

A

# Montadores, Link-editores e o Simulador SPIM

*James R. Larus*  
Microsoft Research  
Microsoft

*O receio do insulto sério não pode  
justificar sozinho a supressão  
da livre expressão.*

**Louis Brandeis**  
Whitney v. California, 1927

<b>A.1</b>	<b>Introdução</b>	<b>A-2</b>
<b>A.2</b>	<b>Montadores</b>	<b>A-7</b>
<b>A.3</b>	<b>Link-editores</b>	<b>A-13</b>
<b>A.4</b>	<b>Carga</b>	<b>A-14</b>
<b>A.5</b>	<b>Uso da memória</b>	<b>A-14</b>
<b>A.6</b>	<b>Convenção para chamadas de procedimento</b>	<b>A-16</b>
<b>A.7</b>	<b>Exceções e interrupções</b>	<b>A-24</b>
<b>A.8</b>	<b>Entrada e saída</b>	<b>A-28</b>
<b>A.9</b>	<b>SPIM</b>	<b>A-29</b>
<b>A.10</b>	<b>Assembly do MIPS R2000</b>	<b>A-33</b>
<b>A.11</b>	<b>Comentários finais</b>	<b>A-60</b>
<b>A.12</b>	<b>Exercícios</b>	<b>A-61</b>

## A.1

### Introdução

Codificar instruções como números binários é algo natural e eficiente para os computadores. Os humanos, porém, têm muita dificuldade para entender e manipular esses números. As pessoas lêem e escrevem símbolos (palavras) muito melhor do que longas seqüências de dígitos. O Capítulo 2 mostrou que não precisamos escolher entre números e palavras, pois as instruções do computador podem ser representadas de muitas maneiras. Os humanos podem escrever e ler símbolos, e os computadores podem executar os números binários equivalentes. Este apêndice descreve o processo pelo qual um programa legível ao ser humano é traduzido para um formato que um computador pode executar, oferece algumas dicas sobre a escrita de programas em assembly e explica como executar esses programas no SPIM, um simulador que executa programas MIPS. As versões UNIX, Windows e Mac OS X do simulador SPIM estão disponíveis no CD.

*Assembly* é a representação simbólica da codificação binária – **linguagem de máquina** – de um computador. O assembly é mais legível do que a linguagem de máquina porque utiliza símbolos no lugar de bits. Os símbolos no assembly nomeiam padrões de bits que ocorrem comumente, como opcodes (códigos de operação) e especificadores de registradores, de modo que as pessoas possam ler e lembrar-se deles. Além disso, o assembly permite que os programadores utilizem *rótulos* para identificar e nomear palavras particulares da memória que mantêm instruções ou dados.

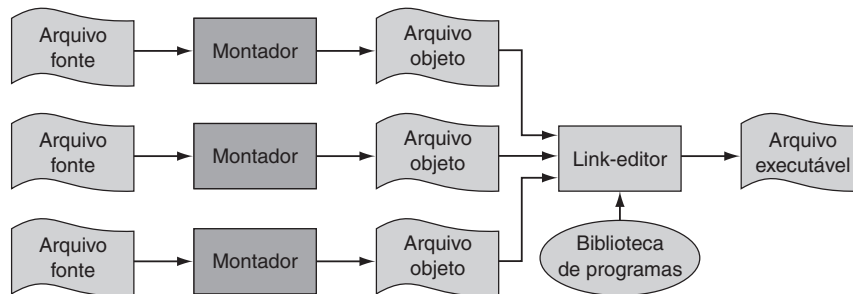
Uma ferramenta chamada **montador** traduz do assembly para instruções binárias. Os montadores oferecem uma representação mais amigável do que os 0s e 1s de um computador, o que simplifica a escrita e a leitura de programas. Nomes simbólicos para operações e locais são uma faceta dessa representação. Outra faceta são as facilidades de programação que aumentam a clareza de um programa. Por exemplo, as **macros**, discutidas na Seção A.2, permitem que um programador estenda o assembly, definindo novas operações.

Um montador lê um único *arquivo fonte* em assembly e produz um *arquivo objeto* com instruções de máquina e informações de trabalho que ajudam a combinar vários arquivos-objeto em um programa. A Figura A.1.1 ilustra como um programa é montado. A maioria dos programas consiste em vários arquivos – também chamados de *módulos* – que são escritos, compilados e montados de forma independente. Um programa também pode usar rotinas pré-escritas fornecidas em uma *biblioteca de programa*. Um módulo normalmente contém *referências* a sub-rotinas e dados definidos em outros módulos e em bibliotecas. O código em um módulo não pode ser executado quan-

**linguagem de máquina**  
A representação binária utilizada para a comunicação dentro de um sistema computacional.

**montador**  
Um programa que traduz uma versão simbólica de uma instrução para a versão binária.

**macro** Uma facilidade de combinação e substituição de padrões que oferece um mecanismo simples para nomear uma seqüência de instruções utilizada com freqüência.



**FIGURA A.1.1 O processo que produz um arquivo executável.** Um montador traduz um arquivo em assembly para um arquivo objeto, que é link-editado a outros arquivos e bibliotecas para um arquivo executável.

**referência não resolvida** Uma referência que exige mais informações de um arquivo externo para estar completa.

**link-editor** Também chamado linker. Um programa de sistema que combina programas em linguagem de máquina montados independentemente e resolve todos os rótulos indefinidos em um arquivo executável.

**diretiva do montador** Uma operação que diz ao montador como traduzir um programa, mas não produz instruções de máquina; sempre começa com um ponto.

do contém **referências não resolvidas** para rótulos em outros arquivos-objeto ou bibliotecas. Outra ferramenta, chamada **link-editor**, combina uma coleção de arquivos-objeto e biblioteca em um *arquivo executável*, que um computador pode executar.

Para ver a vantagem do assembly, considere a seguinte seqüência de figuras, todas contendo uma pequena sub-rotina que calcula e imprime a soma dos quadrados dos inteiros de 0 a 100. A Figura A.1.2 mostra a linguagem de máquina que um computador MIPS executa. Com esforço considerável, você poderia usar as tabelas de formato de opcode e instrução do Capítulo 2 para traduzir as instruções em um programa simbólico, semelhante à Figura A.1.3. Essa forma de rotina é muito mais fácil de ler, pois as operações e os operandos estão escritos com símbolos, em vez de padrões de bits. No entanto, esse assembly ainda é difícil de acompanhar, pois os locais da memória são indicados por seus endereços, e não por um rótulo simbólico.

A Figura A.1.4 mostra o assembly que rotula endereços de memória com nomes mnemônicos. A maior parte dos programadores prefere ler e escrever dessa forma. Os nomes que começam com um ponto, por exemplo, `.data` e `.globl`, são **diretivas do montador**, que dizem ao montador como traduzir um programa, mas não produzem instruções de máquina. Nomes seguidos por um sinal de dois pontos, como `str` ou `main`, são rótulos que dão nome ao próximo local da memória. Esse programa é tão legível quanto a maioria dos programas em assembly (exceto por uma óbvia falta de comentários), mas ainda é difícil de acompanhar, pois muitas operações simples são exigidas para realizar ta-

```

0010011110111101111111111111100000
1010111110111111100000000000010100
10101111101001001000000000000100000
101011111010010100000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000011100111000000000000011001
00100101110010000000000000000001
0010100100000010000000001100101
10101111101010000000000000011100
000000000000000001111100000010010
00000011000011111100100000100001
0001010000100000111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
0000001111100000000000000001000
000000000000000000100000100001
    
```

**FIGURA A.1.2 Código em linguagem de máquina MIPS para uma rotina que calcula e imprime a soma dos quadrados dos inteiros entre 0 a 100.**

```

addiu    $29, $29, -32
sw       $31, 20($29)
sw       $4, 32($29)
sw       $5, 36($29)
sw       $0, 24($29)
sw       $0, 28($29)
lw       $14, 28($29)
lw       $24, 24($29)
multu    $14, $14
addiu    $8, $14, 1
slti     $1, $8, 101
sw       $8, 28($29)
mflo     $15
addu     $25, $24, $15
bne      $1, $0, -9
sw       $25, 24($29)
lui      $4, 4096
lw       $5, 24($29)
jal      1048812
addiu    $4, $4, 1072
lw       $31, 20($29)
addiu    $29, $29, 32
jr       $31
move     $2, $0
  
```

**FIGURA A.1.3 A mesma rotina escrita em assembly.** Entretanto, o código para a rotina não rotula registradores ou locais de memória, nem inclui comentários.

refas simples e porque a falta de construções de fluxo de controle do assembly oferece poucos palpites sobre a operação do programa.

Ao contrário, a rotina em C na Figura A.1.5 é mais curta e mais clara, pois as variáveis possuem nomes mnemônicos e o loop é explícito, em vez de construído com desvios. Na verdade, a rotina em C é a única que escrevemos. As outras formas do programa foram produzidas por um compilador C e um montador.

Em geral, o assembly desempenha duas funções (ver Figura A.1.6). A primeira função é como uma linguagem de saída dos compiladores. Um *compilador* traduz um programa escrito em uma *linguagem de alto nível* (como C ou Pascal) para um programa equivalente em linguagem de máquina ou assembly. A linguagem de alto nível é chamada de **linguagem fonte**, e a saída do compilador é sua *linguagem destino*.

A outra função do assembly é como uma linguagem para a escrita de programas. Essa função costumava ser a dominante. Hoje, porém, devido a memórias maiores e compiladores melhores, a maioria dos programadores utiliza uma linguagem de alto nível e raramente ou nunca vê as instruções que um computador executa. Apesar disso, o assembly ainda é importante para a escrita de programas em que a velocidade ou o tamanho são críticos, ou para explorar recursos do hardware que não possuem correspondentes nas linguagens de alto nível.

Embora este apêndice focalize o assembly do MIPS, a programação assembly na maioria das outras máquinas é muito semelhante. As outras instruções e os modos de endereçamento nas máquinas CISC, como o VAX, podem tornar os programas assembly mais curtos, mas não mudam o processo de montagem de um programa, nem oferecem ao assembly as vantagens das linguagens de alto nível, como verificação de tipos e fluxo de controle estruturado.

**linguagem fonte** A linguagem de alto nível em que um programa é escrito originalmente.

```

        .text
        .align    2
        .globl   main
main:
        subu    $sp,$sp,32
        sw     $ra,20($sp)
        sd     $a0,32($sp)
        sw     $0,24($sp)
        sw     $0,28($sp)
loop:
        lw     $t6,28($sp)
        mul   $t7,$t6,$t6
        lw     $t8,24($sp)
        addu  $t9,$t8,$t7
        sw     $t9,24($sp)
        addu  $t0,$t6,1
        sw     $t0,28($sp)
        ble   $t0,100,loop
        la    $a0,str
        lw    $a1,24($sp)
        jal   printf
        move  $v0,$0
        lw    $ra,20($sp)
        addu  $sp,$sp,32
        jr    $ra

        .data
        .align    0
str:
        .asciiz  "The sum from 0 ..100 is %d \n"

```

**FIGURA A.1.4 A mesma rotina escrita em assembly com rótulos, mas sem comentários.** Os comandos que começam com pontos são diretivas do montador (ver páginas A-34 a A-35). `.text` indica que as linhas seguintes contêm instruções. `.data` indica que elas contêm dados. `.align n` indica que os itens nas linhas seguintes devem ser alinhados em um limite de  $2^n$  bytes. Logo, `.align 2` significa que o próximo item deverá estar em um limite de word. `.globl main` declara que `main` é um símbolo global, que deverá ser visível ao código armazenado em outros arquivos. Finalmente, `.asciiz` armazena na memória uma string terminada em nulo.

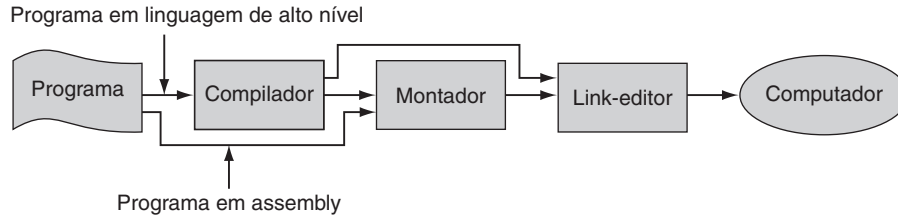
```

#include <stdio.h>

int
main (int argc,char *argv [ ])
{
    int i;
    int sum =0;
    for (i =0;i <=100;i =i +1)sum =sum +i *i;
    printf ("The sum from 0 ..100 is %d \n",sum);
}

```

**FIGURA A.1.5 A rotina escrita na linguagem de programação C.**



**FIGURA A.1.6** O assembly é escrito por um programador ou é a saída de um compilador.

## Quando usar o assembly

O motivo principal para programar em assembly, em vez de uma linguagem de alto nível disponível, é que a velocidade ou o tamanho de um programa possuem importância crítica. Por exemplo, imagine um computador que controla um mecanismo qualquer, como os freios de um carro. Um computador incorporado em outro dispositivo, como um carro, é chamado de *computador embutido*. Esse tipo de computador precisa responder rápida e previsivelmente aos eventos no mundo exterior. Como um compilador introduz incerteza sobre o custo de tempo das operações, os programadores podem achar difícil garantir que um programa em linguagem de alto nível responderá dentro de um intervalo de tempo definido – digamos, 1 milissegundo após um sensor detectar que um pneu está derrapando. Um programador assembly, por outro lado, possui mais controle sobre as instruções executadas. Além disso, em aplicações embutidas, reduzir o tamanho de um programa, de modo que caiba em menos chips de memória, reduz o custo do computador embutido.

Uma técnica híbrida, em que a maior parte de um programa é escrita em uma linguagem de alto nível e seções críticas são escritas em assembly, aproveita os pontos fortes das duas linguagens. Os programas normalmente gastam a maior parte do seu tempo executando uma pequena fração do código-fonte do programa. Essa observação é exatamente o princípio da localidade em que as caches se baseiam (ver Seção 7.2, no Capítulo 7).

O perfil do programa mede onde um programa gasta seu tempo e pode localizar as partes de tempo crítico de um programa. Em muitos casos, essa parte do programa pode se tornar mais rápida com melhores estruturas de dados ou algoritmos. No entanto, às vezes, melhorias de desempenho significativas só são obtidas com a recodificação de uma parte crítica de um programa em assembly.

Essa melhoria não é necessariamente uma indicação de que o compilador da linguagem de alto nível falhou. Os compiladores costumam ser melhores do que os programadores na produção uniforme de código de máquina de alta qualidade por um programa inteiro. Entretanto, os programadores entendem os algoritmos e o comportamento de um programa em um nível mais profundo do que um compilador e podem investir esforço e engenhosidade consideráveis melhorando pequenas seções do programa. Em particular, os programadores em geral consideram vários procedimentos simultaneamente enquanto escrevem seu código. Os compiladores compilam cada procedimento isoladamente e precisam seguir convenções estritas governando o uso dos registradores nos limites de procedimento. Retendo valores comumente usados nos registradores, até mesmo entre os limites dos procedimentos, os programadores podem fazer um programa ser executado com mais rapidez.

Outra vantagem importante do assembly está na capacidade de explorar instruções especializadas, por exemplo, instruções de cópia de string ou combinação de padrões. Os compiladores, na maior parte dos casos, não podem determinar se um loop de programa pode ser substituído por uma única instrução. Contudo, o programador que escreveu o loop pode substituí-lo facilmente por uma única instrução.

Hoje, a vantagem de um programador sobre um compilador tornou-se difícil de manter, pois as técnicas de compilação melhoram e os pipelines das máquinas aumentaram de complexidade (Capítulo 6).

O último motivo para usar assembly é que não existe uma linguagem de alto nível disponível em um computador específico. Computadores muito antigos ou especializados não possuem um compilador, de modo que a única alternativa de um programador é o assembly.

## Desvantagens do assembly

O assembly possui muitas desvantagens, que argumentam fortemente contra seu uso generalizado. Talvez sua principal desvantagem seja que os programas escritos em assembly são inerentemente específicos à máquina e precisam ser reescritos para serem executados em outra arquitetura de computador. A rápida evolução dos computadores, discutida no Capítulo 1, significa que as arquiteturas se tornam obsoletas. Um programa em assembly permanece firmemente ligado à sua arquitetura original, mesmo depois que o computador for substituído por máquinas mais novas, mais rápidas e mais econômicas.

Outra desvantagem é que os programas em assembly são maiores do que os programas equivalentes escritos em uma linguagem de alto nível. Por exemplo, o programa em C da Figura A.1.5 possui 11 linhas de extensão, enquanto o programa em assembly da Figura A.1.4 possui 31 linhas. Em programas mais complexos, a razão entre o assembly e a linguagem de alto nível (seu *fator de expansão*) pode ser muito maior do que o fator de três nesse exemplo. Infelizmente, estudos empíricos mostraram que os programadores escrevem quase o mesmo número de linhas de código por dia em assembly e em linguagens de alto nível. Isso significa que os programadores são aproximadamente  $x$  vezes mais produtivos em uma linguagem de alto nível, onde  $x$  é o fator de expansão do assembly.

Para aumentar o problema, programas maiores são mais difíceis de ler e entender e contêm mais bugs. O assembly realça esse problema, devido à sua completa falta de estrutura. Os idiomas de programação comuns, como instruções *if-then* e loops, precisam ser criados a partir de desvios e jumps. Os programas resultantes são difíceis de ler, pois o leitor precisa recriar cada construção de nível mais alto a partir de suas partes e cada instância de uma instrução pode ser ligeiramente diferente. Por exemplo, veja a Figura A.1.4 e responda a estas perguntas: que tipo de loop é utilizado? Quais são seus limites inferior e superior?

**Detalhamento:** os compiladores podem produzir linguagem de máquina diretamente, em vez de contar com um montador. Esses compiladores executam muito mais rapidamente do que aqueles que invocam um montador como parte da compilação. Todavia, um compilador que gera linguagem de máquina precisa realizar muitas tarefas que um montador normalmente trata, como resolver endereços e codificar instruções como números binários. A escolha é entre velocidade de compilação e simplicidade do compilador.

**Detalhamento:** apesar dessas considerações, algumas aplicações embutidas são escritas em uma linguagem de alto nível. Muitas dessas aplicações são programas grandes e complexos, que precisam ser muito confiáveis. Os programas em assembly são maiores e mais difíceis de escrever e ler do que os programas em linguagem de alto nível. Isso aumenta bastante o custo da escrita de um programa em assembly e torna muito difícil verificar a exatidão desse tipo de programa. Na verdade, essas considerações levaram o Departamento de Defesa, que paga por muitos sistemas embutidos complexos, a desenvolver a Ada, uma nova linguagem de alto nível para a escrita de sistemas embutidos.

## A.2

## Montadores

Um montador traduz um arquivo de instruções em assembly para um arquivo de instruções de máquina binárias e dados binários. O processo de tradução possui duas etapas principais. A primeira etapa é encontrar locais de memória com rótulos, de modo que o relacionamento entre os nomes simbólicos e endereços é conhecido quando as instruções são traduzidas. A segunda etapa é traduzir cada instrução assembly combinando os equivalentes numéricos dos opcodes, especificadores de registradores e rótulos em uma instrução válida. Como vemos na Figura A.1.1, o montador produz um arquivo de saída, chamado de *arquivo objeto*, que contém as instruções de máquina, dados e informações de manutenção.



Um arquivo objeto normalmente não pode ser executado porque referencia procedimentos ou dados em outros arquivos. Um **rótulo é externo** (também chamado **global**) se o objeto rotulado puder ser referenciado a partir de arquivos diferentes de onde está definido. Um rótulo é *local* se o objeto só puder ser usado dentro do arquivo em que está definido. Na maior parte dos montadores, os rótulos são locais por padrão e precisam ser declarados como globais explicitamente. As sub-rotinas e variáveis globais exigem rótulos externos, pois são referenciados a partir de muitos arquivos em um programa. **Rótulos locais** ocultam nomes que não devem ser visíveis a outros módulos – por exemplo, funções estáticas em C, que só podem ser chamadas por outras funções no mesmo arquivo. Além disso, nomes gerados pelo compilador – por exemplo, um nome para a instrução no início de um loop – são locais, de modo que o compilador não precisa produzir nomes exclusivos em cada arquivo.

**rótulo externo** Também chamado rótulo global. Um rótulo que se refere a um objeto que pode ser referenciado a partir de arquivos diferentes daquele em que está definido.

**rótulo local** Um rótulo que se refere a um objeto que só pode ser usado dentro do arquivo em que está definido.

## RÓTULOS LOCAIS E GLOBAIS

Considere o programa na Figura A.1.4. A sub-rotina possui um rótulo externo (global) `main`. Ela também contém dois rótulos locais – `loop` e `str` – visíveis apenas dentro do seu arquivo em assembly. Finalmente, a rotina também contém uma referência não resolvida a um rótulo externo `printf`, que é a rotina da biblioteca que imprime valores. Quais rótulos na Figura A.1.4 poderiam ser referenciados a partir de outro arquivo?

Somente os rótulos globais são visíveis fora de um arquivo, de modo que o único rótulo que poderia ser referenciado por outro arquivo é `main`.

### EXEMPLO

### RESPOSTA

Como o montador processa cada arquivo em um programa individual e isoladamente, ele só sabe os endereços dos rótulos locais. O montador depende de outra ferramenta, o link-editor, para combinar uma coleção de arquivos-objeto e bibliotecas em um arquivo executável, resolvendo os rótulos externos. O montador auxilia o link-editor, oferecendo listas de rótulos e referências não resolvidas.

No entanto, até mesmo os rótulos locais apresentam um desafio interessante a um montador. Ao contrário dos nomes na maioria das linguagens de alto nível, os rótulos em assembly podem ser usados antes de serem definidos. No exemplo, na Figura A.1.4, o rótulo `str` é usado pela instrução `la` antes de ser definido. A possibilidade de uma **referência à frente**, como essa, força um montador a traduzir um programa em duas etapas: primeiro encontre todos os rótulos e depois produza as instruções. No exemplo, quando o montador vê a instrução `la`, ele não sabe onde a palavra rotulada com `str` está localizada ou mesmo se `str` rotula uma instrução ou um dado.

A primeira passada de um montador lê cada linha de um arquivo em assembly e a divide em suas partes componentes. Essas partes, chamadas *lexemas*, são palavras, números e caracteres de pontuação individuais. Por exemplo, a linha

```
ble $t0, 100, loop
```

contém seis lexemas: o opcode `ble`, o especificador de registrador `$t0`, uma vírgula, o número `100`, uma vírgula e o símbolo `loop`.

Se uma linha começa com um rótulo, o montador registra em sua **tabela de símbolos** o nome do rótulo e o endereço da word de memória que a instrução ocupa. O montador, então, calcula quantas words de memória ocupará a instrução na linha atual. Acompanhando os tamanhos das instruções, o montador pode determinar onde a próxima instrução entrará. Para calcular o tamanho de uma instrução de tamanho variável, como aquelas no VAX, um montador precisa examiná-la em detalhes. Por outro lado, instruções de tamanho fixo, como aquelas no MIPS, exigem apenas um exame superficial. O montador realiza um cálculo semelhante para calcular o espaço exigido para instruções de dados. Quando o montador atinge o final de um arquivo assembly, a tabela de símbolos registra o local de cada rótulo definido no arquivo.

**referência à frente**  
Um rótulo usado antes de ser definido.

**tabela de símbolos**  
Uma tabela que faz a correspondência entre os nomes dos rótulos e os endereços das words de memória que as instruções ocupam.



O montador utiliza as informações na tabela de símbolos durante uma segunda passada pelo arquivo, que, na realidade, produz o código de máquina. O montador novamente examina cada linha no arquivo. Se a linha contém uma instrução, o montador combina as representações binárias de seu opcode e operandos (especificadores de registradores ou endereço de memória) em uma instrução válida. O processo é semelhante ao usado na Seção 2.4 do Capítulo 2. As instruções e as words de dados que referenciam um símbolo externo definido em outro arquivo não podem ser completamente montadas (elas não estão resolvidas) porque o endereço do símbolo não está na tabela de símbolos. Um montador não reclama sobre referências não resolvidas porque o rótulo correspondente provavelmente estará definido em outro arquivo.

## Colocando em perspectiva

**backpatching** Um método para traduzir do assembly para instruções de máquina, em que o montador monta uma representação binária (possivelmente incompleta) de cada instrução em uma passada por um programa e depois retorna para preencher rótulos previamente indefinidos.

O assembly é uma linguagem de programação. Sua principal diferença das linguagens de alto nível, como BASIC, Java e C, é que o assembly oferece apenas alguns tipos simples de dados e fluxo de controle. Os programas em assembly não especificam o tipo de valor mantido em uma variável. Em vez disso, um programador precisa aplicar as operações apropriadas (por exemplo, adição de inteiro ou ponto flutuante) a um valor. Além disso, em assembly, os programas precisam implementar todo o fluxo de controle com *go tos*. Os dois fatores tornam a programação em assembly para qualquer máquina – MIPS ou 80x86 – mais difícil e passível de erro do que a escrita em uma linguagem de alto nível.

**Detalhamento:** se a velocidade de um montador for importante, esse processo em duas etapas pode ser feito em uma passada pelo arquivo assembly com uma técnica conhecida como **backpatching**. Em sua passada pelo arquivo, o montador monta uma representação binária (possivelmente incompleta) de cada instrução. Se a instrução referencia um rótulo ainda não definido, o montador consulta essa tabela para encontrar todas as instruções que contêm uma referência à frente ao rótulo. O montador volta e corrige sua representação binária para incorporar o endereço do rótulo. O backpatching agiliza o assembly porque o montador só lê sua entrada uma vez. Contudo, isso exige que um montador mantenha uma representação binária inteira de um programa na memória, de modo que as instruções possam sofrer backpatching. Esse requisito pode limitar o tamanho dos programas que podem ser montados. O processo é complicado por máquinas com diversos tipos de desvios que se espalham por diferentes intervalos de instruções. Quando o montador vê inicialmente um rótulo não resolvido em uma instrução de desvio, ele precisa usar o maior desvio possível ou arriscar ter de voltar e reajustar muitas instruções para criar espaço para um desvio maior.

### segmento de texto

O segmento de um arquivo objeto do UNIX que contém o código em linguagem de máquina para as rotinas no arquivo de origem.

### segmento de dados

O segmento de um objeto ou arquivo executável do UNIX que contém uma representação binária dos dados inicializados, usados pelo programa.

## Formato do arquivo objeto

Os montadores produzem arquivos objeto. Um arquivo objeto no UNIX contém seis seções distintas (ver Figura A.2.1):

- O **cabeçalho do arquivo objeto** descreve o tamanho e a posição das outras partes do arquivo.
- O **segmento de texto** contém o código em linguagem de máquina para rotinas no arquivo de origem. Essas rotinas podem ser não-executáveis devido a referências não resolvidas.
- O **segmento de dados** contém uma representação binária dos dados no arquivo de origem. Os dados também podem estar incompletos devido a referências não resolvidas a rótulos em outros arquivos.

Cabeçalho do arquivo objeto	Segmento de texto	Segmento de dados	Informações de relocação	Tabela de símbolos	Informações de depuração
-----------------------------	-------------------	-------------------	--------------------------	--------------------	--------------------------

**FIGURA A.2.1 Arquivo objeto.** Um montador do UNIX produz um arquivo objeto com seis seções distintas.

- As **informações de relocação** identificam instruções e words de dados que dependem de **endereços absolutos**. Essas referências precisam mudar se partes do programa forem movidas na memória.
- A *tabela de símbolos* associa endereços a rótulos externos no arquivo de origem e lista referências não resolvidas.
- As *informações de depuração* contêm uma descrição concisa do modo como o programa foi compilado, de modo que um depurador possa descobrir quais endereços de instrução correspondem às linhas em um arquivo de origem e imprimir as estruturas de dados em formato legível.

**informações de relocação** O segmento de um arquivo objeto do UNIX que identifica instruções e words de dados que dependem de endereços absolutos.

**endereço absoluto** O endereço real na memória de uma variável ou rotina.

O montador produz um arquivo objeto que contém uma representação binária do programa e dos dados, além de informações adicionais para ajudar a link-editar partes de um programa. Essas informações de relocação são necessárias porque o montador não sabe quais locais da memória um procedimento ou parte de dados ocupará depois de ser link-editado com o restante do programa. Os procedimentos e dados de um arquivo são armazenados em uma parte contígua da memória, mas o montador não sabe onde essa memória estará localizada. O montador também passa algumas entradas da tabela de símbolos para o link-editor. Em particular, o montador precisa registrar quais símbolos externos são definidos em um arquivo e quais referências não resolvidas ocorrem em um arquivo.

**Detalhamento:** por conveniência, os montadores consideram que cada arquivo começa no mesmo endereço (por exemplo, posição 0) com a expectativa de que o link-editor *reposicione* o código e os dados quando receberem locais na memória. O montador produz *informações de relocação*, que contém uma entrada descrevendo cada instrução ou word de dados no arquivo que referencia um endereço absoluto. No MIPS, somente as instruções call, load e store da sub-rotina referenciam endereços absolutos. As instruções que usam endereçamento relativo ao PC, como desvios, não precisam ser relocadas.

## Facilidades adicionais

Os montadores oferecem diversos recursos convenientes que ajudam a tornar os programas em assembly mais curtos e mais fáceis de escrever, mas não mudam fundamentalmente o assembly. Por exemplo, as *diretivas de layout de dados* permitem que um programador descreva os dados de uma maneira mais concisa e natural do que sua representação binária.

Na Figura A.1.4, a diretiva

```
.ascii "The sum from 0 .. 100 is %d\n"
```

armazena caracteres da string na memória. Compare essa linha com a alternativa de escrever cada caractere como seu valor ASCII (a Figura 2.21, no Capítulo 2, descreve a codificação ASCII para os caracteres):

```
.byte 84, 104, 101, 32, 115, 117, 109, 32  
.byte 102, 114, 111, 109, 32, 48, 32, 46  
.byte 46, 32, 49, 48, 48, 32, 105, 115  
.byte 32, 37, 100, 10, 0
```

A diretiva `.ascii` é mais fácil de ler porque representa caracteres como letras, e não como números binários. Um montador pode traduzir caracteres para sua representação binária muito mais rapidamente e com mais precisão do que um ser humano. As diretivas de layout de dados especificam os dados em um formato legível aos seres humanos, que um montador traduz para binário. Outras diretivas de layout são descritas na Seção A.10.

**DIRETIVA DE STRING****EXEMPLO**

Defina a seqüência de bytes produzida por esta diretiva:

```
.asciiz "The quick brown fox jumps over the lazy dog"
```

**RESPOSTA**

```
.byte 84, 104, 101, 32, 113, 117, 105, 99
.byte 107, 32, 98, 114, 111, 119, 110, 32
.byte 102, 111, 120, 32, 106, 117, 109, 112
.byte 115, 32, 111, 118, 101, 114, 32, 116
.byte 104, 101, 32, 108, 97, 122, 121, 32
.byte 100, 111, 103, 0
```

As *macros* são uma facilidade de combinação e troca de padrão, que oferece um mecanismo simples para nomear uma seqüência de instruções usada com freqüência. Em vez de digitar repetidamente as mesmas instruções toda vez que forem usadas, um programador chama a macro e o montador substitui a chamada da macro pela seqüência de instruções correspondente. As macros, como as sub-rotinas, permitem que um programador crie e nomeie uma nova abstração para uma operação comum. No entanto, diferente das sub-rotinas, elas não causam uma chamada e um retorno de sub-rotina quando o programa é executado, pois uma chamada de macro é substituída pelo corpo da macro quando o programa é montado. Depois dessa troca, a montagem resultante é indistinguível do programa equivalente, escrito sem macros.

**MACROS****EXEMPLO**

Como um exemplo, suponha que um programador precise imprimir muitos números. A rotina de biblioteca `printf` aceita uma string de formato e um ou mais valores para imprimir como seus argumentos. Um programador poderia imprimir o inteiro no registrador `$7` com as seguintes instruções:

```
.data
int_str: .asciiz"%d"
.text
la    $a0, int_str # Carrega endereço da string
                        # no primeiro argumento
mov   $a1, $7      # Carrega o valor no
                        # segundo argumento
jal   printf      # Chama a rotina printf
```

A diretiva `.data` diz ao montador para armazenar a string no segmento de dados do programa, e a diretiva `.text` diz ao montador para armazenar as instruções em seu segmento de texto.

Entretanto, a impressão de muitos números dessa maneira é tediosa e produz um programa extenso, difícil de ser entendido. Uma alternativa é introduzir uma macro, `print_int`, para imprimir um inteiro:

```
.data
int_str: .asciiz"%d"
.text
.macro print_int($arg)
la    $a0, int_str # Carrega endereço da string
                        # no primeiro argumento
mov   $a1, $arg    # Carrega o parâmetro da macro
                        # ($arg) no segundo argumento
```

```
        jal printf      # Chama a rotina printf
        .end_macro
print_int($7)
```

A macro possui um **parâmetro formal**, `$arg`, que nomeia o argumento da macro. Quando a macro é expandida, o argumento de uma chamada é substituído pelo parâmetro formal em todo o corpo da macro. Depois, o montador substitui a chamada pelo corpo recém-expandido da macro. Na primeira chamada em `print_int`, o argumento é `$7`, de modo que a macro se expande para o código

```
la $a0, int_str
mov $a1, $7
jal printf
```

Em uma segunda chamada em `print_int`, digamos, `print_int($t0)`, o argumento é `$t0`, de modo que a macro expande para

```
la $a0, int_str
mov $a1, $t0
jal printf
```

Para o que a chamada `print_int($a0)` se expande?

```
la $a0, int_str
mov $a1, $a0
jal printf
```

Esse exemplo ilustra uma desvantagem das macros. Um programador que utiliza essa macro precisa estar ciente de que `print_int` utiliza o registrador `$a0` e por isso não pode imprimir corretamente o valor nesse registrador.

**parâmetro formal** Uma variável que é o argumento de um procedimento ou macro; substituída por esse argumento quando a macro é expandida.

**RESPOSTA**

---

## Interface hardware/software

Alguns montadores também implementam *pseudo-instruções*, que são instruções fornecidas por um montador mas não implementadas no hardware. O Capítulo 2 contém muitos exemplos de como o montador MIPS sintetiza pseudo-instruções e modos de endereçamento do conjunto de instruções de hardware do MIPS. Por exemplo, a Seção 2.6, no Capítulo 2, descreve como o montador sintetiza a instrução `blt` a partir de duas outras instruções: `slt` e `bne`. Estendendo o conjunto de instruções, o montador MIPS torna a programação em assembly mais fácil sem complicar o hardware. Muitas pseudo-instruções também poderiam ser simuladas com macros, mas o montador MIPS pode gerar um código melhor para essas instruções, pois pode usar um registrador dedicado (`$at`) e é capaz de otimizar o código gerado.

---

**Detalhamento:** os montadores *montam condicionalmente* partes de código, o que permite que um programador inclua ou exclua grupos de instruções quando um programa é montado. Esse recurso é particularmente útil quando várias versões de um programa diferem por um pequeno valor. Em vez de manter esses programas em arquivos separados – o que complica bastante o reparo de bugs no código comum –, os programadores normalmente mesclam as versões em um único arquivo. O código particular a uma versão é montado condicionalmente, de modo que possa ser excluído quando outras versões do programa forem montadas.

Se as macros e a montagem condicional são tão úteis, por que os montadores para sistemas UNIX nunca ou quase nunca as oferecem? Um motivo é que a maioria dos programadores nesses sistemas escreve programas em linguagens de alto nível, como C. A maior parte do código assembly é produzida por compiladores, que acham mais conveniente repetir o código do que definir macros. Outro motivo é que outras ferramentas no UNIX – como `cpp`, o pré-processador C, ou `m4`, um processador de macro de uso geral – podem oferecer macros e montagem condicional para programas em assembly.

## A.3

## Link-editores

**compilação separada** Dividir um programa em muitos arquivos, cada qual podendo ser compilado sem conhecimento do que está nos outros arquivos.

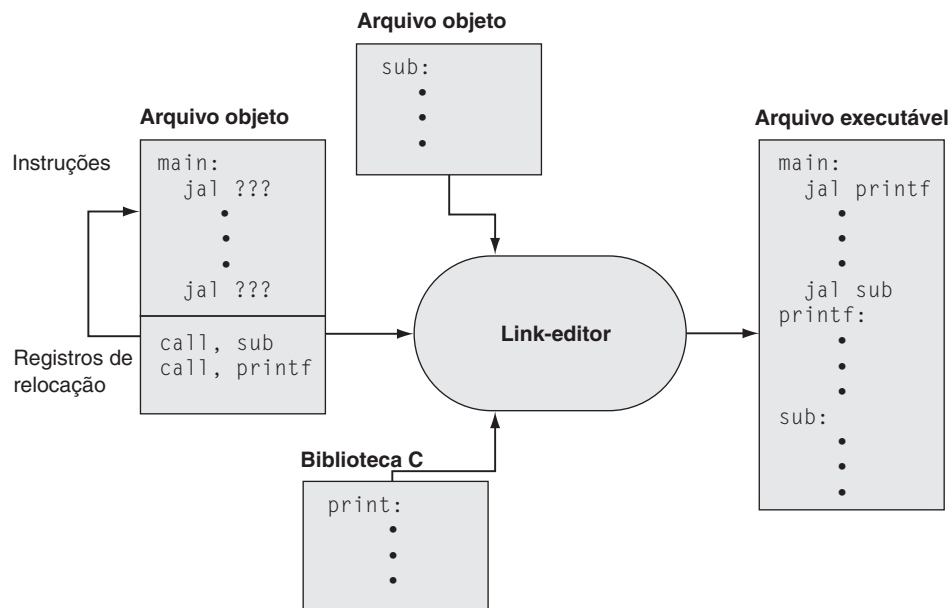
A **compilação separada** permite que um programa seja dividido em partes que são armazenadas em arquivos diferentes. Cada arquivo contém uma coleção logicamente relacionada de sub-rotinas e estruturas de dados que formam um *módulo* de um programa maior. Um arquivo pode ser compilado e montado independente de outros arquivos, de modo que as mudanças em um módulo não exigem a recompilação do programa inteiro. Conforme já discutimos, a compilação separada necessita da etapa adicional de link-edição para combinar os arquivos-objeto de módulos separados e consertar suas referências não resolvidas.

A ferramenta que mescla esses arquivos é o *link-editor* (ver Figura A.3.1). Ele realiza três tarefas:

- Pesquisa as bibliotecas de programa para encontrar rotinas de biblioteca usadas pelo programa
- Determina os locais da memória que o código de cada módulo ocupará e reloca suas instruções ajustando referências absolutas
- Resolve referências entre os arquivos

A primeira tarefa de um link-editor é garantir que um programa não contenha rótulos indefinidos. O link-editor combina os símbolos externos e as referências não resolvidas a partir dos arquivos de um programa. Um símbolo externo em um arquivo resolve uma referência de outro arquivo se ambos se referirem a um rótulo com o mesmo nome. As referências não combinadas significam que um símbolo foi usado, mas não definido em qualquer lugar do programa.

Referências não resolvidas nesse estágio do processo de link-edição não necessariamente significam que um programador cometeu um erro. O programa poderia ter referenciado uma rotina de biblioteca cujo código não estava nos arquivos-objeto passados ao link-editor. Depois de combinar os símbolos no programa, o link-editor pesquisa as bibliotecas de programa do sistema para encontrar sub-rotinas e estruturas de dados predefinidas que o programa referencia. As bibliotecas básicas con-



**FIGURA A.3.1** O link-editor pesquisa uma coleção de arquivos-objeto e bibliotecas de programa para encontrar rotinas usadas em um programa, combina-as em um único arquivo executável e resolve as referências entre as rotinas em arquivos diferentes.

têm rotinas que lêem e escrevem dados, alocam e liberam memória, e realizam operações numéricas. Outras bibliotecas contêm rotinas para acessar bancos de dados ou manipular janelas de terminal. Um programa que referencia um símbolo não resolvido que não está em qualquer biblioteca é errôneo e não pode ser link-editado. Quando o programa usa uma rotina de biblioteca, o link-editor extrai o código da rotina da biblioteca e o incorpora ao segmento de texto do programa. Essa nova rotina, por sua vez, pode depender de outras rotinas de biblioteca, de modo que o link-editor continua a buscar outras rotinas de biblioteca até que nenhuma referência externa esteja não resolvida ou até que uma rotina não possa ser encontrada.

Se todas as referências externas forem resolvidas, o link-editor em seguida determina os locais da memória que cada módulo ocupará. Como os arquivos foram montados isoladamente, o montador não poderia saber onde as instruções ou os dados de um módulo seriam colocados em relação a outros módulos. Quando o link-editor coloca um módulo na memória, todas as referências absolutas precisam ser *relocadas* para refletir seu verdadeiro local. Como o link-editor possui informações de relocação que identificam todas as referências relocáveis, ele pode eficientemente localizar e remendar essas referências.

O link-editor produz um arquivo executável que pode ser executado em um computador. Normalmente, esse arquivo tem o mesmo formato de um arquivo objeto, exceto que não contém referências não resolvidas ou informações de relocação.

## A.4

### Carga

Um programa que link-edita sem um erro pode ser executado. Antes de ser executado, o programa reside em um arquivo no armazenamento secundário, como um disco. Em sistemas UNIX, o kernel do sistema operacional traz o programa para a memória e inicia sua execução. Para iniciar um programa, o sistema operacional realiza as seguintes etapas:

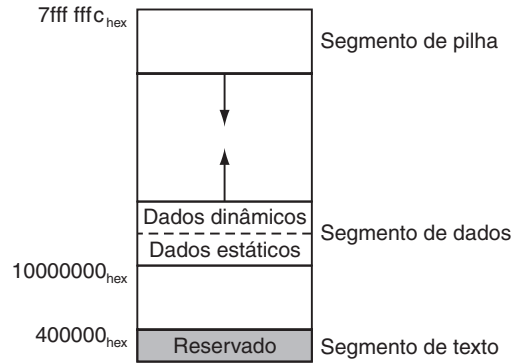
1. Lê o cabeçalho do arquivo executável para determinar o tamanho dos segmentos de texto e de dados.
2. Cria um novo espaço de endereçamento para o programa. Esse espaço de endereçamento é grande o suficiente para manter os segmentos de texto e de dados, junto com um segmento de pilha (ver a Seção A.5).
3. Copia instruções e dados do arquivo executável para o novo espaço de endereçamento.
4. Copia argumentos passados ao programa para a pilha.
5. Inicializa os registradores da máquina. Em geral, a maioria dos registradores é apagada, mas o stack pointer precisa receber o endereço do primeiro local da pilha livre (ver a Seção A.5).
6. Desvia para a rotina de partida, que copia os argumentos do programa da pilha para os registradores e chama a rotina `main` do programa. Se a rotina `main` retornar, a rotina de partida termina o programa com a chamada do sistema `exit`.

## A.5

### Uso da memória

As próximas seções elaboram a descrição da arquitetura MIPS apresentada anteriormente no livro. Os capítulos anteriores focalizaram principalmente o hardware e seu relacionamento com o software de baixo nível. Essas seções tratavam principalmente de como os programadores assembly utilizam o hardware do MIPS. Essas seções descrevem um conjunto de convenções seguido em muitos siste-





**FIGURA A.5.1** Layout da memória.

mas MIPS. Em sua maior parte, o hardware não impõe essas convenções. Em vez disso, elas representam um acordo entre os programadores para seguirem o mesmo conjunto de regras, de modo que o software escrito por diferentes pessoas possa atuar junto e fazer uso eficaz do hardware MIPS.

Os sistemas baseados em processadores MIPS normalmente dividem a memória em três partes (ver Figura A.5.1). A primeira parte, próxima do início do espaço de endereçamento (começando no endereço  $400000_{hex}$ ), é o *segmento de texto*, que mantém as instruções do programa.

**dados estáticos** A parte da memória que contém dados cujo tamanho é conhecido pelo compilador e cujo tempo de vida é a execução inteira do programa.

A segunda parte, acima do segmento de texto, é o *segmento de dados*, dividido ainda mais em duas partes. Os **dados estáticos** (começando no endereço  $10000000_{hex}$ ) contêm objetos cujo tamanho é conhecido pelo compilador e cujo tempo de vida – o intervalo durante o qual um programa pode acessá-los – é a execução inteira do programa. Por exemplo, em C, as variáveis globais são alocadas estaticamente, pois podem ser referenciadas a qualquer momento durante a execução de um programa. O link-editor atribui objetos estáticos a locais no segmento de dados e resolve referências a esses objetos.

## Interface hardware/software

Como o segmento de dados começa muito acima do programa, no endereço  $10000000_{hex}$ , as instruções `load` e `store` não podem referenciar diretamente os objetos de dados com seus campos de offset de 16 bits (ver Seção 2.4, no Capítulo 2). Por exemplo, para carregar a `word` no segmento de dados no endereço  $10010020_{hex}$  para o registrador `$v0`, são necessárias duas instruções:

```
lui $s0, 0x1001 # 0x1001 significa 1001 base 16
lw $v0, 0x0020($s0) # 0x10010000 + 0x0020 = 0x10010020
```

(O `0x` antes de um número significa que ele é um valor hexadecimal. Por exemplo,  $0x8000$  é  $8000_{hex}$  ou  $32.768_{dec}$ .)

Para evitar repetir a instrução `lui` em cada `load` e `store`, os sistemas MIPS normalmente dedicam um registrador (`$gp`) como um *ponteiro global* para o segmento de dados estático. Esse registrador contém o endereço  $10008000_{hex}$ , de modo que as instruções `load` e `store` podem usar seus campos de 16 bits com sinal para acessar os primeiros 64KB do segmento de dados estático. Com esse ponteiro global, podemos reescrever o exemplo como uma única instrução:

```
lw $v0, 0x8020($gp)
```

Naturalmente, um ponteiro global torna os locais de endereçamento entre  $10000000_{hex}$ – $10010000_{hex}$  mais rápidos do que outros locais do heap. O compilador MIPS normalmente armazena *variáveis globais* nessa área, pois essas variáveis possuem locais fixos e se ajustam melhor do que outros dados globais, como arrays.



Imediatamente acima dos dados estáticos estão os *dados dinâmicos*. Esses dados, como seu nome sugere, são alocados pelo programa enquanto ele é executado. Nos programas C, a rotina de biblioteca `malloc` localiza e retorna um novo bloco de memória. Como um compilador não pode prever quanta memória um programa alocará, o sistema operacional expande a área de dados dinâmica para atender à demanda. Conforme indica a seta para cima na figura, `malloc` expande a área dinâmica com a chamada do sistema `sbrk`, que faz com que o sistema operacional acrescente mais páginas ao espaço de endereçamento virtual do programa (ver Seção 7.4, no Capítulo 7) imediatamente acima do segmento de dados dinâmico.

A terceira parte, o **segmento de pilha** do programa, reside no topo do espaço de endereçamento virtual (começando no endereço `7fffffffhexa`). Assim como os dados dinâmicos, o tamanho máximo da pilha de um programa não é conhecido antecipadamente. À medida que o programa coloca valores na pilha, o sistema operacional expande o segmento de pilha para baixo, em direção ao segmento de dados.

Essa divisão de três partes da memória não é a única possível. Contudo, ela possui duas características importantes: os dois segmentos dinamicamente expansíveis são bastante distantes um do outro, e eles podem crescer para usar o espaço de endereços inteiro de um programa.

**segmento de pilha** A parte da memória usada por um programa para manter frames de chamada de procedimento.

## A.6

## Convenção para chamadas de procedimento

As convenções que governam o uso dos registradores são necessárias quando os procedimentos em um programa são compilados separadamente. Para compilar um procedimento em particular, um compilador precisa saber quais registradores pode usar e quais são reservados para outros procedimentos. As regras para usar os registradores são chamadas de **convenções para uso dos registradores** ou **convenções para chamadas de procedimento**. Como o nome sugere, essas regras são, em sua maior parte, convenções seguidas pelo software, em vez de regras impostas pelo hardware. No entanto, a maioria dos compiladores e programadores tenta seguir essas convenções estritamente, pois sua violação causa bugs traiçoeiros.

A convenção para chamadas descrita nesta seção é aquela utilizada pelo compilador `gcc`. O compilador nativo do MIPS utiliza uma convenção mais complexa, que é ligeiramente mais rápida.

A CPU do MIPS contém 32 registradores de uso geral, numerados de 0 a 31. O registrador \$0 contém o valor fixo 0.

**convenção para uso dos registradores** Também chamada **convenção para chamadas de procedimento**. Um protocolo de software que governa o uso dos registradores por procedimentos.

- Os registradores \$at (1), \$k0 (26) e \$k1 (27) são reservados para o montador e o sistema operacional e não devem ser usados por programas do usuário ou compiladores.
- Os registradores \$a0-\$a3 (4-7) são usados para passar os quatro primeiros argumentos às rotinas (os argumentos restantes são passados na pilha). Os registradores \$v0 e \$v1 (2, 3) são usados para retornar valores das funções.
- Os registradores \$t0-\$t9 (8-15, 24, 25) **registradores salvos pelo caller**, que são usados para manter quantidades temporárias que não precisam ser preservadas entre as chamadas (ver Seção 2.7, no Capítulo 2).
- Os registradores \$s0-\$s7 (16-23) são **registradores salvos pelo callee**, que mantêm valores de longa duração, que devem ser preservados entre as chamadas.
- O registrador \$gp (28) é um ponteiro global que aponta para o meio de um bloco de 64K de memória no segmento de dados estático.
- O registrador \$sp (29) é o stack pointer, que aponta para o último local na pilha. O registrador \$fp (30) é o frame pointer. A instrução `jal` escreve no registrador \$ra (31), o endereço de retorno de uma chamada de procedimento. Esses dois registradores são explicados na próxima seção.

**registrador salvo pelo caller** Um registrador salvo pela rotina que faz uma chamada de procedimento.

**registrador salvo pelo callee** Um registrador salvo pela rotina sendo chamada.

**frame de chamada de procedimento** Um bloco de memória usado para manter valores passados a um procedimento como argumentos, para salvar registradores que um procedimento pode modificar mas que o caller não deseja que sejam alterados, e para fornecer espaço para variáveis locais a um procedimento.

As abreviações e nomes de duas letras para esses registradores – por exemplo, \$sp para o stack pointer – refletem os usos intencionados na convenção de chamada de procedimento. Ao descrever essa convenção, usaremos os nomes em vez de números de registrador. A Figura A.6.1 lista os registradores e descreve seus usos intencionados.

### Chamadas de procedimento

Esta seção descreve as etapas que ocorrem quando um procedimento (*caller*) invoca outro procedimento (*callee*). Os programadores que escrevem em uma linguagem de alto nível (como C ou Pascal) nunca vêem os detalhes de como um procedimento chama outro, pois o compilador cuida dessa manutenção de baixo nível. Contudo, os programadores assembly precisam implementar explicitamente cada chamada e retorno de procedimento.

A maior parte da manutenção associada a uma chamada gira em torno de um bloco de memória chamado **frame de chamada de procedimento**. Essa memória é usada para diversas finalidades:

Nome do registrador	Número	Uso
\$zero	0	constante 0
\$at	1	reservado para o montador
\$v0	2	avaliação de expressão e resultados de uma função
\$v1	3	avaliação de expressão e resultados de uma função
\$a0	4	argumento 1
\$a1	5	argumento 2
\$a2	6	argumento 3
\$a3	7	argumento 4
\$t0	8	temporário (não preservado pela chamada)
\$t1	9	temporário (não preservado pela chamada)
\$t2	10	temporário (não preservado pela chamada)
\$t3	11	temporário (não preservado pela chamada)
\$t4	12	temporário (não preservado pela chamada)
\$t5	13	temporário (não preservado pela chamada)
\$t6	14	temporário (não preservado pela chamada)
\$t7	15	temporário (não preservado pela chamada)
\$s0	16	temporário salvo (preservado pela chamada)
\$s1	17	temporário salvo (preservado pela chamada)
\$s2	18	temporário salvo (preservado pela chamada)
\$s3	19	temporário salvo (preservado pela chamada)
\$s4	20	temporário salvo (preservado pela chamada)
\$s5	21	temporário salvo (preservado pela chamada)
\$s6	22	temporário salvo (preservado pela chamada)
\$s7	23	temporário salvo (preservado pela chamada)
\$t8	24	temporário (não preservado pela chamada)
\$t9	25	temporário (não preservado pela chamada)
\$k0	26	reservado para o kernel do sistema operacional
\$k1	27	reservado para o kernel do sistema operacional
\$gp	28	ponteiro para área global
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	endereço de retorno (usado por chamada de função)

**FIGURA A.6.1 Registradores do MIPS e convenção de uso.**

- Para manter valores passados a um procedimento como argumentos
- Para salvar registradores que um procedimento pode modificar, mas que o caller não deseja que sejam alterados
- Para oferecer espaço para variáveis locais a um procedimento

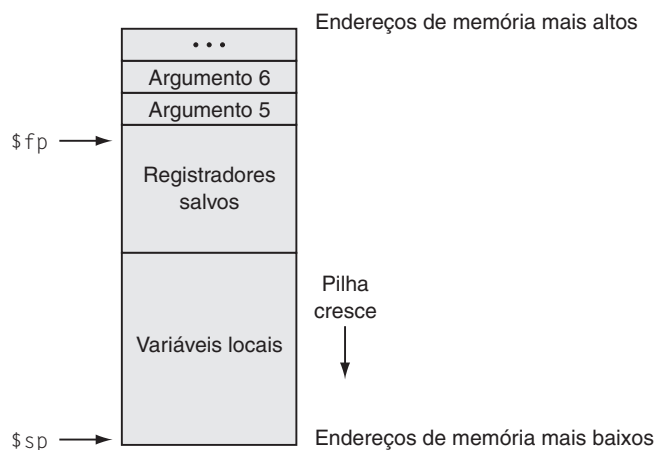
Na maioria das linguagens de programação, as chamadas e retornos de procedimento seguem uma ordem estrita do tipo último a entrar, primeiro a sair (LIFO – Last-In, First-Out), de modo que essa memória pode ser alocada e liberada como uma pilha, motivo pelo qual esses blocos de memória às vezes são chamados frames de pilha.

A Figura A.6.2 mostra um frame de pilha típico. O frame consiste na memória entre o frame pointer (\$fp), que aponta para a primeira word do frame, e o stack pointer (\$sp), que aponta para a última word do frame. A pilha cresce para baixo a partir dos endereços de memória mais altos, de modo que o frame pointer aponta para acima do stack pointer. O procedimento que está executando utiliza o frame pointer para acessar rapidamente os valores em seu frame de pilha. Por exemplo, um argumento no frame de pilha pode ser lido para o registrador \$v0 com a instrução

```
lw $v0, 0($fp)
```

Um frame de pilha pode ser construído de muitas maneiras diferentes; porém, o caller e o callee precisam combinar a seqüência de etapas. As etapas a seguir descrevem a convenção de chamada utilizada na maioria das máquinas MIPS. Essa convenção entra em ação em três pontos durante uma chamada de procedimento: imediatamente antes do caller invocar o callee, assim que o callee começa a executar e imediatamente antes do callee retornar ao caller. Na primeira parte, o caller coloca os argumentos da chamada de procedimento em locais padrões e invoca o callee para fazer o seguinte:

1. Passar argumentos. Por convenção, os quatro primeiros argumentos são passados nos registradores \$a0-\$a3. Quaisquer argumentos restantes são colocados na pilha e aparecem no início do frame de pilha do procedimento chamado.
2. Salvar registradores salvos pelo caller. O procedimento chamado pode usar esses registradores (\$a0-\$a3 e \$t0-\$t9) sem primeiro salvar seu valor. Se o caller espera utilizar um desses registradores após uma chamada, ele deverá salvar seu valor antes da chamada.
3. Executar uma instrução jal (ver Seção 2.7 do Capítulo 2), que desvia para a primeira instrução do callee e salva o endereço de retorno no registrador \$ra.



**FIGURA A.6.2 Layout de um frame de pilha.** O frame pointer (\$fp) aponta para a primeira word do frame de pilha do procedimento em execução. O stack pointer (\$sp) aponta para a última word do frame. Os quatro primeiros argumentos são passados em registradores, de modo que o quinto argumento é o primeiro armazenado na pilha.

Antes que uma rotina chamada comece a executar, ela precisa realizar as seguintes etapas para configurar seu frame de pilha:

1. Alocar memória para o frame, subtraindo o tamanho do frame do stack pointer.
2. Salvar os registradores salvos pelo callee no frame. Um callee precisa salvar os valores desses registradores (\$s0-\$s7, \$fp e \$ra) antes de alterá-los, pois o caller espera encontrar esses registradores inalterados após a chamada. O registrador \$fp é salvo para cada procedimento que aloca um novo frame de pilha. No entanto, o registrador \$ra só precisa ser salvo se o callee fizer uma chamada. Os outros registradores salvos pelo callee, que são utilizados, também precisam ser salvos.
3. Estabelecer o frame pointer somando o tamanho do frame de pilha menos 4 a \$sp e armazenando a soma no registrador \$fp.

---

## Interface hardware/software

A convenção de uso dos registradores do MIPS oferece registradores salvos pelo caller e pelo callee, pois os dois tipos de registradores são vantajosos em circunstâncias diferentes. Os registradores salvos pelo caller são usados para manter valores de longa duração, como variáveis de um programa do usuário. Esses registradores só são salvos durante uma chamada de procedimento se o caller espera utilizar o registrador. Por outro lado, os registradores salvos pelo callee são usados para manter quantidades de curta duração, que não persistem entre as chamadas, como valores imediatos em um cálculo de endereço. Durante uma chamada, o caller não pode usar esses registradores para valores temporários de curta duração.

---

Finalmente, o callee retorna ao caller executando as seguintes etapas:

1. Se o callee for uma função que retorna um valor, coloque o valor retornado no registrador \$v0.
2. Restaure todos os registradores salvos pelo callee que foram salvos na entrada do procedimento.
3. Remova o frame de pilha somando o tamanho do frame a \$sp.
4. Retorne desviando para o endereço no registrador \$ra.

**Detalhamento:** uma linguagem de programação que não permite procedimentos recursivos – procedimentos que chamam a si mesmos direta ou indiretamente, por meio de uma cadeia de chamadas – não precisa alocar frames em uma pilha. Em uma linguagem não-recursiva, o frame de cada procedimento pode ser alocado estaticamente, pois somente uma invocação de um procedimento pode estar ativa ao mesmo tempo. As versões mais antigas de Fortran proibiam a recursão porque frames alocados estaticamente produziam código mais rápido em algumas máquinas mais antigas. Todavia, em arquiteturas load-store, como MIPS, os frames de pilha podem ser muito rápidos porque o registrador frame pointer aponta diretamente para o frame de pilha ativo, que permite que uma única instrução load ou store acesse valores no frame. Além disso, a recursão é uma técnica de programação valiosa.

### procedimentos recursivos

Procedimentos que chamam a si mesmos direta ou indiretamente através de uma cadeia de chamadas.

## Exemplo de chamada de procedimento

Como exemplo, considere a rotina em C

```
main ( )
{
    printf ("The factorial of 10 is %d\n", fact (10));
}
```

```

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

```

que calcula e imprime 10! (o fatorial de 10,  $10! = 10 \times 9 \times \dots \times 1$ ). `fact` é uma rotina recursiva que calcula  $n!$  multiplicando  $n$  vezes  $(n - 1)!$ . O código assembly para essa rotina ilustra como os programas manipulam frames de pilha.

Na entrada, a rotina `main` cria seu frame de pilha e salva os dois registradores salvos pelo callee que serão modificados: `$fp` e `$ra`. O frame é maior do que o exigido para esses dois registradores, pois a convenção de chamada exige que o tamanho mínimo de um frame de pilha seja 24 bytes. Esse frame mínimo pode manter quatro registradores de argumento (`$a0`-`$a3`) e o endereço de retorno `$ra`, preenchidos até um limite de double word (24 bytes). Como `main` também precisa salvar o `$fp`, seu frame de pilha precisa ter duas words a mais (lembre-se de que o stack pointer é mantido alinhado em um limite de double word).

```

    .text
    .globl main
main:
    subu    $sp,$sp,32    # Frame de pilha tem 32 bytes
    sw     $ra,20($sp)    # Salva endereço de retorno
    sw     $fp,16($sp)    # Salva frame pointer antigo
    addiu  $fp,$sp,28    # Prepara frame pointer

```

A rotina `main`, então, chama a rotina de fatorial e lhe passa o único argumento 10. Depois que `fact` retorna, `main` chama a rotina de biblioteca `printf` e lhe passa uma string de formato e o resultado retornado de `fact`:

```

    li     $a0,10        # Coloca argumento (10) em $a0
    jal   fact          # Chama função fatorial

    la    $a0,$LC       # Coloca string de formato em $a0
    move  $a1,$v0       # Move resultado de fact para $a1
    jal   printf        # Chama a função print

```

Finalmente, depois de imprimir o fatorial, `main` retorna. Entretanto, primeiro, ela precisa restaurar os registradores que salvou e remover seu frame de pilha:

```

    lw    $ra,20($sp)    # Restaura endereço de retorno
    lw    $fp,16($sp)    # Restaura frame pointer
    addiu $sp,$sp,32    # Remove frame de pilha
    jr    $ra           # Retorna a quem chamou

```

```

    .rdata
$LC:
    .ascii "The factorial of 10 is %d\n\000"

```

A rotina de fatorial é semelhante em estrutura a `main`. Primeiro, ela cria um frame de pilha e salva os registradores salvos pelo callee que serão usados por ela. Além de salvar `$ra` e `$fp`, `fact` também salva seu argumento (`$a0`), que ela usará para a chamada recursiva:

```

    .text
fact:
    subu    $sp,$sp,32    # Frame de pilha tem 32 bytes
    sw     $ra,20($sp)    # Salva endereço de retorno
    sw     $fp,16($sp)    # Salva frame pointer

```

```

addiu    $fp,$sp,28    # Prepara frame pointer
sw       $a0,0($fp)    # Salva argumento (n)

```

O coração da rotina `fact` realiza o cálculo do programa em C. Ele testa se o argumento é maior do que 0. Caso contrário, a rotina retorna o valor 1. Se o argumento for maior do que 0, a rotina é chamada recursivamente para calcular `S` e multiplica esse valor por `n`:

```

lw       $v0,0($fp)    # Carrega n
bgtz    $v0,$L2        # Desvia se n > 0
li       $v0,1         # Retorna 1
jr       $L1           # Desvia para o código de retorno

$L2:
lw       $v1,0($fp)    # Carrega n
subu    $v0,$v1,1     # Calcula n - 1
move    $a0,$v0       # Move valor para $a0
jal     fact          # Chama função de fatorial
lw       $v1,0($fp)    # Carrega n
mul     $v0,$v0,$v1   # Calcula fact(n-1) * n

```

Finalmente, a rotina de fatorial restaura os registradores salvos pelo callee e retorna o valor no registrador `$v0`:

```

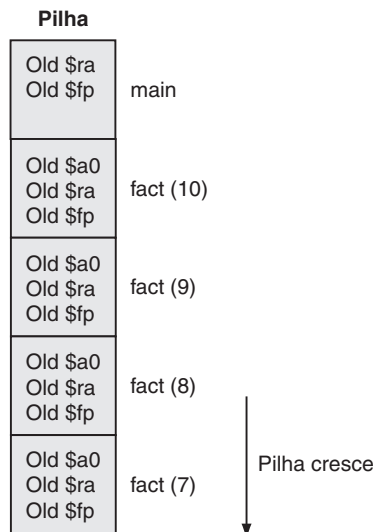
$L1:
lw       $ra, 20($sp)  # Restaura $ra
lw       $fp, 16($sp) # Restaura $fp
addiu   $sp, $sp, 32  # Remove o frame de pilha
jr       $ra           # Retorna a quem chamou

```

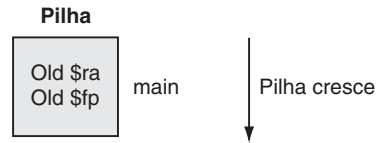
## PILHA EM PROCEDIMENTOS RECURSIVOS

### EXEMPLO

A Figura A.6.3 mostra a pilha na chamada `fact(7)`. `main` executa primeiro, de modo que seu frame está mais abaixo na pilha. `main` chama `fact(10)`, cujo frame de pilha vem em seguida na pilha. Cada invocação chama `fact` recursivamente para calcular o próximo fatorial mais inferior. Os frames de pilha fazem um paralelo com a ordem LIFO dessas chamadas. Qual é a aparência da pilha quando a chamada a `fact(10)` retorna?



**FIGURA A.6.3** Frames da pilha durante a chamada de `fact(7)`.


**RESPOSTA**

**Detalhamento:** a diferença entre o compilador MIPS e o compilador gcc é que o compilador MIPS normalmente não usa um frame pointer, de modo que esse registrador está disponível como outro registrador salvo pelo callee, \$s8. Essa mudança salva algumas das instruções na chamada de procedimento e seqüência de retorno. Contudo, isso complica a geração do código, porque um procedimento precisa acessar seu frame de pilha com \$sp, cujo valor pode mudar durante a execução de um procedimento se os valores forem colocados na pilha.

### Outro exemplo de chamada de procedimento

Como outro exemplo, considere a seguinte rotina que calcula a função tak, que é um benchmark bastante utilizado, criado por Ikuo Takeuchi. Essa função não calcula nada de útil, mas é um programa altamente recursivo que ilustra a convenção de chamada do MIPS.

```
int tak (int x, int y, int z)
{
    if (y < x)
        return 1 + tak (tak (x - 1, y, z),
            tak (y - 1, z, x),
            tak (z - 1, x, y));
    else
        return z;
}

int main ( )
{
    tak(18, 12, 6);
}
```

O código assembly para esse programa está logo em seguida. A função tak primeiro salva seu endereço de retorno em seu frame de pilha e seus argumentos nos registradores salvos pelo callee, pois a rotina pode fazer chamadas que precisam usar os registradores \$a0-\$a2 e \$ra. A função utiliza registradores salvos pelo callee, pois eles mantêm valores que persistem por toda a vida da função, o que inclui várias chamadas que potencialmente poderiam modificar registradores.

```
.text
.globl tak

tak:
    subu    $sp, $sp, 40
    sw     $ra, 32($sp)

    sw     $s0, 16($sp) # x
    move  $s0, $a0
    sw     $s1, 20($sp) # y
    move  $s1, $a1
    sw     $s2, 24($sp) # z
    move  $s2, $a2
    sw     $s3, 28($sp) # temporário
```



A rotina, então, inicia a execução testando se  $y < x$ . Se não, ela desvia para o rótulo L1, que aparece a seguir.

```
bge    $s1, $s0, L1    # if (y < x)
```

Se  $y < x$ , então ela executa o corpo da rotina, que contém quatro chamadas recursivas. A primeira chamada usa quase os mesmos argumentos do seu pai:

```
addiu   $a0, $s0, -1
move    $a1, $s1
move    $a2, $s2
jal     tak          # tak (x - 1, y, z)
move    $s3, $v0
```

Observe que o resultado da primeira chamada recursiva é salvo no registrador \$s3, de modo que pode ser usado mais tarde.

A função agora prepara argumentos para a segunda chamada recursiva.

```
addiu   $a0, $s1, -1
move    $a1, $s2
move    $a2, $s0
jal     tak          # tak (y - 1, z, x)
```

Nas instruções a seguir, o resultado dessa chamada recursiva é salvo no registrador \$s0. No entanto, primeiro, precisamos ler, pela última vez, o valor salvo do primeiro argumento a partir desse registrador.

```
addiu   $a0, $s2, -1
move    $a1, $s0
move    $a2, $s1
move    $s0, $v0
jal     tak          # tak (z - 1, x, y)
```

Depois de três chamadas recursivas mais internas, estamos prontos para a chamada recursiva final. Depois da chamada, o resultado da função está em \$v0 e o controle desvia para o epílogo da função.

```
move    $a0, $s3
move    $a1, $s0
move    $a2, $v0
jal     tak          # tak (tak(...), tak(...), tak(...))
addiu   $v0, $v0, 1
j       L2
```

Esse código no rótulo L1 é a consequência da instrução *if-then-else*. Ele apenas move o valor do argumento z para o registrador de retorno e entra no epílogo da função.

```
L1:
move    $v0, $s2
```

O código a seguir é o epílogo da função, que restaura os registradores salvos e retorna o resultado da função a quem chamou.

```
L2:
lw      $ra, 32($sp)
lw      $s0, 16($sp)
lw      $s1, 20($sp)
lw      $s2, 24($sp)
lw      $s3, 28($sp)
```

```

addiu   $sp, $sp, 40
jr      $ra

```

A rotina `main` chama a função `tak` com seus argumentos iniciais, e depois pega o resultado calculado (7) e o imprime usando a chamada ao sistema do SPIM para imprimir inteiros:

```

.global  main
main:
subu    $sp, $sp, 24
sw      $ra, 16($sp)

li      $a0, 18
li      $a1, 12
li      $a2, 6
jal     tak                # tak(18, 12, 6)

move    $a0, $v0
li      $v0, 1            # syscall print_int
syscall

lw      $ra, 16($sp)
addiu   $sp, $sp, 24
jr      $ra

```

## A.7 Exceções e interrupções

A Seção 5.6 do Capítulo 5 descreve o mecanismo de exceção do MIPS, que responde a exceções causadas por erros durante a execução de uma instrução e a interrupções externas causadas por dispositivos de E/S. Esta seção descreve o **tratamento de interrupção** e exceção com mais detalhes.<sup>1</sup> Nos processadores MIPS, uma parte da CPU, chamada *co-processador 0*, registra as informações de que o software precisa para lidar com exceções e interrupções. O simulador SPIM do MIPS não implementa todos os registradores do co-processador 0, pois muitos não são úteis em um simulador ou fazem parte do sistema de memória, que o SPIM não modela. Contudo, o SPIM oferece os seguintes registradores do co-processador 0:

**handler de interrupção**  
Um trecho de código executado como resultado de uma exceção ou interrupção.

Nome do registrador	Número do registrador	Uso
BadVAddr	8	Endereço de memória em que ocorreu uma referência de memória problemática
Count	9	Temporizador
Compare	11	valor comparado com o temporizador que causa interrupção quando combinam
Status	12	máscara de interrupções e bits de habilitação
Cause	13	tipo de exceção e bits de interrupções pendentes
EPC	14	endereço da instrução que causou a exceção
Config	16	configuração da máquina

1. Esta seção discute as exceções na arquitetura MIPS32, que é o que o SPIM implementa na Versão 7.0 em diante. As versões anteriores do SPIM implementaram a arquitetura MIPS-I, que tratava exceções de forma ligeiramente diferente. A conversão de programas a partir dessas versões para execução no MIPS32 não deverá ser difícil, pois as mudanças são limitadas aos campos dos registradores `Status` e `Cause` e à substituição da instrução `rfe` pela instrução `eret`.

Esses sete registradores fazem parte do conjunto de registradores do co-processor 0. Eles são acessados pelas instruções `mfc0` and `mtc0`. Após uma exceção, o registrador EPC contém o endereço da instrução executada quando a exceção ocorreu. Se a exceção foi causada por uma interrupção externa, então a instrução não terá iniciado a execução. Todas as outras exceções são causadas pela execução da instrução no EPC, exceto quando a instrução problemática está no delay slot de um branch ou jump. Nesse caso, o EPC aponta para a instrução de branch ou jump e o bit BD é ligado no registrador Cause. Quando esse bit está ligado, o handler de exceção precisa procurar a instrução problemática em `EPC + 4`. No entanto, de qualquer forma, um handler de exceção retoma o programa corretamente, retornando à instrução no EPC.

Se a instrução que causou a exceção fez um acesso à memória, o registrador `BadVAddr` contém o endereço do local de memória referenciado.

O registrador Count é um timer que incrementa em uma taxa fixa (como padrão, a cada 10 milissegundos) enquanto o SPIM está executando. Quando o valor no registrador Count for igual ao valor no registrador Compare, ocorre uma interrupção de hardware com nível de prioridade 5.

A Figura A.7.1 mostra o subconjunto dos campos do registrador Status implementados pelo simulador SPIM do MIPS. O campo `interrupt mask` contém um bit para cada um dos seis níveis de interrupção de hardware e dois de software. Um bit de máscara 1 permite que as interrupções nesse nível interrompam o processador. Um bit de máscara 0 desativa as interrupções nesse nível. Quando uma interrupção chega, ela liga seu bit de interrupção pendente no registrador Cause, mesmo que o bit de máscara esteja desligado. Quando uma interrupção está pendente, ela interromperá o processador quando seu bit de máscara for habilitado mais tarde.

O bit de modo do usuário é 0 se o processador estiver funcionando no modo kernel e 1 se estiver funcionando no modo usuário. No SPIM, esse bit é fixado em 1, pois o processador SPIM não implementa o modo kernel. O bit de nível de exceção normalmente é 0, mas é colocado em 1 depois que ocorre uma exceção. Quando esse bit é 1, as interrupções são desativadas e o EPC não é atualizado se outra exceção ocorrer. Esse bit impede que um handler de exceção seja incomodado por uma interrupção ou exceção, mas deve ser reiniciado quando o handler termina. Se o bit `interrupt enable` for 1, as interrupções são permitidas. Se for 0, elas estão inibidas.

A Figura A.7.2 mostra o subconjunto dos campos do registrador Cause que o SPIM implementa. O bit de branch delay é 1 se a última exceção ocorreu em uma instrução executada no slot de retardo de um desvio. Os bits de interrupções pendentes tornam-se 1 quando uma interrupção é gerada em determinado nível de hardware ou software. O registrador de código de exceção descreve a causa de uma exceção por meio dos seguintes códigos:

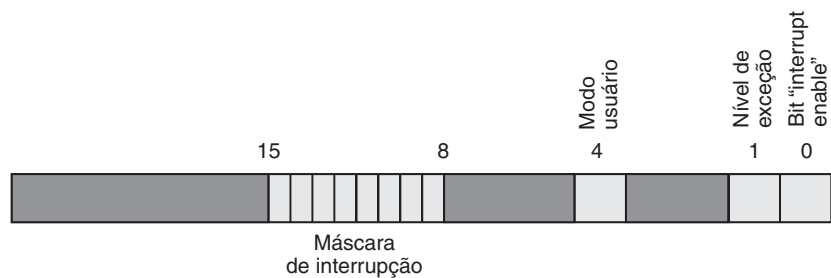


FIGURA A.7.1 O registrador Status.

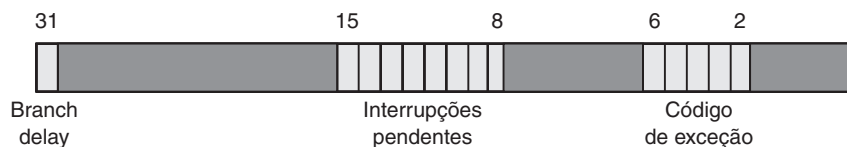


FIGURA A.7.2 O registrador Cause.



Número	Nome	Causa da exceção
00	Int	interrupção (hardware)
04	AdEL	exceção de erro de endereço (load ou busca de instrução)
05	AdES	exceção de erro de endereço (store)
06	IBE	erro de barramento na busca da instrução
07	DBE	erro de barramento no load ou store de dados
08	Sys	exceção de syscall
09	Bp	exceção de breakpoint
10	RI	exceção de instrução reservada
11	CpU	co-processador não implementado
12	Ov	exceção de overflow aritmético
13	Tr	trap
15	FPE	ponto flutuante

Exceções e interrupções fazem com que um processador MIPS desvie para uma parte do código, no endereço `80000180hexa` (no espaço de endereçamento do kernel, não do usuário), chamada *handler de exceção*. Esse código examina a causa da exceção e desvia para um ponto apropriado no sistema operacional. O sistema operacional responde a uma exceção terminando o processo que causou a exceção ou realizando alguma ação. Um processo que causa um erro, como a execução de uma instrução não implementada, é terminado pelo sistema operacional. Por outro lado, outras exceções, como faltas de página, são solicitações de um processo para o sistema operacional realizar um serviço, como trazer uma página do disco. O sistema operacional processa essas solicitações e retoma o processo. O último tipo de exceções são interrupções de dispositivos externos. Elas em geral fazem com que o sistema operacional mova dados de ou para um dispositivo de E/S e retome o processo interrompido.

O código no exemplo a seguir é um handler de exceção simples, que invoca uma rotina para imprimir uma mensagem a cada exceção (mas não interrupções). Esse código é semelhante ao handler de exceção (`exceptions.s`) usado pelo simulador SPIM.

## HANDLER DE EXCEÇÕES

O handler de exceção primeiro salva o registrador `$at`, que é usado em pseudo-instruções no código do handler, depois salva `$a0` e `$a1`, que mais tarde utiliza para passar argumentos. O handler de exceção não pode armazenar os valores antigos a partir desses registradores na pilha, como faria uma rotina comum, pois a causa da exceção poderia ter sido uma referência de memória que usou um valor incorreto (como 0) no stack pointer. Em vez disso, o handler de exceção armazena esses registradores em um registrador de handler de exceção (`$k1`, pois não pode acessar a memória sem usar `$at`) e dois locais da memória (`save0` e `save1`). Se a própria rotina de exceção pudesse ser interrompida, dois locais não seriam suficientes, pois a segunda exceção gravaria sobre valores salvos durante a primeira exceção. Entretanto, esse handler de exceção simples termina a execução antes de permitir interrupções, de modo que o problema não surge.

### EXEMPLO

```
.ktext 0x80000180
mov $k1, $at    # Salva o registrador $at
sw  $a0, save0 # Handler não é reentrante e não pode
sw  $a1, save1 # usar a pilha para salvar $a0, $a1
                # Não precisa salvar $k0/$k1
```

O handler de exceção, então, move os registradores `Cause` e `EPC` para os registradores da CPU. Os registradores `Cause` e `EPC` não fazem parte do conjunto de registradores da CPU. Em vez disso, eles são registradores no co-processador 0, que é a parte da CPU que trata das exceções. A instrução

mfc0 \$k0, \$13 move o registrador 13 do co-processador 0 (o registrador Cause) para o registrador da CPU \$k0. Observe que o handler de exceção não precisa salvar os registradores \$k0 e \$k1, pois os programas do usuário não deveriam usar esses registradores. O handler de exceção usa o valor do registrador Cause para testar se a exceção foi causada por uma interrupção (ver a tabela anterior). Se tiver sido, a exceção é ignorada. Se a exceção não foi uma interrupção, o handler chama `print_excp` para imprimir uma mensagem.

```
mfc0    $k0, $13    # Move Cause para $k0

sr1     $a0, $k0, 2 # Extrai o campo ExcCode
andi   $a0, $a0, 0xf

bgtz   $a0, done   # Desvia se ExcCode for Int(0)

mov     $a0, $k0   # Move Cause para $a0
mfc0   $a1, $14   # Move EPC para $a1
jal    print_excp # Imprime mensagem de erro de exceção
```

Antes de retornar, o handler de exceção apaga o registrador Cause; reinicia o registrador Status para ativar interrupções e limpar o bit EXL, para permitir que exceções subsequentes mudem o registrador EPC; e restaura os registradores \$a0, \$a1 e \$at. Depois, ele executa a instrução `eret` (retorno de exceção), que retorna à instrução apontada pelo EPC. Esse handler de exceção retorna à instrução após aquela que causou a exceção, a fim de não reexecutar a instrução que falhou e causar a mesma exceção novamente.

```
done: mfc0    $k0, $14    # Muda EPC
      addiu   $k0, $k0, 4 # Não reexecuta
                                # instrução que falhou
      mtc0    $k0, $14    # EPC

      mtc0    $0, $13    # Apaga registrador Cause

      mfc0    $k0, $12    # Repara registrador Status
      andi   $k0, 0xffffd # Apaga bit EXL
      ori    $k0, 0x1     # Ativa interrupções

      mtc0    $k0, $12

      lw     $a0, save0   # Restaura registradores
      lw     $a1, save1
      mov    $at, $k1

      eret                    # Retorna ao EPC

      .kdata
save0: .word 0
save1: .word 0
```

**Detalhamento:** em processadores MIPS reais, o retorno de um handler de exceção é mais complexo. O handler de exceção não pode sempre desviar para a instrução após o EPC. Por exemplo, se a instrução que causou a exceção estivesse em um delay slot de uma instrução de desvio (ver Capítulo 6), a próxima instrução a executar pode não ser a instrução seguinte na memória.

# A.8

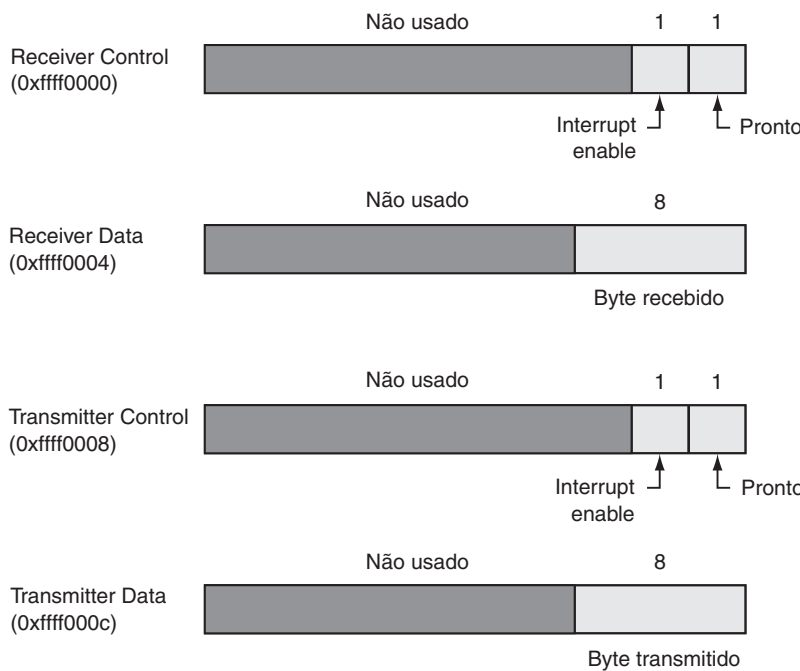
## Entrada e saída

O SPIM simula um dispositivo de E/S: um console mapeado em memória em que um programa pode ler e escrever caracteres. Quando um programa está executando, o SPIM conecta seu próprio terminal (ou uma janela de console separada na versão `xspim` do X-Windows ou na versão `PCSpim` do Windows) ao processador. Um programa MIPS executando no SPIM pode ler os caracteres que você digita. Além disso, se o programa MIPS escreve caracteres no terminal, eles aparecem no terminal do SPIM ou na janela de console. Uma exceção a essa regra é Control-C: esse caractere não é passado ao programa, mas, em vez disso, faz com que o SPIM pare e retorne ao modo de comando. Quando o programa para de executar (por exemplo, porque você digitou Control-C ou porque o programa atingiu um ponto de interrupção), o terminal é novamente conectado ao SPIM para que você possa digitar comandos do SPIM.

Para usar a E/S mapeada em memória (ver a seguir), o `spim` ou o `xspim` precisam ser iniciados com o flag `-mapped_io`. `PCSpim` pode ativar a E/S mapeada em memória por meio de um flag de linha de comando ou pela caixa de diálogo “Settings” (configurações).

O dispositivo de terminal consiste em duas unidades independentes: um *receptor* e um *transmissor*. O receptor lê caracteres digitados no teclado. O transmissor exibe caracteres no vídeo. As duas unidades são completamente independentes. Isso significa, por exemplo, que os caracteres digitados no teclado não são ecoados automaticamente no monitor. Em vez disso, um programa ecoa um caractere lendo-o do receptor e escrevendo-o no transmissor.

Um programa controla o terminal com quatro registradores de dispositivo mapeados em memória, como mostra a Figura A.8.1. “Mapeado em memória” significa que cada registrador aparece como uma posição de memória especial. O *registrador Receiver Control* está na posição `ffff0000hexa`. Somente dois de seus bits são realmente usados. O bit 0 é chamado “pronto”: se for 1, isso significa que um caractere chegou do teclado, mas ainda não foi lido do registrador Receiver Data. O bit de



**FIGURA A.8.1 O terminal é controlado por quatro registradores de dispositivo, cada um deles parecendo com uma posição de memória no endereço indicado.** Somente alguns bits desses registradores são realmente utilizados. Os outros sempre são lidos como 0s e as escritas são ignoradas.

pronto é apenas de leitura: tentativas de sobrescrevê-lo são ignoradas. O bit de pronto muda de 0 a 1 quando um caractere é digitado no teclado, e ele muda de 1 para 0 quando o caractere é lido do registrador Receiver Data.

O bit 1 do registrador Receiver Control é o “interrupt enable” do teclado. Esse bit pode ser lido e escrito por um programa. O interrupt enable inicialmente é 0. Se ele for colocado em 1 por um programa, o terminal solicita uma interrupção no nível de hardware 1 sempre que um caractere é digitado e o bit de pronto se torna 1. Todavia, para que a interrupção afete o processador, as interrupções também precisam estar ativadas no registrador Status (ver Seção A.7). Todos os outros bits do registrador Receiver Control não são utilizados.

O segundo registrador de dispositivo de terminal é o *registrador Receiver Data* (no endereço `ffff0004hexa`). Os 8 bits menos significativos desse registrador contêm o último caractere digitado no teclado. Todos os outros bits contêm 0s. Esse registrador é apenas de leitura e muda apenas quando um novo caractere é digitado no teclado. A leitura do registrador Receiver Data reinicia o bit de pronto no registrador Receiver Control para 0. O valor nesse registrador é indefinido se o registrador Receiver Control for 0.

O terceiro registrador de dispositivo de terminal é o *registrador Transmitter Control* (no endereço `ffff0008hexa`). Somente os 2 bits menos significativos desse registrador são usados. Eles se comportam de modo semelhante aos bits correspondentes no registrador Receiver Control. O bit 0 é chamado de “pronto” e é apenas de leitura. Se esse bit for 1, o transmissor estará pronto para aceitar um novo caractere para saída. Se for 0, o transmissor ainda está ocupado escrevendo o caractere anterior. O bit 1 é “interrupt enable” e pode ser lido e escrito. Se esse bit for colocado em 1, então o terminal solicita uma interrupção no nível de hardware 0 sempre que o transmissor estiver pronto para um novo caractere e o bit de pronto se torna 1.

O registrador de dispositivo final é o *registrador Transmitter Data* (no endereço `ffff000chexa`). Quando um valor é escrito nesse local, seus 8 bits menos significativos (ou seja, um caractere ASCII como na Figura 2.21, no Capítulo 2) são enviados para o console. Quando o registrador Transmitter Data é escrito, o bit de pronto no registrador Transmitter Control é retornado para 0. Esse bit permanece sendo 0 até passar tempo suficiente para transmitir o caractere para o terminal; depois, o bit de pronto se torna 1 novamente. O registrador Transmitter Data só deverá ser escrito quando o bit de pronto do registrador Transmitter Control for 1. Se o transmissor não estiver pronto, as escritas no registrador Transmitter Data são ignoradas (as escritas parecem ter sucesso, mas o caractere não é enviado).

Computadores reais exigem tempo para enviar caracteres para um console ou terminal. Esses retardos de tempo são simulados pelo SPIM. Por exemplo, depois que o transmissor começa a escrever um caractere, o bit de pronto do transmissor torna-se 0 por um tempo. O SPIM mede o tempo em instruções executadas, e não em tempo de clock real. Isso significa que o transmissor não fica pronto novamente até que o processador execute um número fixo de instruções. Se você interromper a máquina e examinar o bit de pronto, ele não mudará. Contudo, se você deixar a máquina executar, o bit por fim mudará de volta para 1.

## A.9

## SPIM

SPIM é um simulador de software que executa programas em assembly escritos para processadores que implementam a arquitetura MIPS32, especificamente o Release 1 dessa arquitetura com um mapeamento de memória fixo, sem caches e apenas os co-processadores 0 e 1.<sup>2</sup> O nome do SPIM é sim-

2. As primeiras versões do SPIM (antes da 7.0) implementaram a arquitetura MIPS-I utilizada nos processadores MIPS R2000 originais. Essa arquitetura é quase um subconjunto apropriado da arquitetura MIPS32, sendo que a diferença é a maneira como as exceções são tratadas. O MIPS32 também introduziu aproximadamente 60 novas instruções, que são aceitas pelo SPIM. Os programas executados nas versões anteriores do SPIM e que não usavam exceções deverão ser executados sem modificação nas versões mais recentes do SPIM. Os programas que usavam exceções exigirão pequenas mudanças.



plesmente MIPS ao contrário. O SPIM pode ler e executar os arquivos em assembly. O SPIM é um sistema autocontido para executar programas do MIPS. Ele contém um depurador e oferece alguns serviços de forma semelhante ao sistema operacional. SPIM é muito mais lento do que um computador real (100 ou mais vezes). Entretanto, seu baixo custo e grande disponibilidade não têm comparação com o hardware real!

Uma pergunta óbvia é: por que usar um simulador quando a maioria das pessoas possui PCs que contém processadores executando muito mais rápido do que o SPIM? Um motivo é que o processador nos PCs são 80x86s da Intel, cuja arquitetura é muito menos regular e muito mais complexa de entender e programar do que os processadores MIPS. A arquitetura do MIPS pode ser a síntese de uma máquina RISC simples e limpa.

Além disso, os simuladores podem oferecer um ambiente melhor para a programação em assembly do que uma máquina real, pois podem detectar mais erros e oferecer uma interface melhor do que um computador real.

Finalmente, os simuladores são uma ferramenta útil no estudo de computadores e dos programas neles executados. Como eles são implementados em software, não em silício, os simuladores podem ser examinados e facilmente modificados para acrescentar novas instruções, criar novos sistemas, como os multiprocessadores, ou apenas coletar dados.

## Simulação de uma máquina virtual

Uma arquitetura MIPS básica é difícil de programar diretamente, por causa dos *delayed branches*, *delayed loads* e modos de endereçamento restritos. Essa dificuldade é tolerável, pois esses computadores foram projetados para serem programados em linguagens de alto nível e apresentam uma interface criada para compiladores, em vez de programadores assembly. Uma boa parte da complexidade da programação é resultante de instruções *delayed*. Um *delayed branch* exige dois ciclos para executar (veja as seções “Detalhamentos” nas páginas 287 e 318 do Capítulo 6). No segundo ciclo, a instrução imediatamente após o desvio é executada. Essa instrução pode realizar um trabalho útil que normalmente teria sido feito antes do desvio. Ela também pode ser um *nop* (nenhuma operação), que não faz nada. De modo semelhante, os *delayed loads* exigem 2 ciclos para trazer um valor da memória, de modo que a instrução imediatamente após um *load* não pode usar o valor (ver Seção 6.2 do Capítulo 6).

O MIPS sabiamente escolheu ocultar essa complexidade fazendo com que seu montador implemente uma **máquina virtual**. Esse computador virtual parece ter *branches* e *loads* não *delayed* e um conjunto de instruções mais rico do que o hardware real. O montador *reorganiza* instruções para preencher os *delay slots*. O computador virtual também oferece *pseudo-instruções*, que aparecem como instruções reais nos programas em assembly. O hardware, porém, não sabe nada a respeito de *pseudo-instruções*, de modo que o montador as traduz para seqüências equivalentes de instruções de máquina reais. Por exemplo, o hardware do MIPS só oferece instruções para desvio quando um registrador é igual ou diferente de 0. Outros desvios condicionais, como aquele que desvia quando um registrador é maior do que outro, são sintetizados comparando-se os dois registradores e desviando quando o resultado da comparação é verdadeiro (diferente de zero).

Como padrão, o SPIM simula a máquina virtual mais rica, pois essa é a máquina que a maioria dos programadores achará útil. Todavia, o SPIM também pode simular os *branches* e *loads delayed* no hardware real. A seguir, descrevemos a máquina virtual e só mencionamos rapidamente os recursos que não pertencem ao hardware real. Ao fazer isso, seguimos a convenção dos programadores (e compiladores) assembly do MIPS, que normalmente utilizam a máquina estendida como se estivesse implementada em silício.



**máquina virtual** Um computador virtual que parece ter *branches* e *loads* não *delayed* e um conjunto de instruções mais rico do que o hardware real.



## Introdução ao SPIM

O restante deste apêndice é uma introdução ao SPIM e ao Assembly R2000 do MIPS. Muitos detalhes nunca deverão preocupá-lo; porém, o grande volume de informações às vezes poderá obscure-

cer o fato de que o SPIM é um programa simples e fácil de usar. Esta seção começa com um tutorial rápido sobre o uso do SPIM, que deverá permitir que você carregue, depure e execute programas MIPS simples.

O SPIM vem em diferentes versões para diferentes tipos de sistemas. A única constante é a versão mais simples, chamada `spim`, que é um programa controlado por linha de comandos, executado em uma janela de console. Ele opera como a maioria dos programas desse tipo: você digita uma linha de texto, pressiona a tecla Enter (ou Return) e o `spim` executa seu comando. Apesar da falta de uma interface sofisticada, o `spim` pode fazer tudo que seus primos mais sofisticados fazem.

Existem dois primos sofisticados do `spim`. A versão que roda no ambiente X-Windows de um sistema UNIX ou Linux é chamada `xspim`. `xspim` é um programa mais fácil de aprender e usar do que o `spim`, pois seus comandos sempre são visíveis na tela e porque ele continuamente apresenta os registradores e a memória da máquina. A outra versão sofisticada se chama `PCspim` e roda no Microsoft Windows. As versões do SPIM para UNIX e Windows estão neste CD (clique em Tutoriais) . Os tutoriais (em inglês) sobre `xspim`, `pcSpim`, `spim` e as opções da linha de comandos do `spim`  estão neste CD (clique em Software).

Se você for executar o `spim` em um PC com o Microsoft Windows, deverá primeiro dar uma olhada no tutorial sobre `PCSpim`  neste CD. Se você executar o `spim` em um computador com UNIX ou Linux, deverá ler o tutorial (em inglês) sobre `xspim`  (clique em Tutoriais).

## Recursos surpreendentes

Embora o SPIM fielmente simule o computador MIPS, o SPIM é um simulador, e certas coisas não são idênticas a um computador real. As diferenças mais óbvias são que a temporização da instrução e o sistema de memória não são idênticos. O SPIM não simula caches ou a latência da memória, nem reflete com precisão os atrasos na operação de ponto flutuante ou nas instruções de multiplicação e divisão. Além disso, as instruções de ponto flutuante não detectam muitas condições de erro, o que deveria causar exceções em uma máquina real.

Outra surpresa (que também ocorre na máquina real) é que uma pseudo-instrução se expande para várias instruções de máquina. Quando você examina a memória passo a passo, as instruções que encontra são diferentes daquelas do programa fonte. A correspondência entre os dois conjuntos de instruções é muito simples, pois o SPIM não reorganiza as instruções para preencher delay slots.

## Ordem de bytes

Os processadores podem numerar os bytes dentro de uma word de modo que o byte com o número mais baixo seja o mais à esquerda ou o mais à direita. A convenção usada por uma máquina é considerada sua *ordem de bytes*. Os processadores MIPS podem operar com a ordem de bytes *big-endian* ou *little-endian*. Por exemplo, em uma máquina *big-endian*, a diretiva `.byte 0, 1, 2, 3` resultaria em uma word de memória contendo

Byte #			
0	1	2	3

enquanto, em uma máquina *little-endian*, a word seria

Byte #			
3	2	1	0

O SPIM opera com duas ordens de bytes. A ordem de bytes do SPIM é a mesma ordem de bytes da máquina utilizada para executar o simulador. Por exemplo, em um Intel 80x86, o SPIM é *little-endian*, enquanto em um Macintosh ou Sun SPARC, o SPIM é *big-endian*.

## Chamadas ao sistema

O SPIM oferece um pequeno conjunto de serviços semelhantes aos oferecidos pelo sistema operacional, por meio da instrução de chamada ao sistema (`syscall`). Para requisitar um serviço, um programa carrega o código da chamada ao sistema (ver Figura A.9.1) no registrador `$v0` e os argumentos nos registradores `$a0–$a3` (ou `$f12`, para valores de ponto flutuante). As chamadas ao sistema que retornam valores colocam seus resultados no registrador `$v0` (ou `$f0` para resultados de ponto flutuante). Por exemplo, o código a seguir imprime “the answer = 5”:

```
.data
str:
.asciiz "the answer = "
.text
li    $v0, 4      # código de chamada ao sistema para print_str
la    $a0, str    # endereço da string a imprimir
syscall          # imprime a string

li    $v0, 1      # código de chamada ao sistema para print_int
li    $a0, 5      # inteiro a imprimir
syscall          # imprime
```

Serviço	Código de chamada do sistema	Argumentos	Resultado
<code>print_int</code>	1	<code>\$a0 = integer</code>	
<code>print_float</code>	2	<code>\$f12 = float</code>	
<code>print_double</code>	3	<code>\$f12 = double</code>	
<code>print_string</code>	4	<code>\$a0 = string</code>	
<code>read_int</code>	5		integer (em <code>\$v0</code> )
<code>read_float</code>	6		float (em <code>\$f0</code> )
<code>read_double</code>	7		double (em <code>\$f0</code> )
<code>read_string</code>	8	<code>\$a0 = buffer, \$a1 = tamanho</code>	
<code>sbrk</code>	9	<code>\$a0 = valor</code>	endereço (em <code>\$v0</code> )
<code>exit</code>	10		
<code>print_char</code>	11	<code>\$a0 = char</code>	
<code>read_char</code>	12		char (em <code>\$a0</code> )
<code>open</code>	13	<code>\$a0 = nome de arquivo (string), \$a1 = flags, \$a2 = modo</code>	descritor de arquivo (em <code>\$a0</code> )
<code>read</code>	14	<code>\$a0 = descritor de arquivo, \$a1 = buffer, \$a2 = tamanho</code>	número de caracteres lidos (em <code>\$a0</code> )
<code>write</code>	15	<code>\$a0 = descritor de arquivo, \$a1 = buffer, \$a2 = tamanho</code>	número de caracteres escritos (em <code>\$a0</code> )
<code>close</code>	16	<code>\$a0 = descritor de arquivo</code>	
<code>exit2</code>	17	<code>\$a0 = resultado</code>	

**FIGURA A.9.1** Serviços do sistema.

A chamada ao sistema `print_int` recebe um inteiro e o imprime no console. `print_float` imprime um único número de ponto flutuante; `print_double` imprime um número de precisão dupla; e `print_string` recebe um ponteiro para uma string terminada em nulo, que ele escreve no console.

As chamadas ao sistema `read_int`, `read_float` e `read_double` lêem uma linha inteira da entrada, até o caractere de newline, inclusive. Os caracteres após o número são ignorados. `read_string` possui a mesma semântica da rotina de biblioteca do UNIX `fgets`. Ela lê até  $n - 1$  caracteres para um buffer e termina a string com um byte nulo. Se menos de  $n - 1$  caracteres estiverem na linha atual, `read_string` lê até o caractere de newline, inclusive, e novamente termina a string com nulo.

*Aviso:* os programas que usam essas syscalls para ler do terminal não deverão usar E/S mapeada em memória (ver Seção A.8).

sbrk retorna um ponteiro para um bloco de memória contendo  $n$  bytes adicionais. `exit` interrompe o programa que o SPIM estiver executando. `exit2` termina o programa SPIM, e o argumento de `exit2` torna-se o valor retornado quando o próprio simulador SPIM termina.

`print_char` e `read_char` escrevem e lêem um único caractere, respectivamente. `open`, `read`, `write` e `close` são as chamadas da biblioteca padrão do UNIX.

## A.10

### Assembly do MIPS R2000

Um processador MIPS consiste em uma unidade de processamento de inteiros (a CPU) e uma coleção de co-processadores que realizam tarefas auxiliares ou operam sobre outros tipos de dados, como números de ponto flutuante (ver Figura A.10.1). O SPIM simula dois co-processadores. O co-processador 0 trata de exceções e interrupções. O co-processador 1 é a unidade de ponto flutuante. O SPIM simula a maior parte dos aspectos dessa unidade.

#### Modos de endereçamento

O MIPS é uma arquitetura load-store, o que significa que somente instruções load e store acessam a memória. As instruções de cálculo operam apenas sobre os valores nos registradores. A máquina pura oferece apenas um modo de endereçamento de memória:  $c(rx)$ , que usa a soma do  $c$  imediato e do registrador  $rx$  como endereço. A máquina virtual oferece os seguintes modos de endereçamento para instruções load e store:

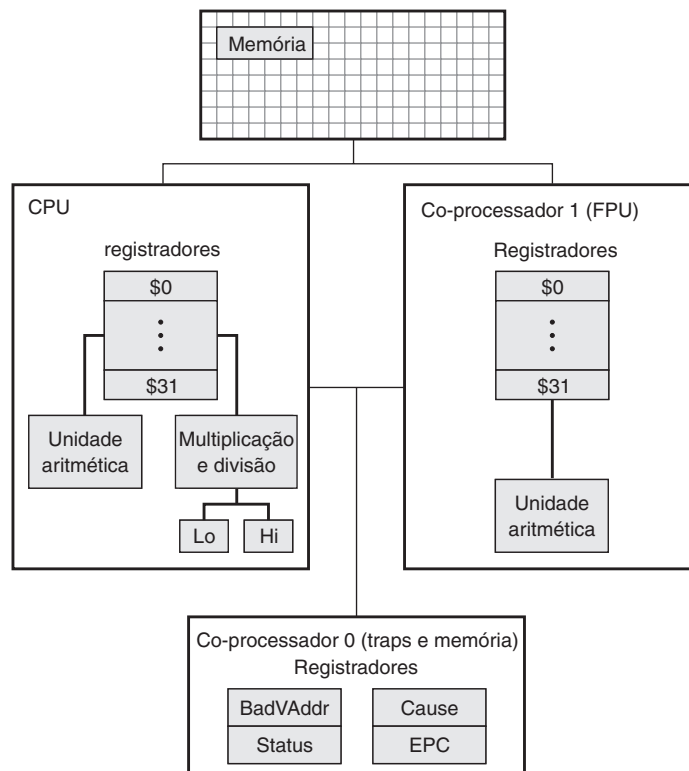


FIGURA A.10.1 CPU e FPU do MIPS R2000.

Formato	Cálculo de endereço
(registrador)	conteúdo do registrador
imm	imediato
imm (registrador)	imediato + conteúdo do registrador
rótulo	endereço do rótulo
rótulo ± imediato	endereço do rótulo + ou – imediato
rótulo ± imediato (registrador)	endereço do rótulo + ou – (imediato + conteúdo do registrador)

A maior parte das instruções `load` e `store` opera apenas sobre dados alinhados. Uma quantidade está *alinhada* se seu endereço de memória for um múltiplo do seu tamanho em bytes. Portanto, um objeto `halfword` precisa ser armazenado em endereços pares e um objeto `word` precisa ser armazenado em endereços que são múltiplos de quatro. No entanto, o MIPS oferece algumas instruções para manipular dados não alinhados (`lwl`, `lwr`, `swl` e `swr`).

**Detalhamento:** o montador MIPS (e SPIM) sintetiza os modos de endereçamento mais complexos, produzindo uma ou mais instruções antes que o `load` ou o `store` calculem um endereço complexo. Por exemplo, suponha que o rótulo `table` referenciasse o local de memória `0x10000004` e um programa tivesse a instrução

```
ld $a0, table + 4($a1)
```

O montador traduziria essa instrução para as instruções

```
lui $at, 4096
addu $at, $at, $a1
lw $a0, 8($at)
```

A primeira instrução carrega os bits mais significativos do endereço do rótulo no registrador `$at`, que é o registrador que o montador reserva para seu próprio uso. A segunda instrução acrescenta o conteúdo do registrador `$a1` ao endereço parcial do rótulo. Finalmente, a instrução `load` utiliza o modo de endereçamento de hardware para adicionar a soma dos bits menos significativos do endereço do rótulo e o offset da instrução original ao valor no registrador `$at`.

## Sintaxe do montador

Os comentários nos arquivos do montador começam com um sinal `#`. Tudo desde esse sinal até o fim da linha é ignorado.

Os identificadores são uma seqüência de caracteres alfanuméricos, símbolos `_` e pontos (`.`), que não começam com um número. Os opcodes de instrução são palavras reservadas que *não podem* ser usadas como identificadores. Rótulos são declarados por sua colocação no início de uma linha e seguidos por um sinal de dois-pontos, por exemplo:

```
.data
item: .word 1
.text
.globl main      # Precisa ser global
main: lw $t0, item
```

Os números estão na base 10 por padrão. Se eles forem precedidos por `0x`, serão interpretados como hexadecimais. Logo, `256` e `0x100` indicam o mesmo valor.

As strings são delimitadas com aspas (`"`). Caracteres especiais nas strings seguem a convenção da linguagem C:

- nova linha (`\n`)
- tabulação (`\t`)
- aspas (`\"`)

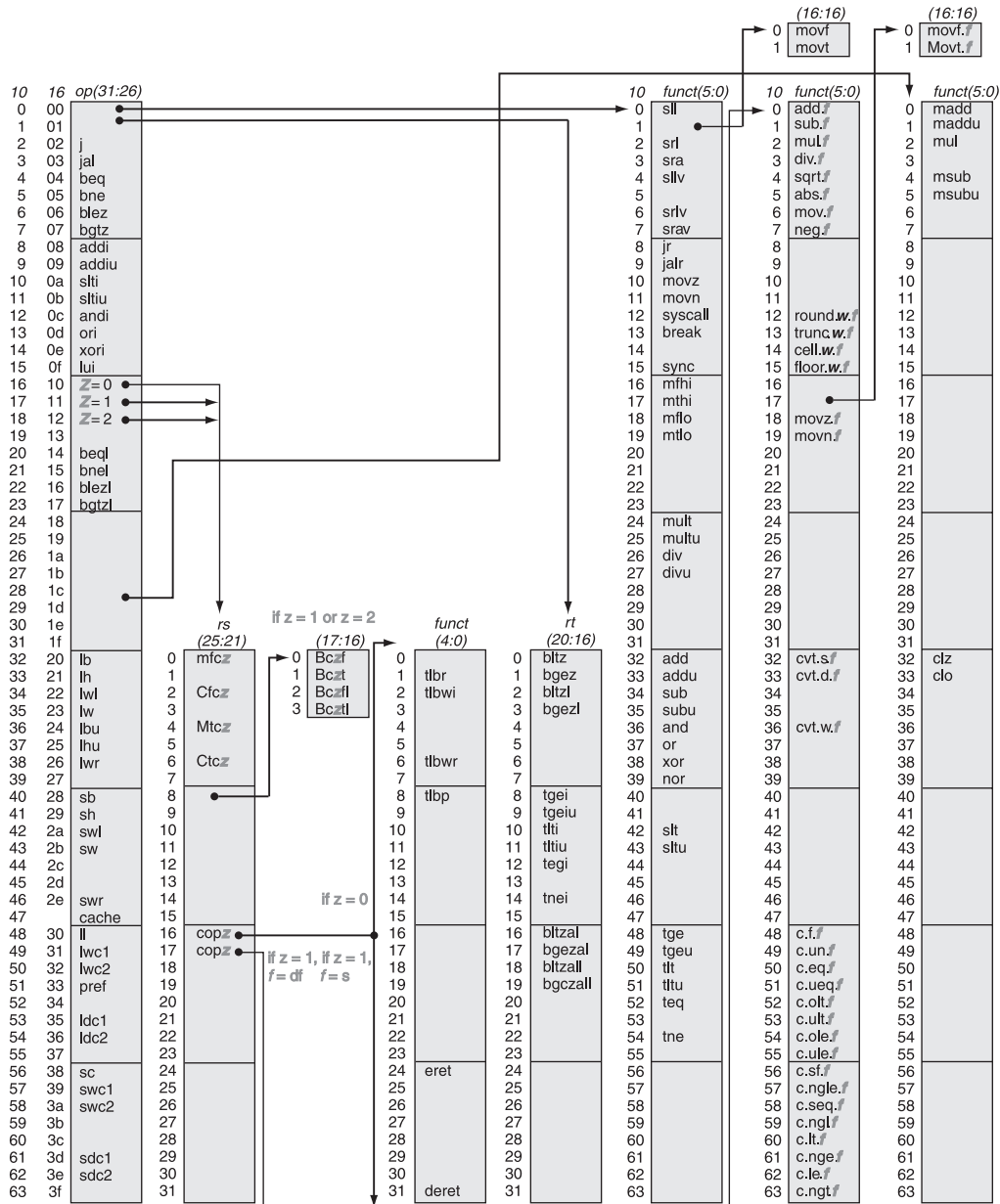
O SPIM admite um subconjunto das diretivas do montador do MIPS:

<code>.align n</code>	Alinha o próximo dado em um limite de $2^n$ bytes. Por exemplo, <code>.align 2</code> alinha o próximo valor em um limite de word. <code>.align 0</code> desativa o alinhamento automático das diretivas <code>.half</code> , <code>.word</code> , <code>.float</code> e <code>.double</code> até a próxima diretiva <code>.data</code> OU <code>.kdata</code> .
<code>.ascii str</code>	Armazena a string <i>str</i> na memória, mas não a termina com nulo.
<code>.asciiz str</code>	Armazena a string <i>str</i> na memória e a termina com nulo.
<code>.byte b1, ..., bn</code>	Armazena os <i>n</i> valores em bytes sucessivos da memória.
<code>.data &lt;end&gt;</code>	Itens subseqüentes são armazenados no segmento de dados. Se o argumento opcional <i>end</i> estiver presente, os itens subseqüentes são armazenados a partir do endereço <i>end</i> .
<code>.double d1, ..., dn</code>	Armazena os <i>n</i> números de precisão dupla em ponto flutuante em locais de memória sucessivos.
<code>.extern sym tamanho</code>	Declara que o dado armazenado em <i>sym</i> possui <i>tamanho</i> bytes de extensão e é um rótulo global. Essa diretiva permite que o montador armazene o dado em uma parte do segmento de dados que é acessado eficientemente por meio do registrador <code>\$gp</code> .
<code>.float f1, ..., fn</code>	Armazena os números de precisão simples em ponto flutuante nos locais de memória sucessivos.
<code>.globl sym</code>	Declara que o rótulo <i>sym</i> é global e pode ser referenciado a partir de outros arquivos.
<code>.half h1, ..., hn</code>	Armazena as <i>n</i> quantidades de 16 bits em halfwords sucessivas da memória.
<code>.kdata &lt;end&gt;</code>	Itens de dados subseqüentes são armazenados no segmento de dados do kernel. Se o argumento opcional <i>end</i> estiver presente, itens subseqüentes são armazenados a partir do endereço <i>end</i> .
<code>.ktext &lt;end&gt;</code>	Itens subseqüentes são colocados no segmento de texto do kernel. No SPIM, esses itens só podem ser instruções ou words (ver a diretiva <code>.word</code> , mais adiante). Se o argumento opcional <i>end</i> estiver presente, os itens subseqüentes são armazenados a partir do endereço <i>end</i> .
<code>.set noat</code> e <code>.set at</code>	A primeira diretiva impede que o SPIM reclame sobre instruções subseqüentes que utilizam o registrador <code>\$at</code> . A segunda diretiva reativa a advertência. Como as pseudo-instruções se expandem para o código que usa o registrador <code>\$at</code> , os programadores precisam ter muito cuidado ao deixar valores nesse registrador.
<code>.space n</code>	Aloca <i>n</i> bytes de espaço no segmento atual (que precisa ser o segmento de dados no SPIM).
<code>.text &lt;end&gt;</code>	Itens subseqüentes são colocados no segmento de texto do usuário. No SPIM, esses itens só podem ser instruções ou words (ver a diretiva <code>.word</code> a seguir). Se o argumento opcional <i>end</i> estiver presente, os itens subseqüentes são armazenados a partir do endereço <i>end</i> .
<code>.word w1, ..., wn</code>	Armazena as <i>n</i> quantidades de 32 bits em words de memória sucessivas.

O SPIM não distingue as várias partes do segmento de dados (`.data`, `.rdata` e `.sdata`).

### Codificando instruções do MIPS

A Figura A.10.2 explica como uma instrução MIPS é codificada em um número binário. Cada coluna contém codificações de instrução para um campo (um grupo de bits contíguo) a partir de uma instrução. Os números na margem esquerda são valores para um campo. Por exemplo, o opcode `j` possui um valor 2 no campo opcode. O texto no topo de uma coluna nomeia um campo e especifica quais bits ele ocupa em uma instrução. Por exemplo, o campo `op` está contido nos bits 26-31 de uma instrução. Esse campo codifica a maioria das instruções. No entanto, alguns grupos de instruções utilizam



**FIGURA A.10.2 Mapa de opcode do MIPS.** Os valores de cada campo aparecem à sua esquerda. A primeira coluna mostra os valores na base 10 e a segunda mostra a base 16 para o campo `op` (bits 31 a 26) na terceira coluna. Esse campo `op` especifica completamente a operação do MIPS, exceto para 6 valores de `op`: 0, 1, 16, 17, 18 e 19. Essas operações são determinadas pelos outros campos, identificados por ponteiros. O último campo (`funct`) utiliza “`f`” para indicar “`s`” se `rs` = 16 e `op` = 17 ou “`d`” se `rs` = 17 e `op` = 17. O segundo campo (`rs`) usa “`z`” para indicar “0”, “1”, “2” ou “3” se `op` = 16, 17, 18 ou 19, respectivamente. Se `rs` = 16, a operação é especificada em outro lugar: se `z` = 0, as operações são especificadas no quarto campo (bits 4 a 0); se `z` = 1, então as operações são no último campo com `f` = `s`. Se `rs` = 17 e `z` = 1, então as operações estão no último campo com `f` = `d`.

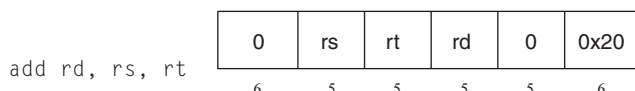


campos adicionais para distinguir instruções relacionadas. Por exemplo, as diferentes instruções de ponto flutuante são especificadas pelos bits 0-5. As setas a partir da primeira coluna mostram quais opcodes utilizam esses campos adicionais.

## Formato de instrução

O restante deste apêndice descreve as instruções implementadas pelo hardware MIPS real e as pseudo-instruções fornecidas pelo montador MIPS. Os dois tipos de instruções podem ser distinguidos facilmente. As instruções reais indicam os campos em sua representação binária. Por exemplo, em

### Adição (com overflow)



a instrução add consiste em seis campos. O tamanho de cada campo em bits é o pequeno número abaixo do campo. Essa instrução começa com 6 bits em 0. Os especificadores de registradores começam com um *r*, de modo que o próximo campo é um especificador de registrador de 5 bits chamado *rs*. Esse é o mesmo registrador que é o segundo argumento no assembly simbólico à esquerda dessa linha. Outro campo comum é  $imm_{16}$ , que é um número imediato de 16 bits.

As pseudo-instruções seguem aproximadamente as mesmas convenções, mas omitem a informação de codificação de instrução. Por exemplo:

### Multiplicação (sem overflow)

mul rdest, rsrc1, src2      *pseudo-instrução*

Nas pseudo-instruções, *rdest* e *rsrc1* são registradores, e *src2* é um registrador ou um valor imediato. Em geral, o montador e o SPIM traduzem uma forma mais geral de uma instrução (por exemplo, add \$v1, \$a0, 0x55) para uma forma especializada (por exemplo, addi \$v1, \$a0, 0x55).

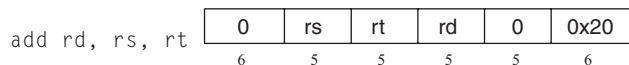
## Instruções aritméticas e lógicas

### Valor absoluto

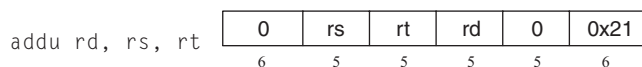
Coloca o valor absoluto do registrador *rsrc* no registrador *rdest*.

abs, rdest, rsrc      *pseudo-instrução*

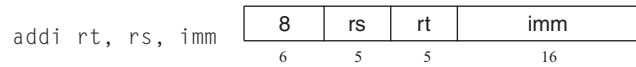
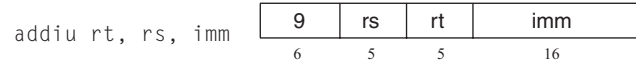
### Adição (com overflow)



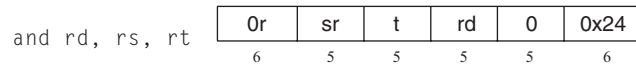
### Adição (sem overflow)



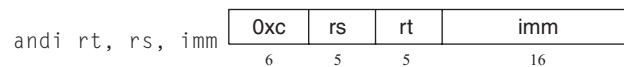
Coloca a soma dos registradores *rs* e *rt* no registrador *rd*.

**Adição imediato (com overflow)**

**Adição imediato (sem overflow)**


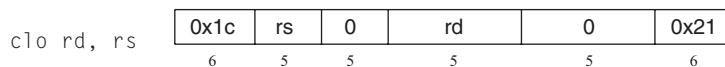
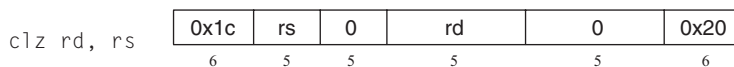
Coloca a soma do registrador rs e o imediato com sinal estendido no registrador rt.

**AND**


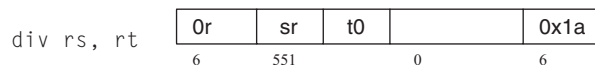
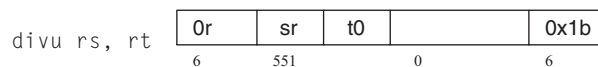
Coloca o AND lógico dos registradores rs e rt no registrador rd.

**AND imediato**


Coloca o AND lógico do registrador rs e o imediato estendido com zeros no registrador rt.

**Contar uns iniciais**

**Contar zeros iniciais**


Conta o número de uns (zeros) iniciais da word no registrador rs e coloca o resultado no registrador rd. Se uma word contém apenas uns (zeros), o resultado é 32.

**Divisão (com overflow)**

**Divisão (sem overflow)**


Divide o registrador rs pelo registrador rt. Deixa o quociente no registrador lo e o resto no registrador hi. Observe que, se um operando for negativo, o restante não será especificado pela arquitetura MIPS e dependerá da convenção da máquina em que o SPIM é executado.

**Divisão (com overflow)**

div rdest, rsrc1, src2 *pseudo-instrução*

**Divisão (sem overflow)**

divu rdest, rsrc1, src2 *pseudo-instrução*

Coloca o quociente do registrador rsrc1 pelo src2 no registrador rdest.

**Multiplicação**

mult rs, rt

0	rs	rt	0	0x18
6	5	5	10	6

**Multiplicação sem sinal**

multu rs, rt

0	rs	rt	0	0x19
6	5	5	10	6

Multiplica os registradores rs e rt. Deixa a word menos significativa do produto no registrador lo e a word mais significativa no registrador hi.

**Multiplicação (sem overflow)**

mul rd, rs, rt

0x1c	rs	rt	rd	0	2
6	5	5	5	5	6

Coloca os 32 bits menos significativos do produto de rs e rt no registrador rd.

**Multiplicação (com overflow)**

mulo rdest, rsrc1, src2 *pseudo-instrução*

**Multiplicação sem sinal (com overflow)**

mulou rdest, rsrc1, src2 *pseudo-instrução*

Coloca os 32 bits menos significativos do produto do registrador rsrc1 e src2 no registrador rdest.

**Multiplicação adição**

madd rs, rt

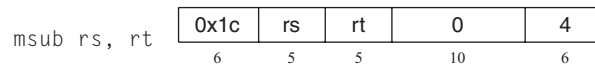
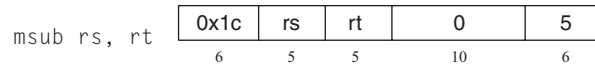
0x1c	rs	rt	0	0
6	5	5	10	6

**Multiplicação adição sem sinal**

maddu rs, rt

0x1c	rs	rt	0	1
6	5	5	10	6

Multiplica os registradores rs e rt e soma o produto de 64 bits resultante ao valor de 64 bits nos registradores concatenados lo e hi.

**Multiplicação subtração**

**Multiplicação subtração sem sinal**


Multiplica os registradores rs e rt e subtrai o produto de 64 bits resultante do valor de 64 bits nos registradores concatenados lo e hi.

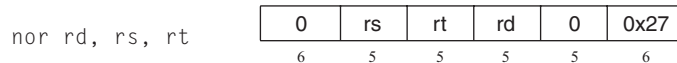
**Negar valor (com overflow)**

neg rdest, rsrc      *pseudo-instrução*

**Negar valor (sem overflow)**

negu rdest,rsrc      *pseudo-instrução*

Coloca o negativo do registrador rsrc no registrador rdest.

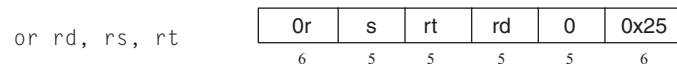
**NOR**


Coloca o NOR lógico dos registradores rs e rt para o registrador rd.

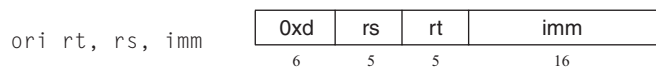
**NOT**

not rdest, rsrc      *pseudo-instrução*

Coloca a negação lógica bit a bit do registrador rsrc no registrador rdest.

**OR**


Coloca o OR lógico dos registradores rs e rt no registrador rd.

**OR imediato**


Coloca o OR lógico do registrador rs e o imediato estendido com zero no registrador rt.

**Resto**

rem rdest, rsrc1, rsrc2 *pseudo-instrução*

**Resto sem sinal**

remu rdest, rsrc1, rsrc2 *pseudo-instrução*

Coloca o resto do registrador rsrc1 dividido pelo registrador rsrc2 no registrador rdest. Observe que se um operando for negativo, o resto não é especificado pela arquitetura MIPS e depende da convenção da máquina em que o SPIM é executado.

**Shift lógico à esquerda**

sll rd, rt, shamt

0	rs	rt	rd	shamt	0
6	5	5	5	5	6

**Shift lógico à esquerda variável**

sllv rd, rt, rs

0	rs	rt	rd	0	4
6	5	5	5	5	6

**Shift aritmético à direita**

sra rd, rt, shamt

0	rs	rt	rd	shamt	3
6	5	5	5	5	6

**Shift aritmético à direita variável**

srav rd, rt, rs

0	rs	rt	rd	0	7
6	5	5	5	5	6

**Shift lógico à direita**

srl rd, rt, shamt

0	rs	rt	rd	shamt	2
6	5	5	5	5	6

**Shift lógico à direita variável**

srlv rd, rt, rs

0	rs	rt	rd	0	6
6	5	5	5	5	6

Desloca o registrador rt à esquerda (direita) pela distância indicada pelo shamt imediato ou pelo registrador rs e coloca o resultado no registrador rd. Observe que o argumento rs é ignorado para sll, sra e srl.

**Rotate à esquerda**

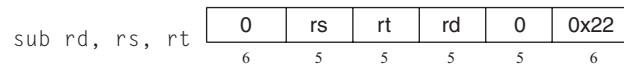
rol rdest, rsrc1, rsrc2 *pseudo-instrução*

**Rotate à direita**

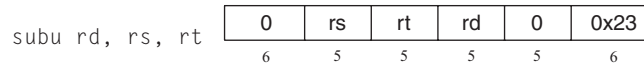
ror rdest, rsrc1, rsrc2 *pseudo-instrução*

Gira o registrador rsrc1 à esquerda (direita) pela distância indicada por rsrc2 e coloca o resultado no registrador rdest.

### Subtração (com overflow)

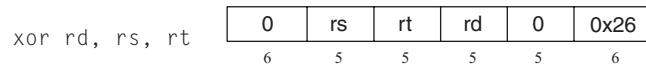


### Subtração (sem overflow)



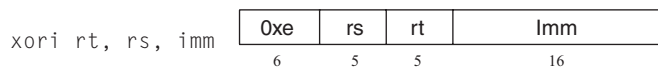
Coloca a diferença dos registradores rs e rt no registrador rd.

### OR exclusivo



Coloca o XOR lógico dos registradores rs e rt no registrador rd.

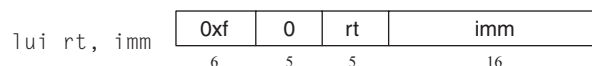
### XOR imediato



Coloca o XOR lógico do registrador rs e o imediato estendido com zeros no registrador rt.

## Instruções para manipulação de constantes

### Load superior imediato



Carrega a halfword menos significativa do imediato imm na halfword mais significativa do registrador rt. Os bits menos significativos do registrador são colocados em 0.

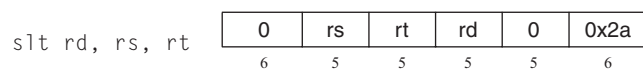
### Load imediato

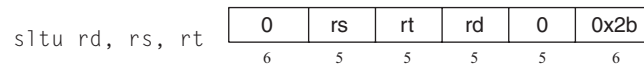
li rdest, imm *pseudo-instrução*

Move o imediato imm para o registrador rdest.

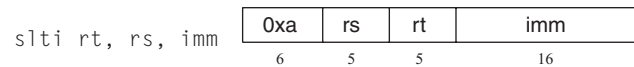
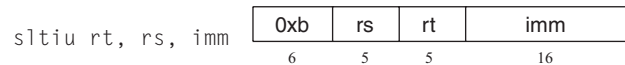
## Instruções de comparação

### Set se menor que



**Set se menor que sem sinal**

Coloca o registrador rd em 1 se o registrador rs for menor que rt; caso contrário, coloca-o em 0.

**Set se menor que imediato****Set se menor que imediato sem sinal**

Coloca o registrador rt em 1 se o registrador rs for menor que o imediato estendido com sinal, e em 0 em caso contrário.

**Set se igual**

seq rdest, rsrc1, rsrc2 *pseudo-instrução*

Coloca o registrador rdest em 1 se o registrador rsrc1 for igual a rsrc2, e em 0 caso contrário.

**Set se maior ou igual**

sge rdest, rsrc1, rsrc2 *pseudo-instrução*

**Set se maior ou igual sem sinal**

sgeu rdest, rsrc1, rsrc2 *pseudo-instrução*

Coloca o registrador rdest em 1 se o registrador rsrc1 for maior ou igual a rsrc2, e em 0 caso contrário.

**Set se maior que**

sgt rdest, rsrc1, rsrc2 *pseudo-instrução*

**Set se maior que sem sinal**

sgtu rdest, rsrc1, rsrc2 *pseudo-instrução*

Coloca o registrador rdest em 1 se o registrador rsrc1 for maior que rsrc2, e em 0 caso contrário.

**Set se menor ou igual**

sle rdest, rsrc1, rsrc2 *pseudo-instrução*

**Set se menor ou igual sem sinal**

sleu rdest, rsrc1, rsrc2 *pseudo-instrução*

Coloca o registrador rdest em 1 se o registrador rsrc1 for menor ou igual a rsrc2, e em 0 caso contrário.



### Set se diferente

sne rdest, rsrc1, rsrc2      *pseudo-instrução*

Coloca o registrador rdest em 1 se o registrador rsrc1 não for igual a rsrc2, e em 0 caso contrário.

### Instruções de desvio

As instruções de desvio utilizam um campo *offset* de instrução de 16 bits com sinal; logo, elas podem desviar  $2^{15} - 1$  *instruções* (não bytes) para a frente ou  $2^{15}$  instruções para trás. A instrução *jump* contém um campo de endereço de 26 bits. Em processadores MIPS reais, as instruções de desvio são *delayed branches*, que não transferem o controle até que a instrução após o desvio (seu “delay slot”) tenha sido executado (ver Capítulo 6). Os *delayed branches* afetam o cálculo de *offset*, pois precisam ser calculados em relação ao endereço da instrução do *delay slot* (PC + 4), que é quando o desvio ocorre. O SPIM não simula esse *delay slot*, a menos que os flags *-bare* ou *-delayed\_branch* sejam especificados.

No código *assembly*, os *offsets* normalmente não são especificados como números. Em vez disso, uma instrução desvia para um rótulo, e o montador calcula a distância entre o desvio e a instrução destino.

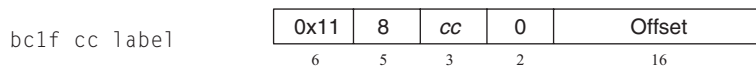
No MIPS32, todas as instruções de desvio condicional reais (não *pseudo*) têm uma variante “provável” (por exemplo, a variável *provável* de *beq* é *beql*), que *não* executa a instrução no *delay slot* do desvio se o desvio não for tomado. Não use essas instruções; elas poderão ser removidas em versões subseqüentes da arquitetura. O SPIM implementa essas instruções, mas elas não são descritas daqui por diante.

### Branch

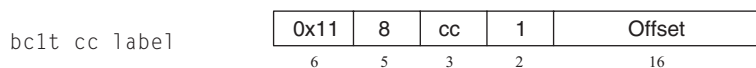
b label      *pseudo-instrução*

Desvia incondicionalmente para a instrução no rótulo.

### Branch co-processor falso



### Branch co-processor verdadeiro



Desvia condicionalmente pelo número de instruções especificado pelo *offset* se o flag de condição de ponto flutuante numerado como *cc* for falso (verdadeiro). Se *cc* for omitido da instrução, o flag de código de condição 0 é assumido.

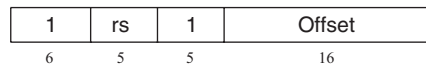
### Branch se for igual



Desvia condicionalmente pelo número de instruções especificado pelo *offset* se o registrador *rs* for igual a *rt*.

**Branch se for maior ou igual a zero**

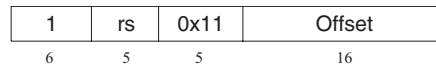
bgez rs, label



Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador rs for maior ou igual a 0.

**Branch se for maior ou igual a zero e link**

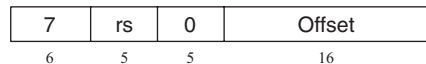
bgezal rs, label



Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador rs for maior ou igual a 0. Salva o endereço da próxima instrução no registrador 31.

**Branch se for maior que zero**

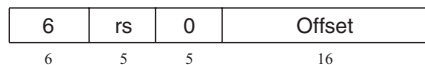
bgtz rs, label



Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador rs for maior que 0.

**Branch se for menor ou igual a zero**

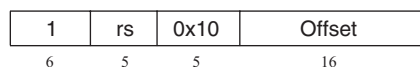
blez rs, label



Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador rs for menor ou igual a 0.

**Branch se for menor e link**

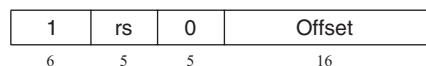
bltzal rs, label



Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador rs for menor que 0. Salva o endereço da próxima instrução no registrador 31.

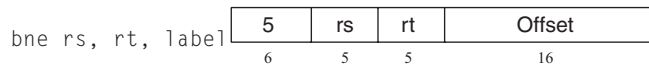
**Branch se for menor que zero**

bltz rs, label



Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador rs for menor que 0.

**Branch se for diferente**



Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador rs não for igual a rt.

**Branch se for igual a zero**

beq rsrc, label *pseudo-instrução*

Desvia condicionalmente para a instrução no rótulo se rsrc for igual a 0.

**Branch se for maior ou igual**

bge rsrc1, rsrc2, label *pseudo-instrução*

**Branch se for maior ou igual com sinal**

bgeu rsrc1, rsrc2, label *pseudo-instrução*

Desvia condicionalmente até a instrução no rótulo se o registrador rsrc1 for maior ou igual a rsrc2.

**Branch se for maior**

bgt rsrc1, src2, label *pseudo-instrução*

**Branch se for maior sem sinal**

bgtu rsrc1, src2, label *pseudo-instrução*

Desvia condicionalmente para a instrução no rótulo se o registrador rsrc1 for maior do que src2.

**Branch se for menor ou igual**

ble rsrc1, src2, label *pseudo-instrução*

**Branch se for menor ou igual sem sinal**

bleu rsrc1, src2, label *pseudo-instrução*

Desvia condicionalmente para a instrução no rótulo se o registrador rsrc1 for menor ou igual a rsrc2.

**Branch se for menor**

blt rsrc1, rsrc2, label *pseudo-instrução*

**Branch se for menor sem sinal**

bltu rsrc1, rsrc2, label *pseudo-instrução*

Desvia condicionalmente para a instrução no rótulo se o registrador rsrc1 for menor do que src2.

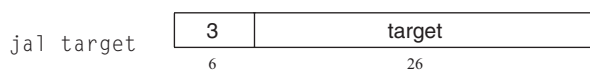
**Branch se não for igual a zero**

bnez rsrc, label *pseudo-instrução*

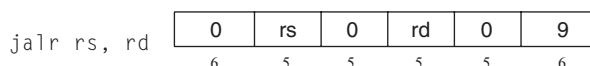
Desvia condicionalmente para a instrução no rótulo se o registrador rsrc não for igual a 0.

**Instruções de jump****Jump**

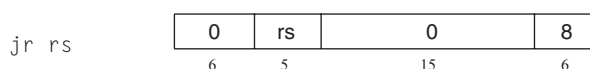
Desvia incondicionalmente para a instrução no destino.

**Jump-and-link**

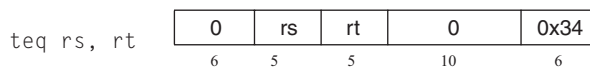
Desvia incondicionalmente para a instrução no destino. Salva o endereço da próxima instrução no registrador \$ra.

**Jump-and-link registrador**

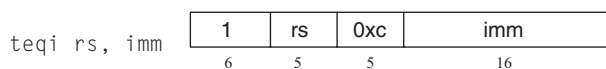
Desvia incondicionalmente para a instrução cujo endereço está no registrador rs. Salva o endereço da próxima instrução no registrador rd (cujo default é 31).

**Jump registrador**

Desvia incondicionalmente para a instrução cujo endereço está no registrador rs.

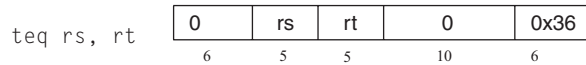
**Instruções de trap****Trap se for igual**

Se o registrador rs for igual ao registrador rt, gera uma exceção de Trap.

**Trap se for igual imediato**

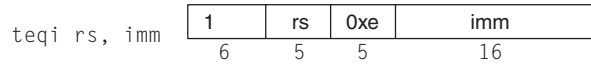
Se o registrador rs for igual ao valor de imm com sinal estendido, gera uma exceção de Trap.

**Trap se não for igual**



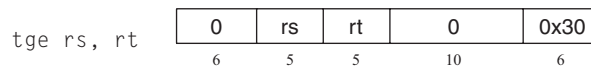
Se o registrador rs não for igual ao registrador rt, gera uma exceção de Trap.

**Trap se não for igual imediato**

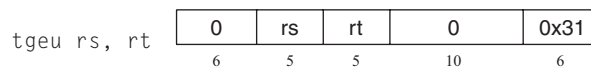


Se o registrador rs não for igual ao valor de imm com sinal estendido, gera uma exceção de Trap.

**Trap se for maior ou igual**

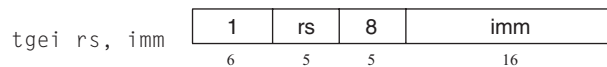


**Trap sem sinal se for maior ou igual**

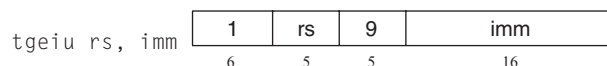


Se o registrador rs for maior ou igual ao registrador rt, gera uma exceção de Trap.

**Trap se for maior ou igual imediato**

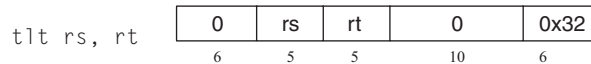


**Trap sem sinal se for maior ou igual imediato**

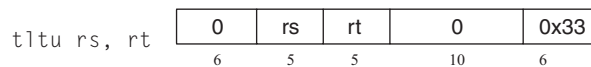


Se o registrador rs for maior ou igual ao valor de imm com sinal estendido, gera uma exceção de Trap.

**Trap se for menor**

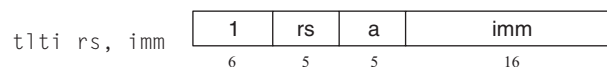


**Trap sem sinal se for menor**



Se o registrador rs for menor que o registrador rt, gera uma exceção de Trap.

**Trap se for menor imediato**



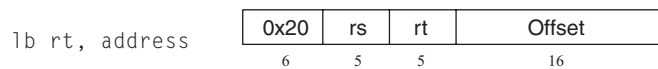
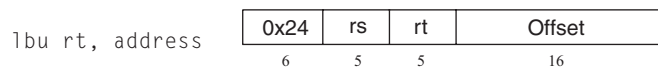
**Trap sem sinal se for menor imediato**

Se o registrador rs for menor do que o valor de imm com sinal estendido, gera uma exceção de Trap.

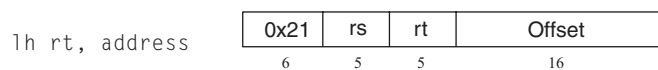
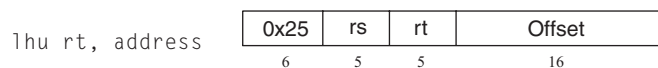
**Instruções load****Load endereço**

la rdest, address *pseudo-instrução*

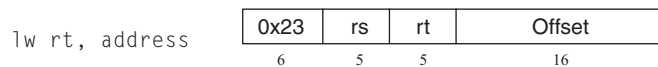
Carrega o *endereço* calculado – não o conteúdo do local – para o registrador rdest.

**Load byte****Load byte sem sinal**

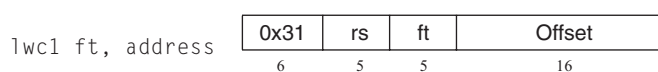
Carrega o byte no *endereço* para o registrador rt. O byte tem sinal estendido por 1b, mas não por 1bu.

**Load halfword****Load halfword sem sinal**

Carrega a quantidade de 16 bits (halfword) no *endereço* para o registrador rt. A halfword tem sinal estendido por lh, mas não por lhu.

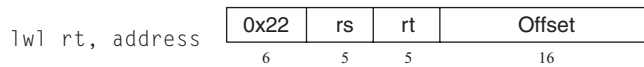
**Load word**

Carrega a quantidade de 32 bits (word) no *endereço* para o registrador rt.

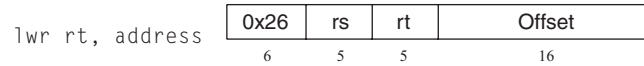
**Load word co-processor 1**

Carrega a word no *endereço* para o registrador ft da unidade de ponto flutuante.

### Load word à esquerda



### Load word à direita



Carrega os bytes da esquerda (direita) da word do *endereço* possivelmente não alinhado para o registrador rt.

### Load doubleword

ld rdest, address *pseudo-instrução*

Carrega a quantidade de 64 bits no *endereço* para os registradores rdest e rdest + 1.

### Load halfword não alinhada

ulh rdest, address *pseudo-instrução*

### Load halfword sem sinal não alinhada

ulhu rdest, address *pseudo-instrução*

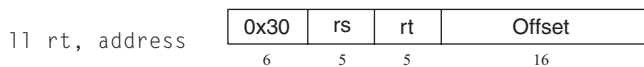
Carrega a quantidade de 16 bits (halfword) no *endereço* possivelmente não alinhado para o registrador rdest. A halfword tem extensão de sinal por ulh, mas não ulhu.

### Load word não alinhada

ulw rdest, address *pseudo-instrução*

Carrega a quantidade de 32 bits (word) no *endereço* possivelmente não alinhado para o registrador rdest.

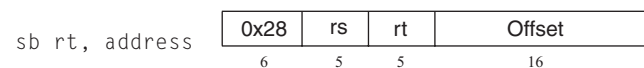
### Load Linked



Carrega a quantidade de 32 bits (word) no *endereço* para o registrador rt e inicia uma operação ler-modificar-escrever indivisível. Essa operação é concluída por uma instrução de armazenamento condicional (sc), que falhará se outro processador escrever no bloco que contém a word carregada. Como o SPIM não simula processadores múltiplos, a operação de armazenamento condicional sempre tem sucesso.

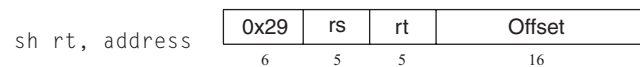
## Instruções store

### Store byte

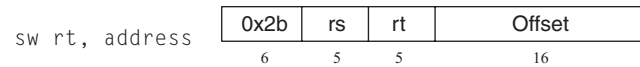


Armazena o byte baixo do registrador rt no *endereço*.

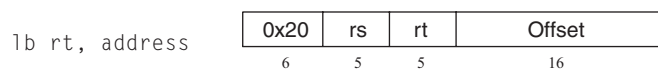


**Store halfword**

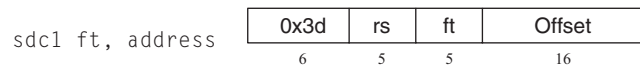
Armazena a halfword baixa do registrador rt no *endereço*.

**Store word**

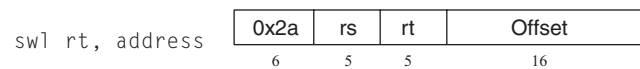
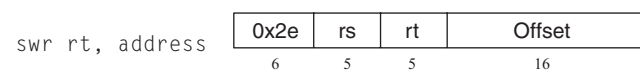
Armazena a word do registrador rt no *endereço*.

**Store word co-processador 1**

Armazena o valor de ponto flutuante no registrador ft do co-processador de ponto flutuante no *endereço*.

**Store double co-processador 1**

Armazena o valor de ponto flutuante da dupla word nos registradores ft e ft + 1 do co-processador de ponto flutuante em *endereço*. O registrador ft precisa ser um número par.

**Store word à esquerda****Store word à direita**

Armazena os bytes da esquerda (direita) do registrador rt no *endereço* possivelmente não alinhado.

**Store doubleword**

sd rsrc, address *pseudo-instrução*

Armazena a quantidade de 64 bits nos registradores rsrc e rsrc + 1 no *endereço*.

**Store halfword não alinhada**

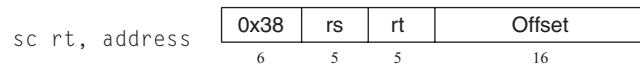
ush rsrc, address *pseudo-instrução*

Armazena a halfword baixa do registrador rsrc no *endereço* possivelmente não alinhado.

**Store word não alinhada**

usw rsrc, address *pseudo-instrução*

Armazena a word do registrador rsrc no *endereço* possivelmente não alinhado.



### Store condicional

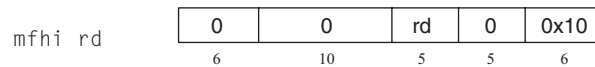
Armazena a quantidade de 32 bits (word) no endereço rt para a memória no *endereço* e completa uma operação ler-modificar-escrever indivisível. Se essa operação indivisível tiver sucesso, a word da memória será modificada e o registrador rt será colocado em 1. Se a operação indivisível falhar porque outro processador escreveu em um local no bloco contendo a word endereçada, essa instrução não modifica a memória e escreve 0 no registrador rt. Como o SPIM não simula diversos processadores, a instrução sempre tem sucesso.

## Instruções para movimentação de dados

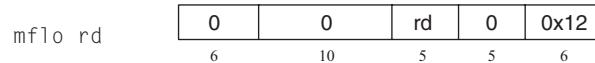
### Move

move rdest, rsrc *pseudo-instrução*

Move o registrador rsrc para rdest.

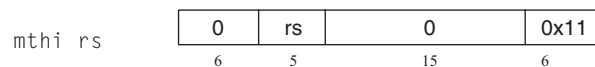


### Move de hi

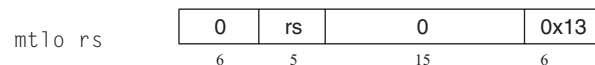


### Move de lo

A unidade de multiplicação e divisão produz seu resultado em dois registradores adicionais, hi e lo. Essas instruções movem os valores de e para esses registradores. As pseudo-instruções de multiplicação, divisão e resto que fazem com que essa unidade pareça operar sobre os registradores gerais movem o resultado depois que o cálculo terminar. Move o registrador hi (lo) para o registrador rd.

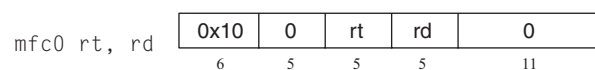


### Move para hi

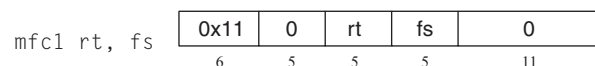


### Move para lo

Move o registrador rs para o registrador hi (lo).



### Move do co-processador 0



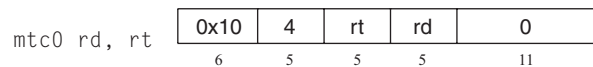
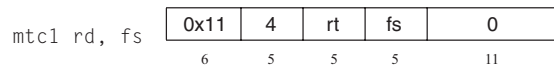
**Move do co-processador 1**

Os co-processadores têm seus próprios conjuntos de registradores. Essas instruções movem valores entre esses registradores e os registradores da CPU. Move o registrador rd em um co-processador (registrador fs na FPU) para o registrador rt da CPU. A unidade de ponto flutuante é o co-processador 1.

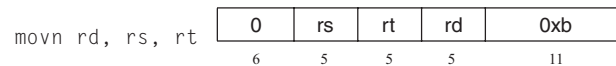
**Move double do co-processador 1**

`mfcl.d rdest, frsrc1` *pseudo-instrução*

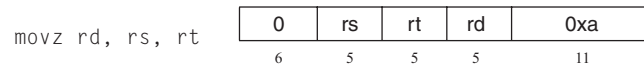
Move os registradores de ponto flutuante frsrc1 e frsrc1 + 1 para os registradores da CPU rdest e rdest + 1.

**Move para co-processador 0****Move para co-processador 1**

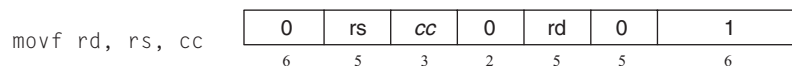
Move o registrador da CPU rt para o registrador rd em um co-processador (registrador fs na FPU).

**Move condicional diferente de zero**

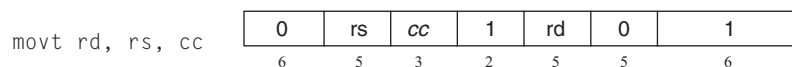
Move o registrador rs para o registrador rd se o registrador rt não for 0.

**Move condicional zero**

Move o registrador rs para o registrador rd se o registrador rt for 0.

**Move condicional em caso de FP falso**

Move o registrador da CPU rs para o registrador rd se o flag de código de condição da FPU número cc for 0. Se cc for omitido da instrução, o flag de código de condição 0 será assumido.

**Move condicional em caso de FP verdadeiro**

Move o registrador da CPU rs para o registrador rd se o flag de código de condição da FPU número cc for 1. Se cc for omitido da instrução, o bit de código de condição 0 é assumido.

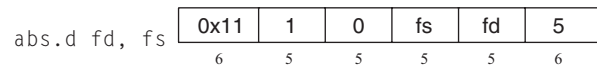
## Instruções de ponto flutuante

O MIPS possui um co-processador de ponto flutuante (número 1) que opera sobre números de ponto flutuante de precisão simples (32 bits) e precisão dupla (64 bits). Esse co-processador tem seus próprios registradores, que são numerados de \$f0 a \$f31. Como esses registradores possuem apenas 32 bits, dois deles são necessários para manter doubles, de modo que somente registradores de ponto flutuante com números pares podem manter valores de precisão dupla. O co-processador de ponto flutuante também possui 8 flags de código de condição (cc), numerados de 0 a 7, que são alterados por instruções de comparação e testados por instruções de desvio (bc1f ou bc1t) e instruções move condicionais.

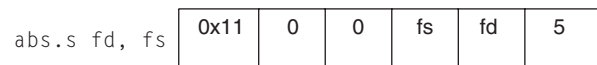
Os valores são movidos para dentro e para fora desses registradores uma word (32 bits) de cada vez pelas instruções lwc1, swc1, mtc1 e mfc1 ou um double (64 bits) de cada vez por ldc1 e sdc1, descritos anteriormente, ou pela pseudo-instruções l.s, l.d, s.s e s.d, descritas a seguir.

Nas instruções reais a seguir, os bits 21-26 são 0 para precisão simples e 1 para precisão double. Nas pseudo-instruções a seguir, fdest é um registrador de ponto flutuante (por exemplo, \$f2).

### Valor absoluto de ponto flutuante double

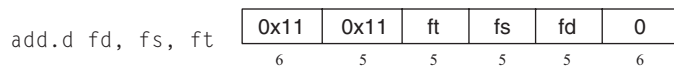


### Valor absoluto de ponto flutuante single

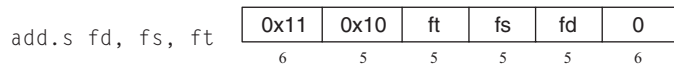


Calcula o valor absoluto do double (single) de ponto flutuante no registrador fs e o coloca no registrador fd.

### Adição de ponto flutuante double

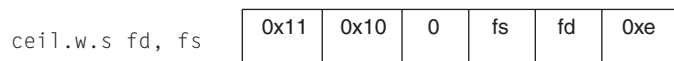
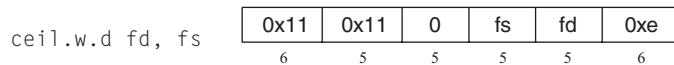


### Adição de ponto flutuante single



Calcula a soma dos doubles (singles) de ponto flutuante nos registradores fs e ft e a coloca no registrador fd.

### Teto de ponto flutuante para word



Calcula o teto do double (single) de ponto flutuante no registrador fs, converte para um valor de ponto fixo de 32 bits e coloca a word resultante no registrador fd.

**Comparação igual double**

c.eq.d cc fs, ft	0x11	0x11	ft	fs	cc	0	FC	2
	6	5	5	5	3	2	2	4

**Comparação igual single**

c.eq.s cc fs, ft	0x11	0x10	ft	fs	cc	0	FC	2
	6	5	5	5	3	2	2	4

Compara o double (single) de ponto flutuante no registrador fs com aquele em ft e coloca o flag de condição de ponto flutuante cc em 1 se forem iguais. Se cc for omitido, o flag de código de condição 0 é assumido.

**Comparação menor ou igual double**

c.le.d cc fs, ft	0x11	0x11	ft	fs	cc	0	FC	0xe
	6	5	5	5		2	2	4

**Comparação menor ou igual single**

c.le.s cc fs, ft	0x11	0x10	ft	fs	cc	0	FC	0xe
	6	5	5	5	3	2	2	4

Compara o double (single) de ponto flutuante no registrador fs com aquele no ft e coloca o flag de condição de ponto flutuante cc em 1 se o primeiro for menor ou igual ao segundo. Se o cc for omitido, o flag de código de condição 0 é assumido.

**Comparação menor que double**

c.lt.d cc fs, ft	0x11	0x11	ft	fs	cc	0	FC	0xc
	6	5	5	5	3	2	2	4

**Comparação menor que single**

c.lt.s cc fs, ft	0x11	0x10	ft	fs	cc	0	FC	0xc
	6	5	5	5	3	2	2	4

Compara o double (single) de ponto flutuante no registrador fs com aquele no ft e coloca o flag de condição de ponto flutuante cc em 1 se o primeiro for menor que o segundo. Se o cc for omitido, o flag de código de condição 0 é assumido.

**Converte single para double**

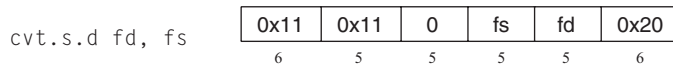
cvt.d.s fd, fs	0x11	0x10	0	fs	fd	0x21
	6	5	5	5	5	6

**Converte integer para double**

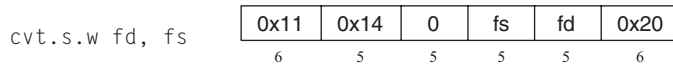
cvt.d.w fd, fs	0x11	0x14	0	fs	fd	0x21
	6	5	5	5	5	6

Converte o número de ponto flutuante de precisão simples ou inteiro no registrador fs para um número de precisão dupla (simples) e o coloca no registrador fd.

**Converte double para single**

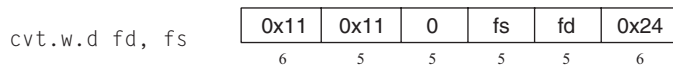


**Converte integer para single**

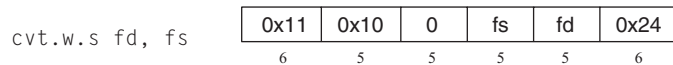


Converte o número de ponto flutuante de precisão dupla ou inteiro no registrador fs para um número de precisão simples e o coloca no registrador fd.

**Converte double para integer**

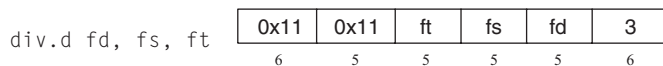


**Converte single para integer**

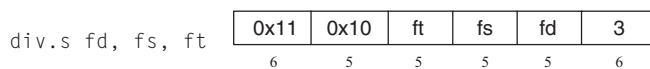


Converte o número de ponto flutuante de precisão dupla ou simples no registrador fs para um inteiro e o coloca no registrador fd.

**Divisão de ponto flutuante double**

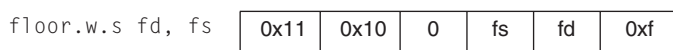
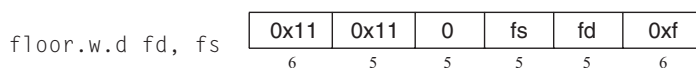


**Divisão de ponto flutuante single**



Calcula o quociente dos números de ponto flutuante de precisão dupla (simples) nos registradores fs e ft e o coloca no registrador fd.

**Piso de ponto flutuante para word**



Calcula o piso do número de ponto flutuante de precisão dupla (simples) no registrador fs e coloca a word resultante no registrador fd.

**Carrega double de ponto flutuante**

l.d fdest, address      *pseudo-instrução*

**Carrega single de ponto flutuante**

l.s fdest, address *pseudo-instrução*

Carrega o número de ponto flutuante de precisão dupla (simples) em address para o registrador fdest.

**Move ponto flutuante double**

mov.d fd, fs	0x11	0x11	0	fs	fd	6
	6	5	5	5	5	6

**Move ponto flutuante single**

mov.s fd, fs	0x11	0x10	0	fs	fd	6
	6	5	5	5	5	6

Move o número de ponto flutuante de precisão dupla (simples) do registrador fs para o registrador fd.

**Move condicional de ponto flutuante double se falso**

movf.d fd, fs, cc	0x11	0x11	cc	0	fs	fd	0x11
	6	5	3	2	5	5	6

**Move condicional de ponto flutuante single se falso**

movf.s fd, fs, cc	0x11	0x10	cc	0	fs	fd	0x11
	6	5	3	2	5	5	6

Move o número de ponto flutuante de precisão dupla (simples) do registrador fs para o registrador fd se o flag do código de condição cc for 0. Se o cc for omitido, o flag de código de condição 0 é assumido.

**Move condicional de ponto flutuante double se verdadeiro**

movt.d fd, fs, cc	0x11	0x11	cc	1	fs	fd	0x11
	6	5	3	2	5	5	6

**Move condicional de ponto flutuante single se verdadeiro**

movt.s fd, fs, cc	0x11	0x10	cc	1	fs	fd	0x11
	6	5	3	2	5	5	6

Move o double (single) de ponto flutuante do registrador fs para o registrador fd se o flag do código de condição cc for 1. Se o cc for omitido, o flag do código de condição 0 será assumido.

**Move ponto flutuante double condicional se não for zero**

movn.d fd, fs, rt	0x11	0x11	rt	fs	fd	0x13
	6	5	5	5	5	6

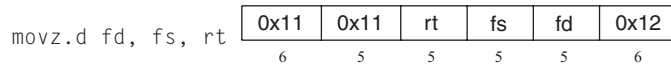
**Move ponto flutuante single condicional se não for zero**

movn.s fd, fs, rt	0x11	0x10	rt	fs	fd	0x13
	6	5	5	5	5	6

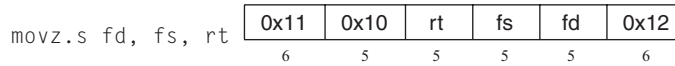
Move o número de ponto flutuante double (single) do registrador fs para o registrador fd se o registrador rt do processador não for 0.



**Move ponto flutuante double condicional se for zero**

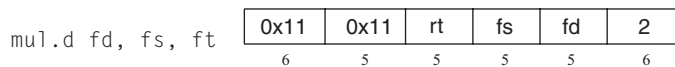


**Move ponto flutuante single condicional se for zero**

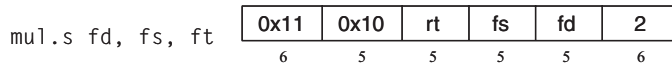


Move o número de ponto flutuante double (single) do registrador fs para o registrador fd se o registrador rt do processador for 0.

**Multiplicação de ponto flutuante double**

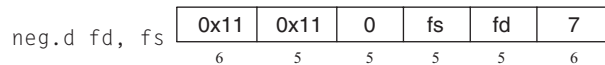


**Multiplicação de ponto flutuante single**

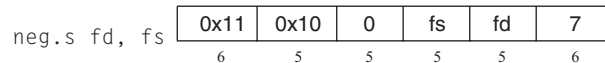


Calcula o produto dos números de ponto flutuante double (single) nos registradores fs e ft e o coloca no registrador fd.

**Negação double**

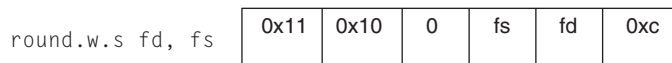
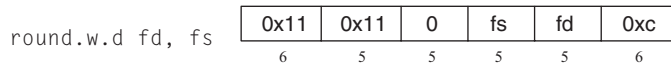


**Negação single**



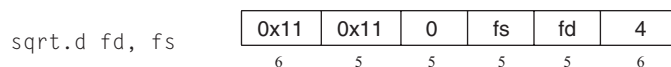
Nega o número de ponto flutuante double (single) no registrador fs e o coloca no registrador fd.

**Arredondamento de ponto flutuante para word**



Arredonda o valor de ponto flutuante double (single) no registrador fs, converte para um valor de ponto fixo de 32 bits e coloca a word resultante no registrador fd.

**Raiz quadrada de double**



**Raiz quadrada de single**

sqrt.s fd, fs	0x11	0x10	0	fs	fd	4
	6	5	5	5	5	6

Calcula a raiz quadrada do número de ponto flutuante double (single) no registrador fs e a coloca no registrador fd.

**Store de ponto flutuante double**

s.d fdest, address *pseudo-instrução*

**Store de ponto flutuante single**

s.s fdest, address *pseudo-instrução*

Armazena o número de ponto flutuante double (single) no registrador fdest em *address*.

**Subtração de ponto flutuante double**

sub.d fd, fs, ft	0x11	0x11	ft	fs	fd	1
	6	5	5	5	5	6

**Subtração de ponto flutuante single**

sub.s fd, fs, ft	0x11	0x10	ft	fs	fd	1
	6	5	5	5	5	6

Calcula a diferença dos números de ponto flutuante double (single) nos registradores fs e ft e a coloca no registrador fd.

**Truncamento de ponto flutuante para word**

trunc.w.d fd, fs	0x11	0x11	0	fs	fd	0xd
	6	5	5	5	5	6

trunc.w.s fd, fs	0x11	0x10	0	fs	fd	0xd
------------------	------	------	---	----	----	-----

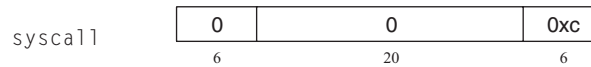
Trunca o valor de ponto flutuante double (single) no registrador fs, converte para um valor de ponto fixo de 32 bits e coloca a word resultante no registrador fd.

**Instruções de exceção e interrupção****Retorno de exceção**

eret	0x10	1	0	0x18
	6	1	19	6

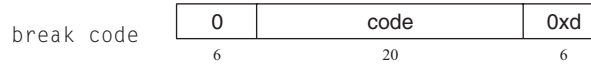
Coloca em 0 o bit EXL no registrador Status do co-processador 0 e retorna à instrução apontada pelo registrador EPC do co-processador 0.

### Chamada ao sistema



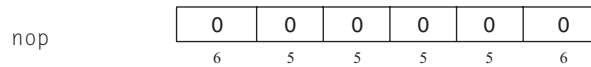
O registrador \$v0 contém o número da chamada ao sistema (ver Figura A.9.1) fornecido pelo SPIM.

### Break



Causa a exceção *código*. A Exceção 1 é reservada para o depurador.

### Nop



Não faz nada.

## A.11

## Comentários finais

A programação em assembly exige que um programador escolha entre os recursos úteis das linguagens de alto nível – como estruturas de dados, verificação de tipo e construções de controle – e o controle completo sobre as instruções que um computador executa. Restrições externas sobre algumas aplicações, como o tempo de resposta ou o tamanho do programa, exigem que um programador preste muita atenção a cada instrução. No entanto, o custo desse nível de atenção são programas em assembly maiores, mais demorados para escrever e mais difícil de manter do que os programas em linguagem de alto nível.

Além do mais, três tendências estão reduzindo a necessidade de escrever programas em assembly. A primeira tendência é em direção à melhoria dos compiladores. Os compiladores modernos produzem código comparável ao melhor código escrito manualmente – e, às vezes, melhor ainda. A segunda tendência é a introdução de novos processadores, que não apenas são mais rápidos, mas, no caso de processadores que executam várias instruções ao mesmo tempo, também mais difíceis de programar manualmente. Além disso, a rápida evolução dos computadores modernos favorece os programas em linguagem de alto nível que não estejam presos a uma única arquitetura. Finalmente, temos testemunhado uma tendência em direção a aplicações cada vez mais complexas, caracterizadas por interfaces gráficas complexas e muito mais recursos do que seus predecessores. Grandes aplicações são escritas por equipes de programadores e exigem recursos de modularidade e verificação semântica fornecidos pelas linguagens de alto nível.

### Leitura adicional

Aho, A., R. Sethi e J. Ullman [1985]. *Compilers: Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley.

*Ligeiramente desatualizado e faltando a cobertura das arquiteturas modernas, mas ainda é a referência padrão sobre compiladores.*

Sweetman, D. [1999]. *See MIPS Run*, San Francisco CA: Morgan Kaufmann Publishers.

*Uma introdução completa, detalhada e envolvente sobre o conjunto de instruções do MIPS e a programação em assembly nessas máquinas.*

A documentação detalhada sobre a arquitetura MIPS32 está disponível na Web:

**MIPS32™ Architecture for Programmers Volume I: Introduction to the MIPS32 Architecture** (<http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/ArchitectureProgrammingPublicationsforMIPS32/MD00082-2B-MIPS32INT-AFP-02.00.pdf/getDownload>)

**MIPS32™ Architecture for Programmers Volume II: The MIPS32 Instruction Set** (<http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/ArchitectureProgrammingPublicationsforMIPS32/MD00086-2B-MIPS32BIS-AFP-02.00.pdf/getDownload>)

**MIPS32™ Architecture for Programmers Volume III: The MIPS32 Privileged Resource Architecture** (<http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/ArchitectureProgrammingPublicationsforMIPS32/MD00090-2B-MIPS32PRA-AFP-02.00.pdf/getDownload>)

## A.12

## Exercícios

- A.1** [5] <§A.5> A Seção A.5 descreveu como a memória é dividida na maioria dos sistemas MIPS. Proponha outra maneira de dividir a memória, que cumpra os mesmos objetivos.
- A.2** [20] <§A.6> Reescreva o código para fact utilizando menos instruções.
- A.3** [5] <§A.7> É seguro que um programa do usuário utilize os registradores \$k0 ou \$k1?
- A.4** [25] <§A.7> A Seção A.7 contém código para um handler de exceção muito simples. Um problema sério com esse handler é que ele desativa as interrupções por um longo tempo. Isso significa que as interrupções de um dispositivo de E/S rápido podem ser perdidas. Escreva um handler de exceção melhor, que não possa ser interrompido mas que ative as interrupções o mais rápido possível.
- A.5** [15] <§A.7> O handler de exceção simples sempre desvia para a instrução após a exceção. Isso funciona bem, a menos que a instrução que causou a exceção esteja no delay slot de um desvio. Nesse caso, a próxima instrução será o alvo do desvio. Escreva um handler melhor, que use o registrador EPC para determinar qual instrução deverá ser executada após a exceção.
- A.6** [5] <§A.9> Usando o SPIM, escreva e teste um programa de calculadora que leia inteiros repetidamente e os adicione a um acumulador. O programa deverá parar quando receber uma entrada 0, imprimindo a soma nesse ponto. Use as chamadas do sistema do SPIM descritas nas páginas A-32 e A-33.
- A.7** [5] <§A.9> Usando o SPIM, escreva e teste um programa que leia três inteiros e imprima a soma dos dois maiores desses três. Use as chamadas do sistema do SPIM descritas nas páginas A-32 e A-33.
- A.8** [5] <§A.9> Usando o SPIM, escreva e teste um programa que leia um inteiro positivo usando as chamadas do sistema do SPIM. Se o inteiro não for positivo, o programa deverá terminar com a mensagem “Entrada inválida”; caso contrário, o programa deverá imprimir os nomes dos dígitos dos inteiros por extenso, delimitados por exatamente um espaço. Por exemplo, se o usuário informou “728”, a saída deverá ser “Sete Dois Oito”.
- A.9** [25] <§A.9> Escreva e teste um programa em assembly do MIPS para calcular e imprimir os 100 primeiros números primos. Um número  $n$  é primo se ele só puder ser dividido exatamente por ele mesmo e por 1. Duas rotinas deverão ser implementadas:
- `testa_primo (n)` Retorna 1 se  $n$  for primo e 0 se  $n$  não for primo.
  - `main ( )` Percorre os inteiros, testando se cada um deles é primo. Imprime os 100 primeiros números primos.
- Teste seus programas executando-os no SPIM.

**A.10** [10] <§§A.6, A.9> Usando o SPIM, escreva e teste um programa recursivo para solucionar um problema matemático clássico, denominado Torres de Hanói. (Isso exigirá o uso de frames de pilha para admitir a recursão.) O problema consiste em três pinos (1, 2 e 3) e  $n$  discos (o número  $n$  pode variar; os valores típicos poderiam estar no intervalo de 1 a 8). O disco 1 é menor que o disco 2, que, por sua vez, é menor que o disco 3, e assim por diante, com o disco  $n$  sendo o maior. Inicialmente, todos os discos estão no pino 1, começando com o disco  $n$  na parte inferior, o disco  $n - 1$  acima dele, e assim por diante, até o disco 1 no topo. O objetivo é mover todos os discos para o pino 2. Você só pode mover um disco de cada vez, ou seja, o disco superior de qualquer um dos três pinos para o topo de qualquer um dos outros dois pinos. Além do mais, existe uma restrição: você não pode colocar um disco maior em cima de um disco menor.

O programa em C a seguir pode ser usado como uma base para a escrita do seu programa em assembly:

```
/* move n discos menores de start para finish usando extra */

void hanoi(int n, int start, int finish, int extra){
    if(n != 0){
        hanoi(n-1, start, extra, finish);
        print_string("Move disco");
        print_int(n);
        print_string("do pino");
        print_int(start);
        print_string("para o pino");
        print_int(finish);
        print_string(".\n");
        hanoi(n-1, extra, finish, start);
    }
}

main( ){
    int n;
    print_string("Entre com o número de discos>");
    n = read_int( );
    hanoi(n, 1, 2, 3);
    return 0;
}
```