Capítulo 11

Programação em *Assembly*

When words are scarce they are seldom spent in vain.
William Shakespeare, Richard the Second, II,1.

No relatório intitulado *First Draft of a Report on the EDVAC*, publicado em 1945, John von Neumann definiu o que se entende por *computador com programa armazenado*. Nesse computador, a memória é nada mais, nada menos, do que um vetor de bits, e a interpretação dos bits é determinada pelo programador [vN93]. Este modelo tem algumas implicações interessantes. Senão, vejamos:

(i) o código fonte que você escreve na linguagem C é "um programa" ou é "dados"?

(ii) o código *assembly* produzido pelo montador a partir do seu código C é "um programa" ou é "dados"?

(iii) o arquivo `a.out` produzido pelo ligador, a partir do código *assembly*, é "um programa" ou é "dados"?

A resposta, segundo o modelo de von Neumann, para as três perguntas é *dados*.

Como é que é?

É exatamente o que você acaba de ler. O código C é mantido num arquivo texto, que é a entrada para o compilador, portanto dados. O código *assembly* é mantido num arquivo texto, que é a entrada para o montador, logo, dados. O código binário mantido no arquivo `a.out` está em formato de executável, mas é apenas "um arquivo cheio de bits". Esse arquivo só se transforma em "um programa" no instante em que for carregado em memória e estiver prestes a ser executado pelo processador.

Uma vez que o programa esteja carregado em memória e a execução se inicia, quais seriam as fases de execução de uma instrução? A Figura 11.1 mostra um diagrama de blocos de um computador com programa armazenado. Por razões que serão enunciadas adiante[1], a memória do nosso computador é dividida em duas partes: (i) uma memória de instruções. e (ii) uma memória de dados.

Entre as duas memórias está o processador. Um registrador chamado de `PC`, ou *Program Counter*, mantém o endereço da próxima instrução[2] que será executada. O registrador `instr`

[1] © Roberto André Hexsel, 2012-2021.

[2] Esse registrador é chamado de *instruction pointer* na documentação dos processadores da Intel, nome que melhor descreve a função deste registrador.

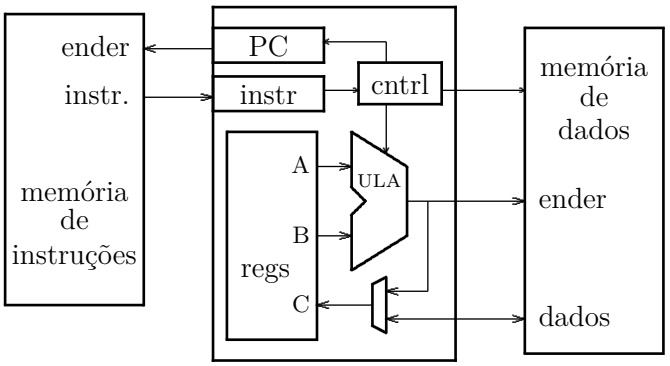


Figura 11.1: Modelo de computador com programa armazenado.

mantém a *instrução corrente*, que está sendo executada. O bloco de registradores (`regs`) contém 32 registradores que armazenam os valores temporários das variáveis do programa que está a executar. A unidade de lógica e aritmética (ULA) efetua a operação determinada pela instrução corrente sobre os conteúdos de dois registradores e armazena o resultado num terceiro registrador.

O circuito que controla o funcionamento interno do processador sequencia a execução de cada instrução em quatro fases:

1. indexado pelo PC, busca na memória uma instrução, que é a "instrução corrente";
2. decodifica a instrução corrente;
3. executa a operação definida pela instrução; e
4. armazena o resultado da operação e retorna para 1.

O Capítulo 12 apresenta os detalhes sobre cada uma das fases. A Figura 11.2 indica as quatro fases da execução de uma instrução de adição. No topo da figura está a instrução **`add`** com seus três operandos, `r3` que é o registrador de resultado, `r1` e `r2` que são os registradores com as duas parcelas por somar. O caractere '#' é o indicador de comentário que reza: "ao registrador `r3` é atribuída a soma dos conteúdos de `r1` e `r2`".

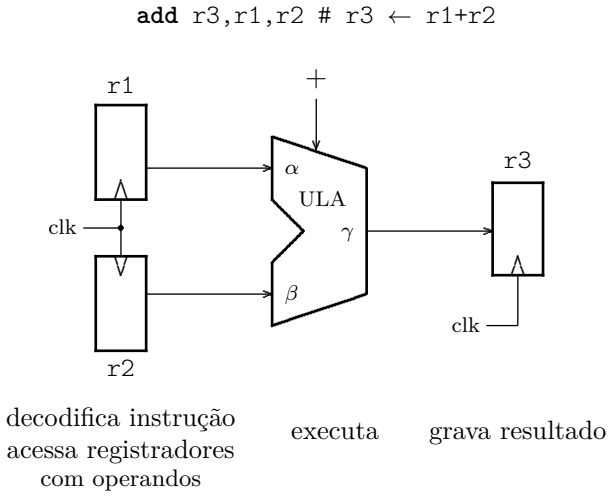


Figura 11.2: Acesso aos registradores, execução e resultado da instrução `add`.

Os registradores `r1`, `r2` e `r3` pertencem ao banco de registradores – `regs` na Figura 11.1, e a cunha indica a ligação ao sinal de relógio. O trapézio chanfrado é a unidade de lógica e aritmética do processador.

Durante a decodificação da instrução, o circuito de controle seleciona a operação da ULA, que é uma soma neste exemplo, e os conteúdos dos registradores `r1` e `r2` são apresentados às suas entradas α e β. Durante a execução os sinais se propagam através da ULA, e na fase de resultado, a soma é armazenada no registrador de destino, `r3`. Isso feito, o ciclo se repete e uma nova instrução é buscada.

O conjunto de instruções de um certo processador determina quais operações aquele processador é capaz de efetuar, bem como os tamanhos dos operandos das operações. Como um mínimo, as operações de aritmética e lógica são suportadas – sobre operandos de 8, 16, 32 ou 64 bits – além de instruções para acesso à memória, mais instruções para a tomada de decisões.

O processador é um circuito sequencial que implementa todas as operações do conjunto de instruções. Diz-se então que *o conjunto de intruções é a* especificação *do processador*, e ainda que *um determinado processador* é uma implementação *de seu conjunto de instruções*. Alternativamente, a *arquitetura* de um processador é definida pelo seu conjunto de instruções, seus modos de endereçamento e seu mecanismo de interrupções, a implementação da arquitetura é chamada de *microarquitetura*.

No que se segue, o conjunto de instruções dos processadores MIPS de 32 bits é definido, ao mesmo tempo em que introduzimos a programação na linguagem *assembly* MIPS32, revisão 2.

11.1 A linguagem de montagem MIPS32r2

A cada processador é associado o seu conjunto de instruções, e portanto uma linguagem de montagem que lhe é exclusiva. As instruções do MIPS são diferentes das instruções dos processadores da família x86, tanto em termos de formato, quanto em termos da funcionalidade de cada instrução. Em geral, não há portabilidade entre linguagens de montagem, embora elas possam ser similares entre si. Nossa referência para o conjunto de instruções é o documento publicado pela própria MIPS [MIP05b].

Nos interessa a linguagem *assembly* do MIPS. Essa linguagem é extremamente simples, e um programa montador simplificado, porém completo, que traduz *assembly* para binário pode ser escrito em algo como 200 linhas de código C.

Estamos habituados a usar a forma infixada para as operações, tal como "$c \leftarrow (a + b)$". A notação empregada nas instruções de *assembly* é prefixada, e o exemplo anterior é grafado como **`add`** `rc,ra,rb`, com o significado "$c \leftarrow \ + (a, b)$". Nos comentários ao longo do código usamos a forma infixada. No que se segue, as instruções são grafadas em negrito.

Note que a "instrução **`add`**" tem três operandos, que são três registradores, enquanto a operação de soma tem dois operandos e um resultado. Na descrição da instrução **`add`**, 'operandos' são os operandos da instrução **`add`**, ao passo que dois deles são operandos da operação *soma* que é efetuada por aquela instrução.

Espaço em branco proposital.

Cada instrução é identificada pelo seu *opcode*, ou seu *operation code*, que é um padrão de bits exclusivo àquela instrução. Para facilitar a vida de quem programa, a linguagem *asembly* representa os *opcodes* com mnemônicos que são mais facilmente memorizáveis. Por exemplo, ao invés do padrão de 32 bits mostrado abaixo para representar a instrução de soma, `00000000010000110000100000100000`, empregamos o mnemônico `add r1, r2, r3`.

Uma instrução de lógica e aritmética como a instrução **`add`** é dividida em seis campos:

```
000000 00010 00011 00001 00000 100000
opcode  rs    rt    rd   shamt  func
```

O campo `opcode`, com 6 bits, identifica a instrução. Os três campos de 5 bits `rs` (*source*), `rt` (*target*), e `rd` (*destination*) são os três operandos da instrução, cada um com 5 bits. O campo `shamt` (*shift amount*) indica o número de posições por deslocar nas instruções de deslocamento, também com 5 bits. O campo `func` tem 6 bits e determina a função da ULA, que neste caso é uma soma. A codificação das instruções é retomada na Seção 12.1.1.

Programas em linguagem de montagem do MIPS podem fazer uso de três tamanhos, ou tipos, de dados: byte, meia-palavra (*half word*, com 16 bits) e palavra (*word*, com 32 bits). As 'variáveis' dos programas podem ser armazenadas em dois conjuntos de variáveis: nos 32 registradores do processador, e num vetor de bytes que é a própria memória do computador. Estritamente falando, estes conjuntos disjuntos são dois *espaços de nomes*.

Um programa em *assembly* é editado num arquivo texto, com uma instrução por linha. Cada linha pode conter até três campos, todos opcionais: um *label*, uma instrução, e um comentário. O Programa 11.1 mostra um trecho com três linhas de código *assembly*. Um *label* é um nome simbólico que representa o endereço da instrução associada; o *label* `.L1:` representa o endereço da instrução **`add`** – note o caractere ':', que indica que o nome `.L1` é um *label* e não uma instrução. Um comentário inicia no caractere '`#`' e se estende até o final da linha. Nos comentários, a atribuição é denotada por '`<-`'.

Programa 11.1: Exemplo de código *assembly*.

```
.L1:    add r1, r2, r3    # r1 <- r2 + r3
        sub r5, r6, r7    # r5 <- r6 - r7
fim:    j   .L1           # salta para endereço com .L1
```

Na década de 1940, uma grande parte da tarefa "escrever um programa" incluía "montar os códigos de operação" – além de determinar a sequência de instruções do programa, a programadora devia traduzir as instruções para binário, que então eram gravadas diretamente na memória do computador.

O programa montador (*assembler*) traduz a *linguagem de montagem* (*assembly language*) para a *linguagem de máquina*, que é o código binário interpretado pelo processador. Além da tradução, montadores sofisticados provêm uma série de comodidades ao programador, como veremos no que se segue.

Vejamos três exemplos simples de tradução de código C para *assembly*. Nestes exemplos usaremos a convenção de chamar os operandos das instruções com os nomes das variáveis, prefixados pela letra 'r', de registrador.

C	*assembly*
`a = b + c;`	`add ra, rb, rc   # ra <- rb + rc`

Há uma correspondência direta entre o comando (simples) em C e sua tradução para *assembly*. Nosso segundo exemplo tira proveito da associatividade da soma. O resultado intermediário é acumulado no registrador `ra`.

C	*assembly*

`a = b + c + d + e;`	`add ra, rb, rc  # ra <- rb + rc`
	`add ra, ra, rd  # ra <- ra + rd`
	`add ra, ra, re  # ra <- ra + re`

No terceiro exemplo podemos tirar proveito da associatividade, ou então escrever código ligeiramente mais complexo para demonstrar o uso de dois registradores temporários (`t0` e `t1`), para a computação de expressões que não sejam triviais.

C	*assembly*

`f = (g+h) - (i+j);`	`add t0, rg, rh  # t0 <- rg + rh`
	`add t1, ri, rj  # t1 <- ri + rj`
	`sub rf, t0, t1  # rf <- t0 - t1`

Nos processadores MIPS de 32 bits as instruções de lógica e aritmética sempre têm 3 operandos, e estes são *sempre* registradores. Essa escolha de projeto simplifica o circuito que decodifica as instruções, como discutido na Seção 12.2.1. A largura de todos os circuitos de dados no MIPS é 32 bits: os registradores, a unidade de lógica e aritmética, as interfaces com a memória, o PC e os circuitos de controle de fluxo de execução.

O MIPS tem 32 *registradores visíveis*, que chamaremos de `r0` a `r31` por uma questão de gosto do autor, embora o programa montador `gas`[3] os chame de `$0` a `$31`.

O último exemplo, agora empregando números dos registradores ao invés de nomes de variáveis, mostra a linguagem que é aceita pelo montador `as`, com as variáveis `f..j` alocadas aos registradores `$16..$20`.

C	*assembly*

`f = (g+h) - (i+j);`	`add $8, $17, $18`
	`add $9, $19, $20  # $8=t1, $9=t2`
	`sub $16, $8, $9`

O registrador `$0` (`r0` ou `$zero`) retorna sempre o valor zero, e escritas neste registrador não tem qualquer efeito. Este registrador permite usar uma constante que é assaz popular. Por uma convenção da linguagem *assembly*, o registrador `r1` (ou `$1`) é usado como uma variável temporária para montador, e não deve ser usado nos programas em *assembly*.

11.1.1 Instruções de lógica e aritmética

A representação para números inteiros usada na linguagem de montagem do MIPS é o complemento de dois. Operações com inteiros podem ter operandos positivos e negativos, e talvez, dependendo da aplicação, um programa deva detectar a ocorrência de *overflow*, quando a soma de dois números de 32 bits produz um resultado que só pode ser representado corretamente em 33 bits.

[3] Tipicamente, os montadores são chamados de `as` (*assembler*). A versão em *software* livre de nossa preferência se chama *GNU assembler*, logo o nome do executável é `gas`. Em geral, emprega-se um apelido de tal forma que `gas` é chamado de `as`.

A linguagem C ignora solenemente a ocorrência de *overflow*, enquanto linguagens como C++ e Java permitem ao programador tratar o evento de acordo com as especifidades de cada aplicação. No caso de C, a programadora é responsável por detectar a ocorrência de *overflow* e tomar a ação corretiva que seja conveniente.

No MIPS as instruções de aritmética tem duas variedades, a variedade *signed* sinaliza a ocorrência de *overflow* com uma *excessão*, enquanto que a a variedade *unsigned* os ignora. Estes nomes são uma escolha infeliz porque ambíguos: *todas* as instruções operam com números em complemento de dois, portanto números inteiros com sinal. O que *signed* e *unsigned* indicam é a forma de tratamento da ocorrência de eventuais resultados errados por causa de *overflow*. O mecanismo de excessões do MIPS é apresentado na Seção **??**.

Instruções *unsigned* são empregadas no cálculo de endereços porque operações com endereços são sempre sem-sinal – todos os 32 bits compõem o endereço. Por exemplo, quando representa um endereço, `0x8000.0000` é um endereço válido e não o maior número negativo.

Vejamos mais algumas instruções de lógica e aritmética. Para descrever o efeito das instruções, ou sua *semântica*, empregaremos uma notação que é similar a VHDL: '&' denota a concatenação, `(x:y)` denota um vetor de bits com índices `x` e `y`.

```
add   r1, r2, r3        # r1 <- r2 + r3
addu  r1, r2, r3        # r1 <- r2 + r3
addi  r1, r2, const16   # r1 <- r2 + extSin(const16)
addiu r1, r2, const16   # r1 <- r2 + extSin(const16)
```

Como já vimos, as instruções **`add`** e **`addu`** somam o conteúdo de dois operandos e gravam o resultado no primeiro operando. As instruções **`addi`** e **`addiu`** somam o conteúdo de um registrador ao conteúdo de uma constante de 16 bits, que é estendida para 32 bits, com a operação `extSin()`. As instruções *com* sufixo `u` são ditas *unsigned* e não sinalizam a ocorrência de *overflow*; as instruções *sem* o sufixo `u` são ditas *signed* e sinalizam a ocorrência de *overflow*.

Por que estender o sinal? É necessário estender o sinal para transformar um número de 16 bits, representado em complemento de dois, em número de 32 bits? A extensão é necessária para garantir que o número de 32 bits mantenha a mesma magnitude e sinal. Considere as representações em binário, em 16 e em 32 bits para os números $+4$ e -4. Nas representações em binário, o bit de sinal do número em 16 bits está indicado em negrito, bem como o bit de sinal para a representação em 32 bits.

$+4 =$ `0b`**`0`**`000.0000.`**`0`**`000.0100 = 0x0004` $\rightsquigarrow$ `0x0000.0004`

$-4 =$ `0b`**`1`**`111.1111.`**`1`**`111.1100 = 0xfffc` $\rightsquigarrow$ `0xffff.fffc`

Vejamos algumas das instruções de lógica. As instruções **`and`**, **`or`**, **`nor`**, e **`xor`** efetuam a operação lógica Φ sobre pares de bits, um bit de cada registrador, e o resultado é atribuído ao bit correspondente do registrador de destino, como especifica a Equação 11.1.

$$\Phi\ r1, r2, r3 \quad \equiv \quad r1_i \leftarrow r2_i\ \Phi\ r3_i, \quad i \in [0, 31] \tag{11.1}$$

As instruções **`andi`**, **`ori`** e **`xori`** efetuam a operação lógica sobre o conteúdo de um registrador e da constante de 16 bits estendida com zeros (`extZero()`), e não com o sinal. Ao contrário das instruções de aritmética, quando se efetua operações lógicas, o que se deseja representar é uma constante lógica e não uma constante numérica positiva ou negativa.

A instrução **`not`** produz o complemento do seu operando `r2`.

A instrução **sll** (*shift left logical*) desloca seu segundo operando do número de posições indicadas no terceiro operando, que pode ser um registrador, ou uma constante de 5 bits; no primeiro caso, somente os 5 bits menos significativos são considerados.

As instruções **srl** (*shift right logical*) e **sra** (*shift right arithmetic*) deslocam seu segundo operando para a direita, de forma similar ao **sll**, exceto que a **sra** replica o sinal do número deslocado. A Tabela 11.1 define as instruções de lógica e aritmética.

Tabela 11.1: Instruções de lógica e aritmética.

```
add   r1, r2, r3        # r1 <- r2 + r3                    [1]
addi  r1, r2, const16   # r1 <- r2 + extSin(const16)       [1]
sub   r1, r2, r3        # r1 <- r2 - r3                    [1]

addu  r1, r2, r3        # r1 <- r2 + r3                    [2]
addiu r1, r2, const16   # r1 <- r2 + extSin(const16)       [2]
subu  r1, r2, r3        # r1 <- r2 - r3                    [2]

and   r1, r2, r3        # r1 <- r2 AND r3
or    r1, r2, r3        # r1 <- r2 OR r3
not   r1, r2            # r1 <- NOT(r2)
nor   r1, r2, r3        # r1 <- NOT(r2 OR r3)
xor   r1, r2, r3        # r1 <- r2 XOR r3

andi r1, r2, const16    # r1 <- r2 AND extZero(const16)
ori  r1, r2, const16    # r1 <- r2 OR  extZero(const16)
xori r1, r2, const16    # r1 <- r2 XOR extZero(const16)

sll   r1, r2, r3        # r1 <- (r2 << r3(4..0))
sll   r1, r2, const5    # r1 <- (r2 << const5)

srl   r1, r2, r3        # r1 <- (r2 >> r3(4..0))
srl   r1, r2, const5    # r1 <- (r2 >> const5)

sra   r1, r2, r3        # r1 <- (r2 >> r3(4..0))           [3]
sra   r1, r2, const5    # r1 <- (r2 >> const5)             [3]

lui   r1, const16       # r1 <- const16 & 0x0000

la    r1, const32       # r1 <- const32                    [4]
li    r1, const16       # r1 <- 0x0000 & const16           [4]
```

[1] sinaliza ocorrência de *overflow*, [2] ignora ocorrência de *overflow*,
[3] replica sinal, [4] pseudoinstrução.

Já sabemos como trabalhar com constantes de até 16 bits. Como se faz para obter constantes em 32 bits? São necessárias duas instruções: a instrução **lui** (*load upper immediate*) carrega uma constante de 16 bits na parte mais significativa de um registrador e preenche os 16 bits menos significativos com zeros.

```
lui r1, const16      # r1 <- const16 & 0x0000
```

Espaço em branco proposital.

Combinando-se **lui** com **ori**, é possível atribuir uma constante de 32 bits a um registrador:

```
lui r1, 0x0080       # r1 <- 0x0080 & 0x0000 = 0x0080.0000
ori r1, r1, 0x4000   # r1 <- 0x0080.0000 OR 0x0000.4000
                     #    <- 0x0080.4000
```

Essa operação é usada frequentemente para efetuar o acesso à estruturas de dados em memória – o endereço da estrutura deve ser atribuído a um registrador que então aponta para seu endereço inicial. Por causa da popularidade dessa concatenação, o montador nos oferece a *pseudoinstrução* **la** (*load address*) que é um apelido para o par **lui** seguido de **ori**.

```
la  r1, 0x0080.4000  # r1 <- 0x0080.4000
```

Quando o endereço é um *label*, ao invés de uma constante numérica, o montador faz uso dos operadores `%hi()` e `%lo()` para extrair as partes mais e menos significativas do seu operando. Suponha que o endereço associado ao *label* `.L1` seja `0x0420.3610`, então

```
la r1, .L1
```

é expandido para

```
lui r1, %hi(.L1)     # r1 <- 0x0420 & 0000
ori r1, r1, %lo(.L1) # r1 <- 0x0420.0000 OR 0x0000.3610
```

A pseudoinstrução **li** (*load immediate*) é usada para carregar constantes que não são endereços. Se a constante pode ser representada em 16 bits, **li** é expandida para **ori** ou **addi**, dependendo se a constante é um inteiro ou uma constante lógica. Se a constante é maior do que ±32K, então **li** é expandida com o par **lui**;**ori**.

11.1.2 Acesso à variáveis em memória

Até agora, empregamos somente registradores para manter os valores intermediários em nossas computações. Programas realistas usam um número de variáveis maior do que os 32 registradores – variáveis, vetores e estruturas de dados são alocados em memória. No MIPS, os operandos de todas as instruções que operam sobre dados são registradores, e por isso os operandos devem ser trazidos da memória para que sejam utilizados.

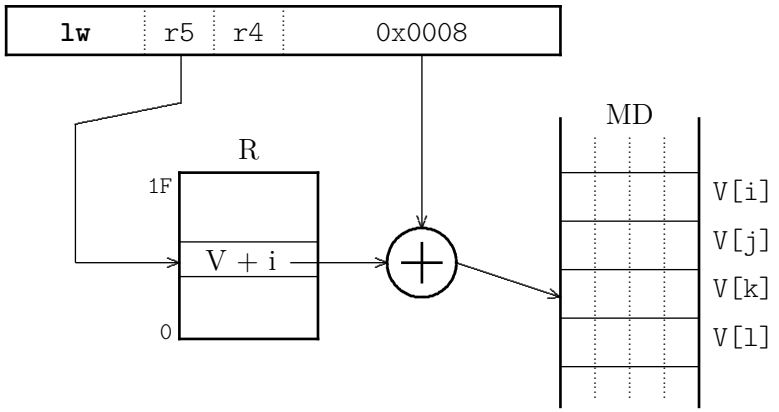
O modelo de memória do MIPS é um vetor com tipo `byte`: `M`[2^{32}], com 4Gbytes de capacidade. Um *endereço em memória* é o índice **i** do vetor `M`[**i**].

Bytes são armazenados em endereços consecutivos do vetor `M[ ]`, enquanto que palavras de 32 bits são armazenadas em endereços múltiplos de 4 – a memória acomoda 2^{30} palavras. Para simplificar a interface do processador com a memória, as referências devem ocorrer para *endereços alinhados*. Uma referência a um inteiro (uma palavra) deve empregar um endereço que é múltiplo de quatro, enquanto que uma referência a um `short` (*half word*) deve empregar um endereço par. Um `long long` (*double word*) deve ser referenciado num endereço múltiplo de 8. Referências a `char` são naturalmente alinhadas.

São de dois tipos as instruções para acessar a memória: *loads* e *stores*. Essas instruções existem em tamanho *word* (4 bytes), *half word* (2 bytes) e *byte*. Por enquanto, vejamos as de tamanho *word*. A instrução **lw** (*load word*) permite copiar o conteúdo de uma palavra em memória para um registrador. A instrução **sw** (*store word*) copia o conteúdo de um registrador para uma palavra em memória.

```
lw r1, desl16(r2)    # r1 <- M[ r2 + extSin(desl16) ]
sw r3, desl16(r4)    # M[ r4 + extSin(desl16) ] <- r3
```

O campo `desl16` é um inteiro representado em 16 bits. A soma do conteúdo do *registrador base* (`r2` e `r4`) com a constante estendida é chamada de *endereço efetivo*. O deslocamento pode ser negativo, quando o endereço efetivo é menor do que o apontado pelo registrador base, ou positivo, quando o endereço efetivo é maior do que o apontado pelo registrador base.




```
lw r4, 8(r5) # r4 ← M( r5 + 8 )
```

Figura 11.3: Cálculo do endereço para acessar o elemento `V[k]`.

A Figura 11.3 mostra a instrução que efetua um acesso de leitura no terceiro elemento de um vetor de inteiros, indicada abaixo em C e em *assembly*. O endereço de um vetor, na linguagem C ou em *assembly*, é representado pelo nome do vetor, que é `V` no nosso exemplo. Esse mesmo endereço pode ser representado, verbosamente, por `&(V[0])`.

C	*assembly*
`i = V[k];`	`lw r4, 8(r5)`

O registrador `r5` aponta para o endereço inicial do vetor `V`. Ao conteúdo de `r5` é somado o deslocamento, que é $2 \times 4 = 8$, e o endereço efetivo é `&(V[0])+8`. Essa posição de memória é acessada e seu conteúdo é copiado para o registrador `r4`. O deslocamento com relação à base do vetor, apontado por `r5`, é de 8 bytes porque cada elemento do vetor ocupa quatro bytes consecutivos na memória.

11.1.3 Estruturas de dados em C: vetores e matrizes

Vejamos como acessar estruturas de dados em *assembly*. Antes de mais nada, recordemos os tamanhos das 'coisas' representáveis em C. 'Coisa' não chega a ser um termo técnico elegante, mas a palavra não é sobrecarregada como seria o caso da palavra 'objeto'. A função da linguagem C **`sizeof`**`(x)` retorna o número de bytes necessários para representar a 'coisa' `x`. A Tabela 11.2 indica o tamanho das 'coisas' básicas da linguagem C – aqui, por 'coisa' entenda-se os tipos básicos das variáveis e constantes representáveis em C.

Talvez o mais surpreendente seja a constatação de que ponteiros para caracteres e *strings*, para inteiros, para qualquer 'coisa' enfim, (**`char`**`*`, **`int`**`*`, **`void`**`*`, **`int`**`**`) são *endereços* que *sempre* tem o mesmo tamanho, que é de 32 bits no MIPS.

A Figura 11.4 mostra como seria a alocação em memória de três vetores, de tipos **`char`**, **`short`**, e **`int`**, a partir do endereço 20. Elementos contíguos de vetores e estruturas de dados são

Tabela 11.2: Tamanho das 'coisas' representáveis em C.

tipo de dado	`sizeof()`
`char`	1
`short`	2
`int`	4
`long long`	8
`float`	4
`double`	8
`char[12]`	12
`short[6]`	12
`int[3]`	12
`char *`	4
`short *`	4
`int *`	4
`void *`	4

alocados em endereços contíguos: `V[i+1]` é alocado no endereço seguinte a `V[i]`; o endereço do 'elemento seguinte' depende do tipo dos elementos do vetor `V`.

endereço	20	21	22	23	24	25	26	27
`char`	`c[0]`	`c[1]`	`c[2]`	`c[3]`	`c[4]`	`c[5]`	`c[6]`	`c[7]`
`short`	`s[0]`		`s[1]`		`s[2]`		`s[3]`	
`int`	`i[0]`				`i[1]`			

Figura 11.4: Endereços de vetores de tipo `char`, `short`, e `int` em C.

Quem programa em *assembly* fica responsável por gerenciar o acesso a *todas* as estruturas de dados. A programadora é responsável por acessar palavras de 4 em 4 bytes, elementos de vetores alocados em endereços que dependem do tipo dos elementos, elementos de vetores de estruturas em endereços que dependem do **`sizeof()`** dos elementos, e assim por diante. Ao programador *assembly* não é dado o luxo de empregar as abstrações providas por linguagens de alto nível tais como C.

Na linguagem C, uma matriz é alocada em memória como um vetor de vetores. Para elementos de tipo τ, linhas com κ colunas e λ linhas, o endereço do elemento de índices i, j é obtido com a Equação 11.2. A diagrama na Figura 11.5 indica a relação entre o endereço base da matriz ($M = \&(M[0][0])$), linhas e colunas de uma matriz de elementos do tipo τ.

$$\&(M[i][j]) = \&(M[0][0]) + |\tau|(\kappa \cdot i + j) \tag{11.2}$$

Espaço em branco proposital.

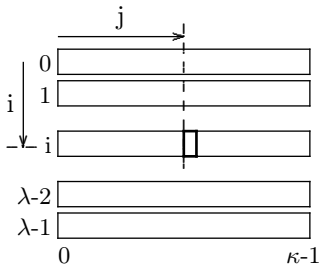


Figura 11.5: Endereço do elemento i, j, de tipo τ, numa matriz $\lambda \times \kappa$.

Exemplo 11.1 Considere o acesso ao vetor de inteiros `V`, mostrado no trecho de código abaixo e sua tradução para *assembly*. Ao registrador `r4` é atribuído o conteúdo de `V[1]` – note o deslocamento de 4 bytes para acessar o segundo inteiro – e ao registrador `r6` é atribuído o conteúdo de `V[2]` com deslocamento de 8 bytes. Ao registrador `r7` é atribuído o valor a ser armazenado no elemento zero do vetor `V`. Os comentários mostram o cálculo do *endereço efetivo*: à base do vetor (em `r1`) é adicionado o deslocamento de `i*4`, para índice `i`.

C

```
int V[NNN];
...
V[0] = V[1] + V[2]*16;
```

assembly

```
la  r1, V      # r1 <- &(V[0])
lw  r4, 4(r1)  # r4 <- M[r1+1*4]
lw  r6, 8(r1)  # r6 <- M[r1+2*4]
sll r6, r6, 4  # r6*16 = r6<<4
add r7, r4, r6
sw  r7, 0(r1)  # M[r1+0*4] <- r4+r6
```

O código deste exemplo tem uma característica importante: em tempo de compilação – quando o compilador examina o código – é possível determinar sem ambiguidade os deslocamentos com relação à base do vetor, e é por isso que o código emprega deslocamentos fixos nas instruções **`lw`** e **`sw`**. A Figura 11.6 indica os deslocamentos com relação à base do vetor, que é `&(V[0])`. ◁

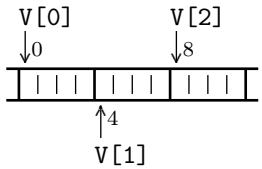


Figura 11.6: Deslocamento com relação à base de `V` dos elementos 0, 1, e 2.

Exemplo 11.2 O trecho de código abaixo é praticamente o mesmo do Exemplo 11.1, exceto que os índices são variáveis, e não constantes – no exemplo anterior, os deslocamentos estão fixados em tempo de compilação, e portanto a geração do código pode usar a informação quanto aos deslocamentos fixados no código. Neste caso, o código deve computar o endereço efetivo em função dos índices que variam ao longo da execução.

Para facilitar a leitura, as variáveis `i`,`j`,`k` são armazenadas nos registradores `ri`,`rj`,`rk`. A multiplicação por 16 é obtida pelo deslocamento de 4 posições para a esquerda, com a instrução `sll r6,r6,4`.

No código deste exemplo, os índices são variáveis e portanto os deslocamentos com relação à base do vetor devem ser computados explicitamente: ao endereço base do vetor é somado o índice do elemento multiplicado por 4, porque cada inteiro ocupa 4 bytes.

C

```
int V[NNN];
...
V[i] = V[j] + V[k]*16;
```

assembly

```
la   r1, V       # r1 <- &(V[0])
sll  r2, rj, 2   # r2 <- j * 4
addu r3, r2, r1  # r3 <- V + j*4
lw   r4, 0(r3)   # r4 <- M[V + j*4]
sll  r2, rk, 2   # r2 <- k * 4
addu r3, r2, r1  # r3 <- V + k*4
lw   r6, 0(r3)   # r6 <- M[V + k*4]
sll  r6, r6, 4   # r6 <- r6*16
add  r7, r4, r6
sll  r2, ri, 2   # r2 <- i * 4
addu r3, r2, r1  # r3 <- V + i*4
sw   r7, 0(r3)   # M[V + i*4] <- r7
```

O próximo exemplo mostra o código para acessar uma estrutura de dados mais complexa do que um vetor de inteiros. ◁

Espaço em branco proposital.

Exemplo 11.3 Considere a estrutura `aType`, com 4 elementos inteiros, `x,y,z,w`, e o vetor `V`, com 16 elementos do tipo `aType`. Como **`sizeof`**`(aType)`=16, o deslocamento de `V[3]` com relação à base do vetor é:

3 elementos x 4 palavras/elemento x 4 bytes/palavra = 48 bytes = 0x30 bytes.

Para simplificar o exemplo, suponha que o vetor `V` foi alocado no endereço `0x0080.0000`. Note que o código *assembly* está otimizado para o índice que é a constante 3.

C

```
typedef struct A {
   int x;
   int y;
   int z;
   int w;
} aType;
...
aType V[16];
...
   m = V[3].y;
   n = V[3].w;
   V[3].x = m+n;
```

assembly

```
la   r5, 0x00800000   # r5 <- &(V[0])
lw   r8, (48+4)(r5)   # r8 <- V[3].y
lw   r9, (48+12)(r5)  # r9 <- V[3].w
add  r5, r8, r9
sw   r5, (48+0)(r5)   # V[3].x <- m+n
```

A Figura 11.7 indica os deslocamentos com relação à base do vetor de estruturas `aType`, que é `&(V[0])`. Lembre que cada elemento ocupa 4 inteiros, ou 16 bytes. ◁

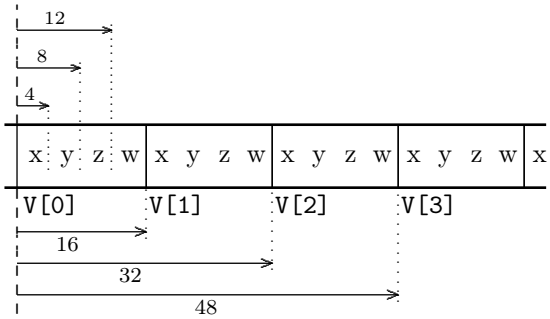


Figura 11.7: Deslocamento com relação à `V[0]` dos elementos 0, 1, e 2, do tipo `aType`.

Exemplo 11.4 Vejamos um exemplo com indexação indireta de um vetor. O conteúdo da i-ésima posição do vetor `X` é usado para indexar o vetor `Y`.

```
int a, i;
int X[2048], Y[256];
...
a = a + X[i] + Y[ (X[i] % 256) ]; // MOD
```

O resto da divisão inteira é obtido com uma divisão, que é uma operação custosa. Ao invés da divisão pode-se usar o seguinte truque: se $P = 2^k,\ k > 1$, então $n \,\%\, P = n \wedge (P-1)$. Para $P = 16$, temos que $n \,\%\, 16 \in [0, 15]$, e $16 - 1 = 15 = 1111_2$. A conjunção de qualquer número com 15 resulta num número que é, no máximo, 15.

```
la   rx,X             # rx <- &(X[0])
la   ry,Y             # ry <- &(Y[0])

sll  t1, ri, 2        # i*4
add  t2, t1, rx       # X + i*4
lw   t3, 0(t2)        # t3 <- X[i]
andi t4, t3, (256-1)  # t3 % 256 = t3 AND 0x0ff
sll  t4, t4, 2        # (t4 % 256)*4
add  t4, t4, ry       # Y + (t4 % 256)*4
lw   t5, 0(t4)        # t5 <- Y[ X[i] ]

add  t6, t5, t3       # X[i] + Y[ X[i] ]
add  ra, ra, t6       # a = a + X[i] + Y[ X[i] ]
```

O cálculo do índice módulo 256 é usado para garantir que, no acesso a `Y`, o índice não extrapola o espaço alocado àquele vetor. ◁

As instruções de movimentação de dados entre registradores e memória são mostradas na Tabela 11.3. As instruções **`lw`** e **`sw`** foram definidas nos parágrafos anteriores. Nesta tabela, por questões de espaço, a função `extSin()` é a função é indicada como `eS()`.

As instruções **`lh`** e **`lhu`** copiam para o registrador de destino os 16 bits apontados pelo endereço efetivo. A instrução **`lh`** (*load half*) estende o sinal do valor lido da memória, enquanto que a instrução **`lhu`** (*load half unsigned*) estende o valor lido para 32 bits com 16 zeros. O mesmo se aplica às instruções **`lb`** (*load byte*) e **`lbu`** (*load byte unsigned*), exceto que para acessos a 8 bits.

Tabela 11.3: Instruções para acesso à memória.

```
lw  r1, des16(r2) # r1 <- M[r2 + eS(des16)]

lh  r1, des16(r2) # r1 <- eS(M[r2 + eS(des16)](15..0))
lhu r1, des16(r2) # r1 <- 0x0000 & M[r2 + eS(des16)](15..0)

lb  r1, des16(r2) # r1 <- eS(M[r2 + eS(des16)](7..0))
lbu r1, des16(r2) # r1 <- 0x000000 & M[r2 + eS(des16)](7..0)

sw  r1, des16(r2) # M[r2 + eS(des16)] <- r1
sh  r1, des16(r2) # M[r2 + eS(des16)](15..0) <- r1(15..0)
sb  r1, des16(r2) # M[r2 + eS(des16)](7..0)  <- r1(7..0)
```

A instrução **sh** (*store half*) copia os 16 bits menos significativos do seu operando para os dois bytes apontados pelo endereço efetivo. A instrução **sb** (*store byte*) copia para a memória o byte menos significativo do seu operando. Os deslocamentos nos acessos à palavra (**lw**, **sw**), e nos acessos à meia-palavra (**lh**, **sh**) devem ser alinhados de 4 e 2 bytes, respectivamente. Acessos à bytes são naturalmente alinhados.

Exercícios

Ex. 11.1 Traduza para *assembly* do MIPS o trecho de código C abaixo. A constante `NN` é suficientemente grande.

```
int P[NN]; int Q[NN]; int V[NN];
int i,j,k;

V[4] = 5;
V[k] = 5;

V[ V[4] ] = 5;
V[ V[m] ] = 5;

V[ V[4] ] = V[ V[3] ];
V[ V[p] ] = V[ V[q] ];
```

11.1.4 Controle de fluxo de execução

Com o que vimos até o momento temos condições de escrever programas que acessam a memória e efetuam operações de lógica e aritmética com variáveis que foram copiadas para registradores. Isso é muito interessante, mas nos faltam as instruções que permitem decidir se uma determinada operação será ou não efetuada sobre uma dada variável. As instruções que nos permitem incluir a tomada de decisões no código são os *desvios*.

Vejamos um exemplo de decisão, no trecho de programa em C, mostrado abaixo. Dependendo dos valores em `a` e em `b`, ou ocorre uma atribuição à `c`, ou uma atribuição à `f`; qualquer das duas é seguida da atribuição à `j`.

```
if ( a == b ) {
   c = d + e;
} else {
   f = g - h;
}
j = k * l;
```

As sequências podem ser

```
  c = d + e ; j = k * l;
```

ou

```
  f = g - h ; j = k * l;
```

O compilador gera o código tal que somente uma das cláusulas do **if**() seja executada. Ao programar em *assembly*, a sua tarefa é garantir que o fluxo de execução que é definido no código seja obedecido pela versão traduzida.

Desvios condicionais

O *fluxo sequencial de execução* é determinado pela próxima instrução, que é aquela em PC+4, com relação à instrução corrente. Tomar um desvio significa desviar do fluxo sequencial, e saltar sobre uma ou mais instruções. São duas as instruções que nos permitem avaliar uma condição e então desviar do fluxo sequencial: **`beq`** e **`bne`**, ou *branch if equal*, e *branch if not equal*.

Estas instruções nos permitem codificar os comandos de C **`if()`**, **`while()`** e **`for(;;)`**, por exemplo. Vejamos pois como se dão os desvios da conduta sequencial. O *endereço de destino* é o endereço da instrução para onde o fluxo de execução do programa deve desviar.

As instruções de desvio comparam o conteúdo de dois registradores e desviam para o endereço de destino se os conteúdos são iguais – **`beq`** – ou diferentes – **`bne`**. Se é para desviar, então PC ← `dest` e a execução prossegue daquela instrução; senão PC ← PC+4 e a instrução após o desvio é executada. Esta não é uma descrição exata dos desvios; sanearemos esta falha em breve.

```
beq r1, r2, dest # branch if equal: desvia se r1 == r2
bne r1, r2, dest # branch if not equal;: desvia se r1 != r2
```

O trecho de código abaixo mostra um desvio, e todas as instruções estão enumeradas para facilitar a descrição. Se os conteúdos de `r1` e `r2` são iguais, a próxima instrução é o **`sw`** em `L5`, do contrário, é o **`add`** em `L2`. A programadora identifica o destino do desvio, que é a instrução que deve ser executada caso a condição do teste seja verdadeira, e o montador traduz o símbolo `L5` para um endereço, e este endereço é o terceiro operando da instrução de desvio.

```
L1:  beq r1, r2, L5 # salta para L5 de r1==r2
L2:  add r5, r6, r7
L3:  sub r8, r9, r10
L4:  xor r11, r12, r13
L5:  sw  r14, 0(r15)
```

Os 'nomes' atribuídos às instruções no exemplo acima, seus *labels*, servem justamente para nomear uma instrução ou uma variável. O *label* associa um *nome simbólico* ao endereço em que aparece. O nome simbólico é algo de relevante ao programador, enquanto que seu endereço é necessário ao processo de tradução de *assembly* para binário. Para o montador, um *label* é um símbolo que inicia por '.', '_' ou uma letra, é seguido de letras, dígitos, '.' ou '_', e terminado por ':'. Os símbolos `_start:`, `.L23:`, e `a_b_c:` são *labels* válidos; `23x:` e `add` são inválidos.

No MIPS, o destino de um desvio é o número de instruções por saltar, tomando por base a instrução seguinte ao desvio. O endereço de destino é determinado pelo *deslocamento* com relação ao PC+4, que é o endereço da instrução seguinte ao desvio. A razão para tal é apresentada na Seção 12.3.

A definição completa da instrução **`beq`** `r1, r2, desloc` é: se desvio for tomado, o *endereço de destino* é (PC+4) + `extSin(desloc)`×4; ou (PC+4) se o desvio não for tomado. A constante `desloc` é o número de instruções por saltar com relação a PC+4. O deslocamento é representado em 16 bits, em complemento de dois, o que possibilita desvios com distância de até 32K instruções para adiante (`desloc` > 0) ou até de 32K instruções para trás (`desloc` < 0). A multiplicação por quatro ajusta o deslocamento para saltar instruções ao invés de bytes.

Agora que sabemos computar o endereço de destino dos desvios, qual é o efeito das sequências abaixo? A instrução **nop**, ou *no-operation*, não tem efeito algum, mas é uma instrução muito mais útil do que seu despretensioso nome possa indicar. Lembre que o destino é o deslocamento, contado em instruções de 4 bytes, com relação PC+4, que é instrução imediatamente após o desvio.

```
L1: nop
L2: beq r1, r1, -1
L3: nop

L4: nop
L5: beq r1, r1, 0
L6: nop

L7: nop
L8: beq r1, r1, +1
L9: nop
```

No desvio em `L2`, o deslocamento de -1 com relação a `L3` é `L2` ((PC+4)$-$4) e este é um laço infinito. O desvio em `L5` salta zero instruções com relação a `L6` e portanto se comporta como mais um **nop** ((PC+4)+0). O desvio em `L8` pula +1 instrução com relação a `L9`, e desvia portanto para a instrução após `L9` ((PC+4)+4).

Comparação de magnitude

Comparações de igualdade são muito úteis mas comparar magnitude é igualmente necessário. A instrução **slt**, ou *set on less than*, compara a magnitude de seus segundo e terceiro operandos e atribui 1 ou 0 ao primeiro operando, dependendo do resultado da comparação. Aqui, como na linguagem C, 0 significa *falso* enquanto que 1 significa *verdadeiro*. O comando em C, que equivale à **slt** é a atribuição condicional:

```
rd = ( (r1 < r2) ? 1 : 0 );
```

A comparação de magnitude é efetuada com uma subtração; se o conteúdo de `r1` é menor do que o de `r2`, então o resultado deve ser negativo, e o bit de sinal do resultado é atribuído ao registrador `rd`.

São quatro as instruções de comparação de magnitude. **slt** pode gerar uma excessão de *overflow* caso a subtração dos operandos (`r1`$-$`r2`) produza resultado que não é representável em 32 bits. **sltu** não sinaliza a ocorrência de overflow. **slti** compara o conteúdo de um registrador com o valor de uma constante de 16 bits com o sinal estendido, e sinaliza a ocorrência de *overflow*. **sltiu** não sinaliza a ocorrência de *overflow*.

```
slt   rd, r1, r2       # rd <- (r1 < r2) ? 1 : 0
sltu  rd, r1, r2       # rd <- (r1 < r2) ? 1 : 0
slti  rd, r1, const16  # rd <- (r1 < extSin(const16)) ? 1 : 0
sltiu rd, r1, const16  # rd <- (r1 < extSin(const16)) ? 1 : 0
```

Espaço em branco proposital.

Combinando a comparação de magnitude com a comparação de igualdade, pode-se decidir em função da magnitude de operandos. São necessárias duas instruções, um **slt** e um desvio. O par de instruções abaixo equivale a **blt**, ou *branch on less than*. Lembre que r0 é sempre zero e representa *falso*.

```
slt r1, r2, r3   # r1 <- (r2 < r3)     TRUE=1, FALSE=0
bne r1, r0, dest # salta para dest se r1=TRUE != FALSE=r0
```

Esta sequência equivale a **bge**, ou *branch if greater or equal*.

```
slt r1, r2, r3   # r1 <- (r2 < r3)      TRUE=1, FALSE=0
beq r1, r0, dest # salta para dest se r1=FALSE == FALSE=r0
```

Saltos incondicionais

Os *saltos incondicionais* efetuam uma mudança no fluxo de execução incondicionalmente. O conjunto de instruções do MIPS nos oferece três saltos incondicionais: **j**, ou *jump*, **jr**, ou *jump register*, e **jal**, ou *jump and link*.

```
j    ender26       # PC <- ender26 & 00
jr   rt            # PC <- rt
jal  ender26       # PC <- ender26 & 00 , r31 <- PC+4
```

Nos interessa agora a instrução **j**, que atribui ao PC o endereço de uma instrução, quer dizer, o argumento de 26 bits é concatenado com dois zeros, representando portanto um endereço alinhado na fronteira de uma palavra. Com esta instrução pode-se efetuar saltos sobre 2^{26} instruções, ou aproximadamente, 6,4 milhões de linhas de código C.

> *Folclore: na média, uma linha de código C é traduzida para 10 instruções de assembly. Evidentemente, a proporção varia em função do processador e da linguagem, mas esta é uma boa estimativa para o tamanho do problema.*

Adiante, quando estudarmos a implementação de funções, veremos como se estende o alcance de um salto para 2^{30} instruções.

A Tabela 11.4 mostra as instruções de controle de fluxo. Nesta tabela, por questões de espaço, a função `extSin()` é a função é indicada como `eS()`.

Isso tudo posto, vejamos como traduzir código com saltos e desvios. Nestes exemplos os registradores são identificados com os nomes das variáveis.

Exemplo 11.5 Como traduzir um **if**() simples?

C	*assembly*
`if (a == b) goto Labl;`	`beq ra, rb, Labl`
`  c = d + e;`	`add rc, rd, re`
`Labl: f = g - h;`	`Labl: sub rf, rg, rh`

Se os conteúdos de a e b são iguais, então somente a subtração é efetuada; do contrário, ambas, soma e a subtração, são efetuadas. O **goto** é usado neste exemplo somente para aproximar o C do *assembly* – esta construção só é adequada quando se deseja escrever código indecifrável. ◁

Tabela 11.4: Instruções para saltos e desvios.

```
slt   rd, r1, r2      # rd <- (r1 < r2 ? 1 : 0)           [1]
slti  rd, r1, cnst16  # rd <- (r1 < eS(cnst16) ? 1 : 0)   [1]

sltu  rd, r1, r2      # rd <- (r1 < r2 ? 1 : 0)           [2]
sltiu rd, r1, cnst16  # rd <- (r1 < eS(cnst16) ? 1 : 0)   [2]

beq r1, r2, dst16     # PC <- PC+4+(r1==r2 ? eS(dst16) : 0)
bne r1, r2, dst16     # PC <- PC+4+(r1!=r2 ? eS(dst16) : 0)

j   ender26           # PC <- ender26 & 00
jr  rt                # PC <- rt
jal ender26           # PC <- ender26 & 00 , r31 <- PC+4
```

[1] sinaliza ocorrência de *overflow*, [2] ignora ocorrência de *overflow*.

Exemplo 11.6 Aqui as cláusulas do `if()` são mutuamente exclusivas: *ou* a cláusula do `if` é executada; *ou* a cláusula do `else` é executada, mas nunca as duas em sequência.

C

```
if (a == b)
   c = d + e;
else
   f = g - h;
```

assembly

```
      bne ra, rb, Else
If:   add rc, rd, re
      j Exit          # salta else
Else: sub rf, rg, rh
Exit: nop
```

A função do *jump* é justamente saltar sobre as instruções da cláusula `else`. ◁

Exemplo 11.7 Este exemplo é um tanto mais interessante. O laço procura o índice do primeiro elemento de um vetor que é diferente de `c`. Supondo que os elementos do vetor sejam inteiros, o índice é multiplicado por quatro (`r9 <- ri*4`) e então adicionado à base do vetor `vet`. Recorde que `vet = &(vet[0])`.

C

```
while (vet[i] == c)
   i = i + j;
```

assembly

```
   la  r7, vet      # r7 <- vet
L: sll r9, ri, 2    # r9 <- i*4
   add r9, r7, r9   # r9 <- i*4+vet
   lw  r8, 0(r9)    # r8 <- M[ r9 ]
   bne r8, rc, End  # vet[i] != c ?
   add ri, ri, rj   # i <- i + j
   j   L            # repete
End: nop
```

O valor lido da memória é comparado com `c`, o que causa a terminação do laço com o `bne`, ou sua continuação (`j L`). ◁

Exemplo 11.8 Vejamos um laço que inclui o código do Exemplo 11.4.

```
int a, i;
int x[2048], y[256];
...
i=0; a=0;
while (i < 1024) {
   a = a + x[i] + y[ (x[i] % 256) ]; // MOD
   i = i + 1;
}
```

O teste do laço usa uma instrução `slti` para fazer a comparação de magnitude, que resulta em verdadeiro=1 ou falso=0, e o registrador `$zero` é usado na comparação com falso.

```
        la   rx, x            # rx <- &(x[0])
        la   ry, y            # ry <- &(y[0])
        li   ri, 0            # i <- 0
        li   ra, 0            # a <- 0

while:  slti t0, ri, 1024     # t0 <- (ri < 1024)
        beq  t0, $zero, fim   # t0 == FALSE -> fim
        sll  t1, ri, 2        # i*4
        add  t2, t1, rx       # x + i*4
        lw   t3, 0(t2)        # t3 <- x[i]
        andi t4, t3, (256-1)  # t3 % 256 = t3 AND 255
        sll  t4, t4, 2        # (t4 % 256)*4
        add  t4, t4, ry       # y + (t4 % 256)*4
        lw   t5, 0(t4)        # t5 <- y[ x[i] ]
        add  t6, t5, t3       # x[i] + y[ x[i] ]
        add  ra, ra, t6       # a = a + x[i] + y[ x[i] ]
        addi ri, ri, 1
        j    while

fim:    nop
```

Como o limite do laço é uma constante, o teste poderia estar no final do laço, o que eliminaria a instrução `j`, e tornaria o código mais eficiente. Se o limite fosse uma variável, então o teste deveria estar no topo do laço porque não é possível prever quantas voltas seriam executadas. ◁

Os exercícios contém outras construções com laços, e sua codificação e verificação com o simulador `MARS` é enfaticamente recomendada.

Espaço em branco proposital.

11.1.5 Estruturas de dados em C: cadeias de caracteres

Na linguagem C, cadeias de caracteres, ou *strings*, são vetores de caracteres terminados por `'\0'`. Uma *string* é um vetor do tipo **`char`**, de tamanho não definido, sendo o final da *string* sinalizado pelo caractere `'\0'`, que é `0x00`.

No código fonte, *strings* são representadas entre aspas duplas, enquanto que caracteres são representados entre aspas simples. Quando lemos código C, na *string* `"palavra"` não vemos o caractere o `'\0'`, mas ele ocupa o espaço necessário para sinalizar o fim da cadeia. Supondo que esta *string* seja alocada em memória a partir do endereço 0x400, o que é armazenado é o vetor mostrado na Figura 11.8. Quando se computa o tamanho de uma *string*, o `'\0'` deve ser contado porque ele ocupa espaço, embora seja invisível. Veja a Seção **??** para a codificação de texto e o alfabeto ASCII e suas extensões.

endereço	400	401	402	403	404	405	406	407
caractere	'p'	'a'	'l'	'a'	'v'	'r'	'a'	'\0'

Figura 11.8: Leiaute de uma *string* em memória.

Exemplo 11.9 O trecho de código no Programa 11.2 copia uma cadeia de caracteres, do vetor `fte`, para o vetor de caracteres `dst`, e no Programa 11.3 está a sua tradução para *assembly*.

A condição do laço contém uma leitura da memória (`fte[i]`), e no corpo do laço uma leitura em `fte` e uma escrita em `dst`. A segunda leitura é desnecessária porque o valor que é usado para testar a condição é o mesmo a ser usado na atribuição. A caractere `'\0'` não é atribuído ao destino no corpo do laço, e por isso é atribuído após o seu final.

Na tradução para *assembly* é necessário lembrar que cada elemento dos vetores ocupa um byte e portanto as instruções para acessar `fte` e `dst` devem ser **`lbu`** e **`sb`**. Os elementos dos vetores são caracteres representados em 8 bits – não são inteiros de 8 bits – e por isso a instrução *load byte unsigned* (**`lbu`**) é usada: quando o byte é carregado para o registrador de 32 bits, o valor é estendido com 24 zeros e não com o sinal (bit 7) do valor lido. Lembre que para representar a concatenação usamos o '&', do VHDL.

O registrador `r5` recebe o caractere de `fte[i]` e este é estendido com 24 zeros na esquerda. Quando o caractere `'\0'` é lido, ao registrador `r5` são atribuídos 32 zeros (24 & 8) e é por isso que o `'\0'` é comparado com `r0` no **`beq`** `r5,r0`. A instrução *store byte* (**`sb`**) escreve somente o byte menos significativo na memória e portanto não é necessário nenhum tipo de extensão.

O compilador, ou o montador, aloca o espaço necessário em memória para acomodar os vetores fonte e destino, e os endereços destas variáveis podem ser referenciados pela programadora, ao usar os nomes `fte` e `dst`.

No Programa 11.3, o índice `i` é incrementado dentro do laço, mas o endereço `&(dst[i])` *não* é computado explicitamente após o teste `fte[i] != '\0'`. O deslocamento de 1 no **`sb`** da última instrução tem o mesmo efeito que adicionar `r4` a `r19` após a saída do laço. ◁

Programa 11.2: Laço que copia uma *string* para um vetor de `char`.

```
char fte[16]="abcd-efgh-ijkl-"; // sizeof(fte)=16, mais '\0'
char dst[32];
int i;

i = 0;
while ( fte[i] != '\0' ) {       // terminou?
   dst[i] = fte[i];
   i = i + 1;
}
dst[i] = '\0';                   // atribui o '\0'
```

Programa 11.3: Versão em *assembly* do laço que copia uma *string*.

```
        la   r8, fte        # r8 <- fte
        la   r9, dst        # r9 <- dst
        add  r4, r0, r0     # i = 0;
                            # while ( fte[i] != '\0' ) {
lasso:  add  r18, r8, r4    #    r18 <- fte+i
        lbu  r5, 0(r18)     #    r5 <- 0x0000.00 & fte[i]
        beq  r5, r0, fim    #    (r5 == '\0') ? -> terminou
        add  r19, r9, r4    #    r19 <- dst+i
        sb   r5, 0(r19);    #    dst[i] <- (char)fte[i]
        addi r4, r4, 1      #    i = i + 1;
        j    lasso          # }
fim:    sb   r0, 1(r19)     # dst[i] = '\0';
```

Exemplo 11.10 O trecho de código no Programa 11.4 percorre uma lista encadeada, cujos elementos são do tipo `elemType`. O primeiro componente da estrutura é um apontador para o próximo elemento da lista, e o segundo componente é um vetor de seis inteiros.

Antes do laço, o apontador é inicializado com o endereço da estrutura de dados – lembre que o nome da estrutura equivale a `&(estrut[0])`. Se a lista é não vazia então o apontador não é nulo, e os elementos do vetor de inteiros são inicializados. Isso feito, o teste é repetido para o próximo elemento da lista.

Na versão em assembly, no Programa 11.5, o registrador `rp` é carregado com o endereço do primeiro elemento do vetor, e as constantes são carregadas em seis registradores. Do ponto de vista de eficiência da execução do código, estas constantes *devem* ser carregadas *fora* do laço para evitar a repetição destas operações, cujo resultado é constante, no corpo do laço.

O teste compara o valor do apontador para o próximo elemento (`rn = p->next`) com `NULL` e o laço termina se `rn == r0`. Os elementos do vetor são inicializados com deslocamentos de 4 (apontador) mais o índice multiplicado por 4.

No final do laço, o apontador é de-referenciado para que `rn` aponte para o próximo elemento da lista, e o teste é então repetido. Note que o conteúdo de um apontador é um endereço, que pode ser usado diretamente como tal. ◁

Programa 11.4: Laço que percorre uma lista encadeada com *pointers*.

```
typedef struct elem {
   elem *next;
   int   vet[6];
} elemType;

elemType *p;
elemType estrut[256];
   ...
   p = estrut;     // p <- &(estrut[0])
   while (p->next != NULL) {
      p->vet[0] = 1;
      p->vet[1] = 2;
      p->vet[2] = 4;
      p->vet[3] = 8;
      p->vet[4] = 16;
      p->vet[5] = 32;
      p = p->next;
   }
```

Programa 11.5: Versão em *assembly* do laço que percorre uma lista encadeada com *pointer*.

```
       la rp, estrut   # rp <- &(estrut[0])
       li r1, 1        # estes são inicializados FORA do laço
       li r2, 2        #   porque são valores constantes
       li r3, 4
       li r4, 8
       li r5, 16
       li r6, 32

lasso: lw   rn, 0(rp)   # rn <- p->next
       beq  rn, r0, fim # (p->next == NULL) ? terminou
       sw   r1, 4(rp)   # vet[0] <- 1; deslocamento = 4*(i+1)
       sw   r2, 8(rp)   # vet[1] <- 2
       sw   r3, 12(rp)  # vet[2] <- 4
       sw   r4, 16(rp)  # vet[3] <- 8
       sw   r5, 20(rp)  # vet[4] <- 16
       sw   r6, 24(rp)  # vet[5] <- 32
       move rp, rn      # próximo: p <- p->next
       j    lasso

fim:   nop
```

Exercícios

Ex. 11.2 Traduza para *assembly* do MIPS os trechos de programa em C abaixo. Lembre que em C o valor de um comando de atribuição é o valor atribuído.

```
// (a) ----------------------------------------
#define NN 1024
int i, sum, v[NN];
...
sum = 0;
for (i=0; i < NN; i+=2)
   sum += v[i];

// (b) ----------------------------------------
char *fte, *dst;
...
while ( ( *dst = *fte ) != '\0') {
   dst++; fte++; i++;
}

// (c) ----------------------------------------
char fte[NN]; char dst[NN];
int i, num;
...
i = 0;
while (fte[i] != '\0') {
   i = i + 1;
}
num = i;
for (i=0; num > 0; num--, i++) {
   dst[i] = fte[num - 1];
}

// (d) ----------------------------------------
#define SZ 1024
typedef struct A {
   int   x;
   short z[4];
   char  s[8];
} aType;
aType V[ SZ ]; // compilador aloca V em 0x0040.0000
int i,a,b,c;
...
a = b = c = 0;
for (i=0; i < SZ; i+=4) {
   a = a + V[i].x + (int)V[i].z[1];
   b = b + (int)(V[i].s[1] + V[i].s[7]);
   c = c + V[i].x - (int)V[i].z[3];
}

// Se P é uma potência de dois, então x % P = x AND (P-1)
p = q = r = 0;
for (i=0; i < SZ; i+=16) {
   p = V[i].x;
   q = q + (int)(V[i].s[(p % 8)] + V[i].s[(p % 8)]);
   r = r + V[i].x - (int)V[i].z[(q % 4)];
}
```

11.2 Implementação de Funções em *Assembly*

Nesta seção examinamos com algum detalhe a implementação em *assembly* de funções escritas na linguagem C.

11.2.1 Definição e declaração de funções em C

A *definição* de uma função declara o tipo do valor a ser retornado, declara os tipos dos parâmetros, e contém o corpo da função que computa o valor da função a partir dos parâmetros. Além dos comandos, o corpo da função pode conter declarações das variáveis locais à função. O Programa 11.6 mostra o esqueleto da definição de uma função na linguagem C [KR88, KP90].

Programa 11.6: Definição de uma função em C.

```
tipo nome_da_função( parâmetros formais ) { // cabeçalho
   declarações                              // corpo da função
   comandos
}
```

Para que uma função que é definida em outro arquivo com código fonte possa ser usada, é necessário que ela seja *declarada* antes da primeira invocação, porque sem a declaração, o compilador não tem como gerar o código para invocar a função. O mesmo vale para funções definidas em bibliotecas. A declaração contém somente os tipos dos parâmetros e do valor retornado pela função. Tipicamente, as declarações de funções são agrupadas num arquivo de cabeçalho que é incluído nos arquivos que necessitam daquelas funções. O Programa 11.7 mostra uma declaração de função.

Programa 11.7: Declaração de uma função em C.

```
tipo nome_da_função( lista de tipos dos parâmetros );
```

O Programa 11.8 mostra algumas declarações de funções. Uma função que não retorna um valor tem o 'tipo' `void`. Uma função sem parâmetros tem *um* parâmetro de 'tipo' `void`. A palavra 'tipo' aparece entre aspas porque `void` não é exatamente um tipo, no sentido estrito do termo, mas sim um *placeholder* para o tipo ou argumento nulo. A função `j()` é similar à função `printf()`, que recebe um argumento que, ao ser interpretado em tempo de execução, determina quantos e quais os tipos dos demais parâmetros.

Programa 11.8: Exemplos de declarações de funções em C.

```
void f(void);        // sem argumentos e não retorna valor
int  g(void);        // sem argumentos e retorna inteiro
int  h(int, char);   // dois argumentos, retorna inteiro
int  j(const char *, ...); // número variável de argumentos
```

11.2.2 Avaliação de expressões e de funções

Como são avaliados os comandos e expressões em C? Da esquerda para a direita, em avaliação preguiçosa, e com efeitos colaterais. *Avaliação preguiçosa* consiste em avaliar uma expressão

somente até que seu valor seja determinado. Por exemplo, a avaliação preguiçosa da expressão $0 \wedge X$ ignora o valor de X porque a conjunção de 0 com qualquer coisa é 0; da mesma forma, a avaliação de $1 \vee Y$ ignora o valor de Y porque a disjunção de 1 com qualquer coisa é 1.

Uma linguagem com *efeitos colaterais* permite que os efeitos da avaliação de uma subexpressão alterem a avaliação de outras subexpressões de uma mesma expressão. Por exemplo, este comando é válido em C e tem um efeito colateral da avaliação de `E`:

```
a = E + (E = a*b) + z*E + w/E;
```

O lado direito da atribuição é avaliado da esquerda para a direita e o "valor de uma atribuição" é o "valor atribuído". Na primeira parcela da soma, `E` tem o valor que lhe fora atribuído anteriormente, enquanto que nas demais parcelas `E` vale `a*b`. Programas com efeitos colaterais podem ser extremamente difíceis de depurar porque os valores das subexpressões mudam durante a avaliação da expressão que as contêm. Facilmente, o código pode se tornar ininteligível.

Pior ainda, o manual que define a linguagem C informa que a ordem de avaliação pode ser escolhida arbitrariamente pelo compilador e que o resultado não é portável[4]. Por *código portável* entende-se o código fonte que, compilado em qualquer máquina e com qualquer compilador, produz resultados idênticos[5].

Os argumentos de uma função também são avaliados da esquerda para a direita, com avaliação preguiçosa e efeitos colaterais, com este algoritmo:

1. Cada expressão na lista de argumentos é avaliada, da esquerda para a direita;
2. se necessário, os valores das expressões são convertidos para o tipo do parâmetro formal, e o valor é atribuído ao argumento correspondente no corpo da função;
3. o corpo da função é executado;
4. se um comando **`return`** é executado, o controle é devolvido à função que chamou;
5. se o **`return`** inclui uma expressão, seu valor é computado e o tipo convertido para o tipo do valor de retorno da função. Se o **`return`** não contém uma expressão, nenhum valor útil é retornado. Se o corpo da função não inclui um **`return`**, então o controle é devolvido quando a execução do corpo da função chegar ao seu último comando;
6. todos os argumentos são passados "por valor" (*call by value*), mesmo que o 'valor' seja um endereço (*pointer*).

Como é avaliado o comando `a = f(p*2, q, g(r,s,t), q/2, x+4, y*z);` ?

1. o valor de `p*2` é atribuído ao primeiro argumento;
2. o conteúdo da variável `q` é atribuído ao segundo argumento;
3. a função `g()` é avaliada com argumentos `r`, `s` e `t`, e seu valor atribuído ao terceiro argumento;
4. `q/2` é avaliado e atribuído ao quarto argumento;
5. `x+4` é avaliado e atribuído ao quinto argumento;
6. `y*z` é avaliado e atribuído ao sexto argumento; e
7. a função é invocada e seu valor de retorno atribuído à variável `a`.

[4] Conheço um exemplo de código que produz resultados distintos para duas versões do *mesmo* compilador.

[5] De acordo com `stackoverflow.com`, o manual de C99, informa que *"the order of evaluation of the function designator, the actual arguments, and subexpressions within the actual arguments is unspecified, but there is a sequence point before the actual call"*.

Regras de escopo

O valor de identificadores, ou os seus conteúdos, só pode ser acessado nos blocos em que são declarados.

Classes de armazenagem (*storage classes*)

As *classes de armazenagem*, que determinam o local em que uma determinada variável é armazenada, são:

`auto` variáveis declaradas dentro de um bloco (variáveis locais), armazenadas na pilha;

`extern` variáveis declaradas fora do corpo de uma função; seu escopo se estende a todas as funções que aparecem após sua declaração. Funções podem ser declaradas como **`extern`**;

`register` indica ao compilador que variável deve, se possível, ser alocada num registrador físico (raramente implementado nos geradores de código/compiladores);

`static` variáveis declaradas como **`static`** num bloco retém seus valores entre execuções do bloco;

`static (external)` variáveis declaradas fora de um bloco mas com escopo restrito ao arquivo em que são declaradas. Funções declaradas como **`static`** são visíveis apenas no arquivo em que são declaradas.

Variáveis das classes **`extern`** e **`static`**, se não forem inicializadas pelo programador, são inicializadas em 0 pelo compilador.

11.2.3 Implementação de funções no MIPS32

O conjunto de instruções MIPS32r2 provê duas instruções para o suporte a funções, *viz*:

`jal` `end` *jump and link*, com dois efeitos: salta para o endereço indicado no argumento e salva o endereço de retorno em `r31=ra` (*return address*), que é o *link*:
```
jal ender # PC <- ender , ra <- PC+4
```

`jr` `reg` *jump register*, que salta para o endereço de ligação/retorno, que foi armazenado em `ra` por **`jal`**:
```
jr ra    # PC <- ra
```

A Tabela 11.5 mostra a convenção de uso dos registradores definida na *Application Binary Interface* (ABI) do MIPS32 [SCO96]. Somente os registradores `r0` e `r31` tem usos determinados pelo *hardware*; a utilização de todos os demais é fruto de convenção de *software*.

Tabela 11.5: Convenção de uso de registradores para chamadas de função.

REG	FUNÇÃO	NÚMERO
`$zero`	sempre zero (em *hardware*)	`r0`
`at`	temporário para montador (*assembly temporary*)	`r1`
`v0-v1`	dois registradores para retornar valores (*value*)	`r2,r3`
`a0-a3`	quatro regs. para passar argumentos	`r4-r7`
`s0..s7`	regs. 'salvos' são preservados (*saved*)	`r16-r23`
`t0..t9`	regs 'temporários' não são preservados	`r8-r15,r24,r25`
`k0,k1`	temporários para o SO (*kernel*)	`r26,r27`
`gp`	*global pointer* (dados estáticos 'pequenos')	`r28`
`sp`	apontador de pilha (*stack pointer*)	`r29`
`fp`	apontador do registro de ativação (*frame pointer*)	`r30`
`ra`	endereço de retorno (*return address*, em *hardware*)	`r31`

A cada chamada de função encontrada num programa, o compilador deve gerar instruções para efetuar os sete passos listados abaixo. O Programa 11.9 mostra o código *assembly* para a implementação do comando `z = int f(int x);`. Os números das linhas indicadas referem-se ao Programa 11.9.

1. Alocar os argumentos onde o corpo da função possa encontrá-los (linha 1);
2. transferir controle para a função e armazenar *link* no registrador `ra` (linha 2);
3. o corpo da função deve alocar o espaço necessário na pilha para computar seu resultado (linha 5);
4. executar as instruções do corpo da função (linha 6);
5. colocar o valor computado onde a função que chamou possa encontrá-lo (linha 7);
6. devolver o espaço alocado em pilha (linha 8); e
7. retornar controle ao ponto de invocação da função (linha 9).

Programa 11.9: Protocolo de invocação de função.

```
1      move a0,rx        # prepara argumento
2      jal  f            # salta para a função e salva link
3      move rz,v0        # valor da função, end. de retorno=3
4      ...
5  f:  addi sp, sp, -32  # aloca espaço na pilha, 32 bytes
6      ...               # computa valor
7      move v0, t0       # prepara valor por retornar
8      addi sp, sp, 32   # devolve espaço alocado na pilha
9      jr   ra           # retorna, usando o link
```

11.2.4 Registro de ativação

Qual é a estrutura de dados necessária para suportar funções? Por que?

Uma *função folha* é uma função que não invoca outra(s) função(ões). Um *registro de ativação* (*stack frame*) é alocado para cada função não-folha e para cada função folha que necessita

alocar espaço para variáveis locais. A pilha cresce de endereços mais altos para endereços mais baixos. A Figura 11.9 mostra o leiaute de um registro de ativação completo.

	registro da função que chamou
	argumentos a mais que 4 (5,6,7...)
fp →	registradores com argumentos 1 a 4, se existem (a0..a3)
	endereço de retorno (ra)
	registradores salvos, se alterados pela função (s0..s7)
	variáveis locais, se existem
sp →	área para construir argumentos a mais que 4, se existem
	registro da próxima função a ser chamada

Figura 11.9: Registro de ativação no MIPS-32.

Um registro de ativação deve conter espaço para:

variáveis locais e temporárias declaradas no escopo da função;

registradores salvos espaço só é alocado para aqueles registradores que devem ser preservados. Uma função não-folha deve salvar `ra`. Se qualquer dentre `s0`-`s7` (`r16`-`r23`) e `sp`,`fp`,`ra` (`r29`-`r31`) são alterados no corpo da função, estes devem ser preservados na pilha e restaurados antes do retorno da função. Registradores são empilhados na ordem de número, com registradores de números maiores armazenados em endereços mais altos. A área de salvamento de registradores deve ser alinhada como *doubleword* (8 bytes);

área para argumentos de chamada de função numa função não-folha, o espaço necessário para todos os argumentos que podem ser usados para invocar outras funções deve ser reservado na pilha. No mínimo, quatro palavras devem ser sempre reservadas, mesmo que o número de argumentos passados a qualquer função seja menor que quatro palavras; e

alinhamento a convenção (válida para o SO, inclusive) exige que um registro de ativação seja alinhado como *doubleword*. O alinhamento é em *doubleword* porque este é o maior tamanho de palavra que pode ser empilhado, que é um valor de ponto flutuante do tipo `double`, no caso do MIPS32r2.

Uma função aloca seu registro de ativação ao subtrair do *stack pointer* (`sp`) o tamanho de seu registro, no início de seu código. O ajuste no `sp` deve ocorrer antes que aquele registrador seja usado na função, e antes de qualquer instrução de salto ou desvio. A desalocação do registro deve ocorrer no último bloco básico da função, que inclui todas as instruções após o último salto ou desvio do código até a instrução de retorno (*jump-register*).

A ordem de armazenagem dos componentes no registro de ativação deve ser respeitada mesmo que o código de uma função não os utilize todos.

A Tabela 11.6 mostra quais recursos devem ser preservados pelo código de uma função. Do ponto de vista da função que chama, nenhum dos recursos do lado esquerdo da tabela é alterado pela função chamada. Os detalhes sórdidos estão em [SCO96], páginas 3.11 a 3.21.

Tabela 11.6: Preservação de conteúdos entre chamadas de funções.

PRESERVADOS	NÃO PRESERVADOS
`s0-s7` (regs. salvos)	`t0-t9` (temporários)
`fp` (*frame pointer*)	`a0-a3` (argumentos)
`sp` (*stack pointer*)	`v0,v1` (valores de retorno)
`ra` (*return address*)	`at, k0, k1` (*assembler temporary, kernel*)
pilha acima do `sp`	pilha abaixo do `sp`

O programador em *assembly* é responsável por definir o leiaute do registro de ativação de cada função em função de quantos e quais são os argumentos da função, quais as variáveis locais, e quais os registradores que são alterados no corpo da função. As variáveis locais e registradores salvos devem ser referenciados usando o apontador da pilha como registrador base. O compilador emprega rotinas que executam estas mesmas funções.

Exemplo 11.11 O código da função `int g(int x, int y, int z);` é mostrado no Programa 11.10. A função `g()` declara três variáveis locais em seu corpo, e é uma função não-folha. Seu registro de ativação, mostrado na Figura 11.10, deve acomodar 3 argumentos, o registrador com o endereço de retorno, e as três variáveis locais, perfazendo 28 bytes, que alinhado como *doubleword*, resulta em 32 bytes.

A ordem em que os registradores são preservados, e depois recuperados não é importante – o que importa é que o endereço em que cada registrador é salvo no início da função seja exatamente o mesmo de onde seu conteúdo é recuperado logo antes do retorno. ◁

Programa 11.10: Parte do código da função `g(x,y,z)`.

```
1     # w = g(x,y,z);
2     move  a0,rx       # prepara 3 argumentos
3     move  a1,ry
4     move  a2,rz
5     jal   g             # salta e armazena link em ra
6     move  rw,v0       # recebe valor de retorno
7     ...
8  g: addiu sp,sp,-32   # espaço para 3 args + ra + 3 vars
9     sw    ra,12(sp)   # empilha endereço de retorno
10    sw    a0,16(sp)   # empilha a0
11    sw    a1,20(sp)   # empilha a1
12    sw    a2,24(sp)   # empilha a2
13    ...         # corpo de g()
14    move  v0,rw       # valor de retorno
15    lw    ra,12(sp)   # recompõe endereço de retorno
16    lw    a0,16(sp)   # recompõe a0
17    lw    a1,20(sp)   # recompõe a1
18    lw    a2,24(sp)   # recompõe a2
19    addiu sp,sp,32    # desaloca espaço na pilha
20    jr    ra            # retorna
```

	...	sp + 28
	a2	sp + 24
	a1	sp + 20
	a0	sp + 16
	ra	sp + 12
	var loc1	sp + 8
	var loc2	sp + 4
sp →	var loc3	sp + 0

Figura 11.10: Registro de ativação do Programa 11.10.

Exemplo 11.12 Vejamos a tradução para *assembly* de uma função simples, empregando as convenções de programação do MIPS.

```
int fun(int g, int h, int i, int j) {
   int f = 0;

   f = (g+h)-(i+j);
   return (f*4);
}
```

Os quatro argumentos são armazenados, da esquerda para a direita nos registradores `a0` a `a3`. O valor da função é retornado em `v0`. A função `f` é uma função folha e portanto é desnecessário empilhar o endereço de retorno, assim como os registradores com os argumentos. O trecho inicial de código mostra a preparação dos argumentos – as variáveis são copiadas para os registradores, e o valor da função é copiado para a variável `k` após o retorno.

Espaço na pilha é alocado para a variável local `f`, num registro de ativação alinhado como *doubleword*. O registro de ativação contém somente a variável local, que é alocada no endereço apontado por `sp`. As operações intermediárias salvam seus resultados em registradores temporários (`t0` a `t3`). O valor intermediário é salvado na variável local, recuperado e então multiplicado por quatro com um deslocamento para a esquerda. O espaço na pilha é desalocado antes do retorno da função.

```
main: ...
      move a0, rg          # quatro argumentos
      move a1, rh
      move a2, ri
      move a3, rj
      jal  fun             # salta para fun()
      move rk, v0          # valor de retorno
      ...
fun: addiu sp, sp, -8      # aloca f na pilha, alinhado
     sw    r0, 0(sp)       # f <- 0
     add   t0, a0, a1      # t0 <- g + h
     add   t1, a2, a3      # t1 <- i + j
     sub   t2, t0, t1
     sw    t2, 0(sp)       # f <- (g+h)-(i+j);
     lw    t3, 0(sp)
     sll   v0, t3, 2       # v0 <- f*4
     addiu sp, sp, 8       # desaloca espaço na pilha
     jr    ra              # retorna
```

Esta função está codificada em 10 instruções, sendo três delas acessos à memória, que são operações deveras custosas. Este estilo de código é o produzido por um compilador, ao compilar sem nenhuma otimização, tal como com `gcc -O0`. ◁

Exemplo 11.13 Vejamos algumas otimizações para reduzir o tamanho e complexidade no código do Exemplo 11.12.

O corpo da função é tão simples, que a variável local pode ser mantida num registrador e portanto não é necessário salvar nada na pilha e o registro de ativação da função é vazio. Economia: duas instruções para manipular a pilha, inicialização de `f`, salvamento e leitura desta variável – todos os acessos à pilha foram eliminados porque desnecessários.

```
fun: add   a0, a0, a1     # a0 <- g + h
     add   a2, a2, a3     # a2 <- i + j
     sub   a2, a0, a2     # a2 <- (g+h)-(i+j);
     sll   v0, a2, 2      # v0 <- f*4
     jr    ra             # retorna
```

Os registradores temporários também são desnecessários porque o corpo da função é simples – tão simples que uma macro seria suficiente. Os valores intermediários são computados nos registradores de argumentos.

A versão otimizada tem cinco instruções e nenhum acesso à memória, além dos acessos inevitáveis para buscar as instruções da função. Esta economia é obtida quando o compilador otimiza o código, por exemplo, com `gcc -O2`. ◁

Exercícios

Ex. 11.3 Traduza para *assembly* a função abaixo. Seu código *assembly* deve empregar as convenções de programação do MIPS. Todas as variáveis estão declaradas e tem os tipos e tamanhos adequados.

```
int fun(int a, int b, int c, int d, int e, int f);
   ...
   int a, p, q, z, w, v[N];
   ...
   x = fun(16*a, z*w, gun(p,q,r,s), v[3], v[z], z-2);
   ...
```

Ex. 11.4 Traduza para *assembly* a função abaixo. Seu código *assembly* deve empregar as convenções de programação do MIPS.

```
int fati(int n) {
   int i,j;
   j=1;
   if(n > 1)
      for(i = 1; i <= n; i++)
        j = j*i;
   return(j);
}
```

Ex. 11.5 Traduza para *assembly* a função abaixo. Seu código *assembly* deve empregar as convenções de programação do MIPS.

```
int fatr(int n) {
   if(n < 1)
      return (0);
   else
      return (n * fatr(n-1));
}
```

Ex. 11.6 Traduza para *assembly* a função abaixo. Seu código *assembly* deve empregar as convenções de programação do MIPS.

```
int log2(int n) {
   if (n < 2) then
      return 0;
   else
      return (1 + log2(n/2));
}
   ...
   x = log2(96000);   // maior do que |32.767|
   ...
```

Ex. 11.7 Traduza para *assembly* a função abaixo. Seu código *assembly* deve empregar as convenções de programação do MIPS. Não escreva o código para `print()`; somente prepare os argumentos para sua invocação.

```
void print(char *, int); // não escreva o código desta função

int fib(int n) {
  if ( n == 0 )
     return 0;
  else
     if ( n == 1 )
        return 1;
  else
    return ( fib(n-1) + fib(n-2) );
}

void main() {
  int c;

  for (c = 1 ; c < 6 ; c++)
     print("%d\n", fib(c));
}
```

Espaço em branco proposital.

Ex. 11.8 Traduza para *assembly* a função abaixo. Seu código *assembly* deve empregar as convenções de programação do MIPS.

```
typedef elem {
   elemType *next;
   int  vet[3];
} elemType;

elemType *x;
elemType strut[256];
   ...
   x = insert( strut, strut, j) );
   x->vet[2] = 512;
   ...

elemType* insert(elemType *p, elemType *s, int i) {
  while (p != NULL) {
     p = p->next;
  }
  p->next = &(s[i]);
  (p->next)->next = NULL;
  return p->next;
}
```

11.3 Montadores

Computadores não executam diretamente programas escritos em linguagens de alto nível tais como C ou Pascal. O formato dos programas que eles executam diretamente chama-se de *executável*, e sob o modelo de VonNeuman, tal código é nada mais que uma sequência de bits, cujo significado é atribuído pelo projetista do *conjunto de instruções* (CdI), ou pelo *arquiteto* do sistema/computador.

Cada *instrução* ao computador consiste de um certo número de bits, 32 no caso do MIPS, e a cada padrão de 32 bits corresponde a uma ação distinta do computador. Felizmente, computadores podem ser muito úteis mesmo sem executar as 2^{32} ações distintas – 4 bilhões é um número deveras grande.

O *montador* (*assembler*) é um programa que 'monta' o código em *linguagem de montagem* (*assembly language*), gerando o código binário que é interpretado pelo computador. Como veremos adiante, instruções em *assembly* são algo mais compreensíveis do que um número binário de 32 dígitos. No caso do MIPS, a instrução que soma o conteúdo dos registradores `$5` e `$3`, e armazena o resultado no registrador `$3` é:

```
  addu $3,$5,$3  00000000101000110001100000100000     0x00a31820
                 000000 00101 00011 00011 00000 100000
                 opcode  rs    rt    rd  shamt  fun
```

Breve lição de História. O nome 'completo' do montador disponível em nosso sistema é `gas`, abreviatura para *Gnu ASsembler*. O nome do "montador nativo" em sistemas Unix é `as`, enquanto que o nome do "compilador nativo" para C é `cc`, ou *C Compiler*. O nome `gcc` é uma abreviatura para *Gnu C Compiler* – ignoro a razão pela qual o compilador é chamado de `gcc` mas o montador é chamado de `as`, ao invés de `gas`. *Fim da lição.*

Por *compilador nativo*, ou *montador nativo*, entende-se o programa que traduz código para ser executado no mesmo computador em que o tradutor executa. Um *cross-compilador* é um compilador que produz código para ser executado num processador distinto. No nosso caso, usaremos `mips-gcc` e `mips-as` para traduzir código C e *assembly* para ser executado num processador que executa as instruções do MIPS, e não para o processador nativo, que é algum membro da família estendida dos x86.

11.3.1 O processo de compilação

Considere uma aplicação cujo código fonte está separado em dois arquivos, `x.c` que contém a função `main()`, e `y.c` que contém a função `fun()`. A Figura 11.11 mostra um diagrama com as etapas da compilação dos dois arquivos para produzir o executável `a.out`. Os círculos contém os programas que traduzem "o programa", de código C para código executável; os nomes sobre as setas indicam os arquivos gerados em cada etapa do processo de compilação.

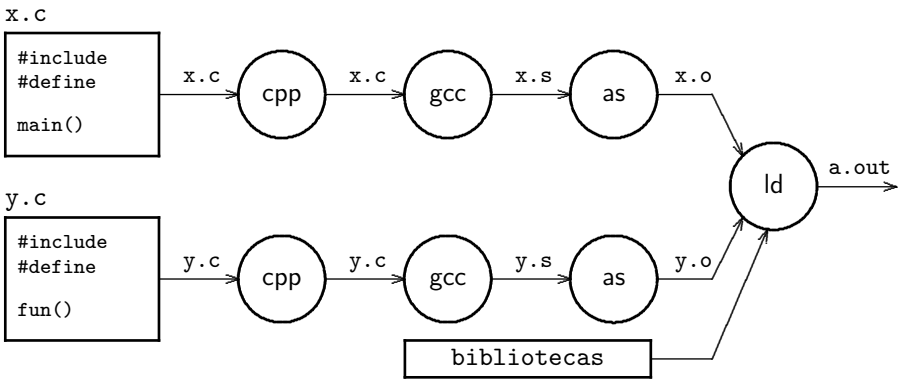


Figura 11.11: Etapas do processo de compilação.

Os arquivos fonte contém diretivas do tipo `#include` e `#define` que são processadas pelo preprocessador cpp, que faz a expansão das macros, a inclusão dos arquivos com cabeçalhos e remove os comentários. A saída do cpp é entregue ao compilador gcc, que faz a tradução de C para linguagem de montagem. Ao contrário do cpp, que só manipula texto, o compilador traduz comandos da linguagem C para instruções em *assembly*.

O código em *assembly* é processado pelo montador as, que produz um *arquivo!objeto*, já com as instruções traduzidas para seus equivalentes em binário, além da tabela de símbolos – veja a Seção 11.3.3 para uma descrição do processo de montagem.

Os arquivos objeto, e se necessário, o código mantido em bibliotecas, são agregados pelo *ligador* (ld), que finalmente produz o arquivo executável `a.out`. O processo de ligação é descrito no Capítulo **??** e seguintes.

Felizmente, tudo que a criatura deve fazer é invocar um único comando, tal como

```
gcc -Wall x.c y.c -lm && ./a.out
```

para que todo o processo de compilação seja efetuado por um comando chamado gcc. O uso do `gcc` é detalhado no Capítulo **??**.

11.3.2 Anatomia e fisiologia de um montador

Examinemos então a entrada e a saída de um montador, bem como alguns de seus componentes internos mais importantes.

Entrada e saída do montador

A entrada para o programa montador (*assembler*) é um ou mais arquivo(s) texto com código em linguagem de montagem (*assembly*), que foi gerado por uma pessoa ou pelo compilador. A saída do montador é um *arquivo objeto* formatado para ser usado como entrada por um programa ligador. Em "sistemas pequenos" o montador pode produzir um executável diretamente – este não é o caso de sistemas Unix/Linux.

Os arquivos de entrada do montador tem sufixo `.s` (*aSsembly*) e o arquivo de saída tem sufixo `.o`, para "arquivo Objeto".

Além das instruções da linguagem de montagem, montadores aceitam *diretivas*, que são instruções para o montador, e não para o processador. Diretivas permitem reservar espaço para variáveis, determinam se determinado trecho do código fonte corresponde a instruções ou a dados, além de várias outras tarefas administrativas. Geralmente, diretivas iniciam com um ponto, para diferenciá-las das instruções do *assembly*, como por exemplo '`.space`'.

Seções geradas pelo montador

O arquivo com o código objeto gerado pelo mips-as é armazenado em *seções* denominadas `.text`, `.data`, `.bss`, `.absolute`, `.undefined`, pelo menos. Uma lista reduzida dos conteúdos de cada seção é mostrada abaixo. A lista completa inclui ainda vários outras seções um tanto obscuras – detalhes na Seção **??**. As siglas "RO, RW, EX" significam "Read Only", "Readable and Writable", e "EXecutable".

`.text` instruções do programa e constantes, geralmente RO,EX;

`.data` as variáveis de um programa, geralmente RW;

`.bss` variáveis não-inicializadas e *common*s, variáveis inicializadas com zero (*block started by symbol*);

`.absolute` símbolos com endereço absoluto, em seção que será relocada pelo ligador para endereços absolutos em tempo de execução; e

`.undefined` lista de símbolos com endereço indefinido; durante a ligação, estes endereços devem ser preenchidos pelo ligador.

Location counter

O *location counter* é um contador mantido pelo montador que é incrementado a cada byte emitido (traduzido), e que aponta para o endereço em que algo está sendo gerado/montado.

Um ponto (ou *dot*) é o símbolo que contém o endereço do *location counter*. O trecho de código abaixo mostra um exemplo de uso do *dot*. No exemplo, o símbolo '`aqui`' contem seu próprio

endereço, que é o endereço de uma palavra. A diretiva **.word** reserva espaço para uma palavra em memória e este espaço é identificado pelo seu *label* 'aqui:'. O conteúdo da palavra apontada por 'aqui:' é o próprio valor do símbolo, que é o valor corrente do *location counter*, ou o *dot*.

```
aqui: .word .      # define símbolo para o endereço corrente
```

A expressão .=.+4 ('aqui' recebe 'aqui' mais quatro) equivale a reservar o espaço correspondente a quatro bytes. O mesmo efeito pode ser obtido com a diretiva **.space**:

```
meuint: .space 4  # define símbolo para um inteiro
```

Símbolos

Símbolos são usados para nomear endereços, na montagem, ligação e depuração. Símbolos iniciam por um caractere dentre '$', '.', '_', minúsculas ou maiúsculas e dígitos.

Comentários iniciam com '#' ou ';' e se estendem até o fim da linha.

Um *label* é um símbolo terminado por ':' que representa o valor do *location counter* naquele ponto da montagem. Um *label* corresponde a um endereço, que pode ser de uma instrução – função, destino de goto – ou variável, ou estrutura de dados.

Um símbolo pode representar um valor, que lhe é atribuído por uma diretiva tal como **.set**, **.size** ou **.equ**.

Diretivas

Diretivas são comandos ao montador e não fazem parte do conjunto de instruções de nenhum processador, mas permitem a alocação de espaço para variáveis (**.word**, **.byte**), definição de escopo de visibilidade de nomes (**.global**), além de outras funções de caráter administrativo.

De novo e paradoxalmente: diretivas fazem parte da linguagem *assembly* de um certo montador, mas não pertencem ao conjunto de instruções de nenhum processador.

As diretivas do montador mips-as estão documentadas no espartano manual (man as) e fartamente documentadas na página HTML do montador. Algumas diretivas são brevemente descritas abaixo – elas serão empregadas nas próximas aulas, quando então sua utilidade ficará mais evidente.

.text o que segue deve ser alocado na seção **.text** – código;

.data o que segue deve ser alocado na seção **.data** – dados;

.section texto que segue é montado na seção nomeada;

.previous troca esta seção pela que foi referenciada mais recentemente;

.ent endereço de "entrada" no código – primeira instrução da função;

.end marca o final de código, não monta nada até a próxima diretiva **.ent**;

.align expr ajusta contador de locação para valor múltiplo de 2^{expr};

.global torna símbolo visível ao ligador, aumentando seu escopo;

.comm declara símbolo na seção BSS (*block started by symbol*);

.ascii aloca espaço para cadeias, sem `'\0'`;

.asciiz aloca espaço para cadeias, com `'\0'`;

.byte resolve expressões e as aloca em bytes consecutivos;

.set `symb expr` resolve expressão e atribui valor ao símbolo;

.size `symb expr` resolve expressão e atribui tamanho [bytes] ao símbolo;

.equ `symb, expr` resolve expressão a atribui valor ao símbolo;

.type `symb tipo` atribui tipo ao símbolo: função ou variável;

.mask .fmask máscaras dos registradores usados no código.

Os valores de **.mask** e **.fmask** indicam quais registradores são usados: se o bit 19 está em 1 então o registrador $19 é usado. Estes valores são usados pelo ligador para gerar a seção `.reginfo` do arquivo objeto de processadores MIPS. A disjunção dos valores dos **.mask** indica quais registradores de inteiros são usados no código; os valores em **.fmask** indicam quais registradores de ponto flutuante são usados – a justificativa para tal é apresentada na Seção **??**.

Exemplo 11.14 O trecho de código C no Programa 11.11 foi compilado para gerar código *assembly* do MIPS, o que nos permite observar a saída do compilador, que é a entrada para o montador.

O código nos Programas 11.12 e 11.13 foi gerado com o comando `mips-gcc -S -O1 strcopy.c strcopy.s`. Os números à esquerda não fazem parte da saída, e servem apenas para identificar as linhas do programa.

Programa 11.11: Código fonte de `strcopy()`.

```
int strcopy(char x[], char y[]) {
  int i=0;
  while ( (x[i] = y[i]) != '\0' ) // copia e testa fim da cadeia
    i = i+1;
  return(i);
}

char fnte[] = "abcdefgh";     // variável global
char dest[] = "ABCDEFGH";     // variável global

int main (int argc, char** argv) {
  int num=0;
  num = strcopy(fnte, dest);
  return(num);
}
```

As linhas 1–8 são o cabeçalho do arquivo (prólogo) e descrevem seu conteúdo e características. O símbolo `strcopy` é declarado como um símbolo global (linha 6) e portanto sua visibilidade se estende para além deste arquivo. A linha 4 indica que o que segue deve ser gerado numa seção de código (`.text`).

Programa 11.12: Início da versão *assembly* de `strcopy()`.

```
1           .file   1 "strcpy.c"
2           .section .mdebug.abi32
3           .previous
4           .text
5           .align  2
6           .global strcopy
7           .set    nomips16
8           .ent    strcopy
9
10  strcopy:
11          .frame  $sp,0,$ra   # vars=0, regs=0/0, args=0
12          .mask   0x00000000,0
13          .fmask  0x00000000,0
14          .set    noreorder
15          .set    nomacro
16
17          move    $a2,$zero
18          lbu     $v0,0($a1)
19          beq     $v0,$zero,.L6
20          sb      $v0,0($a0)
21
22  .L4:
23          addiu   $a2,$a2,1
24          addu    $v0,$a0,$a2
25          addu    $v1,$a1,$a2
26          lbu     $v1,0($v1)
27          bne     $v1,$zero,.L4
28          sb      $v1,0($v0)
29
30  .L6:
31          jr      $31
32          move    $v0,$a2
```

As linhas 10–15 não avançam o *location counter*; portanto o endereço de `strcopy` é o da instrução `move` na linha 17.

Programa 11.13: Final da versão *assembly* de `strcopy()`.

```
33          .set    macro
34          .set    reorder
35          .end    strcopy
36          .size   strcopy, .-strcopy
37          .globl  fnte
38          .data
39          .align  2
40          .type   fnte, @object
41          .size   fnte, 9
42  fnte:
43          .ascii  "abcdefgh\000"
44          .global dest
45          .align  2
46          .type   dest, @object
47          .size   dest, 9
48  dest:
49          .ascii  "ABCDEFGH\000"
50          .text
51          .align  2
52          .global main
53          .set    nomips16
54          .ent    main
55  main:
56          .frame  $sp,24,$ra # vars=0, regs=1/0, args=16
57          .mask   0x80000000,-8
58          .fmask  0x00000000,0
59          .set    noreorder
60          .set    nomacro
61
62          addiu   $sp,$sp,-24
63          sw      $ra,16($sp)
64          lui     $a0,%hi(fnte)
65          addiu   $a0,$a0,%lo(fnte)
66          lui     $a1,%hi(dest)
67          jal     strcopy
68          addiu   $a1,$a1,%lo(dest)
69
70          lw      $ra,16($sp)
71          jr      $ra
72          addiu   $sp,$sp,24
73
74          .set    reorder
75          .end    main
76          .size   main, .-main
77          .ident  "GCC:␣(GNU)␣3.4.4␣mipssde-6.06.01-20070420"
```

As linhas 33-41 indicam que as constantes inicializadas são geradas na seção `.data` (linha 38), e que os nomes `fnte` e `dest` são símbolos globais e visíveis fora deste arquivo (linhas 37 e 44). `main()` é gerada na seção `.text` (linha 50) e as linhas 74-77 contém o epílogo do programa. ◁

Os *labels* gerados pelo compilador iniciam com '`.`' (`.L1:`) porque o compilador C não produz símbolos que iniciam com '`.`' . Isso garante que símbolos como `main:` sejam facilmente diferenciados dos símbolos que representam fim/início de laços, tais como `.L4` e `.L6` .

11.3.3 Algoritmo e estruturas de dados

Para entender a operação do montador, é necessário uma rapidíssima descrição do processo de compilação. Considere o processo de compilação dos arquivos `x.c` e `y.c`, mostrado na Figura 11.11. `y.c` contém a definição de várias funções que são empregadas em `x.c`. Os dois arquivos fonte são compilados e montados separadamente; o programa *ligador* edita o arquivo que resulta da compilação de `x.c` e ajusta os endereços das funções referenciadas naquele arquivo, mas definidas em `y.c`, para que estas referências reflitam os endereços no código gerado para `y.c`. Não se preocupe se, no momento, parece um tanto confuso porque várias páginas são dedicadas aos copiosos detalhes que faltam nesta explanação.

A estrutura de dados principal de um montador é sua *tabela de símbolos* (TS), que contém os símbolos declarados no programa e seus valores. Durante a montagem, alguns dos valores podem estar momentaneamente indefinidos, ou permanecer indefinidos até o final da execução do montador. A implementação mais simples de um montador consiste de "duas passadas" sobre o código *assembly*: (i) a tradução das instruções; e (ii) o ajuste dos endereços, que são indicados na Figura 11.12 e detalhados no que se segue.

primeira passada	lê código fonte
	traduz todas instruções sem endereços
	insere símbolos na tabela de símbolos
	gera arquivo intermediário (saída parcial)
segunda passada	lê arquivo intermediário
	pesquisa na tab. de símbolos e ajusta endereços
	gera saída completa

Figura 11.12: Algoritmo de montagem em dois passos.

Na *primeira passada*, o montador lê o arquivo com o código fonte e traduz cada instrução que esteja completamente definida, tal como uma adição. Se um operando de uma instrução é um endereço que ainda está indefinido, o montador insere o símbolo correspondente àquele endereço na tabela de símbolos, e marca a instrução como "incompletamente traduzida".

A cada instrução traduzida, o montador avança o *location counter*. O mesmo vale para a seção de dados – a cada variável declarada no programa, o espaço necessário é reservado na seção apropriada, e o *location counter* daquela seção é avançado de tantos bytes quanto necessário para acomodar a variável ou estrutura de dados.

Ao final da primeira passada, todos os símbolos do programa foram armazenados na tabela de símbolos, e seus valores (endereços) podem ser determinados pelo montador. Se o endereço de um símbolo não pode ser determinado em tempo de montagem, esta informação é repassada para o ligador, que então resolverá o valor do símbolo. Se isso não é possível então ocorreu um erro de compilação, ou de ligação, e o programa não pode ser executado porque uma função, ou uma variável, está com um endereço indeterminado.

Na *segunda passada*, o montador percorre novamente o arquivo com o código, e para cada

instrução incompletamente traduzida, a tabela de símbolos é consultada para resolver o símbolo que não foi resolvido no primeiro passo. Ao consultar a tabela, o valor do símbolo é usado para alterar o arquivo de saída que então reflete a informação atualizada.

Ao final da segunda passada, o arquivo de saída é gerado, e possivelmente, todas as instruções estão completamente traduzidas, com as informações de endereço completas. Caso algum símbolo não tenha sido resolvido, como uma invocação de `printf()`, por exemplo, esta informação é armazenada no arquivo de saída para que o ligador se encarregue de alterar a instrução que invoca a função `printf()`, fazendo a ligação entre o símbolo `printf:` no arquivo recém-montado e o endereço correspondente ao símbolo, que é a primeira instrução daquela função na biblioteca `libc.a`.

De volta ao exemplo desta seção: se o arquivo `x.o` contém símbolos indefinidos, tal como a invocação de `fun()`, o ligador cria uma nova tabela de símbolos, com os símbolos de `x.o` e com os símbolos definidos em `y.o`. O arquivo de saída com a junção de `x.o` e `y.o` é o executável `a.out`, *se e somente se*, todos símbolos indefinidos em `x.o` foram resolvidos por símbolos definidos em `y.o`, ou na biblioteca `libc.a` é ligada a estes dois arquivos objeto.

Vários detalhes importantes foram omitidos nesta descrição, tal como os detalhes da ligação com funções de biblioteca; estes serão investigados no Capítulo **??** e seguintes.

Num processador como o MIPS, no qual todas as instruções tem o mesmo tamanho, o processamento em duas passadas pode parecer exagero porque basta contar as instruções para gerar todos os endereços no arquivo objeto. Na montagem de código para processadores que usam instruções de tamanho variável, como é o caso do x86, cujas instruções tem de um a 17 bytes, é necessário traduzir as instruções para binário no primeiro passo, porque só então é possível determinar todos os endereços de todas as instruções.

Exercícios

Ex. 11.9 É possível efetuar a montagem com uma única passagem sobre o código? Se sim, indique as estruturas de dados e o algoritmo para fazê-lo.

Ex. 11.10 Talvez seja mais eficiente efetuar a montagem em três passadas sobre o código. Se sim, indique as estruturas de dados e o algoritmo para fazê-lo, e explique em quais condições a montagem em três passadas poderia ser mais eficiente do que duas.

Ex. 11.11 (a) Dê dois exemplos de diretivas do montador que *não* causam a inclusão de bits adicionais no arquivo objeto e explique suas funções; (b) dê dois exemplos de diretivas do montador que produzem saída no arquivo objeto e explique suas funções; (c) qual a função da diretiva `.align` ? Esta diretiva não pode estar incluída nas suas respostas anteriores.

Ex. 11.12 (a) Escreva uma função em C que computa a redução por *ou-exclusivo* de um vetor de inteiros ($x = \bigoplus_{i=0}^{n-1} V_i$), que é apontado pelo primeiro argumento, conforme o protótipo abaixo; (b) traduza sua função para o *assembly* do MIPS; e (c) escreva em *assembly* o trecho de código em que `reduz()` é invocada, e mostre o leiaute do registro de ativação. Seu código *assembly* deve respeitar as convenções para a codificação de funções. `int reduz(int *v, int n);`

Ex. 11.13 Traduza para *assembly* do MIPS os trechos de programa abaixo. Seu código *assembly* deve empregar as convenções de programação do MIPS.

```
// (a)
int a, x[2048], y[64];
   ...
   a = fun(x, 2048, y, 64);
   ...
int fun(int *p, int np, int *q, int nq) {
   int i=1;
   int s=0;
   while (i < np) {
      s = s + p[i] + p[ q[i%nq] % np ]; // MOD, MOD
      i = i * 2;
   }
   return s;
}

// (b) ------------------------------------------
#define SIZE 1024
typedef struct x {
   int a;
   int b;
   int c;
   short x;
   short y;
} xType;

xType V[SIZE], Z[SIZE];

void reduz(int lim, xType *v, xType *z, int pot) {
   int i=0;
   while (i < lim) {
      v[i].a = z[i].b + z[i].c;
      v[i].x = z[i].x <<pot;
      i = i + 1;
   }
}
   ...
   reduz(SIZE/4, V,  Z, 4);
   ...

// (c) ------------------------------------------
#define SZ 1024
int A[SZ], B[SZ];
...
int escalar(int tam, int *a, int *b, int const) {
   int i,j, soma;
   for (i=1, j=0, soma=0; i < tam; i=i*2, j=j+1) {
      b[j] = a[i]*const;
      soma += a[j];
   }
   return soma;
}
   ...
   escalar(SZ, A, B, 16);
   ...
```

Índice Remissivo

W

X

Z