo, 241
projeto do conjunto de instruções, 244
resumo, 250
tempo de execução, 252
vazão, 252
visão geral, 240–252
pistas de vetor, 448
pixels, 14
ponteiros
 arrays *versus*, 123–127
 frame, 90
 globais, 89
 incrementando, 125
 pilha, 85, 89
ponteiros de frame, 90
ponteiros globais, 89

ponto flutuante, [171–195](#), [197](#)
arquitetura SSE2, [197–198](#)
arredondamento, [192](#)
cálculos intermediários, [192](#)
codificação de instruções, [187](#)
compilando programas, [188–191](#)
conversão binário para decimal, [176](#)
desafios, [204–205](#)
desvio, [185](#)
dígitos de guarda, [192](#)
divisão, [185](#)
forma, [172](#)
formato empacotado, [197](#)
frequência de instrução MIPS, [208](#)
instruções MIPS, [185–187](#)
linguagem assembly, [186](#)
linguagem de máquina, [186](#)
multiplicação e adição reunidas, [194](#)
no x86, [197](#)
operandos, [186](#)
overflow, [173](#)
padrão IEEE 754, [173](#), [174](#)
precisão, [203](#)
procedimento com matrizes bidimensionais, [189–191](#)
registradores, [191](#)
representação, [172–176](#)

sinal e magnitude, 172
subtração, 185
underflow, 173
unidades, 192

pop, 85

portas, B-2, B-5
 AND, B-9
 atrasos, B-37
 NAND, B-6
 NOR, B-6, B-40

potência
 eficiência, 300–301
 natureza crítica, 44
 relativa, 35
 taxa de clock e, 34

precisão dupla, *Ver também* precisão simples
 definição, 173
 representação, 175

precisão simples, *Ver também* precisão dupla
 definição, 173
 representação binária, 175

predição, 10
 desvio dinâmico, 280–283
 esquema de 2 bits, 282
 estado fixo, 280
 loops, 280–283

precisão, 280, 284
prefetching, 421
previsão de desvio
 buffers, 280, 282
 como solução do hazard de controle, 249
 definição, 249
 dinâmica, 249, 280–283
 estática, 293
previsão de desvio dinâmica, 280–283, *Ver também* hazards de controle
 buffer de previsão de desvio, 280
 loops e, 280–283
 previsão de desvio estática, 293
previsão de estado fixo, 280
previsão de falha, 367
previsor de correlação, 283
previsores de desvio
 correlação, 283
 informação, 284
 precisão, 282
 torneio, 284
previsores de desvio de torneio, 284
previsores de hardware dinâmicos, 249
primeira word crítica, 343
primeira word requisitada, 343
procedimentos, 84–92
 aninhados, 87–89

compilando, 85
compilando, mostrando ligação de procedimento aninhado, 88–89
cópia de string, 94–95
etapas de execução, 84
folha, 87
frames, 90
para definir arrays em zero, 124
recursivos, 92, A-20
sort, 117–121
strcpy, 94–95
swap, 116
procedimentos aninhados, 87–89
 compilando procedimento recursivo mostrando, 88–89
procedimentos de folha, *Ver também* procedimentos
 definição, 87
 exemplo, 95
procedimentos recursivos, 92, A-20, *Ver também* procedimentos
 invocação de clone, 87
 pilha, A-22–23
processadores, 215–311
 caminho de dados, 17
 como núcleos, 36
 controle, 17
 crescimento do desempenho, 37
 definição, 14, 17
 despacho duplo, 294–295

despacho múltiplo, 291
despacho múltiplo dinâmico, 291
despacho múltiplo estático, 291, 292–296
especulação, 292–292
execução fora da ordem, 301, 364
superescalar, 296, 451
tecnologias para criação, 20–24
vetor, 444–446
VLIW, 293

processadores de despacho múltiplo estático, 291, 292–296, *Ver também* [despacho múltiplo](#)
com MIPS ISA, 293–295
conjuntos de instruções, 293
hazards de controle, 293–294

processadores de vetor, 444–446, *Ver também* processadores
comparação de código convencional, 445–446
escalares *versus*, 446
extensões de multimídia, 446–448
instruções, 446

processadores dinâmicos de despacho múltiplo, 291, 296–298, *Ver também* [despacho múltiplo](#)
escalonamento de pipeline, 296–298
superescalar, 296

processadores múltiplos, 484–487
produto, 159
programas
iniciando, 107–115

Java, iniciando, [114–115](#)
linguagem assembly, [107](#)
processamento paralelo, [439–443](#)
traduzindo, [107–115](#)
programas de processamento paralelo, [439–443](#)
definição, [438](#)
dificuldade na criação, [439–443](#)
para espaço de endereços compartilhado, [454–456](#)
para passagem de mensagens, [454–456](#)
uso, [490](#)
Programmable Array Logic (PAL), [B-63](#)
Programmable Logic Arrays (PLAs)
definição, [B-9](#)
exemplo, [B-10](#)
ilustração, [B-9](#)
ilustração de pontos de componente, [B-13](#)
implementação de tabela verdade, [B-10](#)
ROMs, [B-12](#)
Programmable Logic Devices (PLDs), [B-63](#)
projeto
caminho de dados, [223](#)
comprometimentos, [141](#)
conjuntos de instruções de pipeline, [244](#)
hierarquia de memória, desafios, [403](#)
lógico, [220–223](#), [B-1–64](#)
unidade de controle principal, [231–234](#)

projeto do genoma humano, 3
propagação
 definição, B-31
 exemplo, B-34
 super, B-32
proteção
 definição, 375
 implementando, 389–391
 VMs para, 371
protocolos de invalidação de escrita, 409, 410
protocolo snooping, 409–410
pseudoinstruções
 definição, 108
 resumo, 109
pseudo MIPS
 conjunto de instruções, 207
 definição, 205
Pthreads (POSIX threads), 473
push
 definição, 85
 usando, 87

Q

quad words, 135
quantidade de deslocamento, 71
Quicksort, 360, 361

quociente, 165

R

Radix sort, 360, 361

RAID, *Ver Redundant Arrays of Inexpensive Disks (RAID)*

raiz quadrada, instruções, A-59

RAM, 7

Ray Casting (RC), 481

Read-Only Memories (ROMs), B-11–12

codificação da função lógica, B-12

PLAs, B-12

programáveis (PROM), B-11

realização de serviço, 366

Receiver Control, registrador, A-30

Receiver Data, registrador, A-30

rede de clusters, 471

redes, 20

crossbar, 469

largura de banda, 469

locais (LANs), 20

multiestágios, 469

remotas (WANs), 20

totalmente conectadas, 469

vantagens, 20

redes crossbar, 469

redes locais (LANs), 20, *Ver também* redes

redes multiestágio, [469](#)
redes totalmente conectadas, [469](#)
redução, [454](#)
referências
 absolutas, [110](#)
 forward, [A-8](#)
 não resolvidas, [A-3](#), [A-A-13](#)
 referências absolutas, [110](#)
 referências de forwarding, [A-8](#)
 referências não resolvidas
 definição, [A-3](#)
 link-editores e, [A-A-13](#)
registrador Cause
 campos, [A-26](#), [A-27](#)
 definição, [286](#)
registrador contador, [A-26](#)
registrador de status
 campos, [A-26](#), [A-27](#)
registradores, [133](#), [133–135](#)
 arquiteturais, [285–290](#)
 base, [59](#)
 Cause, [A-27](#)
 compilando a atribuição C com, [58](#)
 contador, [A-26](#)
 convenções MIPS, [92](#)
 definição, [57](#)

destino, 72, 232
especificação numérica, 224
mapeando, 69
metade dierita, 256
metade esquerda, 256
pipeline, 270, 271, 273
ponto flutuante, 191
primitivos, 57
Receiver Control, A-30
Receiver Data, A-30
renomeando, 295
salvo pelo callee, A-17
salvo pelo caller, A-17
spilling, 61
Status, 286, A-27
tabela de página, 378
tempo de ciclo de clock, 57
temporários, 57, 86
Transmitter Control, A-30–31
Transmitter Data, A-31
variáveis, 57
registradores arquiteturais, 304
registradores de base, 59
registradores de pipeline
antes do forwarding, 271
dependências, 270

seleção de unidade de forwarding, 273

registradores de uso geral, 131

registradores temporários, 57, 86

registrador salvo pelo callee, A-17

registrador salvo pelo caller, A-17

reinício antecipado, 343

replicação, 409

representação em complemento a dois, 65

- atalho de extensão de sinal, 67
- atalho de negação, 66
- regra, 69
- vantagem, 65–66

resto

- definição, 165
- instruções, A-42

restrição de alinhamento, 59–60

retorno de exceção (ERET), 390

ripple carry

- somador, B-23

velocidade de carry lookahead *versus*, B-36

RISCs de desktop e servidor *Ver também* Reduced Instruction Set Computer (RISC), arquiteturas

ROMs programáveis (PROMs), B-11

roofline, modelo, 474–475, 476, 477

- com áreas sobrepostas sombreadas, 479
- com dois kernels, 479
- com tetos, 478, 479

desempenho de pico da memória, [475](#)
desempenho de ponto flutuante de pico, [474](#)
gerações de Opteron, [475](#), [476](#)
ilustração, [475](#)
roofline computacional, [477](#)
rotina de envio de mensagem, [464](#)
rotina receber mensagem, [464](#)
rótulos
 globais, [A-7](#), [A-8](#)
 locais, [A-8](#)
rótulos externos, [A-7](#)
rótulos locais, [A-8](#)

S

saturação, [158](#)
sb (Store Byte), [55](#)
SCALAPAK, [203](#)
sc (Store Conditional), [55](#)
segmentação, [377](#)
segmento de dados, [A-9](#)
segmento de pilha, [A-17](#)
segmento de texto, [A-9](#)
seletores de dados, [218](#)
sem alocação de escrita, [345](#)
semicondutores, [21](#)
serialização da escrita, [408](#)

servidores, OL5, *Ver também* RISCs de desktop e servidor
custo e capacidade, 3
setores, 333
sh (Store Halfword), 55
significandos, 173
adição, 177
multiplicação, 180
silício, 21
como tecnologia básica do hardware, 44
definição, 22
lingote de cristal, 22
wafers, 22
SIMD (Single Instruction Multiple Data), 443–444, 489
arquitetura de vetor, 444–446
no x86, 444
Simple Programmable Logic Devices (SPLDs), B-63
simplicidade, 141
sinais
ativados, 222, B-3
controle, 222, 232–234
desativados, 222, B-3
sinais ativados, 222, B-3
sinais de controle
ALUOp, 232
bits múltiplos, 234
caminhos de dados em pipeline, 263–265

definição, [222](#)
efeito, [234](#)
sinais desativados, [222](#), [B-3](#)
sinal e magnitude, [172](#)
sincronização, [105–107](#), [483](#)
 definição, [453](#)
 lock, [105](#)
 overhead, reduzindo, [37–38](#)
 unlock, [105](#)
sincronização de desbloqueio, [105](#)
sincronização de lock, [105](#)
sincronizadores
 definição, [B-61](#)
 de flip-flop D, [B-61](#)
 falha, [B-62](#)
Single Error Correcting/Double Error Correcting (SEC/DEC), [368–369](#)
Single Instruction Single Data (SISD), [443](#)
sistema síncrono, [B-38](#)
sistemas operacionais
 definição, [10](#)
 encapsulamento, [19](#)
sll (Shift Left Logical), [55](#)
slti (Set Less Than Imm.), [55](#)
sltiu (Set Less Than Imm.Unsigned), [55](#)
slt (Set Less Than), [55](#)
sltu (Set Less Than Unsig.), [55](#)

smartphones, 5

software

camadas, 10-11

como serviço, 5, 467, 489

multiprocessador, 437

paralelo, 438

sistemas, 10

software de sistemas, 10

software paralelo, 438

soldagem, 22

somadores carry save, 164

soma dos produtos, B-8, B-9

Sort, procedimento, 117–121, *Ver também* procedimentos

alocação de registrador, 117

chamada de procedimento, 120

código para o corpo, 117–119

passando parâmetros, 120

preservando registradores, 120

procedimento completo, 120–121

SPEC

benchmark de CPU, 39–40

benchmark de potência, 40–41

SPEC2006, 205

SPECrate, 471–472

SPECratio, 39

spilling de registradores, 61, 85

SPIM, [A-31–34](#)

chamadas do sistema, [A-33–34](#)

introdução, [A-32](#)

ordem de byte, [A-33](#)

recursos, [A-32–33](#)

simulação de máquina virtual, [A-32](#)

suporte a diretivas do montador MIPS, [A-36–38](#)

velocidade, [A-32](#)

versões, [A-32](#)

sra (Shift Right Arith.), [A-42](#)

srl (Shift Right Logical), [55](#)

stack pointers

ajuste, [87](#)

definição, [85](#)

valores, [87](#)

stalls, [246](#)

buffer de escrita, [349](#)

esquema write-back, [349](#)

evitando com reordenação de código, [246](#)

hazards de dados e, [274–277](#)

inserção em pipeline, [276](#)

memória, [350](#)

solução para hazard de controle, [249](#)

uso de load, [278](#)

stalls de pipeline, [246](#)

evitando com reordenação de código, [247](#)

hazards de dados e, 274–277
inserção, 276
solução para hazards de controle, 249
uso de load, 278
stalls de uso de load, 278

Static Random Access Memories (SRAMs), 331, 331, B-47–51
buffers de três estados, B-48
definição, 17, B-47
estrutura básica, B-49
grandes, B-48
início de leitura/escrita, B-48
organização de array, B-51
síncronas (SSRAMs), B-51
tempo de acesso fixo, B-47

store word, 61

Strcpy, procedimento, 94–95, *Ver também* procedimentos
ponteiros, 95
procedimento de folha, 95

Stream, benchmark, 479

Streaming SIMD Extension 2 (SSE2)
arquitetura de ponto flutuante, 197

Streaming SIMD Extensions (SSE) e extensões avançadas de vetor no x86, 197–198

strings
definição, 93
em Java, 95–97
representação, 93

strip mining, [446](#)
sub.d (FP Subtract Double), [A-60](#)
subnormais, [195](#)
sub.s (FP Subtract Single), [A-60](#)
sub (Subtract), [55](#)
subtração, [155–159](#), *Ver também* aritmética
 binária, [155–156](#)
 instruções, [A-43](#)
 número negativo, [156](#)
 overflow, [156](#)
 ponto flutuante, [185](#), [A-60](#)
subu (Subtract Unsigned), [103](#)
supercomputadores
 definição, [4](#)
superescalares
 definição, [296](#)
 escalonamento de pipeline dinâmico, [296](#)
 opções de multithreading, [451](#)
swap, espaço, [380](#)
Swap, procedimento, [116](#), *Ver também* procedimentos
 alocação de registradores, [116](#)
 código do corpo, [117](#)
 completo, [117](#), [120–121](#)
swc1 (Store FP Single), [A-55](#)
sw (Store Word), [55](#)
Synchronous DRAM (SRAM), [331–333](#), [B-48](#), [B-53](#)

Synchronous SRAM (SSRAM), [B-51](#)

T

tabelas de página, [399](#)

atualizando, [378](#)

definição, [378](#)

ilustração, [380](#)

indexando, [378](#)

invertidas, [381](#)

memória principal, [382](#)

níveis, [381–382](#)

registrador, [378](#)

técnicas de redução de armazenamento, [381–382](#)

VMM, [396](#)

tabelas de página invertidas, [381](#)

tabelas de símbolos, [109](#), [A-9](#), [A-10](#)

tabelas verdade, [B-3](#)

definição, [230](#)

exemplo, [B-3](#)

implementação de PLA, [B-10](#)

para bits de controle, [230–231](#)

tablets, [5](#)

tags

definição, [336](#)

localizando bloco, [356](#)

tabelas de página, [380](#)

tamanho, 358

tail call, 92

taxa de acerto, 329

taxa de clock

- definição, 28
- frequência comutada como função da, 35
- potência, 34

taxas de falhas locais, 364

taxas de falta

- cache de dados, 398
- cache repartido, 348
- definição, 329
- globais, 364

Intrinsics FastMATH, processador, 348

locais, 364

melhoria, 342–343

origens de falta, 403

tamanho de bloco *versus*, 343

taxas de perda globais, 364

tebibyte (TiB), 4

tecnologias de memória, 331–335

- memória em disco, 333–335
- memória flash, 333

tecnologia DRAM, 331, 331–333

tecnologia SRAM, 331, 331

telas gráficas

LCD, [14](#)
suporte de hardware do computador, [14](#)
telefones celulares, [5](#)
tempo de acerto
definição, [329](#)
desempenho de cache, [351–352](#)
tempo de execução
como medida de desempenho válida, [43](#)
CPU, [27](#), [28–29](#)
pipelining, [252](#)
tempo de preparação, [B-43](#)
tempo de resposta, [25](#)
tempo de suspensão, [B-43](#)
tempo de transferência, [335](#)
temporização
duas fases, [B-60](#)
entradas assíncronas, [B-61–62](#)
metodologias, [B-58–62](#)
sensível ao nível, [B-60–61](#)
terabyte (TB), [4](#)
definição, [4](#)
thrashing, [396](#)
TLB, falhas, [384](#), *Ver também* Translation-Lookaside Buffer (TLB)
handler, [393](#)
ocorrência, [391](#)
ponto de entrada, [393](#)

problema, 397
tratamento, 391–396
tolerância a falhas, 367
topologias de rede, 468–471
implementando, 469
multiestágios, 471
touchscreen, 15
tradução de endereço
definição, 375
para o ARM cortex-A8, 411
para o Intel Core i7, 411
rápida, 383–384
TLB para, 383–384
transistor de passagem, B-51
transistores, 21
Translation-Lookaside Buffer (TLB), 383–384, *Ver também* TLB, falhas
associatividades, 384
ilustração, 383
integração, 385–386
Intrinsics FastMATH, 385
valores típicos, 384
Transmitter Control, registrador, A-30–31
Transmitter Data, registrador, A-31
tratamento de interrupção, A-25
trilhas, 334
troca de contexto, 391

troca indivisível, 105

tubos de vácuo, 21

U

underflow, 173

Unicode

alfabetos, 95

alfabetos de exemplo, 96

definição, 96

unidade lógica e aritmética (ALU), *Ver também ALU, controle, Ver também unidades de controle*

1 bit, B-20–23

32 bits, B-23–30

antes do encaminhamento, 271

caminho de dados do desvio, 225

entrada imediata com sinal, 273

hardware, 157

operações em formato R, 224

para valores de registrador, 224

uso de instrução de referência à memória, 218

unidades

commit, 296–297, 300

controle, 219–220, 229–231

definição, 192

detecção de hazard, 274, 275–276

para implementação de load/store, 226

ponto flutuante, 192

unidades de commit
buffer, 296–297
definição, 296–297
no controle de atualização, 300

unidades de controle, 219, *Ver também* unidade lógica e aritmética (ALU)
ilustração, 235
principais, projetando, 231–234
saída, 229–231

unidades de detecção de hazard, 274–275
conexões de pipeline, 275
funções, 275

Uniform Memory Access (UMA), 453
multiprocessadores, 454

usado menos recentemente (LRU)
definição, 358
estratégia de substituição de bloco, 400
páginas, 380

V

valores de seletor, B-7

variáveis
classe de armazenamento, 89
estáticas, 89
linguagem C, 89
linguagem de programação, 57
registrador, 57

tipo, 89

variáveis estáticas, 89

vazão

- despacho múltiplo, 299
- pipelining, 252, 299

verificação de redundância cíclica, 371

Verilog

- atribuição bloqueante, B-18
- atribuição não bloqueante, B-18
- definição, B-15
- definição comportamental da ALU do MIPS, B-19
- definição da ALU do MIPS, B-28–30
- especificação comportamental, B-16
- especificação estrutural, B-16
- especificação lógica sequencial, B-45–47
- estrutura do programa, B-17
- fios, B-16–17
- lista de sensitividade, B-18
- lógica combinacional, B-18–20
- módulos, B-18
- operadores, B-17
- reg, B-16–17
- tipos de dados, B-16–17

Very Large-Scale Integrated (VLSI), circuitos, 21

Very Long Instruction Word (VLIW)

- definição, 292–293

- processadores, 293
- VHDL, B-16
- Virtual Machine Monitors (VMMs)
 - atitude *laissez-faire*, 421
 - definição, 371
 - implementando, 420–421
 - na melhoria do desempenho, 374
 - requisitos, 373
 - tabelas de páginas, 396
- W**
 - wafers, 22
 - defeitos, 22
 - dies, 22
 - yield, 22
 - Warehouse Scale Computers (WSCs), 5, 465–467, 489
 - warps, 463
 - while, loops, 80–81
 - Wide Area Networks (WANs), 20, Ver também redes
 - words
 - acessando, 58
 - definição, 57
 - duplas, 133
 - load, 58, 61
 - quad, 135
 - store, 61

words duplas, [133](#)
World Wide Web, [3](#)
write-back, estágio
 instrução load, [258](#)
 instrução store, [258](#)
 linha de controle, [265](#)
write-through, caches, *Ver também* caches
 definição, [344](#), [400](#)
 divergência de tag, [345](#)
 vantagens, [401](#)

X

x86, [130–138](#)
 Advanced Vector Extensions no, [198](#)
 codificação de instruções, [136](#)
 codificação do especificador do primeiro endereço, [138](#)
 conclusão, [137–138](#)
 crescimento do conjunto de instruções, [141](#)
 evolução, [130–133](#)
 formatos de instrução, [137](#)
 instruções/funções típicas, [136](#)
 linha de tempo histórica, [130–133](#)
 modos de endereçamento de dados, [133](#), [133–135](#)
 operações com inteiros, [133–136](#)
 operações típicas, [137](#)
 registradores, [133](#), [133–135](#)

SIMD no, [443–444](#)

Streaming SIMD Extensions no, [197–198](#)

tipos de instruções, [133](#)

XMM, [197](#)

Y

Yahoo! Cloud Serving Benchmark (YCSB), [473](#)

yield, [22](#)

YMM, [198](#)

Z

zettabyte, [4](#)

MIPS – Guia de Referência

MIPS Guia de Referência

CONJUNTO BÁSICO DE INSTRUÇÕES

NOME, MNEMÔNICO	MATO	OPERAÇÃO (em Verilog)	FUNÇÃO (Hexa)	OPCODE/
Add	add	R[Rd] = R[rs] + R[rt]	(1) 0 / 20 _{hex}	
Add Immediate	addi	I R[rt] = R[rs] + SignExtImm (1,2)	8 _{hex}	
Add Imm. Unsigned	addiu	I R[rt] = R[rs] + SignExtImm (2)	9 _{hex}	
Add Unsigned	addu	R R[rd] = R[rs] + R[rt]	0/21 _{hex}	
And	and	R R[rd] = R[rs] & R[rt]	0/24 _{hex}	
And Immediate	andi	I R[rt] = R[rs] & ZeroExtImm (3)	c _{hex}	
Branch On Equal	beq	I PC=PC+4+BranchAddr (4)	4 _{hex}	
Branch On Not Equal	bne	I PC=PC+4+BranchAddr (4)	5 _{hex}	
Jump	j	J PC=JumpAddr (5)	2 _{hex}	
Jump And Link	jal	J R[31]=PC+8;PC=JumpAddr (5)	3 _{hex}	
Jump Register	jr	R PC=R[rs] 0 / 08 _{hex}		
Load Byte Unsigned	lbu	I R[rt]=(2 ⁴ b0,M[R[rs]]+SignExtImm)(7:0) (2)	24 _{hex}	
Load Halfword Unsigned	lhu	I R[rt]=(16'b0,M[R[rs]]+SignExtImm)(15:0) (2)	25 _{hex}	
Load Linked	ll	I R[rt]=M[R[rs]]+SignExtImm (2,7)	30 _{hex}	
Load Upper Imm.	lui	I R[rt]=[imm,16'b0] f _{hex}		
Load Word	lw	I R[rt]=M[R[rs]]+SignExtImm (2)	23 _{hex}	
Nor	nor	R R[rd] = -(R[rs] R[rt]) 0 / 27 _{hex}		
Or	or	R R[rd] = R[rs] R[rt] 0 / 25 _{hex}		
Or Immediate	ori	I R[rt] = R[rs] ZeroExtImm (3)	d _{hex}	
Set Less Than	slt	R R[rd] = R[rs] < R[rt] ? 1 : 0 0 / 2a _{hex}		
Set Less Than Imm.	slti	I R[rt] = R[rs] < SignExtImm (2) a _{hex} ? 1 : 0 (2,6)	b _{hex}	
Set Less Than Imm. Unsigned	sltiu	I R[rt] = R[rs] < SignExtImm ? 1 : 0		
Set Less Than Unsigned	sltu	R R[rd] = (R[rs] < R[rt]) ? 1 : 0 (6) 0 / 2b _{hex}		
Shift Left Logical	sll	R R[rd] = R[rt] << shamt 0/00 _{hex}		
Shift Right Logical	srl	R R[rd] = R[rt] >> shamt 0/02 _{hex}		
Store Byte	sb	I M[R[rs]]+SignExtImm(7:0) = R[rt](7:0) (2)	28 _{hex}	
Store Conditional	sc	I M[R[rs]]+SignExtImm = R[rt]; if(R[rs] < R[rt]) ? 1 : 0 (2,7)	38 _{hex}	
Store Halfword	sh	I M[R[rs]]+SignExtImm(15:0) = R[rt](15:0) (2)	29 _{hex}	
Store Word	sw	I M[R[rs]]+SignExtImm = R[rt] (2)	2b _{hex}	
Subtract	sub	R R[rd] = R[rs] - R[rt] (1)	0/22 _{hex}	
Subtract Unsigned	subu	R R[rd] = R[rs] - R[rt] 0 / 23 _{hex}		
(1) Pode causar exceção de overflow				
(2) SignExtImm = [16 imediato[15]], imediato]				
(3) ZeroExtImm = [16 b'0], imediato]				
(4) BranchAddr = [14 imediato[15]], imediato, 2'b0]				
(5) JumpAddr = [PC+4 31:28], endereço, 2'b0]				
(6) Operандos considerados números sem sinal (versus complemento de 2)				
(7) Par indivisível testar&definir; R[rt] = 1 se par indivisível, 0 se não indivisível				

FORMATOS BÁSICOS DE INSTRUÇÃO

R	opcode	rs	rt	rd	shamt	função	
	31	26 25	21 20	16 15	11 10	6 5	0
I	opcode	rs	rt			imediato	
	31	26 25	21 20	16 15			0
J	opcode				endereço		0
	31	26 25					

© 2014 by Elsevier, Inc. Todos os direitos reservados. De Patterson e Hennessy, Organização e Projeto de Computadores, 5a. ed.



CONJUNTO BÁSICO DE INSTRUÇÕES ARITMÉTICAS

① NOME, MNEMÔNICO	MATO	FOR- OPERAÇÃO (Hexa)	② OPCODE/ FMT/FT/ FUNÇÃO	
Branch On FP True	bclt	FI if(FPcond)PC = PC + 4 + Branch- (4)	11/8/1/-	
Branch On FP False	bclf	FI if(! FPcond)PC = PC + 4 + Bran- (4)	11/8/0/-	
Divide	div	R Lo = R[rs]/R[rt]; Hi = R[rs] % R[rt]	0/-/-/1a	
Divide Unsigned	divu	R Lo = R[rs]/R[rt]; Hi = R[rs] % R[rt]	0/-/-/1b	
FP Add Single	add.s	FR F[fd] = F[fs] + F[ft]	11/10/-/-0	
FP Add Double	add.d	FR [F[fd],F[fd + 1]] = {F[fs],F[fs + 1]} + {F[ft],F[ft + 1]}	11/11/-/-0	
FP Compare Single	c.x.s*	FR FPCond = {F[fs].op F[rt]} ? 1 : 0	11/10/-/-y	
FP Compare Double	c.x.d*	FR FPCond = {F[fs].op F[rt]} ? 1 : 0	11/11/-/-y	
*(x é eq, l t ou l e) (op pe ==, < ou <=) (y é 32, 3c ou 3g)				
FP Divide Single	div.s	FR F[fd] = F[fs] / F[rt]	11/10/-/-3	
FP Divide Double	div.d	FR [F[fd],F[fd + 1]] = {F[fs],F[fs + 1]}/ {F[ft],F[ft + 1]}	11/11/-/-3	
FP Multiply Single	mul.s	FR F[fd] = F[fs] * F[rt]	11/10/-/-2	
FP Multiply Double	mul.d	FR [F[fd],F[fd + 1]] = {F[fs],F[fs + 1]} *	11/11/-/-2	
FP Subtract Single	sub.s	FR F[fd] = F[fs] - F[rt]	11/10/-/-1	
FP Subtract Double	sub.d	FR [F[fd],F[fd + 1]] = {F[fs],F[fs + 1]} - {F[ft],F[ft + 1]}	11/11/-/-1	
Load FP Single	lwcl	I F[rt] = M[R[rs]] + SignExtImm (2)	31/-/-/-/-	
Load FP Double	ldcl	I F[rt] = M[R[rs]] + SignExtImm; (2) F[rt + 1] = M[R[rs]] + SignExt- Imm + 4]	35/-/-/-/-	
Move From Hi	mfhi	R R[rd] = Hi	0/-/-/10	
Move From Lo	mflo	R R[rd] = Lo	0/-/-/12	
Move From Control	mfco	R R[rd] = CR[rs]	10/0/-/-0	
Multiply	mult	R [Hi,Lo] = R[rs] * R[rt]	0/-/-/18	
Multiply Unsigned	multu	R [Hi,Lo] = R[rs] * R[rt]	0/-/-/19	
Shift Right Arith.	sra	R R[rd] = R[rt] > shamt	0/-/-/13	
Store FP Single	swcl	I M[R[rs]] + SignExtImm = F[rt] (2)	39/-/-/-/-	
Store FP Double	sdcl	I M[R[rs]] + SignExtImm = F[rt]; (2) M[R[rs]] + SignExtImm + 4] = F[rt + 1]	3d/-/-/-/-	

FORMATOS DE INSTRUÇÃO DE PONTO FLUTUANTE

FR	opcode	fmt	ft	fs	fd	função
	31	26 25	21 20	16 15	11 10	6 5 0
FI	opcode	fmt	ft		imediato	
	31	26 25	21 20	16 15		0

CONJUNTO DE PSEUDOINSTRUÇÕES

NOME	MNEMÔNICO	OPERAÇÃO
Branch Less Than	bit	if(R[rs] < R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs] > R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs] <= R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs] >= R[rt]) PC = Label
Load Immediate	li	R[rd] = imediato
Move	move	R[rd] = R[rs]

NO ME DE REGISTRADOR, NÚMERO, USO, PRESERVAÇÃO EM UMA CHAMADA

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

MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card
2. Fold bottom side (columns 3 and 4) together

OPCODES, CONVERSÃO DE BASE, SÍMBOLOS ASCII

Op- code	(1) Função MIPS (31:26)	(2) Função MIPS (5:0)	Binário	Deci- mal	Hexa- decimal	Ca- rac- tere ASCII	Deci- mal	Hexa- decimal	Carac- tere ASCII
MIPS									
MIPS									
(1)	sll	add.f	00 0000	0	0	NUL	64	40	@
		sub.f	00 0001	1	1	SOH	65	41	A
j	srl	mul.f	00 0010	2	2	STX	66	42	B
jal	sra	div.f	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt.f	00 0100	4	4	EOT	68	44	D
bne		abs.f	00 0101	5	5	ENQ	69	45	E
blez	sriv	mov.f	00 0110	6	6	ACK	70	46	F
bgtz	srav	neg.f	00 0111	7	7	BEL	71	47	G
addi	j r		00 1000	8	8	BS	72	48	H
addiu	jalr		00 1001	9	9	HT	73	49	I
siti	movz		00 1010	10	a	LF	74	4a	J
sltiu	movn		00 1011	11	b	VT	75	4b	K
andi	syscallround.w.f		00 1100	12	c	FF	76	4c	L
ori	break	trunc.w.f	00 1101	13	d	CR	77	4d	M
xorl		ceil.w.f	00 1110	14	e	SO	78	4e	N
lui	sync	floor.w.f	00 1111	15	f	SI	79	4f	O
	mfhi		01 0000	16	10	DLE	80	50	P
(2)	mthi		01 0001	17	11	DC1	81	51	Q
mflo	movz.f		01 0010	18	12	DC2	82	52	R
mtlo	movn.f		01 0011	19	13	DC3	83	53	S
			01 0100	20	14	DC4	84	54	T
			01 0101	21	15	NAK	85	55	U
			01 0110	22	16	SYN	86	56	V
			01 0111	23	17	ETB	87	57	W
mult			01 1000	24	18	CAN	88	58	X
multu			01 1001	25	19	EM	89	59	Y
div			01 1010	26	la	SUB	90	5a	Z
divu			01 1011	27	lb	ESC	91	5b	[
			01 1100	28	lc	FS	92	5c	\
			01 1101	29	ld	GS	93	5d]
			01 1110	30	le	RS	94	5e	^
			01 1111	31	lf	US	95	5f	-
lb	add	cvt.s.f	10 0000	32	20	Space	96	60	'
lh	addu	cvt.d.f	10 0001	33	21	!	97	61	a
lw	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c
lbu	and	cvt.w.f	10 0100	36	24	\$	100	64	d
lhu	or		10 0101	37	25	%	101	65	e
lwr	xor		10 0110	38	26	&	102	66	f
	nor		10 0111	39	27	*	103	67	g
sb			10 1000	40	28	{	104	68	h
sh			10 1001	41	29)	105	69	i
swl	slt		10 1010	42	2a	*	106	6a	j
sw	sltu		10 1011	43	2b	+	107	6b	k
			10 1100	44	2c	,	108	6c	l
			10 1101	45	2d	-	109	6d	m
swr			10 1110	46	2e	.	110	6e	n
cache			10 1111	47	2f	/	111	6f	o
l1	tge	c.f.f	11 0000	48	30	0	112	70	p
lwcl	tgeu	c.un.f	11 0001	49	31	1	113	71	q
lwc2	tilt	c.eq.f	11 0010	50	32	2	114	72	r
pref	tiltu	c.ueq.f	11 0011	51	33	3	115	73	s
	teq	c.lt.f	11 0100	52	34	4	116	74	t
idc1		c.ulit.f	11 0101	53	35	5	117	75	u
ldc2	tne	c.ole.f	11 0110	54	36	6	118	76	v
		c.ule.f	11 0111	55	37	7	119	77	w
sc		c.sf.f	11 1000	56	38	8	120	78	x
swcl		c.ngle.f	11 1001	57	39	9	121	79	y
swc2		c.seq.f	11 1010	58	3a	:	122	7a	z
		c.ngl.f	11 1011	59	3b	:	123	7b	[
sdc1		c.lt.f	11 1100	60	3c	<	124	7c]
sdc2		c.nge.f	11 1101	61	3d	=	125	7d]
		c.le.f	11 1110	62	3e	>	126	7e	~
		c.ngt.f	11 1111	63	3f	?	127	7f	DEL

(1) $\text{opcode}(31:26) == 0$

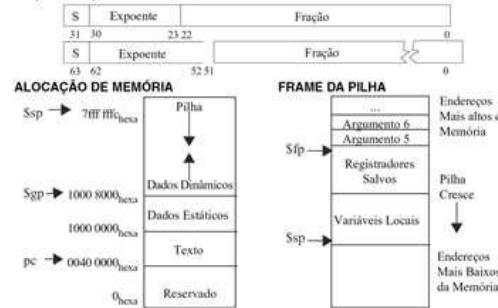
(2) $\text{opcode}(31:26) == 17_{\text{dec}} (11_{\text{hex}})$; $\text{se fmt}(25:21) == 16_{\text{dec}} (10_{\text{hex}})$ $f = s$ (simples);
 $\text{se fmt}(25:21) == 17_{\text{dec}} (11_{\text{hex}})$ $f = d$ (dupla)

PADRÃO DE PONTO FLUTUANTE
IEEE 754

$(-1)^5 \times (1 + \text{Fração}) \times 2^{(\text{Expoente} - \text{Bias})}$

Formatos IEEE de Precisão Simples e Dupla:

Símbolos IEEE 754		
Exponente	Fracção	Objeto
0	0	± 0
0	$\neq 0$	# Desnorm
1 até MAX - 1	qualquer coisa	$\pm \text{NaN}$, Pt, Fl
MAX	0	$\pm \infty$
MAX	$\neq 0$	NaN
S.P. MAX = 255, D.P. MAX = 2047		



ALINHAMENTO DE DADOS

Dupla Word							
Word				Word			
Halfword		Halfword		Halfword		Halfword	
Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0	1	2	3	4	5	6	7

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

REGISTRADORES DE CONTROLE DE EXCEÇÃO: CAUSA E STATUS

B D	Máscara de Interrupção	Código de Exceção
31	15	8 6 2
	Interrupção Pendente	U M

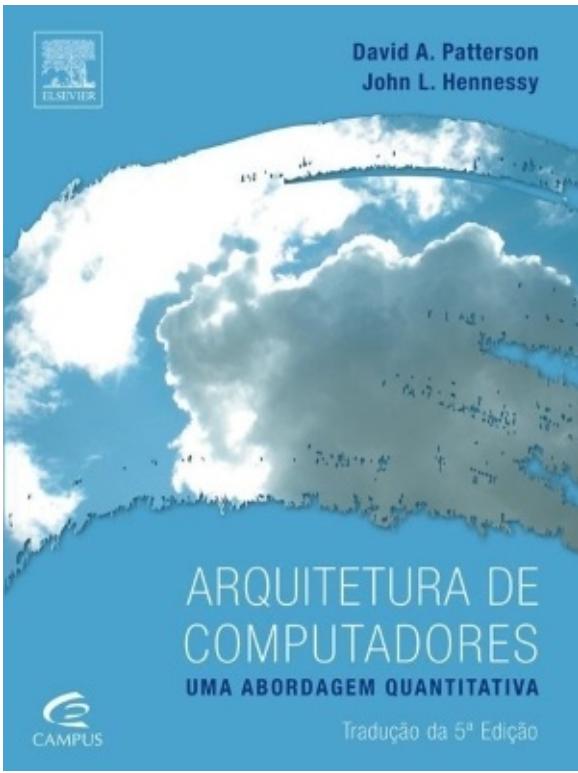
BD = Branch Delay (desvio), UM = User Mode (modo usuário), EL = Exception Level (nível de exceção), IE = Interrupt Enable (interrupção habilitada)

CÓDIGOS DE EXCEÇÃO

Número	Nome	Causa da Exceção	Número	Nome	Causa da Exceção
0	Int	Interrupção (hardware)	9	Bp	Exceção de Ponto de Interrupção
4	AdEL	Exceção de Erro de Endereço (load ou busca de instrução)	10	RI	Exceção de Instrução Revertida
5	AdES	Exceção de Erro de Endereço (store)	11	CpU	Co-processor Não Implementado
6	IBE	Erro de Barramento ou Busca de Instrução	12	Ov	Exceção de Overflow Aritmético
7	DBE	Erro de Barramento no Load ou Store	13	Tr	Trap
8	Sys	Exceção de Syscall	15	FPE	Exceção de Ponto Flutuante

PREFIXOS DE TAMANHO

PREFI- XO	SIM- BOLO	TAMA- NHO	PREFI- XO	SIM- BOLO	TAMA- NHO	PREFI- XO	SIM- BOLO	TAMA- NHO	PRE- FEXO	SIM- BOLO
Kilo-	K	^{2¹⁰}	Kibi-	Ki	^{10¹³}	Peta-	P	^{2³⁰}	Pebi-	Pi
Mega-	M	^{2²⁰}	Mebi-	Mi	^{10¹⁸}	Exa-	E	^{2⁴⁰}	Exbi-	Ei
Giga-	G	^{2³⁰}	Gibi-	Gi	^{10²¹}	Zetta-	Z	^{2⁷⁰}	Zebi-	Zi
Tera-	T	^{2⁴⁰}	Tebi-	Ti	^{10²⁴}	Yotta-	Y	^{2⁸⁰}	Yobi-	Yi



Tradução da 5^a Edição

Arquitetura de Computadores

Hennessy, John L.

9788535264111

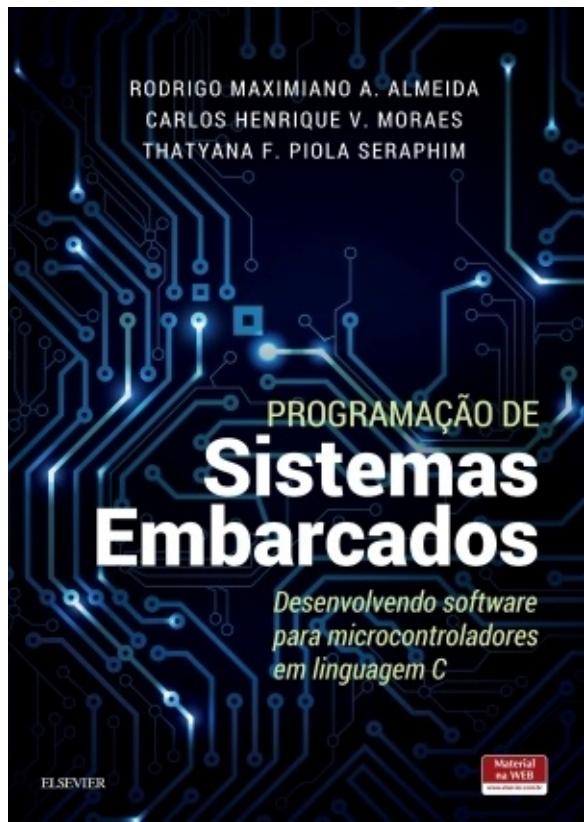
760 páginas

[Compre agora e leia](#)

Atualizado para cobrir a revolução da computação móvel, o livro campeão de vendas sobre arquitetura de computadores, concentra-se nas novas plataformas – dispositivos móveis pessoais e sistemas de computação em escala de depósito – e novas arquiteturas – multicore e GPUs. Esta nova edição abrange essas duas áreas em profundidade, mantendo-se fiel à sua missão original de desmistificar arquitetura de computadores através de um enfoque quantitativo e de uma análise de sistemas reais. Cada capítulo inclui dois exemplos do mundo real, um celular e um datacenter, para ilustrar esta mudança revolucionária e desenvolve temas comuns como potência, desempenho, custo, confiabilidade, proteção, modelos de programação e tendências emergentes. Arquitetura

~~de programação e conceitos emergentes na arquitetura de Computadores – Uma abordagem quantitativa é relevante para estudantes de cursos avançados relacionados às áreas de arquitetura e projeto de computador, bem como profissionais que tenham interesse em atualizar seus conhecimentos.~~

[Compre agora e leia](#)



Programação de Sistemas Embarcados

Almeida, Rodrigo Maximiano Antunes de
9788535285192
488 páginas

[Compre agora e leia](#)

Os sistemas embarcados são dispositivos que podem ser encontrados em qualquer lugar, de aplicações residenciais a controladores de processos críticos como aviação ou equipamentos médicos. Aprender a programar estes dispositivos envolve conhecer o hardware, os periféricos e a interação entre eles. Este livro traz os conhecimentos e ferramentas necessárias para que o leitor possa entender estes dispositivos e desenvolver aplicações com segurança e rapidez.

Existe uma carência de livros na área de sistemas embarcados em português. A maioria incorre em dois problemas: ou é extremamente superficial no tema de programação de embarcados ou é focada em um tipo específico de processador, limitando sua utilidade para um fabricante ou item específico. Com relação ao tempos

Um fascinante e novo campo. Com relações ao tempo, o momento é bastante propício para este tipo de literatura. Com a popularidade da plataforma Arduino várias pessoas têm começado seus estudos nesta área. Outro grande motivador é o advento da "internet das coisas", plataformas embarcadas que possuem acesso a internet, várias empresas multinacionais vem investindo nesta área (Intel, Oracle, Advantech, Dell) e as universidades começam a perceber a escassez de mão de obra no mercado. Oferece conhecimento técnico sobre embarcados e periféricos; Possui um acervo de projetos práticos; Referencial teórico sobre programação; Prepara o leitor para iniciar seus estudos em tópicos mais avançados como sistemas operacionais de tempo real (RTOS), Linux embarcado ou Android; Exercícios

[Compre agora e leia](#)

Andrews

ATLAS CLÍNICO DE DOENÇAS DA PELE

William D. James • Dirk M. Elston • Patrick J. McMahon



ELSEVIER

Andrews atlas clínico de doenças da pele

James, William D.

9788535290295

624 páginas

[Compre agora e leia](#)

Este livro apresenta uma coleção de mais de 3.000 imagens de alta qualidade, incluindo novas doenças e condições raras, além de cabelo, unha e descobertas na membrana mucosa para atender as necessidades dos dermatologistas no momento do diagnóstico. Apresenta ainda um texto introdutório conciso para cada capítulo com uma visão geral e compreensiva para auxiliar no diagnóstico. Veja mais de 3.000 fotografias coloridas de alta qualidade que retratam o aspecto completo de doenças da pele em todos os tipos de pele em adultos, crianças e recém-nascidos; Destaca uma grande variedade de subtipos de condições comuns, tais como lichen planus, granuloma

annulare e psoriase; Novas descobertas de doenças em cabelo, unha e membrana mucosa são apresentados no livro; Inclui representações de importantes condições sistêmicas como sarcoidose, lúpus eritematoso e doenças infecciosas; Apresenta imagens nunca antes publicadas contribuídas por 54 líderes globais em dermatologia; Texto introdutório conciso em cada capítulo fornece aos leitores uma visão geral rápida e compreensiva da doença abordada.

[Compre agora e leia](#)

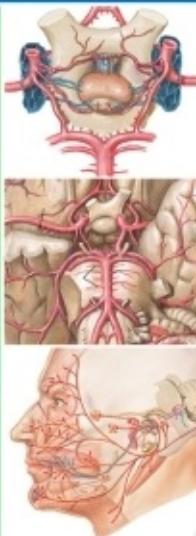


David L. Felten · Anil N. Shetty

Netter
ATLAS
DE NEUROCIÊNCIA



TRADUÇÃO DA 2ª EDIÇÃO



Netter atlas de neurociência

Felten, David L.

9788535246261

464 páginas

[Compre agora e leia](#)

A nova edição do Netter Atlas de Neurociência oferece rica orientação visual, combinada a um texto conciso para ajudar você a dominar os princípios complexos, porém importantes, da neurociência. Uma cobertura de fácil entendimento dividida em três partes — uma visão geral do sistema nervoso, neurociência regional e neurociência sistêmica — permite o estudo das estruturas e sistemas neurais em múltiplos contextos.

No conteúdo, você encontrará:

- Informações atualizadas e novas figuras que refletem os atuais conhecimentos dos componentes neurais e de tecido conjuntivo, regiões e sistemas do cérebro, medula espinal e periferia para garantir que você conheça os avanços mais recentes.
- Novas imagens coloridas em 3D de vias comissurais de associação e de projeção do

cerébro. • Quase 400 ilustrações com a excelência e o estilo Netter que destacam os conceitos-chave da neurociência e as correlações clínicas, proporcionando uma visão geral rápida e fácil de memorizar da anatomia, da função e da relevância clínica. • Imagens de alta qualidade — Imagens de Ressonância Magnética (RM) de alta resolução nos planos coronal e axial (horizontal), além de cortes transversais do tronco encefálico — bem como angiografia e venografia por RM e arteriografia clássica — o que oferece uma melhor perspectiva da complexidade do sistema nervoso. • Anatomia esquemática transversa do tronco encefálico e anatomia cerebral axial e coronal — com RM — para melhor ilustrar a correlação entre neuroanatomia e neurologia. • Uma organização regional do sistema nervoso periférico, da medula espinal, do tronco encefálico, cerebelo e prosencéfalo — e uma organização sistêmica dos sistemas sensitivos, sistemas motores (incluindo o cerebelo e os núcleos da base) e dos sistemas límbicos/hipotalâmicos/autonômicos — que torna as referências mais fáceis e mais eficientes. • Novos quadros de correlação clínica que enfatizam a aplicação clínica das neurociências fundamentais. A compra deste

livro permite acesso gratuito ao site studentconsult.com, um site com ilustrações para download para uso pessoal, links para material de referência adicional e conteúdo original do livro em inglês.

[Compre agora e leia](#)

A maneira inteligente de estudar

Student Consult

Robbins
**PATOLOGIA
BÁSICA**

TRADUÇÃO DA
10^a EDIÇÃO



KUMAR
ABBAS
ASTER

ELSEVIER

Robbins Patologia Básica

Kumar, Vinay

9788535288551

952 páginas

[Compre agora e leia](#)

Parte da confiável família Robbins e Cotran, Robbins Patologia Básica 10^a edição proporciona uma visão geral bem ilustrada, concisa e de leitura agradável dos princípios de patologia humana, ideal para os atarefados estudantes de hoje em dia. Esta edição cuidadosamente atualizada continua a ter forte ênfase na patogênese e nas características clínicas da doença, acrescentando novas ilustrações e diagramas mais esquemáticos para ajudar ainda mais no resumo dos principais processos patológicos e expandir o já impressionante programa de ilustrações.

[Compre agora e leia](#)