Introdução a Organização de Computadores e Linguagens de Montagem

Ricardo Anido

© *Draft date 1 de Março de 2012*

Capítulo 4

Procedimentos e funções

Neste capítulo vamos estudar as facilidades que os processadores oferecem para a implementação do conceito de procedimentos. Considere o seguinte trecho de programa na linguagem C da Figura 4.1, que inclui a declaração e o uso de um procedimento denominado `troca`:

```
 1
 2  ...
 3
 4  int x,y,z; // declaração de algumas variáveis
 5
 6  void troca(int *a, int *b)
 7  {
 8      int tmp;
 9
10      tmp = *a;
11      *a = *b;
12      *b = tmp;
13  }
14
15  int main(void)
16  {
17      ...
18      troca(&x,&y);
19      z=x+1;
20      troca(&z,&x);
21      x=y-1;
22      ...
23  }
```

Figura 4.1: Trecho com declaração e uso de procedimento em C.

A sequência de comandos executados durante as duas invocações do procedimento `troca` mostradas no trecho de programa são descritas na seguinte Tabela:

Linhas	Comentário
18	Primeira invocação de `troca`. O fluxo de execução desvia para o corpo do procedimento.
10, 11, 12	Execução dos comandos do corpo do procedimento.
19	Ao final do corpo do procedimento, desvia e executa este comando.
20	Segunda invocação de `troca`. O fluxo de execução desvia para o corpo do procedimento.
10, 11, 12	Execução dos comandos do corpo do procedimento.
21	Ao final do corpo do procedimento, desvia e executa este comando.

Podemos ver que a invocação de um procedimento envolve dois desvios no fluxo de execução de um programa. Na chamada do procedimento há um desvio para o início do procedimento, e ao final da execução do procedimento o fluxo desvia de volta para o comando seguinte à chamada de procedimento. Assim, na primeira invocação, após a execução do comando na linha 12 é executado o comando na linha 19; na segunda invocação, após a execução do comando na linha 12 é executado o comando na linha 21. E aí fica clara a dificuldade de se implementar procedimentos com as instruções de desvio que vimos até agora: a cada invocação do procedimento, o desvio ao término de sua execução deve ser feito para endereços diferentes.

O endereço para o qual o fluxo de execução deve ser desviado ao final do procedimento é chamado de *endereço de retorno* do procedimento. Note que o endereço de retorno pode ser calculado no momento da chamada do procedimento: é o endereço da instrução seguinte à chamada de procedimento, no programa que faz a chamada.

Uma primeira abordagem para a implementação de procedimentos é a seguinte. A chamada de procedimento pode ser implementada por uma instrução especial que armazena o endereço de retorno na primeira palavra do procedimento. Mais especificamente, suponha que exista uma instrução especial `jsr` (desvia para subrotina – em inglês, *jump to subroutine*), que tem um único operando, o endereço do procedimento:

JSR

Desvia para subrotina

Sintaxe	Operação	Flags	Codificação
`jsr` $expr_{32}$	$mem[imd_{32}] \leftarrow ip + 8$ $ip \leftarrow imd_{32} + 4$	–	31 … 0: | `f0` | | | | 31 … 0: | imd_{32} |

Ao ser executada, essa instrução armazena o valor do endereço de retorno na pri-

meira palavra do procedimento chamado. A primeira palavra do procedimento deve portanto ser reservada, ou seja, não deve conter uma instrução. Note que o endereço de retorno, supondo que o valor de `ip` no início da execução da instrução `jsr` seja X, é $X + 8$, já que essa instrução é codificada em duas palavras. Com esse esquema, quando uma chamada de procedimento é executada, o endereço de retorno a chamada fica armazenado em uma posição de memória conhecida. Ao final da execução do procedimento, para retornar da chamada, podemos recuperar o endereço de retorno, armazenado na primeira palavra do procedimento, e desviar para esse endereço.

Por exemplo, considere o seguinte trecho em linguagem de montagem, em que uma chamada ao procedimento de nome `procP` é feita usando uma instrução `jsr`:

```
 1           ...
 2          jsr procP       ; uma chamada ao procedimento de nome proc_P
 3          xor r1,r2       ; esta é a instrução que deve ser executada
 4                          ; no retorno do procedimento
 5          ...
 6
 7  ; esta é a declaração do procedimento
 8  procP:
 9          ds  4           ; a primeira palavra do procedimento é reservada
10                          ; para armazenar o endereço de retorno
11          set r5,-1       ; a primeira instrução do procedimdento está no
12                          ; endereço procP+4
13          ...
```

Suponha que o montador tenha produzido o seguinte código a partir do trecho acima:

```
Endereço  Código       | Programa
--------  -----------   --------------------------
   ...        ...      |     ...
[001000]  80 00 00 00  |     jsr   procP
[001004]  00 00 04 00  |
[001008]  50 00 01 02  |     xor   r1,r2
   ...        ...      |     ...
[004000]               | procP:
          00 00 00 00  |     ds    4
[004008]  01 ff 05 00  |     set   r5,-1
   ...        ...      |     ...
```

Ou seja, o montador monta a instrução de chamada de procedimento no endereço 1000h, e o procedimento `procP` é montado a partir do endereço 4000h. Nesse caso, quando a instrução `jsr` é executada, o endereço de retorno (que vale 1008h) é armazenado na posição de memória 4000h (que é a primeira palavra do procedimento `procP`). A instrução `jsr` também altera o valor de `ip` para 4004h (endereço da primeira instrução do procedimento chamado), de forma que a execução do corpo do procedimento é iniciada.

O retorno do procedimento pode ser executado pela seguinte sequência de instruções:

```
1          ...
2          ld  r1,procP    ; carrega endereço de retorno, previamente
3                          ; armazenado no endereço procP
4          jmp r1          ; e desvia para esse endereço
5          ...
```

A instrução `ld` carrega o registrador `r1` com o valor 1008h, que é o endereço de retorno previamente armazenado no endereço 4000h, e a instrução `jmp` efetua o desvio para esse endereço, o que faz com que a execução do programa que fez a chamada ao procedimento seja retomada.

Essa solução para implementação de procedimentos foi adotada em alguns processadores antigos, como o IBM-1130, da década de 60. Ela no entanto tem várias limitações. Por exemplo, a instrução `jsr` não funcionaria em sistemas operacionais de hoje, pois estes não permitem que sejam realizados acessos para escrita na região de memória que contém código de programa. Um outro problema, bem mais grave conceitualmente, é que esta solução não permite recursão: uma nova chamada ao procedimento de dentro do próprio procedimento faria com que o endereço de retorno da primeira chamada fosse destruído, já que o endereço dessa segunda chamada seria armazenado no mesmo lugar que o da primeira chamada. Na época, uma das linguagens mais utilizadas era FORTRAN-77, que não permitia recursão, e essa limitação não era tão incômoda.

Obviamente essa primeira abordagem não é interessante hoje em dia, pois recursão é uma funcionalidade indispensável em linguagens de programação. Como permitir recursão? Sabemos que recursão envolve o uso de pilhas – a solução portanto é o uso de uma pilha pelo processador.

4.0.1 Instruções que manipulam pilhas

Uma pilha pode ser facilmente implementada no Faíska com um registrador de propósito geral, como `r0`. Inicialmente, ele deve ser inicializado para apontar para uma região de memória disponível. Um dado é empilhado na pilha apontada por `r0` decrementando-se `r0` de 4 e escrevendo-se o dado na nova posição apontada por `r0`. Neste esquema, a pilha "cresce" de endereços altos para endereços baixos:

```
1   ; implementando uma pilha com o registrador r0 como apontador de pilha
2   ; exemplo que empilha valor do registrador r10
3        sub   r0,4            ; reserva espaço para novo elemento
4        st    (r0),r10        ; e coloca valor de r10 no topo da pilha
```

Para retirar o elemento do topo da pilha procede-se à operação inversa: lê-se o valor da palavra apontada por `r0` e incrementa-se `r0` de 4, de forma a que ele aponte para o novo topo da pilha:

```
1   ; implementando uma pilha com o registrador r0 como apontador de pilha
2   ; exemplo que retira o valor do topo da pilha e armazena em r9
3       ld    r9,(r0)        ; carrega r9 com valor no topo da pilha
4       add   r0,4           ; r0 aponta para novo topo da pilha
```

Manipulações de pilhas são tão freqüentes que os processadores incluem instruções específicas e um registrador especial, normalmente chamado `sp` (do inglês *stack pointer*), que funciona como apontador de pilha. No Faíska o apontador de pilha é na verdade um dos registradores de uso geral, `r15`. Para comodidade, a linguagem de montagem aceita `sp` como outro nome do registrador `r15`. Duas instruções são disponíveis no Faíska para manipulação direta da pilha pelo usuário: `push` (empilha registrador) e `pop` (desempilha registrador).

PUSH

Empilha			
Sintaxe	**Operação**	**Flags**	**Codificação**
push r_{fonte}	$sp \leftarrow sp - 4$ $mem[sp] \leftarrow r_{fonte}$	–	31 … 0: | 50 | | | r_{fonte} |

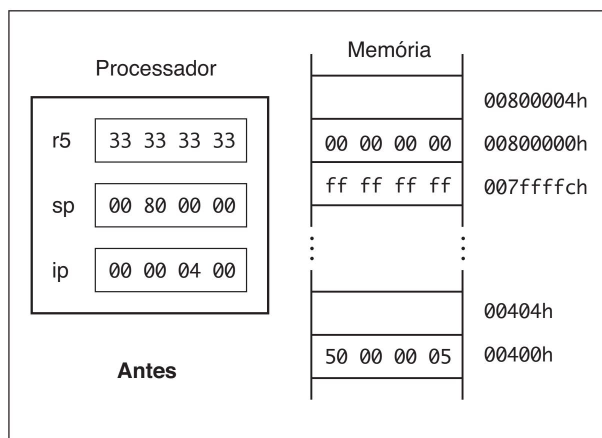
POP

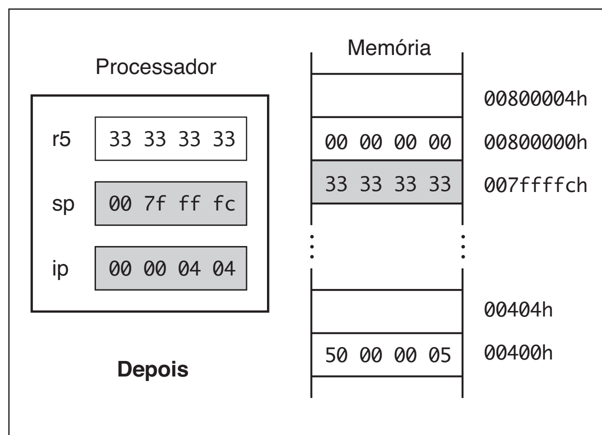
Desempilha			
Sintaxe	**Operação**	**Flags**	**Codificação**
pop	$r_{dest} \leftarrow mem[sp]$ $sp \leftarrow sp + 4$	–	31 … 0: | 51 | r_{dest} | | |

Os operandos das instruções `push` e `pop` são sempre registradores. Assim, somente é possível empilhar e desempilhar palavras inteiras. Conjuntos menores de bits, como bytes, não podem ser empilhados individualmente. O Exemplo 4.1 ilustra a execução da instrução `push`, e o exemplo 4.2 ilustra a execução da instrução `pop`.

Exemplo 4.1 *Uso e execução da instrução* `push`.

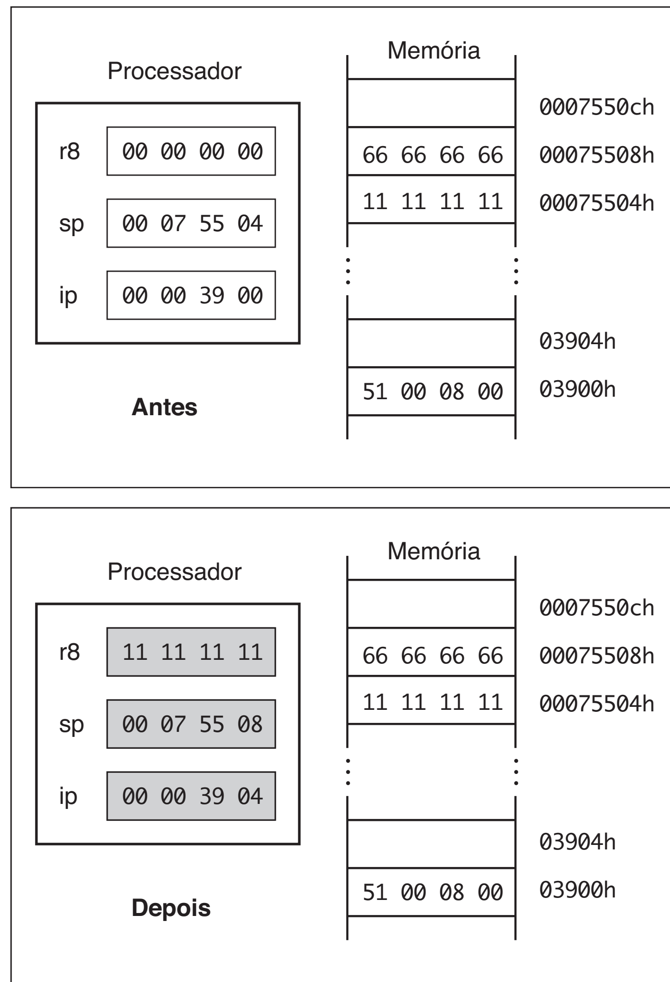
```
Endereço  Código      | Programa
--------  ----------- --------------------------
[000400]  50 00 00 05 |     push r5
```





Exemplo 4.2 *Uso e execução da instrução* `pop`.

```
Endereço  Código      | Programa
--------  ----------- --------------------------
[000400]  51 00 0a 00 |     pop r10
```



É importante notar que, antes de utilizar instruções que manipulam a pilha, é necessário reservar a região de memória que será utilizada pela pilha. Como a pilha cresce de endereços altos para endereços baixos, um exemplo de alocação de espaço para a pilha é:

```
 1  ; aloca uma região da memória para a pilha
 2  ; assume que a pilha utilizará no máximo 1 KByte
 3
 4         TAM_PILHA equ 1024
 5
 6  ; aloca espaço (em área de memória reservada para dados)
 7  fim_pilha:
 8         ds  TAM_PILHA
 9  ini_pilha:
10
11         ...
12
13  ; inicializa apontador de pilha, antes de executar
14  ; qualquer instrução que manipule a pilha
15
16         set  sp,ini_pilha
```

Uma outra opção é alocar a pilha no final da memória física instalada, se essa informação for conhecida:

```
1  ; aloca a pilha no final da memória
2  ; assume que a memória instalada é apenas 64 KBytes
3
4         FINAL_MEMORIA equ 10000h
5
6  ; inicializa apontador de pilha, antes de executar
7  ; qualquer instrução que manipule a pilha
8
9         set  sp,FINAL_MEMORIA
```

4.1 Chamada e retorno de procedimento

A implementação de procedimentos nos processadores atuais faz uso do registrador apontador de pilha. No Faíska são definidas duas instruções específicas, que manipulam a pilha de forma indireta: uma para a chamada de procedimento, e outra para o retorno do procedimento.

CALL

Chamada de procedimento			
Sintaxe	**Operação**	**Flags**	**Codificação**
`call` $expr_{32}$	$sp \leftarrow sp - 4$ $mem[sp] \leftarrow ip + 8$ $ip \leftarrow imd_{32}$	–	31–0: 54 | | | 31–0: imd_{32}
`call` r_{dest}	$sp \leftarrow sp - 4$ $mem[sp] \leftarrow ip + 4$ $ip \leftarrow r_{dest}$	–	31–0: 55 | r_{dest} | |

A instrução chamada de procedimento (`call`) empilha o endereço de retorno e executa o desvio para o início do procedimento alvo, que é codificado na instrução. Como na instrução de desvio incondicional, o endereço alvo (início do procedimento) pode ser especificado através de um valor imediato ou de um registrador. O endereço de retorno empilhado é o endereço da instrução imediatamente seguinte à instrução `call`.

RET

Retorno de procedimento			
Sintaxe	**Operação**	**Flags**	**Codificação**
`ret`	$ip \leftarrow mem[sp]$ $sp \leftarrow sp + 4$	–	31–0: 56 | | |

A instrução retorno de procedimento (`ret`) simplesmente desempilha a palavra no topo da pilha e executa o desvio para o endereço desempilhado, retomando a execução do fluxo de programa que realizou a chamada de procedimento.

Exemplo 4.3 *Vamos ilustrar a execução das instruções* `call` *e* `ret` *com o seguinte trecho de programa, que inclui uma chamada a um procedimento de rótulo* `ProcExemplo`*:*

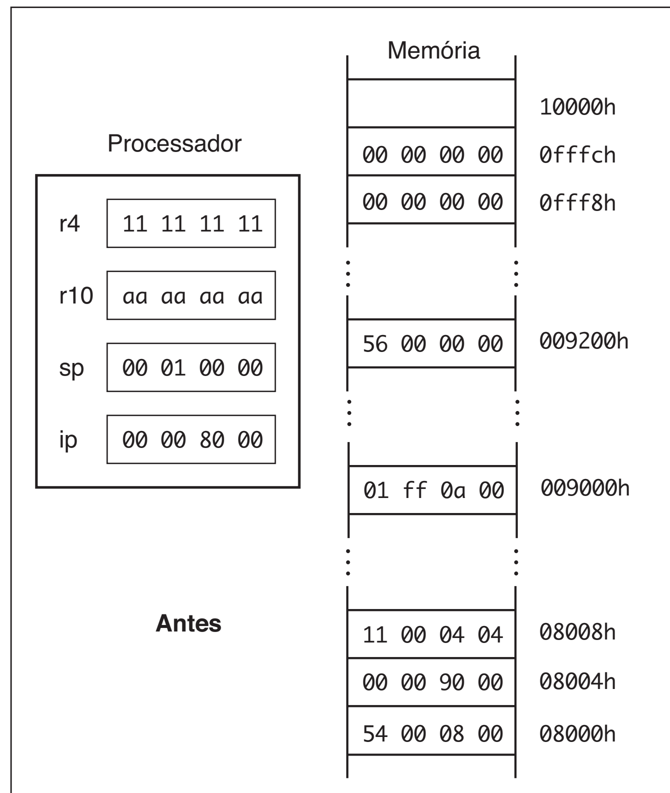
```
1        ...
2        call  ProcExemplo    ; uma chamada a procedimento
3        mov   r4,r5          ; esta é a primeira instrução após a
4                             ; chamada ao procedimento
5        ...
6
7   ProcExemplo:
8        set   r10,-1         ; primeira instrução do procedimento
9        ...
10       ret                  ; procedimento termina, retorna da chamada
```

Suponha que o montador tenha produzido o seguinte código a partir do trecho acima:

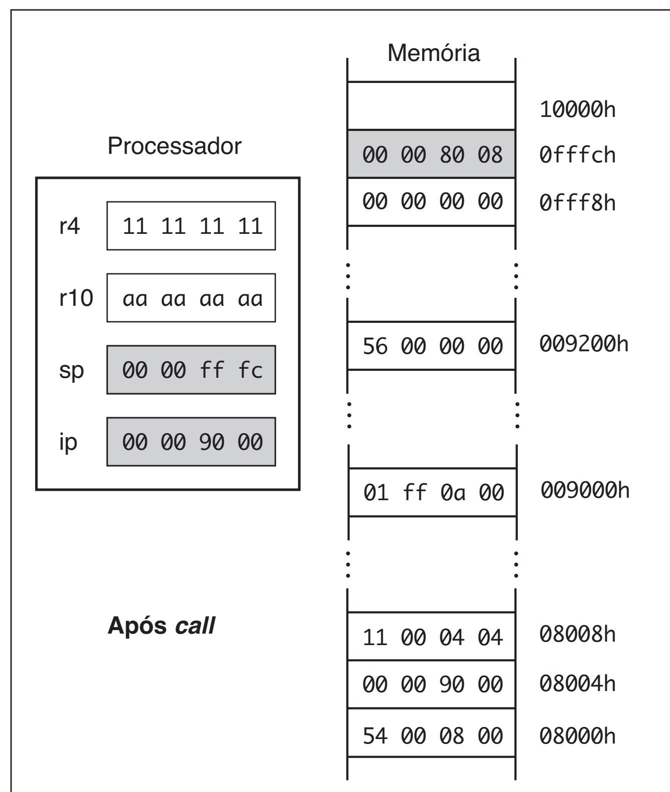
```
Endereço  Código       | Programa
--------  -----------    -------------------------
   ...        ...      |     ...
[008000]  54 00 00 00 |     call ProcExemplo
[008004]  00 00 90 00 |
[008008]  11 00 04 04 |     add  r4,r4
                       |
   ...        ...      |     ...
                       |
[009000]               | ProcExemplo:
          01 ff 0a 00 |     set  r10,-1

   ...        ...      |     ...

[009200]  56 00 00 00 |     ret
   ...        ...      |     ...
```

Ou seja, o procedimento é montado a partir do endereço 9000h, e a chamada ao procedimento é montada no endereço 8000h, de forma que o endereço de retorno para essa chamada é 8008h (endereço da instrução `add`.

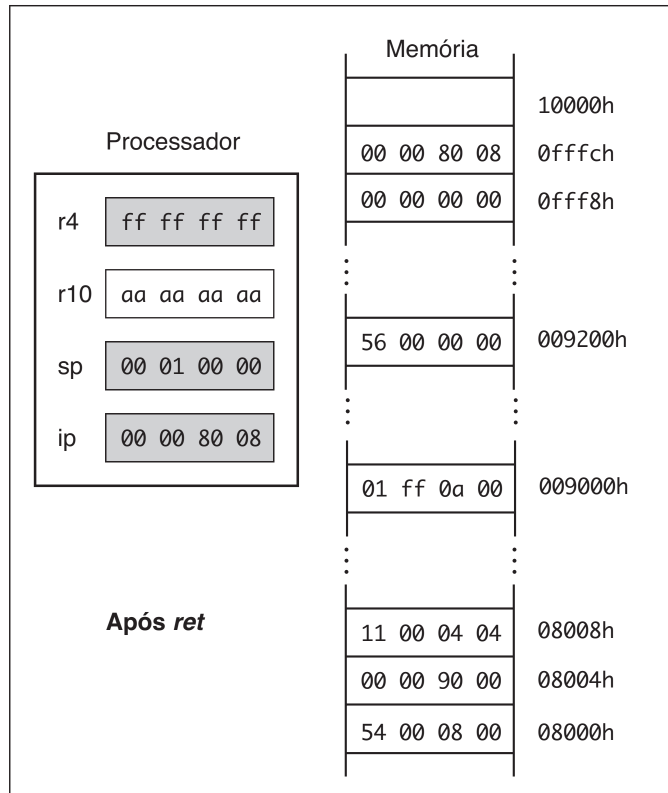
Suponha ainda que o apontador de pilha tenha o valor 10000h antes da execução da instrução `call` e que a configuração dos registradores e memória seja:



A instrução `call` empilha o endereço de retorno e desvia para o início do procedimento; a configuração passa a ser:



A instrução `ret` no endereço 9200h, quando executada, retira da pilha o endereço de retorno e desvia para esse endereço, de forma que após a sua execução a configuração é:



Com o novo valor de `ip`, o programa retoma a execução a partir da instrução seguinte à chamada de procedimento.

Exercício 4.1 Escrever um procedimento para zerar os registradores `r0`, `r1`, `r2` e `r3`.

```
 1  ; ********
 2  ; ZeraRegs
 3  ; ********
 4  ; Zera os registradores r0, r1, r2 e r3
 5  ;   entrada: nenhuma
 6  ;   saída: r0, r1, r2 e r3 zerados
 7  ;   destrói: nada
 8
 9  ZeraRegs:
10          set r0,0
11          set r1,0
12          set r2,0
13          set r3,0
14          ret
```

Um exemplo de chamada ao procedimento ZeraRegs é:

```
1   ; exemplo de chamada de ZeroRegs
2
3       call   ZeraRegs          ; após a chamada, r1=r2=r3=0
```

É interessante ressaltar que em linguagem de montagem o conceito de procedimento é muito mais flexível que em uma linguagem de alto nível. O 'procedimento' é apenas um bloco de instruções que contém uma instrução `ret`, e não é feita qualquer verificação pelo montador se o operando de um comando `call` representa mesmo o endereço inicial de um procedimento. Isto apresenta algumas vantagens mas muitas desvantagens –– é preciso disciplina para programar corretamente. Por outro lado, essa flexibilidade de estruturação permite construções que não são possíveis em linguagens de comando de alto nível. Por exemplo, em linguagem de montagem é possível utilizar o mesmo trecho de código para mais de um "procedimento", como no exemplo seguinte, que apresenta dois procedimentos com um mesmo ponto de retorno: um procedimento para zerar os registradores `r0`, `r1`, `r2`, `r3`, `r4` e `r5`; e outro para zerar apenas os registradores `r0`, `r1`, `r2` e `r3`.

```
 1  ; *********************
 2  ; Zera6Regs e Zera4Regs
 3  ; *********************
 4  ; Zera6Regs zera os registradores r0, r1, r2, r3, r4 e r5
 5  ; Zera4Regs zera os registradores r0, r1, r2 er3
 6  ;   entrada: nenhuma
 7  ;   destrói: nada
 8
 9  ; aqui é o ponto de entrada do primeiro procedimento
10  Zera6Regs:
11         set r5,0
12         set r4,0
13  ; aqui é o ponto de entrada do segundo procedimento
14  Zera4Regs:
15         set r3,0
16         set r2,0
17         set r1,0
18         set r0,0
19         ret
```

4.2 Passagem de parâmetros

Até o momento consideramos apenas procedimentos sem parâmetros. Como podemos implementar a passagem de parâmetros? Uma primeira abordagem é utilizar registradores.

4.2.1 Passagem de parâmetros por registradores

Este método é bastante eficiente, e pode ser usado se o procedimento não é recursivo e há registradores disponíveis em número suficiente para acomodar os parâmetros (o que nem sempre é o caso).

Exercício 4.2 Escrever um procedimento `Preenche`, com funcionalidade similar ao procedimento *memset*, da bilbioteca padrão de C. Ou seja, o procedimento `Preenche` deve preencher uma região de memória com um valor de byte dado como parâmetro. Os parâmetros são passados por registradores; `r0` contém o valor do byte a ser usado no preenchimento, `r1` contém endereço inicial da região de memória, `r2` contém o número de bytes a serem preenchidos com o valor dado.

```
 1  ; ********
 2  ; Preenche
 3  ; ********
 4  ; Preenche região de memória com um byte dado
 5  ;   entrada:
 6  ;       r0 tem byte a ser usado no preenchimento
 7  ;       r1 tem endereço inicial da região de memória
 8  ;       r2 tem o número de bytes da região que devem ser preenchidos
 9  ;   saída: nenhuma
10  ;   destrói: r1, r2 e flags
11
12  Preenche:
13       sub   r2,1           ; continua a preencher?
14       js    final          ; desvia se terminou de preencher
15       stb   (r1),r0        ; preenche um byte com valor dado
16       add   r1,1           ; avança apontador
17       jmp   Preenche
18  final:
19       ret                  ; e retorna, região foi preenchida
```

Para preencher 100 bytes a partir da posição 1000h com o valor 0ffh, podemos utilizar o procedimento `Preenche` da seguinte maneira:

```
1        ...
2                             ; exemplo de chamada ao procedimento Preenche
3        set   r0,0ffh        ; valor de byte para preencher memória
4        set   r1,1000h       ; endereço inicial
5        set   r2,100         ; número de bytes a preencher
6        call Preenche
7        ...
```

4.3 Passagem de parâmetros pela pilha

Apesar de eficiente, a passagem de parâmetros por registradores tem muitas limitações e não pode ser utilizada em todos os casos, particularmente no caso de procedimento recursivos. Nesses casos, é necessário utilizar a pilha para passagem de parâmetros.

Para passar parâmetros utilizando a pilha, devemos empilhar os parâmetros antes da chamada do procedimento. Dentro do procedimento, podemos acessar os parâmetros utilizando o registrador apontador de pilha. Suponha um procedimento `ProcParam` com dois parâmetros *param1* e *param2*. O código abaixo mostra a preparação da chamada do procedimento, com os dois parâmetros sendo empilhados antes da instrução de chamada de procedimento:

```
1  ; supondo que param1 esteja armazenado em r4 e param2 armazenado em r5
2  ; exemplo de chamada de procedimento
3         push r4          ; empilha primeiro parâmetro
4         push r5          ; empilha segundo parâmetro
5         call ProcParam   ; agora efetua a chamada
6         ...
```

A configuração da pilha após a execução da chamada é mostrada na Figura 4.3.

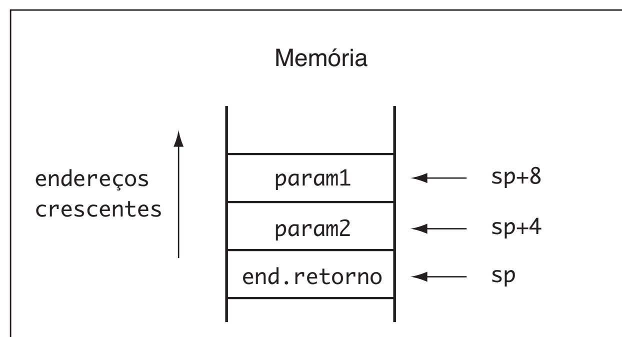


Figura 4.2: Diagrama ilustrando a configuração da pilha durante a execução de um procedimento com dois parâmetros.

Para acessar os parâmetros, o código do procedimento deve usar o registrador apontador de pilha:

```
1         ...
2  ProcParam:
3         ld  r0,(sp+8)    ; carrega primeiro parâmetro em r0
4         ld  r1,(sp+4)    ; e carrega segundo parâmetro em r1
5         ...
```

Para acessar os parâmetros na pilha não deve ser usada a instrução `pop`, pois o endereço de retorno também está na pilha (foi empilhado após os parâmetros, pela ins-

trução `call`) e seria desempilhado. Portanto, os parâmetros empilhados devem permanecer na pilha até o retorno do procedimento, e devem ser desempilhados após o retorno do procedimento. O código completo do exemplo de chamada de procedimento é

```
1  ; supondo que param1 esteja armazenado em r4 e param2 armazenado em r5
2  ; exemplo de chamada de procedimento
3         push r4         ; empilha primeiro parâmetro
4         push r5         ; empilha segundo parâmetro
5         call Proc       ; agora efetua a chamada
6         add  sp,8       ; retira os parâmetros da pilha
```

Note que não é necessário utilizar várias instruções `pop` para retirar os parâmetros da pilha; é mais eficiente manipular diretamente o apontador de pilha com a instrução `add`. Dessa forma, várias palavras podem ser "desempilhadas" com uma única instrução.

Diferentes linguagens de alto nível podem utilizar diferentes regras para empilhar os parâmetros. Nas implementações da linguagem C, por exemplo, os parâmetros são empilhados na ordem inversa em que foram declarados. Para o procedimento `ParamC` declarado abaixo

```
1  void ParamC(int a, int b)
```

os parâmetros seriam empilhados assim:

```
1       ...
2       ld    r0,b          ; último parâmetro declarado
3       push  r0            ; é o primeiro a ser empilhado
4       ld    r0,a          ; primeiro parâmetro declarado
5       push  r0            ; é o último a ser empilhado
6       call  ParamC        ; faz a chamada
7       add   sp,8          ; retira os dois parâmetros da pilha.
8       ...
```

Em linguagem de montagem, podemos fazer a nossa própria convenção sobre a ordem com que empilhamos os parâmetros. Mas se um procedimento escrito em linguagem de montagem vai fazer parte de uma biblioteca, podendo ser invocado por programas escritos em uma linguagem de alto nível, é necessário obedecer às convenções estabelecidas pela implementação da linguagem em questão.

4.4 Retorno de valores de funções

No caso de funções (procedimentos que retornam valores), o valor ou valores podem ser retornados ou em registradores ou pela pilha.

Exercício 4.3 Escrever uma função `Multiplica` que calcula o resultado da multiplicação de dois números inteiros pelo método de adições sucessivas. Os dois operandos são passados nos registradores `r1` e `r2`, e o resultado deve ser retornado no registrador `r0`.

```
 1  ; **********
 2  ; Multiplica
 3  ; **********
 4  ; Calcula o produto de dois números inteiros positivos pelo método de
 5  ; adições sucessivas. Considera que o resultado cabe em um registrador
 6  ; (ou seja, não considera overflow).
 7  ;   entrada:
 8  ;       r1 tem um número inteiro, operando da multiplicação
 9  ;       r2 tem um número inteiro, operando da multiplicação
10  ;   saída:
11  ;       r0 tem o valor do produto dos operandos
12  ;   destrói:
13  ;       r0, r1, r2 e flags
14
15  Multiplica:
16       cmp   r1,r2          ; para minimizar os passos da multiplicação
17       jnc   mult1          ; usa menor valor para controlar repetição
18       mov   r0,r1          ; troca valores de r1 e r2 usando
19       mov   r1,r2          ; r0 como temporário
20       mov   r2,r0
21  mult1:
22       set   r0,0           ; inicializa valor do produto
23  mult2:
24       sub   r2,1           ; vamos realizar r2 adições
25       js    mult3          ; desvia se terminamos
26       add   r0,r1          ; adiciona mais uma parcela
27       jmp   mult2
28  mult3:
29       ret                  ; retorna quando todas as adições terminaram
```

Embora simples, essa função de multiplicação não é muito eficiente. No pior caso, ela executa N operações de adição, onde N é o valor do menor operando. Considerando que os números estão representados em binário, podemos utilizar um outro método. Considere que a operação é $res = op1 \times op2$. Inicie com res igual a zero. Repita, enquanto o valor corrente de $op2$ for maior que zero, os seguintes passos:

- Se o bit 0 (menos significativo) do valor corrente de $op2$ é 1, adicione o valor corrente de $op1$ ao resultado.

- Multiplique $op1$ por 2 e divida $op2$ por 2.

Exercício 4.4 Escrever uma função `MultiplicaBin` que utiliza o método descrito.

```
 1  ; **************
 2  ; MultiplicaBin
 3  ; **************
 4  ; Calcula o produto de dois números inteiros positivos com log N
 5  ; adições sucessivas. Considera que o resultado cabe em um registrador
 6  ; (ou seja, não considera overflow).
 7  ;   entrada:
 8  ;       r1 tem um número inteiro, operando da multiplicação
 9  ;       r2 tem um número inteiro, operando da multiplicação
10  ;   saída:
11  ;       r0 tem o valor do produto dos operandos
12  ;   destrói:
13  ;       r0, r1, r2 e flags
14
15  MultiplicaBin:
16       cmp   r1,r2          ; para minimizar os passos da multiplicação
17       jnc   mult1          ; escolhe o menor valor para controlar
18                            ; a repetição
19       mov   r0,r1          ; troca valores de r1 e r2 usando
20       mov   r1,r2          ; r0 como temporário
21       mov   r2,r0
22  mult1:
23                            ; aqui r1 >= r2; r1 é op1, r2 é op2
24       set   r0,0           ; inicializa valor do produto
25  mult2:
26       shr   r1,1           ; bit 0 de op1 vai para flag C
27       jnc   mult3          ; bit era 0, não adiciona parcela
28       add   r0,r2          ; adiciona parcela (valor corrente de op2)
29  mult3:
30       shl   r2,1           ; multiplica op2 por 2
31       cmp   r1,0           ; terminamos?
32       jnz   mult2          ; não, continua
33       ret                  ; retorna quando todas as adições terminaram
```

Com esse método são realizadas $log_2 N$ adições, onde N é o valor do menor operando, o que representa um enorme ganho de desempenho em relação à implementação anterior da função de multiplicação.

Exercício 4.5 Escrever uma função `ContaUns` que devolve em `r0` o número de bits 1 de uma palavra de 32 bits passada como parâmetro pela pilha.

```
 1  ; ********
 2  ; ContaUns
 3  ; ********
 4  ; Conta o número de bits 1 de uma palavra passada pela pilha
 5  ;   entrada: palavra de 32 bits passada pela pilha
 6  ;   saída: número de bits 1 em r0
 7  ;   destrói: r1 e flags
 8
 9  ContaUns:
10       ld    r1,(sp+4)      ; carrega parâmetro
11       xor   r0,r0          ; zera registrador resultado
12  proxbit:
13       ror   r1,1           ; testa mais um bit
14       jnc   testafim       ; bit deslocado é 1?
15       add   r0,1           ; conta este bit
16  testafim:
17       jnz   proxbit        ; se não verificou todos os bits, continua
18       ret                  ; retorna com valor em r0
```

Vamos ilustrar o uso da função `ContaUns` escrevendo uma outra função que a utiliza.

Exercício 4.6 Escrever uma função `ComparaUns` que verifica se duas palavras de 32 bits passadas como parâmetro na pilha têm ambas o mesmo número de bits 1. Caso tenham, o registrador `r0` deve retornar com o número de bits 1. Caso contrário, `r0` deve retornar com -1.

```
 1  ; **********
 2  ; ComparaUns
 3  ; **********
 4  ; Compara o número de bits 1 de duas palavras
 5  ;   entrada: duas palavras de 32 bits passadas pela pilha
 6  ;   saída: se iguais, número de bits 1 em r0
 7  ;          se diferentes, r0 igual a -1
 8  ;   destrói: r1 e flags
 9  ;   usa: ContaUns
10
11  ComparaUns:
12       ld    r0,(sp+8)      ; carrega primeiro parâmetro
13       push  r0             ; empilha como parâmetro para ContaUm
14       call  ContaUns       ; conta o número de bits 1 do primeiro
15       add   sp,4           ; retira parametro da pilha
16       mov   r1,r0          ; guarda resultado intermediário em r1
17       ld    r0,(sp+4)      ; carrega segundo parâmetro
18       push  r0             ; empilha como parâmetro para ContaUm
19       call  ContaUns       ; conta o número de bits 1 do segundo
20       add   sp,4           ; retira parametro da pilha
21       cmp   r0,r1          ; compara resultados
22       jz    final          ; se iguais, retorna
23       set   r0,-1          ; caso contrário, indica com r0=-1
24  final:
25       ret                  ; retorna com valor em r0
```

Esta versão de `ComparaUns` apresenta um problema, porque `ContaUns` destrói o registrador `r1`, que é usado em `ComparaUns` para guardar o número de bits 1 do primeiro parâmetro. Quando `ContaUns` é invocado pela segunda vez, o resultado da primeira chamada é perdido.

Uma solução óbvia para este problema é utilizar um outro registrador que não `r1`, que sabidamente é alterado por `ContaUns`, para armazenar o resultado da primeira chamada. No entanto, nem sempre há um outro registrador disponível para ser usado. Em uma aplicação embarcada, pequena, pode ser apropriado realizar um mapeamento do uso dos registradores, com o objetivo de minimizar o conflito de registradores utilizados pelos diversos procedimentos do programa. Mas no caso geral esse é um problema difícil, e uma outra solução é geralmente utilizada: o uso da pilha para armazenamento temporário de valores. Antes da chamada do procedimento os valores dos registradores que se deseja preservar devem ser empilhados, e após o retorno do procedimento os registradores devem ser restaurados com os valores anteriores. Vamos re-escrever o procedimento `ComparaUns` armazenando o registrador `r1` temporariamente na pilha:

```
1   ; **********
2   ; ComparaUns -- segunda versão
3   ; **********
4   ; Compara o número de bits 1 de duas palavras
5   ;   entrada: duas palavras de 32 bits passadas pela pilha
6   ;   saída: se iguais, número de bits 1 em r0
7   ;          se diferentes, r0 igual a -1
8   ;   destrói: r1
9
10  ComparaUns:
11       ld    r0,(sp+4)      ; carrega primeiro parâmetro
12       push  r0             ; empilha como parâmetro para ContaUm
13       call  ContaUns       ; conta bits 1 do primeiro parâmetro
14       add   sp,4           ; retira parâmetro da pilha
15       push  r0             ; salva resultado para nao ser destruído
16       ld    r0,(sp+12)     ; carrega segundo parâmetro
17                            ; note que o deslocamento em relação ao
18                            ; topo da pilha mudou (12 ao invés de 8)
19       push  r0             ; empilha como parâmetro para ContaUm
20       call  ContaUns       ; conta bits 1 do segundo parâmetro
21       add   sp,4           ; retira parametro da pilha
22       pop   r1             ; recupera primeiro resultado
23       cmp   r0,r1          ; compara resultados
24       jz    final          ; se iguais, retorna
25       set   r1,-1          ; caso contrário, indica com r1=-1
26  final:
27       ret                  ; retorna com valor em r0
```

Note no entanto que essa solução de armazenamento temporário faz com que os deslocamentos dos parâmetros em relação ao apontador de pilha sejam modificados enquanto os valores temporários estão na pilha. Ou seja, um parâmetro de um procedimento que normalmente é acessado no endereço `(sp+4)` passa a ser acessado, após o armazenamento temporário de dois registradores na pilha, no endereço `(sp+12)`. Se o procedimento é extenso, essa solução pode deixar o código mais complexo. Na seção 4.4.2 veremos uma forma mais estruturada de armazenar valores temporários dentro de um procedimento.

Exercício 4.7 Escrever um procedimento que inverte uma cadeia de caracteres cujo endereço inicial é passado na pilha. O final da cadeia é marcado com um byte de valor 0.

```
1  ; *******
2  ; Inverte
3  ; *******
4  ; Inverte uma cadeia de caracteres
5  ;   entrada: endereço da cadeia; final da cadeia é byte 0
6  ;   saída: nenhuma
7  ;   destrói: r1 e flags
8
9  Inverte:
10      ld    r2,(sp+4)      ; carrega endereço inicial da cadeia
11                           ; primeiro procura final da cadeia
12      mov   r1,r2          ; guarda apontador inicio da cadeia
13 achaFinal:
14      ldb   r0,(r2)
15      add   r2,1           ; avança apontador
16      cmp   r0,0           ; chegamos ao final?
17      jnz   achaFinal      ; desvia se ainda nao é final
18      sub   r2,2           ; r2 aponta para último elemento da cadeia
19                           ; agora começa a a inversão, que continua
20                           ; enquanto o apontador do inicio é menor do
21                           ; que apontador do final da cadeia
22 proxchar:
23      cmp   r1,r2          ; verifica se cadeia terminou
24      jnc    final          ; sim, terminou, vai retornar
25      ldb   r0,(r1)        ; elemento do início
26      ldb   r3,(r2)        ; elemento do final
27      stb   (r1),r3
28      stb   (r2),r0        ; faz a troca
29      add   r1,1           ; avança ponteiro do inicio
30      sub   r2,1           ; retrocede ponteiro do final
31      jmp   proxchar
32 final:
33      ret
```

Esta solução tem um detalhe interessante. A execução de um desvio é em geral mais demorada se o desvio é tomado (ou seja, se a condição do desvio é verdadeira) do que se o desvio não é tomado e a execução sequencial de instruções continua. No processador Intel Pentium, por exemplo, a execução de um desvio condicional pode ser até 20 vezes mais demorada se o desvio é tomado. No procedimento `Inverte`, o laço onde é feita a inversão de caracteres poderia ser re-escrito com o sentido da condição das linhas 23~24 invertido:

```
22  proxchar:
23       cmp   r2,r1          ; verifica se cadeia terminou
24       jnc   continua       ; nao terminou, vai inverter
25       ret
26  continua:
27       ldb   r0,(r1)        ; elemento do início
28       ldb   r3,(r2)        ; elemento do final
29       stb   (r1),r3
30       stb   (r2),r0        ; faz a troca
31       add   r1,1           ; avança ponteiro do inicio
32       sub   r2,1           ; retrocede ponteiro do final
33       jmp   proxchar
```

O trecho acima executa exatamente o mesmo trabalho que o trecho original, mas leva muito mais tempo, pois na maioria das vezes a condição da linha 3 é verdadeira e o processador toma o desvio. Portanto, este simples "detalhe" faz com que o programa execute mais lentamente. É necessário estar sempre atento ao sentido da condição dentro de laços, especialmente no caso de laços encaixados, quando o efeito danoso é multiplicado.

4.4.1 Passagem de parâmetros por referência e por valor

Parâmetros em linguagens de alto nível como C ou Pascal podem ser passados por referência ou por valor, como na declaração abaixo, em que os parâmetros `c` e `v` são passados por valor, e o parâmetro `r` é passado por referência:

```
void ExemploValorRef(char c, int v, int *r)
```

Em linguagens de alto nível, como C ou C++, o compilador cuida de tratar corretamente esses dois tipos de parâmetros. Em linguagem de montagem, o controle é feito exclusivamente pelo programador. Se um *valor* é passado para um procedimento (pela pilha ou por registrador), o esquema é obviamente de passagem por valor. Se um *endereço* é passado como parâmetro, de forma que o procedimento pode alterar o valor contido naquele endereço, a passagem é por referência. Mas como em linguagem de montagem não há verificação de tipos, é necessário tomar muito cuidado para não cometer erros na passagem de parâmetros.

Como exemplo, uma chamada ao procedimento `ExemploValorRef` acima

```
ExemploValorRef('a', val, &ref);
```

poderia ser traduzida em linguagem de montagem para

```
1        ...
2        set  r0,61h          ; empilha primeiro parâmetro
3                             ; note que a instrução set é usada
4
5        push r0              ; valor da constante 'a' é empilhado
6        ld   r0,val          ; empilha valor do segundo parâmetro
7                             ; note que a instrução ld é usada
8
9        push r0              ; valor contido no endereço val é empilhado
10       set  r0,ref          ; empilha endereço do terceiro parâmetro
11                            ; note que a instrução set é usada
12
13       push r0              ; o endereço do rótulo ref é empilhado
14       call ExemploValorRef
15       add  sp, 12          ; desempilha parâmetros
16       ...
```

No caso dos parâmetros `c` e `v`, passados por valor, são empilhados os valores dos argumentos, respectivamente o valor da constante `61h` (codificação ASCII do caractere 'a') e o valor da variável `val`. Já no caso do parâmetro `r`, é empilhado o endereço da variável passada como argumento.

Exercício 4.8 Implementar o procedimento `troca`, similar ao utilizado na Figura 4.1 no início deste capítulo, para trocar os valores de duas variáveis inteiras passadas como referência pela pilha.

```
1   ; *****
2   ; Troca
3   ; *****
4   ; Troca os valores de duas variáveis passadas por referêcia na pilha
5   ;   entrada: dois endereços de variáveis inteiras, pela pilha
6   ;   saída: nenhuma
7   ;   destrói: r0,r1,r2,r3
8
9   Troca:
10       ld    r0,(sp+4)      ; carrega endereço da variável a
11       ld    r1,(sp+8)      ; carrega endereço da variável b
12       ld    r2,(r0)        ; valor da variável a
13       ld    r3,(r1)        ; valor da variável b
14       st    (r0),r3        ; armazena valor de b no endereço de a
15       st    (r1),r2        ; armazena valor de a no endereço de b
16       ret
```

Note que não nesse caso a variável temporária `tmp` da Figura 4.1 não é necessária. O seguinte trecho de código ilustra uma chamada ao procedimento `Troca`:

```
 1
 2       ...
 3
 4       set   r0,x          ; carrega endereço da variável x
 5       push  r0            ; empilha parâmtro
 6       set   r0,y          ; carrega endereço da variável y
 7       push  r0            ; empilha parâmetro
 8       call  Troca         ; faz a chamada
 9       add   sp,8          ; retira os dois parâmetros da pilha
10
11       ...
12
13  ; aqui estão definidas as variáveis inteiras
14  x:   ds    4
15  y:   ds    4
16
```

Esse exemplo mostra ainda a alocação de espaço para duas variáveis inteiras `x` e `y`. Esse tipo de alocação corresponde à definição de variáveis globais do programa, como as definidas na linha 4 da Figura 4.1. Esse tipo de alocação é chamado de *estático*, pois uma vez alocado permanece alocado até o término da execução do programa. Para variáveis que serão utilizadas apenas dentro de um procedimento a alocação estática não é adequada, uma vez, entre outros problemas, não permitiria recursão.

4.4.2 Variáveis locais a procedimentos

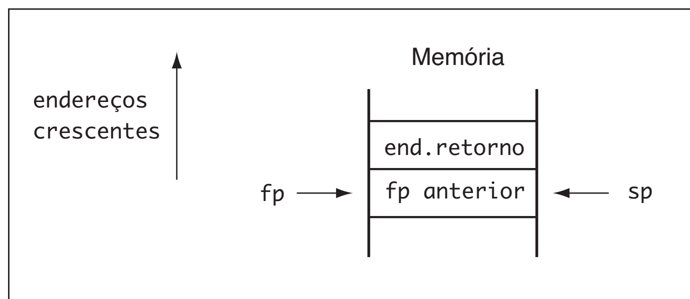
Variáveis locais a um procedimento também devem ser alocadas na pilha se o procedimento for recursivo, ou se utiliza muitas variáveis locais e o número de registradores não é suficiente para alocá-las. O espaço para as variáveis locais deve ser reservado na entrada do procedimento, e desalocado ao final do procedimento.

Para alocar espaço na pilha, basta decrementar `sp` pelo número de bytes desejado. No entanto, ao reservarmos espaço para variáveis locais na pilha dentro de um procedimento, o deslocamento necessário para acessar os parâmetros do procedimento se altera, em relação a `sp`, de forma que os acessos a parâmetros deve levar em conta o número de bytes reservados na pilha para variáveis locais. Ou seja, apesar de o endereço normal para acessar um dado parâmetro ser `(sp+4)`, se houver uma variável temporária de uma palavra na pilha esse endereço passa a ser `(sp+8)`. Para evitar que os deslocamentos sejam alterados durante a execução do procedimento, gerando confusão, é comum utilizarmos mais um registrador, geralmente chamado apontador de quadro (em inglês, *frame pointer*), que é mantido fixo, apontando para a posição que `sp` aponta no início do procedimento. Dessa forma, os deslocamentos dos parâmetros em relação a `fp` se mantêm fixos durante toda a execução do procedimento. No Faíska, o registrador `r14` é usado para esse fim, e a linguagem de montagem aceita o nome `fp` como sinônimo de `r14`. No início do procedimento, devemos preparar o uso do apontador de quadro, copiando o valor do apontador de pilha. Mas como o apontador de quadro pode es-

tar sendo utilizado (provavelmente em outra invocação de procedimento), é necessário preservar o seu valor, empilhando-o:

```
1  ; preparando o apontador de quadro para uso, no início do procedimento
2       ...
3       push  fp              ; preserva valor anterior de fp
4       mov   fp,sp           ; e faz cópia do apontador da pilha corrente
5       ...
```

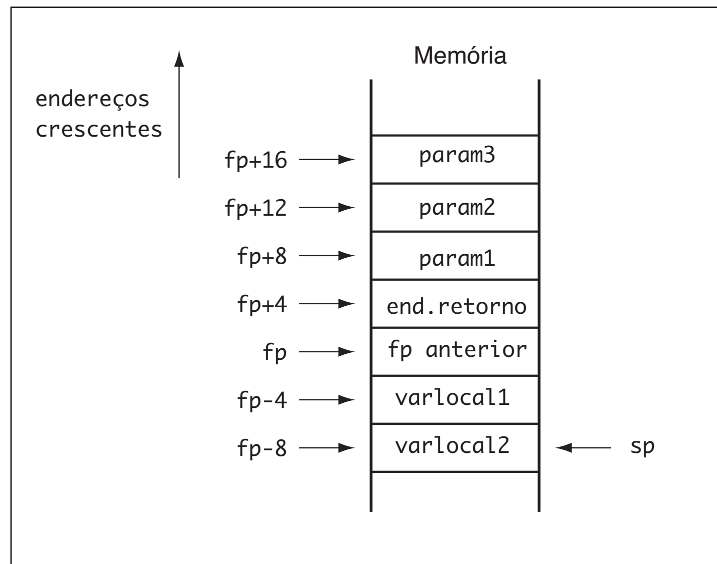
A figura abaixo mostra a configuração da pilha e dos registradores sp e fp após a preparação do apontador de quadro. Note que, em razão do empilhamento do valor antigo do apontador de quadro fp, o primeiro parâmetro (se o procedimento tem parâmetro) está na posição fp+8 (ou sp+8, já que o apontador de pilha e o apontador de quadro têm o mesmo valor nesse momento):



A partir desse ponto, o apontador de quadro é mantido fixo, enquanto o apontador de pilha pode ser alterado para reservar espaço para variáveis locais:

```
1  ; preparando o apontador de quadro para uso, no início do procedimento
2       ...
3       push  fp              ; preserva valor anterior de fp
4       mov   fp,sp           ; e faz cópia do apontador da pilha corrente
5       sub   sp,8            ; reserva espaço na pilha para duas palavras,
6                             ; correspondendo a duas variáveis locais
7       ...
```

Por exemplo, durante a execução de um procedimento com três parâmetros e duas variáveis locais, após a preparação do apontador de quadro e reserva de espaço para as variáveis locais a pilha tem a configuração mostrada na figura abaixo:



Note que, em relação a `fp`, parâmetros têm deslocamentos positivos e variáveis locais têm deslocamentos negativos, e esses deslocamentos se mantêm constantes durante a execução do procedimento, independente de mais valores serem empilhados.

Ao final do procedimento, a preparação do apontador de quadro deve ser desfeita. Para isso basta executar, logo antes do retorno do procedimento:

```
1   ; desfazendo a preparação do apontador de quadro
2        ...
3        mov   sp,fp          ; volta sp ao ponto de origem
4        pop   fp             ; recupera valor de fp anterior
5        ret                  ; e retorna
6        ...
```

Exercício 4.9 Re-escrever o procedimento `ComparaUns` usando uma variável local.

```
1  ; **********
2  ; ComparaUns --- versão com variável local
3  ; **********
4  ; compara o número de bits 1 de duas palavras passadas como parâmetro
5  ;   entrada: duas palavras de 32 bits passadas pela pilha
6  ;   saída: se iguais, número de bits 1 em r0 e flag C desligada
7  ;          se diferentes, flag C ligada
8  ;   destrói: r1 e flags
9  ;   usa: ContaUns
10
11 ComparaUns:
12      push  fp             ; guarda fp antigo
13      mov   fp,sp          ; apontador de quadro
14      sub   sp,4           ; reserva espaço para variável local i
15      ld    r0,(fp+8)      ; carrega primeiro parâmetro
16      push  r0             ; empilha como parâmetro para ContaUm
17      call  ContaUns       ; conta bits 1 do primeiro parâmetro
18      add   sp,4           ; retira parâmetro da pilha
19      st    ,r0            ; guarda resultado na variável local i
20      ld    r0,(fp+4)      ; carrega segundo parâmetro
21      push  r0             ; empilha como parâmetro para ContaUm
22      call  ContaUns       ; conta bits 1 do segundo parâmetro
23      add   sp,4           ; retira parametro da pilha
24      ld    r1,(fp-2)      ; recupera resultado da primeira chamada
25      cmp   r0,r1          ; compara resultados
26      jz    final          ; se iguais, pode retornar
27      set   r1,-1          ; caso contrário, indica com r1=-1
28 final:
29      mov   sp,fp          ; desaloca variável local
30      pop   fp
31      ret                  ; retorna com valor em r0
```

4.5 Recursão

Recursão é uma técnicas de programação muito utilizada, sendo apropriada quando a solução de um problema depende da solução de instâncias menores do mesmo problema. Agora que sabemos como utilizar a pilha para passar parâmetros e guardar variáveis locais, podemos escrever procedimentos que utilizam recursão.

Um exemplo muito comum quando estudamos recursão é o cálculo do fatorial de um número inteiro positivo. O fatorial de um inteiro positivo n é definido como o produto de todos os números inteiros positivos menores ou iguais a n:

$$fatorial(n) = \begin{cases} 1 & se\ n \leq 1; \\ n \times fatorial(n-1) & se\ n > 1. \end{cases}$$

Uma implementação recursiva em C é:

```
1  int fatorial(int n)
2  {
3      if (n<=1)
4          return 1;
5      else
6          return n*fatorial(n-1);
7  }
```

Uma possível implementação em linguagem de montagem do Faíska, fazendo uma tradução direta da implementação em C é:

```
 1  ; ********
 2  ; Fatorial
 3  ; ********
 4  ; Calcula o fatorial de um número inteiro positivo
 5  ; para multiplicação, utiliza função MultiplicaBin, descrita
 6  ; anteriormente
 7  ;   entrada: valor de n>=0, passado pela pilha
 8  ;   saída: registrador r0 tem valor de fatorial(n)
 9  ;   destrói: r1
10
11  Fatorial:
12       ld    r0,(sp+4)      ; valor de n
13       cmp   r0,1
14       ja    fat1
15       set   r0,1
16       ret                  ; retorna fatorial(0) ou fatorial(1)
17  fat1:
18       sub   r0,1           ; calcula recursivamente fatorial(n-1)
19       push  r0             ; empilha parâmetro
20       call  Fatorial       ; chama recursivamente
21       add   sp,4           ; retira parâmetro da pilha
22                            ; neste ponto, r0 tem valor de fatorial(n-1)
23       ld    r1,(sp+4)      ; recupera novamente valor de n
24       mov   r2,r0          ; argumentos para multiplica em r1 e r2
25       call  MultiplicaBin  ; efetua produto
26       ret                  ; e retorna com valor em r0
```

Um outro exemplo também utilizado quando estudamos recursão é o cálculo do *n-ésimo* número da sequência de Fibonacci. A sequência de Fibonacci é definida como:

$$fib(n) = \begin{cases} 0 & se\ n \leq 0; \\ 1 & se\ n \leq 1; \\ fib(n-1) + fib(n-2) & se\ n > 1. \end{cases}$$

Uma implementação recursiva em C é:

```
1   int fib(int n)
2   {
3       if (n<=1)
4           return n;
5       else
6           return fib(n-1)+fib(n-2);
7   }
```

Fazendo uma tradução direta da implementação em C, uma implementação em linguagem de montagem do Faíska seria:

```
 1  ; *********
 2  ; Fibonacci
 3  ; *********
 4  ; Calcula o n-ésimo número da sequência de Fibonacci usando recursão
 5  ;
 6  ;   entrada: valor de n, passado pela pilha
 7  ;   saída: valor de fib(n), no registrador r0
 8  ;   destrói: r1
 9
10  Fibonacci:
11       ld    r0,(sp+4)      ; valor de n
12       cmp   r0,1
13       ja    fib1
14       ret                  ; retorna valor de fib(0) ou fib(1)
15  fib1:
16       sub   r0,1           ; vamos calcular recursivamente fib(n-1)
17       push  r0             ; empilha parâmetro
18       call  Fibonacci      ; chama recursivamente
19       add   sp,4           ; retira parâmetro da pilha
20                            ; neste ponto, r0 tem valor de fib(n-1)
21       push  r0             ; armazena temporariamente valor de fib(n-1)
22       ld    r0,(sp+8)      ; recupera novamente valor de n; note que
23                            ; o deslocamento agora é 8
24       sub   r0,2           ; calcula recursivamente fib(n-2)
25       push  r0
26       call  Fibonacci      ; chama recursivamente
27       add   sp,4           ; retira parâmetro da pilha
28                            ; neste ponto, r0 tem valor de fib(n-2)
29       pop   r1             ; recupera valor de fib(n-1)
30       add   r0,r1          ; calcula valor de fib(n)
31       ret                  ; e retorna com valor em r0
```

Embora os dois exemplos de procedimentos recursivos que vimos sejam bons em relação à didática, por serem fáceis de entender e codificar, é preciso ressaltar que as versões iterativas desses procedimentos são muito mais eficientes do que as versões recursivas.

4.6 Exercícios adicionais

4.6.1. Escreva um procedimento em linguagem de montagem do Faíska que receba uma palavra de 32 bits no registrador `r1` e determine a *paridade* dessa palavra de memória. A paridade é definida como 1 se o número de bits dessa palavra de memória é par, e 0 se o número de bits dessa palavra de memória é ímpar. O procedimento deve retornar o resultado em `r0`.

4.6.2. Implemente em linguagem de montagem do Faíska as funções de biblioteca de C abaixo. Considere que o valor de retorno é feito pelo registrador `r0`, e lembre que as cadeias de caracteres em C são terminadas por um byte de valor 0, e a constante NULL seja definida como uma palavra de valor 0.

- `int strlen(char *s)`, que retorna o número de caracteres da cadeia `s`.
- `char* strchr(const char *s, int c)`, que retorna o endereço da primeira ocorrência do caractere `c` na cadeia de caracteres `s`, ou NULL se o caractere `c` não ocorre em `s`.
- `void strcpy(char *s1, const char *s2)`, que copia a cadeia de caracteres `s2` em `s1` (note que o caractere terminador da cadeia deve ser também copiado).
- `int strcmp(const char *s1, const char *s2)`, que compara lexicograficamente duas cadeias de caracteres e retorna um valor inteiro maior que zero se a cadeia `s1` é maior (lexicograficamente) do que a cadeia `s2`, retorna um valor menor que zero se `s1` é menor que `s2` e retorna zero se as cadeias são iguais.

4.6.3. Escreva um procedimento em linguagem de montagem do Faíska que implemente uma operação de deslocamento de K bits para a direita (*shift right*) em uma palavra de 256 bits armazenada na memória em bytes sucessivos. Os seguintes argumentos são passados pela pilha (empilhados na ordem dada):

- endereço inicial da palavra
- o valor de K ($0 \leq K \leq 256$)

4.6.4. Escreva um procedimento `Impares` que receba como parâmetro, pela pilha, o endereço de um vetor de inteiros sem sinal, cujo comprimento é também passado como parâmetro pela pilha. O procedimento deve retornar no registrador r0 o número de elementos ímpares do vetor.

4.6.5. Escreva uma versão iterativa da função `Fatorial`.

4.6.6. Escreva uma versão iterativa da função `Fibonacci`.