

# O Processador

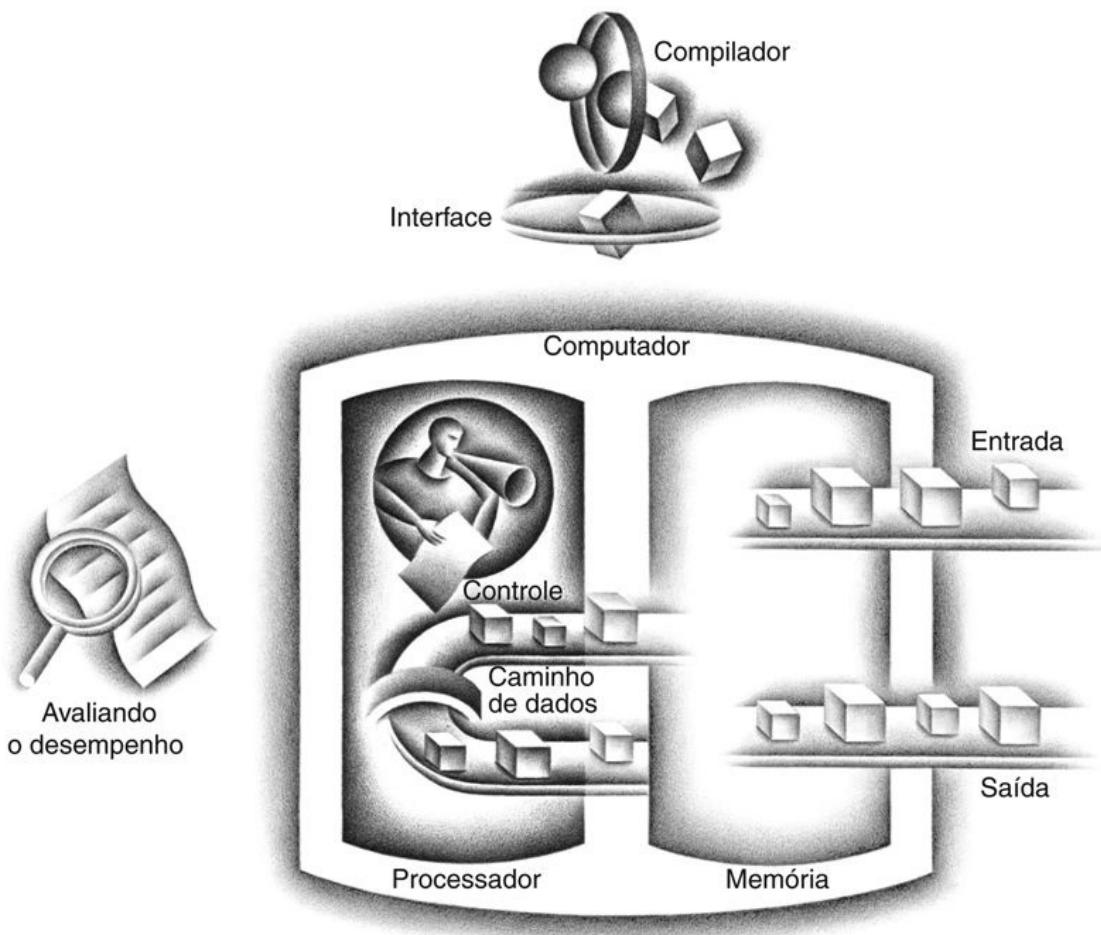
---

*Em um assunto importante, nenhum detalhe é pequeno.*

*Provérbio francês*

- 4.1 Introdução
- 4.2 Convenções lógicas de projeto
- 4.3 Construindo um caminho de dados
- 4.4 Um esquema de implementação simples
- 4.5 Visão geral de pipelining
- 4.6 Caminho de dados e controle usando pipeline
- 4.7 Hazards de dados: forwarding *versus* stalls
- 4.8 Hazards de controle
- 4.9 Exceções
- 4.10 Paralelismo e paralelismo avançado em nível de instrução
- 4.11 Vida real: pipelines do ARM Cortex-A8 e Intel Core i7
- 4.12 Mais rápido: Paralelismo em nível de instrução e multiplicação matricial
- 4.13 Faláncias e armadilhas
- 4.14 Comentários finais
- 4.15 Exercícios

# Os cinco componentes clássicos de um computador

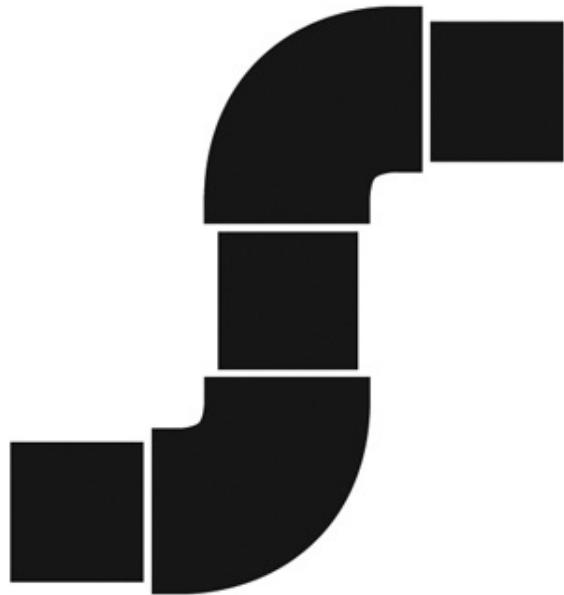


## 4.1. Introdução

O [Capítulo 1](#) explica que o desempenho de um computador é determinado por três fatores principais: contagem de instruções, tempo de ciclo de clock e *ciclos de clock por instrução* (CPI). O [Capítulo 2](#) explica que o compilador e a arquitetura do conjunto de instruções determinam a contagem de instruções necessária para um determinado programa. Entretanto, tanto o tempo de ciclo de

clock quanto o número de ciclos de clock por instrução são determinados pela implementação do processador. Neste capítulo, construímos o caminho de dados e a unidade de controle para duas implementações diferentes do conjunto de instruções MIPS.

Este capítulo contém uma explicação dos princípios e das técnicas usadas na implementação de um processador, começando com uma sinopse altamente abstrata e simplificada nesta seção. Ela é seguida de uma seção que desenvolve um caminho de dados e constrói uma versão simples de um processador, suficiente para implementar conjuntos de instruções como o MIPS. O corpo do capítulo descreve uma implementação MIPS em **pipelining** mais realista, seguida de uma seção que desenvolve conceitos necessários para implementar conjuntos de instruções mais complexos, como o x86.



## PIPELINING

Para o leitor interessado em entender a interpretação de alto nível de instruções e seu impacto sobre o desempenho do programa, esta seção inicial e a [Seção 4.5](#) apresentam os conceitos básicos do pipelining. Tendências recentes são abordadas na [Seção 4.10](#), e a [Seção 4.11](#) descreve as arquiteturas recentes Intel Core i7 e ARM Cortex-A8. A [Seção 4.12](#) mostra como usar o paralelismo

em nível de instrução para mais do que dobrar o desempenho da multiplicação matricial da [Seção 3.8](#). Estas seções oferecem uma base suficiente para entender os conceitos de pipeline em um alto nível.

Para os leitores que desejam um entendimento do processador e seu desempenho com mais profundidade, as [Seções 4.3, 4.4](#) e [4.6](#) serão úteis. Aqueles interessados em aprender como montar um processador também devem ler as [Seções 4.2, 4.7, 4.8](#) e [4.9](#).

## Uma implementação MIPS básica

Analisaremos uma implementação que inclui um subconjunto do conjunto de instruções MIPS básico.

- As instruções de referência à memória *load word* (*lw*) e *store word* (*sw*).
- As instruções lógicas e aritméticas *add*, *sub*, *AND*, *OR* e *slt*.
- As instruções *brench equal* (*beq*) e *jump* (*j*), que acrescentamos depois.

Esse subconjunto não inclui todas as instruções de inteiro (por exemplo, *shift*, *multiply* e *divide* estão ausentes), nem inclui qualquer instrução de ponto flutuante. Entretanto, os princípios básicos usados na criação de um caminho de dados e no projeto do controle são ilustrados. A implementação das outras instruções é semelhante.

Examinando a implementação, teremos a oportunidade de ver como o conjunto de instruções determina muitos aspectos da implementação e como a escolha de várias estratégias de implementação afeta a velocidade de clock e o CPI para o computador. Muitos dos princípios básicos de projeto apresentados no [Capítulo 1](#) podem ser ilustrados, considerando-se a implementação, como o princípio *A simplicidade favorece a regularidade*. Além disso, a maioria dos conceitos usados para implementar o subconjunto MIPS, neste capítulo e no próximo, envolvem as mesmas ideias básicas usadas para construir um amplo espectro de computadores, desde servidores de alto desempenho a microprocessadores de finalidade geral e processadores embutidos.

## Uma sinopse da implementação

No [Capítulo 2](#) vimos instruções MIPS básicas, incluindo as instruções lógicas e aritméticas, as de referência à memória e as de desvio. Muito do que precisa ser feito para implementar essas instruções é igual, independentemente da classe exata da instrução. Para cada instrução, as duas primeiras etapas são idênticas:

1. Enviar o *contador de programa* (PC) à memória que contém o código e

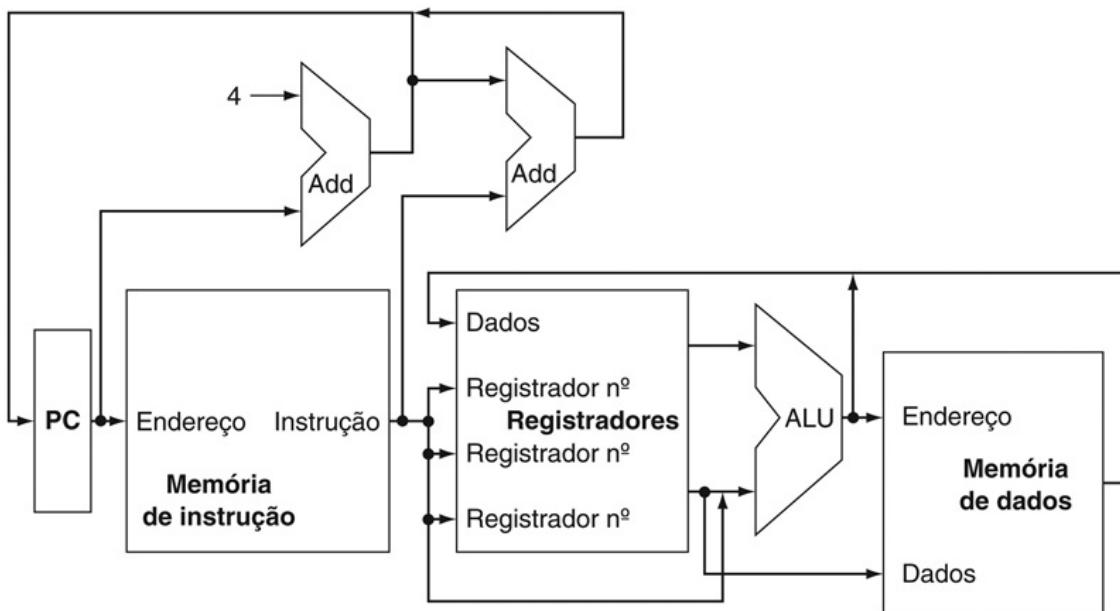
buscar a instrução dessa memória.

2. Ler um ou mais registradores, usando campos da instrução para selecionar os registradores a serem lidos. Para a instrução load word, precisamos ler apenas um registrador, mas a maioria das outras instruções exige a leitura de dois registradores.

Após essas duas etapas, as ações necessárias para completar a instrução dependem da classe da instrução. Felizmente, para cada uma das três classes de instrução (referência à memória, lógica e aritmética, e desvios), as ações são quase as mesmas, seja qual for a instrução exata. A simplicidade e a regularidade do conjunto de instruções simplifica a implementação tornando semelhantes as execuções de muitas das classes de instrução.

Por exemplo, todas as classes de instrução, exceto jump, usam a unidade lógica e aritmética (ALU) após a leitura dos registradores. As instruções de referência à memória usam a ALU para o cálculo de endereço, as instruções lógicas e aritméticas para a execução da operação e desvios para comparação. Após usar a ALU, as ações necessárias para completar várias classes de instrução diferem. Uma instrução de referência à memória precisará acessá-la, a fim de escrever dados para um load ou ler dados para um store. Uma instrução lógica e aritmética precisa escrever os dados da ALU de volta a um registrador. Finalmente, para uma instrução de desvio, podemos ter de mudar o próximo endereço de instrução com base na comparação; caso contrário, o PC deve ser incrementado em 4, a fim de chegar ao endereço da próxima instrução.

A [Figura 4.1](#) mostra a visão em alto nível de uma implementação MIPS, focando as várias unidades funcionais e sua interconexão. Embora essa figura mostre a maioria do fluxo de dados pelo processador, ela omite dois importantes aspectos da execução da instrução.



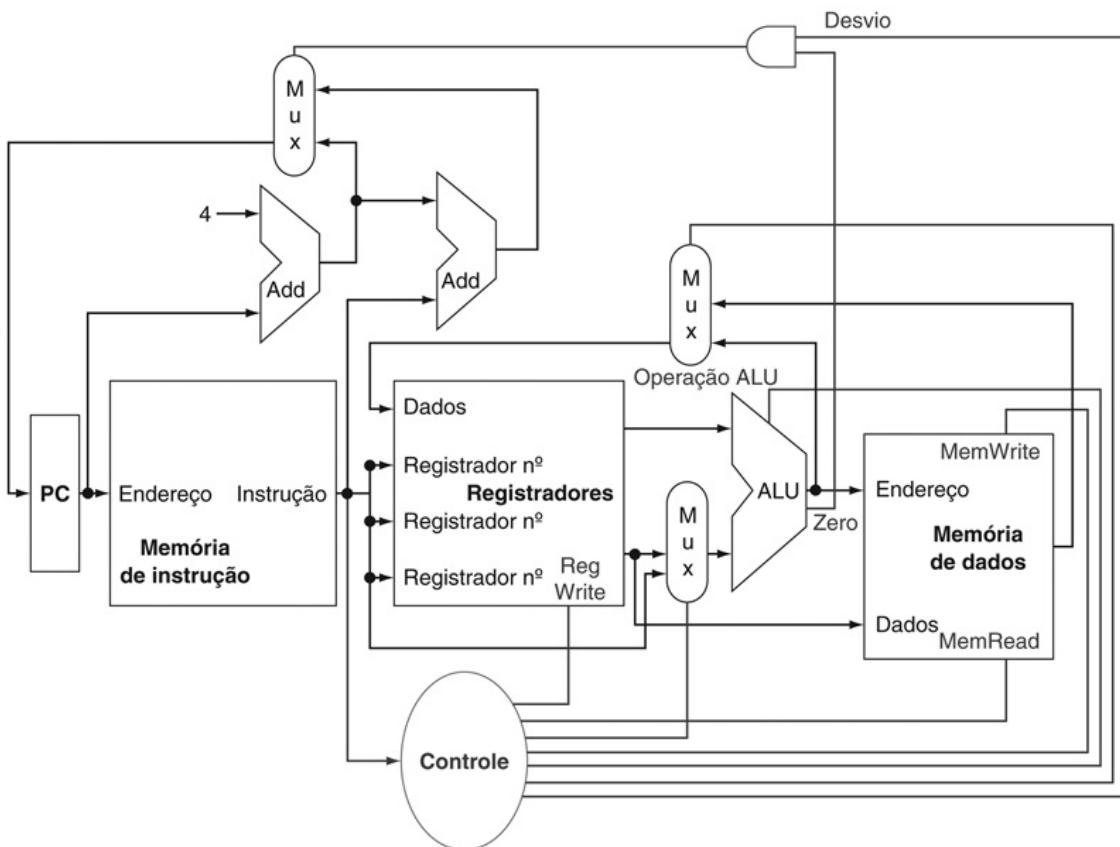
**FIGURA 4.1** Uma visão abstrata da implementação do subconjunto MIPS mostrando as principais unidades funcionais e as principais conexões entre elas.

Todas as instruções começam usando o contador de programa para fornecer o endereço de instrução para a memória de instruções. Depois que a instrução é trazida, os registradores usados como operandos pela instrução são especificados por campos dessa instrução. Uma vez que os operandos tenham sido trazidos, eles podem ser operados de modo a calcular um endereço de memória (para um load ou store), calcular um resultado aritmético (para uma instrução lógica ou aritmética) ou a comparação (para um desvio). Se a instrução for uma instrução lógica ou aritmética, o resultado da ALU precisa ser escrito em um registrador. Se a operação for um load ou store, o resultado da ALU é usado como um endereço com a finalidade de armazenar o valor de um registrador ou ler um valor da memória para um registrador. O resultado da ALU ou memória é escrito de volta no banco de registradores. Os desvios exigem o uso da saída da ALU para determinar o endereço da próxima instrução, que vem da ALU (em que o offset do PC e do desvio são somados) ou de um somador que incrementa o PC atual em 4. As linhas grossas interconectando as unidades funcionais representam barramentos, que consistem em múltiplos sinais. As setas são usadas para guiar o leitor sobre como as informações fluem. Como as linhas de sinal podem se cruzar, mostramos explicitamente quando as linhas que se cruzam estão conectadas pela presença de um ponto no local do cruzamento.

Primeiro, em vários lugares, a [Figura 4.1](#) mostra os dados indo para uma determinada unidade, vindo de duas origens diferentes. Por exemplo, o valor escrito no PC pode vir de dois somadores, os dados escritos no banco de registradores podem vir da ALU ou da memória de dados, e a segunda entrada da ALU pode vir de um registrador ou do campo imediato da instrução. Na prática, essas linhas de dados não podem simplesmente ser interligadas; precisamos adicionar um elemento que escolha dentre as diversas origens e conduza uma dessas origens a seu destino. Essa seleção normalmente é feita com um dispositivo chamado *multiplexador*, embora uma melhor denominação desse dispositivo seria *seletor de dados*. O Apêndice B descreve o multiplexador, que seleciona entre várias entradas com base na configuração de suas linhas de controle. As linhas de controle são definidas principalmente com base na informação tomada da instrução sendo executada.

A segunda omissão na [Figura 4.1](#) é que várias das unidades precisam ser controladas de acordo com o tipo da instrução. Por exemplo, a memória de dados precisa ler em um load e escrever em um store. O banco de registradores precisa ser escrito apenas em uma instrução load ou em uma instrução lógica ou aritmética. E, é claro, a ALU precisa realizar uma de várias operações. (O Apêndice B descreve o projeto detalhado da ALU.) Assim como os multiplexadores, essas operações são direcionadas por linhas de controle que são definidas com base nos vários campos das instruções.

A [Figura 4.2](#) mostra o caminho de dados da [Figura 4.1](#) com os três multiplexadores necessários acrescentados, bem como as linhas de controle para as principais unidades funcionais. Uma *unidade de controle*, que tem a instrução como uma entrada, é usada para determinar como definir as linhas de controle para as unidades funcionais e dois dos multiplexadores. O terceiro multiplexador – que determina se  $PC + 4$  ou o endereço de destino do desvio é escrito no PC – é definido com base na saída zero da ALU, usada para realizar a comparação da instrução **beq**. A regularidade e a simplicidade do conjunto de instruções MIPS significam que um simples processo de decodificação pode ser usado no sentido de determinar como definir as linhas de controle.



**FIGURA 4.2** A implementação básica do subconjunto MIPS incluindo os multiplexadores necessários e as linhas de controle.

O multiplexador superior (“Mux”) controla que valor substitui o PC ( $PC + 4$  ou o endereço de destino do desvio); o multiplexador é controlado pela porta que realiza um AND da saída Zero da ALU com um sinal de controle que indica que a instrução é de desvio. O multiplexador do meio, cuja saída retorna para o banco de registradores, é usado para conduzir a saída da ALU (no caso de uma instrução lógica ou aritmética) ou a saída da memória de dados (no caso de um load) a ser escrita no banco de registradores. Finalmente, o multiplexador da parte inferior é usado de modo a determinar se uma segunda entrada da ALU vem dos registradores (para uma instrução lógica-aritmética OU um desvio) ou do campo offset da instrução (para um load ou store). As linhas de controle acrescentadas são simples e determinam a operação realizada pela ALU, se a memória de dados deve ler ou escrever e se os registradores devem realizar uma operação de escrita. As linhas de controle são mostradas em tons de cinza para que sejam vistas com mais facilidade.

No restante do capítulo, refinamos essa visão para preencher os detalhes, o que exige que acrescentemos mais unidades funcionais, aumentemos o número das conexões entre unidades e, é claro, adicionemos uma unidade de controle, a fim de controlar que ações são realizadas para diferentes classes de instrução. As [Seções 4.3 e 4.4](#) descrevem uma implementação simples que usa um único ciclo de clock longo para cada instrução e segue a forma geral das [Figuras 4.1 e 4.2](#). Nesse primeiro projeto, cada instrução começa a execução em uma transição do clock e completa a execução na próxima transição do clock.

Embora seja mais fácil de entender, esse método não é prático, já que o ciclo de clock precisa ser bastante esticado para acomodar a instrução mais longa. Após projetar o controle desse computador simples, veremos uma implementação em pipeline com todas as suas complexidades, incluindo as exceções.

## Verifique você mesmo

Quantos dos cinco componentes clássicos de um computador — mostrados no início deste capítulo — as Figuras 4.1 e 4.2 contêm?

## 4.2. Convenções lógicas de projeto

Para tratar do projeto de um computador, precisamos decidir como a implementação lógica do computador irá operar e como esse computador está sincronizado. Esta seção examina algumas ideias básicas na lógica digital que usaremos em todo o capítulo. Se você tiver pouco ou nenhum conhecimento em lógica digital, provavelmente será útil ler o Apêndice B antes de continuar.

### elemento combinacional

Um elemento operacional, como uma porta AND ou uma ALU.

Os elementos do caminho de dados na implementação MIPS consistem em dois tipos diferentes de elementos lógicos: aqueles que operam nos valores dos dados e os que contêm estado. Os elementos que operam nos valores dos dados são todos **combinacionais**, significando que suas saídas dependem apenas das entradas atuais. Dada a mesma entrada, um elemento combinacional sempre produz a mesma saída. A ALU mostrada na [Figura 4.1](#) e discutida no Apêndice B é um exemplo de elemento combinacional. Dado um conjunto de entradas, ele

sempre produz a mesma saída porque não possui qualquer armazenamento interno.

Outros elementos no projeto não são combinacionais, mas contêm *estado*. Um elemento contém estado se tiver algum armazenamento interno. Chamamos esses elementos de **elementos de estado**, pois, se desligássemos o computador da tomada, poderíamos reiniciá-lo carregando os elementos de estado com os valores que continham antes de interrompermos a energia. Além disso, se salvássemos e armazenássemos novamente os elementos de estado, seria como se o computador nunca tivesse sido desligado. Na [Figura 4.1](#), as memórias de instruções e de dados, bem como os registradores, são exemplos de elementos de estado.

## elemento de estado

Um elemento da memória, como um registrador ou uma memória.

Um elemento de estado possui pelo menos duas entradas e uma saída. As entradas necessárias são os valores dos dados a serem escritos no elemento e o *clock*, que determina quando o valor dos dados deve ser escrito. A saída de um elemento de estado fornece o valor escrito em um ciclo de *clock* anterior. Por exemplo, um dos elementos de estado mais simples logicamente é um flip-flop tipo D (Apêndice B), que possui exatamente essas duas entradas (um valor e um *clock*) e uma saída. Além dos flip-flops, nossa implementação MIPS também usa dois outros tipos de elementos de estado: memórias e registradores, ambos aparecendo na [Figura 4.1](#). O *clock* é usado para determinar quando se deve escrever no elemento de estado; um elemento de estado pode ser lido a qualquer momento.

Os componentes lógicos que contêm estado também são chamados de *sequenciais* porque suas saídas dependem de suas entradas e do conteúdo do estado interno. Por exemplo, a saída da unidade funcional representando os registradores depende dos números de registrador fornecidos e do que foi escrito nos registradores anteriormente. Tanto a operação dos elementos combinacionais e sequenciais, quanto sua construção são discutidas em mais detalhes no Apêndice B.

## Metodologia de clocking

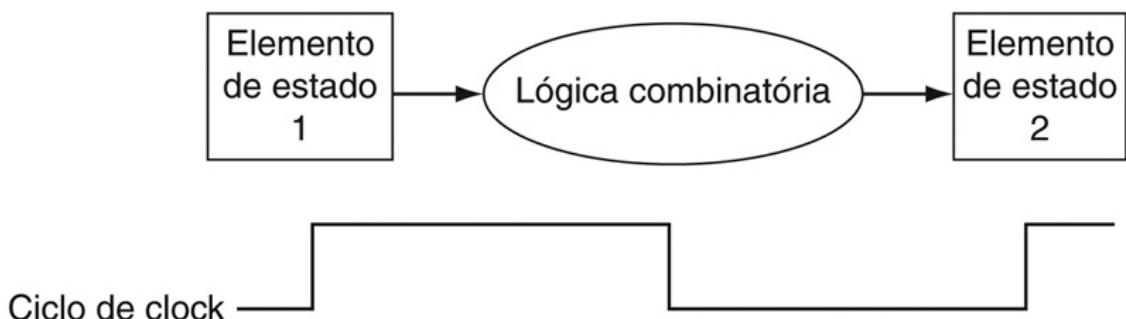
Uma **metodologia de clocking** define quando os sinais podem ser lidos e

quando podem ser escritos. Ela é importante para especificar a sincronização das leituras e escritas porque, se um sinal fosse escrito ao mesmo tempo em que fosse lido, o valor da leitura poderia corresponder ao valor antigo, ao valor recém-escrito ou mesmo alguma combinação dos dois! Obviamente, os projetos de computadores não podem tolerar essa imprevisibilidade. Uma metodologia de clocking tem o objetivo de garantir a previsibilidade.

## metodologia de clocking

O método usado para determinar quando os dados são válidos e estáveis em relação ao clock.

Para simplificar, consideraremos uma metodologia de **sincronização acionada por transição**. Uma metodologia de sincronização acionada por transição significa que quaisquer valores armazenados em um elemento lógico sequencial são atualizados apenas em uma transição do clock, que é uma transição rápida de baixo para alto ou vice-versa ([Figura 4.3](#)). Como apenas os elementos de estado podem armazenar valores de dados, qualquer coleção de lógica combinatória precisa ter suas entradas vindo de um conjunto de elementos de estado e suas saídas escritas em um conjunto de elementos de estado. As entradas são valores escritos em um ciclo de clock anterior, enquanto as saídas são valores que podem ser usados em um ciclo de clock seguinte.



**FIGURA 4.3** A lógica combinacional, os elementos de estado e o clock estão intimamente relacionados.

Em um sistema digital síncrono, o clock determina quando os elementos com estado escreverão valores no armazenamento interno. Quaisquer entradas em um elemento de estado precisam atingir um valor estável (ou seja, ter alcançado um valor do qual não mudarão até após a transição do clock) antes

que a transição ativa do clock faça com que o estado seja atualizado. Todos os elementos de estado neste capítulo, incluindo a memória, são considerados acionados por transição positiva; ou seja, eles mudam na transição de subida do clock.

## sincronização acionada por transição

Um esquema de clocking em que todas as mudanças de estado ocorrem em uma transição do clock.

A [Figura 4.3](#) mostra os dois elementos de estado em volta de um bloco de lógica combinacional, que opera em um único ciclo de clock: todos os sinais precisam se propagar desde o elemento de estado 1, passando pela lógica combinacional e indo até o elemento 2 no tempo de um ciclo de clock. O tempo necessário para os sinais alcançarem o elemento 2 define a duração do ciclo de clock.

Para simplificar, não mostraremos um **sinal de controle** de escrita quando um elemento de estado é escrito em cada transição ativa de clock. Por outro lado, se um elemento de estado não for atualizado em cada clock, um sinal de controle de escrita explícito é necessário. Tanto o sinal de clock quanto o sinal de controle de escrita são entradas, e o elemento de estado só é alterado quando o sinal de controle de escrita está ativo e ocorre uma transição do clock.

## sinal de controle

Um sinal usado para seleção de multiplexador ou para direcionar a operação de uma unidade funcional; contrasta com um *sinal de dados*, que contém informações operadas por uma unidade funcional.

Usaremos o termo **ativo** para indicar um sinal que está logicamente alto, o termo *ativar* para especificar que um sinal deve ser conduzido a logicamente alto, e *desativar* ou **inativo** para representar o que é logicamente baixo. Usamos os termos ativar e desativar porque, ao implementarmos o hardware, às vezes 1 representa um sinal lógico alto, mas também pode representar um sinal lógico baixo.

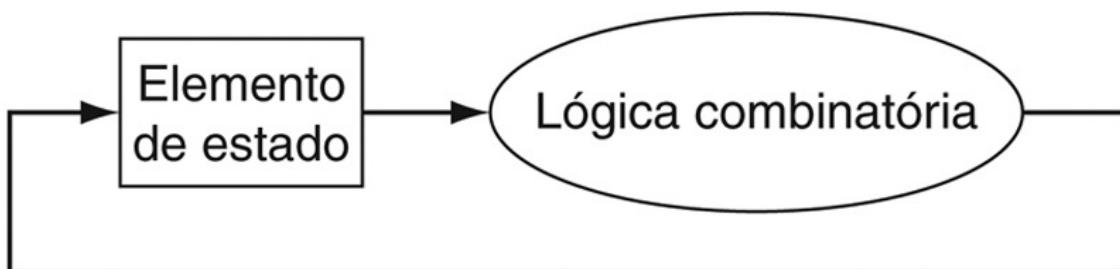
## ativo

O sinal está logicamente alto ou verdadeiro.

## inativo

O sinal está logicamente baixo ou falso.

Uma metodologia acionada por transição permite ler o conteúdo de um registrador, enviar o valor por meio de alguma lógica combinatória e escrever nesse registrador no mesmo ciclo de clock. A [Figura 4.4](#) mostra um exemplo genérico. Não importa se consideramos que todas as escritas ocorrem na transição de subida do clock ou na transição de descida, já que as entradas no bloco de lógica combinatória não podem mudar exceto na transição de clock escolhida. Com uma metodologia de sincronização acionada por transição, não há qualquer feedback dentro de um único ciclo de clock, e a lógica na [Figura 4.4](#) funciona corretamente. No Apêndice B, discutimos brevemente as outras limitações (como os tempos de setup e hold), bem como outras metodologias de sincronização.



**FIGURA 4.4** Uma metodologia acionada por transição permite que um elemento de estado seja lido e escrito no mesmo ciclo de clock sem criar uma disputa que poderia levar a valores de dados indeterminados.

É claro que o ciclo de clock ainda precisa ser longo o suficiente para que os valores de entrada sejam estáveis quando houver transição ativa do clock. O feedback não pode ocorrer dentro de um ciclo de clock devido à atualização acionada por transição do elemento de estado. Se o feedback fosse possível, esse projeto não poderia funcionar corretamente. Nossos projetos neste capítulo e no próximo se baseiam na metodologia de sincronização acionada por transição e em estruturas como a mostrada nesta figura.

Para a arquitetura MIPS de 32 bits, quase todos esses elementos de estado e lógicos terão entradas e saídas contendo 32 bits de extensão, já que essa é a extensão da maioria dos dados manipulados pelo processador. Sempre que uma unidade tiver uma entrada ou saída diferente de 32 bits de extensão, deixaremos isso claro. As figuras indicarão *barramentos* (que são sinais mais largos do que 1 bit), com linhas mais grossas. Algumas vezes, desejaremos combinar vários barramentos para formar um barramento mais largo; por exemplo, podemos querer obter um barramento de 32 bits combinando dois de 16 bits. Nesses casos, rótulos nas linhas de barramento indicarão que estamos concatenando barramentos para formar um mais largo. Setas também são incluídas para ajudar a esclarecer a direção do fluxo dos dados entre elementos. Finalmente, o **realce** indica um sinal de controle em oposição a um sinal que conduz dados; essa distinção se tornará mais clara enquanto avançarmos neste capítulo.

### Verifique você mesmo

Verdadeiro ou falso: como o banco de registradores é lido e escrito no mesmo ciclo de clock, qualquer caminho de dados MIPS usando escritas acionadas por transição precisa ter mais de uma cópia do banco de registradores.

### Detalhamento

Há também uma versão de 64 bits da arquitetura MIPS e, naturalmente, a maioria dos caminhos em sua implementação teria 64 bits de largura.

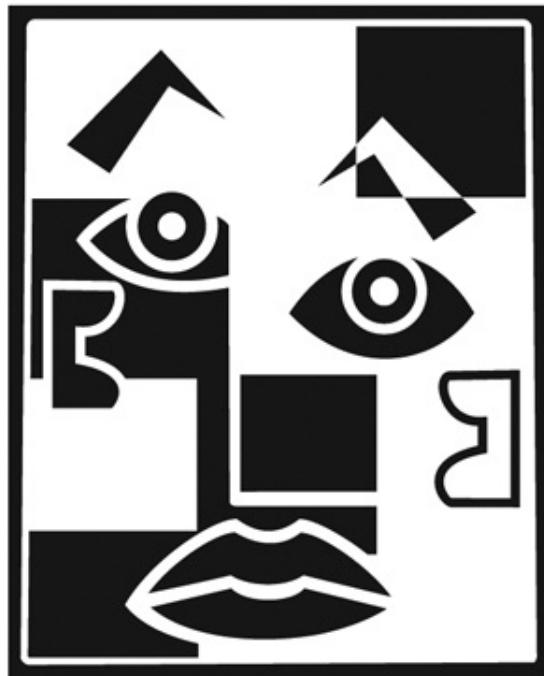
## 4.3. Construindo um caminho de dados

Uma maneira razoável de iniciar um projeto de caminho de dados é examinar os principais componentes necessários para executar cada classe de instrução MIPS. Vamos começar olhando quais **elementos do caminho de dados** cada instrução precisa, e depois desceremos por todos os níveis de **abstração**. Quando mostramos os elementos do caminho de dados, também mostramos seus sinais de controle. Usamos a abstração nesta explicação, começando de baixo para cima.

### elemento do caminho de dados

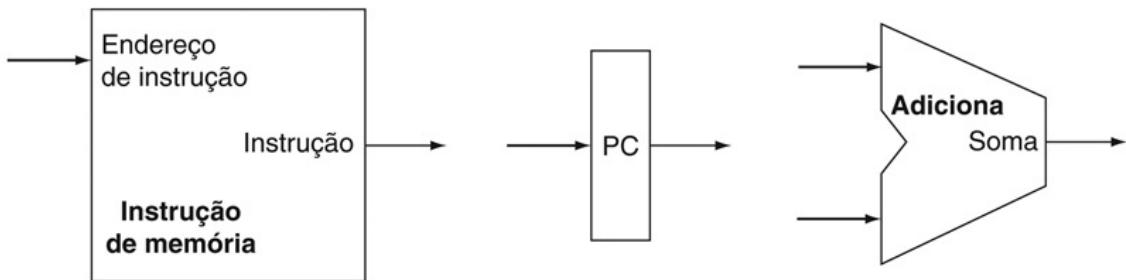
Uma unidade funcional usada para operar sobre os dados ou conter esses

dados dentro de um processador. Na implementação MIPS, os elementos do caminho de dados incluem as memórias de instruções e de dados, o banco de registradores, a unidade lógica e aritmética (ALU) e os somadores.



## A B S T R A Ç Ã O

A [Figura 4.5a](#) mostra o primeiro elemento de que precisamos: uma unidade de memória para armazenar as instruções de um programa e fornecer instruções dado um endereço. A [Figura 4.5b](#) mostra um registrador, que podemos chamar de **contador de programa (PC)**, que, como vimos no [Capítulo 2](#), é um registrador que contém o endereço da instrução atual. Finalmente, precisaremos de um somador a fim de incrementar o PC para o endereço da próxima instrução. Esse somador, que é combinacional, pode ser construído a partir da ALU que descrevemos em detalhes no Apêndice B, simplesmente interligando as linhas de controle de modo que o controle sempre especifique uma operação de adição. Representaremos uma ALU desse tipo com o rótulo *Soma*, como na [Figura 4.5](#), para indicar que ela se tornou permanentemente um somador e não pode realizar as outras funções da ALU.



a. Instrução de memória

b. Contador de programa

c. Adicionador

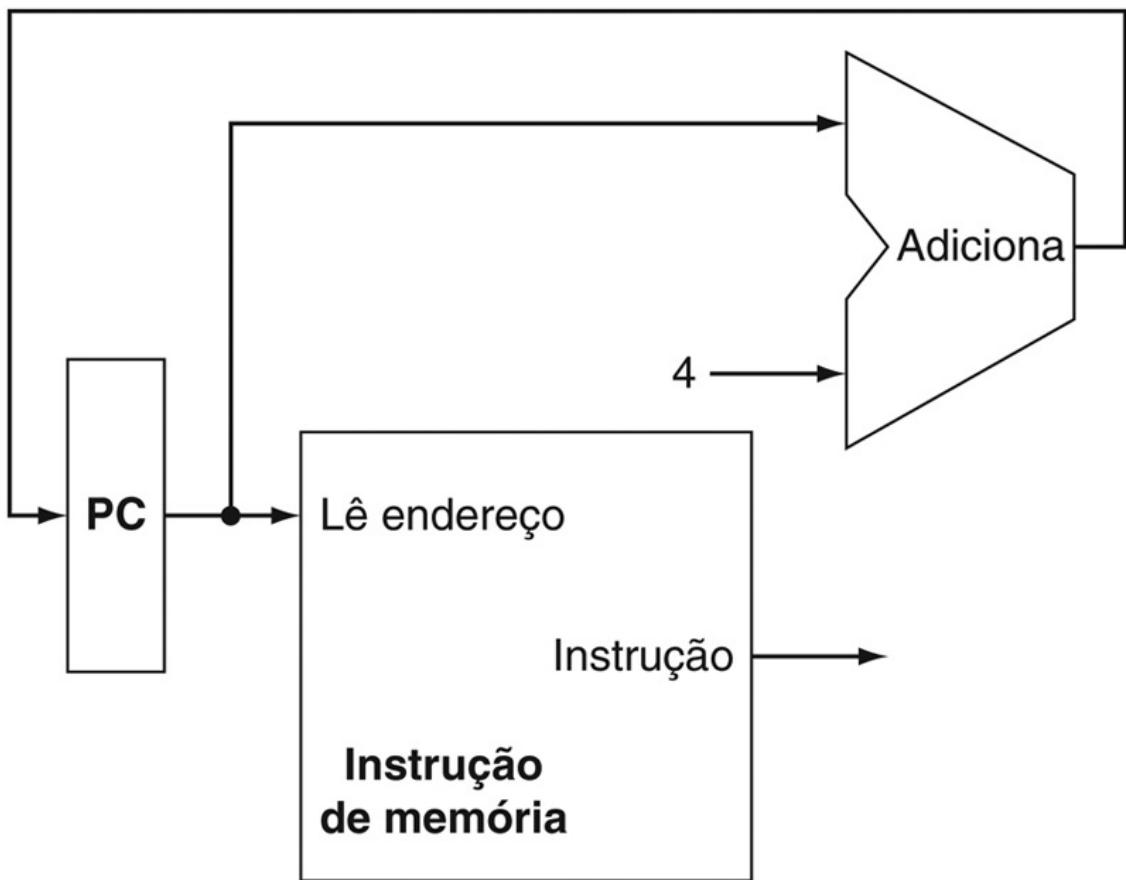
**FIGURA 4.5** **Dois elementos de estado são necessários para armazenar e acessar instruções, e um somador é necessário para calcular o endereço da próxima instrução.**

Os elementos de estado são a memória de instruções e o contador de programa. A memória de instruções só precisa fornecer acesso de leitura porque o caminho de dados não escreve instruções. Como a memória de instruções apenas é lida, nós a tratamos como lógica combinatória: a saída em qualquer momento reflete o conteúdo do local especificado pela entrada de endereço, e nenhum sinal de controle de leitura é necessário. (Precisaremos escrever na memória de instruções quando carregarmos o programa; isso não é difícil de incluir e o ignoramos em favor da simplicidade.) O contador de programa é um registrador de 32 bits que é escrito no final de cada ciclo de clock e, portanto, não precisa de um sinal de controle de escrita. O somador é uma ALU configurada para sempre realizar a adição das suas duas entradas de 32 bits e colocar o resultado em sua saída.

## contador de programa (PC)

O registrador que contém o endereço da instrução do programa sendo executado.

Para executar qualquer instrução, precisamos começar buscando a instrução na memória. A fim de preparar para executar a próxima instrução, também temos de incrementar o contador de programa de modo que aponte para a próxima instrução, 4 bytes depois. A [Figura 4.6](#) mostra como combinar os três elementos da [Figura 4.5](#) para formar um caminho de dados que busca instruções e incrementa o PC de modo a obter o endereço da próxima instrução sequencial.



**FIGURA 4.6** Uma parte do caminho de dados usada para buscar instruções e incrementar o contador do programa. A instrução buscada é usada por outras partes do caminho de dados.

Agora, vamos considerar as instruções de formato R ([Figura 2.20](#)). Todas elas leem dois registradores, realizam uma operação na ALU com o conteúdo dos registradores e escrevem o resultado em um registrador. Chamamos essas instruções de *instruções tipo R* ou *instruções lógicas ou aritméticas* (já que elas realizam operações lógicas ou aritméticas). Essa classe de instrução inclui add, sub, AND, OR e slt, que foram apresentadas no [Capítulo 2](#). Lembre-se de que um caso típico desse tipo de instrução é add \$t1, \$t2, \$t3, que lê \$t2 e \$t3 e escreve em \$t1.

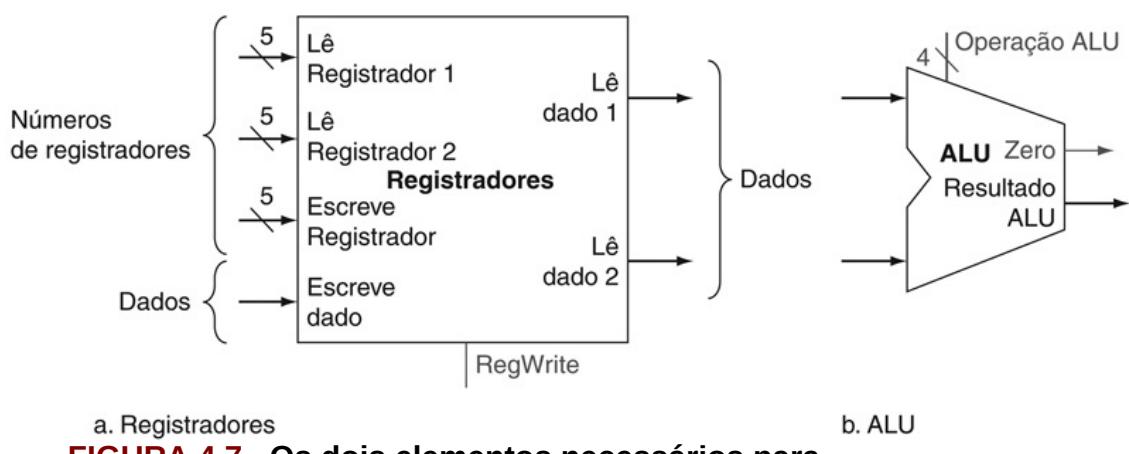
Os registradores de uso geral de 32 bits do processador são armazenados em uma estrutura chamada **banco de registradores**. Um banco de registradores é uma coleção de registradores em que qualquer registrador pode ser lido ou escrito especificando o número do registrador no banco. O banco de registradores contém o estado dos registradores do computador. Além disso,

precisaremos que uma ALU opere nos valores lidos dos registradores.

## banco de registradores

Um elemento de estado que consiste em um grupo de registradores que podem ser lidos e escritos fornecendo um número de registrador a ser acessado.

Devido às instruções de formato R terem três operandos de registrador, precisaremos ler duas palavras de dados do banco de registradores e escrever uma palavra de dados no banco de registradores para cada instrução. A fim de que cada palavra de dados seja lida dos registradores, precisamos de uma entrada no banco de registradores que especifique o número do registrador a ser lido e uma saída do banco de registradores que conduzirá o valor lido dos registradores. Para escrever uma palavra de dados, precisaremos de duas entradas: uma para especificar o *número do registrador* a ser escrito e uma para fornecer os *dados* a serem escritos no registrador. O banco de registradores sempre gera como saída o conteúdo de quaisquer números de registrador que estejam nas entradas Registrador de leitura. As escritas, entretanto, são controladas pelo sinal de controle de escrita, que precisa estar ativo para que uma escrita ocorra na transição do clock. A Figura 4.7a mostra o resultado; precisamos de um total de quatro entradas (três para números de registrador e uma para dados) e duas saídas (ambas para dados). As entradas de número de registrador possuem 5 bits de largura para especificar um dos 32 registradores ( $32 = 2^5$ ), enquanto a entrada de dados e os dois barramentos de saída de dados possuem 32 bits de largura cada um.



**FIGURA 4.7** Os dois elementos necessários para

### **implementar operações para a ALU no formato R são o banco de registradores e a ALU.**

O banco de registradores contém todos os registradores e possui duas portas para leitura e uma porta para escrita. O projeto dos bancos de registradores de várias portas é discutido na Seção B.8 do **Apêndice B**. O banco de registradores sempre gera como saídas os conteúdos dos registradores correspondentes às entradas Registrador de leitura nas saídas; nenhuma outra entrada de controle é necessária. Ao contrário, uma escrita em um registrador precisa ser explicitamente indicada ativando o sinal de controle de escrita. Lembre-se de que as escritas são acionadas por transição, de modo que todas as entradas de escrita (por exemplo, o valor a ser escrito, o número do registrador e o sinal de controle de escrita) precisam ser válidas na transição do clock. Como as escritas no banco de registradores são acionadas por transição, nosso projeto pode ler e escrever sem problemas no mesmo registrador dentro de um ciclo de clock: a leitura obterá o valor escrito em um ciclo de clock anterior, enquanto o valor escrito estará disponível para uma leitura em um ciclo de clock subsequente. Todas as entradas com o número do registrador para o banco de registradores possuem 5 bits de largura, enquanto as linhas com os valores de dados possuem 32 bits de largura. A operação a ser realizada pela ALU é controlada com o sinal de operação da ALU, que terá largura de 4 bits, usando a ALU projetada no **Apêndice B**. Em breve, usaremos a saída de detecção Zero da ALU para implementar desvios. A saída de overflow não será necessária até a [Seção 4.9](#), quando discutiremos as exceções; até lá, elas serão omitidas.

A [Figura 4.7b](#) mostra a ALU, que usa duas entradas de 32 bits e produz um resultado de 32 bits, bem como um sinal de 1 bit se o resultado for 0. O sinal de controle de quatro bits da ALU é descrito em detalhes no Apêndice B; examinaremos o controle da ALU brevemente quando precisarmos saber como defini-lo.

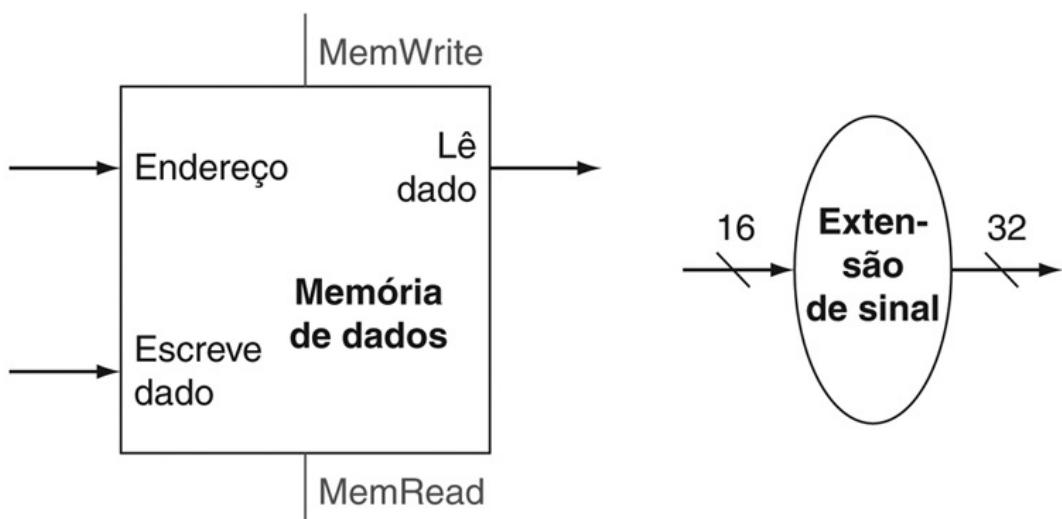
A seguir, considere as instruções MIPS load word e store word, que possuem o formato `lw $t1, offset_value($t2)` ou `sw $t1, offset_value($t2)`. Essas instruções calculam um endereço de memória somando o registrador de base, que é `$t2`, com o campo offset de 16 bits com sinal contido na instrução. Se a instrução for um store, o valor a ser armazenado também precisará ser lido do banco de registradores em que reside, em `$t1`. Se a instrução for um load, o valor lido da memória precisará ser escrito no banco de registradores no

registrador especificado, que é  $\$t1$ . Consequentemente, precisaremos do banco de registradores e da ALU da Figura 4.7.

## banco de registradores

Um elemento de estado que consiste em um grupo de registradores que podem ser lidos e escritos fornecendo um número de registrador a ser acessado.

Além disso, precisaremos de uma unidade a fim de **estender o sinal** do campo offset de 16 bits da instrução para um valor com sinal de 32 bits, e de uma unidade de memória da para ler ou escrever. A memória de dados precisa ser escrita com instruções store; portanto, ela tem sinais de controle de leitura e escrita, uma entrada de endereço e uma entrada para os dados serem escritos na memória. A Figura 4.8 mostra esses dois elementos.



a. Unidade de memória de dados      b. Unidade de extensão de sinal

**FIGURA 4.8** As duas unidades necessárias para implementar loads e stores, além do banco de registradores e da ALU da Figura 4.7, são a unidade de memória de dados e a unidade de extensão de sinal.

A unidade de memória é um elemento de estado com entradas para os endereços e os dados de escrita, e uma única saída para o resultado da leitura. Existem controles de leitura e escrita separados, embora apenas um deles possa estar ativado em qualquer clock específico. A unidade de memória precisa de um

sinal de leitura, já que, diferente do banco de registradores, ler o valor de um endereço inválido pode causar problemas, como veremos no [Capítulo 5](#). A unidade de extensão de sinal possui uma entrada de 16 bits que tem o seu sinal estendido para que um resultado de 32 bits apareça na saída ([Capítulo 2](#)).

Consideramos que a memória de dados é acionada por transição para as escritas. Na verdade, os chips de memória padrão possuem um sinal “write enable” que é usado para escritas. Embora o write enable não seja acionado por transição, nosso projeto acionado por transição poderia facilmente ser adaptado para funcionar com chips de memória reais. Consulte a Seção B.8 do [Apêndice B](#) para ver uma discussão mais detalhada de como funcionam os chips de memória reais.

A instrução `beq` possui três operandos, dois registradores comparados para igualdade e um offset de 16 bits para calcular o **endereço de destino do desvio** relativo ao endereço da instrução desvio. Sua forma é `beq $t1,$t2,offset`. Para implementar essa instrução, precisamos calcular o endereço de destino somando o campo offset estendido com sinal da instrução com o PC. Há dois detalhes na definição de instruções de desvio ([Capítulo 2](#)) para os quais precisamos prestar atenção:

- O conjunto de instruções especifica que a base para o cálculo do endereço de desvio é o endereço da instrução seguinte ao desvio. Como calculamos  $PC + 4$  (o endereço da próxima instrução) no caminho de dados para busca de instruções, é fácil usar esse valor como a base para calcular o endereço de destino do desvio.
- A arquitetura também diz que o campo offset é deslocado 2 bits para a esquerda, de modo que é um offset de uma palavra; esse deslocamento aumenta a faixa efetiva do campo offset por um fator de quatro vezes.

## endereço de destino do desvio

O endereço especificado em um desvio, que se torna o novo contador do programa (PC) se o desvio for tomado. Na arquitetura MIPS, o destino do desvio é dado pela soma do campo offset da instrução e o endereço da instrução seguinte ao desvio.

Para lidar com a última complicação, precisaremos deslocar o campo offset de dois bits.

Além de calcular o endereço de destino do desvio, também precisamos determinar se a próxima instrução é a instrução que acompanha sequencialmente ou a instrução no endereço de destino do desvio. Quando a condição é verdadeira (isto é, os operandos são iguais), o endereço de destino do desvio se torna o novo PC e dizemos que o **desvio é tomado**. Se os operandos não forem iguais, o PC incrementado deve substituir o PC atual (exatamente como para qualquer outra instrução normal); nesse caso, dizemos que o **desvio é não tomado**.

## desvio tomado

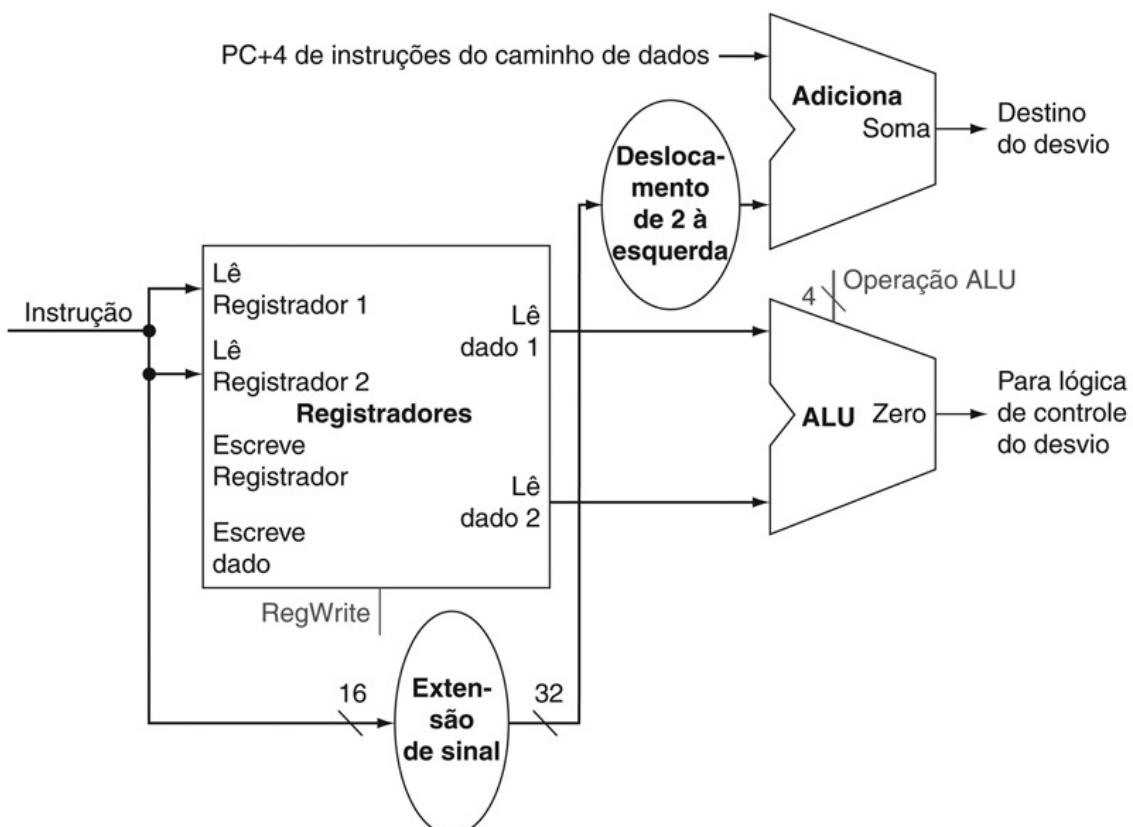
Um desvio em que a condição de desvio é satisfeita, e o contador do programa (PC) se torna o destino do desvio. Todos os desvios incondicionais são desvios tomados.

## desvio não tomado

Um desvio em que a condição de desvio é falsa e o contador do programa (PC) se torna o endereço da instrução que acompanha sequencialmente o desvio.

Portanto, o caminho de dados de desvio precisa realizar duas operações: calcular o endereço de destino do desvio e comparar o conteúdo do registrador. (Os desvios também afetam a parte da busca de instrução do caminho de dados, como veremos em breve.) A [Figura 4.9](#) mostra a estrutura do segmento do caminho de dados que lida com os desvios. Para calcular o endereço de destino do desvio, o caminho de dados de desvio inclui uma unidade de extensão de sinal, exatamente como a da [Figura 4.8](#), e um somador. Para realizar a comparação, precisamos usar o banco de registradores mostrado na [Figura 4.7a](#) a fim de fornecer os dois operandos (embora não precisemos escrever no banco de registradores). Além disso, a comparação pode ser feita usando a ALU que projetamos no Apêndice B. Como essa ALU fornece um sinal de saída que indica se o resultado era 0, podemos enviar os dois operandos de registrador para a ALU com o conjunto de controle de modo a fazer uma subtração. Se o sinal Zero da ALU estiver ativo, sabemos que os dois valores são iguais. Embora a saída de Zero sempre sinalize quando o resultado é 0, nós a estaremos usando apenas para implementar o teste de igualdade dos desvios. Mais adiante, mostraremos exatamente como conectar os sinais de controle da ALU para uso

no caminho de dados.



**FIGURA 4.9** O caminho de dados para um desvio usa a ALU a fim de avaliar a condição de desvio e um somador separado para calcular o destino do desvio como a soma do PC incrementado e os 16 bits mais baixos da instrução com sinal estendido (o deslocamento do desvio), deslocados de 2 bits para a esquerda.

A unidade rotulada como *Deslocamento de 2 à esquerda* é simplesmente um direcionamento dos sinais entre entrada e saída que acrescenta  $00_{bin}$  à extremidade de baixa ordem do campo offset com sinal estendido; nenhum hardware de deslocamento real é necessário, já que a quantidade de “deslocamento” é constante. Como sabemos que o offset teve o sinal dos seus 16 bits estendido, o deslocamento irá descartar apenas “bits de sinal”. A lógica de controle é usada para decidir se o PC incrementado ou o destino do desvio deve substituir o PC, com base na saída Zero da ALU.

A instrução jump funciona substituindo os 28 bits menos significativos do PC

pelos 26 bits menos significativos da instrução deslocados de 2 bits à esquerda. Esse deslocamento é realizado simplesmente concatenando 00 ao offset do jump, como descrito no [Capítulo 2](#).

## Detalhamento

No conjunto de instruções MIPS, os **desvios são atrasados**, isso significa que a instrução imediatamente posterior ao desvio é sempre executada, *independentemente* da condição de desvio ser verdadeira ou falsa. Quando a condição é falsa, a execução se parece com um desvio normal. Quando a condição é verdadeira, um desvio atrasado primeiro executa a instrução imediatamente posterior ao desvio na ordem sequencial antes de desviar para o endereço de destino do desvio. A motivação para os desvios atrasados surge de como o pipelining afeta os desvios (Seção 4.8). Para simplificar, geralmente ignoramos os desvios atrasados neste capítulo e implementamos uma instrução beq como não sendo atrasado.

### desvio atrasado

Um tipo de desvio em que a instrução imediatamente seguinte ao desvio é sempre executada, independente de a condição do desvio ser verdadeira ou falsa.

## Criando um caminho de dados simples

Agora que examinamos os componentes do caminho de dados necessários para as classes de instrução individualmente, podemos combiná-los em um único caminho de dados e acrescentar o controle para completar a implementação. O caminho de dados mais simples pode tentar executar todas as instruções em um único ciclo de clock. Isso significa que nenhum recurso do caminho de dados pode ser usado mais de uma vez por instrução e, portanto, qualquer elemento necessário mais de uma vez precisa ser duplicado. Então, precisamos de uma memória para instruções separada da memória para dados. Embora algumas unidades funcionais precisem ser duplicadas, muitos dos elementos podem ser compartilhados por diferentes fluxos de instrução.

Para compartilhar um elemento do caminho de dados entre duas classes de instrução diferentes, talvez tenhamos de permitir múltiplas conexões com a entrada de um elemento usando um multiplexador e um sinal de controle para

selecionar entre as múltiplas entradas.

## Construindo um caminho de dados

### Exemplo

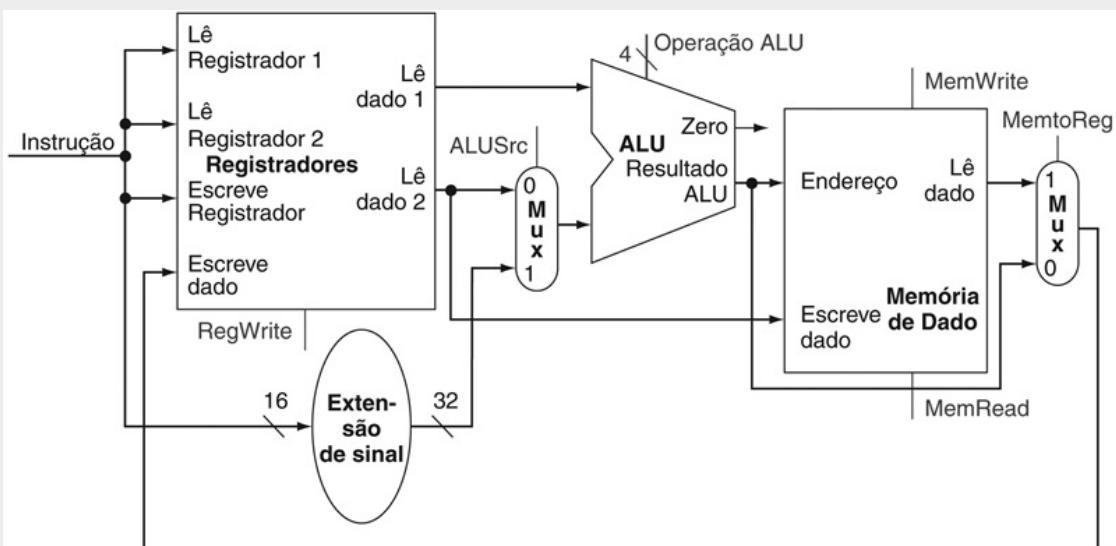
As operações do caminho de dados das instruções lógicas e aritméticas (ou tipo R) e das instruções de acesso à memória são muito semelhantes. As principais diferenças são as seguintes:

- As instruções lógicas e aritméticas usam a ALU com as entradas vindas de dois registradores. As instruções de acesso à memória também podem usar a ALU para fazer o cálculo do endereço, embora a segunda entrada seja o campo offset de 16 bits com sinal estendido da instrução.
- O valor armazenado em um registrador de destino vem da ALU (para uma instrução tipo R) ou da memória (para um load).

Mostre como construir um caminho de dados para a parte operacional das instruções de referência à memória e instruções lógicas e aritméticas, que use um único banco de registradores e uma única ALU para manipular os dois tipos de instrução, incluindo quaisquer multiplexadores necessários.

### Resposta

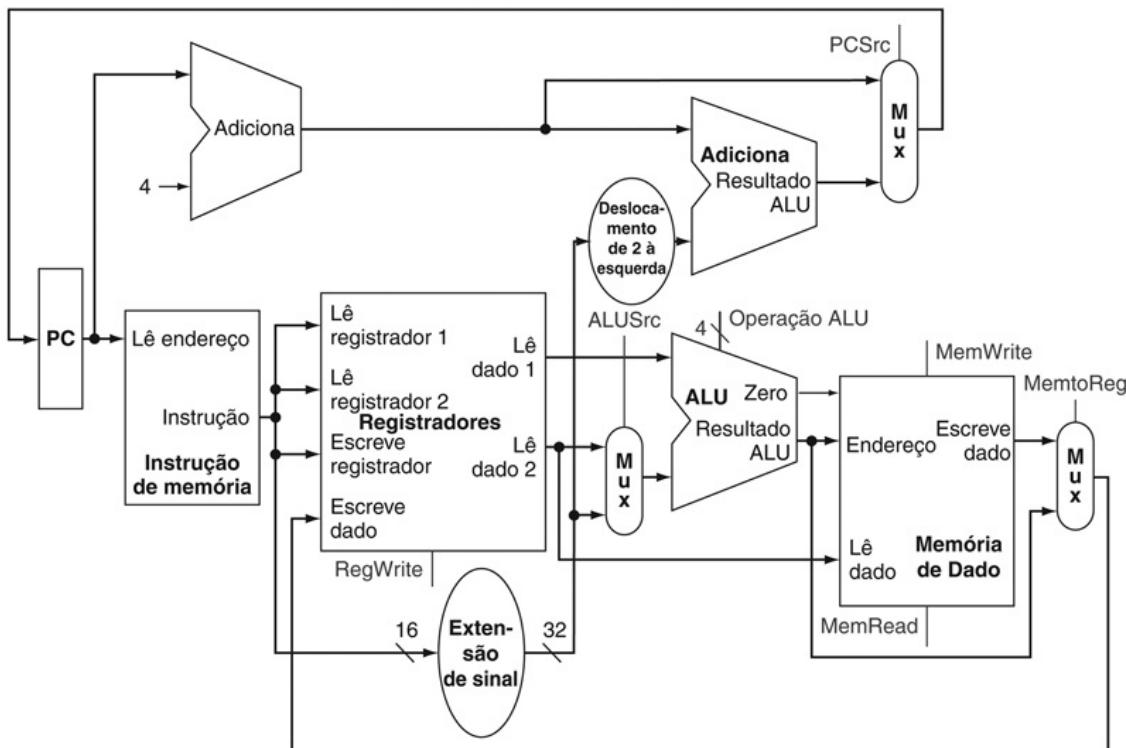
Para criar um caminho de dados com apenas um único banco de registradores e uma única ALU, precisamos suportar duas origens diferentes para a segunda entrada da ALU, bem como duas origens diferentes para os dados armazenados no banco de registradores. Portanto, um multiplexador é colocado na entrada da ALU e outro na entrada de dados para o banco de registradores. A Figura 4.10 mostra a parte operacional do caminho de dados combinado.



**FIGURA 4.10** O caminho de dados para as instruções de acesso à memória e as instruções tipo R.

Este exemplo mostra como um único caminho de dados pode ser montado, a partir das partes nas Figuras 4.7 e 4.8 acrescentando multiplexadores. Dois multiplexadores são necessários, como descrito no exemplo.

Agora, podemos combinar todas as partes de modo a criar um caminho de dados simples para a arquitetura do núcleo MIPS incluindo um caminho de dados para busca de instruções (Figura 4.6), o caminho de dados das instruções tipo R e de acesso à memória (Figura 4.10) e o caminho de dados para desvios (Figura 4.9). A Figura 4.11 mostra o caminho de dados que obtemos compondo as partes separadas. A instrução de desvio usa a ALU principal para comparação dos operandos registradores, de modo que precisamos manter o somador da Figura 4.9, a fim de calcular o endereço de destino do desvio. Um multiplexador adicional é necessário para selecionar o endereço de instrução seguinte ( $PC + 4$ ) ou o endereço de destino do desvio a ser escrito no PC.



**FIGURA 4.11** O caminho de dados simples para a arquitetura MIPS combina os elementos necessários para diferentes classes de instrução.

Os componentes vêm das [Figuras 4.6, 4.9 e 4.10](#). Este caminho de dados pode executar as instruções básicas (load-store word, operações da ALU e desvios) em um único ciclo de clock.

Apenas um multiplexador adicional é necessário para integrar os desvios. O suporte para jumps será incluído mais tarde.

Agora que completamos este caminho de dados simples, podemos acrescentar a unidade de controle. A unidade de controle precisa ser capaz de ler entradas e gerar um sinal de escrita para cada elemento de estado, o controle seletor de cada multiplexador e o controle da ALU. O controle da ALU é diferente de várias maneiras e será útil projetá-lo primeiro, antes de projetarmos o restante da unidade de controle.

# Verifique você mesmo

- I. Qual das seguintes afirmativas é correta para uma instrução load?  
Consulte a Figura 4.10.

  - a. MemtoReg deve ser definido para fazer com que os dados da memória sejam enviados ao banco de registradores.

- b. MemtoReg deve ser definido para fazer com que o registrador de destino correto seja enviado ao banco de registradores.
- c. Não precisamos nos importar com MemtoReg para loads.

- II. O caminho de dados de ciclo único descrito conceitualmente nesta seção *precisa* ter memórias de instrução e dados separadas, porque:
- a. os formatos dos dados e das instruções são diferentes no MIPS, e, portanto, memórias diferentes são necessárias.
  - b. ter memórias separadas é menos dispendioso.
  - c. o processador opera em um ciclo e não pode usar uma memória de porta simples para dois acessos diferentes dentro desse ciclo.

## 4.4. Um esquema de implementação simples

Nesta seção, veremos o que poderia ser considerado a implementação mais simples possível do nosso subconjunto MIPS. Construímos essa implementação simples usando o caminho de dados da última seção e acrescentando uma função de controle simples. Essa implementação simples cobre as instruções *load word* (lw), *store word* (sw), *branch equal* (beq) e as instruções lógicas e aritméticas add, sub, AND, OR e set on less than. Posteriormente, desenvolveremos o projeto para incluir uma instrução jump (j).

### O controle da ALU

A ALU MIPS no Apêndice B define as 6 combinações a seguir das quatro entradas de controle:

Linhas de controle da ALU	Função
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Dependendo da classe de instrução, a ALU precisará realizar uma dessas cinco primeiras funções. (NOR é necessária para outras partes do conjunto de instruções MIPS não encontradas no subconjunto que estamos implementando.) Para as instruções load word e store word, usamos a ALU para calcular o

endereço de memória por adição. Para instruções tipo R, a ALU precisa realizar uma das cinco ações (AND, OR, subtract, add ou set on less than), dependendo do valor do campo funct (ou function – função) de 6 bits nos bits menos significativos da instrução ([Capítulo 2](#)). Para branch equal, a ALU precisa realizar uma subtração.

Podemos gerar a entrada do controle da ALU de 4 bits usando uma pequena unidade de controle que tenha como entradas o campo funct da instrução e um campo control de 2 bits, que chamamos de ALUOp. ALUOp indica se a operação a ser realizada deve ser add (00) para loads e stores, subtract (01) para beq ou determinada pela operação codificada no campo funct (10). A saída da unidade de controle da ALU é um sinal de 4 bits que controla diretamente a ALU gerando uma das combinações de 4 bits mostradas anteriormente.

Na [Figura 4.12](#), mostramos como definir as entradas do controle da ALU com base no controle ALUOp de 2 bits e no código de função de 6 bits. Mais adiante neste capítulo, veremos como os bits de ALUOp são gerados na unidade de controle principal.

Opcode da instrução	OpALU	Operação da instrução	Campo funct	Ação da ALU desejada	Entrada do controle da ALU
LW	00	load word		add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
tipo R	10	add	100000	add	0010
tipo R	10	subtract	100010	subtract	0110
tipo R	10	AND	100100	AND	0000
tipo R	10	OR	100101	OR	0001
tipo R	10	set on less than	101010	set on less than	0111

**FIGURA 4.12** A forma como os bits de controle da ALU são definidos, dependendo dos bits de controle de ALUOp e dos diferentes códigos de função para as instruções tipo R.

O opcode, que aparece na primeira coluna, determina a definição dos bits de ALUOp. Todas as codificações são mostradas em binário. Observe que quando o código de ALUOp é 00 ou 01, a ação da ALU desejada não depende do campo de código de função; nesse caso, dizemos que “não nos importamos” (don’t care) com o valor do código de função e o campo funct aparece como XXXXXX. Quando o valor de ALUOp é 10, então o código de função é usado para definir a entrada do controle da ALU. Veja o [Apêndice B](#).

Esse estilo de usar vários níveis de decodificação — ou seja, a unidade de controle principal gera os bits de ALUOp, que, então, são usados como entrada para o controle da ALU que gera os sinais reais para controlar a ALU — é uma técnica de implementação comum. Usar níveis múltiplos de controle pode reduzir o tamanho da unidade de controle principal. Usar várias unidades de controle menores também pode aumentar a velocidade da unidade de controle. Essas otimizações são importantes, pois a velocidade da unidade de controle normalmente é essencial para o tempo de ciclo de clock.

Há várias maneiras diferentes de implementar o mapeamento do campo ALUOp de 2 bits e do campo funct de 6 bits para os 3 bits de controle de operação da ALU. Como apenas um pequeno número dos 64 valores possíveis do campo funct são de interesse e o campo funct é usado apenas quando os bits de ALUOp são iguais a 10, podemos usar uma pequena lógica que reconhece o subconjunto dos valores possíveis e faz a definição correta dos bits de controle da ALU.

Como uma etapa no projeto dessa lógica, é útil criar uma tabela verdade para as combinações interessantes do campo de código funct e dos bits de ALUOp, como fizemos na [Figura 4.13](#); essa **tabela verdade** mostra como o controle da ALU de 4 bits é definido, de acordo com esses dois campos de entrada. Como a tabela verdade inteira é muito grande ( $2^8 = 256$  entradas) e não nos importamos com o valor do controle da ALU para muitas dessas combinações de entrada, mostramos apenas as entradas para as quais o controle da ALU precisa ter um valor específico. Em todo este capítulo, usaremos essa prática de mostrar apenas as entradas da tabela verdade que precisam ser declaradas e não mostrar as que estão zeradas ou que não nos interessam.

ALUOp		Campo funct						Operação
ALUOp1	ALUOp2	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

**FIGURA 4.13** A tabela verdade para os 4 bits de controle da ALU (chamados Operação).

As entradas são ALUOp e o campo de código de função. Apenas

as entradas para as quais o controle da ALU é ativado são mostradas. Algumas entradas don't care foram incluídas. Por exemplo, como ALUOp não usa a codificação 11, a tabela verdade pode conter entradas 1X e X1, em vez de 10 e 01. Além disso, quando o campo funct é usado, os dois primeiros bits (F5 e F4) dessas instruções são sempre 10; portanto, eles são termos don't care e são substituídos por XX na tabela verdade.

## tabela verdade

Pela lógica, uma representação de uma operação lógica listando todos os valores das entradas e em seguida, em cada caso, mostrando quais deverão ser as saídas resultantes.

Como, em muitos casos, não nos interessamos pelos valores de algumas das **entradas** e para mantermos as tabelas compactas, também incluímos **termos don't care**. Um termo don't care nessa tabela verdade (representado por um X em uma coluna de entrada) indica que a saída não depende do valor da entrada correspondente a essa coluna. Por exemplo, quando os bits de ALUOp são 00, como na primeira linha da tabela na [Figura 4.13](#), sempre definimos o controle da ALU em 0010, independente do código funct. Nesse caso, então, as entradas do código funct serão don't care nessa linha da tabela verdade. Depois, veremos exemplos de outro tipo de termo don't care. Se você não estiver familiarizado com o conceito de termos don't care, veja o Apêndice B para obter mais informações.

## termo don't care

Um elemento de uma função lógica em que a saída não depende dos valores de todas as entradas. Os termos don't care podem ser especificados de diversas maneiras.

Uma vez construída a tabela, ela pode ser otimizada e depois transformada em portas lógicas. Esse processo é completamente mecânico.

## Projetando a unidade de controle principal

Agora que descrevemos como projetar uma ALU que usa o código de função e

um sinal de 2 bits como suas entradas de controle, podemos voltar a considerar o restante do controle. Para começar esse processo, vamos identificar os campos de uma instrução e as linhas de controle necessárias para o caminho de dados construído na [Figura 4.11](#). A fim de entender como conectar os campos de uma instrução com o caminho de dados, é útil examinar os formatos das três classes de instrução: as instruções tipo R, as instruções de desvio e as instruções load-store. A [Figura 4.14](#) mostra esses formatos.

Campo	0	rs	rt	rd	shamt	funct
Posição de bit	31:26	25:21	20:16	15:11	10:6	5:0
a. Instrução do tipo R						
Campo	35 ou 43	rs	rt	address		
Posição de bit	31:26	25:21	20:16	15:0		
b. Carrega ou armazena instrução						
Campo	4	rs	rt	address		
Posição de bit	31:26	25:21	20:16	15:0		
c. Instrução de desvio						

**FIGURA 4.14 As três classes de instrução (tipo R, load/store e desvio) usam dois formatos de instrução diferentes.**

As instruções jump usam outro formato, que será discutido em breve. (a) Formato de instrução para instruções tipo R, as quais possuem todas opcode 0. Essas instruções possuem três registradores como operandos: rs, rt e rd. Os campos rs e rt são origens e rd é o destino. A função da ALU está no campo funct e é decodificada pelo projeto de controle da ALU da seção anterior. As instruções tipo R que implementamos são add, sub, AND, OR e slt. O campo shamt é usado apenas para deslocamentos; nós o ignoraremos neste capítulo. (b) Formato de instrução para instruções load (opcode = 35<sub>dec</sub>) e store (opcode = 43<sub>dec</sub>). O registrador rs é o registrador de base adicionado ao campo address de 16 bits de modo a formar o endereço de memória. Com os loads, rt é o registrador de destino para o valor lido. Com stores, rt é o registrador de origem cujo valor deve ser armazenado na memória. (c) Formato de instrução para branch equal (opcode = 4). Os registradores rs e rt são os registradores de origem que são comparados por igualdade. O campo address de 16 bits tem seu sinal estendido, é deslocado e somado ao PC + 4 para calcular o endereço de

destino do desvio.

Existem várias observações importantes sobre esses formatos de instrução em que nos basearemos:

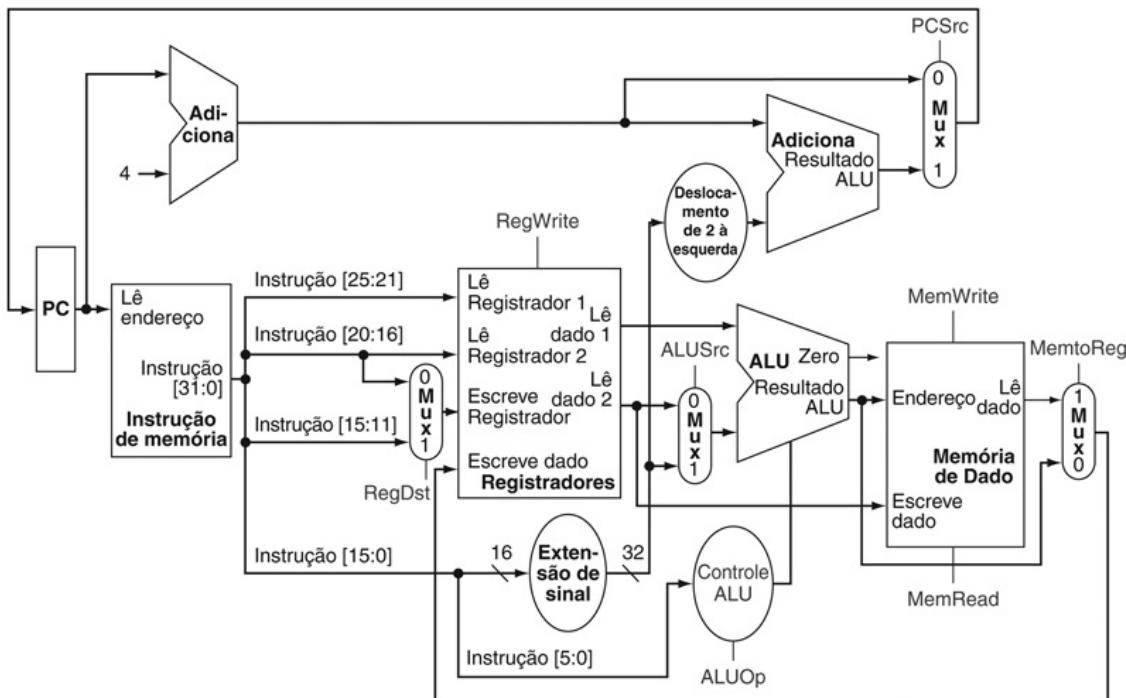
- Campo op, também chamado **opcode** no [Capítulo 2](#), está sempre contido nos bits 31:26. Iremos nos referir a esse campo como Op[5:0].
- Os dois registradores a serem lidos sempre são especificados pelos campos rs e rt, nas posições 25:21 e 20:16. Isso é verdade para as instruções tipo R, branch equal e store.
- Registrador de base para as instruções load e store está sempre nas posições de bit 25:21 (rs).
- Offset de 16 bits para branch equal, load e store está sempre nas posições 15:0.
- Registrador de destino está em um de dois lugares. Para um load, ele está nas posições 20:16 (rt), enquanto para uma instrução tipo R, ele está nas posições 15:11 (rd). Portanto, precisaremos incluir um multiplexador a fim de selecionar que campo da instrução será usado para indicar o número de registrador a ser escrito.

## opcode

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

A vantagem do primeiro princípio de projeto do [Capítulo 2](#) — *a simplicidade favorece a regularidade* — aparece aqui na especificação do controle.

Usando essas informações, podemos acrescentar os rótulos de instrução e o multiplexador extra (para a entrada Escreve registrador do banco de registradores) no caminho de dados simples. A [Figura 4.15](#) mostra essas adições, além do bloco de controle da ALU, os sinais de escrita para elementos de estado, o sinal de leitura para a memória de dados e os sinais de controle para os multiplexadores. Como todos os multiplexadores possuem duas entradas, cada um deles requer uma única linha de controle.



**FIGURA 4.15** O caminho de dados da [Figura 4.11](#) com todos os multiplexadores necessários e todas as linhas de controle identificadas.

As linhas de controle são mostradas em tons de cinza. O bloco de controle da ALU também foi acrescentado. O PC não exige um controle de escrita, já que ele é escrito uma vez no fim de cada ciclo de clock; a lógica de controle de desvio determina se ele é escrito com o PC incrementado ou o endereço de destino do desvio.

A [Figura 4.15](#) mostra sete linhas de controle de um único bit mais o sinal de controle ALUOp de 2 bits. Já definimos como o sinal de controle ALUOp funciona e é útil definir o que fazem os outros sete sinais de controle informalmente antes de determinarmos como definir esses sinais de controle durante a execução da instrução. A [Figura 4.16](#) descreve a função dessas sete linhas de controle.

Nome do sinal	Efeito quando inativo	Efeito quando ativo
RegDst	O número do registrador destino para entrada Registrador para escrita vem do campo rt (bits 20:16).	O número do registrador destino para a entrada Registrador para escrita vem do campo rd (bits 15:11).
EscreveReg	Nenhum.	O registrador na entrada Registrador para escrita é escrito com o valor da entrada Dados para escrita.
OrigALU	O segundo operando da ALU vem da segunda saída do banco de registradores (Dados da leitura 2).	O segundo operando da ALU consiste nos 16 bits mais baixos da instrução com sinal estendido.
OrigPC	O PC é substituído pela saída do somador que calcula o valor de PC + 4.	O PC é substituído pela saída do somador que calcula o destino do desvio.
LeMem	Nenhum.	O conteúdo da memória de dados designado pela entrada Endereço é colocado na saída Dados da leitura.
EscreveMem	Nenhum.	O conteúdo da memória de dados designado pela entrada Endereço é substituído pelo valor na entrada Dados para escrita.
MemparaReg	O valor enviado à entrada Dados para escrita do banco de registradores vem da ALU.	O valor enviado à entrada Dados para escrita do banco de registradores vem da memória de dados.

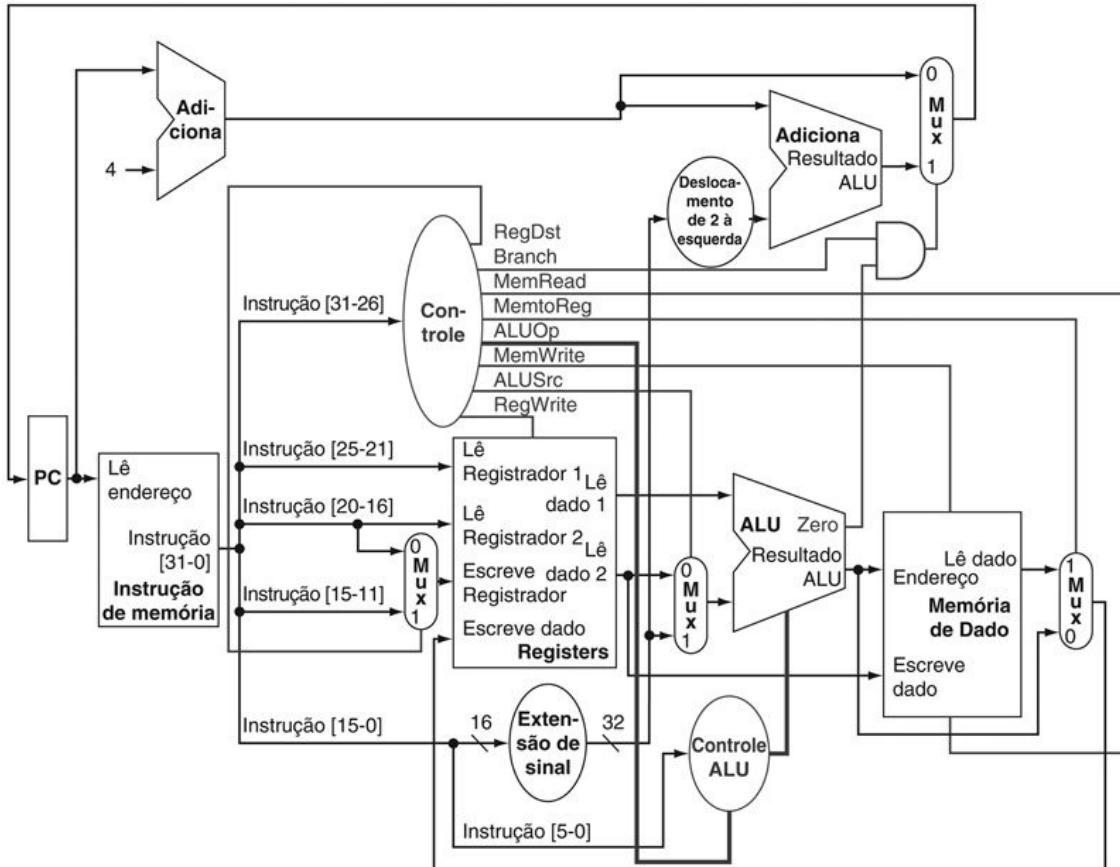
**FIGURA 4.16 O efeito de cada um dos sete sinais de controle.**

Quando o controle de um bit de largura, para um multiplexador com duas entradas, está ativo, o multiplexador seleciona a entrada correspondente a 1. Caso contrário, se o controle não estiver ativo, o multiplexador seleciona a entrada 0. Lembre-se de que todos os elementos de estado têm o clock como uma entrada implícita e que o clock é usado para controlar escritas. O clock nunca vem externamente para um elemento de estado, já que isso pode criar problemas de sincronização. (Veja o **Apêndice B** para obter mais detalhes sobre esse problema.)

Agora que examinamos a função de cada um dos sinais de controle, podemos ver como defini-los. A unidade de controle pode definir todos menos um dos sinais de controle unicamente com base no campo opcode da instrução. A exceção é a linha de controle PCScrPCScr. Essa linha de controle deve ser ativada se a instrução for branch on equal (uma decisão que a unidade de controle pode tomar) e a saída Zero da ALU, usada para comparação de igualdade, for verdadeira. Para gerar o sinal PCScrPCScr, precisaremos realizar um AND de um sinal da unidade de controle, que chamamos *Branch*, com o sinal Zero da ALU.

Esses nove sinais de controle (sete da [Figura 4.16](#) e dois para ALUOp) podem agora ser definidos baseados nos seis sinais de entrada da unidade de controle, que são os bits de opcode 31 a 26. A [Figura 4.17](#) mostra o caminho de dados

com a unidade de controle e os sinais de controle.



**FIGURA 4.17** O caminho de dados simples com a unidade de controle.

A entrada para a unidade de controle é o campo opcode de 6 bits da instrução. As saídas da unidade de controle consistem em três sinais de 1 bit usados para controlar multiplexadores (RegDst, ALUScrALUScr e MemtoReg), três sinais para controlar leituras e escritas no banco de registradores e na memória de dados (WriteReg, ReadMem e WriteMem), um sinal de 1 bit usado na determinação de um possível desvio (Branch) e um sinal de controle de 2 bits para a ALU (ALUOp). Uma porta AND é usada de modo a combinar o sinal de controle de desvio com a saída Zero da ALU; a saída da porta AND controla a seleção do próximo PC. Observe que PCScr é agora um sinal derivado, em vez de um sinal vindo diretamente da unidade de controle. Portanto, descartamos o nome do sinal nas próximas figuras.

Antes de tentarmos escrever um conjunto de equações ou uma tabela verdade

para a unidade de controle, será útil definir a função de controle informalmente. Como a definição das linhas de controle depende apenas do opcode, definimos se cada sinal de controle deve ser 0, 1 ou don't care (X) para cada um dos valores de opcode. A [Figura 4.18](#) descreve como os sinais de controle devem ser definidos para cada opcode; essas informações seguem diretamente das [Figuras 4.12, 4.16 e 4.17](#).

Instrução	RegDst	OrigALU	MemparaReg	EscreveReg	LeMem	EscreveMem	Branch	ALUOp1	ALUOp0
formato R	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

**FIGURA 4.18** A definição das linhas de controle é completamente determinada pelos campos opcode da instrução.

A primeira linha da tabela corresponde às instruções formato R (add, sub, AND, OR e slt). Para todas essas instruções, os campos registradores de origem são rs e rt, e o campo registrador de destino é rd; isso especifica como os sinais ALUScr e RegDst são definidos. Além disso, uma instrução tipo R escreve em um registrador (WriteReg = 1), mas não escreve ou lê a memória de dados. Quando o sinal de controle Branch é 0, o PC é incondicionalmente substituído por PC + 4; caso contrário, o PC é substituído pelo destino do desvio se a saída Zero da ALU também está ativa. O campo ALUOp para as instruções tipo R é definido como 10 a fim de indicar que o controle da ALU deve ser gerado do campo funct. A segunda e a terceira linhas dessa tabela fornecem as definições dos sinais de controle para lw e sw.

Esses campos ALUScr e ALUOp são definidos para realizar o cálculo do endereço. ReadMem e WriteMem são definidos para realizar o acesso à memória. Finalmente, RegDst e WriteReg são definidos para que um load faça o resultado ser armazenado no registrador rt. A instrução branch é semelhante à operação no formato R, já que ela envia os registradores rs e rt para a ALU. O campo ALUOp para um desvio é definido como uma subtração (controle da ALU = 01), usada para testar a igualdade. Repare que o campo MemtoReg é irrelevante quando o sinal WriteReg é 0: como o registrador não está sendo escrito, o valor dos dados na entrada Dados para escrita do banco de registradores não é usado. Portanto, a entrada MemtoReg nas duas últimas linhas da tabela é substituída por X (don't care). Os don't care também podem ser adicionados a RegDst quando WriteReg é 0. Esse tipo de don't care precisa ser acrescentado pelo projetista, uma

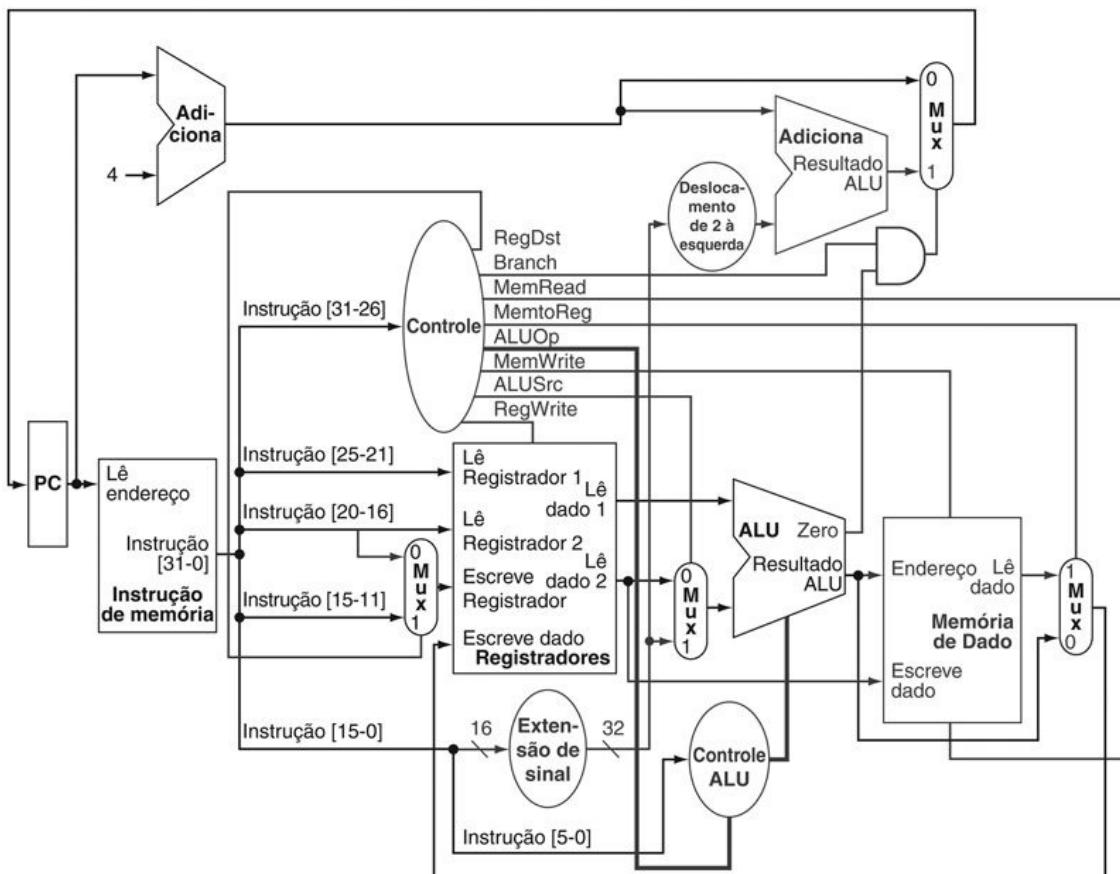
vez que ele depende do conhecimento de como o caminho de dados funciona.

## Operação do caminho de dados

Com as informações contidas nas [Figuras 4.16 e 4.18](#), podemos projetar a lógica da unidade de controle. Antes de fazer isso, porém, vejamos como cada instrução usa o caminho de dados. Nas próximas figuras, mostramos o fluxo das três classes de instrução diferentes por meio do caminho de dados. Os sinais de controle ativos e os elementos do caminho de dados ativos são destacados em cada uma das figuras. Observe que um multiplexador cujo controle é 0 tem uma ação definida, mesmo se sua linha de controle não estiver destacada. Sinais de controle de vários bits são destacados se qualquer sinal constituinte estiver ativo.

A [Figura 4.19](#) mostra a operação do caminho de dados para uma instrução tipo R, como `add $t1,$t2,$t3`. Embora tudo ocorra em um ciclo de clock, podemos pensar em quatro etapas para executar a instrução; essas etapas são ordenadas pelo fluxo da informação:

1. A instrução é buscada e o PC é incrementado.



**FIGURA 4.19** O caminho de dados em operação para uma instrução tipo R como add \$t1,\$t2,\$t3.

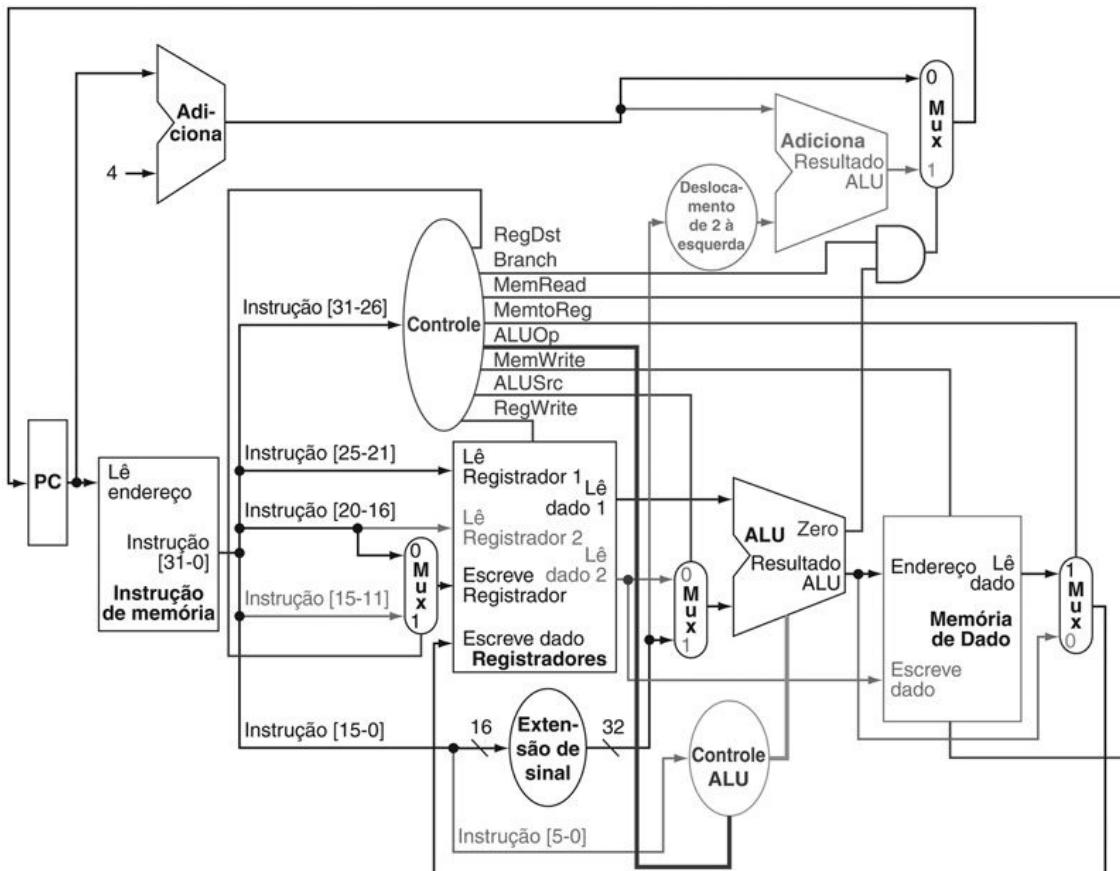
As linhas de controle, as unidades do caminho de dados e as conexões que estão ativas aparecem destacadas.

2. Dois registradores, \$t2 e \$t3, são lidos do banco de registradores, e a unidade de controle principal calcula a definição das linhas de controle também durante essa etapa.
  3. A ALU opera nos dados lidos do banco de registradores, usando o código de função (bits 5:0, que é o campo funct, da instrução) para gerar a função da ALU.
  4. O resultado da ALU é escrito no banco de registradores usando os bits 15:11 da instrução para selecionar o registrador de destino (\$t1).
- Da mesma forma, podemos ilustrar a execução de um load word, como

lw \$t1, offset(\$t2)

em um estilo semelhante à [Figura 4.19](#). A [Figura 4.20](#) mostra as unidades funcionais ativas e as linhas de controle ativas para um load. Podemos pensar em uma instrução load como operando em cinco etapas (semelhante ao tipo R executado em quatro):

1. Uma instrução é buscada da memória de instruções e o PC é incrementado.



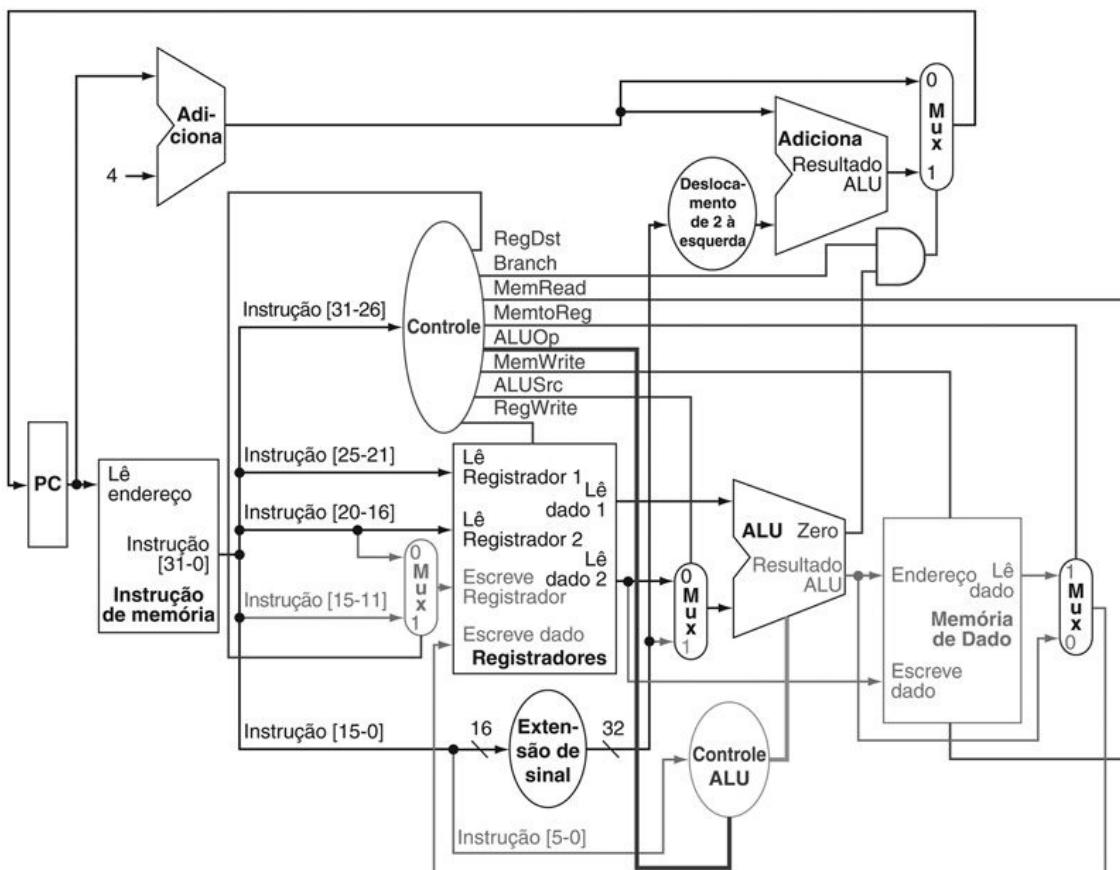
**FIGURA 4.20** O caminho de dados em operação para uma instrução load.

As linhas de controle, as unidades do caminho de dados e as conexões ativas aparecem destacadas. Uma instrução store operaria de maneira muito semelhante. A principal diferença seria que o controle da memória indicaria uma escrita em vez de uma leitura, a segunda leitura do valor de um registrador seria usada para os dados a serem armazenados e a operação de escrita do valor da memória de dados no banco de registradores não ocorreria.

2. Um valor de registrador ( $$t_2$ ) é lido do banco de registradores.
3. A ALU calcula a soma do valor lido do banco de registradores com os 16 bits menos significativos com sinal estendido da instrução (offset).
4. A soma da ALU é usada como o endereço para a memória de dados.
5. Os dados da unidade de memória são escritos no banco de registradores; o registrador de destino é fornecido pelos bits 20:16 da instrução ( $$t_1$ ).

Finalmente, podemos mostrar a operação da instrução branch-on-equal, como `beq $t1,$t2,offset`, da mesma maneira. Ela opera de forma muito parecida com uma instrução de formato R, mas a saída da ALU é usada para determinar se o PC é escrito com  $PC + 4$  ou o endereço de destino do desvio. A Figura 4.21 mostra as quatro etapas da execução:

1. Uma instrução é buscada da memória de instruções e o PC é incrementado.



**FIGURA 4.21** O caminho de dados em operação para uma instrução branch-on-equal.

As linhas de controle, as unidades do caminho de dados e as conexões que estão ativas aparecem destacadas. Após usar o

banco de registradores e a ALU para realizar a comparação, a saída Zero é usada na seleção do próximo contador de programa dentre os dois candidatos.

2. Dois registradores,  $\$t_1$  e  $\$t_2$ , são lidos do banco de registradores.
3. A ALU realiza uma subtração dos valores de dados lidos do banco de registradores. O valor de  $PC + 4$  é somado aos 16 bits menos significativos com sinal estendido (offset) deslocados de dois para a esquerda; o resultado é o endereço de destino do desvio.
4. O resultado Zero da ALU é usado para decidir qual resultado do somador deve ser armazenado no PC.

### Finalizando o controle

Agora que vimos como as instruções operam em etapas, vamos continuar com a implementação do controle. A função de controle pode ser definida com precisão usando o conteúdo da [Figura 4.18](#). As saídas são as linhas de controle, e a entrada é o campo opcode de 6 bits,  $Op [5:0]$ . Portanto, podemos criar uma tabela verdade para cada uma das saídas com base na codificação binária dos opcodes.

A [Figura 4.22](#) mostra a lógica na unidade de controle como uma grande tabela verdade que combina todas as saídas e que usa os bits de opcode como entradas. Ela especifica completamente a função de controle, e podemos implementá-la diretamente em portas lógicas de uma maneira automatizada.

<b>Entrada ou saída</b>	<b>Nome do sinal</b>	<b>formato R</b>	<b>lw</b>	<b>sw</b>	<b>beq</b>
Entradas	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Saídas	RegDst	1	0	X	X
	OrigALU	0	1	1	0
	MemparaReg	0	1	X	X
	EscreveReg	1	1	0	0
	LeMem	0	1	0	0
	EscreveMem	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

**FIGURA 4.22 A função de controle para a implementação de ciclo único simples é completamente especificada por essa tabela verdade.**

A parte superior da tabela fornece combinações de sinais de entrada que correspondem aos quatro opcodes, um por coluna, que determinam as definições de saída do controle. (Lembre-se de que Op [5:0] corresponde aos bits 31:26 da instrução, que é o campo op.) A parte inferior da tabela fornece as saídas para cada um dos quatro opcodes. Portanto, a saída WriteReg é ativada para duas combinações diferentes das entradas. Se considerarmos apenas os quatro opcodes mostrados nessa tabela, então, poderemos simplificar a tabela verdade usando don't care na parte da entrada. Por exemplo, podemos detectar uma instrução no formato R com a expressão [see original, pg 269], uma vez que isso é suficiente para distinguir as instruções no formato R das instruções lw, sw e beq. Não tiramos vantagem dessa simplificação, já que o restante dos opcodes MIPS é usado em uma implementação completa.

Agora que temos uma **implementação de ciclo único** da maioria dos conjuntos de instruções MIPS básico, vamos acrescentar a instrução jump para mostrar como o caminho de dados básico e o controle podem ser estendidos ao lidar com outras instruções no conjunto de instruções.

## implementação de ciclo único

Também chamada de implementação de ciclo de clock único. Uma

implementação em que uma instrução é executada em um único ciclo de clock.

## Implementando jumps

### Exemplo

A Figura 4.17 mostra a implementação de várias instruções vistas no Capítulo 2. Uma classe de instruções ausente é a da instrução jump. Estenda o caminho de dados e o controle da Figura 4.17 para incluir a instrução jump. Descreva como definir quaisquer novas linhas de controle.

### Resposta

A instrução jump, mostrada na Figura 4.23, se parece um pouco com uma instrução branch, mas calcula o PC de destino de maneira diferente e não é condicional. Como um branch, os 2 bits menos significativos de um endereço jump são sempre  $00_{bin}$ . Os próximos 26 bits menos significativos desse endereço de 32 bits vêm do campo imediato de 26 bits na instrução. Os 4 bits superiores do endereço que deve substituir o PC vêm do PC da instrução jump mais 4. Portanto, podemos implementar um jump armazenando no PC a concatenação de:

- os 4 bits superiores do PC atual + 4 (esses são bits 31:28 do endereço da instrução imediatamente seguinte);
- campo de 26 bits imediato da instrução jump;
- os bits  $00_{bin}$ .

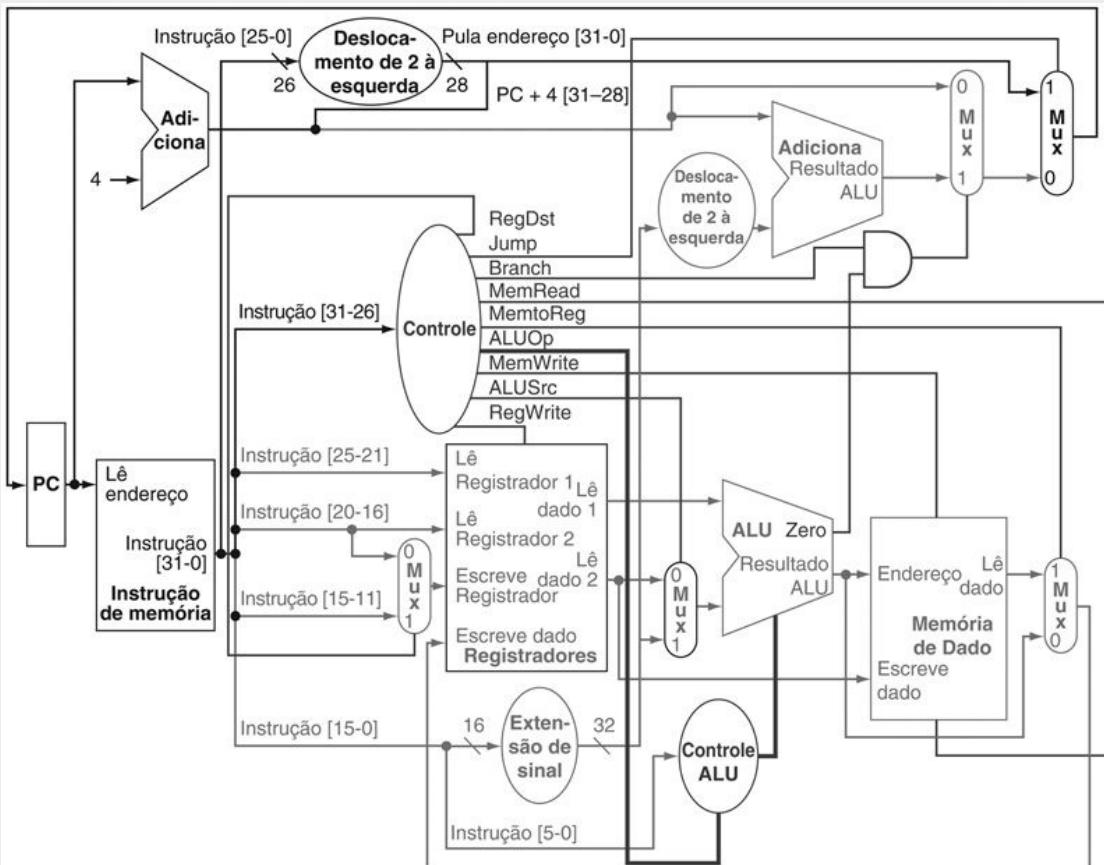
Campo	000010	endereço
Posições dos bits	31:26	25:0

**FIGURA 4.23 Formato de instrução para a instrução jump (opcode = 2).**

O endereço de destino para uma instrução jump é formado pela concatenação dos 4 bits superiores do PC atual + 4, com o campo endereço de 26 bits na instrução jump e pela adição de 00 como os dois bits menos significativos.

A Figura 4.24 mostra a adição do componente para jump à Figura 4.17. Um outro multiplexador é usado na seleção da origem para o novo valor do PC,

que pode ser o PC incrementado ( $PC + 4$ ), o PC de destino de um branch ou o PC de destino de um jump. Um sinal de controle adicional é necessário para o multiplexador adicional. Esse sinal de controle, chamado *Jump*, é ativado apenas quando a instrução é um jump — ou seja, quando o opcode é 2.



**FIGURA 4.24** O controle e o caminho de dados simples são estendidos para lidar com a instrução jump.

Um multiplexador adicional (no canto superior direito) é usado na escolha entre o destino de um jump e o destino de um desvio ou a instrução sequencial seguinte a esta. Esse multiplexador é controlado pelo sinal de controle *Jump*. O endereço de destino do jump é obtido deslocando-se os 26 bits inferiores da instrução jump de 2 bits para a esquerda, efetivamente adicionando 00 como os bits menos significativos, e, depois, concatenando os 4 bits mais significativos do  $PC + 4$  como os bits mais significativos, produzindo, assim, um endereço de 32 bits.

## Por que uma implementação de ciclo único não é usada hoje

Embora o projeto de ciclo único funcionasse corretamente, ele não seria usado nos projetos modernos porque é ineficiente. Para ver o porquê disso, observe que o ciclo de clock precisa ter a mesma duração para cada instrução nesse projeto de ciclo único. É claro, o ciclo de clock é determinado pelo caminho mais longo possível no processador. Esse caminho é, quase certamente, uma instrução load, que usa cinco unidades funcionais em série: a memória de instruções, o banco de registradores, a ALU, a memória de dados e o banco de registradores. Embora o CPI seja 1 ([Capítulo 1](#)), o desempenho geral de uma implementação de ciclo único provavelmente será fraco, já que o ciclo de clock é muito longo.

O ônus de usar o projeto de ciclo único com um ciclo de clock fixo é significativo, mas poderia ser considerado aceitável para este pequeno conjunto de instruções. Historicamente, os primeiros computadores com conjuntos de instruções muito simples usavam essa tecnologia de implementação. Entretanto, se tentássemos implementar a unidade de ponto flutuante ou um conjunto de instruções com orientações mais complexas, esse projeto de ciclo único decididamente não funcionaria bem.

Como precisamos considerar que o ciclo de clock é igual ao atraso do pior caso para todas as instruções, não podemos usar técnicas de implementação que reduzem o atraso do caso comum, mas não melhoram o tempo de ciclo de pior caso. Uma implementação de ciclo único, portanto, viola o nosso princípio básico de projeto do [Capítulo 2](#) de tornar o **caso comum veloz**.



**CASO COMUM VELOZ**

Na próxima seção, veremos outra técnica de implementação, chamada pipelining, que usa um caminho de dados muito semelhante ao caminho de dados de ciclo único, mas é muito mais eficiente por ter uma vazão muito mais alta. O pipelining melhora a eficiência executando múltiplas instruções simultaneamente.

### Verifique você mesmo

Veja os sinais de controle na Figura 4.22. Você consegue combinar alguns deles? Algum sinal de controle na figura pode ser substituído pelo inverso de outro? (Dica: leve em conta os don't care.) Nesse caso, você pode trocar um sinal pelo outro sem incluir um inverSOR?

## 4.5. Visão geral de pipelining

*Nunca perca tempo.*

*Provérbio americano*

**Pipelining** é uma técnica de implementação em que várias instruções são sobrepostas na execução. Hoje, a técnica de pipelining é praticamente universal.

### pipelining

Uma técnica de implementação em que várias instruções são sobrepostas na execução, semelhante a uma linha de montagem.

Esta seção utiliza bastante uma analogia para dar uma visão geral dos termos e aspectos da técnica de pipelining. Se você estiver interessado apenas no quadro geral, deverá se concentrar nesta seção e depois pular para as [Seções 4.10 e 4.11](#), a fim de ver uma introdução às técnicas de pipelining avançadas, utilizadas nos processadores mais recentes, como o Intel Core i7 e o ARM Cortex-A8. Se estiver interessado em explorar a anatomia de um computador com pipeline, esta seção é uma boa introdução às [Seções de 4.6 a 4.9](#).

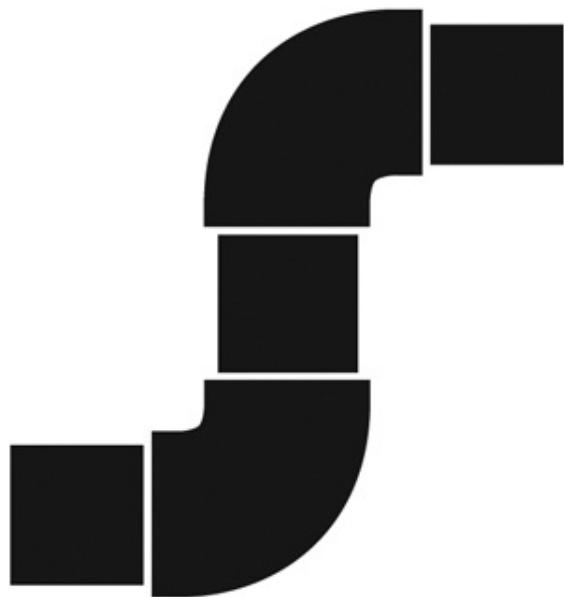
Qualquer um que tenha lavado muitas roupas intuitivamente já usou pipelining. A técnica sem pipeline para lavar roupas seria:

1. Colocar a trouxa de roupa suja na lavadora.
2. Quando a lavadora terminar, colocar a trouxa de roupa molhada na

secadora.

3. Quando a secadora terminar, colocar a trouxa de roupa seca na mesa e passar.
4. Quando terminar de passar, pedir ao seu colega de quarto para guardar as roupas.

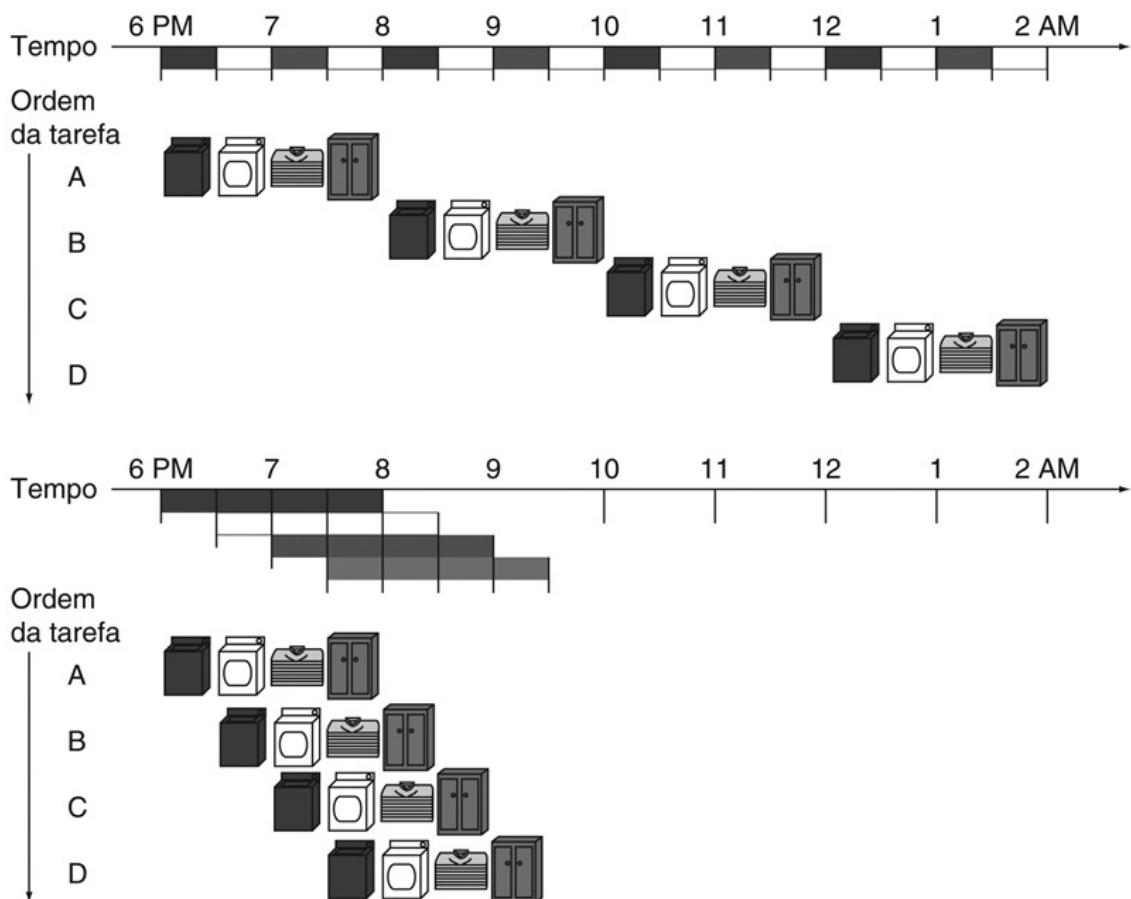
Quando seu colega terminar, então comece novamente com a próxima trouxa de roupa suja.



## PIPELINING

A técnica com *pipeline* leva muito menos tempo, como mostra a [Figura 4.25](#). Assim que a lavadora terminar com a primeira trouxa e ela for colocada na secadora, você carrega a lavadora com a segunda trouxa de roupa suja. Quando a primeira trouxa estiver seca, você a coloca na tábua para começar a passar e dobrar, move a trouxa de roupa molhada para a secadora e a próxima trouxa de roupa suja para a lavadora. Em seguida, você pede a seu colega para guardar a primeira remessa, começa a passar e dobrar a segunda, a secadora está com a terceira remessa e você coloca a quarta na lavadora. Nesse ponto, todas as etapas — denominadas *estágios* em pipelining — estão operando simultaneamente. Desde que haja recursos separados para cada estágio, podemos usar um pipeline

para as tarefas.



**FIGURA 4.25** A analogia da lavagem de roupas para pipelining.

Ana, Beto, Catarina e Davi possuem roupas sujas para serem lavadas, secadas, passadas e guardadas. O lavador, o secador, o passador e o guardador levam 30 minutos para sua tarefa. A lavagem sequencial levaria oito horas para quatro trouxas de roupas, enquanto a lavagem com pipeline levaria apenas 3,5 horas. Mostramos o estágio do pipeline de diferentes trouxas com o passar do tempo, mostrando cópias dos quatro recursos nessa linha de tempo bidimensional, mas na realidade temos apenas um de cada recurso.

O paradoxo da técnica de pipelining é que o tempo desde a colocação de uma única trouxa de roupa suja na lavadora até que ela esteja seca, e seja passada e guardada não é mais curto para a técnica de pipelining; o motivo pelo qual a

técnica de pipelining é mais rápida para muitas trouxas é que tudo está trabalhando em paralelo, de modo que mais trouxas são terminadas por hora. A técnica de pipelining melhora a vazão do sistema de lavanderia. Logo, a técnica de pipelining não diminuiria o tempo para concluir uma trouxa de roupas, mas, quando temos muitas trouxas para lavar, a melhoria na vazão diminui o tempo total de conclusão do trabalho.

Se todos os estágios levarem aproximadamente o mesmo tempo e houver trabalho suficiente para realizar, então o ganho de velocidade devido à técnica de pipelining será igual ao número de estágios do pipeline, neste caso, quatro: lavar, secar, passar e guardar. Assim, a lavanderia com pipeline é potencialmente quatro vezes mais rápida do que a sem pipeline: 20 trouxas levariam cerca de cinco vezes o tempo de uma trouxa, enquanto 20 trouxas de lavagem sequencial levariam 20 vezes o tempo de uma trouxa. O ganho foi de apenas 2,3 vezes na [Figura 4.25](#) porque mostramos apenas quatro trouxas. Observe que, no início e no final da carga de trabalho na versão com pipeline da [Figura 4.25](#), o pipeline não está completamente cheio. Esse efeito no início e no fim afeta o desempenho quando o número de tarefas não é grande em comparação com a quantidade de estágios do pipeline. Se o número de trouxas for muito maior que 4, então os estágios estarão cheios na maior parte do tempo e o aumento na vazão será muito próximo de 4.

Os mesmos princípios se aplicam a processadores em que usamos pipeline para a execução da instrução. As instruções MIPS normalmente exigem cinco etapas:

1. Buscar instrução da memória.
2. Ler registradores enquanto a instrução é decodificada. O formato das instruções MIPS permite que a leitura e a decodificação ocorram simultaneamente.
3. Executar a operação ou calcular um endereço.
4. Acessar um operando na memória de dados.
5. Escrever o resultado em um registrador.

Logo, o pipeline MIPS que exploramos neste capítulo possui cinco estágios. O exemplo a seguir mostra que a técnica de pipelining agiliza a execução da instrução, assim como agiliza a lavagem de roupas.

## Desempenho de ciclo único *versus* desempenho com pipeline

## Exemplo

Para tornar esta discussão concreta, vamos criar um pipeline. Neste exemplo, e no restante deste capítulo, vamos limitar nossa atenção a oito instruções: load word (lw), store word (sw), add (add), subtract (sub), AND (and), OR (or), set-less-than (slt) e branch-on-equal (beq).

Compare o tempo médio entre as instruções de uma implementação em ciclo único, em que todas as instruções levam um ciclo de clock, com uma implementação com pipeline. Os tempos de operação para as principais unidades funcionais neste exemplo são de 200 ps para acesso à memória, 200 ps para operação com ALU e 100 ps para leitura ou escrita de registradores. No modelo de ciclo único, cada instrução leva exatamente um ciclo de clock, de modo que o ciclo precisa ser esticado para acomodar a instrução mais lenta.

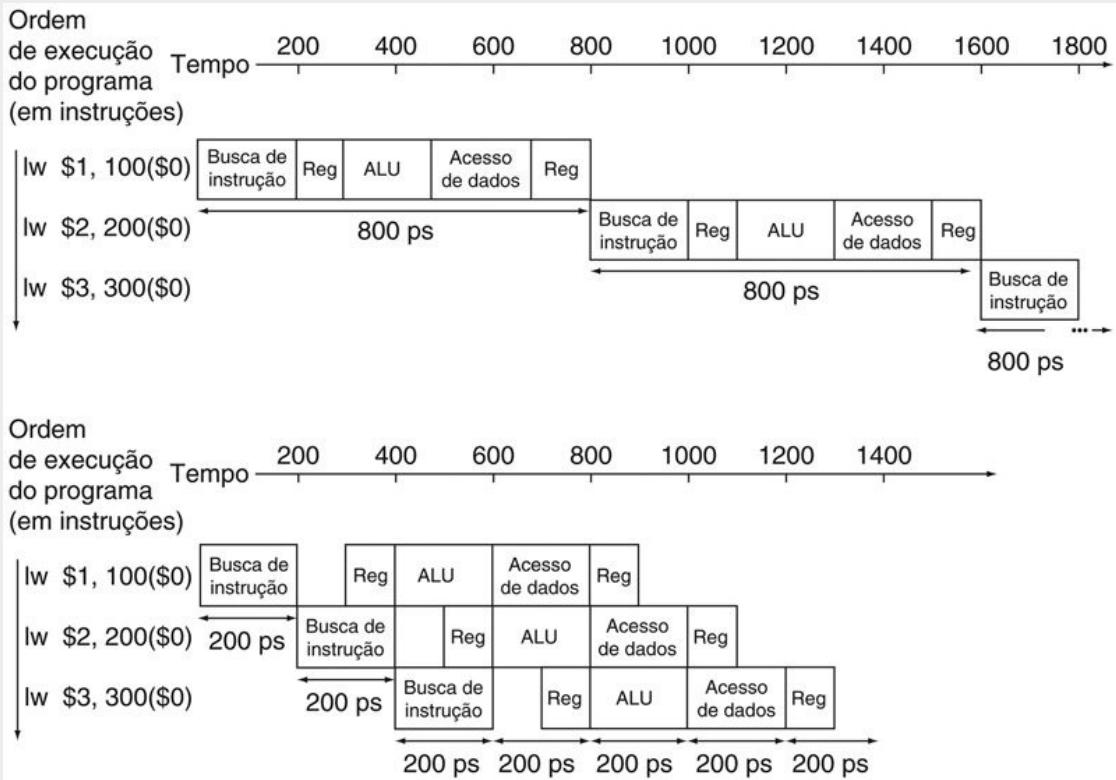
## Resposta

A Figura 4.26 mostra o tempo exigido para cada uma das oito instruções. O projeto de ciclo único precisa contemplar a instrução mais lenta — na Figura 4.26, ela é lw — de modo que o tempo exigido para cada instrução é 800 ps. Assim como na Figura 4.25, a Figura 4.27 compara a execução sem pipeline e com pipeline de três instruções load word. Desse modo, o tempo entre a primeira e a quarta instrução no projeto sem pipeline é  $3 \times 800$  ps ou 2.400 ps.

Classe de instrução	Busca de instrução	Leitura do registrador	Operação ALU	Acesso de dados	Escrita do registrador	Tempo Total
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

**FIGURA 4.26** Tempo total para cada instrução calculada a partir do tempo para cada componente.

Esse cálculo considera que os multiplexadores, unidade de controle, acessos ao PC e unidade de extensão de sinal não possuem atraso.



**FIGURA 4.27 Em cima, execução em ciclo único, sem pipeline, versus execução com pipeline, embaixo.**

Ambas utilizam os mesmos componentes de hardware, cujo tempo está listado na Figura 4.26. Neste caso, vemos um ganho de velocidade de quatro vezes no tempo médio entre as instruções, de 800 ps para 200 ps. Compare com a Figura 4.25. Para a lavanderia, consideramos que todos os estágios eram iguais. Se a secadora fosse mais lenta, então o estágio da secadora definiria o tempo do estágio. Os tempos de estágio do pipeline dos computadores são limitados pelo recurso mais lento, seja a operação da ALU ou o acesso à memória. Consideramos que a escrita no banco de registradores ocorre na primeira metade do ciclo de clock e a leitura do banco de registradores ocorre na segunda metade. Usamos essa suposição por todo este capítulo.

Todos os estágios do pipeline utilizam um único ciclo de clock, de modo que ele precisa ser grande o suficiente para acomodar a operação mais lenta. Assim como o projeto de ciclo único de clock precisa levar o tempo do ciclo de clock no pior caso, de 800 ps, embora algumas instruções possam ser tão rápidas quanto 500 ps, o ciclo de clock da execução com pipeline precisa ter o ciclo de clock, no pior caso, de 200 ps, embora alguns estágios levem apenas

100 ps. O uso de pipeline ainda oferece uma melhoria de desempenho de quatro vezes: o tempo entre a primeira e a quarta instruções é de  $3 \times 200$  ps ou 600 ps.

Podemos converter a discussão sobre ganho de velocidade com a técnica de pipelining em uma fórmula. Se os estágios forem perfeitamente balanceados, então o tempo entre as instruções no processador com pipeline — assumindo condições ideais — é igual a:

$$\text{Tempo entre instruções}_{\text{com pipeline}} = \frac{\text{Tempo entre instruções}_{\text{sem pipeline}}}{\text{Número de estágios do pipeline}}$$

Sob condições ideais e com uma grande quantidade de instruções, o ganho de velocidade com a técnica de pipelining é aproximadamente igual ao número de estágios do pipeline; um pipeline de cinco estágios é quase cinco vezes mais rápido.

A fórmula sugere que um pipeline de cinco estágios deve oferecer uma melhoria de quase cinco vezes sobre o tempo sem pipeline de 800 ps ou um ciclo de clock de 160 ps. Entretanto, o exemplo mostra que os estágios podem ser mal平衡ados. Além disso, a técnica de pipelining envolve algum overhead, cuja origem se tornará mais clara adiante. Assim, o tempo por instrução no processador com pipeline será superior ao mínimo possível, e o ganho de velocidade será menor que o número de estágios do pipeline.

Além do mais, até mesmo nossa afirmação de uma melhoria de quatro vezes para nosso exemplo não está refletida no tempo de execução total para as três instruções: são 1.400 ps versus 2.400 ps. Naturalmente, isso acontece porque o número de instruções não é grande. O que aconteceria se aumentássemos o número de instruções? Poderíamos estender os valores anteriores para 1.000.003 instruções. Acrescentaríamos 1.000.000 instruções no exemplo com pipeline; cada instrução acrescenta 200 ps ao tempo de execução total. O tempo de execução total seria  $1.000.000 \times 200$  ps + 1.400 ps, ou 200.001.400 ps. No exemplo sem pipeline, acrescentaríamos 1.000.000 instruções, cada uma exigindo 800 ps de modo que o tempo de execução total seria  $1.000.000 \times 800$  ps + 2.400 ps ou 800.002.400 ps. Sob essas condições ideais, a razão entre os tempos de execução total para os programas reais nos processadores sem

pipeline e com pipeline é próximo da razão de tempos entre as instruções:

$$\frac{800.002.400 \text{ ps}}{200.001.400 \text{ ps}} \simeq \frac{800 \text{ ps}}{200 \text{ ps}} \simeq 4,00$$

A técnica de pipelining melhora o desempenho *aumentando a vazão de instruções, em vez de diminuir o tempo de execução de uma instrução individual*, mas a vazão de instruções é a medida importante, pois os programas reais executam bilhões de instruções.

## Projetando conjuntos de instruções para pipelining

Mesmo com essa explicação simples sobre pipelining, podemos entender melhor o projeto do conjunto de instruções MIPS, projetado para execução com pipeline.

Primeiro, todas as instruções MIPS têm o mesmo tamanho. Essa restrição torna muito mais fácil buscar instruções no primeiro estágio do pipeline e decodificá-las no segundo estágio. Em um conjunto de instruções como o x86, no qual as instruções variam de 1 byte a 15 bytes, a técnica de pipelining é muito mais desafiadora. As implementações recentes da arquitetura x86 na realidade traduzem instruções x86 em micro-operações simples, que se parecem com instruções MIPS e depois usam um pipeline de micro-operações no lugar das instruções x86 nativas! ([Seção 4.10.](#))

Em segundo lugar, o MIPS tem apenas alguns poucos formatos de instrução, com os campos do registrador de origem localizados no mesmo lugar em cada instrução. Essa simetria significa que o segundo estágio pode começar a ler o banco de registradores ao mesmo tempo em que o hardware está determinando que tipo de instrução foi lida. Se os formatos de instrução do MIPS não fossem simétricos, precisaríamos dividir o estágio 2, resultando em seis estágios de pipeline. Logo veremos a desvantagem dos pipelines mais longos.

Em terceiro lugar, os operandos em memória só aparecem em loads ou stores no MIPS. Essa restrição significa que podemos usar o estágio de execução para calcular o endereço de memória e depois acessar a memória no estágio seguinte. Se pudéssemos operar sobre os operandos na memória, como na arquitetura x86, os estágios 3 e 4 se expandiriam para estágio de endereço, estágio de memória e,

em seguida, estágio de execução.

Em quarto lugar, conforme discutimos no [Capítulo 2](#), os operandos precisam estar alinhados na memória. Logo, não precisamos nos preocupar com uma única instrução de transferência de dados exigindo dois acessos à memória de dados; os dados solicitados podem ser transferidos entre o processador e a memória em um único estágio do pipeline.

## Hazards de pipeline

Existem situações em pipelining em que a próxima instrução não pode ser executada no ciclo de clock seguinte. Esses eventos são chamados *hazards* e existem três tipos diferentes.

### Hazards estruturais

O primeiro hazard é chamado **hazard estrutural**. Ele significa que o hardware não pode admitir a combinação de instruções que queremos executar no mesmo ciclo de clock. Um hazard estrutural na lavanderia aconteceria se usássemos uma combinação lavadora-secadora no lugar de lavadora e secadora separadas ou se nosso colega estivesse ocupado com outra coisa e não pudesse guardar as roupas. Nosso pipeline, cuidadosamente programado, fracassaria.

#### **hazard estrutural**

Uma ocorrência em que uma instrução planejada não pode ser executada no ciclo de clock apropriado, pois o hardware não admite a combinação de instruções definidas para executar em determinado ciclo de clock.

Como dissemos, o conjunto de instruções MIPS foi projetado para ser executado em um pipeline, tornando muito fácil para os projetistas evitar hazards estruturais quando projetaram o pipeline. Contudo, suponha que tivéssemos uma única memória, em vez de duas. Se o pipeline da [Figura 4.27](#) tivesse uma quarta instrução, veríamos que, no mesmo ciclo de clock em que a primeira instrução está acessando dados da memória, a quarta instrução está buscando uma instrução dessa mesma memória. Sem duas memórias, nosso pipeline poderia ter um hazard estrutural.

### Hazards de dados

Os **hazards de dados** ocorrem quando o pipeline precisa ser interrompido

porque uma etapa precisa esperar até que outra seja concluída. Suponha que você tenha encontrado uma meia na mesa de passar para a qual não exista um par. Uma estratégia possível é correr até o seu quarto e procurar em sua gaveta para ver se consegue encontrar o par. Obviamente, enquanto você está procurando, as roupas que ficaram secas e estão prontas para serem passadas, e aquelas que acabaram de ser lavadas e estão prontas para secarem deverão esperar.

## hazard de dados

Também chamado **hazard de dados do pipeline**. Uma ocorrência em que uma instrução planejada não pode ser executada no ciclo de clock correto porque os dados necessários para executar a instrução ainda não estão disponíveis.

Em um pipeline de computador, os hazards de dados surgem quando uma instrução depende de uma anterior, que ainda está no pipeline (um relacionamento que não existe realmente quando se lavam roupas). Por exemplo, suponha que tenhamos uma instrução add seguida imediatamente por uma instrução subtract que usa a soma (\$s0):

```
add    $s0, $t0, $t1  
sub    $t2, $s0, $t3
```

Sem intervenção, um hazard de dados poderia prejudicar o pipeline severamente. A instrução add não escreve seu resultado até o quinto estágio, significando que teríamos de desperdiçar três ciclos de clock no pipeline.

Embora pudéssemos contar com compiladores para remover todos esses hazards, os resultados não seriam satisfatórios. Essas dependências acontecem com muita frequência, e o atraso simplesmente é muito longo para se esperar que o compilador nos tire desse dilema.

A solução principal é baseada na observação de que não precisamos esperar que a instrução termine antes de tentar resolver o hazard de dados. Para a sequência de código anterior, assim que a ALU cria a soma para o add, podemos

fornecê-la como uma entrada para a subtração. O acréscimo de hardware extra para ter o item que falta antes do previsto, diretamente dos recursos internos, é chamado de **forwarding** ou **bypassing**.

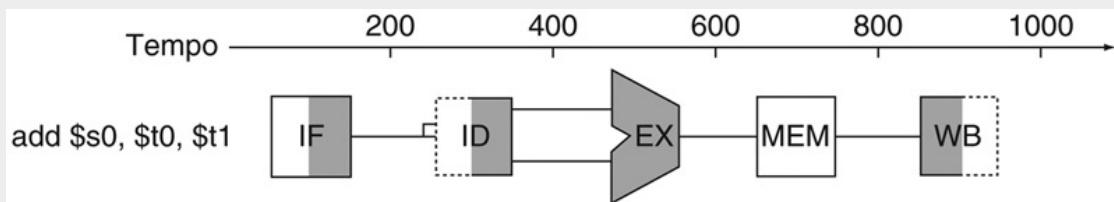
## forwarding

Também chamado **bypassing**. Um método para resolver um hazard de dados utilizando o elemento de dados que falta a partir de buffers internos, em vez de esperar que chegue nos registradores visíveis ao programador ou na memória.

## Forwarding com duas instruções

### Exemplo

Para as duas instruções anteriores, mostre quais estágios do pipeline estariam conectados pelo forwarding. Use o desenho da Figura 4.28 para representar o caminho de dados durante os cinco estágios do pipeline. Alinhe a cópia do caminho de dados para cada instrução, semelhante ao pipeline da lavanderia, na Figura 4.25.



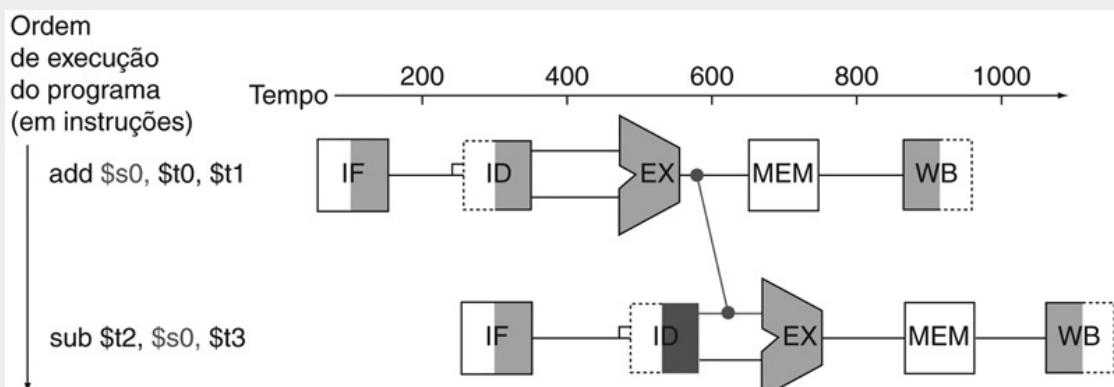
**FIGURA 4.28** Representação gráfica do pipeline de instrução, semelhante em essência ao pipeline da lavanderia, na Figura 4.25.

Aqui, usamos símbolos representando os recursos físicos com as abreviações para estágios de pipeline usados no decorrer do capítulo. Os símbolos para os cinco estágios: *IF* para o estágio de busca de instrução, com a caixa representando a memória de instrução; *ID* para o estágio de leitura de decodificação de instrução/banco de registradores, com o desenho mostrando o banco de registradores sendo lido; *EX* para o estágio de execução, com o desenho representando a ALU; *MEM* para o estágio de acesso à memória, com a caixa representando a memória de dados; e *WB* para o estágio

write-back, com o desenho mostrando o banco de registradores sendo escrito. O sombreamento indica que o elemento é usado pela instrução. Logo, MEM tem um fundo branco porque add não acessa a memória de dados. O sombreamento na metade direita do banco de registradores ou memória significa que o elemento é lido nesse estágio, e o sombreamento da metade esquerda significa que ele é escrito nesse estágio. Logo, a metade direita do ID é sombreada no segundo estágio porque o banco de registradores é lido, e a metade esquerda do WB é sombreada no quinto estágio, pois o banco de registradores é escrito.

## Resposta

A Figura 4.29 mostra a conexão para o forwarding do valor em \$s0 após o estágio de execução da instrução add como entrada para o estágio de execução da instrução sub.



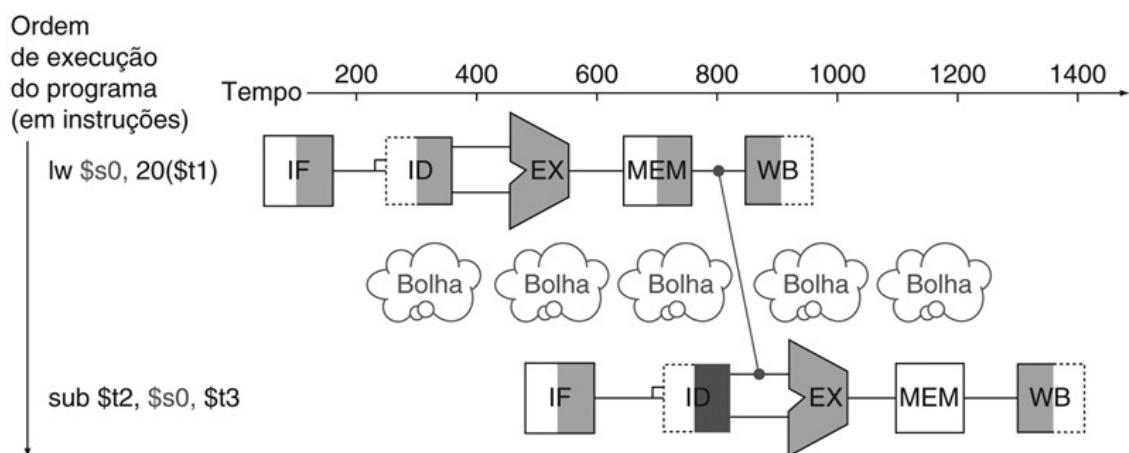
**FIGURA 4.29 Representação gráfica do forwarding.**

A conexão mostra o caminho do forwarding desde a saída do estágio EX de add até a entrada do estágio EX para sub, substituindo o valor do registrador \$s0 lido no segundo estágio de sub.

Nessa representação gráfica dos eventos, os caminhos de forwarding só são válidos se o estágio de destino estiver mais adiante no tempo do que o estágio de origem. Por exemplo, não pode haver um caminho de forwarding válido da saída do estágio de acesso à memória na primeira instrução para a entrada do estágio

de execução da instrução seguinte, pois isso significaria voltar no tempo.

O forwarding funciona muito bem e é descrito com detalhes na [Seção 4.7](#). Entretanto, ele não pode impedir todos os stalls do pipeline. Por exemplo, suponha que a primeira instrução fosse um load de  $\$s0$  em vez de um add. Como podemos imaginar examinando a [Figura 4.29](#), os dados desejados só estariam disponíveis *depois* do quarto estágio da primeira instrução na dependência, que é muito tarde para a *entrada* do terceiro estágio de sub. Logo, até mesmo com o forwarding, teríamos de atrasar um estágio para um **hazard de dados no uso de load**, como mostra a [Figura 4.30](#). Essa figura mostra um conceito importante de pipeline, conhecido oficialmente como **pipeline stall**, mas normalmente recebendo o apelido de **bolha**. Veremos os stalls em outros lugares do pipeline. A [Seção 4.7](#) mostra como podemos tratar de casos assim, usando a detecção de hardware e stalls ou software que reordena o código para evitar stalls de pipeline no uso de load, como este exemplo ilustra.



**FIGURA 4.30** Precisamos de um stall até mesmo com forwarding, quando uma instrução do formato R após um load tenta usar os dados.

Sem o stall, o caminho da saída do estágio de acesso à memória para a entrada do estágio de execução estaria ao contrário no tempo, o que é impossível. Essa figura, na realidade, é uma simplificação, pois não podemos saber, antes que a instrução de subtração seja lida e decodificada, se um stall será necessário. A [Seção 4.7](#) mostra os detalhes do que realmente acontece no caso de um hazard.

## **hazard de dados no uso de load**

Uma forma específica de hazard de dados em que os dados solicitados por uma instrução load ainda não estão disponíveis quando necessários para outra instrução.

## **pipeline stall**

Também chamado **bolha**. Um stall iniciado a fim de resolver um hazard.

## **Reordenando o código para evitar pipeline stalls**

### **Exemplo**

Considere o seguinte segmento de código em C:

$$\begin{aligned} a &= b + e; \\ c &= b + f; \end{aligned}$$

Aqui está o código MIPS gerado para esse segmento, supondo que todas as variáveis estejam na memória e sejam endereçáveis como offsets, a partir de \$t0:

```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
add    $t3, $t1,$t2
sw      $t3, 12($t0)
lw      $t4, 8($t0)
add    $t5, $t1,$t4
sw      $t5, 16($t0)
```

Encontre os hazards no segmento de código a seguir e reordene as instruções para evitar quaisquer pipeline stalls.

## Resposta

As duas instruções add possuem um hazard, devido à respectiva dependência da instrução lw imediatamente anterior. Observe que o bypassing elimina vários outros hazards em potencial, incluindo a dependência do primeiro add no primeiro lw e quaisquer hazards para instruções store. Subir a terceira instrução lw elimina os dois hazards:

```
lw    $t1, 0($t0)
lw    $t2, 4($t0)
lw    $t4, 8($t0)
add  $t3, $t1,$t2
sw    $t3, 12($t0)
add  $t5, $t1,$t4
sw    $t5, 16($t0)
```

Em um processador com pipeline com forwarding, a sequência reordenada será completada em dois ciclos a menos do que a versão original.

O forwarding leva a outro detalhe da arquitetura MIPS, além dos quatro mencionados anteriormente. Cada instrução MIPS escreve no máximo um resultado e faz isso no último estágio do pipeline. O forwarding é mais difícil se houver vários resultados para encaminhar por instrução, ou se for preciso, escrever um resultado mais cedo na execução da instrução.

## Detalhamento

o nome “forwarding” vem da ideia de que o resultado é passado adiante (em inglês *forwarded*) a partir de uma instrução anterior para uma instrução posterior. “Bypassing” vem de contornar (do inglês *bypass*) o resultado pelo banco de registradores até chegar à unidade desejada.

## Hazards de controle

O terceiro tipo de hazard é chamado **hazard de controle**, vindo da necessidade de tomar uma decisão com base nos resultados de uma instrução enquanto outras estão sendo executadas.

## **hazard de controle**

**Também chamado hazard de desvio** Quando a instrução apropriada não pode ser executada no devido ciclo de clock de pipeline porque a instrução buscada não é aquela necessária; ou seja, o fluxo de endereços de instrução não é o que o pipeline esperava.

Suponha que nosso pessoal da lavanderia receba a tarefa feliz de limpar os uniformes de um time de futebol. Como a roupa é muito suja, temos de determinar se o detergente e a temperatura da água que selecionamos são fortes o suficiente para limpar os uniformes, mas não tão forte para desgastá-los antes do tempo. Em nosso pipeline de lavanderia, temos de esperar até o segundo estágio e examinar o uniforme seco para ver se precisamos ou não mudar as opções da lavadora. O que fazer?

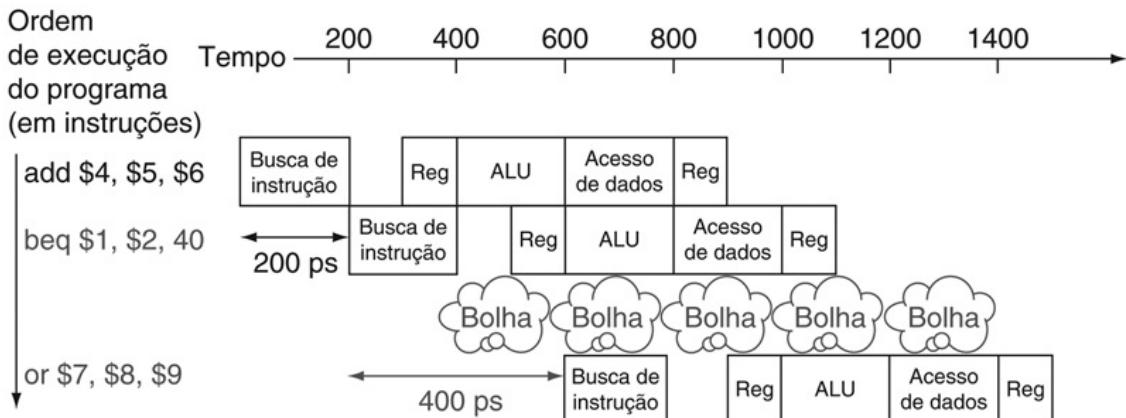
Aqui está a primeira das duas soluções para controlar os hazards na lavanderia e seu equivalente nos computadores.

**Stall:** Basta operar sequencialmente até que o primeiro lote esteja seco e depois repetir até você ter a fórmula correta.

Essa opção conservadora certamente funciona, mas é lenta.

A tarefa de decisão equivalente em um computador é a instrução de desvio. Observe que temos de começar a buscar a instrução após o desvio no próximo ciclo de clock. Contudo, o pipeline possivelmente não saberá qual deve ser a próxima instrução, pois ele *acabou de receber* da memória a instrução de desvio! Assim como na lavanderia, uma solução possível é ocasionar um stall no pipeline imediatamente após buscarmos um desvio, esperando até que o pipeline determine o resultado do desvio para saber em qual endereço apanhar a próxima instrução.

Vamos supor que colocamos hardware extra suficiente para que possamos testar registradores, calcular o endereço de desvio e atualizar o PC durante o segundo estágio do pipeline ([Seção 4.8](#)). Até mesmo com esse hardware extra, o pipeline envolvendo desvios condicionais se pareceria com a [Figura 4.31](#). A instrução `lw`, executada se o desvio não for tomado, fica em stall durante um ciclo de clock extra de 200 ps antes de iniciar



**FIGURA 4.31** Pipeline mostrando o stall em cada desvio condicional como solução para controlar os hazards.

Este exemplo considera que o desvio condicional é tomado e a instrução no destino do desvio é a instrução OR. Existe um stall de um estágio no pipeline ou bolha, após o desvio. Na realidade, o processo de criação de um stall é ligeiramente mais complicado, conforme veremos na [Seção 4.8](#). No entanto, o efeito sobre o desempenho é o mesmo que ocorreria se uma bolha fosse inserida.

## Desempenho do “stall no desvio”

### Exemplo

Estime o impacto nos *ciclos de clock por instrução* (CPI) do stall nos desvios. Suponha que todas as outras instruções tenham um CPI de 1.

### Resposta

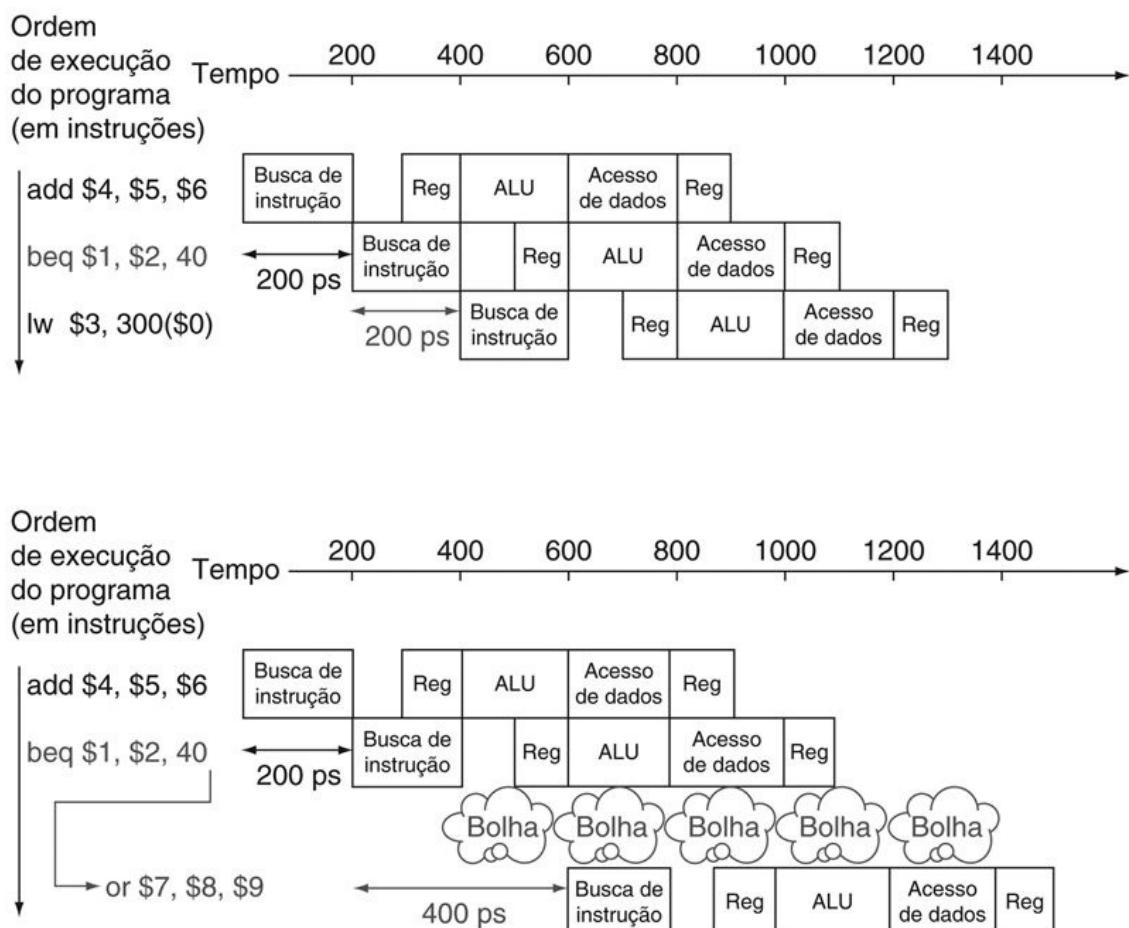
A Figura 3.27 no Capítulo 3 mostra que os desvios são 17% das instruções executadas no SPECint2006. Como as outras instruções possuem um CPI de 1 e os desvios tomaram um ciclo de clock extra para o stall, então veríamos um CPI de 1,17 e, portanto, um stall de 1,17 em relação ao caso ideal.

Se não pudermos resolver o desvio no segundo estágio, como normalmente acontece para pipelines maiores, então veríamos um atraso ainda maior se ocorresse um stall nos desvios. O custo dessa opção é muito alto para a maioria dos computadores utilizar, e isso motiva uma segunda solução para o hazard de controle, usando uma de nossas grandes ideias do [Capítulo 1](#):

*Prever:* se você estiver certo de que tem a fórmula correta para lavar os uniformes, então basta *prever* que ela funcionará e lavar a segunda remessa enquanto espera que a primeira seque.

Essa opção não atrasa o pipeline quando você estiver correto. Entretanto, quando estiver errado, você terá de refazer a remessa que foi lavada enquanto pensa na decisão.

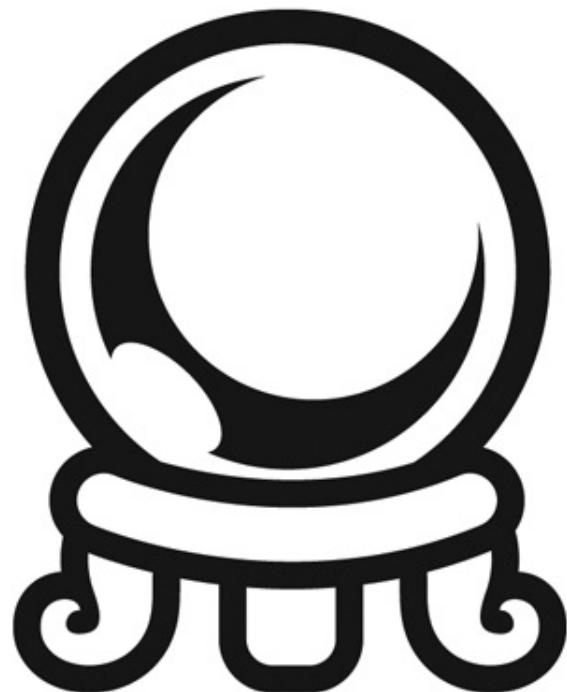
Os computadores realmente utilizam a **predição** para tratar dos desvios. Uma técnica simples é sempre prever que os desvios não serão tomados. Quando você estiver certo, o pipeline prosseguirá a toda velocidade. Somente quando os desvios são tomados é que o pipeline sofre um stall. A [Figura 4.32](#) mostra um exemplo assim.



**FIGURA 4.32** Prevendo que os desvios não serão tomados como solução para o hazard de controle.

O desenho superior mostra o pipeline quando o desvio não é tomado. O desenho inferior mostra o pipeline quando o desvio é tomado. Conforme observamos na [Figura 4.31](#), a inserção de uma bolha nesse padrão simplifica o que realmente acontece, pelo menos durante o primeiro ciclo de clock, imediatamente após o desvio. A [Seção 4.8](#) esclarecerá os detalhes.

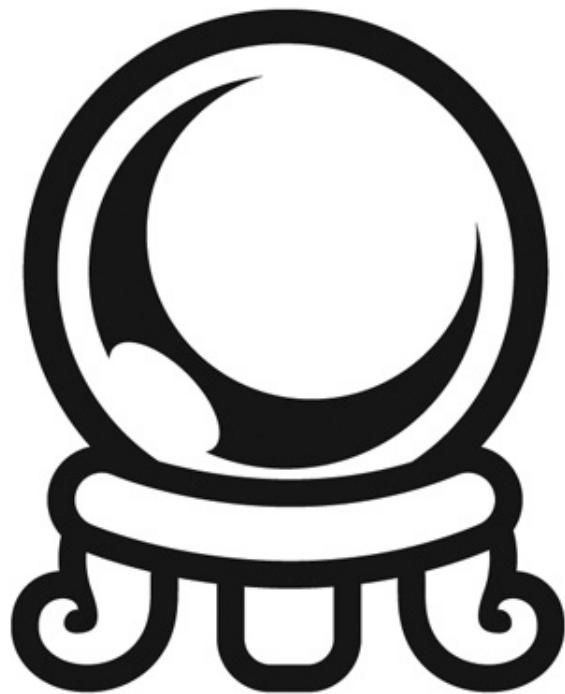
Uma versão mais sofisticada de **previsão de desvio** teria alguns desvios previstos como tomados e alguns como não tomados. Em nossa analogia, os uniformes escuros ou de casa poderiam usar uma fórmula, enquanto os uniformes claros ou de sair poderiam usar outra. No caso da programação, no final dos loops existem desvios que voltam para o início do loop. Como provavelmente serão tomados e desviam para trás, sempre poderíamos prever como tomados os desvios para um endereço anterior.



## P R E D I Ç Ã O

**previsão de desvio**

Um método de resolver um hazard de desvio que considera um determinado resultado para o desvio e prossegue a partir dessa suposição, em vez de esperar para verificar o resultado real.



## P R E D I Ç Ã O

Essas técnicas rígidas para o desvio contam com o comportamento estereotipado e não levam em consideração a individualidade de uma instrução de desvio específica. Previsores de hardware *dinâmicos*, ao contrário, fazem suas escolhas dependendo do comportamento de cada desvio, e podem mudar as previsões para um desvio durante a vida de um programa. Seguindo nossa analogia, na previsão dinâmica, uma pessoa veria como o uniforme estava sujo e escolheria a fórmula, ajustando a próxima escolha dependendo do sucesso das escolhas recentes.

Uma técnica comum para a previsão dinâmica de desvios é manter um histórico de cada desvio como tomado ou não tomado, e depois usar o comportamento passado recente para prever o futuro. Como veremos mais adiante, a quantidade e o tipo de histórico mantido têm se tornado extensos,

resultando em previsores de desvio dinâmicos que podem prever os desvios corretamente, com uma precisão superior a 90% ([Seção 4.8](#)). Quando a escolha estiver errada, o controle do pipeline terá de garantir que as instruções após o desvio errado não tenham efeito, reiniciando o pipeline a partir do endereço de desvio apropriado. Em nossa analogia de lavanderia, temos de deixar de aceitar novas remessas para poder reiniciar a remessa prevista incorretamente.

Como no caso de todas as outras soluções para controlar hazards, pipelines mais longos aumentam o problema, neste caso, aumentando o custo do erro de previsão. As soluções para controlar os hazards são descritas com mais detalhes na [Seção 4.8](#).

## Detalhamento

Existe uma terceira técnica para o hazard de controle, chamada *decisão adiada*. Em nossa analogia, sempre que você tiver de tomar uma decisão sobre a lavanderia, basta colocar uma remessa de roupas que não sejam de futebol na lavadora, enquanto espera que os uniformes de futebol sequem. Desde que você tenha roupas sujas suficientes, que não sejam afetadas pelo teste, essa solução funcionará bem.

Chamado de *delayed branch* (desvio adiado) nos computadores, essa é a solução realmente usada pela arquitetura MIPS. O delayed branch sempre executa a próxima instrução sequencial, com o desvio ocorrendo *após* esse atraso de uma instrução. Isso fica escondido do programador assembly do MIPS, pois o montador pode arrumar as instruções automaticamente para conseguir o comportamento de desvio desejado pelo programador. O software MIPS colocará uma instrução imediatamente após a instrução de delayed branch, que não é afetada pelo desvio, e um desvio tomado muda o endereço da instrução que vem *após* essa instrução segura. Em nosso exemplo, a instrução add antes do desvio na Figura 4.31 não o afeta, e pode ser movida para depois dele, a fim de esconder totalmente seu atraso. Como os delayed branches são úteis quando os desvios são curtos, nenhum processador usa um delayed branch de mais de um ciclo. Para atrasos em desvios maiores, a previsão de desvio baseada em hardware normalmente é usada.

## Resumo da visão geral de pipelining

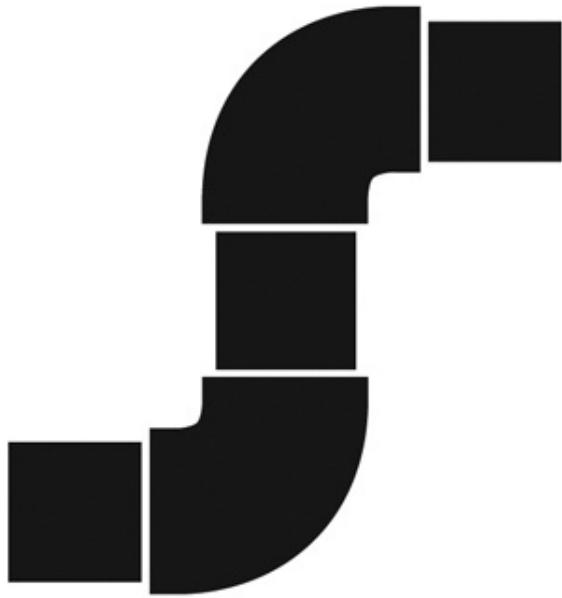
Pipelining é uma técnica que explora o **paralelismo** entre as instruções em um

fluxo de instruções sequenciais. Ela tem a vantagem substancial de que, diferente de programar um multiprocessador, ela é fundamentalmente invisível ao programador.



## PARALELISMO

Nas próximas seções deste capítulo, abordamos o conceito de pipelining usando o subconjunto de instruções MIPS da implementação de ciclo único na [Seção 4.4](#) e mostramos uma versão simplificada de seu pipeline. Depois, examinamos os problemas que a técnica de **pipelining** gera e o desempenho alcançável em situações típicas.



## PIPELINING

Se você quiser saber mais sobre o software e as implicações de desempenho da técnica de pipelining, agora terá base suficiente para pular para a [Seção 4.10](#). A [Seção 4.10](#) apresenta conceitos avançados de pipelining, como o escalonamento superescalar e dinâmico, e a [Seção 4.11](#) examina os pipelines de microprocessadores recentes.

Como alternativa, se você estiver interessado em entender como a técnica de pipelining é implementada e os desafios de lidar com hazards, poderá prosseguir para examinar o projeto de um caminho de dados com pipeline, explicado na [Seção 4.6](#). Depois, você poderá usar esse conhecimento para explorar a implementação do forwarding e stalls na [Seção 4.7](#). Você poderá, então, ler a [Seção 4.8](#) e aprender mais sobre soluções para hazards de desvio, e depois ver como as exceções são tratadas, na [Seção 4.9](#).

### Verifique você mesmo

Para cada sequência de código a seguir, indique se ela deverá sofrer stall, pode evitar stalls usando apenas forwarding ou pode ser executada sem stall ou forwarding:

Sequência 1	Sequência 2	Sequência 3
lw \$t0 \$t0		

add \$t1,\$t0,\$t0	add \$t1,\$t0,\$t0 addi \$t2,\$t0,#5 addi \$t4,\$t1,#5	addi \$t1,\$t0,#1 addi \$t2,\$t0,#2 addi \$t3,\$t0,#2 addi \$t3,\$t0,#4 addi \$t5,\$t0,#5
--------------------	--	---

## Entendendo o desempenho dos programas

Fora do sistema de memória, a operação eficaz do pipeline normalmente é o fator mais importante para determinar o CPI do processador e, portanto, seu desempenho. Conforme veremos na Seção 4.10, compreender o desempenho de um processador moderno com múltiplos problemas é algo complexo e exige a compreensão de mais do que apenas as questões que surgem em um processador com pipeline simples. Apesar disso, os hazards estruturais, de dados e de controle continuam sendo importantes em pipelines simples e mais sofisticados.

Para pipelines modernos, os hazards estruturais costumam girar em torno da unidade de ponto flutuante, que pode não ser totalmente implementada com pipeline, enquanto os hazards de controle costumam ser um problema maior nos programas de inteiros, que costumam ter maiores frequências de desvio, além de desvios menos previsíveis. Os hazards de dados podem ser gargalos de desempenho em programas de inteiros e de ponto flutuante. Em geral, é mais fácil lidar com hazards de dados em programas de ponto flutuante porque a menor frequência de desvios e os padrões de acesso mais regulares permitem que o compilador tente escalarizar instruções para evitar os hazards. É mais difícil realizar essas otimizações em programas de inteiros, que possuem acesso menos regular e envolvem um maior uso de ponteiros. Conforme veremos na Seção 4.10, existem técnicas de compilação e de hardware mais ambiciosas que reduzem as dependências de dados para o escalonamento.



## PIPELINING

### Colocando em perspectiva

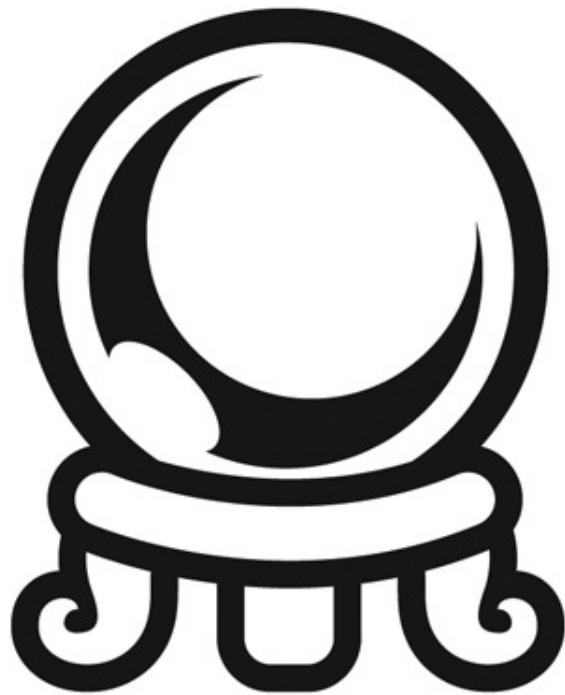
A técnica de pipelining aumenta o número de instruções em execução simultânea e a velocidade em que as instruções são iniciadas e concluídas. A técnica de pipelining não reduz o tempo gasto para completar uma instrução individual, também chamado de **latência**. Por exemplo, o pipeline de cinco estágios ainda usa cinco ciclos de clock para completar a instrução. Nos termos usados no Capítulo 1, a técnica de pipelining melhora a *vazão* de instruções, e não o *tempo de execução* ou *latência* das instruções individualmente.

### latência (pipeline)

O número de estágios em um pipeline ou o número de estágios entre duas instruções durante a execução.

Os conjuntos de instruções podem simplificar ou dificultar a vida dos projetistas do pipeline, que já precisam enfrentar hazards estruturais, de controle e de dados. A **previsão** de desvio, o forwarding e os stalls ajudam a tornar um

computador rápido enquanto ainda gera as respostas certas.



## P R E D I Ç Ã O

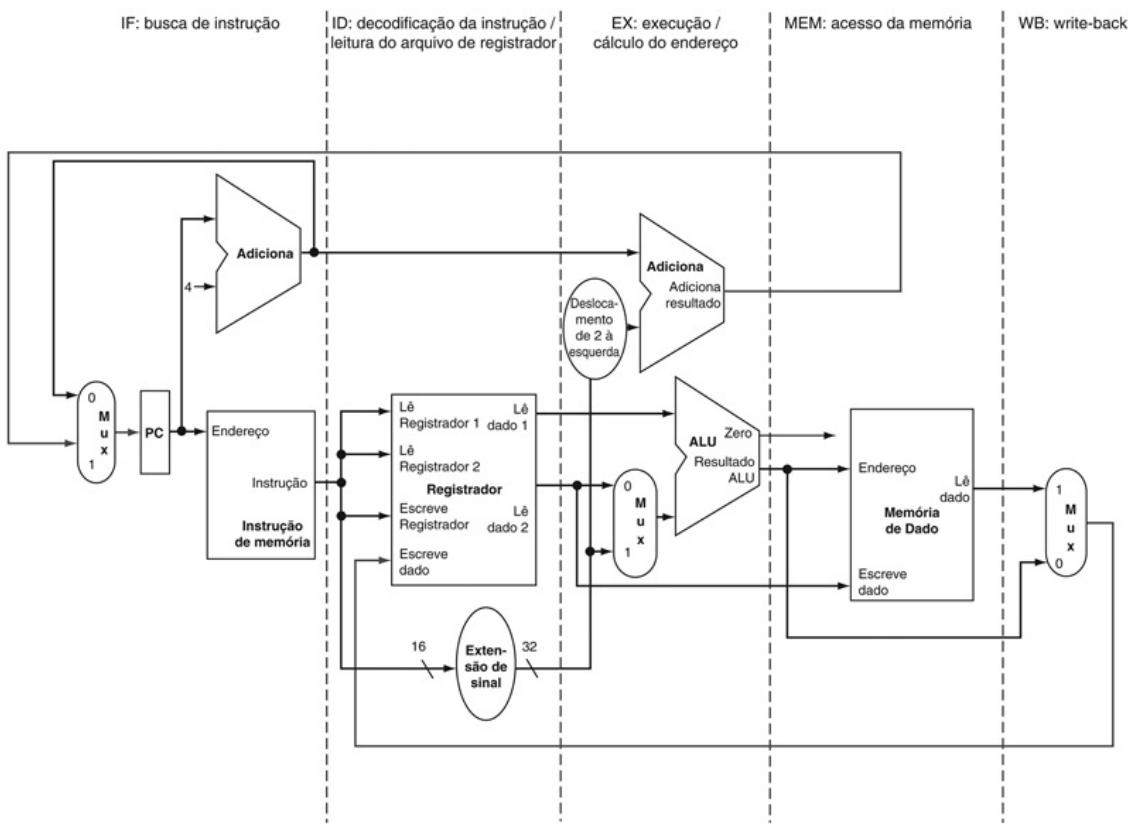
### 4.6. Caminho de dados e controle usando pipeline

*Há menos coisa nisso do que os olhos podem ver.*

*Tallulah Bankhead, comentário para Alexander Woollcott, 1922*

A Figura 4.33 mostra o caminho de dados de ciclo único da Seção 4.4 com os estágios de pipeline identificados. A divisão de uma instrução em cinco estágios significa um pipeline de cinco estágios que, por sua vez, significa que até cinco instruções estarão em execução durante qualquer ciclo de clock. Assim, temos de separar o caminho de dados em cinco partes, com cada parte possuindo um nome correspondente a um estágio da execução da instrução:

1. IF (Instruction Fetch): Busca de instruções.



**FIGURA 4.33** O caminho de dados da [Seção 4.4](#) (semelhante à [Figura 4.17](#)).

Cada etapa da instrução pode ser mapeada no caminho de dados da esquerda para a direita. As únicas exceções são a atualização do PC e a etapa de escrita do resultado, mostrada em destaque, que envia o resultado da ALU ou os dados da memória para a esquerda, a fim de serem escritos no banco de registradores. (Normalmente, usamos linhas coloridas para controle, mas são linhas de dados.)

2. ID (Instruction Decode): Decodificação de instruções e leitura do banco de registradores.
3. EX: Execução ou cálculo de endereço.
4. MEM: Acesso à memória de dados.
5. WB (Write Back): Escrita do resultado.

Na [Figura 4.33](#), esses cinco componentes correspondem aproximadamente ao modo como o caminho de dados é desenhado; as instruções e os dados em geral se movem da esquerda para a direita pelos cinco estágios enquanto completam a execução. Voltando à nossa analogia da lavanderia, as roupas ficam mais limpas, mais secas e mais organizadas à medida que prosseguem na fila, e nunca se

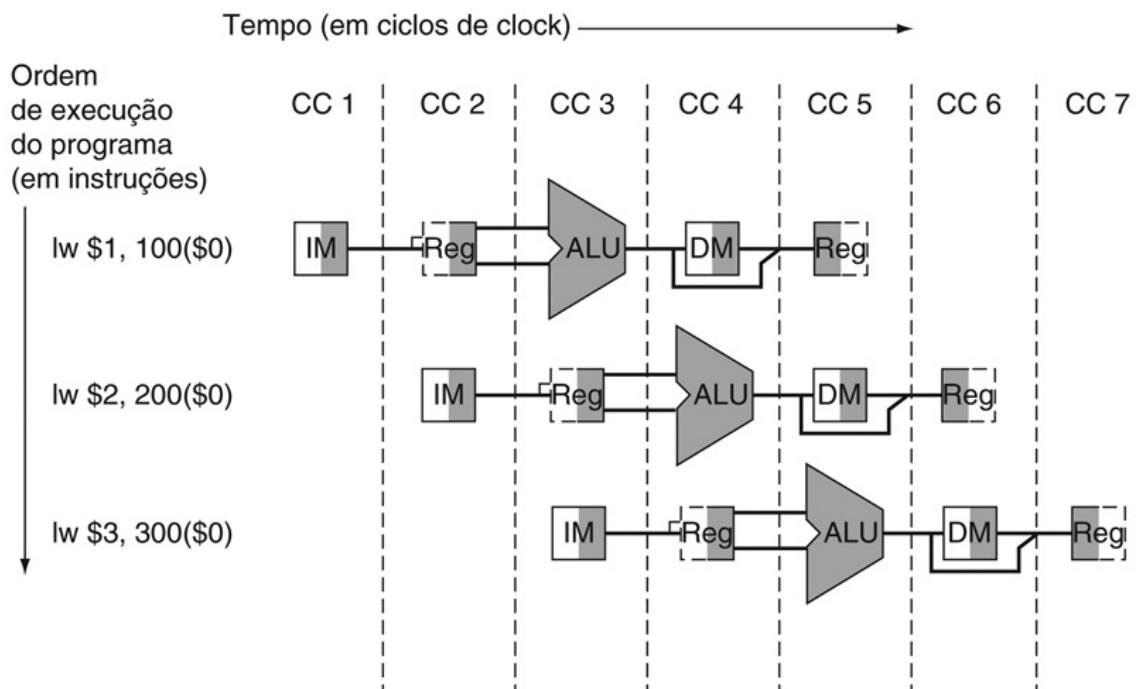
movem para trás.

Entretanto, existem duas exceções para esse fluxo de informações da esquerda para a direita:

- O estágio de escrita do resultado, que coloca o resultado de volta no banco de registradores, no meio do caminho de dados;
- A seleção do próximo valor do PC, escolhendo entre o PC incrementado e o endereço de desvio do estágio MEM.

Os dados fluindo da direita para a esquerda não afetam a instrução atual; somente as instruções seguintes no pipeline são influenciadas por esses movimentos de dados reversos. Observe que a primeira seta da direita para a esquerda pode levar a hazards de dados e a segunda ocasiona hazards de controle.

Uma maneira de mostrar o que acontece na execução com pipeline é fingir que cada instrução tem seu próprio caminho de dados e depois colocar esses caminhos de dados em uma linha de tempo para mostrar seu relacionamento. A Figura 4.34 mostra a execução das instruções na Figura 4.27, exibindo seus caminhos de dados privados em uma linha de tempo comum. Usamos uma versão estilizada do caminho de dados na Figura 4.33 para mostrar os relacionamentos na Figura 4.34.



**FIGURA 4.34 Instruções executadas usando o caminho de dados de ciclo único na Figura 4.33, assumindo a execução com pipeline.**

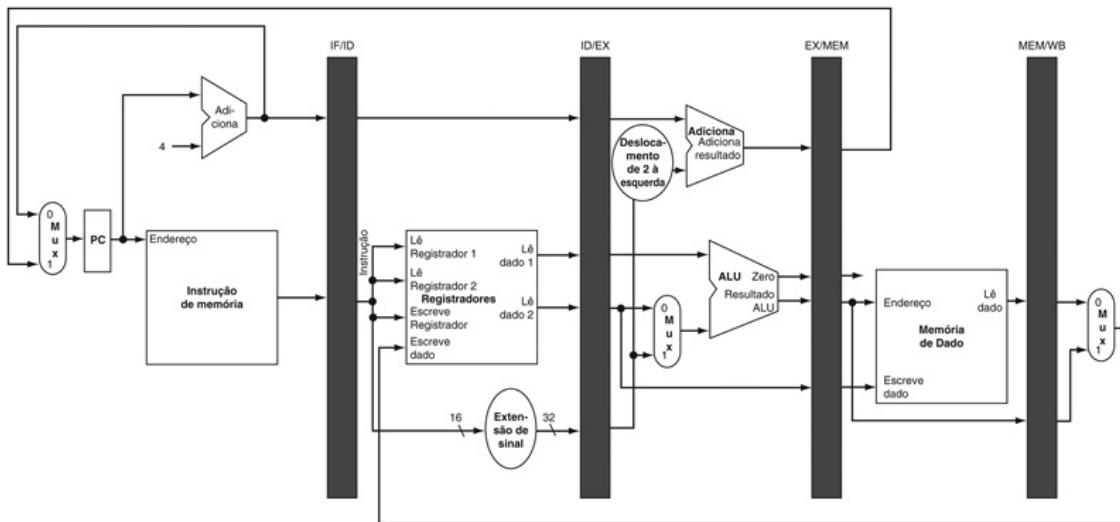
Semelhante às [Figuras de 4.28 a 4.30](#), esta figura finge que cada instrução possui seu próprio caminho de dados e pinta cada parte de acordo com o uso. Ao contrário daquelas figuras, cada estágio é rotulado pelo recurso físico usado nesse estágio, correspondendo às partes do caminho de dados na [Figura 4.33](#). *IM* representa a memória de instruções e o PC no estágio de busca da instrução, *Reg* significa banco de registradores e extensor de sinal no estágio de decodificação de instruções/leitura do banco de registradores (*ID*), e assim por diante. Para manter a ordem de tempo correta, esse caminho de dados estilizado divide o banco de registradores em duas partes lógicas: leitura de registradores durante a busca de registradores (*ID*) e registradores escritos durante a escrita do resultado (*WB*). Esse uso dual é representado pelo desenho da metade esquerda não sombreada do banco de registradores, usando linhas tracejadas no estágio *ID*, quando ele não estiver sendo escrito e a metade direita não sombreada usando linhas tracejadas do estágio *WB*, quando não estiver sendo lido. Como antes, consideramos que o banco de registradores é escrito na primeira metade do ciclo de clock e é lido durante a segunda metade.

A [Figura 4.34](#) parece sugerir que três instruções precisam de três caminhos de dados. Em vez disso, acrescentamos registradores para manter dados de modo que partes do caminho de dados pudessem ser compartilhadas durante a execução da instrução.

Por exemplo, como mostra a [Figura 4.34](#), a memória de instruções é usada durante apenas um dos cinco estágios de uma instrução, permitindo que seja compartilhada por outras instruções durante os outros quatro estágios. A fim de reter o valor de uma instrução individual para seus outros quatro estágios, o valor lido da memória de instruções precisa ser salvo em um registrador. Argumentos semelhantes se aplicam a cada estágio do pipeline, de modo que precisamos colocar registradores sempre que existam linhas divisórias entre os estágios na [Figura 4.33](#). Retornando à nossa analogia da lavanderia, poderíamos ter um cesto entre cada par de estágios contendo as roupas para a próxima etapa.

A [Figura 4.35](#) mostra o caminho de dados usando pipeline com os registradores do pipeline destacados. Todas as instruções avançam durante cada ciclo de clock de um registrador do pipeline para o seguinte. Os registradores recebem os nomes dos dois estágios separados por esse registrador. Por

exemplo, o registrador do pipeline entre os estágios IF e ID é chamado de IF/ID.



**FIGURA 4.35** A versão com pipeline do caminho de dados na [Figura 4.33](#).

Os registradores do pipeline, em cinza, separam cada estágio do pipeline. Eles são rotulados pelos nomes dos estágios que separam; por exemplo, o primeiro é rotulado com *IF/ID* porque separa os estágios de busca de instruções e decodificação de instruções. Os registradores precisam ser grandes o suficiente para armazenar todos os dados correspondentes às linhas que passam por eles. Por exemplo, o registrador IF/ID precisa ter 64 bits de largura, pois precisa manter a instrução de 32 bits lida da memória e o endereço incrementado de 32 bits no PC. Vamos expandir esses registradores no decorrer deste capítulo, mas, por enquanto, os outros três registradores de pipeline contêm 128, 97 e 64 bits, respectivamente.

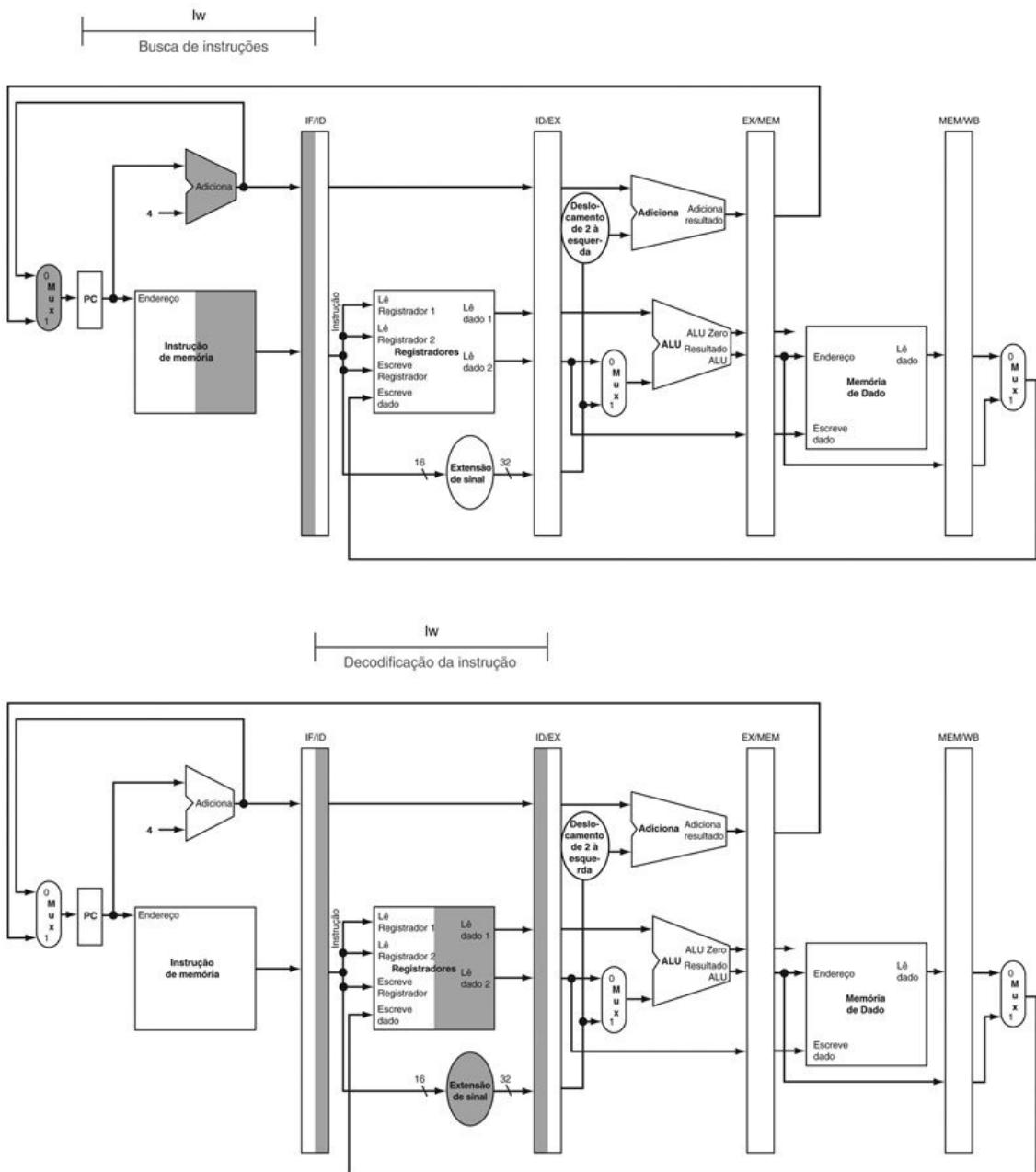
Observe que não existe um registrador de pipeline no final do estágio de escrita do resultado (WB). Todas as instruções precisam atualizar algum estado no processador — o banco de registradores, memória ou o PC —, assim, um registrador de pipeline separado é redundante para o estado que é atualizado. Por exemplo, uma instrução load colocará seu resultado em um dos 32 registradores, e qualquer instrução posterior que precise desses dados simplesmente lerá o registrador apropriado.

Naturalmente, cada instrução atualiza o PC, seja incrementando-o ou atribuindo a ele o endereço de destino de um desvio. O PC pode ser considerado

um registrador de pipeline: um que alimenta o estágio IF do pipeline. Contudo, diferente dos registradores de pipeline sombreados na [Figura 4.35](#), o PC faz parte do estado arquitetônico visível; seu conteúdo precisa ser salvo quando ocorre uma exceção, enquanto o conteúdo dos registradores de pipeline pode ser descartado. Na analogia da lavanderia, você poderia pensar no PC como correspondendo ao cesto que mantém a remessa de roupas sujas antes da etapa de lavagem.

Para mostrar como funciona a técnica de pipelining, no decorrer deste capítulo, apresentamos sequências de figuras para demonstrar a operação com o tempo. Essas páginas extras parecem exigir muito mais tempo para você entender. Mas não tema; as sequências levam muito menos tempo do que parece, pois você pode compará-las e ver que mudanças ocorrem em cada ciclo do clock. A [Seção 4.7](#) descreve o que acontece quando existem hazards de dados entre instruções em um pipeline; ignore-as por enquanto.

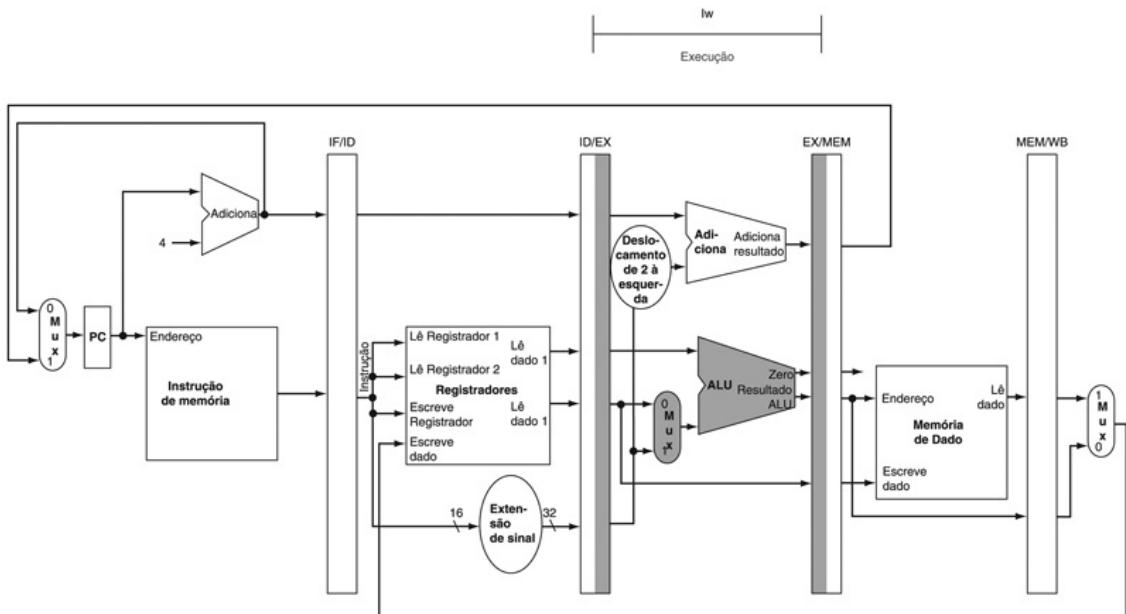
As [Figuras 4.36 a 4.38](#), nossa primeira sequência, mostram as partes ativas do caminho de dados destacadas, enquanto uma instrução de load passa pelos cinco estágios de execução do pipeline. Mostramos um load primeiro porque ele ativa todos os cinco estágios. Como nas [Figuras de 4.28 a 4.30](#), destacamos a *metade direita* dos registradores ou memória quando estão sendo *lidos* e destacamos a *metade esquerda* quando estão sendo *escritos*.



**FIGURA 4.36** IF e ID: primeiro e segundo estágios do pipe de uma instrução, com as partes ativas do caminho de dados da [Figura 4.35](#) em destaque.

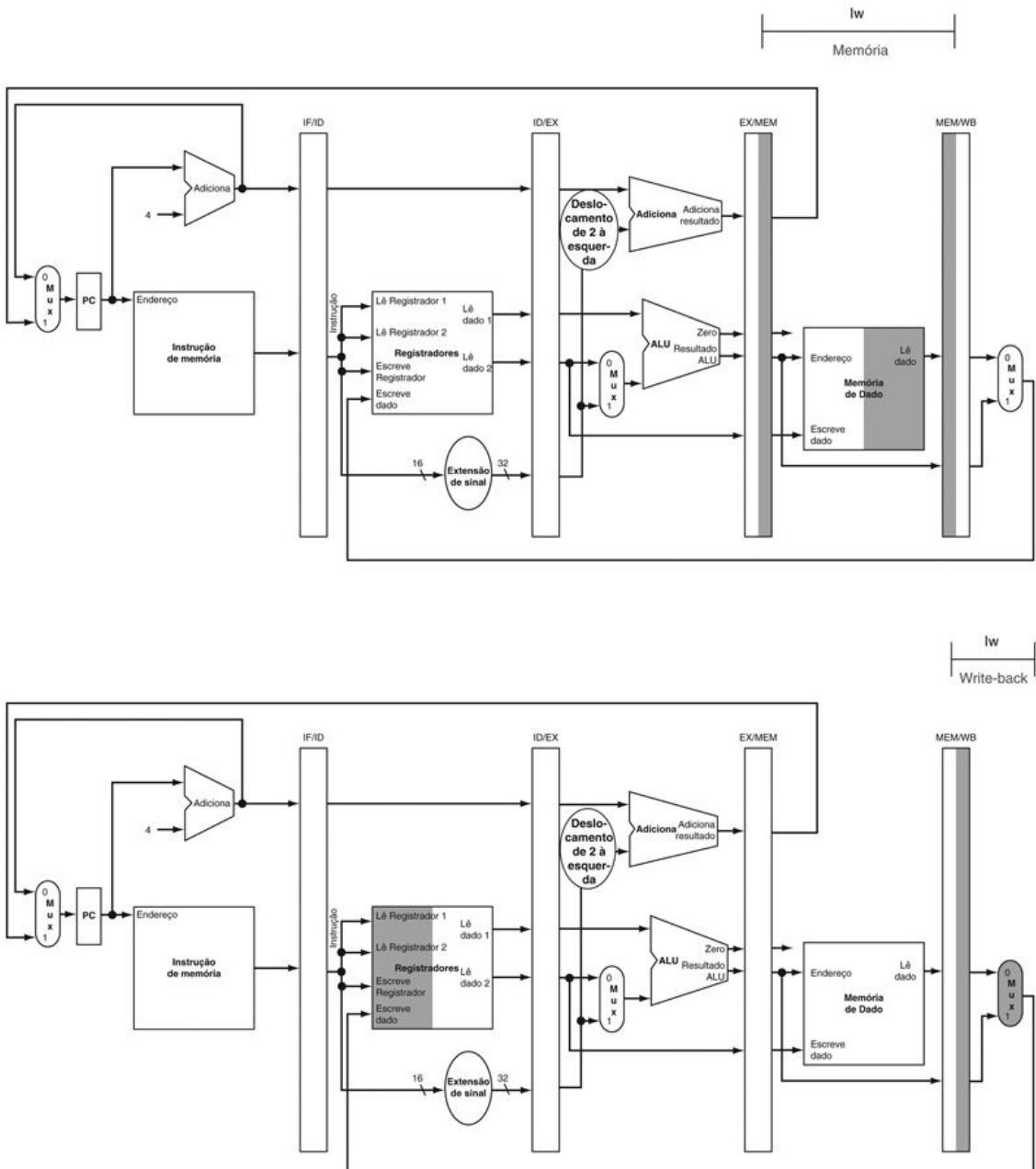
A convenção de destaque é a mesma utilizada na [Figura 4.28](#). Como na [Seção 4.2](#), não há confusão quando se lê e escreve nos registradores, pois o conteúdo só muda na transição do clock. Embora o load só precise do registrador de cima no estágio 2, o processador não sabe qual instrução está sendo decodificada, de modo que estende o sinal da constante de 16 bits e lê os dois registradores para o registrador de pipeline

ID/EX. Não precisamos de todos os três operandos, mas simplifica o controle manter todos os três.



**FIGURA 4.37** EX: o terceiro estágio do pipe de uma instrução load, destacando as partes do caminho de dados da [Figura 4.35](#) usadas neste estágio do pipe.

O registrador é acrescentado ao imediato com sinal estendido, e a soma é colocada no registrador de pipeline EX/MEM.



**FIGURA 4.38** MEM e WB: o quarto e quinto estágios do pipe de uma instrução load, destacando as partes do caminho de dados da [Figura 4.35](#) usadas nesses estágios do pipe.

A memória de dados é lida por meio do endereço no registrador de pipeline EX/MEM e os dados são colocados no registrador de pipeline MEM/WB. Em seguida, os dados são lidos do registrador de pipeline MEM/WB e escritos no banco de registradores, no meio do caminho de dados. Nota: existe um bug nesse projeto, que foi consertado na [Figura 4.41](#).

Mostramos a abreviação da instrução `lw` com o nome do estágio do pipeline que está ativo em cada figura. Os cinco estágios são os seguintes:

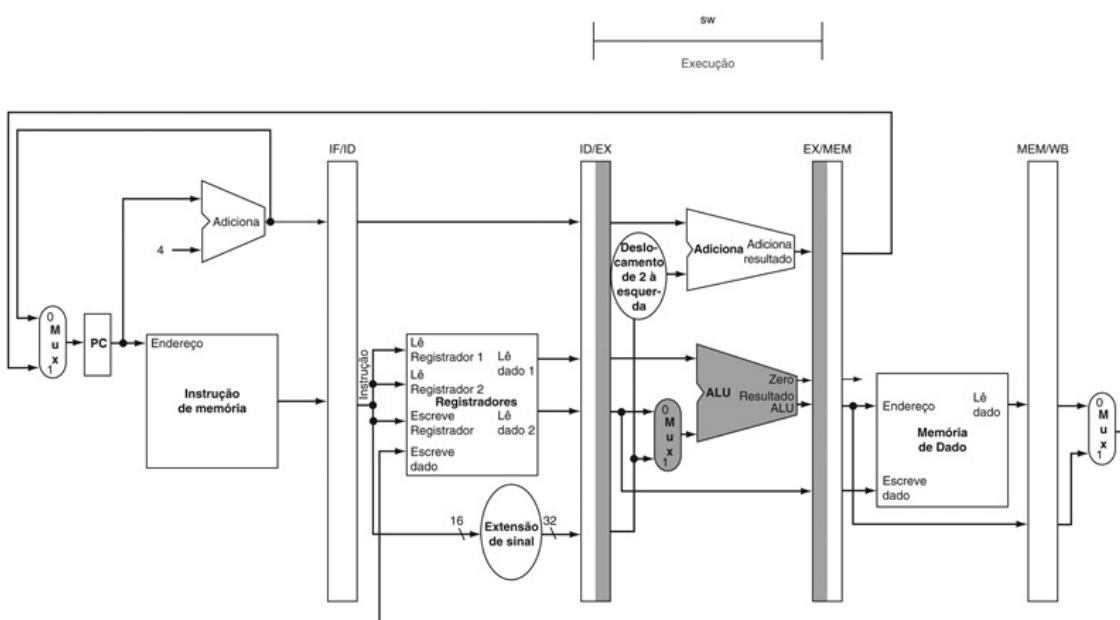
1. *Busca de instruções*: A parte superior da [Figura 4.36](#) mostra a instrução sendo lida da memória usando o endereço no PC e depois colocada no registrador de pipeline IF/ID. O endereço do PC é incrementado em 4 e, depois, escrito de volta ao PC, para que fique pronto para o próximo ciclo de clock. Esse endereço incrementado também é salvo no registrador de pipeline IF/ID caso seja necessário mais tarde para uma instrução, como `beq`. O computador não tem como saber que tipo de instrução está sendo buscada, de modo que precisa se preparar para qualquer instrução, passando pelo pipeline informações potencialmente necessárias.
2. *Decodificação de instruções e leitura do banco de registradores*: A parte inferior da [Figura 4.36](#) mostra a parte relativa à instrução do registrador de pipeline IF/ID, fornecendo o campo imediato de 16 bits, que tem seu sinal estendido para 32 bits, e os números dos dois registradores para leitura. Todos os três valores são armazenados no registrador de pipeline ID/EX, assim como o endereço no PC incrementado. Novamente, transferimos tudo o que possa ser necessário por qualquer instrução, durante um ciclo de clock posterior.
3. *Execução ou cálculo de endereço*: A [Figura 4.37](#) mostra que a instrução `load` lê o conteúdo do registrador 1 e o imediato com o sinal estendido do registrador de pipeline ID/EX e os soma usando a ALU. Essa soma é colocada no registrador de pipeline EX/MEM.
4. *Acesso à memória*: A parte superior da [Figura 4.38](#) mostra a instrução `load` lendo a memória de dados por meio do endereço vindo do registrador de pipeline EX/MEM e carregando os dados no registrador de pipeline MEM/WB.
5. *Escrita do resultado*: A parte inferior da [Figura 4.38](#) mostra a etapa final: lendo os dados do registrador de pipeline MEM/WB e escrevendo-os no banco de registradores, no meio da figura.

Essa revisão da instrução `load` mostra que qualquer informação necessária em um estágio posterior do pipe precisa ser passada a esse estágio por meio de um registrador de pipeline. A revisão de uma instrução `store` mostra a semelhança na execução da instrução, bem como a passagem da informação para os estágios posteriores. Aqui estão os cinco estágios do pipe da instrução `store`:

1. *Busca de instruções*: A instrução é lida da memória usando o endereço no PC e depois é colocada no registrador de pipeline IF/ID. Esse estágio

ocorre antes que a instrução seja identificada, de modo que a parte superior da [Figura 4.36](#) funciona para store e também para load.

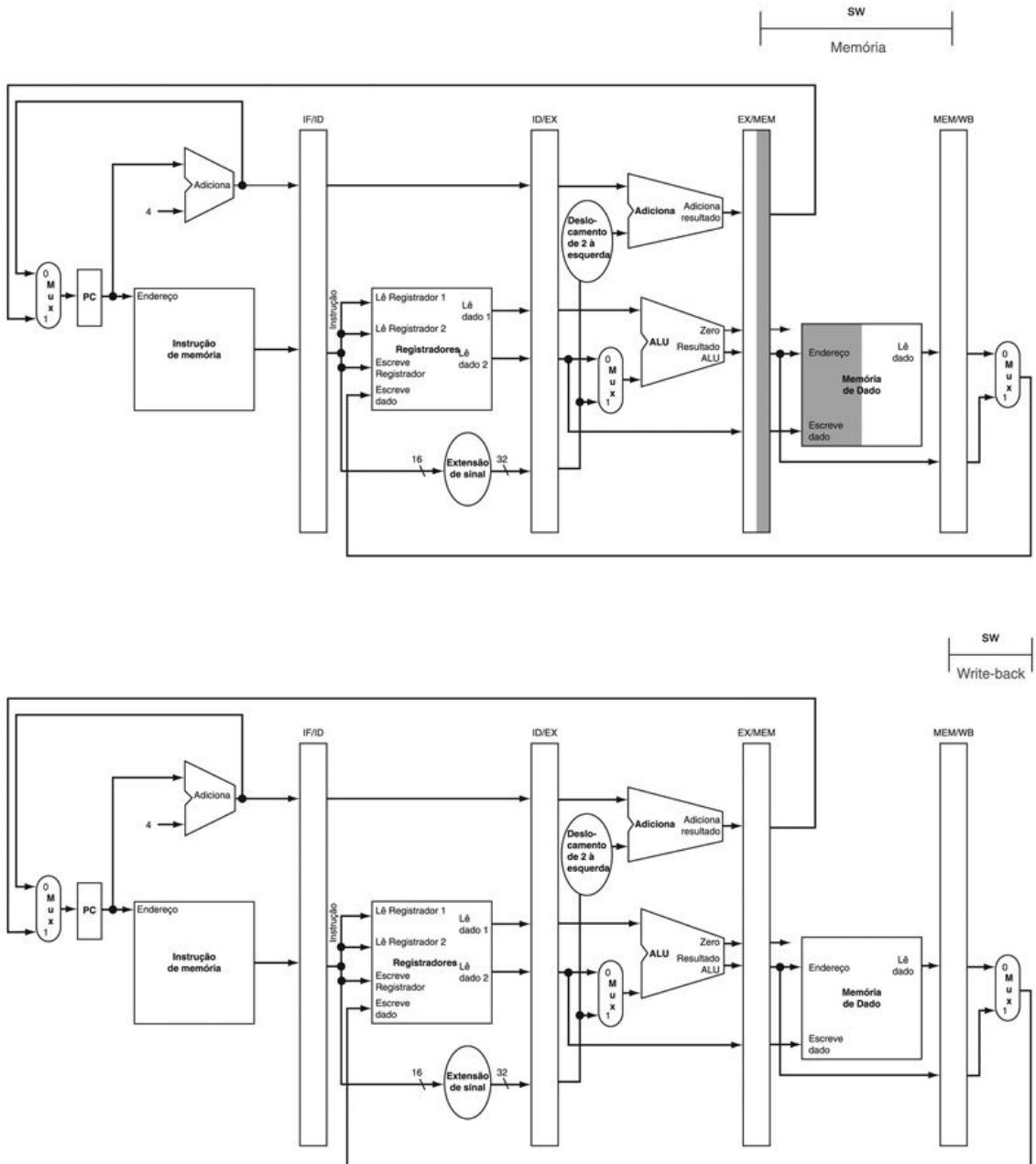
2. *Decodificação de instruções e leitura do banco de registradores:* A instrução no registrador de pipeline IF/ID fornece os números de dois registradores para leitura e estende o sinal do imediato de 16 bits. Esses três valores de 32 bits são armazenados no registrador de pipeline ID/EX. A parte inferior da [Figura 4.36](#) para instruções load também mostra as operações do segundo estágio para stores. Esses dois primeiros estágios são executados por todas as instruções, pois é muito cedo para saber o tipo da instrução.
3. *Execução e cálculo de endereço:* A [Figura 4.39](#) mostra a terceira etapa; o endereço efetivo é colocado no registrador de pipeline EX/MEM.



**FIGURA 4.39 EX: o terceiro estágio do pipe de uma instrução store.**

Ao contrário do terceiro estágio da instrução load na [Figura 4.37](#), o segundo valor do registrador é carregado no registrador de pipeline EX/MEM a ser usado no próximo estágio. Embora não faça mal algum sempre escrever esse segundo registrador no registrador de pipeline EX/MEM, escrevemos o segundo registrador apenas em uma instrução store para tornar o pipeline mais fácil de entender.

4. Acesso à memória: A parte superior da [Figura 4.40](#) mostra os dados sendo escritos na memória. Observe que o registrador contendo os dados a serem armazenados foi lido em um estágio anterior e armazenado no ID/EX. A única maneira de disponibilizar os dados durante o estágio MEM é colocar os dados no registrador de pipeline EX/MEM no estágio EX, assim como armazenar o endereço efetivo em EX/MEM.



**FIGURA 4.40 MEM e WB: o quarto e quinto estágios do pipe de uma instrução store.**

No quarto estágio, os dados são escritos na memória de dados para o store. Observe que os dados vêm do registrador de pipeline EX/MEM e que nada é mudado no registrador de pipeline MEM/WB. Uma vez que os dados são escritos na memória, não há nada mais a fazer para a instrução store, de modo que nada acontece no estágio 5.

5. *Escrita do resultado:* A parte inferior da Figura 4.40 mostra a última etapa

do store. Para essa instrução, nada acontece no estágio de escrita do resultado. Como cada instrução por trás do store já está em progresso, não temos como acelerar essas instruções. Logo, uma instrução passa por um estágio mesmo que não haja nada a fazer, pois as instruções posteriores já estão prosseguindo em velocidade máxima.

A instrução store novamente ilustra que, para passar algo de um estágio anterior do pipe a um estágio posterior, a informação precisa ser colocada em um registrador de pipeline; caso contrário, a informação é perdida quando a próxima instrução entrar nesse estágio do pipeline. Para a instrução store, precisamos passar um dos registradores lidos no estágio ID para o estágio MEM, onde é armazenado na memória. Os dados foram colocados inicialmente no registrador de pipeline ID/EX e depois passados para o registrador de pipeline EX/MEM.

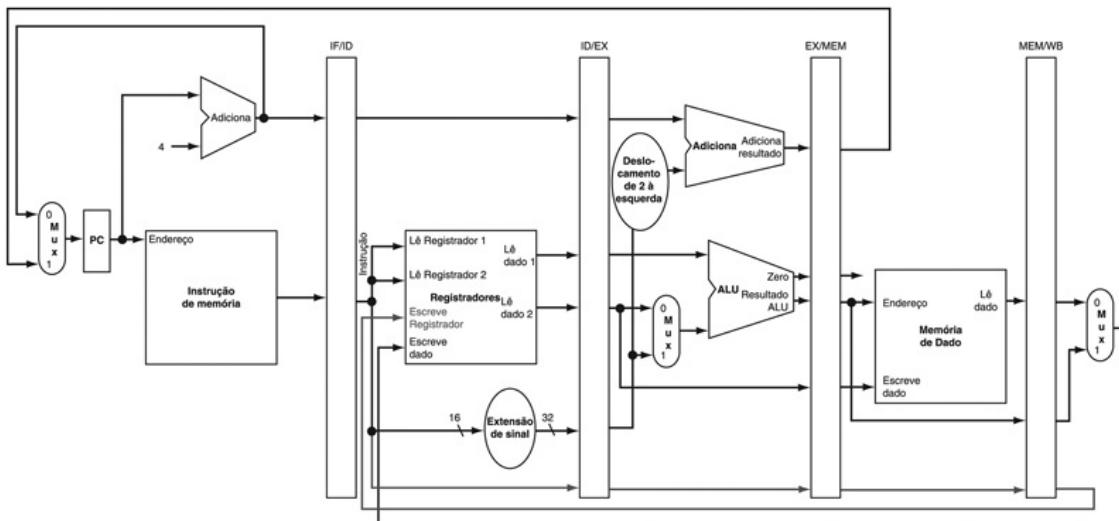
Load e store ilustram um segundo ponto importante: cada componente lógico do caminho de dados — como memória de instruções, portas para leitura de registradores, ALU, memória de dados e porta para escrita de registradores — só pode ser usado dentro de um único estágio do pipeline. Caso contrário, teríamos um *hazard estrutural* (ver Seção Hazards estruturais”, anteriormente neste capítulo). Logo, esses componentes e seu controle podem ser associados a um único estágio do pipeline.

Agora, podemos expor um bug no projeto da instrução load. Você conseguiu ver? Qual registrador é alterado no estágio final da leitura? Mais especificamente, qual instrução fornece o número do registrador de escrita? A instrução no registrador de pipeline IF/ID fornece o número do registrador de escrita, embora essa instrução ocorra consideravelmente *depois* da instrução load!

Logo, precisamos preservar o número do registrador de destino da instrução load. Assim como store passou o *conteúdo* do registrador do ID/EX aos registradores de pipeline EX/MEM para uso no estágio MEM, load precisa passar o número do *registraror* de ID/EX por EX/MEM ao registrador de pipeline MEM/WB, para uso no estágio WB. Outra maneira de pensar sobre a passagem do número de registrador é que, para compartilhar o caminho de dados em pipeline, precisávamos preservar a instrução lida durante o estágio IF, de modo que cada registrador de pipeline contenha uma parte da instrução necessária para esse estágio e para os estágios posteriores.

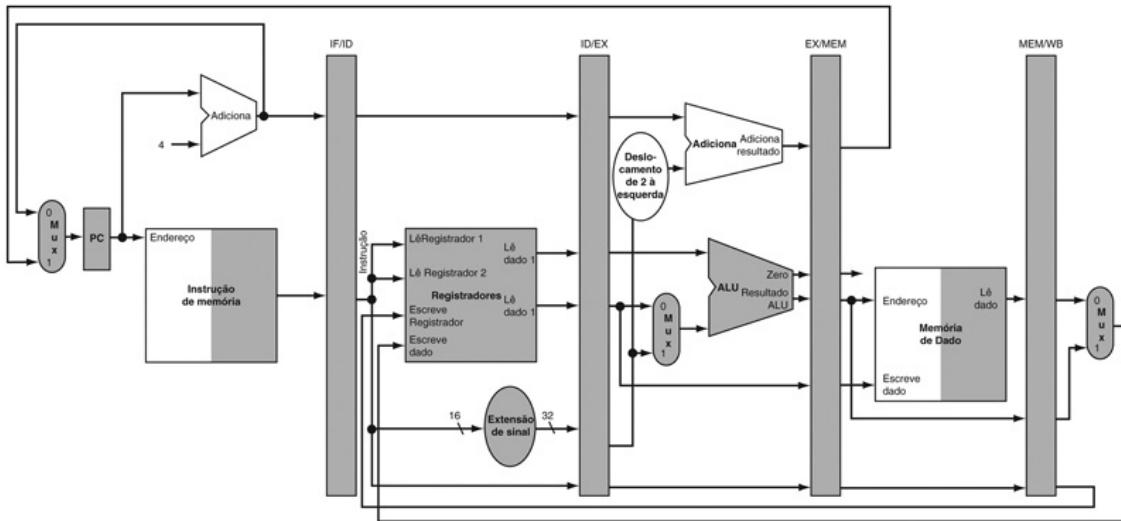
A [Figura 4.41](#) mostra a versão correta do caminho de dados, passando o número do registrador de escrita primeiro ao registrador ID/EX, depois ao registrador EX/MEM e finalmente ao registrador MEM/WB. O número do

registrador é usado durante o estágio WB de modo a especificar o registrador a ser escrito. A [Figura 4.42](#) é um desenho simples do caminho de dados corrigido, destacando o hardware utilizado em todos os cinco estágios da instrução load word nas [Figuras de 4.36 a 4.38](#). Veja na [Seção 4.8](#) uma explicação de como fazer a instrução branch funcionar como esperado.



**FIGURA 4.41** O caminho de dados em pipeline corrigido para lidar corretamente com a instrução load.

O número do registrador de escrita agora vem do registrador de pipeline MEM/WB junto com os dados. O número do registrador é passado do estágio do pipe ID até alcançar o registrador de pipeline MEM/WB, acrescentando mais 5 bits aos três últimos registradores de pipeline. Esse novo caminho aparece em destaque.



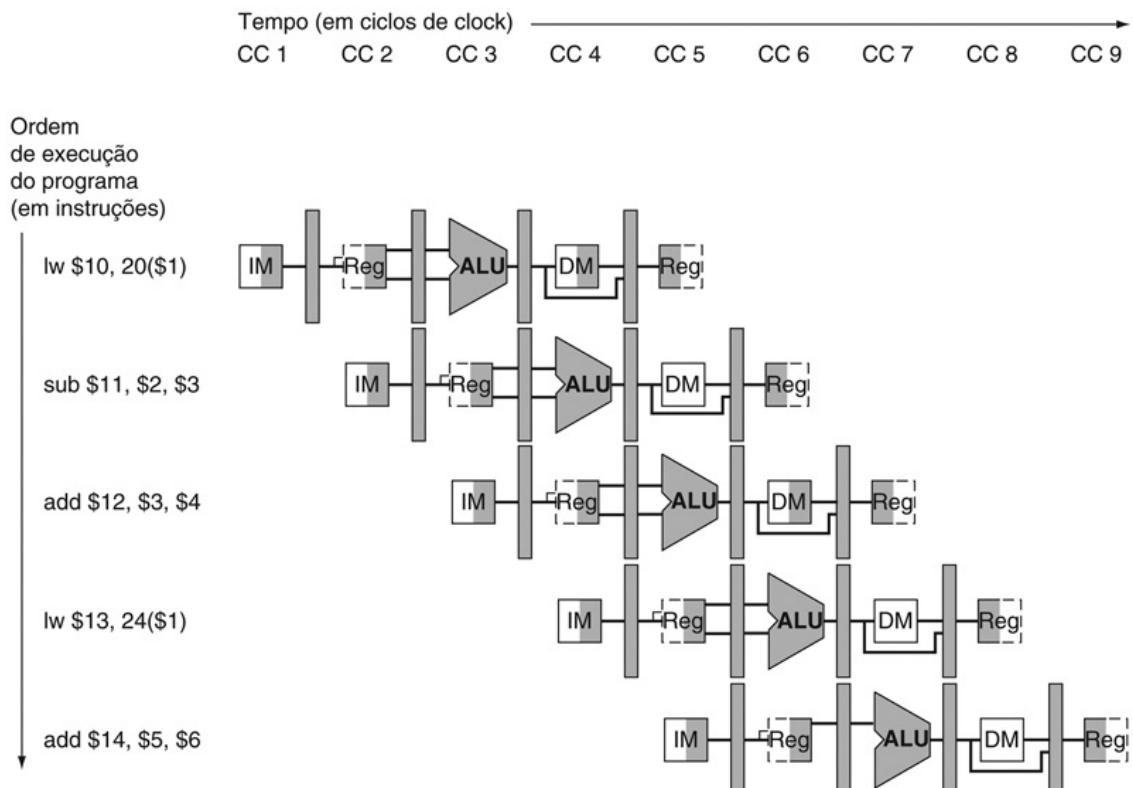
**FIGURA 4.42** A parte do caminho de dados na [Figura 4.41](#) usada em todos os cinco estágios de uma instrução load.

## Representando pipelines graficamente

Pipelining pode ser difícil de entender, pois muitas instruções estão executando simultaneamente em um único caminho de dados em cada ciclo de clock. Para ajudar na compreensão, existem dois estilos básicos de figuras de pipeline: *diagramas de pipeline com múltiplos ciclos de clock*, como a [Figura 4.34](#), e *diagramas de pipeline com único ciclo de clock*, como as [Figuras de 4.36 a 4.40](#). Os diagramas com múltiplos ciclos de clock são mais simples, mas não contêm todos os detalhes. Por exemplo, considere esta sequência de cinco instruções:

lw	\$10, 20(\$1)
sub	\$11, \$2, \$3
add	\$12, \$3, \$4
lw	\$13, 24(\$1)
add	\$14, \$5, \$6

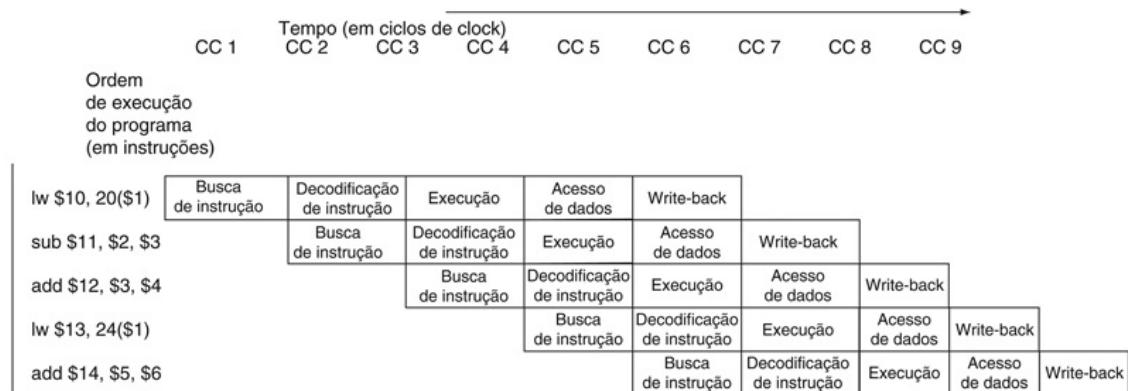
A Figura 4.43 mostra o diagrama de pipeline com múltiplos ciclos de clock para essas instruções. O tempo avança da esquerda para a direita na horizontal, semelhante ao pipeline da lavanderia, na Figura 4.25. Uma representação dos estágios do pipeline é colocada em cada parte do eixo de instruções, ocupando os ciclos de clock apropriados. Esses caminhos de dados estilizados representam os cinco estágios do nosso pipeline, mas um retângulo indicando o nome de cada estágio do pipe também funciona bem. A Figura 4.44 mostra a versão mais tradicional do diagrama de pipeline com múltiplos ciclos de clock. Observe que a Figura 4.43 mostra os recursos físicos utilizados em cada estágio, enquanto a Figura 4.44 usa o *nome* de cada estágio.



**FIGURA 4.43** Diagrama de pipeline com múltiplos ciclos de clock das cinco instruções.

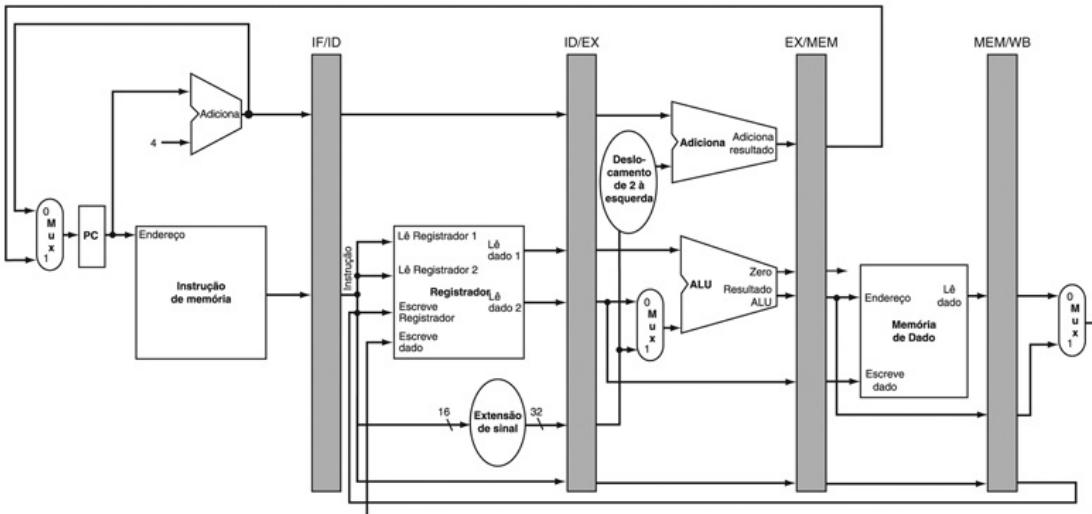
Este estilo de representação de pipeline mostra a execução completa das instruções em uma única figura. As instruções são listadas por ordem de execução, de cima para baixo e os ciclos de clock se movem da esquerda para a direita. Ao contrário da Figura 4.28, aqui, mostramos os registradores de pipeline entre cada estágio. A Figura 4.44 mostra a maneira tradicional de

desenhar esse diagrama.



**FIGURA 4.44** Diagrama de pipeline tradicional com múltiplos ciclos de clock, com as cinco instruções da [Figura 4.43](#).

Os diagramas de pipeline de ciclo único de clock mostram o estado do caminho de dados inteiro durante um único ciclo de clock, e normalmente todas as cinco instruções no pipeline são identificadas por rótulos acima de seus respectivos estágios do pipeline. Usamos esse tipo de figura para mostrar os detalhes do que está acontecendo dentro do pipeline durante cada ciclo de clock; normalmente, os desenhos aparecem em grupos, para mostrar a operação do pipeline durante uma sequência de ciclos de clock. Usamos diagramas de ciclo múltiplo de clock, a fim de oferecer sinopses de situações de pipelining. Um diagrama de ciclo único de clock representa uma fatia vertical de um conjunto do diagrama com múltiplos ciclos de clock, mostrando o uso do caminho de dados em cada uma das instruções do pipeline no ciclo de clock designado. Por exemplo, a [Figura 4.45](#) mostra o diagrama com ciclo único de clock correspondente ao ciclo de clock 5 das [Figuras 4.43 e 4.44](#). Obviamente, os diagramas com ciclo único de clock possuem mais detalhes e ocupam muito mais espaço para mostrar o mesmo número de ciclos de clock. Os exercícios pedem que você crie esses diagramas para outras sequências de código.



**FIGURA 4.45** O diagrama com ciclo único de clock correspondente ao ciclo de clock 5 do pipeline das Figuras 4.43 e 4.44.

Como você pode ver, uma figura com ciclo único de clock é uma fatia vertical de um diagrama com múltiplos ciclos de clock.

## Verifique você mesmo

Um grupo de alunos discutia sobre a eficiência de um pipeline de cinco estágios quando um deles apontou que nem todas as instruções estão ativas em cada estágio do pipeline. Depois de decidir ignorar os efeitos dos hazards, eles fizeram as quatro afirmações a seguir. Quais delas estão corretas?

1. Permitir que jumps, branches e instruções da ALU utilizem menos estágios do que os cinco necessários pela instrução load, aumentará o desempenho do pipeline sob todas as circunstâncias.
2. Tentar permitir que algumas instruções utilizem menos ciclos não ajuda, pois a vazão é determinada pelo ciclo do clock; o número de estágios do pipe por instrução afeta a latência, e não a vazão.
3. Você não pode fazer com que as instruções da ALU utilizem menos ciclos, devido à escrita do resultado, mas os branches e jumps podem utilizar menos ciclos, de modo que existe alguma oportunidade de melhoria.
4. Em vez de tentar fazer com que as instruções utilizem menos ciclos de clock, devemos explorar um meio de tornar o pipeline mais longo, de

modo que as instruções utilizem mais ciclos, porém com ciclos mais curtos. Isso poderia melhorar o desempenho.

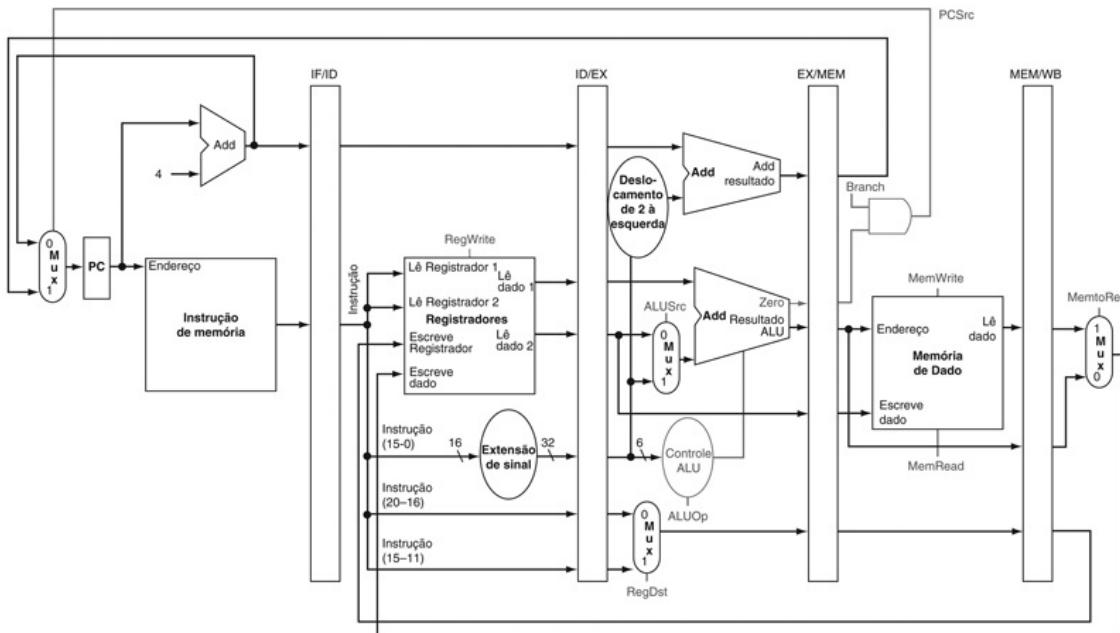
## Controle em pipeline

*No computador 6600, talvez ainda mais do que em qualquer computador anterior, o sistema de controle faz a diferença.*

*James Thornton, Design of a Computer: The Control Data 6600, 1970*

Assim como acrescentamos controle ao caminho de dados simples na [Seção 4.3](#), agora acrescentamos controle ao caminho de dados de um pipeline. Começamos com um projeto simples, que vê o problema por meio de óculos cor-de-rosa.

O primeiro passo é rotular as linhas de controle no caminho de dados existente. A [Figura 4.46](#) mostra essas linhas. Pegamos o máximo possível emprestado do controle para o caminho de dados simples da [Figura 4.17](#). Em particular, usamos a mesma lógica de controle da ALU, lógica de desvio, multiplexador do registrador destino e linhas de controle. Essas funções são definidas nas [Figuras 4.12, 4.16 e 4.18](#). Reproduzimos as principais informações nas [Figuras 4.47 a 4.49](#) em uma única página, de modo a facilitar o acompanhamento do restante do texto.



**FIGURA 4.46** O caminho de dados em pipeline da [Figura 4.41](#) com sinais de controle identificados.

Esse caminho de dados toma emprestado a lógica de controle para a origem do PC, o número do registrador destino e o controle da ALU, da [Seção 4.4](#). Observe que agora precisamos do campo funct (código de função) de 6 bits da instrução no estágio EX como entrada para o controle da ALU, de modo que esses bits também precisam ser incluídos no registrador de pipeline ID/EX. Lembre-se de que esses 6 bits também são os 6 bits menos significativos do campo imediato da instrução, de modo que o registrador de pipeline ID/EX pode fornecê-los a partir do campo imediato, já que a extensão do sinal deixa esses bits inalterados.

Opcode da instrução	OpALU	Operação da instrução	Campo funct	Ação da ALU desejada	Entrada do controle da ALU
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Desvio igual	01	Desvio igual	XXXXXX	subtract	0110
Tipo R	10	add	100000	add	0010
Tipo R	10	subtract	100010	subtract	0110
Tipo R	10	AND	100100	AND	0000
Tipo R	10	OR	100101	OR	0001
Tipo R	10	definir com menos de	101010	definir com menos de	0111

**FIGURA 4.47** Uma cópia da [Figura 4.12](#).

Essa figura mostra como os bits do controle da ALU são definidos dependendo dos bits de controle ALUOp e dos

diferentes códigos de função para instruções tipo R.

Nome do sinal	Efeito quando inativo (0)	Efeito quando ativo (1)
RegDst	O número do registrador destino para a entrada Registrador para escrita vem do campo rt (bits 20:16).	O número do registrador destino para a entrada Registrador para escrita vem do campo rd (bits 15:11).
EscreveReg	Nenhum.	O registrador na entrada Registrador para escrita é escrito com o valor na entrada Dados para escrita.
OrigALU	O segundo operando da ALU vem da segunda saída do banco de registradores (Dados da leitura 2).	O segundo operando da ALU consiste nos 16 bits mais baixos da instrução com sinal estendido.
OrigPC	O PC é substituído pela saída do somador que calcula o valor de PC + 4.	O PC é substituído pela saída do somador que calcula o destino do desvio.
LeMem	Nenhum.	O conteúdo da memória de dados designado pela entrada Endereço é colocado na saída Dados da leitura.
EscreveMem	Nenhum.	O conteúdo da memória de dados designado pela entrada Endereço é substituído pelo valor na entrada Dados para escrita.
MemparaReg	O valor enviado para a entrada Dados para escrita do banco de registradores vem da ALU.	O valor enviado para a entrada Dados para escrita do banco de registradores vem da memória de dados.

**FIGURA 4.48** Uma cópia da [Figura 4.16](#).

A função de cada um dos sete sinais de controle é definida. As linhas de controle da ALU (ALUOp) são definidas na segunda coluna da [Figura 4.47](#). Quando um controle de 1 bit para um multiplexador bidirecional é ativado, o multiplexador seleciona a entrada correspondente a 1. Caso contrário, se o controle for desativado, o multiplexador seleciona a entrada 0. Observe que PCScr é controlado por uma porta lógica AND na [Figura 4.46](#). Se o sinal Branch e o sinal Zero da ALU estiverem ativos, então PCScr é 1; caso contrário, ele é 0. O controle define o sinal Branch somente durante uma instrução beq; caso contrário, o PCScr é 0.

Instrução	Linhas de controle do estágio de execução/cálculo de endereço				Linhas de controle do estágio de acesso à memória			Linhas de controle do estágio de escrita do resultado	
	RegDst	OpALU1	OpALU0	OrigALU	Desvio	Le Mem	Escreve Mem	Escreve Reg	Mem para Reg
Formato R	1	1	0	0	0	0	0	1	0
Iw	0	0	0	1	0	1	0	1	1
Sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

**FIGURA 4.49** Os valores das linhas de controle são iguais aos da [Figura 4.18](#), mas foram reorganizados em três grupos, correspondentes aos três últimos estágios do pipeline.

Assim como ocorreu com a implementação com ciclo único, consideramos que o PC é escrito a cada ciclo de clock, de modo que não existe um sinal de

escrita separado para o PC. Pelo mesmo argumento, não existem sinais de escrita para os registradores de pipeline (IF/ID, ID/EX, EX/MEM e MEM/WB), pois os registradores de pipeline também são escritos durante cada ciclo de clock.

A fim de especificar o controle para o pipeline, só precisamos definir os valores de controle durante cada estágio do pipeline. Como cada linha de controle está associada a um componente ativo em apenas um estágio do pipeline, podemos dividir as linhas de controle em cinco grupos, de acordo com o estágio do pipeline.

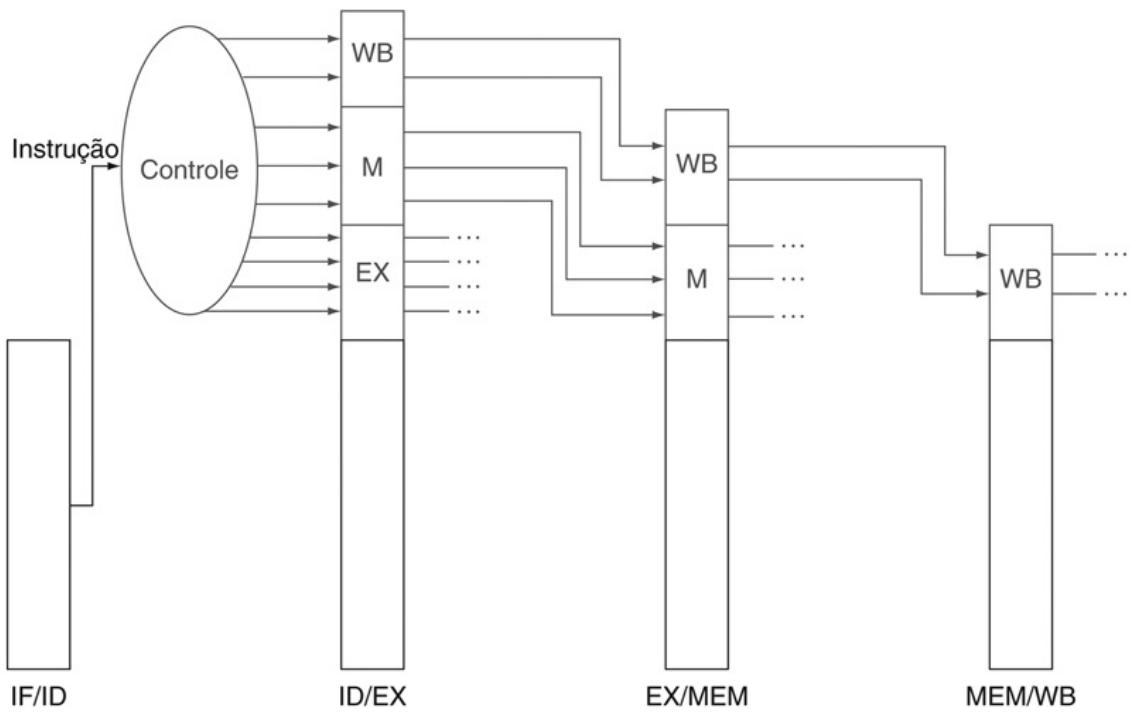
1. *Busca de instruções*: Os sinais de controle para ler a memória de instruções e escrever o PC sempre são ativados, de modo que não existe nada de especial para controlar nesse estágio do pipeline.
2. *Decodificação de instruções/leitura do banco de registradores*: Como no estágio anterior, o mesmo acontece em cada ciclo de clock, de modo que não existem linhas de controle opcionais para definir.
3. *Execução/cálculo de endereço*: Os sinais a serem definidos são RegDst, ALUOp e ALUScr ([Figuras 4.47 e 4.48](#)). Os sinais selecionam o registrador Resultado, a operação da ALU e Dados da leitura 2 ou um imediato com sinal estendido para a ALU.
4. *Acesso à memória*: As linhas de controle definidas nesse estágio são Branch, ReadMem e WriteMem. Esses sinais são definidos pelas instruções branch equal, load e store, respectivamente. Lembre-se de que o PCScr na [Figura 4.48](#) seleciona o próximo endereço sequencial, a menos que o controle ative Branch e o resultado da ALU seja zero.
5. *Escrita do resultado*: As duas linhas de controle são MemtoReg, que decide entre enviar o resultado da ALU ou o valor da memória para o banco de registradores, e WriteRegWriteReg, que escreve o valor escolhido.

Como a utilização de um pipeline no caminho de dados deixa inalterado o significado das linhas de controle, podemos usar os mesmos valores de controle de antes. A [Figura 4.49](#) tem os mesmos valores da [Seção 4.4](#), mas agora as nove linhas de controle estão agrupadas por estágio do pipeline.

A implementação do controle significa definir as nove linhas de controle desses valores em cada estágio, para cada instrução. A maneira mais simples de fazer isso é estender os registradores do pipeline de modo a incluir informações de controle.

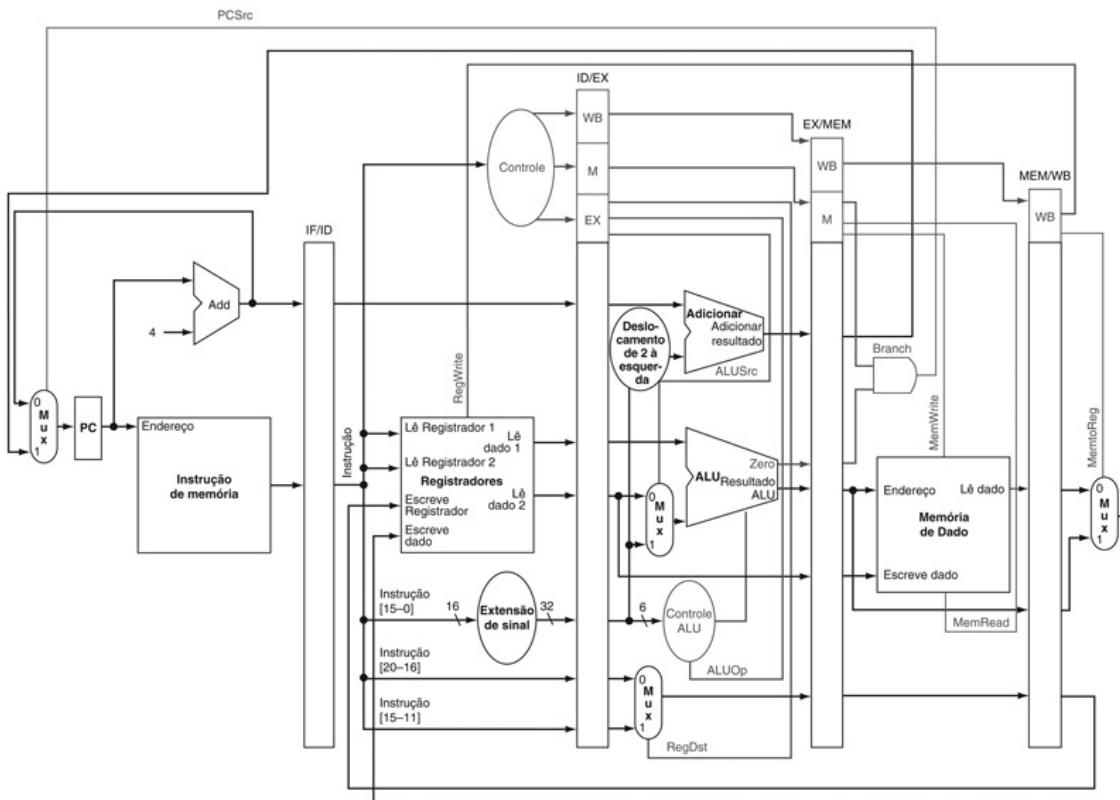
Como as linhas de controle começam com o estágio EX, podemos criar a informação de controle durante a decodificação da instrução. A [Figura 4.50](#)

mostra que esses sinais de controle são usados no respectivo estágio do pipeline, à medida que a instrução se move por ele, assim como o número do registrador destino para loads desce pelo pipeline da [Figura 4.41](#). A [Figura 4.51](#) mostra o caminho de dados completo, com os registradores de pipeline estendidos e com as linhas de controle conectadas ao estágio apropriado



**FIGURA 4.50** As linhas de controle para os três estágios finais.

Observe que quatro das nove linhas de controle são usadas na fase EX, com as cinco linhas de controle restantes passadas adiante para o registrador de pipeline EX/MEM, a fim de manter as linhas de controle; três são usadas durante o estágio MEM, e as duas últimas são passadas a MEM/WB, para uso no estágio WB.



**FIGURA 4.51** O caminho de dados em pipeline da [Figura 4.46](#), com os sinais de controle conectados às partes de controle dos registradores de pipeline.

Os valores de controle para os três últimos estágios são criados durante o estágio de decodificação de instruções e, depois, colocados no registrador de pipeline ID/EX. As linhas de controle para cada estágio do pipe são usadas, e as linhas de controle restantes depois disso são passadas ao próximo estágio do pipeline.

## 4.7. Hazards de dados: forwarding versus stalls

*Como assim por que teve de ser criado? É um bypass. Você precisa criar bypasses.*

*Douglas Adams, The Hitchhiker's Guide to the Galaxy, 1979*

Os exemplos da seção anterior mostram o poder da execução em pipeline e como o hardware realiza a tarefa. Agora é hora de retirarmos os óculos cor-de-rosa e examinarmos o que acontece com os programas reais. As instruções nas

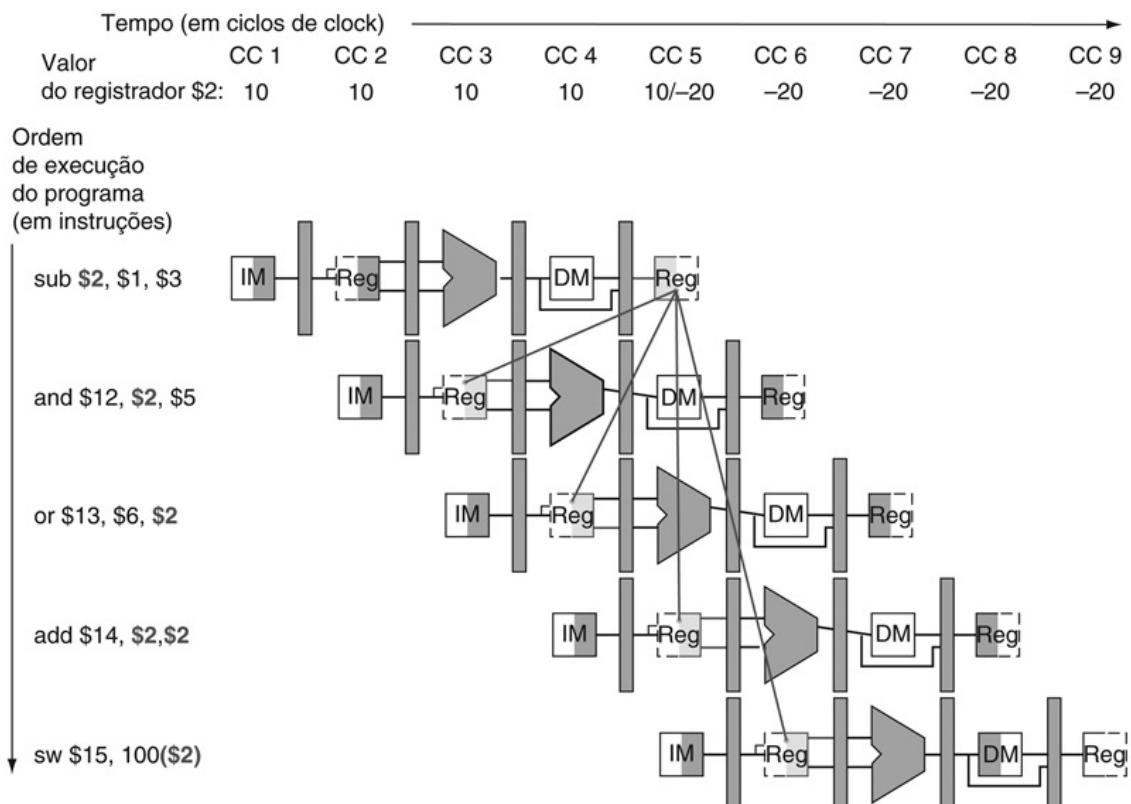
[Figuras de 4.43](#) a [4.45](#) eram independentes; nenhuma delas usava os resultados calculados por qualquer uma das outras. Mesmo assim, na [Seção 4.5](#), vimos que os hazards de dados são obstáculos para a execução em pipeline.

Vejamos uma sequência com muitas dependências, indicadas com realce:

```
sub    $2, $1,$3      # Registrador $2 escrito por sub
and    $12,$2,$5      # 1º operando ($2) depende de sub
or     $13,$6,$2      # 2º operando ($2) depende de sub
add    $14,$2,$2      # 1º e 2º operandos ($2) dependem de sub
sw     $15,100($2)    # Base ($2) depende de sub
```

As quatro últimas instruções são todas dependentes do resultado no registrador \$2 da primeira instrução. Se o registrador \$2 tivesse o valor 10 antes da instrução subtract e -20 depois dela, o programador desejaria que -20 fosse usado nas instruções seguintes, que se referem ao registrador \$2.

Como essa sequência funcionaria com nosso pipeline? A [Figura 4.52](#) ilustra a execução dessas instruções usando uma representação de pipeline com múltiplos ciclos de clock. Para demonstrar a execução dessa sequência de instruções em nosso pipeline atual, o topo da [Figura 4.52](#) mostra o valor do registrador \$2, que muda durante o ciclo de clock 5, quando a instrução sub escreve seu resultado.



**FIGURA 4.52 Dependências em pipeline em uma sequência de cinco instruções usando caminhos de dados simplificados para mostrar as dependências.**

Todas as ações dependentes são mostradas em cinza, e “CC 1” no alto da figura significa o ciclo de clock 1. A primeira instrução escreve em \$2, e todas as instruções seguintes leem de \$2. Esse registrador é escrito no ciclo de clock 5, de modo que o valor correto está indisponível antes do ciclo de clock 5. (Uma leitura de um registrador durante um ciclo de clock retorna o valor escrito no final da primeira metade do ciclo, quando ocorre tal escrita.) As linhas coloridas do caminho de dados do topo para os inferiores mostram as dependências. Aquelas que precisam retornar no tempo são os *hazards de dados do pipeline*.

O último hazard em potencial pode ser resolvido pelo projeto do hardware do banco de registradores: o que acontece quando um registrador é lido e escrito no mesmo ciclo de clock? Consideramos que a escrita está na primeira metade do ciclo de clock e a leitura está na segunda metade, de modo que esta fornece o que foi escrito. Como acontece para muitas implementações dos bancos de registradores, não temos hazard de dados nessa situação.

A Figura 4.52 mostra que os valores lidos para o registrador \$2 *não* seriam o

resultado da instrução sub, a menos que a leitura ocorresse durante o ciclo de clock 5 ou posterior. Assim, as instruções que receberiam o valor correto de -20 são add e sw; as instruções AND e OR receberiam o valor incorreto de 10! Usando esse estilo de desenho, esses problemas se tornam aparentes quando uma linha de dependência retorna no tempo.

Conforme dissemos na [Seção 4.5](#), o resultado desejado está disponível no final do estágio EX ou no ciclo de clock 3. Quando os dados são realmente necessários pelas instruções AND e OR? No início do estágio EX ou nos ciclos de clock 4 e 5, respectivamente. Assim, podemos executar esse segmento sem stalls se simplesmente os dados sofrerem *forwarding* assim que estiverem disponíveis para quaisquer unidades que precisam deles, antes de estarem disponíveis para leitura do banco de registradores.

Como funciona o forwarding? Para simplificar o restante desta seção, consideramos apenas o desafio de forwarding para uma operação no estágio EX, que pode ser uma operação da ALU ou um cálculo de endereço efetivo. Isso significa que, quando uma instrução tenta usar um registrador em seu estágio EX, que uma instrução anterior pretende escrever em seu estágio WB, na realidade precisamos dos valores como entradas para a ALU.

Uma notação que nomeia os campos dos registradores de pipeline permite uma notação mais precisa das dependências. Por exemplo, “ID/EX.RegistradorRs” refere-se ao número de um registrador cujo valor se encontra no registrador de pipeline ID/EX; ou seja, aquele da primeira porta de leitura do banco de registradores. A primeira parte do nome, à esquerda do ponto, é o nome do registrador de pipeline; a segunda parte é o nome do campo nesse registrador. Usando essa notação, os dois pares de condições de hazard são:

- 1a. EX/MEM.RegistradorRd = ID/EX.RegistradorRs
- 1b. EX/MEM.RegistradorRd = ID/EX.RegistradorRt
- 2a. MEM/WB.RegistradorRd = ID/EX.RegistradorRs
- 2b. MEM/WB.RegistradorRd = ID/EX.RegistradorRt

O primeiro hazard na sequência da [Seção 4.7](#) está no registrador \$2, entre o resultado de sub \$2,\$1,\$3 e o primeiro operando de leitura de and \$12,\$2,\$5. Esse hazard pode ser detectado quando a instrução and está no estágio EX, e a instrução anterior está no estágio MEM, de modo que este é o hazard 1a:

$$\text{EX/MEM.RegistradorRd} = \text{ID/EX.RegistradorRs} = \$2$$

## Detecção de dependência

### Exemplo

Classifique as dependências nesta sequência da Seção 4.7:

```
sub $2,    $1, $3  # Registrador $2 escrito por sub
and $12,   $2, $5  # 1º operando ($2) escrito por sub
or  $13,   $6, $2  # 2º operando ($2) escrito por sub
add $14,   $2, $2  # 1º e 2º operandos ($2) escrito por sub
sw   $15, 100($2) # Base ($2) escrito por sub
```

### Resposta

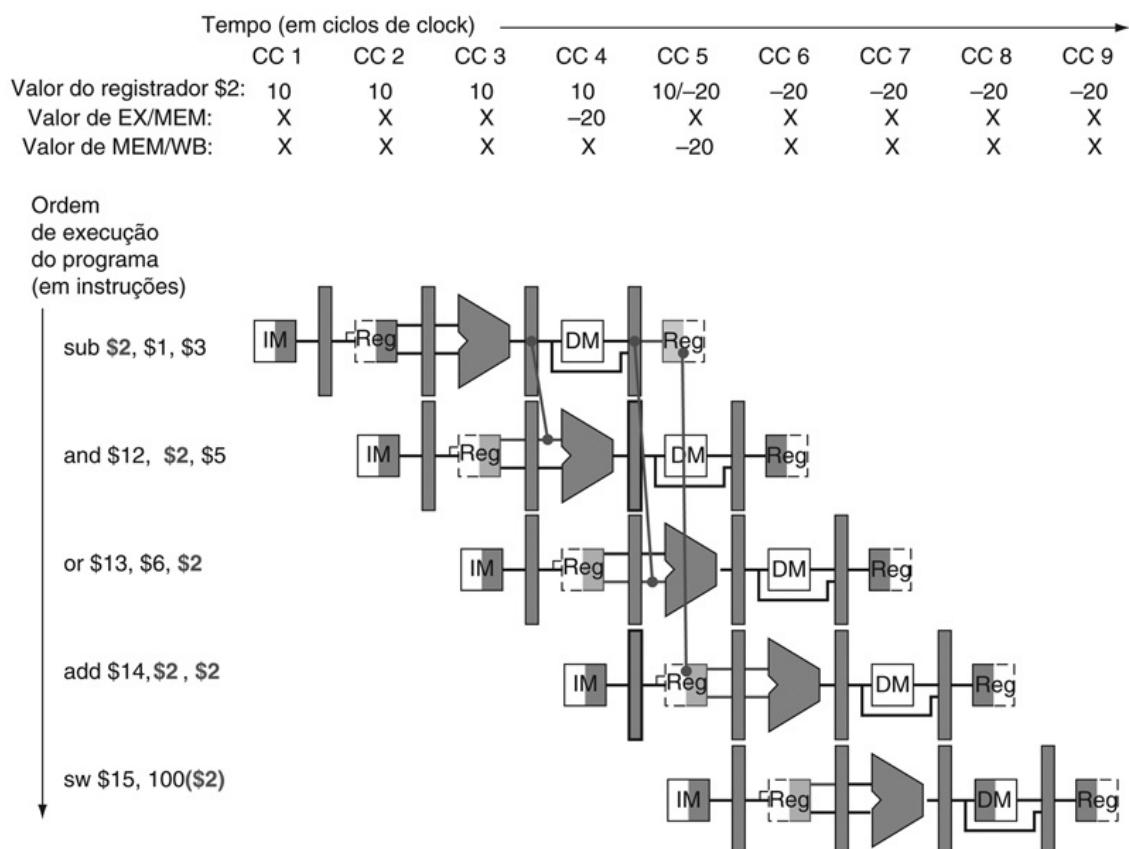
Conforme já mencionamos, o sub-and é um hazard tipo 1a. Os outros hazards são

- sub-or é um hazard tipo 2b:  
MEM/WB.RegistradorRd = ID/EX.RegistradorRt = \$2
- As duas dependências em sub-add não são hazards, pois o banco de registradores fornece os dados apropriados durante o estágio ID de add.
- Não existe hazard de dados entre sub e sw, porque sw lê \$2 no ciclo de clock depois que sub escreve \$2.

Como algumas instruções não escrevem em registradores, essa política não é exata; às vezes, poderia haver forwarding indevidamente. Uma solução é simplesmente verificar se o sinal WriteReg estará ativo: examinando o campo de controle WB do registrador de pipeline durante os estágios EX e MEM, é possível determinar se WriteReg está ativo. Lembre-se de que o MIPS exige que cada uso de \$0 como operando deve gerar um valor de operando 0. Se uma instrução no pipeline tiver \$0 como seu destino (por exemplo, sll \$0,\$1,2), queremos evitar o forwarding do seu valor possivelmente diferente de zero. Não encaminhar os resultados destinados a \$0 libera o programador assembly e o compilador de qualquer requisito para evitar o uso de \$0 como destino. As condições anteriores, portanto, funcionam corretamente desde que acrescentemos EX/MEM.RegistradorRd ≠ 0 à primeira condição de hazard e MEM/WB.RegistradorRd ≠ 0 à segunda.

Agora que podemos detectar os hazards, metade do problema está resolvido — mas ainda precisamos fazer o forwarding dos dados corretos.

A Figura 4.53 mostra as dependências entre os registradores de pipeline e as entradas da ALU para a mesma sequência de código da Figura 4.52. A mudança é que a dependência começa por um registrador de *pipeline*, em vez de esperar pelo estágio WB para escrever no banco de registradores. Assim, os dados exigidos existem a tempo para as instruções posteriores, com os registradores de pipeline mantendo os dados para forwarding.



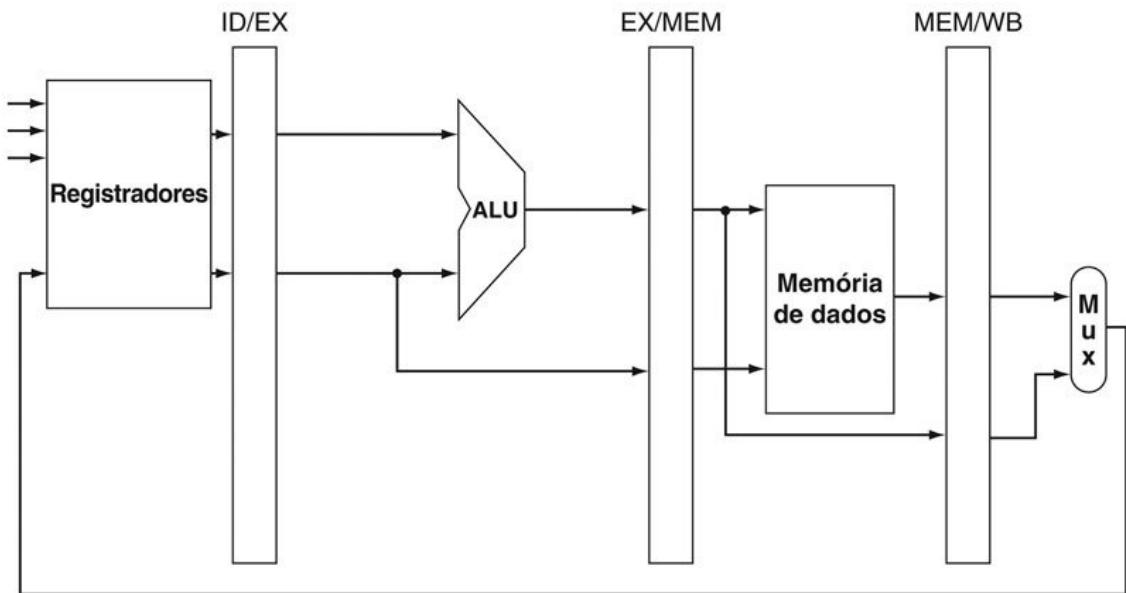
**FIGURA 4.53** As dependências entre os registradores de pipeline se movem para a frente no tempo, de modo que é possível fornecer as entradas para a ALU necessárias para a instrução AND e para a instrução OR fazendo forwarding dos resultados encontrados nos registradores de pipeline.

Os valores nos registradores de pipeline mostram que o valor desejado está disponível antes de ser escrito no banco de registradores. Consideraremos que o banco de registradores encaminha valores lidos e escritos durante o mesmo ciclo de

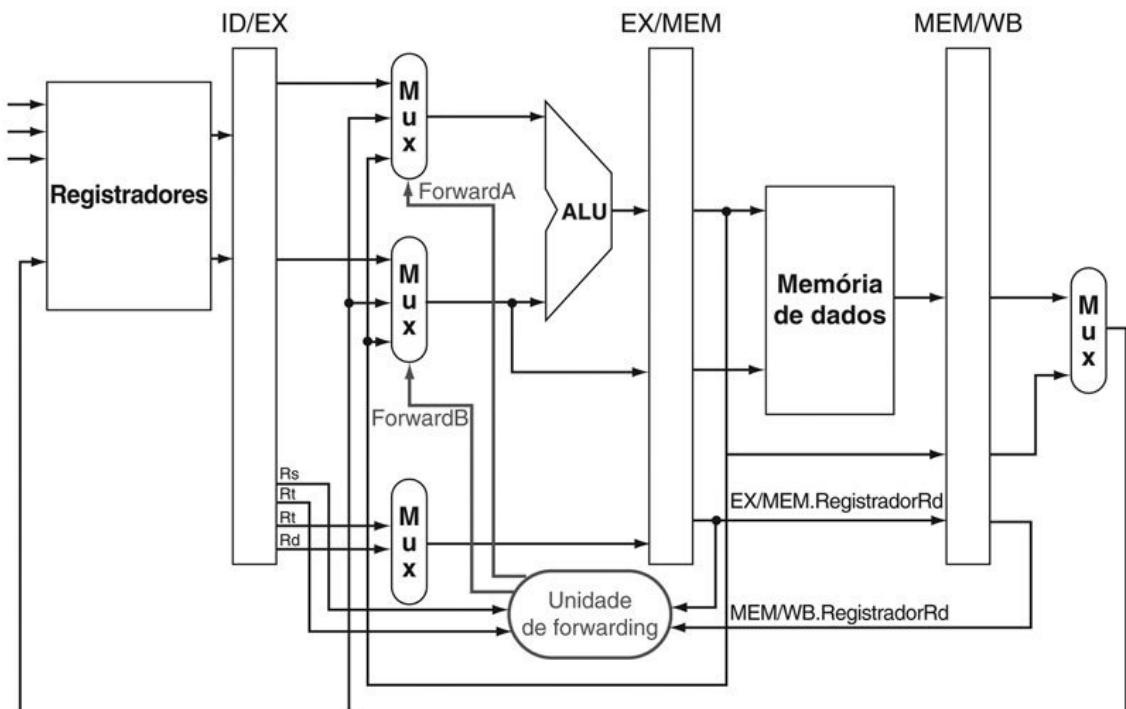
`clock`, de modo que `add` não causa stall, mas os valores vêm do banco de registradores e não de um registrador de pipeline. O “forwarding” do banco de registradores — ou seja, a leitura apanha o valor da escrita nesse ciclo de `clock` — é o motivo pelo qual o ciclo de `clock` 5 mostra o registrador `$2`, tendo o valor 10 no início e -20 no final do ciclo de `clock`. Como no restante desta seção, tratamos de todo o forwarding, exceto o valor a ser armazenado por uma instrução `store`.

Se pudermos pegar as entradas da ALU de *qualquer* registrador de pipeline, e não apenas de ID/EX, então podemos fazer o forwarding dos dados corretos. Acrescentando multiplexadores à entrada da ALU e com os controles apropriados, podemos executar o pipeline em velocidade máxima na presença dessas dependências de dados.

Por enquanto, vamos considerar que as únicas instruções para as quais precisamos de forwarding são as quatro instruções no formato R: `add`, `sub`, `AND` e `OR`. A [Figura 4.54](#) mostra um detalhe da ALU e do registrador de pipeline, antes e depois de acrescentar o forwarding. A [Figura 4.55](#) mostra os valores das linhas de controle para os multiplexadores da ALU que selecionam os valores do banco de registradores ou um dos valores de forwarding.



a. Sem forwarding



b. Com forwarding

**FIGURA 4.54** Em cima estão a ALU e os registradores de pipeline antes da inclusão do forwarding.

Embaixo, os multiplexadores foram expandidos para acrescentar os caminhos de forwarding e mostramos a unidade de forwarding. O hardware novo aparece em um destaque. No

entanto, essa figura é um desenho estilizado, omitindo os detalhes do caminho de dados completo, como o hardware de extensão de sinal. Observe que o campo ID/EX.RegistradorRt aparece duas vezes, uma para conectar ao Mux e uma para a unidade de forwarding, mas esse é um sinal único. Como na discussão anterior, isso ignora o forwarding de um valor armazenado por uma instrução store. Observe também que esse mecanismo funciona, inclusive, para instruções slt.

<b>Controle do Mux</b>	<b>Origem</b>	<b>Explicação</b>
ForwardA = 00	ID/EX	O primeiro operando da ALU vem do banco de registradores.
ForwardA = 10	EX/MEM	O primeiro operando da ALU sofre forwarding do resultado anterior da ALU.
ForwardA = 01	MEM/WB	O primeiro operando da ALU sofre forwarding da memória de dados ou de um resultado anterior da ALU.
ForwardB = 00	ID/EX	O segundo operando da ALU vem do banco de registradores.
ForwardB = 10	EX/MEM	O segundo operando da ALU sofre forwarding do resultado anterior da ALU.
ForwardB = 01	MEM/WB	O segundo operando da ALU sofre forwarding da memória de dados ou de um resultado anterior da ALU.

**FIGURA 4.55 Os valores de controle para os multiplexadores de forwarding da Figura 4.54.**

O imediato com sinal que é outra entrada da ALU é descrito na Seção “Detalhamento” ao final desta seção.

Esse controle de forwarding estará no estágio EX porque os multiplexadores de forwarding da ALU são encontrados nesse estágio. Assim, temos de passar os números dos registradores operando do estágio ID por meio do registrador de pipeline ID/EX, para determinar se os valores devem sofrer forwarding. Já temos o campo rt (bits 20-16). Antes do forwarding, o registrador ID/EX não precisava incluir espaço a fim de manter o campo rs. Logo, rs (bits 25-21) é acrescentado a ID/EX.

Agora, vamos escrever as duas condições para detectar hazards e os sinais de controle para resolvê-los:

1. *Hazard EX:*

```

if (EX/MEM.WriteReg
and (EX/MEM.RegistradorRd ≠ 0)
and (EX/MEM.RegistradorRd = ID/EX.RegistradorRs)) ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegistradorRd ≠ 0)
and (EX/MEM.RegistradorRd = ID/EX.RegistradorRt)) ForwardB = 10

```

Observe que o campo EX/MEM.RegistradorRd é o destino de registrador para uma instrução da ALU (que vem do campo Rd da instrução) ou um load (que vem do campo Rt).

Esse caso faz o forwarding do resultado da instrução anterior para qualquer entrada da ALU. Se a instrução anterior tiver de escrever no banco de registradores e o número do registrador de escrita combinar com o número do registrador de leitura das entradas A ou B da ALU, desde que não seja o registrador 0, então direcione o multiplexador para que pegue o valor do registrador de pipeline EX/MEM.

## 2. Hazard MEM:

```

if (MEM/WB.WriteReg
and (MEM/WB.RegistradorRd ≠ 0)
and (MEM/WB.RegistradorRd = ID/EX.RegistradorRs)) ForwardA = 01
if (MEM/WB.WriteReg
and (MEM/WB.RegistradorRd ≠ 0)
and (MEM/WB.RegistradorRd = ID/EX.RegistradorRt)) ForwardB = 01

```

Como dissemos, não existe hazard no estágio WB porque consideramos que o banco de registradores fornece o resultado correto se a instrução no estágio ID ler o mesmo registrador escrito pela instrução no estágio WB. Tal banco de registradores realiza outra forma de forwarding, mas isso ocorre dentro do banco de registradores.

Uma complicação são os hazards de dados em potencial entre o resultado da instrução no estágio WB, o resultado da instrução no estágio MEM e o operando de origem da instrução no estágio ALU. Por exemplo, ao somar um vetor de números em um único registrador, uma sequência de instruções lerá e escreverá no mesmo registrador:

```

add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
    .
    .
    .

```

Nesse caso, o resultado sofre forwarding do estágio MEM, pois o resultado no estágio MEM é o mais recente. Assim, o controle para o hazard em MEM seria (com os acréscimos destacados):

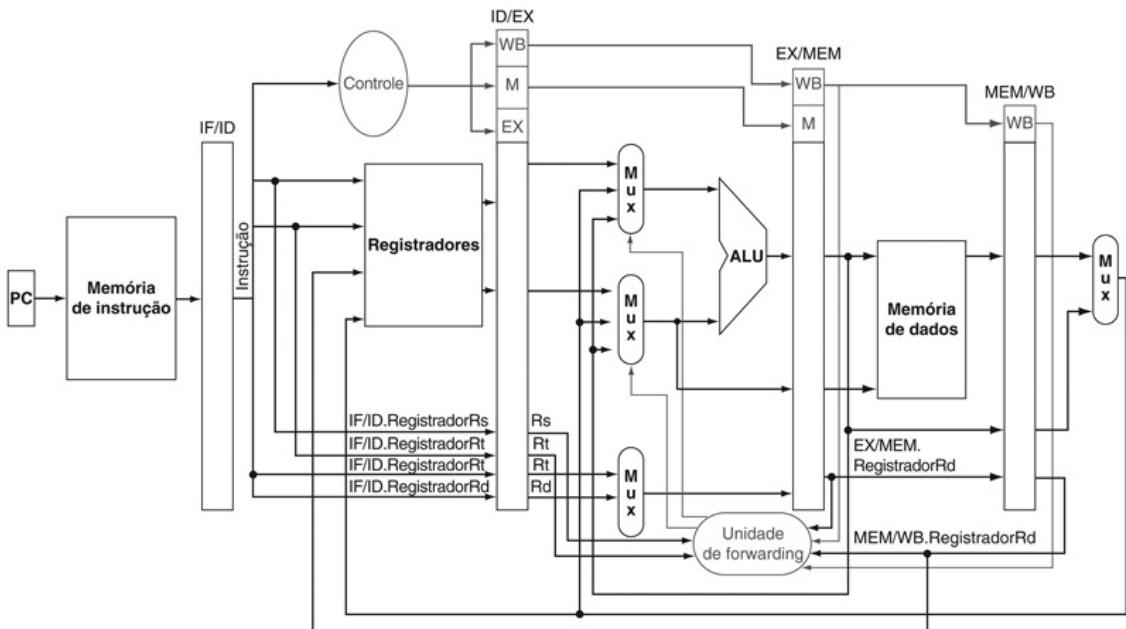
```

if (MEM/WB.WriteReg
and (MEM/WB.RegistradorRd ≠ 0)
and not(EX/MEM.WriteReg and (EX/MEM.RegistradorRd ≠ 0)
        and (EX/MEM.RegistradorRd ≠ ID/EX.RegistradorRs))
and (MEM/WB.RegistradorRd = ID/EX.RegistradorRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegistradorRd ≠ 0))
        and (EX/MEM.RegistradorRd ≠ ID/EX.RegistradorRt)
and (MEM/WB.RegistradorRd = ID/EX.RegistradorRt)) ForwardB = 01

```

A [Figura 4.56](#) mostra o hardware necessário para dar suporte ao forwarding para operações que utilizam resultados durante o estágio EX. Observe que o campo EX/MEM.RegistradorRd é o destino do registrador para uma instrução ALU (que vem do campo Rd da instrução) ou um load (que vem do campo Rt).



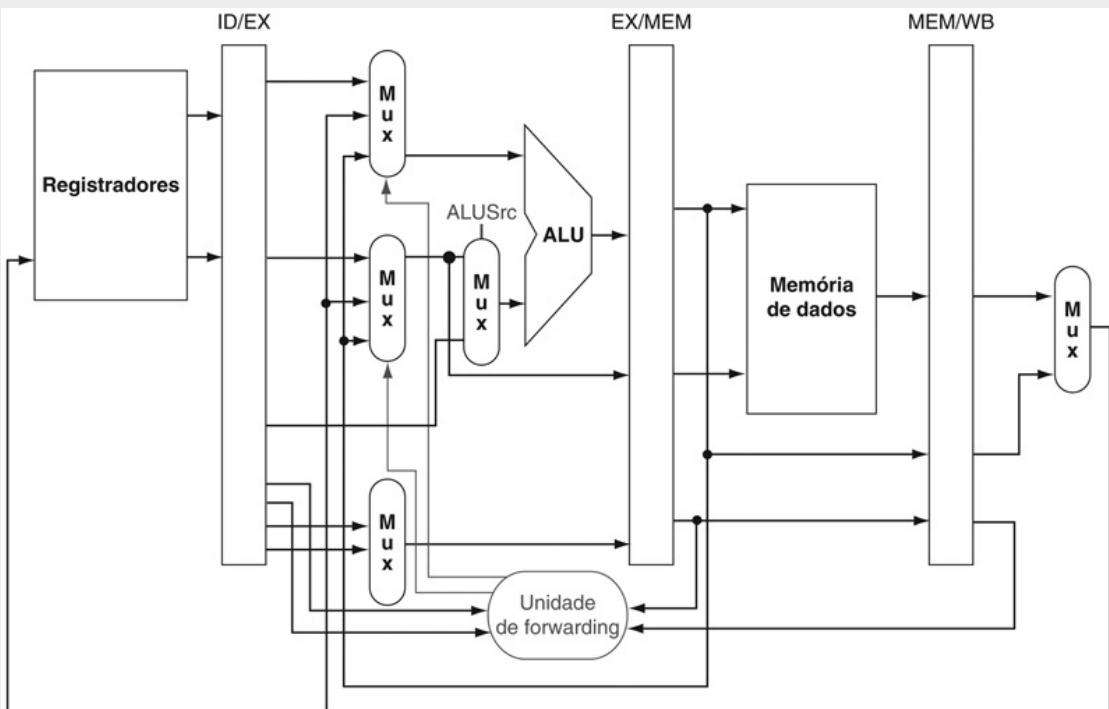
**FIGURA 4.56** O caminho de dados modificado para resolver os hazards via forwarding.

Em comparação com o caminho de dados da [Figura 4.51](#), os acréscimos são os multiplexadores para as entradas da ALU. Contudo, essa figura é um desenho mais estilizado, omitindo detalhes do caminho de dados completo, como o hardware de desvio e o hardware de extensão de sinal.

## Detalhamento

O forwarding também pode ajudar com hazards quando instruções store dependem de outras instruções. Como elas utilizam apenas um valor de dados durante o estágio MEM, o forwarding é fácil. Mas considere os loads imediatamente seguidos por stores, útil quando se realiza cópias da memória para a memória na arquitetura MIPS. Como as cópias são frequentes, precisamos acrescentar mais hardware de forwarding, a fim de fazer com que as cópias de memória para memória se tornem mais rápidas. Se tivéssemos de redesenhar a Figura 4.53, substituindo as instruções sub e AND por lw e sw, veríamos que é possível evitar um stall, pois os dados existem no registrador MEM/WB de uma instrução load, em tempo para seu uso no estágio MEM de uma instrução store. Para essa opção, teríamos de acrescentar o forwarding para o estágio de acesso à memória. Deixamos essa modificação como um exercício para o leitor.

Além disso, a entrada imediata com sinal para a ALU, necessária para loads e stores, não existe no caminho de dados da Figura 4.56. Como o controle central decide entre registrador e imediato, e como a unidade de forwarding escolhe o registrador de pipeline para uma entrada de registrador para a ALU, a solução mais fácil é acrescentar um multiplexador 2:1 que escolha entre a saída do multiplexador ForwardB e o imediato com sinal. A Figura 4.57 mostra esse acréscimo.



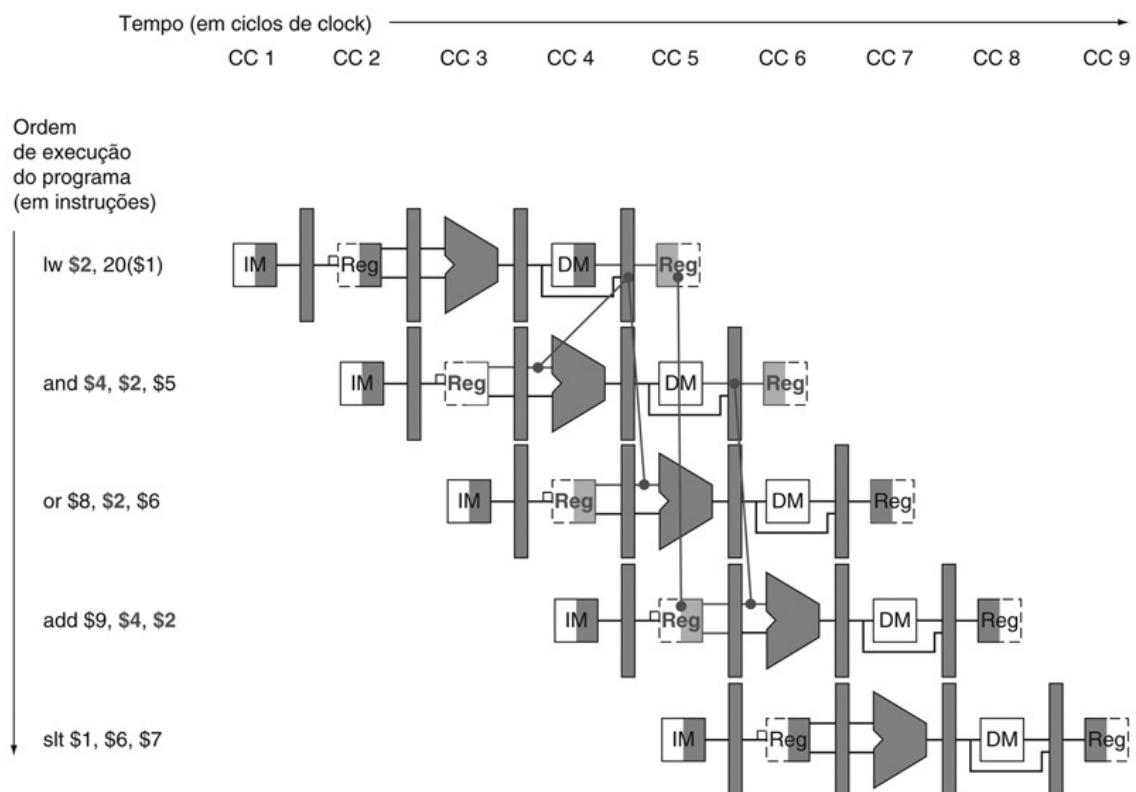
**FIGURA 4.57** Uma visão de perto do caminho de dados da Figura 4.54 mostra um multiplexador 2:1, que foi acrescentado para selecionar o imediato com sinal como uma entrada para a ALU

## Hazards de dados e stalls

*Se a princípio você não obteve sucesso, redefina sucesso.*

*Anônimo*

Conforme dissemos na [Seção 4.5](#), um caso em que o forwarding não pode salvar o dia é quando uma instrução tenta ler um registrador após uma instrução load que escreve no mesmo registrador. A [Figura 4.58](#) ilustra o problema. Os dados ainda são lidos da memória no ciclo de clock 4, enquanto a ALU está realizando a operação para a instrução seguinte. Algo precisa ocasionar um stall no pipeline para a combinação de load seguida por uma instrução que lê seu resultado.



**FIGURA 4.58** Uma sequência de instruções em pipeline.

Como a dependência entre o load e a instrução seguinte (and) recua no tempo, esse hazard não pode ser resolvido pelo forwarding. Logo, essa combinação precisa resultar em um stall pela unidade de detecção de hazard.

Logo, além de uma unidade de forwarding, precisamos de uma *unidade de detecção de hazard*. Ela opera durante o estágio ID, de modo que pode inserir o stall entre o load e seu uso. Verificando as instruções load, o controle para a unidade de detecção de hazard é esta condição única:

```

if (ID/EX.ReadMem and
((ID/EX.RegistradorRt = IF/ID.RegistradorRs) or
(ID/EX.RegistradorRt = IF/ID.RegistradorRt)))
ocasiona stall no pipeline

```

A primeira linha testa se a instrução é um load: a única instrução que lê a memória de dados é um load. As duas linhas seguintes verificam se o campo do registrador destino do load no estágio EX combina com qualquer registrador origem da instrução no estágio ID. Se a condição permanecer, a instrução ocasiona um stall de um ciclo de clock no pipeline. Depois desse stall de um ciclo, a lógica de forwarding pode lidar com a dependência e a execução prossegue. (Se não houvesse forwarding, então as instruções na [Figura 4.58](#) precisariam de outro ciclo de stall.)

Se a instrução no estágio ID sofrer um stall, então a instrução no estágio IF também precisa sofrer; caso contrário, perderíamos a instrução lida da memória. Evitar que essas duas instruções tenham progresso é algo feito simplesmente impedindo-se que o registrador PC e o registrador de pipeline IF/ID sejam alterados. Desde que esses registradores sejam preservados, a instrução no estágio IF continuará a ser lida usando o mesmo PC, e os registradores no estágio ID continuarão a ser lidos usando os mesmos campos de instrução no registrador de pipeline IF/ID. Retornando à nossa analogia favorita, é como se você reiniciasse a lavadora com as mesmas roupas e deixasse a secadora continuar a trabalhar vazia. Naturalmente, assim como a secadora, a metade do pipeline que começa com o estágio EX precisa estar fazendo algo; o que ela está fazendo é executar instruções que não têm efeito algum: **nops**.

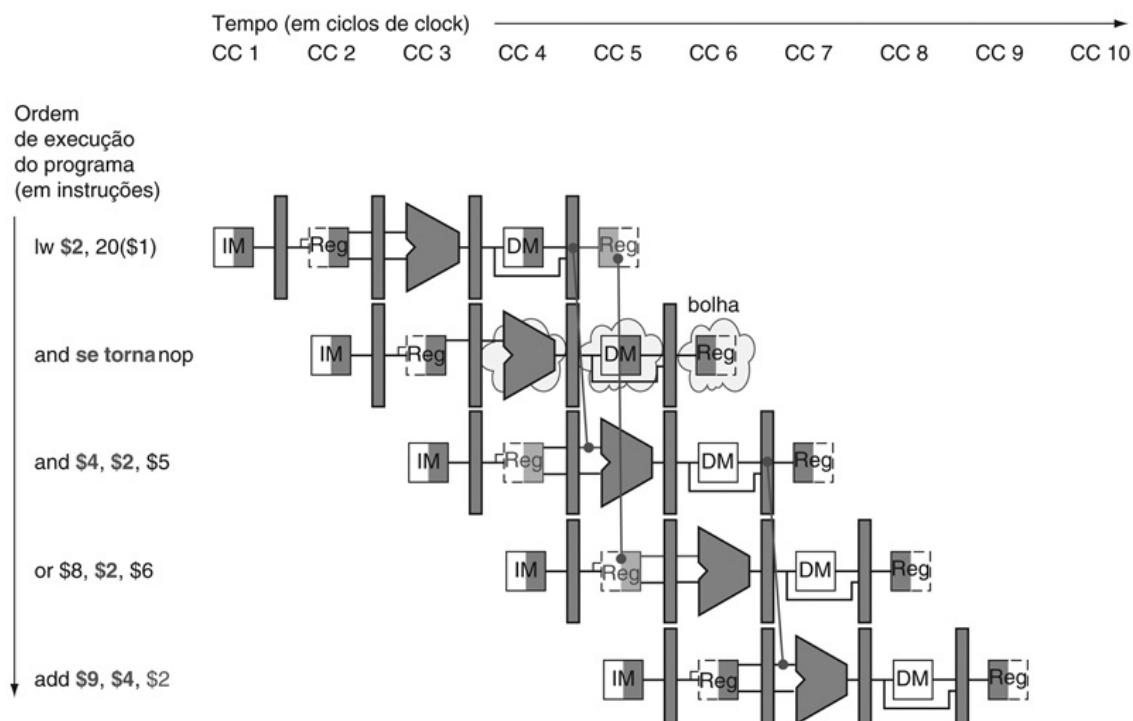
## nop

Uma instrução que não realiza operação para mudar de estado.

Como podemos inserir esses nops, que atuam como bolhas, no pipeline? Na [Figura 4.49](#), vimos que a desativação de todos os nove sinais de controle (colocando-os em 0) nos estágios EX, MEM e WB criará uma instrução que “não faz nada” ou nop. Identificando o hazard no estágio ID, podemos inserir uma bolha no pipeline alterando os campos de controle EX, MEM e WB do registrador de pipeline ID/EX para 0. Esses valores de controle benignos são filtrados adiante em cada ciclo de clock com o efeito correto: nenhum registrador

ou memória serão modificados se os valores forem todos 0.

A Figura 4.59 mostra o que realmente acontece no hardware: o slot de execução do pipeline associado com a instrução AND transforma-se em um nop, e todas as instruções começando com a instrução AND são atrasadas um ciclo. Assim como uma bolha de ar em um cano de água, uma bolha de stall retarda tudo o que está atrás dela e prossegue pelo pipe de instruções um estágio a cada ciclo, até que saia no final. Neste exemplo, o hazard força as instruções AND e OR a repetir no ciclo de clock 4 o que fizeram no ciclo de clock 3: AND lê registradores e decodifica, e OR é apanhado novamente da memória de instruções. Esse trabalho repetido é um stall, mas seu efeito é esticar o tempo das instruções AND e OR e atrasar a busca da instrução add.

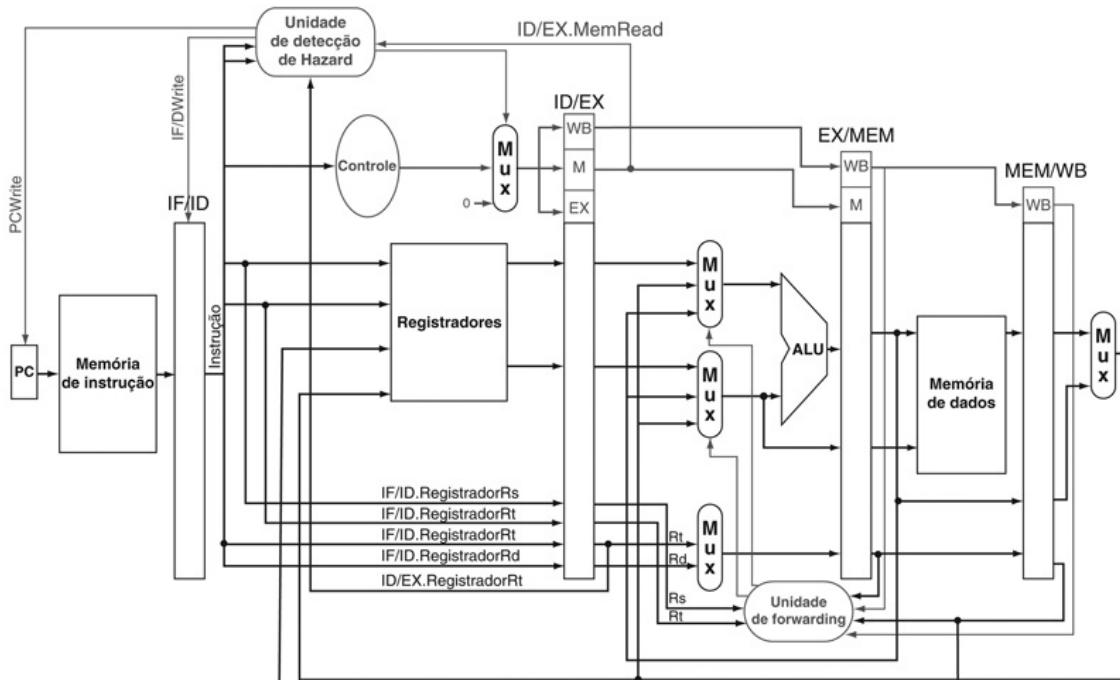


**FIGURA 4.59** O modo como os stalls são realmente inseridos no pipeline.

Uma bolha é inserida a partir do ciclo de clock 4, alterando a instrução and para um nop. Observe que a instrução and na realidade é buscada e decodificada nos ciclos de clock 2 e 3, mas seu estágio EX é atrasado até o ciclo de clock 5 (ao contrário da posição sem stall no ciclo de clock 4). Da mesma forma, a instrução or é apanhada no ciclo de clock 3, mas seu estágio ID é atrasado até o ciclo de clock 5 (ao contrário da

posição não atrasada no ciclo de clock 4). Após a inserção da bolha, todas as dependências seguem à frente no tempo, e nenhum outro hazard acontece.

A Figura 4.60 destaca as conexões do pipeline para a unidade de detecção de hazard e a unidade de forwarding. Como antes, a unidade de forwarding controla os multiplexadores da ALU, a fim de substituir o valor de um registrador de uso geral pelo valor do registrador de pipeline apropriado. A unidade de detecção de hazard controla a escrita dos registradores PC e IF/ID mais o multiplexador que escolhe entre os valores de controle reais e 0s. A unidade de detecção de hazard insere um stall e desativa os campos de controle se o teste de hazard do uso do load for verdadeiro.



**FIGURA 4.60** Visão geral do controle em pipeline, mostrando os dois multiplexadores para forwarding, a unidade de detecção de hazard e a unidade de forwarding.

Embora os estágios ID e EX tenham sido simplificados — a lógica de extensão de sinal imediato e de desvio estão faltando —, este desenho mostra a essência dos requisitos do hardware de forwarding.

## Colocando em perspectiva

Embora o compilador geralmente conte com o hardware para resolver dependências de hazard e garantir assim a execução correta, o compilador precisa compreender o pipeline, a fim de alcançar o melhor desempenho. Caso contrário, stalls inesperados reduzirão o desempenho do código compilado.

## Detalhamento

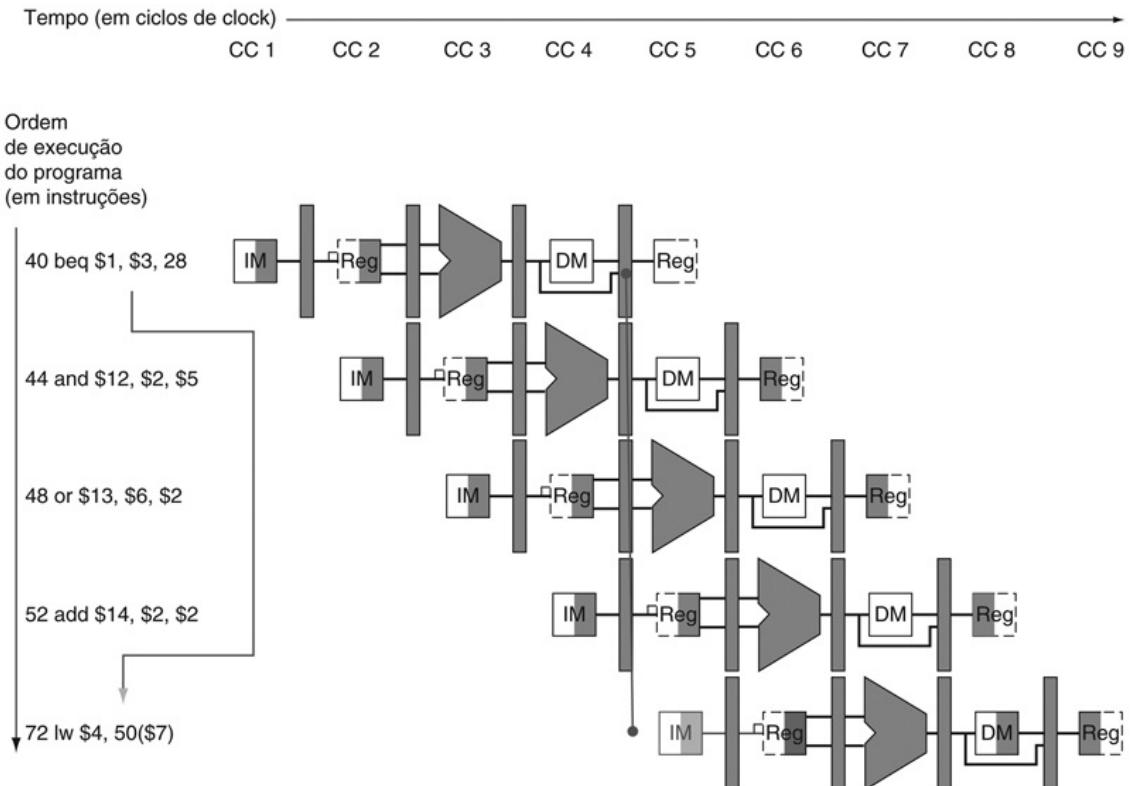
Com relação ao comentário anterior sobre a colocação das linhas de controle em 0 para evitar a escrita de registradores ou memória: somente os sinais WriteReg e WriteMem precisam ser 0, enquanto os outros sinais de controle podem ser “don’t care”.

## 4.8. Hazards de controle

*Para cada mal que está batendo na raiz há milhares pendurados nos galhos.*

*Henry David Thoreau, Walden, 1854*

Até aqui, preocupamo-nos apenas com os hazards envolvendo operações aritméticas e transferências de dados. Entretanto, como vimos na [Seção 4.5](#), também existem hazards de pipeline envolvendo desvios. A [Figura 4.61](#) mostra uma sequência de instruções e indica quando o desvio ocorreria nesse pipeline. Uma instrução precisa ser buscada a cada ciclo de clock para sustentar o pipeline, embora, em nosso projeto, a decisão sobre o desvio não ocorra até o estágio MEM do pipeline. Conforme mencionamos na [Seção 4.5](#), esse atraso para determinar a instrução própria a ser buscada é chamado de *hazard de controle* ou *hazard de desvio*, ao contrário dos *hazards de dados*, que acabamos de examinar.

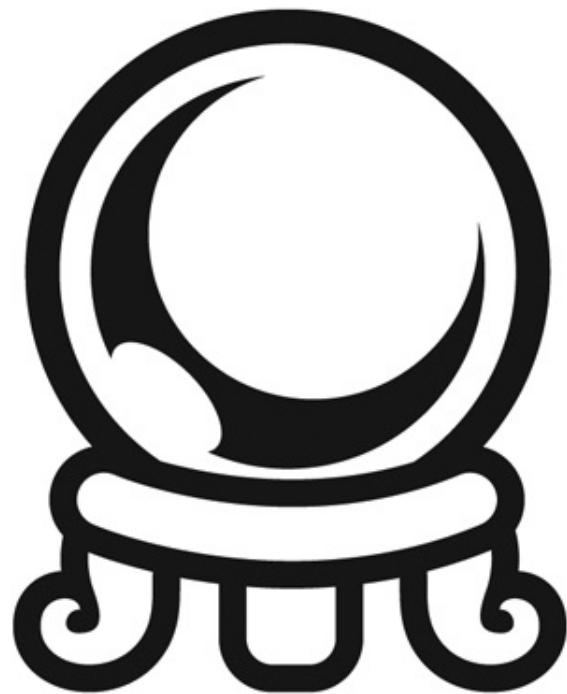


**FIGURA 4.61** O impacto do pipeline sobre a instrução branch.

Os números à esquerda da instrução (40, 44, ...) são os endereços das instruções. Como a instrução branch decide se deve desviar no estágio MEM — ciclo de clock 4 para a instrução beq, anterior —, as três instruções sequenciais que seguem o branch serão buscadas e iniciarão sua execução. Sem intervenção, essas três instruções seguintes começarão a executar antes que o beq desvie para lw na posição 72. (A Figura 4.31 considerou um hardware extra para reduzir o hazard de controle a um ciclo de clock; esta figura usa o caminho de dados não otimizado.)

Esta seção sobre hazards de controle é mais curta do que as seções anteriores, sobre hazards de dados, porque os hazards de controle são relativamente simples de entender, e ocorrem com menos frequência que os hazards de dados. Além disso, não há nada tão eficiente contra os hazards de controle como o forwarding é contra os hazards de dados. Logo, usamos esquemas mais simples. Veremos dois esquemas para resolver os hazards de controle e uma otimização para melhorar esses esquemas.

## Considere que o desvio não foi tomado



## P R E D I Ç Ã O

Como vimos na [Seção 4.5](#), fazer um stall até que o desvio termine é muito lento. Uma melhoria comum ao stall do desvio é **prever** que o desvio não será tomado e, portanto, continuar no fluxo sequencial das instruções. Se o desvio for tomado, as instruções que estão sendo buscadas e decodificadas precisam ser descartadas. A execução continua no destino do desvio. Se os desvios não são tomados na metade das vezes, e se custar pouco descartar as instruções, essa otimização reduz ao meio o custo dos hazards de controle.

Para descartar instruções, simplesmente alteramos os valores de controle para 0, assim como fizemos para o stall no hazard de dados no caso do load. A diferença é que também precisamos alterar as três instruções nos estágios IF, ID e EX quando o desvio atingir o estágio MEM; para os stalls no uso de load, simplesmente alteramos o controle para 0 no estágio ID e o deixamos prosseguir no pipeline. Descartar instruções, então, significa que precisamos ser capazes de dar **flush** nas instruções nos estágios IF, ID e EX do pipeline.

## flush

Descarte de instruções em um pipeline, normalmente devido a um evento inesperado.

## Reduzindo o atraso dos desvios

Uma forma de melhorar o desempenho do desvio é reduzir o custo do desvio tomado. Até aqui, consideramos que o próximo PC para um desvio é selecionado no estágio MEM, mas, se movermos a execução do desvio para um estágio anterior do pipeline, então menos instruções precisam sofrer flush. A arquitetura do MIPS foi criada para dar suporte a desvios rápidos de ciclo único, que poderiam passar pelo pipeline com uma pequena penalidade no desvio. Os projetistas observaram que muitos desvios contam apenas com testes simples (igualdade ou sinal, por exemplo) e que esses testes não exigem uma operação completa da ALU, mas podem ser feitos com, no máximo, algumas portas lógicas. Quando uma decisão de desvio mais complexa é exigida, a comparação realizada por uma ALU, através de instrução separada, é requisitada — uma situação semelhante ao uso de códigos de condição para os desvios ([Capítulo 2](#)).

Levar a decisão do desvio para cima exige que duas ações ocorram mais cedo: calcular o endereço de destino do desvio e avaliar a decisão do desvio. A parte fácil dessa mudança é subir com o cálculo do endereço de desvio. Já temos o valor do PC e o campo imediato no registrador de pipeline IF/ID, de modo que só movemos o somador do desvio do estágio EX para o estágio ID; naturalmente, o cálculo do endereço de destino do desvio será realizado para todas as instruções, mas só será usado quando for necessário.

A parte mais difícil é a própria decisão do desvio. Para branch equal, compararíamos os dois registradores lidos durante o estágio ID para ver se são iguais. A igualdade pode ser testada, primeiro realizando um OR exclusivo de seus respectivos bits e, depois, um OR de todos os resultados. Mover o teste de desvio para o estágio ID implica hardware adicional de forwarding e detecção de hazard, visto que um desvio dependente de um resultado ainda no pipeline precisará funcionar corretamente com essa otimização. Por exemplo, a fim de implementar branch-on-equal (e seu inverso), teremos de fazer um forwarding dos resultados para a lógica do teste de igualdade que opera durante o estágio ID. Existem dois fatores que comprometem o procedimento:

1. Durante o estágio ID, temos de decodificar a instrução, decidir se é necessário um bypass para a unidade de igualdade e completar a

comparação de igualdade de modo que, se a instrução for um desvio, possamos atribuir ao PC o endereço de destino do desvio. O forwarding para os operandos dos desvios foi tratado anteriormente pela lógica de forwarding da ALU, mas a introdução da unidade de teste de igualdade no estágio ID exigirá nova lógica de forwarding. Observe que os operandos-fonte de um desvio que sofreram bypass podem vir dos latches do pipeline ALU/MEM ou MEM/WB.

2. Como os valores em uma comparação de desvio são necessários durante o estágio ID, mas podem ser produzidos mais adiante no tempo, é possível que ocorra um hazard de dados e um stall seja necessário. Por exemplo, se uma instrução da ALU imediatamente antes de um desvio produz um dos operandos para a comparação no desvio, um stall será exigido, já que o estágio EX para a instrução da ALU ocorrerá depois do ciclo de ID do desvio. Por extensão, se um load for imediatamente seguido por um desvio condicional que está no resultado do load, dois ciclos de stall serão necessários, pois o resultado do load aparece no final do ciclo MEM, mas é necessário no início do ID do desvio.

Apesar dessas dificuldades, mover a execução do desvio para o estágio ID é uma melhoria, pois reduz a penalidade de um desvio a apenas uma instrução se o desvio for tomado, a saber, aquela sendo buscada atualmente. Os exercícios exploram os detalhes da implementação do caminho de forwarding e a detecção do hazard.

Para fazer um flush das instruções no estágio IF, acrescentamos uma linha de controle, chamada IF.Flush, que zera o campo de instrução do registrador de pipeline IF/ID. Apagar o registrador transforma a instrução buscada em um nop, uma instrução que não possui ação e não muda estado algum.

## Desvios no pipeline

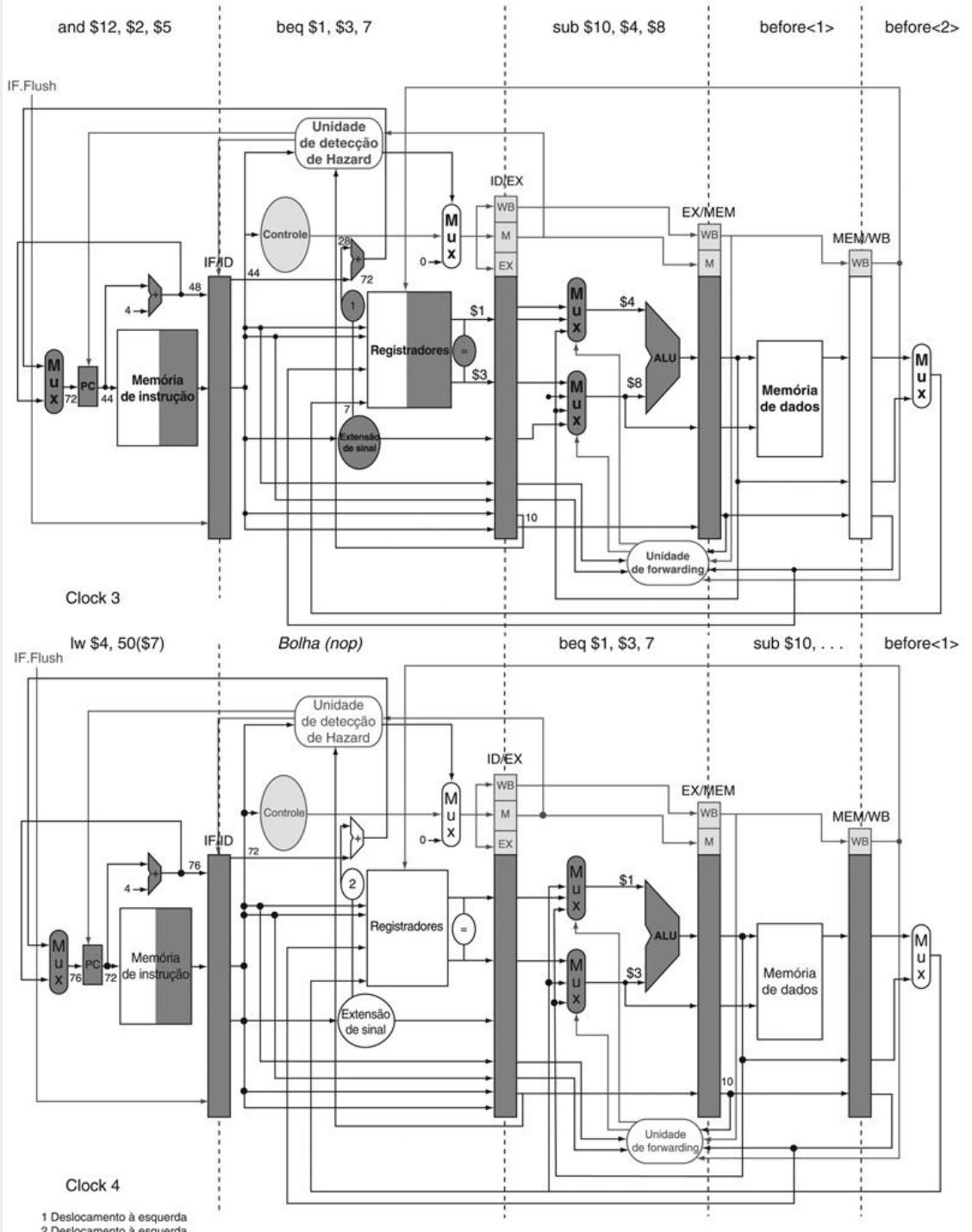
### Exemplo

Mostre o que acontece quando o desvio é tomado nesta sequência de instruções, considerando que o pipeline está otimizado para desvios que não são tomados e que movemos a execução do desvio para o estágio ID:

```
36 sub $10, $4, $8
40 beq $1, $3, 7 # desvio relativo ao PC para 40+4+7*4 = 72
44 and $12, $2, $5
48 or $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. .
72 lw $4, 50($7)
```

## Resposta

A Figura 4.62 mostra o que acontece quando um desvio é tomado. Diferente da Figura 4.61, há somente uma bolha no pipeline para o desvio tomado.

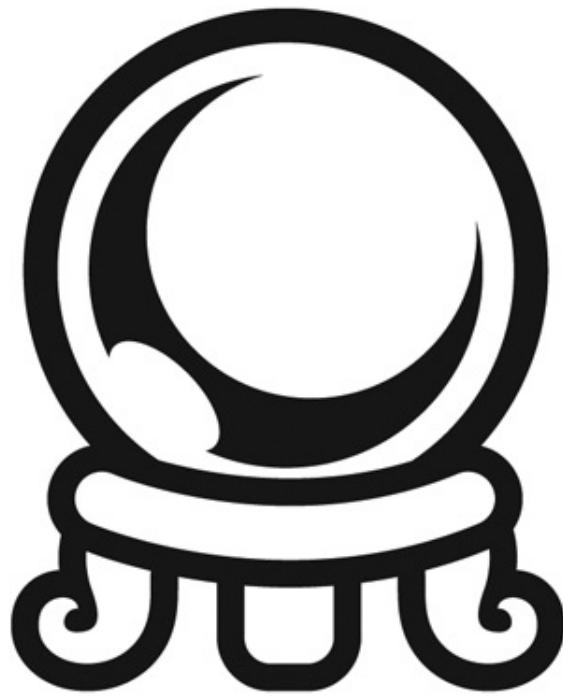


resultado do desvio tomado. (Como o `nop` na realidade é `s11 $0,$0,0`, é discutível se o estágio ID no clock 4 deve ou não ser destacado.)

## Previsão dinâmica de desvios

Supor que um desvio não seja tomado é uma forma simples de *previsão de desvios*. Nesse caso, prevemos que os desvios não são tomados, fazendo um flush no pipeline quando estivermos errados. Para o pipeline simples, com cinco estágios, essa técnica, possivelmente acoplada com a previsão baseada no compilador, deverá ser adequada. Com pipelines mais profundos, a penalidade do desvio aumenta quando medida em ciclos de clock. Da mesma forma, com a questão múltipla ([Seção 4.10](#)), a penalidade do desvio aumenta em termos de instruções perdidas. Essa combinação significa que, em um pipeline agressivo, um esquema de previsão estática provavelmente desperdiçará muito desempenho. Como mencionamos na [Seção 4.5](#), com mais hardware, é possível tentar **prever** o comportamento do desvio durante a execução do programa.

Uma técnica é pesquisar o endereço da instrução para ver se um desvio foi tomado na última vez que essa instrução foi executada e, se foi, começar a buscar novas instruções a partir do mesmo lugar da última vez. Essa técnica é chamada **previsão dinâmica de desvios**.



## P R E D I Ç Ã O

### previsão dinâmica de desvios

Previsão de desvios durante a execução, usando informações em tempo de execução.

Uma implementação dessa técnica é um **buffer de previsão de desvios** ou **tabela de histórico de desvios**. Um buffer de previsão de desvios é uma pequena memória indexada pela parte menos significativa do endereço da instrução de desvio. A memória contém um bit que diz se o desvio foi tomado recentemente ou não.

### buffer de previsão de desvios

Também chamado **tabela de histórico de desvios**. Uma pequena memória indexada pela parte menos significativa do endereço da instrução de desvio e que contém um ou mais bits indicando se o desvio foi tomado recentemente ou não.

Esse é o tipo de buffer mais simples; na verdade, não sabemos se a previsão é a correta — ela pode ter sido colocada lá por outro desvio, que tem os mesmos bits de endereço menos significativos. Mas isso não afeta a exatidão. A previsão é apenas um palpite considerado correto, de modo que a busca começa na direção prevista. Se o palpite estiver errado, as instruções previstas incorretamente são excluídas, o bit de previsão é invertido e armazenado de volta, e a sequência apropriada é buscada e executada.

Esse esquema simples de previsão de 1 bit tem um problema de desempenho: mesmo que um desvio quase sempre seja tomado, provavelmente faremos uma previsão incorreta duas vezes, em vez de uma, quando ele não for tomado. O exemplo a seguir mostra esse dilema.

## Loops e previsão

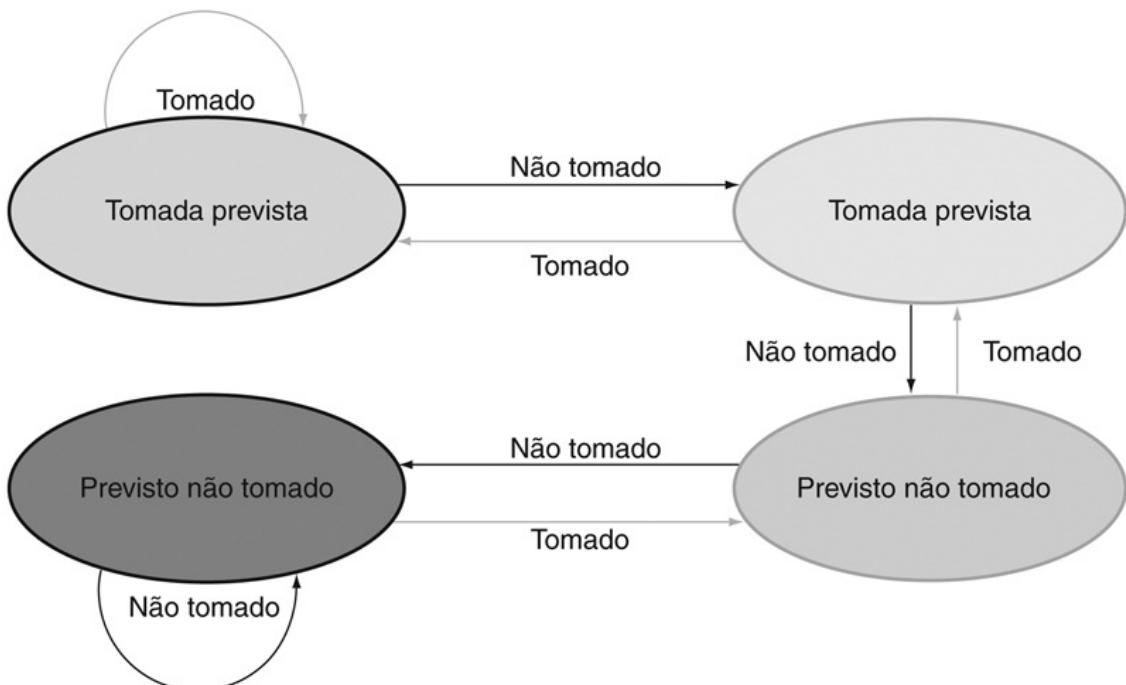
### Exemplo

Considere um desvio de loop que se desvia nove vezes seguidas, depois não é tomado uma vez. Qual é a exatidão da previsão para esse desvio, supondo que o bit de previsão para o desvio permaneça no buffer de previsão?

### Resposta

O comportamento da previsão de estado fixo fará uma previsão errada na primeira e última iterações do loop. O erro de previsão na última iteração é inevitável, pois o bit de previsão dirá “tomado”, já que o desvio foi tomado nove vezes seguidas nesse ponto. O erro de previsão na primeira iteração acontece porque o bit é invertido na execução anterior da última iteração do loop, pois o desvio não foi tomado nessa iteração final. Assim, a exatidão da previsão para esse desvio tomado 90% do tempo é apenas de 80% (duas previsões incorretas contra oito corretas).

O ideal é que a previsão do sistema combine com a frequência de desvio tomado para esses desvios altamente regulares. Para remediar esse ponto fraco, os esquemas de previsão de 2 bits são utilizados com frequência. Em um esquema de 2 bits, uma previsão precisa estar errada duas vezes antes de ser alterada. A [Figura 4.63](#) mostra a máquina de estados finitos para um esquema de previsão de 2 bits.



**FIGURA 4.63** Os estados em um esquema de previsão de 2 bits.

Usando 2 bits em vez de 1, um desvio que favoreça bastante a situação “tomado” ou “não tomado” — como muitos desvios fazem — será previsto incorretamente apenas uma vez. Os 2 bits são usados para codificar os quatro estados no sistema. O esquema de 2 bits é um caso geral de uma previsão baseada em contador, incrementado quando a previsão é exata e decrementado em caso contrário, utilizando o ponto intermediário desse intervalo como divisão entre desvio tomado e não tomado.

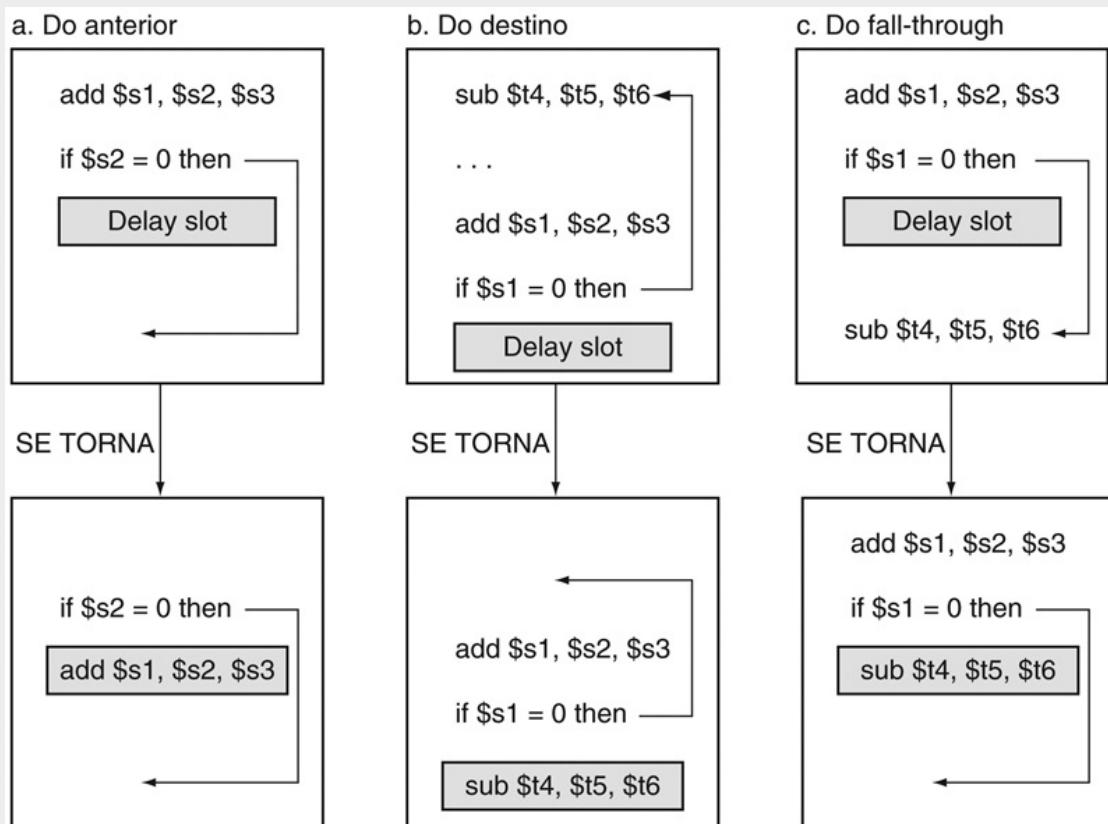
Um buffer de previsão de desvio pode ser implementado como um pequeno buffer especial, acessado com o endereço da instrução durante o estágio do pipe IF. Se a instrução for prevista como tomada, a busca começa a partir do destino assim que o PC for conhecido; conforme mencionamos anteriormente, isso pode ser até mesmo no estágio ID. Caso contrário, a busca e a execução sequencial continuam. Se a previsão for errada, os bits de previsão são trocados, como mostra a [Figura 4.63](#).

## Detalhamento

Conforme descrevemos na Seção 4.5, em um pipeline de cinco estágios, podemos tornar o hazard de controle em um recurso, redefinindo o desvio.

Um delayed branch sempre executa a seguinte instrução, mas a segunda instrução após o desvio será afetada pelo desvio.

Os compiladores e os montadores tentam colocar uma instrução que sempre executa após o desvio no **delay slot do desvio**. A tarefa do software é tornar as instruções sucessoras válidas e úteis. A Figura 4.64 mostra as três maneiras como o delay slot do desvio pode ser escalonado.



**FIGURA 4.64 Escalonando o delay slot do desvio.**

Em cada par de quadros, o quadro de cima mostra o código antes do escalonamento; o quadro de baixo mostra o código escalonado. Em (a), o delay slot é escalonado com uma instrução independente de antes do desvio. Essa é a melhor opção. As estratégias (b) e (c) são usadas quando (a) não é possível. Nas sequências de código para (b) e (c), o uso de  $\$s1$  na condição de desvio impede que a instrução `add` (cujo destino é  $\$s1$ ) seja movida para o delay slot do desvio. Em (b), o delay slot de desvio é escalonado a partir do destino do desvio; normalmente, a instrução de destino precisará ser copiada, pois pode ser alcançada por outro caminho. A

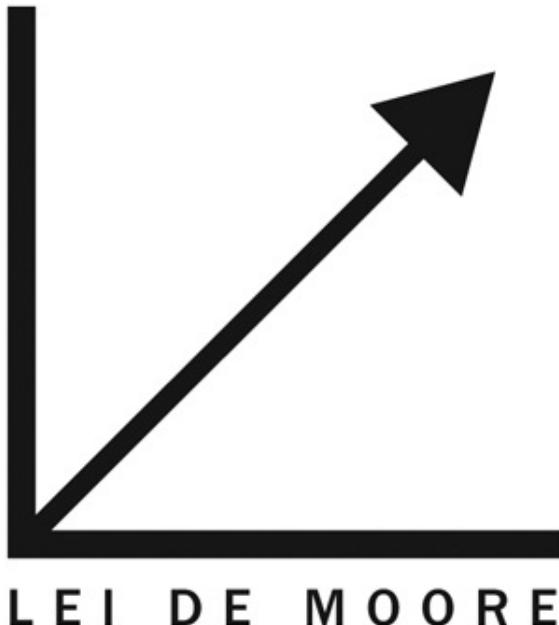
estratégia (b) é preferida quando o desvio é tomado com alta probabilidade, como em um desvio de loop. Finalmente, o desvio pode ser escalonado a partir da sequência não tomada, como em (c). Para tornar essa otimização válida para (b) ou (c), deve ser válido executar a instrução `sub` quando o desvio seguir na direção inesperada. Com “válido”, queremos dizer que o trabalho é desperdiçado, mas o programa ainda será executado corretamente. Esse é o caso, por exemplo, se `$t4` fosse um registrador temporário não utilizado quando o desvio entrasse na direção inesperada.

As limitações sobre o escalonamento com delayed branch surgem de (1) as restrições sobre as instruções escalonadas nos delay slots e (2) nossa capacidade de prever durante a compilação se um desvio provavelmente será tomado ou não.

O delayed branch foi uma solução simples e eficaz para um pipeline de cinco estágios despachando uma instrução a cada ciclo de clock. À medida que os processadores utilizam pipelines maiores, despachando múltiplas instruções por ciclo de clock (Seção 4.10), o atraso do desvio torna-se maior e um único delay slot é insuficiente. Logo, o delayed branch perdeu popularidade em comparação com as técnicas dinâmicas mais dispendiosas, porém mais flexíveis. Simultaneamente, o crescimento em transistores disponíveis por chip, devido à **Lei de Moore**, tornou a previsão dinâmica relativamente mais barata.

## delay slot do desvio

O slot diretamente após a instrução de delayed branch, que na arquitetura MIPS é preenchido por uma instrução que não afeta o desvio.



## Detalhamento

Um previsor de desvios nos diz se um desvio é tomado ou não, mas ainda exige o cálculo do destino do desvio. No pipeline de cinco estágios, esse cálculo leva um ciclo, significando que os desvios tomados terão uma penalidade de um ciclo. Os delayed branches são uma técnica para eliminar essa penalidade. Outra técnica é usar uma cache para manter o contador de programa de destino ou instrução de destino, usando um **buffer de destino de desvios**.

O esquema de previsão dinâmica de 2 bits usa apenas informações sobre um determinado desvio. Os pesquisadores notaram que o uso de informações sobre um desvio local e um comportamento global de desvios executados recentemente, juntos, geram maior exatidão da previsão para o mesmo número de bits de previsão. Essas técnicas são chamadas de previsor correlato. Um **previsor correlato** simples poderia ter dois previsores de 2 bits para cada desvio, com a escolha entre os previsões feita com base em se o último desvio executado foi tomado ou não. Assim, o comportamento de desvio global pode ser imaginado como acrescentando bits de índice adicionais para a previsão.

Uma inovação mais recente na previsão de desvios é o uso de previsões de torneio. Um **previsor de torneio** utiliza vários previsores, rastreando, para

cada desvio, qual previsor gera os melhores resultados. Um previsor de torneio típico poderia conter duas previsões para cada índice de desvio: uma baseada em informações locais e uma baseada no comportamento do desvio global. Um seletor escolheria qual previsor usar para qualquer previsão dada. O seletor pode operar semelhantemente a um previsor de 1 ou 2 bits, favorecendo qualquer um dos dois previsores que tenha sido mais preciso. Muitos microprocessadores avançados mais recentes utilizam esses previsores rebuscados.

## buffer de destino de desvios

Uma estrutura que coloca em cache o PC de destino ou a instrução de destino para um desvio. Ele normalmente é organizado como uma cache com tags, tornando-o mais dispendioso do que um buffer de previsão simples.

## previsor correlato

Um previsor de desvio que combina o comportamento local de determinado desvio e informações globais sobre o comportamento de algum número recente de desvios executados.

## previsor de desvio de torneio

Um previsor de desvios com múltiplas previsões para cada desvio e um mecanismo de seleção que escolhe qual previsor deve ser usado para determinado desvio

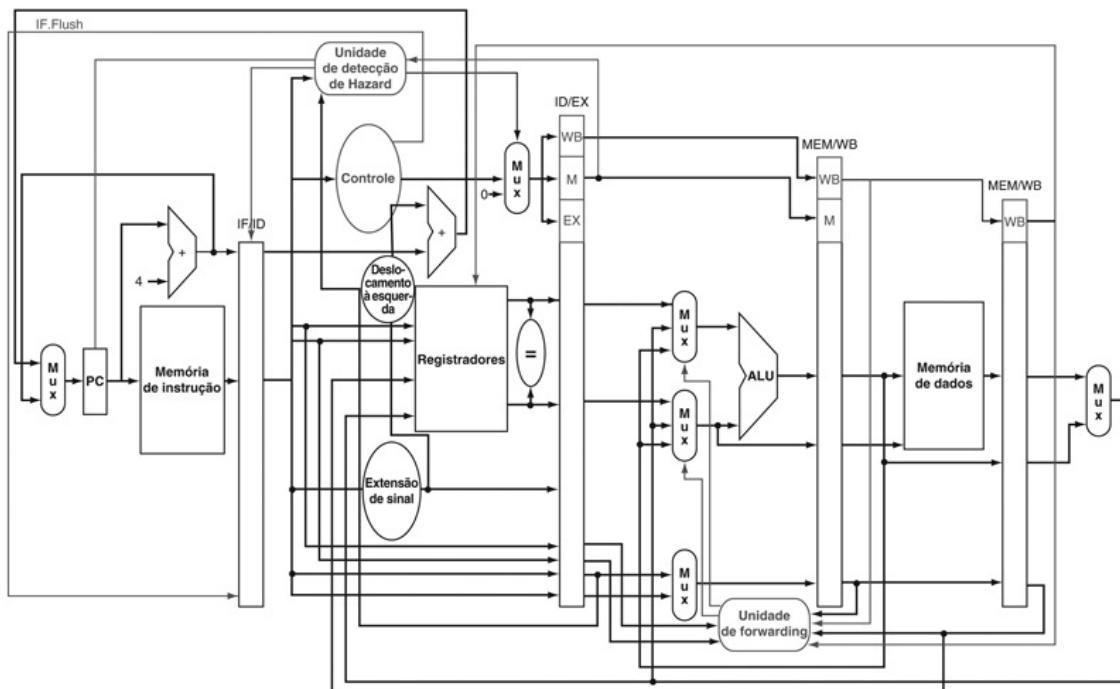
## Detalhamento

Uma maneira de reduzir o número de desvios condicionais é acrescentar instruções de *move condicional*. Em vez de mudar o PC com um desvio condicional, a instrução muda condicionalmente o registrador de destino do move. Se a condição falha, o move atua como um nop. Por exemplo, uma versão da arquitetura do conjunto de instruções MIPS tem duas novas instruções chamadas `movn` (move if not zero) e `movz` (move if zero). Assim, `movn $8,$11,$4` copia o conteúdo do registrador 11 para o registrador 8, desde que o valor no registrador 4 seja diferente de zero; caso contrário, ela não faz nada.

O conjunto de instruções ARMv7 tem um campo de condição na maioria das instruções. Assim, os programas ARM poderiam ter menos desvios condicionais que os programas MIPS.

## Resumo sobre pipeline

Começamos na lavanderia, mostrando princípios de pipelining em um ambiente cotidiano. Usando essa analogia como um guia, explicamos o pipelining de instruções passo a passo, começando com um caminho de dados de ciclo único e depois acrescentando registradores de pipeline, caminhos de forwarding, detecção de hazard de dados, previsão de desvio e com flushing de instruções em exceções. A [Figura 4.65](#) mostra o caminho de dados e controle final para este capítulo.



**FIGURA 4.65** O caminho de dados e controle final para este capítulo.

Observe que essa é uma figura estilizada, em vez de um caminho de dados detalhado, de modo que não contém o mux ALUScr da [Figura 4.57](#) e os controles multiplexadores da [Figura 4.51](#).

## Verifique você mesmo

Considere três esquemas de previsão de desvios: desvio não tomado, previsão tomada e previsão dinâmica. Suponha que todos eles tenham penalidade zero quando preveem corretamente e 2 ciclos quando estão errados. Suponha que a exatidão média da previsão do previsor dinâmico seja de 90%. Qual previsor é a melhor escolha para os seguintes desvios?

1. Um desvio tomado com frequência de 5%.
2. Um desvio tomado com frequência de 95%.
3. Um desvio tomado com frequência de 70%.

## 4.9. Exceções

*Fazer um computador com facilidades automáticas de interrupção de programa se comportar [sequencialmente] não foi uma tarefa fácil, pois o número de instruções em diversos estágios do processamento, quando ocorre um sinal de interrupção, pode ser muito grande.*

*Fred Brooks Jr., Planning a Computer System: Project Stretch, 1962*

Controle é o aspecto mais desafiador do projeto do processador: ele é a parte mais difícil de se acertar e a parte mais difícil de tornar mais rápida. Uma das partes mais difíceis do controle é implementar **exceções** e **interrupções** — eventos diferentes dos desvios ou saltos, que mudam o fluxo normal da execução da instrução. Eles foram criados inicialmente para tratar de eventos inesperados de dentro do processador, como o overflow aritmético. O mesmo mecanismo básico foi estendido para os dispositivos de E/S se comunicarem com o processador, conforme veremos no [Capítulo 5](#).

### exceção

Também chamada **interrupção**. Um evento não programado que interrompe a execução do programa; usada para detectar overflow.

### interrupção

Uma exceção que vem de fora do processador. (Algumas arquiteturas utilizam

o termo *interrupção* para todas as exceções.)

Muitas arquiteturas e autores não fazem distinção entre interrupções e exceções, normalmente usando o nome mais antigo *interrupção* para se referirem aos dois tipos de eventos. Por exemplo, o Intel x86 usa interrupção. Seguimos a convenção do MIPS, usando o termo *exceção* para indicar *qualquer* mudança inesperada no fluxo de controle, sem distinguir se a causa é interna ou externa; usamos o termo *interrupção* apenas quando o evento é causado externamente. Aqui estão alguns exemplos mostrando se a situação é gerada internamente pelo processador ou se é gerada externamente:

Tipo de evento	De onde?	Terminologia MIPS
Solicitação de dispositivo de E/S	Externa	Interrupção
Chamar o sistema operacional do programa do usuário	Interna	Exceção
Overflow aritmético	Interna	Exceção
Usar uma instrução indefinida	Interna	Exceção
Defeitos do hardware	Ambos	Exceção ou interrupção

Muitos dos requisitos para dar suporte a exceções vêm da situação específica que causa a ocorrência de uma exceção. Consequentemente, retornaremos a esse assunto no [Capítulo 5](#), quando entenderemos melhor a motivação para as capacidades adicionais no mecanismo de exceção. Nesta seção, lidamos com a implementação de controle de modo a detectar dois tipos de exceções que surgem das partes do conjunto de instruções e da implementação que já discutimos.

Detectar condições excepcionais e tomar a ação apropriada normalmente está no percurso de temporização crítico de um processador, que determina o tempo de ciclo de clock e, portanto, o desempenho. Sem a devida atenção às exceções durante o projeto da unidade de controle, as tentativas de acrescentar exceções a uma implementação complicada podem reduzir o desempenho significativamente, bem como complicar a tarefa de corrigir o projeto.

## Como as exceções são tratadas em uma arquitetura MIPS

Os dois tipos de exceções que nossa implementação atual pode gerar são: a execução de uma instrução indefinida e um overflow aritmético. Usaremos o

overflow aritmético na instrução `add $1,$2,$1` como exemplo de exceção nas próximas páginas. A ação básica que o processador deve realizar quando ocorre uma exceção é salvar o endereço da instrução causadora no *contador de programa de exceção* (Exception Program Counter — EPC) e depois transferir o controle para o sistema operacional em algum endereço especificado.

O sistema operacional pode então tomar a ação apropriada, que pode ser fornecer algum serviço ao programa do usuário, tomar alguma ação predefinida em resposta a um overflow ou terminar a execução do programa e informar um erro. Depois de realizar qualquer ação necessária devido à exceção, o sistema operacional pode terminar o programa ou pode continuar sua execução, usando o EPC para determinar onde reiniciar a execução do programa. No [Capítulo 5](#), veremos mais de perto a questão da retomada da execução.

Para o sistema operacional tratar da exceção, ele precisa conhecer o motivo da exceção, além da instrução que a causou. Existem dois métodos principais usados para comunicar o motivo de uma exceção. O método da arquitetura MIPS inclui um registrador de status (chamado *registrador Cause*), que mantém um campo que indica o motivo da exceção.

Um segundo método usa **interrupções vetorizadas**. Em uma interrupção vetorizada, o endereço ao qual o controle é transferido é determinado pela causa da exceção. Por exemplo, para acomodar os dois tipos de exceção listados anteriormente, poderíamos definir os dois endereços de vetor de exceção a seguir:

## interrupção vetorizada

Uma interrupção para a qual o endereço para onde o controle é transferido é determinado pela causa da exceção.

Tipo de exceção	Endereço do vetor de exceção (em hexa)
Instrução indefinida	8000 0000 <sub>hexa</sub>
Overflow aritmético	8000 0180 <sub>hexa</sub>

O sistema operacional sabe o motivo para a exceção pelo endereço em que ela é iniciada. Os endereços são separados por 32 bytes ou oito instruções, e o sistema operacional precisa registrar o motivo para a exceção e pode realizar algum processamento limitado nessa sequência. Quando a exceção não é vetorizada, um único ponto de entrada para todas as exceções pode ser utilizado,

e o sistema operacional decodifica o registrador de status para encontrar a causa.

Podemos realizar o processamento exigido para exceções acrescentando alguns registradores e sinais de controle extras à nossa implementação básica e estendendo o controle ligeiramente. Vamos supor que estejamos implementando o sistema de exceção utilizado na arquitetura MIPS, com o único ponto de entrada sendo o endereço 8000 0180<sub>hexa</sub>. (A implementação de exceções vetorizadas não é mais difícil.) Precisaremos acrescentar dois registradores adicionais à nossa implementação MIPS atual:

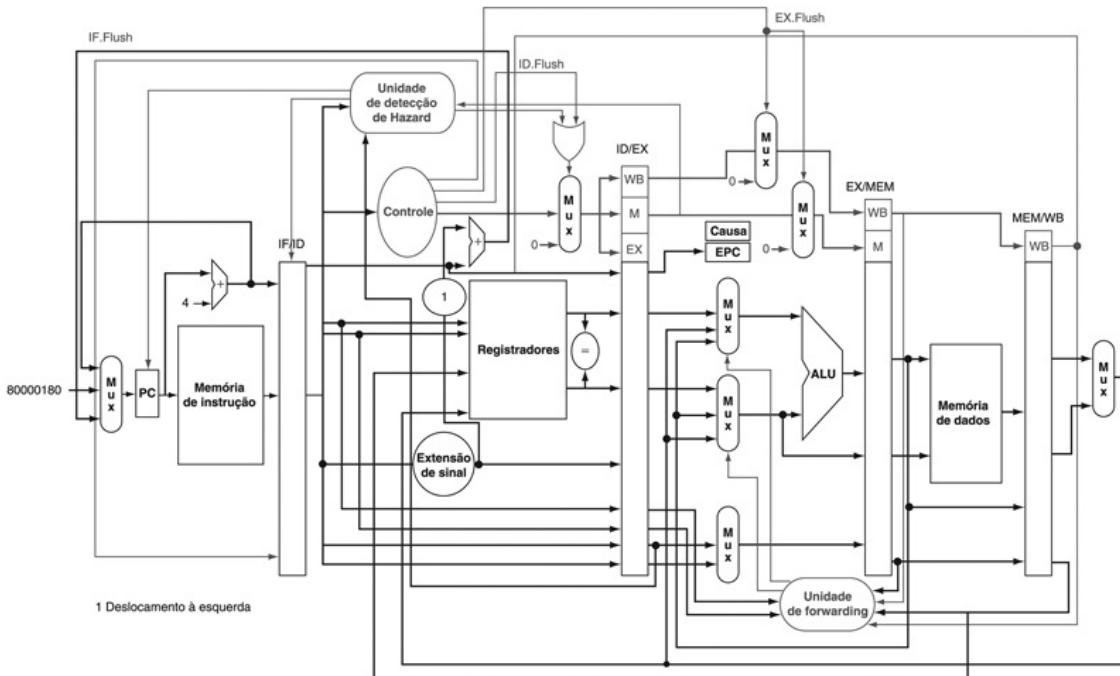
- *EPC*: Um registrador de 32 bits usado para manter o endereço da instrução afetada. (Esse registrador é necessário mesmo quando as exceções são vetorizadas.)
- *Cause*: Um registrador usado para registrar a causa da exceção. Na arquitetura MIPS, esse registrador tem 32 bits, embora alguns bits atualmente não sejam utilizados. Suponha que haja um campo de cinco bits que codifica as duas fontes de informação possíveis mencionadas anteriormente, com 10 representando uma instrução indefinida e 12 representando o overflow aritmético.

## Exceções em uma implementação em pipeline

Uma implementação em pipeline trata exceções como outra forma de hazard de controle. Por exemplo, suponha que haja um overflow aritmético em uma instrução add. Assim como fizemos para o desvio tomado na seção anterior, temos de dar flush nas instruções que vêm após a instrução add do pipeline e começar a buscar instruções do novo endereço. Usaremos o mesmo mecanismo que usamos para os desvios tomados, mas, desta vez, a exceção causa a desativação das linhas de controle.

Quando lidamos com um desvio mal previsto, vimos como dar flush na instrução no estágio IF, transformando-a em um nop. Para dar flush nas instruções no estágio ID, usamos o multiplexador já presente no estágio ID que zera os sinais de controle para stalls. Um novo sinal de controle, chamado ID.Flush, realiza um OR com o sinal de stall da unidade de detecção de hazards, a fim de dar flush durante o ID. Para dar flush na instrução na fase EX, usamos um novo sinal, chamado EX.Flush, fazendo com que novos multiplexadores zerem as linhas de controle. Para começar a buscar instruções do local 8000 0180<sub>hexa</sub>, que é o local da exceção para o overflow aritmético, simplesmente acrescentamos uma entrada adicional ao multiplexador do PC, que envia 8000

$0180_{\text{hexa}}$  ao PC. A Figura 4.66 mostra essas mudanças.



**FIGURA 4.66** O caminho de dados com controles para lidar com exceções.

Os principais acréscimos incluem uma nova entrada, com o valor  $8000\ 0180_{\text{hexa}}$ , no multiplexador que fornece o novo valor do PC; um registrador Cause para registrar a causa da exceção; e um registrador PC de Exceção (Exception Program Counter — EPC) para salvar o endereço da instrução que causou a exceção. A entrada  $8000\ 0180_{\text{hexa}}$  para o multiplexador é o endereço inicial para começar a buscar instruções no caso de uma exceção.

Embora não apareça, o sinal de overflow da ALU é uma entrada para a unidade de controle.

Este exemplo aponta um problema com as exceções: se não pararmos a execução no meio da instrução, o programador não poderá ver o valor original do registrador \$1 que ajudou a causar o overflow, pois funcionará como registrador de destino da instrução add. Devido ao planejamento cuidadoso, a exceção de overflow é detectada durante o estágio EX; logo, podemos usar o sinal EX.Flush para impedir que a instrução no estágio EX escreva seu resultado no estágio WB. Muitas exceções exigem que, por fim, completemos a instrução que causou a exceção como se ela fosse executada normalmente. O modo mais

fácil de fazer isso é dar flush na instrução e reiniciá-la desde o início após a exceção ser tratada.

A etapa final é salvar o endereço da instrução problemática no *Exception Program Counter* (EPC). Na realidade, salvamos o endereço +4, de modo que a rotina de tratamento da exceção, primeiro deve subtrair 4 do valor salvo. A Figura 4.66 mostra uma versão estilizada do caminho de dados, incluindo o hardware de desvio e as acomodações necessárias para tratar das exceções.

## Exceção em um computador com pipeline

### Exemplo

Dada esta sequência de instruções

40 <sub>hex</sub>	sub	\$11,	\$2,	\$4
44 <sub>hex</sub>	and	\$12,	\$2,	\$5
48 <sub>hex</sub>	or	\$13,	\$2,	\$6
4C <sub>hex</sub>	add	\$1,	\$2,	\$1
50 <sub>hex</sub>	slt	\$15,	\$6,	\$7
54 <sub>hex</sub>	lw	\$16,	50(\$7)	
• • •				

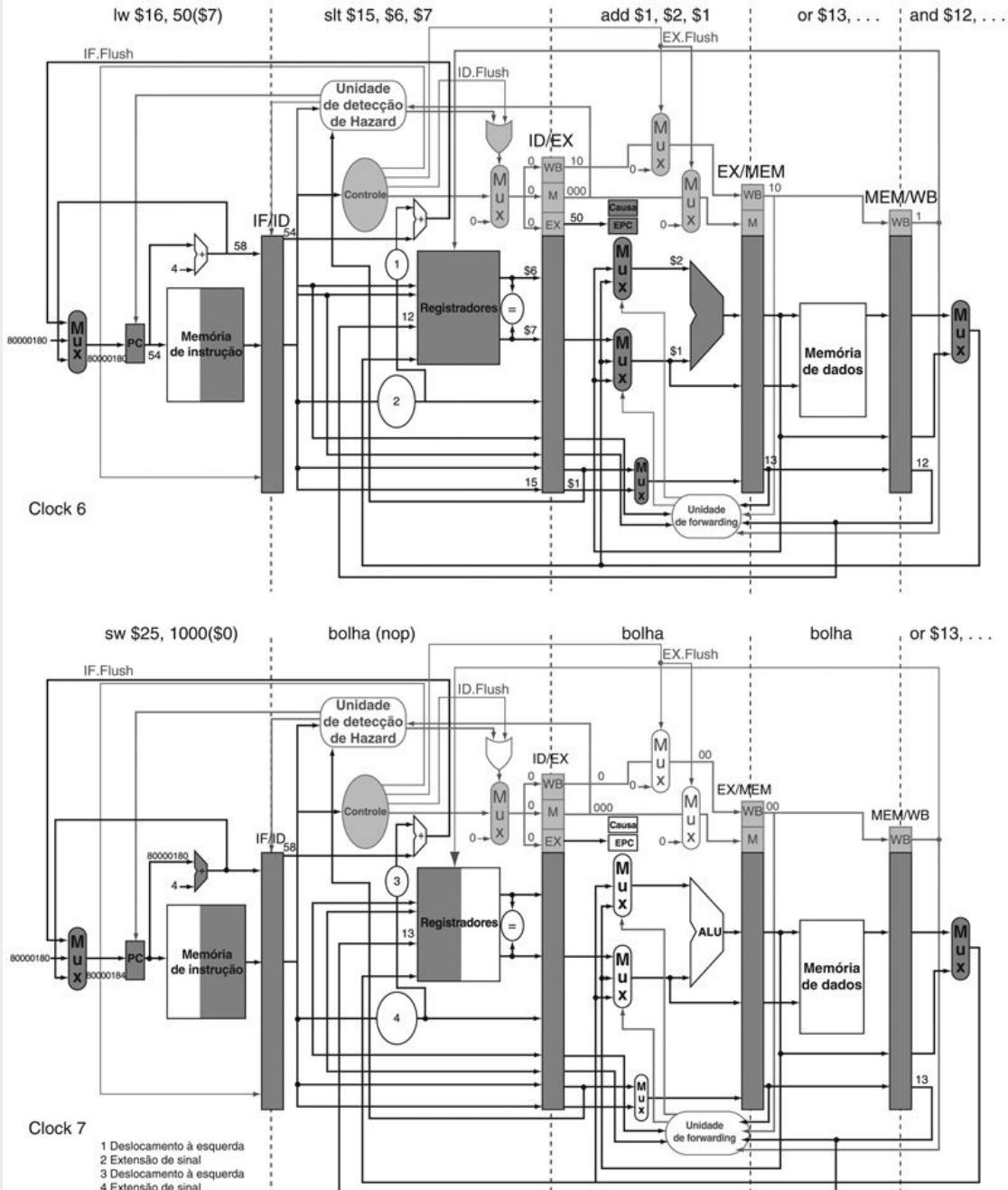
considere que as instruções a serem invocadas em uma exceção comecem desta forma:

$80000180_{\text{hex}}$	SW	\$26, 1000(\$0)
$80000184_{\text{hex}}$	SW	\$27, 1004(\$0)
. . .		

Mostre o que acontece no pipeline se houver uma exceção de overflow na instrução add.

## Resposta

A Figura 4.67 mostra os eventos, começando com a instrução add no estágio EX. O overflow é detectado durante essa fase, e  $8000\ 0180_{\text{hexa}}$  é forçado para o PC. O ciclo de clock 7 mostra que o add e as instruções seguintes sofrem flush, e a primeira instrução do código de exceção é buscada. Observe que o endereço da instrução *seguinte* ao add é salvo:  $4C_{\text{hexa}} + 4 = 50_{\text{hexa}}$ .



**FIGURA 4.67** O resultado de uma exceção devido a um overflow aritmético na instrução add.

O overflow é detectado durante o estágio EX do clock 6, salvando o endereço após o add no registrador EPC ( $4C + 4 = 50_{hexa}$ ). O overflow faz com que todos os sinais Flush sejam ativados perto do final desse ciclo de clock, desativando os valores de controle (colocando-os em 0) para o add. O ciclo de clock 7 mostra as instruções convertidas para bolhas no

pipeline, mas a busca da primeira instrução da rotina de exceção — `sw $25,1000($0)` — a partir do local da instrução `8000 0180hexa`. Observe que as instruções `AND` e `OR`, que estão antes do `add`, ainda completam. Embora não apareça, o sinal de overflow da ALU é uma entrada para a unidade de controle.

Mencionamos cinco exemplos de exceções na tabela da [Seção 4.9](#), e veremos outros no [Capítulo 5](#). Com cinco instruções ativas em qualquer ciclo de clock, o desafio é associar uma exceção à instrução apropriada. Além do mais, várias exceções podem ocorrer simultaneamente em um único ciclo de clock. A solução é priorizar as exceções de modo que seja fácil determinar qual será atendida primeiro. Na maioria das implementações MIPS, o hardware ordena as exceções de modo que a instrução mais antiga seja interrompida.

Solicitações de dispositivos de E/S e defeitos do hardware não estão associados a uma instrução específica, de modo que a implementação possui alguma flexibilidade quanto ao momento de interromper o pipeline. Logo, usar o mecanismo utilizado para outras exceções funciona muito bem.

O EPC captura o endereço das instruções interrompidas, e o registrador Cause do MIPS registra todas as exceções possíveis em um ciclo de clock, de modo que o software de exceção precisa combinar a exceção à instrução. Uma dica importante é saber em que estágio do pipeline um tipo de exceção pode ocorrer. Por exemplo, uma instrução indefinida é descoberta no estágio ID, e a chamada ao sistema operacional ocorre no estágio EX. As exceções são coletadas no registrador Cause em um campo de exceção pendente, de modo que o hardware possa interromper com base em exceções posteriores, uma vez que a mais antiga tenha sido atendida.

## Interface hardware/software

O hardware e o sistema operacional precisam trabalhar em conjunto para que as exceções se comportem conforme o esperado. O contrato do hardware normalmente é interromper a instrução problemática no meio do caminho, deixar que todas as instruções anteriores terminem, dar flush em todas as instruções seguintes, definir um registrador para mostrar a causa da exceção, salvar o endereço da instrução problemática e depois desviar para um endereço previamente arranjado. O contrato do sistema operacional é

examinar a causa da exceção e atuar de forma apropriada. Para uma instrução indefinida, falha de hardware ou exceção por overflow aritmético, o sistema operacional normalmente encerra o programa e retorna um indicador do motivo. Para uma solicitação de dispositivo de E/S ou uma chamada de serviço ao sistema operacional, o próprio sistema salva o estado do programa, realiza a tarefa desejada e, em algum ponto no futuro, restaura o programa para continuar a execução. No caso das solicitações do dispositivo de E/S, normalmente podemos escolher executar outra tarefa antes de retomar a tarefa que requisitou a E/S, pois essa tarefa em geral pode não ser capaz de prosseguir até que a E/S termine. É por isso que é fundamental a capacidade de salvar e restaurar o estado de qualquer tarefa. Um dos usos mais importantes e frequentes das exceções é o tratamento de faltas de página e exceções de TLB; o Capítulo 5 descreve essas exceções e seu tratamento com mais detalhes.

## Detalhamento

A dificuldade de sempre associar a exceção correta à instrução correta nos computadores em pipeline levou alguns projetistas de computador a relaxarem esse requisito em casos não críticos. Alguns processadores são considerados como tendo **interrupções imprecisas** ou **exceções imprecisas**. No exemplo anterior, o PC normalmente teria 58<sub>hexa</sub> no início do ciclo de clock, depois que a exceção for detectada, embora a instrução com problema esteja no endereço 4C<sub>hexa</sub>. Um processador com exceções imprecisas poderia colocar 58<sub>hexa</sub> no EPC e deixar que o sistema operacional determinasse qual instrução causou o problema. O MIPS e a grande maioria dos computadores de hoje admitem **interrupções precisas** ou **exceções precisas**. (Um motivo é para dar suporte à memória virtual, que veremos no Capítulo 5.)

### interrupção imprecisa

Também chamada **exceção imprecisa**. As interrupções ou exceções nos computadores em pipeline não estão associadas à instrução exata que foi a causa da interrupção ou exceção.

### interrupção precisa

Também chamada **exceção precisa**. Uma interrupção ou exceção que está sempre associada à instrução correta nos computadores em pipeline.

## Detalhamento

Embora o MIPS utilize o endereço de entrada de exceção  $8000\ 0180_{\text{hexa}}$  para quase todas as exceções, ele usa o endereço  $8000\ 0000_{\text{hexa}}$  de modo a melhorar o desempenho do tratador de exceção para exceções de falta de TLB (Capítulo 5).

## Verifique você mesmo

Qual exceção deverá ser reconhecida primeiro nesta sequência?

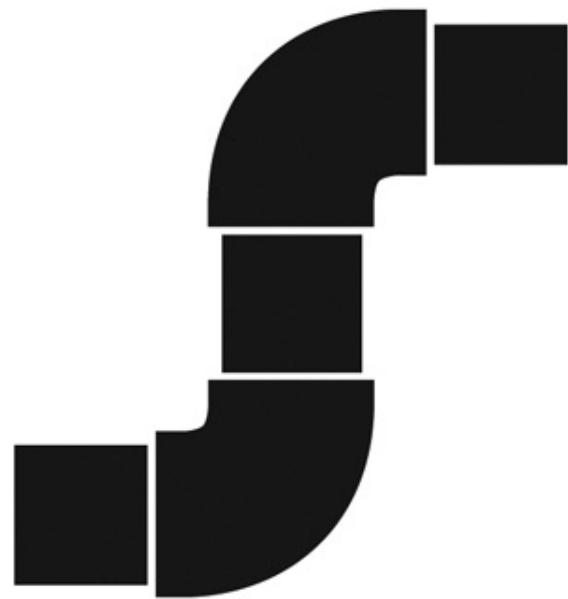
1. add \$1, \$2, \$1 # overflow aritmético
2. XXX \$1, \$2, \$1 # instrução indefinida
3. sub \$1, \$2, \$1 # erro de hardware

## 4.10. Paralelismo e paralelismo avançado em nível de instrução

Esteja avisado de que esta seção é uma breve introdução de assuntos fascinantes, porém avançados. Se você quiser saber mais detalhes, deverá consultar nosso livro mais avançado, *Computer Architecture: A Quantitative Approach*, 5<sup>a</sup> edição (Morgan Kaufmann, 2012), quarta edição, no qual o material explicado nas próximas páginas é expandido para mais de 200 páginas (incluindo Apêndices)!

A técnica de **pipelining** explora o **paralelismo** em potencial entre as instruções. Esse paralelismo é chamado de **paralelismo em nível de instrução** (ILP — Instruction-Level Parallelism). Existem dois métodos principais para aumentar a quantidade em potencial de paralelismo em nível de instrução. O primeiro é aumentar a profundidade do pipeline para sobrepor mais instruções. Usando nossa analogia da lavanderia e considerando que o ciclo da lavadora

fosse maior do que os outros, poderíamos dividir nossa lavadora em três máquinas que lavam, enxaguam e centrifugam, como as etapas de uma lavadora tradicional. Poderíamos, então, passar de um pipeline de quatro para seis estágios. Para ganhar o máximo de velocidade, precisamos rebalancear as etapas restantes de modo que tenham o mesmo tamanho, nos processadores ou na lavanderia. A quantidade de paralelismo sendo explorada é maior, pois existem mais operações sendo sobrepostas. O desempenho é potencialmente maior, pois o ciclo de clock pode ser encurtado.



**PIPELINING**



## PARALELISMO

### paralelismo em nível de instrução

O paralelismo entre as instruções.

Outra técnica é replicar os componentes internos do computador de modo que ele possa iniciar várias instruções em cada estágio do pipeline. O nome geral para essa técnica é **despacho múltiplo**. Uma lavanderia com despacho múltiplo substituiria nossa lavadora e secadora doméstica por, digamos, três lavadoras e três secadoras. Você também teria de recrutar mais auxiliares para passar e guardar três vezes a quantidade de roupas no mesmo período. A desvantagem é o trabalho extra de manter todas as máquinas ocupadas e transferir as trouxas de roupa para o próximo estágio do pipeline.

### despacho múltiplo

Um esquema pelo qual múltiplas instruções são disparadas em 1 ciclo de clock.

Disparar várias instruções por estágio permite que a velocidade de execução da instrução exceda a velocidade de clock ou, de forma alternativa, que o CPI seja menor do que 1. Como dissemos no [Capítulo 1](#), às vezes, é útil inverter a

métrica e usar o IPC ou *instruções por ciclo de clock*. Logo, um microprocessador de despacho múltiplo quádruplo de 4 GHz pode executar uma velocidade de pico de 16 bilhões de instruções por segundo e ter um CPI de 0,25 no melhor dos casos ou um IPC de 4. Considerando um pipeline de cinco estágios, esse processador teria 20 instruções executando em determinado momento. Os microprocessadores mais potentes de hoje tentam despachar de três a oito instruções a cada ciclo de clock. Entretanto, normalmente existem muitas restrições sobre os tipos das instruções que podem ser executadas simultaneamente e o que acontece quando surgem dependências.

Existem duas maneiras importantes de implementar um processador de despacho múltiplo, sendo que a principal diferença está na divisão de trabalho entre o compilador e o hardware. Como a divisão do trabalho indica se as decisões estão sendo feitas estaticamente (ou seja, durante a compilação) ou dinamicamente (ou seja, durante a execução), as técnicas, às vezes, são chamadas de **despacho múltiplo estático** e **despacho múltiplo dinâmico**. Como veremos, as duas técnicas possuem outros nomes, usados mais comumente, que podem ser menos precisos ou mais restritivos.

## despacho múltiplo estático

Uma técnica para implementar um processador de despacho múltiplo em que muitas decisões são tomadas pelo compilador antes da execução.

## despacho múltiplo dinâmico

Uma técnica para implementar um processador de despacho múltiplo em que muitas decisões são tomadas pelo processador durante a execução.

Existem duas responsabilidades principais e distintas que precisam ser tratadas em um pipeline de despacho múltiplo:

1. Empacotar as instruções em **slots de despacho**: como o processador determina quantas e quais instruções podem ser despachadas em determinado ciclo de clock? Na maioria dos processadores de despacho estático, esse processo é tratado, pelo menos, parcialmente pelo compilador; nos projetos de despacho dinâmico, isso normalmente é tratado durante a execução pelo processador, embora o compilador em geral já tenha tentado ajudar a melhorar a velocidade do despacho colocando as instruções em uma ordem benéfica.

2. Lidar com hazards de dados e de controle: em processadores de despacho estático, algumas ou todas as consequências dos hazards de dados e controle são tratadas estaticamente pelo compilador. Ao contrário, a maioria dos processadores de despacho dinâmico tenta aliviar pelo menos algumas classes de hazards usando técnicas de hardware operando durante a execução.

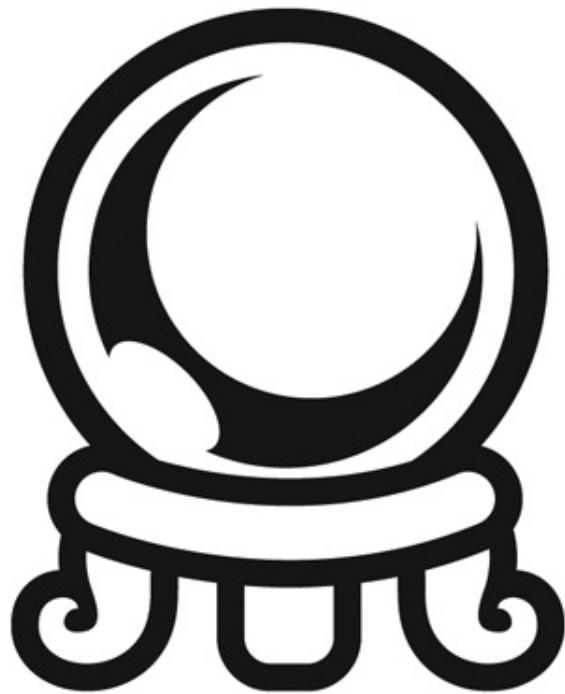
### slots de despacho

As posições das quais as instruções poderiam ser despachadas em determinado ciclo de clock; por analogia, correspondem a posições nos blocos iniciais para uma atividade.

Embora as tenhamos descrito como técnicas distintas, na realidade, cada técnica pega algo emprestado da outra e nenhuma pode afirmar ser perfeitamente pura.

## O conceito de especulação

Um dos métodos mais importantes para localizar e explorar mais ILP é a especulação. Com base na grande ideia da **predição, especulação** é uma técnica que permite que o compilador ou o processador “adivinhem” as propriedades de uma instrução, de modo a permitir que a execução comece para outras instruções que possam depender da instrução especulada. Por exemplo, poderíamos especular a respeito do resultado de um desvio, de modo que as instruções após o desvio pudessem ser executadas mais cedo. Outro exemplo é que poderíamos especular que um store que precede um load não se refere ao mesmo endereço, o que permitiria que o load fosse executado antes do store. A dificuldade com a especulação é que ela pode estar errada. Assim, qualquer mecanismo de especulação deve incluir tanto um método para verificar se a escolha foi certa quanto um método para retornar ou retroceder os efeitos das instruções executadas de forma especulativa. A implementação dessa capacidade de retrocesso aumenta a complexidade.



## P R E D I Ç Ã O

### especulação

Uma técnica pela qual o compilador ou processador adivinha o resultado de uma instrução para removê-la como uma dependência na execução de outras instruções.

A especulação pode ser feita pelo compilador ou pelo hardware. Por exemplo, o compilador pode usar a especulação para reordenar as instruções, fazendo uma instrução passar por um desvio ou um load passar por um store. O hardware do processador pode realizar a mesma transformação durante a execução, usando técnicas que discutiremos mais adiante nesta seção.

Os mecanismos de recuperação usados para a especulação incorreta são bem diferentes. No caso da especulação em software, o compilador normalmente insere instruções adicionais que verificam a precisão da especulação e oferecem uma rotina de reparo para usar quando a especulação tiver sido incorreta. Na especulação em hardware, o processador normalmente coloca os resultados

especulativos em um buffer até que saiba que não são mais especulativos. Se a especulação estiver correta, as instruções são concluídas, permitindo que o conteúdo dos buffers seja escrito nos registradores ou na memória. Se a especulação estiver incorreta, o hardware faz um flush nos buffers e executa novamente, mas na sequência de instruções correta.

A especulação apresenta outro problema possível: especular sobre certas instruções pode gerar exceções que, anteriormente, não estavam presentes. Por exemplo, suponha que uma instrução load seja movida de uma maneira especulativa, mas o endereço que usa não é válido quando a especulação for incorreta. O resultado é que ocorrerá uma exceção que não deveria ter ocorrido. O problema é complicado pelo fato de que, se a instrução load não fosse especulativa, então, a exceção deveria ocorrer! Na especulação feita pelo compilador, esses problemas são evitados pelo acréscimo de suporte especial à especulação, que permite que tais exceções sejam ignoradas, até que esteja claro que elas realmente devam ocorrer. Na especulação por hardware, as exceções são simplesmente mantidas em um buffer até que fique claro que a instrução que as causam não é mais especulativa e está pronta para terminar; nesse ponto, a exceção é gerada, e prossegue o tratamento normal da exceção.

Como a especulação pode melhorar o desempenho quando realizada corretamente e diminuir o desempenho quando feita descuidadamente, é preciso haver muito esforço na decisão de quando a especulação é apropriada. Mais adiante, nesta seção, vamos examinar as técnicas estática e dinâmica para a especulação.

## Despacho múltiplo estático

Todos os processadores de despacho múltiplo estático utilizam o compilador para ajudar no empacotamento de instruções e no tratamento de hazards. Em um processador de despacho estático, você pode pensar no conjunto de instruções despachadas em determinado ciclo de clock, o que é chamado **pacote de despacho**, como uma grande instrução com várias operações. Essa visão é mais do que uma analogia. Como um processador de despacho múltiplo estático normalmente restringe o mix de instruções que podem ser iniciadas em determinado ciclo de clock, é útil pensar no pacote de despacho como uma única instrução, permitindo várias operações em certos campos predefinidos. Essa visão levou ao nome original para essa técnica: **VLIW (Very Long Instruction Word** — palavra de instrução muito longa).

## **pacote de despacho**

O conjunto de instruções despachadas juntas em um ciclo de clock; o pacote pode ser determinado estaticamente, pelo compilador, ou dinamicamente, pelo processador.

## **VLIW (Very Long Instruction Word)**

Um estilo de arquitetura de conjunto de instruções que dispara muitas operações definidas para serem independentes em uma única instrução larga, normalmente com muitos campos de opcode separados.

A maioria dos processadores de despacho estático também conta com o compilador para assumir alguma responsabilidade por tratar de hazards de dados e controle. As responsabilidades do compilador podem incluir previsão estática de desvios e escalonamento de código, para reduzir ou impedir todos os hazards. Vejamos uma versão simples do despacho estático de um processador MIPS, antes de descrevermos o uso dessas técnicas em processadores mais agressivos.

### **Um exemplo: despacho múltiplo estático com a ISA do MIPS**

Para que você tenha uma ideia do despacho múltiplo estático, consideramos um processador MIPS simples capaz de despachar duas instruções por ciclo, sendo que uma das instruções pode ser uma operação da ALU com inteiros e a outra pode ser um load ou um store. Esse projeto é como aquele utilizado em alguns processadores MIPS embutidos. O despacho de duas instruções por ciclo exigirá a busca e a decodificação de 64 bits de instruções. Em muitos processadores de despacho múltiplo, e basicamente em todos os processadores VLIW, o layout do despacho de instruções simultâneas é restrito para simplificar a decodificação e o despacho da instrução. Logo, exigiremos que as instruções sejam emparelhadas e alinhadas em um limite de 64 bits, com a parte da ALU ou desvio aparecendo primeiro. Além do mais, se uma instrução do par não puder ser usada, exigimos que ela seja substituída por um nop. Assim, as instruções sempre são despachadas em pares, possivelmente com um nop em um slot. A [Figura 4.68](#) mostra como as instruções aparecem enquanto entram no pipeline em pares.

Tipo de instrução	Estágios do pipe						
	IF	ID	EX	MEM	WB		
Instrução da ALU ou desvio	IF	ID	EX	MEM	WB		
Instrução load ou store	IF	ID	EX	MEM	WB		
Instrução da ALU ou desvio		IF	ID	EX	MEM	WB	
Instrução load ou store		IF	ID	EX	MEM	WB	
Instrução da ALU ou desvio			IF	ID	EX	MEM	WB
Instrução load ou store			IF	ID	EX	MEM	WB
Instrução da ALU ou desvio				IF	ID	EX	MEM
Instrução load ou store				IF	ID	EX	WB

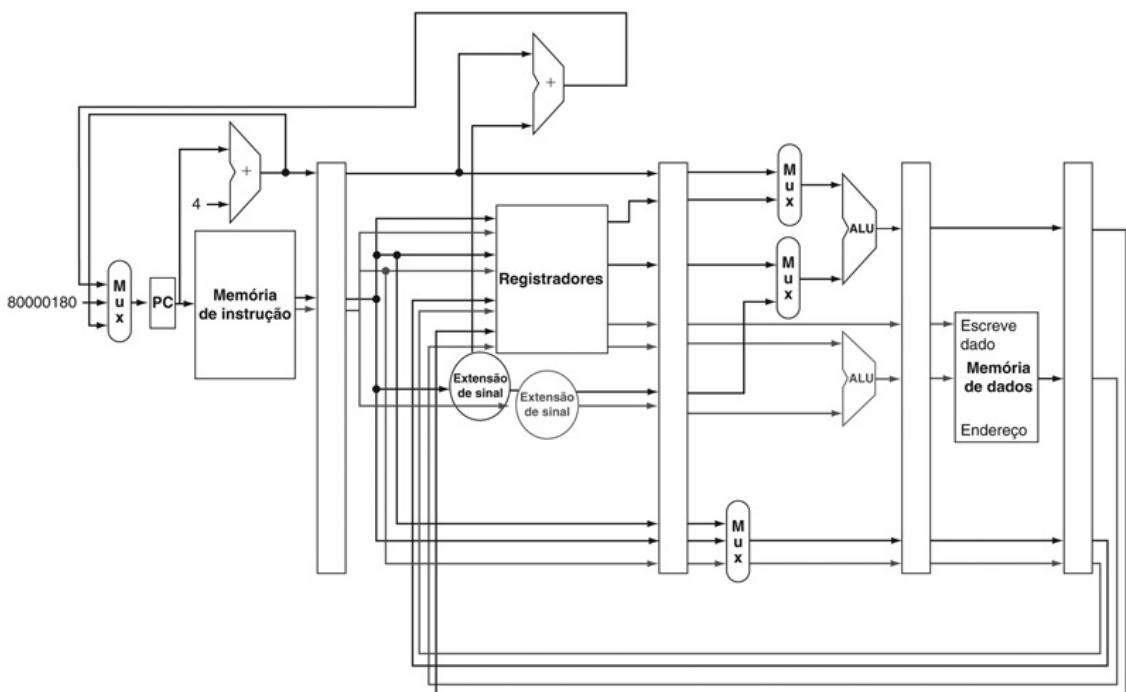
**FIGURA 4.68 Pipeline com despacho estático de duas instruções em operação.**

As instruções da ALU e de transferência de dados são despachadas ao mesmo tempo. Aqui, consideramos a mesma estrutura de cinco estágios utilizada para o pipeline de despacho único. Embora isso não seja estritamente necessário, possui algumas vantagens. Em particular, manter as escritas de registrador no final do pipeline simplifica o tratamento de exceções e a manutenção de um modelo de exceção preciso, que se torna mais difícil em processadores de despacho múltiplo.

Os processadores de despacho múltiplo estático variam no modo como lidam com hazards de dados e controle em potencial. Em alguns projetos, o compilador tem responsabilidade completa por remover *todos* os hazards, escalonando o código e inserindo no-ops de modo que o código execute sem qualquer necessidade de detecção de hazard ou stalls gerados pelo hardware. Em outros, o hardware detecta os hazards de dados e gera stalls entre dois pacotes de despacho, enquanto exige que o compilador evite todas as dependências dentro de um par de instruções. Mesmo assim, um hazard geralmente força o pacote de despacho inteiro contendo a instrução dependente a sofrer stall. Se o software precisa lidar com todos os hazards ou apenas tentar reduzir a fração de hazards entre pacotes de despacho separados, a aparência de haver uma única grande instrução com várias operações é reforçada. Ainda assumiremos a segunda técnica para esse exemplo.

Para emitir uma operação da ALU e uma operação de transferência de dados em paralelo, a primeira necessidade para o hardware adicional — além da lógica normal de detecção de hazard e stall — são portas extras no banco de registradores (Figura 4.69). Em um ciclo de clock, podemos ter de ler dois registradores para a operação da ALU e mais dois para um store, e também uma porta de escrita para uma operação da ALU e uma porta de escrita para um load. Como a ALU está presa à operação da ALU, também precisamos de um

somador separado, a fim de calcular o endereço efetivo para as transferências de dados. Sem esses recursos extras, nosso pipeline com despacho duplo seria atrapalhado pelos hazards estruturais.



**FIGURA 4.69 Um caminho de dados com despacho duplo estático.**

Os acréscimos necessários para o despacho duplo estão destacados: outros 32 bits da memória de instruções, mais duas portas de leitura e mais uma porta de escrita no banco de registradores, e outra ALU. Suponha que a ALU inferior trate dos cálculos de endereço para transferências de dados e a ALU superior trate de todo o restante.

Claramente, esse processador com despacho duplo pode melhorar o desempenho por um fator de até 2. Entretanto, fazer isso exige que o dobro de instruções seja superposto na execução e essa sobreposição adicional aumenta a perda de desempenho relativa aos hazards de dados e controle. Por exemplo, em nosso pipeline simples de cinco estágios, os loads possuem uma **latência de uso** de um ciclo de clock, o que impede que uma instrução use o resultado sem sofrer stall. No pipeline com despacho duplo e cinco estágios, o resultado de uma instrução load não pode ser usado no próximo *ciclo de clock*. Isso significa que as *duas* instruções seguintes não podem usar o resultado do load sem sofrer stall.

Além do mais, as instruções da ALU que não tiveram latência de uso no pipeline simples de cinco estágios, agora possuem uma latência de uso de uma instrução, pois os resultados não podem ser usados no load ou store emparelhados. Para explorar com eficiência o paralelismo disponível em um processador com despacho múltiplo, é preciso utilizar técnicas mais ambiciosas de escalonamento de compilador ou hardware, e o despacho múltiplo estático requer que o compilador assuma essa função.

## latência de uso

Número de ciclos de clock entre uma instrução load e uma instrução que pode usar o resultado do load sem stall do pipeline.

## Escalonamento de código simples para despacho múltiplo

### Exemplo

Como este loop seria escalonado em um pipeline com despacho duplo estático para o MIPS?

```
Loop: lw      $t0, 0($s1)    # $t0=elemento do array
      addu   $t0,$t0,$s2# add escalar em $s2
      sw      $t0, 0($s1)# resultado do store
      addi   $s1,$s1,-4# decrementa ponteiro
      bne    $s1,$zero,Loop# desvia se $s1!=0
```

Reordene as instruções para evitar o máximo de stalls do pipeline possível. Considere que os desvios são previstos, de modo que os hazards de controle sejam tratados pelo hardware.

### Resposta

As três primeiras instruções possuem dependências de dados, bem como as duas últimas. A Figura 4.70 mostra o melhor escalonamento para essas instruções. Observe que apenas um par de instruções possui os dois slots utilizados. São necessários quatro clocks por iteração do loop; com quatro

clocks para executar cinco instruções, obtemos o CPI decepcionante de 0,8 *versus* o melhor caso de 0,5 ou um IPC de 1,25 *versus* 2,0. Observe que, no cálculo do CPI ou do IPC, não contamos quaisquer nops executados como instruções úteis. Isso melhoraria o CPI, mas não o desempenho!

	Instrução da ALU ou desvio	Instrução de transferência de dados	Ciclo de clock
Loop:		lw \$t0, 0(\$s1)	1
	addi \$s1,\$s1,-4		2
	addu \$t0,\$t0,\$s2		3
	bne \$s1,\$zero,Loop	sw \$t0, 4(\$s1)	4

**FIGURA 4.70** O código escalonado conforme apareceria em um pipeline MIPS com despacho duplo.

Os slots vazios são nops.

Uma importante técnica de compilador para conseguir mais desempenho dos loops é o **desdobramento de loop** (loop unrolling), em que são feitas várias cópias do corpo do loop. Após o desdobramento, haverá mais ILP disponível pela sobreposição de instruções de diferentes iterações.

## desdobramento de loop (loop unrolling)

Uma técnica para conseguir mais desempenho dos loops que acessam arrays, em que são feitas várias cópias do corpo do loop e instruções de diferentes iterações são escalonadas juntas.

## Desdobramento de loop para pipelines com despacho múltiplo

### Exemplo

Veja como o trabalho de desdobramento do loop e escalonamento funciona no exemplo anterior. Para simplificar, suponha que o índice do loop seja um múltiplo de quatro.

### Resposta

Para escalar o loop sem quaisquer atrasos, acontece que precisamos fazer

quatro cópias do corpo do loop. Depois de desdobrar e eliminar as instruções de overhead de loop desnecessárias, o loop terá quatro cópias de `lw`, `add` e `sw`, mais um `addi` e um `bne`. A Figura 4.71 mostra o código desdoblado e escalonado.

	Instrução da ALU ou desvio	Instrução de transferência de dados	Ciclo de clock
Loop:	<code>addi \$s1,\$s1,-16</code>	<code>lw \$t0, 0(\$s1)</code>	1
		<code>lw \$t1,12(\$s1)</code>	2
	<code>addu \$t0,\$t0,\$s2</code>	<code>lw \$t2, 8(\$s1)</code>	3
	<code>addu \$t1,\$t1,\$s2</code>	<code>lw \$t3, 4(\$s1)</code>	4
	<code>addu \$t2,\$t2,\$s2</code>	<code>sw \$t0, 16(\$s1)</code>	5
	<code>addu \$t3,\$t3,\$s2</code>	<code>sw \$t1,12(\$s1)</code>	6
		<code>sw \$t2, 8(\$s1)</code>	7
	<code>bne \$s1,\$zero,Loop</code>	<code>sw \$t3, 4(\$s1)</code>	8

**FIGURA 4.71** O código desdoblado e escalonado da Figura 4.70 conforme apareceria no pipeline MIPS com despacho duplo estático.

Os slots vazios são nops. Como a primeira instrução no loop decrementa `$s1` em 16, os endereços lidos são o valor original de `$s1`, depois esse endereço menos 4, menos 8 e menos 12.

Durante o processo de desdoblamento, o compilador introduziu registradores adicionais (`$t1`, `$t2`, `$t3`). O objetivo desse processo, chamado **renomeação de registradores**, é eliminar dependências que não são dependências de dados verdadeiras, mas que poderiam levar a hazards em potencial ou impedir que o compilador escalonasse o código de forma flexível. Considere como o código não desdoblado apareceria usando apenas `$t0`. Haveria instâncias repetidas de `lw $t0,0($s1)`, `addu $t0,$t0,$s2` seguidas por `sw $t0,4($s1)`, mas essas sequências, apesar do uso de `$t0`, na realidade são completamente independentes — nenhum valor de dados flui entre um par dessas instruções e o par seguinte. É isso que é chamado de **antidependência** ou **dependência de nome**, que é uma ordenação forçada puramente pela reutilização de um nome, em vez de uma dependência de dados real, que também é chamada de dependência verdadeira.

Renomear os registradores durante o processo de desdoblamento permite que o compilador move subsequentemente essas instruções independentes, de modo a escalar melhor o código. O processo de renomeação elimina as dependências de nome, enquanto preserva as dependências verdadeiras.

Observe agora que 12 das 14 instruções no loop são executadas como um par. São necessários oito clocks para quatro iterações do loop ou dois clocks por iteração, o que gera um CPI de  $8/14 = 0,57$ . O desdobramento e o escalonamento do loop com despacho dual nos deram um fator de melhoria de quase dois, parcialmente pela redução das instruções de controle de loop e parcialmente pela execução do despacho dual. O custo dessa melhoria de desempenho é usar quatro registradores temporários em vez de um, além de um aumento significativo no tamanho do código.

## renomeação de registradores

O restante dos registradores é usado, pelo compilador ou hardware, para remover antidependências.

## antidependência

Também chamada **dependência de nome**. Uma ordenação forçada pela reutilização de um nome, normalmente um registrador, em vez de uma dependência verdadeira que transporta um valor entre duas instruções.

# Processadores com despacho múltiplo dinâmico

Os processadores de despacho múltiplo dinâmico também são conhecidos como processadores **superescalares** ou simplesmente superescalares. Nos processadores superescalares mais simples, as instruções são despachadas em ordem e o processador decide se zero, uma ou mais instruções podem ser despachadas em determinado ciclo de clock. Obviamente, conseguir um bom desempenho em tal processador ainda exige que o compilador tente escalar instruções para separar as dependências e, com isso, melhorar a velocidade de despacho de instruções. Mesmo com esse escalonamento de compilador, existe uma diferença importante entre essa arquitetura superescalar simples e um processador VLIW: o código, seja ele escalarizado ou não, é garantido pelo hardware que será executado corretamente. Além do mais, o código compilado sempre será executado corretamente, independente da velocidade de despacho ou estrutura do pipeline do processador. Em alguns projetos VLIW, isso não tem acontecido, e a recompilação foi necessária quando da mudança por diferentes modelos de processador; em outros processadores de despacho estático, o código seria executado corretamente em diversas implementações, mas constantemente

de uma forma tão pouco eficiente que torna a compilação necessária.

## superescalar

Uma técnica de pipelining avançada que permite que o processador execute mais de uma instrução por ciclo de clock selecionando-as durante a execução.

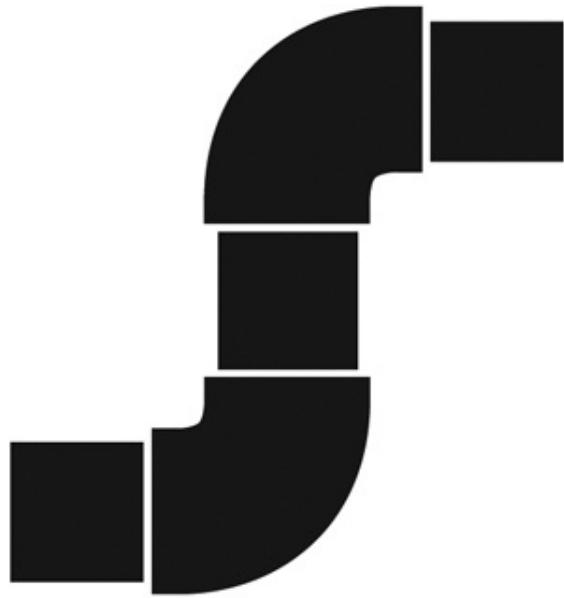
Muitas arquiteturas superescalares estendem a estrutura básica das decisões de despacho dinâmico para incluir **escalonamento dinâmico em pipeline**. O escalonamento dinâmico em pipeline escolhe quais instruções serão executadas em determinado ciclo de clock, enquanto tenta evitar hazards e stalls. Vamos começar com um exemplo simples de impedimento de um hazard de dados. Considere a seguinte sequência de código:

```
lw      $t0, 20($s2)
addu   $t1, $t0, $t2
sub    $s4, $s4, $t3
slt i $t5, $s4, 20
```

## escalonamento dinâmico em pipeline

Suporte do hardware para modificar a ordem de execução das instruções de modo a evitar stalls.

Embora a instrução sub esteja pronta para executar, ela precisa esperar que lw e addu terminem primeiro, o que poderia exigir muitos ciclos de clock se a memória for lenta. (O Capítulo 5 explica as faltas de cache, motivo pelo qual os acessos à memória às vezes são muito lentos.) O escalonamento dinâmico em **pipeline** permite que tais hazards sejam evitados total ou parcialmente.

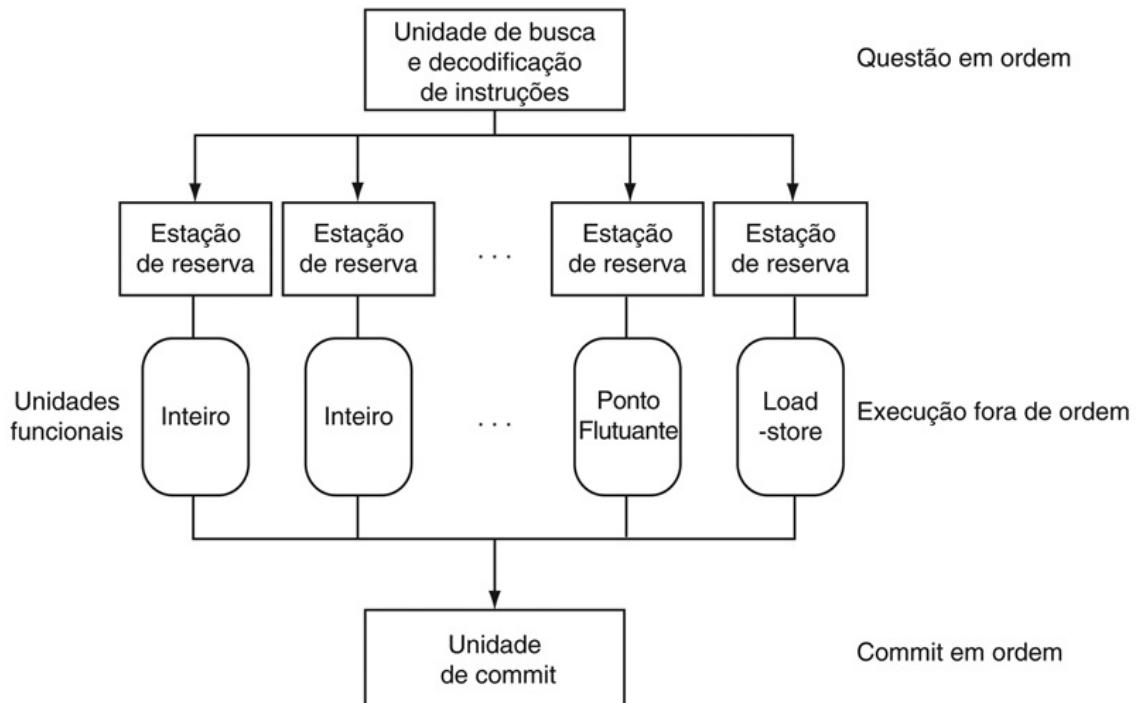


## PIPELINING

### Escalonamento dinâmico em pipeline

O escalonamento dinâmico em pipeline escolhe quais instruções serão executadas em seguida, possivelmente reordenando-as para evitar stalls. Nestes processadores, o pipeline é dividido em três unidades principais: uma unidade de busca e despacho de instruções, várias unidades funcionais (uma dezena ou mais nos projetos de alto nível em 2013) e uma **unidade de commit**. A [Figura 4.72](#) mostra o modelo. A primeira unidade busca instruções, decodifica-as e envia cada instrução a uma unidade funcional correspondente para execução. Cada unidade funcional possui buffers, chamados **estações de reserva**, que mantêm os operandos e a operação. (Na seção de Detalhamento, discutiremos uma alternativa às estações de reserva utilizadas por muitos processadores recentes.) Assim que o buffer tiver todos os seus operandos e a unidade funcional estiver pronta para executar, o resultado será calculado. Quando o resultado for completado, ele será enviado a quaisquer estações de reserva esperando por esse resultado em particular, bem como a unidade de commit, que mantém o resultado em um buffer até que seja seguro colocar o resultado no banco de registradores ou, para um store, na memória. O buffer na unidade de commit, normalmente chamado de **buffer de reordenação**, também é usado para

fornecer operandos, mais ou menos da mesma maneira como a lógica de forwarding faz em um pipeline escalonado estaticamente. Quando um resultado é submetido ao banco de registradores, ele pode ser apanhado diretamente de lá, como em um pipeline normal.



**FIGURA 4.72 As três unidades principais de um pipeline escalonado dinamicamente.**

A etapa final da atualização do estado também é chamada de reforma ou graduação.

## unidade de commit

A unidade em um pipeline de execução dinâmica ou fora de ordem que decide quando é seguro liberar o resultado de uma operação aos registradores e memória visíveis ao programador.

## estação de reserva

Um buffer dentro de uma unidade funcional que mantém os operandos e a operação.

## buffer de reordenação

O buffer que mantém resultados em um processador escalonado dinamicamente até que seja seguro armazenar os resultados na memória ou em um registrador.

A combinação de operandos em buffers nas estações de reserva e os resultados no buffer de reordenação oferecem uma forma de renomeação de registradores, assim como aquela utilizada pelo compilador em nosso exemplo anterior de desdobramento de loop, anteriormente neste capítulo. Para ver como isso funciona conceitualmente, considere as seguintes etapas:

1. Quando uma instrução é despachada, ela é copiada para uma estação de reserva para a unidade funcional apropriada. Quaisquer operandos que estejam disponíveis no banco de registradores ou no buffer de reordenação também serão copiados para a estação de reserva imediatamente. A instrução é mantida em um buffer até que todos os operandos e a unidade funcional estejam disponíveis. Para a instrução despachada, a cópia do registrador operando não é mais necessária, e se houvesse uma escrita nesse registrador, o valor poderia ser reescrito.
2. Se um operando não estiver no banco de registradores ou no buffer de reordenação, ele terá de esperar para ser produzido por uma unidade funcional. O nome da unidade funcional que produzirá o resultado é rastreado. Quando essa unidade por fim produz o resultado, ele é copiado diretamente para a estação de reserva, que estava aguardando, a partir da unidade funcional, sem passar pelos registradores.

Essas etapas efetivamente utilizam o buffer de reordenação e as estações de reserva para implementar a renomeação de registradores.

Conceitualmente, você pode pensar em um pipeline escalonado de forma dinâmica como uma análise da estrutura de fluxo de dados de um programa. O processador executa as instruções em alguma ordem que preserva a ordem do fluxo de dados do programa. Esse estilo de execução é chamado de **execução fora de ordem**, pois as instruções podem ser executadas em uma ordem diferente daquela em que foram apanhadas.

## execução fora de ordem

Uma situação na execução em pipeline quando uma instrução com execução bloqueada não faz com que as instruções seguintes esperem.

Para fazer com que os programas se comportem como se estivessem executando em um pipeline simples em ordem, a unidade de busca e decodificação de instruções precisa despachar instruções em ordem, o que permite que as dependências sejam acompanhadas, e a unidade de commit precisa escrever resultados nos registradores e na memória na ordem de execução do programa. Esse modo conservador é chamado de **commit em ordem**. Logo, se houver uma exceção, o computador poderá apontar para a última instrução executada, e os únicos registradores atualizados serão aqueles escritos pelas instruções antes da instrução que causa a exceção. Apesar de o front end (busca e despacho) e o back end (commit) do pipeline executarem em ordem, as unidades funcionais são livres para iniciar a execução sempre que os dados de que precisam estiverem disponíveis. Hoje, todos os pipelines escalonados dinamicamente utilizam o commit em ordem.

## commit em ordem

Um commit em que os resultados da execução em pipeline são escritos no estado visível ao programador na mesma ordem em que as instruções são buscadas.

Em geral, o escalonamento dinâmico é estendido pela inclusão da especulação baseada em hardware, especialmente para resultados de desvios. Prevendo a direção de um desvio, um processador escalonado dinamicamente pode continuar a buscar e executar instruções ao longo do caminho previsto. Como as instruções possuem um commit em ordem, sabemos se o desvio foi previsto corretamente ou não, antes que quaisquer instruções do caminho previsto tenham seus resultados atualizados pelas unidades de commit. Um pipeline especulativo, escalonado dinamicamente, também pode admitir especulação nos endereços de load, permitindo uma reordenação load-store e usando a unidade de commit para evitar a especulação incorreta. Na próxima seção, veremos o uso do escalonamento dinâmico com especulação no projeto do Intel Core i7.

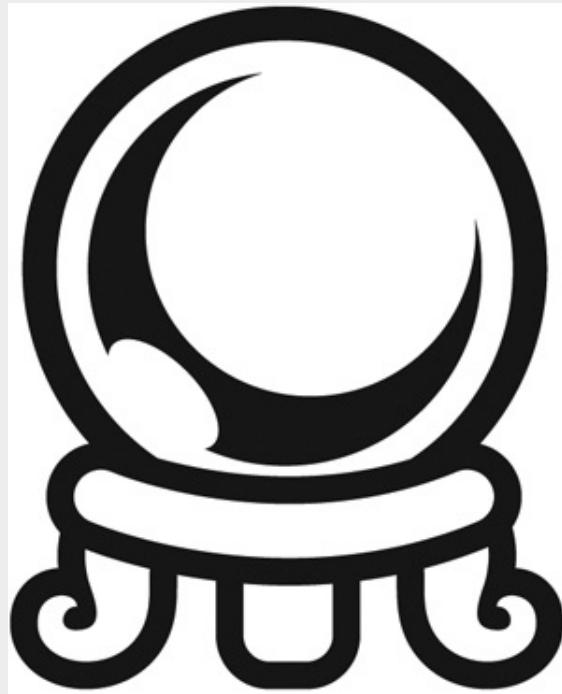
## Entendendo o desempenho dos programas

Dado que os compiladores também podem escalonar o código em torno das dependências de dados, você poderia perguntar por que um processador superescalar usaria o escalonamento dinâmico. Existem três motivos principais. Primeiro, nem todos os stalls são previsíveis. Em particular, as

fallas de cache (Capítulo 5) na **hierarquia de memória** causam stalls imprevisíveis. O escalonamento dinâmico permite que o processador oculte alguns desses stalls continuando a executar instruções enquanto esperam que o stall termine.



Segundo, se o processador especula sobre resultados de desvio usando a **predição** de desvio dinâmica, ele não pode saber a ordem exata das instruções durante a compilação, pois isso depende do comportamento previsto e real dos desvios. A incorporação da especulação dinâmica para explorar mais o *parallelismo em nível de instrução* (ILP) sem incorporar o escalonamento dinâmico restringiria significativamente os benefícios de tal especulação.



## P R E D I Ç Ã O

Terceiro, como a latência do pipeline e a largura do despacho mudam de uma implementação para outra, a melhor maneira de compilar uma sequência de código também muda. Por exemplo, a forma de escalonar uma sequência de instruções dependentes é afetada tanto pela largura quanto pela latência do despacho. A estrutura do pipeline afeta o número de vezes que um loop precisa ser desdobrado para evitar stalls e também o processo de renomeação de registradores feito pelo compilador. O escalonamento dinâmico permite que o hardware oculte a maioria desses detalhes. Assim, os usuários e os distribuidores de software não precisam se preocupar em ter várias versões de um programa para diferentes implementações do mesmo conjunto de instruções. De modo semelhante, o código antigo legado receberá grande parte do benefício de uma nova implementação sem a necessidade de recompilação.

### Colocando em perspectiva

Tanto a técnica de **pipelining** quanto a execução com despacho múltiplo

aumentam a vazão máxima de instruções e tentam explorar o **parallelismo** em nível de instrução (ILP). No entanto, as dependências de dados e controle nos programas oferecem um limite superior sobre o desempenho sustentado, pois o processador, às vezes, precisa esperar que uma dependência seja resolvida. As técnicas centradas no software para a exploração do ILP contam com a capacidade do compilador de encontrar e reduzir os efeitos de tais dependências, enquanto as técnicas centradas no hardware contam com extensões para o pipeline e mecanismos de despacho. A especulação, realizada pelo compilador ou pelo hardware, pode aumentar a quantidade de ILP que pode ser explorada por meio da **predição**, embora se deva ter cuidado, visto que a especulação incorreta provavelmente reduzirá o desempenho.





PARALELISMO



PREDIÇÃO

## Interface hardware/software

Processadores modernos, de alto desempenho, são capazes de despachar várias instruções por clock; infelizmente, é muito difícil sustentar essa taxa de despacho. Por exemplo, apesar da existência de processadores com despacho de quatro a seis instruções por clock, muito poucas aplicações podem sustentar mais do que duas instruções por clock. Existem dois motivos principais para isso.

Primeiro, dentro do pipeline, os principais gargalos no desempenho surgem das dependências que não podem ser aliviadas, reduzindo assim o paralelismo entre as instruções e a velocidade de despacho sustentada. Embora pouca coisa possa ser feita sobre as verdadeiras dependências dos dados, normalmente o compilador ou o hardware não sabe exatamente se uma dependência existe ou não e, por isso, precisa considerar de forma conservadora que a dependência existe. Por exemplo, o código que utiliza ponteiros, principalmente os que criam mais aliasing, levará a dependências em potencial mais implícitas. Ao contrário, a maior regularidade dos acessos a um array normalmente permite que um compilador deduza que não existem dependências. De modo semelhante, os desvios que não podem ser previstos com precisão, seja em tempo de execução ou de compilação, limitarão a capacidade de explorar o ILP. Em geral, o ILP adicional está disponível, mas a capacidade do compilador ou o hardware encontrar ILP que possa estar bastante separado (às vezes, pela execução de milhares de instruções) é limitada.

Em segundo lugar, as perdas na **hierarquia da memória** (o tópico do Capítulo 5) também limitam a capacidade de manter o pipeline cheio. Alguns stalls do sistema de memória podem ser escondidos, mas quantidades limitadas de ILP também limitam a extensão à qual esses stalls podem ser escondidos.



## Eficiência de potência e pipelining avançado

A desvantagem do aumento da exploração do paralelismo em nível de instrução por meio do despacho múltiplo dinâmico e especulação é a eficiência de potência. Cada inovação foi capaz de transformar mais transistores em desempenho, mas geralmente eles faziam isso de modo muito ineficaz. Agora que atingimos o muro da potência, estamos vendo projetos com múltiplos processadores por chip em que os processadores não são tão profundamente dispostos em pipeline ou tão agressivamente especulativos quanto seus predecessores.

A crença é que, embora os processadores mais simples não sejam tão rápidos quanto seus irmãos sofisticados, eles oferecem melhor desempenho por watt, de modo que podem oferecer mais desempenho por chip quando os projetos são restritos mais por potência do que por número de transistores.

A [Figura 4.73](#) mostra o número de estágios de pipeline, largura do despacho, nível de especulação, taxa de clock, núcleos por chip e potência de vários microprocessadores do passado e recentes. Observe a queda nos estágios de pipeline e potência enquanto as empresas passam para projetos multicore.

Microprocesso	Ano	Taxa de clock	Estágios de pipeline	Largura do despacho	Fora de ordem/especulação	Núcleos/chip	Potência	
Intel 486	1989	25 MHz	5	1	Não	1	5	W
Intel Pentium	1993	66 MHz	5	2	Não	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Sim	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Sim	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Sim	1	103	W
Intel Core	2006	2930 MHz	14	4	Sim	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Sim	1	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Sim	8	77	W

**FIGURA 4.73 Registro dos microprocessadores Intel em termos de complexidade de pipeline, número de cores (núcleos) e potência.**

Os estágios de pipeline do Pentium 4 não incluem os estágios de commit. Se os incluíssemos, os pipelines do Pentium 4 seriam ainda mais profundos.

## Detalhamento

Uma unidade de commit controla atualizações no banco de registradores e na memória. Alguns processadores escalonados dinamicamente atualizam o banco de registradores imediatamente durante a execução, usando registradores extras para implementar a função de renomeação e preservar a cópia mais antiga de um registrador até que a instrução atualizando o registrador não seja mais especulativa. Outros processadores mantêm o resultado em buffer, normalmente em uma estrutura chamada buffer de reordenação e a atualização real no banco de registradores ocorre depois, como parte do commit. Stores na memória precisam ser colocados em buffer até o momento do commit, seja em um *buffer de store* (Capítulo 5) ou no buffer de reordenação. A unidade de commit permite que o store escreva na memória, a partir do buffer, quando ele tiver um endereço com dados válidos e quando o store não for mais dependente de desvios previstos.

## Detalhamento

Os acessos à memória se beneficiam das *caches sem bloqueio*, que continuam a atender acessos da cache durante uma falta de cache (Capítulo 5). Os processadores com execução fora de ordem precisam do projeto de cache para permitir que as instruções sejam executadas durante uma falha.

## Verifique você mesmo

Indique se as técnicas ou componentes a seguir estão associados principalmente a uma técnica baseada em software ou hardware para a exploração do ILP. Em alguns casos, a resposta pode ser “ambos”.

1. Previsão de desvio
2. Despacho múltiplo
3. VLIW
4. Superescalar
5. Escalonamento dinâmico
6. Execução fora de ordem
7. Especulação
8. Buffer de reordenação
9. Renomeação de registradores

## 4.11. Vida real: pipelines do ARM Cortex-A8 e Intel Core i7

A Figura 4.74 descreve os dois microprocessadores que examinaremos nesta seção, cujos destinos são os dois exemplos típicos da era pós-PC.

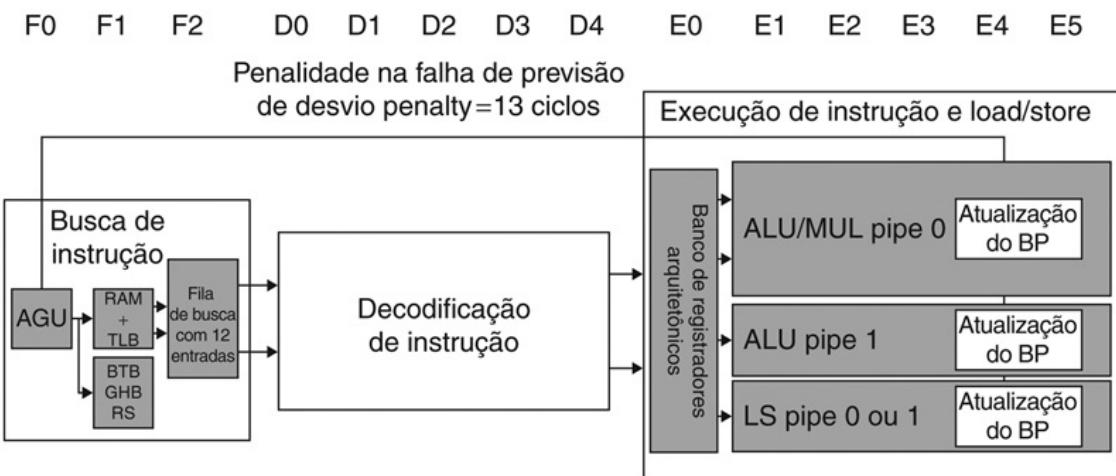
Processador	ARM A8	Intel Core i7 920
Mercado	Dispositivo móvel pessoal	Servidor, Nuvem
Potência do projeto térmico	2 Watts	130 Watts
Taxa de clock	1 GHz	2.66 GHz
Núcleos/chip	1	4
Ponto flutuante?	Não	Sim
Despacho múltiplo?	Dinâmico	Dinâmico
Pico de instruções/ciclo de clock	2	4
Estágios do pipeline	14	14
Escalonamento de pipeline	Estático em ordem	Dinâmico fora de ordem com especulação
Predição de desvio	2-níveis	2-níveis
Caches de 1º nível/núcleo	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
Cache de 2º nível/núcleo	128–1024 KiB	256 KiB
Cache de 3º nível (compartilhado)	–	2–8 MiB

**FIGURA 4.74** Especificação do ARM Cortex-A8 e do Intel

Core i7 920.

## O ARM Cortex-A8

O ARM Cortex-A8 roda a 1 GHz com um pipeline de 14 estágios. Ele usa o despacho múltiplo dinâmico, com duas instruções por ciclo de clock. Ele é um pipeline estático em ordem, no qual as instruções são despachadas, executadas e confirmadas ordenadamente. O pipeline consiste em três seções para busca de instruções, decodificação de instruções e execução. A [Figura 4.75](#) mostra o pipeline geral.



**FIGURA 4.75** O pipeline do A8.

Os três primeiros estágios leem instruções para um buffer de busca de instrução com 12 entradas. A *unidade de geração de endereço* (AGU — Address Generation Unit) usa um *buffer de destino de desvio* (BTB — Branch Target Buffer), *buffer de histórico global* (GHB — Global History Buffer) e *pilha de retorno* (RS — Return Stack) para prever desvios a fim de tentar manter cheia a fila de busca. A decodificação de instruções tem cinco estágios e a execução de instruções tem seis estágios.

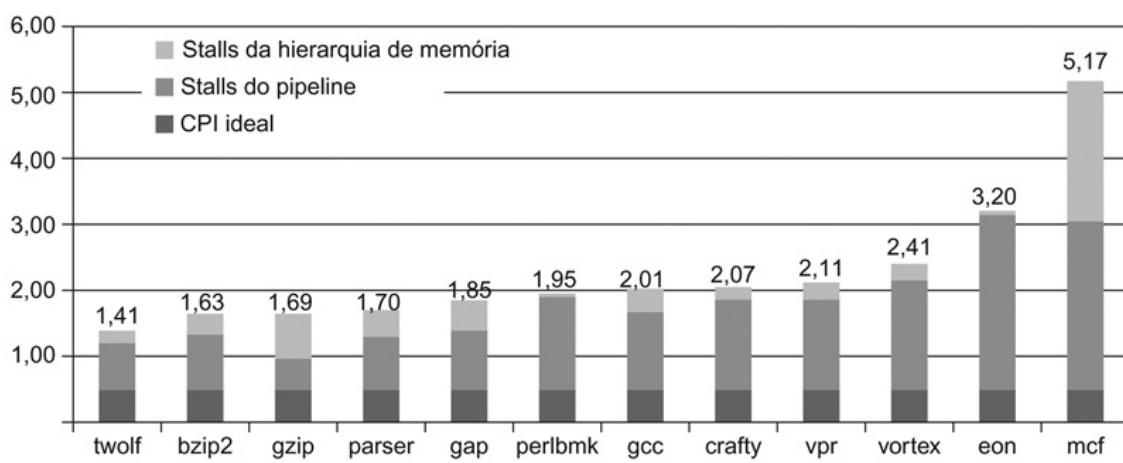
Os três primeiros estágios buscam duas instruções de uma só vez e tentam manter cheio um buffer de pré-busca com 12 instruções. Ele usa um previsor de desvio de dois níveis usando um buffer de destino de 512 entradas, um buffer de histórico global de 4096 entradas e uma pilha de retorno de 8 entradas. Quando a previsão de desvio é errada, ele esvazia o pipeline, resultando em uma

penalidade na falha de previsão de desvio com 13 ciclos de clock.

Os cinco estágios do pipeline de decodificação determinam se existem dependências entre um par de instruções, o que forçaria a execução sequencial, e em qual pipeline dos estágios de execução as instruções são enviadas.

Os seis estágios da seção de execução de instrução oferecem um pipeline para instruções load e store e dois pipelines para operações aritméticas, embora somente o primeiro do par possa lidar com multiplicações. Qualquer instrução do par pode ser despachada ao pipeline de load-store. Os estágios de execução possuem bypassing pleno entre os três pipelines.

A [Figura 4.76](#) mostra o CPI do A8 usando pequenas versões de programas derivados dos benchmarks SPEC2000. Embora o CPI ideal seja 0,5, o melhor caso é 1,4, o caso mediano é 2,0 e o pior caso é 5,2. Para o caso mediano, 80% dos stalls devem-se aos hazards de pipelining e 20% são stalls devidos à hierarquia de memória. Os stalls do pipeline são causados por falhas de previsão de desvio, hazards estruturais e dependências de dados entre pares de instruções. Devido ao pipeline estático do A8, fica a critério do compilador tentar evitar hazards estruturais e dependências de dados.



**FIGURA 4.76** CPI no ARM Cortex A8 para os benchmarks Minnespec, que são pequenas versões dos benchmarks SPEC2000.

Estes benchmarks usam as entradas muito menores para reduzir o tempo de execução em várias ordens de grandeza. O tamanho menor *subestima* significativamente o impacto do CPI da hierarquia de memória ([Capítulo 5](#)).

## Detalhamento

O Cortex-A8 é um núcleo (core) configurável que tem suporte para a arquitetura do conjunto de instrução ARMv7. Ele é entregue como um *núcleo de propriedade intelectual (IP — Intellectual Property)*. Núcleos IP são a forma dominante de entrega de tecnologia nos mercados de dispositivos móveis embutidos, pessoais e relacionados; bilhões de processadores ARM e MIPS foram criados a partir desses núcleos IP.

Observe que os núcleos IP são diferentes dos núcleos nos computadores Intel i7 multicore. Um núcleo IP (que pode, por si só, ser um multicore) é projetado para ser incorporado com outra lógica (daí ele ser um “núcleo” de um chip), incluindo processadores específicos da aplicação (como um codificador ou decodificador para vídeo), interfaces de E/S e interfaces de memória, e depois fabricado para produzir um processador otimizado para uma aplicação em particular. Embora o núcleo do processador seja quase idêntico, os chips resultantes possuem muitas diferenças. Um parâmetro é o tamanho da cache L2, que pode variar por um fator de oito.

## O Intel Core i7 920

Os microprocessadores x86 empregam técnicas sofisticadas de pipelining, usando o despacho múltiplo dinâmico e o escoamento de pipeline dinâmico com execução fora de ordem e especulação para o seu pipeline de 14 estágios. Porém, esses processadores ainda enfrentam o desafio de implementar o complexo conjunto de instruções do x86, descrito no [Capítulo 2](#). O processador Intel busca instruções x86 e as traduz em instruções internas tipo MIPS, que a Intel chama de *micro-operações*. As micro-operações são então executadas por um pipeline sofisticado, especulativo e dinamicamente escalonado, capaz de sustentar a taxa de execução de até seis micro-operações por ciclo de clock. Esta seção é focada nesse pipeline de micro-operações.

Quando consideramos o projeto de processadores sofisticados, escalonados dinamicamente, o projeto das unidades funcionais, da cache e banco de registradores, do despacho de instruções e do controle geral do pipeline tornam-se algo combinado, dificultando a separação entre o caminho de dados e o pipeline. Por causa disso, muitos engenheiros e pesquisadores têm adotado o termo **microarquitetura** para se referirem à arquitetura interna detalhada de um processador.

## microarquitetura

A organização do processador, incluindo as principais unidades funcionais, sua interconexão e controle.

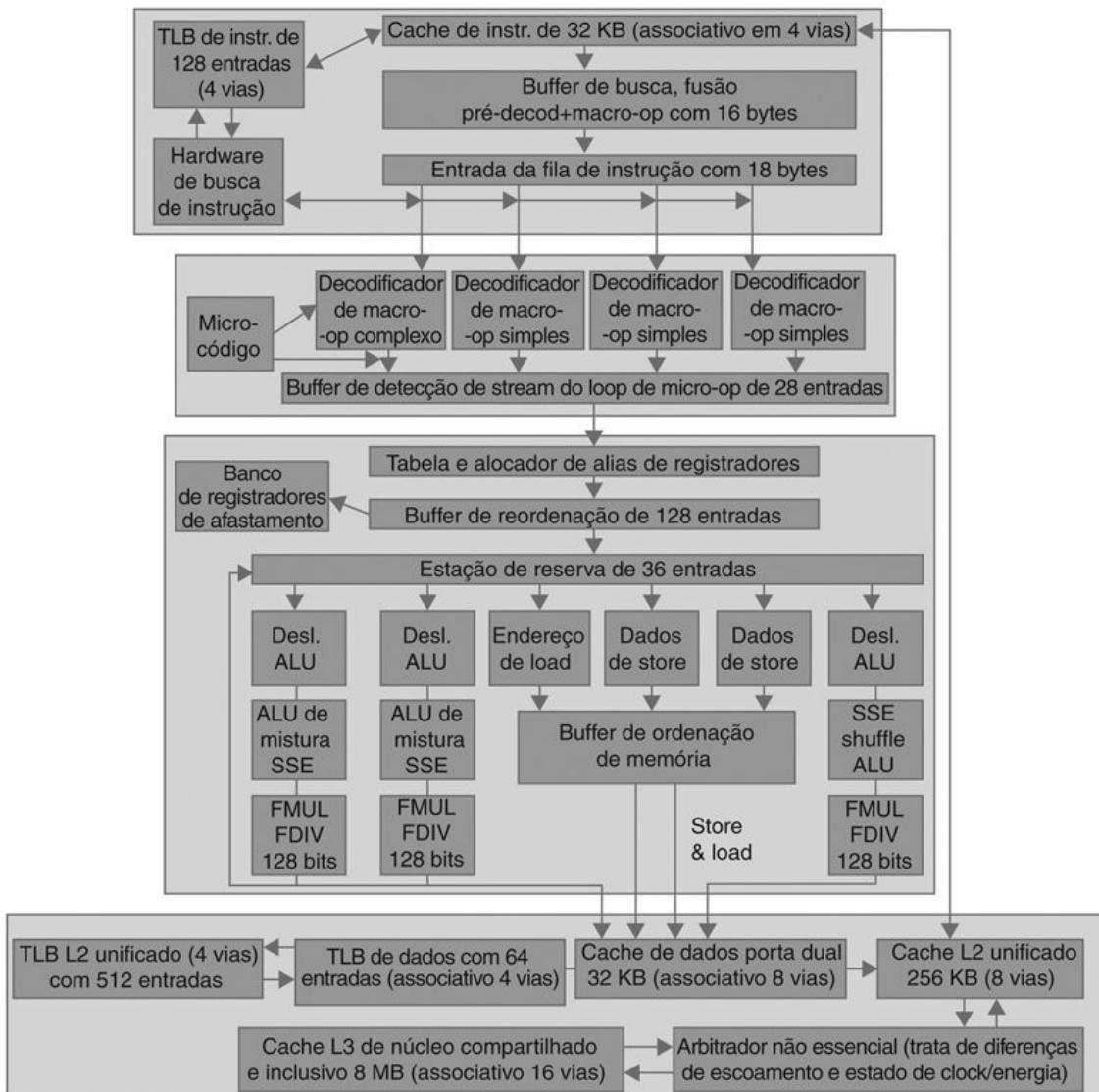
O Intel Core i7 utiliza um esquema para resolver as antidependências e a especulação incorreta, que usa um buffer de reordenação junto com a renomeação de registradores. A renomeação de registradores renomeia explicitamente os **registradores arquitetônicos** em um processador (16 no caso da versão de 64 bits da arquitetura x86) para um conjunto maior de registradores físicos. O Core i7 utiliza a renomeação de registradores para remover as antidependências. A renomeação de registradores exige que o processador mantenha um mapa entre os registradores arquitetônicos e os registradores físicos, indicando qual registrador físico é a cópia mais atual de um registrador arquitetônico. Registrando as renomeações que ocorreram, a técnica oferece outra forma de recuperação no caso de especulação incorreta: basta desfazer os mapeamentos que ocorreram desde a primeira instrução especulada incorretamente. Isso fará com que o estado do processador retorne à última instrução executada corretamente, mantendo o mapeamento correto entre os registradores arquitetônicos e físicos.

## registradores arquitetônicos

O conjunto de instruções dos registradores visíveis de um processador; por exemplo, no MIPS, existem 32 registradores de inteiros e 32 de ponto flutuante.

A [Figura 4.77](#) mostra a organização geral e o pipeline do Core i7. A seguir estão as oito etapas pelas quais uma instrução x86 passa para a sua execução.

1. Busca de instrução — O processador utiliza um buffer de destino de desvio multinível para obter um equilíbrio entre velocidade e exatidão na previsão. Há também uma pilha de endereços de retorno para agilizar o retorno de função. Falhas de previsão causam uma penalidade de aproximadamente 15 ciclos. Usando o endereço previsto, a unidade de busca de instruções lê 16 bytes da cache de instruções.



**FIGURA 4.77** O pipeline do Core i7 com componentes de memória.

A profundidade total do pipeline é de 14 estágios, com as falhas de previsão de desvio custando 17 ciclos de clock. Esse projeto pode manter em buffer 48 loads e 32 stores. As seis unidades independentes podem iniciar a execução de uma operação RISC pronta a cada ciclo de clock.

2. Os 16 bytes são colocados no buffer de instrução pré-decodificação — O estágio de pré-decodificação transforma os 16 bytes em instruções x86 individuais. Essa pré-decodificação é não trivial, pois o comprimento de uma instrução x86 pode ser de 1 a 15 bytes, e o pré-decodificador precisa percorrer uma série de bytes, antes de descobrir o comprimento da

instrução. As instruções x86 individuais são colocadas na filha de instruções com 18 entradas.

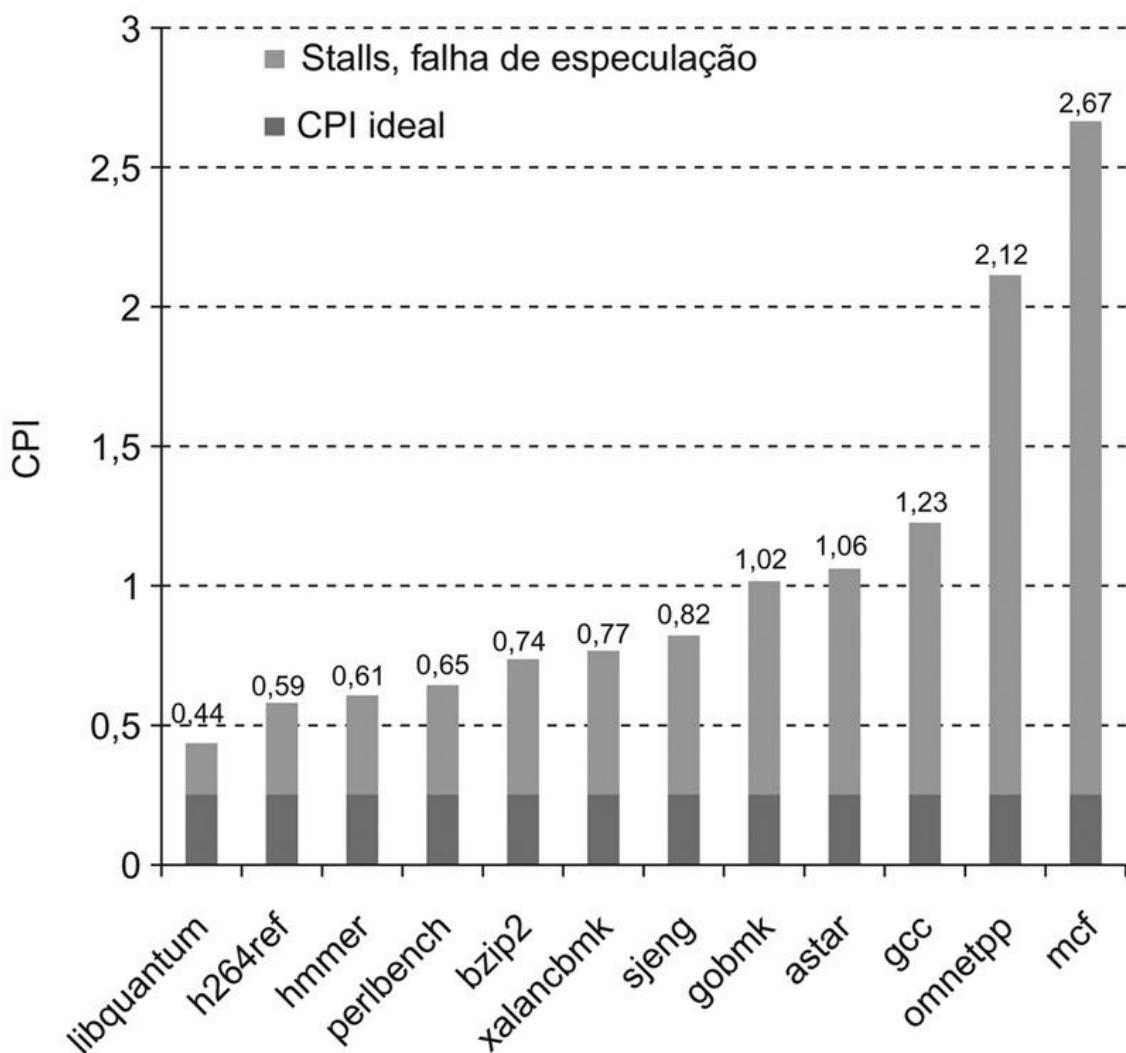
3. Decodificação de micro-operação — As instruções x86 individuais são traduzidas em micro-operações (micro-ops). Três dos decodificadores tratam das instruções x86 que se traduzem diretamente em uma micro-op. Para instruções x86 que possuem semântica mais complexa, existe um mecanismo de microcódigo usado para produzir a sequência de micro-op; ele pode produzir até quatro micro-ops a cada ciclo e continua até que a sequência de micro-ops necessária tenha sido gerada. As micro-ops são posicionadas de acordo com a ordem das instruções x86 no buffer de micro-ops com 28 entradas.
4. O buffer de micro-op realiza a *detecção de stream do loop* — Se houver uma pequena sequência de instruções (menos de 28 instruções ou 256 bytes de extensão) que compreende um loop, o detector de stream do loop encontrará o loop e despachará diretamente as micro-ops a partir do buffer, eliminando a necessidade de ativação dos estágios de busca e decodificação de instruções.
5. Realizar o despacho básico da instrução — Pesquisar o local do registrador nas tabelas de registradores, renomear os registradores, alocar uma entrada no buffer de reordenação e buscar quaisquer resultados dos registradores ou buffer de reordenação antes de enviar as micro-ops para as estações de reserva.
6. O i7 usa uma estação de reserva centralizada com 36 entradas, compartilhada por seis unidades funcionais. Até seis micro-ops podem ser despachadas para as unidades funcionais a cada ciclo de clock.
7. As unidades funcionais individuais executam micro-ops e depois os resultados são enviados de volta a qualquer estação de reserva aguardando, bem como para a unidade de afastamento de registrador, onde atualizarão o estado do registrador, quando se souber que a instrução não é mais especulativa. A entrada correspondente à instrução no buffer de reordenação é marcada como completa.
8. Quando uma ou mais instruções no início do buffer de reordenação forem marcadas como completas, as escritas pendentes na unidade de afastamento de registrador são executadas, e as instruções são removidas do buffer de reordenação.

## Detalhamento

O hardware na segunda e quarta etapas pode combinar ou *fundir* operações para reduzir o número de operações que precisam ser realizadas. A *fusão de macro-op* na segunda etapa exige que se combine instruções x86, como uma comparação seguida de um desvio, e que se junte em uma única operação. A *microfusão* na quarta etapa combina pares de micro-operações, como load/operação ALU e operação ALU/store, e os despacha para uma única estação de reserva (onde ainda poderão ser despachados independentemente), aumentando assim a utilização do buffer. Em um estudo da arquitetura Intel Core, que também incorporou a microfusão e a macrofusão, Bird et al. (2007) descobriram que a microfusão tinha pouco impacto sobre o desempenho, enquanto a macrofusão parece ter um impacto positivo moderado sobre o desempenho de inteiros e pouco impacto sobre o desempenho de ponto flutuante.

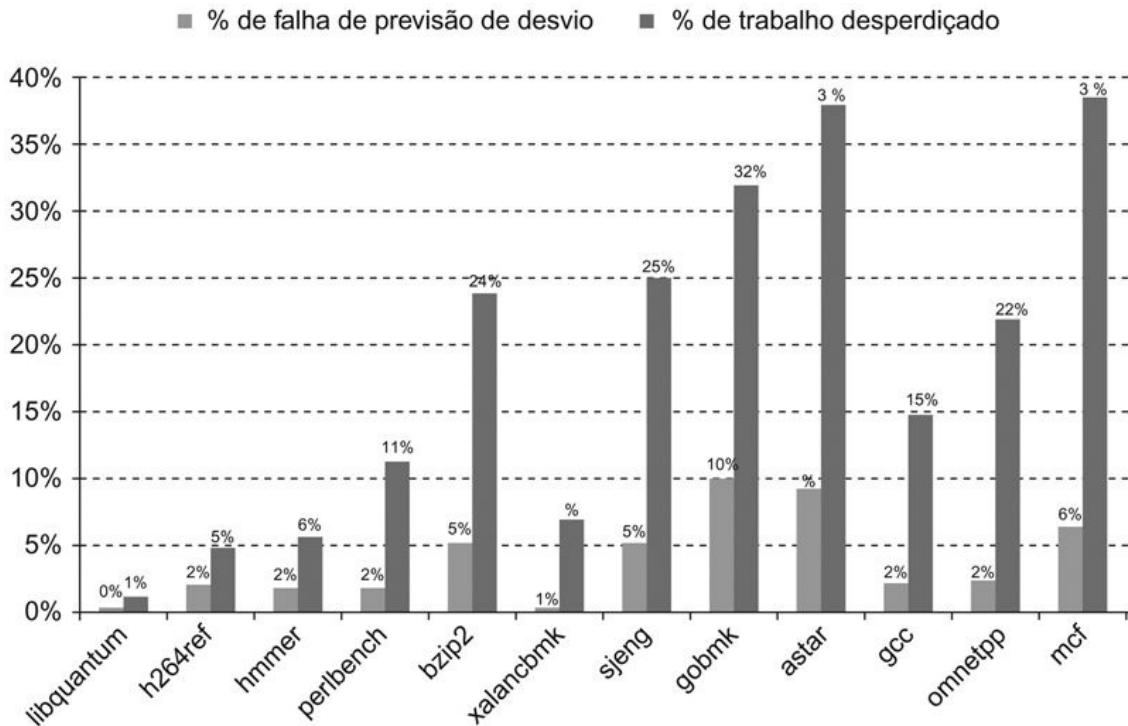
## Desempenho do Intel Core i7 920

A [Figura 4.78](#) mostra o CPI do Intel Core i7 para cada um dos benchmarks SPEC2006. Embora o CPI ideal seja 0,25, o melhor caso é 0,44, o caso mediano é 0,79 e o pior caso é 2,67.



**FIGURA 4.78** CPI do Intel Core i7 920 rodando benchmarks de inteiros do SPEC2006.

Embora seja difícil diferenciar entre stalls do pipeline e stalls da memória em um pipeline de execução dinâmica fora de ordem, podemos mostrar a eficácia da previsão de desvio e da especulação. A Figura 4.79 mostra a porcentagem dos desvios mal previstos e a porcentagem do trabalho (medida pelo número de micro-ops despachadas para o pipeline) que não se ausenta (ou seja, seus resultados são anulados) em relação a todos os despachos de micro-op. O mínimo, mediano e máximo das falhas de previsão de desvio são 0%, 2% e 10%. Para o trabalho desperdiçado, eles são 1%, 18% e 39%.



**FIGURA 4.79** Porcentagem de erros de previsão de desvio e trabalho desperdiçado devido à especulação improdutiva do Intel Core i7 920 rodando benchmarks de inteiros do SPEC2006.

O trabalho desperdiçado, em alguns casos, corresponde de perto às taxas de falha na previsão de desvio, como para os benchmarks gobmk e astar. Em vários casos, como em mcf, o trabalho desperdiçado parece ser relativamente maior do que a taxa de falha de previsão. Essa divergência provavelmente se deve ao comportamento da memória. Com taxas de falta de cache de dados muito altas, mcf despachará muitas instruções durante uma especulação incorreta, desde que haja estações de reserva suficientes à disposição para as referências de memória adiadas. Quando um desvio entre as muitas instruções especuladas for finalmente mal previsto, as micro-ops correspondentes a todas essas instruções sofrerão flush.

## Entendendo o desempenho dos programas

O Intel Core i7 combina um pipeline de 14 estágios e despacho múltiplo agressivo para conseguir alto desempenho. Mantendo baixas as latências para operações back to back, o impacto das dependências de dados é reduzido. Quais são os gargalos de desempenho em potencial mais sérios para os

programas executados nesse processador? A lista a seguir inclui alguns problemas de desempenho em potencial, com os três últimos podendo se aplicar, de alguma forma, a qualquer processador com pipeline de alto desempenho.

- O uso de instruções x86 que não são mapeadas para algumas micro-operações simples.
- Desvios que são difíceis de se prever, causando stalls e reinícios mal previstos quando a especulação falha.
- Dependências longas — normalmente causadas por instruções duradouras ou pela **hierarquia de memória** — que causam stalls.
- Atrasos de desempenho que surgem no acesso à memória (Capítulo 5), fazendo com que o processador sofra stall.



## 4.12. Mais rápido: Paralelismo em nível de instrução e multiplicação matricial

Retornando ao exemplo do DGEMM do Capítulo 3, podemos ver o impacto do paralelismo em nível de instrução desdobrando o loop de modo que o processador com execução de despacho múltiplo, fora de ordem, tenha mais

instruções com que trabalhar. A [Figura 4.80](#) mostra a versão desdoblada da [Figura 3.23](#), que contém os intrínsecos da linguagem C para produzir as instruções AVX.

```
1 #include <x86intrin.h>
2 #define UNROLL (4)
3
4 void dgemm (int n, double* A, double* B, double* C)
5 {
6     for ( int i = 0; i < n; i+=UNROLL*4 )
7         for ( int j = 0; j < n; j++ ) {
8             __m256d c[4];
9             for ( int x = 0; x < UNROLL; x++ )
10                 c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12             for( int k = 0; k < n; k++ )
13             {
14                 __m256d b = _mm256_broadcast_sd(B+k+j*n);
15                 for (int x = 0; x < UNROLL; x++)
16                     c[x] = _mm256_add_pd(c[x],
17                                         _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
18             }
19
20             for ( int x = 0; x < UNROLL; x++ )
21                 _mm256_store_pd(C+i+x*4+j*n, c[x]);
22         }
23 }
```

**FIGURA 4.80** Versão C otimizada do DGEMM usando intrínsecos da linguagem C para gerar as instruções paralelas de subword AVX para o x86 ([Figura 3.23](#)) e desdobramento de loop para criar mais oportunidades de paralelismo em nível de instrução.

A [Figura 4.81](#) mostra o código em linguagem assembly produzido pelo compilador para o loop mais interno, que desdobra os três corpos de loop for a fim de expor o paralelismo em nível de instrução.

Assim como o exemplo de desdobramento na [Figura 4.71](#), vamos desdobrar o loop 4 vezes. (Usamos a constante UNROLL no código C para controlar a quantidade de desdobramento caso queiramos experimentar outros valores.) Em vez de desdobrar manualmente o loop em C fazendo 4 cópias de cada um dos intrínsecos da [Figura 3.23](#), podemos contar com o compilador gcc para fazer o desdobramento na otimização -O3. Delimitamos cada intrínseco com um loop

*for* simples com 4 iterações (linhas 9, 14 e 20) e substituímos o escalar `c0` da [Figura 3.23](#) por um array de 4 elementos `c[]` (linhas 8, 10, 16 e 21).

A [Figura 4.81](#) mostra a saída em linguagem assembly do código desdobrado. Conforme esperado, na [Figura 4.81](#), existem 4 versões de cada uma das instruções AVX da [Figura 3.24](#), com uma exceção. Só precisamos de uma cópia da instrução `vbroadcastsd`, pois podemos usar as quatro cópias do elemento B no registrador `%ymm0` repetidamente por todo o loop. Assim, as 5 instruções AVX da [Figura 3.24](#) tornam-se 17 na [Figura 4.81](#), e as 7 instruções de inteiros aparecem em ambas, embora as constante e o endereçamento mudem para levar em conta o desdobramento. Portanto, apesar de desdobrar 4 vezes, o número de instruções no corpo do loop só dobra: de 12 para 24.

```

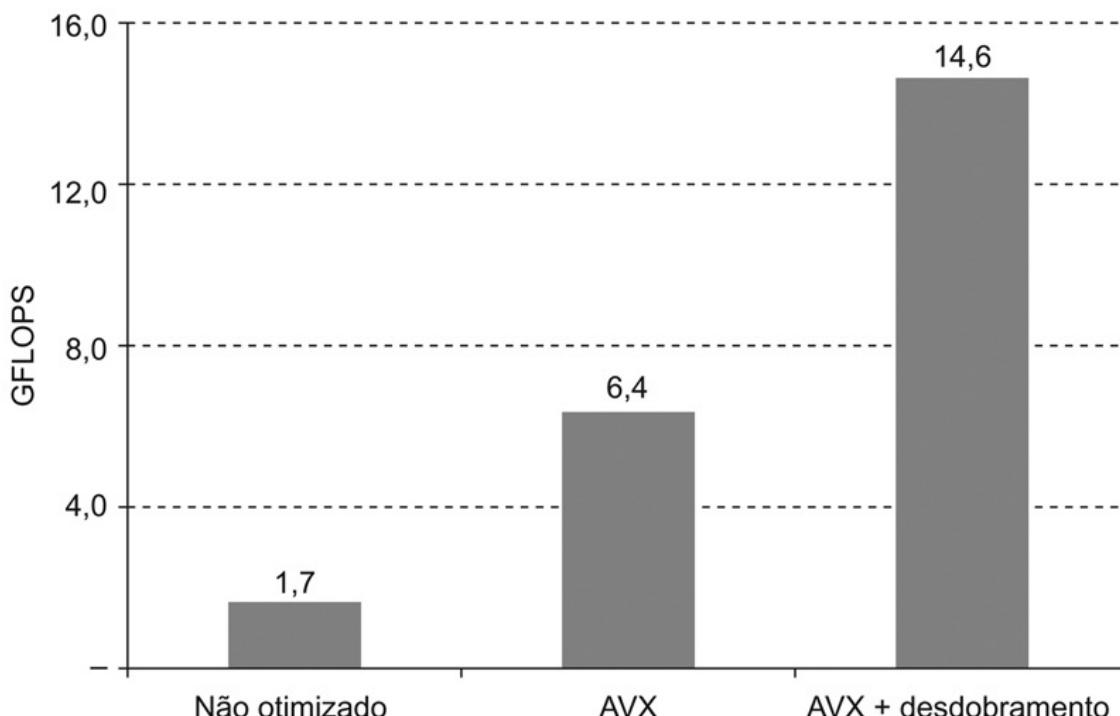
1  vmovapd (%r11),%ymm4          # Lê 4 elementos de C para %ymm4
2  mov    %rbx,%rax              # Registrador %rax = %rbx
3  xor    %ecx,%ecx              # Registrador %ecx = 0
4  vmovapd 0x20(%r11),%ymm3      # Lê 4 elementos de C para %ymm3
5  vmovapd 0x40(%r11),%ymm2      # Lê 4 elementos de C para %ymm2
6  vmovapd 0x60(%r11),%ymm1      # Lê 4 elementos de C para %ymm1
7  vbroadcastsd (%rcx,%r9,1),%ymm0  # Faz 4 cópias do elemento B
8  add    $0x8,%rcx              # Registrador %rcx = %rcx + 8
9  vmulpd (%rax),%ymm0,%ymm5      # Mul paralelo de %ymm1,4 elementos A
10 vaddpd %ymm5,%ymm4,%ymm4      # Add paralelo de %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5  # Mul paralelo de %ymm1,4 elementos A
12 vaddpd %ymm5,%ymm3,%ymm3      # Add paralelo de %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5  # Mul paralelo de %ymm1,4 elementos A
14 vmulpd 0x60(%rax),%ymm0,%ymm0  # Mul paralelo de %ymm1,4 elementos A
15 add    %r8,%rax              # Registrador %rax = %rax + %r8
16 cmp    %r10,%rcx              # Compara %r8 com %rax
17 vaddpd %ymm5,%ymm2,%ymm2      # Add paralelo de %ymm5, %ymm2
18 vaddpd %ymm0,%ymm1,%ymm1      # Add paralelo de %ymm0, %ymm1
19 jne    68 <dgemm+0x68>        # Salta se não %r8 != %rax
20 add    $0x1,%esi              # Registrador %esi = %esi + 1
21 vmovapd %ymm4,(%r11)          # Salva %ymm4 em 4 elementos C
22 vmovapd %ymm3,0x20(%r11)      # Salva %ymm3 em 4 elementos C
23 vmovapd %ymm2,0x40(%r11)      # Salva %ymm2 em 4 elementos C
24 vmovapd %ymm1,0x60(%r11)      # Salva %ymm1 em 4 elementos C

```

**FIGURA 4.81** A linguagem assembly x86 para o corpo dos

**loops aninhados gerados pela compilação do código C desdobrado na Figura 4.80.**

A Figura 4.82 mostra o aumento de desempenho do DGEMM para matrizes  $32 \times 32$  passando de não otimizado para AVX e depois para AVX com desdobramento. O desdobramento mais do que duplica o desempenho, passando de 6,4 GFLOPS para 14,6 GFLOPS. As otimizações para **paralelismo de subword** e **paralelismo em nível de instrução** resultam em um ganho de velocidade geral de 8,8 *versus* o DGEMM não otimizado da Figura 3.21.



**FIGURA 4.82 Desempenho de três versões do DGEMM para matrizes  $32 \times 32$ .**

O paralelismo de subword e o paralelismo em nível de instrução levaram a um ganho de velocidade de quase 9 em relação ao código não otimizado da Figura 3.21.

## Detalhamento

Como dissemos no Detalhamento da Seção 3.8, esses resultados são com o modo Turbo desativado. Se o ativarmos, como no Capítulo 3, melhoramos

todos os resultados pelo aumento temporário na taxa de clock de  $3,3/2,6 = 1,27$ , passando a 2,1 GFLOPS para o DGEMM não otimizado, 8,1 GFLOPS com AVX e 18,6 GFLOPS com desdobramento e AVX. Como dissemos na Seção 3.8, o modo Turbo funciona particularmente bem nesse caso, porque está usando apenas um único núcleo de um chip de oito núcleos.

## Detalhamento

Não existem stalls de pipeline apesar da reutilização do registrador `%ymm5` nas linhas de 9 a 17 da Figura 4.81, pois o pipeline do Intel Core i7 renomeia os registradores.

## Verifique você mesmo

Indique se cada uma das afirmações a seguir é verdadeira ou falsa.

1. O Intel Core i7 utiliza um pipeline com despacho múltiplo para executar as instruções X86 diretamente.
2. Tanto o A8 quanto o Core i7 utilizam o despacho múltiplo dinâmico.
3. A microarquitetura do Core i7 possui muito mais registradores do que o x86 exige.
4. O Intel Core i7 usa menos de metade dos estágios do pipeline do Intel Pentium 4 Prescott mais antigo (Figura 4.73).

## 4.13. Falácia e armadilhas

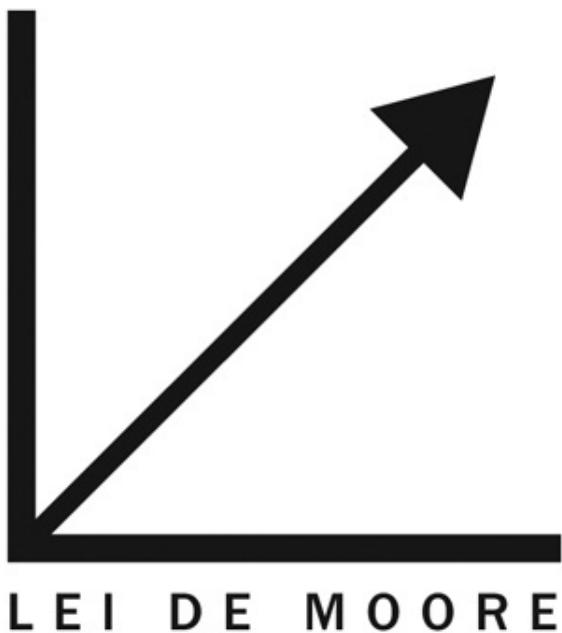
Falácia: Pipelining é fácil.

Nossos livros comprovam a sutileza da execução correta de um pipeline. Nossa livro avançado tinha um bug no pipeline em sua primeira edição, apesar de ter sido revisado por mais de 100 pessoas e testado nas salas de aula de 18 universidades. O bug só foi descoberto quando alguém tentou montar um computador com aquele livro. O fato de que o Verilog para descrever um pipeline como esse do Intel Core i7 terá milhares de linhas é uma indicação da complexidade. Esteja atento!

Falácia: As ideias de pipelining podem ser implementadas independentes da

tecnologia.

Quando o número de transistores no chip e a velocidade dos transistores tornaram um pipeline de cinco estágios a melhor solução, então o delayed branch (veja o primeiro “*Detalhamento*” da Seção “Previsão dinâmica de desvios”) foi uma solução simples para controlar os hazards. Com pipelines maiores, a execução superescalar e a previsão dinâmica de desvios, isso agora é redundante. No início da década de 1990, o escalonamento dinâmico em pipeline exigia muitos recursos e não era necessário para o alto desempenho, mas, à medida que a quantidade de transistores continuava a dobrar, devido à **Lei de Moore**, a lógica se tornava muito mais rápida do que a memória, então as múltiplas unidades funcionais e os pipelines dinâmicos fizeram mais sentido. Hoje, a preocupação com a potência está levando a projetos menos agressivos.



Armadilha: A falha em considerar o projeto do conjunto de instruções pode afetar o pipeline de forma adversa.

Muitas das dificuldades em pipelining surgem por causa das complicações do

conjunto de instruções. Aqui estão alguns exemplos:

- Tamanhos de instrução e tempos de execução muito variáveis podem causar desequilíbrio entre estágios do pipeline e complicar bastante a detecção de hazards em um projeto com pipeline, no nível do conjunto de instruções. Esse problema foi contornado, inicialmente no DEC VAX 8500, no final da década de 1980, usando o esquema de micro-operações e micropipeline que o Intel Core i7 emprega hoje. Naturalmente, o overhead da tradução e a manutenção da correspondência entre as micro-operações e as instruções permanecem.
- Modos de endereçamento sofisticados podem levar a diferentes tipos de problemas. Os modos de endereçamento que atualizam registradores complicam a detecção de hazards. Outros modos de endereçamento que exigem múltiplos acessos à memória complicam bastante o controle do pipeline e tornam difícil manter o pipeline fluindo tranquilamente.
- Talvez o melhor exemplo seja o DEC Alpha e o DEC NVAX. Em uma tecnologia comparável, o conjunto de instruções mais recente do Alpha permitiu uma implementação cujo desempenho tem mais do que o dobro da velocidade do NVAX. Em outro exemplo, Bhandarkar e Clark [1991] compararam o MIPS M/2000 e o DEC VAX 8700 contando os ciclos de clock dos benchmarks SPEC; eles concluíram que, embora o MIPS M/2000 execute mais instruções, o VAX na média executa 2,7 vezes mais ciclos de clock, de modo que o MIPS é mais rápido.

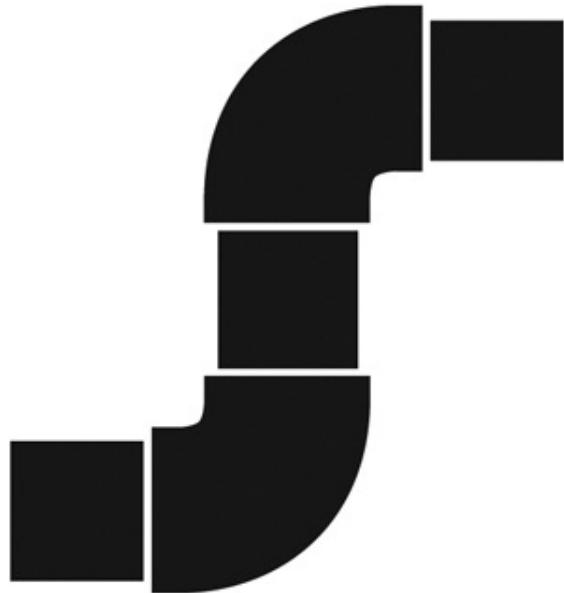
## 4.14. Comentários finais

*Noventa por cento da sabedoria consiste em ser sensato no tempo.*

*Provérbio americano*

Como vimos neste capítulo, tanto o caminho de dados quanto o controle para um processador podem ser projetados começando com a arquitetura do conjunto de instruções e o conhecimento das características básicas da tecnologia. Na [Seção 4.3](#), vimos como o caminho de dados para um processador MIPS poderia ser construído com base na arquitetura e na decisão de criar uma implementação de ciclo único. Naturalmente, a tecnologia básica também afeta muitas decisões de projeto, ditando quais componentes podem ser usados no caminho de dados e também se uma implementação de ciclo único sequer faz sentido.

A técnica de **pipelining** melhora a vazão, mas não o tempo de execução inerente (ou **latência de instrução**) das instruções; para algumas instruções, a latência é semelhante, em duração, à técnica de ciclo único. O despacho de instrução múltiplo acrescenta um hardware adicional ao caminho de dados para permitir que várias instruções sejam iniciadas a cada ciclo de clock, mas com um aumento na latência efetiva. O pipelining foi apresentado como capaz de reduzir o tempo de ciclo de clock do caminho de dados do ciclo único simples. O despacho múltiplo de instruções, em comparação, focaliza claramente na redução dos *ciclos de clock por instrução* (CPI).



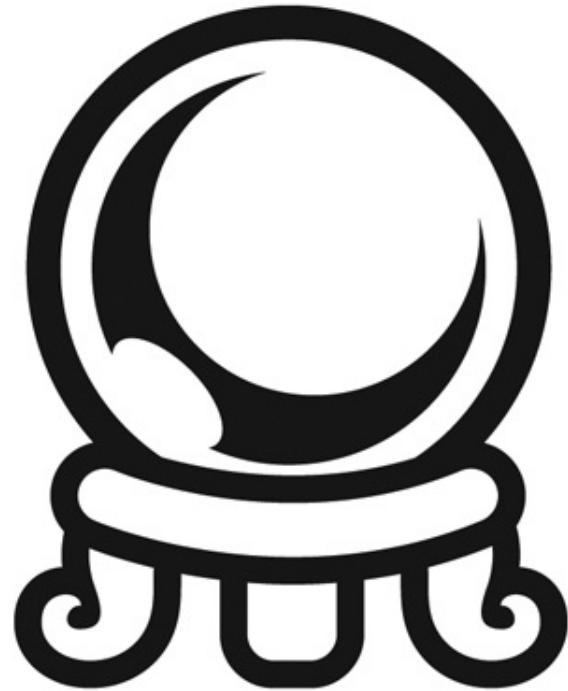
## PIPELINING

### latência de instrução

O tempo de execução inerente para uma instrução.

A técnica de pipelining e o despacho múltiplo tentam explorar o paralelismo em nível de instrução. A presença de dependências de dados e de controle, que podem se tornar hazards, são as principais limitações para a exploração do paralelismo. Escalonamento e especulação por **predição**, tanto no hardware

quanto no software, são as principais técnicas utilizadas para reduzir o impacto das dependências sobre o desempenho.



## P R E D I Ç Ã O

Mostramos que o desdobramento do loop DGEMM expõe quatro vezese mais instruções que poderiam tirar proveito do mecanismo de execução fora de ordem do Core i7 para mais do que dobrar o desempenho.

A passagem para pipelines maiores, despacho de instruções múltiplas e escalonamento dinâmico em meados da década de 1990 ajudou a sustentar os 60% de aumento anual de desempenho dos processadores que começou no início da década de 1980. Como dissemos no [Capítulo 1](#), esses microprocessadores preservaram o modelo de programação sequencial, mas por fim se chocaram com o muro da potência. Assim, a indústria foi forçada a testar multiprocessadores, que exploram o paralelismo em níveis menos minuciosos (o assunto do [Capítulo 6](#)). Essa tendência também fez com que os projetistas reavaliassem as implicações de desempenho da potência de algumas invenções desde meados da década de 1990, resultando em uma simplificação dos pipelines

em versões mais recentes das microarquiteturas.

Para sustentar os avanços no desempenho de processamento por meio de processadores paralelos, a lei de Amdahl sugere que outra parte do sistema se torne o gargalo. Esse gargalo é o assunto do próximo capítulo: a **hierarquia da memória**.



## 4.15. Exercícios

**4.1** Considere a seguinte instrução:

Instrução: AND Rd, Rs, Rt

Interpretação:  $\text{Reg}[\text{Rd}] = \text{Reg}[\text{Rs}] \text{ AND } \text{Reg}[\text{Rt}]$

**4.1.1** [5] <§4.1> Quais são os valores dos sinais de controle gerados pelo controle na [Figura 4.2](#) para esta instrução?

**4.1.2** [5] <§4.1> Quais recursos (blocos) realizam uma função útil para essa instrução?

**4.1.3** [10] <§4.1> Quais recursos (blocos) produzem saídas, mas suas saídas não são usadas para essa instrução? Quais recursos não produzem saídas para ela?

**4.2** A implementação básica de ciclo único do MIPS na [Figura 4.2](#) só pode implementar algumas instruções. Novas instruções podem ser acrescentadas

a uma ISA (Instruction Set Architecture) existente, mas a decisão de fazer isso ou não depende, entre outras coisas, do custo e da complexidade que tal acréscimo introduz no caminho de dados e controle do processador. Os três primeiros problemas neste exercício referem-se a esta nova instrução:

Instrução: LWI Rt, Rd(Rs)

Interpretação:  $\text{Reg}[\text{Rt}] = \text{Mem}[\text{Reg}[\text{Rd}] + \text{Reg}[\text{Rs}]]$

**4.2.1 [10] <§4.1>** Quais blocos existentes (se houver) podem ser usados para essa instrução?

**4.2.2 [10] <§4.1>** De quais novos blocos funcionais (se houver) precisamos para essa instrução?

**4.2.3 [10] <§4.1>** De quais novos sinais da unidade de controle (se houver) precisamos para dar suporte a essa instrução?

**4.3** Quando os projetistas de processador consideram uma melhoria possível no caminho de dados do processador, a decisão normalmente depende da escolha de custo/desempenho. Nos três problemas a seguir, considere que estamos começando com um caminho de dados da [Figura 4.2](#), em que os blocos I-Mem, Add, Mux, ALU, Regs, D-Mem e de Controle têm latências de 400 ps, 100 ps, 30 ps, 120 ps, 200 ps, 350 ps e 100 ps, respectivamente, e custos de 1000, 30, 10, 100, 200, 2000 e 500, respectivamente.

Considere o acréscimo de um multiplicador à ALU. Esse acréscimo somará 300 ps à latência da ALU e acrescentará um custo de 600 à ALU. O resultado será 5% menos instruções executadas, pois não precisaremos mais emular a instrução MUL.

**4.3.1 [10] <§4.1>** Qual é o tempo de ciclo de clock com e sem essa melhoria?

**4.3.2 [10] <§4.1>** Qual é o ganho de velocidade obtido acrescentando essa melhoria?

**4.3.3 [10] <§4.1>** Compare a razão custo/desempenho com e sem essa melhoria.

**4.4** Os problemas neste exercício consideram que os blocos lógicos necessários para implementar o caminho de dados de um processador têm as seguintes latências:

I-Mem	Add	Mux	ALU	Regs	D-Mem	Extensão de sinal	Shift-esq-2
200 ps	70 ps	20 ps	90 ps	90 ps	250 ps	15 ps	10 ps

**4.4.1 [10] <§4.3>** Se a única coisa que precisássemos fazer em um processador fosse buscar instruções consecutivas ([Figura 4.6](#)), qual seria o tempo do ciclo?

**4.4.2 [10] <§4.3>** Considere um caminho de dados semelhante ao da [Figura 4.11](#), mas para um processador que só tem um tipo de instrução: desvio incondicional relativo ao PC. Qual seria o tempo de ciclo para esse caminho de dados?

**4.4.3 [10] <§4.3>** Repita o Exercício 4.4.2, mas desta vez precisamos dar suporte apenas a desvios *condicionais* relativos ao PC.

Os três problemas restantes neste exercício referem-se ao elemento Shift-esq-2 no caminho de dados:

**4.4.4 [10] <§4.3>** Quais tipos de instruções exigem esse recurso?

**4.4.5 [20] <§4.3>** Para que tipos de instruções (se houver) esse recurso está no caminho crítico?

**4.4.6 [10] <§4.3>** Supondo que só temos suporte para instruções beq e add, discuta como as mudanças na latência indicada desse recurso afetam o tempo de ciclo do processador. Suponha que as latências de outros recursos não mudem.

**4.5** Para os problemas neste exercício, considere que não existem stalls de pipeline e que o desmembramento das instruções executadas seja o seguinte:

add	addi	not	beq	lw	sw
20%	20%	0%	25%	25%	10%

**4.5.1 [10] <§4.3>** Em que fração de todos os ciclos a memória de dados é utilizada?

**4.5.2 [10] <§4.3>** Em que fração de todos os ciclos a entrada do circuito por extensão de sinal é necessária? O que esse circuito está fazendo nos ciclos em que sua entrada não é necessária?

**4.6** Quando os chips de silício são fabricados, os defeitos nos materiais (por exemplo, o silício) e os erros de manufatura podem resultar em circuitos defeituosos. Um defeito muito comum é quando um fio afeta o sinal em outro. Isso é chamado de falha cross-talk. Uma classe especial de falhas cross-talk é quando um sinal está conectado a um fio que tem um valor lógico constante (por exemplo, um fio da fonte de alimentação). Nesse caso, temos uma falha stuck-at-0 ou stuck-at-1, e o sinal afetado sempre tem um valor lógico 0 ou 1, respectivamente. Os problemas a seguir referem-se ao bit 0 do Registrador Escrita no banco de registradores da [Figura 4.24](#).

**4.6.1 [10] <§§4.3, 4.4>** Vamos supor que o teste do processador seja feito preenchendo o PC, registradores e memórias de dados e instruções com alguns valores (você pode escolher quais valores), permitindo que uma única

instrução seja executada e depois lendo o PC, memórias e registradores. Esses valores são então examinados para determinar se uma falha em particular está presente. Você conseguiria criar um teste (valores para PC, memórias e registradores) que determinaria se existe uma falha stuck-at-0 nesse sinal?

**4.6.2 [10] <§§4.3, 4.4>** Repita o Exercício 4.6.1 para uma falha stuck-at-1.

Você conseguiria usar um único teste para stuck-at-0 e stuck-at-1? Caso afirmativo, explique como; se não, explique por que não.

**4.6.3 [60] <§§4.3, 4.4>** Se soubermos que o processador tem uma falha stuck-at-1 nesse sinal, o processador ainda é utilizável? Para isso, temos de converter qualquer programa que execute em um processador MIPS normal em um programa que funcione nesse processador. Você pode considerar que existe memória de instrução e memória de dados livre suficiente para tornar o programa maior e armazenar dados adicionais. Dica: o processador é utilizável se cada instrução “rompida” por essa falha puder ser substituída por uma sequência de instruções “funcionais” que conseguem o mesmo efeito.

**4.6.4 [10] <§§4.3, 4.4>** Repita o Exercício 4.6.1, mas agora o teste é se o sinal de controle “MemRead” torna-se 0 se o sinal de controle RegDst for 0; caso contrário, nenhuma falha.

**4.6.5 [10] <§§4.3, 4.4>** Repita o Exercício 4.6.4, mas agora o teste é se o sinal de controle “Jump” torna-se 0 se o sinal de cont RegDst for 0; caso contrário, nenhuma falha.

**4.7** Neste exercício, examinamos detalhadamente como uma instrução é executada em um caminho de dados de ciclo único. Os problemas neste exercício referem-se a um ciclo de clock em que o processador busca a seguinte word de instrução:

101011000110001000000000000010100

Considere que a memória de dados contém apenas zeros e que os registradores do processador possuem os seguintes valores no início do ciclo em que a word de instrução anterior é apanhada:

r0	r1	r2	r3	r4	r5	r6	r8	r12	r31
0	-1	2	-3	-4	10	6	8	2	-16

**4.7.1 [5] <§4.4>** Quais são as saídas da unidade de extensão de sinal e salto “Shift left 2” (topo da [Figura 4.24](#)) para essa palavra de instrução?

**4.7.2 [10] <§4.4>** Quais são os valores das entradas da unidade de controle da

ALU para essa instrução?

**4.7.3 [10] <§4.4>** Qual é o novo endereço do PC após a execução dessa instrução? Destaque o caminho através do qual esse valor é determinado.

**4.7.4 [10] <§4.4>** Para cada Mux, mostre os valores de sua saída de dados durante a execução dessa instrução e esses valores de registrador.

**4.7.5 [10] <§4.4>** Para a ALU e as duas unidades de soma, quais são seus valores de entrada de dados?

**4.7.6 [10] <§4.4>** Quais são os valores de todas as entradas para a unidade de “Registradores”?

**4.8** Neste exercício, examinamos como o pipelining afeta o tempo do ciclo de clock do processador. Os problemas neste exercício consideram que os estágios individuais do caminho de dados têm as seguintes latências:

IF	ID	EX	MEM	WB
250 ps	350 ps	150 ps	300 ps	200 ps

Além disso, considere que as instruções executadas pelo processador são desmembradas da seguinte forma:

alu	beq	lw	sw
45%	20%	20%	15%

**4.8.1 [5] <§4.5>** Qual é o tempo do ciclo de clock em um processador com e sem pipeline?

**4.8.2 [10] <§4.5>** Qual é a latência total de uma instrução LW em um processador com e sem pipeline?

**4.8.3 [10] <§4.5>** Se pudessemos dividir um estágio do caminho de dados com pipeline em dois novos estágios, cada um com metade da latência do estágio original, que estágio dividiríamos e qual é o novo tempo do ciclo de clock do processador?

**4.8.4 [10] <§4.5>** Supondo que não haja stalls ou hazards, qual é a utilização da memória de dados?

**4.8.5 [10] <§4.5>** Supondo que não haja stalls ou hazards, qual é a utilização da porta de escrita de registrador da unidade “Registradores”?

**4.8.6 [30] <§4.5>** Em vez de uma organização de ciclo único, podemos usar uma organização multiciclos, em que cada instrução ocupa múltiplos ciclos, mas uma instrução termina antes que outra seja apanhada. Nessa organização, uma instrução só percorre os estágios que ela realmente precisa

(por exemplo, ST só ocupa quatro ciclos, pois não precisa do estágio WB). Compare os tempos do ciclo de clock e os tempos de execução com a organização em ciclo único, multiciclos e em pipeline.

**4.9** Neste exercício, examinamos como as dependências de dados afetam a execução no pipeline básico de cinco estágios descrito na [Seção 4.5](#). Os problemas neste exercício referem-se a esta sequência de instruções:

```
or r1,r2,r3
or r2,r1,r4
or r1,r1,r2
```

Além disso, considere os seguintes tempos do ciclo de clock para cada uma das opções relacionadas ao forwarding:

Sem forwarding	Com forwarding completo	Apenas com forwarding ALU-ALU
250 ps	300 ps	290 ps

- 4.9.1** [10] <§4.5> Indique as dependências e seu tipo.
- 4.9.2** [10] <§4.5> Suponha que não haja forwarding nesse processador em pipeline. Indique hazards e acrescente instruções nop para eliminá-los.
- 4.9.3** [10] <§4.5> Suponha que haja forwarding completo. Indique os hazards e acrescente instruções nop para eliminá-los.
- 4.9.4** [10] <§4.5> Qual é o tempo de execução total dessa sequência de instruções sem forwarding e com forwarding completo? Qual é o ganho de velocidade obtido, acrescentando-se forwarding completo a um pipeline que não tinha forwarding?
- 4.9.5** [10] <§4.5> Acrescente instruções nop a esse código para eliminar hazards se houver apenas forwarding ALU-ALU (nenhum forwarding do estágio MEM para EX).
- 4.9.6** [10] <§4.5> Qual é o tempo de execução total dessa sequência de instruções apenas com forwarding ALU-ALU? Qual é o ganho de velocidade em relação a um pipeline sem forwarding?
- 4.10** Neste exercício, examinamos como os hazards de recursos, os hazards de

controle e o projeto da ISA podem afetar a execução em pipeline. Os problemas neste exercício referem-se ao seguinte fragmento de código MIPS:

```
sw r16,12(r6)
lw r16,8(r6)
beq r5,r4,Label # Considere que r5!=r4
add r5,r1,r4
slt r5,r15,r4
```

Considere que os estágios de pipeline individuais possuem as seguintes latências:

IF	ID	EX	MEM	WB
200 ps	120 ps	150 ps	190 ps	100 ps

**4.10.1 [10] <§4.5>** Para este problema, suponha que todos os desvios sejam perfeitamente previstos (isso elimina todos os hazards de controle) e que nenhum slot de delay seja utilizado. Se tivermos apenas uma memória (para instruções e dados), haverá um hazard estrutural toda vez que precisarmos apanhar uma instrução no mesmo ciclo em que outra instrução acessa dados. Para garantir o processo do forwarding, esse hazard sempre precisa ser resolvido em favor da instrução que acessa dados. Qual é o tempo de execução total dessa sequência de instruções no pipeline de cinco estágios que tem apenas uma memória? Vimos que os hazards de dados podem ser eliminados acrescentando nops ao código. Você conseguiria fazer o mesmo com esse hazard estrutural? Por quê?

**4.10.2 [20] <§4.5>** Para este problema, suponha que todos os desvios sejam perfeitamente previstos (isso elimina todos os hazards de controle) e que nenhum slot de delay seja utilizado. Se mudarmos as instruções load/store para usar um registrador (sem um offset) como endereço, essas instruções não precisam mais usar a ALU. Como resultado, os estágios MEM e EX podem ser sobrepostos e o pipeline tem apenas quatro estágios. Mude esse código para acomodar essa ISA alterada. Supondo que essa mudança não

afete o tempo do ciclo de clock, que ganho de velocidade é obtido nessa sequência de instruções?

**4.10.3** [10] <§4.5> Considerando stall-on-branch e nenhum slot de delay, que ganho de velocidade é obtido nesse código se os resultados do desvio forem determinados no estágio ID, em relação à execução em que os resultados do desvio são determinados no estágio EX?

**4.10.4** [10] <§4.5> Dadas essas latências de estágio de pipeline, repita o cálculo de ganho de velocidade de 4.10.2, mas leve em conta a (possível) mudança no tempo do ciclo de clock. Quando EX e MEM são feitos em um único estágio, a maior parte do trabalho pode ser feita em paralelo. Como resultado, o estágio EX/MEM resultante tem uma latência que é a maior das duas originais, mais 20 ps necessários para o trabalho que poderia ser feito em paralelo.

**4.10.5** [10] <§4.5> Dadas essas latências de estágio em pipeline, repita o cálculo de ganho de velocidade de 4.10.3, mas leve em conta a (possível) mudança no tempo do ciclo de clock. Suponha que a latência do estágio ID aumente em 50% e a latência do estágio EX diminua em 10 ps quando a resolução do resultado do desvio é passada de EX para ID.

**4.10.6** [10] <§4.5> Considerando stall-on-branch e nenhum slot de delay, qual é o novo tempo do ciclo de clock e tempo de execução dessa sequência de instruções se o cálculo de endereço de beq for passado para o estágio MEM? Qual é o ganho de velocidade decorrente dessa mudança? Suponha que a latência do estágio EX seja reduzida em 20 ps e a latência do estágio MEM fique inalterada quando a resolução do resultado do desvio for passada de EX para MEM.

**4.11** Considere o loop a seguir.

```

loop: lw r1,0(r1)
      and r1,r1,r2
      lw r1,0(r1)
      lw r1,0(r1)
      beq r1,r0,loop

```

Considere que a previsão de desvio perfeita é utilizada (sem stalls devido aos hazards de controle), que não existem slots de delay e que o pipeline possui suporte para forwarding completo. Considere também que muitas iterações desse loop são executadas antes que o loop termine.

**4.11.1 [10] <§4.6>** Mostre um diagrama de execução de pipeline para a terceira iteração desse loop, do ciclo em que apanhamos a primeira instrução dessa iteração até (mas não incluindo) o ciclo em que apanhamos a primeira instrução da iteração seguinte. Mostre todas as instruções que estão no pipeline durante esses ciclos (não apenas aquelas da terceira iteração).

**4.11.2 [10] <§4.6>** Com que frequência (como uma porcentagem de todos os ciclos) temos um ciclo em que todos os cinco estágios do pipeline estão realizando trabalho útil?

**4.12** Este exercício tem por finalidade ajudá-lo a entender as escolhas entre custo/complexidade/desempenho do forwarding em um processador com pipeline. Os problemas neste exercício referem-se aos caminhos de dados em pipeline da [Figura 4.45](#). Esses problemas consideram que, de todas as instruções executadas em um processador, a fração dessas instruções a seguir tem um tipo particular de dependência de dados RAW. O tipo de dependência de dados RAW é identificado pelo estágio que produz o resultado (EX ou MEM) e a instrução que consome o resultado (1<sup>a</sup> instrução que segue aquela que produz o resultado, 2<sup>a</sup> instrução que a segue ou ambas). Consideramos que a escrita do registrador é feita na primeira metade do ciclo de clock e que as leituras do registrador são feitas na segunda metade do

ciclo, de modo que dependências “EX para 3<sup>a</sup>” e “MEM para 3<sup>a</sup>” não são contadas, pois não podem resultar em hazards de dados. Além disso, considere que o CPI do processador é 1 se não houver hazards de dados.

EX para 1 <sup>a</sup> somente	MEM para 1 <sup>a</sup> somente	EX para 2 <sup>a</sup> somente	MEM para 2 <sup>a</sup> somente	EX para 1 <sup>a</sup> e MEM para 2 <sup>a</sup>	Outras dependências RAW
5%	20%	5%	10%	10%	10%

Considere as seguintes latências para estágios individuais do pipeline. No estágio EX, as latências são dadas separadamente para um processador sem forwarding e um processador com diferentes tipos de forwarding.

IF	ID	EX (sem FW)	EX (FW completo)	EX (FW apenas de EX/MEM)	EX (FW apenas de MEM/WB)	MEM	WB
150 ps	100 ps	120 ps	150 ps	140 ps	130 ps	120 ps	100 ps

- 4.12.1 [10] <§4.7>** Se não usarmos forwarding, em que fração dos ciclos estamos realizando stall devido aos hazards de dados?
- 4.12.2 [5] <§4.7>** Se usarmos o forwarding completo (encaminhar todos os resultados que podem ser encaminhados), em que fração dos ciclos estamos realizando stall devido aos hazards de dados?
- 4.12.3 [10] <§4.7>** Vamos supor que não tenhamos recursos para ter Muxes de três entradas que são necessários para o forwarding completo. Temos de decidir se é melhor encaminhar apenas do registrador de pipeline EX/MEM (forwarding do próximo ciclo) ou apenas do registrador de pipeline MEM/WB (forwarding de dois ciclos). Qual das duas opções resulta em menos ciclos de stall de dados?
- 4.12.4 [10] <§4.7>** Para as possibilidades de hazard e latências de estágio de pipeline indicadas, qual é o ganho de velocidade obtido acrescentando-se forwarding completo a um pipeline que não tinha forwarding?
- 4.12.5 [10] <§4.7>** Qual seria o ganho de velocidade adicional (relativo a um processador com forwarding) se acrescentássemos o forwarding de retorno no tempo que elimina todos os hazards de dados? Suponha que o circuito de retorno no tempo ainda a ser inventado acrescente 100 ps à latência do estágio EX de forwarding completo.
- 4.12.6 [20] <§4.7>** Repita o Exercício 4.12.3, mas desta vez determine quais das duas opções resulta em menor tempo por instrução.
- 4.13** Este exercício tem por finalidade ajudá-lo a entender o relacionamento

entre forwarding, detecção de hazard e projeto de ISA. Os problemas neste exercício referem-se a estas sequências de instrução, e considere que ele é executado em um caminho de dados com pipeline em cinco estágios.

```
add r5,r2,r1
lw  r3,4(r5)
lw  r2,0(r2)
or  r3,r5,r3
sw  r3,0(r5)
```

**4.13.1** [5] <§4.7> Se não houver forwarding ou detecção de hazard, insira nops para garantir a execução correta.

**4.13.2** [10] <§4.7> Repita o Exercício 4.13.1, mas agora use nops somente quando um hazard não puder ser evitado alterando ou rearrumando essas instruções. Você pode considerar que o registrador R7 pode ser usado para manter valores temporários no seu código modificado.

**4.13.3** [10] <§4.7> Se o processador tem forwarding, mas nos esquecemos de implementar a unidade de detecção de hazard, o que acontece quando esse código é executado?

**4.13.4** [20] <§4.7> Se houver forwarding, para os cinco primeiros ciclos durante a execução desse código, especifique quais sinais são ativados em cada ciclo pelas unidades de detecção de hazard e forwarding na [Figura 4.60](#).

**4.13.5** [10] <§4.7> Se não houver forwarding, que novas entradas e sinais de saída precisamos para a unidade de detecção de hazard da [Figura 4.60](#)? Usando essa sequência de instruções como exemplo, explique por que cada sinal é necessário.

**4.13.6** [20] <§4.7> Para a unidade de detecção de hazard do Exercício 4.13.5, especifique quais sinais de saída ela ativa em cada um dos cinco primeiros ciclos durante a execução desse código.

**4.14** Este exercício tem por finalidade ajudá-lo a entender o relacionamento

entre slots de delay, hazards de controle e execução de desvio em um processador com pipeline. Neste exercício, consideramos que o código MIPS a seguir é executado em um processador com um pipeline em cinco estágios, forwarding completo e um previsor de desvio tomado:

```
lw r2,0(r1)
label1: beq r2,r0,label2 # não tomado uma vez, depois tomado
        lw r3,0(r2)
        beq r3,r0,label1 # tomado
        add r1,r3,r1
label2: sw r1,0(r2)
```

**4.14.1** [10] <§4.8> Desenhe um diagrama de execução de pipeline para este código, considerando que não existam slots de delay e que os desvios sejam executados no estágio EX.

**4.14.2** [10] <§4.8> Repita o Exercício 4.14.1, mas considere que os slots de delay sejam utilizados. No código apresentado, a instrução que vem após o desvio agora é a instrução do slot de delay para esse desvio.

**4.14.3** [20] <§4.8> Uma maneira de mover a resolução do desvio para um estágio anterior é não precisar de uma operação da ALU nos desvios condicionais. As instruções de desvio seriam “bez rd,label” e “bnez rd,label”, e haveria desvio se o registrador tivesse e não tivesse um valor 0, respectivamente. Mude esse código para usar essa instrução de desvio em vez de beq. Você pode considerar que o registrador R8 está disponível como um registrador temporário, e que uma instrução tipo R seq (set if equal) pode ser usada.

A [Seção 4.8](#) descreve como a rigidez dos hazards de controle pode ser reduzida movendo-se a execução do desvio para o estágio ID. Essa técnica envolve um comparador dedicado no estágio ID, como mostra a [Figura 4.62](#). Porém, essa técnica tem o potencial de aumentar a latência do estágio ID, além de requerer lógica adicional de forwarding e detecção de hazard.

**4.14.4** [10] <§4.8> Usando como exemplo a primeira instrução de desvio no código apresentado, descreva a lógica de detecção de hazard necessária para dar suporte à execução do desvio no estágio ID como na [Figura 4.62](#). Que tipo de hazard essa nova lógica deveria detectar?

**4.14.5** [10] <§4.8> Para o código apresentado, qual é o ganho de velocidade

alcançado movendo-se a execução do desvio para o estágio ID? Explique sua resposta. No seu cálculo de ganho de velocidade, considere que a comparação adicional no estágio ID não afeta o tempo do ciclo de clock.

**4.14.6 [10] <§4.8>** Usando como exemplo a primeira instrução de desvio no código apresentado, descreva o suporte para forwarding que precisa ser acrescentado para dar suporte à execução do desvio no estágio ID. Compare a complexidade dessa nova unidade de forwarding com a complexidade da unidade de forwarding existente na [Figura 4.62](#).

**4.15 A** importância de ter um bom previsor de desvio depende da frequência com que os desvios condicionais são executados. Juntamente com a precisão do previsor de desvio, isso determinará quanto tempo será gasto com stall devido a desvios mal previstos. Neste exercício, considere o desmembramento das instruções dinâmicas em diversas categorias de instrução, como a seguir:

Tipo R	BEQ	JMP	LW	SW
40%	25%	5%	25%	5%

Além disso, considere as seguintes precisões do previsor de desvio:

Sempre tomado	Sempre não tomado	2 bits
45%	55%	85%

**4.15.1 [10] <§4.8>** Os ciclos de stall ocasionados por desvios mal previstos aumentam o CPI. Qual é o CPI extra devido a desvios mal previstos com o previsor sempre tomado? Considere que os resultados do desvio sejam determinados no estágio EX, que não existem hazards de dados e que nenhum slot de delay seja utilizado.

**4.15.2 [10] <§4.8>** Repita o Exercício 4.15.1 para o previsor “sempre não tomado”.

**4.15.3 [10] <§4.8>** Repita o Exercício 4.15.1 para o previsor de 2 bits.

**4.15.4 [10] <§4.8>** Com um previsor de 2 bits, que ganho de velocidade seria alcançado se pudéssemos converter metade das instruções de desvio de um modo que substitua uma instrução de desvio por uma instrução da ALU? Suponha que instruções previstas correta e incorretamente tenham a mesma chance de serem substituídas.

**4.15.5 [10] <§4.8>** Com um previsor de 2 bits, que ganho de velocidade seria obtido se pudéssemos converter metade das instruções de desvio de um

modo que substituísse cada instrução de desvio por duas instruções da ALU? Suponha que instruções previstas correta e incorretamente tenham a mesma chance de serem substituídas.

**4.15.6** [10] <§4.8> Algumas instruções de desvio são muito mais previsíveis do que outras. Se soubermos que 80% de todas as instruções de desvio executadas são desvios loop-back fáceis de prever, que sempre são previstos corretamente, qual é a precisão do previsor de 2 bits nos 20% restantes das instruções de desvio?

**4.16** Este exercício examina a precisão de vários previsores de desvios para o seguinte padrão repetitivo (como em um loop) de resultados do desvio: T, NT, T, T, NT

**4.16.1** [5] <§4.8> Qual é a precisão dos previsores sempre tomado e sempre não tomado para essa sequência dos resultados do desvio?

**4.16.2** [5] <§4.8> Qual é a precisão do previsor de dois bits para os quatro primeiros desvios nesse padrão, supondo que o previsor comece no estado inferior esquerdo da [Figura 4.63](#) (previsão não tomada)?

**4.16.3** [10] <§4.8> Qual é a precisão do previsor de dois bits se esse padrão for repetido indefinidamente?

**4.16.4** [30] <§4.8> Crie um previsor que alcance uma precisão perfeita se esse padrão for repetido indefinidamente. Seu previsor deverá ser um circuito sequencial com uma saída que oferece uma previsão (1 para tomado, 0 para não tomado) e nenhuma entrada que não seja o clock e o sinal de controle que indica que a instrução é um desvio condicional.

**4.16.5** [10] <§4.8> Qual é a precisão do seu previsor do Exercício 4.16.4 se ele receber um padrão repetitivo que é o oposto exato deste?

**4.16.6** [20] <§4.8> Repita o Exercício 4.16.4, mas agora o seu previsor deverá ser capaz de, mais cedo ou mais tarde (após um período de aquecimento durante o qual poderá fazer previsões erradas), começar a prever perfeitamente esse padrão e seu oposto. Seu previsor deverá ter uma entrada que lhe diga qual foi o resultado real. Dica: essa entrada permite que seu previsor determine qual dos dois padrões repetitivos ele recebe.

**4.17** Este exercício explora como o tratamento de exceção afeta o projeto do pipeline. Os três primeiros problemas neste exercício referem-se às duas instruções a seguir:

Instrução 1	Instrução 2
BNE R1, R2, Label	LW R1, 0(R1)

**4.17.1** [5] <§4.9> Quais exceções cada uma dessas instruções pode disparar?

Para cada uma dessas exceções, especifique o estágio do pipeline em que ela é detectada.

**4.17.2** [10] <§4.9> Se houver um endereço de handler separado para cada exceção, mostre como a organização do pipeline deve ser mudada para ser capaz de tratar dessa exceção. Você pode considerar que os endereços desses handlers são conhecidos quando o processador é projetado.

**4.17.3** [10] <§4.9> Se a segunda instrução dessa tabela for apanhada logo após a instrução da primeira tabela, descreva o que acontece no pipeline quando a primeira instrução causa a primeira exceção que você listou no Exercício 4.17.1. Mostre o diagrama de execução do pipeline do momento em que a primeira instrução é apanhada até o momento em que a primeira instrução do handler de exceção é concluída.

**4.17.4** [20] <§4.9> No tratamento de exceção com vetor, a tabela de endereços do handler de exceção está na memória de dados em um endereço conhecido (fixo). Mude o pipeline para implementar esse mecanismo de tratamento de exceção. Repita o Exercício 4.17.3 usando esse pipeline modificado e o tratamento de exceção com vetor.

**4.17.5** [15] <§4.9> Queremos simular o tratamento de exceção com vetor (descrito no Exercício 4.17.4) em uma máquina que tem apenas um endereço de handler fixo. Escreva o código que deverá estar nesse endereço fixo. Dica: esse código deverá identificar a exceção, obter o endereço correto da tabela de vetor de exceção e transferir a execução para esse handler.

**4.18** Neste exercício, comparamos o desempenho dos processadores de um despacho e processadores de dois despachos, levando em conta as transformações do programa que podem ser feitas para otimizar a execução em dois despachos. Os problemas neste exercício referem-se ao seguinte loop (escrito em C):

```
for( i=0 ; i != j ; i+=2 )
    b[i]=a[i]-a[i+1];
```

Ao escrever código MIPS, considere que as variáveis são mantidas em registradores da seguinte forma, e que todos os registradores, com exceção

daqueles indicados como Livre, são usados para manter diversas variáveis, de modo que não podem mais ser usados.

i	j	a	b	c	Livre
R5	R6	R1	R2	R3	R10, R11, R12

**4.18.1** [10] <§4.10> Traduza esse código C para instruções MIPS. Sua tradução deverá ser direta, sem rearrumar as instruções para conseguir melhor desempenho.

**4.18.2** [10] <§4.10> Se o loop sair depois de executar apenas duas iterações, desenhe um diagrama de pipeline para o seu código MIPS do Exercício 4.18.1 executado em um processador com dois despachos mostrado na [Figura 4.69](#). Suponha que o processador tenha previsão de desvio perfeita e possa buscar quaisquer duas instruções (não apenas instruções consecutivas) no mesmo ciclo.

**4.18.3** [10] <§4.10> Rearrume o seu código do Exercício 4.18.1 para alcançar o melhor desempenho em um processador de dois despachos estático, da [Figura 4.69](#).

**4.18.4** [10] <§4.10> Repita o Exercício 4.18.2, mas desta vez use seu código MIPS do Exercício 4.18.3.

**4.18.5** [10] <§4.10> Qual é o ganho de velocidade ao passar de um processador de um despacho para o de dois despachos da [Figura 4.69](#)? Use o seu código do Exercício 4.18.1 para um despacho e dois despachos, e considere que 1.000.000 iterações do loop são executadas. Assim como no Exercício 4.18.2, considere que o processador tenha previsões de desvio perfeitas, e que um processador de dois despachos possa buscar duas instruções quaisquer no mesmo ciclo.

**4.18.6** [10] <§4.10> Repita o Exercício 4.18.5, mas desta vez considere que, no processador de dois despachos, uma das instruções a serem executadas em um ciclo possa ser de qualquer tipo e a outra uma instrução que não seja de memória.

**4.19** Este exercício explora a eficiência de energia e seu relacionamento com o desempenho. Os problemas neste exercício consideram o consumo de energia a seguir para a atividade na Memória de Instrução, Registradores e Memória de Dados. Você pode considerar que os outros componentes do caminho de dados gastam uma quantidade de energia insignificante.

I-Mem	1 Leitura de Registrador	Escrita de Registrador	Leitura de Mem D	Escrita de Mem D
-------	--------------------------	------------------------	------------------	------------------

140 pJ	70 pJ	60 pJ	140 pJ	120 pJ
--------	-------	-------	--------	--------

Suponha que os componentes no caminho de dados tenham as latências a seguir. Você pode considerar que os outros componentes do caminho de dados têm latência insignificante.

I-Mem	Controle	Registrador de leitura ou escrita	ALU	Leitura ou escrita de Mem D
200 ps	150 ps	90 ps	90 ps	250 ps

**4.19.1** [10] <§§4.3, 4.6, 4.14> Quanta energia é gasta para executar uma instrução ADD em um projeto de ciclo único e no projeto em pipeline com cinco estágios?

**4.19.2** [10] <§§4.6, 4.14> Qual é a instrução MIPS no pior caso em termos do consumo de energia, e qual é a energia gasta para executá-la?

**4.19.3** [10] <§§4.6, 4.14> Se a redução de energia é fundamental, como você mudaria o projeto em pipeline? Qual é a redução percentual na energia gasta por uma instrução LW após essa mudança?

**4.19.4** [10] <§§4.6, 4.14> Qual é o impacto das suas mudanças do Exercício 4.19.3 sobre o desempenho?

**4.19.5** [10] <§§4.6, 4.14> Podemos eliminar o sinal de controle MemRead e fazer com que a memória de dados seja lida em cada ciclo, ou seja, podemos ter MemRead = 1 permanentemente. Explique por que o processador ainda funciona corretamente após essa mudança. Qual é o efeito dessa mudança sobre a frequência de clock e consumo de energia?

**4.19.6** [10] <§§4.6, 4.14> Se uma unidade ociosa gasta 10% da potência que gastaria se estivesse ativa, qual é a energia gasta pela memória de instrução em cada ciclo? Que porcentagem da energia geral gasta pela memória de instrução essa energia ociosa representa?

## Respostas das Seções “Verifique você mesmo”

§4.1, página 220: 3 de 5: Controle, Caminho de dados, Memória, Entrada e Saída estão faltando.

§4.2, página 222: falso. Elementos de estado disparados na borda tornam a leitura e escrita simultâneas tanto possíveis quanto não ambíguas.

§4.3, página 229: I. a. II. c.

§4.4, página 240: Sim, Desvio e ALUOp0 são idênticos. Além disso, MemtoReg e RegDst são opostos um do outro. Você não precisa de um inverter; basta usar o outro sinal e inverter a ordem das entradas para o

multiplexador!

§4.5, página 251: 1. Stall no resultado  $1w$ . 2. Bypassing do primeiro resultado de add escrito em  $$t1$ . 3. Nenhum stall ou bypassing é necessário.

§4.6, página 261: Afirmações 2 e 4 estão corretas; o restante está incorreto.

§4.8, página 285: 1. Previsão não tomada. 2. Previsão tomada. 3. Previsão dinâmica.

§4.9, página 290: A primeira instrução, pois ela é executada logicamente antes das outras.

§4.10, página 301: 1. Ambos. 2. Ambos. 3. Software. 4. Hardware. 5. Hardware. 6. Hardware. 7. Ambos. 8. Hardware. 9. Ambos.

§4.12, página 308: Duas primeiras são falsas e duas últimas são verdadeiras.