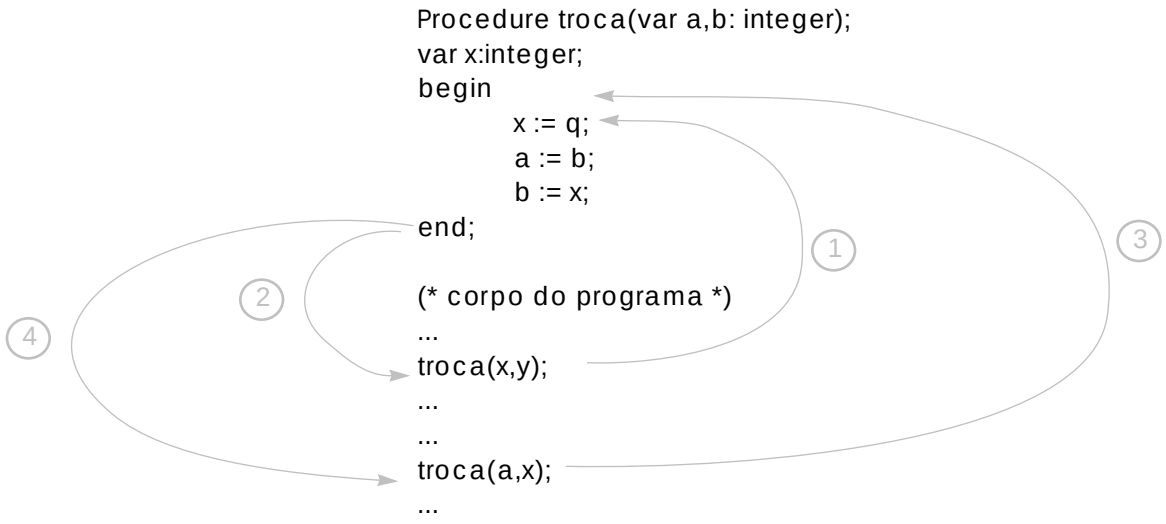
5 *Procedimentos e funções*

Nesta seção vamos estudar as facilidades que os processadores oferecem para a implementação do conceito de procedimentos. Considere o que ocorre na execução de um procedimento em Pascal como o abaixo. O procedimento é um trecho de código que deve ser executado quando invocado. O diagrama da figura abaixo ilustra de forma esquemática o que deve acontecer quando o procedimento *troca* é invocado em dois pontos distintos de um mesmo programa.



Como esperado, a cada invocação do procedimento o programa deve

executar um desvio para o início do procedimento (indicações 1 e 3 no diagrama). E ao final da execução do procedimento deve ser feito um desvio para a instrução seguinte à chamada do procedimento (2 e 4 no diagrama). E aí aparece a dificuldade de se implementar procedimentos com as instruções de desvio que vimos até agora: a cada chamada de procedimento, o retorno deve ser feito para endereços diferentes.

Uma primeira abordagem para a implementação de procedimentos é a seguinte. A chamada de procedimento é implementada por uma instrução especial que armazena o *endereço de retorno* (endereço da instrução seguinte à chamada de procedimento) na primeira palavra do procedimento. Mais especificamente, suponha que exista uma instrução especial JSR (*desvia para subrotina),* que tem como operando o endereço do procedimento a ser invocado:

```
jsr     rótulo
```

e que a operação efetuada pelo processador é

mem[rótulo] ← *ip + 4*
ip ← *rótulo + 4*

Ou seja, no trecho de programa abaixo

```
        add     r1,r0           ; uma instrução qualquer
        jsr     troca           ; chamada de procedimento
; este é o endereço de retorno
ret_aqui:
        add     r0,4            ; uma outra instrução qualquer
```

a execução da instrução *jsr* faz com que o o endereço do rótulo *ret_aqui* seja armazenado na posição de memória *troca.*

Na montagem do programa, o montador deixa livre a primeira palavra de memória do procedimento, montando o código a partir da segunda palavra de memória. Com esse esquema, quando uma chamada de procedimento é executada, o endereço de retorno para aquela chamada fica armazenado em uma posição de memória conhecida. O retorno do procedimento pode ser executado pela seguinte seqüência de

instruções:

```
ld      r1, troca
jmp     r1
```

Esta solução foi adotada em alguns processadores antigos, como o IBM1130 da década de 60. Ela no entanto tem várias limitações. Por exemplo, ela não funciona se o código do programa está sendo executado a partir de uma memória de leitura apenas, como ROMs ou EPROMs, muito comuns hoje em dia. Um outro problema, bem mais grave, 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 (a linguagem FORTRAN, utilizada na época, não permitia recursão, e essa limitação não era tão incômoda).

Como permitir recursão? Sabemos que recursão envolve o uso de pilhas – a solução portanto é o uso de uma pilha pelo processador.

5.1 Instruções que manipulam a pilha

Uma pilha pode ser implementada no Faíska com um registrador de propósito geral, como *r0* . Inicialmente, ele deve ser inicializado para apontar para uma região de memória disponível para escrita. 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). 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.

Manipulações de pilhas são tão freqüentes que os processadores modernos incluem instruções específicas e um registrador especial, normalmente chamado *sp* (do inglês *stack pointer*), que funciona como um apontador de pilha. No Faíska o apontador de pilha é na verdade um dos registradores de uso geral, *r31*. Para comodidade, a linguagem de montagem aceita *sp* como outro nome do registrador *r31*. Duas

operações são disponíveis para manipulação direta da pilha pelo usuário: *push* e *pop.*

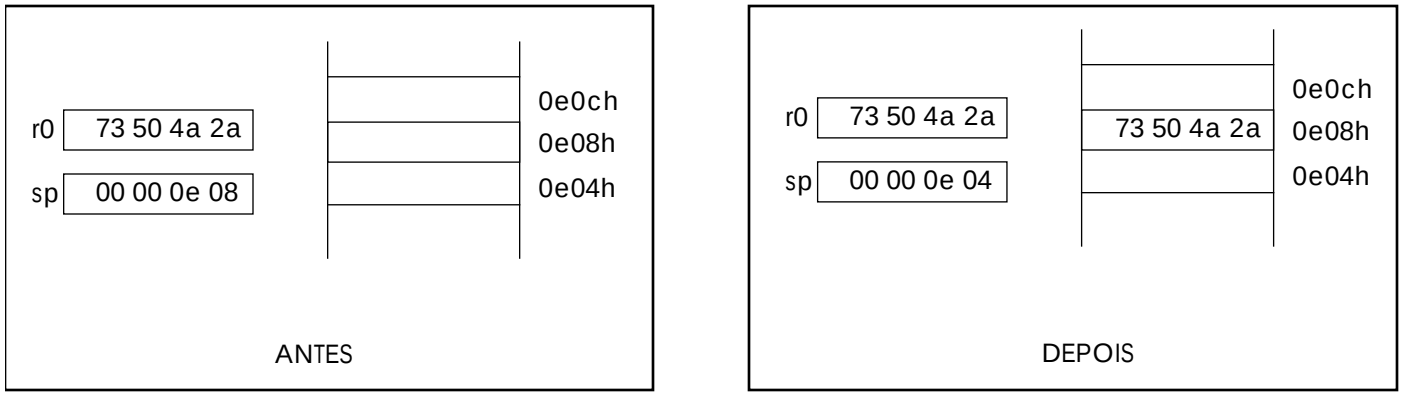
PUSH	Empilha			
	Formato	Operação	Flags	Codificação
	push *rsrc*	*sp* ← *sp* – 4 *mem[sp]* ← *rsrc*	–	40 | – | – | rsrc

POP	Desempilha			
	Formato	Operação	Flags	Codificação
	pop *rdst*	*rdst* ← *mem[sp]* *sp* ← *sp* + 4	–	41 | – | rdst | –

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. A figura ilustra a execução das instruções *push* e *pop.*

Exemplo 5.1:

```
push    r0              ; exemplo de instrução push
```



5.2 Chamada e retorno de procedimento

A implementação de procedimentos em processadores modernos faz uso do registrador apontador de pilha. No Faíska duas instruções específicas, que manipulam a pilha de forma indireta, são utilizadas. Uma para a chamada de procedimento, que empilha o endereço de retorno e desvia para o início do procedimento; e uma para o retorno do procedimento, que retira o endereço de retorno da pilha e executa o desvio correspondente:

CALL	Chama procedimento			
	Formato	Operação	Flags	Codificação
	call *label*	*sp* ← *sp* – *4* *mem[sp]* ← *ip* *ip* ← *imm32*	–	42 | – | – | – imm32
	call *rdst*	*sp* ← *sp* – *4* *mem[sp]* ← *ip* *ip* ← *rdst*	–	43 | – | rdst | –

RET	Retorno de procedimento			
	Formato	Operação	Flags	Codificação
	ret	*ip* ← *mem[sp]* *sp* ← *sp + 4*	–	44 | – | rdst | –

A instrução chamada de procedimento empilha o endereço de retorno na pilha e executa o desvio para o início do procedimento. Como na instrução de desvio incondicional, o endereço alvo (início do procedimento) pode ser especificado através de um rótulo ou de um registrador.

A instrução retorno de procedimento simplesmente desempilha o endereço de retorno da pilha e executa o desvio correspondente.

Exemplo 5.2: Procedimento para zerar os registradores r0, r1, r2 e r3.

```
; ZeraRegs                                                          1
;       procedimento para zerar os registradores r0, r1, r2 e r3    2
;       entrada: nenhuma                                            3
;       saída: r0, r1, r2 e r3 zerados                              4
;       destroi: nada                                               5
ZeraRegs:                                                           6
        set     r0, 0                                               7
        set     r1, 0                                               8
        set     r2, 0                                               9
        set     r3, 0                                               10
        ret                                                         11
```

É interessante ressaltar que em linguagem de montagem o conceito de procedimento é muito mais flexível que em uma linguagem de alto nível: não é feita qualquer verificação se o operando de uma instrução call representa mesmo o endereço inicial de um procedimento (isto apresenta algumas vantagens mas muitas desvantagens – é preciso muita disciplina para programar corretamente). Assim, é possível utilizar o mesmo trecho de código para mais de um “procedimento”, como no

exemplo seguinte.

Exemplo 5.3: Dois procedimentos: um para zerar os registradores r0, r1, r2, r3, r4 e r5; e outro para zerar apenas os registradores r0, r1, r2 e r3.

```
; Zera6Regs e Zero4Regs                                    1
;       Zera6Regs zera os registradores r0, r1, r2, r3, r4 e r5   2
;       Zera4Regs zera os registradores r0, r1, r2 er3      3
;       entrada: nenhuma                                   4
;       destroi: nada                                      5
; aqui é o ponto de entrada do primeiro procedimento       6
Zera6Regs:                                                 7
        set     r5, 0                                      8
        set     r4, 0                                      9
; aqui é o ponto de entrada do segundo procedimento        10
Zera4Regs:                                                 11
        set     r3, 0                                      12
        set     r2, 0                                      13
        set     r1, 0                                      14
        set     r0, 0                                      15
        ret                                                16
```

5.3 Passagem de parâmetros

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

5.3.1 Passagem de parâmetros por registradores

Este método é bastante eficiente, e pode ser usado se o procedimento não é recursivo e o número de parâmetros é pequeno em relação ao número de registradores disponíveis.

Exemplo...

5.3.2 Passagem de parâmetros pela pilha

No caso geral, a melhor maneira de passar parâmetros é utilizando a pilha. Desta forma não só os endereços de retorno, mas também os parâmetros ficam “protegidos” e o procedimento pode ser utilizado

recursivamente.

Para passar parâmetros utilizando a pilha, devemos empilhar os parâmetros antes da chamada do procedimento. Dentro do procedimento, para acessar os parâmetros utilizamos o registrador apontador de pilha. Suponha que desejemos chamar um procedimento *proc* com dois parâmetros *par1* e *par2*. O código abaixo mostra a preparação da chamada do procedimento, com os dois parâmetros sendo empilhados antes da instrução de chamada de procedimento:

```
; supondo que par1 esteja armazenado em r4 e par2 armazenado em r5
; exemplo de chamada de procedimento
        push    r4              ; empilha primeiro parâmetro
        push    r5              ; empilha segundo parâmetro
        call    proc            ; agora efetua a chamada
```

A configuração da pilha após a execução da chamada é mostrada na Figura 5.2. Para acessar os parâmetros, o procedimento deve executar algo como:

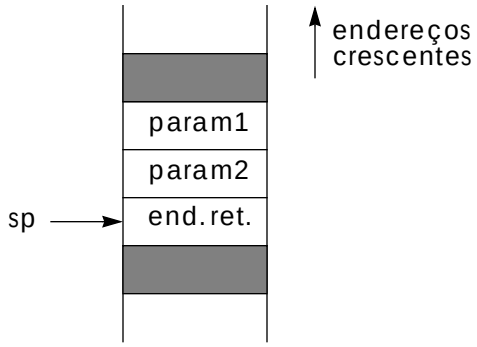


Fig. 5.2 Estado da pilha com os parâmetros após chamada do procedimento.

```
proc:
        ld      r0, (sp + 4)    ; carrega segundo parâmetro em r0
        ld      r1, (sp + 8)    ; e primeiro parâmetro em r1
        ...
```

Note que para acessar os parâmetros na pilha não deve ser usada instrução *pop*, pois o endereço de retorno também está na pilha e seria desempilhado. Portanto, os parâmetros empilhados permanecem 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

```
;supondo que par1 esteja armazenado em r4 e par2 armazenado em r5
; exemplo de chamada de procedimento
        push    r4              ; empilha primeiro parâmetro
        push    r5              ; empilha segundo parâmetro
        call    proc            ; agora efetua a chamada
        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 desalocadas com uma única instrução.

Diferentes linguagens de alto nível utilizam diferentes esquemas 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

```
void TesteC(a: int, b: int)
```

os parâmetros seriam empilhados assim:

```
        ld      r0, b           ; último parâmetro declarado        1
        push    r0                                                  2
        ld      r0, a           ; primeiro parâmetro declarado      3
        push    r0                                                  4
        call    TesteC                                              5
        add     sp, 8           ; retira os dois parâmetros da pilha. 6
```

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 usado por programas escritos em uma linguagem de alto nível, é necessário obedecer as convenções estabelecidas pela implementação da linguagem em questão.

Exemplo 5.4: Escreva um procedimento que devolva em *r0* o número de bits 1 de uma palavra de 32 bits passada como parâmetro pela pilha.

```
; ContaUm                                                           1
;       procedimento para contar o número de bits 1                 2
```

```
;       entrada: palavra de 32 bits passada pela pilha                    3
;       saída: número de bits 1 em r0                                     4
;       destroi: r1, r2 e r3                                              5
ContaUm:                                                                  6
        ld      r1, (sp+4)      ; carrega parâmetro                       7
        set     r3, 32          ; vamos contar 32 bits                    8
        xor     r0, r0          ; zera registrador resultado              9
proxbit:                                                                  10
        mov     r2, r1          ; copia em rascunho                       11
        and     r2,1            ; isola bit menos significativo           12
        add     r0, r2          ; e soma ao resultado                     13
        ror     r1, 1           ; prepara para proximo bit                14
        sub     r3              ; verifica se chegou ao final             15
        jnz     proxbit         ; se nao verificou todos os bits, desvia  16
        ret                     ; retorna com valor em r0                 17
```

Exemplo 5.5: Escreva um procedimento que verifique se duas palavras de 32 bits passadas como parâmetro na pilha têm o mesmo número de bits 1. Caso ambas tenham o mesmo número, o procedimento deve retornar o bit de estado *C* desligado, e o registrador *r0* deve conter o número de bits 1. Caso contrário, o bit de estado *C* deve retornar ligado.

```
; ComparaUm                                                                1
;       procedimento para comparar o número de bits 1                      2
;       entrada: duas palavras de 32 bits passadas pela pilha              3
;       saída:  se iguais, número de bits 1 em r0 e flag C desligada       4
;               se diferentes, flag C ligada                               5
;       destroi: r1 e flags                                                6
ComparaUm:                                                                 7
        ld      r0, (sp + 8)    ; carrega primeiro parâmetro               8
        push    r0              ; empilha como parâmetro para ContaUm      9
        call    ContaUm         ; conta o número de bits 1 do primeiro parâmetro 10
        add     sp, 4           ; retira parametro da pilha                11
        mov     r1, r0          ; guarda resultado em registrador temporário 12
        ld      r0, (sp + 4)    ; carrega segundo parâmetro                13
        push    r0              ; empilha como parâmetro para ContaUm      14
        call    ContaUm         ; conta o número de bits 1 do segundo parâmetro 15
        add     sp, 4           ; retira parametro da pilha                16
        cmp     r0, r1          ; compara resultados                       17
        jz      final           ; se iguais, pode retornar (e C está desligado) 18
        stc                     ; caso contrário, liga bit de estado C (se... 19
                                ; r0 > r1 ainda estava desligado)          20
final:                                                                     21
```

```
        ret                     ; retorna com valor em r0                   22
```

Esta solução apresenta um problema, se o procedimento ContaUm for o apresentado no Exemplo 5.4. Isto porque aquela versão de ContaUm destroi o registrador *r1*, que é usado em ComparaUm para guardar o número de bits 1 do primeiro parâmetro. Quando ContaUm é invocado pela segunda vez, o resultado da primeira chamada é perdido. Uma solução para este problema seria utilizar um outro registrador que não *r1*, *r2* ou *r3*, que sabidamente são alterados por ContaUm, para armazenar o resultado da primeira chamada. Mas como o número de registradores é finito, em um programa real em algum momento será necessário reutilizar registradores.

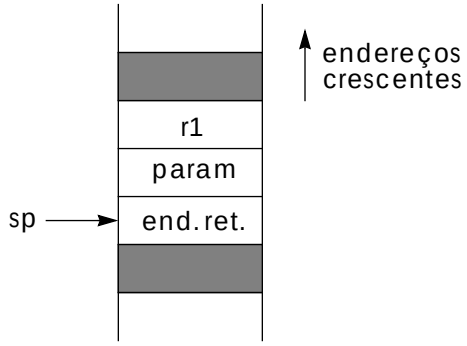


Fig. 5.3 Configuração da pilha com armazenagem temporária do registrador

Como fazer para preservar o valor dos registradores durante uma chamada de procedimento, já que eles podem ser usados dentro do procedimento? A resposta é mais uma vez o uso da pilha: antes da chamada do procedimento os valores dos registradores que se deseja preservar podem ser guardados na pilha, como mostra a Figura 5.3. Após o retorno do procedimento, registradores podem ser restaurados com os valores anteriores. Vamos re-escrever o procedimento *ComparaUm* utilizando esta idéia:

```
; ComparaUm --- segunda versão                                              1
;       procedimento para comparar o número de bits 1                       2
;       entrada: duas palavras de 32 bits passadas pela pilha               3
;       saída:  se iguais, número de bits 1 em r0 e flag C desligada        4
;               se diferentes, flag C ligada                                5
;       destroi: r1 e flags                                                 6
ComparaUm:                                                                  7
        ld      r0, (sp + 4)    ; carrega primeiro parâmetro                8
        push    r0              ; empilha como parâmetro para ContaUm       9
```

```
        call    ContaUm         ; conta bits 1 do primeiro parâmetro        10
        add     sp, 4           ; retira parametro da pilha                 11
        mov     r1, r0          ; guarda resultado em registrador temporário 12
        ld      r0, (sp + 8)    ; carrega segundo parâmetro                 13
        push    r1              ; salva para nao ser destruido por ContaUm  14
        push    r0              ; empilha como parâmetro para ContaUm       15
        call    ContaUm         ; conta bits 1 do segundo parâmetro         16
        add     sp, 4           ; retira parametro da pilha                 17
        pop     r1              ; recupera resultado para primeiro parâmetro 18
        cmp     r0, r1          ; compara resultados                        19
        jz      final           ; se iguais, pode retornar (e C está desligado) 20
        stc                     ; caso contrário, liga bit de estado C (se... 21
                                ; r0 > r1 ainda estava desligado)           22
final:                                                                      23
        ret                     ; retorna com valor em r0                   24
```

(Nesse caso particular é possível diminuir o código do procedimento, alterando as instruções das linhas 12~14 para

```
        push    r0              ; guarda resultado na pilha                 1
        ld      r0, (sp + 8)    ; carrega segundo parâmetro                 2
; linha suprimida                                                           3
```

mas o importante no exemplo é a ilustração da técnica de salvar temporáriamente o valor de registradores na pilha.)

Se um programa está sendo todo escrito em linguagem de montagem, e não envolve recursão, é possível (e necessário, embora custoso) fazer um mapeamento dos registradores de forma a minimizar o conflito de registradores utilizados pelos diversos procedimentos. Um bom compilador para uma linguagem de alto nível também faz esse trabalho de otimização do uso de registradores, principalmente para máquinas com muitos registradores (o processador MIPS, da ??, tem 256 registradores de propósito geral).

Exemplo 5.6: Escreva um procedimento que inverta uma cadeia de caracteres cujo endereço inicial é passado na pilha. O final da cadeia é marcado com o caractere '$'.

```
; Inverte                                                                   1
;       procedimento para inverter uma cadeia                               2
;       entrada: endereço da cadeia; final da cadeia é caractere ' '        3
;       saída:  nenhuma                                                     4
```

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

Esta solução tem um detalhe interessante. Nos processadores modernos, a execução de um desvio é bem 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 seqüencial de instruções continua. No processador Intel Pentium, por exemplo, a execução de um desvio condicional é cerca de 20 vezes mais demorada se o desvio é tomado (veja as razões na seção x). Na solução acima, o loop onde é feita a inversão poderia ser re-escrito com o sentido da condição da linha 22 invertido:

```
proxchar:                                                            1
        cmp     r1, r2          ; verifica se cadeia terminou        2
        jnc     continua        ; nao terminou, vai inverter         3
        ret                                                          4
continua:                                                            5
        ldb     r0, (r1)        ; elemento do início                 6
        ldb     r3, (r2)        ; elemento do final                  7
        stb     (r1), r3                                             8
        stb     (r2), r0        ; faz a troca                        9
```

```
add     r1,1            ; avança ponteiro do inicio             10
sub     r2,1            ; retrocede ponteiro do final           11
jmp     proxchar                                                12
```

O trecho acima executa exatamente o mesmo trabalho que o trecho original, mas leva muito mais tempo, pois a 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 vinte vezes mais lento. É necessário estar sempre atento ao sentido da condição dentro de loops, especialmente no caso de loops encaixados, quando o efeito é multiplicado.

5.4 Variáveis locais a procedimentos

Quando um procedimento utiliza muitas variáveis locais, e o número de registradores não é suficiente para alocá-las, a pilha é também utilizada para armazenar estas variáveis locais.

Considere a seguinte declaração em C:

```
Boolean
ComparaUm(int palavra1,int palavra2)
{
        int i;

        i = ContaUm(palavra1);
        return (i > ContaUm(palavra2);
}
```

O espaço para as variáveis locais é reservado na entrada do procedimento, e desalocado ao final do procedimento.

Até o momento temos usado o próprio *sp* para acessar os parâmetros na pilha. Ao reservarmos espaço para variáveis locais, o deslocamento necessário para acessar os parâmetros se altera, em relação a procedimentos sem variáveis locais. Além disso, como vimos, o valor de *sp* pode variar durante a execução do procedimento, pela necessidade de armazenar temporariamante algum valor na pilha. Isto também faz com que os deslocamentos necessários para acessar os parâmetros e variáveis locais se altere. Ou seja, apesar de o deslocamento normal para

acessar o primeiro parâmetro ser 4, se houver um dado temporário na pilha esse valor passa para 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 frame, que é mantido fixo, apontando para a posição inicial de *sp* no procedimento. No Faíska, o registrador *r30* é usado para esse fim, e a linguagem de montagem aceita o nome *fp* como sinônimo de *r30*.

Como o valor antigo de *fp* não deve ser destruído, é necessário que o empilhemos logo na entrada do procedimento. Durante a execução de um procedimento com parâmetros e variáveis locais a pilha tem portanto a configuração mostrada na Figura 5.4.

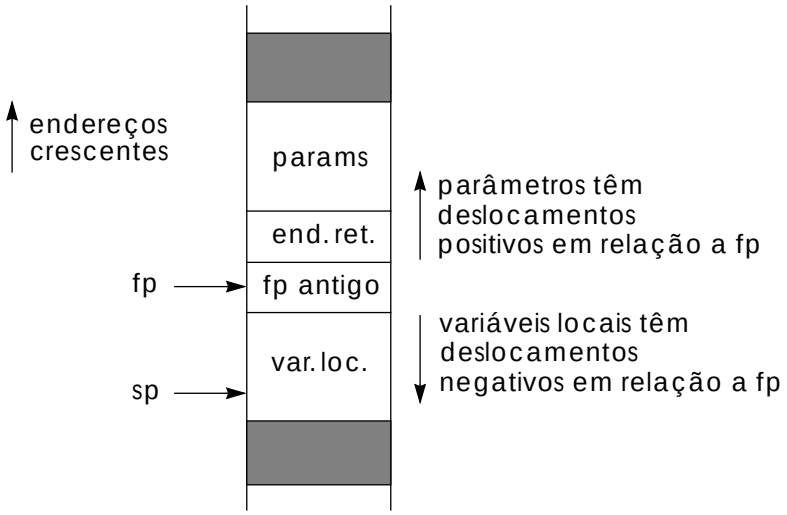


Fig. 5.4 Configuração da pilha com parâmetros e variáveis locais ao procedimento.

Como exemplo, vamos re-escrever o procedimento ComparaUm usando uma variável local:

```
; ComparaUm --- versão com variável local                              1
;       procedimento para comparar o número de bits 1                  2
;       entrada: duas palavras de 32 bits passadas pela pilha          3
;       saída:  se iguais, número de bits 1 em r0 e flag C desligada   4
;               se diferentes, flag C ligada                           5
;       destroi: r1 e flags                                            6
ComparaUm:                                                             7
        push    fp              ; guarda fp antigo                     8
        mov     fp, sp          ; apontador de frame                   9
        sub     sp, 4           ; reserva espaço para variável local i 10
        ld      r0, (fp + 8)    ; carrega primeiro parâmetro           11
        push    r0              ; empilha como parâmetro para ContaUm  12
        call    ContaUm         ; conta bits 1 do primeiro parâmetro   13
        add     sp, 4           ; retira parâmetro da pilha            14
```

```
        st      (fp – 2), r0    ; guarda resultado na variável local        15
        ld      r0, (fp + 4)    ; carrega segundo parâmetro                 16
        push    r0              ; empilha como parâmetro para ContaUm       17
        call    ContaUm         ; conta bits 1 do segundo parâmetro         18
        add     sp, 4           ; retira parametro da pilha                 19
        ld      r1, (fp – 2)    ; recupera resultado para primeiro parâmetro 20
        cmp     r0, r1          ; compara resultados                        21
        jz      final           ; se iguais, pode retornar (e C está desligado) 22
        stc                     ; caso contrário, liga bit de estado C (se... 23
                                ; r0 > r1 ainda estava desligado)           24
final:                                                                      25
        add     sp, 4           ; desaloca variável local                   26
        pop     fp                                                          27
        ret                     ; retorna com valor em r0                   28
```

Note como os deslocamentos em relação ao apontador de frame *fp* para acessar os parâmetros têm valor negativo, e variáveis locais têm deslocamento positivo em relação a *fp*, e estes deslocamentos são sempre fixos durante a execução do procedimento.

Como *fp* é empilhado a cada entrada de procedimento, no caso de chamadas encaixadas de procedimentos forma-se na pilha uma cadeia de apontadores, como ilustra a Figura 5.5.

5.4.1 Passagem de parâmetros por referência e por valor

Parâmetros em linguagens de alto nível como Pascal ou C podem ser passados por referência ou por valor, como na declaração Pascal abaixo:

```
procedure ValRef(c: char; i: integer; var x: integer);
```

Em linguagem de montagem, a distinção entre referência e valor é similar, mas 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, de forma que o procedimento pode alterar o valor contido naquele endereço, a passagem é por referência. Mas como não há verificação de tipos, é necessário tomar bastante cuidado para não cometer erros na passagem de parâmetros.

Uma chamada ao procedimento ValRef acima, como por exemplo

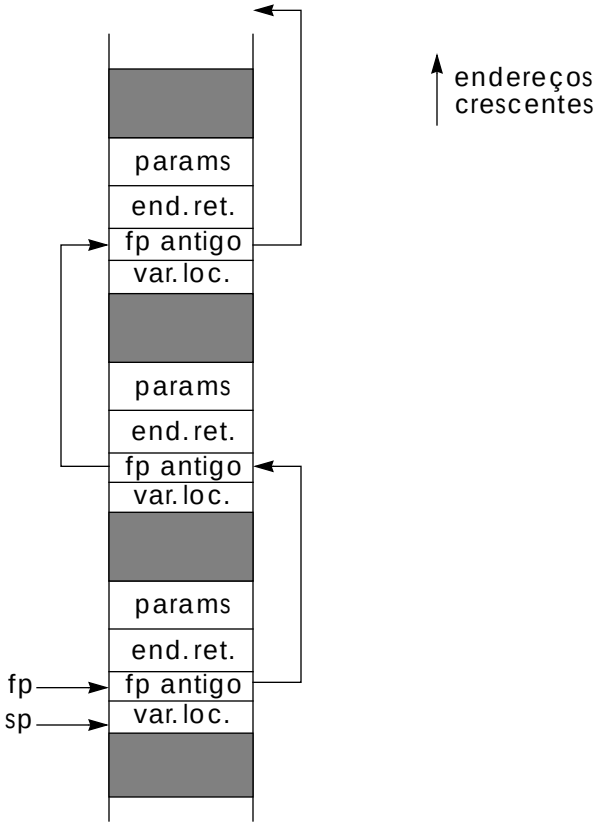


Fig. 5.5 Cadeia de apontadores *fp* durante chamadas encaixadas de procedimentos.

```
ValRef('a', val, ref);    (* assumindo val e ref sejam declarados como inteiros *)
```

seria traduzida em linguagem de montagem para

```
ldb     r0,'a'          ; carrega primeiro parâmetro
push    r0
ld      r0, val         ; carrega valor do segundo parâmetro
push    r0
set     r0, ref         ; carrega endereço do terceiro parâmetro
push    r0
call    ValRef
add     sp, 12          ; desempilha parâmetros
```

Exemplo 5.7: Implemente em linguagem de montagem o procedimento Pascal a seguir:

```
procedure troca(var x, y: integer);
var tmp: integer;
begin
        tmp := x;
        x := y;
        y := tmp;
end;
```

```
; Troca                                                              1
;       procedimento para trocar os valores de duas variáveis inteiras  2
;       entrada: dois endereços de variáveis inteiras, passados na pilha 3
;       saída:  nenhuma                                              4
;       destrói: r0, r1, r2 e r3                                     5
                                                                     6
Troca:                                                               7
        ld      r1, (sp+4)      ; endereço da primeira variável      8
        ld      r2, (sp+8)      ; endereço da segunda variável       9
        ld      r0, (r1)        ; r0 tem valor da primeira variável  10
        ld      r3, (r2)        ; r3 tem valor da segunda variável   11
        st      (r1), r3        ; e guardamos nas variáveis trocadas 12
st      (r2), r0                                                     13
ret                                                                  14
```

5.5 Recursão

Uma grande vantagem de utilizar a pilha para passagem de passagem de parâmetros e armazenamento de variáveis locais é que os procedimentos podem ser recursivos.

Suponha que a estrutura que represente um vértice de uma árvore binária seja implementada em linguagem de montagem com as seguintes declarações:

```
VAL     equ     0       ; conteúdo do nó, inteiro sem sinal
DIR     equ     4       ; apontador para filho direito
ESQ     equ     8       ; apontador para filho esquerdo
NIL     equ     0       ; marcador de apontador vazio
```

Vamos escrever um procedimento que percorra uma árvore binária e devolva como resultado a soma dos valores de seus vértices.

```
; SomaNós --- soma nós de árvore                                     1
;       procedimento que percorre uma árvore binária e acumula       2
;               o valor dos nós                                      3
```

```
;       entrada: endereço da raiz, passado pela pilha                    4
;       saída:  soma dos valores dos nós em r0                           5
;       destrói: r1, r2 e flags                                          6
SomaNós:                                                                 7
        set     r0,0            ; valor inicial da soma                  8
percorre:                                                                9
        ld      r1, (sp+4)      ; pega raiz da árvore da pilha           10
        cmp     r1, NIL         ; se árvore vazia...                     11
        jz      final           ; nada a fazer                           12
        ld      r2, (r1+VAL)    ; acumula ...                            13
        add     r0, r2          ; mais este nó                           14
        ld      r2, (r1+DIR)    ; filho da direita                       15
        push    r2              ; usa como nova raiz,                    16
        call    percorre        ; percorre recursivamente                17
        add     sp, 4           ; retira parâmetro                       18
        ld      r1, (sp+4)      ; pega novamente raiz da árvore da pilha 19
        ld      r2, (r1+ESQ)    ; filho da esquerda                      20
        push    r2              ; usa como nova raiz                     21
        call    percorre        ; percorre recursivamente                22
        add     sp, 4           ; retira parâmetro                       23
final:                                                                   24
        ret                     ; e retorna                              25
```

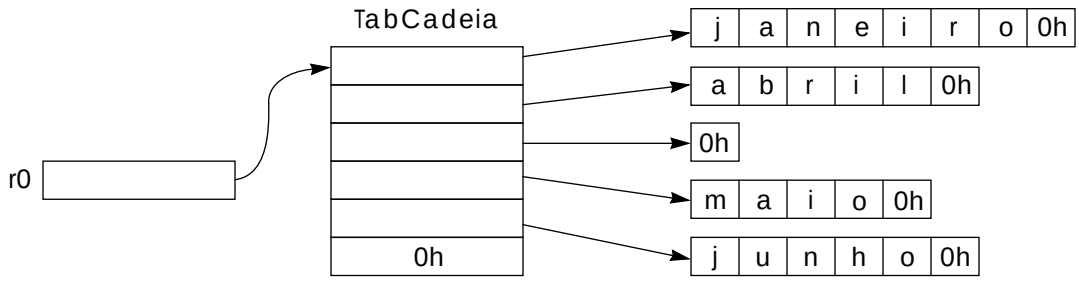
Note que na linha 19 tomamos o cuidado de recarregar, da pilha, o apontador para a raiz da árvore, já que a chamada recursiva da linha 17 destrói o valor previamente carregado em *r1*.

5.6 Exercícios

1. Um esquema em que cada procedimento cuide de não estragar o valor dos registradores que usar (empilhando os registradores no início do procedimento, desempilhando antes do retorno) não é em geral uma boa idéia. Explique por que.

2. Escreva um procedimento *Compr* que receba como parâmetro (passado na pilha) um apontador para uma cadeia de caracteres e devolva em *r0* o comprimento da cadeia (ou seja, o número de caracteres). A cadeia, possivelmente vazia, é terminada pelo valor 00h (este caractere de terminação não entra no cômputo do comprimento da cadeia).

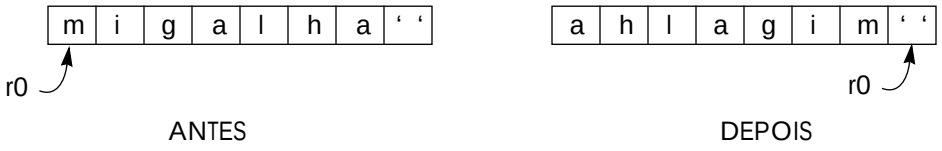
3. Considere que o endereço *TabCadeia* contenha um vetor onde cada elemento é um apontador para uma cadeia de caracteres terminada pelo valor 00h, como no Exercício 2. O final do vetor *TabCadeia* é indicado pelo valor 00000000h. Escreva um procedimento Busca que receba em *r1* o endereço inicial da tabela e, utilizando o procedimento *Compr* do Exercício 2, devolva em *r0* o comprimento da cadeia com o maior número de caracteres e em *r1* o endereço de início dessa cadeia. Se houver mais de uma cadeia com comprimento máximo deve ser retornado o endereço da primeira cadeia.



4. Escreva um procedimento que calcule a soma de dois números inteiros positivos, onde cada inteiro é representado por um vetor de elementos de 4 bits cada (cada elemento representa um dígito do número). O primeiro elemento do vetor indica quantos dígitos tem o número (cada número pode portanto ter até 16 dígitos); o segundo elemento é o dígito menos significativo, o último elemento o dígito mais significativo. São passados como argumento, pela pilha, os endereços iniciais dos dois vetores que se deseja somar, e o endereço inicial do vetor resultado.

5. Escreva um procedimento *Inverte* para inverter a ordem dos caracteres de uma cadeia de caracteres apontada por *r1*. O final da cadeia

é marcado pelo caracter ‘ ‘ (espaço em branco). Ao terminar a execução, *r1* deve apontar para o final da cadeia (caracter ‘ ‘).



6. Utilizando o procedimento *Inverte*, do Exemplo 5.6, escreva um trecho de programa que inverta as palavras de uma frase representada por uma cadeia de caracteres terminada pelo caractere ‘$’ (palavras são sub-cadeias separadas por ‘ ‘). Por exemplo, a frase ‘isto é um exemplo$’) ao ser invertida produziria ‘otsi é mu olpmexe$’). Considere que a frase seja apontada por *r1*, contém pelo menos uma palavra, e o caractere de terminação ‘$’ ocorra sempre após um caractere ‘ ‘.

7. Escreva um procedimento 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.