

# Instruções: A Linguagem dos Computadores

---

*Eu falo espanhol com Deus, italiano com as mulheres, francês com os homens e alemão com meu cavalo.*

*Charles V, imperador romano (1500-1558)*

- 2.1 Introdução
- 2.2 Operações do hardware do computador
- 2.3 Operandos do hardware do computador
- 2.4 Números com sinal e sem sinal
- 2.5 Representando instruções no computador
- 2.6 Operações lógicas
- 2.7 Instruções para tomada de decisões
- 2.8 Suporte a procedimentos no hardware do computador
- 2.9 Comunicando-se com as pessoas
- 2.10 Endereçamento no MIPS para operandos imediatos e endereços de 32 bits
- 2.11 Paralelismo e instruções: Sincronização
- 2.12 Traduzindo e iniciando um programa
- 2.13 Um exemplo de ordenação em C para juntar tudo isso
- 2.14 Arrays *versus* ponteiros
- 2.15 Vida real: instruções ARMv7 (32 bits)

2.16 Vida real: Instruções x86

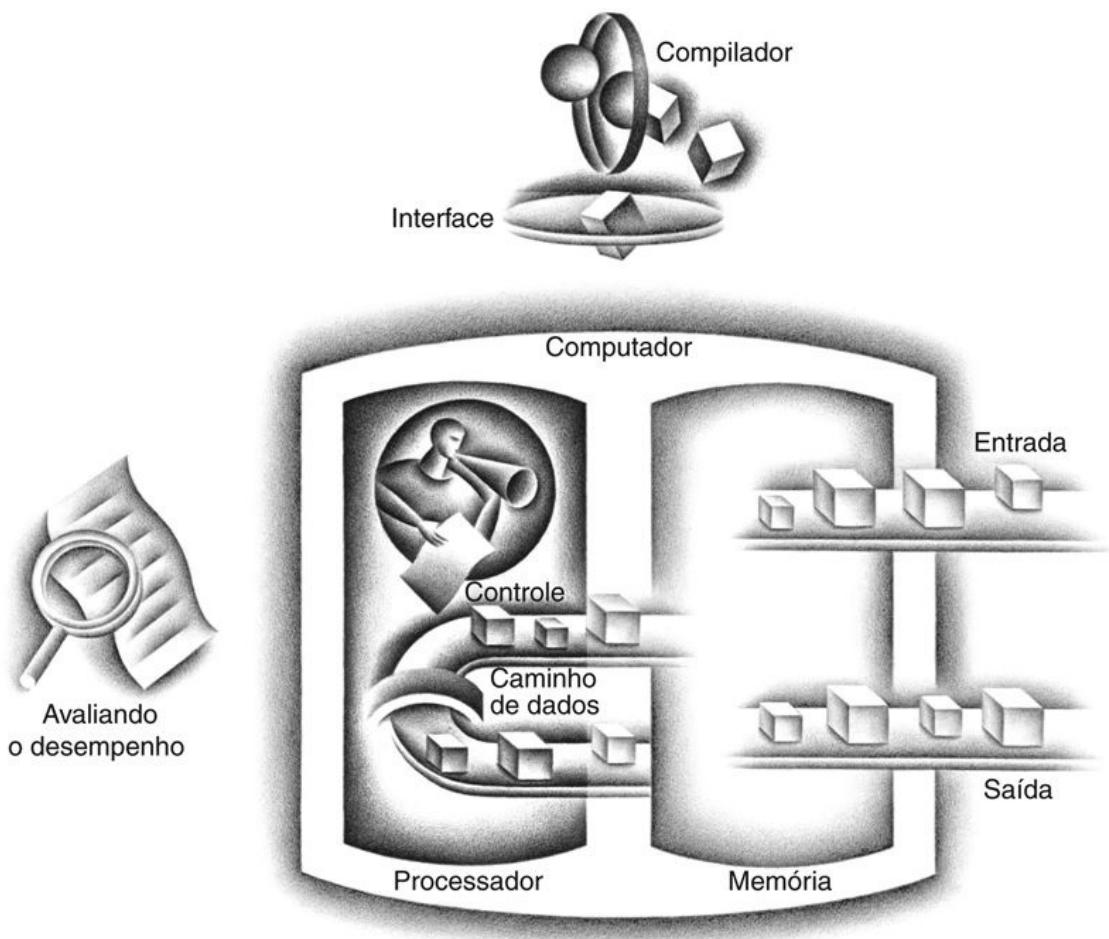
2.17 Vida real: instruções ARMv8 (64 bits)

2.18 Falácia e armadilhas

2.19 Comentários finais

2.20 Exercícios

# Os cinco componentes clássicos de um computador



## 2.1. Introdução

Para controlar o hardware de um computador é preciso falar sua linguagem. As palavras da linguagem de um computador são chamadas *instruções* e seu vocabulário é denominado **conjunto de instruções**. Neste capítulo você verá o conjunto de instruções de um computador real, tanto na forma escrita pelos humanos quanto na forma lida pelo computador. Apresentamos as instruções em

um padrão *top-down*. Começando com uma notação parecida com uma linguagem de programação restrita; nós a refinamos passo a passo, até que você veja a linguagem real de um computador. O [Capítulo 3](#) continua nosso caminho, expondo o hardware para a aritmética e a representação dos números de ponto flutuante.

## conjunto de instruções

O vocabulário dos comandos entendidos por uma determinada arquitetura.

Você poderia pensar que as linguagens dos computadores fossem tão diversificadas quanto as dos humanos, mas, na realidade, as linguagens de computação são muito semelhantes, mais parecidas com dialetos regionais do que linguagens independentes. Logo, quando você aprender uma, será fácil entender as outras.

O conjunto de instruções escolhido vem da *MIPS Technologies*, e é um exemplo elegante dos conjuntos de instruções projetados desde a década de 1980. Para demonstrar como é fácil selecionar outros conjuntos de instruções, daremos uma olhada rápida em três outros conjuntos de instruções populares.

1. ARMv7 é semelhante ao MIPS. Mais de 9 bilhões de chips com processadores ARM foram fabricados em 2011, tornando-o o conjunto de instruções mais popular no mundo.
2. O segundo exemplo é o Intel x86, que controla tanto o PC quanto a nuvem da era pós-PC.
3. O terceiro exemplo é o ARMv8, que amplia o tamanho de endereçamento do ARMv7 de 32 bits para 64 bits. Ironicamente, conforme veremos, esse conjunto de instruções de 2013 está mais próximo do MIPS do que do ARMv7.

A semelhança dos conjuntos de instruções ocorre porque todos os computadores são construídos a partir de tecnologias de hardware baseadas em princípios básicos semelhantes e porque existem algumas operações básicas que todos os computadores precisam oferecer. Além do mais, os projetistas de computador possuem um objetivo comum: encontrar uma linguagem que facilite o projeto do hardware e do compilador enquanto maximiza o desempenho e minimiza o custo. Este objetivo é antigo; a citação a seguir foi escrita antes que você pudesse comprar um computador e é tão verdadeira hoje quanto era em 1947:

*É fácil ver, por métodos lógicos formais, que existem certos [conjuntos de instruções] que são adequados para controlar e causar a execução de qualquer sequência de operações... As considerações realmente decisivas, do ponto de vista atual, na seleção de um [conjunto de instruções], são mais de natureza prática: a simplicidade do equipamento exigido pelo [conjunto de instruções] e a clareza de sua aplicação para os problemas realmente importantes, junto com a velocidade com que tratam esses problemas.*

*Burks, Goldstine e von Neumann, 1947*

A “simplicidade do equipamento” é uma consideração tão valiosa para os computadores de hoje, quanto foi para os da década de 1950. O objetivo deste capítulo é ensinar um conjunto de instruções que siga esse conselho, mostrando como ele é representado no hardware e o relacionamento entre as linguagens de programação de alto nível e essa linguagem mais primitiva. Nossos exemplos estão na linguagem de programação C.

Aprendendo como representar as instruções, você também descobrirá o segredo da computação: o **conceito de programa armazenado**. Você também verá o impacto das linguagens de programação e das otimizações do compilador sobre o desempenho. Concluímos com uma visão da evolução histórica dos conjuntos de instruções e uma visão geral dos outros dialetos do computador.

## conceito de programa armazenado

A ideia de que as instruções e os dados de muitos tipos podem ser armazenados na memória como números, levando ao computador do programa armazenado.

Revelamos o conjunto de instruções do MIPS aos poucos, mostrando o raciocínio em conjunto com as estruturas do computador. Esse tutorial passo a passo entrelaça os componentes com suas explicações, tornando a linguagem de máquina mais fácil de digerir. A [Figura 2.1](#) oferece uma prévia do conjunto de instruções abordado neste capítulo.

### Operandos do MIPS

Nome	Exemplo	Comentários
32 registradores	$\$s0-\$s7$ , $\$t0-\$t9$ , $\$zero$ , $\$a0-\$a3$ , $\$v0-\$v1$ , $\$gp$ , $\$fp$ , $\$sp$ , $\$ra$ , $\$at$	Localizações rápidas para dados. No MIPS, os dados precisam estar em regiões para realizar aritmética. O registrador $\$zero$ sempre é igual a 0 e o registrador $\$at$ é reservado pelo montador para lidar com constantes grandes.
$2^{30}$ palavras de memória	Memória[0], Memória[4], ..., Memória[4294967292]	Acessada apenas pelas instruções de transferência de dados. O MIPS utiliza endereços de byte, de modo que os endereços sequenciais de dados diferem em 4. A memória mantém estruturas de dados, matrizes e registros espalhados.

### Linguagem assembly do MIPS

Categoria	Instrução	Exemplo	Significado	Comentários
Aritmética	add	add \$s1,\$s2,\$s3	$$s1 = \$s2 + \$s3$	Três operandos; dados nos registradores
	subtract	sub \$s1,\$s2,\$s3	$$s1 = \$s2 - \$s3$	Três operandos; dados nos registradores
	add immediate	addi \$s1,\$s2,20	$\$s1 = \$s2 + 20$	Usada para somar constantes
Transferência de dados	load word	lw \$s1,20(\$s2)	$\$s1 = \text{Memória}[\$s2 + 20]$	Dados da memória para o registrador
	store word	sw \$s1,20(\$s2)	$\text{Memória}[\$s2 + 20] = \$s1$	Dados do registrador para a memória
	load half	lh \$s1,20(\$s2)	$\$s1 = \text{Memória}[\$s2 + 20]$	Halfword da memória para registrador
	load half unsigned	lhu \$s1,20(\$s2)	$\$s1 = \text{Memória}[\$s2 + 20]$	Halfword da memória para registrador
	store half	sh \$s1,20(\$s2)	$\text{Memória}[\$s2 + 20] = \$s1$	Halfword de um registrador para memória
	load byte	lb \$s1,20(\$s2)	$\$s1 = \text{Memória}[\$s2 + 20]$	Byte da memória para registrador
	load byte unsigned	lbu \$s1,20(\$s2)	$\$s1 = \text{Memória}[\$s2 + 20]$	Byte da memória para registrador
	store byte	sb \$s1,20(\$s2)	$\text{Memória}[\$s2 + 20] = \$s1$	Byte de um registrador para memória
	load linked word	l1 \$s1,20(\$s2)	$\$s1 = \text{Memória}[\$s2 + 20]$	Carrega word como 1ª metade do swap atômico
	store condition, word	sc \$s1,20(\$s2)	$\text{Memória}[\$s2 + 20] = \$s1; \$s1 = 0 \text{ or } 1$	Armazena word como 2ª metade do swap atômico
Lógica	load upper immed.	lui \$s1,20	$\$s1 = 20 * 2^{16}$	Carrega constante nos 16 bits mais altos
	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Três operadores em registrador; AND bit a bit
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2   \$s3$	Três operadores em registrador; OR bit a bit
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2   \$s3)$	Três operadores em registrador; NOR bit a bit
	and immediate	andi \$s1,\$s2,20	$\$s1 = \$s2 & 20$	AND bit a bit registrador com constante
	or immediate	ori \$s1,\$s2,20	$\$s1 = \$s2   20$	OR bit a bit registrador com constante
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Deslocamento à esquerda por constante
Desvio condicional	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Deslocamento à direita por constante
	branch on equal	beq \$s1,\$s2,25	if( $\$s1 == \$s2$ ) go to PC + 4 + 100	Testa igualdade; desvio relativo ao PC
	branch on not equal	bne \$s1,\$s2,25	if( $\$s1 != \$s2$ ) go to PC + 4 + 100	Testa desigualdade; relativo ao PC
	set on less than	slt \$s1,\$s2,\$s3	if( $\$s2 < \$s3$ ) $\$s1 = 1$ ; else $\$s1 = 0$	Compara menor que; usado com beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	if( $\$s2 < \$s3$ ) $\$s1 = 1$ ; else $\$s1 = 0$	Compara menor que sem sinal
	set less than immediate	slti \$s1,\$s2,20	if( $\$s2 < 20$ ) $\$s1 = 1$ ; else $\$s1 = 0$	Compara menor que constante
	set less than immediate unsigned	sltiu \$s1,\$s2,20	if( $\$s2 < 20$ ) $\$s1 = 1$ ; else $\$s1 = 0$	Compara menor que constante sem sinal
Desvio incondicional	jump	j 2500	go to 10000	Desvia para endereço de destino
	jump register	jr \$ra	go to $\$ra$	Para switch e retorno de procedimento
	jump and link	jal 2500	$\$ra = \text{PC} + 4$ ; go to 10000	Para chamada de procedimento

### FIGURA 2.1 Assembly do MIPS revelado neste capítulo.

Esta informação também pode ser encontrada na Coluna 1 do Guia de Referência do MIPS no final deste livro.

## 2.2. Operações do hardware do computador

Certamente é preciso haver instruções para realizar as operações aritméticas fundamentais.

*Burks, Goldstine e von Neumann, 1947*

Todo computador precisa ser capaz de realizar aritmética. A notação em assembly do MIPS

```
add a, b, c
```

instrui um computador a somar as duas variáveis b e c para colocar sua soma em a.

Essa notação é rígida no sentido de que cada instrução aritmética do MIPS realiza apenas uma operação e sempre precisa ter exatamente três variáveis. Por exemplo, suponha que queiramos colocar a soma das variáveis b, c, d e e na variável a. (Nesta seção, estamos sendo deliberadamente vagos com relação ao que é uma “variável”; na próxima seção, vamos explicar com detalhes.)

Esta sequência de instruções soma as quatro variáveis:

```
add a, b, c # A soma b + c é colocada em a.  
add a, a, d # A soma b + c + d agora está em a.  
add a, a, e # A soma b + c + d + e agora está em a.
```

Portanto, são necessárias três instruções para somar quatro variáveis.

As palavras à direita do símbolo (#) em cada linha acima são *comentários* para o leitor humano, e os computadores os ignoram. Note que, diferentemente de outras linguagens de programação, cada linha desta linguagem pode conter, no máximo, uma instrução. Outra diferença para a linguagem C é que comentários sempre terminam no final da linha.

O número natural de operandos para uma operação como a adição é três: os dois números sendo somados e um local para colocar a soma. Exigir que cada instrução tenha exatamente três operações, nem mais nem menos, está de acordo com a filosofia de manter o hardware simples: o hardware para um número variável de operandos é mais complicado do que o hardware para um número fixo. Essa situação ilustra o primeiro dos quatro princípios básicos de projeto do hardware:

*Princípio de Projeto 1: Simplicidade favorece a regularidade.*

Agora podemos mostrar, nos dois exemplos a seguir, o relacionamento dos programas escritos nas linguagens de programação de mais alto nível com os programas nessa notação mais primitiva.

## Compilando duas instruções de atribuição em C no MIPS

### Exemplo

Este segmento de um programa em C contém as cinco variáveis a, b, c, d e e. Como o Java evoluiu a partir da linguagem C, este exemplo e os próximos funcionam para qualquer uma dessas linguagens de programação de alto nível:

```
a = b + c;  
d = a - e;
```

A tradução de C para as instruções em linguagem assembly do MIPS é realizada pelo *compilador*. Mostre o código do MIPS produzido por um compilador.

### Resposta

Uma instrução MIPS opera sobre dois operandos de origem e coloca o resultado em um operando de destino. Logo, as duas instruções simples anteriores são compiladas diretamente nessas duas instruções em assembly do MIPS:

```
add a, b, c  
sub d, a, e
```

## Compilando uma atribuição em C complexa no MIPS

### Exemplo

Uma instrução um tanto complexa contém as cinco variáveis f, g, h, i e j:

$$f = (g + h) - (i + j);$$

O que um compilador C poderia produzir?

## Resposta

O compilador precisa desmembrar essa instrução em várias instruções assembly, pois somente uma operação é realizada por instrução MIPS. A primeira instrução MIPS calcula a soma de g e h. Temos de colocar o resultado em algum lugar, de modo que o compilador crie uma variável temporária, chamada t0:

```
add t0,g,h # variável temporária t0 contém g + h
```

Embora a próxima operação seja subtrair, precisamos calcular a soma de i e j antes de podermos subtrair. Assim, a segunda instrução coloca a soma de i e j em outra variável temporária criada pelo compilador, chamada t1:

```
add t1,i,j # variável temporária t1 contém i + j
```

Finalmente, a instrução de subtração subtrai a segunda soma da primeira e coloca a diferença na variável f, completando o código compilado:

```
sub f,t0,t1 # f recebe t0 - t1, que é (g + h) - (i + j)
```

## Verifique você mesmo

Para determinada função, que linguagem de programação provavelmente utiliza mais linhas de código? Coloque as três representações a seguir em ordem.

1. Java
2. C
3. Assembly do MIPS

## Detalhamento

Para aumentar a portabilidade, o Java foi idealizado originalmente como um interpretador de software. O conjunto de instruções desse interpretador é chamado *bytecode Java*, que é muito diferente do conjunto de instruções do MIPS. Para chegar a um desempenho próximo ao programa em C equivalente, os sistemas Java de hoje normalmente compilam os bytecodes Java para os conjuntos de instruções nativos, como MIPS. Como essa compilação em geral é feita muito mais tarde do que para programas em C, esses compiladores Java normalmente são denominados compiladores JIT (*Just In Time* — no momento exato). A Seção 2.12 mostra como os JITs são usados mais tarde que os compiladores C no processo de inicialização e a Seção 2.13 mostra as consequências no desempenho de compilar *versus* interpretar programas Java.

## 2.3. Operandos do hardware do computador

Ao contrário dos programas em linguagens de alto nível, os operandos das instruções aritméticas são restritos; precisam ser de um grupo limitado de locais especiais, embutidos diretamente no hardware, chamados *registradores*. Os registradores são primitivas usadas no projeto do hardware que também são visíveis ao programador quando o projeto do computador é concluído, portanto, você pode pensar nos registradores como os “tijolos” na construção do computador. O tamanho de um registrador na arquitetura MIPS é de 32 bits; os grupos de 32 bits ocorrem com tanta frequência que recebem o nome de **word (palavra)** na arquitetura MIPS.

### Word (palavra)

A unidade de acesso natural de um computador, normalmente um grupo de 32 bits; corresponde ao tamanho de um registrador na arquitetura MIPS.

Uma diferença importante entre as variáveis de uma linguagem de programação e os registradores é o número limitado de registradores, normalmente 32 nos computadores atuais, como o MIPS. Assim, continuando em nossa evolução passo a passo da representação simbólica da linguagem MIPS, nesta seção, incluímos a restrição de que cada um dos três operandos das instruções aritméticas do MIPS precisa ser escolhido a partir de um dos 32 registradores de 32 bits.

A razão para o limite dos 32 registradores pode ser encontrada no segundo dos três princípios de projeto básicos da tecnologia de hardware:

*Princípio de Projeto 2: Menor significa mais rápido.*

Uma quantidade muito grande de registradores pode aumentar o tempo do ciclo do clock simplesmente porque os sinais eletrônicos levam mais tempo quando precisam atravessar uma distância maior.

Orientações como “menor significa mais rápido” não são absolutas; 31 registradores podem não ser mais rápidos do que 32. Mesmo assim, a verdade por trás dessas observações faz com que os projetistas de computador as levem a sério. Nesse caso, o projetista precisa equilibrar o desejo dos programas por mais registradores, com o desejo do projetista de manter o ciclo de clock rápido. Outro motivo para não usar mais de 32 é o número de bits que seria necessário no formato da instrução, como demonstra a [Seção 2.5](#).

O [Capítulo 4](#) mostra o papel central que os registradores desempenham na construção do hardware; como veremos neste capítulo, o uso eficaz dos registradores é fundamental para o desempenho do programa.

Embora pudéssemos simplesmente escrever instruções usando números para os registradores, de 0 a 31, a convenção do MIPS é usar nomes com um sinal de cifrão seguido por dois caracteres para representar um registrador. A [Seção 2.8](#) explicará os motivos por trás desses nomes. Por enquanto, usaremos `$s0, $s1...` para os registradores que correspondem às variáveis dos programas em C e Java, e `$t0, $t1...` para os registradores temporários necessários para compilar o programa nas instruções MIPS.

## Compilando uma atribuição em C usando registradores

### Exemplo

É tarefa do compilador associar variáveis do programa aos registradores. Considere, por exemplo, a instrução de atribuição do nosso exemplo anterior:

$$f = (g + h) - (i + j);$$

As variáveis `f`, `g`, `h`, `i` e `j` são associadas aos registradores `$s0, $s1, $s2, $s3` e `$s4`, respectivamente. Qual é o código MIPS compilado?

## Resposta

O programa compilado é muito semelhante ao exemplo anterior, exceto que substituímos as variáveis pelos nomes dos registradores mencionados anteriormente, mais dois registradores temporários, `$t0` e `$t1`, que correspondem às variáveis temporárias de antes:

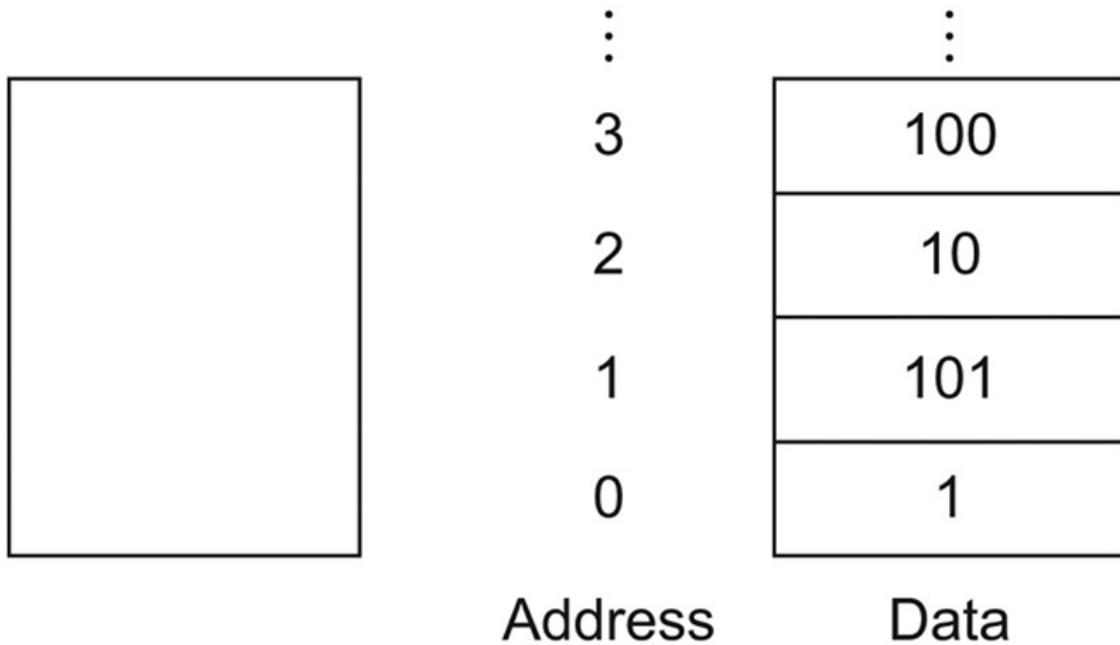
```
add $t0,$s1,$s2 # registrador $t0 contém g + h  
add $t1,$s3,$s4 # registrador $t1 contém i + j  
sub $s0,$t0,$t1 # f recebe $t0 - $t1, que é (g + h) - (i + j)
```

## Operandos em memória

As linguagens de programação possuem variáveis simples, que contêm elementos de dados isolados, como nesses exemplos, mas também possuem estruturas de dados mais complexas — arrays (ou sequências) e estruturas. Essas estruturas de dados complexas podem conter muito mais elementos de dados do que a quantidade de registradores em um computador. Logo, como um computador pode representar e acessar estruturas tão grandes?

Lembre-se dos cinco componentes de um computador, apresentados no [Capítulo 1](#) e desenhados no início deste capítulo. O processador só pode manter uma pequena quantidade de dados nos registradores, mas a memória do computador contém milhões de elementos de dados. Logo, as estruturas de dados (arrays e estruturas) são mantidas na memória.

Conforme explicamos, as operações aritméticas só ocorrem com registradores nas instruções MIPS; assim, o MIPS precisa incluir instruções que transferem dados entre a memória e os registradores. Essas instruções são denominadas **instruções de transferência de dados**. Para acessar uma palavra na memória, a instrução precisa fornecer o **endereço** de memória. A memória é apenas uma sequência grande e unidimensional, com o endereço atuando como índice para esse array, começando de 0. Por exemplo, na [Figura 2.2](#), o endereço do terceiro elemento de dados é 2 e o valor de Memória[2] é 10.



**FIGURA 2.2** Endereços de memória e conteúdo da memória nesses locais.

Se esses elementos fossem palavras, esses endereços estariam incorretos, pois o MIPS, na realidade, usa endereços de bytes, com cada palavra representando quatro bytes. A [Figura 2.3](#) mostra o endereçamento para palavras sequenciais na memória.

## instrução de transferência de dados

Um comando que move dados entre a memória e os registradores.

### endereço

Um valor usado para delinear o local de um elemento de dados específicos dentro de uma sequência da memória.

A instrução de transferência de dados que copia dados da memória para um

registrador tradicionalmente é chamada de *load*. O formato da instrução load é o nome da operação seguido pelo registrador a ser carregado, depois uma constante e o registrador usado para acessar a memória. A soma da parte constante da instrução com o conteúdo do segundo registrador forma o endereço da memória. O nome MIPS real para essa instrução é *lw*, significando *load word* (carregar palavra).

## Compilando uma atribuição quando um operando está na memória

### Exemplo

Vamos supor que *A* seja uma sequência de 100 palavras e que o compilador tenha associado as variáveis *g* e *h* aos registradores *\$s1* e *\$s2*, como antes. Vamos supor também que o endereço inicial da sequência, ou *endereço base*, esteja em *\$s3*. Compile esta instrução de atribuição em C:

```
g = h + A[8];
```

### Resposta

Embora haja uma única operação nessa instrução de atribuição, um dos operandos está na memória, de modo que primeiro precisamos transferir *A[8]* para um registrador. O endereço desse elemento da sequência é a soma da base da sequência *A*, encontrada no registrador *\$s3*, com o número para selecionar o elemento 8. Os dados devem ser colocados em um registrador temporário, para uso na próxima instrução. Com base na Figura 2.2, a primeira instrução compilada é

```
lw $t0,8($s3) # Registrador temporário $t0 recebe A[8]
```

(A seguir, faremos um pequeno ajuste nessa instrução, mas usaremos essa versão simplificada, por enquanto.) A seguinte instrução pode operar sobre o valor em *\$t0* (que é igual a *A[8]*), já que está em um registrador. A instrução precisa somar *h* (contido em *\$s2*) com *A[8]* (*\$t0*) e colocar a soma no

registrador correspondente a g (associado a \$s1):

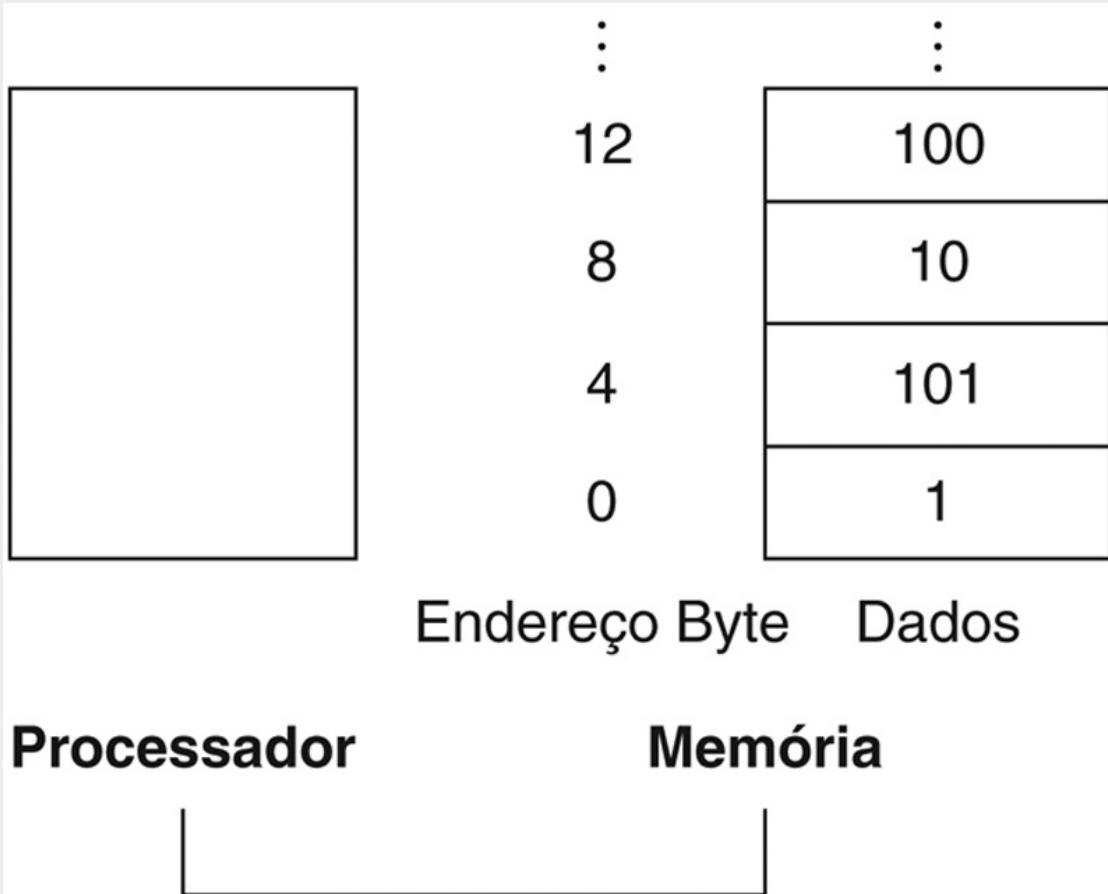
```
add $s1,$s2,$t0 # g = h + A[8]
```

A constante na instrução de transferência de dados (8) é chamada de *offset* e o registrador acrescentado para formar o endereço (\$s3) é chamado de *registraror base*.

## Interface hardware/software

Além de associar variáveis a registradores, o compilador aloca estruturas de dados, como arrays e estruturas, em locais na memória. O compilador pode, então, colocar o endereço inicial apropriado nas instruções de transferência de dados.

Como os *bytes* de 8 bits são úteis em muitos programas, a maioria das arquiteturas atuais endereça bytes individuais. Portanto, o endereço de uma palavra combina os endereços dos 4 bytes dentro da palavra. Logo, os endereços sequenciais das palavras diferem em quatro vezes. Por exemplo, a Figura 2.3 mostra os endereços MIPS reais para a Figura 2.2; o endereço em bytes da terceira palavra é 8.



**FIGURA 2.3** Endereços reais de memória do MIPS e conteúdo da memória para essas palavras.

A mudança de endereços está destacada para comparar com a Figura 2.2. Como o MIPS endereça cada byte, endereços de palavras são múltiplos de 4: existem 4 bytes em uma palavra.

No MIPS, palavras precisam começar em endereços que sejam múltiplos de 4. Esse requisito é denominado **restrição de alinhamento** e muitas arquiteturas o exigem. (O Capítulo 4 explica por que o alinhamento ocasiona transferências de dados mais rápidas.)

### restrição de alinhamento

Um requisito de que os dados estejam alinhados na memória em limites naturais.

Os computadores se dividem entre aqueles que utilizam o endereço do byte mais à esquerda, ou *big end*, como endereço da palavra e os que utilizam o

byte mais à direita, ou *little end*. O MIPS está no campo do *big endian*. Visto que a ordem só importa se você acessar os dados idênticos tanto como uma palavra quanto como quatro bytes, poucos precisam estar cientes da ordenação dos bytes. (O Apêndice A mostra as duas opções para numerar os bytes de uma palavra.)

O endereçamento em bytes também afeta o índice do array. Para obter o endereço em bytes apropriado no código anterior, o *offset a ser somado ao registrador base \$s3 precisa ser  $4 \times 8$ , ou 32*, de modo que o endereço de load selecione  $A[8]$ , e não  $A[8/4]$ . (Veja a armadilha relacionada na Seção 2.18.)

A instrução complementar ao load tradicionalmente é chamada de *store*; ela copia dados de um registrador para a memória. O formato de um store é semelhante ao de um load: o nome da operação, seguido pelo registrador a ser armazenado, depois o offset para selecionar o elemento do array e finalmente o registrador base. Mais uma vez, o endereço MIPS é especificado, em parte, por uma constante e, em parte, pelo conteúdo de um registrador. O nome real no MIPS é *sw*, significando *store word* (armazenar palavra).

## Interface hardware/software

Como os endereços nos loads e stores são números binários, podemos ver por que a DRAM para a memória vem em tamanhos binários, e não em tamanhos decimais. Ou seja, em gibibytes ( $2^{30}$ ) ou tebibytes ( $2^{40}$ ), e não em gigabytes ( $10^9$ ) ou terabytes ( $10^{12}$ ) (Figura 1.1).

## Compilando com load e store

### Exemplo

Suponha que a variável *h* esteja associada ao registrador *\$s2* e o endereço base do array *A* esteja em *\$s3*. Qual é o código assembly do MIPS para a instrução de atribuição em C a seguir?

```
A[12] = h + A[8];
```

### Resposta

Embora haja uma única operação na instrução em C, agora dois dos operandos estão na memória, de modo que precisamos de ainda mais instruções MIPS. As duas primeiras instruções são iguais às do exemplo anterior, exceto que, desta vez, usamos o offset apropriado para o endereçamento do byte na instrução load word, a fim de selecionar A[8], e a instrução add coloca a soma em \$t0:

```
lw $t0,32($s3) # Registrador temporário $t0 recebe A[8]
add $t0,$s2,$t0 # Registrador temporário $t0 recebe h + A[8]
```

A instrução final armazena a soma em A[12], usando 48 ( $4 \times 12$ ) como offset e o registrador \$s3 como registrador base.

```
sw $t0,48($s3) # Armazena h + A[8] de volta em A[12]
```

Load word e store word são as instruções que copiam words entre memória e registradores na arquitetura MIPS. Outras marcas de computadores utilizam outras instruções juntamente com load e store para transferir dados. Uma arquitetura com essas alternativas é a Intel x86, descrita na [Seção 2.16](#).

## Interface hardware/software

Muitos programas possuem mais variáveis do que os computadores possuem registradores. Consequentemente, o compilador tenta manter as variáveis usadas com mais frequência nos registradores e coloca o restante na memória, usando loads e stores para mover variáveis entre os registradores e a memória. O processo de colocar as variáveis menos utilizadas (ou aquelas necessárias mais adiante) na memória é chamado de *spilled registers* (ou *registradores derramados*).

O princípio de hardware relacionando tamanho e velocidade sugere que a memória deve ser mais lenta que os registradores, pois existem menos registradores. Isso realmente acontece; os acessos aos dados são mais rápidos se os dados estiverem nos registradores, ao invés de estarem na memória.

Além do mais, os dados são mais úteis quando em um registrador. Uma instrução aritmética MIPS pode ler dois registradores, operar sobre eles e escrever o resultado. Uma instrução de transferência de dados MIPS só lê um operando ou escreve um operando, sem operar sobre ele.

Assim, os registradores MIPS levam menos tempo para serem acessados e possuem maior vazão do que a memória, tornando os dados nos registradores mais rápidos de acessar e mais simples de usar. O acesso aos registradores também usa menos energia do que o acesso à memória. Para conseguir o melhor desempenho e economizar energia, uma arquitetura de conjunto de instruções precisa ter um número suficiente de registradores, e os compiladores precisam usar os registradores de modo eficaz.

## Constantes ou operandos imediatos

Muitas vezes, um programa usará uma constante em uma operação — por exemplo, ao incrementar um índice a fim de apontar para o próximo elemento de um array. Na verdade, mais da metade das instruções aritméticas do MIPS possuem uma constante como operando quando executam os benchmarks SPEC2006.

Usando apenas as instruções vistas até aqui, teríamos de ler uma constante da memória para utilizá-la. (As constantes teriam de ser colocadas na memória quando o programa fosse carregado.) Por exemplo, para somar a constante 4 ao registrador \$s3, poderíamos usar o código

```
lw $t0, AddrConstant4($s1)    # $t0 = constante 4
add $s3,$s3,$t0                # $s3 = $s3 + $t0 ($t0 == 4)
```

supondo que \$s1 + AddrConstant4 seja o endereço de memória da constante 4.

Uma alternativa que evita a instrução load é oferecer versões das instruções aritméticas em que o operando seja uma constante. Essa instrução add rápida, com uma constante no lugar do operando, é chamada *add imediato* ou *addi*. Para somar 4 ao registrador \$s3, simplesmente escrevemos

```
addi $s3,$s3,4          # $s3 = $s3 + 4
```

Os operandos constantes ocorrem com frequência e, incluindo constantes dentro das instruções aritméticas, as operações são muito mais rápidas e usam menos energia do que se as constantes fossem lidas da memória.

A constante zero tem outro emprego, que é simplificar o conjunto de instruções por oferecer variações úteis. Por exemplo, a operação move é apenas uma instrução de soma na qual cada operando é zero. Portanto, o MIPS dedica o registrador \$zero para ter sempre o valor zero. (Como você deve esperar, ele é o registrador número 0.) O uso da frequência para justificar as inclusões de constantes é outro exemplo da grande ideia de tornar o **caso comum veloz**.



## CASO COMUM VELOZ

### Verifique você mesmo

Dada a importância dos registradores, qual é a taxa de aumento no número de registradores em um chip com o passar do tempo?

1. Muito rápida: eles aumentam tão rapidamente quanto a Lei de Moore, que prevê a duplicação do número de transistores em um chip a cada 18 meses.
2. Muito lenta: como os programas normalmente são distribuídos em linguagem de máquina, existe uma inércia na arquitetura do conjunto de instruções, e, por isso, o número de registradores aumenta apenas quando novos conjuntos de instruções se tornam viáveis.

### Detalhamento

## **Detalhamento**

Embora os registradores MIPS neste livro tenham 32 bits de largura, existe uma versão de 64 bits do conjunto de instruções MIPS, definido com 32 registradores de 64 bits. Para distingui-los, eles são chamados oficialmente de MIPS-32 e MIPS-64. Neste capítulo, usamos um subconjunto do MIPS-32. As Seções 2.16 e 2.18 mostram a diferença muito maior entre o ARMv7 com endereços de 32 bits e o seu sucessor de 64 bits, o ARMv8.

## **Detalhamento**

O endereçamento formado pelo registrador-base mais o offset do MIPS é uma combinação excelente para as estruturas e os arrays, pois o registrador pode apontar para o início da estrutura, e o offset pode selecionar o elemento desejado. Veremos esse exemplo na Seção 2.13.

## **Detalhamento**

O registrador nas instruções de transferência de dados foi criado originalmente para manter o índice do array com o offset utilizado para o endereço inicial do array. Assim, o registrador-base também é chamado *registraror índice*. As memórias de hoje são muito maiores e o modelo de software para alocação de dados é mais sofisticado, de modo que o endereço-base do array normalmente é passado em um registrador, pois não caberá no offset, conforme veremos.

## **Detalhamento**

Como o MIPS admite constantes negativas, a subtração imediata não é necessária no MIPS.

## **2.4. Números com sinal e sem sinal**

Primeiro, vamos revisar rapidamente como um computador representa números. Os humanos são ensinados a pensar na base 10, mas os números podem ser representados em qualquer base. Por exemplo, 123 base 10 = 1111011 base 2.

Os números são mantidos no hardware do computador como uma série de sinais eletrônicos altos e baixos, e por isso são considerados números de base 2.

(Assim como os números de base 10 são chamados números *decimais*, os números de base 2 são chamados números *binários*.)

Um único dígito de um número binário, portanto, é o “átomo” da computação, pois toda a informação é composta de **dígitos binários** ou *bits*. Esse bloco de montagem fundamental pode assumir dois valores, que podem ser imaginados como várias alternativas: alto ou baixo, ligado ou desligado, verdadeiro ou falso ou 1 ou 0.

## dígito binário

Também chamado **bit**. Um dos dois números na base 2 (0 ou 1), que são os componentes básicos da informação.

Generalizando, em qualquer base numérica, o valor do  $i$ -ésimo dígito  $d$  é

$$d \times \text{Base}^i$$

em que  $i$  começa com 0 e aumenta da direita para a esquerda. Isso leva a um modo óbvio de numerar os bits na word: basta usar a potência da base para esse bit. Subscritamos os números decimais com *dec* e os números binários com *bin*. Por exemplo,

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

Representa

$$\begin{aligned} & (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{\text{dec}} \\ & = (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{\text{dec}} \\ & = 8 + 0 + 2 + 1_{\text{dec}} \\ & = 11_{\text{dec}} \end{aligned}$$

Logo, os bits são numerados com 0, 1, 2, 3, ... da *direita para a esquerda* em uma palavra. O desenho a seguir mostra a numeração dos bits dentro de uma

word MIPS e o posicionamento do número  $1011_{\text{bin}}$ :

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	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1		

(32 bits wide)

Como as palavras são desenhadas vertical e horizontalmente, esquerda e direita podem não ser termos muito claros. Logo, o termo **bit menos significativo** é usado para se referir ao bit mais à direita (bit 0, no exemplo anterior) e **bit mais significativo** para o bit mais à esquerda (bit 31).

## bit menos significativo

O bit mais à direita em uma palavra MIPS.

## bit mais significativo

O bit mais à esquerda em uma palavra MIPS.

Cada palavra no MIPS possui 32 bits de largura, de modo que podemos representar  $2^{32}$  padrões diferentes de 32 bits. É natural deixar que essas representações mostrem os números de 0 a  $2^{32} - 1$  ( $4.294.967.295_{\text{dec}}$ ):

0000	0000	0000	0000	0000	0000	0000	0000	0000	$0_{\text{bin}} = 0_{\text{dec}}$
0000	0000	0000	0000	0000	0000	0000	0001	$1_{\text{bin}} = 1_{\text{dec}}$	
0000	0000	0000	0000	0000	0000	0000	0010	$2_{\text{bin}} = 2_{\text{dec}}$	
...							...		
1111	1111	1111	1111	1111	1111	1111	1101	$4.294.967.293_{\text{dec}}$	
1111	1111	1111	1111	1111	1111	1111	1110	$4.294.967.294_{\text{dec}}$	
1111	1111	1111	1111	1111	1111	1111	1111	$4.294.967.295_{\text{dec}}$	

Ou seja, os números binários de 32 bits podem ser representados em termos do valor do bit vezes uma potência de 2 (aqui,  $x_i$  significa o  $i$ -ésimo bit de  $x$ ):

$$(x_{31} \times 2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \dots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Por motivos que veremos em breve, esses números positivos são denominados números sem sinal.

## Interface hardware/software

A base 2 não é natural para os seres humanos; temos 10 dedos e, portanto, a base 10 é natural. Por que os computadores não usaram decimal? Na verdade, o primeiro computador comercial *oferecia* aritmética decimal. O problema foi que o computador ainda usava sinais do tipo ligado e desligado, de modo que um dígito decimal era simplesmente representado por vários dígitos binários. O sistema decimal mostrou ser tão ineficaz que os computadores subsequentes passaram a usar apenas binário, convertendo para a base 10 apenas para eventos de entrada/saída relativamente pouco frequentes.

Lembre-se de que os padrões de bits binários que acabamos de mostrar simplesmente *representam* os números. Os números, na realidade, possuem uma quantidade infinita de dígitos, com quase todos sendo 0, exceto por alguns dos dígitos mais à direita. Só que, normalmente, não mostramos os 0s à esquerda.

O hardware pode ser projetado para somar, subtrair, multiplicar e dividir esses padrões de bits. Se o número que é o resultado correto de tais operações não puder ser representado por esses bits de hardware mais à direita, diz-se que houve um *overflow* (ou *estouro*). Fica a critério da linguagem de programação, do sistema operacional e do programa determinar o que fazer quando isso ocorre.

Os programas de computador calculam números positivos e negativos, de modo que precisamos de uma representação que faça a distinção entre o positivo e o negativo. A solução mais óbvia é acrescentar um sinal separado, que convenientemente possa ser representado em um único bit; o nome dessa representação é *sinal e magnitude*.

Infelizmente, a representação com sinal e magnitude possui várias desvantagens. Primeiro, não é óbvio onde colocar o bit de sinal. À direita? À esquerda? Os primeiros computadores tentaram ambos. Segundo, os somadores de sinal e magnitude podem precisar de uma etapa extra para definir o sinal, pois não podemos saber, com antecedência, qual será o sinal correto. Finalmente, um bit de sinal separado significa que a representação com sinal e magnitude possui

um zero positivo e um zero negativo, o que pode ocasionar problemas para os programadores desatentos. Como resultado desses problemas, a representação com sinal e magnitude logo foi abandonada.

Em busca de uma alternativa mais atraente, levantou-se a questão com relação a qual seria o resultado, para números sem sinal, se tentássemos subtrair um número grande de um número pequeno. A resposta é que ele tentaria pegar emprestado de uma sequência de 0s à esquerda, de modo que o resultado seria uma sequência de 1s à esquerda.

Como não havia uma alternativa melhor óbvia, a solução final foi escolher a representação que tornasse o hardware simples: 0s iniciais significa positivo e 1s iniciais significa negativo. Essa convenção para representar os números binários com sinal é chamada representação por *complemento de dois*:

0000 0000 0000 0000 0000 0000 0000 0000 <sub>bin</sub>	= 0 <sub>dec</sub>
0000 0000 0000 0000 0000 0000 0000 0001 <sub>bin</sub>	= 1 <sub>dec</sub>
0000 0000 0000 0000 0000 0000 0000 0010 <sub>bin</sub>	= 2 <sub>dec</sub>
...	...
0111 1111 1111 1111 1111 1111 1111 1101 <sub>bin</sub>	= 2.147.483.645 <sub>dec</sub>
0111 1111 1111 1111 1111 1111 1111 1110 <sub>bin</sub>	= 2.147.483.646 <sub>dec</sub>
0111 1111 1111 1111 1111 1111 1111 1111 <sub>bin</sub>	= 2.147.483.647 <sub>dec</sub>
1000 0000 0000 0000 0000 0000 0000 0000 <sub>bin</sub>	= -2.147.483.648 <sub>dec</sub>
1000 0000 0000 0000 0000 0000 0000 0001 <sub>bin</sub>	= -2.147.483.647 <sub>dec</sub>
1000 0000 0000 0000 0000 0000 0000 0010 <sub>bin</sub>	= -2.147.483.646 <sub>dec</sub>
...	...
1111 1111 1111 1111 1111 1111 1111 1101 <sub>bin</sub>	= -3 <sub>dec</sub>
1111 1111 1111 1111 1111 1111 1111 1110 <sub>bin</sub>	= -2 <sub>dec</sub>
1111 1111 1111 1111 1111 1111 1111 1111 <sub>bin</sub>	= -1 <sub>dec</sub>

A metade positiva dos números, de 0 a 2.147.483.647<sub>dec</sub> ( $2^{31} - 1$ ), utiliza a mesma representação de antes. O padrão de bits seguinte (1000 ... 0000<sub>bin</sub>) representa o número mais negativo -2.147.483.648<sub>dec</sub> ( $-2^{31}$ ). Ele é seguido por um conjunto decrescente de números negativos: -2.147.483.647<sub>dec</sub> (1000 ... 0001<sub>bin</sub>) até -1<sub>dec</sub> (1111 ... 1111<sub>bin</sub>).

A representação em complemento de dois possui um número negativo, -2.147.483.648<sub>dec</sub>, que não possui um número positivo correspondente. Este desequilíbrio era uma preocupação para o programador desatento, mas a

representação com sinal e magnitude gerava problemas para o programador e para o projetista do hardware. Consequentemente, todo computador, hoje em dia, utiliza a representação de números binários por complemento de dois para os números com sinal.

A representação por complemento de dois tem a vantagem de que todos os números negativos possuem 1 no bit mais significativo. Consequentemente, o hardware só precisa testar esse bit para ver se um número é positivo ou negativo (com 0 considerado positivo). Esse bit normalmente é denominado *bit de sinal*. Reconhecendo o papel do bit de sinal, podemos representar números positivos e negativos de 32 bits em termos do valor do bit vezes uma potência de 2:

$$(x_{31} \times -2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \dots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

O bit de sinal é multiplicado por  $-2^{31}$  e o restante dos bits é multiplicado pelas versões positivas de seus respectivos valores de base.

## Conversão de binário para decimal

### Exemplo

Qual é o valor decimal deste número em complemento de dois com 32 bits?

1111 1111 1111 1111 1111 1111 1111 1100<sub>bin</sub>

### Resposta

Substituindo os valores dos bits do número na fórmula anterior:

$$\begin{aligned} & (1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \dots + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) \\ & = -2^{31} + 2^{30} + 2^{29} + \dots + 2^2 + 0 + 0 \\ & = -2.147.483.648_{\text{dec}} + 2.147.483.644_{\text{dec}} \\ & = -4_{\text{dec}} \end{aligned}$$

Logo, veremos um atalho para simplificar a conversão de negativo para positivo.

Assim como uma operação com números sem sinal pode ocasionar overflow na capacidade do hardware de representar o resultado, uma operação com números em complemento de dois também pode. O overflow ocorre quando o bit mais à esquerda da representação binária do hardware não é igual ao número infinito de dígitos à esquerda (o bit de sinal está incorreto): 0 à esquerda do padrão de bits quando o número é negativo ou 1 quando o número é positivo.

## Interface hardware/software

Com sinal e sem sinal se aplica a loads e também à aritmética. A *função* de um load com sinal é copiar o sinal repetidamente para preencher o restante do registrador — chamado *extensão do sinal* —, mas sua *finalidade* é colocar uma representação correta do número dentro desse registrador. Os loads sem sinal simplesmente preenchem com 0s à esquerda dos dados, pois o número representado pelo padrão de bits é sem sinal.

Ao carregar uma word de 32 bits em um registrador de 32 bits, o ponto é discutível; loads com sinal e sem sinal são idênticos. O MIPS oferece dois tipos de loads de byte: *load byte* (1b) trata o byte como um número com sinal e, assim, estende por sinal para preencher os 24 bits mais à esquerda do registrador, enquanto *load byte unsigned* (1bu) trabalha com inteiros sem sinal. Como os programas em C quase sempre usam bytes para representar caracteres, em vez de considerar bytes como inteiros com sinal muito curtos, 1bu é usada de modo praticamente exclusivo para os loads de byte.

## Interface hardware/software

Diferente dos números discutidos anteriormente, os endereços de memória naturalmente começam com 0 e continuam até o maior endereço. Em outras palavras, endereços negativos não fazem sentido. Assim, os programas desejam lidar às vezes com números que podem ser positivos ou negativos e às vezes com números que só podem ser positivos. Algumas linguagens de programação refletem essa distinção. A linguagem C, por exemplo, chama os primeiros de *integers* ou inteiros (declarados como `int` no programa), e os últimos de *unsigned integers* ou inteiros sem sinal (`unsigned int`). Alguns

guias de estilo C recomendam ainda declarar os primeiros como `signed int`, para deixar a distinção clara.

Vamos examinar dois atalhos úteis quando trabalhamos com os números em complemento de dois. O primeiro atalho é um modo rápido de negar um número binário no complemento de dois. Basta inverter cada 0 para 1 e cada 1 para 0, depois somar um ao resultado. Este atalho é baseado na observação de que a soma de um número e sua representação invertida precisa ser  $111 \dots 111_{\text{bin}}$ , que representa  $-1$ . Como  $x + \bar{x} = -1$ , portanto,  $x + \bar{x} + 1 = 0$  ou  $\bar{x} + 1 = -x$ . (Usamos a notação  $\bar{x}$  para significar inverter cada bit em  $x$  de 0 para 1 e vice-versa.)

## Atalho para negação

### Exemplo

Negue  $2_{\text{dec}}$  e depois verifique o resultado negando  $-2_{\text{dec}}$ .

### Resposta

$$2_{\text{dec}} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{bin}}$$

Negando esse número, invertendo os bits e somando um,

$$\begin{array}{r} 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{\text{bin}} \\ + \qquad 1_{\text{bin}} \\ \hline = \qquad 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{bin}} \\ = \qquad -2_{\text{dec}} \end{array}$$

Na outra direção,

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

primeiro é invertido e depois incrementado:

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{bin}} \\
 + \hspace{10em} 1_{\text{bin}} \\
 \hline
 = \hspace{1em} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{bin}} \\
 = \hspace{1em} 2_{\text{dec}}
 \end{array}$$

O próximo atalho nos diz como converter um número binário representado em  $n$  bits para um número representado com mais de  $n$  bits. Por exemplo, o campo imediato nas instruções *load*, *store*, *branch*, *add* e *set on less than* contém um número de 16 bits em complemento de dois, representando de  $-32.768_{\text{dec}}$  ( $-2^{15}$ ) a  $32.767_{\text{dec}}$  ( $2^{15} - 1$ ). Para somar o campo imediato a um registrador de 32 bits, o computador precisa converter esse número de 16 bits para o seu equivalente em 32 bits. O atalho é pegar o bit mais significativo da menor quantidade — o bit de sinal — e replicá-lo para preencher os novos bits na quantidade maior. Os bits antigos são simplesmente copiados para a parte da direita da nova word. Esse atalho normalmente é chamado de *extensão de sinal*.

## Atalho para extensão de sinal

## Exemplo

Converta as versões binárias de 16 bits de  $2_{dec}$  e  $-2_{dec}$  para números binários de 32 bits.

## Resposta

A versão binária de 16 bits do número 2 é

$$0000\ 0000\ 0000\ 0010_{\text{bin}} = 2_{\text{dec}}$$

Ele é convertido para um número de 32 bits criando-se 16 cópias do valor

do bit mais significativo (0) e colocando-as na metade esquerda da word. A metade direita recebe o valor antigo:

0000 0000 0000 0000 0000 0000 0000 0010<sub>bin</sub> = 2<sub>dec</sub>

Vamos negar a versão de 16 bits de 2 usando o atalho anterior. Assim,

0000 0000 0000 0010<sub>bin</sub>

torna-se

$$\begin{array}{r} 1111 \ 1111 \ 1111 \ 1101_{\text{bin}} \\ + \qquad \qquad \qquad 1_{\text{bin}} \\ \hline = \ 1111 \ 1111 \ 1111 \ 1110_{\text{bin}} \end{array}$$

Criar uma versão de 32 bits do número negativo significa copiar o bit de sinal 16 vezes e colocá-lo à esquerda:

1111 1111 1111 1111 1111 1111 1111 1110<sub>bin</sub> = -2<sub>dec</sub>

Esse truque funciona porque os números positivos em complemento de dois realmente possuem uma quantidade infinita de 0s à esquerda e os que são negativos em complemento de dois possuem uma quantidade infinita de 1s. O padrão binário que representa um número esconde os bits iniciais para caber na largura do hardware; a extensão do sinal simplesmente restaura alguns deles.

## Resumo

O ponto principal desta seção é que precisamos representar inteiros positivos e negativos dentro de uma palavra do computador e, embora existam prós e contras a qualquer opção, a escolha predominante desde 1965 tem sido o complemento de dois.

## Detalhamento

Para números decimais sem sinal, usamos “—” para representar negativo, pois não existem limites para o tamanho de um número decimal. Dado um tamanho de word fixo, strings com bits binários e hexadecimais (Figura 2.4) podem codificar o sinal; logo, normalmente não usamos “+” ou “-” com notação binária ou hexadecimal.

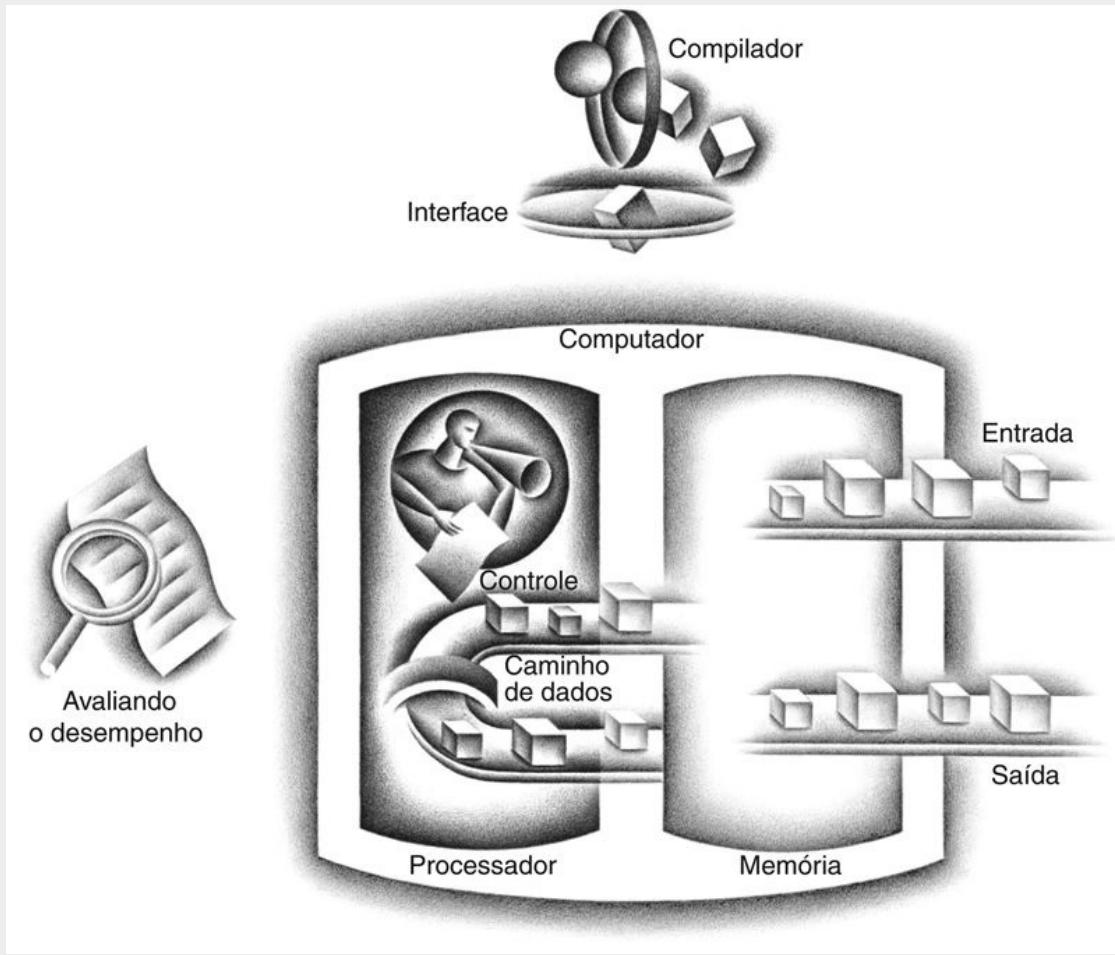
Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal
0 <sub>hexa</sub>	0000 <sub>bin</sub>	4 <sub>hexa</sub>	0100 <sub>bin</sub>	8 <sub>hexa</sub>	1000 <sub>bin</sub>	c <sub>hexa</sub>	1100 <sub>bin</sub>
1 <sub>hexa</sub>	0001 <sub>bin</sub>	5 <sub>hexa</sub>	0101 <sub>bin</sub>	9 <sub>hexa</sub>	1001 <sub>bin</sub>	d <sub>hexa</sub>	1101 <sub>bin</sub>
2 <sub>hexa</sub>	0010 <sub>bin</sub>	6 <sub>hexa</sub>	0110 <sub>bin</sub>	a <sub>hexa</sub>	1010 <sub>bin</sub>	e <sub>hexa</sub>	1110 <sub>bin</sub>
3 <sub>hexa</sub>	0011 <sub>bin</sub>	7 <sub>hexa</sub>	0111 <sub>bin</sub>	b <sub>hexa</sub>	1011 <sub>bin</sub>	f <sub>hexa</sub>	1111 <sub>bin</sub>

**FIGURA 2.4** A tabela de conversão hexadecimal-binário.

Basta substituir um dígito hexadecimal pelos quatro dígitos binários correspondentes e vice-versa. Se o tamanho do número binário não for um múltiplo de 4, prossiga da direita para a esquerda.

## Verifique você mesmo

Qual é o valor decimal deste número de 64 bits em complemento de dois?



1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111  
 1111 1000<sub>bin</sub>

- 1) -4<sub>dec</sub>
- 2) -8<sub>dec</sub>
- 3) -16<sub>dec</sub>
- 4) 18.446.744.073.709.551.609<sub>dec</sub>

## Detalhamento

O complemento de dois recebe esse nome em decorrência da regra de que a soma sem sinal de um número de  $n$  bits e seu negativo de  $n$  bits é  $2^n$ ; logo, o

complemento ou a negação de um número em complemento de dois  $x$  é  $2^n - x$ .

Uma terceira representação alternativa para o complemento de dois, de sinal e magnitude é chamada **complemento de um**. O negativo de um complemento de um é encontrado invertendo-se cada bit, de 0 para 1 e de 1 para 0 ou  $x^-$ . Essa relação ajuda a explicar seu nome, pois o complemento de  $x$  é  $2^n - x - 1$ . Essa também foi uma tentativa de ser uma solução melhor do que a técnica de sinal e magnitude, e vários computadores científicos utilizaram a notação. Essa representação é semelhante ao complemento de dois, exceto que também possui dois 0s:  $00\dots00_{\text{bin}}$  é o 0 positivo, e  $11\dots11_{\text{bin}}$  é o 0 negativo. O maior número negativo  $10\dots000_{\text{bin}}$  representa  $-2.147.483.647_{\text{dec}}$  e, por isso, os positivos e negativos são balanceados. Os que aderiram ao complemento de um precisaram de uma etapa extra para subtrair um número e, por isso, o complemento de dois domina hoje.

## complemento de um

Uma notação que representa o valor mais negativo por  $10\dots000_{\text{bin}}$  e o valor mais positivo por  $01\dots11_{\text{bin}}$ , deixando um número igual de negativos e positivos, mas terminando com dois zeros, um positivo ( $00\dots00_{\text{bin}}$ ) e um negativo ( $11\dots11_{\text{bin}}$ ). O termo também é usado para significar a inversão de cada bit em um padrão: 0 para 1 e 1 para 0.

Uma notação final, que veremos quando tratarmos de ponto flutuante no Capítulo 3, é representar o valor mais negativo por  $00\dots000_{\text{bin}}$  e o valor mais positivo por  $11\dots11_{\text{bin}}$ , com 0 normalmente tendo o valor  $10\dots00_{\text{bin}}$ . Isso é chamado de **notação deslocada** (*biased notation*), pois desloca o número de modo que o número mais o deslocamento tenha uma representação não negativa.

## notação deslocada

Uma notação que representa o valor mais negativo por  $00\dots000_{\text{bin}}$  e o valor mais positivo por  $11\dots11_{\text{bin}}$ , com 0 normalmente tendo o valor  $10\dots00_{\text{bin}}$ , deslocando assim o número, de modo que o número mais o deslocamento têm uma representação não negativa.

## 2.5. Representando instruções no computador

Agora, estamos prontos para explicar a diferença entre o modo como os humanos instruem os computadores e como os computadores veem as instruções.

As instruções são mantidas no computador como uma série de sinais eletrônicos altos e baixos e podem ser representadas como números. Na verdade, cada parte da instrução pode ser considerada um número individual e a colocação desses números lado a lado forma a instrução.

Como os registradores são referenciados por quase todas as instruções, é preciso haver uma convenção para mapear nomes de registrador em números. Na linguagem assembly do MIPS, os registradores \$s0 a \$s7 são mapeados nos registradores de 16 a 23 e os registradores \$t0 a \$t7 são mapeados nos registradores de 8 a 15. Logo, \$s0 significa o registrador 16, \$s1 significa o registrador 17, \$s2 significa o registrador 18, ..., \$t0 significa o registrador 8, \$t1 significa o registrador 9, e assim por diante. Nas próximas seções, descreveremos a convenção para o restante dos 32 registradores.

### Traduzindo uma instrução assembly MIPS para uma instrução de máquina

#### Exemplo

Realizaremos a próxima etapa no refinamento da linguagem do MIPS como um exemplo. Mostraremos a versão da linguagem real do MIPS para a instrução representada simbolicamente por

```
add$t0, $s1, $s2
```

primeiro como uma combinação dos números decimais e depois dos números binários.

#### Resposta

A representação decimal é:

0	17	18	8	0	32
---	----	----	---	---	----

Cada um desses segmentos de uma instrução é chamado de *campo*. O primeiro e o último campos (contendo 0 e 32, nesse caso) combinados dizem ao computador MIPS que essa instrução realiza soma. O segundo campo indica o número do registrador que é o primeiro operando de origem da operação de soma ( $17 = \$s1$ ) e o terceiro campo indica o outro operando fonte para a soma ( $18 = \$s2$ ). O quarto campo contém o número do registrador que deverá receber a soma ( $8 = \$t0$ ). O quinto campo não é utilizado nessa instrução, de modo que é definido como 0. Assim, a instrução soma o registrador  $\$s1$  ao registrador  $\$s2$  e coloca a soma no registrador  $\$t0$ .

Essa instrução também pode ser representada com campos em números binários, em vez de decimal:

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Esse layout da instrução é chamado **formato de instrução**. Como você pode ver pela contagem do número de bits, essa instrução MIPS ocupa exatamente 32 bits — o mesmo tamanho da palavra de dados. Acompanhando nosso princípio de projeto, de que a simplicidade favorece a regularidade, todas as instruções MIPS possuem 32 bits de extensão.

## formato de instrução

Uma forma de representação de uma instrução, composta de campos de números binários.

Para distinguir do assembly, chamamos a versão numérica das instruções de **linguagem de máquina**, e a sequência dessas instruções é o *código de máquina*.

## linguagem de máquina

Representação binária utilizada para a comunicação dentro de um sistema computacional.

Pode parecer que agora você estará lendo e escrevendo sequências longas e cansativas de números binários. Evitamos esse tédio usando uma base maior do que a binária, que pode ser convertida com facilidade para binária. Como quase todos os tamanhos de dados no computador são múltiplos de 4, os números **hexadecimais** (base 16) são muito comuns. Como a base 16 é uma potência de 2, podemos converter facilmente substituindo cada grupo de quatro dígitos binários por um único dígito hexadecimal e vice-versa. A [Figura 2.4](#) converte hexadecimal para binário e vice-versa.

## hexadecimal

Números na base 16.

Visto que frequentemente lidamos com bases numéricas diferentes, para evitar confusão, vamos anexar em subscrito *dec* aos números decimais, *bin* aos números binários e *hex* aos números hexadecimais. (Se não houver um subscrito, a base padrão é 10.) A propósito, C e Java utilizam a notação `0xnnnn` para os números hexadecimais.

## Binário para hexadecimal e vice-versa

### Exemplo

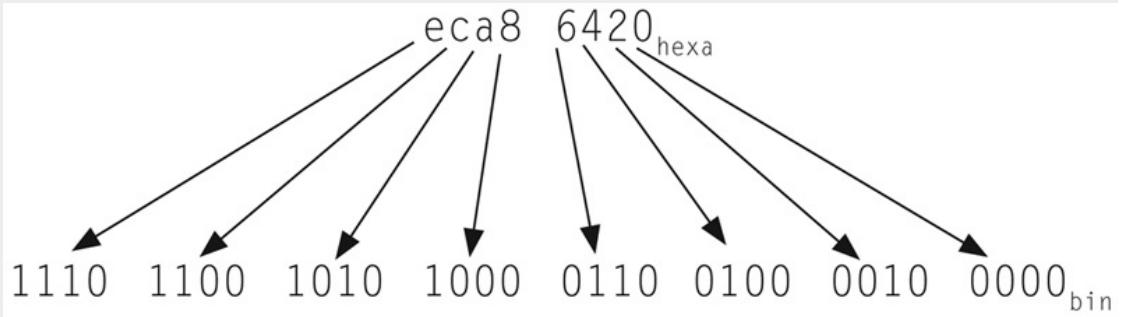
Converta os seguintes números hexadecimais e binários para a outra base:

eca8 6420<sub>hexa</sub>

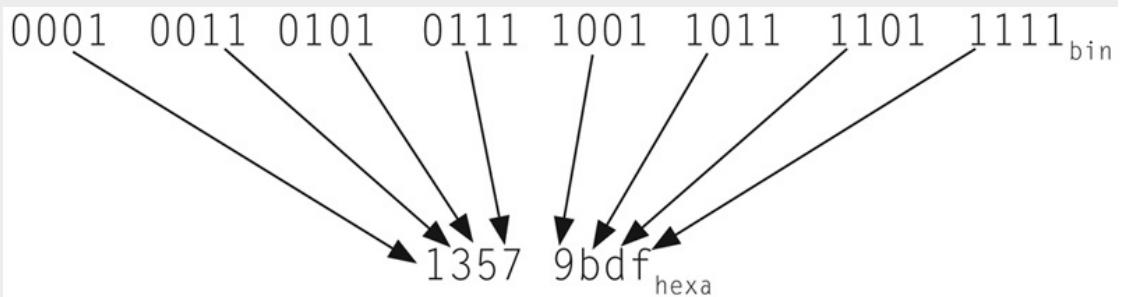
0001 0011 0101 0111 1001 1011 1101 1111<sub>bin</sub>

### Resposta

Usando a Figura 2.4, temos a solução ao olhar na tabela em uma direção:



E depois na outra direção:



## Campos do MIPS

Os campos do MIPS recebem nomes para facilitar seu tratamento:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Aqui está o significado de cada nome nos campos das instruções MIPS:

- **op**: operação básica da instrução, tradicionalmente chamado de **opcode**.
- **rs**: o primeiro registrador do operando fonte.
- **rt**: o segundo registrador do operando fonte.
- **rd**: o registrador do operando de destino. Ele recebe o resultado da operação.
- **shamt**: “Shift amount” (quantidade de deslocamento). (A [Seção 2.6](#) explica as instruções de shift e esse termo; ele não será usado até lá, e, por isso, o campo contém zero nesta seção.)
- **funct**: função. Esse campo, normalmente chamado *código de função*,

seleciona a variante específica da operação no campo op.

## opcode

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

Existe um problema quando uma instrução precisa de campos maiores do que aqueles mostrados. Por exemplo, a instrução load word precisa especificar dois registradores e uma constante. Se o endereço tivesse de usar um dos campos de 5 bits no formato anterior, a constante dentro da instrução load word seria limitada a apenas  $2^5$  ou 32. Esta constante é utilizada para selecionar elementos dos arrays ou estruturas de dados e normalmente precisa ser muito maior do que 32. Esse campo de 5 bits é muito pequeno para realizar algo útil.

Logo, temos um conflito entre o desejo de manter todas as instruções com o mesmo tamanho e o desejo de ter um formato de instrução único. Isso nos leva ao último princípio de projeto de hardware:

*Princípio de Projeto 3: Um bom projeto exige bons compromissos.*

O compromisso escolhido pelos projetistas do MIPS é manter todas as instruções com o mesmo tamanho, exigindo, assim, diferentes tipos de formatos para diferentes tipos de instruções. Por exemplo, o formato anterior é chamado de *tipo-R* (de registrador) ou *formato R*. Um segundo tipo de formato de instrução é chamado *tipo I* (de imediato) ou *formato I*, e é utilizado pelas instruções imediatas e de transferência de dados. Os campos do formato I são:

op	rs	rt	constante ou endereço
6 bits	5 bits	5 bits	16 bits

O endereço de 16 bits significa que uma instrução load word pode carregar qualquer palavra dentro de uma região de  $\pm 2^{15}$  ou 32.768 bytes ( $\pm 2^{13}$  ou 8.192 words) do endereço no registrador base rs. De modo semelhante, a soma imediata é limitada a constantes que não sejam maiores do que  $\pm 2^{15}$ . Vemos que o uso de mais de 32 registradores seria difícil nesse formato, pois os campos rs e rt precisariam cada um de outro bit, tornando mais difícil encaixar tudo em uma palavra.

Vejamos a instrução load word apresentada anteriormente:

```
lw      $t0,32($s3)      # Registrador temporário $t0 recebe A[8]
```

Aqui, 19 (para `$s3`) é colocado no campo `rs`, 8 (para `$t0`) é colocado no campo `rt`, e 32 é colocado no campo de endereço. Observe que o significado do campo `rt` mudou para essa instrução: em uma instrução load word, o campo `rt` especifica o registrador de *destino*, que recebe o resultado do load.

Embora o uso de vários formatos complique o hardware, podemos reduzir a complexidade mantendo os formatos semelhantes. Por exemplo, os três primeiros campos nos formatos de tipo R e tipo I possuem o mesmo tamanho e têm os mesmos nomes; o tamanho do quarto campo no tipo I é igual à soma dos tamanhos dos três últimos campos do tipo R.

Caso você esteja curioso, os formatos são diferenciados pelos valores no primeiro campo: cada formato recebe um conjunto distinto de valores no primeiro campo (`op`), de modo que o hardware sabe se deve tratar a última metade da instrução como três campos (tipo R) ou como um único campo (tipo I). A [Figura 2.5](#) mostra os números utilizados em cada campo para as instruções MIPS descritas aqui.

Instrução	Formato	op	rs	rt	rd	shamt	funct	endereço
add	R	0	reg	reg	reg	0	$32_{dec}$	n.a.
sub (subtract)	R	0	reg	reg	reg	0	$34_{dec}$	n.a.
add immediate	I	$8_{dec}$	reg	reg	n.a.	n.a.	n.a.	constante
lw (load word)	I	$35_{dec}$	reg	reg	n.a.	n.a.	n.a.	endereço
sw (store word)	I	$43_{dec}$	reg	reg	n.a.	n.a.	n.a.	endereço

**FIGURA 2.5 Codificação de instruções MIPS.**

Na tabela, “reg” significa um número de registrador entre 0 e 31, “endereço” significa um endereço de 16 bits, e “n.a.” (não se aplica) significa que esse campo não aparece nesse formato.

Observe que as instruções `add` e `sub` têm o mesmo valor no campo `op`; o hardware usa o campo `funct` para decidir sobre a variante da operação: somar (32) ou subtrair (34).

## Traduzindo do assembly MIPS para a linguagem de máquina

### Exemplo

Agora, já podemos usar um exemplo completo, daquilo que o programador escreve até o que o computador executa. Se \$t1 possui a base do array A e \$s2 corresponde a h, então a instrução de atribuição

```
A[300] = h + A[300];
```

é compilada para

```
lw $t0,1200($t1) # Reg. temporário $t0 recebe A[300]
add $t0,$s2,$t0 # Reg. temporário $t0 recebe h + A[300]
sw $t0,1200($t1) # Armazena h + A[300] de volta para A[300]
```

Qual o código em linguagem de máquina MIPS para essas três instruções?

## Resposta

Por conveniência, primeiro vamos representar as instruções em linguagem de máquina usando os números decimais. Pela Figura 2.5, podemos determinar as três instruções em linguagem de máquina:

op	rs	rt	rd	endereço/shamt	funct
35	9	8		1200	
0	18	8	8	0	32
43	9	8		1200	

A instrução lw é identificada por 35 (Figura 2.5) no primeiro campo (op). O registrador base 9 (\$t1) é especificado no segundo campo (rs), e o registrador de destino 8 (\$t0) é especificado no terceiro campo (rt). O offset para selecionar A[300] ( $1200 = 300 \times 4$ ) aparece no campo final (endereço).

A instrução add, que vem em seguida, é especificada com 0 no primeiro campo (op) e 32 no último campo (funct). Os três operandos de registrador (18, 8 e 8) aparecem no segundo, no terceiro e no quarto campos e correspondem a \$s2, \$t0 e \$t0.

A instrução sw é identificada com 43 no primeiro campo. O restante dessa última instrução é idêntico à instrução lw.

Como  $1200_{dec} = 0000\ 0100\ 1011\ 0000_{bin}$ , o equivalente binário ao formato

decimal é o seguinte:

10 <b>0</b> 11	01001	01000	0000 0100 1011 0000			
000000	10010	01000	01000		00000	100000
10 <b>1</b> 011	01001	01000	0000 0100 1011 0000			

Observe a semelhança das representações binárias da primeira e última instruções. A única diferença está no terceiro bit a partir da esquerda, que está destacado.

## Interface hardware/software

O desejo de manter todas as instruções com o mesmo tamanho está em conflito com o desejo de ter o máximo de registradores possível. Qualquer aumento no número de registradores usa, pelo menos, um bit a mais em cada campo de registrador do formato da instrução. Dadas essas restrições e o princípio de projeto de que menor é mais rápido, a maior parte dos conjuntos de instruções hoje possui 16 ou 32 registradores de uso geral.

A [Figura 2.6](#) resume as partes do assembly do MIPS descritas nesta seção. Como veremos no [Capítulo 4](#), a semelhança das representações binárias de instruções relacionadas simplifica o projeto do hardware. Essas instruções são outro exemplo da regularidade da arquitetura MIPS.

Linguagem de máquina do MIPS							
Nome	Formato	Exemplo					Comentários
add	R	0	18	19	17	0	32
sub	R	0	18	19	17	0	34
addi	I	8	18	17	100		
lw	I	35	18	17	100		
sw	I	43	18	17	100		
Tamanho do campo		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Formato R	R	op	rs	rt	rd	shamt	funct
Formato I	I	op	rs	rt	Endereço		

**FIGURA 2.6** Arquitetura MIPS revelada até a [Seção 2.5](#).

Os dois formatos de instrução MIPS até aqui são R e I. Os 16 primeiros bits são iguais: ambos contêm um campo *op*, indicando a operação básica; um campo *rs*, indicando um dos operandos origem; e um campo *rt*, que especifica o outro operando origem,

exceto para load word, em que especifica o registrador destino. O formato R divide os 16 últimos bits em um campo *rd*, especificando o registrador destino; um campo *shamt*, explicado na [Seção 2.6](#); e o campo *funct*, que particulariza a operação específica das instruções no formato R. O formato I mantém os 16 bits finais como um único campo de endereço.

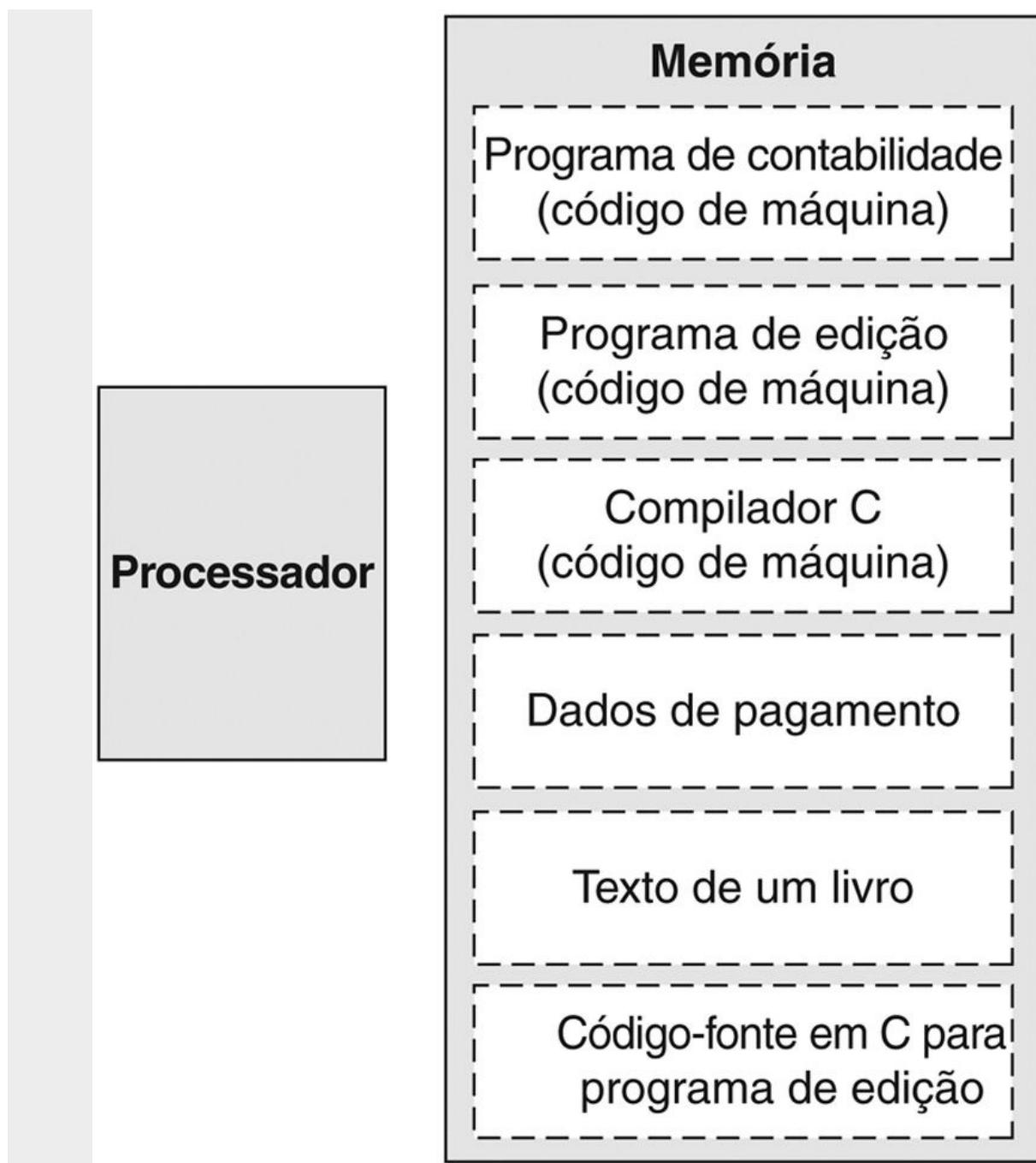
## Linguagem de máquina do MIPS

### Colocando em perspectiva

Os computadores de hoje são baseados em dois princípios fundamentais:

1. As instruções são representadas como números.
2. Os programas são armazenados na memória para serem lidos ou escritos, assim como os números.

Esses princípios levam ao conceito de *programa armazenado*; sua invenção permite que o “gênio da computação saia de sua garrafa”. A Figura 2.7 mostra o poder do conceito; especificamente, a memória pode conter o código-fonte de um editor de textos, o código de máquina compilado correspondente, o texto que o programa compilado está usando e até mesmo o compilador que gerou o código de máquina.



**FIGURA 2.7** O conceito de programa armazenado.

Os programas armazenados permitem que um computador que realiza contabilidade se torne, em um piscar de olhos, um computador que ajuda um autor a escrever um livro. A troca acontece simplesmente carregando a memória com programas e dados e depois dizendo ao computador para iniciar a execução em determinado local na memória. Tratar as instruções da mesma maneira que os dados, simplifica

bastante tanto o hardware da memória quanto o software dos sistemas computacionais. Especificamente, a tecnologia de memória necessária para os dados também pode ser usada para programas, e programas como compiladores, por exemplo, podem traduzir o código escrito em uma notação muito mais conveniente para os humanos, em código que o computador consiga entender.

Uma consequência de instruções em forma de números é que os programas normalmente são entregues como arquivos de números binários. A implicação comercial é que os computadores podem herdar softwares já prontos, desde que sejam compatíveis com um conjunto de instruções existente. Essa “compatibilidade binária” normalmente alinha o setor em torno de uma quantidade muito pequena de arquiteturas de conjuntos de instruções.

## Verifique você mesmo

Que instrução MIPS isto representa? Escolha entre uma das quatro opções a seguir.

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

1. sub \$s0, \$s1, \$s2
2. add \$s2, \$s0, \$s1
3. add \$s2, \$s1, \$s0
4. sub \$s2, \$s0, \$s1

## 2.6. Operações lógicas

“Ao contrário”, continuou Tweedledee, “se foi assim, poderia ser; e se assim fosse, seria; mas como não é, então não é. Isso é lógico.”

Lewis Carroll, Alice no país das maravilhas, 1865

Embora os primeiros computadores se concentrassem em palavras completas, logo ficou claro que era útil atuar sobre campos de bits dentro de uma palavra ou até mesmo sobre bits individuais. Examinar os caracteres dentro de uma palavra, cada um dos quais armazenados como 8 bits, é um exemplo dessa operação ([Seção 2.9](#)). Instruções foram acrescentadas às linguagens de programação e às arquiteturas de conjunto de instruções para simplificar, entre outras coisas, o empacotamento e o desempacotamento dos bits em words. Essas instruções são chamadas operações lógicas. A [Figura 2.8](#) mostra as operações lógicas em C, Java e MIPS.

Operações lógicas	Operadores C	Operadores Java	Instruções MIPS
Shift à esquerda	<<	<<	sll
Shift à direita	>>	>>>	srl
AND bit a bit	&	&	and, andi
OR bit a bit			or, ori
NOT bit a bit	~	~	nor

**FIGURA 2.8** Operadores lógicos em C e Java e suas instruções MIPS correspondentes.

MIPS implementa NOT usando um NOR com um operando sendo zero.

A primeira classe dessas operações é chamada de *shifts* (deslocamentos). Elas movem todos os bits de uma word para a esquerda ou direita, preenchendo os bits que ficaram vazios com 0s. Por exemplo, se o registrador \$s0 tivesse

0000 0000 0000 0000 0000 0000 0000 0000 1001<sub>bin</sub> = 9<sub>dec</sub>

e fosse executada a instrução para deslocar 4 bits à esquerda, o novo valor se pareceria com:

0000 0000 0000 0000 0000 0000 1001 0000<sub>bin</sub> = 144<sub>dec</sub>

O dual de um shift à esquerda é um shift à direita. Os nomes reais das duas instruções shift no MIPS são *shift left logical* (sll) e *shift right logical* (srl). A instrução a seguir realiza essa operação, supondo que o valor original estava no

registrador \$t0 e o resultado deverá ir para o registrador \$t2:

```
sll$t2,$s0,4    # reg $t2 = reg. $s0 << 4 bits
```

Adiamos até agora a explicação do campo *shamt*, do formato R. O nome significa *shift amount* (quantidade de deslocamento) e é usado nas instruções de deslocamento. Logo, a versão em linguagem de máquina da instrução anterior é

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

A codificação de s11 é 0 nos campos op e funct, rd contém 10 (Registrador \$t2), rt contém \$s0 e shamt contém 4. O campo rs não é utilizado e, por isso, é definido como 0.

O deslocamento lógico à esquerda oferece um benefício adicional. O deslocamento à esquerda de  $i$  bits gera o mesmo resultado que multiplicar por  $2^i$ , assim como o deslocamento de um número decimal por  $i$  dígitos é equivalente a multiplicar por  $10^i$ . Por exemplo, a instrução s11 anterior desloca de 4, o que gera o mesmo resultado que multiplicar por  $2^4$  ou 16. O primeiro padrão de bits descrito anteriormente representa 9, e  $9 \times 16 = 144$ , o valor do segundo padrão de bits.

Outra operação útil que isola os campos é **AND**, uma operação bit a bit que deixa um 1 no resultado somente se os dois bits dos operandos forem 1. Por exemplo, se o Registrador \$t2 tiver

## AND

Uma operação lógica bit a bit com dois operandos, que calcula um 1 somente se houver um 1 em *ambos* os operandos.

0000 0000 0000 0000 0000 1101 0000 0000<sub>bin</sub>

e o Registrador \$t1 tiver

0000 0000 0000 0000 0011 1100 0000 0000<sub>bin</sub>

então, depois de executar a instrução MIPS

and \$t0,\$t1,\$t2 # reg. \$t0=reg. \$t1 & reg. \$t2

o valor do registrador \$t0 seria

0000 0000 0000 0000 0000 1100 0000 0000<sub>bin</sub>

Como você pode ver, o AND pode aplicar um padrão de bits a um conjunto de bits para forçar 0s onde houver um 0 no padrão de bits. Esse padrão de bits, em conjunto com o AND, tradicionalmente é chamado de *máscara*, pois a máscara “oculta” alguns bits.

Para colocar um valor em um desses 0s, existe o dual do AND, chamado **OR**. Essa é uma operação bit a bit, que coloca 1 no resultado se *qualquer um* dos bits do operando for 1. Exemplificando, se os registradores \$t1 e \$t2 não tiverem sido alterados do exemplo anterior, o resultado da instrução MIPS

or \$t0,\$t1,\$t2 # reg. \$t0 = reg. \$t1 | reg. \$t2

é este valor no registrador \$t0:

0000 0000 0000 0000 0011 1101 1100 0000<sub>bin</sub>

## OR

Uma operação lógica bit a bit com dois operandos, que calcula um 1 se houver um 1 em *qualquer um* dos operandos.

A última operação lógica é um contrário. O **NOT** apanha um operando e coloca um 1 no resultado se um bit do operando for 0, e vice-versa. Usando nossa notação anterior, ele calcula  $\bar{X}$ .

## NOT

Uma operação lógica bit a bit com um operando, que inverte os bits; ou seja, ela substitui cada 1 por um 0, e cada 0 por um 1.

## NOT

Uma operação lógica bit a bit com dois operandos, que calcula o NOT do OR dos dois operandos. Ou seja, ela calcula um 1 somente se houver um 0 em *ambos* os operandos.

Acompanhando o formato de três operandos, os projetistas do MIPS decidiram incluir a instrução **NOR** (NOT OR) no lugar de NOT. Se um operando for zero, então ele é equivalente a NOT:  $A \text{ NOR } 0 = \text{NOT}(A)$   $A \text{ NOR } 0 = \text{NOT}(A)$

Se o registrador \$t1 não tiver mudado desde o exemplo anterior e o registrador \$t3 tiver o valor 0, o resultado da instrução MIPS

```
nor $t0,$t1,$t3 # reg. $t0 = ~ (reg. $t1 | reg. $t3)
```

é este valor no registrador \$t0:

1111 1111 1111 1111 1100 0011 1111 1111<sub>bin</sub>

A [Figura 2.8](#) mostrou o relacionamento entre os operadores em C e Java e as instruções MIPS. As constantes são úteis nas operações lógicas AND e OR, assim como nas operações aritméticas, de modo que o MIPS também oferece as instruções *and imediato* (*andi*) e *or imediato* (*ori*). As constantes são raras para NOR, pois seu uso principal é inverter os bits de um único operando; assim, a arquitetura do conjunto de instruções do MIPS não possui uma versão imediata do NOR.

## Detalhamento

O conjunto de instruções MIPS completo também inclui exclusive or (XOR), que define o bit como 1 quando dois bits correspondentes diferem, e como 0 quando eles são iguais. C permite que *campos de bit* ou *campos* sejam definidos dentro das palavras, ambos permitindo que os objetos sejam empacotados com uma palavra e combinem com uma interface imposta externamente, como um dispositivo de E/S. Todos os campos precisam caber dentro de uma única palavra. Os campos recebem inteiros sem sinal que podem ser tão curtos quanto 1 bit. Os compiladores C inserem e extraem campos usando instruções lógicas no MIPS: and, or, sll e srl.

## Detalhamento

AND lógico imediato e OR lógico imediato colocam 0s nos 16 bits superiores para formar uma constante de 32 bits, diferente do add imediato, que realiza extensão de sinal.

## Verifique você mesmo

Que operações podem isolar um campo em uma word?

1. AND
2. Um deslocamento à esquerda seguido por um deslocamento à direita

## 2.7. Instruções para tomada de decisões

*A utilidade de um computador automático se encontra na possibilidade de usar determinada sequência de instruções repetidamente; o número de vezes em que ela é repetida depende dos resultados do cálculo. Essa escolha pode depender do sinal de um número (zero é considerado positivo para as finalidades da máquina). Consequentemente, apresentamos uma [instrução] (a [instrução] de transferência condicional) que, dependendo do sinal de determinado número, causa a execução de uma dentre duas rotinas.*

Burks, Goldstine e von Neumann, 1947

O que distingue um computador de uma calculadora simples é a sua capacidade

de tomar decisões. Com base nos dados de entrada e nos valores criados durante o cálculo, diferentes instruções são executadas. A tomada de decisão normalmente é representada nas linguagens de programação usando a instrução *if*, às vezes combinadas com instruções *go to* e rótulos (labels). O assembly do MIPS inclui duas instruções para tomada de decisões, semelhantes a uma instrução *if* com um *go to*. A primeira instrução é:

```
beq registrador1, registrador2, L1
```

Essa instrução significa ir até a instrução chamada *L1* se o valor no *registrador1* for igual ao valor no *registrador2*. O mnemônico *beq* significa *branch if equal* (desviar se for igual). A segunda instrução é:

```
bne registrador1, registrador2, L1
```

Ela significa ir até a instrução chamada *L1* se o valor no *registrador1* *não* for igual ao valor no *registrador2*. O mnemônico *bne* significa *branch if not equal* (desviar se não for igual). Essas duas instruções tradicionalmente são denominadas **desvios condicionais**.

## desvio condicional

Uma instrução que requer a comparação de dois valores e que leva em conta uma transferência de controle subsequente para um novo endereço no programa, com base no resultado da comparação.

## Compilando if-then-else em desvios condicionais

### Exemplo

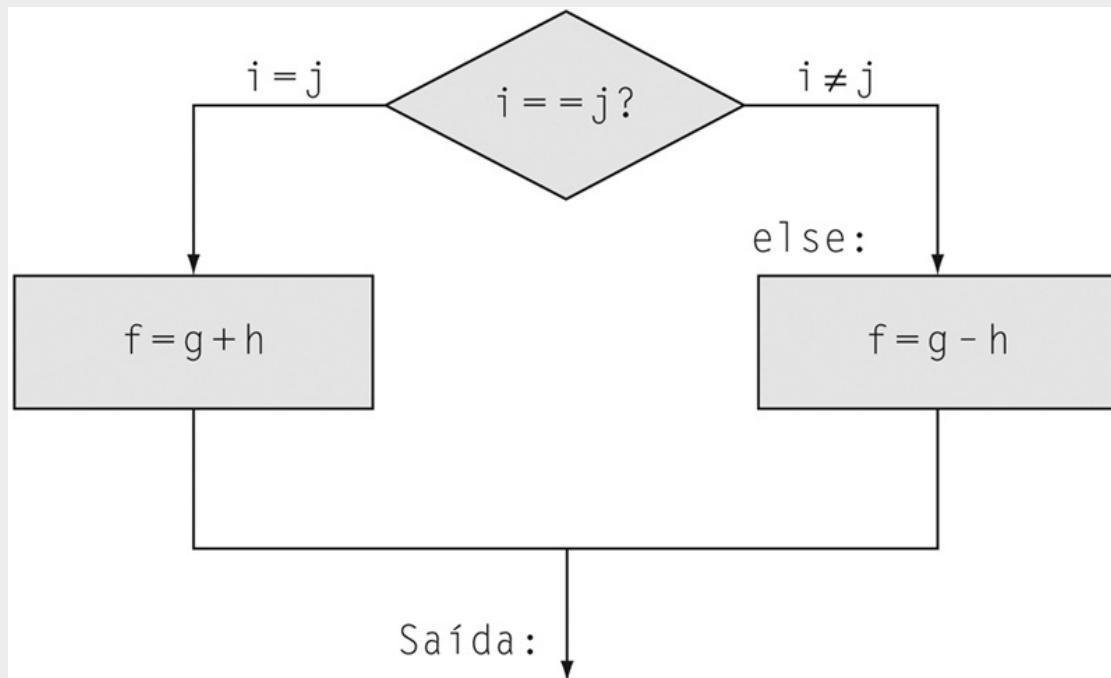
No segmento de código a seguir, *f*, *g*, *h*, *i* e *j* são variáveis. Se as cinco variáveis de *f* a *j* correspondem aos cinco registradores de *\$s0* a *\$s4*, qual é o código MIPS compilado para esta instrução *if* em C?

```
if (I == j) f = g + h; else f = g - h;
```

## Resposta

A Figura 2.9 é um fluxograma de como deve ser o código MIPS. A primeira expressão compara a igualdade, de modo que poderíamos querer desviar se os registradores forem a instrução de igual (beq). De modo geral, o código será mais eficiente se testarmos a condição oposta ao desvio no lugar do código que realiza a parte *then* subsequente do *if* (o rótulo Else é definido a seguir) e assim usamos o desvio se os registradores forem a instrução de *não igual* (bne):

```
bne $s3,$s4,Else # vá para Else se I ≠ j
```



**FIGURA 2.9 Ilustração das opções na instrução if acima.**

A caixa da esquerda corresponde à parte *then* da instrução *if*, e a caixa da direita corresponde à parte *else*.

A próxima instrução de atribuição realiza uma única operação, e se todos os

operандos estiverem em registradores, é apenas uma instrução:

```
add $s0,$s1,$s2 # f = g + h (ignorada se I ≠ j)
```

Agora, precisamos ir até o final da instrução *if*. Este exemplo apresenta outro tipo de desvio, normalmente chamado *desvio incondicional*. Essa instrução diz que o processador sempre deverá seguir o desvio. Para distinguir entre os desvios condicionais e incondicionais, o nome MIPS para esse tipo de instrução é *jump*, abreviado como *j* (o rótulo *Exit* é definido a seguir).

```
j Exit # vá para Exit
```

A instrução de atribuição na parte *else* da instrução *if* pode novamente ser compilada para uma única instrução. Só precisamos anexar um rótulo *Else* a essa instrução. Também mostramos o rótulo *Exit* que está após essa instrução, mostrando o final do código compilado de *if-then-else*:

```
Else:sub $s0,$s1,$s2 #f = g - h (ignorada se I = j)
      Exit:
```

Observe que o montador alivia o compilador e o programador assembly do trabalho de calcular endereços para os desvios, assim como evita o cálculo dos endereços de dados para loads e stores ([Seção 2.12](#)).

## Interface hardware/software

Os compiladores constantemente criam desvios e rótulos onde não aparecem na linguagem de programação. Evitar o trabalho de escrever rótulos e desvios explícitos é um benefício em linguagens de programação de alto nível e um dos motivos para a codificação ser mais rápida nesse nível.

## Loops

Decisões são importantes tanto para escolher entre duas alternativas — como encontramos nas instruções *if* — quanto para repetir um cálculo — como nos loops. As mesmas instruções assembly são os blocos de montagem para os dois casos.

## Compilando um loop WHILE em C

### Exemplo

Aqui está um loop tradicional em C:

```
while (save[i] == k)
    i += 1;
```

Suponha que *i* e *k* correspondam aos registradores \$s3 e \$s5 e a base do array *save* esteja em \$s6. Qual é o código assembly MIPS correspondente a esse segmento C?

### Resposta

O primeiro passo é carregar *save[i]* em um registrador temporário. Antes que possamos carregar *save[i]* em um registrador temporário, precisamos ter seu endereço. Antes que possamos somar *i* à base do array *save* para formar o endereço, temos de multiplicar *i* por 4, em razão do problema do endereçamento em bytes. Felizmente, podemos usar o deslocamento lógico à esquerda, pois o deslocamento em 2 bits à esquerda multiplica por  $2^2$  ou 4 (Seção “Operações Lógicas”, anteriormente neste capítulo). Precisamos acrescentar o rótulo *Loop* para podermos desviar de volta a essa instrução no final do loop:

```
Loop: sll $t1,$s3,2      # Registrador temporário $t1 = i * 4
```

Para obter o endereço de *save[i]*, temos de somar *\$t1* e a base do array *save* em *\$t6*:

```
add $t1,$t1,$s6      # $t1 = endereço de save[i]
```

Agora, podemos usar esse endereço para carregar `save[i]` em um registrador temporário:

```
lw $t0,0($t1)      # Registro temporário $t0 = save[i]
```

A próxima instrução realiza o teste do loop, terminando se `save[i] ≠ k`:

```
bne $t0,$s5, Exit    # vá para Exit se save[i] ≠ k
```

A próxima instrução soma 1 a `i`:

```
add $s3,$s3,1      # I = i + 1
```

O final do loop desvia de volta ao teste do *while* no início do loop. Simplesmente acrescentamos o rótulo `Exit` depois dele e terminamos:

```
J      Loop      # vá para o Loop  
Exit:
```

(Veja nos exercícios uma otimização para essa sequência.)

## Interface hardware/software

Essas sequências de instruções que terminam em um desvio são tão fundamentais para a compilação que recebem seu próprio termo: um **bloco básico** é uma sequência de instruções sem desvios, exceto, possivelmente, no final, e sem destinos de desvio ou rótulos de desvio, exceto, possivelmente, no início. Uma das primeiras fases da compilação é desmembrar o programa em blocos básicos.

## bloco básico

Uma sequência de instruções sem desvios (exceto, possivelmente, no final) e sem destinos de desvio ou rótulos de desvio (exceto, possivelmente, no início).

O teste de igualdade ou desigualdade provavelmente é o teste mais comum, mas às vezes é útil ver se uma variável é menor do que outra. Por exemplo, um loop *for* pode querer testar se a variável de índice é menor do que 0. Essas comparações são realizadas em assembly do MIPS com uma instrução que compara dois registradores e atribui 1 a um terceiro registrador se o primeiro for menor do que o segundo; caso contrário, é atribuído 0. A instrução MIPS é chamada *set on less than* (atribuir se menor que) ou *slt*. Por exemplo,

```
slt      $t0, $s3, $s4    # $t0 = 1 se $s3 < $s4
```

significa que é atribuído 1 ao registrador *\$t0* se o valor no registrador *\$s3* for menor do que o valor no registrador *\$s4*; caso contrário, é atribuído 0 ao registrador *\$t0*.

Operadores constantes são comuns nas comparações, de modo que existe uma versão imediata da instrução “*set on less than*”. Para testar se o registrador *\$s2* é menor do que a constante 10, podemos simplesmente escrever

```
slti     $t0,$s2,10    # $t0 = 1 se $s2 < 10
```

## Interface hardware/software

Os compiladores MIPS utilizam as instruções *slt*, *slti*, *beq*, *bne* e o valor fixo 0 (sempre à disposição com a leitura do registrador *\$zero*) para criar todas as condições relativas: igual, diferente, menor que, menor ou igual, maior que, maior ou igual.

Atentando para a advertência de von Neumann quanto à simplicidade do “equipamento”, a arquitetura do MIPS não inclui “desvio se menor que”, pois

isso é muito complicado; ou ela esticaria o tempo do ciclo de clock ou exigiria ciclos de clock extras por instrução. Duas instruções mais rápidas são mais úteis.

## Interface hardware/software

As instruções de comparação precisam lidar com a dicotomia entre números com sinal e sem sinal. Às vezes, um padrão de bits com 1 no bit mais significativo representa um número negativo e, naturalmente, é menor que qualquer número positivo, que precisa ter um 0 no bit mais significativo. Com inteiros sem sinal, por outro lado, um 1 no bit mais significativo representa um número que é *maior* que qualquer um que comece com um 0. (Logo tiraremos proveito desse significado dual do bit mais significativo para reduzir o custo da verificação dos limites do array.)

MIPS oferece duas versões da comparação “set on less than” para tratar dessas alternativas. *Set on less than* (`slt`) e *set on less than immediate* (`slti`) funcionam com inteiros com sinal. Os inteiros sem sinal são comparados por meio de *set on less than unsigned* (`sltu`) e *set on less than immediate unsigned* (`sltiu`).

## Comparação de números com sinal e sem sinal

### Exemplo

Suponha que o registrador `$s0` tenha o número binário

1111 1111 1111 1111 1111 1111 1111 1111<sub>bin</sub>

e que o registrador `$s1` tenha o número binário

0000 0000 0000 0000 0000 0000 0000 0001<sub>bin</sub>

Quais são os valores dos registradores `$t0` e `$t1` após essas duas instruções?

```
slt    $t0, $s0, $s1 # comparação com sinal  
sltu   $t1, $s0, $s1 # comparação sem sinal
```

## Resposta

O valor no registrador  $\$s0$  representa  $-1_{dec}$  se for um inteiro e  $4.294.967.295_{dec}$  se for um inteiro sem sinal. O valor no registrador  $\$s1$  representa  $1_{dec}$  em qualquer caso. O registrador  $\$t0$  tem o valor 1, pois  $-1_{dec} < 1_{dec}$ , e o registrador  $\$t1$  tem o valor 0, desde  $4.294.967.295_{dec} > 1_{dec}$ .

Tratar números com sinal como se fossem sem sinal é um modo de baixo custo para verificar se  $0 \leq x < y$ , que corresponde à verificação de índice fora de limite dos arrays. O principal é que os inteiros negativos na notação de complemento de dois se parecem com números grandes na notação sem sinal; ou seja, o bit mais significativo é um bit de sinal na primeira notação, mas uma grande parte do número na segunda. Assim, uma comparação sem sinal de  $x < y$  também verifica se  $x$  é negativo, bem como se  $x$  é menor que  $y$ .

## Atalho para verificação de limites

### Exemplo

Use este atalho para reduzir uma verificação de índice fora dos limites: salte para `IndexOutOfBoundsException` se  $\$s1 \geq \$t2$  ou se  $\$s1$  é negativo.

### Resposta

O código de verificação só usa `sltu` para realizar as duas verificações:

```
sltu $t0,$s1,$t2 # $t0 = 0 se $s1 >= tamanho ou $s1 < 0  
beq $t0,$zero,IndexOutOfBoundsException #se erro, goto Error
```

## Instrução Case/Switch

A maioria das linguagens de programação possui uma instrução *case* ou *switch*, para o programador poder selecionar uma dentre muitas alternativas, dependendo de um único valor. O modo mais simples de implementar *switch* é por meio de uma sequência de testes condicionais, transformando a instrução *switch* em uma cadeia de instruções *if-then-else*.

Às vezes, as alternativas podem ser codificadas de forma mais eficiente como uma tabela de endereços de sequências de instruções alternativas, chamada **tabela de endereços de desvio** ou **tabela de desvio**, e o programa só precisa indexar na tabela e depois desviar para a sequência apropriada. A tabela de desvios é, então, apenas um array de palavras com endereços que correspondem aos rótulos no código. O programa carrega a entrada apropriada a partir da tabela de desvio para um registrador. Depois, ele precisa desviar usando o endereço no registrador. Para apoiar tais situações, computadores como o MIPS incluem uma instrução *jump register* (*jr*), significando um desvio incondicional para o endereço especificado em um registrador. Depois desvia para o endereço apropriado usando essa instrução. Veremos um uso ainda mais popular de *jr* na próxima seção.

## tabela de endereços de desvio

Também chamada de **tabela de desvios**. Uma tabela de endereços de sequências de instruções alternativas.

## Interface hardware/software

Embora existam muitas instruções para decisões e loops em linguagens de programação como C e Java, a instrução básica que as implementa no nível do conjunto de instruções é o desvio condicional.

## Detalhamento

Se você já ouviu falar em *delayed branches*, explicados no Capítulo 4, não se preocupe: o montador do MIPS os torna invisíveis ao programador assembly.

## Verifique você mesmo

- I. A linguagem C possui muitas instruções para decisões e loops, enquanto o MIPS possui poucas. Quais dos seguintes itens explicam ou não esse

desequilíbrio? Por quê?

1. Mais instruções de decisão tornam o código mais fácil de ler e entender.
2. Menos instruções de decisão simplificam a tarefa da camada inferior responsável pela execução.
3. Mais instruções de decisão significam menos linhas de código, o que geralmente reduz o tempo de codificação.
4. Mais instruções de decisão significam menos linhas de código, o que geralmente resulta na execução de menos operações.

II. Por que a linguagem C oferece dois conjuntos de operadores para AND (& e &&) e dois conjuntos de operadores para OR (| e ||), enquanto o MIPS não faz isso?

1. As operações lógicas AND e OR implementam & e |, enquanto os desvios condicionais implementam && e ||.
2. A afirmativa anterior é o contrário: && e || correspondem a operações lógicas, enquanto & e | são mapeados para desvios condicionais.
3. Elas são redundantes e significam a mesma coisa: && e || são simplesmente herdados da linguagem de programação B, a antecessora do C.

## 2.8. Suporte a procedimentos no hardware do computador

Um **procedimento** ou função é uma ferramenta que os programadores utilizam a fim de estruturar programas, tanto para torná-los mais fáceis de entender quanto para permitir que o código seja reutilizado. Os procedimentos permitem que o programador se concentre em apenas uma parte da tarefa de cada vez, com os parâmetros atuando como uma interface entre o procedimento e o restante do programa e dos dados, pois eles passam valores e retornam resultados. Os procedimentos são uma forma de implementar **abstração** no software.

### procedimento

Uma sub-rotina armazenada que realiza uma tarefa com base nos parâmetros que lhe são passados.



## A B S T R A Ç Ã O

Você pode pensar em um procedimento como um espião que sai com um plano secreto, adquire recursos, realiza a tarefa, sobre seus rastros e depois retorna ao ponto de origem com o resultado desejado. Nada mais deverá ter sido alterado depois que a missão terminar. Além do mais, um espião opera apenas sobre aquilo que ele “precisa saber”, de modo que não pode fazer suposições sobre seu patrão.

1. De modo semelhante, na execução de um procedimento, o programa precisa seguir estas seis etapas:
2. Colocar parâmetros em um lugar onde o procedimento possa acessá-los.
3. Transferir o controle para o procedimento.
4. Adquirir os recursos de armazenamento necessários para o procedimento.
5. Realizar a tarefa desejada.
6. Colocar o valor de retorno em um local onde o programa que o chamou possa acessá-lo.
7. Retornar o controle para o ponto de origem, pois um procedimento pode ser chamado de vários pontos em um programa.

Como já dissemos, os registradores são o local mais rápido para manter dados em um computador, de modo que queremos usá-los ao máximo. O software do

MIPS utiliza a seguinte convenção na alocação de seus 32 registradores:

- \$a0-\$a3: quatro registradores de argumento, para passar parâmetros
- \$v0-\$v1: dois registradores de valor, para valores de retorno
- \$ra: um registrador de endereço de retorno, para retornar ao ponto de origem

Além de alocar esses registradores, o assembly do MIPS inclui uma instrução apenas para os procedimentos: ela desvia para um endereço e simultaneamente salva o endereço da instrução seguinte no registrador \$ra. A **instrução de jump-and-link** (jal) é escrita simplesmente como

```
jal EndereçoProcedimento
```

## instrução de jump-and-link

Uma instrução que salta para um endereço e simultaneamente salva o endereço da instrução seguinte em um registrador (\$ra no MIPS).

A parte do *link* no nome da instrução significa que um endereço ou link é formado de modo a apontar para o local de chamada, permitindo que o procedimento retorne ao endereço correto. Esse “link”, armazenado no registrador \$ra, é denominado **endereço de retorno**. O endereço de retorno é necessário porque o mesmo procedimento poderia ser chamado de várias partes do programa.

## endereço de retorno

Um link para o local de chamada, permitindo que um procedimento retorne ao endereço correto; no MIPS, ele é armazenado no registrador \$ra.

Para dar suporte a tais situações, computadores como o MIPS utilizam uma instrução de *jump register* (jr), apresentada anteriormente para ajudar com as instruções case, significando um desvio incondicional para o endereço especificado em um registrador:

```
jr $ra
```

A instrução de jump register pula para o endereço armazenado no registrador \$ra — que é exatamente o que queremos. Assim, o programa que chama, ou **caller**, coloca os valores de parâmetro em \$a0-\$a3 e utiliza jal x para desviar para o procedimento x (às vezes denominado **callee**). O callee, então, realiza os cálculos, coloca os resultados em \$v0-\$v1 e retorna o controle para o caller usando jr \$ra.

### caller

O programa que instiga um procedimento e oferece os valores de parâmetro necessários.

### callee

Um procedimento que executa uma série de instruções armazenadas com base nos parâmetros fornecidos pelo caller e depois retorna o controle para o caller novamente.

Num programa armazenado é necessário ter um registrador para manter o endereço da instrução atual sendo executada. Por motivos históricos, esse registrador quase sempre é denominado **contador de programa**, abreviado como *PC* (Program Counter) na arquitetura MIPS, embora um nome mais sensato teria sido *registraror de endereço de instrução*. A instrução jal salva o PC + 4 no registrador \$ra para o link com a instrução seguinte, a fim de preparar o retorno do procedimento.

### contador de programa (PC)

O registrador que contém o endereço da instrução sendo executada no programa.

## Usando mais registradores

Suponha que um compilador precise de mais registradores para um procedimento do que os quatro registradores para argumentos e os dois para valores de retorno. Como temos de cobrir nossos rastros após o término desta missão, quaisquer registradores necessários ao caller deverão ser restaurados aos valores que possuíam *antes* de o procedimento ser chamado. Essa situação é um

exemplo em que são usados os spilled registers em memória, conforme mencionamos na Seção “Interface hardware/software”.

A estrutura de dados ideal para armazenar os spilled registers é uma **pilha** — uma fila do tipo “último a entrar, primeiro a sair”. Uma pilha precisa de um ponteiro para o endereço alocado mais recentemente na pilha, a fim de mostrar onde o próximo procedimento deverá colocar os spilled registers ou onde os valores antigos dos registradores estão localizados. O **stack pointer** é ajustado em uma palavra para cada registrador salvo ou restaurado. O software MIPS reserva o registrador 29 para o stack pointer, dando-lhe o nome óbvio \$sp. As pilhas são tão comuns que possuem seus próprios termos para transferir dados da pilha e para ela: colocar dados na pilha é denominado **push**, e remover dados da pilha é denominado **pop**.

## pilha (stack)

Uma estrutura de dados utilizada para armazenar os registradores, organizada como uma fila do tipo “último a entrar, primeiro a sair”.

## stack pointer

Um valor indicando o endereço alocado mais recentemente em uma pilha, que mostra onde os registradores devem ser armazenados ou onde os valores antigos dos registradores podem ser localizados.

## push

Acrescentar elemento à pilha.

## pop

Remover elemento da pilha.

Por motivos históricos, as pilhas “crescem” de endereços maiores para endereços menores. Essa convenção significa que você põe valores na pilha subtraindo do valor do stack pointer. Somar ao stack pointer diminui essa pilha, removendo seus valores.

## Compilando um procedimento em C que não

## chama outro procedimento

### Exemplo

Vamos transformar o exemplo da Seção 2.2 em um procedimento em C:

```
int exemplo_folha (int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

Qual é o código assembly do MIPS compilado?

### Resposta

As variáveis de parâmetro `g`, `h`, `i` e `j` correspondem aos registradores de argumento `$a0`, `$a1`, `$a2` e `$a3`, e `f` corresponde a `$s0`. O programa compilado começa com o rótulo do procedimento:

```
exemplo_folha:
```

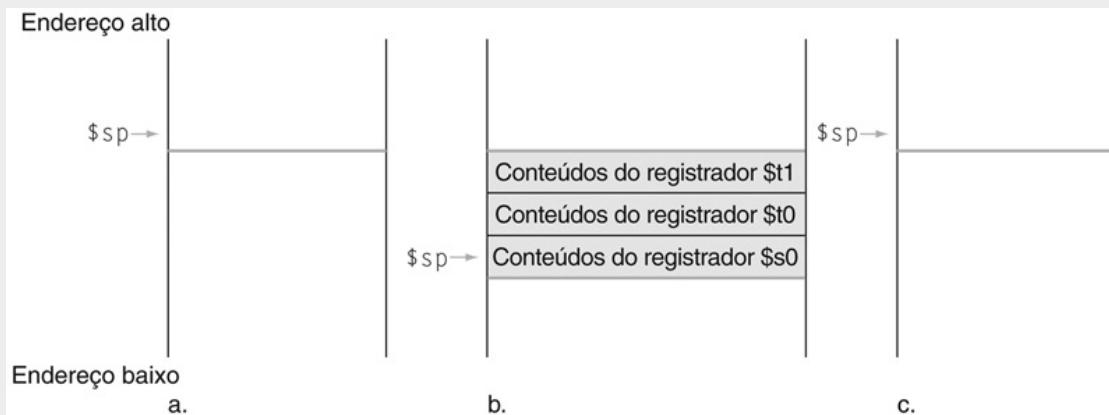
O próximo passo é salvar os registradores usados pelo procedimento. A instrução de atribuição em C no corpo do procedimento é idêntica ao exemplo da Seção 2.2, que usa dois registradores temporários. Assim, precisamos salvar três registradores: `$s0`, `$t0` e `$t1`. “Empilhamos” os valores antigos, criando espaço para três palavras (12 bytes) na pilha e depois as armazenamos:

```

addi $sp, $sp, -12    # ajusta pilha, criando espaço para 3 itens
sw $t1, 8($sp)        # salva reg. $t1 para usar depois
sw $t0, 4($sp)        # salva reg. $t0 para usar depois
sw $s0, 0($sp)        # salva reg. $s0 para usar depois

```

A Figura 2.10 mostra a pilha antes, durante e após a chamada do procedimento.



**FIGURA 2.10** Os valores do stack pointer e a pilha (a) antes, (b) durante e (c) após a chamada do procedimento.  
O stack pointer sempre aponta para o “topo” da pilha ou para a última palavra na pilha neste desenho.

As três instruções seguintes correspondem ao corpo do procedimento, que segue o exemplo da Seção 2.2:

```

add $t0,$a0,$a1      # reg. $t0 contém g + h
add $t1,$a2,$a3      # reg. $t1 contém i + j
sub $s0,$t0,$t1      # f = $t0 - $t1, que é (g + h)-(i + j)

```

Para retornar o valor de f, nós o copiamos para um registrador de valor de retorno:

```
add$v0,$s0,$zero      # retorna f($v0 = $s0 + 0)
```

Antes de retornar, restauramos os três valores antigos dos registradores que salvamos, desempilhando-os:

```
lw $s0, 0($sp)      # restaura reg. $s0 para o caller  
lw $t0, 4($sp)      # restaura reg. $t0 para o caller  
lw $t1, 8($sp)      # restaura reg. $t1 para o caller  
addi $sp,$sp,12      # ajusta pilha para excluir 3 itens
```

O procedimento termina com um jump register usando o endereço de retorno:

```
jr      $ra      # desvia de volta à rotina que chamou
```

No exemplo anterior, usamos registradores temporários e consideramos que seus valores antigos precisam ser salvos e restaurados. Para evitar salvar e restaurar um registrador cujo valor nunca é utilizado, o que poderia acontecer com um registrador temporário, o software do MIPS separa 18 dos registradores em dois grupos:

- \$t0–\$t9: registradores temporários que *não* são preservados pelo callee (procedimento chamado) em uma chamada de procedimento
- \$s0–\$s7: registradores salvos que precisam ser preservados em uma chamada de procedimento (se forem usados, o calee os salva e restaura)

Essa convenção simples reduz o armazenamento de registradores. No exemplo anterior, como o caller não espera que os registradores \$t0 e \$t1 sejam preservados durante uma chamada de procedimento, podemos descartar dois stores e dois loads do código. Ainda temos de salvar e restaurar \$s0, pois o procedimento chamado deve considerar que o caller precisa de seu valor.

## Procedimentos aninhados

Os procedimentos que não chamam outros são denominados procedimentos *folha*. A vida seria simples se todos os procedimentos fossem procedimentos folha, mas não são. Assim como um espião poderia雇用 outros espiões como parte de uma missão, que, por sua vez, poderiam utilizar ainda mais espiões, os procedimentos também chamam outros procedimentos. Além do mais, os procedimentos recursivos ainda chamam “clones” de si mesmos. Assim como precisamos ter cuidado ao usar registradores nos procedimentos, também precisamos ter mais cuidado ao chamar procedimentos não folha.

Por exemplo, suponha que o programa principal chame o procedimento A com um argumento 3, colocando o valor 3 no registrador \$a0 e depois usando `jal A`. Depois, suponha que o procedimento A chame o procedimento B por meio de `jal B` com um argumento 7, também colocado em \$a0. Como A ainda não terminou sua tarefa, existe um conflito com relação ao uso do registrador \$a0. De modo semelhante, existe um conflito em relação ao endereço de retorno no registrador \$ra, pois ele agora tem o endereço de retorno para B. A menos que tomemos medidas para evitar o problema, esse conflito eliminará a capacidade do procedimento A de retornar para o procedimento que o chamou.

Uma solução é empilhar todos os outros registradores que precisam ser preservados, assim como fizemos com os registradores salvos. O caller empilha quaisquer registradores de argumento (\$a0–\$a3) ou registradores temporários (\$t0–\$t9) que sejam necessários após a chamada. O callee empilha o registrador do endereço de retorno \$ra e quaisquer registradores salvos (\$s0–\$s7) usados por ele. O stack pointer \$sp é ajustado para levar em consideração a quantidade de registradores colocados na pilha. No retorno, os registradores são restaurados da memória e o stack pointer é reajustado.

## Compilando um procedimento C recursivo, mostrando a ligação do procedimento aninhado

### Exemplo

Vamos realizar um procedimento recursivo que calcula o fatorial:

```

int fact (int n)
{
    if (n < 1) return (1);
    else return (n * fact(n - 1));
}

```

Qual é o código assembly do MIPS?

## Resposta

A variável de parâmetro *n* corresponde ao registrador de argumento \$a0. O programa compilado começa com o rótulo do procedimento e depois salva dois registradores na pilha, o endereço de retorno e \$a0:

```

fact:
    addi $sp, $sp, -8    # ajusta pilha para 2 itens
    sw   $ra, 4($sp)    # salva o endereço de retorno
    sw   $a0, 0($sp)    # salva o argumento n

```

Na primeira vez que *fact* é chamado, *sw* salva um endereço do programa que chamou *fact*. As duas instruções seguintes testam se *n* é menor do que 1, indo para L1 se *n*  $\geq 1$ .

```

slti $t0,$a0,1    # teste para n < 1
beq $t0,$zero,L1 # se n >= 1, vai para L1

```

Se *n* for menor do que 1, *fact* retorna 1, colocando 1 em um registrador de valor: ele soma 1 a 0 e coloca essa soma em \$v0. Depois, ele retira os dois valores salvos da pilha e desvia para o endereço de retorno:

```
addi $v0,$zero,1 # retorna 1  
addi $sp,$sp,8   # retira 2 itens da pilha  
jr   $ra          # retorna para quem chamou
```

Antes de retirar dois itens da pilha, poderíamos ter restaurado \$a0 e \$ra. Como \$a0 e \$ra não mudam quando n é menor do que 1, pulamos essas instruções.

Se n não for menor do que 1, o argumento n é diminuído e depois fact é chamado novamente com o valor reduzido.

```
L1: addi $a0,$a0,-1 # n >= 1: argumento recebe (n - 1)  
jal fact           # chama fact com (n - 1)
```

A próxima instrução é onde fact retorna. Agora, o endereço de retorno antigo e o argumento antigo são restaurados, juntamente com o stack pointer:

```
lw    $a0, 0($sp)  # retorna jal: restaura argumento n  
lw    $ra, 4($sp)  # restaura o endereço de retorno  
addi $sp, $sp, 8   # ajusta stack pointer para retirar 2 itens
```

Em seguida, o registrador de valor \$v0 recebe o produto do argumento antigo \$a0 e o valor atual do registrador de valor. Consideraremos que exista uma instrução de multiplicação à disposição, embora isso não seja explicado antes do Capítulo 3:

```
mul $v0,$a0,$v0      # retorna n * fact(n - 1)
```

Finalmente, fact salta novamente para o endereço de retorno:

```
jr   $ra          # retorna para o procedimento que chamou
```

## Interface hardware/software

Uma variável em C é um local na memória e sua interpretação depende tanto do seu *tipo* quanto da sua *classe de armazenamento*. Alguns exemplos são inteiros e caracteres (Seção 2.9). A linguagem C possui duas classes de armazenamento: *automática* e *estática*. As variáveis automáticas são locais a um procedimento e são descartadas quando o procedimento termina. As variáveis estáticas permanecem durante entradas e saídas de procedimentos. As variáveis C declaradas fora de todos os procedimentos são consideradas estáticas, assim como quaisquer variáveis declaradas por meio da palavra reservada *static*. As outras são automáticas. Para simplificar o acesso aos dados estáticos, o software do MIPS reserva outro registrador, chamado **ponteiro global** ou \$gp.

## ponteiro global

O registrador reservado para apontar para a área estática.

A [Figura 2.11](#) resume o que é preservado em uma chamada de procedimento. Observe que vários esquemas preservam a pilha, garantindo que o caller receberá os mesmos dados em um load da pilha que foram armazenados nela. A pilha acima de \$sp é preservada simplesmente verificando se o procedimento chamado não escreve acima de \$sp; \$sp é preservado pelo procedimento chamado, somando-se exatamente o mesmo valor que foi subtraído dele, e os outros registradores são preservados por serem salvos na pilha (se forem usados) e restaurados de lá.

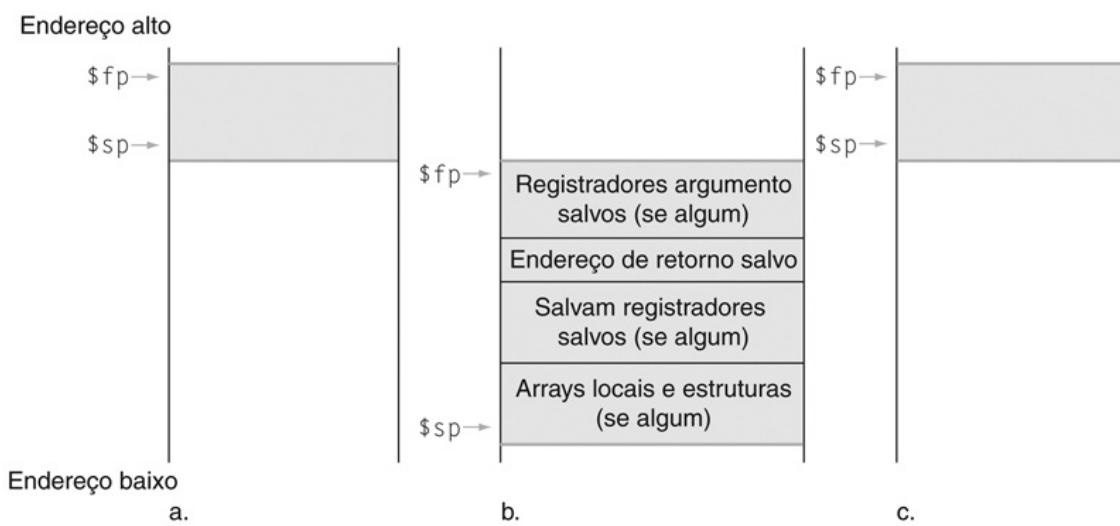
Preservado	Não preservado
Registradores salvos: \$s0-\$s7	Registradores temporários: \$t0-\$t9
Registrador de stack pointer: \$sp	Registradores de argumento: \$a0-\$a3
Registrador de endereço de retorno: \$ra	Registradores de valores de retorno: \$v0-\$v1
Pilha acima do stack pointer	Pilha abaixo do stack pointer

**FIGURA 2.11** O que é e o que não é preservado durante uma chamada de procedimento.

Se o software contar com o registrador de frame pointer ou com o registrador de ponteiro global, discutidos nas próximas seções, eles também serão preservados.

## Alocando espaço para novos dados na pilha

A complexidade final é que a pilha também é utilizada para armazenar variáveis que são locais ao procedimento, que não cabem nos registradores, como arrays ou estruturas locais. O segmento da pilha que contém os registradores salvos e as variáveis locais de um procedimento é chamado **frame de procedimento** ou **registro de ativação**. A Figura 2.12 mostra o estado da pilha antes, durante e após a chamada de um procedimento.



**FIGURA 2.12 Ilustração da alocação de pilha (a) antes, (b) durante e (c) após a chamada de um procedimento.**

O frame pointer (\$fp) aponta para a primeira palavra do frame, normalmente um registrador de argumento salvo, e o stack pointer (\$sp) aponta para o topo da pilha. A pilha é ajustada de modo a criar espaço para todos os registradores salvos e quaisquer variáveis locais residentes na memória. Como o stack pointer pode mudar durante a execução do programa, é mais fácil para os programadores referenciarem variáveis por meio do frame pointer estável, embora isso também pudesse ser feito por meio do stack pointer e um pouco de aritmética de endereços.

Se não houver variáveis locais na pilha dentro de um procedimento, o compilador ganhará tempo não atribuindo um

endereço ao frame pointer, e depois, restaurando-o. Quando um frame pointer é usado, ele é inicializado usando o endereço que está no \$sp em uma chamada, e o \$sp é restaurado usando o valor do \$fp. Essa informação também aparece na Coluna 4 do Guia de Referência do MIPS, no final deste livro.

## frame de procedimento

Também chamado **registro de ativação**. O segmento da pilha contendo os registradores salvos e as variáveis locais de um procedimento.

Alguns softwares MIPS utilizam o **frame pointer** (\$fp) a fim de apontar para a primeira word do registro de ativação de um procedimento. O stack pointer poderia mudar durante o procedimento, e assim as referências a uma variável local na memória poderiam ter offsets diferentes, dependendo de onde estiverem no procedimento, o que torna o procedimento mais difícil de entender. Como alternativa, um frame pointer oferece um registrador base estável dentro de um procedimento para as referências locais à memória. Observe que um registro de ativação aparece na pilha independentemente de o frame pointer explícito ser utilizado. Evitamos o \$fp impedindo mudanças no \$sp dentro de um procedimento: em nossos exemplos, a pilha é ajustada apenas na entrada e na saída do procedimento.

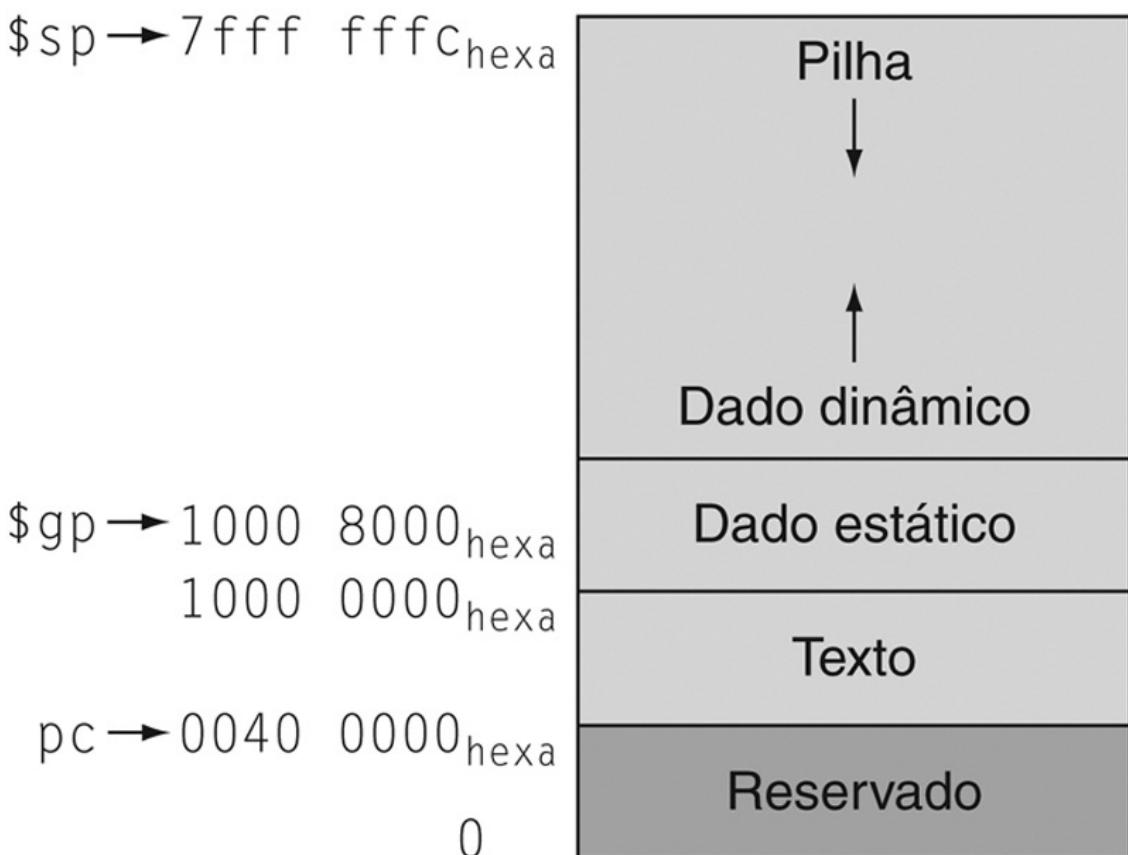
## frame pointer

Um valor indicando o local dos registradores salvos e as variáveis locais para um determinado procedimento.

## Alocando espaço para novos dados no heap

Além das variáveis automáticas que são locais aos procedimentos, os programadores de C precisam de espaço na memória para as variáveis globais e para estruturas de dados dinâmicas. A [Figura 2.13](#) mostra a convenção do MIPS para a alocação de memória. A pilha começa na parte alta da memória e cresce para baixo. A primeira parte da extremidade baixa da memória é reservada, seguida pelo lar do código de máquina do MIPS, tradicionalmente denominado **segmento de texto**. Acima do código existe o *segmento de dados estáticos*, que é o local para constantes e outras variáveis estáticas. Embora os arrays

costumem ter um tamanho fixo e, portanto, correspondam muito bem ao segmento de dados estático, estruturas de dados como listas encadeadas costumam crescer e diminuir durante suas vidas. O segmento para tais estruturas de dados é tradicionalmente chamado de *heap* e fica posicionado logo a seguir na memória. Observe que essa alocação permite que a pilha e o heap cresçam um em direção ao outro, permitindo, assim, o uso eficiente da memória enquanto os dois segmentos aumentam e diminuem.



**FIGURA 2.13** A alocação de memória do MIPS para programas e dados.

Esses endereços são apenas uma convenção do software e não fazem parte da arquitetura MIPS. De cima para baixo, o stack pointer é inicializado com  $7fff\ ffff_{hexa}$  e cresce para baixo, em direção ao segmento de dados. Na outra extremidade, o código do programa (“texto”) começa em  $0040\ 0000_{hexa}$ . Os dados estáticos começam em  $1000\ 0000_{hexa}$ . Os dados dinâmicos, alocados por `malloc` em C e por `new` em Java, vêm em seguida e crescem para cima em direção à pilha, em uma área chamada

heap. O ponteiro global, \$gp, é definido como endereço para facilitar o acesso aos dados. Ele é inicializado com  $1000\ 8000_{hexa}$  para poder acessar de  $1000\ 0000_{hexa}$  até  $1000\ ffff_{hexa}$  usando os offsets de 16 bits positivos e negativos a partir do \$gp. Essa informação também aparece na Coluna 4 do Guia de Referência do MIPS, no final deste livro.

## segmento de texto

O segmento de um arquivo-objeto Unix que contém o código em linguagem de máquina para as rotinas do arquivo-fonte.

A linguagem C aloca e libera espaço no heap com funções explícitas. `malloc()` aloca espaço no heap e retorna um ponteiro para ela, e `free()` libera o espaço no heap para o qual o ponteiro está apontando. A alocação da memória é controlada por programas em C e essa é a fonte de muitos bugs comuns e difíceis de serem encontrados. Esquecer de liberar espaço ocasiona um “vazamento de memória”, que, por fim, pode ocupar tanta memória que venha a causar a falha do sistema operacional. Liberar espaço muito cedo ocasiona “ponteiros pendentes”, podendo fazer os ponteiros apontarem para áreas que o programa nunca desejou. Java utiliza alocação de memória e coleta de lixo automáticas, justamente para evitar esses bugs.

A [Figura 2.14](#) resume as convenções de registrador para o assembly do MIPS. Esta convenção é outro exemplo de tornar o **caso comum veloz**: a maioria dos procedimentos pode ser satisfeita com até 4 argumentos, 2 registradores para um valor de retorno, 8 registradores salvos e 10 registradores temporários sem sequer ir para a memória.

Nome	Número do registrador	Uso	Preservado na chamada?
\$zero	0	O valor constante 0	n.a.
\$v0-\$v1	2-3	Valores para resultados e avaliação de expressões	não
\$a0-\$a3	4-7	Argumentos	não
\$t0-\$t7	8-15	Temporários	não
\$s0-\$s7	16-23	Valores salvos	sim
\$t8-\$t9	24-25	Mais temporários	não
\$gp	28	Ponteiro global	sim
\$sp	29	Stack pointer	sim
\$fp	30	Frame pointer	sim
\$ra	31	Endereço de retorno	sim

**FIGURA 2.14 Convenções de registradores MIPS.**

O registrador 1, chamado \$at, é reservado para o montador ([Seção 2.12](#)), e os registradores 26-27, chamados \$k0-\$k1, são reservados para o sistema operacional. Essa informação também aparece na Coluna 2 do Guia de Referência do MIPS, no final deste livro.

## Detalhamento

E se houver mais do que quatro parâmetros? A convenção do MIPS é colocar os parâmetros extras na pilha, logo acima do stack pointer. O procedimento, então, espera que os quatro primeiros parâmetros estejam nos registradores de \$a0 a \$a3, e que o restante esteja na memória, endereçável por meio do frame pointer.

Conforme dissemos na legenda da Figura 2.12, o frame pointer é conveniente porque todas as referências a variáveis na pilha dentro de um procedimento terão o mesmo offset. Contudo, o frame pointer não é necessário. O compilador C para MIPS sob licença GNU utiliza um frame pointer, mas não o compilador C do MIPS; ele trata o registrador 30 como outro registrador de valor salvo (\$a8).

## Detalhamento

Alguns procedimentos recursivos podem ser implementados iterativamente sem o uso de recursão. A iteração pode melhorar significativamente o desempenho, removendo o overhead associado a chamadas de procedimento. Por exemplo, considere um procedimento usado para acumular uma soma:

```

int sum(int n, int acc) {
    if (n > 0)
        return sum(n - 1, acc + n);
    else
        return acc;
}

```

Considere a chamada de procedimento `sum(3,0)`. Isso resultará em chamadas recursivas a `sum(2,3)`, `sum(1,5)` e `sum(0,6)`, e depois o resultado 6 será retornado quatro vezes. Essa chamada recursiva de `sum` é conhecida como *tail call* e esse exemplo de uso da recursão tail pode ser implementado de modo muito eficiente (suponha que `$a0 = n` e `$a1 = acc`):

```

sum: slti $t0, $a0, 1          # testa se n <= 0
      bne $t0, $zero, sum_exit # vai para sum_exit se n <= 0
      add$a1, $a1, $a0         # soma n a acc
      addi$a0, $a0, -1         # subtrai 1 de n
      j sum                   # vai para sum
sum_exit:
      add$v0, $a1, $zero       # retorna valor acc
      jr $ra                  # retorna ao caller

```

## Verifique você mesmo

Quais das seguintes afirmações sobre C e Java geralmente são verdadeiras?

1. Os programadores C gerenciam os dados explicitamente, enquanto isso é automático em Java.
2. A linguagem C leva a mais problemas de ponteiro e vazamento de memória do que Java.

## 2.9. Comunicando-se com as pessoas

*!(@ | = > (wow open tab at bar is great)*

*Quarta linha do poema de teclado “Hatless Atlas”, 1991 (alguns dão nomes aos caracteres ASCII: “!” é “wow”, “(” é open, “|” é bar, e assim por diante)*

Os computadores foram inventados para devorar números, mas, assim que se tornaram comercialmente viáveis, eles foram usados para processar textos. A maioria dos computadores hoje utiliza bytes de 8 bits para representar caracteres; o *American Standard Code for Information Interchange* (ASCII) é a representação que quase todos seguem. A [Figura 2.15](#) resume o código ASCII.

Valor ASCII	Sinal										
32	Espaç	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

**FIGURA 2.15** Representação dos caracteres no código ASCII.

Observe que as letras maiúsculas e minúsculas diferem exatamente em 32; essa observação pode levar a atalhos na verificação ou mudança entre maiúsculas e minúsculas. Os valores não mostrados incluem caracteres de formatação. Por exemplo, 8 representa backspace, 9 representa o caractere de tabulação e 13, um *carriage return*. Outro valor útil é 0 para null, o valor que a linguagem de programação C utiliza para marcar o final de uma string. Essa informação também aparece na Coluna 3 do Guia de Referência do MIPS, no final deste livro.

## ASCII e números binários

## Exemplo

Poderíamos representar números como strings de dígitos ASCII em vez de inteiros. Em quanto o armazenamento aumenta se o número 1 bilhão for representado em ASCII em vez de inteiro com 32 bits?

## Resposta

Um bilhão é 1.000.000.000, de modo que ele precisaria de 10 dígitos ASCII, cada um com 8 bits de extensão. Assim, a expansão no armazenamento seria  $(10 \times 8)/32$  ou 2,5. Além da expansão no armazenamento, o hardware para somar, subtrair, multiplicar e dividir esses números decimais é difícil e consumiria mais energia. Essas dificuldades explicam por que os profissionais de computação são levados a crer que o binário é natural e que o computador decimal ocasional é bizarro.

Diversas instruções podem extrair um byte de uma palavra, de modo que load word e store word são suficientes para transferir bytes e também palavras. Entretanto, em razão da popularidade do texto em alguns programas, o MIPS oferece instruções para mover bytes. *Load byte* (lb) lê um byte da memória, colocando-o nos 8 bits mais à direita de um registrador. *Store byte* (sb) separa o byte mais à direita de um registrador e o escreve na memória. Assim, copiamos um byte com a sequência

```
lb $t0,0($sp)      # Lê byte da origem  
sb $t0,0($gp)      # Escreve byte no destino
```

Os caracteres normalmente são combinados em strings, que possuem uma quantidade variável de caracteres. Existem três opções para representar uma string: (1) a primeira posição da string é reservada para indicar o tamanho de uma string, (2) uma variável acompanhante possui o tamanho da string (como em uma estrutura) ou (3) a última posição da string é ocupada por um caractere que serve para marcar o final da string. A linguagem C utiliza a terceira opção, terminando uma string com um byte cujo valor é 0 (denominado null, em ASCII). Assim, a string “Cal” é representada em C pelos 4 bytes a seguir, em forma de números decimais: 67, 97, 108, 0. (Como veremos, Java utiliza a

primeira opção.)

## Compilando um procedimento de cópia de string, para demonstrar o uso de strings em C

### Exemplo

O procedimento `strcpy` copia a string `y` para a string `x`, usando a convenção de término com byte nulo da linguagem C:

```
void strcpy (char x[], char y[])
{
    int i;
    i = 0;
    while ((x[i] = y[i]) != '\0') /* copia e testa byte */
        i += 1;
}
```

Qual é o código assembly correspondente no MIPS?

### Resposta

A seguir está o segmento básico em código assembly do MIPS. Considere que os endereços base para os arrays `x` e `y` são encontrados em `$a0` e `$a1`, enquanto `i` está em `$s0`. `strcpy` ajusta o stack pointer e depois salva o registrador de valores salvos `$s0` na pilha:

```
strcpy:
    addi   $sp,$sp,-4    # ajusta pilha para mais 1 item
    sw     $s0, 0($sp)    # salva $s0
```

Para inicializar `i` como 0, a próxima instrução define `$s0` como 0, somando 0 a 0 e colocando essa soma em `$s0`:

```
add    $s0,$zero,$zero  # i = 0 + 0
```

Esse é o início do loop. O endereço de  $y[i]$  é formado inicialmente pela soma de  $i$  a  $y[]$ :

```
L1: add    $t1,$s0,$a1  # endereço de y[i] em $t1
```

Observe que não temos de multiplicar  $i$  por 4, pois  $y$  é um array de *bytes*, e não de palavras, como nos exemplos anteriores.

Para carregar o caractere em  $y[i]$ , usamos load byte unsigned, que coloca o carácter em  $\$t2$ :

```
lbu    $t2, 0($t1)  # $t2 = y[i]
```

Um cálculo de endereço semelhante coloca o endereço de  $x[i]$  em  $\$t3$ , e depois o carácter em  $\$t2$  é armazenado nesse endereço.

```
add    $t3,$s0,$a0  # endereço de x[i] em $t3  
sb    $t2, 0($t3)  # x[i] = y[i]
```

Em seguida, saímos do loop se o carácter foi 0; ou seja, esse é o último carácter da string:

```
beq    $t2,$zero,L2  # se y[i] == 0, vai para L2
```

Se não, incrementamos  $i$  e voltamos ao loop:

```
addi   $s0, $s0,1    # i = i + 1  
j      L1           # vai para L1
```

Se não voltamos, então esse foi o último caracter da string; restauramos \$s0 e o stack pointer, para depois retornar.

```
L2:    lw $s0, 0($sp)      # y[i] == 0: fim da string.  
          # Restaura $s0 antigo  
    addi $sp,$sp,4      # retira 1 palavra da pilha  
    jr $ra               # retorna
```

As cópias de string normalmente utilizam ponteiros no lugar de arrays em C, para evitar as operações com *i* no código anterior. Veja, na Seção 2.14, uma explicação sobre arrays e ponteiros.

Como o procedimento `strcpy` anterior é um procedimento folha, o compilador poderia alocar *i* a um registrador temporário e evitar as operações de salvar e restaurar \$s0. Por essa razão, em vez de pensar nos registradores \$t como sendo apenas para valores temporários, podemos pensar neles como registradores que o procedimento chamado deve utilizar sempre que for conveniente. Quando um compilador encontra um procedimento folha, ele esgota todos os registradores temporários antes de usar registradores que precisa salvar.

## Caracteres e strings em Java

*Unicode* é uma codificação universal dos alfabetos da maior parte das linguagens humanas. A [Figura 2.16](#) é uma lista de alfabetos Unicode; existem tantos *alfabetos* em Unicode quanto *símbolos* úteis em ASCII. Para ser mais específico, Java utiliza Unicode para os caracteres. Como padrão, ela utiliza 16 bits a fim de representar um caracter.

Latim	Malaiala	Apurahuano	Pontuação Geral
Grego	Sinhala	Khmer	Letras de Modificação de Espaço
Cirílico	Tailandês	Mongol	Símbolos de Moedas
Armênio	Laociano	Limbu	Marcas Diacríticas Combinadas
Hebraico	Tibetano	Tai Le	Marcas para Símbolos Combinadas
Árabe	Birmanês	Kangxi	Superescritos e Subescritos
Sírio	Georgiano	Hiragana	Formas de Números
Taana	Hangul	Katakana	Operadores Matemáticos
Devanágari	Etiópe	Bopomofo	Símbolos Matemáticos Alfanuméricos
Bengali	Cherokee	Kanbun	Braille
Gurmukhi	Silábico Unificado dos Aborígenes Canadenses	Shavian	Reconhecimento Ótico de Caracteres
Gujarati	Ogham	Osmanya	Símbolos Musicais Bizantinos
Oriá	Runic	Silabário Cipriota	Símbolos Musicais
Tamil	Tagalo	Símbolos Tai Xuan Jing	Setas
Telugu	Hanunoo	Símbolos I Ching	Elementos de Bloco
Canarês	Buhid	Números Egeus	Formas Geométricas

**FIGURA 2.16 Exemplos de alfabetos em Unicode.**

O Unicode versão 4.0 possui mais de 160 “blocos”, que é o nome para uma coleção de símbolos. Cada bloco é um múltiplo de 16. Por exemplo, Grego começa em 0370<sub>hexa</sub> e Cirílico em 0400<sub>hexa</sub>. As três primeiras colunas mostram 48 blocos que correspondem a linguagens humanas em ordem numérica aproximada no Unicode. A última coluna possui 16 blocos que são multilíngues e não estão em ordem. Uma codificação de 16 bits, chamada UTF-16, é o padrão. Uma codificação de tamanho variável, chamada UTF-8, mantém o subconjunto ASCII como 8 bits e utiliza 16 ou 32 bits para os outros caracteres. UTF-32 utiliza 32 bits por caractere. Para saber mais sobre isso, consulte [www.unicode.org](http://www.unicode.org).

O conjunto de instruções do MIPS possui instruções explícitas para carregar e armazenar quantidades de 16 bits, chamadas *halfwords*. Load half (1h) lê uma halfword da memória, colocando-a nos 16 bits mais à direita de um registrador. Assim como load byte, load half (1h) trata a halfword como um número com sinal e, portanto, estende o sinal para preencher os 16 bits mais à esquerda do registrador, enquanto load halfword unsigned (1hu) trabalha com inteiros sem sinal. Assim, 1hu é o mais comum dos dois. Store half (sh) separa a halfword correspondente aos 16 bits mais à direita de um registrador e a escreve na memória. Copiamos uma halfword com a sequência

```
lhu $t0,0($sp) # Lê halfword (16 bits) da origem  
sh $t0,0($gp) # Escreve halfword (16 bits) no destino
```

As strings são uma classe padrão do Java, com suporte interno especial e métodos predefinidos para concatenação, comparação e conversão. Ao contrário da linguagem C, o Java inclui uma palavra que indica o tamanho da string, semelhante aos arrays Java.

## Detalhamento

O software do MIPS tenta manter a pilha alinhada em endereços de palavra, permitindo que o programa sempre use `lw` e `sw` (que precisam estar alinhados) para acessar a pilha. Essa convenção significa que uma variável `char` alocada na pilha ocupa 4 bytes, embora precise de menos. Contudo, uma variável `string` ou um array de bytes em C agrupará 4 bytes por palavra, e uma variável `string` ou array de shorts em Java agrupará 2 halfwords por palavra.

## Detalhamento

Refletindo a natureza internacional da Web, a maioria das páginas web hoje utiliza Unicode ao invés de ASCII.

## Verifique você mesmo

- I. Quais das seguintes afirmações sobre caracteres e strings em C e Java são verdadeiras?
  1. Uma string em C utiliza cerca da metade da memória da mesma string em Java.
  2. Strings são apenas um nome informal para arrays de uma única dimensão de caracteres em C e Java.
  3. As strings em C e Java utilizam null (0) para marcar o fim de uma string.
  4. As operações sobre strings, como saber seu tamanho, são mais rápidas em C do que em Java.
- II. Que tipo de variável que pode conter  $1.000.000.000_{\text{dec}}$  ocupa mais espaço na memória?
  1. `int` em C

2. string em C
3. string em Java

## 2.10. Endereçamento no MIPS para operandos imediatos e endereços de 32 bits

Embora manter todas as instruções MIPS com 32 bits simplifique o hardware, existem ocasiões em que seria conveniente ter uma constante de 32 bits ou endereço de 32 bits. Esta seção começa com a solução geral para constantes grandes e depois apresenta as otimizações para endereços de instruções usados em desvios condicionais e jumps.

### Operandos imediatos de 32 bits

Embora as constantes normalmente sejam curtas e caibam em um campo de 16 bits, às vezes elas são maiores. O conjunto de instruções MIPS inclui a instrução *load upper immediate* (*lui*) especificamente para atribuir os 16 bits mais altos de uma constante a um registrador, permitindo que uma instrução subsequente atribua os 16 bits mais baixos da constante. A [Figura 2.17](#) mostra a operação de *lui*.

A versão do código de máquina <code>lui \$t0, 255</code>	# \$t0 é o registro 8:
001111	00000 01000 0000 0000 1111 1111
Conteúdos do registro \$t0 depois de executar <code>lui \$t0, 255</code> :	
0000 0000 1111 1111	0000 0000 0000 0000

**FIGURA 2.17** O efeito da instrução *lui*.

A instrução *lui* transfere o valor do campo de constante imediata de 16 bits para os 16 bits mais à esquerda do registrador, preenchendo os 16 bits de menor ordem (direita) com 0s.

### Carregando uma constante de 32 bits

#### Exemplo

Qual é o código assembly do MIPS para carregar esta constante de 32 bits no registrador `$s0`?

```
0000 0000 0011 1101 0000 1001 0000 0000
```

## Resposta

Primeiro, carregaríamos os 16 bits mais altos, que é 61 em decimal, usando a instrução lui:

```
lui $s0, 61 # 61 decimal = 0000 0000 0011 1101 binário
```

O valor do registrador \$s0 depois disso é

```
0000 0000 0011 1101 0000 0000 0000 0000
```

O próximo passo é acrescentar os 16 bits inferiores, cujo valor decimal é 2304:

```
ori $s0, $s0, 2304 # 2304 decimal = 0000 1001 0000 0000
```

O valor final no registrador \$s0 é o valor desejado:

```
0000 0000 0011 1101 0000 1001 0000 0000
```

## Interface hardware/software

Tanto o compilador quanto o montador precisam desmembrar constantes grandes em partes e depois remontá-las em um registrador. Como você poderia esperar, a restrição de tamanho do campo imediato pode ser um problema para endereços de memória em loads e stores, e também para constantes em instruções imediatas. Se esse trabalho recair para o montador, como acontece para o software do MIPS, então o montador precisa ter um registrador temporário disponível, onde criará valores longos. Esse é um

motivo para o registrador \$at (assembler temporary), que é reservado para o montador.

Logo, a representação simbólica da linguagem de máquina do MIPS não está mais limitada pelo hardware, mas a qualquer coisa que o criador de um montador decidir incluir (Seção 2.12). Vamos examinar o hardware de perto para explicar a arquitetura do computador, indicando quando usarmos a linguagem avançada do montador que não se encontra no processador.

## Detalhamento

A criação de constantes de 32 bits requer cuidado. A instrução addi copia o bit mais à esquerda do campo imediato de 16 bits da instrução para todos os 16 bits mais altos de uma palavra. O operador *lógico ou imediato*, da Seção 2.6, carrega 0s nos 16 bits superiores e, portanto, é usado pelo montador em conjunto com lui para criar constantes de 32 bits.

## Endereçamento em desvios condicionais e jumps

As instruções de jump no MIPS possuem o endereçamento mais simples possível. Elas utilizam o último formato de instrução do MIPS, chamado *tipo J*, que consiste em 6 bits para o campo de operação e o restante dos bits para o campo de endereço. Assim,

```
j    10000    # vai para a posição 10000
```

poderia ser gerada neste formato (normalmente, isso é um pouco mais complicado, como veremos):

2	10000
6 bits	26 bits

em que o valor do código da operação de jump é 2 e o endereço destino é 10000.

Ao contrário da instrução de jump, a instrução de desvio condicional precisa

especificar dois operandos além do endereço de desvio. Assim,

```
bne    $s0,$s1,Exit  # vai para Exit se $s0 ≠ $s1
```

é gerada nesta instrução, deixando apenas 16 bits para o endereço de desvio:

5	16	17	Exit
6 bits	5 bits	5 bits	16 bits

Se os endereços do programa tivessem de caber nesse campo de 16 bits, nenhum programa poderia ser maior do que  $2^{16}$ , que é muito pequeno para ser uma opção real nos dias atuais. Uma alternativa seria especificar um registrador que sempre seria somado ao endereço de desvio, de modo que uma instrução de desvio pudesse calcular o seguinte:

Contador de programa = Registrador + Endereço de desvio

Essa soma permite que o contador de programa tenha até  $2^{32}$  bits e ainda possa usar desvios condicionais, solucionando o problema do tamanho do endereço de desvio. A questão, portanto, é: qual registrador?

A resposta vem da observação de como os desvios condicionais são usados. Eles são encontrados em loops e em instruções *if*, de modo que costumam desviar para uma instrução próxima. Por exemplo, cerca de metade de todos os desvios condicionais nos benchmarks SPEC vão para locais a menos de 16 instruções de distância. Como o *contador de programa* (PC) contém o endereço da instrução atual, podemos desviar dentro de  $\pm 2^{15}$  palavras da instrução atual se usarmos o PC como o registrador a ser somado ao endereço. Quase todos os loops e as instruções *if* são muito menores do que  $2^{16}$  palavras, de modo que o PC é a opção ideal.

Essa forma de endereçamento de desvio é denominada **endereçamento relativo ao PC**. Conforme veremos no [Capítulo 4](#), é conveniente que o hardware incremente o PC desde cedo, a fim de que aponte para a próxima instrução. Logo, o endereço MIPS, na realidade, é relativo ao endereço da instrução

seguinte (PC + 4), em vez da instrução atual (PC). Este é outro exemplo de tornar o **caso comum veloz**, que neste caso significa endereçar instruções próximas.

### endereçamento relativo ao PC

Um regime de endereçamento em que o endereço é a soma do *contador de programa* (PC) e uma constante na instrução.



## CASO COMUM VELOZ

Como na maioria dos computadores atuais, o MIPS utiliza o endereçamento relativo ao PC para todos os desvios condicionais, pois o destino dessas instruções provavelmente estará próximo do desvio. Por outro lado, instruções de jump-and-link chamam procedimentos que não têm motivo para estarem próximos à chamada e, por isso, normalmente utilizam outras formas de endereçamento. Logo, a arquitetura MIPS oferece endereços longos para chamadas de procedimento, usando o formato do tipo J para instruções de jump e jump-and-link.

Como todas as instruções MIPS possuem 4 bytes de extensão, o MIPS aumenta a distância do desvio fazendo com que o endereçamento relativo ao PC se refira ao número de *palavras* até a próxima instrução, no lugar do número de bytes. Assim, o campo de 16 bits pode se desviar para uma distância quatro vezes maior, interpretando o campo como um endereço relativo à palavra, e não um endereço relativo a byte. De modo semelhante, o campo de 26 bits nas instruções de jump também é um endereço de palavra, significando que representa um endereço de byte com 28 bits.

## Detalhamento

Como o contador de programa (PC) utiliza 32 bits, 4 bits precisam vir de outro lugar para os jumps. A instrução de jump do MIPS substitui apenas os 28 bits menos significativos do PC, deixando os 4 bits mais significativos inalterados. O loader e o link-editor (Seção 2.12) precisam ter cuidado para evitar colocar um programa entre um limite de endereços de 256MB (64 milhões de instruções); caso contrário, um jump precisa ser substituído por uma instrução de jump register precedida por outras instruções, a fim de carregar o endereço de 32 bits inteiro em um registrador.

## Mostrando o offset do desvio em linguagem de máquina

### Exemplo

O loop *while* da Seção 2.7 foi compilado para este código em assembly do MIPS:

```
Loop:s11    $t1,$s3,2      # Reg. temporário $t1 = 4 * I
            add    $t1,$t1,$s6      # $t1 = endereço de save[i]
            lw     $t0,0($t1)      # Reg. temporário $t0 = save[i]
            bne   $t0,$s5, Exit    # vai para Exit se save[i] ≠ k
            addi  $s3,$s3,1      # i = i + 1
            j     Loop           # vai para Loop
Exit:
```

Se consideramos que o loop inicia na posição 80000 da memória, qual é o código de máquina do MIPS para esse loop?

### Resposta

As instruções montadas e seus endereços são:

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8		0	
80012	5	8	21		2	
80016	8	19	19		1	
80020	2			20000		
80024	...					

Lembre-se de que as instruções do MIPS possuem endereços em bytes, de modo que os endereços das palavras sequenciais diferem em 4, a quantidade de bytes em uma palavra. A instrução `bne` na quarta linha acrescenta 2 palavras ou 8 bytes ao endereço da instrução *seguinte* (80016), especificando o destino do desvio em relação à instrução seguinte ( $8 + 80016$ ), e não em relação à instrução de desvio ( $12 + 80012$ ) ou ao uso do endereço de destino completo (80024). A instrução de salto na última linha utiliza o endereço completo ( $20000 \times 4 = 80000$ ), correspondente ao rótulo `Loop`.

## Interface hardware/software

A maioria dos desvios condicionais é feita para um local nas proximidades, mas, ocasionalmente, eles se desviam para um ponto mais distante do que pode ser representado nos 16 bits da instrução de desvio condicional. O montador vem ao auxílio como fez com endereços ou constantes grandes: ele insere um jump incondicional para o destino do desvio, e inverte a condição de modo que o desvio decida se irá pular o jump.

## Desviando para um lugar mais distante

### Exemplo

Dado um desvio em que o registrador `$s0` é igual ao registrador `$s1`,

```
beq    $s0, $s1, L1
```

substitua-o por um par de instruções que ofereça uma distância de desvio

muito maior.

## Resposta

Estas instruções substituem o desvio condicional com endereço curto:

```
bne    $s0, $s1, L2
j      L1
L2:
```

## Resumo dos modos de endereçamento no MIPS

As diversas formas de endereçamento geralmente são denominadas **modos de endereçamento**. A [Figura 2.18](#) mostra como os operandos são identificados para cada modo de endereçamento. Os modos de endereçamento do MIPS são os seguintes:

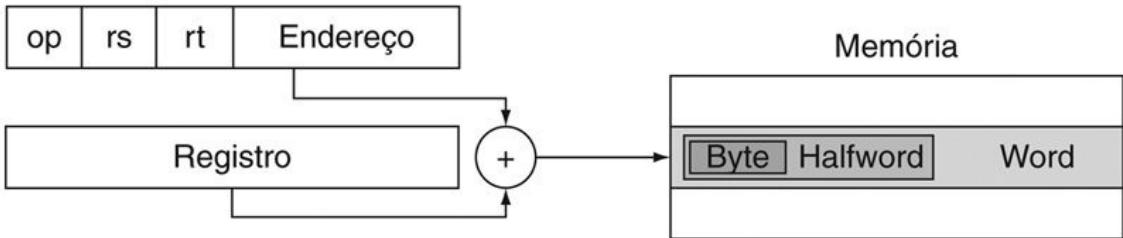
1. Endereçamento imediato



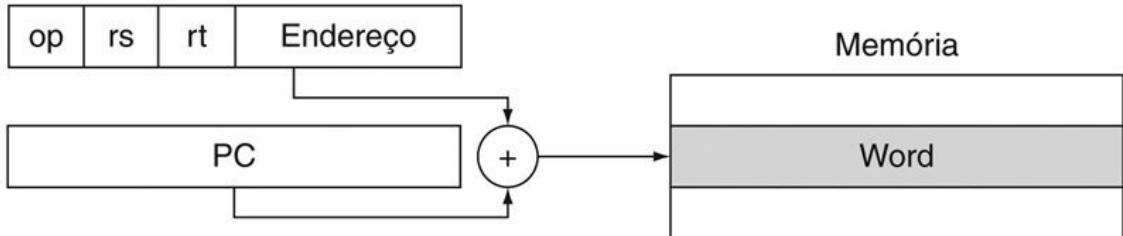
2. Endereçamento em registrador



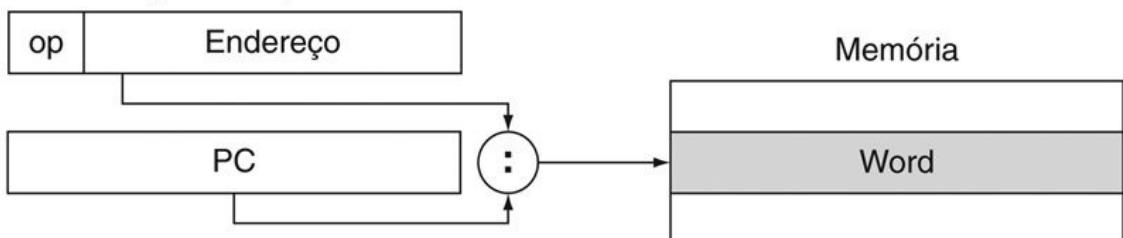
3. Endereçamento de base



4. Endereçamento relativo ao PC



5. Endereçamento pseudodireto



**FIGURA 2.18 Ilustração dos cinco modos de endereçamento do MIPS.**

Os operandos estão sombreados na figura. O operando do modo 3 está na memória, enquanto o operando para o modo 2 é um registrador. Observe que as versões de load e store acessam bytes, halfwords ou palavras. Para o modo 1, o operando é formado pelos 16 bits da própria instrução. Os modos 4 e 5 endereçam as instruções na memória, com o modo 4

adicionando um endereço de 16 bits deslocado à esquerda em 2 bits ao PC, e o modo 5 concatenando um endereço de 26 bits deslocado à esquerda em 2 bits com os 4 bits superiores do PC. Observe que uma única operação pode usar mais de um modo de endereçamento. Add, por exemplo, utiliza um endereçamento imediato (`addi`) e por registrador (`add`).

1. *Endereçamento imediato*, em que o operando é uma constante dentro da própria instrução.
2. *Endereçamento em registrador*, no qual o operando é um registrador.
3. *Endereçamento de base ou deslocamento*, em que o operando está no local de memória cujo endereço é a soma de um registrador e uma constante na instrução.
4. *Endereçamento relativo ao PC*, no qual o endereço de desvio é a soma do PC e uma constante na instrução.
5. *Endereçamento pseudodireto*, em que o endereço de jump são os 26 bits da instrução concatenados com os bits mais altos do PC.

## modo de endereçamento

Um dos diversos regimes de endereçamento delimitados por seu uso variado de operandos e/ou endereços.

## Interface hardware/software

Embora tenhamos mostrado a arquitetura MIPS como tendo endereços de 32 bits, quase todos os microprocessadores (incluindo o MIPS) possuem extensões de endereço de 64 bits ([Seção 2.17](#)). Essas extensões foram a resposta às necessidades de software para programas maiores. O processo de extensão do conjunto de instruções permite que as arquiteturas se expandam de modo que o software possa prosseguir de forma compatível para a próxima geração da arquitetura.

## Decodificando a linguagem de máquina

Às vezes, você é forçado a usar engenharia reversa na linguagem de máquina para criar o código assembly original. Um exemplo é quando se examina um “dump de memória”. A [Figura 2.19](#) mostra a codificação dos campos para a linguagem de máquina do MIPS. Essa figura ajuda na tradução manual entre o

assembly e a linguagem de máquina.

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)	formato R	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	F1Pt						
3(011)								
4(100)	load byte	load half	lw1	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	sw1	store word			swr	
6(110)	load linked word	lwcl						
7(111)	store cond. word	swcl						

op(31:26)=010000 (TLB), 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)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

op(31:26)=000000 (format R), funct(5:0)								
2-0 5-3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	sraw
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mfl o	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

**FIGURA 2.19 Codificação de instruções do MIPS.**

Essa notação indica o valor de um campo por linha e por coluna.

Por exemplo, a parte superior da figura mostra **load word** na linha número 4 (100<sub>bin</sub> para os bits 31-29 da instrução) e a 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>. Formato R na linha 0 e coluna 0 (op = 000000<sub>bin</sub>) é definido na parte inferior da figura. Logo, **subtract** na linha 4 e coluna 2 da seção inferior significa que o campo funct (bits 5-0) da instrução é 100010<sub>bin</sub> e o campo op (bits 31-26) é 000000<sub>bin</sub>.

O valor do floating point na linha 2, coluna 1, é definido na [Figura 3.18](#), no [Capítulo 3](#). bltz/gez é o opcode para quatro instruções encontradas no Apêndice A: bltz, bgez, bltzal e bgezal. Este capítulo descreve as instruções indicadas com nome completo usando destaque, enquanto o [Capítulo 3](#) descreve as instruções indicadas em mnemônicos do mesmo jeito. O Apêndice A abrange todas as instruções.

## Decodificando a linguagem de máquina

### Exemplo

Qual é a instrução em assembly correspondente a esta instrução de máquina?

00af8020hexa

### Resposta

O primeiro passo na conversão de hexadecimal para binário é encontrar os campos op:

(Bits: 31 28 26    5 2 0)

0000 0000 1010 1111 1000 0000 0010 0000

Examinamos o campo op para determinar a operação. Consultando a Figura 2.19, quando os bits 31-29 são 000 e os bits 28-26 são 000, essa é uma instrução no Formato R. Vamos reformatar a instrução binária para campos no Formato R, listado na Figura 2.20:

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

Nome	Campos						Comentários
Tamanho do campo	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Todas as instruções do MIPS têm 32 bits
Formato R	op	rs	rt	rd	shamt	funct	Formato das instruções aritméticas
Formato I	op	rs	rt	endereço/imediato			Formato imediato para transferências e desvios
Formato J	op	endereço de destino					Formato das instruções de jump

**FIGURA 2.20 Formatos das instruções do MIPS.**

A parte inferior da Figura 2.19 determina a operação de uma instrução no Formato R. Nesse caso, os bits 5-3 são 100 e os bits 2-0 são 000, o que significa que esse padrão binário representa uma instrução add.

Decodificamos as demais instruções examinando os valores de campo. Os valores decimais são 5 para o campo rs, 15 para rt, 16 para rd (shamt não é usado). A Figura 2.14 diz que esses números representam os registradores \$a1, \$a7 e \$s0. Agora, podemos mostrar a instrução assembly:

```
add $s0, $a1, $t7
```

A [Figura 2.20](#) mostra todos os formatos de instrução do MIPS. A [Figura 2.1](#) mostrou a linguagem assembly do MIPS revelada neste capítulo; a outra parte ainda oculta das instruções MIPS trata principalmente de aritmética e números reais, que serão abordados no próximo capítulo.

## Verifique você mesmo

- Qual é o intervalo de endereços para desvios condicionais no MIPS ( $K = 1024$ )?
  - Endereços entre 0 e  $64K - 1$
  - Endereços entre 0 e  $256K - 1$
  - Endereços desde cerca de  $32K$  antes do desvio até cerca de  $32K$  depois
  - Endereços desde cerca de  $128K$  antes do desvio até cerca de  $128K$  depois
- Qual é o intervalo de endereços para jump e jump and link no MIPS ( $M = 1024K$ )?
  - Endereços entre 0 e  $64M - 1$
  - Endereços entre 0 e  $256M - 1$
  - Endereços desde cerca de  $32M$  antes do desvio até cerca de  $32M$

depois

4. Endereços desde cerca de 128M antes do desvio até cerca de 128M depois
5. Qualquer lugar dentro de um bloco de 64M endereços, em que o PC fornece os 6 bits mais altos
6. Qualquer lugar dentro de um bloco de 256M endereços, em que o PC fornece os 4 bits mais altos

III. Qual é a instrução em assembly do MIPS correspondente à instrução de máquina com o valor 0000 0000<sub>hexa</sub>?

1. j
2. Formato R
3. addi
4. sll
5. mfc0
6. Opcode indefinido: não existe uma instrução válida que corresponda a 0.

## 2.11. Paralelismo e instruções: Sincronização

A **execução paralela** é mais fácil quando as tarefas são independentes, mas frequentemente elas precisam cooperar. A cooperação normalmente significa que algumas tarefas estão escrevendo novos valores que outras precisam ler. Para saber quando uma tarefa terminou de escrever, de modo que é seguro que outra tarefa leia, elas precisam de sincronização. Se elas não estiverem sincronizadas, haverá um perigo de **data race**, em que os resultados do programa podem mudar, dependendo de como os eventos ocorram.



### PARALELISMO

#### **data race**

Dois acessos à memória formam uma data race se eles forem de threads diferentes para o mesmo local, pelo menos um é de escrita, e eles ocorrem um após o outro.

Por exemplo, lembre-se da analogia citada no [Capítulo 1](#) dos oito repórteres escrevendo um artigo. Suponha que um repórter precise ler todas as seções anteriores antes de escrever uma conclusão. Logo, temos de saber quando os

outros repórteres terminaram suas seções, de modo que ele não se preocupe se será alterado depois disso. Ou seja, é melhor que eles sincronizem a escrita e leitura de cada seção, para que a conclusão seja coerente com o que é impresso nas seções anteriores.

Na computação, os mecanismos de sincronização normalmente estão embutidos em rotinas de software em nível de usuário que contam com as instruções de sincronização fornecidas pelo hardware. Nesta seção, focalizamos a implementação das operações de sincronização *lock* e *unlock*. Lock e unlock podem ser usados facilmente para criar regiões nas quais apenas um único processador possa operar, algo chamado *exclusão mútua*, além de implementar mecanismos de sincronização mais complexos.

A habilidade fundamental que exigimos para implementar a sincronização em um multiprocessador é um conjunto de primitivos de hardware com a capacidade de ler e modificar um local de memória *atomicamente*. Ou seja, nada mais pode se interpor entre a leitura e a escrita do local da memória. Sem essa capacidade, o custo da montagem de primitivos de sincronização básicos será muito alto e aumentará à medida que crescer a quantidade de processadores.

Existem diversas formulações alternativas das primitivas de hardware básicas, todas oferecendo a capacidade de ler e modificar um local atomicamente, junto com algum modo de dizer se a leitura e escrita foram realizadas atomicamente. Em geral, os arquitetos não esperam que os usuários empreguem as primitivas de hardware básicas, mas em vez disso esperam que as primitivas sejam usadas pelos programadores de sistemas para montar uma biblioteca de sincronização, um processo que normalmente é complexo e intrincado.

Vamos começar com uma primitiva de hardware desse tipo, mostrando como ela pode ser usada para montar uma primitiva de sincronização básica. Uma operação típica para a montagem de operações de sincronização é a *troca atômica* ou *swap atômico*, que troca um valor em um registrador por um valor na memória.

Para ver como usar isso a fim de montar uma primitiva de sincronização básica, suponha que queremos montar um bloqueio simples, em que o valor 0 é usado para indicar que o bloqueio está livre e 1 é usado para indicar que o bloqueio está indisponível. Um processador tenta definir o bloqueio realizando uma troca de 1, que está em um registrador, com o endereço de memória correspondendo ao bloqueio. O valor retornado da instrução de troca é 1 se algum outro processador já tiver solicitado acesso, e 0 em caso contrário. No segundo caso, o valor também é trocado para 1, impedindo que qualquer outra

troca em outro processador também recupere um 0.

Por exemplo, considere dois processadores que tentam cada um realizar a troca simultaneamente; essa race é interrompida, pois exatamente um dos processadores realizará a troca primeiro, retornando 0, e o segundo processador retornará 1 quando realizar a troca. A chave para o uso da primitiva de troca para implementar a sincronização é que a operação seja atômica: a troca é indivisível, e duas trocas simultâneas serão ordenadas pelo hardware. É impossível que dois processadores tentando definir a variável de sincronização dessa maneira pensem que definiram a variável simultaneamente.

A implementação de uma única operação de memória atômica apresenta alguns desafios no projeto do processador, pois requer uma leitura e uma escrita na memória em uma única instrução ininterrupta.

Uma alternativa é ter um par de instruções em que a segunda instrução retorna um valor, mostrando se o par de instruções foi executado como se fosse atômico. O par de instruções é efetivamente atômico se parecer que todas as outras operações executadas por qualquer processador ocorreram antes ou depois do par. Assim, quando um par de instruções é efetivamente atômico, nenhum outro processador pode alterar o valor entre o par de instruções.

No MIPS, esse par de instruções inclui um load especial, chamado *load vinculado*, e um store especial, chamado *store condicional*. Essas instruções são usadas em sequência: se o conteúdo do local de memória especificado pelo load vinculado for alterado antes que ocorra o store condicional para o mesmo endereço, então o store condicional falha. O store condicional é definido para armazenar o valor de um registrador na memória e alterar o valor desse registrador para 1 se tiver sucesso e para 0 se falhar. Como o load vinculado retorna o valor inicial, e o store condicional retorna 1 somente se tiver sucesso, a sequência a seguir implementa uma troca atômica no local de memória especificado pelo conteúdo de \$s1:

```
again: addi $t0,$zero,1      ;copia valor da troca
      li    $t1,0($s1)      ;load linked
      sc    $t0,0($s1)      ;store condicional
      beq   $t0,$zero,again ;desvia se store falhar
      add   $s4,$zero,$t1    ;coloca valor de load em $s4
```

Sempre que um processador intervém e modifica o valor na memória entre as instruções `l1` e `sc`, o script retorna 0 em `$t0`, fazendo com que a sequência de código tente novamente. Ao final dessa sequência, o conteúdo de `$s4` e o local de memória especificado por `$s1` foram trocados atomicamente.

## Detalhamento

Embora fosse apresentada para sincronização de multiprocessador, a troca atômica também é útil para o sistema operacional lidar com múltiplos processos em um único processador. A fim de garantir que nada interfira em um único processador, o store condicional também falha se o processador realizar uma troca de contexto entre as duas instruções (Capítulo 5).

Uma vantagem do mecanismo de load vinculado/store condicional é que ele pode ser usado para montar outras primitivas de sincronização, como *compare and swap atômicos* ou *fetch-and-increment atômicos*, que são usados em alguns modelos de programação paralela. Estes envolvem mais instruções entre o `l1` e o `sc`.

Como o store condicional falhará após outro store atraído ao endereço do load vinculado ou em qualquer exceção, deve-se ter o cuidado na escolha de quais instruções são inseridas entre as duas instruções. Em particular, somente instruções registrador-registrador podem ser permitidas com segurança; caso contrário, é possível criar situações de impasse, em que o processador nunca possa completar o `sc`, em consequência das faltas de páginas repetidas. Além disso, o número de instruções entre o load vinculado e o store condicional deve ser pequeno, para minimizar a probabilidade de que um evento não relacionado ou um processador concorrente faça com que o store condicional falhe com frequência.

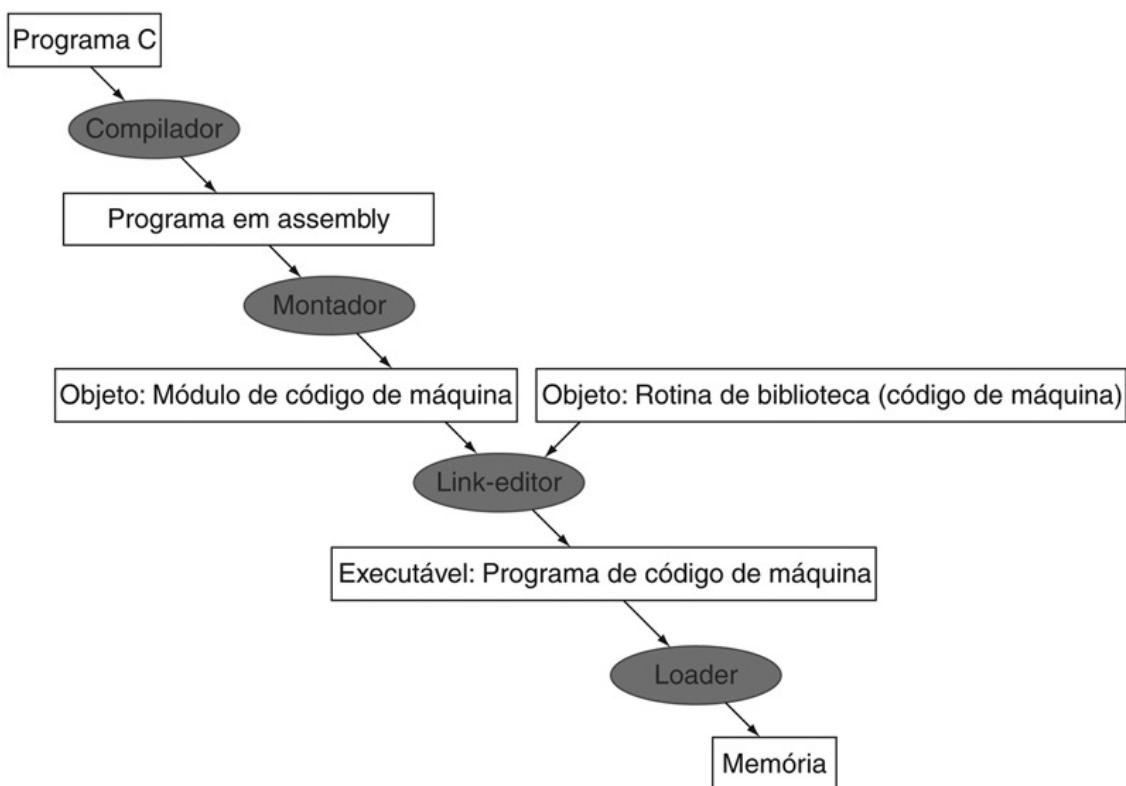
## Verifique você mesmo

Quando você usará primitivas como load vinculado e store condicional?

1. Quando threads em cooperação de um programa paralelo precisarem ser sincronizados para obter um comportamento apropriado para leitura e escrita de dados compartilhados
2. Quando processos em cooperação em um processador precisarem ser sincronizados para a leitura e escrita de dados compartilhados

## 2.12. Traduzindo e iniciando um programa

Esta seção descreve as quatro etapas para a transformação de um programa em C, armazenado em um arquivo no disco, para um programa executando em um computador. A [Figura 2.21](#) mostra a hierarquia de tradução. Alguns sistemas combinam estas etapas para reduzir o tempo de tradução, mas elas são as quatro fases lógicas pelas quais os programas passam. Esta seção segue essa hierarquia de tradução.



**FIGURA 2.21** Uma hierarquia de tradução para a linguagem C.

Um programa em linguagem de alto nível é inicialmente compilado para um programa em assembly e depois montado em um módulo-objeto em linguagem de máquina. O link-editor combina vários módulos com as rotinas de biblioteca para resolver todas as referências. O loader, então, coloca o código de máquina nos locais apropriados da memória, de modo a ser executado pelo processador. Para agilizar o processo de tradução, algumas etapas são puladas ou combinadas. Alguns compiladores produzem módulos-objeto diretamente e alguns sistemas utilizam loaders com link-editores, que realizam as

duas últimas etapas. A fim de identificar o tipo de arquivo, o UNIX segue uma convenção de sufixo para os arquivos: os arquivos-fonte em C são chamados `x.c`, os arquivos em assembly são `x.s`, os arquivos-objeto são `x.o`, as rotinas de biblioteca link-editadas estaticamente são `x.a`, as rotinas de biblioteca link-editadas dinamicamente são `x.so`, e os arquivos executáveis, como padrão, são chamados `a.out`. O MS-DOS utiliza os sufixos `.C`, `.ASM`, `.OBJ`, `.LIB`, `.DLL` e `.EXE` para indicar a mesma coisa.

## Compilador

O compilador transforma o programa C em um *programa em assembly*, uma forma simbólica daquilo que a máquina entende. Os programas em linguagem de alto nível usam muito menos linhas de código do que a linguagem assembly, de modo que a produtividade do programador é muito mais alta.

Em 1975, muitos sistemas operacionais e montadores foram escritos em **linguagem assembly**, pois as memórias eram pequenas e os compiladores eram ineficientes. O aumento de um milhão de vezes na capacidade de memória em um único chip de DRAM reduziu os problemas com tamanho de programas e os compiladores otimizadores de hoje podem produzir programas em assembly quase tão bem quanto um especialista em assembly, e às vezes ainda melhores, para programas grandes.

### linguagem assembly

Uma linguagem simbólica, que pode ser traduzida para o formato binário.

## Montador

Como a linguagem assembly é a interface com o software de nível superior, o montador (ou *assembler*) também pode cuidar de variações comuns das instruções em linguagem de máquina como se fossem instruções propriamente ditas. O hardware não precisa implementar essas instruções; porém, seu aparecimento em assembly simplifica a tradução e a programação. Essas instruções são conhecidas como **pseudoinstruções**.

## pseudoinstrução

Uma variação comum das instruções em assembly, normalmente tratada como se fosse uma instrução propriamente dita.

Como já dissemos, o hardware do MIPS garante que o registrador \$zero sempre tenha o valor 0. Ou seja, sempre que o registrador \$zero é utilizado, ele fornece um 0, e o programador não pode alterar o valor do registrador \$zero. O registrador \$zero é usado para criar a instrução em linguagem assembly que copia o conteúdo de um registrador para outro. Assim, o montador do MIPS aceita esta instrução, embora ela não se encontre na arquitetura do MIPS:

```
move $t0,$t1 # registrador $t0 recebe registrador $t1
```

O montador converte essa instrução em assembly para o equivalente em linguagem de máquina da seguinte instrução:

```
add $t0,$zero,$t1 # registrador $t0 recebe 0 + reg. $t1
```

O montador do MIPS também converte blt (branch on less than) para as duas instruções slt e bne mencionadas no segundo exemplo da [Seção 2.5](#). Outros exemplos são bgt, bge e ble. Ele também converte desvios a locais distantes para um desvio e um jump. Como já dissemos, o montador do MIPS permite que constantes de 32 bits sejam atribuídas a um registrador, apesar do limite de 16 bits das instruções imediatas.

Resumindo, as pseudoinstruções dão ao MIPS um conjunto mais rico de instruções em linguagem assembly do que é implementado pelo hardware. O único custo é reservar um registrador, \$at, para ser usado pelo montador. Se você tiver de escrever programas em assembly, use pseudoinstruções de modo a simplificar seu trabalho. Contudo, para entender a arquitetura do MIPS e ter certeza de que obterá o melhor desempenho, estude as instruções reais do MIPS, encontradas nas [Figuras 2.1 e 2.19](#).

Os montadores também aceitarão números em diversas bases. Além de binário e decimal, eles normalmente aceitam uma base mais sucinta do que o binário, mas que seja convertida facilmente para um padrão de bits. Montadores MIPS

utilizam hexadecimal.

Esses recursos são convenientes, mas a tarefa principal de um montador é a montagem para código de máquina. O montador transforma o programa assembly em um *arquivo objeto*, que é uma combinação de instruções de linguagem de máquina, dados e informações necessárias a fim de colocar instruções corretamente na memória.

Para produzir a versão binária de cada instrução no programa em assembly, o montador precisa determinar os endereços correspondentes a todos os rótulos. Os montadores registram os rótulos utilizados nos desvios e nas instruções de transferência de dados por meio de uma **tabela de símbolos**. Como você poderia esperar, a tabela contém pares de símbolo e endereço.

## tabela de símbolos

Uma tabela que combina nomes de rótulos aos endereços das palavras na memória ocupados pelas instruções.

O arquivo-objeto para os sistemas UNIX normalmente contém seis partes distintas:

- O *cabeçalho do arquivo objeto* descreve o tamanho e a posição das outras partes do arquivo objeto.
- O *segmento de texto* contém o código na linguagem de máquina.
- O *segmento de dados estáticos* contém os dados alocados por toda a vida do programa. (O UNIX permite que os programas usem *dados estáticos*, que são alocados para o programa inteiro ou *dados dinâmicos*, que podem crescer ou diminuir conforme a necessidade do programa. Veja a [Figura 2.13](#).)
- As *informações de relocação* identificam instruções e palavras de dados que dependem de endereços absolutos quando o programa é carregado na memória.
- A *tabela de símbolos* contém os rótulos restantes que não estão definidos, como referências externas.
- As *informações de depuração* contêm uma descrição resumida de como os módulos foram compilados, para o depurador poder associar as instruções de máquina aos arquivos-fonte em C e tornar as estruturas de dados legíveis.

A próxima subseção mostra como juntar rotinas que já foram montadas, como as rotinas de biblioteca.

## **LINK-EDITOR**

O que apresentamos até aqui sugere que uma única mudança em uma linha de um procedimento exige a compilação e a montagem do programa inteiro. A tradução completa é um desperdício terrível de recursos computacionais. Essa repetição é um desperdício principalmente para rotinas de bibliotecas padrão, pois os programadores estariam compilando e montando rotinas que, por definição, quase nunca mudam. Uma alternativa é compilar e montar cada procedimento de forma independente, de modo que uma mudança em uma linha só exija a compilação e a montagem de um procedimento. Essa alternativa requer um novo programa de sistema, chamado **link-editor** ou **linker**, que apanha todos os programas em linguagem de máquina montados independentes e os “remenda”.

### **linker**

Também chamado **link-editor**, é um programa de sistema que combina programas em linguagem de máquina montados de maneira independente e traduz todos os rótulos indefinidos para um arquivo executável.

Existem três etapas realizadas por um link-editor:

1. Colocar os módulos de código e dados simbolicamente na memória.
2. Determinar os endereços dos rótulos de dados e instruções.
3. Remendar as referências internas e externas.

O link-editor utiliza a informação de relocação e a tabela de símbolos em cada módulo- -objeto para resolver todos os rótulos indefinidos. Essas referências ocorrem em instruções de desvio e endereços de dados, de modo que a tarefa desse programa é muito semelhante à de um editor: ele encontra os endereços antigos e os substitui pelos novos. A edição é a origem do nome “link-editor” ou linker para abreviar. O uso de um link-editor faz sentido porque é muito mais rápido remendar o código do que recompilá-lo e remontá-lo.

Se todas as referências externas forem resolvidas, o link-editor em seguida determina os locais da memória que cada módulo ocupará. Lembre-se de que a [Figura 2.13](#), na [Seção 2.8](#), mostra a convenção do MIPS para alocação de programas e dados na memória. Como os arquivos foram montados isoladamente, o montador não poderia saber onde as instruções e os dados do módulo serão colocados em relação a outros módulos. Quando o link-editor coloca um módulo na memória, todas as referências *absolutas*, ou seja,

endereços de memória que não são relativos a um registrador, precisam ser *relocadas* a fim de refletir seu verdadeiro local.

O link-editor produz um **arquivo executável** que pode ser executado em um computador. Normalmente, esse arquivo possui o mesmo formato de um arquivo-objeto, exceto que não contém referências não resolvidas. É possível ter arquivos parcialmente link-editados, como rotinas de biblioteca, que ainda possuem endereços não resolvidos e, portanto, resultam em arquivos-objeto.

## arquivo executável

Um programa funcional no formato de um arquivo-objeto, que não contém referências não resolvidas. Ele pode conter tabelas de símbolos e informações de depuração. Um “executável despido” não contém essa informação. As informações de relocação podem ser incluídas para o loader.

## Link-edição de arquivos-objeto

### Exemplo

Link-edite os dois arquivos-objeto a seguir. Mostre os endereços atualizados das primeiras instruções do arquivo executável gerado. Mostramos as instruções em assembly só para tornar o exemplo comprehensível; na realidade, as instruções seriam números.

Observe que, nos arquivos-objeto, destacamos os endereços e símbolos que precisam ser atualizados no processo de link-edição: as instruções que se referem a endereços dos procedimentos A e B e as instruções que se referem aos endereços das words X e Y.

Cabeçalho do arquivo-objeto			
	Nome	Procedimento A	
	Tamanho do texto	100 <sub>hexa</sub>	
	Tamanho dos dados	20 <sub>hexa</sub>	
Segmento de texto	Endereço	Instrução	
	0	lw \$a0, @(\$gp)	
	4	jal @	
	...	...	
Segmento de dados	0	(X)	
	...	...	
Informações de relocação	Endereço	Tipo de instrução	Dependência

Informações de relocação	Endereço	Tipo de instrução	Dependência
	0	lw	X
	4	jal	B
Tabela de símbolos	Rótulo	Endereço	
	X	—	
	B	—	
<b>Cabeçalho do arquivo-objeto</b>			
	Nome	Procedimento B	
	Tamanho do texto	200 <sub>hexa</sub>	
	Tamanho dos dados	30 <sub>hexa</sub>	
Segmento de texto	Endereço	Instrução	
	0	sw \$a1, 0(\$gp)	
	4	jal 0	
	...	...	
Segmento de dados	0	(Y)	
	...	...	
Informações de relocação	Endereço	Tipo de instrução	Dependência
	0	sw	Y
	4	jal	A
Tabela de símbolos	Rótulo	Endereço	
	Y	—	
	A	—	

## Resposta

O procedimento A precisa encontrar o endereço para a variável cujo rótulo é X, a fim de colocá-lo na instrução load e encontrar o endereço do procedimento B para colocá-lo na instrução jal. O procedimento B precisa do endereço da variável cujo rótulo é Y para a instrução store e o endereço do procedimento A para sua instrução jal.

Pela Figura 2.13, sabemos que o segmento de texto começa no endereço 40 0000<sub>hexa</sub> e o segmento de dados em 1000 0000<sub>hexa</sub>. O texto do procedimento A é colocado no primeiro endereço e seus dados no segundo. O cabeçalho do arquivo-objeto para o procedimento A diz que seu texto possui 100<sub>hexa</sub> bytes e seus dados possuem 20<sub>hexa</sub> bytes, de modo que o endereço inicial para o texto do procedimento B é 40 0100<sub>hexa</sub> e seus dados começam em 1000 0020<sub>hexa</sub>.

<b>Cabeçalho do arquivo executável</b>		
	Tamanho do texto	300 <sub>hexa</sub>

	Tamanho dos dados	50hexa
Segmento de texto	Endereço	Instrução
	0040 0000 <sub>hexa</sub>	lw\$a0, 8000 <sub>hexa</sub> (\$gp)
	0040 0004 <sub>hexa</sub>	jal 40 0100 <sub>hexa</sub>
	...	...
	0040 0100 <sub>hexa</sub>	sw\$a1, 8020 <sub>hexa</sub> (\$gp)
	0040 0104 <sub>hexa</sub>	jal 40 0000 <sub>hexa</sub>
	...	...
Segmento de dados	Endereço	
	1000 0000 <sub>hexa</sub>	(x )
	...	...
	1000 0020 <sub>hexa</sub>	(y )
	...	...

A Figura 2.13 também mostra que o segmento de texto começa no endereço 40 0000<sub>hexa</sub> e o segmento de dados no 1000 0000<sub>hexa</sub>. O texto do procedimento A é colocado no primeiro endereço e seus dados no segundo. O cabeçalho do arquivo objeto para o procedimento A diz que seu texto é 100<sub>hexa</sub> bytes e seus dados 20<sub>hexa</sub> bytes, então o começo do endereço do texto do procedimento B é 40 0100<sub>hexa</sub>, e seus dados começam em 1000 0020<sub>hexa</sub>.

Agora, o link-editor atualiza os campos de endereço das instruções. Ele usa o campo de tipo de instrução para saber o formato do endereço a ser editado. Temos dois tipos aqui:

1. Os jal são fáceis porque utilizam o endereçamento pseudodireto. O jal no 40 0004<sub>hexa</sub> recebe 40 0100<sub>hexa</sub> (o endereço do procedimento B) em seu campo de endereço, e o jal em 40 0104<sub>hexa</sub> recebe 40 0000<sub>hexa</sub> (o endereço do procedimento A) em seu campo de endereço.
2. Os endereços de load e store são mais difíceis, pois são relativos a um registrador de base. Este exemplo utiliza o ponteiro global como registrador de base. A Figura 2.13 mostra que \$gp é inicializado com 1000 8000<sub>hexa</sub>. Para obter o endereço 1000 0000<sub>hexa</sub> (o endereço da palavra X), colocamos 8000<sub>hexa</sub> no campo de endereço da instrução lw, no endereço 40 0000<sub>hexa</sub>. De modo semelhante, colocamos 8020<sub>hexa</sub> no campo de endereço da instrução sw no endereço 40 0100<sub>hexa</sub> para obter o endereço 1000 0020<sub>hexa</sub> (o endereço da palavra Y).

## Detalhamento

Lembre-se de que as instruções MIPS são alinhadas na palavra, de modo que `jal` remove os dois bits da direita para aumentar a faixa de endereços da instrução. Assim, ele usa 26 bits para criar um endereço de byte de 28 bits. Logo, o endereço real nos 26 bits inferiores da instrução `jal` neste exemplo é `10 0040hexa`, em vez de `40 0100hexa`.

## Loader

Agora que o arquivo executável está no disco, o sistema operacional o lê para a memória e o inicia. O **loader** segue estas etapas nos sistemas UNIX:

1. Lê o cabeçalho do arquivo executável para determinar o tamanho dos segmentos de texto e de dados.
2. Cria um espaço de endereçamento grande o suficiente para o texto e os dados.
3. Copia as instruções e os dados do arquivo executável para a memória.
4. Copia os parâmetros (se houver) do programa principal para a pilha.
5. Inicializa os registradores da máquina e define o stack pointer para o primeiro local livre.
6. Desvia para uma rotina de inicialização, que copia os parâmetros para os registradores de argumento e chama a rotina principal do programa.  
Quando a rotina principal retorna, a rotina de inicialização termina o programa com uma chamada `exit` do sistema.

### loader

Um programa de sistema que coloca o programa-objeto na memória principal, de modo que esteja pronto para ser executado.

As Seções A.3 e A.4 no Apêndice A descrevem os link-editores e os loaders com mais detalhes.

## Dinamically Linked Libraries (DLLs)

A primeira parte desta seção descreve a técnica tradicional para a link-edição de bibliotecas antes de o programa ser executado. Embora essa técnica estática seja o modo mais rápido de chamar rotinas de biblioteca, ela possui algumas desvantagens:

- As rotinas de biblioteca se tornam parte do código executável. Se uma nova versão da biblioteca for lançada para reparar os erros ou dar suporte a novos dispositivos de hardware, o programa link-editado estaticamente continua usando a versão antiga.
- Ela carrega todas as rotinas na biblioteca que são chamadas de qualquer lugar no executável, mesmo que essas chamadas não sejam executadas. A biblioteca pode ser grande em relação ao programa; por exemplo, a biblioteca padrão da linguagem C possui 2,5MB.

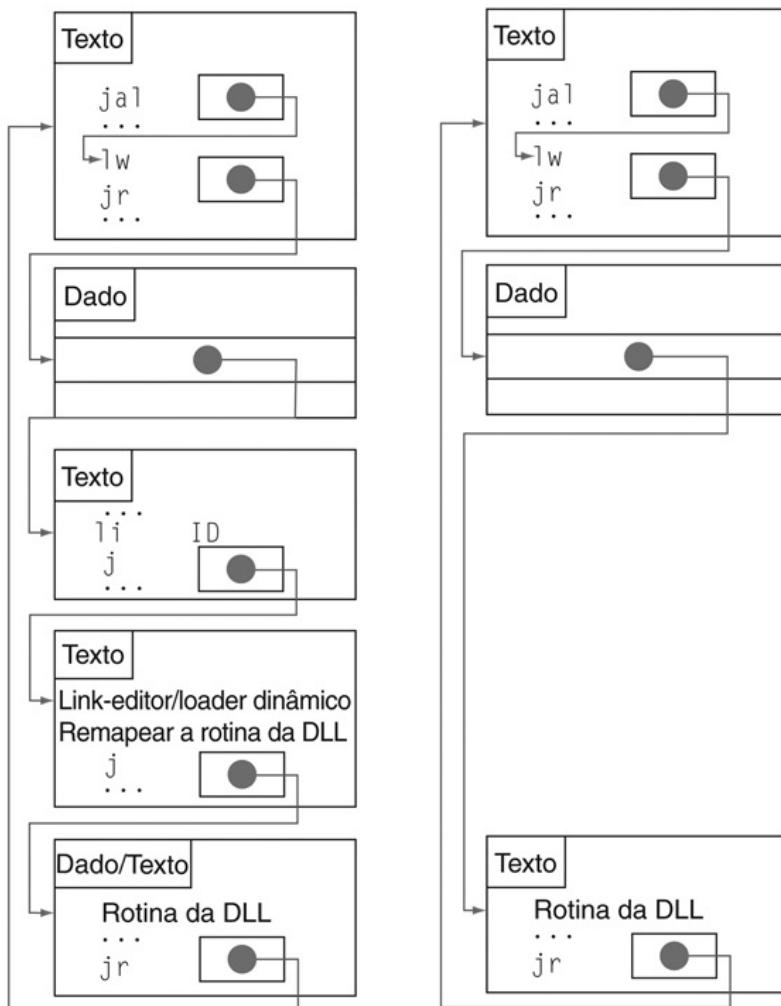
Essas desvantagens levaram às **Dynamic Linked Libraries (DLLs)**, nas quais as rotinas da biblioteca não são link-editadas e carregadas até que o programa seja executado. Tanto o programa quanto as rotinas da biblioteca mantêm informações extras sobre a localização dos procedimentos não locais e seus nomes. Na versão inicial das DLLs, o loader executava um link-editor dinâmico, usando as informações extras no arquivo para descobrir as bibliotecas apropriadas e atualizar todas as referências externas.

## Dynamically Linked Libraries (DLLs)

Rotinas de bit que são vinculadas a um programa durante a execução.

A desvantagem da versão inicial das DLLs era que elas ainda link-editavam todas as rotinas da biblioteca que poderiam ser chamadas, quando apenas algumas são chamadas durante a execução do programa. Essa observação levou à versão da link-edição de procedimento tardio das DLLs, no qual cada rotina só é link-editada *depois* de chamada.

Como muitas inovações em nosso campo, esse truque conta com um certo nível de indireção. A [Figura 2.22](#) mostra a técnica. Ela começa com as rotinas não locais chamando um conjunto de rotinas fictícias no final do programa, com uma entrada por rotina não local. Essas entradas fictícias contêm, cada uma, um jump indireto.



a. Primeiro chamado para a rotina da DLL      b. Chamados subsequentes para a rotina da DLL

### FIGURA 2.22 DLL por meio da link-edição de procedimento tardio.

(a) Etapas para a primeira vez em que uma chamada é feita à rotina da DLL. (b) As etapas para encontrar a rotina, remapeá-la e link-editá-la são puladas em chamadas subsequentes.

Conforme veremos no [Capítulo 5](#), o sistema operacional pode evitar copiar a rotina desejada remapeando-a por meio do gerenciamento de memória virtual.

Na primeira vez em que a rotina da biblioteca é chamada, o programa chama a entrada fictícia e segue o jump indireto. Ele aponta para o código que coloca um número em um registrador para identificar a rotina de biblioteca desejada e depois desvia para o loader com link-editor dinâmico. O loader com link-editor encontra a rotina desejada, remapeia essa rotina e altera o endereço do desvio indireto, de modo a apontar para essa rotina. Depois, ele desvia para ela. Quando

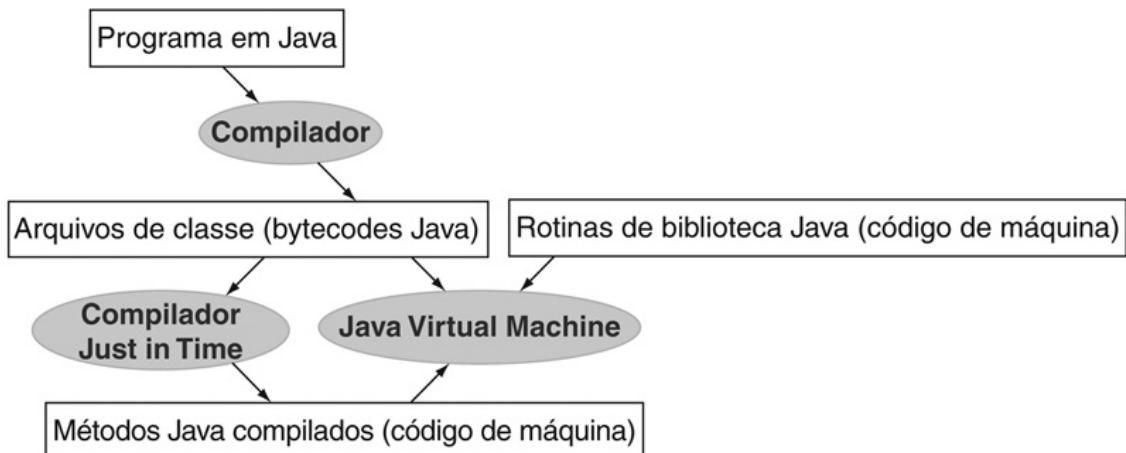
a rotina termina, ele retorna ao local de chamada original. Depois disso, a chamada para rotina de biblioteca desvia indiretamente para a rotina, sem os desvios extras.

Resumindo, as DLLs exigem espaço extra para as informações necessárias à link-edição dinâmica, mas não exigem que as bibliotecas inteiras sejam copiadas ou link-editadas. Elas realizam muito trabalho extra na primeira vez em que uma rotina é chamada, mas executam somente um desvio indireto depois disso. Observe que o retorno da biblioteca não realiza trabalho extra. O Microsoft Windows conta bastante com as DLLs dessa forma e esse também é um modo normal de executar programas nos sistemas UNIX atuais.

## Iniciando um programa Java

A discussão anterior captura o modelo tradicional de execução de um programa, no qual a ênfase está no tempo de execução rápido para um programa voltado a uma arquitetura específica ou mesmo para uma implementação específica dessa arquitetura. Na verdade, é possível executar programas Java da mesma forma que programas C. No entanto, o Java foi inventado com objetivos diferentes. Um deles era funcionar rapidamente e de forma segura em qualquer computador, mesmo que isso pudesse aumentar o tempo de execução.

A [Figura 2.23](#) mostra as etapas típicas de tradução e execução para os programas em Java. Em vez de compilar para assembly de um computador de destino, Java é compilado primeiro para instruções fáceis de interpretar: o conjunto de instruções do **bytecode Java**. Esse conjunto de instruções foi criado para ser próximo da linguagem Java, de modo que essa etapa de compilação seja trivial. Praticamente nenhuma otimização é realizada. Assim como o compilador C, o compilador Java verifica os tipos dos dados e produz a operação apropriada a cada tipo. Os programas em Java são distribuídos na versão binária desses bytecodes.



**FIGURA 2.23** Uma hierarquia de tradução para Java.

Um programa em Java primeiro é compilado para uma versão binária dos bytecodes Java, com todos os endereços definidos pelo compilador. O programa em Java agora está pronto para ser executado no interpretador, chamado Java Virtual Machine (JVM — máquina virtual Java). A JVM link-edita os métodos desejados na biblioteca Java enquanto o programa está sendo executado. Para conseguir melhor desempenho, a JVM pode chamar o compilador Just In Time (JIT), que compila os métodos seletivamente para a linguagem nativa da máquina em que está executando.

## bytecode Java

Instruções de um conjunto de instruções projetado para interpretar programas em Java.

Um interpretador Java, chamado **Java Virtual Machine** (JVM), pode executar os bytecodes Java. Um interpretador é um programa que simula um conjunto de instruções. Não é necessária uma etapa de montagem separada, pois ou a tradução é tão simples que o compilador preenche os endereços ou a JVM os encontra durante a execução.

## Java Virtual Machine (JVM)

O programa que interpreta os bytecodes Java.

A vantagem da interpretação é a portabilidade. A disponibilidade das

máquinas virtuais Java em software significou que muitos puderam escrever e executar programas Java pouco tempo depois que o Java foi anunciado. Hoje, as máquinas virtuais Java são encontradas em centenas de milhões de dispositivos, em tudo desde telefones celulares até navegadores da Internet.

A desvantagem da interpretação é o desempenho fraco. Os avanços incríveis no desempenho dos anos 80 e 90 do século passado tornaram a interpretação viável para muitas aplicações importantes, mas um fator de atraso de 10 vezes, em comparação com os programas C compilados tradicionalmente, tornou o Java pouco atraente para algumas aplicações.

A fim de preservar a portabilidade e melhorar a velocidade de execução, a fase seguinte do desenvolvimento do Java foram compiladores que traduziam *enquanto* o programa estava sendo executado. Esses **compiladores Just In Time (JIT)** normalmente traçam o perfil do programa em execução para descobrir onde estão os métodos “quentes”, e depois os compilam para o conjunto de instruções nativo em que a máquina virtual está executando. A parte compilada é salva para a próxima vez em que o programa for executado, de modo que possa ser executado mais rapidamente cada vez que for executado. Esse equilíbrio entre interpretação e compilação evolui com o tempo, de modo que os programas Java executados com frequência sofrem muito pouco com o trabalho extra da interpretação.

## compilador Just In Time (JIT)

O nome normalmente dado a um compilador que opera durante a execução, traduzindo os segmentos de código interpretados para o código nativo do computador.

À medida que os computadores ficam mais rápidos, de modo que os compiladores possam fazer mais, e os pesquisadores inventam meios melhores de compilar Java durante a execução, a lacuna de desempenho entre Java e C ou C ++ está se fechando.

## Verifique você mesmo

Qual das vantagens de um interpretador em relação a um tradutor você acredita que tenha sido mais importante para os criadores do Java?

1. Facilidade de escrita de um interpretador
2. Melhores mensagens de erro

- 3. Código-objeto menor
- 4. Independência de máquina

## 2.13. Um exemplo de ordenação em C para juntar tudo isso

Um perigo de mostrar o código em assembly em partes é que você não terá ideia de como se parece um programa inteiro em assembly. Nesta seção, deduzimos o código do MIPS a partir de dois procedimentos escritos em C: um para trocar elementos do array e outro para ordená-los.

### O procedimento swap

Vamos começar com o código para o procedimento swap na [Figura 2.24](#). Esse procedimento simplesmente troca os conteúdos de duas posições de memória. Ao traduzir de C para assembly manualmente, seguimos estas etapas gerais:

1. Alocar registradores a variáveis do programa.
2. Produzir código para o corpo do procedimento.
3. Preservar registradores durante a chamada do procedimento.

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

**FIGURA 2.24** Um procedimento em C que troca o conteúdo de duas posições de memória.

Esta subseção utiliza esse procedimento em um exemplo de ordenação.

Esta seção descreve o procedimento swap nessas três partes, concluindo com a junção de todas as partes.

## De registradores para swap

Como mencionamos anteriormente na [Seção 2.8](#), a convenção do MIPS sobre passagem de parâmetros é usar os registradores \$a0, \$a1, \$a2 e \$a3. Como swap tem apenas dois parâmetros, v e k, eles serão encontrados nos registradores \$a0 e \$a1. A única outra variável é temp, que associamos com o registrador \$t0, pois swap é um procedimento folha (Seção “Procedimentos aninhados”). Essa alocação de registradores corresponde às declarações de variável na primeira parte do procedimento swap da [Figura 2.24](#).

## Código do corpo do procedimento swap

As linhas restantes do código em C do swap são

```
temp = v[k];
v[k] = v[k + 1];
v[k + 1] = temp;
```

Lembre-se de que o endereço de memória para o MIPS refere-se ao endereço em *bytes*, e, por isso, as words, na realidade, estão afastadas por 4 bytes. Logo, precisamos multiplicar o índice k por 4 antes de somá-lo ao endereço. *Esquecer que os endereços de palavras sequenciais diferem em 4, em vez de 1, é um erro comum na programação em assembly*. Logo, o primeiro passo é obter o endereço de v[k] multiplicando k por 4 por meio de um deslocamento à esquerda por 2:

```

sll      $t1, $a1,2          # reg. $t1 = k * 4
add      $t1, $a0,$t1        # reg. $t1 = v + (k * 4)
                                # reg. $t1 tem o endereço de v[k]

```

Agora, lemos  $v[k]$  para  $\$t1$ , e depois  $v[k + 1]$  somando 4 a  $\$t1$ :

```

lw       $t0, 0($t1)        # reg. $t0 (temp) = v[k]
lw       $t2, 4($t1)        # reg. $t2 = v[k + 1]
                                # refere-se ao próximo elemento de v

```

Agora, armazenamos  $\$t0$  e  $\$t2$  nos endereços trocados:

```

sw       $t2, 0($t1)        # v[k] = reg. $t2
sw       $t0, 4($t1)        # v[k + 1] = reg. $t0 (temp)

```

Até agora, alocamos registradores e escrevemos o código de modo a realizar as operações do procedimento. O que está faltando é o código para preservar os registradores salvos usados dentro do swap. Como não estamos usando registradores salvos nesse procedimento folha, não há nada para preservar.

## O procedimento swap completo

Agora, estamos prontos para a rotina inteira, que inclui o rótulo do procedimento e o jump de retorno. A fim de facilitar o acompanhamento, identificamos na [Figura 2.25](#) cada bloco de código com sua finalidade no procedimento.

Corpo do procedimento		
swap:	sll \$t1, \$a1, 2 add \$t1, \$a0, \$t1 lw \$t0, 0(\$t1) lw \$t2, 4(\$t1) sw \$t2, 0(\$t1) sw \$t0, 4(\$t1)	# reg \$t1 = k * 4 # reg \$t1 = v + (k * 4) # reg \$t1 tem o endereço de v[k] # reg \$t0 (temp) = v[k] # reg \$t2 = v[k + 1] # refere-se ao próximo elemento de v # v[k] = reg \$t2 # v[k+1] = reg \$t0 (temp)
Retorno do procedimento		
	jr \$ra	# retorna à rotina que chamou

**FIGURA 2.25** Código assembly do MIPS do procedimento swap na [Figura 2.24](#).

## O procedimento sort

Para garantir que você apreciará o rigor da programação em assembly, vamos experimentar um segundo exemplo, maior. Nesse caso, montaremos uma rotina que chama o procedimento swap. Esse programa ordena um array de inteiros, usando ordenação por bolha ou trocas, que é uma das mais simples, mas não a mais rápida. A [Figura 2.26](#) mostra a versão em C do programa. Mais uma vez, apresentamos esse procedimento em várias etapas, concluindo com o procedimento completo.

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
            swap(v, j);
        }
    }
}
```

**FIGURA 2.26** Um procedimento em C que realiza uma ordenação no array v.

## Alocação de registradores para sort

Os dois parâmetros do procedimento `sort`, `v` e `n`, estão nos registradores de parâmetro `$a0` e `$a1`, e alocamos o registrador `$s0` a `i` e o registrador `$s1` a `j`.

## Código para o corpo do procedimento `sort`

O corpo do procedimento consiste em dois loops *for* aninhados e uma chamada a `swap` que inclui parâmetros. Vamos desvendar o código de fora para o meio.

O primeiro passo de tradução é o primeiro loop *for*:

```
for (i = 0; i <n; i += 1) {
```

Lembre-se de que a instrução *for* em C possui três partes: inicialização, teste de loop e incremento da iteração. É necessário apenas uma instrução para inicializar `i` como 0, a primeira parte da instrução *for*:

```
move $s0, $zero    # i = 0
```

(Lembre-se de que `move` é uma pseudo-instrução fornecida pelo montador para a conveniência do programador em assembly; ver seção “Montador”, anteriormente, neste capítulo.) Também é necessária apenas uma instrução para incrementar `i`, a última parte da instrução *for*:

```
addi $s0, $s0, 1    # i += 1
```

O loop deverá terminar se  $i < n$  *não* for verdadeiro ou, em outras palavras, deverá terminar se  $i \geq n$ . A instrução `set on less than` atribui 1 ao registrador `$t0` para 1 se  $\$s0 < \$a1$ ; caso contrário, ele é 0. Como queremos testar se  $\$s0 \geq \$a1$ , desviamos se o registrador `$t0` for zero. Esse teste utiliza duas instruções:

```
for1tst:slt$t0, $s0, $a1      # reg. $t0 = 0 se $s0 ≥ $a1 (i ≥ n)
      beq    $t0, $zero,exit1   # vá para exit1 se $s0 ≥ $a1 (i ≥ n)
```

O final do loop só retorna para o teste do loop:

```
j forltst      # desvia para o teste do loop mais externo exit1:
```

O código da estrutura do primeiro loop *for* é, então,

```
move    $s0, $zero          # i = 0
forltst:slt$t0, $s0, $a1      # reg. $t0 = 0 se $s0 ≥ $a1 (i ≥ n)
    beq    $t0, $zero,exit1 # vai para exit1 se $s0 ≥ $a1 (i ≥ n)
        ...
        (corpo do primeiro loop for)
        ...
addi    $s0, $s0, 1      # i += 1
j forltst            # salta para teste do loop externo
exit1:
```

*Voilà!* (Os exercícios exploram a escrita de código mais rápido para loops semelhantes.)

O segundo loop *for* se parece com o seguinte em C:

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
```

A parte de inicialização desse loop novamente é uma instrução:

```
addi    $s1, $s0, -1 # j = i - 1
```

O decremento de j no final do loop também tem uma instrução:

```
addi    $s1, $s1, -1 # j -= 1
```

O teste do loop possui duas partes. Saímos do loop se a condição falhar, de modo que o primeiro teste precisa terminar o loop se falhar ( $j < 0$ ):

```
for2tst: slti $t0, $s1, 0      # reg. $t0 = 1 se $s1 < 0 (j < 0)
          bne   $t0, $zero, exit2# vai para exit2 se $s1 < 0 (j < 0)
```

Esse desvio pulará o segundo teste de condição. Se não pular, então  $j \geq 0$ .

O segundo teste termina se  $v[j] > v[j + 1]$  não for verdadeiro, ou seja, termina se  $v[j] \leq v[j + 1]$ . Primeiro, criamos o endereço multiplicando  $j$  por 4 (pois precisamos de um endereço em bytes) e somamos ao endereço base de  $v$ :

```
sll      $t1, $s1, 2      # reg. $t1 = j * 4
          add    $t2, $a0, $t1      # reg. $t2 = v + (j * 4)
```

Agora, lemos o conteúdo de  $v[j]$ :

```
lw      $t3, 0($t2)      # reg. $t3 = v[j]
```

Como sabemos que o segundo elemento é exatamente a palavra seguinte, somamos 4 ao endereço no registrador  $\$t2$  para obter  $v[j + 1]$ :

```
lw      $t4, 4($t2)      # reg. $t4 = v[j + 1]
```

O teste de  $v[j] \leq v[j + 1]$  é o mesmo que  $v[j + 1] \geq v[j]$ , de modo que as duas instruções do teste de saída são

```

    slt      $t0, $t4, $t3      # reg. $t0 = 0 se      $t4 ≥ $t3
    beq      $t0, $zero, exit2 # vai para exit2 se $t4 ≥ $t3

```

O final do loop retorna para o teste do loop interno:

```

j      for2tst      # salta para teste do loop interno

```

Combinando as partes, a estrutura do segundo loop *for* se parece com o seguinte:

```

        addi $s1, $s0, -1      # j = i - 1
for2tst: slti $t0, $s1, 0      # reg. $t0 = 1 se $s1 < 0 (j < 0)
        bne $t0, $zero, exit2 # vai para exit2 se $s1 < 0 (j < 0)
        sll $t1, $s1, 2      # reg. $t1 = j * 4
        add $t2, $a0, $t1      # reg. $t2 = v + (j * 4)
        lw   $t3, 0($t2)      # reg. $t3 = v[j]
        lw   $t4, 4($t2)      # reg. $t4 = v[j + 1]
        slt $t0, $t4, $t3      # reg. $t0 = 0 se $t4 ≥ $t3
        beq $t0, $zero, exit2 # vai para exit2 se $t4 ≥ $t3
        . .
        (corpo do segundo loop for)
        . .
        addi $s1, $s1, -1      # j -= 1
        j for2tst      # salta para teste do loop interno
exit2:

```

## A chamada de procedimento em sort

A próxima etapa é o corpo do segundo loop *for*:

```
swap(v,j);
```

Chamar swap é muito fácil:

```
jal    swap
```

## Passando parâmetros em sort

O problema vem quando queremos passar parâmetros, porque o procedimento sort precisa dos valores nos registradores \$a0 e \$a1, enquanto o procedimento swap precisa que seus parâmetros sejam colocados nesses mesmos registradores. Uma solução é copiar os parâmetros para sort em outros registradores antes do procedimento, deixando os registradores \$a0 e \$a1 disponíveis para a chamada de swap. (Essa cópia é mais rápida do que salvar e restaurar na pilha.) Primeiro, copiamos \$a0 e \$a1 para \$s2 e \$s3 durante o procedimento:

```
move  $s2, $a0      # copia parâmetro $a0 para $s2  
move  $s3, $a1      # copia parâmetro $a1 para $s3
```

Depois, passamos os parâmetros para swap com estas duas instruções:

```
move  $a0, $s2      # primeiro parâmetro de swap é v  
move  $a1, $s1      # segundo parâmetro de swap é j
```

## Preservando registradores em sort

O único código restante é o salvamento e a restauração dos registradores. Com certeza, temos de salvar o endereço de retorno no registrador \$ra, pois sort é um procedimento que foi chamado por outro procedimento. O procedimento sort também utiliza os registradores salvos \$s0, \$s1, \$s2 e \$s3, de modo que precisam ser salvos. O prólogo do procedimento sort, portanto, é

```
addi    $sp,$sp,-20      # cria espaço na pilha para 5 registradores  
sw      $ra,16($sp)     # salva $ra na pilha  
sw      $s3,12($sp)     # salva $s3 na pilha  
sw      $s2, 8($sp)      # salva $s2 na pilha  
sw      $s1, 4($sp)      # salva $s1 na pilha  
sw      $s0, 0($sp)      # salva $s0 na pilha
```

O final do procedimento simplesmente reverte todas essas instruções, depois acrescenta um `jr` para retornar.

## O procedimento sort completo

Agora, juntamos todas as partes na [Figura 2.27](#), tendo o cuidado de substituir as referências aos registradores `$a0` e `$a1` nos loops *for* por referências aos registradores `$s2` e `$s3`. Novamente para tornar o código mais fácil de acompanhar, identificamos cada bloco de código com sua finalidade no procedimento. Neste exemplo, nove linhas do procedimento `sort` em C tornaram-se 35 em assembly do MIPS.

Salvando registradores		
	sort:	<pre> addi    \$sp,\$sp, -20      # cria espaço na pilha para 5 registradore sw      \$ra, 16(\$sp)## salva \$ra na pilha sw      \$s3,12(\$sp)       # salva \$s3 na pilha sw      \$s2, 8(\$sp)## salva \$s2 na pilha sw      \$s1, 4(\$sp)## salva \$s1 na pilha sw      \$s0, 0(\$sp)## salva \$s0 na pilha </pre>
Corpo do procedimento		
Move parâmetros		<pre> move   \$s2, \$a0 # copia parâmetro \$a0 para \$s2 (salva \$a0) move   \$s3, \$a1 # copia parâmetro \$a1 para \$s3 (salva \$a1) </pre>
Loop externo	for1tst:	<pre> move   \$s0, \$zero## i = 0        slt\$t0, \$s0, \$s3 ## reg. \$t0 = 0 se \$s0 &lt;= \$s3 (in)≥        beq   \$t0, \$zero, exit1. # vai para exit1 se \$s0 ≥ \$s3 (i ≥ n) </pre>
Loop interno	for2tst:	<pre> addi  \$s1, \$s0, -1## j = i - 1        slti\$t0, \$s1, 0    # reg \$t0 = 1 se \$s1 &lt; 0 (j &lt; 0)        bne   \$t0, \$zero, exit2 # vai para exit2 se \$s1 &lt; 0 (j &lt; 0)        sll   \$t1, \$s1, 2## reg. \$t1 = j * 4        add   \$t2, \$s2, \$t1## reg. \$t2 = v + (j * 4)        lw    \$t3, 0(\$t2)## reg. \$t3 = v[j]        lw    \$t4, 4(\$t2)## reg. \$t4 = v[j+1]        slt   \$t0, \$t4, \$t3## reg. \$t0 = 1 se \$t4 &gt;= \$t3        beq   \$t0, \$zero, exit2 # vai para exit2 se \$t4 &gt;= \$t3 </pre>
Passa parâmetros e chama		<pre> move   \$a0, \$s2          # 1º parâmetro de swap é v (antigo \$a0) move   \$a1, \$s1          # 1º parâmetro de swap é j jal    swap              # código de swap mostrado na Figura 2.25 </pre>
Loop interno		<pre> addi  \$s1, \$s1, -1## j -= 1        for2tst           # desvia para teste do loop interno </pre>
Loop externo	exit2:	<pre> addi  \$s0, \$s0, 1        # i += 1        for1tst           # desvia para teste do loop externo </pre>
Restaurando registradores		
	exit1:	<pre> lw    \$s0, 0(\$sp)        # restaura \$s0 da pilha lw    \$s1, 4(\$sp)## restaura \$s1 da pilha lw    \$s2, 8(\$sp)## restaura \$s2 da pilha lw    \$s3,12(\$sp)         # restaura \$s3 da pilha lw    \$ra,16(\$sp)         # restaura \$ra da pilha addi \$sp,\$sp, 20          # restaura stack pointer </pre>
Retorno do procedimento		
	jr	\$ra
		# retorna à rotina que chamou jr

**FIGURA 2.27** Versão em assembly do MIPS para o procedimento sort da Figura 2.26.

## Detalhamento

Uma otimização que funciona com este exemplo é a utilização de *procedimentos inline*. Em vez de passar argumentos em parâmetros e invocar o código com uma instrução `jal`, o compilador copiaria o código do corpo do procedimento `swap` onde a chamada para `swap` aparece no código. Essa otimização evitaria quatro instruções neste exemplo. A desvantagem da otimização que utiliza procedimentos inline é que o código compilado seria maior se o procedimento inline fosse chamado de vários locais. Essa expansão de código poderia ter um desempenho *inferior* se aumentasse a taxa de falhas

na cache; ver Capítulo 5.

## Entendendo o desempenho dos programas

A Figura 2.28 mostra o impacto da otimização do compilador sobre o desempenho do programa de ordenação, tempo de compilação, ciclos de clock, contagem de instruções e CPI. Observe que o código não otimizado tem o melhor CPI, e a otimização O1 tem a menor contagem de instruções, porém O3 é a mais rápida, lembrando que o tempo é a única medida precisa do desempenho do programa.

Otimização gcc	Desempenho relativo	Ciclos de clock (milhões)	Contador de instruções (milhões)	CPI
Nenhuma	1,00	158.615	114.938	1,38
O1 (média)	2,37	66.990	37.470	1,79
O2 (completa)	2,38	66.521	39.993	1,66
O3 (integração de procedimentos )	2,41	65.747	44.993	1,46

**FIGURA 2.28 Comparando desempenho, contagem de instruções e CPI usando otimizações do compilador para o Bubble Sort.**

Os programas ordenaram 100.000 words com o array inicializado com valores aleatórios. Estes programas foram executados em um Pentium 4 com clock de 3,06GHz e um barramento de 533MHz com 2GB de memória SDRAM DDR PC2100. Ele usava o Linux versão 2.4.20.

A Figura 2.29 compara o impacto das linguagens de programação, compilação *versus* interpretação, e os algoritmos sobre o desempenho das ordenações. A quarta coluna mostra que o programa C otimizado é 8,3 vezes mais rápido do que o código Java interpretado para o Bubble Sort. O uso do compilador JIT torna o programa em Java 2,1 vezes *mais rápido* do que o programa em C não otimizado e dentro de um fator de 1,13 mais rápido do que o código C mais otimizado. As razões não são tão próximas para o Quicksort na coluna 5, possivelmente porque é mais difícil amortizar o custo da compilação em runtime pelo tempo de execução mais curto. A última coluna demonstra o impacto de um algoritmo melhor, oferecendo um aumento no desempenho de três ordens de grandeza quando são ordenados 100.000 itens. Mesmo comparando o programa Java interpretado na coluna 5 com o

programa C compilado com as melhores otimizações na coluna 4, o Quicksort vence o Bubble Sort por um fator de 50 (0,05 x 2468 ou 123 vezes mais rápido que o código C não otimizado *versus* 2,41).

Linguagem	Método de execução	Otimização	Desempenho relativo ao Bubble Sort	Desempenho relativo ao Quicksort	Ganho do Quicksort versus Bubble Sort
C	compilador	nenhuma	1,00	1,00	2468
	compilador	01	2,37	1,50	1562
	compilador	02	2,38	1,50	1555
	compilador	03	2,41	1,91	1955
	interpretador	-	0,12	0,05	1050
Java	Compilador Just-In-Time	-	2,13	0,29	338

**FIGURA 2.29 Desempenho de dois algoritmos de ordenação em C e Java usando interpretação e compiladores otimizadores em relação à versão C não otimizada.**

A última coluna mostra a vantagem no desempenho do Quicksort em relação ao Bubble Sort para cada linguagem e opção de execução. Esses programas foram executados no mesmo sistema da Figura 2.28. A JVM é a versão 1.3.1, e o JIT é o Hotspot versão 1.3.1, ambos da Sun.

## Detalhamento

Os compiladores MIPS sempre reservam espaço na pilha para os argumentos, caso precisem ser armazenados, de modo que, na realidade, eles sempre decrementam o \$sp de 16, de modo a dar espaço para todos os quatro registradores de argumento (16 bytes). Um motivo é que a linguagem C oferece vararg que permite que um ponteiro recolha, digamos, o terceiro argumento de um procedimento. Quando, o compilador encontra o raro vararg, ele copia os quatro registradores de argumento para os quatro locais reservados na pilha.

## 2.14. Arrays *versus* ponteiros

Um tópico desafiador para qualquer programador C novo é entender os ponteiros. A comparação entre o código assembly que usa arrays e índices de array para o código assembly que usa ponteiros fornece esclarecimentos sobre ponteiros. Esta seção mostra as versões C e assembly do MIPS de dois procedimentos para zerar (*clear*) uma sequência de palavras na memória: uma

usando índices de array e uma usando ponteiros. A [Figura 2.30](#) mostra os dois procedimentos em C.

```
clear1(int array[], int size)
{
    int i;
    for (i = 0; i < size; i += 1)
        array[i] = 0;
}

clear2(int *array, int size)
{
    int *p;
    for (p = &array[0]; p < &array[size]; p = p + 1)
        *p = 0;
}
```

**FIGURA 2.30** Dois procedimentos em C para definir um array com todos os valores iguais a zero.

`clear1` usa índices, enquanto `clear2` usa ponteiros. O segundo procedimento precisa de alguma explicação para os que não estão acostumados com C. O endereço de uma variável é indicado por `&`, e a referência ao objeto apontando por um ponteiro é indicada por `*`. As declarações indicam que `array` e `p` são ponteiros para inteiros. A primeira parte do loop `for` em `clear2` atribui o endereço do primeiro elemento do `array` ao ponteiro `p`. A segunda parte do loop `for` testa se o ponteiro está apontando além do último elemento do `array`. Incrementar um ponteiro em um, na última parte do loop `for`, significa mover o ponteiro para o próximo objeto sequencial do seu tamanho declarado. Como `p` é um ponteiro para inteiros, o compilador gerará instruções MIPS para incrementar `p` de quatro, o número de bytes de um inteiro MIPS. A atribuição no loop coloca 0 no objeto apontado por `p`.

A finalidade desta seção é mostrar como os ponteiros são mapeados em instruções MIPS, e não endossar um estilo de programação ultrapassado. Ao

final da seção, veremos o impacto das otimizações do compilador moderno sobre esses dois procedimentos.

## Versão de clear usando arrays

Vamos começar com a versão que usa arrays, `clear1`, focalizando o corpo do loop e ignorando o código de ligação do procedimento. Consideramos que os dois parâmetros `array` e `size` são encontrados nos registradores `$a0` e `$a1`, e que `i` é alocado ao registrador `$t0`.

A inicialização de `i`, a primeira parte do loop *for*, é simples:

```
move      $t0,$zero    # i = 0 (reg. $t0 = 0)
```

Para definir `array[i]` como 0, temos primeiro de obter seu endereço. Comece multiplicando `i` por 4, para obter o endereço em bytes:

```
loop1: sll      $t1,$t0,2    # $t1 = i * 4
```

Como o endereço inicial do array está em um registrador, temos de somá-lo ao índice para obter o endereço de `array[i]` usando uma instrução `add`:

```
add      $t2,$a0,$t1  # $t2 = endereço de array[i]
```

Finalmente, podemos armazenar 0 nesse endereço:

```
sw      $zero, 0($t2)    # array[i] = 0
```

Essa instrução é o final do corpo do loop, de modo que o próximo passo é incrementar `i`:

```
addi     $t0,$t0,1        # i = i + 1
```

O teste do loop verifica se  $i$  é menor do que  $size$ :

```
slt      $t3,$t0,$a1      # $t3 = (i < size)  
bne      $t3,$zero,loop1    # se (i < size) vai para loop1
```

Agora, já vimos todas as partes do procedimento. Aqui está o código MIPS para zerar um array usando índices:

```
move    $t0,$zero      # i = 0  
loop1: sll    $t1,$t0,2      # $t1 = i * 4  
        add    $t2,$a0,$t1      # $t2 = endereço de array[i]  
        sw     $zero, 0($t2)      # array[i] = 0  
        addi   $t0,$t0,1      # i = i + 1  
        slt    $t3,$t0,$a1      # $t3 = (i < size)  
        bne    $t3,$zero,loop1    # se (i < size) vai para loop1
```

(Esse código funciona desde que  $size$  seja maior que 0; o ANSI C requer um teste de tamanho antes do loop, mas pularemos essa conformidade aqui.)

## Versão de clear usando ponteiros

O segundo procedimento que usa ponteiros aloca os dois parâmetros  $array$  e  $size$  aos registradores  $$a0$  e  $$a1$  e aloca  $p$  ao registrador  $$t0$ . O código para o segundo procedimento começa com a atribuição do ponteiro  $p$  ao endereço do primeiro elemento do array:

```
move    $t0,$a0      # p = endereço de array[0]
```

O código seguinte é o corpo do loop *for*, que simplesmente armazena 0 em p:

```
loop2: sw      $zero,0($t0) # Memory[p]=0
```

Essa instrução implementa o corpo do loop, de modo que o próximo código é o incremento da iteração, que muda p de modo que aponte para a próxima palavra:

```
addi      $t0,$t0,4      # p = p + 4
```

Incrementar um ponteiro em 1 significa mover o ponteiro para o próximo objeto sequencial em C. Como p é um ponteiro para inteiros, cada um usando 4 bytes, o compilador incrementa p de 4.

O teste do loop vem em seguida. O primeiro passo é calcular o endereço do último elemento de array. Comece multiplicando size por 4 para obter seu endereço em bytes:

```
sll      $t1,$a1,2      # $t1 = size * 4
```

e depois acrescentamos o produto ao endereço inicial do array para obter o endereço da primeira word *após* o array:

```
add      $t2,$a0,$t1  # $t2 = endereço de array[size]
```

O teste do loop é simplesmente para ver se p é menor do que o último elemento de array:

```
slt      $t3,$t0,$t2      # $t3 = (p < &array[size])
```

```
bne      $t3,$zero,loop2    # se (p < &array[size]) vai para loop2
```

Com todas essas partes completadas, podemos mostrar uma versão do código para zerar um array usando ponteiros:

```
move $t0,$a0          # p = endereço de array[0]
loop2: sw   $zero,0($t0)    # Memória[p] = 0
       addi $t0,$t0,4        # p = p + 4
       sll  $t1,$a1,2        # $t1 = size * 4
       add  $t2,$a0,$t1        # $t2 = endereço de array[size]
       slt  $t3,$t0,$t2        # $t3 = (p < &array[size])
       bne $t3,$zero,loop2 # se (p < &array[size]) vai para loop2
```

Como no primeiro exemplo, esse código considera que size é maior do que 0. Observe que esse programa calcula o endereço do final do array em cada iteração do loop, embora não mude. Uma versão mais rápida do código move esse cálculo para fora do loop:

```
move $t0,$a0          # p = endereço de array[0]
       sll  $t1,$a1,2        # $t1 = size * 4
       add  $t2,$a0,$t1        # $t2 = endereço de array[size]
loop2: sw   $zero,0($t0)    # Memória[p] = 0
       addi $t0,$t0,4        # p = p + 4
       slt $t3,$t0,$t2        # $t3 = (p < &array[size])
       bne $t3,$zero,loop2 # se (p < &array[size]) vai para loop2
```

## Comparando as duas versões de clear

A comparação das duas sequências lado a lado ilustra a diferença entre os índices de array e ponteiros (as mudanças introduzidas pela versão de ponteiro estão destacadas):

move \$t0,\$zero	# i = 0	move \$t0,\$a0	# p = & array[0]
loop1: sll \$t1,\$t0,2	# \$t1=i * 4	sll \$t1,\$a1,2	# \$t1=size * 4
add \$t2,\$a0,\$t1	# \$t2=&array[i]	add \$t2,\$a0,\$t1	# \$t2=&array[size]
sw \$zero, 0(\$t2)	# array[i]=0	loop2: sw \$zero,0(\$t0)	# Memória[p]=0
addi \$t0,\$t0,1	# i = i + 1	addi \$t0,\$t0,4	# p = p + 4
slt \$t3,\$t0,\$a1	# \$t3=(i < size)	slt \$t3,\$t0,\$t2	# \$t3=(p < &array[size])
bne \$t3,\$zero,loop1	# if () vai para loop1	bne \$t3,\$zero,loop2	# se () vai para loop2

A versão da esquerda precisa ter a “multiplicação” e a soma dentro do loop, porque *i* é incrementado e cada endereço precisa ser recalculado a partir do novo índice. A versão usando ponteiros para a memória, à direita, incrementa o ponteiro *p* diretamente. A versão usando ponteiros move o deslocamento em escala e a adição do limite de array para fora do loop, reduzindo assim as instruções executadas por iteração de 6 para 4. Essas otimizações manuais correspondem a otimizações do compilador, chamadas redução de força (deslocamento em vez de multiplicação) e eliminação da variável de indução (eliminando cálculos de endereço de array dentro dos loops).

## Detalhamento

Como mencionamos, o compilador C acrescentaria um teste para garantir que *size* seja maior do que 0. Uma maneira seria acrescentar um desvio, imediatamente antes da primeira instrução do loop, para a instrução *slt*.

## Entendendo o desempenho dos programas

As pessoas costumavam ser ensinadas a usar ponteiros em C para conseguir mais eficiência do que era possível com os arrays: “Use ponteiros, mesmo que você não consiga entender o código”. Os compiladores com otimizações modernos podem produzir um código usando arrays tão bom quanto. A maioria dos programadores de hoje prefere que o compilador realize o

trabalho pesado.

## 2.15. Vida real: instruções ARMv7 (32 bits)

ARM é a arquitetura de conjunto de instruções mais comum para dispositivos embutidos, com mais de nove bilhões de dispositivos em 2011 usando ARM, e o crescimento recente tem sido de 2 bilhões por ano. Preparada originalmente para a Acorn RISC Machine, mais tarde modificada para Advanced RISC Machine, ARM surgiu no mesmo ano em que o MIPS e seguiu filosofias semelhantes. A Figura 2.31 lista as semelhanças. A principal diferença é que MIPS tem mais registradores e ARM tem mais modos de endereçamento.

	<b>ARM</b>	<b>MIPS</b>
Data do anúncio	1985	1985
Tamanho da instrução (bits)	32	32
Espaço de endereços (tamanho, modelo)	32 bits, plano	32 bits, plano
Alinhamento de dados	Alinhado	Alinhado
Modos de endereçamento de dados	9	3
Registradores de inteiros (número, modelo, tamanho)	15 GPR × 32 bits	31 GPR × 32 bits
E/S	Mapeado na memória	Mapeado na memória

**FIGURA 2.31** Semelhanças nos conjuntos de instruções ARM e MIPS.

Existe um núcleo semelhante dos conjuntos de instruções para instruções aritmética-lógica e de transferência de dados para o MIPS e ARM, como mostra a Figura 2.32.

	Nome da instrução	ARM	MIPS
Registrador-registrador	Add	add	addu, addiu
	Add (trap if overflow)	adds; swivs	add
	Subtract	sub	subu
	Subtract (trap if overflow)	subs; swivs	sub
	Multiply	mul	mult, multu
	Divide	—	div, divu
	And	and	and
	Or	orr	or
	Xor	eor	xor
	Load high part register	—	lui
	Shift left logical	lsl <sup>1</sup>	sllv, sll
	Shift right logical	lsr <sup>1</sup>	srlv, srl
	Shift right arithmetic	asr <sup>1</sup>	sra, sra
	Compare	cmp, cmn, tst, teq	slt/i, slt/iu
Transferência de dados	Load byte signed	ldrsb	lb
	Load byte unsigned	ldrb	lbu
	Load halfword signed	ldrsh	lh
	Load halfword unsigned	ldrh	lhu
	Load word	ldr	lw
	Store byte	strb	sb
	Store halfword	strh	sh
	Store word	str	sw
	Read, write special registers	mrs, msr	move
	Atomic Exchange	swp, swpb	li;sc

**FIGURA 2.32 Instruções ARM registrador-registrador e transferência de dados equivalentes ao núcleo MIPS.**

Os traços significam que a operação não está disponível nessa arquitetura ou não é sintetizada em poucas instruções. Se houver várias escolhas de instruções equivalentes ao núcleo MIPS, elas são separadas por vírgulas. ARM inclui deslocamentos como parte de cada instrução de operação de dados, de modo que os shift com sobrescrito 1 são apenas uma variação de uma instrução move, como  $1sr^1$ . Observe que o ARM não possui instrução de divisão.

## Modos de endereçamento

A Figura 2.33 mostra os modos de endereçamento de dados admitidos pelo ARM. Diferente do MIPS, o ARM não reserva um registrador para conter 0. Embora o MIPS tenha apenas três modos de endereçamento de dados simples (Figura 2.18), ARM tem nove, incluindo cálculos bastante complexos. Por

exemplo, ARM tem um modo de endereçamento que pode deslocar um registrador por qualquer quantidade, somá-lo aos outros registradores a fim de formar o endereço, e depois atualizar um registrador com esse novo endereço.

Modo de endereçamento	ARM	MIPS
Operando de registrador	X	X
Operando imediato	X	X
Registrador + offset (deslocamento ou base)	X	X
Registrador + registrador (indexado)	X	—
Registrador + registrador escalado (escalado)	X	—
Registrador + offset e registrador de atualização	X	—
Registrador + registrador e registrador de atualização	X	—
Autoincremento, autodecremento	X	—
Dados relativos ao PC	X	—

**FIGURA 2.33 Resumo dos modos de endereçamento de dados.**

ARM tem modos de endereçamento separados, registrador indireto e registrador + offset, em vez de colocar apenas 0 no deslocamento do segundo modo. Para obter um maior intervalo de endereçamento, o ARM desloca o offset à esquerda, 1 ou 2 bits se o tamanho dos dados for halfword ou uma palavra inteira.

## Comparação e desvio condicional

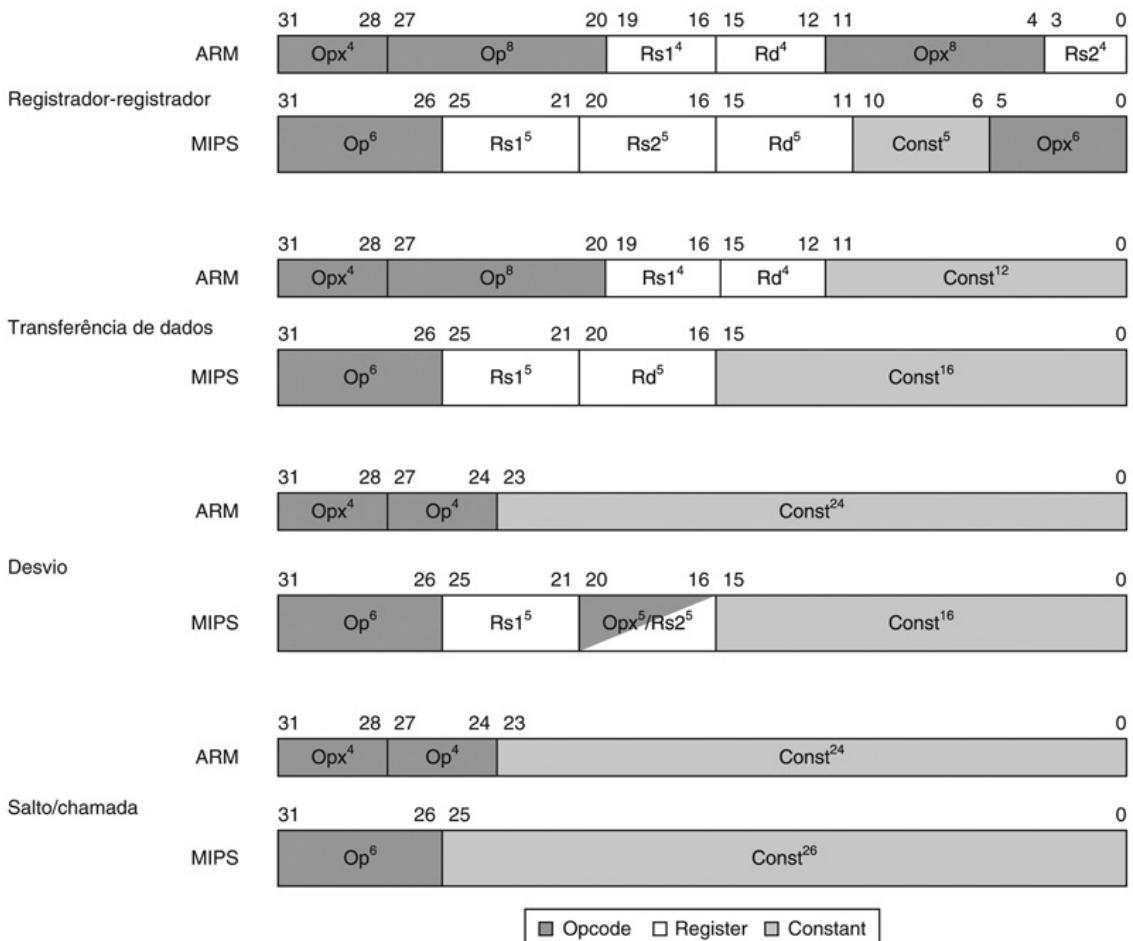
MIPS usa o conteúdo dos registradores para avaliar desvios condicionais. ARM usa os quatro bits de código de condição tradicionais armazenados na palavra de status do programa: *negativo*, *zero*, *carry* e *overflow*. Eles podem ser definidos em qualquer instrução aritmética ou lógica; diferente das arquiteturas anteriores, essa configuração é opcional em cada instrução. Uma opção explícita leva a menos problemas em uma implementação em pipeline. ARM utiliza desvios condicionais para testar os códigos de condição a fim de determinar todas as relações sem sinal e com sinal possíveis.

CMP subtrai um operando do outro, e a diferença define os códigos de condição. CMN *compare negative soma* um operando ao outro, e a soma define os códigos de condição. TST realiza um AND lógico sobre os dois operandos para definir todos os códigos de condição menos overflow, enquanto TEQ utiliza

OR exclusivo a fim de definir os três primeiros códigos de condição.

Um recurso incomum do ARM é que cada instrução tem a opção de executar condicionalmente, dependendo dos códigos de condição. Cada instrução começa com um campo de 4 bits que determina se ele atuará como uma instrução de nenhuma operação (nop) ou como uma instrução real, dependendo dos códigos de condição. Logo, os desvios condicionais são corretamente considerados como executando condicionalmente a instrução de desvio incondicional. A execução condicional permite evitar que um desvio salte sobre uma instrução isolada. É preciso menos espaço de código e tempo para apenas executar uma instrução condicionalmente.

A [Figura 2.34](#) mostra os formatos de instrução para ARM e MIPS. As principais diferenças são o campo de execução condicional de 4 bits em cada instrução e o campo de registrador menor, pois ARM tem metade do número de registradores.



**FIGURA 2.34 Formatos de instrução, ARM e MIPS.**

As diferenças resultam de arquiteturas com 16 ou 32 registradores.

## Recursos exclusivos do ARM

A Figura 2.35 mostra algumas instruções aritmética-lógica não encontradas no MIPS. Por não possuir um registrador dedicado para 0, ARM tem opcodes separados para realizar algumas operações que MIPS pode fazer com \$zero. Além disso, ARM tem suporte para aritmética de múltiplas palavras.

Nome	Definição	ARM	MIPS
Load imediato	$Rd = Imm$	mov	addi, \$0,
Not	$Rd = \sim(Rs1)$	mvn	nor, \$0,
Move	$Rd = Rs1$	mov	or, \$0,
Rotação para direita	$Rd = Rs\ i \gg\ i$ $Rd_{0\dots i-1} = Rs_{31-i\dots 31}$	ror	
And not	$Rd = Rs1 \& \sim(Rs2)$	bic	
Subtração Reversa	$Rd = Rs2 - Rs1$	rsb, rsc	
Apoio à adição de inteiros em múltiplas palavras	CarryOut, $Rd = Rd + Rs1 + OldCarryOut$	adcs	—
Apoio à subtração de inteiros em múltiplas palavras	CarryOut, $Rd = Rd - Rs1 + OldCarryOut$	sbc	—

**FIGURA 2.35 Instruções aritméticas/lógicas do ARM não encontradas no MIPS.**

O campo imediato de 12 bits do ARM tem uma nova interpretação. Os oito bits menos significativos são estendidos em zero a um valor de 32 bits, depois girados para a direita pelo número de bits especificado nos quatro primeiros bits do campo multiplicado por dois. Uma vantagem é que esse esquema pode representar todas as potências de dois em uma palavra de 32 bits. Um estudo interessante seria descobrir se essa divisão realmente recolhe mais imediatos do que um campo simples de 12 bits.

O deslocamento de operandos não é limitado a imediatos. O segundo registrador de todas as operações de processamento aritmético e lógico tem a opção de ser deslocado antes de ser acionado. As opções de deslocamento são *shift left logical*, *shift right logical*, *shift right arithmetic* e *rotate right*.

ARM também possui instruções para salvar grupos de registradores, chamados *loads* e *stores em bloco*. Sob o controle de uma máscara de 16 bits dentro das instruções, qualquer um dos 16 registradores pode ser carregado ou armazenado na memória em uma única instrução. Essas instruções podem salvar e restaurar registradores na entrada e retorno do procedimento. Elas também podem ser usadas para cópia de memória em bloco, sendo este o uso mais importante dessa instrução hoje em dia.

## 2.16. Vida real: instruções x86

*A beleza está toda nos olhos de quem vê.*

*Margaret Wolfe Hungerford, Molly Bawn, 1877*

Os projetistas de conjuntos de instruções às vezes oferecem operações mais poderosas do que aquelas encontradas no ARM e MIPS. O objetivo geralmente é reduzir o número de instruções executadas por um programa. O perigo é que essa redução pode ocorrer ao custo da simplicidade, aumentando o tempo que um programa leva para executar, pois as instruções são mais lentas. Essa lentidão pode ser o resultado de um tempo de ciclo de clock mais lento ou a requisição de mais ciclos de clock do que uma sequência mais simples.

O caminho em direção à complexidade da operação é, portanto, repleto de perigos. A [Seção 2.18](#) demonstra as armadilhas da complexidade.

## A evolução do Intel x86

O ARM e o MIPS foram a visão de pequenos grupos individuais, no ano de 1985; as partes dessas arquiteturas se encaixam muito bem e a arquitetura inteira pode ser descrita de forma sucinta. Isso não acontece com o X86; ele é o produto de vários grupos independentes, que evoluíram a arquitetura por 35 anos, acrescentando novos recursos ao conjunto de instruções original, como alguém acrescentando roupas em uma mala pronta. Aqui estão os marcos importantes do X86:

- **1978:** a arquitetura Intel 8086 foi anunciada como uma extensão compatível com o assembly para o então bem-sucedido Intel 8080, um microprocessador de 8 bits. O 8086 é uma arquitetura de 16 bits, com todos os registradores internos com 16 bits de largura. Ao contrário do MIPS, os registradores possuem usos dedicados, e, por isso, o 8086 não é considerado uma arquitetura com **registradores de uso geral**.

### registradores de uso geral (GPR — General-Purpose Register)

Um registrador que pode ser usado para endereços ou para dados, com praticamente qualquer instrução.

- **1980:** o coprocessador de ponto flutuante Intel 8087 foi anunciado. Essa arquitetura estende o 8086 com cerca de 60 instruções de ponto flutuante. Em vez de usar registradores, ele conta com uma pilha ([Seção 3.7](#)).
- **1982:** o 80286 estendeu a arquitetura 8086, aumentando o espaço de

endereçamento para 24 bits, criando um modelo de mapeamento e proteção de memória elaborado ([Capítulo 5](#)) e acrescentando algumas instruções para preencher o conjunto de instruções e manipular o modelo de proteção.

- **1985:** o 80386 estendeu a arquitetura 80286 para 32 bits. Além de uma arquitetura de 32 bits com registradores de 32 bits e os mesmos 32 bits de espaço de endereçamento, o 80386 acrescentou novos modos de endereçamento e operações adicionais. As instruções adicionais tornam o 80386 quase uma máquina de uso geral. O 80386 também acrescentou suporte para paginação além de endereçamento segmentado ([Capítulo 5](#)). Assim como o 80286, o 80386 possui um modo para executar programas do 8086 sem mudanças.
- **1989-1995:** os posteriores 80486 em 1989, Pentium em 1992 e Pentium Pro em 1995 visaram a um desempenho maior, com apenas quatro instruções acrescentadas ao conjunto de instruções visíveis ao usuário: três para ajudar com o multiprocessamento ([Capítulo 6](#)) e uma instrução move condicional.
- **1997:** depois que o Pentium e o Pentium Pro estavam sendo vendidos, a Intel anunciou que expandiria as arquiteturas Pentium e Pentium Pro com as *Multi Media Extensions* (MMX). Esse novo conjunto de 57 instruções utiliza a pilha de ponto flutuante de modo a acelerar aplicações de multimídia e comunicações. As instruções MMX normalmente operam sobre vários elementos de dados curtos de uma só vez, na tradição das arquiteturas de *única instrução e múltiplos dados* (SIMD — Single Instruction, Multiple Data) ([Capítulo 6](#)). O Pentium II não introduziu novas instruções.
- **1999:** a Intel acrescentou outras 70 instruções, denominadas *Streaming SIMD Extensions* (SSE), como parte do Pentium III. As principais mudanças foram incluir oito registradores separados, dobrar sua largura para 128 bits e incluir um tipo de dados de ponto flutuante com precisão simples. Logo, quatro operações de ponto flutuante de 32 bits podem ser realizadas em paralelo. Para melhorar o desempenho da memória, as SSE incluem instruções de *prefetch* (pré-busca) da cache, mais instruções de armazenamento de streaming, que contornam as caches e escrevem diretamente na memória.
- **2001:** a Intel acrescentou ainda outras 144 instruções, dessa vez denominadas SSE2. O novo tipo de dados tem aritmética de precisão dupla, o que permite pares de operações de ponto flutuante de 64 bits em paralelo. Quase todas essas 144 instruções são versões de instruções MMX e SSE existentes que operam sobre 64 bits de dados em paralelo. Essa mudança não apenas habilita mais operações de multimídia, mas dá ao compilador um alvo

diferente para operações de ponto flutuante do que a arquitetura de pilha única. Os compiladores podem decidir usar os oito registradores SSE como registradores de ponto flutuante, como aqueles encontrados em outros computadores. Essa mudança aumentou o desempenho de ponto flutuante no Pentium 4, o primeiro microprocessador a incluir instruções SSE2.

- **2003:** dessa vez, foi outra empresa, e não a Intel, que melhorou a arquitetura x86. A AMD anunciou um conjunto de extensões arquitetônicas para aumentar o espaço de endereçamento de 32 para 64 bits. Semelhante à transição do espaço de endereçamento de 16 para 32 bits em 1985, com o 80386, o AMD64 alarga todos os registradores para 64 bits. Ele também aumenta a quantidade de registradores para 16 e aumenta o número de registradores SSE de 128 bits para 16. A principal mudança na arquitetura vem da inclusão de um novo modo, chamado *modo longo*, que redefine a execução de todas as instruções x86 com endereços e dados de 64 bits. Para enfrentar a quantidade maior de registradores, ela acrescenta um novo prefixo às instruções. Dependendo de como você conta, o modo longo também acrescenta de 4 a 10 novas instruções e perde 27 antigas. O endereçamento de dados relativo ao PC é outra extensão. O AMD64 ainda possui um modo idêntico ao x86 (*modo legado*) e mais um modo que restringe os programas do usuário ao x86, mas permite que os sistemas operacionais utilizem o AMD64 (*modo de compatibilidade*). Esses modos permitem uma transição mais controlada para o endereçamento de 64 bits do que a arquitetura IA-64 da HP/Intel.
- **2004:** a Intel se rende e abraça o AMD64, trocando seu nome para *Extended Memory 64 Technology* (EM64T). A principal diferença é que a Intel acrescentou uma instrução de comparação e troca atômica de 128 bits, que provavelmente deveria ter sido incluída no AMD64. Ao mesmo tempo, a Intel anunciou outra geração de extensões de mídia. O SSE3 acrescenta 13 instruções para dar suporte à aritmética complexa, operações gráficas sobre arrays de estruturas, codificação de vídeo, conversão de ponto flutuante e sincronismo de threads ([Seção 2.11](#)). A AMD ofereceu o SSE3 nos chips subsequentes e incluiu a instrução de troca atômica que estava faltando no ADM64, para manter a compatibilidade binária com a Intel.
- **2006:** a Intel anuncia 54 novas instruções como parte das extensões do conjunto de instruções SSE4. Essas extensões realizam coisas como soma de diferenças absolutas, produtos escalares para arrays de estruturas, extensão de sinal ou zero de dados estreitos para tamanhos mais largos, contagem de

população e assim por diante. Ela também acrescentou suporte para máquinas virtuais ([Capítulo 5](#)).

- **2007:** a AMD anuncia 170 instruções como parte das SSE5, incluindo 46 instruções do conjunto de instruções básico, que acrescenta três instruções de operando, como MIPS.
- **2011:** a Intel lança a Advanced Vector Extension, que expande a largura de registrador das SSE de 128 para 256 bits, redefinindo, assim, cerca de 250 instruções e acrescentando 128 novas instruções.

Essa história ilustra o impacto das “algemas douradas” da compatibilidade com o x86, pois a base de software existente em cada etapa era muito importante para ser colocada em risco com mudanças arquitetônicas significativas.

Quaisquer que sejam as falhas artísticas do x86, lembre-se de que esse conjunto de instruções impulsionou a geração de computadores PC e ainda domina a parte da nuvem da era pós-PC. A fabricação de 350 milhões de chips x86 por ano pode parecer pequena em comparação com os 9 bilhões de chips ARMv7, mas muitas empresas adorariam controlar esse mercado. Apesar disso, esse ancestral diversificado levou a uma arquitetura difícil de explicar e impossível de amar.

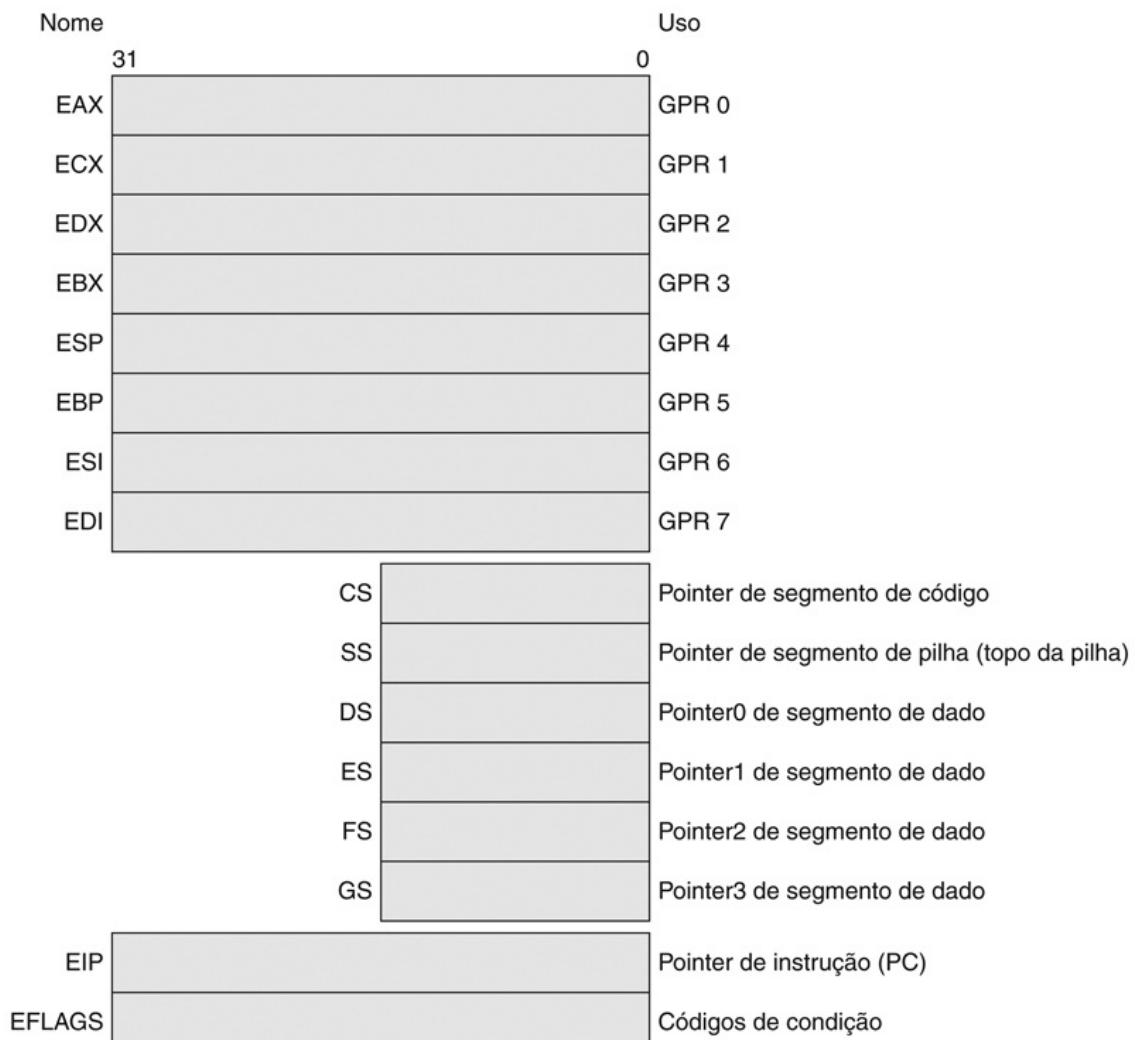
Preste bem atenção ao que você está para ver! *Não tente ler esta seção com o cuidado que precisaria para escrever programas x86; em vez disso, o objetivo é que você tenha alguma familiaridade com os pontos fortes e fracos da arquitetura mais popular do mundo para uso em desktops.*

Em vez de mostrar o conjunto de instruções inteiro de 16, 32 e 64 bits, nesta seção, vamos nos concentrar no subconjunto de 32 bits originado com o 80386. Começamos nossa explicação com os registradores e os modos de endereçamento, prosseguimos para as operações com inteiros e concluímos com um exame da codificação da instrução.

## Registradores e modos de endereçamento de dados x86

Os registradores do 80386 mostram a evolução do conjunto de instruções ([Figura 2.36](#)). O 80386 estendeu todos os registradores de 16 bits (exceto os registradores de segmento) para 32 bits, inserindo um *E* no início de seus nomes para indicar a versão de 32 bits. Vamos nos referir a eles genericamente como registradores de uso geral (ou GPRs — General-Purpose Registers). O 80386 contém apenas oito GPRs. Isso significa que os programas do MIPS podem usar

quatro vezes isso e os ARM, duas vezes.



**FIGURA 2.36 O conjunto de registradores do 80386.**

Começando com o 80386, os oito registradores iniciais foram estendidos para 32 bits e também poderiam ser usados como registradores de uso geral.

A [Figura 2.37](#) mostra que as instruções aritméticas, lógicas e de transferência de dados são instruções de dois operandos. Existem duas diferenças importantes aqui. As instruções aritméticas e lógicas do x86 precisam ter um operando que atue como origem e destino; o ARMv7 e o MIPS admitem registradores separados para origem e destino. Essa restrição coloca mais pressão sobre os registradores limitados, pois um registrador de origem precisa ser modificado. A

segunda diferença importante é que um dos operandos pode estar na memória. Assim, praticamente qualquer instrução pode ter um operando na memória, ao contrário do ARMv7 e do MIPS.

<b>Tipo de operando de origem/destino</b>	<b>Segundo operando de origem</b>
Registrador	Registrador
Registrador	Imediato
Registrador	Memória
Memória	Registrador
Memória	Imediato

**FIGURA 2.37 Tipos de instrução para instruções aritméticas, lógicas e de transferência de dados.**

O x86 permite as combinações mostradas. A única restrição é a ausência de um modo memória-memória. Os imediatos podem ser de 8, 16 ou 32 bits de extensão; um registrador é qualquer um dos 14 principais registradores da [Figura 2.36](#) (não EIP ou EFLAGS).

Os modos de endereçamento de memória, descritos com detalhes a seguir, oferecem dois tamanhos de endereços dentro da instrução. Esses chamados *deslocamentos* podem ser de 8 bits ou de 32 bits.

Embora um operando da memória possa usar qualquer modo de endereçamento, existem restrições em relação a quais registradores podem ser usados em um modo. A [Figura 2.38](#) mostra os modos de endereçamento do x86 e quais GPRs não podem ser usados com cada modo, além de como obter o mesmo efeito usando instruções MIPS.

<b>Modo</b>	<b>Descrição</b>	<b>Restrições de registradores</b>	<b>MIPS equivalente</b>
Registrador indireto	O endereço está em um registrador.	Não é ESP ou EBP	<code>lw \$s0,0(\$s1)</code>
Modo baseado com deslocamento de 8 ou 32 bits	O endereço é o conteúdo do registrador-base com o deslocamento.	Não é ESP	<code>lw \$s0,100(\$s1) #&lt;= 16-bit # deslocamento</code>
Base com o índice escalado	O endereço é a Base + $(2^{\text{Escala}} \times \text{Índice})$ , em que a Escala tem o valor de 0, 1, 2 ou 3.	Base: qualquer GPR Índice: não é ESP	<code>mul \$t0,\$s2,4 add \$t0,\$t0,\$s1 lw \$s0,0(\$t0)</code>
Base com índice escalado e deslocamento de 8 ou 32 bits	O endereço é a Base + $(2^{\text{Escala}} \times \text{Índice}) + \text{Deslocamento}$ , em que a Escala tem o valor de 0, 1, 2 ou 3.	Base: qualquer GPR Índice: não é ESP	<code>mul \$t0,\$s2,4 add \$t0,\$t0,\$s1 lw \$s0,100(\$t0) # 16-bit # deslocamento</code>

**FIGURA 2.38 Modos de endereçamento de 32 bits do x86 com restrições de registrador e o código MIPS equivalente.**

O modo de endereçamento Base mais Índice Escalado, que não

aparece no ARM ou no MIPS, foi incluído para evitar as multiplicações por quatro (fator de escala 2) para transformar um índice de um registrador em um endereço em bytes ([Figuras 2.25 e 2.27](#)). Um fator de escala 1 é usado para dados de 16 bits e um fator de escala 3, para dados de 64 bits. O fator de escala 0 significa que o endereço não é escalado. Se o deslocamento for maior do que 16 bits no segundo ou quarto modos, então o modo MIPS equivalente precisa de mais duas instruções: um `lui` para ler os 16 bits mais altos do deslocamento e um `add` para somar a parte alta do endereço ao registrador base `$s1`. (A Intel oferece dois nomes diferentes para o que é chamado modo de endereçamento com Base — com Base e Indexado —, mas eles são basicamente idênticos, e os combinamos aqui.)

## Operações com inteiros do x86

O 8086 oferece suporte para tipos de dados de 8 bits (*byte*) e 16 bits (*word*). O 80386 acrescenta endereços e dados de 32 bits (*double words*) ao x86. (AMD64 acrescenta endereços e dados de 64 bits, chamados *quad words*; vamos nos ater ao 80386 nesta seção.) As distinções de tipo de dados se aplicam a operações com registrador e também a acessos à memória.

Quase toda operação funciona sobre dados de 8 bits e sobre um tamanho de dados maior. Esse tamanho é determinado pelo modo e é de 16 bits ou de 32 bits.

Logicamente, alguns programas querem operar sobre dados de todos os três tamanhos, de modo que os arquitetos do 80386 ofereceram uma forma conveniente de especificar cada versão sem expandir muito o tamanho do código. Elas decidiram que os dados de 16 bits ou de 32 bits dominam a maioria dos programas e, por isso, faz sentido poder definir um tamanho grande padrão. Esse tamanho de dados padrão é definido por um bit no registrador do segmento de código. Para redefini-lo, um *prefixo* de 8 bits é anexado à instrução a fim de dizer à máquina para usar o outro tamanho grande para essa instrução.

A solução do prefixo foi emprestada do 8086, o que permite que diversos prefixos modifiquem o comportamento da instrução. Os três prefixos originais redefinem o registrador de segmento padrão, bloqueiam o barramento para dar suporte à sincronização ([Seção 2.11](#)) ou repetem a instrução seguinte até o registrador ECX chegar a 0. Esse último prefixo tinha por finalidade estar emparelhado com uma instrução mover byte para mover um número variável de

bytes. O 80386 também acrescentou um prefixo para redefinir o tamanho de endereço padrão.

As operações com inteiros do x86 podem ser divididas em quatro classes principais:

1. Instruções para movimentação de dados, incluindo move, push e pop
2. Instruções aritméticas e lógicas, incluindo operações aritméticas de teste, inteiros e decimais
3. Fluxo de controle, incluindo desvios condicionais, jumps incondicionais, chamadas e retornos
4. Instruções para manipulação de strings, incluindo movimento e comparação de strings

As duas primeiras categorias não precisam de comentários, exceto que as operações de instruções aritméticas e lógicas permitem que o destino seja um registrador ou um local da memória. A [Figura 2.39](#) mostra algumas instruções x86 típicas e suas funções.

Instrução	Função
je nome	se for igual(códigos de condição) {EIP=n} EIP-128 <= nome < EIP+128
jmp nome	EIP=nome
call nome	SP=SP-4; M[SP]=EIP+5; EIP=nome;
movw EBX,[EDI+45]	EBX=M[EDI+45]
push ESI	SP=SP-4; M[SP]=ESI
pop EDI	EDI=M[SP]; SP=SP+4
add EAX,#6765	EAX= EAX+6765
test EDX,#42	Define códigos de condição (flags) com EDX e 42
movsl	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

**FIGURA 2.39** Algumas instruções x86 típicas e suas funções.

Uma lista de operações frequentes aparece na [Figura 2.40](#). O CALL salva na pilha o EIP da próxima instrução. (EIP é o PC — Program Counter — da Intel.)

Os desvios condicionais no x86 são baseados em *códigos de condição* ou *flags*, assim como no ARMv7. Os códigos de condição são definidos como um efeito colateral de uma operação; a maioria é usada para comparar o valor de um resultado com 0. Os desvios, então, testam os códigos de condição. Os endereços de desvio relativos ao PC precisam ser especificados no número de bytes, visto

que, ao contrário do ARMv7 e MIPS, nem todas as instruções do 80386 possuem 4 bytes de extensão.

As instruções para manipulação de strings fazem parte dos antecessores 8080 do x86 e não são comumente executadas na maioria dos programas. Em geral, são mais lentas do que as rotinas de software equivalentes (veja a falácia na [Seção 2.17](#)).

A [Figura 2.40](#) lista algumas das instruções do x86 com inteiros. Muitas das instruções estão disponíveis nos formatos byte e word.

Instrução	Significado
<b>Controle</b>	<b>Desvios condicionais ou incondicionais</b>
jnz, jz	Pula se a condição para o offset EIP + 8-bit; JNE (para JNZ), JE (para JZ) são nomes alternativos
jmp	Pulo incondicional – offset de 8 ou 16 bits
call	Chamada de sub-rotina – offset de 16-bit; o endereço de retorno é empurrado para a pilha
ret	Dispara o endereço de retorno da pilha e pula para ele
loop	Desvio de loop – diminuição do ECX; pula para o deslocamento EIP + 8-but se ECX ≠ 0
<b>Transferência de dados</b>	<b>Movimenta dados entre registradores ou entre registrador e memória</b>
move	Movimento entre dois registradores ou entre um registrador e memória
push, pop	Empurra o operando-fonte na pilha; dispara o operando do topo da pilha para o registrador
les	Carrega o ES a um dos GPRs da memória
<b>Aritmético, lógico</b>	<b>Operações aritméticas e lógicas utilizando dados de registradores e memória</b>
add, sub	Adiciona a fonte ao destino; subtrai a fonte do destino; formato registrador-memória
cmp	Compara fonte e destino; formato registrador-memória
shl, shr, rcr	Desloca à esquerda; desloca à direita lógica; rotacional à direita com o código de condição de carregamento conforme preenchido
cbw	Converte byte em 8 bits direitos de EAX para 16-bit word na direita do EAX
test	Lógico E, fonte e destino configuram os códigos de condição
inc, dec	Incrementa o destino; desincrementa o destino
or, xor	OR lógico; OR exclusive; formato registrador-memória
<b>String</b>	<b>Movimento entre operandos string; comprimento dado por um prefixo repetido</b>
movs	Copia a fonte string para o destino pelo incremento do ESI e EDI; pode ser repetido
lod\$	Carrega um byte, word ou doubleword de uma string em um registrador EAX

**FIGURA 2.40** Algumas operações típicas do x86.

Muitas operações utilizam o formato registrador-memória, no qual a origem ou o destino pode ser a memória e o outro pode ser um registrador ou um operando imediato.

## Codificação de instruções x86

Deixando o pior para o final, a codificação de instruções no 80386 é complexa, com muitos formatos de instrução diferentes. As instruções para o 80386 podem variar de 1 byte, quando não existem operandos, até 15 bytes.

A [Figura 2.41](#) mostra o formato de instrução para várias instruções de exemplo na [Figura 2.39](#). O byte de opcode normalmente contém um bit indicando se o operando é de 8 bits ou de 32 bits. Para algumas instruções, o opcode pode incluir o modo de endereçamento e o registrador; isso acontece em muitas instruções que possuem a forma “registrador = registrador op imediato”. Outras instruções utilizam um “pós-byte” ou byte de opcode extra, rotulado “mod, reg, r/m”, que contém a informação sobre o modo de endereçamento. Esse pós-byte é usado para muitas das instruções que endereçam a memória. O modo “base mais índice escalado” utiliza um segundo pós-byte, rotulado com “sc, índice, base”.

a. JE EIP + deslocamento

4	4	8
JE	Condição	Deslocamento

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m Pós-byte	Deslocamento

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Imediato

f. TEST EDX, #42

7	1	8	32
TEST	w	Pós-byte	Imediato

#### FIGURA 2.41 Formatos típicos de instruções x86.

A Figura 2.42 mostra a codificação do pós-byte. Muitas instruções contêm o campo de 1 bit w, que indica se a operação é de um byte ou double word. O campo d em mov é usado em instruções que podem mover de/para a memória, e mostra a direção do movimento. A instrução ADD requer 32 bits para o campo imediato, visto que no modo de 32 bits, os imediatos são de 8 bits ou de 32 bits. O campo imediato no TEST tem 32 bits de extensão, pois não existe um imediato de 8 bits para testar no modo de 32 bits. Em geral, as instruções podem variar de 1 a 15 bytes de extensão. O tamanho grande vem dos prefixos extras de 1 byte, tendo tanto um imediato de 4 bytes quanto um endereço de deslocamento de 4 bytes, usando um opcode de 2 bytes e usando o especificador do modo de índice escalado, que acrescenta outro byte.

A Figura 2.42 mostra a codificação dos dois especificadores de endereço pós-byte para os modos de 16 e 32 bits. Infelizmente, para entender quais registradores e quais modos de endereçamento estão disponíveis, você precisa ver a codificação de todos os modos de endereçamento e, às vezes, até mesmo a codificação das instruções.

reg	w = 0	w = 1		r/m	mod = 0		mod = 1		mod = 2		mod = 3
		16b	32b		16b	32b	16b	32b	16b	32b	
0	AL	AX	EAX	0	end=BX+SI	=EAX	mesmo	mesmo	mesmo	mesmo	mesmo
1	CL	CX	ECX	1	end=BX+DI	=ECX	end. que	end. que	end. que	end. que	que
2	DL	DX	EDX	2	end=BP+SI	=EDX	mod=0	mod=0	mod=0	mod=0	campo
3	BL	BX	EBX	3	end=BP+SI	=EBX	+ disp8	+ disp8	+ disp16	+ disp32	registro
4	AH	SP	ESP	4	end=SI	=(sib)	SI+disp8	(sib)+disp8	SI+disp8	(sib)+disp32	-
5	CH	BP	EBP	5	end=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	-
6	DH	SI	ESI	6	end=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	-
7	BH	DI	EDI	7	end=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	-

**FIGURA 2.42** A codificação do primeiro especificador de endereço do x86: “mod, reg, r/m”.

As quatro primeiras colunas mostram a codificação do campo reg de 3 bits, que depende do bit w do opcode e se a máquina está no modo de 16 bits (8086) ou no modo de 32 bits (80386).

As demais colunas explicam os campos mod e r/m. O significado do campo r/m de 3 bits depende do valor do campo mod de 2 bits e do tamanho do endereço. Basicamente, os registradores utilizados no cálculo do endereço são listados na sexta e sétima colunas, sob mod = 0, com mod = 1 acrescentando um deslocamento de 8 bits e mod = 2 acrescentando um deslocamento de 16 ou 32 bits, dependendo do modo do endereço. As exceções são: 1) r/m = 6 quando mod = 1 ou mod = 2 no modo de 16 bits seleciona BP mais o deslocamento; 2) r/m = 5 quando mod = 1 ou mod = 2 no modo 16 bits seleciona EBP mais deslocamento; e 3) r/m = 4 no modo de 32 bits quando mod não é igual a 3, em que (sib) significa o uso do modo de índice escalado, mostrado na Figura 2.38. Quando mod = 3, o campo r/m indica um registrador, usando a mesma codificação que o campo reg combinado com o bit w.

## Conclusão sobre o x86

A Intel tinha um microprocessador de 16 bits dois anos antes das arquiteturas mais elegantes de seus concorrentes, como o Motorola 68000, e essa dianteira levou à seleção do 8086 como CPU para o IBM PC. Os engenheiros da Intel

geralmente reconhecem que o x86 é mais difícil de ser montado do que máquinas como ARMv7 e MIPS, mas o mercado maior significou, na era do PC, que a AMD e a Intel podiam abrir mão de mais recursos para ajudar a contornar a complexidade adicional. O que o x86 perde no estilo é compensado na fatia do mercado, tornando-o belo, do ponto de vista apropriado.

O que o salva é que os componentes arquitetônicos mais usados do x86 não são tão difíceis de implementar, como a AMD e a Intel já demonstraram, melhorando rapidamente o desempenho dos programas com inteiros desde 1978. Para obter esse desempenho, os compiladores precisam evitar as partes da arquitetura difíceis de implementar com rapidez.

Entretanto, na era pós-PC, apesar das habilidades consideráveis em termos de arquitetura e manufatura, o x86 ainda não se tornou competitivo no dispositivo móvel pessoal.

## 2.17. Vida real: instruções ARMv8 (64 bits)

Dos muitos problemas em potencial em um conjunto de instruções, aquele que é quase impossível de ser contornado é ter um endereço de memória muito pequeno. Enquanto o x86 foi estendido com sucesso, primeiro para endereços de 32 bits e depois para endereços de 64 bits, muitos de seus irmãos ficaram para trás. Por exemplo, o MOStek 6502 com endereços de 16 bits controlava o Apple II, mas mesmo tendo saído na frente com o primeiro computador pessoal comercialmente bem-sucedido, sua falta de bits de endereço o condenou à caixa de lixo da história.

Os arquitetos do ARM podiam ver o iminente fim de seu computador com 32 bits de endereços, e iniciaram o projeto da versão com endereços de 64 bits do ARM em 2007. Finalmente, ele foi revelado em 2013. Em vez de algumas pequenas mudanças estéticas, para que todos os registradores tivessem 64 bits de largura, basicamente o que aconteceu com o x86, o ARM fez uma reforma completa. A boa notícia é que, se você conhece o MIPS, será muito fácil entender o ARMv8, como é chamada a versão para 64 bits.

Primeiro, em comparação com o MIPS, o ARM descartou praticamente todos os recursos incomuns do v7:

- Não há um campo de execução condicional, como havia em quase toda instrução no v7.
- O campo imediato é simplesmente uma constante de 12 bits, em vez de basicamente uma entrada para uma função que produz uma constante, como

no v7.

- O ARM retirou as instruções Load Multiple e Store Multiple.
- O PC não é mais um dos registradores, o que resultava em desvios inesperados se você escrevesse nele.
- Em segundo lugar, o ARM acrescentou recursos que faltavam e que são úteis no MIPS:
- O V8 possui 32 registradores de uso geral, que os escritores de compilador certamente apreciam muito. Assim como o MIPS, um registrador é fixado em 0, embora em vez disso se refira ao ponteiro de pilha em instruções load e store.
- Seus modos de endereçamento funcionam para todos os tamanhos de word no ARMv8, o que não acontecia no ARMv7.
- Ele inclui uma instrução de divisão, que foi omitida do ARMv7.
- Ele acrescenta o equivalente ao “branch if equal” e o “branch if not equal” do MIPS.

Como a filosofia do conjunto de instruções do v8 é muito mais próxima do MIPS do que do v7, nossa conclusão é que a semelhança principal entre o ARMv7 e o ARMv8 está no nome.

## 2.18. Falácia e armadilhas

Falácia: instruções mais poderosas significam maior desempenho.

Parte do poder do Intel x86 são os prefixos que podem modificar a execução da instrução seguinte. Um prefixo pode repetir a instrução seguinte até que um contador chegue a 0. Assim, para mover dados na memória, pode parecer que a sequência de instruções natural seria usar move com o prefixo de repetição para realizar movimentações de memória para memória em 32 bits.

Um método alternativo, que usa as instruções padrão encontradas em todos os computadores, é carregar os dados nos registradores e depois armazenar os registradores na memória. Essa segunda versão do programa, com o código replicado para reduzir o trabalho extra do loop, copia cerca de 1,5 vez mais rápido. Uma terceira versão, que usa os registradores de ponto flutuante maiores no lugar dos registradores inteiros do x86, copia cerca de 2,0 vezes mais rápido do que a instrução de movimentação complexa.

Falácia: escreva em assembly para obter o maior desempenho.

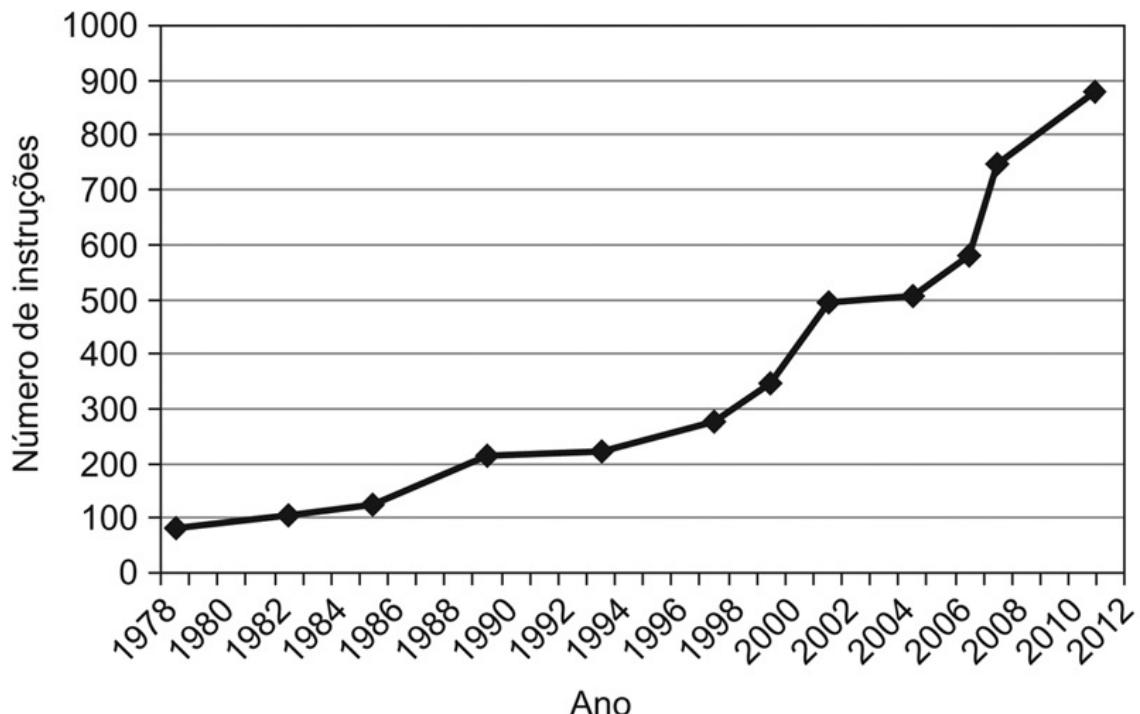
Houve uma época em que os compiladores para as linguagens de programação produziam sequências de instrução ingênuas; a sofisticação cada vez maior dos compiladores significa que a lacuna entre o código compilado e o código produzido à mão está se fechando rapidamente. De fato, para competir com os compiladores atuais, o programador assembly precisa entender perfeitamente os conceitos dos Capítulos 4 e 5 (pipelining do processador e hierarquia de memória).

Essa batalha entre compiladores e codificadores assembly é uma situação em que os humanos estão perdendo terreno. Por exemplo, a linguagem C oferece ao programador uma chance de dar uma sugestão ao compilador sobre quais variáveis manter em registradores, em vez de passar para a memória. Quando os compiladores eram fracos na alocação de registradores, essas sugestões eram vitais para o desempenho. De fato, alguns livros-texto sobre C gastavam muito tempo dando exemplos com sugestões de como usar registradores com eficiência. Os compiladores C de hoje, em geral, ignoram essas sugestões, pois o compilador realiza um trabalho melhor na alocação do que o programador.

Mesmo se a escrita à mão resultasse em código mais rápido, os perigos de escrever em assembly são: maior tempo gasto codificando e depurando, perda de portabilidade e dificuldade de manter esse código. Um dos poucos axiomas aceitos de modo geral na engenharia de software é que a codificação leva mais tempo se você escrever mais linhas, e claramente é preciso mais linhas para escrever um programa em assembly do que em C ou Java. Além do mais, uma vez codificado, o próximo perigo é que ele se torne um programa popular. Esses programas sempre vivem por mais tempo do que o esperado, significando que alguém terá de atualizar o código por vários anos e fazer com que funcione com novas versões dos sistemas operacionais e novos modelos de máquinas. A escrita em linguagem de alto nível no lugar do assembly não apenas permite que os compiladores futuros ajustem o código a máquinas futuras, mas também torna o software mais fácil de manter e permite que o programa execute em mais modelos de computadores.

Falácia: a importância da compatibilidade binária comercial significa que os conjuntos de instruções bem-sucedidos não mudam.

Embora a compatibilidade binária seja sacrossanta, a [Figura 2.43](#) mostra que a arquitetura do x86 cresceu drasticamente. A média é mais de uma instrução por mês no decorrer do seu tempo de vida de 35 anos!



**FIGURA 2.43 Crescimento do conjunto de instruções x86 com o tempo.**

Embora haja um valor técnico claro em algumas dessas extensões, essa mudança rápida também aumenta a dificuldade para outras empresas tentarem montar processadores compatíveis.

Armadilha: esquecer que os endereços sequenciais de palavras em máquinas com endereçamento em bytes não diferem em um.

Muitos programadores assembly têm lutado contra erros cometidos pela suposição de que o endereço da próxima palavra pode ser encontrado incrementando-se o endereço em um registrador por um, em vez do tamanho da palavra em bytes. Prevenir é melhor do que remediar!

**Armadilha:** usando um ponteiro para uma variável automática fora de seu procedimento de definição.

Um engano comum ao lidar com ponteiros é passar um resultado de um procedimento que inclui um ponteiro para um array que é local a esse procedimento. Seguindo a disciplina de pilha da [Figura 2.12](#), a memória que contém o array local será reutilizada assim que o procedimento retornar. Os ponteiros para variáveis automáticas podem levar ao caos.

## 2.19. Comentários finais

*Menos significa mais.*

*Robert Browning, Andrea del Sarto, 1855*

Os dois princípios do computador com *programa armazenado* são o uso de instruções que sejam indiferentes de números e o uso de memória alterável para os programas. Estes princípios permitem que uma única máquina auxilie cientistas ambientais, consultores financeiros e autores de romance em suas especialidades. A seleção de um conjunto de instruções que a máquina possa entender exige um equilíbrio delicado entre a quantidade de instruções necessárias para executar um programa, a quantidade de ciclos de clock necessários por uma instrução e a velocidade do clock. Como ilustramos neste capítulo, três princípios de projeto orientam os autores de conjuntos de instruções a estabelecer esse equilíbrio delicado:

1. *Simplicidade favorece a regularidade.* A regularidade motiva muitos recursos do conjunto de instruções do MIPS: mantendo todas as instruções com um único tamanho, sempre exigindo três operandos de registrador nas instruções aritméticas e mantendo os campos de registrador no mesmo lugar em cada formato de instrução.
2. *Menor é mais rápido.* O desejo de velocidade é o motivo para que o MIPS tenha 32 registradores em vez de muito mais.
3. *Um bom projeto exige bons compromissos.* Um exemplo do MIPS foi o compromisso entre providenciar endereços e constantes maiores nas instruções e manter todas as instruções com o mesmo tamanho.

Também vimos a grande ideia de tornar o **caso comum veloz** aplicada a conjuntos de instruções e também à arquitetura do computador. Alguns

exemplos de tornar o caso comum do MIPS veloz são o endereçamento relativo ao PC para desvios condicionais e o endereçamento imediato para operandos constantes maiores.



## CASO COMUM VELOZ

Acima desse nível de máquina está o assembly, uma linguagem que os humanos podem ler. O montador traduz isso para os números binários que as máquinas podem entender e até mesmo “estende” o conjunto de instruções, criando instruções simbólicas que não estão no hardware. Por exemplo, constantes ou endereços que são muito grandes são divididos em partes com tamanho apropriado, variações comuns de instruções recebem seu próprio nome, e assim por diante. A [Figura 2.44](#) lista as instruções MIPS que abordamos até aqui, tanto instruções reais quanto pseudoinstruções. Ocultar os detalhes do nível mais alto é outro exemplo da grande ideia da **abstração**.

Instruções MIPS	Nome	Formato	PseudoMIPS	Nome	Formato
add	add	R	move	move	R
subtract	sub	R	multiply	mult	R
add immediate	addi	I	multiply immediate	multi	I
load word	lw	I	load immediate	li	I
store word	sw	I	branch less than	blt	I
load half	lh	I	branch less than or equal	ble	I
load half unsigned	lhu	I			
store half	sh	I	branch greater than	bgt	I
load byte	lb	I	branch greater than or equal	bge	I
load byte unsigned	lbu	I			
store byte	sb	I			
load linked	ll	I			
store conditional	sc	I			
load upper immediate	lui	I			
and	and	R			
or	or	R			
nor	nor	R			
and immediate	andi	I			
or immediate	ori	I			
shift left logical	sll	R			
shift right logical	srl	R			
branch on equal	beq	I			
branch on not equal	bne	I			
set less than	slt	R			
set less than immediate	slti	I			
set less than immediate unsigned	sltiu	I			
jump	j	J			
jump register	jr	R			
jump and link	jal	J			

**FIGURA 2.44** O conjunto de instruções do MIPS explicado até aqui, com as instruções MIPS reais à esquerda e as pseudoinstruções à direita.

O Apêndice A ([Seção A.10](#)) descreve a arquitetura MIPS completa. A [Figura 2.1](#) mostra mais detalhes da arquitetura MIPS revelada neste capítulo. As informações que aparecem aqui são encontradas nas colunas 1 e 2 do Guia de Referência do MIPS, no final deste livro.



## A B S T R A Ç Ã O

Cada categoria de instruções MIPS está associada a construções que aparecem nas linguagens de programação:

- As instruções aritméticas correspondem às operações encontradas nas instruções de atribuição.
- As instruções de transferência de dados provavelmente ocorrerão quando se lida com estruturas de dados, como arrays e estruturas.
- Os desvios condicionais são usados em instruções *if* e em loops.
- Os jumps incondicionais são usados em chamadas de procedimento e em retornos, e para instruções *case/switch*.

Essas instruções não nasceram iguais; a popularidade de poucas domina muitas. Por exemplo, a [Figura 2.45](#) mostra a popularidade de cada classe de instruções para o SPE CPUC2006. A popularidade variada das instruções desempenha um papel importante nos capítulos sobre desempenho, caminho de dados, controle e pipelining.

Classe de instrução	Exemplos do MIPS	Correspondência com uma linguagem de alto nível	Frequência	
			Inteiro	Ponto flutuante
Aritmética	add, sub, addi	operações nas instruções de atribuição	16%	48%
Transferência de dados	lw, sw, lb, lbu, lh, lhu, sb, lui	referências a estruturas de dados, como arrays	35%	36%
Lógica	and, or, nor, andi, ori, sll, srl	operações em instruções de atribuição	12%	4%
Desvio condicional	beq, bne, slt, slti, sltiu	instruções if e loops	34%	8%
Jump	j, jr, jal	chamadas de procedimento, retornos e instruções case/switch	2%	0%

**FIGURA 2.45** Classes de instruções MIPS, exemplos, correspondência com construções de linguagem de programação de alto nível e porcentagem média de instruções do MIPS executadas por categoria para os benchmarks médios SPEC CPU2006 de inteiros e ponto flutuante.

A Figura 3.26 no Capítulo 3 mostra a porcentagem das instruções MIPS individuais executadas.

Depois que explicarmos a aritmética do computador no Capítulo 3, revelaremos mais da arquitetura do conjunto de instruções do MIPS.

## 2.20. Exercícios

O Apêndice A descreve o simulador do MIPS, que é útil para estes exercícios. Embora o simulador aceite pseudoinstruções, tente não as usar em qualquer exercício que pedir para produzir código do MIPS. Seu objetivo deverá ser aprender o conjunto de instruções MIPS real, e se você tiver de contar instruções, sua contagem deverá refletir as instruções reais executadas, e não as pseudoinstruções.

Existem alguns casos em que as pseudoinstruções precisam ser usadas (por exemplo, a instrução la, quando um valor real não é conhecido durante a codificação em assembly). Em muitos casos, elas são muito convenientes e resultam em código mais legível (por exemplo, as instruções li e move). Se você decidir usar pseudoinstruções por esses motivos, por favor, acrescente uma sentença ou duas à sua solução, indicando quais pseudoinstruções usou e por quê.

**2.1 [5] <§2.2>** Para a instrução C a seguir, qual é o código assembly do MIPS correspondente? Suponha que as variáveis f, g, h e i sejam dadas e possam ser consideradas inteiros de 32 bits, conforme declarado em um programa C. Use um número mínimo de instruções assembly do MIPS.

$$F = g + (h - 5);$$

**2.2 [5] <§2.2>** Para as instruções assembly do MIPS a seguir, qual é a instrução C correspondente?

add f, g, h

add f, i, f

**2.3 [5] <§§2.2, 2.3>** Para a instrução C a seguir, qual é o código assembly do MIPS correspondente? Suponha que as variáveis f, g, h, i e j sejam atribuídas aos registradores \$s0, \$s1, \$s2, \$s3 e \$s4, respectivamente. Suponha que o endereço de base dos arrays A e B estejam nos registradores \$s6 e \$s7, respectivamente.

$$B[8] = A[i - j];$$

**2.4 [5] <§§2.2, 2.3>** Para as instruções assembly do MIPS a seguir, qual é a instrução C correspondente? Suponha que as variáveis f, g, h, i e j sejam atribuídas aos registradores \$s0, \$s1, \$s2, \$s3 e \$s4, respectivamente. Suponha também que o endereço de base dos arrays A e B estejam nos registradores \$s6 e \$s7, respectivamente.

sll	\$t0, \$s0, 2	# \$t0 = f * 4
add	\$t0, \$s6, \$t0	# \$t0 = &A[f]
sll	\$t1, \$s1, 2	# \$t1 = g * 4
add	\$t1, \$s7, \$t1	# \$t1 = &B[g]
lw	\$s0, \$(\$t0)	# f = A[f]

```

addi    $t2, $t0, 4
lw      $t0, 0($t2)
add    $t0 $t0, $s0
sw      $t0, 0($t1)

```

**2.5 [5] <§§2.2, 2.3>** Para as instruções assembly do MIPS no Exercício 2.4, reescreva o código assembly para diminuir o número de instruções MIPS (se possível) necessárias para executar a mesma função.

**2.6** A tabela a seguir mostra valores de 32 bits de um array armazenado na memória.

Endereço	Dados
24	2
38	4
32	3
36	6
40	1

**2.6.1 [5] <§§2.2, 2.3>** Para os locais de memória na tabela anterior, escreva o código C classificando os dados do mais baixo ao mais alto, colocando o menor valor no menor local de memória mostrado na figura. Suponha que os dados mostrados representem a variável C chamada `Array`, que é um array do tipo `interface`, e que o primeiro número no array mostrado seja o primeiro elemento no array. Suponha que essa máquina em particular seja uma máquina endereçável por byte e uma word consista em 4 bytes.

**2.6.2 [5] <§2.2, 2.3>** Para os locais de memória na tabela anterior, escreva o código MIPS que classifique os dados do mais baixo ao mais alto, colocando o menor valor no menor local de memória. Use um número mínimo de instruções MIPS. Suponha que o endereço de base de `Array` esteja armazenado no registrador `$s6`.

**2.7 [5] <§2.3>** Mostre como o valor `0xabcdef12` seria arrumado na memória de uma máquina little-endian e uma máquina big-endian. Suponha que os dados sejam armazenados a partir do endereço 0.

**2.8 [5] <§2.4>** Traduza `0xabcdef12` para decimal.

**2.9 [5] <§§2.2, 2.3>** Traduza o código C a seguir para MIPS. Suponha que as variáveis `f`, `g`, `h`, `I` e `j` sejam atribuídas aos registradores `$s0`, `$s1`, `$s2`, `$s3` e `$s4`, respectivamente. Suponha que o endereço de base dos arrays `A` e `B` estejam nos registradores `$s6` e `$s7`, respectivamente. Suponha que os elementos dos arrays `A` e `B` sejam words de 4 bytes:

$$B[8] = A[i] + A[j];$$

**2.10 [5]** <§§2.2, 2.3> Traduza o código MIPS a seguir para C. Suponha que as variáveis f, g, h, i e j sejam atribuídas aos registradores \$s0, \$s1, \$s2, \$s3 e \$s4, respectivamente. Suponha que o endereço de base dos arrays A e B estejam nos registradores \$s6 e \$s7, respectivamente.

```
addi    $t0, $s6, 4
add     $t1, $s6, $0
sw      $t1, 0($t0)
lw      $t0, 0($t0)
add     $s0, $t1, $t0
```

**2.11 [5]** <§§2.3, 2.5> Para cada instrução MIPS, mostre o valor dos campos de opcode (OP), registrador fonte (RS) e registrador de destino (RT). Para as instruções tipo I, mostre o valor do campo imediato, e para as instruções tipo R, mostre o valor do campo de registrador de destino (RD).

**2.12** Suponha que os registradores \$s0 e \$s1 mantenham os valores  $0 \times 80000000$  e  $0 \times D0000000$ , respectivamente.

**2.12.1 [5]** <§2.4> Qual é o valor de \$t0 para o código assembly a seguir?

```
add $t0, $s0, $s1
```

**2.12.2 [5]** <§2.4> O resultado em \$t0 é o resultado desejado ou houve overflow?

**2.12.3 [5]** <§2.4> Para o conteúdo dos registradores \$s0 e \$s1, conforme especificado acima, qual é o valor de \$t0 para o código assembly a seguir?

```
sub $t0, $s0, $s1
```

**2.12.4 [5]** <§2.4> O resultado em \$t0 é o resultado desejado ou houve overflow?

**2.12.5 [5] <§2.4>** Para o conteúdo dos registradores \$s0 e \$s1 especificado acima, qual é o valor de \$t0 para o código assembly a seguir?

```
add $t0, $s0, $s1  
add $t0, $t0, $s0
```

**2.12.6 [5] <§2.4>** O resultado em \$t0 é o resultado desejado ou houve overflow?

**2.13** Suponha que \$s0 contenha o valor  $128_{dec}$ .

**2.13.1 [5] <§2.4>** Para a instrução add \$t0, \$s0, \$s1, qual é ou quais são as faixas de valores para \$s1 que resultaria(m) em overflow?

**2.13.2 [5] <§2.4>** Para a instrução sub \$t0, \$s0, \$s1, qual é ou quais são as faixas de valores para \$s1 que resultaria(m) em overflow?

**2.13.3 [5] <§2.4>** Para a instrução sub \$t0, \$s1, \$s0, qual é ou quais são as faixas de valores para \$s1 que resultaria(m) em overflow?

**2.14 [5] <§§2.4, 2.5>** Forneça o tipo e a instrução em linguagem assembly para o seguinte valor binário: 0000 0010 0001 0000 1000 0000 0010 0000<sub>bin</sub>.

**2.15 [5] <§§2.4, 2.5>** Forneça o tipo e a representação hexadecimal da seguinte instrução: sw \$t1, 32(\$t2)

**2.16 [5] <§2.5>** Forneça o tipo, a instrução em linguagem assembly e a representação binária da instrução descrita pelos seguintes campos MIPS:

op = 0, rs = 3, rt = 2, rd = 3, shamt = 0, funct = 34

**2.17 [5] <§2.5>** Forneça o tipo, a instrução em linguagem assembly e a representação binária da instrução descrita pelos seguintes campos MIPS:

op = 0 × 23, rs = 1, rt = 2, const = 0 × 4

**2.18 [5] <§2.5>** Suponha que quiséssemos expandir o arquivo de registradores MIPS para 128 registradores e expandir o conjunto de instruções para conter quatro vezes o número de instruções atuais.

**2.18.1 [5] <§2.5>** Como isso afetaria o tamanho de cada um dos campos de bit

nas instruções do tipo R?

**2.18.2** [5] <§2.5> Como isso afetaria o tamanho de cada um dos campos de bit nas instruções do tipo I?

**2.18.3** [5] <§§2.5, 2.10> Como cada uma das duas mudanças propostas diminuiria o tamanho de um programa em assembly MIPS? Por outro lado, como a mudança proposta aumentaria o tamanho de um programa em assembly MIPS?

**2.19** Considere o seguinte conteúdo de registradores:

$$\$t0 = 0 \times \text{AAAAAAA}, \$t1 = 0 \times 12345678$$

**2.19.1** [5] <§2.6> Para os valores de registradores mostrados acima, qual é o valor de \$t2 para a seguinte sequências de instruções?

sll \$t2, \$t0, 44 ou \$t2, \$t2, \$t1

**2.19.2** [5] <§2.6> Para os valores de registradores mostrados acima, qual é o valor de \$t2 para a seguinte sequências de instruções?

```
srl $t2, $t0, 4  
andi $t2, $t2, -1
```

**2.19.3** [5] <§2.6> Para os valores de registradores mostrados acima, qual é o valor de \$t2 para a seguinte sequência de instruções?

```
srl $t2, $t0, 3  
andi $t2, $t2, 0xFFE
```

**2.20** [5] <§2.6> Ache a sequência mais curta de instruções MIPS que extraia os bits de 16 até 11 do registrador \$t0 e use o valor desse campo para substituir os bits de 31 até 26 no registrador \$t1 sem alterar os outros 26 bits do registrador \$t1.

**2.21** [5] <§2.6> Forneça um conjunto mínimo de instruções MIPS que possa ser utilizado para implementar a seguinte pseudoinstrução:

```
not $t1, $t2 // bit-wise invert
```

- 2.22 [5] <§2.6>** Para a instrução C a seguir, escreva uma sequências mínima de instruções assembly do MIPS que faça a operação idêntica. Suponha que  $\$t1 = A$ ,  $\$t2 = B$ , e  $\$s1$  seja o endereço de base de C.

```
A = C[0] << 4;
```

- 2.23 [5] <§2.7>** Suponha que  $\$t0$  contenha o valor `0x00101000`. Qual é o valor de  $\$t2$  após as instruções a seguir?

```
slt $t2, $0, $t0  
bne $t2, $0, ELSE  
j DONE  
ELSE: addi $t2, $t2, 2  
DONE:
```

- 2.24 [5] <§2.7>** Suponha que o contador de programa (PC) seja definido em `0x2000 0000`. É possível usar a instrução assembly do MIPS jump (j) para definir o PC para o endereço como `0x4000 0000`? É possível usar a instrução assembly do MIPS branch-on-equal (beq) para definir o PC como esse mesmo endereço?

- 2.25** A instrução a seguir não está incluída no conjunto de instruções do MIPS:

```
rpt $t2, loop # se(R[rs]>0) R[rs]=R[rs]-1, PC=PC+4+BranchAddr
```

- 2.25.1 [5] <§2.7>** Se essa instrução tivesse que ser implementada no conjunto de instruções do MIPS, qual seria o formato de instrução mais apropriado?

- 2.25.2 [5] <§2.7>** Qual é a sensibilidade de instruções MIPS mais curta que realiza a mesma operação?

- 2.26** Considere o seguinte loop em MIPS:

```

LOOP: slt $t2, $0, $t1
      beq $t2, $0, DONE
      subi $t1, $t1, 1
      addi $s2, $s2, 2
      j LOOP

DONE:

```

**2.26.1 [5] <§2.7>** Suponha que o registrador \$t1 seja inicializado com o valor 10. Qual é o valor no registrador \$s2, supondo que \$s2 seja inicialmente zero?

**2.26.2 [5] <§2.7>** Para cada um dos loops acima, escreva a rotina equivalente em código C. Suponha que os registradores \$s1, \$s2, \$t1 e \$t2 sejam inteiros A, B, I e temp, respectivamente.

**2.26.3 [5] <§2.7>** Para os loops escritos em assembly MIPS acima, suponha que o registrador \$t1 seja inicializado com o valor N. Quantas instruções MIPS são executadas?

**2.27 [5] <§2.7>** Traduza o código C para o código assembly do MIPS. Use um número mínimo de instruções. Suponha que os valores de a, b, i e j estejam nos registradores \$s0, \$s1, \$t0, \$t1, respectivamente. Além disso, suponha que o registrador \$s2 mantenha o endereço de base do array D.

```

for(i = 0; i < a; i++)
    for(j = 0; j < b; j++)
        D[4*j] = i + j;

```

**2.28 [5] <§2.7>** Quantas instruções MIPS são necessárias para implementar o código C do Exercício 2.27? Se as variáveis a e b forem inicializadas como 10 e 1 e todos os elementos de D forem inicialmente 0, qual é o número total de instruções MIPS que são executadas para completar o loop?

**2.29 [5] <§2.7>** Traduza esses loops para C. Suponha que o inteiro i em nível de C seja mantido no registrador \$t1, \$s2 mantenha o inteiro em nível de C chamado result, e \$s0 mantenha o endereço de base do inteiro MemArray.

```

        addi $t1, $0, $0
LOOP:  lw    $s1, 0($s0)
        add  $s2, $s2, $s1
        addi $s0, $s0, 4
        addi $t1, $t1, 1
        slti $t2, $t1, 100
        bne  $t2, $s0, LOOP

```

**2.30 [5] <§2.7>** Reescreva o loop do Exercício 2.29 para reduzir o número de instruções MIPS executadas.

**2.31 [5] <§2.8>** Implemente o seguinte código C em assembly MIPS. Qual é o número total de instruções MIPS necessárias para executar a função?

```

int fib(int n){
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}

```

**2.32 [5] <§2.8>** As funções normalmente podem ser implementadas pelos compiladores “em linha”. Uma função em linha é quando o corpo da função é copiado para o espaço do programa, permitindo que o overhead da chamada de função seja eliminado. Implemente uma versão “em linha” do código C acima em assembly do MIPS. Qual é a redução no número total de instruções assembly do MIPS necessárias para completar a função? Suponha que a variável C, n, seja inicializada como 5.

**2.33 [5] <§2.8>** Para cada chamada de função, mostre o conteúdo da pilha após

a chamada de função ser feita. Suponha que o ponteiro de pilha esteja originalmente no endereço `0x7fffffff` e siga as convenções de registrador especificadas na [Figura 2.11](#).

**2.34** Traduza a função `f` para a linguagem assembly do MIPS. Se você precisar usar os registradores de `$t0` até `$t7`, use primeiro os registradores de número mais baixo. Suponha que a declaração de função para `func` seja “`int func(int a, int b);`”. O código para a função `f` é o seguinte:

```
int f(int a, int b, int c, int d){  
    return func(func(a,b),c + d);  
}
```

**2.35 [5] <§2.8>** Podemos usar a otimização “tail-call” nesta função? Se negativo, explique por que não. Se afirmativo, qual é a diferença no número de instruções executadas em `f` com e sem a otimização?

**2.36 [5] <§2.8>** Imediatamente antes que a sua função `f` do Exercício 2.34 retorne, o que sabemos sobre o conteúdo dos registradores `$t5`, `$s3`, `$ra` e `$sp`? Lembre-se de que sabemos o conteúdo da função `f`, mas, para a função `func`, só conhecemos sua declaração.

**2.37 [5] <§2.9>** Escreva um programa em linguagem assembly do MIPS para converter uma string de números ASCII contendo strings decimais inteiras positivas e negativas para um inteiro. Seu programa deverá esperar que o registrador `$a0` contenha o endereço de uma string terminada em nulo, contendo alguma combinação dos dígitos de 0 até 9. Seu programa deverá calcular o valor inteiro equivalente a essa string de dígitos, depois colocar o número no registrador `$v0`. Se um caractere que não seja um dígito aparecer em qualquer lugar da string, seu programa deverá parar com um valor -1 no registrador `$v0`. Por exemplo, se o registrador `$a0` apontar para uma sequência de três bytes  $50_{dec}$ ,  $52_{dec}$ ,  $0_{dec}$  (a string “24” terminada em nulo), então, quando o programa terminar, o registrador `$v0` deverá conter o valor  $24_{dec}$ .

**2.38 [5] <§2.9>** Considere o código a seguir:

```
lbu $t0, 0($t1)
sw $t0, 0($t2)
```

Suponha que o registrador \$t1 contenha o endereço 0x1000 0000 e o registrador \$t2 contenha o endereço 0x1000 0010. Observe que a arquitetura MIPS utiliza o endereçamento big-endian. Suponha que os dados (em hexadecimal) no endereço 0x1000 0000 sejam: 0x11223344. Que valor é armazenado no endereço apontado pelo registrador \$t2?

**2.39** [5] <§2.10> Escreva o código assembly MIPS que cria a constante de 32 bits 0010 0000 0000 0001 0100 1001 0010 0100<sub>bin</sub> e armazena esse valor no registrador \$t1.

**2.40** [5] <§§2.6, 2.10> Se o valor atual do PC é 0x00000000, você pode usar uma única instrução jump para chegar ao endereço do PC, como mostra o Exercício 2.39?

**2.41** [5] <§§2.6, 2.10> Se o valor atual do PC é 0x00000600, você pode usar uma única instrução branch para chegar ao endereço do PC, como mostra o Exercício 2.39?

**2.42** [5] <§§2.6, 2.10> Se o valor atual do PC é 0x1FFFF000, você pode usar uma única instrução branch para chegar ao endereço do PC, como mostra o Exercício 2.39?

**2.43** [5] <§2.11> Escreva o código assembly MIPS para implementar o seguinte código em C:

```
lock(1k);
shvar = max(shvar,x);
unlock(1k);
```

Suponha que o endereço da variável 1k esteja em \$a0, o endereço da variável shvar esteja em \$a1 e o valor da variável x esteja em \$a2. Sua seção crítica não deverá conter quaisquer chamadas de função. Use instruções ll/sc a fim de implementar a operação lock(), e a operação unlock() é simplesmente uma instrução store comum.

**2.44** [5] <§2.11> Repita o Exercício 2.43, mas desta vez use ll/sc para realizar uma atualização atômica da variável shvar diretamente, sem usar lock() e

`unlock()`. Observe que, neste problema, não existe uma variável 1k.

**2.45** [5] <§2.11> Usando o seu código do Exercício 2.43 como exemplo, explique o que acontece quando dois processadores começam a executar essa seção crítica ao mesmo tempo, supondo que cada processador executa exatamente uma instrução por ciclo.

**2.46** Suponha que, para determinado processador, o CPI das instruções aritméticas seja 1, o CPI das instruções load/store seja 10 e o CPI das instruções branch seja 3. Suponha que um programa tenha os seguintes desmembramentos de instrução: 500 milhões de instruções aritméticas, 300 milhões de instruções load/store e 100 milhões de instruções branch.

**2.46.1** [5] <§2.19> Suponha que instruções aritméticas novas, mais poderosas, sejam acrescentadas ao conjunto de instruções. Na média, com o uso dessas instruções aritméticas mais poderosas, podemos reduzir em 25% o número de instruções aritméticas necessárias para executar um programa, e em apenas 10% o custo do aumento do tempo de ciclo de clock. Essa é uma boa escolha de projeto? Por quê?

**2.46.2** [5] <§2.19> Suponha que achemos um meio de dobrar o desempenho das instruções aritméticas. Qual é o ganho de velocidade geral de nossa máquina? E se achássemos um modo de melhorar o desempenho das instruções aritméticas em 10 vezes?

**2.47** Suponha que, para determinado programa, 70% das instruções executadas sejam aritméticas, 10% sejam load/store e 20% sejam branch.

**2.47.1** [5] <§2.19> Dada a mistura de instruções e a suposição de que uma instrução aritmética usa 2 ciclos, uma instrução load/store usa 6 ciclos e uma instrução branch usa 3 ciclos, ache o CPI médio.

**2.47.2** [5] <§2.19> Para uma melhoria de 25% no desempenho, quantos ciclos, em média, uma instrução aritmética pode usar se instruções load/store e branch não tiverem qualquer melhoria?

**2.47.3** [5] <§2.19> Para uma melhoria de 50% no desempenho, quantos ciclos, em média, uma instrução aritmética pode usar se instruções load/store e branch não tiverem qualquer melhoria?

## Respostas das Seções “Verifique você mesmo”

§2.2, página 57: MIPS, C, Java

§2.3, página 62: 2) Muito lento

§2.4, página 68: 2)  $-8_{\text{dec}}$

§2.5, página 75: 4) sub \$s2, \$s0, \$s1

§2.6, página 78: Ambos. AND com um padrão de máscara de 1s deixa 0s em todo lugar menos no campo desejado. O deslocamento à esquerda pela quantidade correta remove os bits da esquerda do campo. O deslocamento à direita pela quantidade apropriada coloca o campo nos bits mais à direita da palavra, com 0s no restante da palavra. Observe que AND deixa o campo onde ele estava originalmente e o par deslocado move o campo para a parte mais à direita da palavra.

§2.7, página 83: I. Todos são verdadeiros. II. 1).

§2.8, página 92: Ambos são verdadeiros.

§2.9, página 97: I. 2) II. 3)

§2.10, página 104: I. 4)  $+ -128K$ . II. 6) um bloco de 256M. III. 4) s11

§2.11, página 107: Ambos são verdadeiros.

§2.12, página 115: 4) Independência de máquina.