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_{dec} por 1001_{dec}:

Multiplicando		1000_{dec}
Multiplicador	×	$\underline{1001}_{dec}$
		1000
		0000
		0000
		1000
Produto		1001000_{dec}

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

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

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

1. Basta colocar uma cópia do multiplicando (1 x multiplicando) no lugar apropriado se o dígito do multiplicador for 1, ou
2. Colocar 0 (0 x 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 multiplicado para a esquerda um

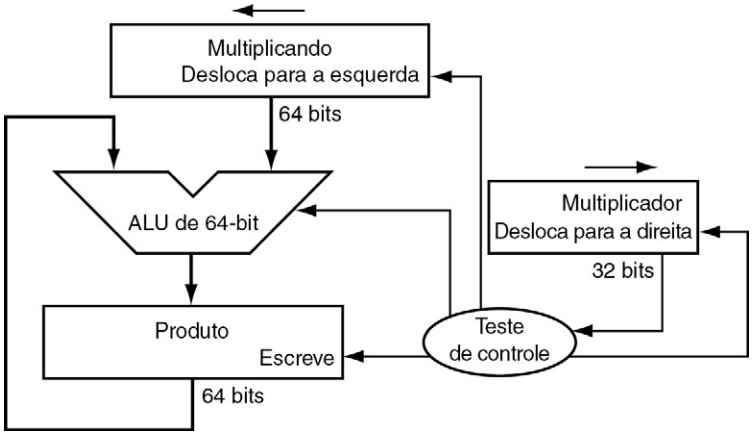

FIGURA 3.4 Primeira versão do hardware de multiplicação. O registrador do Multiplicando, a ALU, e o registrador do Produto possuem 64 bits de largura, apenas com o registrador do Multiplicador contendo 32 bits. (O **Apêndice C** descreve as ALUs.) O multiplicando com 32 bits começa na metade direita 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.

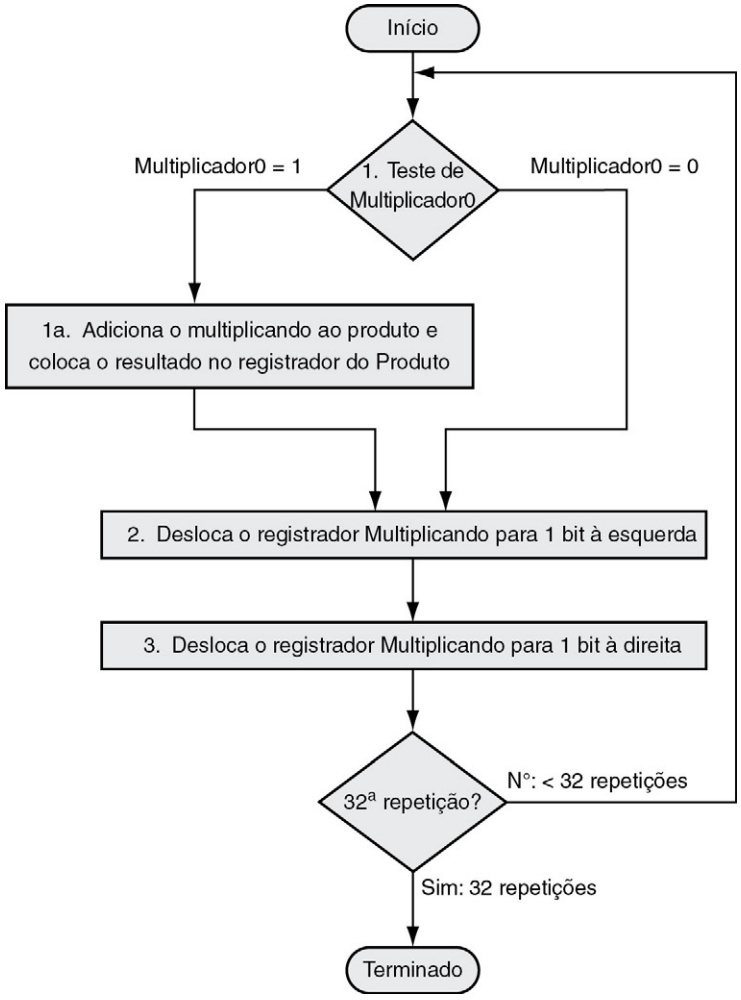

FIGURA 3.5 O primeiro algoritmo de multiplicação, usando o hardware mostrado na Figura 3.4. Se o bit menos significativo do multiplicador for 1, some o multiplicando ao produto. Caso contrário, vá para a etapa seguinte. Desloque o multiplicando para a esquerda e o multiplicador para a direita nas duas 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.

Interface hardware/ software

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.

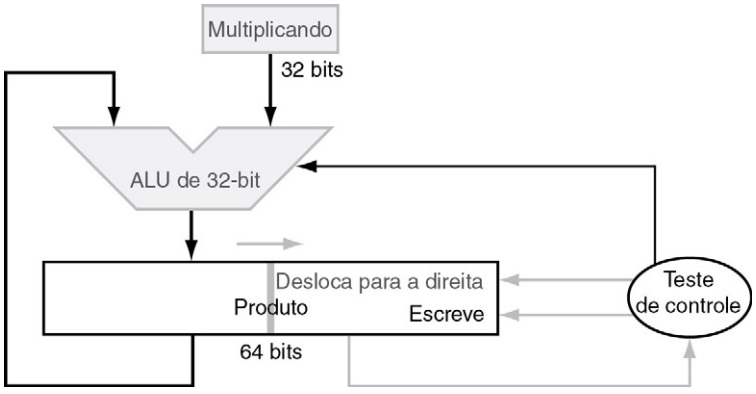

FIGURA 3.6 Versão refinada do hardware de multiplicação. Compare com a primeira versão na Figura 3.4. O registrador Multiplicando, a ALU, e o registrador Multiplicador possuem 32 bits de extensão, com somente o registrador Produto restando nos 64 bits. Agora, o produto é deslocado para a direita. O registrador 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

EXEMPLO

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

RESPOSTA

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 $0000\ 0110_{bin}$ ou 6_{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.

Iteração	Passo	Multiplicador	Multiplicando	Produto
0	Valores iniciais	001(1)	0000 0010	0000 0000
1	1a: 1 ⇒ 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	001(1)	0000 0100	0000 0010
2	1a: 1 ⇒ 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	000(0)	0000 1000	0000 0110
3	1: 0 ⇒ Nenhuma operação	0000	0000 1000	0000 0110
	2: Desloca o Multiplicando à esquerda	0000	0001 0000	0000 0110
	3: Desloca o Multiplicador à direita	000(0)	0001 0000	0000 0110
4	1: 0 ⇒ 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 lembrarmos 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.

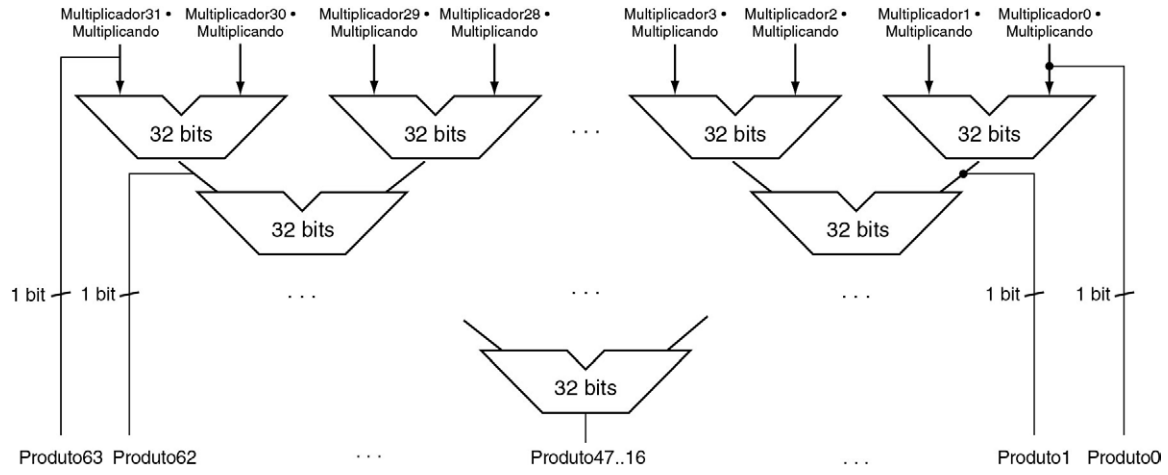

FIGURA 3.8 Hardware da multiplicação rápida. Em vez de usar um único somador de 32 bits 31 vezes, esse hardware "desenrola o loop" para usar 31 somadores e depois os organiza para minimizar o atraso.

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.

Interface hardware/ software

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

3.4 Divisão

Divide et impera.

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

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

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

$$
\begin{array}{rll}
 & 1001_{dec} & \text{Quociente} \\
\text{Divisor } 1000_{dec} & \overline{)\,1001010_{dec}} & \text{Dividendo} \\
 & \underline{-1000} & \\
 & 10 & \\
 & 101 & \\
 & 1010 & \\
 & \underline{-1000} & \\
 & 10_{dec} & \text{Resto}
\end{array}
$$

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. Nosso 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.

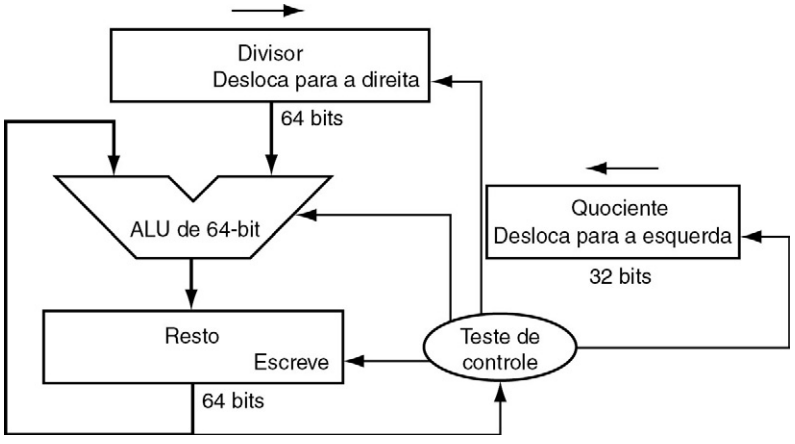

FIGURA 3.9 Primeira versão do hardware de divisão. O registrador Divisor, a ALU e o registrador Resto possuem 64 bits de largura, com apenas o registrador Quociente tendo 32 bits. O divisor de 32 bits começa na metade esquerda do registrador Divisor e é deslocado 1 bit para a direita em cada iteração. 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.

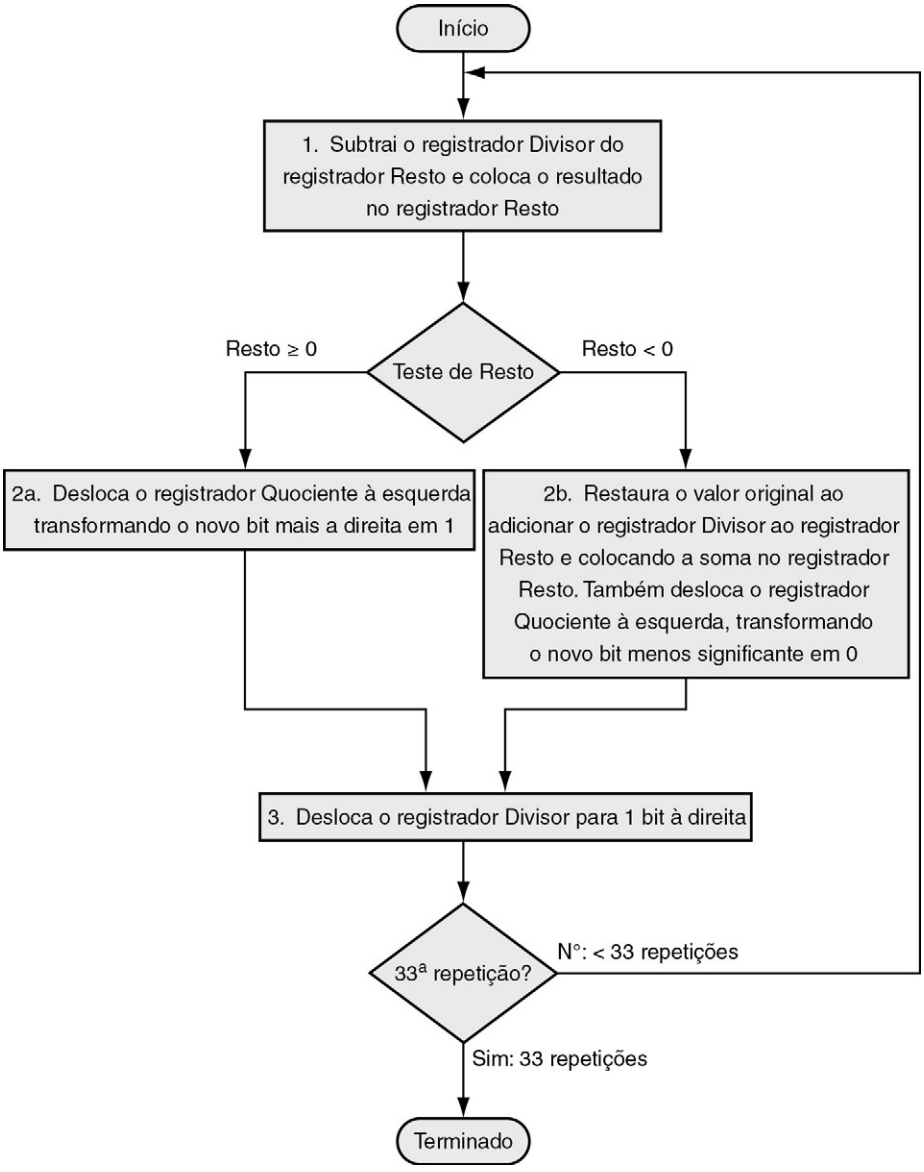

FIGURA 3.10 Um algoritmo de divisão, usando o hardware da Figura 3.9. Se o Resto é positivo, o divisor coube no dividendo, de modo que a etapa 2a gera um 1 no quociente. Um Resto negativo após a etapa 1 significa que o divisor não coube no dividendo, de modo que a etapa 2b gera um 0 no quociente e soma 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

EXEMPLO

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

RESPOSTA

A Figura 3.11 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.

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 ⟹ +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 ⟹ +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 ⟹ +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	⓪000 0011
	2a: Resto ≥ 0 ⟹ 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	⓪000 0001
	2a: Resto≥ 0 ⟹ 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 $\pm 7_{dec}$ por $\pm 2_{dec}$. O primeiro caso é fácil:

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

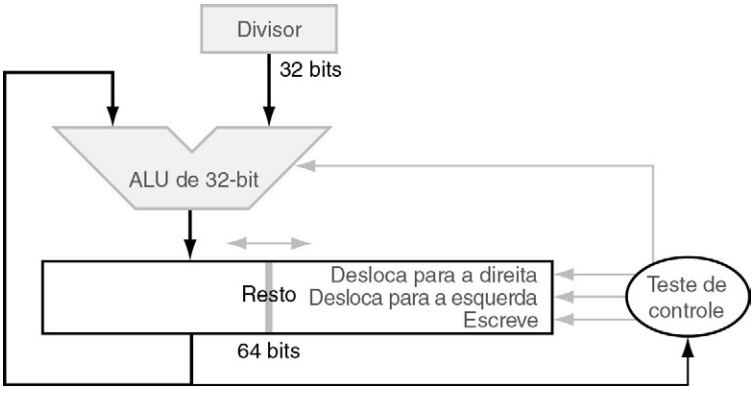

FIGURA 3.12 Uma versão melhorada do hardware de divisão. O registrador Divisor, a ALU e o registrador Quociente possuem 32 bits de largura, com apenas o registrador Resto ficando com 64 bits. Em comparação com a Figura 3.9, os registradores ALU e Divisor são divididos ao meio, e o resto é deslocado à 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.

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.