Linguagens de montagem
Capítulo 7 – Procedimentos e funções

Ricardo Anido
Instituto de Computação
Unicamp

Procedimentos e funções

Uma chamada de procedimento envolve dois trechos de programas, o trecho de programa que executa a chamada e o procedimento que é chamado.

- o trecho de programa que executa a chamada
- o procedimento que é chamado.

```
int x,y,z; // declaração de algumas variáveis

void troca(int *a, int *b) {
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(void) {
    ...
    troca(&x,&y);
    z=x+1;
    troca(&z,&x);
    x=y-1;
    ...
}
```

Procedimentos e funções

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,
- ao final da execução do procedimento o fluxo é desviado de volta para o comando seguinte à chamada de procedimento.
- o endereço para o qual o fluxo de execução deve ser desviado ao final do procedimento é chamado de *endereço de retorno* do procedimento.

Chamada de procedimento: primeira tentativa

Uma instrução usada antigamente:

JSR

Desvia para sub-rotina

Syntax	Operação	Flags	Codificação
jsr *rótulo*	$mem[imd32] \leftarrow ip + 8$ $ip \leftarrow imd32 + 4$	–	31 … 0 `0xff` | – | – | – 31 … 0 *imd*32

Chamada de procedimento: primeira tentativa

- Ao ser executada, a instrução armazena o valor do endereço de retorno na primeira palavra do procedimento chamado.
- A primeira palavra do procedimento deve portanto ser reservada para esse fim, (não deve conter uma instrução)
- Ao final da execução do procedimento, para retornar da chamada, podemos recuperar o endereço de retorno, armazenado no início da execução do procedimento pela instrução JSR, e desviar para esse endereço.

Chamada de procedimento: primeira tentativa

```
00001000 [ff 00 00 00] |     jsr proc     @ uma chamada ao procedimento
         [00 00 40 00] |                   @ de nome proc
00001008 [50 00 01 02] |     xor r1,r2     @ esta é a instrução que deve
                       |                   @ ser executada após retorno
                       |                   @ do procedimento
   ...         ...     |     ...
                       |      .org 0x4000
                       | @ aqui é o início do procedimento
                       | proc:
00004000 [.. .. .. ..] |      .skip 4      @ palavra deixada vazia
                       |                   @ para armazenar end. retorno
00004004 [01 ff 05 00] |      set   r5,-1  @ esta é a primeira instrução
                       |                   @ do procedimento
   ...         ...     |     ...
                       | @ aqui é o final do procedimento
                       | @ efetua retorno para endereço armazenado
00004084 [02 00 0a 00] |      ld   r10, proc   @ recupera end. retorno
00004088 [00 00 40 00] |                   @ guardado pela instrução jsr
00004084 [31 00 0a 00] |      jmp   r10    @ e desvia, retornando
```

Chamada de procedimento: primeira tentativa

Essa solução foi adotada em alguns processadores antigos, como o IBM-1130, da década de 60. Problemas:

- não funcionaria em sistemas operacionais de hoje, pois estes não permitem que sejam realizados acessos para escrita na região de programa.
- mais grave: não permite recursão!

Como permitir recursão?

Pilhas

Podemos implementar uma pilha usando um apontador, como `r0`. Inicialmente, ele deve ser inicializado para apontar para uma região de memória disponível.

- Uma palavra é empilhada na pilha apontada por `r0` decrementando-se `r0` de quatro e escrevendo-se a palavra no novo endereço apontado por `r0`.
- Neste esquema, a pilha "cresce" de endereços altos para endereços baixos.

Pilhas

Operação *Empilhar*:

```
@ implementando uma pilha com o registrador r0 como apontador de pilha
@ exemplo que empilha valor do registrador r10
    sub   r0,4            @ faz apontador de pilha apontar para
                          @ novo elemento
    st    [r0],r10        @ e coloca valor de r10 no topo da pilha
```

Pilhas

Operação *Desempilhar*:

```
@ implementando uma pilha com o registrador r0 como apontador de pilha
@ exemplo que retira o valor do topo da pilha e armazena em r9
    ld    r9,[r0]         @ carrega r9 com valor no topo da pilha
    add   r0,4            @ r0 aponta para novo topo da pilha
```

Pilhas

- manipulações de pilhas são tão frequentes que os processadores incluem instruções específicas e um registrador especial, normalmente chamado `sp` (do inglês *stack pointer*), que funciona como apontador de pilha.
- no LEG o apontador de pilha é na verdade um dos registradores de uso geral, `r15`.
- a linguagem de montagem aceita `sp` como outro nome do registrador `r15`.
- duas instruções no LEG: PUSH (empilha registrador) e POP (desempilha registrador).

Pilhas

PUSH

Empilha registrador			
Sintaxe	Operação	Flags	Codificação
push *rf*	$sp \leftarrow sp - 4$ $mem[sp] \leftarrow rf$	–	31 … 0 0x50 | – | – | *rf*

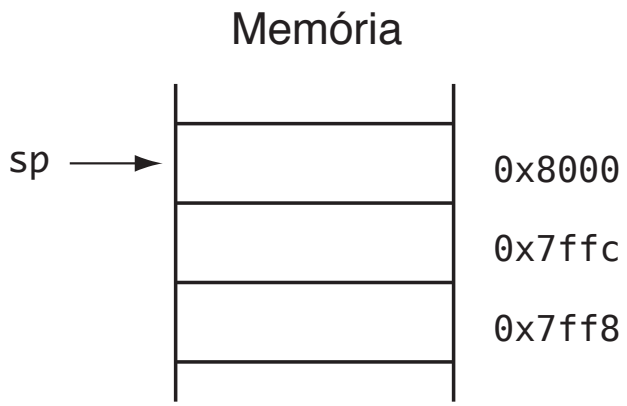
POP

Desempilha registrador			
Sintaxe	Operação	Flags	Codificação
pop *rd*	$rd \leftarrow mem[sp]$ $sp \leftarrow sp + 4$	–	31 … 0 0x51 | – | *rd* | –

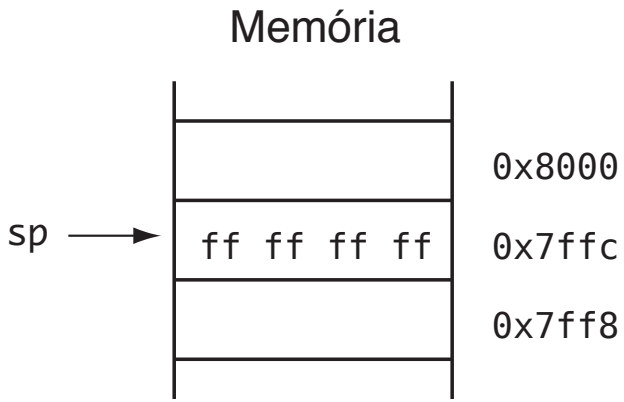
Pilhas

```
                        | @ exemplos de instruções push e pop
00000400 [02 00 0f 00] |    set  sp, 0x8000  @ um valor inicial para sp
         [00 00 80 00] |                      @ e vamos colocar um valor
00000408 [01 ff 02 00] |    set  r2, -1       @ em r2 para ilustrar
0000040c [50 00 00 02] |    push r2
00000410 [51 00 01 00] |    pop  r1
```

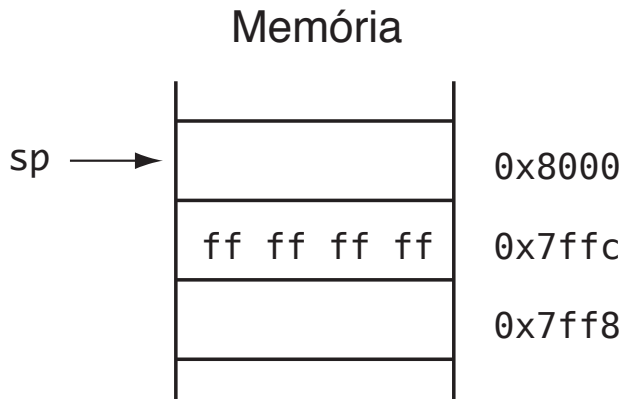
Pilhas



(a) Após a linha 2



(b) Após a linha 5



(c) Após a linha 6

Pilhas

Antes de utilizar instruções que manipulam a pilha, é necessário reservar a região de memória que será utilizada pela pilha, como por exemplo:

```
@ aloca uma região da memória para a pilha
@ assume que a pilha utilizará no máximo 1 KiB
      TAM_PILHA .equ 1024
@ aloca espaço (em área de memória reservada para dados)
fim_pilha:
      .skip  TAM_PILHA
ini_pilha:
      ...
@ inicializa apontador de pilha, antes de executar
@ qualquer instrução que manipule a pilha
      set  sp,ini_pilha
```

Chamada e retorno de procedimento

- A implementação de procedimentos nos processadores atuais faz uso do registrador apontador de pilha.
- No LEG são definidas duas instruções específicas:
  - chamada de procedimento (CALL)
  - retorno do procedimento (RET)

Chamada e retorno de procedimento

CALL

Chamada de procedimento			
Syntax	Operação	Flags	Codificação
call *rótulo*	$sp \leftarrow sp - 4$ $mem[sp] \leftarrow ip$ $ip \leftarrow imd32$	–	31 … 0 0x54 | – | – | – 31 … 0 *imd*32
call *rd*	$sp \leftarrow sp - 4$ $mem[sp] \leftarrow ip$ $ip \leftarrow rd$	–	31 … 0 0x55 | – | *rd* | –

Chamada e retorno de procedimento

- A instrução chamada de procedimento (CALL) empilha o endereço de retorno e executa o desvio para o início do procedimento alvo.
- O endereço alvo (início do procedimento) pode ser especificado através de um rótulo (codificado no campo *imd*32) ou de um registrador *rd*.
- O endereço de retorno empilhado é o endereço da instrução imediatamente seguinte à instrução CALL.

Chamada e retorno de procedimento

RET

Retorno de procedimento			
Sintaxe	Operação	Flags	Codificação
ret	$ip \leftarrow mem[sp]$ $sp \leftarrow sp + 4$	–	31 … 0 0x56 – – –

Chamada e retorno de procedimento

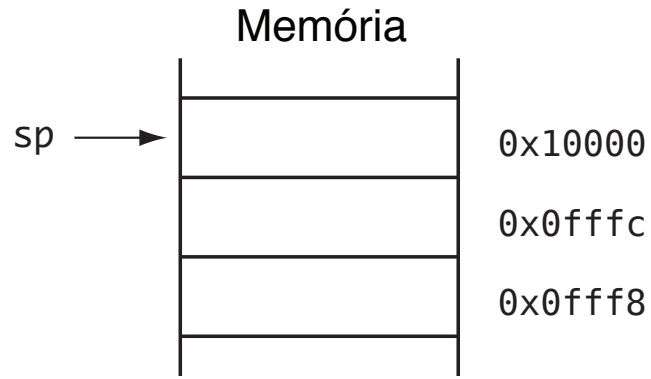
A instrução retorno de procedimento (RET):

- 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.

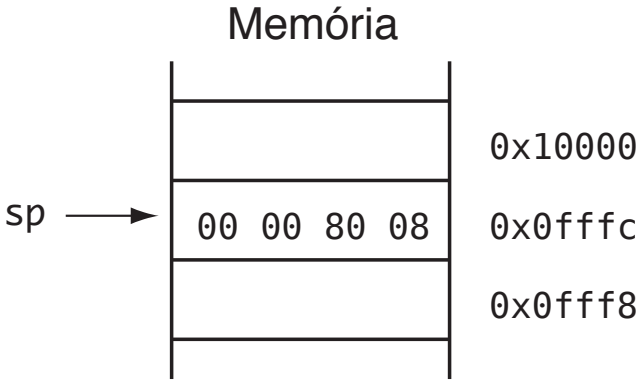
Chamada e retorno de procedimento

```
                          | @ exemplo de chamada de procedimento
00008000 [54 00 00 00]    |    call proc_exe @ uma chamada ao procedimento
00008004 [00 00 90 00]    |                  @ de rótulo proc_exe
00008008 [31 00 01 02]    |    xor r1,r2     @ esta é a instrução que deve
                          |                  @ ser executada após retorno
                          |                  @ do procedimento
   ...         ...        |    ...
                          |    .org 0x9000
                          | @ aqui é o início do procedimento
                          | proc_exe:
00009000 [01 ff 05 00]    |    set   r5,-1   @ esta é a primeira instrução
                          |                  @ do procedimento
   ...         ...        |    ...
                          | @ aqui é o final do procedimento
00009100 [56 00 00 00]    |    ret           @ retorna do procedimento
```

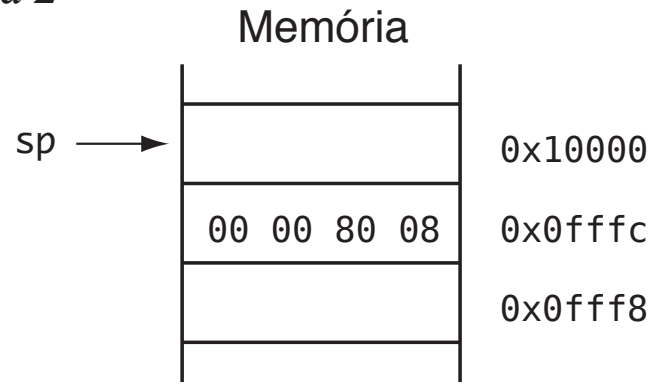
Chamada e retorno de procedimento



(a) Imediatamente antes da instrução CALL na linha 2



(b) Imediatamente após a instrução CALL na linha 2



(c) Imediatamente após a instrução RET na linha 15

Problema

Escreva um procedimento `zera_regs` para zerar os registradores `r0`, `r1`, `r2` e `r3`.

Solução

```
@ ********
@ zera_regs
@ ********
@ Zera os registradores r0, r1, r2 e r3
@   entrada: nenhuma
@   saída: r0, r1, r2 e r3 zerados
@   destrói: nada

zera_regs:
       set r0,0
       set r1,0
       set r2,0
       set r3,0
       ret
...
@ exemplo de chamada do procedimento zera_regs
     call   zera_regs          @ após a chamada, r0=r1=r2=r3=0
```

Procedimentos com parâmetros

- Uma primeira abordagem é usar registradores para armazenar os argumentos na chamada do procedimento.
- Bastante eficiente, e pode ser usado se o procedimento não é recursivo e há registradores disponíveis em número suficiente para acomodar os parâmetros.

Problema

Escreva um procedimento `preenche_mem`, com funcionalidade similar ao procedimento *memset*, da biblioteca padrão de C. Ou seja, o procedimento `preenche_mem` deve preencher uma região de memória com um valor de byte passado 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.

Solução

```
@ ************
@ preenche_mem
@ ************

preenche_mem:
     sub   r2,1                @ continua a preencher?
     jc    preenche_mem_fim    @ desvia se terminou de preencher
     stb   [r1],r0             @ preenche um byte com valor dado
     add   r1,1                @ avança apontador
     jmp   preenche_mem
preenche_mem_fim:
     ret                       @ e retorna, região foi preenchida
```

Solução

```
...
set   r0,0xff          @ valor de byte para preencher memória
set   r1,0x1000        @ endereço inicial
set   r2,100           @ número de bytes a preencher
call  preenche_mem
...
```

Passagem de parâmetros

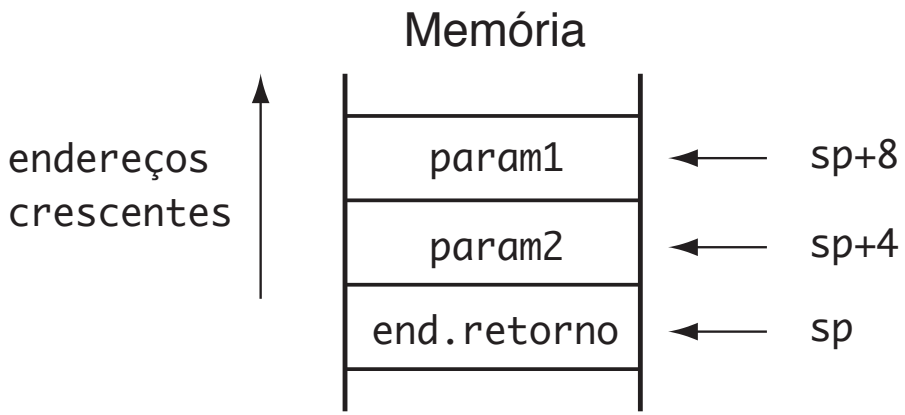
- Apesar de eficiente, a passagem de parâmetros por registradores não pode ser utilizada em todos os casos:
  - procedimentos recursivos
  - número de argumentos é maior do que o número de registradores disponíveis
- Nesses casos, é necessário utilizar a pilha para passagem de parâmetros.

Passagem de parâmetros pela pilha

- 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.

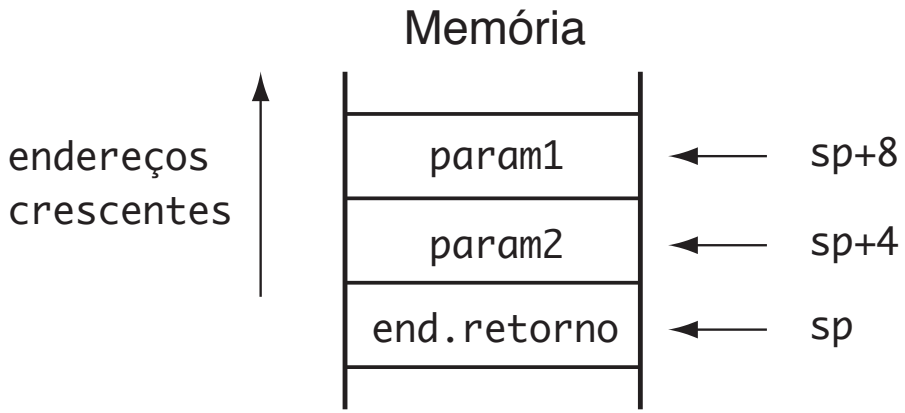
Passagem de parâmetros pela pilha

```
@ supondo que param1 esteja armazenado em r4 e param2 armazenado em r5
@ exemplo de chamada de procedimento
      push r4           @ empilha arg1 correspondente ao param1
      push r5           @ empilha arg2 correspondente ao param2
      call proc_param   @ agora efetua a chamada
      ...
```



Passagem de parâmetros pela pilha

```
@ Acesso a parâmetros durante a execução do procedimento.
        ...
proc_param:
        ld  r0,[sp+8]     @ carrega valor de param2 em r0
        ld  r1,[sp+4]     @ e carrega valor de param1 em r1
        ...
```

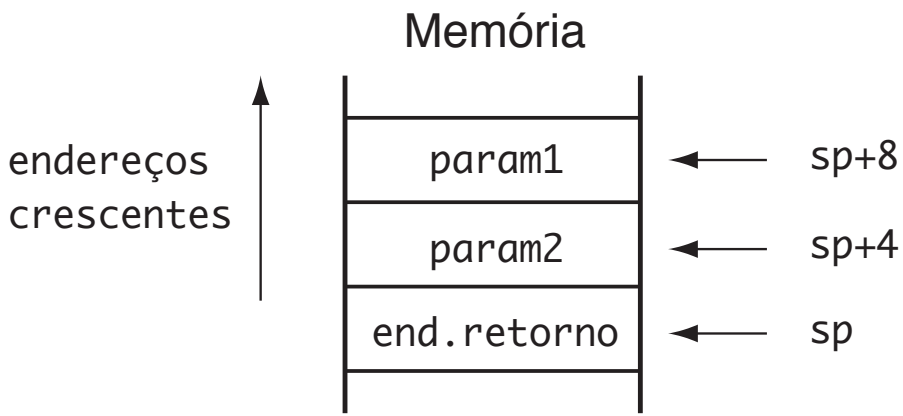


Passagem de parâmetros pela pilha

- A instrução POP NÃO deve ser usada para acessar os parâmetros na pilha, pois o endereço de retorno também está na pilha (foi empilhado após os parâmetros, pela instrução CALL).
- Os parâmetros empilhados antes da chamada devem permanecer na pilha até o retorno do procedimento, quando devem então ser desempilhados.

Passagem de parâmetros pela pilha

```
@ supondo que o argumento correspondente a param1 esteja armazenado em r4
@ e o argumento correspondente a param2 esteja armazenado em r5
@ exemplo de chamada de procedimento
      push r4          @ empilha valor para param1
      push r5          @ empilha valor para param2
      call proc_param  @ agora efetua a chamada
      add  sp,8        @ retira os parâmetros da pilha
```



Passagem de parâmetros pela pilha

- Diferentes linguagens de alto nível podem utilizar diferentes regras para empilhar os parâmetros.
- Em C, os parâmetros são empilhados na ordem inversa em que foram declarados.
- Para o procedimento `param_c` declarado como

```
void param_c(int x, int y)
```

a chamada

```
param_c(a, b)
```

seria implementada como mostrado a seguir.

Passagem de parâmetros pela pilha

```
...
ld    r0,b          @ último parâmetro declarado
push  r0            @ é o primeiro a ser empilhado
ld    r0,a          @ primeiro parâmetro declarado
push  r0            @ é o último a ser empilhado
call  param_c       @ faz a chamada
add   sp,8          @ retira os dois parâmetros da pilha.
...
```

Retorno de valores de funções

- No caso de funções (procedimentos que retornam valores), o valor ou valores podem ser retornados em registradores ou na pilha.
- C permite que apenas tipos escalares sejam retornados por funções, e nesse caso um registrador específico é utilizado.
- Algumas linguagens de programação permitem que funções retornem valores de tipos não escalares, e nesses casos a convenção de passagem de parâmetros utiliza outros registradores ou a pilha.
- Em linguagem de montagem podemos fazer as nossas próprias convenções para o retorno de valores por funções, e registradores são a primeira opção.

Problema

Escreva 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`.

Solução

```
@ **********
@ multiplica
@ **********

multiplica:
     cmp   r1,r2          @ para minimizar os passos da multiplicação
     jnc   mult1          @ usa menor valor para controlar repetição
     mov   r0,r1          @ troca valores de r1 e r2 usando
     mov   r1,r2          @ r0 como temporário
     mov   r2,r0
mult1:
     set   r0,0           @ inicializa valor do produto
mult2:
     sub   r2,1           @ vamos realizar r2 adições
     jc    mult3          @ desvia se terminamos
     add   r0,r1          @ adiciona mais uma parcela
     jmp   mult2
mult3:
     ret                  @ retorna quando todas as adições terminaram
```

Problema

Escreva uma função `conta_uns` que devolve em `r0` o número de bits 1 de uma palavra de 32 bits passada como parâmetro pela pilha.

Solução

```
@ *********
@ conta_uns
@ *********
@ Conta o número de bits 1 de uma palavra passada pela pilha
@   entrada: palavra de 32 bits passada pela pilha
@   saída: número de bits 1 em r0
@   destrói: r1 e flags

conta_uns:
     ld    r1,[sp+4]       @ carrega parâmetro
     xor   r0,r0           @ zera registrador resultado
proxbit:
     shr   r1,1            @ testa mais um bit
     jnc   testafim        @ bit deslocado é 1?
     add   r0,1            @ conta este bit (note que flag Z nunca é um)
testafim:
     jnz   proxbit         @ se não verificou todos os bits, continua
     ret                   @ retorna com valor em r0
```

Problema

Escreva uma função `compara_uns` 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 contido em cada palavra. Caso contrário, `r0` deve retornar com o valor -1.

Tentativa de solução

```
@ ***********
@ compara_uns
@ ***********
@ Compara o número de bits 1 de duas palavras

compara_uns:
     ld    r0,[sp+4]       @ carrega primeiro parâmetro
     push  r0              @ empilha como parâmetro para conta_um
     call  conta_uns       @ conta o número de bits 1 do primeiro
     add   sp,4            @ retira parâmetro da pilha
     mov   r1,r0           @ guarda resultado intermediário em r1
     ld    r0,[sp+8]       @ carrega segundo parâmetro
     push  r0              @ empilha como parâmetro para conta_um
     call  conta_uns       @ conta o número de bits 1 do segundo
     add   sp,4            @ retira parâmetro da pilha
     cmp   r0,r1           @ compara resultados
     jz    final           @ se iguais retorna, r0 com num. bits 1
     set   r0,-1           @ caso contrário, indica com r0=-1
final:
     ret                   @ retorna com valor em r0
```

Tentativa de solução

- a implementação desse exemplo apresenta um problema que ocorre com frequência: como `conta_uns` destrói o registrador `r1`, que é usado em `compara_uns` para guardar o número de bits 1 do primeiro parâmetro, quando `conta_uns` é invocado pela segunda vez, o resultado da primeira chamada, que foi armazenado em `r1`, é perdido.
- Uma solução óbvia para este problema: utilizar outro registrador para armazenar o resultado da primeira chamada.
- No entanto, nem sempre há outro registrador disponível.

Tentativa de solução

- 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
- após o retorno do procedimento os registradores devem ser restaurados com os valores anteriores, armazenados na pilha

Solução

```
@ ****************************
@ compara_uns -- segunda versão
@ ****************************
compara_uns:
     ld    r0,[sp+4]       @ carrega primeiro parâmetro
     push  r0              @ empilha como parâmetro para conta_uns
     call  conta_uns       @ conta bits 1 do primeiro parâmetro
     add   sp,4            @ retira parâmetro da pilha
     push  r0              @ salva resultado para não ser destruído
     ld    r0,[sp+12]      @ carrega segundo parâmetro
                           @ note que o deslocamento em relação ao
                           @ topo da pilha mudou (12 ao invés de 8)
     push  r0              @ empilha como parâmetro para conta_uns
     call  conta_uns       @ conta bits 1 do segundo parâmetro
     add   sp,4            @ retira parâmetro da pilha
     pop   r1              @ recupera primeiro resultado
     cmp   r0,r1           @ compara resultados
     jz    final           @ se iguais retorna, r0 com num. bits 1
     set   r0,-1           @ caso contrário, indica com r0=-1
final:
     ret                   @ retorna com valor em r0
```

Passagem de parâmetros por referência e por valor

- Em linguagens de alto nível o compilador cuida de tratar corretamente esses dois tipos de parâmetros.
- Em linguagem de montagem, o controle é feito exclusivamente pelo programador.

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

Passagem de parâmetros por referência e por valor

```
ld   r0,val              @ empilha valor do terceiro parâmetro
push r0                  @ valor contido no endereço val é empilhado
set  r0,0x61             @ empilha segundo parâmetro
push r0                  @ valor da constante 'a' é empilhado
set  r0,ref              @ empilha endereço do primeiro parâmetro
push r0                  @ o endereço do rótulo ref é empilhado
call exemplo_valor_ref
add  sp,12               @ desempilha parâmetros
```

Problema

Traduza para linguagem de montagem do LEG o procedimento `troca`, declarado no Exemplo **??** no início deste capítulo, para trocar os valores de duas variáveis inteiras passadas como referência pela pilha.

```
int x,y,z; // declaração de algumas variáveis

void troca(int *a, int *b) {
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(void) {
    ...
    troca(&x,&y);
    z=x+1;
    troca(&z,&x);
    x=y-1;
    ...
}
```

Solução

```
@ *****
@ troca
@ *****
@ Troca os valores de duas variáveis passadas por referência na pilha
@   entrada: dois endereços de variáveis inteiras, pela pilha
@   saída: nenhuma
@   destrói: r0,r1,r2,r3

troca:
     ld    r0,[sp+4]       @ carrega endereço da variável a
     ld    r1,[sp+8]       @ carrega endereço da variável b
     ld    r2,[r0]         @ valor da variável a
     ld    r3,[r1]         @ valor da variável b
     st    [r0],r3         @ armazena valor de b no endereço de a
     st    [r1],r2         @ armazena valor de a no endereço de b
     ret
```

Solução

Exemplo de chamada:

```
    ...
    set   r0,y          @ carrega endereço da variável y
    push  r0            @ empilha argumento
    set   r0,x          @ carrega endereço da variável x
    push  r0            @ empilha argumento
    call  troca         @ faz a chamada
    add   sp,8          @ retira os dois argumentos da pilha
    ...
@ aqui estão definidas as variáveis inteiras
x:   .skip   4
y:   .skip   4
```

Variáveis locais a procedimentos

- Uma variável declarada no corpo de um procedimento é chamada de variável *local* ao procedimento.
- Uma variável local é visível e acessível apenas dentro do corpo do procedimento em que é declarada.
- Registradores são normalmente usados para representar variáveis locais a um procedimento.
- Se o número de registradores não é suficiente para alocar as variáveis de um procedimento, ou se o procedimento é recursivo, as variáveis locais devem ser alocadas na pilha.

Variáveis locais a procedimentos

- A alocação de espaço na pilha para variáveis locais deve ser feita no corpo do procedimento (normalmente na entrada).
- O espaço deve ser desalocado antes do retorno do procedimento. Esse tipo de alocação de espaço para variáveis é denominado de alocação *dinâmica* de variáveis porque, em contraste com a alocação estática, o espaço é alocado apenas quando necessário.

Variáveis locais a procedimentos

- Para alocar espaço na pilha, basta decrementar o número de bytes desejado do registrador `sp`.
- Ao reservarmos espaço para variáveis locais na pilha dentro de um procedimento, o deslocamento necessário, em relação a `sp`, para acessar os parâmetros do procedimento se altera.
- Exemplo: apesar de o endereço normal para acessar o primeiro parâmetro ser `sp+4`, se alocarmos duas palavras para variáveis locais na pilha esse endereço passa a ser `sp+12`.

Registrador apontador de quadro

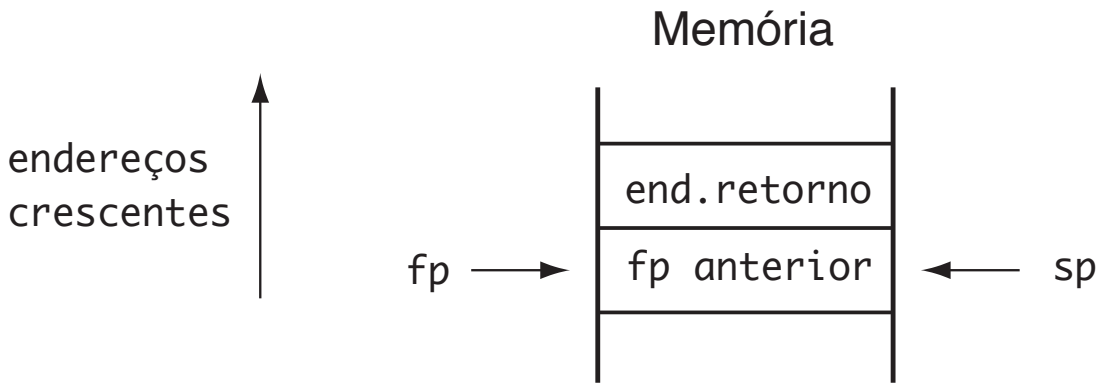
- Para evitar que os deslocamentos sejam alterados durante a execução do procedimento, gerando confusão, é comum utilizarmos mais um registrador, geralmente chamado apontador de quadro (em inglês, *frame pointer*), que é mantido fixo durante a execução do procedimento, apontando para o endereço que `sp` aponta no início do procedimento.
- Os deslocamentos dos parâmetros em relação ao registrador apontador de quadro mantêm-se fixos durante toda a execução do procedimento.
- No LEG, o registrador `r14` pode ser usado como apontador de quadro, 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.
- Como o apontador de quadro pode estar sendo utilizado (provavelmente em uma invocação anterior de procedimento), é necessário preservar o seu valor, empilhando-o.

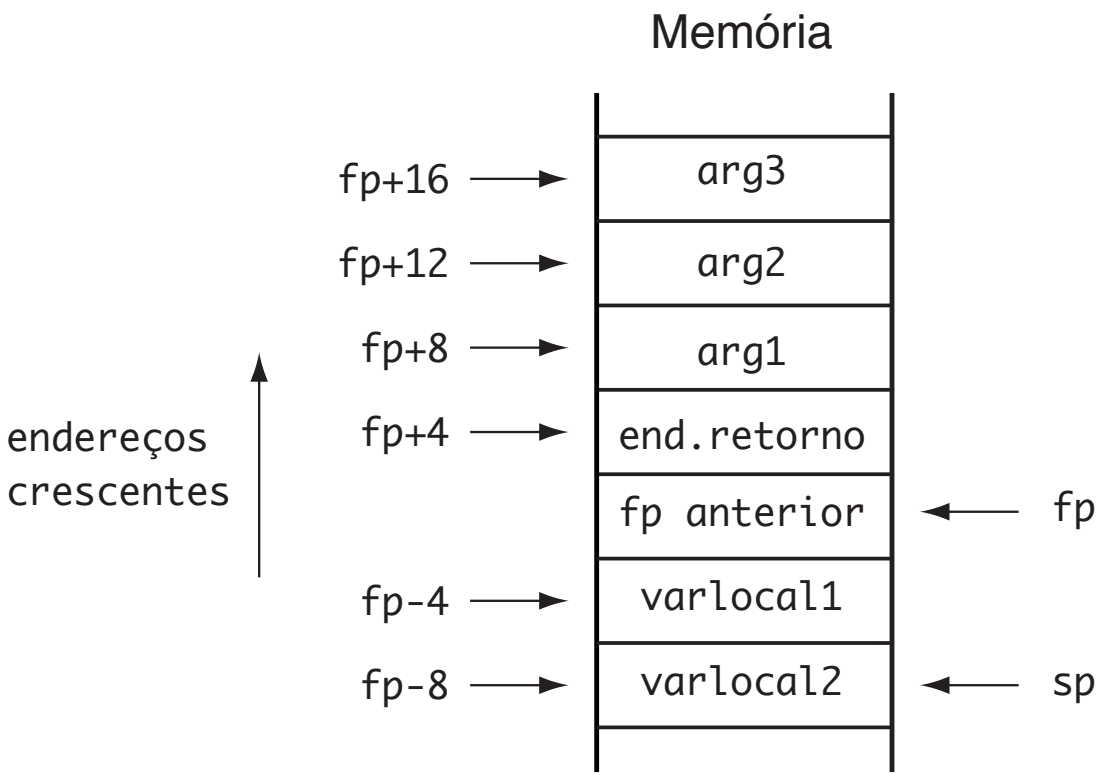
Registrador apontador de quadro

```
     ...
procedim:
     push  fp              @ preserva valor anterior de fp
     mov   fp,sp           @ e faz cópia do apontador da pilha corrente
     ...
```

Registrador apontador de quadro



Registrador apontador de quadro



Registrador apontador de quadro

```
compara_uns:
     push  fp              @ prepara apontador de quadro
     mov   fp,sp
     sub   sp,4            @ reserva espaço para variável local i
     ld    r0,[fp+8]       @ carrega primeiro parâmetro
     push  r0              @ empilha como parâmetro para conta_uns
     call  conta_uns       @ conta bits 1 do primeiro parâmetro
     add   sp,4            @ retira parâmetro da pilha
     st    [fp-4],r0       @ guarda resultado na variável local i
     ld    r0,[fp+12]      @ carrega segundo parâmetro
     push  r0              @ empilha como parâmetro para conta_uns
     call  conta_uns       @ conta bits 1 do segundo parâmetro
     add   sp,4            @ retira parâmetro da pilha
     ld    r1,[fp-4]       @ recupera resultado da primeira chamada
     cmp   r0,r1           @ compara resultados
     jz    final           @ se iguais retorna, r0 com num. bits 1
     set   r0,-1           @ caso contrário, indica com r0=-1
final:
     mov   sp,fp           @ desaloca variável local e
     pop   fp              @ desfaz apontador de quadro
     ret                   @ retorna com valor em r0
```

Recursão

- Recursão é um conceito muito importante em programação, sendo útil 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 recursivos em linguagem de montagem.

Fatorial

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

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

Problema

Traduza a implementação recursiva de fatorial na linguagem C, dada a seguir, para linguagem de montagem do LEG.

```
int fatorial(int n)
{
    if (n<=1)
        return 1;
    else
        return n*fatorial(n-1);
}
```

Solução

```
@ ********
@ fatorial
@ ********
fatorial:
     ld    r0,[sp+4]        @ valor de n
     cmp   r0,1             @ n<=1?
     ja    fat1             @ se não, desvia para calcular fatorial
     set   r0,1             @
     ret                    @ retorna fatorial(0) ou fatorial(1)
fat1:
     sub   r0,1             @ calcula recursivamente fatorial(n-1)
     push  r0               @ empilha parâmetro
     call  fatorial         @ chama recursivamente
     add   sp,4             @ retira parâmetro da pilha
                            @ neste ponto, r0 tem valor de fatorial(n-1)
     ld    r1,[sp+4]        @ recupera novamente valor de n
     mov   r2,r0            @ copia fatorial(n-1) em r2
     call  multiplica_otim  @ efetua r0 <- r1*r2
     ret                    @ retorna, r0 tem valor de fatorial(n)
```