

# 3

*A precisão numérica é a própria alma da ciência.*

**Sir D'arcy Wentworth Thompson**  
*On Growth and Form, 1917*

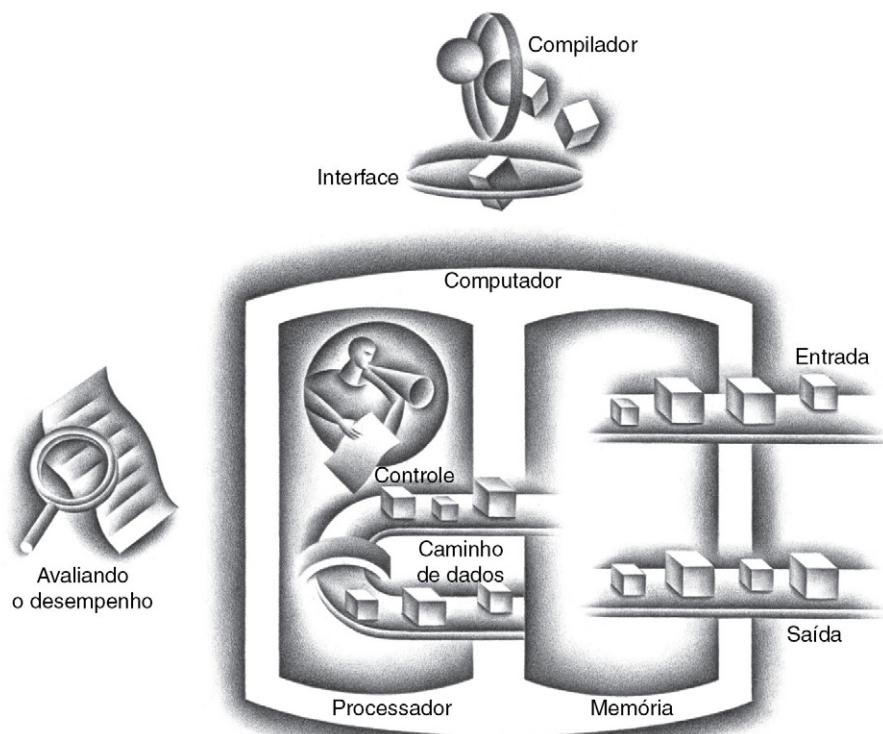
## Aritmética Computacional

- 3.1** **Introdução** 182
- 3.2** **Adição e subtração** 182
- 3.3** **Multiplicação** 186
- 3.4** **Divisão** 191
- 3.5** **Ponto flutuante** 197
- 3.6** **Paralelismo e aritmética computacional:  
Associatividade** 219
- 3.7** **Vida real: ponto flutuante no x86** 220
- 3.8** **Falácia e armadilhas** 222

- 3.9 Comentários finais 226**
- 3.10 Perspectiva histórica e leitura adicional 228**
- 3.11 Exercícios 228**

---

## Os cinco componentes clássicos de um computador



## 3.1 Introdução

As palavras do computador são compostas de bits; assim, podem ser representadas como números binários. O Capítulo 2 mostra que os inteiros podem ser representados em formato decimal ou binário, mas e quanto aos outros números que ocorrem normalmente? Por exemplo:

- Como são representadas frações e outros números reais?
- O que acontece se uma operação cria um número maior do que poderia ser representado?
- E por trás de todas essas perguntas existe um mistério: como o hardware realmente multiplica ou divide números?

O objetivo deste capítulo é desvendar esse mistério, incluindo a representação dos números, algoritmos aritméticos, hardware que acompanha esses algoritmos e as implicações de tudo isso para os conjuntos de instruções. Essas ideias podem ainda explicar truques que você já pode ter encontrado nos computadores

*Subtração: o companheiro esquisito da adição*

No. 10, Top Ten Courses for Athletes at a Football Factory, David Letterman e outros, *Book of Top Ten Lists*, 1990

## 3.2 Adição e subtração

A adição é exatamente o que você esperaria nos computadores. Dígitos são somados bit a bit, da direita para a esquerda, com carries (“vai-uns”), sendo passados para o próximo dígito à esquerda, como você faria manualmente. A subtração utiliza a adição: o operando apropriado é simplesmente negado antes de ser somado.

### EXEMPLO

#### Adição e subtração binária

Vamos tentar somar  $6_{dec}$  a  $7_{dec}$  em binário e depois subtrair  $6_{dec}$  de  $7_{dec}$  em binário.

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{bin} = 7_{dec} \\
 +\quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{bin} = 6_{dec} \\
 =\quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{bin} = 13_{dec}
 \end{array}$$

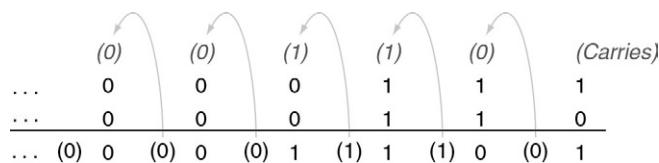
Os 4 bits à direita fazem toda a ação; a [Figura 3.1](#) mostra as somas e os carries. Os carries aparecem entre parênteses, com as setas mostrando como são passados. A subtração de  $6_{dec}$  de  $7_{dec}$  pode ser feita diretamente:

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{bin} = 7_{dec} \\
 -\quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{bin} = 6_{dec} \\
 \hline
 =\quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{bin} = 1_{dec}
 \end{array}$$

ou por meio da soma, usando a representação de complemento de dois de  $-6$ :

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{bin} = 7_{dec} \\
 +\quad 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{bin} = -6_{dec} \\
 \hline
 =\quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{bin} = 1_{dec}
 \end{array}$$

### RESPOSTA



**FIGURA 3.1 Adição binária, mostrando carries da direita para a esquerda.** O bit mais à direita adiciona 1 a 0, resultando em uma soma de 1 e um *carry out* de 0 para esse bit. Logo, a operação para o segundo dígito da direita é  $0 + 1 + 1$ . Isso gera uma soma de 0 e um *carry out* de 1 para esse bit. O terceiro dígito é a soma de  $1 + 1 + 1$ , resultando em um *carry out* de 1 e uma soma de 1 para esse dígito. O quarto bit é  $1 + 0 + 0$ , tendo uma soma de 1 e nenhum *carry*.

Já dissemos que o overflow ocorre quando o resultado de uma operação não pode ser representado com o hardware disponível, nesse caso, uma palavra de 32 bits. Quando pode ocorrer um overflow na adição? Quando se somam operandos com sinais diferentes, não poderá haver overflow. O motivo é que a soma não pode ser maior do que um dos operandos. Por exemplo,  $-10 + 4 = -6$ . Como os operandos cabem nos 32 bits, e a soma não é maior do que um operando, a soma também precisa caber nos 32 bits. Portanto, nenhum overflow pode ocorrer ao somar operandos positivos e negativos.

Existem restrições semelhantes à ocorrência do overflow durante a subtração, mas esse é apenas o princípio oposto: quando os sinais dos operandos são *iguais*, o overflow pode ocorrer. Para ver isso, lembre-se de que  $x - y = x + (-y)$ , pois subtraímos negando o segundo operando e depois somamos. Assim, quando subtraímos operandos do mesmo sinal, acabamos *somando* operandos de sinais *diferentes*. Pelo parágrafo anterior, sabemos que não pode ocorrer overflow também nesse caso.

Tendo examinado quando um overflow pode ocorrer na adição e na subtração, ainda não respondemos como detectar quando ele *ocorre*. Logicamente, a soma ou a subtração de dois números de 32 bits pode gerar um resultado que precisa de 33 bits para ser totalmente expresso. A falta de um 33º bit significa que, quando o overflow ocorre, o bit de sinal está sendo definido com o *valor* do resultado, no lugar do sinal apropriado do resultado. Como precisamos apenas de um bit extra, somente o bit de sinal pode estar errado. Logo, o overflow ocorre quando se somam dois números positivos, e a soma é negativa, ou vice-versa. Isso significa que um carry ocorreu no bit de sinal.

O overflow ocorre na subtração quando subtraímos um número negativo de um número positivo e obtemos um resultado negativo, ou quando subtraímos um número positivo de um número negativo e obtemos um resultado positivo. Isso significa que houve um empréstimo do bit de sinal. A [Figura 3.2](#) mostra a combinação de operações, operandos e resultados que indicam um overflow.

Acabamos de ver como detectar o overflow para os números em complemento de dois em um computador. E com relação aos inteiros sem sinal? Os inteiros sem sinal normalmente são usados para endereços de memória em que os overflows são ignorados.

Logo, o projetista de computador precisa oferecer uma maneira de ignorar o overflow em alguns casos e reconhecê-lo em outros. A solução do MIPS é ter dois tipos de instruções aritméticas para reconhecer as duas escolhas:

- Adição (*add*), adição imediata (*addi*) e subtração (*sub*) causam exceções no overflow.
- Adição sem sinal (*addu*), adição imediata sem sinal (*addiu*) e subtração sem sinal (*subu*) *não* causam exceções no overflow.

Operação	Operando A	Operando B	Resultado indicando overflow
$A + B$	$\geq 0$	$\geq 0$	$< 0$
$A + B$	$< 0$	$< 0$	$\geq 0$
$A - B$	$\geq 0$	$< 0$	$< 0$
$A - B$	$< 0$	$\geq 0$	$\geq 0$

**FIGURA 3.2 Condições de overflow para adição e subtração.**

**Unidade Lógica e Aritmética (ALU)** Hardware que realiza adição, subtração e normalmente operações lógicas como AND e OR.

Como a linguagem C ignora os overflows, os compiladores C do MIPS sempre gerarão as versões sem sinal das instruções aritméticas *addu*, *addiu* e *subu*, sem importar o tipo das variáveis. No entanto, os compiladores Fortran do MIPS apanham as instruções aritméticas apropriadas, dependendo do tipo dos operandos.

○ O Apêndice C descreve o hardware que realiza a adição e subtração, que é chamado de **Unidade Lógica e Aritmética**, ou ULA.

## Interface hardware/software

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

**interrupção** Uma exceção que vem de fora do processador. (Algumas arquiteturas utilizam o termo *interrupção* para todas as exceções.)

O projetista de computador precisa decidir como tratar overflows aritméticos. Embora algumas linguagens como C e Java ignorem o overflow de inteiros, linguagens como Ada e Fortran exigem que o programa seja notificado. O programador ou o ambiente de programação precisa, então, decidir o que fazer quando ocorre o overflow.

O MIPS detecta o overflow com uma **exceção**, também chamada de **interrupção** em muitos computadores. Uma exceção ou interrupção é basicamente uma chamada de procedimento não planejada. O endereço da instrução que gerou o overflow é salvo em um registrador e o computador desvia para um endereço predefinido, a fim de invocar a rotina apropriada para essa exceção. O endereço interrompido é salvo de modo que, em algumas situações, o programa possa continuar após o código corretivo ser executado. (A Seção 4.9 abrange as exceções com mais detalhes; os Capítulos 5 e 6 descrevem outras situações em que ocorrem exceções e interrupções.)

O MIPS inclui um registrador, chamado *contador de programa de exceção* (EPC – Exception Program Counter), para conter o endereço da instrução que causou a exceção. A instrução *move from system control* (*mfc0*) é usada a fim de copiar o EPC para um registrador de uso geral, de modo que o software do MIPS tem a opção de retornar à instrução problemática por meio de uma instrução *jump register*.

## Aritmética para Multimídia

Já que cada microprocessador desktop por definição tem suas próprias telas gráficas, e como as quantidades de transistores aumentaram, foi inevitável que seria acrescentado suporte para operações gráficas.

Muitos sistemas gráficos originalmente usavam 8 bits para representar cada uma das três cores primárias, mais 8 bits para um local de um pixel. O acréscimo de alto-falantes e microfones para teleconferência e jogos de vídeo sugeriu também o suporte para som. As amostras de áudio precisam de mais de 8 bits de precisão, mas 16 bits são suficientes.

Cada microprocessador tem suporte especial, de modo que os bytes e meias palavras ocupam menos espaço quando armazenados na memória (veja Seção 2.9), mas, em razão da infreqüência das operações aritméticas nesses tamanhos de dados nos programas de inteiros típicos existe pouco suporte além das transferências de dados. Os arquitetos reconhecem que muitas aplicações gráficas e de áudio realizariam a mesma operação sobre vetores desses dados. Particionando as cadeias de carry dentro de um somador de 64 bits, um processador poderia realizar operações simultâneas sobre vetores curtos de operandos de 8 bits, quatro operandos de 16 bits ou dois operandos de 32 bits. O custo desses somadores partionados era pequeno. Essas extensões têm sido chamadas de vetor ou SIMD, para única instrução, múltiplos dados (veja Seção 2.17 e Capítulo 7).

Um recurso geralmente não encontrado nos microprocessadores de uso geral é a *saturação* de operações. A saturação significa que, quando um cálculo estoura, o resultado é definido como o maior número positivo ou o maior número negativo, em vez de um cálculo de módulo, como na aritmética do complemento de dois. A saturação provavelmente é o que você deseja para operações de mídia. Por exemplo, o botão de volume em um aparelho de rádio seria frustrante se, enquanto você girasse, o volume aumentasse continuamente

por um tempo e depois imediatamente ficasse baixo. Um botão com saturação pararia no volume mais alto, não importa o quanto você o girasse. A Figura 3.3 mostra as operações aritméticas e lógicas encontradas em muitas extensões de multimídia dos conjuntos de instruções modernos.

Categoría de instrucción	Operandos
Soma/subtração sem sinal	Oito de 8 bits ou quatro de 16 bits
Soma/subtração saturada	Oito de 8 bits ou quatro de 16 bits
Max/min/mínimo	Oito de 8 bits ou quatro de 16 bits
Média	Oito de 8 bits ou quatro de 16 bits
Shift à direita/esquerda	Oito de 8 bits ou quatro de 16 bits

**FIGURA 3.3 Resumo do suporte de multimídia para computadores desktop.**

**Detalhamento:** o MIPS pode interceptar um overflow, mas, diferente de muitos outros computadores, não existe desvio condicional para testar o overflow. A sequência de instruções do MIPS pode descobrir overflow. Para a adição com sinal, a sequência é a seguinte (veja o *Detalhamento* na Seção 2.6 do Capítulo 2, para obter uma descrição da instrução xor):

```

addu $t0, $t1, $t2 # $t0 = soma, mas não intercepta
xor $t3, $t1, $t2 # Verifica se sinais são diferentes
slt $t3, $t3, $zero # $t3 = 1 se os sinais são diferentes
bne $t3, $zero, Sem_overflow # sinais de $t1, $t2 s,
                            # portanto, sem overflow
xor $t3, $t0, $t1 # sinais =; sinal da soma também combina?
                    # $t3 negativo se soma diferente
slt $t3, $t3, $zero # $t3 = 1 se sinal da soma diferente
bne $t3, $zero, Overflow# Todos os três sinais ?; vai para overflow
    
```

Para adição sem sinal ( $$t0 = $t1 + $t2$ ), o teste é

```

addu $t0, $t1, $t2      # $t0 = soma
nor $t3, $t1, $zero     # $t3 = NOT $t1
                        # (compl.adois - 1:232 - $t1 - 1)
slt $t3, $t3, $t2       # (232 - $t1 - 1) < $t2
                        # => 232 - 1 < $t1 + $t2
bne $t3,$zero,Overflow # se(232-1<$t1+$t2) vai para overflow
    
```

## Resumo

A questão principal desta seção é que, independente da representação, o tamanho finito da palavra dos computadores significa que as operações aritméticas podem criar resultados muito grandes para caber nesse tamanho de palavra fixo. É fácil detectar o overflow em números sem sinal, embora quase sempre sejam ignorados, pois os programas não querem detectar overflow para a aritmética de endereço, o uso mais comum dos números naturais. O complemento de dois apresenta um desafio maior, embora alguns sistemas de software exijam detecção de overflow, de modo que, hoje, todos os computadores tenham um meio de detectá-lo.

A crescente popularidade das aplicações de multimídia levou a instruções aritméticas que dão suporte a operações mais estreitas, que podem operar facilmente em paralelo.

## Verifique você mesmo

Algumas linguagens de programação permitem a aritmética de inteiros em complemento de dois com variáveis declaradas com um byte e meio. Que instruções do MIPS seriam usadas?

1. Load com  $lbu$ ,  $lhu$ ; aritmética com  $add$ ,  $sub$ ,  $mult$ ,  $div$ ; depois, armazenamento usando  $sb$ ,  $sh$ .
2. Load com  $lb$ ,  $lh$ ; aritmética com  $add$ ,  $sub$ ,  $mult$ ,  $div$ ; depois, armazenamento usando  $sb$ ,  $sh$ .
3. Load com  $lb$ ,  $lh$ ; aritmética com  $add$ ,  $sub$ ,  $mult$ ,  $div$ ; usando AND para mascarar o resultado com 8 ou 16 bits após cada operação; depois, armazenamento usando  $sb$ ,  $sh$ .

**Detalhamento:** no texto anterior, dissemos que você copia o EPC para um registrador por meio de `mfc0` e depois retorna ao código interrompido por meio de jump register. Isso leva a uma pergunta interessante: já que você primeiro precisa transferir o EPC para um registrador a fim de usar com jump register, como jump register pode retornar ao código interrompido e restaurar os valores originais de todos os registradores? Você restaura os registradores antigos primeiro, destruindo, assim, seu endereço de retorno do EPC, que colocou em um registrador para uso em jump register, ou restaura todos os registradores, menos aquele com o endereço de retorno, para que possa desviar – significando que uma exceção resultaria em alterar esse único registrador a qualquer momento durante a execução do programa! Nenhuma dessas opções é satisfatória.

Para auxiliar o hardware nesse dilema, os programadores MIPS concordaram em reservar os registradores `$k0` e `$k1` para o sistema operacional; esses registradores não são restaurados nas exceções. Assim como os compiladores MIPS evitam o uso do registrador `$at`, de modo que o montador possa utilizá-lo como um registrador temporário (veja a Seção “Interface Hardware/Software”, na Seção 2.10), os compiladores também se abstêm do uso dos registradores `$k0` e `$k1`, de modo que fiquem disponíveis para o sistema operacional. As rotinas de exceção colocam o endereço de retorno em um desses registradores e depois usam o jump register para armazenar o endereço da instrução.

**Detalhamento:** A velocidade da adição é aumentada determinando-se o carry in para os bits de alta ordem mais cedo. Existem diversos esquemas para antecipar o carry, de modo que o cenário do pior caso é uma função do log2 do número de bits no somador. Esses sinais antecipados são mais rápidos, pois percorrem menos portas na sequência, mas exigem mais portas para antecipar o carry apropriado. O mais comum é o *carry lookahead*, descrito na Seção C.6 do  Apêndice C no site.

*Multiplicação é vexação,  
Divisão também é ruim; A  
regra de três me intriga, e a  
prática me deixa louco.*

Anônimo, manuscrito de Elizabeth, 1570

### 3.3

## Multiplicação

Agora que completamos a explicação de adição e subtração, estamos prontos para montar a operação mais vexatória da multiplicação.

Primeiro, vamos rever a multiplicação de números decimais à mão para nos lembrar das etapas e dos nomes dos operandos. Por motivos que logo se tornarão claros, limitamos esse exemplo decimal ao uso apenas dos dígitos 0 e 1. Multiplicando  $1000_{\text{dec}}$  por  $1001_{\text{dec}}$ :

$$\begin{array}{r}
 \text{Multiplicando} & 1000_{\text{dec}} \\
 \text{Multiplicador} \times & \underline{1001}_{\text{dec}} \\
 & 1000 \\
 & 0000 \\
 & 0000 \\
 & 1000 \\
 \hline
 \text{Produto} & 1001000_{\text{dec}}
 \end{array}$$

O primeiro operando é chamado *multiplicando* e o segundo é o *multiplicador*. O resultado final é chamado *produto*. Como você pode se lembrar, o algoritmo aprendido na escola é pegar os dígitos do multiplicador um a um, da direita para a esquerda, calculando a multiplicação do multiplicando pelo único dígito do multiplicador e deslocando o produto intermediário um dígito para a esquerda dos produtos intermediários anteriores.

A primeira observação é que o número de dígitos no produto é muito maior do que o número no multiplicando ou no multiplicador. De fato, se ignorarmos os bits de sinal, o tamanho da multiplicação de um multiplicando de  $n$  bits por um multiplicador de  $m$  bits é um produto que possui  $n + m$  bits de largura. Ou seja,  $n + m$  bits são necessários para representar todos os produtos possíveis. Logo, como na adição, a multiplicação precisa lidar com o overflow, pois constantemente desejamos um produto de 32 bits como resultado da multiplicação de dois números de 32 bits.

Neste exemplo, restringimos os dígitos decimais a 0 e 1. Com somente duas opções, cada etapa da multiplicação é simples:

1. Basta colocar uma cópia do multiplicando ( $1 \times$  multiplicando) no lugar apropriado se o dígito do multiplicador for 1, ou
2. Colocar 0 ( $0 \times$  multiplicando) no lugar apropriado se o dígito for 0.

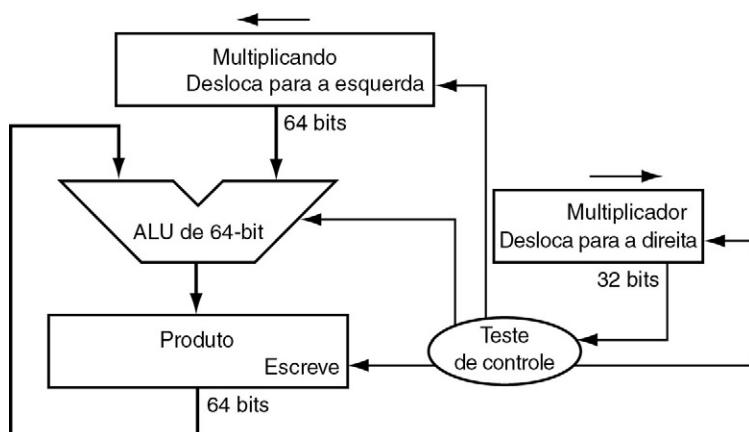
Embora o exemplo decimal anterior utilize apenas 0 e 1, a multiplicação de números binários sempre usa 0 e 1 e, por isso, sempre oferece apenas essas duas opções.

Agora que já revisamos os fundamentos tradicionais da multiplicação, a próxima etapa é mostrar o hardware de multiplicação altamente otimizado. Quebramos essa tradição na crença de que você entenderá melhor vendo a evolução do hardware e do algoritmo de multiplicação no decorrer das diversas gerações. Por enquanto, vamos supor que estamos multiplicando apenas números positivos.

## Versão sequencial do algoritmo e hardware de multiplicação

Esse projeto imita o algoritmo que aprendemos na escola; o hardware aparece na Figura 3.4. Desenhamos o hardware de modo que os dados fluam de cima para baixo, para que fique mais semelhante à técnica do lápis e papel.

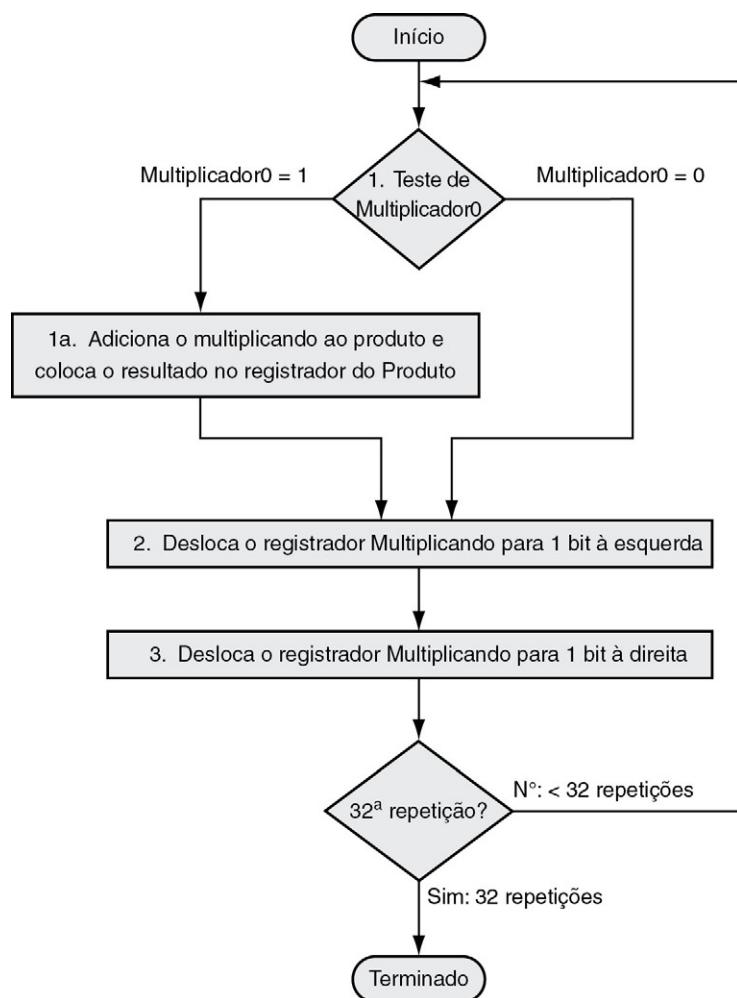
Vamos supor que o multiplicador esteja no registrador Multiplicador de 32 bits e que o registrador Produto de 64 bits esteja inicializado como 0. Pelo exemplo de lápis e papel, visto anteriormente, fica claro que precisaremos mover o multiplicando para a esquerda um



**FIGURA 3.4 Primeira versão do hardware de multiplicação.** O registrador do Multiplicando, a ALU, e o registrador do Produto possuem 64 bits de largura, apenas com o registrador do Multiplicador contendo 32 bits. (O Apêndice C descreve as ALUs.) O multiplicando com 32 bits começa na metade direita do registrador do Multiplicando e é deslocado à esquerda 1 bit em cada etapa. O multiplicador é deslocado na direção oposta em cada etapa. O algoritmo começa com o produto inicializado com 0. O controle decide quando deslocar os registradores Multiplicando e Multiplicador e quando escrever novos valores no registrador do Produto.

dígito a cada passo, pois pode ser somado aos produtos intermediários. Durante 32 etapas, um multiplicando de 32 bits moveria 32 bits para a esquerda. Logo, precisamos de um registrador Multiplicando de 64 bits, inicializado com o multiplicando de 32 bits na metade direita e 0 na metade esquerda. Esse registrador, em seguida, é deslocado 1 bit para a esquerda a cada etapa, de modo a alinhar o multiplicando com a soma sendo acumulada no registrador Produto de 64 bits.

A Figura 3.5 mostra as três etapas clássicas necessárias para cada bit. O bit menos significativo do multiplicador (Multiplicador0) determina se o multiplicando é somado ao registrador Produto. O deslocamento à esquerda na etapa 2 tem o efeito de mover os operandos intermediários para a esquerda, assim como na multiplicação manual. O deslocamento à direita na etapa 3 nos indica o próximo bit do multiplicador a ser examinado na iteração seguinte. Essas três etapas são repetidas 32 vezes, para obter o produto. Se cada etapa usasse um ciclo de clock, esse algoritmo exigiria quase 100 ciclos de clock para multiplicar dois números de 32 bits. A importância relativa de operações aritméticas, como a multiplicação, varia com o programa, mas a soma e a subtração podem ser de 5 a 100 vezes mais comuns do que a multiplicação. Como consequência, em muitas aplicações, a multiplicação pode demorar vários ciclos de clock sem afetar o desempenho de forma significativa. Mesmo assim, a lei de Amdahl (veja Seção 1.8) nos lembra que até mesmo uma frequência moderada para uma operação lenta pode limitar o desempenho.

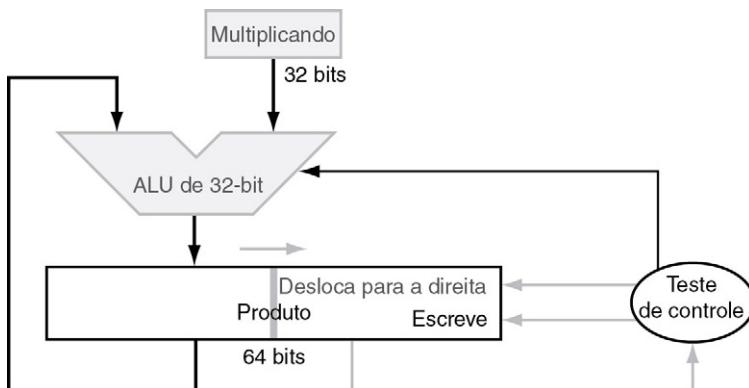


**FIGURA 3.5** O primeiro algoritmo de multiplicação, usando o hardware mostrado na [Figura 3.4](#).  
Se o bit menos significativo do multiplicador for 1, some o multiplicando ao produto. Caso contrário, vá para a etapa seguinte. Desloque o multiplicando para a esquerda e o multiplicador para a direita nas duas etapas seguintes. Essas três etapas são repetidas 32 vezes.

Esse algoritmo e o hardware são facilmente refinados para usar 1 ciclo de clock por etapa. O aumento de velocidade vem da realização das operações em paralelo: o multiplicador e o multiplicando são deslocados enquanto o multiplicando é somado ao produto se o bit do multiplicador for 1. O hardware simplesmente precisa garantir que testará o bit da direita do multiplicador e receberá a versão previamente deslocada do multiplicando. O hardware normalmente é otimizado ainda mais para dividir a largura do somador e dos registradores ao meio, observando onde existem partes não utilizadas dos registradores e somadores. A [Figura 3.6](#) mostra o hardware revisado.

Substituir a aritmética por deslocamentos também pode ocorrer quando se multiplica por constantes. Alguns compiladores substituem multiplicações por constantes curtas com uma série de deslocamentos e adições. Como deslocar um bit à esquerda representa um número duas vezes maior na base 2, o deslocamento de bits para a esquerda tem o mesmo efeito de multiplicar por uma potência de 2. Como dissemos no Capítulo 2, quase todo compilador realizará a otimização por redução de força substituindo uma multiplicação na potência de 2 por um deslocamento à esquerda.

## Interface hardware/software



**FIGURA 3.6 Versão refinada do hardware de multiplicação.** Compare com a primeira versão na [Figura 3.4](#). O registrador Multiplicando, a ALU, e o registrador Multiplicador possuem 32 bits de extensão, com somente o registrador Produto restando nos 64 bits. Agora, o produto é deslocado para a direita. O registrador Multiplicador separado também desapareceu. O multiplicador é colocado na metade direita do registrador Produto. Essas mudanças estão destacadas. (O registrador Produto na realidade deverá ter 65 bits, a fim de manter o carry do somador, mas ele aparece aqui como 64 bits para destacar a evolução da [Figura 3.4](#).)

### Um algoritmo de multiplicação

Usando números de 4 bits para economizar espaço, multiplique  $2_{\text{dec}} \times 3_{\text{dec}}$ , ou  $0010_{\text{bin}} \times 0011_{\text{bin}}$ .

### EXEMPLO

A [Figura 3.7](#) mostra o valor de cada registrador para cada uma das etapas rotuladas de acordo com a [Figura 3.5](#), com o valor final de  $0000\ 0110_{\text{bin}}$  ou  $6_{\text{dec}}$ . A cor é usada para indicar os valores de registrador que mudam nessa etapa e o bit circulado é aquele examinado para determinar a operação da próxima etapa.

### RESPOSTA

Iteração	Passo	Multiplicador	Multiplicando	Produto
0	Valores iniciais	001①	0000 0010	0000 0000
1	1a: $1 \Rightarrow$ Prod = Prod + Multiplicando	0011	0000 0010	0000 0010
	2: Desloca o Multiplicando à esquerda	0011	0000 0100	0000 0010
	3: Desloca o Multiplicador à direita	000①	0000 0100	0000 0010
2	1a: $1 \Rightarrow$ Prod = Prod + Multiplicando	0001	0000 0100	0000 0110
	2: Desloca o Multiplicando à esquerda	0001	0000 1000	0000 0110
	3: Desloca o Multiplicador à direita	000①	0000 1000	0000 0110
3	1: $0 \Rightarrow$ Nenhuma operação	0000	0000 1000	0000 0110
	2: Desloca o Multiplicando à esquerda	0000	0001 0000	0000 0110
	3: Desloca o Multiplicador à direita	000①	0001 0000	0000 0110
4	1: $0 \Rightarrow$ Nenhuma operação	0000	0001 0000	0000 0110
	2: Desloca o Multiplicando à esquerda	0000	0010 0000	0000 0110
	3: Desloca o Multiplicador à direita	0000	0010 0000	0000 0110

**FIGURA 3.7 Exemplo de multiplicação usando o algoritmo da Figura 3.5.** O bit examinado para determinar a próxima etapa está circulado.

### Multiplicação com sinal

Até aqui, tratamos de números positivos. O modo mais fácil de entender como tratar dos números com sinal é primeiro converter o multiplicador e o multiplicando para números positivos e depois lembrar dos sinais originais. Os algoritmos deverão, então, ser executados por 31 iterações, deixando os sinais fora do cálculo. Conforme aprendemos na escola, o produto só será negativo se os sinais originais forem diferentes.

Acontece que o último algoritmo funcionará para números com sinais se nos lembarmos de que os números com que estamos lidando possuem dígitos infinitos e que só os estamos representando com 32 bits. Logo, as etapas de deslocamento precisariam estender o sinal do produto para números com sinal. Quando o algoritmo terminar, a palavra menos significativa terá o produto de 32 bits.

### Multiplicação mais rápida

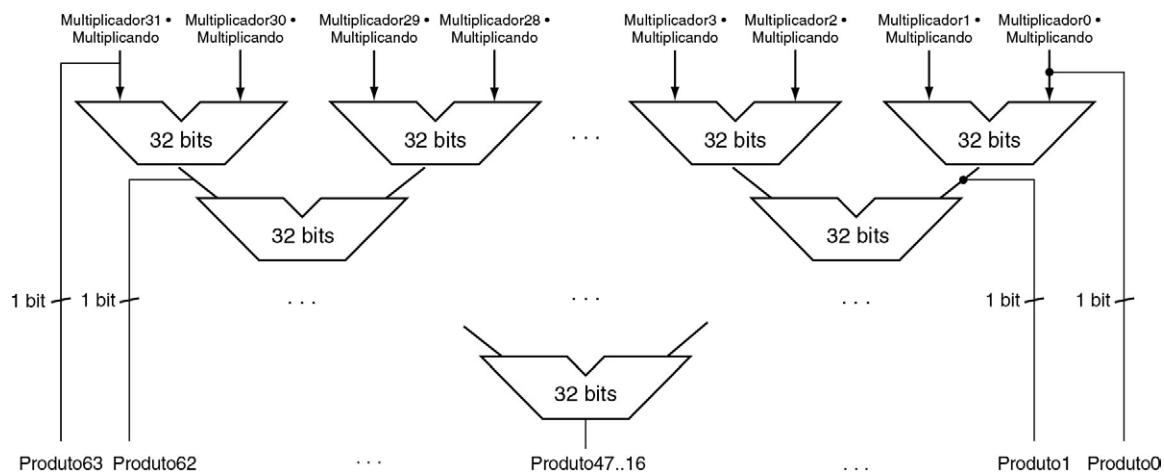
A Lei de Moore ofereceu tantos recursos que os projetistas de hardware agora podem construir um hardware de multiplicação muito mais rápido. Não importa se o multiplicando deve ser somado ou não, isso é conhecido no início da multiplicação analisando cada um dos 32 bits do multiplicador. Multiplicações mais rápidas são possíveis basicamente fornecendo um somador de 32 bits para cada bit do multiplicador: uma entrada é o AND do multiplicando pelo bit do multiplicador e a outra é a saída de um somador anterior.

Uma técnica simples seria conectar as saídas dos somadores à direita das entradas dos somados à esquerda, criando uma pilha de somadores com altura 32. Um modo alternativo de organizar essas 32 adições é em uma árvore paralela, como mostra a Figura 3.8. Em vez de esperar 32 tempos de add, esperamos apenas o  $\log_2(32)$  ou cinco tempos de add com 32 bits. A Figura 3.8 mostra como esse é o modo mais rápido de conectá-los.

De fato, a multiplicação pode se tornar ainda mais rápida do que cinco tempos de add, veja uso de *somadores para salvar carry* (veja Seção C.6 no Apêndice C) e porque é fácil usar um pipeline nesse projeto para que possam ser realizadas muitas multiplicações simultaneamente (veja Capítulo 4).

### Multiplicação no MIPS

O MIPS oferece um par separado de registradores de 32 bits, de modo a conter o produto de 64 bits, chamados *Hi* e *Lo*. Para produzir um produto com ou sem o devido sinal, o MIPS possui duas instruções: *multiply* (*mult*) e *multiply unsigned* (*multu*). Para apanhar o produto de 32 bits inteiro, o programador usa *move from lo* (*mflo*). O montador MIPS gera uma pseudoinstrução para multiplicar, que especifica três registradores de uso geral, gerando instruções *mflo* e *mfhi* que colocam o produto nos registradores.



**FIGURA 3.8 Hardware da multiplicação rápida.** Em vez de usar um único somador de 32 bits 31 vezes, esse hardware “desenrola o loop” para usar 31 somadores e depois os organiza para minimizar o atraso.

## Resumo

A multiplicação é feita pelo hardware simples de deslocamento e adição, derivado do método de lápis e papel que aprendemos na escola. Os compiladores utilizam até mesmo as instruções de deslocamento para multiplicações por potências de dois.

As duas instruções multiply do MIPS ignoram o overflow, de modo que fica a critério do software verificar se o produto é muito grande para caber nos 32 bits. Não existe overflow se Hi for 0 para *multu* ou o sinal replicado de Lo para *mult*. A instrução *move from hi* (*mfhi*) pode ser usada para transferir Hi a um registrador de uso geral, a fim de testar o overflow.

## Interface hardware/software

*Divide et impera.*

Tradução do latim para “Dividir e conquistar”, máxima política antiga, citada por Maquiavel, 1532

## 3.4 Divisão

A operação recíproca da multiplicação é a divisão, ainda menos frequente e ainda mais peculiar. Ela oferece até mesmo a oportunidade de realizar uma operação matematicamente inválida: dividir por 0.

Vamos começar com um exemplo de divisão longa usando números decimais, para lembrar os nomes dos operandos e do algoritmo de divisão que aprendemos na escola. Por motivos semelhantes aos da seção anterior, vamos limitar os dígitos decimais a apenas 0 ou 1. O exemplo é a divisão de  $1.001.010_{dec}$  por  $1000_{dec}$ :

$$\begin{array}{r}
 1001_{dec} \\
 \text{Divisor } 1000_{dec} \overline{)1001010_{dec}} & \text{Quociente} \\
 -1000 \\
 \hline
 10 \\
 101 \\
 1010 \\
 -1000 \\
 \hline
 10_{dec} & \text{Resto}
 \end{array}$$

**dividendo** Um número sendo dividido.

**divisor** Um número pelo qual o dividendo é dividido.

**quociente** O resultado principal de uma divisão; um número que, quando multiplicado pelo divisor e somado ao resto, produz o dividendo.

**resto** O resultado secundário de uma divisão; um número que, quando somado ao produto do quociente pelo divisor, produz o dividendo.

Os dois operandos (**dividendo** e **divisor**) e o resultado (**quociente**) da divisão são acompanhados por um segundo resultado, chamado **resto**. Veja aqui outra maneira de expressar o relacionamento entre os componentes:

$$\text{Dividendo} = \text{Quociente} \times \text{Divisor} + \text{Resto}$$

em que o resto é menor do que o divisor. Raramente os programas utilizam a instrução de divisão só para obter o resto, ignorando o quociente.

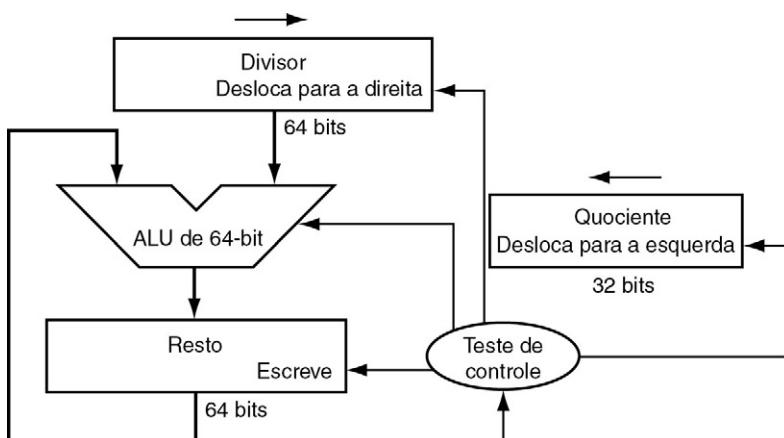
O algoritmo básico de divisão, que aprendemos na escola, tenta ver o quanto um número pode ser subtraído, criando um dígito do quociente em cada tentativa. Nossa exemplo decimal cuidadosamente selecionado usa apenas os números 0 e 1, de modo que é fácil descobrir quantas vezes o divisor cabe na parte do dividendo: deve ser 0 ou 1. Os números binários contêm apenas 0 ou 1, de modo que a divisão binária é restrita a essas duas opções, simplificando, assim, a divisão binária.

Vamos supor que o dividendo e o divisor sejam positivos e, logo, o quociente e o resto sejam não negativos. Os operandos da divisão e os dois resultados são valores de 32 bits, e ignoraremos o sinal por enquanto.

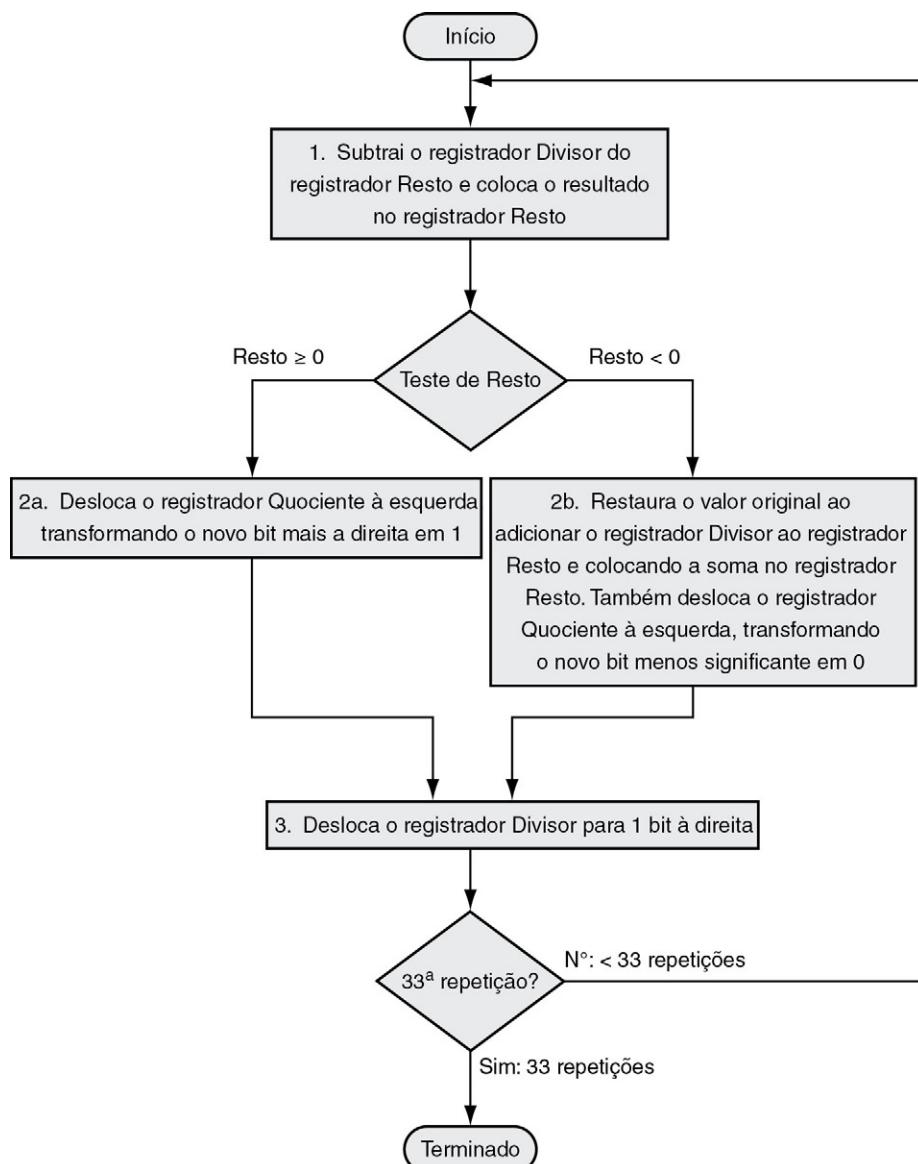
### Algoritmo e hardware de divisão

A Figura 3.9 mostra o hardware para imitar nosso algoritmo da escola. Começamos com o registrador Quociente de 32 bits definido como 0. Cada iteração do algoritmo precisa deslocar o divisor para a direita um dígito, de modo que começaremos com o divisor colocado na metade esquerda do registrador Divisor de 64 bits e o deslocaremos para a direita 1 bit a cada etapa, a fim de alinhá-lo com o dividendo. O registrador Resto é inicializado com o dividendo.

A Figura 3.10 mostra três etapas do primeiro algoritmo de divisão. Ao contrário dos humanos, o computador não é inteligente o bastante para saber, com antecedência, se o divisor é menor do que o dividendo. Ele primeiro precisa subtrair o divisor na etapa 1; lembre-se de que é assim que realizamos a comparação na instrução set on less than. Se o resultado for positivo, o divisor foi menor ou igual ao dividendo, de modo que geramos um 1 no quociente (etapa 2a). Se o resultado é negativo, a próxima etapa é restaurar o valor original, somando o divisor de volta ao resto e gerar um 0 no quociente (etapa 2b). O divisor é deslocado para a direita e depois repetimos. O resto e o quociente serão encontrados em seus registradores de mesmo nome depois que as iterações terminarem.



**FIGURA 3.9 Primeira versão do hardware de divisão.** O registrador Divisor, a ALU e o registrador Resto possuem 64 bits de largura, com apenas o registrador Quociente tendo 32 bits. O divisor de 32 bits começa na metade esquerda do registrador Divisor e é deslocado 1 bit para a direita em cada iteração. O resto é inicializado com o dividendo. O controle decide quando deslocar os registradores Divisor e Quociente e quando escrever o novo valor para o registrador Resto.



**FIGURA 3.10 Um algoritmo de divisão, usando o hardware da Figura 3.9.** Se o Resto é positivo, o divisor coube no dividendo, de modo que a etapa 2a gera um 1 no quociente. Um Resto negativo após a etapa 1 significa que o divisor não coube no dividendo, de modo que a etapa 2b gera um 0 no quociente e soma o divisor ao resto, revertendo, assim, a subtração da etapa 1. O deslocamento final, na etapa 3, alinha o divisor corretamente, em relação ao dividendo, para a próxima iteração. Essas etapas são repetidas 33 vezes.

### Um algoritmo de divisão

Usando uma versão de 4 bits do algoritmo para economizar páginas, vamos tentar dividir  $7_{dec}$  por  $2_{dec}$ , ou  $0000\ 0111_{bin}$  por  $0010_{bin}$ .

### EXEMPLO

A Figura 3.11 mostra o valor de cada registrador para cada uma das etapas, com o quociente sendo  $3_{dec}$  e o resto sendo  $1_{dec}$ . Observe que o teste na etapa 2 (se o resto é positivo ou negativo) simplesmente testa se o bit de sinal do registrador Resto é um 0 ou um 1. O requisito surpreendente desse algoritmo é que ele utiliza  $n + 1$  etapas para obter o quociente e resto corretos.

### RESPOSTA

Iteração	Etapa	Quociente	Divisor	Resto
0	Valores iniciais	0000	0010 0000	0000 0111
1	1: Resto = Resto - Div	0000	0010 0000	①110 0111
	2b: Resto < 0 $\Rightarrow$ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Desloca Div direita	0000	0001 0000	0000 0111
2	1: Resto = Resto - Div	0000	0001 0000	①111 0111
	2b: Resto < 0 $\Rightarrow$ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Desloca Div direita	0000	0000 1000	0000 0111
3	1: Resto = Resto - Div	0000	0000 1000	①111 1111
	2b: Resto < 0 $\Rightarrow$ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Desloca Div direita	0000	0000 0100	0000 0111
4	1: Resto = Resto - Div	0000	0000 0100	②000 0011
	2a: Resto $\geq$ 0 $\Rightarrow$ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Desloca Div direita	0001	0000 0010	0000 0011
5	1: Resto = Resto - Div	0001	0000 0010	②000 0001
	2a: Resto $\geq$ 0 $\Rightarrow$ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Desloca Div direita	0011	0000 0001	0000 0001

**FIGURA 3.11 Exemplo de divisão usando o algoritmo da Figura 3.10.** O bit examinado para determinar a próxima etapa está em destaque.

Esse algoritmo e esse hardware podem ser refinados para que sejam mais rápidos e menos dispendiosos. A rapidez vem do deslocamento dos operandos e do quociente no mesmo momento da subtração. Essa melhoria divide ao meio a largura do somador e dos registradores, observando onde existem partes não usadas dos registradores e somadores. A Figura 3.12 mostra o hardware revisado.

### Divisão com sinal

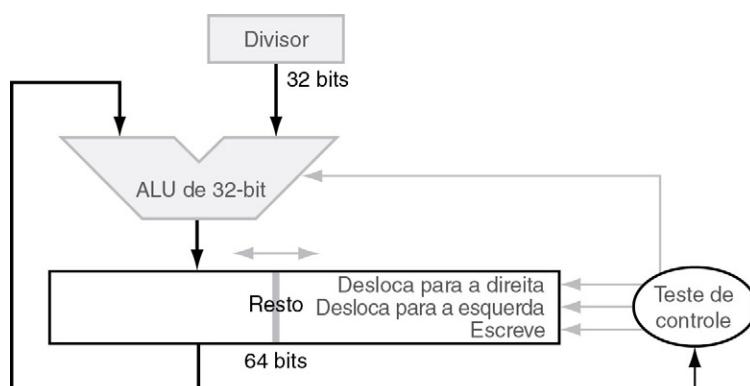
Até aqui, ignoramos os números com sinal na divisão. A solução mais simples é lembrar os sinais do divisor e do dividendo e depois negar o quociente se os sinais forem diferentes.

**Detalhamento:** Uma complicação da divisão com sinal é que também temos de definir o sinal do resto. Lembre-se de que a seguinte equação precisa ser sempre mantida:

$$\text{Dividendo} = \text{Quociente} \times \text{Divisor} + \text{Resto}$$

Para entender como definir o sinal do resto, vejamos o exemplo da divisão de todas as combinações de  $\pm 7_{\text{dec}}$  por  $\pm 2_{\text{dec}}$ . O primeiro caso é fácil:

$$+7 \div +2: \text{Quociente} = +3, \text{Resto} = +1$$



**FIGURA 3.12 Uma versão melhorada do hardware de divisão.** O registrador Divisor, a ALU e o registrador Quociente possuem 32 bits de largura, com apenas o registrador Resto ficando com 64 bits. Em comparação com a Figura 3.9, os registradores ALU e Divisor são divididos ao meio, e o resto é deslocado à esquerda. Essa versão também combina o registrador Quociente com a metade direita do registrador Resto. (Assim como na Figura 3.6, o registrador Resto na realidade deveria ter 65 bits para garantir que o carry do somador não se perca.)

Verificando os resultados:

$$7 = 3 \times 2 + (+1) = 6 + 1$$

Se você mudar o sinal do dividendo, o quociente também precisa mudar:

$$-7 \div +2 : \text{Quociente} = -3$$

Reescrevendo nossa fórmula básica para calcular o resto:

$$\text{Resto} = (\text{Dividendo} - \text{Quociente} \times \text{Divisor}) = -7 - (-3 \times +2) = -7 - (-6) = -1$$

Assim,

$$-7 \div +2 : \text{Quociente} = -3, \text{ Resto} = -1$$

Verificando os resultados novamente:

$$-7 = -3 \times 2 + (-1) = -6 - 1$$

O motivo pelo qual a resposta não é um quociente de  $-4$  e um resto de  $+1$ , que também caberia nessa fórmula, é que o valor absoluto do quociente mudaria dependendo do sinal do dividendo e do divisor! Logicamente, se

$$-(x + y) \neq (-x) \div y$$

a programação seria um desafio ainda maior. Esse comportamento anômalo é evitado seguindo-se a regra de que o dividendo e o resto devem ter os mesmos sinais, não importa quais sejam os sinais do divisor e do quociente.

Calculamos as outras combinações seguindo a mesma regra:

$$+7 \div -2 : \text{Quociente} = -3, \text{ Resto} = +1$$

$$-7 \div -2 : \text{Quociente} = +3, \text{ Resto} = -1$$

Assim, o algoritmo de divisão com sinal nega o quociente se os sinais dos operandos foram opostos e faz com que o sinal do resto diferente de zero corresponda ao dividendo.

## Divisão mais rápida

Usamos muitos somadores para agilizar a multiplicação, mas não podemos fazer o mesmo truque para a divisão. O motivo é que precisamos saber o sinal da diferença antes de podermos realizar a próxima etapa do algoritmo, enquanto, com a multiplicação, poderíamos calcular os 32 produtos parciais imediatamente.

Existem técnicas para produzir mais de um bit do quociente por etapa. A técnica de divisão SRT tenta descobrir vários bits do quociente por etapa, usando uma pesquisa numa tabela baseada nos bits mais significativos do dividendo e do resto. Ela conta com as etapas subsequentes para corrigir escolhas erradas. Um valor comum hoje é 4 bits. A chave é descobrir o valor para subtrair. Com a divisão binária, existe somente uma única opção. Esses algoritmos utilizam 6 bits do resto e 4 bits do divisor para indexar uma tabela que determina a opção para cada etapa.

A precisão desse método rápido depende de haver valores apropriados na tabela de pesquisa. A falácia apresentada na Seção 3.8 mostra o que pode acontecer se a tabela estiver incorreta.

## Divisão no MIPS

Você já pode ter observado que o mesmo hardware sequencial pode ser usado para multiplicação e divisão nas [Figuras 3.6 e 3.12](#). O único requisito é um registrador de 64 bits, que pode deslocar para a esquerda ou para a direita e uma ALU de 32 bits que soma ou subtrai. Logo, o MIPS utiliza os registradores Hi e Lo de 32 bits, tanto para multiplicação quanto para divisão. Como poderíamos esperar do algoritmo anterior, Hi contém o resto, e Lo contém o quociente após o término da instrução de divisão.

Para lidar com inteiros com sinal e inteiros sem sinal, o MIPS possui duas instruções: *divide* (*div*) e *divide unsigned* (*divu*). O montador MIPS permite que as instruções de

divisão especifiquem três registradores, gerando as instruções *mflo* ou *mfhi* para colocar o resultado desejado em um registrador de uso geral.

## Resumo

O suporte de hardware comum para multiplicação e divisão permite que o MIPS ofereça um único par de registradores de 32 bits usados tanto para multiplicar quanto para dividir. A [Figura 3.13](#) resume os acréscimos à arquitetura MIPS das duas últimas seções.

**Linguagem de assembly MIPS**

Categoría	Instrucción	Exemplo	Significado	Comentarios
Aritmética	add	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Três operandos, overflow detectado
	subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	Três operandos, overflow detectado
	add immediate	addi \$s1,\$s2,100	\$s1 = \$s2 + 100	+ constante, overflow detectado
	add unsigned	addu \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Três operandos, overflow detectado
	subtract unsigned	subu \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	Três operandos, overflow detectado
	add immediate unsigned	addiu \$s1,\$s2,100	\$s1 = \$s2 + 100	+ constante, overflow detectado
	move from coprocessor register	mfc0 \$s1,\$epc	\$s1 = \$epc	Copiar Exceção PC + regs especiais
	multiply	mult \$s2,\$s3	Hi, Lo = \$s2 × \$s3	Produto de 64-bit com sinal em Hi, Lo
	multiply unsigned	multu \$s2,\$s3	Hi, Lo = \$s2 × \$s3	Produto de 64-bit sem sinal em Hi, Lo
	divide	div \$s2,\$s3	Lo = \$s2 / \$s3, Hi = \$s2 mod \$s3	Lo = quociente, Hi = resto
Transferência de dados	divide unsigned	divu \$s2,\$s3	Lo = \$s2 / \$s3, Hi = \$s2 mod \$s3	Quociente sem sinal e resto
	move from Hi	mfhi \$s1	\$s1 = Hi	Costumava copiar de Hi
	move from Lo	mflo \$s1	\$s1 = Lo	Costumava copiar de Lo
	load word	lw \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Word da memória para o registrador
	store word	sw \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Word do registrador para a memória
	load half unsigned	lhu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword da memória para o registrador
	store half	sh \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Halfword do registrador para a memória
	load byte unsigned	lbu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte da memória para o registrador
	store byte	sb \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Byte do registrador para a memória
	load linked word	ll \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Carrega word como primeira metade da troca atómica
Lógica	store conditional word	sc \$s1,20(\$s2)	Memory[\$s2+20]=\$s1;\$s1=0 or 1	Armazena word como primeira metade da troca atómica
	load upper immediate	lui \$s1,100	\$s1 = 100 * 2 <sup>16</sup>	Carrega constante nos 16 bits altos
	AND	AND \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	Três operandos registradores; AND bit por bit
	OR	OR \$s1,\$s2,\$s3	\$s1 = \$s2   \$s3	Três operandos registradores; OR bit por bit
	NOR	NOR \$s1,\$s2,\$s3	\$s1 = ~ (\$s2   \$s3)	Três operandos registradores; NOR bit por bit
	AND immediate	andi \$s1,\$s2,100	\$s1 = \$s2 & 100	AND bit por bit com constante
	OR immediate	ori \$s1,\$s2,100	\$s1 = \$s2   100	OR bit por bit com constante
Desvio condicional	shift left logical	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	Desloca à esquerda pela constante
	shift right logical	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	Desloca à direita pela constante
	branch on equal	beq \$s1,\$s2,25	if(\$s1 == \$s2) go to PC + 4 + 100	Teste idêntico; desvio relativo ao PC
	branch on not equal	bne \$s1,\$s2,25	if(\$s1 != \$s2) go to PC + 4 + 100	Teste não-idêntico; relativo ao PC
	set on less than	slt \$s1,\$s2,\$s3	if(\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Comparação menos que; complementos de dois
	set less than immediate	slti \$s1,\$s2,100	if(\$s2 < 100) \$s1 = 1; else \$s1=0	Comparação < constante; complementos de dois
	set less than unsigned	sltu \$s1,\$s2,\$s3	if(\$s2 < \$s3) \$s1 = 1; else \$s1=0	Comparação menos que; números naturais
Pulo incondicional	set less than immediate unsigned	sltiu \$s1,\$s2,100	if(\$s2 < 100) \$s1 = 1; else \$s1 = 0	Comparação < constante; números naturais
	jump	j 2500	go to 10000	Pula para o endereço alvo
	jump register	jr \$ra	go to \$ra	Para troca, procedimento de retorno
	jump and link	jal 2500	\$ra = PC + 4; go to 10000	Para chamada de procedimento

**FIGURA 3.13 Arquitetura MIPS revelada até aqui.** A memória e os registradores da arquitetura MIPS não estão incluídos por questões de espaço, mas esta seção acrescentou os registradores Hi e Lo para dar suporte à multiplicação e à divisão. A linguagem de máquina do MIPS aparece no Guia de referência rápida, no início do livro.

Instruções de divisão MIPS ignoram o overflow, de modo que o software precisa determinar se o quociente é muito grande. Além do overflow, a divisão também pode resultar em um cálculo impróprio: divisão por 0. Alguns computadores distinguem esses dois eventos anômalos. O software MIPS precisa verificar o divisor para descobrir a divisão por 0 e também o overflow.

## Interface hardware/software

**Detalhamento:** um algoritmo ainda mais rápido não soma imediatamente o divisor se o resto for negativo. Ele simplesmente soma o dividendo ao resto deslocado na etapa seguinte, pois  $(r + d) \times 2 - d = r \times 2 + d \times 2 - d = r \times 2 + d$ . Esse algoritmo de divisão *sem restauração*, que usa um clock por etapa, é explorado ainda mais nos exercícios; o algoritmo aqui apresentado é chamado de divisão com *restauração*. Um terceiro algoritmo, que não salva o resultado da subtração se ele for negativo, é chamado algoritmo de divisão *sem o retorno esperado*. Ele reduz em média um terço de operações aritméticas.

## 3.5

## Ponto flutuante

Indo além de inteiros com e sem sinal, as linguagens de programação admitem números com frações, que são chamados *reais* na matemática. Aqui estão alguns exemplos de números reais:

$3,14159265\dots_{\text{dec}}$  (pi)

$2,71828\dots_{\text{dec}}$  ( $e$ )

$0,00000001_{\text{dec}}$  ou  $1,0_{\text{dec}} \times 10^{-9}$  (segundos em um nanosegundo)

$3.155.760.000_{\text{dec}}$  ou  $3,15576_{\text{dec}} \times 10^9$  (segundos em um século típico)

Observe que, no último caso, o número não representou uma fração pequena, mas foi maior do que poderíamos representar com um inteiro de 32 bits com sinal. A notação alternativa para os dois últimos números é chamada **notação científica**, que tem um único dígito à esquerda do ponto decimal. Um número na notação científica que não tem 0s à esquerda do ponto decimal é chamado de número **normalizado**, que é o modo normal como o escrevemos. Por exemplo,  $1,0_{\text{dec}} \times 10^{-9}$  está em notação científica normalizada, mas  $0,1_{\text{dec}} \times 10^{-8}$  e  $10,0_{\text{dec}} \times 10^{-10}$  não estão.

Assim como podemos mostrar números decimais em notação científica, também podemos mostrar números binários em notação científica:

$$1,0_{\text{bin}} \times 2^{-1}$$

Para manter um número binário na forma normalizada, precisamos de uma base que possamos aumentar ou diminuir exatamente pelo número de bits que o número precisa ser deslocado para ter um dígito diferente de zero à esquerda do ponto decimal. Somente uma base de 2 atende à nossa necessidade. Como a base não é 10, também precisamos de um novo nome para o ponto decimal; o *ponto binário* funcionará bem.

A aritmética computacional, que admite tais números, é chamada **ponto flutuante** porque representa os números em que o ponto binário não é fixo, como acontece para os inteiros. A linguagem de programação C utiliza o nome *float* para esses números. Assim como na notação científica, os números são representados como um único dígito diferente de zero à esquerda do ponto binário. Em binário, o formato é

$$1,xxxxxxxx_{\text{bin}} \times 2^{yyyy}$$

*A velocidade não o leva a lugar algum se você estiver na direção errada.*

Provérbio americano

**notação científica** Uma notação que apresenta números com um único dígito à esquerda do ponto decimal.

**normalizado** Um número na notação de ponto flutuante que não possui 0s à esquerda do ponto decimal.

**ponto flutuante** Aritmética computacional que representa os números em que o ponto binário não é fixo.

(Embora o computador represente o expoente na base 2, bem como o restante do número, para simplificar a notação, mostramos o expoente em decimal.)

Uma notação científica padrão para os números reais no formato normalizado oferece três vantagens. Ela simplifica a troca de dados, que incluem números em ponto flutuante; simplifica os algoritmos aritméticos de ponto flutuante, por saber que os números sempre estarão nessa forma; e aumenta a precisão dos números que podem ser armazenados em uma palavra, pois os 0s desnecessários são substituídos por dígitos reais à direita do ponto binário.

## Representação em ponto flutuante

**fração** O valor, geralmente entre 0 e 1, colocado no campo de fração.

**expoente** No sistema de representação numérica da aritmética de ponto flutuante, o valor colocado no campo de expoente.

Um projetista de uma representação em ponto flutuante precisa encontrar um compromisso entre o tamanho da **fração** e o tamanho do **expoente**, pois um tamanho de palavra fixo significa que você precisa tirar um bit de um para acrescentar um bit ao outro. Essa troca é entre a precisão e o intervalo: aumentar o tamanho da fração melhora a precisão da fração, enquanto aumentar o tamanho do expoente aumenta o intervalo de números que podem ser representados. Conforme nosso guia de projetos do Capítulo 2 nos lembra, um bom projeto exige um bom compromisso.

Os números em ponto flutuante normalmente são múltiplos do tamanho de uma palavra. A representação de um número em ponto flutuante MIPS aparece a seguir, em que *s* é o sinal do número de ponto flutuante (1 significa negativo), **expoente** é o valor do campo de expoente com 8 bits (incluindo o sinal do expoente) e **fração** é o número de 23 bits. Essa representação é chamada *sinal e magnitude*, pois o sinal possui um bit separado do restante do número.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>s</i>	expoente																										fração				
1 bit	8 bits																										23 bits				

Em geral, os números em ponto flutuante estão no formato

$$(-1)^s \times F \times 2^E$$

F envolve o valor no campo de fração e E envolve o valor no campo de expoente; o relacionamento exato com esses campos será explicado em breve. (Logo veremos que o MIPS faz algo ligeiramente mais sofisticado.)

Esses tamanhos escolhidos de expoente e fração dão à aritmética do computador MIPS um intervalo extraordinário. Frações quase tão pequenas quanto  $2,0_{\text{dec}} \times 10^{-38}$  e números quase tão grandes quanto  $2,0_{\text{dec}} \times 10^{38}$  podem ser representados em um computador. Infelizmente, extraordinário é diferente de infinito, de modo que ainda é possível que os números sejam muito grandes. Assim, interrupções por overflow podem ocorrer na aritmética de ponto flutuante e também na aritmética de inteiros. Observe que **overflow** aqui significa que o expoente é muito grande para ser representado no campo de expoente.

O ponto flutuante também oferece um novo tipo de evento excepcional. Assim como os programadores desejariam saber quando calcularam um número muito grande para ser representado, também desejariam saber se a fração diferente de zero que estão calculando tornou-se tão pequena que não pode ser representada; os dois eventos poderiam resultar em um programa com respostas incorretas. Para distinguir do overflow, as pessoas chamam esse evento de **underflow**. Essa situação ocorre quando o expoente negativo é muito grande para caber no campo de expoente.

Uma maneira de reduzir as chances de underflow ou overflow é oferecer outro formato que tenha um expoente maior. Em C, esse número é chamado *double*, e as operações sobre doubles são indicadas como aritmética de ponto flutuante com **precisão dupla**; o ponto flutuante com **precisão simples** é o nome do formato anterior.

A representação de um número em ponto flutuante com precisão dupla utiliza duas palavras MIPS, como vemos a seguir, em que *s* ainda é o sinal do número, **expoente** é o valor do campo de expoente em 11 bits, e **fração** é o número de 52 bits na fração.

**precisão dupla** Um valor de ponto flutuante representado em duas palavras de 32 bits.

**precisão simples** Um valor de ponto flutuante representado em uma única palavra de 32 bits.



A precisão dupla do MIPS permite números quase tão pequenos quanto  $2,0_{\text{dec}} \times 10^{-308}$  e quase tão grandes quanto  $2,0_{\text{dec}} \times 10^{308}$ . Embora a precisão dupla não aumente o intervalo do expoente, sua principal vantagem é sua maior precisão, em consequência do significando maior.

Esses formatos vão além do MIPS. Eles fazem parte do *padrão de ponto flutuante IEEE 754*, encontrado em praticamente todo computador inventado desde 1980. Esse padrão melhorou bastante tanto a facilidade de portar programas de ponto flutuante quanto a qualidade da aritmética computacional.

Para colocar ainda mais bits no significando, o IEEE 754 deixa implícito o bit 1 inicial dos números binários normalizados. Logo, o número tem, na realidade, 24 bits de largura na precisão simples (1 implícito e fração de 23 bits) e 53 bits de extensão na precisão dupla (1 + 52). Para ser exato, usamos o termo *significado* a fim de representar o número de 24 ou 53 bits que é 1 mais a fração, e *fração* quando queremos dizer o número de 23 ou 52 bits. Como 0 não possui um 1 inicial, ele recebe o valor de expoente reservado 0, de modo que o hardware não lhe acrescente um 1 inicial.

Assim,  $00\dots00_{\text{bin}}$  representa 0; a representação do restante dos números usa a forma de antes, com o 1 oculto sendo acrescentado:

$$(-1)^S \times (1 + \text{Fração}) \times 2^E$$

em que os bits da fração representam um número entre 0 e 1, e E especifica o valor no campo de expoente, que será explicado em detalhes mais adiante. Se numerarmos os bits da fração da esquerda para a direita de s1, s2, s3, ..., então o valor é

$$(-(-1)^s \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + \dots) \times 2^E)$$

A Figura 3.14 mostra as codificações dos números de ponto flutuante IEEE 754. Outros recursos do IEEE 754 são símbolos especiais para representar eventos incomuns. Por exemplo, em vez de interromper em uma divisão por 0, o software pode definir o resultado para um padrão de bits que represente  $+\infty$  ou  $-\infty$ ; o maior expoente é reservado a esses símbolos especiais. Quando o programador imprime os resultados, o programa imprimirá um símbolo de infinito. (Para os que são matematicamente treinados, a finalidade do infinito é formar o fechamento topológico dos reais.)

Precisão simples		Precisão dupla		Objeto representado
Expoente	Fração	Expoente	Fração	
0	0	0	0	0
0	não zero	0	não zero	± número desnormalizado
1–254	qualquer coisa	1–2046	Anything	± número ponto flutuante
255	0	2047	0	± infinito
255	não zero	2047	não zero	Not a Number (NaN)

**FIGURA 3.14 Codificação IEE 754 dos números de ponto flutuante.** Um bit de sinal separado determina o sinal. Os números desnormalizados são descritos no *Detalhamento* na página 270. Essa informação também é encontrada na coluna 4 do Guia de Instrução Rápida no início deste livro.

O IEEE 754 até mesmo possui um símbolo para o resultado de operações inválidas, como 0/0, ou a subtração entre infinito e infinito. Esse símbolo é *NaN*, de *Not a Number* (não é um número). A finalidade dos NaNs é permitir que os programadores adiem alguns testes e decisões para outro momento no programa, quando for conveniente.

Os projetistas do IEEE 754 também queriam uma representação de ponto flutuante que pudesse ser facilmente processada por comparações de inteiros, especialmente para ordenação. Esse desejo é o motivo pelo qual o sinal está no bit mais significativo, permitindo um teste rápido de menor que, maior que ou igual a 0. (Isso é um pouco mais complicado do que uma ordenação simples de inteiros, pois essa notação é basicamente sinal e magnitude, em vez do complemento de dois.)

Colocar o expoente antes do significando simplifica a ordenação dos números de ponto flutuante usando instruções de comparação de inteiros, pois os números com expoentes maiores são maiores do que os números com expoentes menores, desde que os dois expoentes tenham o mesmo sinal.

Expoentes negativos impõem um desafio à ordenação simplificada. Se usarmos o complemento de dois ou qualquer outra notação em que os expoentes negativos têm um 1 no bit mais significativo do campo de expoente, um expoente negativo se parecerá com um número grande. Por exemplo,  $1,0_{\text{bin}} \times 2^{-1}$  seria representado como

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

(Lembre-se de que o 1 à esquerda do ponto é implícito no significando.) O valor  $1,0_{\text{bin}} \times 2^{+1}$  seria semelhante a um número binário menor

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

A notação desejável, portanto, precisa representar o expoente mais negativo como 00 ... 00<sub>bin</sub> e o mais positivo como 11 ... 11<sub>bin</sub>. Essa convenção é chamada *notação deslocada*, com a inclinação (bias) sendo o número subtraído da representação normal, sem sinal, para determinar o valor real.

O IEEE 754 usa um bias de 127 para a precisão simples, de modo que -1 é representado pelo padrão de bits do valor  $-1 + 127_{\text{dec}} = 126_{\text{dec}} = 0111\ 1110_{\text{bin}}$ , e +1 é representado por  $1 + 127 = 128_{\text{dec}} = 1000\ 0000_{\text{bin}}$ . O expoente deslocado significa que o valor representado por um número em ponto flutuante é, na realidade:

$$(-1)^S \times (1 + \text{Fração}) \times 2^{(\text{Expoente-Bias})}$$

A faixa de números de precisão simples é, então, desde

$$\pm 1.0000000000000000000000000000000_{\text{bin}} \times 2^{-126}$$

até

$$\pm 1.111111111111111111111111111111_{\text{bin}} \times 2^{+127}$$

Vamos mostrar a representação.

### Representação de ponto flutuante

Mostre a representação binária IEEE 754 para o número  $-0,75_{\text{dec}}$  em precisão simples e dupla.

#### EXEMPLO

O número  $-0,75_{\text{dec}}$  também é

$$-3/4_{\text{dec}} \text{ ou } -3/2^2_{\text{dec}}$$

#### RESPOSTA

Ele também é representado pela fração binária

$$-11_{\text{bin}} / 2^2_{\text{dec}} \text{ ou } -0,11_{\text{bin}}$$

Em notação científica, o valor é

$$-0,11_{\text{bin}} \times 2^0$$

e, na notação científica normalizada, ele é

$$-1,1_{\text{bin}} \times 2^{-1}$$

A representação geral para um número de precisão simples é

$$(-1)^S \times (1 + \text{Fração}) \times 2^{(\text{Expoente}-127)}$$

Quando subtraímos o bias 127 do expoente de  $-1,1_{\text{bin}} \times 2^{-1}$ , o resultado é

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000_{\text{bin}}) \times 2^{(126-127)}$$

A representação binária de precisão simples de  $0,75_{\text{dec}}$ , portanto, é

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit                    8 bits

23 bits

A representação em precisão dupla é

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{bin}}) \times 2^{(1022-1023)}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	0	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit                    11 bits

20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

32 bits

Agora, vamos experimentar na outra direção.

### Convertendo ponto flutuante binário para decimal

Que número decimal é representado por este float de precisão simples?

**EXEMPLO**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

O bit de sinal é 1, o campo de expoente contém 129 e o campo de fração contém  $1 \times 2^{-2} = 1/4$ , ou 0,25. Usando a equação básica,

**RESPOSTA**

$$\begin{aligned} (-1)^S \times (1 + \text{Fração}) \times 2^{(\text{Expoente}-Bias)} &= (-1)^1 \times (1 + 0,25) \times 2^{(129-127)} \\ &= -1 \times 1,25 \times 2^2 \\ &= -1,25 \times 4 \\ &= -5,0 \end{aligned}$$

Nas próximas seções, daremos os algoritmos para a adição e multiplicação em ponto flutuante. Em seu núcleo, eles utilizam operações inteiras correspondentes nos significandos, mas é preciso que haja manutenção extra para lidar com os expoentes e normalizar o resultado. Primeiro, oferecemos uma derivação intuitiva dos algoritmos em decimal, e depois uma versão mais detalhada, binária, nas figuras.

**Detalhamento:** em uma tentativa de aumentar o intervalo sem remover bits do significando, alguns computadores antes do padrão IEEE 754 usavam uma base diferente de 2. Por exemplo, os computadores mainframe IBM 360 e 370 usam a base 16. Como mudar o expoente no IBM em um significa deslocar o significando em 4 bits, os números de base 16 “normalizados” podem ter até 3 bits à esquerda do ponto em 0s! Logo, os dígitos hexadecimais significam que até 3 bits precisam ser removidos do significando, o que leva a problemas surpreendentes na precisão da aritmética de ponto flutuante. Mainframes IBM recentes admitem IEEE 754 além do formato hexa.

### Adição em ponto flutuante

Vamos somar os números na notação científica manualmente, para ilustrar os problemas na adição em ponto flutuante:  $9,999_{\text{dec}} \times 10^1 + 1,610_{\text{dec}} \times 10^{-1}$ . Suponha que só possamos armazenar quatro dígitos decimais do significando e dois dígitos decimais do expoente.

Etapa 1. Para poder somar esses números corretamente, temos de alinhar o ponto decimal do número que possui o menor expoente. Logo, precisamos de uma forma do número menor,  $1,610_{\text{dec}} \times 10^{-1}$ , que combine com o expoente maior. Obtemos isso observando que existem várias representações de um número em ponto flutuante não normalizado na notação científica:

$$1,610_{\text{dec}} \times 10^{-1} = 0,1610_{\text{dec}} \times 10^0 = 0,01610_{\text{dec}} \times 10^1$$

O número da direita é a versão que desejamos, pois seu expoente combina com o expoente do maior número,  $9,999_{\text{dec}} \times 10^1$ . Assim, a primeira etapa desloca o significando do menor número à direita até que seu expoente corrigido combine com o do maior número. Contudo, só podemos representar quatro dígitos decimais, de modo que, após o deslocamento, o número é, na realidade:

$$0,016_{\text{dec}} \times 10^1$$

Etapa 2. Em seguida, vem a adição dos significandos:

$$\begin{array}{r} 9,999_{\text{dec}} \\ + 0,016_{\text{dec}} \\ \hline 10,015_{\text{dec}} \end{array}$$

A soma é  $10,015_{\text{dec}} \times 10^1$ .

Etapa 3. Essa soma não está na notação científica normalizada, de modo que precisamos ajustá-la:

$$10,015_{\text{dec}} \times 10^1 = 1,0015_{\text{dec}} \times 10^2$$

Assim, depois da adição, podemos ter de deslocar a soma para colocá-la na forma normalizada, ajustando o expoente de acordo. Esse exemplo mostra o deslocamento para a direita, mas, se um número fosse positivo e o outro negativo, é possível que a soma tenha muitos 0s iniciais, exigindo deslocamentos à esquerda. Sempre que o expoente é aumentado ou diminuído, temos de verificar o overflow ou underflow – ou seja, temos de verificar se o expoente ainda cabe em seu campo.

Etapa 4. Como pressupomos que o significando só pode ter quatro dígitos de extensão (excluindo o sinal), temos de arredondar o número. Em nosso algoritmo que aprendemos

na escola, as regras truncam o número se o dígito à direita do ponto desejado estiver entre 0 e 4 e somamos 1 ao dígito se o número à direita estiver entre 5 e 9. O número

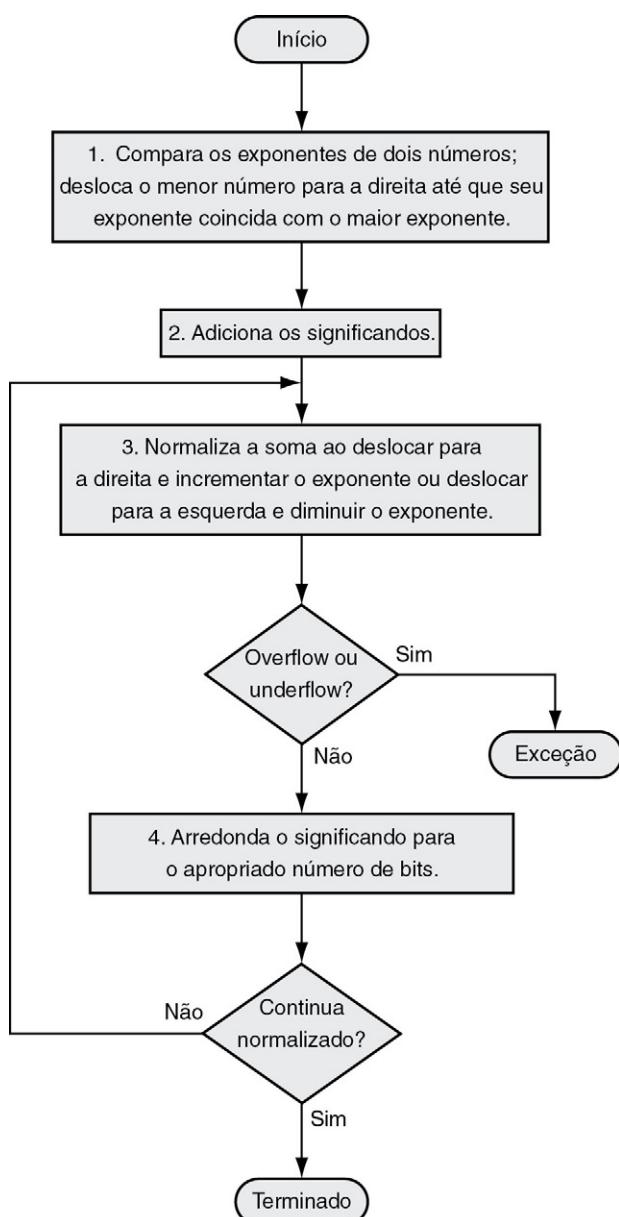
$$1,0015_{\text{dec}} \times 10^2$$

é arredondado para quatro dígitos no significando, passando para

$$1,002_{\text{dec}} \times 10^2$$

pois o quarto dígito à direita do ponto decimal estava entre 5 e 9. Observe que, se não tivermos sorte no arredondamento, como ao somar 1 a uma sequência de 9s, a soma não pode mais ser normalizada, sendo necessário realizar a etapa 3 novamente.

A [Figura 3.15](#) mostra o algoritmo para a adição binária de ponto flutuante que acompanha este exemplo em decimal. As etapas 1 e 2 são semelhantes ao exemplo que discutimos:



**FIGURA 3.15 Adição de ponto flutuante.** O caminho normal é executar as etapas 3 e 4 uma vez, mas se o arredondamento fizer com que a soma fique não normalizada, temos de repetir a etapa 3.

ajustar o significando do número com o menor expoente e depois somar os dois significandos. A etapa 3 normaliza os resultados, forçando uma verificação de overflow ou underflow. O teste de overflow e underflow na etapa 3 depende da precisão dos operandos. Lembre-se de que o padrão de todos os bits zero no expoente é reservado e usado para a representação de ponto flutuante de zero. Além disso, o padrão de todos os bits um no expoente é reservado para indicar valores e situações fora do escopo dos números de ponto flutuante normais (veja a Seção “Detalhamento” na página 270). Assim, para a precisão simples, o expoente máximo é 127, e o expoente mínimo é -126. Os limites para precisão dupla são 1023 e -1022.

## EXEMPLO

## RESPOSTA

### Adição de ponto flutuante em decimal

Tente somar os números  $0,5_{\text{dec}}$  e  $-0,4375_{\text{dec}}$  em binário usando o algoritmo da Figura 3.15.

Primeiro, vejamos a versão binária dos dois números na notação científica normalizada, supondo que mantemos 4 bits de precisão:

$$\begin{aligned} 0,5_{\text{dec}} &= 1/2_{\text{dec}} = 1/2^1_{\text{dec}} \\ &= 0,1_{\text{bin}} = 0,1_{\text{bin}} \times 2^0 = 1,000_{\text{bin}} \times 2^{-1} \\ -0,4375_{\text{dec}} &= -7/16_{\text{dec}} = -7/2^4_{\text{dec}} \\ &= -0,0111_{\text{bin}} = -0,0111_{\text{bin}} \times 2^0 = -1,110_{\text{bin}} \times 2^{-2} \end{aligned}$$

Agora, seguimos o algoritmo:

Etapa 1. O significando do número com o menor expoente ( $-1,11_{\text{bin}} \times 2^{-2}$ ) é deslocado para a direita até seu expoente combinar com o maior número:

$$-1,110_{\text{bin}} \times 2^{-2} = -0,111_{\text{bin}} \times 2^{-1}$$

Etapa 2. Some os significandos:

$$1,000_{\text{bin}} \times 2^{-1} + (-0,111_{\text{bin}} \times 2^{-1}) = 0,001_{\text{bin}} \times 2^{-1}$$

Etapa 3. Normalize a soma, verificando overflow ou underflow:

$$\begin{aligned} 0,001_{\text{bin}} \times 2^{-1} &= 0,010_{\text{bin}} \times 2^{-2} = 0,100_{\text{bin}} \times 2^{-3} \\ &= 1,000_{\text{bin}} \times 2^{-4} \end{aligned}$$

Como  $127 \geq 04 \geq -126$ , não existe overflow ou underflow. (O expoente deslocado seria  $-4 + 127$ , ou 123, que está entre 1 e 254, o menor e o maior expoente deslocado não reservado.)

Etapa 4. Arredonde a soma:

$$1,000_{\text{bin}} \times 2^{-4}$$

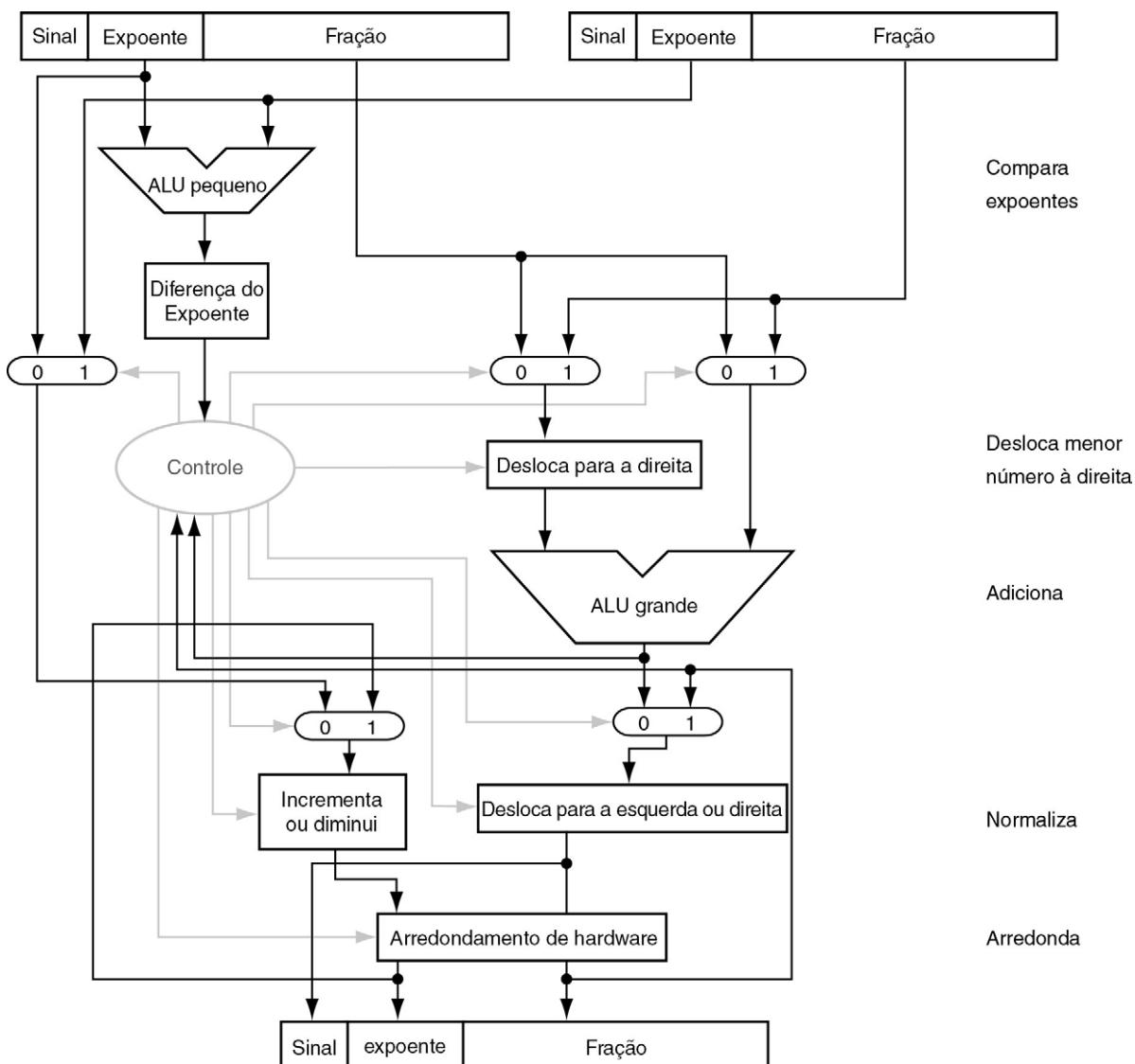
A soma já cabe exatamente em 4 bits, de modo que não há mudança nos bits em razão do arredondamento.

Essa soma é, então

$$\begin{aligned} 1,000_{\text{bin}} \times 2^{-4} &= 0,0001000_{\text{bin}} = 0,0001_{\text{bin}} \\ &= 1/2^4_{\text{dec}} = 1/16_{\text{dec}} = 0,0625_{\text{dec}} \end{aligned}$$

Essa soma é o que esperaríamos da soma de  $0,5_{\text{dec}}$  a  $-0,4375_{\text{dec}}$ .

Muitos computadores dedicam o hardware para executar operações de ponto flutuante o mais rápido possível. A Figura 3.16 esboça a organização básica do hardware para a adição de ponto flutuante.



**FIGURA 3.16 Diagrama de bloco de uma unidade aritmética dedicada à adição em ponto flutuante.** As etapas da Figura 3.15 correspondem a cada bloco, de cima para baixo. Primeiro, o expoente de um operando é subtraído do outro usando a ALU pequena para determinar qual é maior e quanto. Essa diferença controla os três multiplexadores; da esquerda para a direita, eles selecionam o maior expoente, o significando do menor número e o significando do maior número. O menor significando é deslocado para a direita, e depois os significandos são somados usando a ALU grande. A etapa de normalização, então, desloca a soma para a esquerda ou para a direita e incrementa ou diminui o expoente. O arredondamento, então, cria o resultado final, que pode exigir normalização novamente para produzir o resultado final.

### Multiplicação em ponto flutuante

Agora que já explicamos a adição em ponto flutuante, vamos experimentar a multiplicação em ponto flutuante. Começamos multiplicando os números decimais em notação científica na mão:  $1,110_{dec} \times 10^{10} \times 9,200_{dec} \times 10^{-5}$ . Suponha que possamos armazenar apenas quatro dígitos do significando e dois dígitos do expoente.

Etapa 1. Ao contrário da adição, calculamos o expoente do produto simplesmente somando os expoentes dos operandos:

$$\text{Novo expoente} = 10 + (-5) = 5$$

Vamos fazer isso com os expoentes deslocados, para obtermos o mesmo resultado:  $10 + 127 = 137$ , e  $-5 + 127 = 122$ , assim

$$\text{Novo expoente} = 137 + 122 = 259$$

Esse resultado é muito grande para o campo de expoente de 8 bits, de modo que há algo faltando! O problema é com o bias, pois estamos somando os biases e também os expoentes:

$$\text{Novo expoente} = (10 + 127) + (-5 + 127) = (5 + 2 \times 127) = 259$$

*De acordo com isso, para obter a soma deslocada correta quando somamos números deslocados, temos de subtrair o bias da soma:*

$$\text{Novo expoente} = 137 + 122 - 127 = 259 - 127 = 132 = (5 + 127)$$

e 5 é, na realidade, o expoente que calculamos inicialmente.

Etapa 2. Em seguida, vem a multiplicação dos significandos:

$$\begin{array}{r} 1,110_{\text{dec}} \\ \times 9,200_{\text{dec}} \\ \hline 0000 \\ 0000 \\ 2220 \\ 9990 \\ \hline 10212000_{\text{dec}} \end{array}$$

Existem três dígitos à direita do ponto decimal para cada operando, de modo que o ponto decimal é colocado seis dígitos a partir da direita no significando do produto:

$$10,212000_{\text{dec}}$$

Supondo que só possamos manter três dígitos à direita do ponto decimal, o produto é  $10,212 \times 10^5$ .

Etapa 3. Este produto não está normalizado, de modo que precisamos normalizá-lo:

$$10,212_{\text{dec}} \times 10^5 = 1,0212_{\text{dec}} \times 10^6$$

Assim, após a multiplicação, o produto pode ser deslocado para a direita um dígito, a fim de colocá-lo no formato normalizado, somando 1 ao expoente. Nesse ponto, podemos verificar o overflow e o underflow. O underflow pode ocorrer se os dois operandos forem pequenos – ou seja, se ambos tiverem expoentes negativos grandes.

Etapa 4. Consideramos que o significando tem apenas quatro dígitos de largura (excluindo o sinal), de modo que devemos arredondar o número. O número

$$1,0212_{\text{dec}} \times 10^6$$

é arredondado para quatro dígitos no significando, para

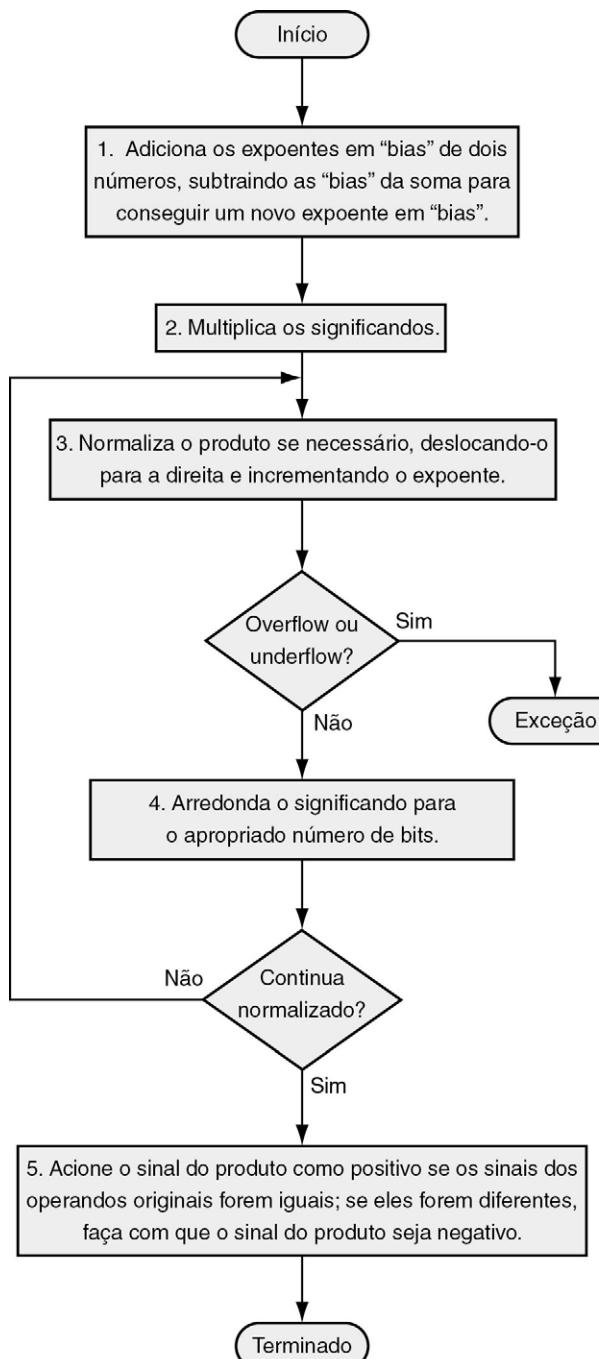
$$1,021_{\text{dec}} \times 10^6$$

Etapa 5. O sinal do produto depende dos sinais dos operandos originais. Se forem iguais, o sinal é positivo; caso contrário, é negativo. Logo, o produto é

$$+1,021_{\text{dec}} \times 10^6$$

O sinal da soma no algoritmo de adição foi determinado pela adição dos significandos; porém, na multiplicação, o sinal do produto é determinado pelos sinais dos operandos.

Mais uma vez, como mostra a [Figura 3.17](#), a multiplicação de números binários em ponto flutuante é muito semelhante às etapas que acabamos de concluir. Começamos calculando o novo expoente do produto, somando os expoentes deslocados, subtraindo um bias para obter o resultado apropriado. Em seguida, está a multiplicação de significandos, seguida por uma etapa de normalização opcional. O tamanho do expoente é verificado, em busca de overflow ou underflow, e depois o produto é arredondado. Se o arredondamento causar mais normalização, uma vez verificamos o tamanho do expoente. Finalmente, definimos o bit de sinal como 1 se os sinais dos operandos forem diferentes (produto negativo) ou como 0 se forem iguais (produto positivo).



**FIGURA 3.17 Multiplicação em ponto flutuante.** O caminho normal é executar as etapas 3 e 4 uma vez, mas se o arredondamento fizer com que a soma fique desnormalizada, temos de repetir a etapa 3.

**EXEMPLO****RESPOSTA****Multiplicação em ponto flutuante em decimal**

Vamos tentar multiplicar os números  $0,5_{dec}$  e  $-0,4375_{dec}$ , usando as etapas na [Figura 3.17](#).

Em binário, a tarefa é multiplicar  $1,000_{bin} \times 2^{-1}$  por  $-1,110_{bin} \times 2^{-2}$ .

Etapa 1. Somando os expoentes sem bias:

$$-1 + (-2) = -3$$

ou então, usando a representação deslocada:

$$\begin{aligned} (-1+127) + (-2+127) - 127 &= (-1-2) + (127+127-127) \\ &= -3+127=124 \end{aligned}$$

Etapa 2. Multiplicando os significados:

$$\begin{array}{r} 1,000_{bin} \\ \times \quad 1,110_{bin} \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1110000_{bin} \end{array}$$

O produto é  $1,110000_{bin} \times 2^{-3}$ , mas precisamos mantê-lo com 4 bits, de modo que é  $1,110_{bin} \times 2^{-3}$ .

Etapa 3. Agora, verificamos o produto para ter certeza de que está normalizado e depois verificamos o expoente em busca de overflow ou underflow. O produto já está normalizado e, como  $127 \geq -3 \geq -126$ , não existe overflow ou underflow. (Usando a representação deslocada,  $254 \geq 124 \geq 1$ , de modo que o expoente cabe.)

Etapa 4. O arredondamento do produto não causa mudança:

$$1,110_{bin} \times 2^{-3}$$

Etapa 5. Como os sinais dos operandos originais diferem, torne o sinal do produto negativo. Logo, o produto é

$$-1,110_{bin} \times 2^{-3}$$

Convertendo para decimal, para verificar nossos resultados:

$$\begin{aligned} -1,110_{bin} \times 2^{-3} &= -0,001110_{bin} = -0,00111_{bin} \\ &= -7/2^5_{dec} = -7/32_{dec} = -0,21875_{dec} \end{aligned}$$

O produto entre  $0,5_{dec}$  e  $-0,4375_{dec}$  é, na realidade,  $-0,21875_{dec}$ .

**Instruções de ponto flutuante no MIPS**

O MIPS admite os formatos de precisão simples e dupla do padrão IEEE 754 com estas instruções:

- *Adição simples* em ponto flutuante (`add.s`) e *adição dupla* (`add.d`)
- *Subtração simples* em ponto flutuante (`sub.s`) e *subtração dupla* (`sub.d`)

- *Multiplicação simples* em ponto flutuante (`mul.s`) e *multiplicação dupla* (`mul.d`)
- *Divisão simples* em ponto flutuante (`div.s`) e *divisão dupla* (`div.d`)
- *Comparação simples* em ponto flutuante (`c.x.s`) e *comparação dupla* (`c.x.d`), em que  $x$  pode ser *igual* (`eq`), *diferente* (`neq`), *menor que* (`lt`), *menor ou igual* (`le`), *maior que* (`gt`) ou *maior ou igual* (`ge`)
- *Desvio verdadeiro* em ponto flutuante (`be1t`) e *desvio falso* (`bc1f`)

A comparação em ponto flutuante define um bit como verdadeiro ou falso, dependendo da condição de comparação, e um desvio de ponto flutuante então decide se desviará ou não, dependendo da condição.

Os projetistas do MIPS decidiram acrescentar registradores de ponto flutuante separados – chamados `$f0`, `$f1`, `$f2...` – usados para precisão simples ou precisão dupla. Logo, eles incluíram loads e stores separados para registradores de ponto flutuante: `lwcl` e `swcl`. Os registradores base para transferências de dados de ponto flutuante continuam sendo registradores inteiros. O código do MIPS para carregar dois números de precisão simples da memória, somá-los e depois armazenar a soma poderia se parecer com isto:

```

lwcl      $f4,x($sp)  # Lê número P.F. 32 bits em F4
lwcl      $f6,y($sp)  # Lê número P.F. 32 bits em F6
add.s    $f2,$f4,$f6   # F2 = F4 + F6 precisão simples
swcl      $f2,z($sp)  # Armazena número P.F. 32 bits de F2
    
```

Um registrador de precisão dupla é, na realidade, um par de registradores (par e ímpar) de precisão simples, usando o número do registrador par como seu nome. Assim, o par de registradores `$f2` e `$f3` também forma o registrador de precisão dupla chamado `$f2`.

A [Figura 3.18](#) resume a parte de ponto flutuante da arquitetura MIPS revelada neste capítulo, com as adições para dar suporte ao ponto flutuante mostradas em destaque. Semelhante à Figura 2.19 no Capítulo 2, mostramos a codificação dessas instruções na [Figura 3.19](#).

Uma questão que os projetistas de computador enfrentam no suporte à aritmética de ponto flutuante é se devem utilizar os mesmos registradores usados pelas instruções com inteiros ou acrescentar um conjunto especial de ponto flutuante. Como os programas normalmente realizam operações com inteiros e operações com ponto flutuante sobre dados diferentes, a separação dos registradores só aumentará ligeiramente o número de instruções necessárias para executar um programa. O maior impacto é criar um conjunto separado de instruções de transferência de dados para mover dados entre os registradores de ponto flutuante e a memória.

Os benefícios dos registradores de ponto flutuante separados são a existência do dobro dos registradores sem utilizar mais bits no formato da instrução, ter o dobro da largura de banda de registrador, com conjuntos de registradores separados para inteiros e números de ponto flutuante, e ser capaz de personalizar registradores para ponto flutuante; por exemplo, alguns computadores convertem todos os operandos dimensionados nos registradores para um único formato inteiro.

## Interface hardware/software

**Operandos de ponto flutuante do MIPS**

Nome	Exemplo	Comentários
32 registradores de ponto flutuante	\$f0, \$f1, \$f2, . . . , \$f31	Registradores MIPS de ponto flutuante são usados em pares para números de precisão dupla.
$2^{30}$ words em memória	Memória[0], Memória[4], . . . , Memória[4294967292]	Acessado apenas por instruções de transferência de dados. O MIPS usa endereços em bytes, de modo que os endereços sequenciais em palavras diferem em 4 vezes. A memória mantém estruturas de dados, como arrays, e spilled registers, como aqueles salvos em chamadas de procedimento.

**Linguagem assembly de ponto flutuante do MIPS**

Categoria	Instrução	Exemplo	Significado	Comentários
Aritmética	FP add single	add.s \$f2,\$f4,\$f6	$\$f2 = \$f4 + \$f6$	adição PF (precisão simples)
	FP subtract single	sub.s \$f2,\$f4,\$f6	$\$f2 = \$f4 - \$f6$	subtração PF (precisão simples)
	FP multiply single	mul.s \$f2,\$f4,\$f6	$\$f2 = \$f4 \times \$f6$	multiplicação PF (precisão simples)
	FP divide single	div.s \$f2,\$f4,\$f6	$\$f2 = \$f4 / \$f6$	divisão PF (precisão simples)
	FP add double	add.d \$f2,\$f4,\$f6	$\$f2 = \$f4 + \$f6$	adição PF (precisão dupla)
	FP subtract double	sub.d \$f2,\$f4,\$f6	$\$f2 = \$f4 - \$f6$	subtração PF (precisão dupla)
	FP multiply double	mul.d \$f2,\$f4,\$f6	$\$f2 = \$f4 \times \$f6$	multiplicação PF (precisão dupla)
	FP divide double	div.d \$f2,\$f4,\$f6	$\$f2 = \$f4 / \$f6$	divisão PF (precisão dupla)
Transferência de dados	load word copr. 1	lwcl \$f1,100(\$s2)	$\$f1 = \text{Memória}[\$s2 + 100]$	dados de 32 bits para um registrador FP
	store word copr. 1	swcl \$f1,100(\$s2)	$\text{Memória}[\$s2 + 100] = \$f1$	dados de 32 bits para memória
Desvio condicional	branch on FP true	bclt 25	if (cond == 1) go to PC + 4 + 100	desvio relativo ao PC se PF cond.
	branch on FP false	bclf 25	if (cond == 0) go to PC + 4 + 100	desvio relativo ao PC se não cond.
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s \$f2,\$f4	if ( $\$f2 < \$f4$ ) cond = 1; else cond = 0	comparação PF menor que, precisão simples
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d \$f2,\$f4	if ( $\$f2 < \$f4$ ) cond = 1; else cond = 0	comparação PF menor que, precisão dupla

**Linguagem de máquina de ponto flutuante do MIPS**

Nome	Formato	Exemplo							Comentários
add.s	R	17	16	6	4	2	0	add.s \$f2,\$f4,\$f6	
sub.s	R	17	16	6	4	2	1	sub.s \$f2,\$f4,\$f6	
mul.s	R	17	16	6	4	2	2	mul.s \$f2,\$f4,\$f6	
div.s	R	17	16	6	4	2	3	div.s \$f2,\$f4,\$f6	
add.d	R	17	17	6	4	2	0	add.d \$f2,\$f4,\$f6	
sub.d	R	17	17	6	4	2	1	sub.d \$f2,\$f4,\$f6	
mul.d	R	17	17	6	4	2	2	mul.d \$f2,\$f4,\$f6	
div.d	R	17	17	6	4	2	3	div.d \$f2,\$f4,\$f6	
lwcl	I	49	20	2	100			lwcl \$f2,100(\$s4)	
swcl	I	57	20	2	100			swcl \$f2,100(\$s4)	
bclt	I	17	8	1	25			bclt 25	
bclf	I	17	8	0	25			bclf 25	
c.lt.s	R	17	16	4	2	0	60	c.lt.s \$f2,\$f4	
c.lt.d	R	17	17	4	2	0	60	c.lt.d \$f2,\$f4	
Tamanho do campo		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Todas as instruções MIPS de 32 bits	

**FIGURA 3.18 Arquitetura de ponto flutuante do MIPS revelada até aqui.** Ver Apêndice B, Seção B.10, para obter mais detalhes. Essa informação também é encontrada na coluna 2 do Guia de Instrução Rápida do MIPS, no início deste livro.

op(31:26):								
28-26	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
31-29								
0(000)	Rfmt	<u>Bltz/gez</u>	j	jal	beq	bne	blez	bgtz
1(001)	addi	addiu	slti	sltiu	andi	ori	xori	lui
2(010)	TLB	<u>FIPt</u>						
3(011)								
4(100)	lb	lh	lw	lw	lbu	lhu	lwr	
5(101)	sb	sh	sw	sw			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26) = 010001 (FIPt), (rt(16:16) = 0 => c = f, rt(16:16) = 1 => c = t), rs(25:21):								
23-21	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
25-24								
0(00)	mfc1		cfcl		mtcl		ctcl	
1(01)	bc1.c							
2(10)	<i>f</i> = single	<i>f</i> = double						
3(11)								

op(31:26) = 010001 (FIPt), (f above: 10000 => f = s, 10001 => f = d), funct(5:0):								
2-0	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
5-3								
0(000)	add. <i>f</i>	sub. <i>f</i>	mul. <i>f</i>	div. <i>f</i>		abs. <i>f</i>	mov. <i>f</i>	neg. <i>f</i>
1(001)								
2(010)								
3(011)								
4(100)	cvt.s. <i>f</i>	cvt.d. <i>f</i>			cvt.w. <i>f</i>			
5(101)								
6(110)	c.f. <i>f</i>	c.un. <i>f</i>	c.eq. <i>f</i>	c.ueq. <i>f</i>	c.olt. <i>f</i>	c.ult. <i>f</i>	c.ole. <i>f</i>	c.ule. <i>f</i>
7(111)	c.sf. <i>f</i>	c.ngle. <i>f</i>	c.seq. <i>f</i>	c.ngl. <i>f</i>	c.lt. <i>f</i>	c.nge. <i>f</i>	c.le. <i>f</i>	c.ngt. <i>f</i>

**FIGURA 3.19 Codificação de instruções de ponto flutuante do MIPS.** Essa notação indica o valor de um campo por linha e por coluna. Por exemplo, na parte superior da figura, *lw* se encontra na linha número 4 (100<sub>bin</sub> para os bits de 31-29 da instrução) e na coluna número 3 (011<sub>bin</sub> para os bits 28-26 da instrução), de modo que o valor correspondente do campo op (bits 31-26) é 100011<sub>bin</sub>. O sublinhado indica que o campo é usado em outro lugar. Por exemplo, FIPt na linha 2 e coluna 1 (op = 010001<sub>bin</sub>) está definido na parte inferior da figura. Logo, sub.f na linha 0 e coluna 1 da seção inferior significa que o campo funct (bits 5-0 da instrução) é 000001<sub>bin</sub> e o campo op (bits 31-26) é 010001<sub>bin</sub>. Observe que o campo rs de 5 bits, especificado na parte do meio da figura, determina se a operação é de precisão simples (*f* = *s*, de modo que rs = 10000) ou precisão dupla (*f* = *d*, de modo que rs = 10001). De modo semelhante, o bit 16 da instrução determina se a instrução bc1.c testa o estado verdadeiro (bit 16 = 1 => bc1.t) ou falso (bit 16 = 1 => bc1.f). As instruções em negrito são descritas nos Capítulos 2 neste capítulo, com o Apêndice B abordando todas as instruções. Essa informação também é encontrada na coluna 2 do Guia de Instrução Rápida do MIPS, no início deste livro.

**EXEMPLO****Compilando um programa C de ponto flutuante em código assembly do MIPS**

Vamos converter uma temperatura em Fahrenheit para Celsius:

```
float f2c (float fahr)
{
    return ((5.0/9.0) * (fahr - 32.0));
}
```

Considere que o argumento de ponto flutuante `fahr` seja passado em `$f12` e o resultado deva ficar em `$f0`. (Ao contrário dos registradores inteiros, o registrador de ponto flutuante 0 pode conter um número.) Qual é o código assembly do MIPS?

**RESPOSTA**

Consideramos que o compilador coloca as três constantes de ponto flutuante na memória para serem alcançadas facilmente por meio do ponteiro global `$gp`. As duas primeiras instruções carregam as constantes 5.0 e 9.0 nos registradores de ponto flutuante:

`f2c:`

```
lwcl $f16,const5($gp) # $f16 = 5.0 (5.0 in memoria)
lwcl $f18,const9($gp) # $f18 = 9.0 (9.0 in memoria)
```

Depois, elas são divididas para que se obtenha a fração 5.0/9.0:

```
div.s $f16, $f16, $f18 # $f16 = 5.0 / 9.0
```

(Muitos compiladores dividiram 5.0 por 9.0 durante a compilação e guardariam uma única constante 5.0/9.0 na memória, evitando, assim, a divisão em tempo de execução.) Em seguida, carregamos a constante 32.0 e depois a subtraímos de `fahr` (`$f12`):

```
lwcl $f18, const32($gp) # $f18 = 32.0
sub.s $f18, $f12, $f18 # $f18 = fahr - 32.0
```

Finalmente, multiplicamos os dois resultados intermediários, colocando o produto em `$f0` como resultado de retorno, e depois retornamos:

```
mul.s $f0, $f16, $f18 # $f0 = (5/9)*(fahr - 32.0)
jr $ra # return
```

Agora, vamos realizar operações de ponto flutuante em matrizes, código comumente encontrado em programas científicos.

**EXEMPLO****Compilando um procedimento em C de ponto flutuante com matrizes bidimensionais no MIPS**

A maioria dos cálculos de ponto flutuante é realizada com precisão dupla. Vamos realizar uma multiplicação de matrizes  $X = X + Y * Z$ . Vamos supor que  $X$ ,  $Y$  e  $Z$  sejam matrizes quadradas com 32 elementos em cada dimensão.

```

void mm (double x[][], double y[][], double z[][])
{
    int i, j, k;

    for (i = 0; i != 32; i = i + 1)
        for (j = 0; j != 32; j = j + 1)
            for (k = 0; k != 32; k = k + 1)
                x[i][j] = x[i][j] + y[i][k] * z[k][j];
}

```

Os endereços iniciais do array são parâmetros, de modo que estão em \$a0, \$a1 e \$a2. Suponha que as variáveis inteiras estejam em \$s0, \$s1 e \$s2, respectivamente. Qual é o código assembly do MIPS para o corpo do procedimento?

Observe que  $x[i][j]$  é usado no loop mais interno. Como o índice do loop é  $k$ , o índice não afeta  $x[i][j]$ , de modo que podemos evitar a leitura e o armazenamento de  $x[i][j]$  a cada iteração. Em vez disso, o compilador lê  $x[i][j]$  em um registrador fora do loop, acumula a soma dos produtos de  $y[i][k]$  e  $z[k][j]$  nesse mesmo registrador, e depois armazena a soma em  $x[i][j]$ , ao terminar o loop mais interno.

Mantemos o código mais simples, usando as pseudoinstruções em assembly `li` (que carrega uma constante em um registrador), e `l.d e s.d` (que o montador transforma em um par de instruções de transferência de dados, `lwcl` ou `swcl`, para um par de registradores de ponto flutuante).

O corpo do procedimento começa salvando o valor de término do loop (32) em um registrador temporário e depois inicializando as três variáveis do loop `for`:

```

mm:...
    li      $t1, 32  # $t1 = 32 (tamanho de linha/fim do loop)
    li      $s0, 0   # i = 0; inicializa 1º loop for
L1:    li      $s1, 0   # j = 0; reinicia 2º loop for
L2:    li      $s2, 0   # k = 0; reinicia 3º loop for

```

Para calcular o endereço de  $x[i][j]$  precisamos saber como um array bidimensional de  $32 \times 32$  é armazenado na memória. Como você poderia esperar, seu layout é como se houvesse 32 arrays unidimensionais, cada um com 32 elementos. Assim, a primeira etapa é pular os  $i$  “arrays unidimensionais”, ou linhas, para obter a que desejamos. Assim, multiplicamos o índice da primeira dimensão pelo tamanho da linha, 32. Como 32 é uma potência de 2, podemos usar um deslocamento em seu lugar:

```
sll  $t2, $s0, 5          # $t2 = i * 25 (tamanho de linha (d x))
```

Agora, acrescentamos o segundo índice para selecionar o elemento  $j$  da linha desejada:

```
addu $t2, $t2, $s1      # $t2 = i * tamanho(linha) + j
```

Para transformar essa soma em um índice em bytes, multiplicamos pelo tamanho de um elemento da matriz em bytes. Como cada elemento tem 8 bytes para a precisão dupla, podemos deslocar à esquerda por 3:

```
sll  $t2, $t2, 3          # $t2 = deslocamento em bytes de [i][j]
```

Em seguida, somamos essa soma ao endereço base de  $x$ , dando o endereço de  $x[i][j]$ , e depois carregamos o número de precisão dupla  $x[i][j]$  em \$f4:

## RESPOSTA

```

addu $t2, $a0, $t2      # $t2 = endereço em bytes de x[i][j]
l.d $f4, 0($t2)        # $f4 = 8 bytes de x[i][j]

```

As cinco instruções a seguir são praticamente idênticas às cinco últimas: calcular o endereço e depois ler o número de precisão dupla  $z[k][j]$ .

```

L3: sll $t0, $s2, 5      # $t0 =  $k * 2^5$  (tamanho de linha de z)
    addu $t0, $t0, $s1 # $t0 =  $k * \text{tamanho(linha)} + j$ 
    sll $t0, $t0, 3      # $t0 = deslocamento em bytes de [k][j]
    addu $t0, $a2, $t0 # $t0 = endereço em bytes de z[k][j]
    l.d $f16, 0($t0)   # $f16 = 8 bytes de z[k][j]

```

De modo semelhante, as cinco instruções a seguir são como as cinco últimas: calcular o endereço e depois carregar o número de precisão dupla  $y[i][k]$ .

```

sll      $t0, $s0, 5      # $t0 =  $i * 2^5$  (tamanho de linha de y)
addu    $t0, $t0, $s2 # $t0 =  $i * \text{tamanho(linha)} + k$ 
sll      $t0, $t0, 3      # $t0 = deslocamento em bytes de [i][k]
addu    $t0, $a1, $t0 # $t0 = endereço em bytes de y[i][k]
l.d     $f18, 0($t0)   # $f18 = 8 bytes de y[i][k]

```

Agora que carregamos todos os dados, finalmente estamos prontos para realizar algumas operações em ponto flutuante! Multiplicamos os elementos de  $y$  e  $z$  localizados nos registradores  $$f18$  e  $$f16$ , e depois acumulamos a soma em  $$f4$ .

```

mul.d $f16, $f18, $f16 # $f16 =  $y[i][k] * z[k][j]$ 
add.d $f4, $f4, $f16    # f4 =  $x[i][j] + y[i][k] * z[k][j]$ 

```

O bloco final incrementa o índice  $k$  e retorna se o índice não for 32. Se for 32, ou seja, o final do loop mais interno, precisamos armazenar em  $x[i][j]$  a soma acumulada em  $$f4$ .

```

addiu $s2, $s2, 1      # $k = k + 1
bne   $s2, $t1, L3     # se ( $k \neq 32$ ) vai para L3
s.d   $f4, 0($t2)      #  $x[i][j] = f4$ 

```

De modo semelhante, essas quatro instruções finais incrementam a variável de índice do loop do meio e do loop mais externo, voltando no loop se o índice não for 32 e saindo se o índice for 32.

```

addiu $s1, $s1, 1      # $j = j + 1
bne   $s1, $t1, L2     # se ( $j \neq 32$ ) vai para L2
addiu $s0, $s0, 1      # $i = i + 1
bne   $s0, $t1, L1     # se ( $i \neq 32$ ) vai para L1
...

```

**Detalhamento:** o layout do array discutido no exemplo, chamado *ordem linhas primeiro*, é usado pela linguagem C e muitas outras linguagens de programação. Fortran, por sua vez, usa a *ordem colunas primeiro*, pela qual o array é armazenado coluna por coluna.

**Detalhamento:** Somente 16 dos 32 registradores de ponto flutuante do MIPS puderam ser usados originalmente para operações de precisão simples:  $$f0, $f2, $f4, \dots, $f30$ . A precisão dupla é calculada usando pares desses registradores. Os registradores de ponto flutuante com números ímpares só foram usados para carregar e armazenar a metade direita

dos números de ponto flutuante de 64 bits. MIPS-32 acrescentou l.d e s.d ao conjunto de instruções. MIPS-32 também acrescentou versões “simples emparelhadas” de todas as instruções de ponto flutuante, em que uma única instrução resulta em duas operações paralelas de ponto flutuante sobre dois operandos de 32 bits dentro de registradores de 64 bits. Por exemplo, add.ps F0, F2, F4 é equivalente a add.s F0, F2, F4, seguido por add.ps F1, F3, F5

**Detalhamento:** Outro motivo para que os registradores inteiros e de ponto flutuante sejam separados é que os microprocessadores na década de 1980 não possuíam transistores suficientes para colocar a unidade ponto flutuante no mesmo chip da unidade de inteiros. Logo, a unidade de ponto flutuante, incluindo os registradores de ponto flutuante, opcionalmente estava disponível como um segundo chip. Esses chips aceleradores opcionais são chamados *coprocessadores* e explicam o acrônimo para os loads de ponto flutuante no MIPS: `lwc1` significa “load word to coprocessor 1” (“leia uma palavra para o coprocessador 1”), que é a unidade de ponto flutuante. (O coprocessador 0 trata da memória virtual, descrita no Capítulo 5.) Desde o início da década de 1990, os microprocessadores têm integrado o ponto flutuante (e praticamente tudo o mais) no chip, e, por isso, o termo “coprocessador” reúne “acumulador” e “memória”.

**Detalhamento:** Conforme mencionamos na Seção 3.4, acelerar a divisão é mais complicado do que a multiplicação. Além de SRT, outra técnica para aproveitar um multiplicador rápido é a *iteração de Newton*, na qual a divisão é redefinida como a localização do zero de uma função para encontrar a recíproca  $1/x$ , que é multiplicada pelo outro operando. As técnicas de iteração não podem ser arredondadas corretamente sem o cálculo de muitos bits extras. Um chip TI soluciona esse problema, calculando uma recíproca de precisão extra.

**Detalhamento:** Java abrange o padrão IEEE 754 por nome em sua definição dos tipos de dados e operações de ponto flutuante Java. Assim, o código no primeiro exemplo poderia muito bem ter sido gerado para um método de classe que convertesse graus Fahrenheit em Celsius.

O segundo exemplo utiliza múltiplos arrays dimensionais, que não são admitidos explicitamente em Java. Java permite arrays de arrays, mas cada array pode ter seu próprio tamanho, ao contrário de vários arrays dimensionais em C. Como os exemplos no Capítulo 2, uma versão Java desse segundo exemplo exigiria muito código de verificação para os limites de array, incluindo um novo cálculo de tamanho no final da linha. Ela também precisaria verificar se a referência ao objeto não é nula.

## Aritmética de precisão

Ao contrário dos inteiros, que podem representar exatamente cada número entre o menor e o maior, os números de ponto flutuante, em geral, são aproximações para um número que não representam realmente. O motivo é que existe uma variedade infinita de números reais entre, digamos, 0 e 1, porém não mais do que  $2^{53}$  podem ser representados com exatidão em ponto flutuante de precisão dupla. O melhor que podemos fazer é utilizar a representação de ponto flutuante próxima ao número real. Assim, o padrão IEEE 754 oferece vários modos de arredondamento para permitir que o programador selecione a aproximação desejada.

O arredondamento parece muito simples, mas arredondar com precisão exige que o hardware inclua bits extras no cálculo. Nos exemplos anteriores, fomos vagos com relação ao número de bits que uma representação intermediária pode ocupar, mas, claramente, se cada resultado intermediário tivesse de ser truncado ao número de dígitos exato, não haveria oportunidade para arredondar. O IEEE 754, portanto, sempre mantém dois bits extras à direita durante adições intermediárias, chamados **guarda** e **arredondamento**, respectivamente. Vamos fazer um exemplo decimal para ilustrar o valor desses dígitos extras.

**guarda** O primeiro dos dois bits extras mantidos à direita durante os cálculos intermediários de números de ponto flutuante, usados para melhorar a precisão do arredondamento.

**arredondamento** Método para fazer com que o resultado de ponto flutuante intermediário se encaixe no formato de ponto flutuante; o objetivo normalmente é encontrar o número mais próximo que pode ser representado no formato.

**EXEMPLO****RESPOSTA****Arredondando com dígitos de guarda**

Some  $2,56_{\text{dec}} \times 10^0$  a  $2,34_{\text{dec}} \times 10^2$ , supondo que temos três dígitos decimais significativos. Arredonde para o número decimal mais próximo com três dígitos decimais significativos, primeiro com dígitos guarda e arredondamento, e depois sem eles.

Primeiro, temos de deslocar o número menor para a direita, a fim de alinhar os expoentes, de modo que  $2,56_{\text{dec}} \times 10^0$  torna-se  $0,0256_{\text{dec}} \times 10^2$ . Como temos dígitos de guarda e arredondamento, podemos representar os dois dígitos menos significativos quando alinharmos os expoentes. O dígito de guarda mantém 5 e o dígito de arredondamento mantém 6. A soma é

$$\begin{array}{r} 2,3400_{\text{dec}} \\ + 0,0256_{\text{dec}} \\ \hline 2,3656_{\text{dec}} \end{array}$$

Assim, a soma é  $2,3656_{\text{dec}} \times 10^2$ . Como temos dois dígitos para arredondar, queremos que os valores de 0 a 49 arredondem para baixo e de 51 a 99 para cima, com 50 sendo o desempate. Arredondar a soma para cima com três dígitos significativos gera  $2,37_{\text{dec}} \times 10^2$ .

Fazer isso *sem* dígitos de guarda e arredondamento remove dois dígitos do cálculo. A nova soma é, então,

$$\begin{array}{r} 2,34_{\text{dec}} \\ + 0,02_{\text{dec}} \\ \hline 2,36_{\text{dec}} \end{array}$$

A resposta é  $2,36_{\text{dec}} \times 10^2$ , arredondando no último dígito da soma anterior.

**unidades na última casa (ulp)** O número de bits com erro nos bits menos significativos do significando entre o número real e o número que pode ser representado.

Como o pior caso para o arredondamento seria quando o número real está a meio caminho entre duas representações de ponto flutuante, a precisão no ponto flutuante normalmente é medida em termos do número de bits em erro nos bits mais significativos do significando; a medida é denominada número de **unidades na última casa**, ou **ulp** (*units in the last place*). Se o número ficou defasado em 2 nos bits menos significativos, ele estaria defasado por 2 ulps. Desde que não haja qualquer overflow, underflow ou exceções de operação inválida, o IEEE 754 garante que o computador utiliza o número que está dentro de meia ulp.

**Detalhamento:** Embora o exemplo anterior, na realidade, precisasse apenas de um dígito extra, a multiplicação pode precisar de dois. Um produto binário pode ter um bit 0 inicial; logo, a etapa de normalização precisa deslocar o produto 1 bit à esquerda. Isso desloca o dígito de guarda para o bit menos significativo do produto, deixando o bit de arredondamento para ajudar no arredondamento mais preciso do produto.

O IEEE 754 tem quatro modos de arredondamento: sempre arredondar para cima (para  $+\infty$ ), sempre arredondar para baixo (para  $-\infty$ ), truncar e arredondar para o próximo par. O modo final determina o que fazer se o número estiver exatamente no meio. A Receita Federal americana sempre arredonda 0,50 dólares para cima, possivelmente para o benefício da Receita. Um modo mais imparcial seria arredondar para cima, nesse caso, na metade do tempo e arredondar para baixo na outra metade. O IEEE 754 diz que, se o bit menos significativo retido em um caso de meio do caminho for ímpar, some um; se for par, trunque. Esse método sempre cria um 0 no bit menos significativo no caso de desempate, dando nome ao arredondamento. Esse modo é o mais utilizado, e o único que o Java admite.

O objetivo dos bits de arredondamento extras é permitir que o computador obtenha os mesmos resultados, como se os resultados intermediários fossem calculados para precisão

infinita e depois arredondados. Para auxiliar nesse objetivo e arredondar para o par mais próximo, o padrão possui um terceiro bit além do bit de guarda e arredondamento; ele é definido sempre que existem bits diferentes de zero à direita do bit de arredondamento. Esse **sticky bit** permite que o computador veja a diferença entre  $0,50 \dots 00_{dec}$  e  $0,50 \dots 01_{dec}$  ao arredondar.

O sticky bit pode ser definido, por exemplo, durante a adição, quando o menor número é deslocado para a direita. Suponha que somemos  $5,01_{dec} \times 10^1$  a  $2,34_{ten} \times 10^2$  no exemplo anterior. Mesmo com os bits de guarda e arredondamento, estariamos somando 0,0050 a 2,34, com uma soma de 2,3450. O sticky bit seria definido, porque existem bits diferentes de zero à direita. Sem o sticky bit para lembrar se quaisquer 1s foram deslocados, consideraríamos que o número é igual a 2.345000...00 e arredondaríamos para o par mais próximo de 2,34. Com o sticky bit para lembrar que o número é maior do que 2,345000...00, arredondaríamos para 2,35.

**sticky bit** Um bit usado no arredondamento além dos bits de guarda e arredondamento, definido sempre que existem bits diferentes de zero à direita do bit de arredondamento.

**Detalhamento:** As arquiteturas PowerPC, SPARC64 e AMD SSE5 oferecem uma única instrução que realiza multiplicação e adição sobre três registradores:  $a = a + (b \times c)$ . Obviamente, essa instrução permite um desempenho de ponto flutuante potencialmente mais alto para essa operação comum. Igualmente importante é que, em vez de realizar dois arredondamentos — depois da multiplicação e após a adição — que aconteceria com instruções separadas, a instrução de multiplicação adição pode realizar um único arredondamento após a adição, o que aumenta a precisão da multiplicação adição. Essas operações com um único arredondamento são chamadas **multiplicação adição fundida**. Isso foi acrescentado no IEEE 754 revisado (veja  [Seção 3.10](#) no site).

## Resumo

A próxima seção “Colocando em perspectiva” reforça o conceito de programa armazenado do Capítulo 2; o significado da informação não pode ser determinado simplesmente examinando-se os bits, pois os mesmos bits podem representar uma série de objetos. Esta seção mostra que a aritmética computacional é finita e, assim, pode não combinar com a aritmética natural. Por exemplo, a representação de ponto flutuante do padrão IEEE 754

$$(-1)^S \times (1 + \text{Fração}) \times 2^{(\text{Expoente} - \text{Bias})}$$

é quase sempre uma aproximação do número real. Os sistemas computacionais precisam ter o cuidado de minimizar essa lacuna entre a aritmética computacional e a aritmética no mundo real, e os programadores às vezes precisam estar cientes das implicações dessa aproximação.

Padrões de bits não possuem significado inherente. Eles podem representar inteiros com sinal, inteiros sem sinal, números de ponto flutuante, instruções e assim por diante. O que é representado depende da instrução que opera sobre os bits na palavra.

A principal diferença entre os números no computador e os números no mundo real é que os números no computador possuem tamanho limitado e, por isso, uma precisão limitada; é possível calcular um número muito grande ou muito pequeno para ser representado em uma palavra. Os programadores precisam se lembrar desses limites e escrever programas de acordo.

## Colocando em perspectiva

Tipo C	Tipo Java	Transferências de dados	Operações
int	int	lw, sw, lui	addu, addiu, subu, mult, div, AND, ANDi, OR, ORi, NOR,slt, slti
unsigned int	—	lw, sw, lui	addu, addiu, subu, mult, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu
char	—	lb, sb, lui	add, addi, sub, mult, div, AND, ANDi, OR, ORi, NOR,slt, slti
—	char	lh, sh, lui	addu, addiu, subu, multu, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu
float	float	lwcl, swcl	add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s
double	double	ld, sd	add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d

## Interface hardware/software

No capítulo anterior, apresentamos as classes de armazenamento da linguagem de programação C (veja a seção Interface Hardware/Software da Seção 2.7). A tabela anterior mostra alguns dos tipos de dados C e Java junto com as instruções de transferência de dados MIPS e instruções que operam sobre aqueles tipos que aparecem aqui e no Capítulo 2. Observe que Java omite inteiros sem sinal.

## Verifique você mesmo

Suponha que houvesse um formato de ponto flutuante IEEE 754 de 16 bits com 5 bits de expoente. Qual seria o intervalo provável de números que ele poderia representar?

1.  $1,0000\ 0000\ 00 \times 2^0$  a  $1,1111\ 1111\ 11 \times 2^{31}, 0$
2.  $\pm 1,0000\ 0000\ 0 \times 2^{-14}$  a  $\pm 1.1111\ 1111\ 1 \times 2^{15}, \pm 0, \pm \infty, \text{NaN}$
3.  $\pm 1,0000\ 0000\ 00 \times 2^{-14}$  a  $\pm 1.1111\ 1111\ 11 \times 2^{15}, \pm 0, \pm \infty, \text{NaN}$
4.  $\pm 1,0000\ 0000\ 00 \times 2^{-15}$  a  $\pm 1.1111\ 1111\ 11 \times 2^{14}, \pm 0, \pm \infty, \text{NaN}$

**Detalhamento:** Para acomodar comparações que possam incluir NaNs, o padrão inclui *ordenada* e *desordenada* como opções para comparações. Logo, o conjunto de instruções MIPS inteiro possui muitos tipos de comparações para dar suporte a NaNs. (Java não admite comparações não ordenadas.)

Em uma tentativa de espremer cada bit de precisão de uma operação de ponto flutuante, o padrão permite que alguns números sejam representados em forma não normalizada. Em vez de ter uma lacuna entre 0 e o menor número normalizado, o IEEE permite *números não normalizados* (também conhecidos como *denorms* ou *subnormals*). Eles têm o mesmo expoente que zero, mas um significando diferente de zero. Eles permitem que um número diminua no significado até se tornar 0, chamado *underflow gradual*. Por exemplo, o menor número normalizado positivo de precisão simples é

$$1,0000\ 0000\ 0000\ 0000\ 0000_{\text{bin}} \times 2^{-126}$$

mas o menor número não normalizado de precisão simples é

$$0,0000\ 0000\ 0000\ 0000\ 0001_{\text{bin}} \times 2^{-126}, \text{ou } 1,0_{\text{bin}} \times 2^{-149}$$

Para a precisão dupla, a lacuna denorm vai de  $1,0 \times 2^{-1022}$  a  $1,0 \times 2^{-1074}$ .

A possibilidade de um operando ocasional não normalizado tem dado dores de cabeça aos projetistas de ponto flutuante que estejam tentando criar unidades de ponto flutuante velozes. Logo, muitos computadores causam uma exceção se um operando for não normalizado, permitindo que o software complete a operação. Embora as implementações de software sejam perfeitamente válidas, seu menor desempenho diminuiu a popularidade dos denorms no software de ponto flutuante portável. Além disso, se os programadores não esperarem os denorms, seus programas poderão ser surpreendidos.

## 3.6

# Paralelismo e aritmética computacional: associatividade

Os programas normalmente têm sido escritos primeiro para executarem sequencialmente antes de simultaneamente, de modo que uma pergunta natural é “as duas versões geram a mesma resposta?”. Se a resposta for não, você pode considerar que existe um defeito na versão paralela, que precisa ser localizado.

Essa técnica considera que a aritmética do computador não afeta os resultados quando passa de sequencial para paralelo. Ou seja, se você tivesse de somar um milhão de números, obteria os mesmos resultados usando 1 processador ou 1.000 processadores. Essa suposição continua para inteiros no complemento de dois, mesmo que o cálculo estoure. Outro modo de dizer isso é que a adição de inteiros é associativa.

Infelizmente, como os números de ponto flutuante são aproximações dos números reais, e como a aritmética computacional tem precisão limitada, isso não é verdade para os números de ponto flutuante. Ou seja, a adição de ponto flutuante não é associativa.

### Testando a associatividade da adição de ponto flutuante

Veja se  $x + (y + z) = (x + y) + z$ . Por exemplo, suponha que  $x = -1,5_{\text{dec}} \times 10^{38}$ ,  $y = 1,5_{\text{dec}} \times 10^{38}$ , e  $z = 1,0$ , e que todos estes sejam números de precisão simples.

### EXEMPLO

Dada a grande faixa de números que podem ser representados em ponto flutuante, ocorrem problemas quando se somam dois números grandes de sinais opostos, mais um número pequeno, conforme veremos:

$$\begin{aligned}x + (y + z) &= -1,5_{\text{dec}} \times 10^{38} + (1,5_{\text{dec}} \times 10^{38} + 1,0) \\&= -1,5_{\text{dec}} \times 10^{38} + (1,5_{\text{dec}} \times 10^{38}) = 0,0 \\(x + y) + z &= (-1,5_{\text{dec}} \times 10^{38} + 1,5_{\text{dec}} \times 10^{38}) + 1,0 \\&= (0,0_{\text{dec}}) + 1,0 \\&= 1,0\end{aligned}$$

### RESPOSTA

Portanto,  $x + (y + z) \neq (x + y) + z$ , de modo que a adição de ponto flutuante não é associativa.

Como os números de ponto flutuante possuem precisão limitada e resultam em aproximações dos resultados reais,  $1,5_{\text{dec}} \times 10^{38}$  é tão maior que  $1,0_{\text{dec}}$  que  $1,5_{\text{dec}} \times 10^{38} + 1,0$  ainda é  $1,5_{\text{dec}} \times 10^{38}$ . É por isso que a soma de  $x$ ,  $y$  e  $z$  é 0,0 ou 1,0, dependendo da ordem das adições de ponto flutuante, e, portanto, a adição de ponto flutuante *não* é associativa.

Uma versão mais irritante dessa armadilha ocorre em um computador paralelo, em que o escalonador do sistema operacional pode usar um número diferente de processadores, dependendo do que outros programas estão executando em um computador paralelo. O programador paralelo desavisado pode se confundir com seu programa obtendo respostas ligeiramente diferentes toda vez que for executada exatamente com o mesmo código e entrada idêntica, pois o número variável de processadores em cada execução faria com que as somas de ponto flutuante fossem calculadas em diferentes ordens.

Por causa desse dilema, os programadores que escrevem código paralelo com números de ponto flutuante precisam verificar se os resultados são confiáveis, mesmo que não deem exatamente a mesma resposta que o código sequencial. O campo que lida com essas questões é a análise numérica, abordada em diversos livros-texto voltados para esse assunto. Esses problemas são um motivo para a popularidade das bibliotecas numéricas, como LAPACK e SCALAPACK, que foram validadas em suas formas sequencial e paralela.

**Detalhamento:** Uma versão sutil do problema de associatividade ocorre quando dois processadores realizam um cálculo redundante que é executado em ordem diferente, de modo que eles recebem respostas ligeiramente diferentes, embora as duas respostas sejam consideradas precisas. O problema ocorre se um desvio condicional compara com um número de ponto flutuante e os dois processadores seguem caminhos diferentes quando o bom senso sugere que eles deveriam seguir o mesmo caminho.

### 3.7

### Vida real: ponto flutuante no x86

A arquitetura x86 possui instruções regulares de multiplicação e divisão que operam inteiramente sobre os registradores normais, em vez de contar com Hi e Lo separados como no MIPS. (Na verdade, as versões posteriores do conjunto de instruções MIPS incluíram instruções semelhantes.)

As diferenças principais são encontradas nas instruções de ponto flutuante. A arquitetura de ponto flutuante x86 é diferente de todos os outros computadores no mundo.

#### A arquitetura de ponto flutuante do x86

O coprocessador de ponto flutuante Intel 8087 foi anunciado em 1980. Essa arquitetura estendeu o 8086 com cerca de 60 instruções de ponto flutuante.

A Intel proveu uma arquitetura de pilha com suas instruções de ponto flutuante: *loads* inserem números na pilha, *operações* encontram operandos nos dois elementos do topo da pilha e *stores* podem retirar elementos da pilha. A Intel complementou essa arquitetura de pilha com instruções e modos de endereçamento que permitem que a arquitetura tenha alguns dos benefícios do modelo registrador-memória. Além de localizar operandos nos dois elementos do topo da pilha, um operando pode estar na memória ou em um dos sete registradores do chip, abaixo do topo da pilha. Assim, um conjunto completo de instruções de pilha é complementado por um conjunto limitado de instruções registrador-memória.

Essa mistura é ainda um modelo registrador-memória restrito, pois os loads sempre movem dados para o topo da pilha enquanto incrementam o ponteiro do topo da pilha, e os stores só podem mover do topo da pilha para a memória. A Intel usa a notação ST para indicar o topo da pilha, e ST(i) para representar o i-ésimo registrador abaixo do topo da pilha.

Outro novo recurso dessa arquitetura é que os operandos são mais largos na pilha de registradores do que são armazenados na memória e todas as operações são realizadas nessa precisão interna larga. Ao contrário do máximo de 64 bits no MIPS, os operandos de ponto flutuante x86 na pilha possuem 80 bits de largura. Os números são convertidos automaticamente para o formato interno de 80 bits em um load e convertidos de volta para o tamanho apropriado em um store. Essa *precisão dupla estendida* não é aceita pelas linguagens de programação, embora tenha sido útil aos programadores de software matemático.

Os dados da memória podem ser números de ponto flutuante de 32 bits (precisão simples) ou de 64 bits (precisão dupla). Antes de realizar a operação, a versão registrador-memória dessas instruções converterá o operando da memória para esse formato de 80 bits da Intel. As instruções de transferência de dados também converterão automaticamente inteiros de 16 e 32 bits para ponto flutuante, e vice-versa, para loads e stores de inteiros.

As operações de ponto flutuante x86 podem ser divididas em quatro classes principais:

1. Instruções de movimentação de dados, incluindo load, load de constante e store
2. Instruções aritméticas, incluindo adição, subtração, multiplicação, divisão, raiz quadrada e módulo absoluto
3. Comparação, incluindo instruções para enviar o resultado ao processador de inteiros de modo que possa se desviar
4. Instruções transcendentais, incluindo seno, cosseno, logaritmo e exponenciação

A Figura 3.20 mostra algumas das 60 operações de ponto flutuante. Observe que obtemos ainda mais combinações quando incluímos os modos de operando para essas operações. A Figura 3.21 mostra as muitas opções para a adição de ponto flutuante.

As instruções de ponto flutuante são codificadas por meio do opcode ESC do 8086 e o especificador de endereço pós-byte (veja Figura 2.47). As operações de memória reservam 2 bits para decidir se o operando é um ponto flutuante de 32 ou de 64 bits, ou um inteiro de 16 ou 32 bits. Esses mesmos 2 bits são usados em versões que não acessam a memória para decidir se o topo da pilha deve ser removido após a operação e se o topo da pilha ou um registrador inferior deve obter o resultado.

O desempenho de ponto flutuante da família x86 tradicionalmente tem ficado atrás de outros computadores. Como resultado, a Intel criou uma arquitetura de ponto flutuante mais tradicional como parte do SSE2.

## A arquitetura de ponto flutuante Streaming SIMD Extension 2 (SSE2) da Intel

O Capítulo 2 observa que, em 2001, a Intel acrescentou 144 instruções à sua arquitetura, incluindo registradores e operações de ponto flutuante com precisão dupla. Isso inclui oito registradores de 64 bits que podem ser usados como operandos de ponto flutuante, dando ao compilador um alvo diferente para as operações de ponto flutuante do que a arquitetura de pilha exclusiva. Os compiladores podem decidir usar os oito registradores SSE2 como

Transferência de dados	Aritmética	Comparação	Transcendentais
F{I}LD mem/ST(i)	F{I}ADD{P} mem/ST(i)	F{I}COM{P}	FPATAN
F{I}ST{P} mem/ST(i)	F{I}SUB{R}{P} mem/ST(i)	F{I}UCOM{P}{P}	F2XM1
FLDPI	F{I}MUL{R}{P} mem/ST(i)	FSTSW AX/mem	FCOS
FLD1F	F{I}SUB{R}{P} mem/ST(i)		FPTAN
FLDZ	FSQRT		FPREM
	FABS		FSIN
	FRNDINT		FYL2X

**FIGURA 3.20 As instruções de ponto flutuante do x86.** Usamos as chaves {} para mostrar variações opcionais das operações básicas: {I} significa que existe uma versão inteira da instrução; {P} significa que essa variação retirará um operando da pilha após a operação; e {R} significa o reverso da ordem dos operandos nessa operação. A primeira coluna mostra as instruções de transferência de dados, que movem dados para a memória ou para um dos registradores abaixo do topo da pilha. As três últimas operações na primeira coluna colocam constantes na pilha: pi, 1,0 e 0,0. A segunda coluna contém as operações aritméticas descritas anteriormente. Observe que as três últimas operam apenas no topo da pilha. A terceira coluna contém as instruções de comparação. Como não existem instruções de desvio de ponto flutuante especiais, o resultado da comparação precisa ser transformado para a CPU de inteiros via instruções FSTSW, seja para o registrador AX ou para a memória, seguida por uma instrução SAHF a fim de definir os códigos de condição. A comparação de ponto flutuante pode, então, ser testada por meio de instruções de desvio inteiros. A última coluna oferece as operações de ponto flutuante de mais alto nível. Nem todas as combinações sugeridas pela notação são fornecidas. Logo, operações F{I}SUB{R}{P} representam estas instruções encontradas no x86: FSUB, FISUB, FSUBR, FI SUBR, FSUBP, FISUBRP. Para as instruções de subtração de inteiros, não existe um pop (FI SUBP) ou um pop reverso (FISUBRP).

Instrução	Operandos	Comentário
FADD		Ambos os operandos na pilha; resultado substitui o topo da pilha.
FADD	ST(i)	Um operando de origem é o registrador na posição i abaixo do topo da pilha; resultado substitui o topo da pilha.
FADD	ST(i), ST	Um operando de origem é o topo da pilha; resultado substitui registrador na posição i abaixo do topo da pilha.
FADD	mem32	Um operando de origem é um local de 32 bits na memória; resultado substitui o topo da pilha.
FADD	mem64	Um operando de origem é um local de 64 bits na memória; resultado substitui o topo da pilha.

**FIGURA 3.21 As variações dos operandos para adição de ponto flutuante na arquitetura x86.**

Transferência de dados	Aritmética	Comparação
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD{SS/PS/SD/PD} xmm, mem/xmm SUB{SS/PS/SD/PD} xmm, mem/xmm	CMP{SS/PS/SD/PD}
MOV{H/L}{PS/PD} xmm, mem/xmm	MUL{SS/PS/SD/PD} xmm, mem/xmm DIV{SS/PS/SD/PD} xmm, mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX{SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD} mem/xmm	

**FIGURA 3.22 As instruções de ponto flutuante SSE/SSE2 do x86.** xmm significa que um operando é um registrador SSE2 de 128 bits, e mem/xmm significa que o outro operando está na memória ou é um registrador SSE2. Usamos as chaves {} para mostrar variações opcionais das operações básicas: {SS} significa ponto flutuante de precisão Scalar Single, ou quatro operandos de 32 bits em um registrador de 128 bits; {SD} significa ponto flutuante de precisão Scalar Double, ou um operando de 64 bits em um registrador de 128 bits; {PD} significa ponto flutuante de precisão Packed Double, ou dois operandos de 64 bits em um registrador de 128 bits; {A} significa que o operando de 128 bits é alinhado na memória; {U} significa que o operando de 128 bits é desalinhado na memória; {H} significa mover a metade alta (high) do operando de 128 bits; e {L} significa mover a metade baixa (low) do operando de 128 bits.

registradores de ponto flutuante, como aqueles encontrados em outros computadores. A AMD expandiu o número para 16, como parte do AMD64, que a Intel passou a chamar de EM64T para seu uso. A Figura 3.22 resume as instruções SSE e SSE2.

Além de manter um número de precisão simples ou de precisão dupla em um registrador, a Intel permite que vários operandos de ponto flutuante sejam encaixados em um único registrador SSE2 de 128 bits: quatro de precisão simples e dois de precisão dupla. Se os operandos podem ser organizados na memória como dados alinhados em 128 bits, então as transferências de dados de 128 bits podem carregar e armazenar vários operandos por instrução. Esse formato de ponto flutuante compactado é aceito por operações aritméticas que podem operar simultaneamente sobre quatro números de precisão simples (PS) ou dois de precisão dupla (PD). Essa arquitetura pode mais do que dobrar o desempenho em relação à arquitetura de pilha.

Assim, a matemática pode ser definida como o assunto em que nunca sabemos do que estamos falando, nem se o que estamos dizendo é verdadeiro.

Bertrand Russell, Recent Words on the Principles of Mathematics, 1901

## 3.8

## Falácia e armadilhas

As falácias e armadilhas aritméticas geralmente advêm da diferença entre a precisão limitada da aritmética computacional e da precisão ilimitada da aritmética natural.

*Falácia: assim como a instrução de deslocamento à esquerda pode substituir uma multiplicação de inteiros por uma potência de 2, um deslocamento à direita é o mesmo que uma divisão de inteiros por uma potência de 2.*

Lembre-se de que um número binário  $x$ , em que  $x_i$  significa o bit na posição  $i$ , representa o número

$$\dots + (x_3 \times 2^3) + (x_2 \times 2^2) + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Deslocar os bits de  $x$  para a direita de  $n$  bits pareceria ser o mesmo que dividir por  $2^n$ . E isso é verdade para inteiros sem sinal. O problema é com os inteiros com sinal. Por exemplo, suponha que queremos dividir  $-5_{\text{dec}}$  por  $4_{\text{dec}}$ ; o quociente seria  $-1_{\text{dec}}$ . A representação no complemento de dois para  $-5_{\text{dec}}$  é

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_{\text{bin}}$$

De acordo com essa falácia, deslocar para a direita por dois deverá dividir por  $4_{\text{dec}}$  ( $2^2$ ):

$$0011\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{bin}}$$

Com um 0 no bit de sinal, esse resultado claramente está errado. O valor criado pelo deslocamento à direita é, na realidade,  $1.073.741.822_{dec}$ , e não  $-1_{dec}$ .

Uma solução seria ter um deslocamento aritmético à direita, que estende o bit de sinal, em vez de colocar 0s à esquerda num deslocamento à direita. Um deslocamento aritmético de 2 bits para a direita de  $-5_{dec}$  produz

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

O resultado é  $-2_{dec}$ , em vez de  $-1_{dec}$ ; próximo, mas não podemos comemorar.

*Armadilha: a instrução MIPS add immediate unsigned (addiu) estende o sinal de seu campo imediato de 16 bits.*

Apesar de seu nome, add immediate unsigned (*addiu*) é usada para somar constantes a inteiros com sinal quando não nos importamos com o overflow. O MIPS não possui uma instrução de subtração imediata e os números negativos precisam de extensão de sinal, de modo que os arquitetos do MIPS decidiram estender o sinal do campo imediato.

*Falácia: somente os matemáticos teóricos se importam com a precisão do ponto flutuante.*

As manchetes dos jornais de novembro de 1994 provam que essa afirmação é uma falácia (veja [Figura 3.23](#)). A seguir, está a história por trás das manchetes.

O Pentium usa um algoritmo de divisão de ponto flutuante padrão, que gera bits de quociente múltiplos por etapa, usando os bits mais significativos do divisor e do dividendo para descobrir os 2 bits seguintes do quociente. A escolha vem de uma tabela de pesquisa contendo  $-2, -1, 0, +1$  ou  $+2$ . A escolha é multiplicada pelo divisor e subtraída do resto a fim de gerar um novo resto. Assim como a divisão sem restauração, se uma escolha anterior obtiver um resto muito grande, o resto parcial é ajustado em uma passada subsequente.

Evidentemente, havia cinco elementos da tabela do 80486 que a Intel pensou que nunca poderiam ser acessados, e eles otimizaram a PLA para retornar 0 no lugar de 2 nessas situa-



**FIGURA 3.23 Uma amostra dos artigos de jornais e revistas de novembro de 1994, incluindo New York Times, San Jose Mercury News, San Francisco Chronicle e Infoworld.** O bug da divisão de ponto flutuante do Pentium chegou até mesmo à “Lista dos 10 maiores” do David Letterman Late Show na televisão. A Intel acabou tendo um custo de US\$300 milhões para substituir os chips com defeito.

ções no Pentium. A Intel estava errada: embora os 11 primeiros bits sempre fossem corretos, erros apareceriam ocasionalmente nos bits de 12 a 52, ou do 4º ao 15º dígito decimal.

A seguir está um roteiro dos fatos que aconteceram referentes ao bug do Pentium:

- *Julho de 1994*: a Intel descobre o bug no Pentium. O custo real para consertar o bug foi de várias centenas de milhares de dólares. Após os procedimentos normais de reparo do bug, levariam meses para fazer a mudança, verificar novamente e colocar o chip corrigido em produção. A Intel planejou colocar os chips bons em produção em janeiro de 1995, estimando que 3 a 5 milhões de Pentiums seriam produzidos com o bug.
- *Setembro de 1994*: um professor de matemática no Lynchburg College, na Virgínia, Thomas Nicely, descobre o bug. Depois de ligar para o suporte técnico da Intel e não receber uma posição oficial, ele posta sua descoberta na Internet. Rapidamente surgiram os seguidores e alguns apontaram que até mesmo erros pequenos se tornam grandes ao multiplicar por grandes números: a fração de pessoas com uma doença rara vezes a população da Europa, por exemplo, poderia levar a uma estimativa errada do número de pessoas doentes.
- *7 de novembro de 1994*: o *Electronic Engineering Times* coloca a matéria em sua capa, que logo foi seguido por outros jornais.
- *22 de novembro de 1994*: a Intel emite um comunicado oficial, chamando-o de “glitch”. O Pentium “pode cometer erros no nono dígito. ... Até mesmo a maioria dos engenheiros e analistas financeiros exige precisão apenas até a quarta ou quinta casa decimal. Usuários de planilhas eletrônicas e processadores de textos não precisam se preocupar. ... Talvez haja algumas dezenas de pessoas a quem isso afetaria. Até aqui, só ouvimos falar de uma. ... [Somente] matemáticos teóricos (com computadores Pentium comprados antes do verão) devem se preocupar”. O que aborreceu a muitos foi que os clientes foram solicitados a descrever sua aplicação à Intel, e depois a Intel decidiria se sua aplicação mereceria ou não um novo Pentium sem um bug de divisão.
- *5 de dezembro de 1994*: a Intel afirma que a falha acontece uma vez em 27.000 anos para o usuário típico de planilha. A Intel considera que um usuário realiza 1.000 divisões por dia e multiplica a taxa de erro supondo que os números de ponto flutuante são aleatórios, o que é um em 9 bilhões, e depois apanha 9 milhões de dias, ou 27.000 anos. As coisas começam a acalmar, apesar de a Intel ter deixado de explicar por que um cliente comum acessaria números de ponto flutuante aleatoriamente.
- *12 de dezembro de 1994*: a IBM Research Division discute o cálculo da Intel quanto à taxa de erros (você pode acessar esse artigo visitando [www.mkp.com/books\\_catalog/cod-links.htm](http://www.mkp.com/books_catalog/cod-links.htm)). A IBM afirma que os programas comuns de planilha, calculando por 15 minutos por dia, poderiam produzir erros relacionados ao bug do Pentium com tanta frequência quanto uma vez a cada 24 dias. A IBM considera 5.000 divisões por segundo, por 15 minutos, gerando 4,2 milhões de divisões por dia, e não considera a distribuição aleatória de números, calculando em vez disso as chances como uma em 100 milhões. Como resultado, a IBM imediatamente deixa de enviar todos os computadores pessoais IBM baseados no Pentium. As coisas se aquecem novamente para a Intel.
- *21 de dezembro de 1994*: a Intel lança o seguinte comunicado, assinado pelo presidente da Intel, pelo diretor executivo, pelo diretor de operações e pelo presidente do comitê:

“Nós, da Intel, queremos sinceramente pedir desculpas por nosso tratamento da falha recentemente publicada do processador Pentium. O símbolo Intel Inside significa que seu computador possui um microprocessador que não fica atrás de nenhum outro em qualidade e desempenho. Milhares de funcionários da Intel trabalham muito para garantir que isso aconteça. Mas nenhum microprocessador é totalmente perfeito. O que a Intel continua a acreditar é que, tecnicamente, um problema extremamente pequeno assumiu vida própria. Embora a Intel mantenha

a qualidade da versão atual do processador Pentium, reconhecemos que muitos usuários possuem problemas. Queremos resolvê-los. A Intel trocará a versão atual do processador Pentium por uma versão atualizada, em que essa falha de divisão de ponto flutuante está corrigida, para qualquer proprietário que o solicite, sem qualquer custo, durante toda a vida de seu computador”.

Os analistas estimam que essa troca custou à Intel cerca de US\$500 milhões, e os funcionários da Intel não receberam um bônus de Natal naquele ano.

Essa história nos faz refletir sobre alguns pontos. Seria mais econômico ter consertado o bug em julho de 1994? Qual foi o custo para reparar o dano causado à reputação da Intel? E qual é a responsabilidade corporativa na divulgação de bugs em um produto tão utilizado e confiado como um microprocessador?

Em abril de 1997, outro bug de ponto flutuante foi revelado nos microprocessadores Pentium Pro e Pentium II. Quando as instruções store de ponto flutuante para inteiro (*fist*, *fistp*) encontram um número de ponto flutuante negativo que seja muito grande para caber em uma word de 16 ou 32 bits sendo convertida para inteiro, elas definem o bit errado na palavra de status FPO (exceção de precisão, no lugar de exceção por operação inválida). Para o crédito da Intel, dessa vez, eles reconheceram publicamente o bug e ofereceram um reparo de software para contorná-lo – uma reação muito diferente da que aconteceu em 1994.

Instruções do núcleo MIPS	Nome	Formato	Núcleo aritmético MIPS	Nome	Formato
add	add	R	multiply	mult	R
add immediate	addi	I	multiply unsigned	multu	R
add unsigned	addu	R	divide	div	R
add immediate unsigned	addiu	I	divide unsigned	divu	R
subtract	sub	R	move from Hi	mfhi	R
subtract unsigned	subu	R	move from Lo	mflo	R
AND	and	R	move from system control (EPC)	mfc0	R
AND immediate	andi	I	floating-point add single	add.s	R
OR	or	R	floating-point add double	add.d	R
OR immediate	ori	I	floating-point subtract single	sub.s	R
NOR	nor	R	floating-point subtract double	sub.d	R
shift left logical	sll	R	floating-point multiply single	mul.s	R
shift right logical	srl	R	floating-point multiply double	mul.d	R
load upper immediate	lui	I	floating-point divide single	div.s	R
load word	lw	I	floating-point divide double	div.d	R
store word	sw	I	load word to floating-point single	lwcl	I
load halfword unsigned	lhu	I	store word to floating-point single	swcl	I
store halfword	sh	I	load word to floating-point double	ldc1	I
load byte unsigned	lbu	I	store word to floating-point double	sdc1	I
store byte	sb	I	branch on floating-point true	bc1t	I
load linked (atualização atômica)	ll	I	branch on floating-point false	bc1f	I
store cond. (atualização atômica)	sc	I	floating-point compare single	c.x.s	R
branch on equal	beq	I	(x = eq, neq, lt, le, gt, ge)		
branch on not equal	bne	I	floating-point compare double	c.x.d	R
jump	j	J	(x = eq, neq, lt, le, gt, ge)		
jump and link	jal	J			
jump register	jr	R			
set less than	slt	R			
set less than immediate	slti	I			
set less than unsigned	sltu	R			
set less than immediate unsigned	sltiu	I			

**FIGURA 3.24 O conjunto de instruções MIPS.** Este livro se concentra nas instruções da coluna da esquerda. Essa informação também se encontra nas colunas 1 e 2 do Guia de Instrução Rápida do MIPS no início deste livro.

### 3.9

### Comentários finais

Um efeito colateral do computador com programa armazenado é que os padrões de bits não possuem significado inerente. O mesmo padrão de bits pode representar um inteiro com sinal, um inteiro sem sinal, um número de ponto flutuante, uma instrução e assim por diante. É a instrução que opera sobre os bits que determina seu significado.

A aritmética computacional é distinguida da aritmética de lápis e papel pelas restrições da precisão limitada. Esse limite pode resultar em operações inválidas, por meio do cálculo de números maiores ou menores do que os limites predefinidos. Essas anomalias, chamadas “overflow” ou “underflow”, podem resultar em exceções ou interrupções, eventos de emergência, semelhantes a chamadas de sub-rotina não planejadas. O Capítulo 4 discute as exceções com mais detalhes.

A aritmética de ponto flutuante tem o desafio adicional de ser uma aproximação de números reais e é preciso tomar cuidado para garantir que o número selecionado pelo computador seja a representação mais próxima do número real. Os desafios da imprecisão e da representação limitada fazem parte da inspiração para o campo da análise numérica. A recente passagem para o paralelismo acenderá a tocha na análise numérica novamente, à medida que soluções que eram consideradas seguras nos computadores sequenciais precisam ser reconsideradas quando se tenta encontrar o algoritmo mais rápido para computadores paralelos, que ainda alcance um resultado correto.

Com o passar dos anos, a aritmética computacional tornou-se padronizada, aumentando bastante a portabilidade dos programas. A aritmética de inteiros binários com complemento de dois e a aritmética de ponto flutuante binário do padrão IEEE 754 são encontradas na grande maioria dos computadores vendidos hoje. Por exemplo, cada computador desktop vendido desde que este livro foi impresso pela primeira vez segue essas convenções.

Com a explicação sobre aritmética computacional deste capítulo vem uma descrição de muito mais do conjunto de instruções do MIPS. Uma questão que gera confusão são as instruções explicadas neste capítulo *versus* as instruções executadas pelos chips MIPS *versus* as instruções aceitas pelos montadores MIPS. As duas figuras seguintes tentam esclarecer isso.

A Figura 3.24 lista as instruções MIPS abordadas neste capítulo e no Capítulo 2. Chamamos o conjunto de instruções da esquerda da figura de *núcleo MIPS*. As instruções à direita são chamadas *núcleo aritmético MIPS*. No lado esquerdo da Figura 3.25 estão as instruções que o processador MIPS executa que não se encontram na Figura 3.24. Chamamos o conjunto completo de instrução de hardware de *MIPS-32*. À direita da Figura 3.25 estão as instruções aceitas pelo montador, que não fazem parte do MIPS-32. Chamamos esse conjunto de instruções de *PseudoMIPS*.

A Figura 3.26 indica a popularidade das instruções MIPS para os benchmarks de inteiro e de ponto flutuante SPEC2006. Todas as instruções listadas foram responsáveis por, pelo menos, 0,2% das instruções executadas.

Observe que, embora os programadores e escritores de compilador possam utilizar MIPS-32 para ter um menu de opções mais rico, as instruções do núcleo MIPS dominam a execução SCPEC2006 de inteiros, e o núcleo de inteiros mais aritmético domina o ponto flutuante SPEC2006, como mostra a tabela a seguir.

Subconjunto de instruções	Inteiros	Pt. Flut.
Núcleo do MIPS	98%	31%
Núcleo aritmético do MIPS	2%	66%
MIPS-32 restante	0%	3%

MIPS-32 restantes	Nome	Formato	PseudoMIPS	Nome	Formato
exclusive or ( $rs \oplus rt$ )	xor	R	absolute value	abs	rd,rs
exclusive or immediate	xori	I	negate (com ou sem sinal)	negs	rd,rs
shift right arithmetic	sra	R	rotate left	rol	rd,rs,rt
shift left logical variable	sllv	R	rotate right	ror	rd,rs,rt
shift right logical variable	srlv	R	multiply and don't check oflw (com ou sem sinal)	muls	rd,rs,rt
shift right arithmetic variable	sraw	R	multiply and check oflw (com ou sem sinal)	mulos	rd,rs,rt
move to Hi	mthi	R	divide and check overflow	div	rd,rs,rt
move to Lo	mtlo	R	divide and don't check overflow	divu	rd,rs,rt
load halfword	lh	I	remainder (com ou sem sinal)	remS	rd,rs,rt
load byte	lb	I	load immediate	li	rd,imm
load word left (não alinhado)	lw1	I	load address	la	rd,addr
load word right (não alinhado)	lwr	I	load double	ld	rd,addr
store word left (não alinhado)	sw1	I	store double	sd	rd,addr
store word right (não alinhado)	swr	I	unaligned load word	ulw	rd,addr
load linked (atualização atômica)	l1	I	unaligned store word	usw	rd,addr
store cond. (atualização atômica)	sc	I	unaligned load halfword (com ou sem sinal)	ulhs	rd,addr
move if zero	movz	R	unaligned store halfword	ush	rd,addr
move if not zero	movn	R	branch	b	Label
multiply and add (com ou sem sinal)	madds	R	branch on equal zero	beqz	rs,L
multiply and subtract (com ou sem sinal)	msubs	I	branch on compare (com ou sem sinal)	bxs	rs,rt,L
branch on $\geq$ zero and link	bgezal	I	( $x = 1t, 1e, gt, ge$ )		
branch on $<$ zero and link	bltzal	I	set equal	seq	rd,rs,rt
jump and link register	jalr	R	set not equal	sne	rd,rs,rt
branch compare to zero	bxz	I	set on compare (com ou sem sinal)	sxS	rd,rs,rt
branch compare to zero likely	bxzl	I	( $x = 1t, 1e, gt, ge$ )		
( $x = 1t, 1e, gt, ge$ )			load to floating point (s ou d)	1.f	rd,addr
branch compare reg likely	bx1	I	store from floating point (s ou d)	s.f	rd,addr
trap if compare reg	tx	R			
trap if compare immediate	txi	I			
( $x = eq, neq, 1t, 1e, gt, ge$ )					
return from exception	rfe	R			
system call	syscall	I			
break (causa exceção)	break	I			
move from FP to integer	mfc1	R			
move to FP from integer	mtc1	R			
FP move (s ou d)	mov.f	R			
FP move if zero (s ou d)	movz.f	R			
FP move if not zero (s ou d)	movn.f	R			
FP square root (s ou d)	sqrt.f	R			
FP absolute value (s ou d)	abs.f	R			
FP negate (s ou d)	neg.f	R			
FP convert (w, s ou d)	cvt.f	R			
FP compare un (s ou d)	c.xn.f	R			

**FIGURA 3.25 Conjuntos de instruções MIPS-32 restantes e “pseudoMIPS”.** f significa instruções de ponto flutuante com precisão simples (s) ou dupla (d) e s significa versões com sinal e sem sinal (u). MIPS-32 também possui instruções de PF para multiply e add/sub (*madd.f/msub.f*), ceiling (*ceil.f*), truncate (*trunc.f*), round (*round.f*) e reciprocal (*recip.f*). O sublinhado representa a letra a ser incluída para representar esse tipo de dados.

Para o restante do livro, vamos nos concentrar nas instruções do núcleo MIPS – o conjunto de instruções de inteiros, excluindo multiplicação e divisão – para facilitar a explicação do projeto do computador. Como podemos ver, o núcleo MIPS inclui as instruções MIPS mais comuns, e tenha certeza de que compreender um computador que execute o núcleo MIPS lhe dará base suficiente para entender computadores com projetos ainda mais ambiciosos.

Núcleo MIPS	Nome	Inteiro	PF	Núcleo aritmético + MIPS-32	Nome	Inteiro	PF
add	add	0,0%	0,0%	FP add double	add.d	0,0%	10,6%
add immediate	add i	0,0%	0,0%	FP subtract double	sub.d	0,0%	4,9%
add unsigned	addu	5,2%	3,5%	FP multiply double	mul.d	0,0%	15,0%
add immediate unsigned	addiu	9,0%	7,2%	FP divide double	div.d	0,0%	0,2%
subtract unsigned	subu	2,2%	0,6%	FP add single	add.s	0,0%	1,5%
and	and	0,2%	0,1%	FP subtract single	sub.s	0,0%	1,8%
and immediate	andi	0,7%	0,2%	FP multiply single	mul.s	0,0%	2,4%
or	or	4,0%	1,2%	FP divide single	div.s	0,0%	0,2%
or immediate	ori	1,0%	0,2%	load word to FP double	l.d	0,0%	17,5%
nor	nor	0,4%	0,2%	store word to FP double	s.d	0,0%	4,9%
shift left logical	sll	4,4%	1,9%	load word to FP single	l.s	0,0%	4,2%
shift right logical	srl	1,1%	0,5%	store word to FP single	s.s	0,0%	1,1%
load upper immediate	lui	3,3%	0,5%	branch on floating-point true	bc1t	0,0%	0,2%
load word	lw	18,6%	5,8%	branch on floating-point false	bc1f	0,0%	0,2%
store word	sw	7,6%	2,0%	floating-point compare double	c.x.d	0,0%	0,6%
load byte	lbu	3,7%	0,1%	multiply	mul	0,0%	0,2%
store byte	sb	0,6%	0,0%	shift right arithmetic	sra	0,5%	0,3%
branch on equal (zero)	beq	8,6%	2,2%	load half	lhu	1,3%	0,0%
branch on not equal (zero)	bne	8,4%	1,4%	store half	sh	0,1%	0,0%
jump and link	jal	0,7%	0,2%				
jump register	jr	1,1%	0,2%				
set less than	slt	9,9%	2,3%				
set less than immediate	slti	3,1%	0,3%				
set less than unsigned	sltu	3,4%	0,8%				
set less than imm. uns.	sltiu	1,1%	0,1%				

**FIGURA 3.26 Frequência das instruções MIPS para o benchmark de inteiros e ponto flutuante SPEC2000.** Todas as instruções responsáveis por, pelo menos, 1% das instruções estão incluídas na tabela. As pseudoinstruções são convertidas em MIPS-32 antes da execução e, portanto, não aparecem aqui.

A Lei de Gresham (“dinheiro ruim expulsa o bom”) para os computadores diria: “o rápido expulsa o lento, mesmo que o rápido seja errado”.

W. Kahan, 1992

## 3.10

### Perspectiva histórica e leitura adicional

Esta seção estuda a história do ponto flutuante desde von Neumann, incluindo o esforço surpreendentemente controvertido dos padrões do IEEE, mais o raciocínio para a arquitetura de pilha de 80 bits para ponto flutuante do x86. Ver Seção 3.10 no site.

Nunca ceda, nunca ceda,  
nunca, nunca, nunca – em  
nada, seja grande ou pequeno,  
importante ou insignificante –  
nunca ceda.

Winston Churchill, discurso na  
Harrow School, 1941

## 3.11

### Exercícios

#### Exercício 3.1

O livro mostra como somar e subtrair números binários e decimais. Porém, outros sistemas de numeração também foram muito populares quando se tratavam de computadores. O sistema de numeração octal (base 8) foi um deles. A tabela a seguir mostra pares de números octais.

	A	B
a.	3174	0522
b.	4165	1654

**3.1.1** [5] <3.2> Qual é a soma de A e B se eles representam números octais de 12 bits sem sinal? O resultado deverá ser escrito em octal. Mostre seu trabalho.

**3.1.2** [5] <3.2> Qual é a soma de A e B se eles representam números octais de 12 bits com sinal armazenados em um formato de sinal e magnitude? O resultado deverá ser escrito em octal. Mostre seu trabalho.

**3.1.3** [10] <3.2> Converta A em um número decimal, supondo que ele é sem sinal. Repita considerando que ele esteja armazenado em formato de sinal e magnitude. Mostre seu trabalho.

A tabela a seguir também mostra pares de números octais.

	<b>A</b>	<b>B</b>
a.	7040	0444
b.	4365	3412

**3.1.4** [5] <3.2> O que é A – B se eles representam números octais de 12 bits sem sinal? O resultado deverá ser escrito em octal. Mostre seu trabalho.

**3.1.5** [5] <3.2> O que é A – B se eles representam números octais de 12 bits com sinal armazenados em formato de sinal e magnitude? O resultado deverá ser escrito em octal. Mostre seu trabalho.

**3.1.6** [10] <3.2> Converta A em um número binário. O que torna a base 8 (octal) um sistema de numeração atraente para representar valores nos computadores?

## Exercício 3.2

Hexadecimal (base 16) também é um sistema de numeração normalmente utilizado para representar valores nos computadores. Na verdade, ele se tornou muito mais comum que octal. A tabela a seguir mostra pares de números hexadecimais.

	<b>A</b>	<b>B</b>
a.	1446	672F
b.	2460	4935

**3.2.1** [5] <3.2> Qual é a soma de A e B se eles representam números hexadecimais de 16 bits sem sinal? O resultado deverá ser escrito em hexadecimal. Mostre seu trabalho.

**3.2.2** [5] <3.2> Qual é a soma de A e B se eles representam números hexadecimais de 16 bits com sinal, armazenados em formato de sinal e magnitude? O resultado deverá ser escrito em hexadecimal. Mostre seu trabalho.

**3.2.3** [10] <3.2> Converta A para um número decimal, supondo que ele esteja sem sinal. Repita considerando que ele está armazenado em formato de sinal e magnitude. Mostre seu trabalho.

A tabela a seguir também mostra pares de números hexadecimais.

	<b>A</b>	<b>B</b>
a.	C352	36AE
b.	5ED4	07A4

**3.2.4** [5] <3.2> O que é A – B se eles representam números hexadecimais de 16 bits sem sinal? O resultado deverá ser escrito em hexadecimal. Mostre seu trabalho.

**3.2.5** [5] <3.2> O que é  $A - B$  se eles representam números hexadecimais de 16 bits com sinal, armazenados em formato de sinal e magnitude? O resultado deverá ser escrito em hexadecimal. Mostre seu trabalho.

**3.2.6** [10] <3.2> Converta  $A$  para um número binário. O que torna a base 16 (hexadecimal) um sistema de numeração atraente para representar valores em computadores?

### Exercício 3.3

O overflow ocorre quando um resultado é muito grande para ser representado com precisão dado um tamanho de palavra finito. O underflow ocorre quando um número é muito pequeno para ser representado corretamente — um resultado negativo quando se realiza aritmética sem sinal, por exemplo. (O caso quando um resultado positivo é gerado pela adição de dois inteiros negativos também é considerado como underflow por muitos, mas neste livro isso é considerado overflow.) A tabela a seguir mostra pares de números decimais.

	<b>A</b>	<b>B</b>
a.	69	90
b.	102	44

**3.3.1** [5] <3.2> Suponha que  $A$  e  $B$  sejam inteiros decimais de 8 bits sem sinal. Calcule  $A - B$ . Existe overflow, underflow ou nenhum deles?

**3.3.2** [5] <3.2> Suponha que  $A$  e  $B$  sejam inteiros decimais de 8 bits com sinal, armazenados em formato de magnitude de sinal. Calcule  $A + B$ . Existe overflow, underflow ou nenhum deles?

**3.3.3** [5] <3.2> Suponha que  $A$  e  $B$  sejam inteiros decimais de 8 bits com sinal, armazenados em formato de magnitude de sinal. Calcule  $A - B$ . Existe overflow, underflow ou nenhum deles?

A tabela a seguir também mostra pares de números decimais.

	<b>A</b>	<b>B</b>
a.	200	103
b.	247	237

**3.3.4** [10] <3.2> Suponha que  $A$  e  $B$  sejam inteiros decimais de 8 bits com sinal, armazenados no formato de complemento de dois. Calcule  $A + B$  usando a aritmética por saturação. O resultado deverá ser escrito em decimal. Mostre seu trabalho.

**3.3.5** [10] <3.2> Suponha que  $A$  e  $B$  sejam inteiros decimais de 8 bits com sinal, armazenados no formato de complemento de dois. Calcule  $A - B$  usando a aritmética por saturação. O resultado deverá ser escrito em decimal. Mostre seu trabalho.

**3.3.6** [10] <3.2> Suponha que  $A$  e  $B$  sejam inteiros de 8 bits sem sinal. Calcule  $A + B$  usando a aritmética por saturação. O resultado deverá ser escrito em decimal. Mostre seu trabalho.

### Exercício 3.4

Vejamos a multiplicação com mais detalhes. Usaremos os números na tabela a seguir.

	<b>A</b>	<b>B</b>
a.	62	12
b.	35	26

**3.4.1** [20] <3.3> Usando uma tabela semelhante à que mostramos na Figura 3.7, calcule o produto dos inteiros octais de 6 bits sem sinal A e B usando o hardware descrito na Figura 3.4. Você deverá mostrar o conteúdo de cada registrador em cada etapa.

**3.4.2** [20] <3.3> Usando uma tabela semelhante à que mostramos na Figura 3.7, calcule o produto dos inteiros hexadecimais de 8 bits sem sinal A e B usando o hardware descrito na Figura 3.6. Você deverá mostrar o conteúdo de cada registrador em cada etapa.

**3.4.3** [60] <3.3> Escreva um programa na linguagem assembly MIPS para calcular o produto dos inteiros A e B sem sinal, usando a técnica descrita na Figura 3.4.

A tabela a seguir mostra pares de números octais.

	A	B
a.	41	33
b.	60	26

**3.4.4** [30] <3.3> Ao multiplicar números com sinal, um modo de obter a resposta correta é converter o multiplicador e multiplicando para números positivos, salvar os sinais originais e depois ajustar o valor final de forma apropriada. Usando uma tabela semelhante à que mostramos na Figura 3.7, calcule o produto de A e B usando o hardware descrito na Figura 3.4. Você deverá mostrar o conteúdo de cada registrador em cada etapa, e incluir a etapa necessária para produzir o resultado sinalizado corretamente. Suponha que A e B estejam armazenados em formato de magnitude de sinal com 6 bits.

**3.4.5** [30] <3.3> Ao deslocar um registrador um bit para a direita, existem várias maneiras de decidir qual será o novo bit entrando. Ele sempre pode ser um 0, ou sempre um 1, ou o bit entrando poderia ser aquele que está sendo empurrado pelo lado direito (transformando um deslocamento em uma rotação), ou o valor que já está no bit mais à esquerda pode simplesmente ser retido (chamado de deslocamento aritmético à direita, pois preserva o sinal do número que está sendo deslocado). Usando uma tabela semelhante à que mostramos na Figura 3.7, calcule o produto dos números em complemento de dois com 6 bits A e B usando o hardware descrito na Figura 3.6. Os deslocamentos à direita deverão ser feitos usando-se um deslocamento aritmético à direita. Observe que o algoritmo descrito no texto terá de ser modificado ligeiramente para que isso funcione — em particular, as coisas precisam ser feitas de forma diferente se o multiplicador for negativo. Você poderá encontrar detalhes pesquisando a Web. Mostre o conteúdo de cada registrador a cada etapa.

**3.4.6** [60] <3.3> Escreva um programa em linguagem assembly MIPS para calcular o produto dos inteiros com sinal A e B. Indique se você está usando a técnica dada no Exercício 3.4.4 ou no Exercício 3.4.5.

### Exercício 3.5

Por muitos motivos, gostaríamos de projetar multiplicadores que exijam menos tempo. Muitas técnicas diferentes foram utilizadas para se realizar esse objetivo. Na tabela a seguir, A representa a largura de um inteiro em bits, e B representa o número de unidades de tempo (ut) necessárias para realizar uma etapa de uma operação.

	A (largura em bits)	B (unidades de tempo)
a.	4	3 ut
b.	32	7 ut

**3.5.1** [10] <3.3> Calcule o tempo necessário para realizar uma multiplicação usando a técnica dada nas Figuras 3.4 e 3.5 se um inteiro tiver A bits de largura e cada etapa da

operação exigir B unidades de tempo. Suponha que, na etapa 1a, uma adição sempre é realizada — ou o multiplicando será somado, ou então um 0 será somado. Suponha também que os registradores já foram inicializados (você está simplesmente contando quanto tempo é necessário para se realizar o próprio loop de multiplicação). Se isso estiver sendo feito no hardware, os deslocamentos do multiplicando e do multiplicador podem ser feitos simultaneamente. Se isso estiver sendo feito no software, eles terão de ser feitos um após o outro. Solucione para cada caso.

**3.5.2** [10] <3.3> Calcule o tempo necessário para realizar uma multiplicação usando a técnica descrita no texto (31 somadores empilhados verticalmente) se um inteiro tiver A bits de largura e um somador exigir B unidades de tempo.

**3.5.3** [20] <3.3> Calcule o tempo necessário para realizar uma multiplicação usando a técnica dada na [Figura 3.8](#), se um inteiro tiver A bits de largura e um somador exigir B unidades de tempo.

### Exercício 3.6

Neste exercício, veremos algumas das outras maneiras de melhorar o desempenho da multiplicação, com base principalmente em realizar mais deslocamentos e menos operações aritméticas. A tabela a seguir mostra pares de números hexadecimais.

	A	B
a.	33	55
b.	8a	6d

**3.6.1** [20] <3.3> Conforme discutimos no texto, uma melhoria possível é realizar um deslocamento e soma em vez de uma multiplicação real. Como  $9 \times 6$ , por exemplo, pode ser escrito como  $(2 \times 2 \times 2 + 1) \times 6$ , podemos calcular  $9 \times 6$  deslocando 6 para a esquerda três vezes e depois somando 6 a esse resultado. Mostre a melhor maneira de calcular  $A \times B$  usando deslocamentos e adições/subtrações. Suponha que A e B sejam inteiros de 8 bits sem sinal.

**3.6.2** [20] <3.3> Mostre a melhor maneira de calcular  $A \times B$  usando deslocamento e somas, se A e B forem inteiros de 8 bits com sinal armazenados em formato de magnitude de sinal.

**3.6.3** [60] <3.3> Escreva um programa em linguagem assembly MIPS que realize uma multiplicação de inteiros com sinal usando deslocamento e somas, usando o enfoque descrito em 3.6.1.

A tabela a seguir mostra outros pares de números hexadecimais.

	A	B
a.	F6	7F
b.	08	55

**3.6.4** [30] <3.3> O algoritmo de Booth é outra técnica que reduz o número de operações aritméticas necessárias para realizar uma multiplicação. Esse algoritmo já existe há muitos anos e envolve identificar ciclos de 1s e 0s e realizar apenas deslocamentos durante os ciclos, ao invés de deslocamentos e adições. Ache uma descrição do algoritmo na internet e explique, com detalhes, como ele funciona.

**3.6.5** [30] <3.3> Mostre o resultado passo a passo da multiplicação de A e B, usando o algoritmo de Booth. Suponha que A e B sejam inteiros de 8 bits em complemento de dois, armazenados em formato hexadecimal.

**3.6.6** [60] <3.3> Escreva um programa em linguagem assembly MIPS para realizar a multiplicação de A e B usando o algoritmo de Booth.

### Exercício 3.7

Vejamos a divisão com maiores detalhes. Usaremos os números octais da tabela a seguir.

	A	B
a.	74	21
b.	76	52

**3.7.1** [20] <3.4> Usando uma tabela semelhante à que mostramos na [Figura 3.11](#), calcule A dividido por B usando o hardware descrito na [Figura 3.9](#). Você deverá mostrar o conteúdo de cada registrador em cada etapa. Suponha que A e B sejam inteiros de 6 bits sem sinal.

**3.7.2** [30] <3.4> Usando uma tabela semelhante à que mostramos na [Figura 3.11](#), calcule A dividido por B usando o hardware descrito na [Figura 3.12](#). Você deverá mostrar o conteúdo de cada registrador em cada etapa. Suponha que A e B sejam inteiros de 6 bits sem sinal. Este algoritmo requer uma técnica ligeiramente diferente daquela mostrada na [Figura 3.10](#). Você deverá pensar bem nisso, realizar um experimento ou dois, ou então vá à Web descobrir como fazer isso funcionar corretamente. (Dica: uma solução possível envolve o uso do fato de que a [Figura 3.12](#) implica que o registrador de resto pode ser deslocado em qualquer direção.)

**3.7.3** [60] <3.4> Escreva um programa em linguagem assembly MIPS para calcular A dividido por B, usando a técnica descrita na [Figura 3.9](#). Suponha que A e B sejam inteiros de 6 bits sem sinal.

A tabela a seguir mostra outros pares de números octais.

	A	B
a.	72	07
b.	75	47

**3.7.4** [30] <3.4> Usando uma tabela semelhante à que mostramos na [Figura 3.11](#), calcule A dividido por B usando o hardware descrito na [Figura 3.9](#). Você deverá mostrar o conteúdo de cada registrador em cada etapa. Suponha que A e B sejam inteiros de 6 bits com sinal em formato de magnitude de sinal. Não se esqueça de incluir como você está calculando os sinais do quociente e resto.

**3.7.5** [30] <3.4> Usando uma tabela semelhante à que mostramos na [Figura 3.11](#), calcule A dividido por B usando o hardware descrito na [Figura 3.12](#). Você deverá mostrar o conteúdo de cada registrador em cada etapa. Suponha que A e B sejam inteiros de 6 bits com sinal em formato de magnitude de sinal. Não se esqueça de incluir como você está calculando os sinais do quociente e do resto.

**3.7.6** [60] <3.4> Escreva um programa na linguagem assembly MIPS para calcular A dividido por B, usando a técnica descrita na [Figura 3.12](#). Suponha que A e B sejam inteiros com sinal.

### Exercício 3.8

A [Figura 3.10](#) descreve um algoritmo de divisão com restauração, pois quando subtrai o divisor do resto produz um resultado negativo, o divisor é somado de volta ao resto (restaurando, assim, o valor). Porém, existem outros algoritmos que foram desenvolvidos para eliminar a adição extra. Muitas referências a esses algoritmos são facilmente encontradas na Web. Exploraremos esses algoritmos usando os pares de números octais na tabela a seguir.

	A	B
a.	26	05
b.	37	15

**3.8.1** [30] <3.4> Usando uma tabela semelhante àquela mostrada na Figura 3.11, calcule A dividido por B usando a divisão sem restauração. Você deverá mostrar o conteúdo de cada registrador a cada etapa. Suponha que A e B sejam inteiros de 6 bits sem sinal.

**3.8.2** [60] <3.4> Escreva um programa em linguagem assembly MIPS para calcular A dividido por B usando a divisão sem restauração. Suponha que A e B sejam inteiros de 6 bits com sinal (complemento de dois).

**3.8.3** [60] <3.4> Compare o desempenho da divisão com e sem restauração. Demonstre exibindo o número de etapas necessárias para calcular A dividido por B usando cada método. Suponha que A e B sejam inteiros de 6 bits com sinal (magnitude de sinal). Você também pode escrever um programa para realizar as divisões com e sem restauração.

A tabela a seguir mostra outros pares de números octais.

	A	B
a.	27	06
b.	54	12

**3.8.4** [30] <3.4> Usando uma tabela semelhante à que mostramos na Figura 3.11, calcule A dividido por B usando a divisão nonperforming. Você deverá mostrar o conteúdo de cada registrador a cada etapa. Suponha que A e B sejam inteiros de 6 bits sem sinal.

**3.8.5** [60] <3.4> Escreva um programa em linguagem assembly MIPS para calcular A dividido por B usando a divisão nonperforming. Suponha que A e B sejam inteiros de 3 bits com sinal em complemento de dois.

**3.8.6** [60] <3.4> Como o desempenho da divisão nonrestoring e nonperforming se comparam? Demonstre exibindo o número de etapas necessárias para calcular A dividido por B usando cada método. Suponha que A e B sejam inteiros de 6 bits com sinal, armazenados em formato de magnitude de sinal. Você também pode escrever um programa para realizar as divisões nonperforming e nonrestoring.

### Exercício 3.9

A divisão é tão demorada e difícil que o guia do CRAY T3E Fortran Optimization afirma: “A melhor estratégia para divisão é evitá-la sempre que for possível.” Este exercício examina as diferentes estratégias para realizar divisões.

a.	Divisão sem restauração
b.	Divisão por multiplicação recíproca

**3.9.1** [30] <3.4> Descreva o algoritmo com detalhes.

**3.9.2** [60] <3.4> Use um fluxograma (ou um trecho de código de alto nível) para descrever como o algoritmo funciona.

**3.9.3** [60] <3.4> Escreva um programa em linguagem assembly MIPS para realizar uma divisão usando o algoritmo.

### Exercício 3.10

Em uma arquitetura de Von Neumann, grupos de bits não possuem significados intrínsecos por si próprios. O que um padrão de bits representa depende totalmente de como ele é utilizado. A tabela a seguir mostra os padrões de bits expressos em notação hexadecimal.

a.	0x0C000000
b.	0xC4630000

**3.10.1** [5] <3.5> Que número decimal o padrão de bits representa se ele for um inteiro em complemento de dois? E um inteiro sem sinal?

**3.10.2** [10] <3.5> Se esse padrão de bits for colocado no Registrador de Instrução, que instrução MIPS será executada?

**3.10.3** [10] <3.5> Que número decimal o padrão de bits representa se ele for um número de ponto flutuante? Use o padrão IEEE 754.

A tabela a seguir mostra números decimais.

a.	63,25
b.	146987, 40625

**3.10.4** [10] <3.5> Escreva a representação binária do número decimal, considerando o formato de precisão simples IEEE 754.

**3.10.5** [10] <3.5> Escreva a representação binária do número decimal, considerando o formato de precisão dupla IEEE 754.

**3.10.6** [10] <3.5> Escreva a representação binária do número decimal considerando que ele foi armazenado usando-se o formato IBM de precisão simples (base 16, em vez da base 2, com 7 bits de expoente).

### Exercício 3.11

No padrão de ponto flutuante IEEE 754, o expoente é armazenado em formato de “bias” (também conhecido como “Excess-N”). Essa técnica foi selecionada porque queremos que um padrão com apenas zeros seja o mais próximo de zero possível. Em razão do uso de um 1 oculto, se tivéssemos de representar o expoente no formato de complemento de dois, um padrão com apenas zeros na realidade seria o número 1! (Lembre-se de que qualquer coisa elevada à potência zero é 1 e, portanto,  $1,0^0 = 1$ .) Há muitos outros aspectos do padrão IEEE 754 que ajudam as unidades de ponto flutuante do hardware a trabalharem mais rapidamente. Porém, em muitas máquinas mais antigas, os cálculos de ponto flutuante eram tratados no software, e, portanto, outros formatos foram utilizados. A tabela a seguir mostra números decimais.

a.	$-1,5625 \times 10^{-1}$
b.	$9,356875 \times 10^2$

**3.11.1** [20] <3.5> Escreva o padrão de bits binário considerando um formato semelhante ao empregado pelo DEC PDP-8 (12 bits da esquerda são o expoente armazenado como um número de complemento de dois, e os 24 bits da direita são a mantissa armazenada como um número de complemento de dois.) Nenhum 1 oculto é utilizado. Compare o intervalo e a precisão desse padrão de 36 bits com os padrões IEEE 754 de precisão simples e dupla.

**3.11.2** [20] <3.5> NVIDIA tem um formato “metade”, que é semelhante ao IEEE 754, exceto que tem apenas 16 bits de largura. O bit mais à esquerda ainda é o bit de sinal, o ex-

poente tem 5 bits de largura e é armazenado no formato Excess-16, e a mantissa tem 10 bits de extensão. Assume-se que existe um 1 oculto. Escreva o padrão de bits considerando uma versão modificada desse formato que utiliza um formato com excesso de 16 para armazenar o expoente. Comente sobre o intervalo e a precisão desse padrão de 16 bits com o padrão IEEE 754 de precisão simples.

**3.11.3** [20] <3.5> Os Hewlett-Packard 2114, 2115 e 2116 usavam um formato com os 16 bits mais à esquerda sendo a mantissa armazenada no formato de complemento de dois, seguida por outro campo de 16 bits que tinha nos 8 bits mais à esquerda uma extensão da mantissa (fazendo com que a mantissa tenha 24 bits de extensão) e os 8 bits mais à direita representando o expoente. Porém, por um capricho interessante, o expoente era armazenado em formato de magnitude de sinal com o bit de sinal no canto direito! Escreva o padrão de bits considerando esse formato. Nenhum 1 oculto é utilizado. Compare o intervalo e a precisão desse padrão de 32 bits com o padrão IEEE 754 de precisão simples.

A tabela a seguir mostra pares de números decimais.

	A	B
a.	$2,6125 \times 10^1$	$4,150390625 \times 10^{-1}$
b.	$-4,484375 \times 10^1$	$1,3953125 \times 10^1$

**3.11.4** [20] <3.5> Calcule a soma de A e B à mão, supondo que A e B sejam armazenados no formato NVIDIA de 16 bits descrito no Exercício 3.11.2 (e também descrito no texto). Considere um bit de guarda, um bit de arredondamento e um sticky bit, e arredonde para o par mais próximo. Mostre todas as etapas.

**3.11.5** [60] <3.5> Escreva um programa em linguagem de máquina do assembly MIPS para calcular a soma de A e B, supondo que estes estejam armazenados no formato NVIDIA de 16 bits descrito no Exercício 3.11.2 (e também descrito no texto). Considere um bit de guarda, um bit de arredondamento e um sticky bit, e arredonde para o par mais próximo.

**3.11.6** [60] <3.5> Escreva um programa em linguagem assembly MIPS para calcular a soma de A e B, considerando que eles estejam armazenados usando o formato descrito no Exercício 3.11.1. Agora, modifique o programa para calcular a soma considerando o formato descrito no Exercício 3.11.3. Que formato é mais fácil para um programador lidar? Compare-os com o formato IEEE 754. (Não se preocupe com os sticky bits nesta questão.)

## Exercício 3.12

A multiplicação de ponto flutuante é ainda mais complicada e desafiadora que a adição de ponto flutuante, e ambas são mínimas se comparadas à divisão de ponto flutuante. A tabela a seguir apresenta pares de números decimais.

	A	B
a.	$-8,0546875 \times 10^0$	$-1,79931640625 \times 10^{-1}$
b.	$8,59375 \times 10^{-2}$	$8,125 \times 10^{-1}$

**3.12.1** [30] <3.5> Calcule o produto de A e B manualmente, considerando que A e B sejam armazenados no formato NVIDIA de 16 bits descrito no Exercício 3.11.2 (e também descrito no texto). Considere um bit de guarda, um bit de arredondamento e um sticky bit, e arredonde para o par mais próximo. Mostre todas as etapas; porém, como acontece no exemplo do texto, você pode realizar a multiplicação em formato legível aos humanos, em vez de usar as técnicas descritas nos Exercícios de 3.4 a 3.6. Indique se existe overflow ou underflow. Escreva sua resposta como um padrão de 16 bits e também como um número decimal. Qual é a precisão do seu resultado? Compare-o com o número que você obtém se realizar a multiplicação em uma calculadora.

**3.12.2** [60] <3.5> Escreva um programa para calcular o produto de A e B, considerando que eles sejam armazenados no formato IEEE 754. Indique se existe overflow ou underflow. (Lembre-se de que o IEEE 754 considera um bit de guarda, um bit de arredondamento e um sticky bit, e arredonde para o par mais próximo.)

**3.12.3** [60] <3.5> Escreva um programa em linguagem assembly MIPS para calcular o produto de A e B, considerando que eles sejam armazenados no formato descrito no Exercício 3.11.1. Agora, modifique o programa para calcular o produto considerando o formato descrito no Exercício 3.11.3. Que formato é mais fácil para um programador lidar? Compare-os com o formato IEEE 754. (Não se preocupe com os sticky bits nesta questão.)

A tabela a seguir mostra outros pares de números decimais.

	<b>A</b>	<b>B</b>
a.	$8,625 \times 10^1$	$-4,875 \times 10^0$
b.	$1,84375 \times 10^0$	$1,3203125 \times 10^0$

**3.12.4** [30] <3.5> Calcule A dividido por B manualmente. Mostre todas as etapas necessárias para se chegar à sua resposta. Suponha que exista um bit de guarda, de arredondamento e um sticky bit, e use-os se for necessário. Escreva a resposta final em formato de ponto flutuante com 16 bits descrito no exercício 3.11.2 e em decimal, comparando o resultado decimal com o que você obtém usando uma calculadora.

Os Livermore Loops<sup>1</sup> são um conjunto de kernels intensos de ponto flutuante tirados de programas científicos executados no Lawrence Livermore Laboratory. A tabela a seguir identifica os kernels individuais do conjunto.

a.	Livermore Loop 3
b.	Livermore Loop 9

**3.12.5** [60] <3.5> Escreva o loop na linguagem assembly MIPS.

**3.12.6** [60] <3.5> Descreva, com detalhes, uma técnica para realizar divisão de ponto flutuante em um computador digital. Não se esqueça de incluir referências às fontes que você utilizou.

### Exercício 3.13

As operações realizadas sobre inteiros de ponto fixo se comportam como se espera — as leis comutativa, associativa e distributiva permanecem. Contudo, isso nem sempre acontece quando se trabalha com números de ponto flutuante. Primeiro, vejamos a lei associativa. A tabela a seguir mostra conjuntos de números decimais.

	<b>A</b>	<b>B</b>	<b>C</b>
a.	$3,984375 \times 10^{-1}$	$3,4375 \times 10^{-1}$	$1,771 \times 10^3$
b.	$3,96875 \times 10^0$	$8,46875 \times 10^0$	$2,1921875 \times 10^1$

**3.13.1** [20] <3.2, 3.5, 3.6> Calcule  $(A + B) + C$  manualmente, considerando que A, B e C são armazenados no formato NVIDIA de 16 bits, descrito no Exercício 3.11.2 (e também descrito no texto). Considere um bit de guarda, um bit de arredondamento e um sticky bit, e arredonde para o par mais próximo. Mostre todas as etapas e escreva sua resposta em formato de ponto flutuante de 16 bits e em decimal.

<sup>1</sup> Você poderá encontrá-los em <http://www.netlib.org/benchmark/livermore>.

**3.13.2** [20] <3.2, 3.5, 3.6> Calcule  $A + (B + C)$  manualmente, considerando que A, B e C são armazenados no formato NVIDIA de 16 bits, descrito no Exercício 3.11.2 (e também descrito no texto). Considere um bit de guarda, um bit de arredondamento e um sticky bit, e arredonde para o par mais próximo. Mostre todas as etapas, e escreva sua resposta em formato de ponto flutuante de 16 bits e em decimal.

**3.13.3** [10] <3.2, 3.5, 3.6> Com base nas suas respostas dos Exercícios 3.13.1 e 3.13.2,  $(A + B) + C = A + (B + C)$ ?

A tabela a seguir mostra outros conjuntos de números decimais.

	A	B	C
a.	$3,41796875 \times 10^{-3}$	$6,34765625 \times 10^{-3}$	$1,05625 \times 10^2$
b.	$1,140625 \times 10^2$	$-9,135 \times 10^2$	$9,84375 \times 10^{-1}$

**3.13.4** [30] <3.3, 3.5, 3.6>  $(A \times B) \times C$  manualmente, considerando que A, B e C são armazenados no formato NVIDIA de 16 bits, descrito no Exercício 3.11.2 (e também descrito no texto). Considere um bit de guarda, um bit de arredondamento e um sticky bit, e arredonde para o par mais próximo. Mostre todas as etapas e escreva sua resposta em formato de ponto flutuante de 16 bits e em decimal.

**3.13.5** [30] <3.3, 3.5, 3.6> Calcule  $A \times (B \times C)$  manualmente, considerando que A, B e C são armazenados no formato NVIDIA de 16 bits, descrito no Exercício 3.11.2 (e também descrito no texto). Considere um bit de guarda, um bit de arredondamento e um sticky bit, e arredonde para o par mais próximo. Mostre todas as etapas e escreva sua resposta em formato de ponto flutuante de 16 bits e em decimal.

**3.13.6** [10] <3.3, 3.5, 3.6> Com base nas suas respostas dos Exercícios 3.13.4 e 3.13.5,  $(A \times B) \times C = A \times (B \times C)$ ?

## Exercício 3.14

A lei associativa não é a única que nem sempre se mantém quando se lidam com números de ponto flutuante. Existem outras coisas estranhas que também ocorrem. A tabela a seguir mostra conjuntos de números decimais.

	A	B	C
a.	$1,666015625 \times 10^0$	$1,9760 \times 10^4$	$-1,9744 \times 10^4$
b.	$3,48 \times 10^2$	$6,34765625 \times 10^{-2}$	$-4,052734375 \times 10^{-2}$

**3.14.1** [30] <3.2, 3.3, 3.5, 3.6> Calcule  $A \times (B + C)$  manualmente, considerando que A, B e C são armazenados no formato NVIDIA de 16 bits, descrito no Exercício 3.11.2 (e também descrito no texto). Considere um bit de guarda, um bit de arredondamento e um sticky bit, e arredonde para o par mais próximo. Mostre todas as etapas e escreva sua resposta em formato de ponto flutuante de 16 bits e em decimal.

**3.14.2** [30] <3.2, 3.3, 3.5, 3.6> Calcule  $(A \times B) + (A \times C)$  manualmente, considerando que A, B e C são armazenados no formato NVIDIA de 16 bits, descrito no Exercício 3.11.2 (e também descrito no texto). Considere um bit de guarda, um bit de arredondamento e um sticky bit, e arredonde para o par mais próximo. Mostre todas as etapas e escreva sua resposta em formato de ponto flutuante de 16 bits e em decimal.

**3.14.3** [10] <3.2, 3.3, 3.5, 3.6> Com base nas suas respostas dos Exercícios 3.14.1 e 3.14.2,  $(A \times B) + (A \times C) = A \times (B + C)$ ?

A tabela a seguir mostra dois pares, cada um consistindo em uma fração e um inteiro.

	<b>A</b>	<b>B</b>
a.	-1/4	4
b.	1/10	10

**3.14.4** [10] <3.5> Usando o formato de ponto flutuante IEEE 754, escreva o padrão de bits que representaria A. Você consegue representar A com exatidão?

**3.14.5** [10] <3.2, 3.3, 3.5, 3.6> O que você obtém se somar A a si mesmo B vezes? Quanto é  $A \times B$ ? Eles são iguais? O que deveriam ser?

**3.14.6** [60] <3.2, 3.3, 3.4, 3.5, 3.6> O que você obtém se apanhar a raiz quadrada de B e depois multiplicar esse valor por si mesmo? O que você deveria obter? Faça isso para números de ponto flutuante com precisão simples e dupla. (Escreva um programa para realizar esses cálculos.)

### Exercício 3.15

Números binários são utilizados no campo de mantissa, mas eles não precisam ser binários. A IBM usou números de base 16, por exemplo, em alguns de seus formatos de ponto flutuante. Existem outras técnicas que também são possíveis, cada uma com suas vantagens e desvantagens em particular. A tabela a seguir mostra frações a serem representadas em diversos formatos de ponto flutuante.

a.	1/3
b.	1/10

**3.15.1** [10] <3.5, 3.6> Escreva o padrão de bits na mantissa considerando um formato de ponto flutuante que usa números binários na mantissa (basicamente, o que você esteve fazendo neste capítulo). Suponha que existam 24 bits e você não precisa normalizar. Essa representação é exata?

**3.15.2** [10] <3.5, 3.6> Escreva o padrão de bits na mantissa considerando um formato de ponto flutuante que usa números Binary Coded Decimal (base 10) na mantissa, em vez da base 2. Suponha que existam 24 bits e você não precisa normalizar. Essa representação é exata?

**3.15.3** [10] <3.5, 3.6> Escreva o padrão de bits supondo que estamos usando números de base 15 na mantissa, em vez da base 2. (Números de base 16 utilizam os símbolos 0-9 e A-F. Números de base 15 usariam 0-9 e A-E.) Suponha que existam 24 bits e você não precisa normalizar. Essa representação é exata?

**3.15.4** [20] <3.5, 3.6> Escreva o padrão de bits supondo que estamos usando números de base 30 na mantissa, em vez da base 2. (Números de base 16 utilizam os símbolos 0-9 e A-F. Números de base 30 usariam 0-9 e A-T.) Suponha que existam 24 bits e você não precisa normalizar. Essa representação é exata? Você consegue ver alguma vantagem no uso dessa técnica?

§3.2: página 229: 3.

§3.4: página 269: 3.

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