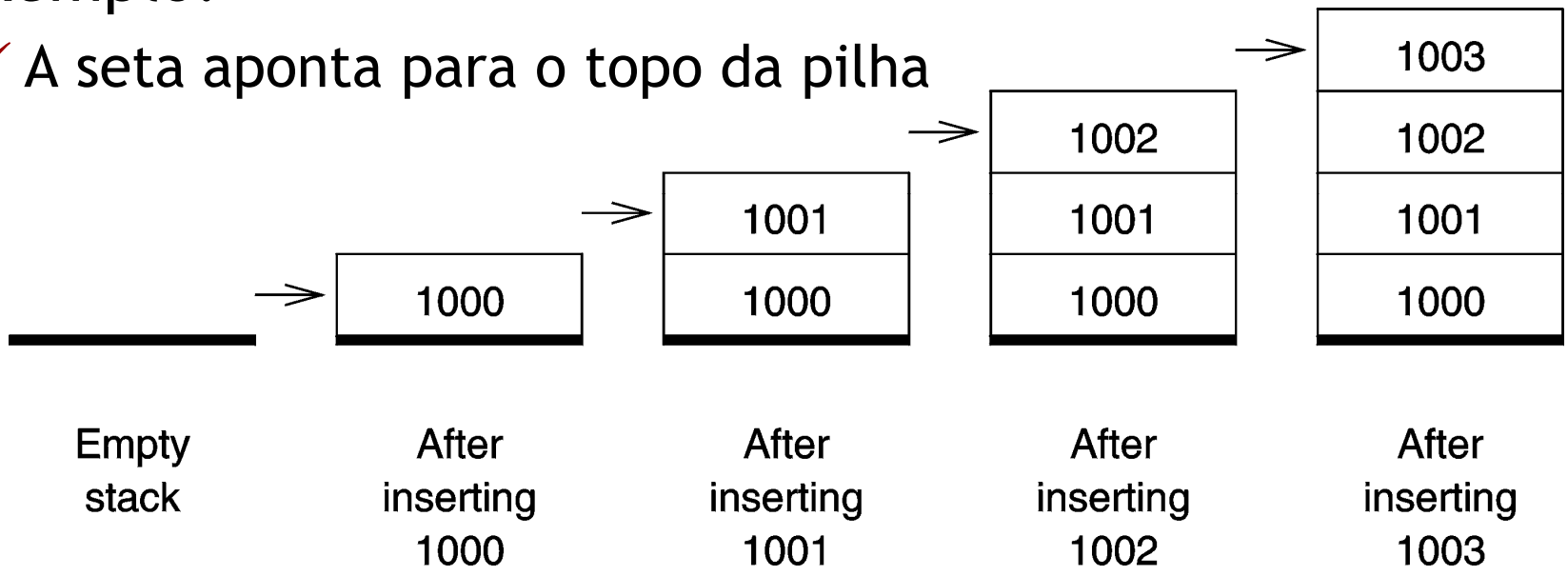
Assembly NASM/IA-32
Subrotinas e a Pilha

Introdução

- Implementação da Pilha no Pentium
- Uso da Pilha
  - ✓ Guardar valores temporários
  - ✓ Subrotina
    - ✓ Passagem de parâmetros
    - ✓ Endereço de retorno
    - ✓ Variáveis locais
- Registo de activação
- Retorno de resultados
- Exemplos de subrotinas
- Programas assembly com vários módulos

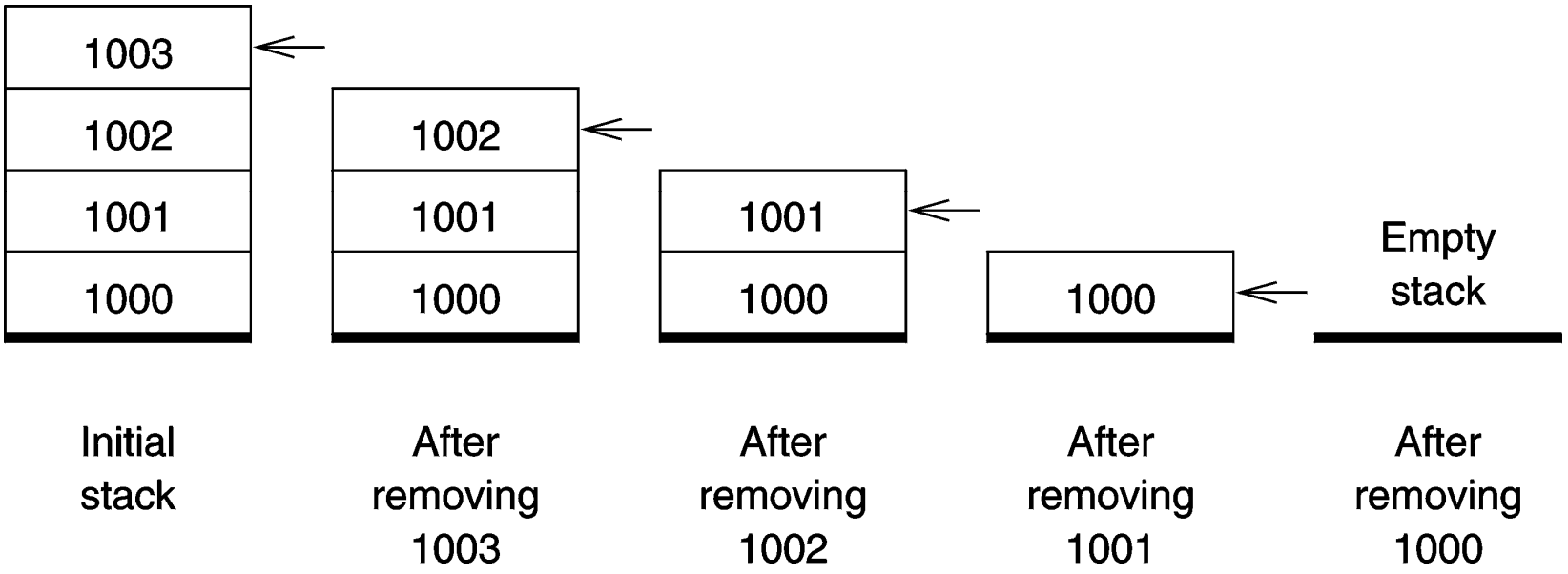
O Que é a Pilha - Revisão

- Estrutura em que o último elemento a ser inserido é o primeiro a sair – LIFO (Last In First Out)
- Duas operações básicas
  - ✓ Push → coloca elemento no topo da pilha
  - ✓ Pop → retira elemento do topo da pilha
- Exemplo:
  - ✓ A seta aponta para o topo da pilha



O Que é a Pilha - Revisão

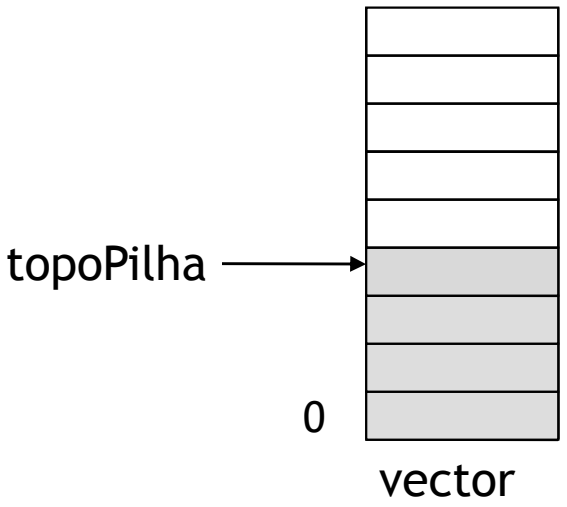
- Estrutura em que o último elemento a ser inserido é o primeiro a sair – LIFO (Last In First Out)
- Exemplo:
  - ✓ A seta aponta para o topo da pilha



Implementação da Pilha

- Possível implementação (sem verificar erros) :

```
tipoE  vector[DimPilha];
int topoPilha = -1;

Por( tipoE E ) {
  topoPilha++;
  vector[topoPilha] = E;
}

tipoE Tirar () {
  tipoE E = vector[topoPilha];
  topoPilha--;
  return E;
}
```



Pilha – Implementação no Pentium

- É utilizado o segmento da pilha
  - ✓ Registo SS → base do segmento
  - ✓ Registo ESP → topo da pilha
- A pilha só suporta elementos de 16 ou 32 bits
- A pilha cresce para baixo
- O topo da pilha (TOS - Top Of Stack) aponta sempre para o último elemento inserido

Pilha – Implementação no Pentium

- Operação de push:

  push src

  ESP ← ESP – 4 (ou 2, se a operação for a 16 bits)

  Mem[ESP] = src

  - ✓ src pode ser um registo, um posição de memória ou uma constante

  - ✓ Exemplos:

    push ebx

    push dword [x]

    push dword 4

Pilha – Implementação no Pentium

- Operação de pop:

  pop dest

  dest = Mem[ESP]

  ESP ← ESP + 4 (ou 2, se a operação for a 16 bits)

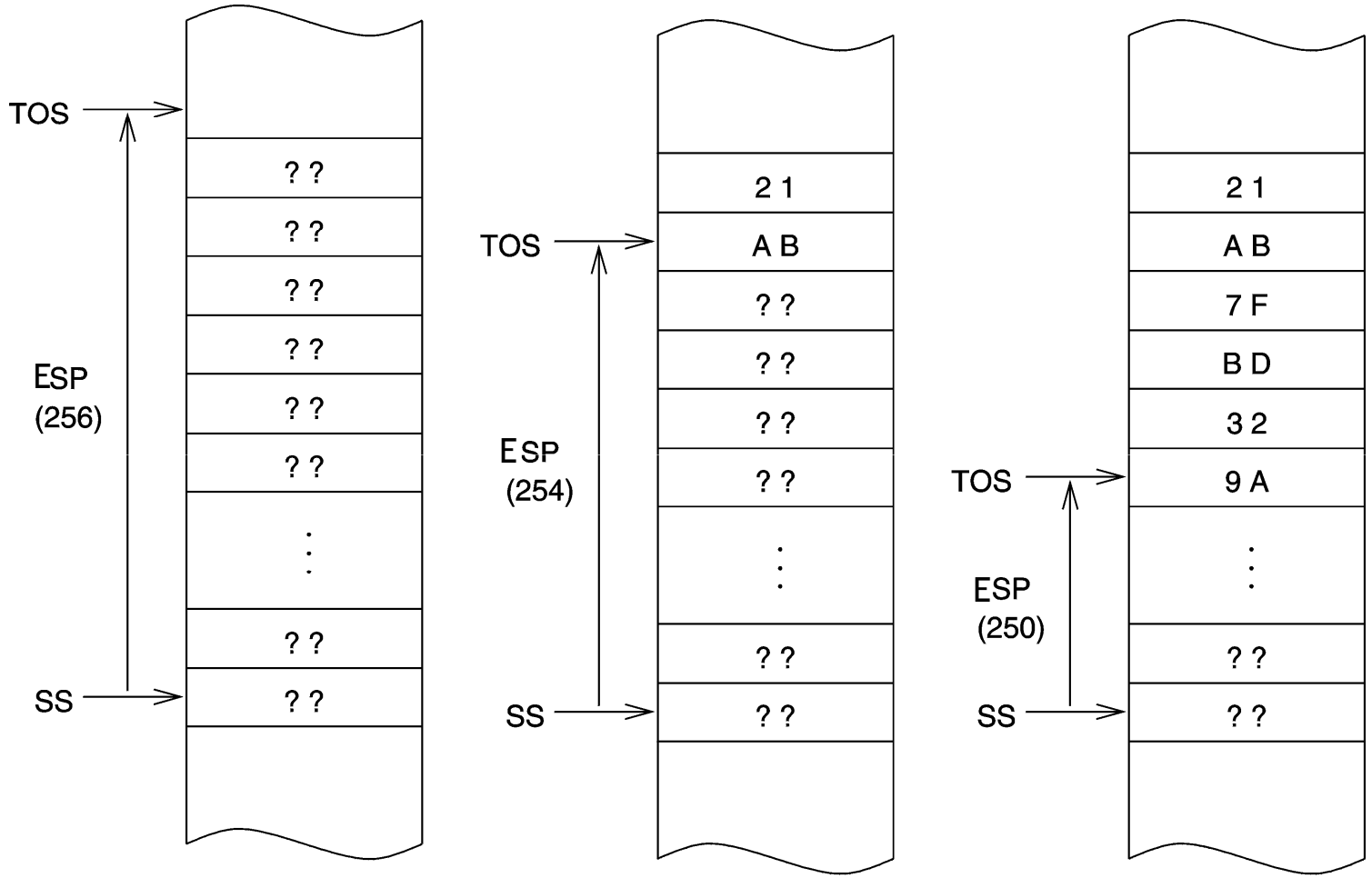
  - ✓ dest pode ser um registo ou uma posição de memória

  - ✓ Exemplos:

    pop ebx

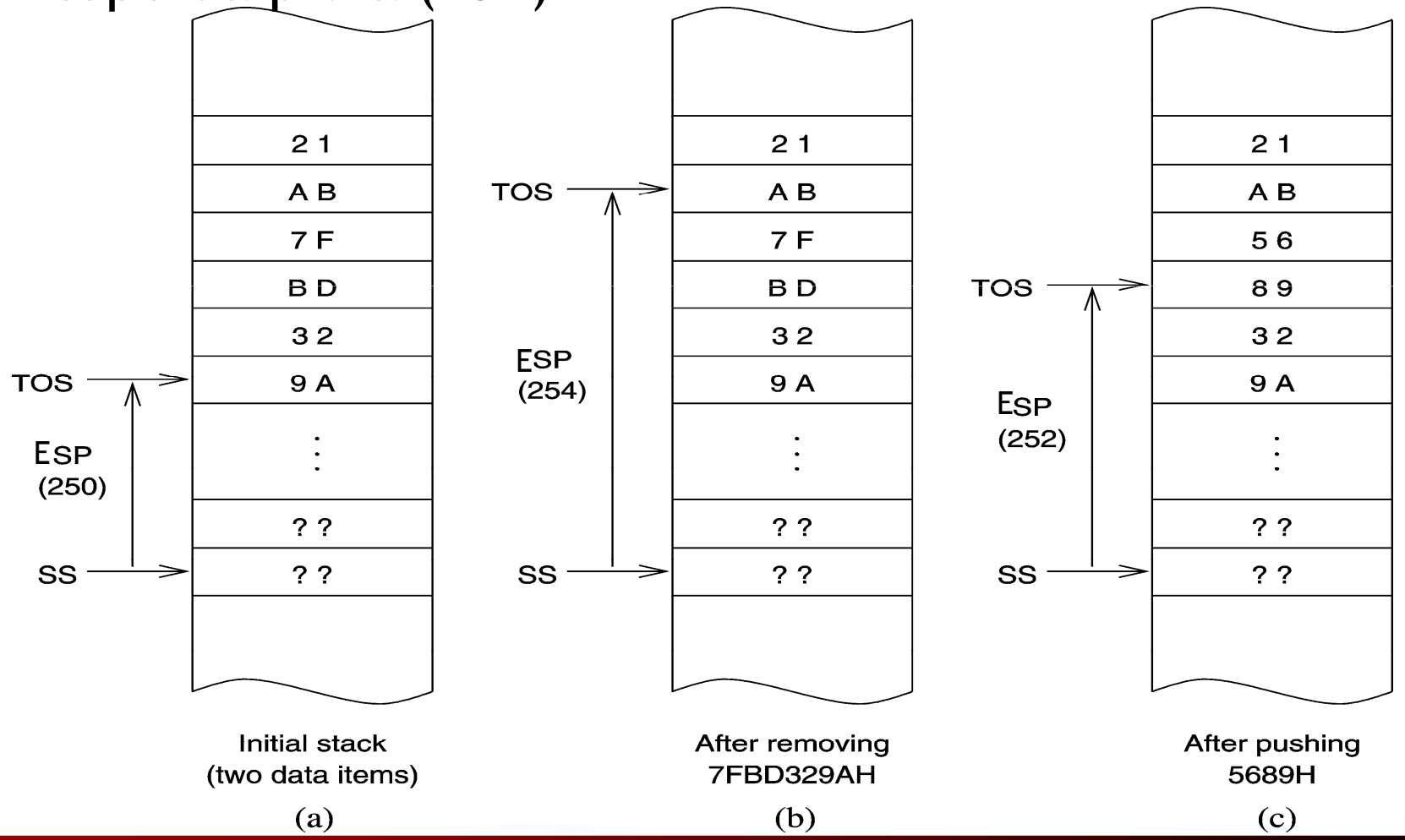
    pop dword [y]

Pilha – Implementação no Pentium



Pilha – Implementação no Pentium

- O pop não apaga os elementos, apenas altera o topo da pilha (ESP)



Pilha – Implementação no Pentium

- Mais instruções que operam sobre a pilha

pushad

- ✓ Guarda os registos EAX, EBX, ECX, EDX, ESP, EBP, ESI e EDI

popad

- ✓ Coloca os valores que estão no topo da pilha nos registos EAX, EBX, ECX, EDX, ESP, EBP, ESI e EDI

pushfd

- ✓ Guarda o registo EFLAGS

popfd

- ✓ Coloca o valor que está no topo da pilha em EFLAGS

A Zona da Pilha de Execução (ou Stack)

- Além do código e dos dados, um programa escrito numa linguagem imperativa, como o C, ou orientada-por-objectos, como o Java, utiliza outra zona de memória como pilha de execução (ou stack).
  - ✓ A dimensão desta varia ao longo da execução do programa
  - ✓ Esta pilha é usada para manter valores temporários, endereços de retorno das subrotinas, parâmetros das subrotinas entre outros
  - ✓ Usam-se as facilidades do CPU para gerir/utilizar esta pilha

Uso da Pilha – Libertar Registos

- Exemplo:

```
; guardar valores de eax e ebx
push eax
push ebx

; usar eax e ebx para alojar outros valores

; recuperar os valores de eax e ebx
pop ebx
pop eax
```

Uso da Pilha – Subrotinas

- Guardar o endereço de retorno
- Passagem de parâmetros
- Guardar as variáveis locais
- Vamos discutir estes pontos em detalhe

CALL, RET e a Pilha

- CALL e RET usam implicitamente a pilha
- A pilha permite a chamada encadeada e recursiva de subrotinas

`call` *S*

Empilha (*push*) o EIP e salta para *S*:

ESP ← ESP – 4
Mem[ESP] ← EIP } push EIP
EIP ← *S*

`ret`

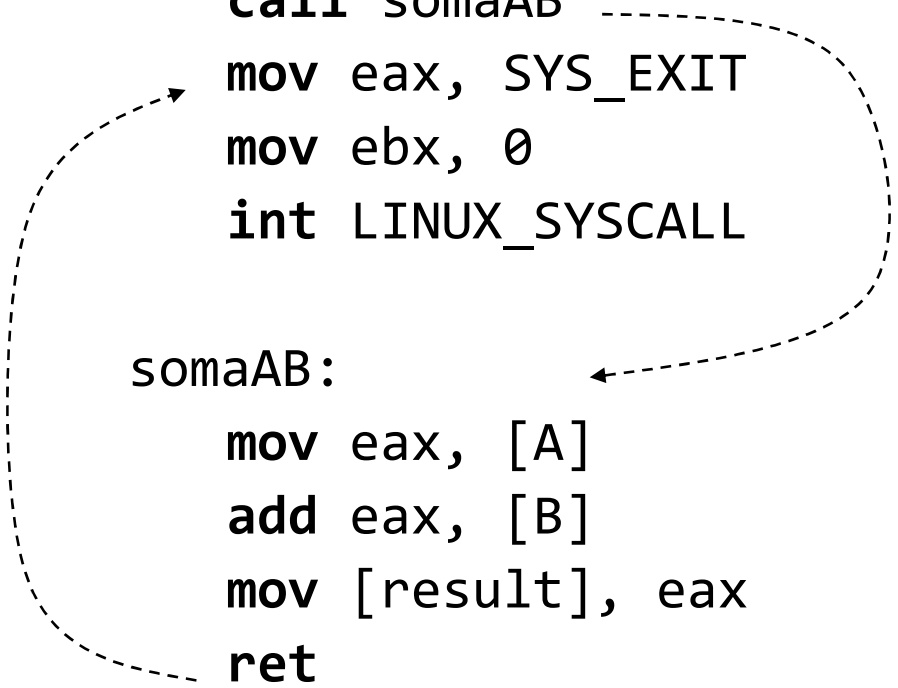
Desempilha (*pop*) para EIP (salta para o endereço guardado no topo da pilha):

EIP ← Mem[ESP]
ESP ← ESP + 4 } pop EIP

Exemplo (usando variáveis globais)

```
SYS_EXIT equ 1
LINUX_SYSCALL equ 0x80

global _start

section .data
  A:            dd 0
  B:            dd 0

section .bss
  result:       resd 1
```

```
section .text
_start:
    mov [A],13
    mov [B], 4
    call somaAB
    mov eax, SYS_EXIT
    mov ebx, 0
    int LINUX_SYSCALL

somaAB:
    mov eax, [A]
    add eax, [B]
    mov [result], eax
    ret
```



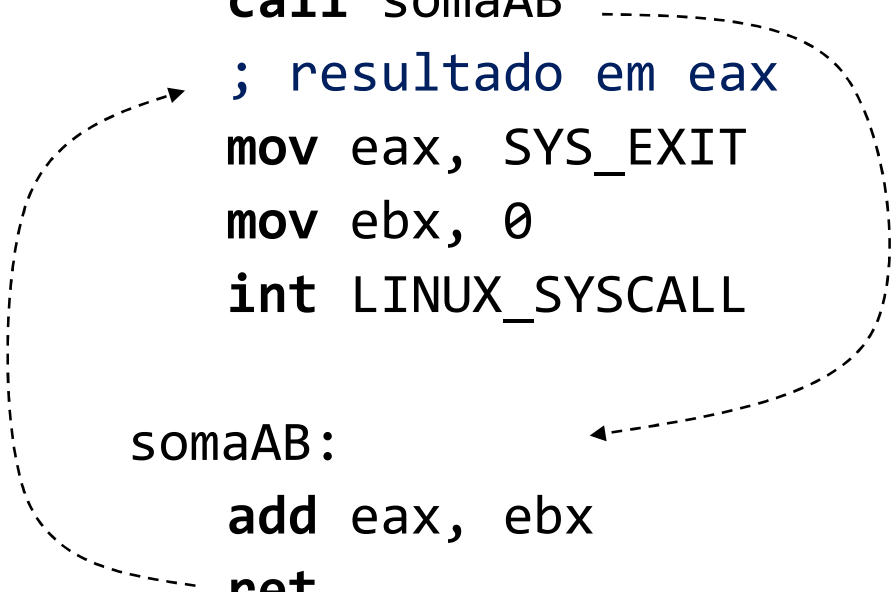
Questões

- Passagem de parâmetros
  - ✓ Onde?
    - ✓ Registos do CPU
    - ✓ Através da pilha de execução
  - ✓ Modo?
    - ✓ Por valor ou cópia
    - ✓ Por referência ou endereço
- Variáveis locais
  - ✓ Onde é que se guardam?
- Resultado (se existir)
  - ✓ Como é que o retorna o resultado de uma função?

Passagem de Parâmetros por Registo

```
SYS_EXIT equ 1
LINUX_SYSCALL equ 0x80

global _start

section .data
  …

section .bss
  …
```

```
section .text
_start:
    mov eax,13
    mov ebx, 4
    call somaAB
    ; resultado em eax
    mov eax, SYS_EXIT
    mov ebx, 0
    int LINUX_SYSCALL

somaAB:
    add eax, ebx
    ret
```



Passagem de Parâmetros por Registo

- Vantagens
  - ✓ Simples
  - ✓ Rápido

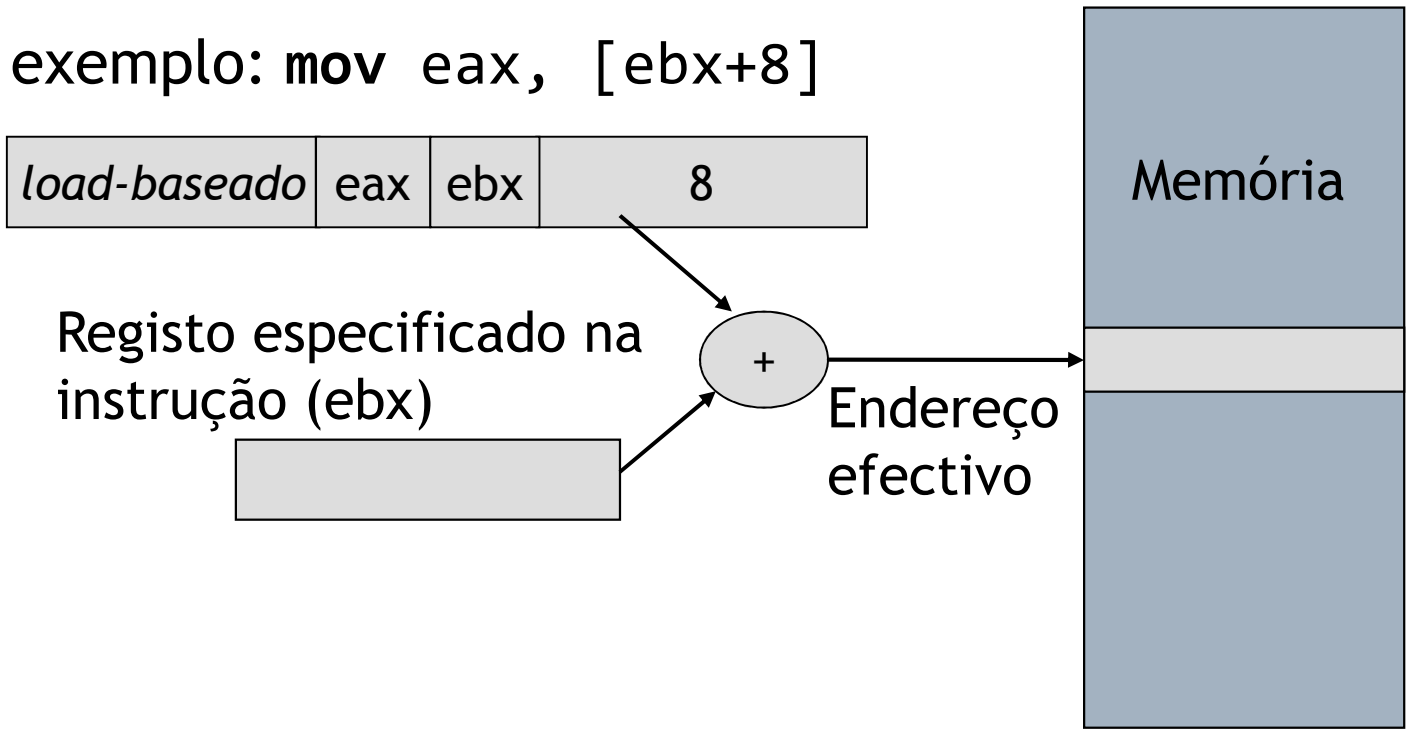
- Desvantagens
  - ✓ Número de parâmetros limitado ao número de registos
  - ✓ Para que os registos estejam livres pode ser necessário guardar o seu valor actual na pilha

Passagem de Parâmetros Através da Pilha

- A solução mais flexível é usar a pilha
- Chamada:
  - ✓ Empilham-se todos os parâmetros (push)
  - ✓ chama-se a subrotina (call)
- Na subrotina:
  - ✓ Acede-se às posições de memória onde ficaram os parâmetros
- Na chamada e na subrotina há que seguir um mesmo conjunto de convenções:
  - ✓ Que parâmetros, sua dimensão, ordem destes na pilha, etc

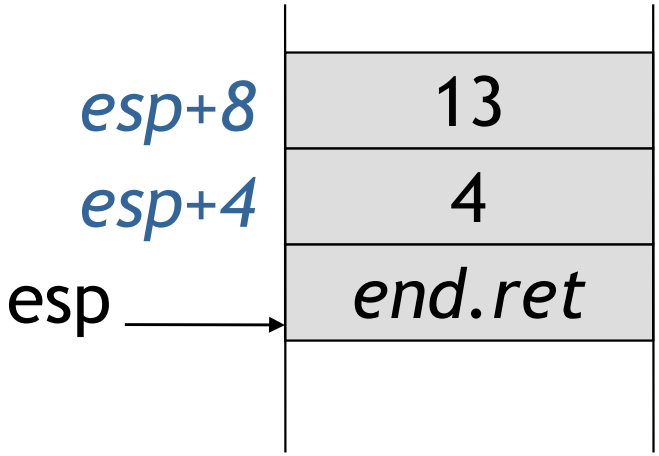
Modo de Endereçamento Baseado

- O endereço efectivo é obtido somando o conteúdo de um registo (registo base ou índice) a uma constante



Passagem de Parâmetros Através da Pilha

```
push dword 13
push dword 4
call myMul
…
…
```




```
myMul:
   …
```

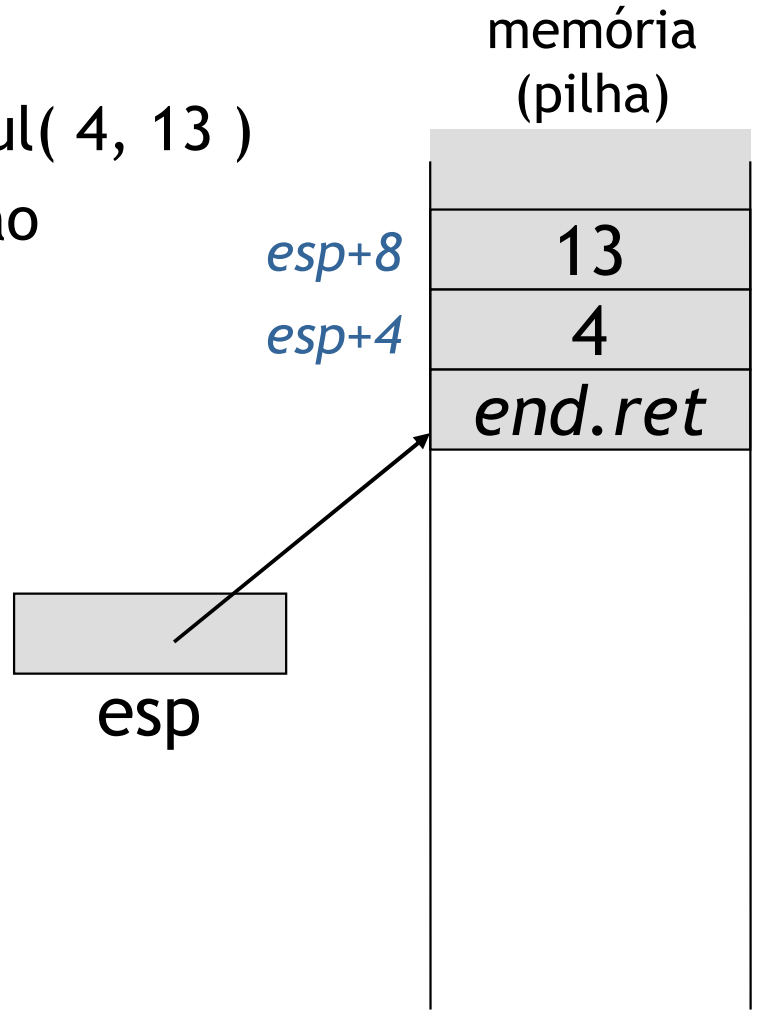
1º argumento → aceder ao endereço dado por esp+4

2º argumento → aceder ao endereço dado por esp+8

…

Acesso aos Parâmetros na Subrotina

- Exemplo:
  - ✓ chamar o procedimento myMul(4, 13)
  - ✓ em myMul queremos aceder ao parâmetro 13: [esp+8]



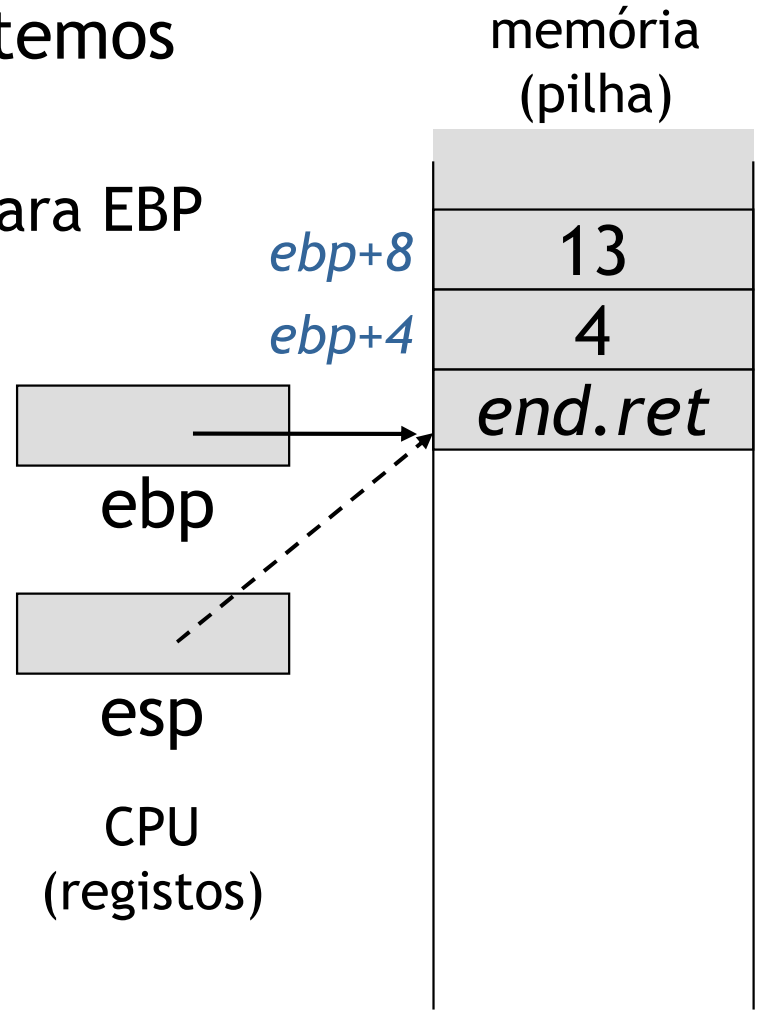

```
    …
    push dword 13
    push dword 4
    call myMul
    …
myMul:
    …
    mov ecx, [esp+8]
    …
    ret
```

O Registo "Frame Pointer"

- Usar o ESP como registo base não é cómodo:
  - ✓ Podem ser feitos pushs e pops o que faz variar distância aos parâmetros
- Normalmente os CPUs suportam outro registo que está intrinsecamente ligado ao stack – o Frame Pointer
  - ✓ O objectivo é manter um endereço de referência ao longo da execução da subrotina (deixando o ESP livre)
  - ✓ Este pode ser usado com registo base para acesso aos parâmetros e às variáveis locais colocadas na pilha (esta zona é o frame de activação da subrotina)
- No Pentium este registo chama-se EBP (Extended Base Pointer)

O Registo "Frame Pointer"

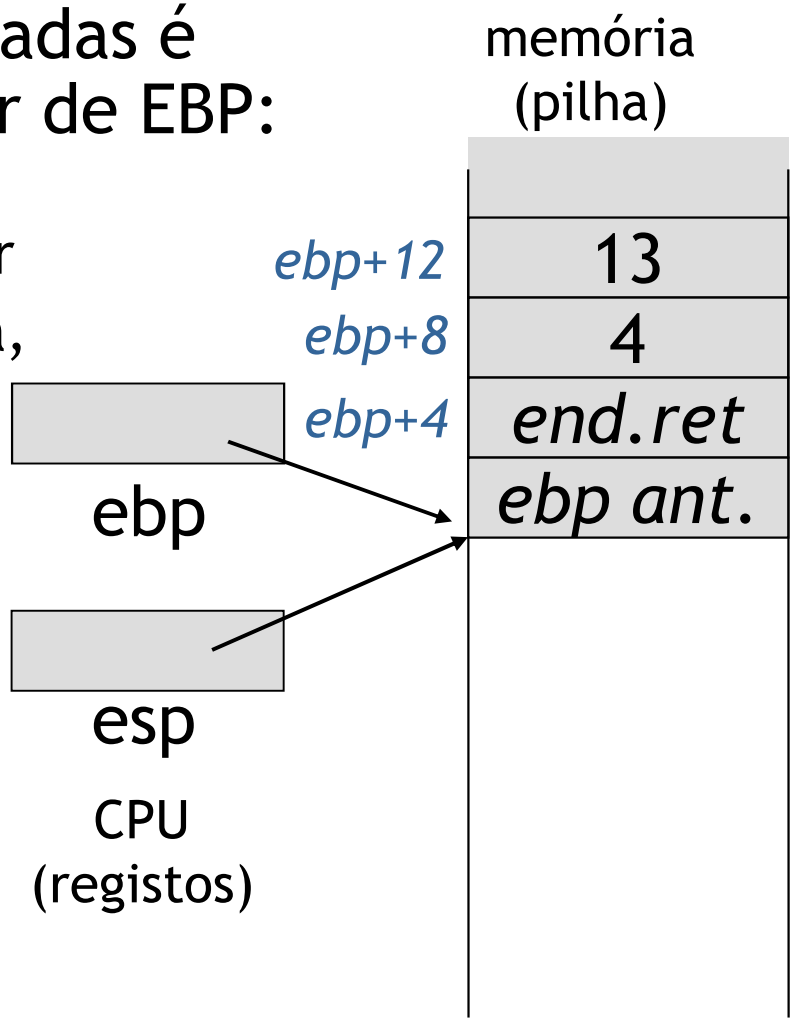
- Para continuar a usar a pilha temos de "libertar" ESP
  - ✓ Copia-se o endereço em ESP para EBP
- Se ESP mudar, os parâmetros ficam na mesma posição relativamente a EBP
  - ✓ Os parâmetros devem ser acessíveis do mesmo modo ao longo de toda a subrotina




```
…
myMul:
    mov ebp, esp
    …
    mov ecx,[ebp+8]
    …
    ret
```

Salvaguarda do Registo "Frame Pointer"

- Quando há chamadas encadeadas é preciso salvar o valor anterior de EBP:
  - ✓ Na entrada, salvar o valor do EBP (no *stack)* antes de o alterar
  - ✓ Quando o procedimento termina, restaurar o EBP (a partir do *stack*) com o endereço anterior




```
    …
myMul:
    push ebp
    mov ebp, esp
    …
    mov ecx,[ebp+12]
    …
    pop ebp
    ret
```

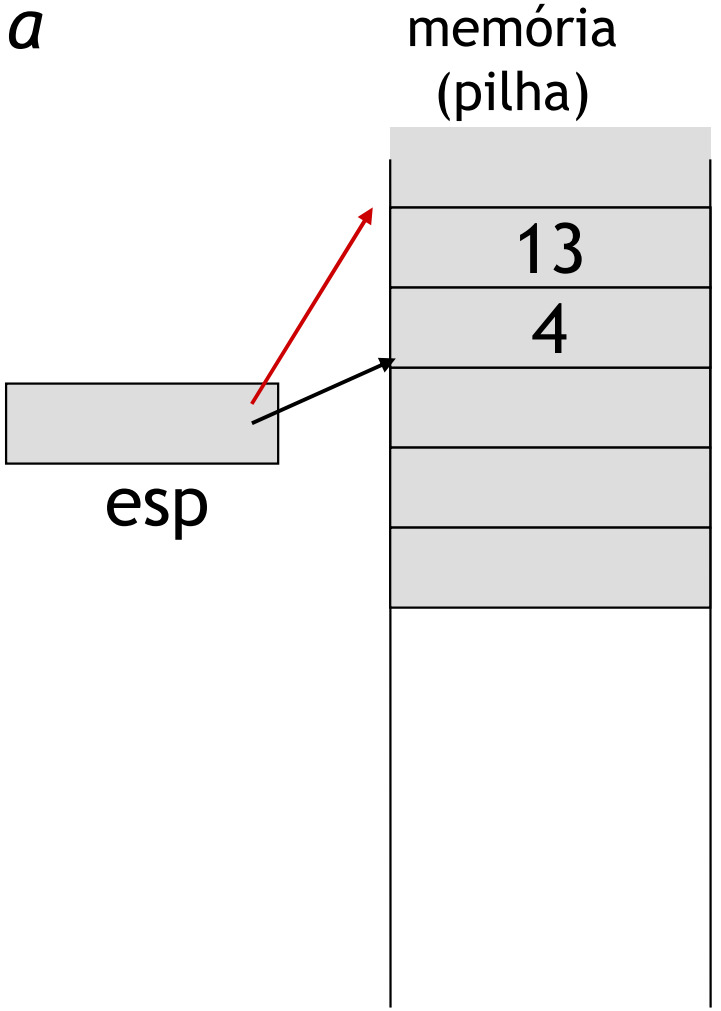
Questões

- A pilha deve ser reposta para que não vá sempre crescendo
  - ✓ Os parâmetros têm de ser retirados da pilha
- Quem é que deve limpar os parâmetros da pilha?
  - ✓ A subrotina que faz a chamada
    - ✓ É necessário colocar código para actualizar o ESP depois de cada chamada
    - ✓ O C utiliza este mecanismo porque permite número variável de parâmetros
  - ✓ A subrotina que é a chamada
    - ✓ Código mais modular, a subrotina trata da sua própria *limpeza*
    - ✓ Não suporta número variável de parâmetros

Limpar os Parâmetros da Pilha
Subrotina Invocadora

- Abordagem *"subrotina que faz a chamada"*

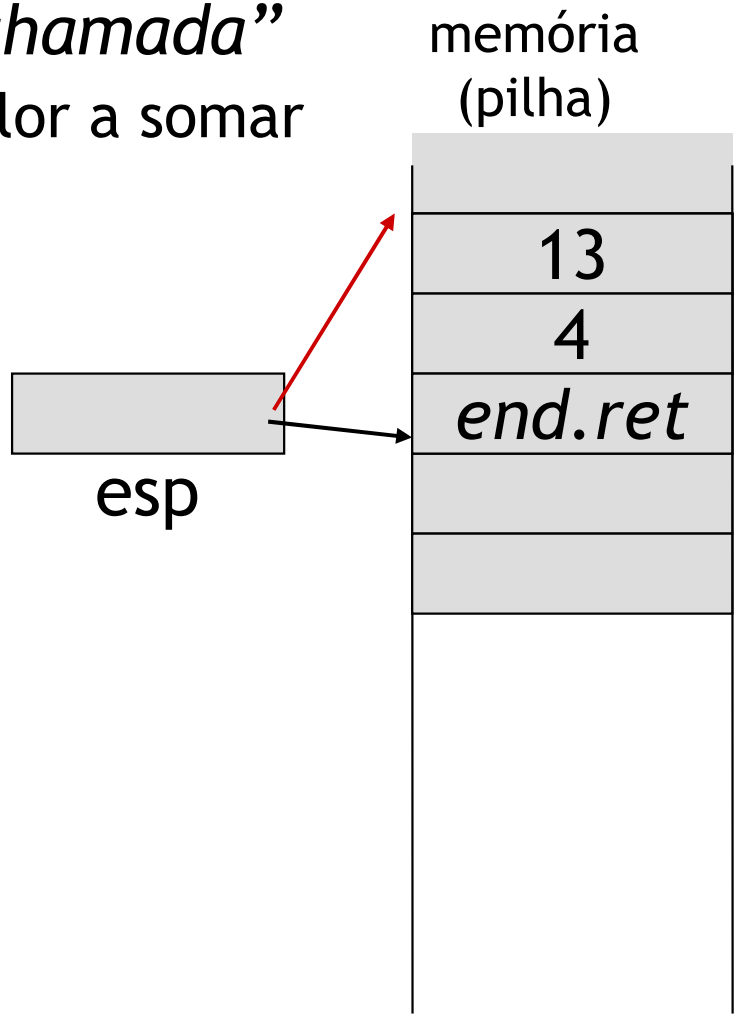
```
    …
    push dword 4
    push dword 13
    call myMul
    add esp, 8
    mov [result], eax
    …
myMul:
    push ebp
    mov ebp, esp
    sub esp, 4
    …
    mov eax, 52
    mov esp, ebp
    pop ebp
    ret
```



Limpar os Parâmetros da Pilha
Subrotina Invocada

- Abordagem *"subrotina que é chamada"*
  - ✓ A instrução **ret** pode ter um valor a somar ao ESP

```
    …
    push dword 4
    push dword 13
    call myMul
    mov [result], eax
    …
myMul:
    push ebp
    mov ebp, esp
    sub esp, 4
    …
    mov eax, 52
    mov esp, ebp
    pop ebp
    ret 8
```



Questões

- Que registos devem ser guardados de forma a que estes possam ser utilizados na subrotina chamada?

```
    mov ecx, 4
    push dword 4
    push dword 13
    call myMul
    mov [result], eax
    cmp ecx, 4 ; qual é o valor de ecx?
    …
myMul:
    push ebp
    mov ebp, esp
    sub esp, 4
    …
    mov ecx, 52
    add eax, ecx
    mov esp, ebp
    pop ebp
    ret
```

Questões

- Que registos devem ser guardados de forma a que estes possam ser utilizados na subrotina chamada?
  - ✓ Os que são utilizados na subrotina
    - ✓ É preciso saber que registos é preciso guardar e quem é que os guarda
      - ✓ Subrotina que faz a chamada
      - ✓ Subrotina que é a chamada
  - ✓ Todos → **pushad**
    - ✓ Operação muito pesada
    - ✓ No Pentium demora 5 ciclos comparado com apenas 1 para guardar um registo

Questões

- Quem é que guarda os valores dos registos?
  - ✓ Subrotina que faz a chamada
    - ✓ Tem de saber quais são os registos utilizados pela subrotina a chamar
    - ✓ É necessário incluir instruções para guardar e recuperar o conteúdo dos registos em cada chamada
    - ✓ Registos que por convenção são guardados pela subrotina que faz a chamada dá-se o nome de **caller-saved**
  - ✓ Subrotina que é a chamada
    - ✓ É o método utilizado normalmente pois torna o código mais limpo e modular
    - ✓ Registos que por convenção são guardados pela subrotina que é chamada dá-se o nome de **callee-saved**

Variáveis Locais

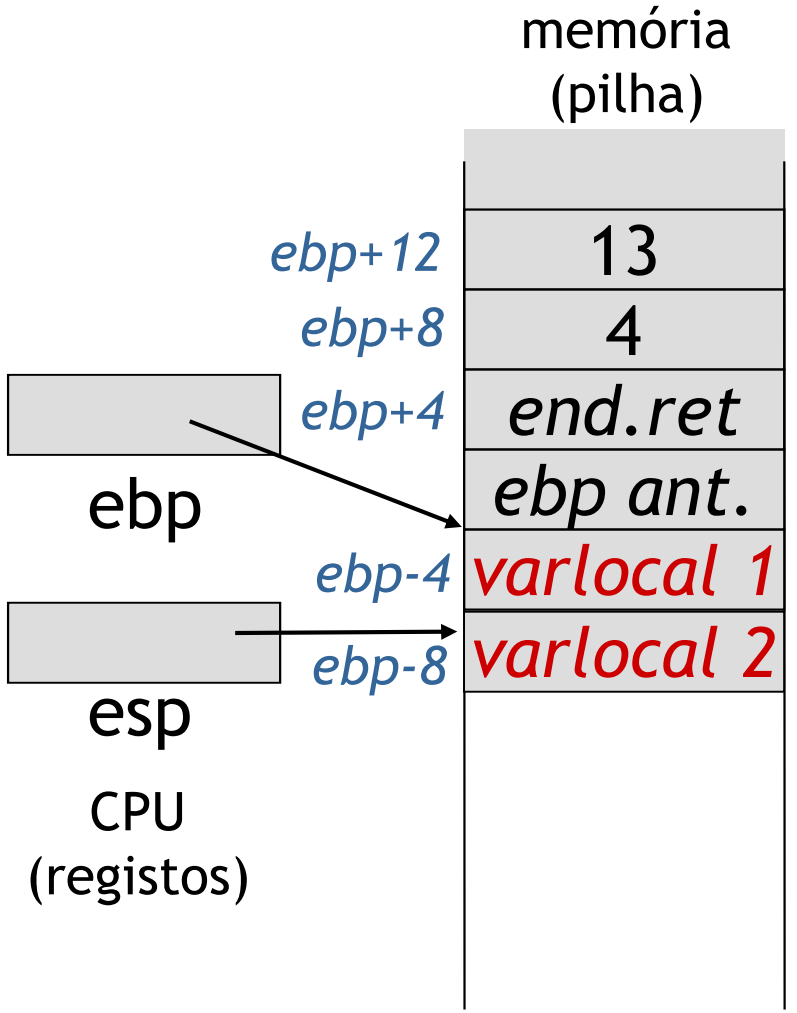
- As variáveis locais são dinâmicas
  - ✓ Só existem enquanto a função (subrotina) executa
  - ✓ O espaço reservado no início é libertado no fim

- Estas variáveis não podem ir para a secção de dados (data e bss) porque:
  - ✓ Esse espaço é estático
  - ✓ Não suporta recursividade

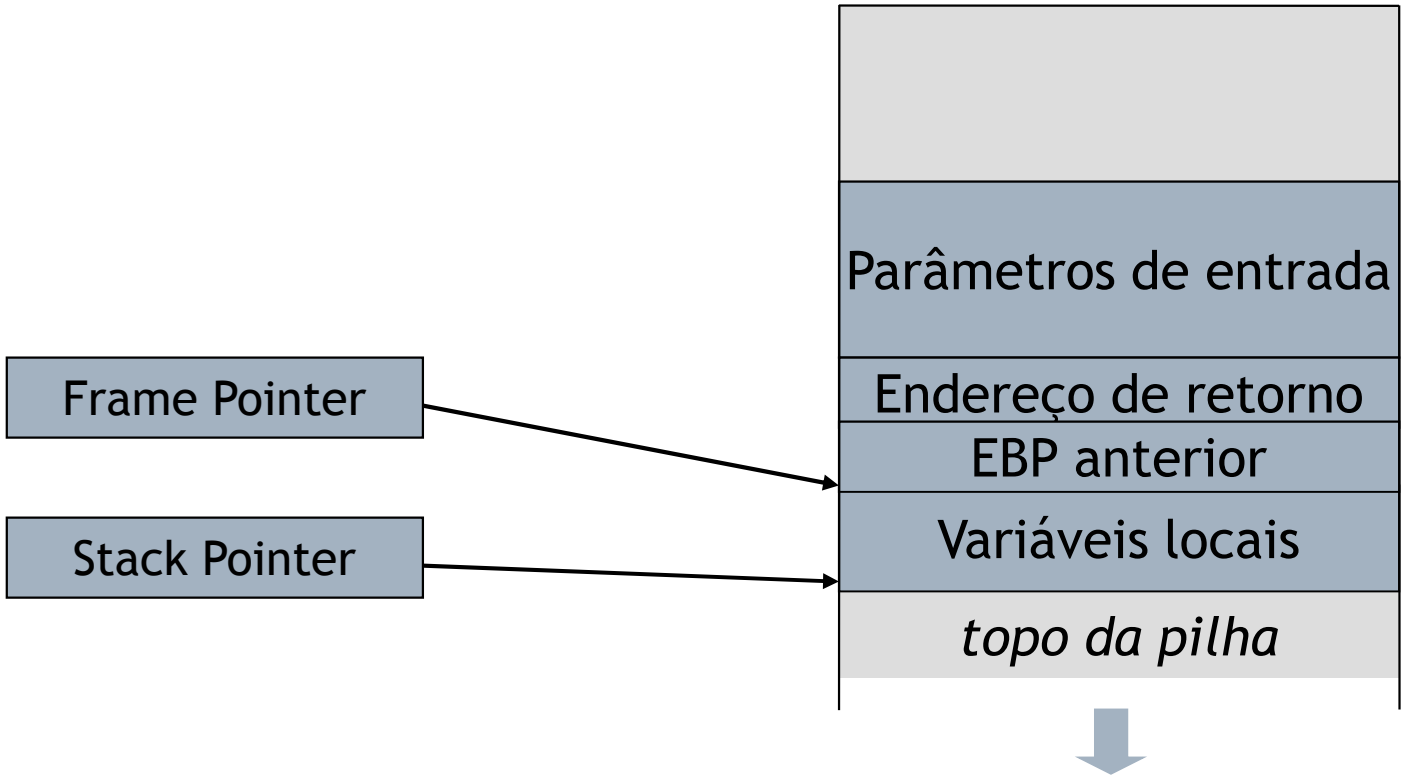
- Variável local → zona de memória no ambiente de execução (frame) da subrotina
  - ✓ Ou seja, na pilha

Variáveis Locais

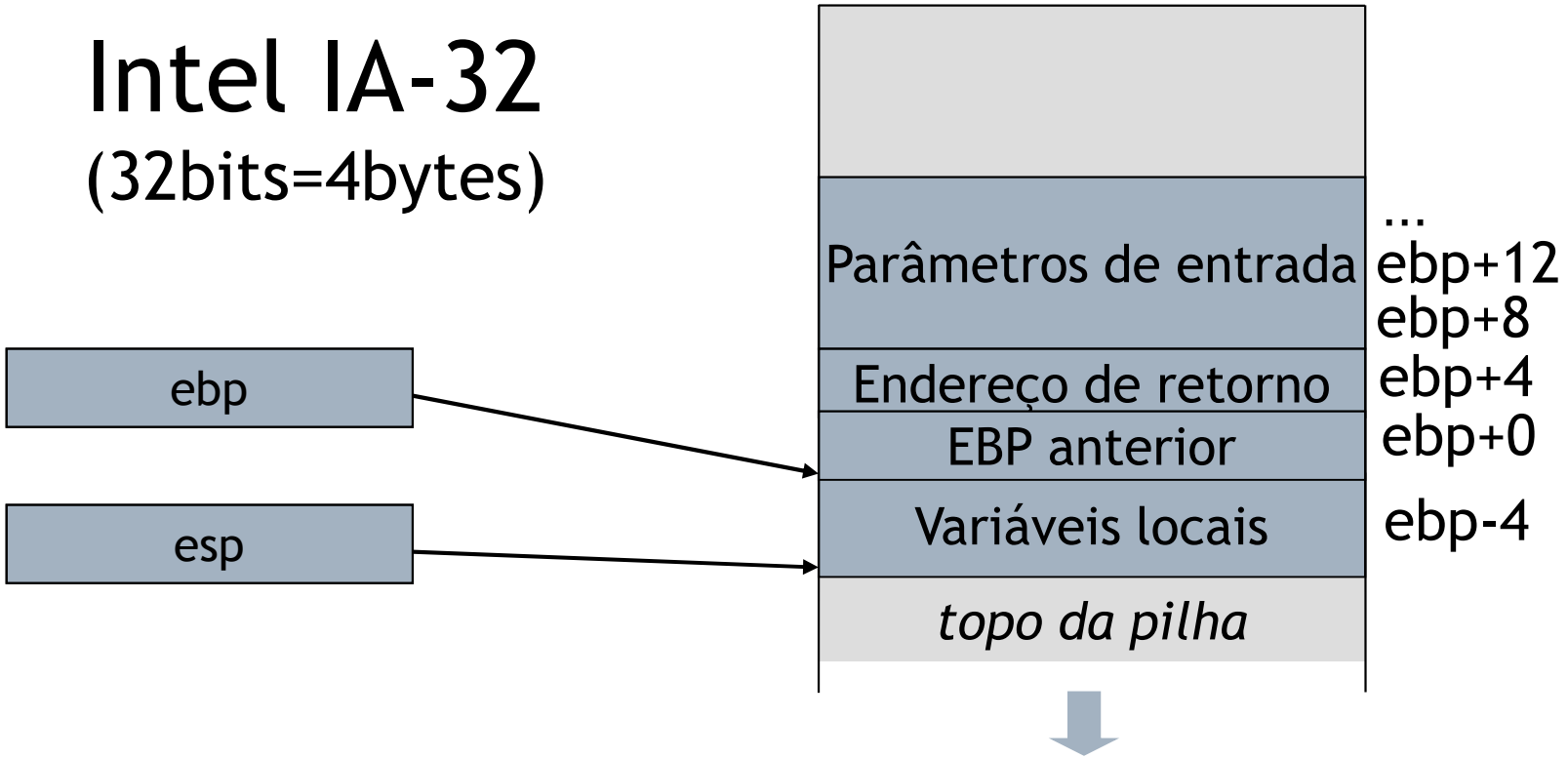
```
    …
myMul:
    push ebp
    mov ebp, esp
    sub esp, 8
    …
    ; aceder à 1ª variável local
    mov [ebp-4], 5
    ; aceder à 2ª variável local
    mov [ebp-8], 10
    …
    mov esp, ebp
    pop ebp
    ret
```



Frame (ou Registo) de Activação



Frame (ou Registo) de Activação



Retorno de Resultados

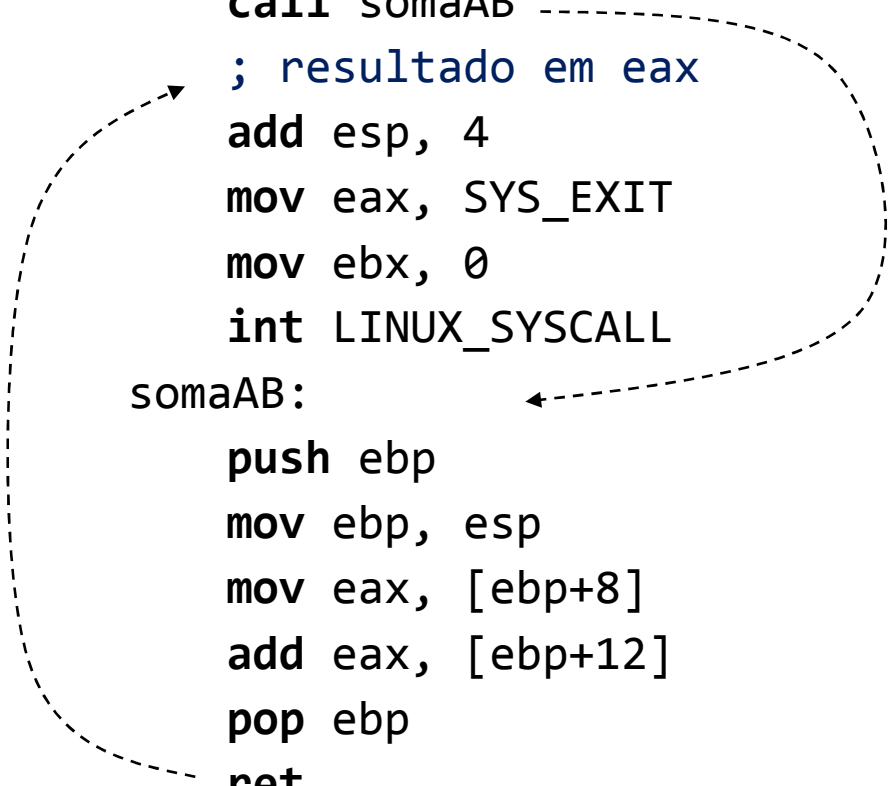
- Se a subrotina é uma função retorna um resultado
  - ✓ Normalmente é usado um registo para retornar o resultado (no IA-32 está convencionado o uso do EAX)

```
    …
    push dword 13
    push dword 4
    call myMul
    mov [result], eax
    …
myMul:
    push ebp
    mov ebp, esp
    sub esp, 4
    …
    mov eax, 52
    mov esp, ebp
    pop ebp
    ret
```

Passagem de Parâmetros por Pilha

```
SYS_EXIT equ 1
LINUX_SYSCALL equ 0x80

global _start

section .data
  …

section .bss
  …
```

```
section .text
_start:
    push dword 4
    push dword 13
    call somaAB
    ; resultado em eax
    add esp, 4
    mov eax, SYS_EXIT
    mov ebx, 0
    int LINUX_SYSCALL
somaAB:
    push ebp
    mov ebp, esp
    mov eax, [ebp+8]
    add eax, [ebp+12]
    pop ebp
    ret
```



Exemplo: factorial recursivo

- Subrotinas recursivas (que se chamam a si próprias)
- Exemplo: função factorial

$$\text{factorial}(n) = \begin{cases} n * \text{factorial}(n-1) & \text{se } n > 1 \\ 1 & \text{se } n = 0 \text{ ou } n = 1 \end{cases}$$

- Uma implementação em C:

```
long factorial (int n) {
  if (n < 2)
    return 1;
  return n * factorial(n-1);
}
```

Exemplo: factorial recursivo

- Chamar factorial(n), consideremos que n está em ebx:

```
push ebx            ; push do argumento (n)
call factorial      ; o resultado fica em eax
add esp, 4          ; repor a pilha
```

- Implementação de factorial(n):

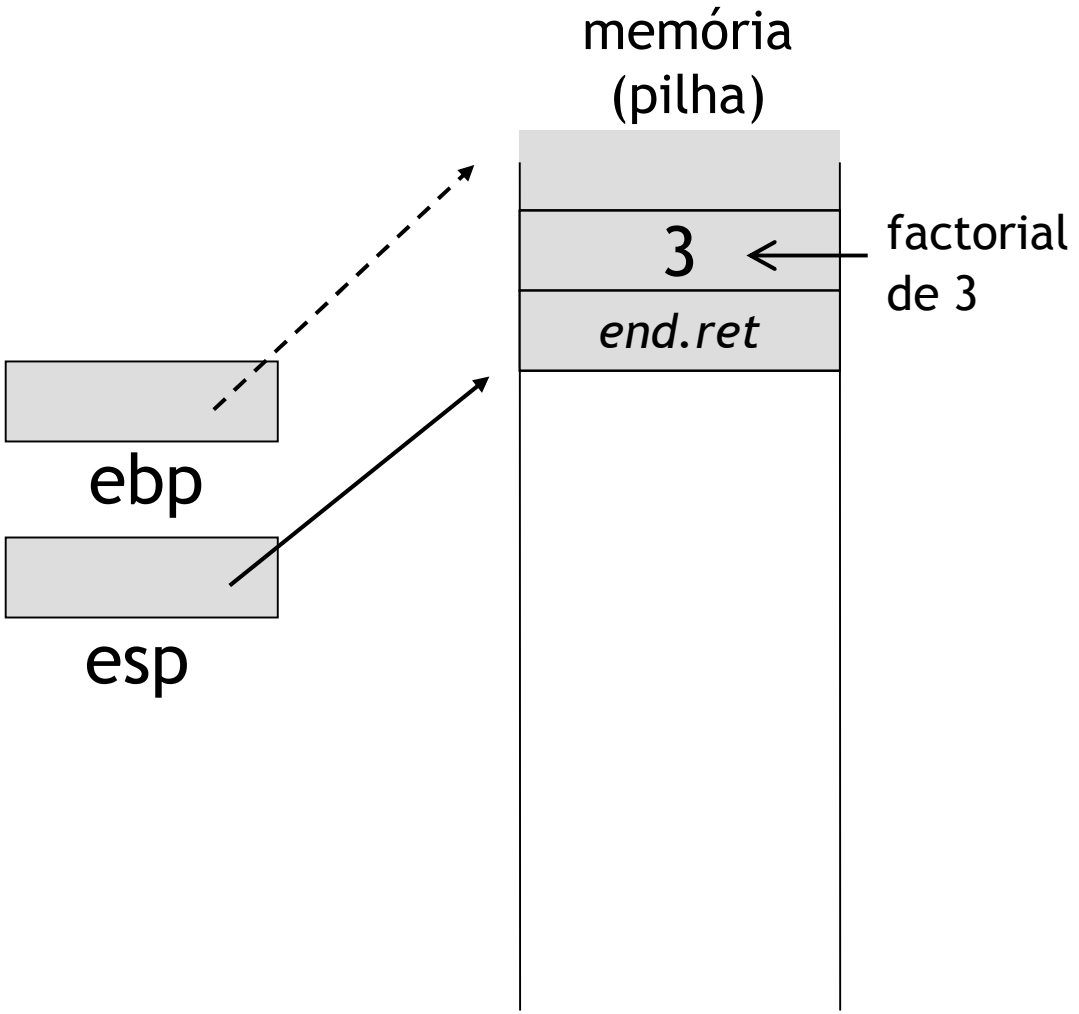
```
factorial:
    …
    push …          ; push do argumento (n-1)
    call factorial  ; chamada recursiva
                    ; o resultado fica em eax
    add esp, 4      ; repor a pilha
    …
```

Exemplo: factorial recursivo

- Vamos calcular o factorial de 3, suponhamos que ebx contém esse valor

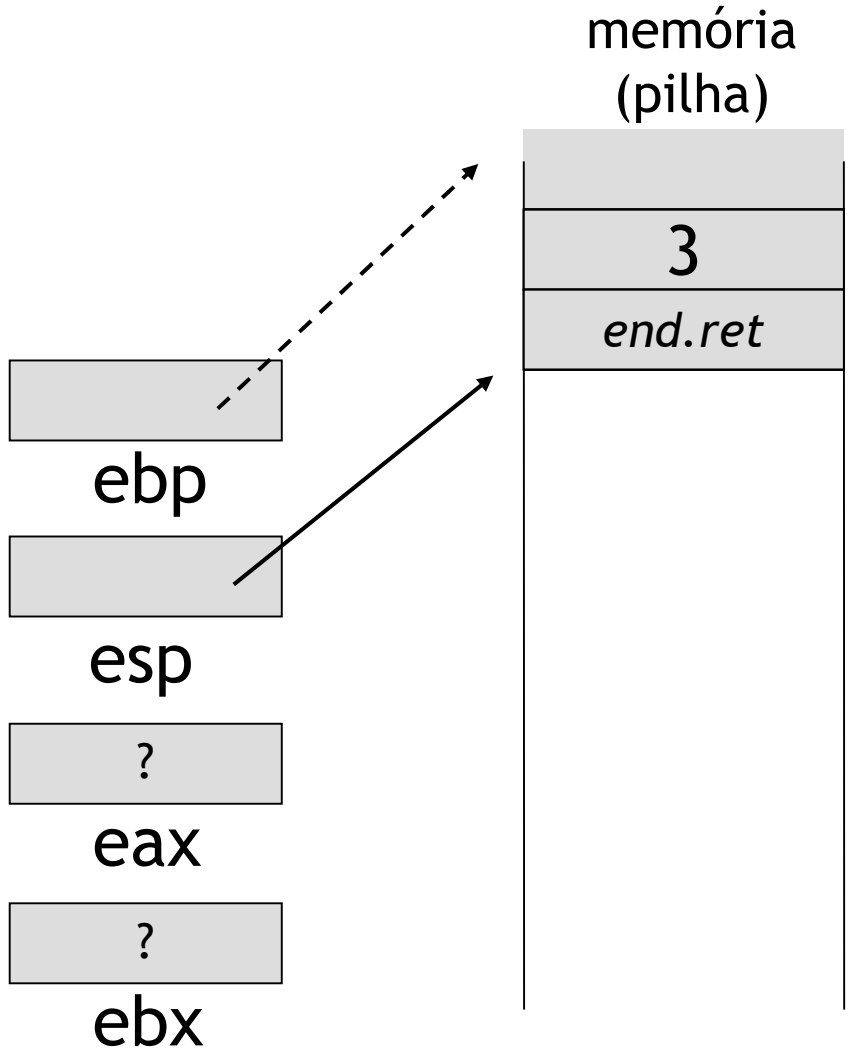
- O código da chamada é:

```
…
push ebx
call factorial
…
```



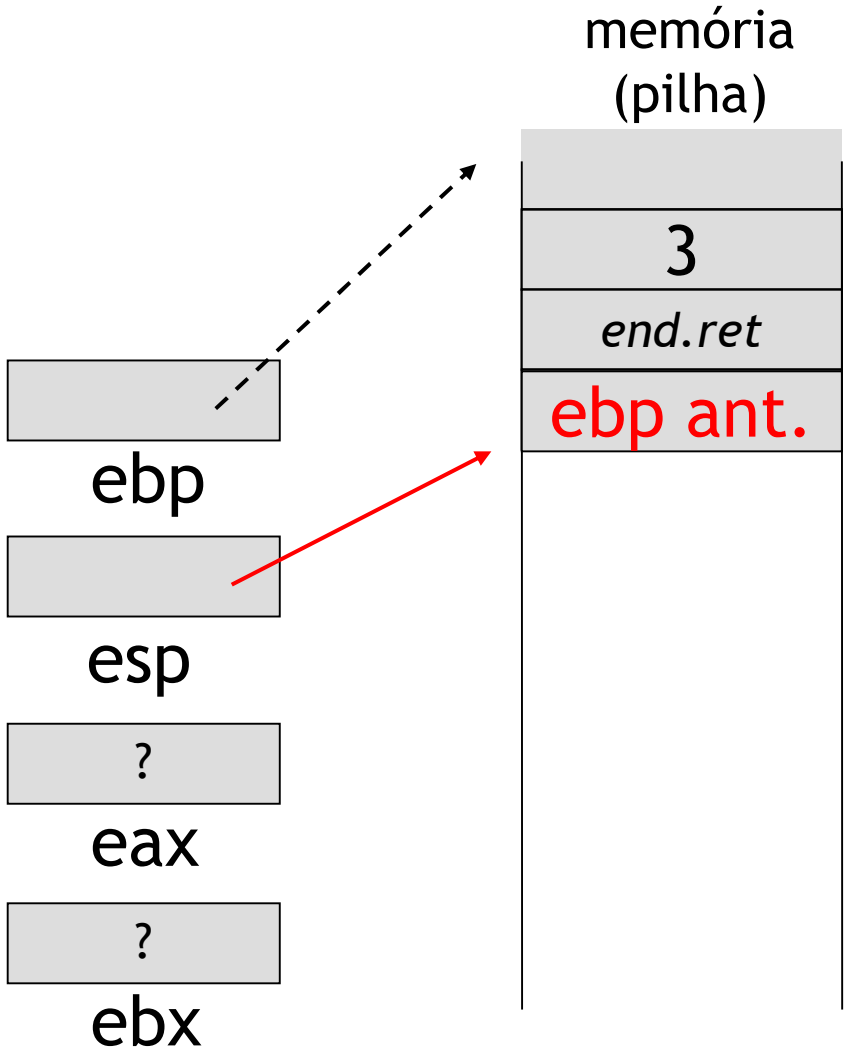
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



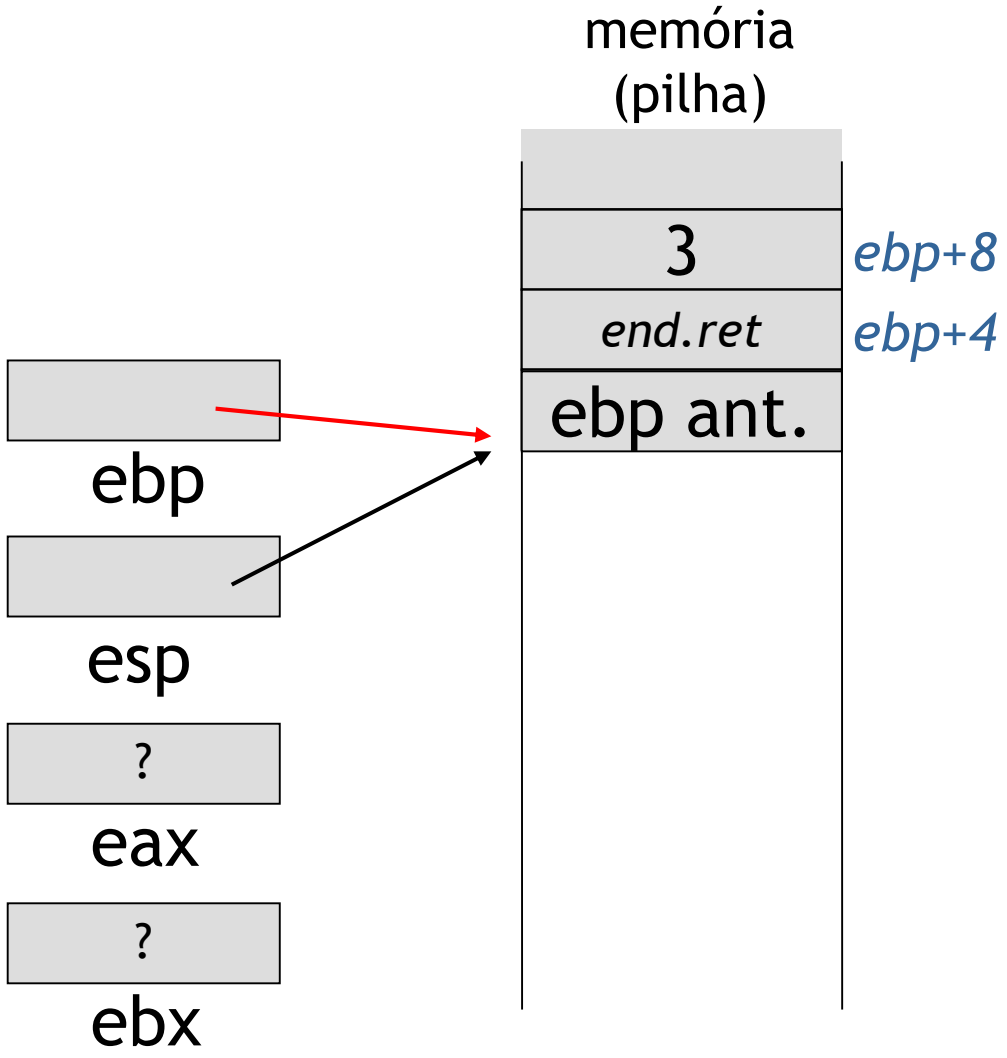
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



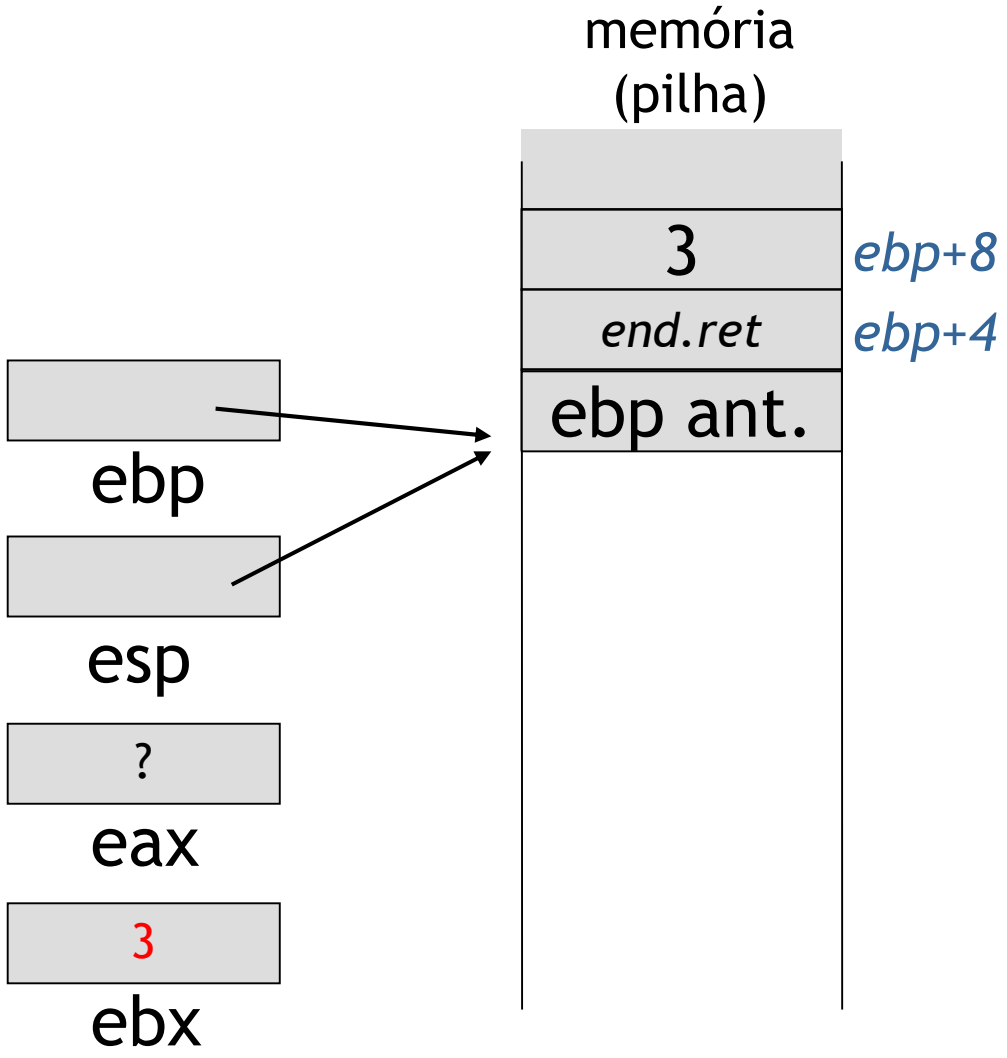
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



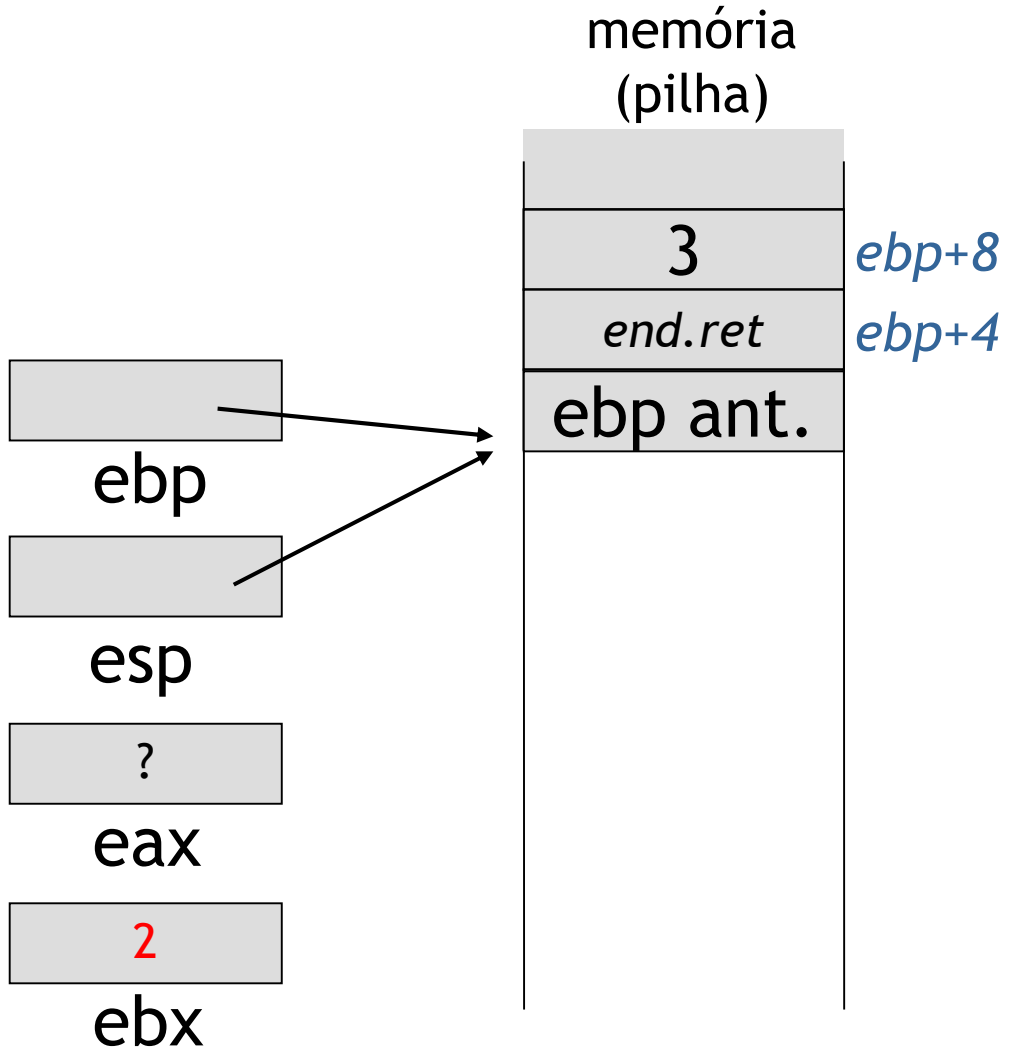
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



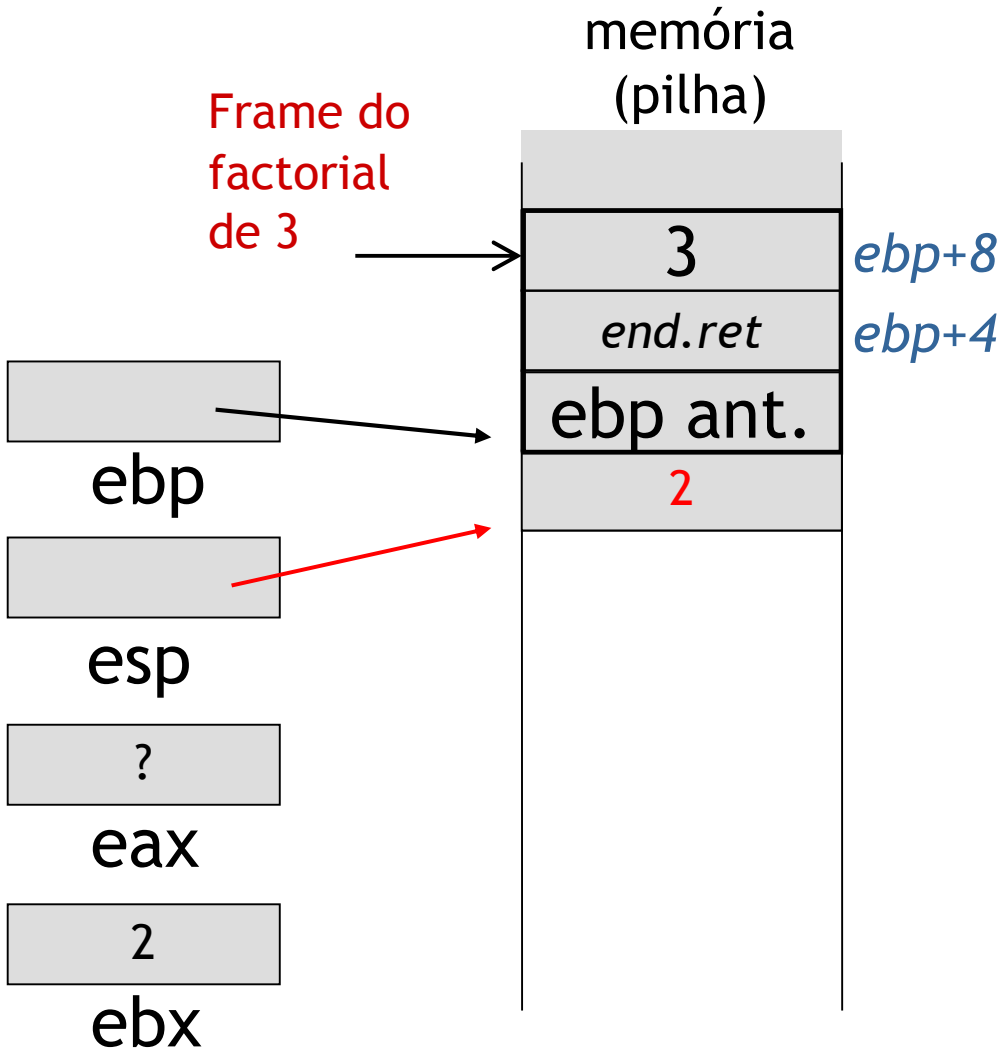
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



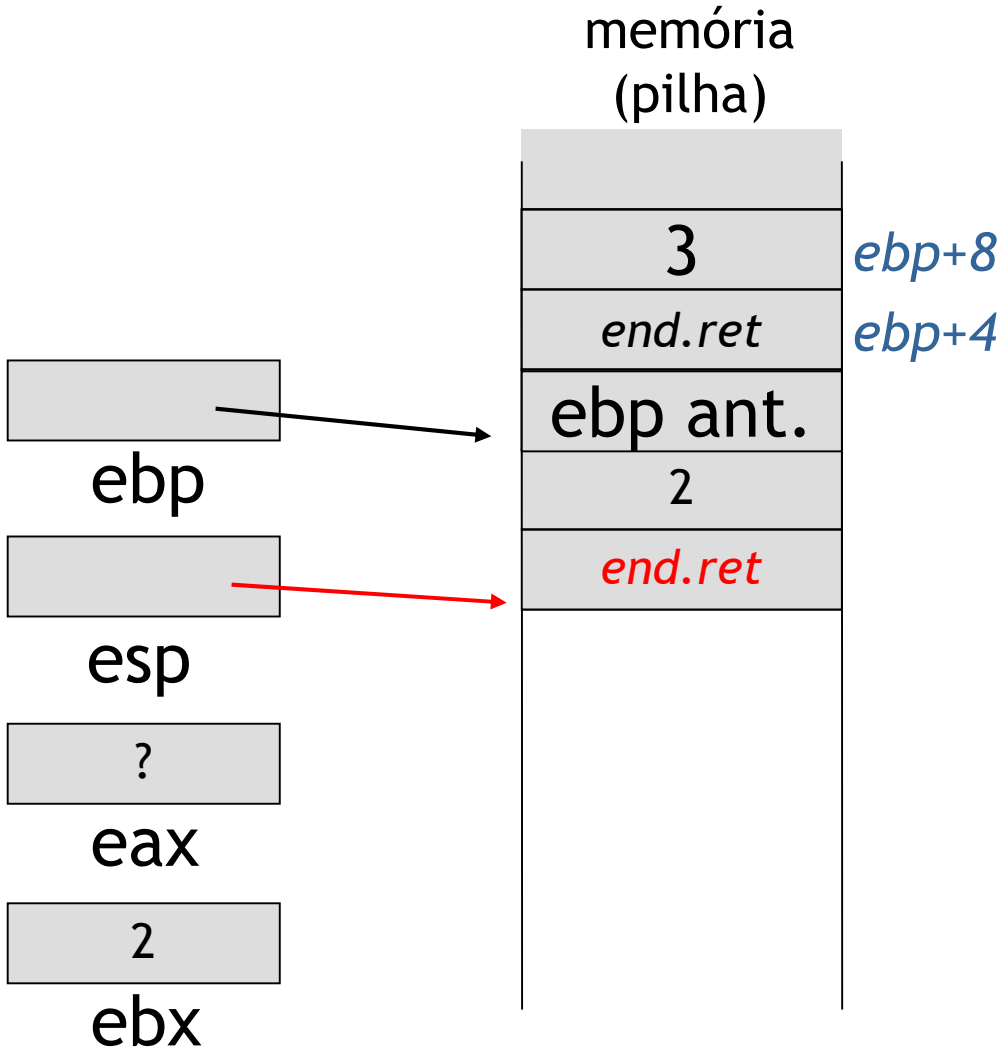
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



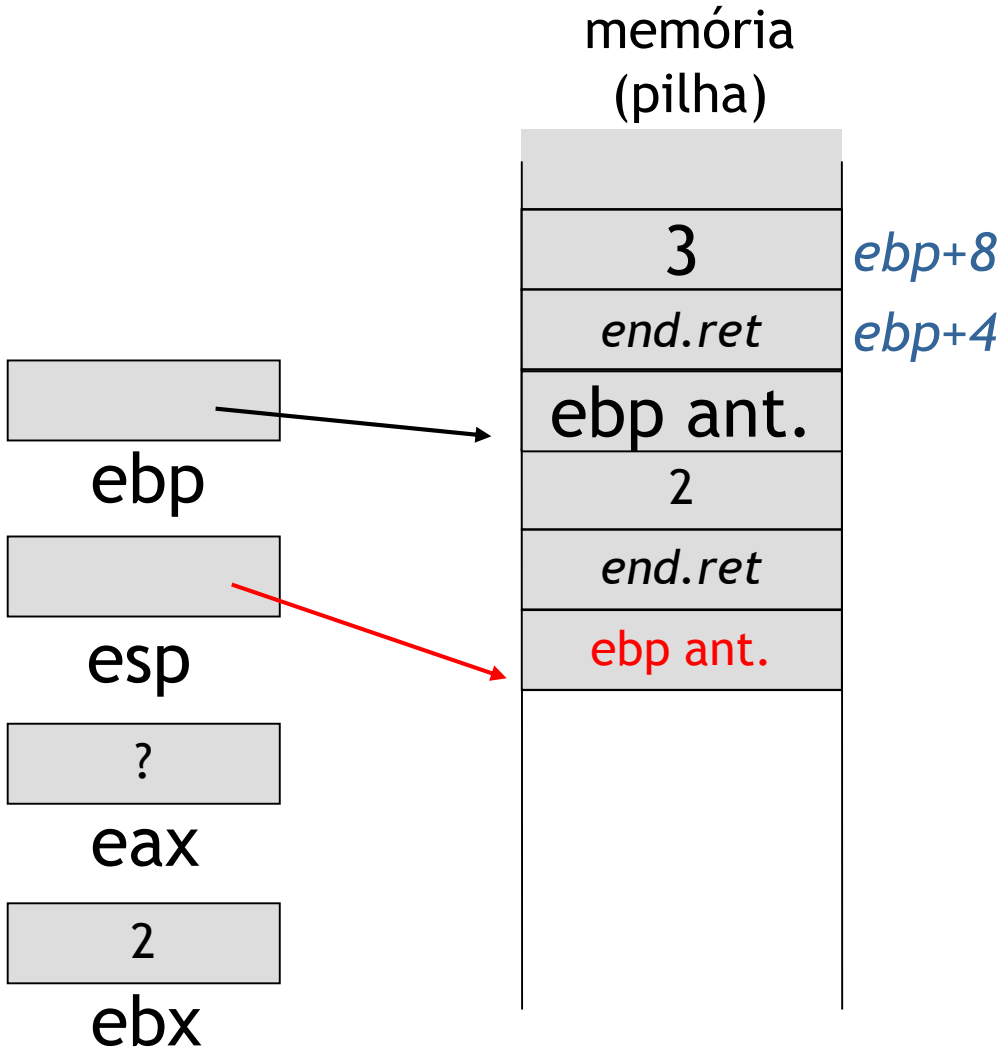
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



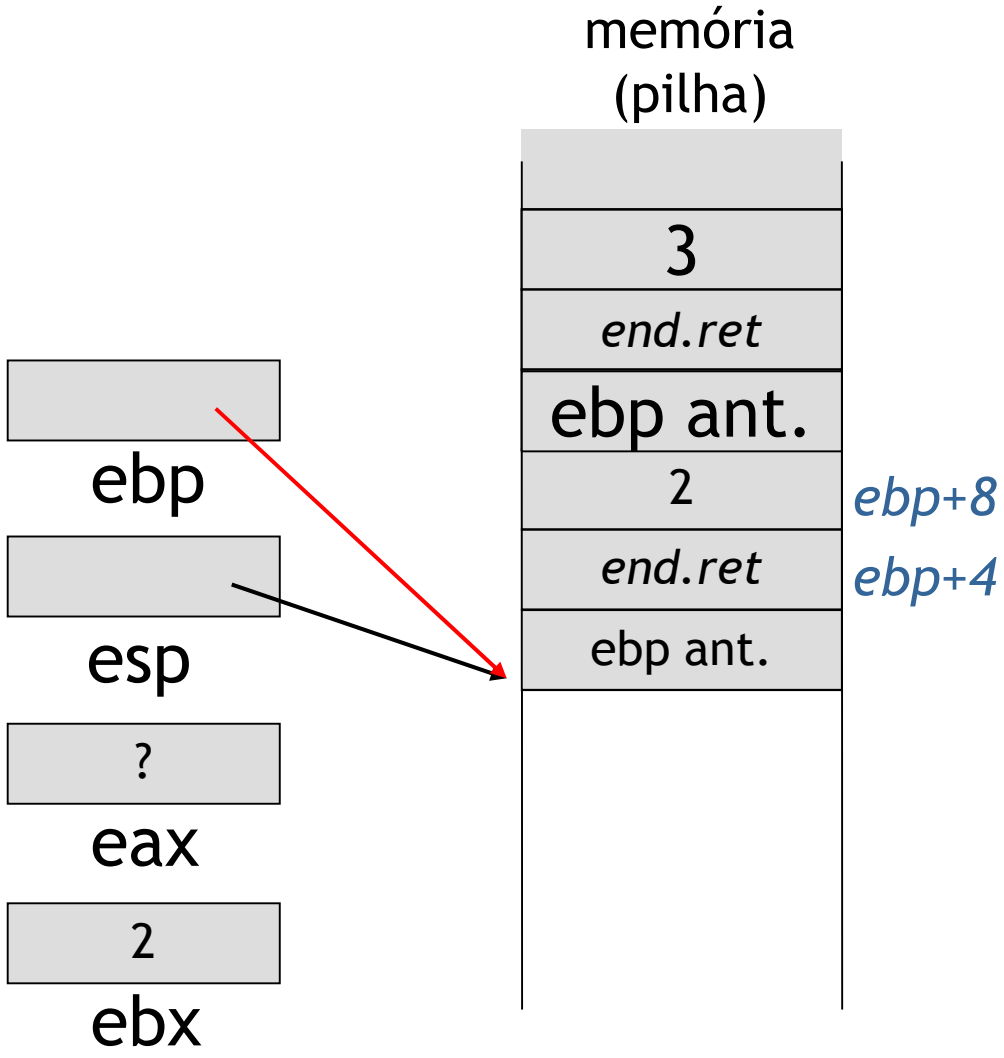
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



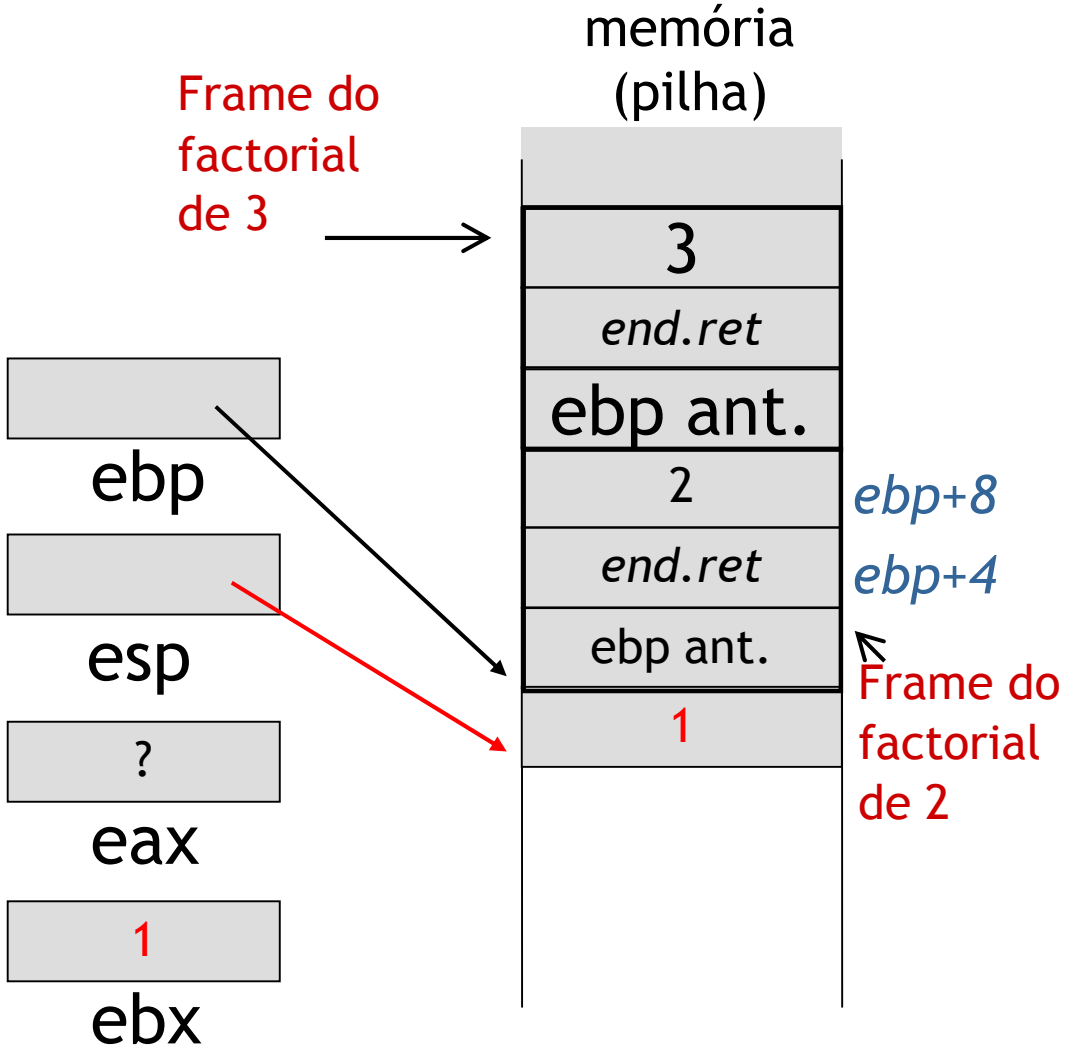
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



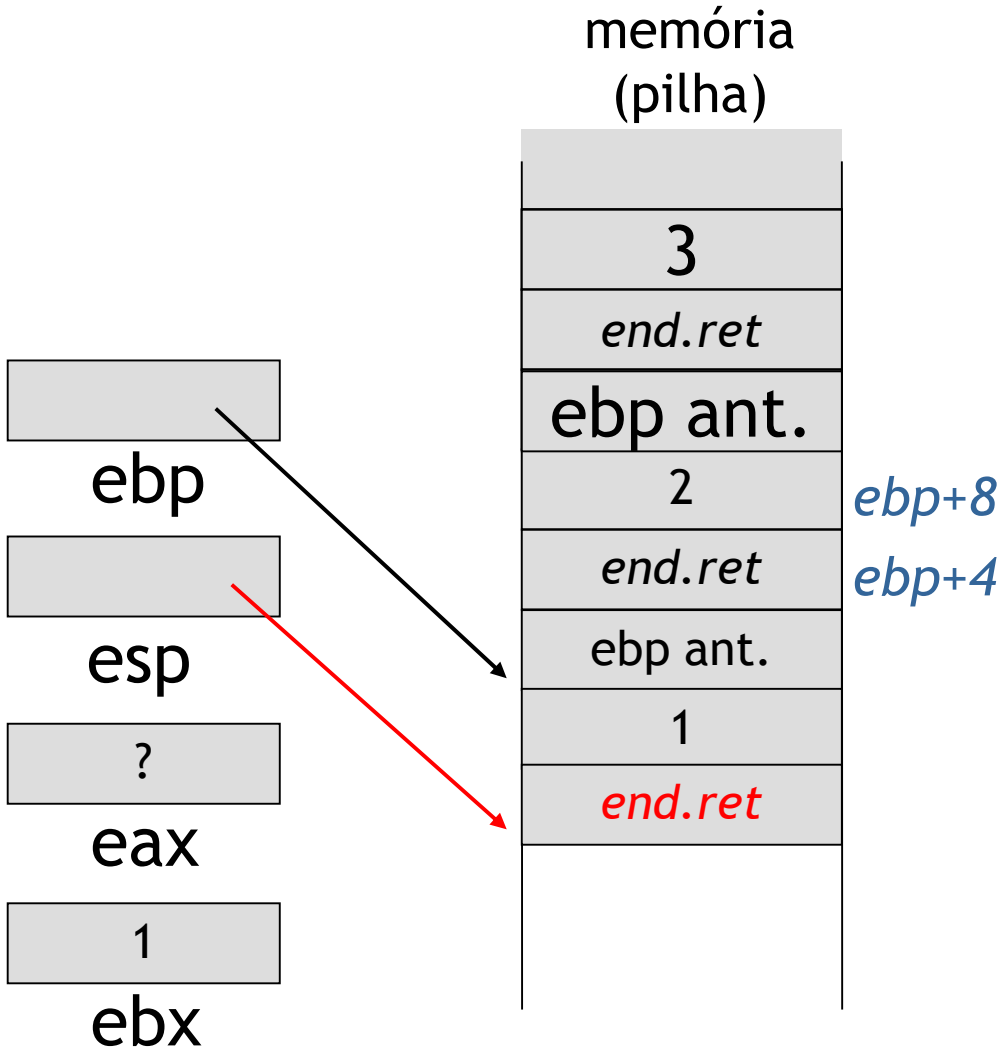
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



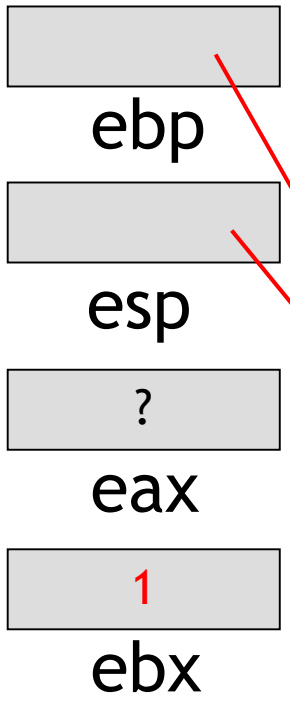
Exemplo: factorial recursivo

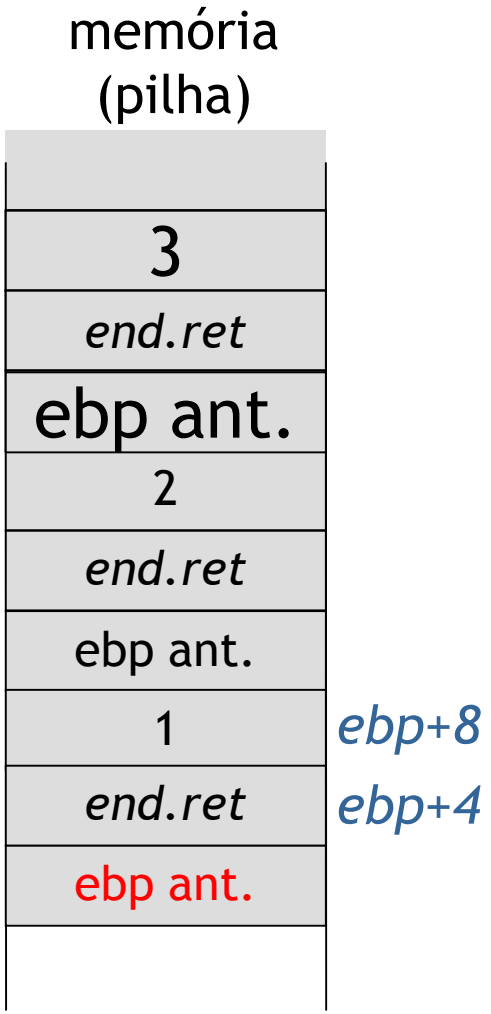
```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



Exemplo: factorial recursivo

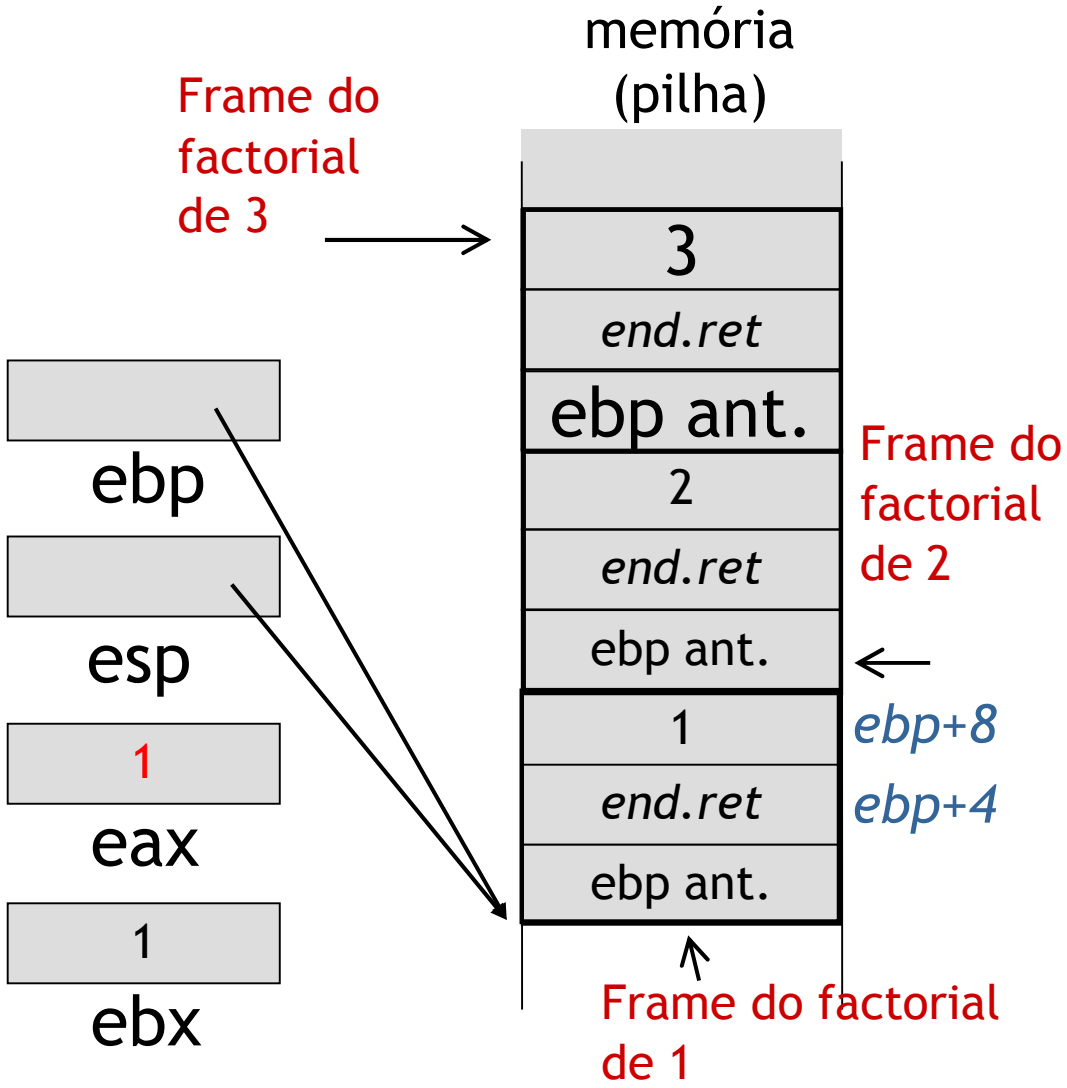
```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```





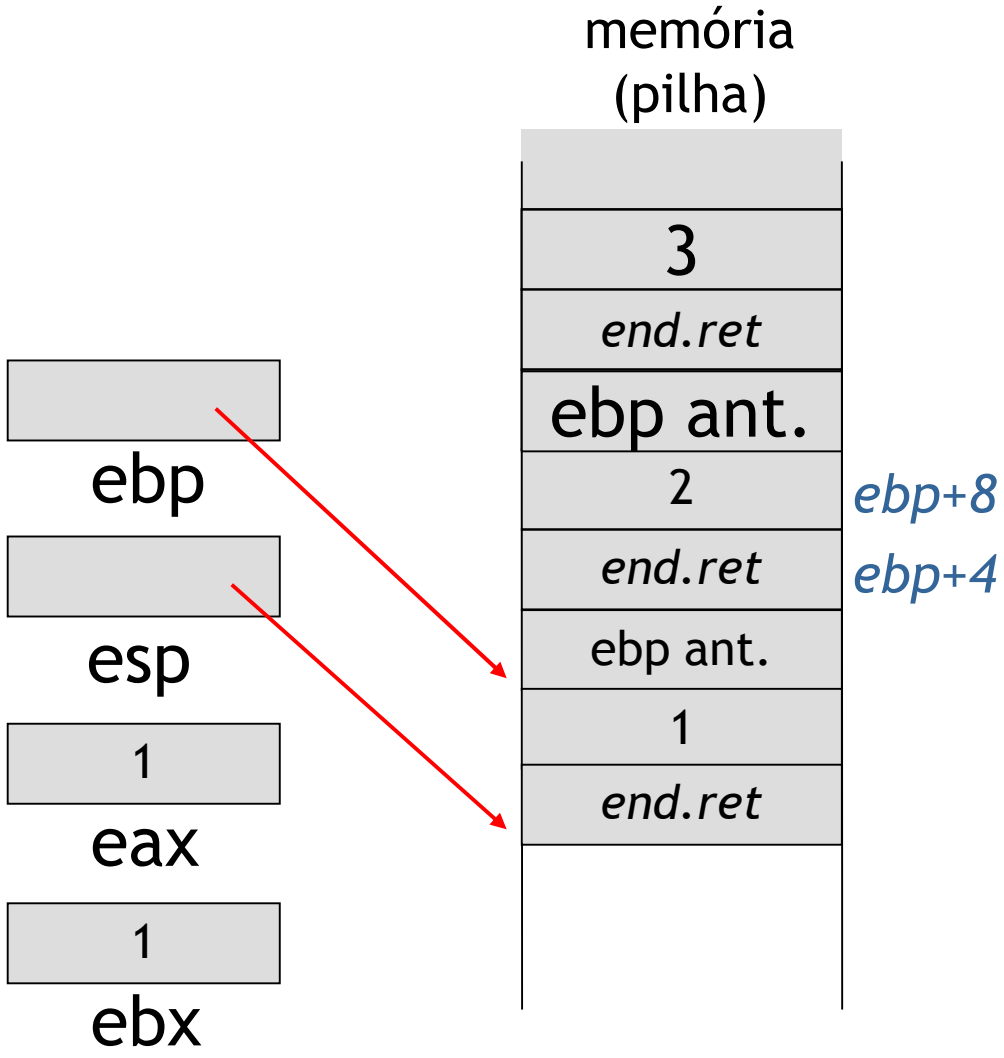
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



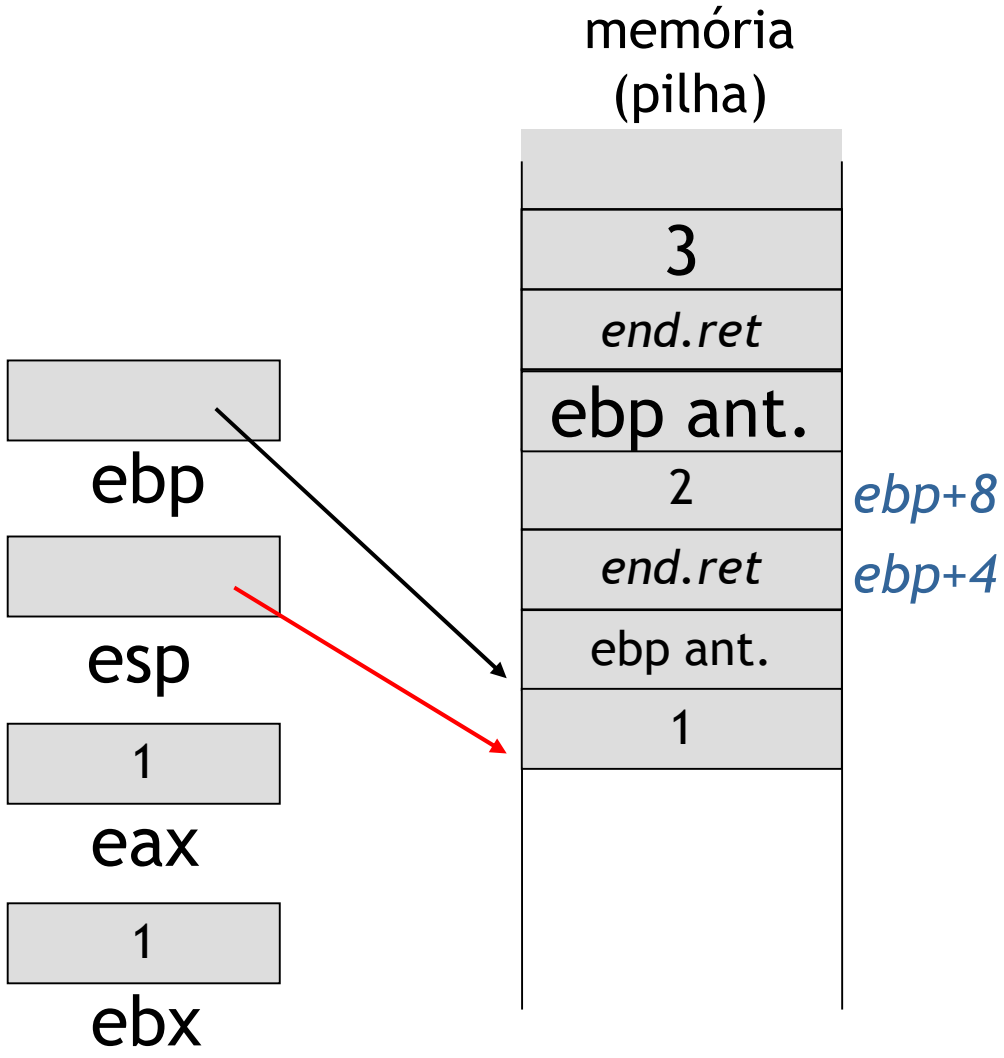
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



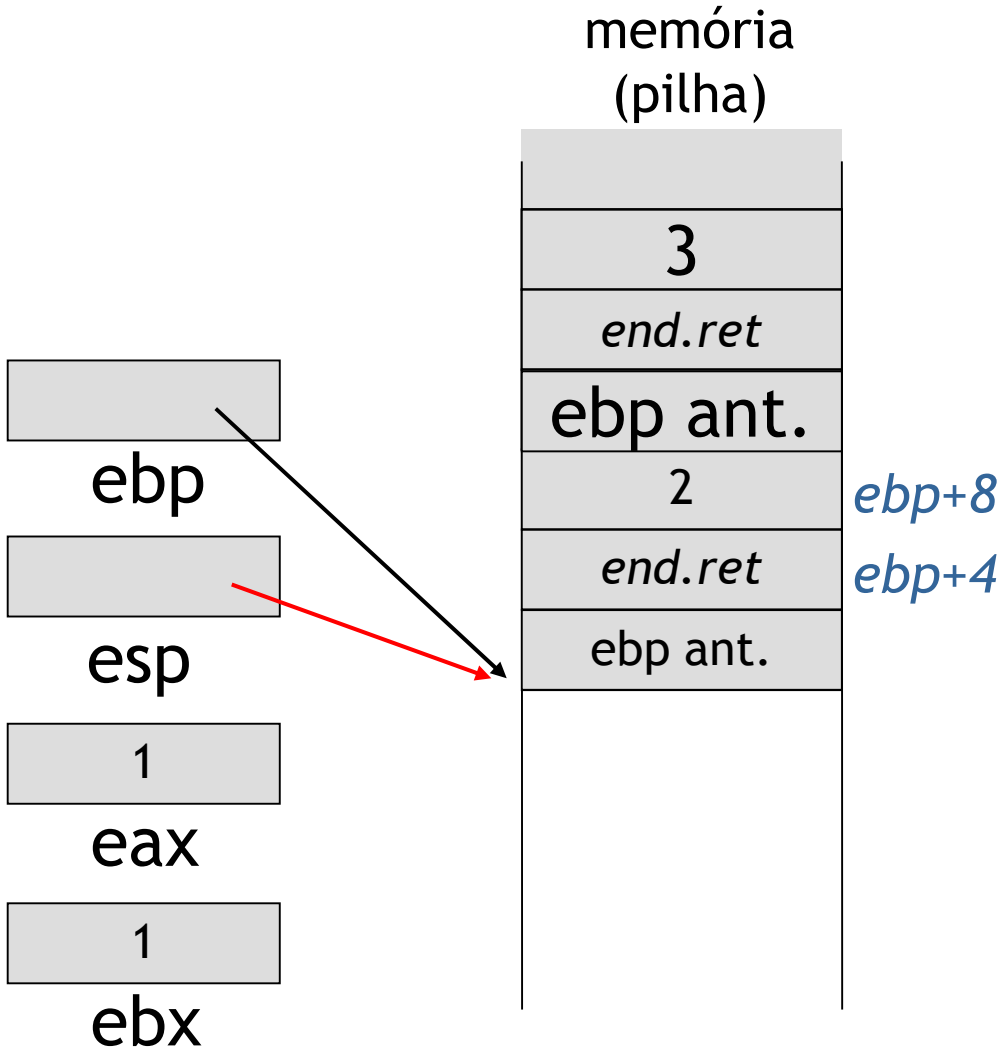
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



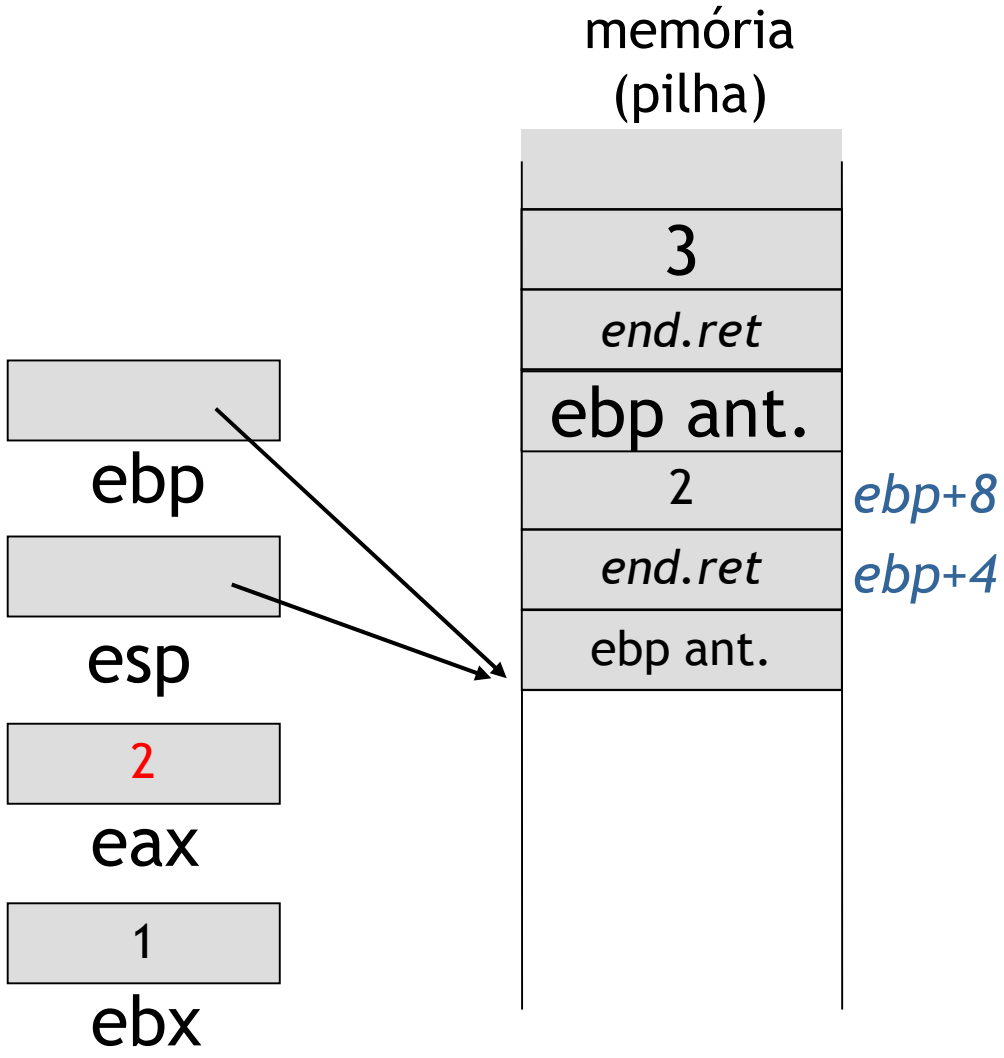
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



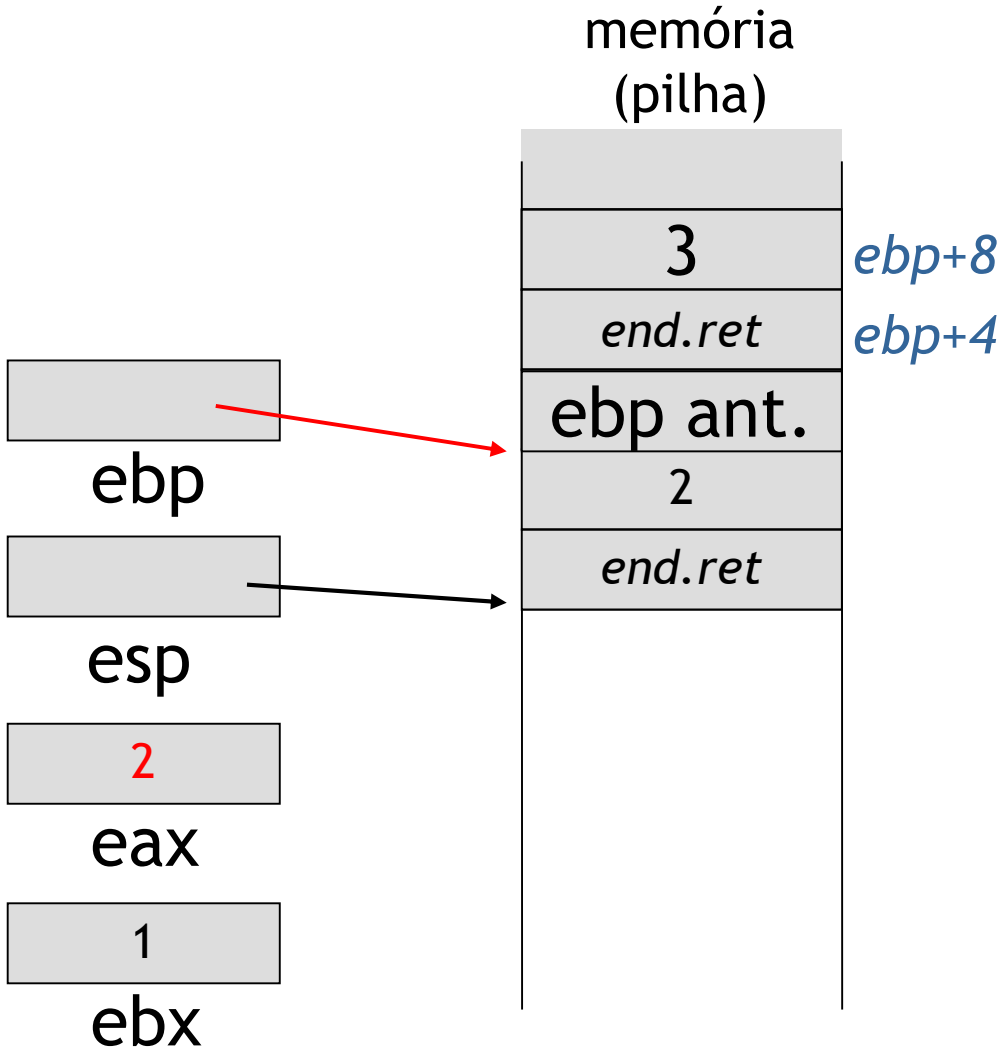
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



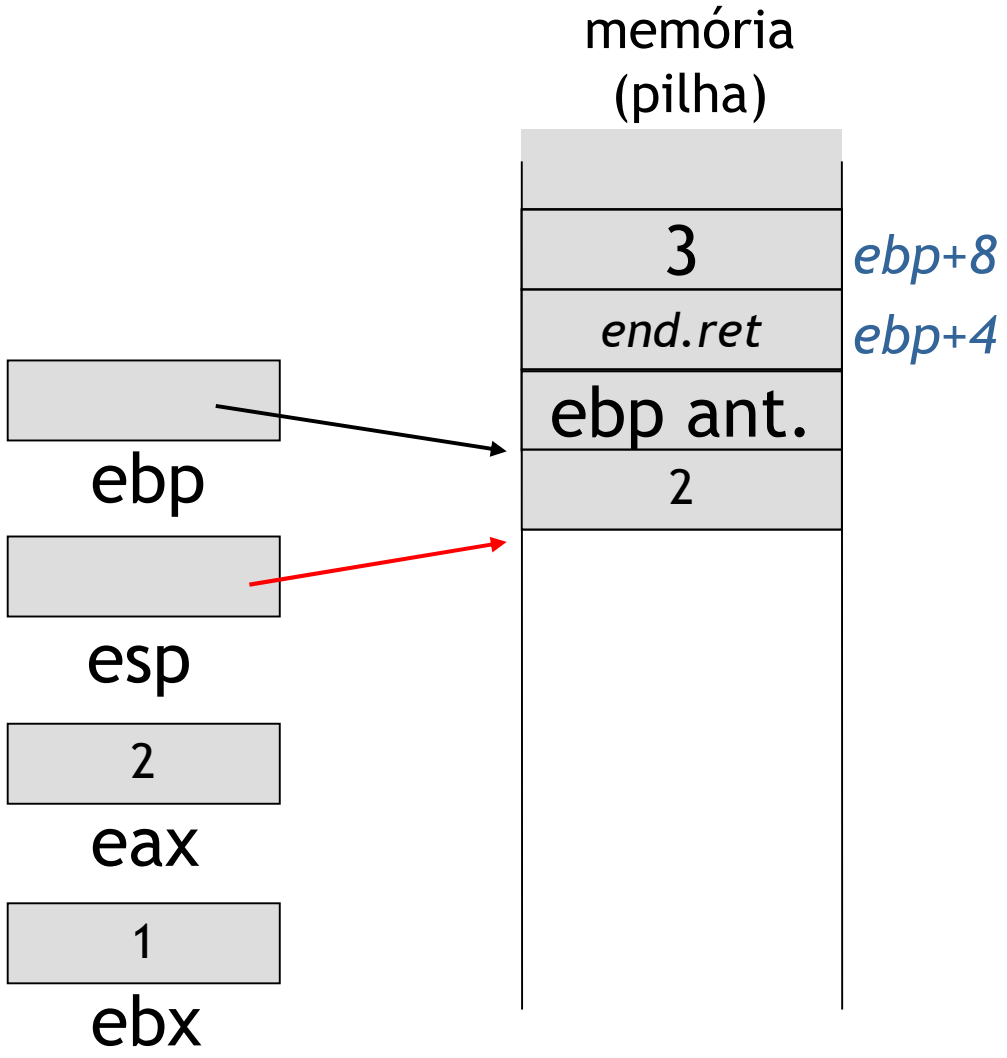
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



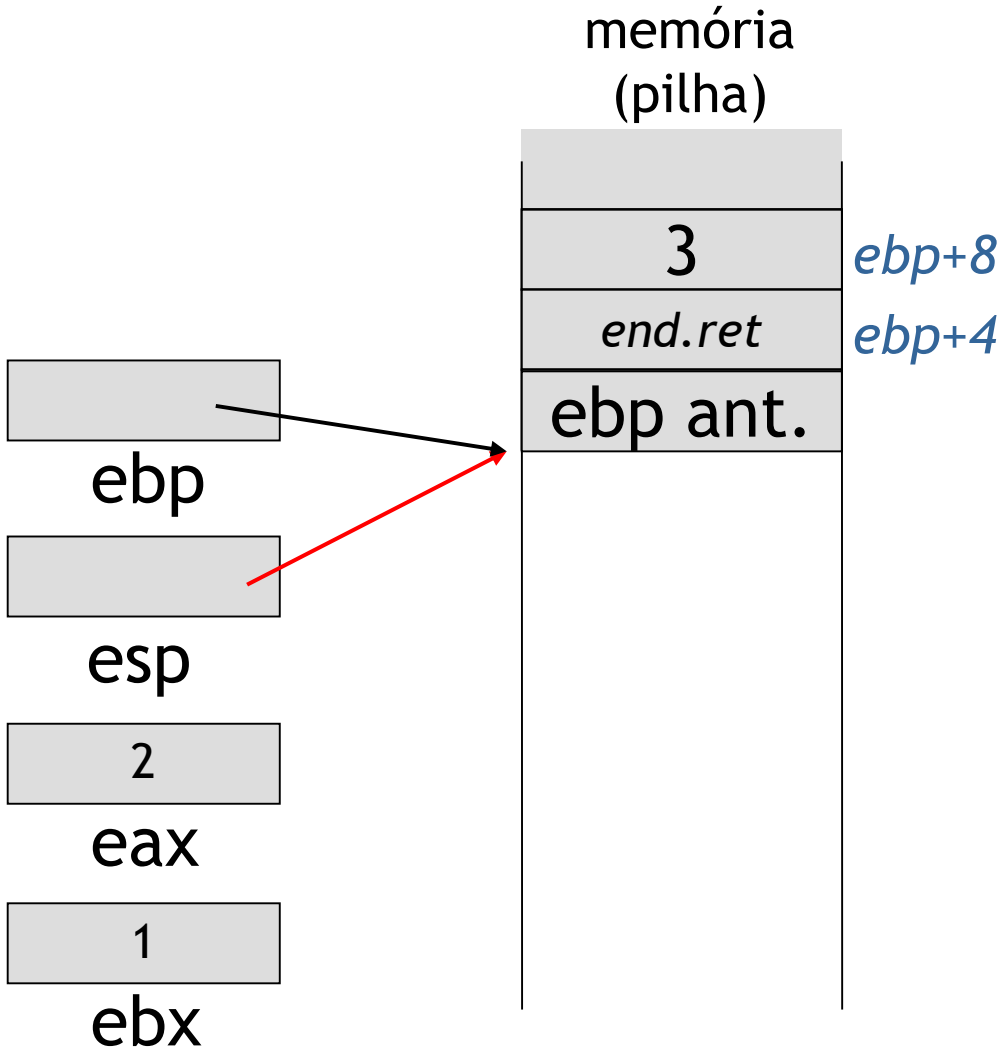
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



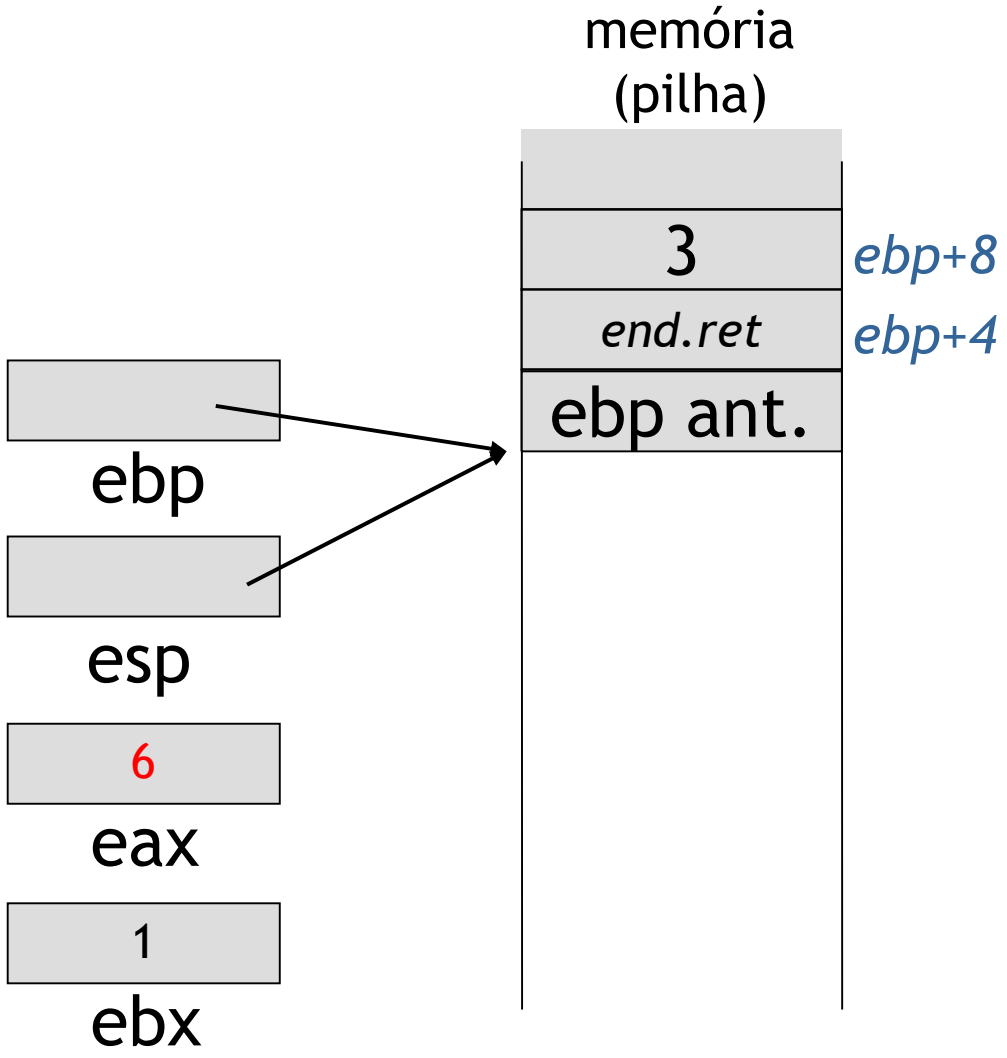
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



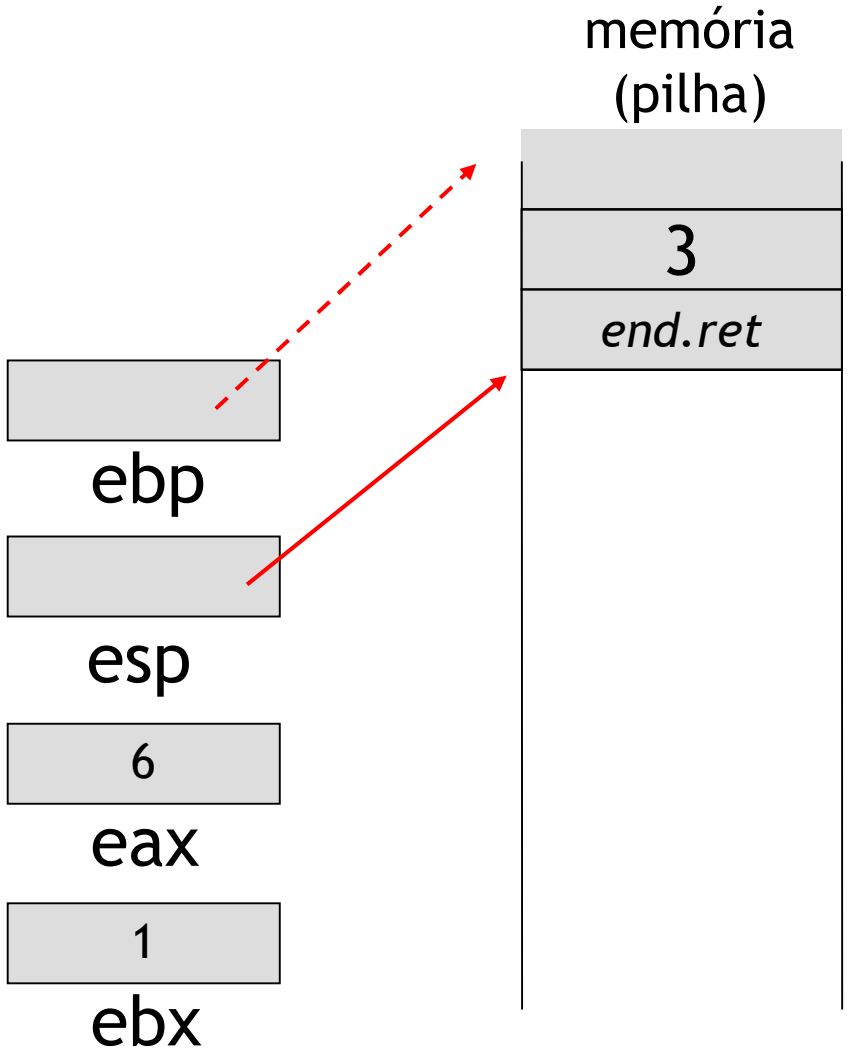
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



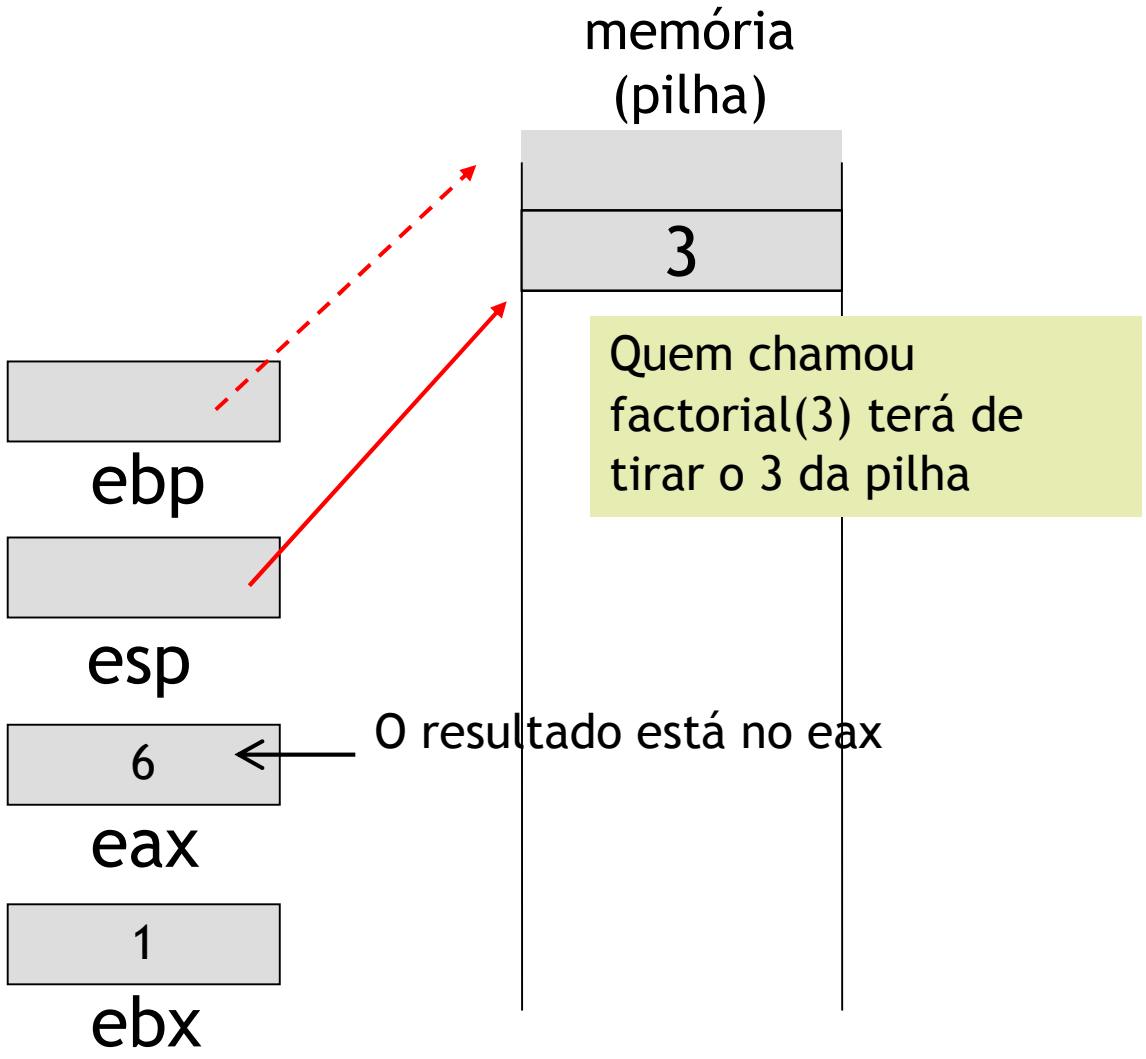
Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



Exemplo: factorial recursivo

```
factorial:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    cmp ebx, 1
    jle one
    dec ebx
    push ebx
    call factorial
    add esp, 4
    mul dword [ebp+8]
    jmp endfact
one:
    mov eax, 1
endfact:
    pop ebp
    ret
```



Um Exemplo Para estudar em Casa

- A função de Fibonnacci

$$fib(n) = \begin{cases} fib(n-1) + fib(n-2) & \text{se } n>1 \\ 1 & \text{caso contrário} \end{cases}$$

  - ✓ Temos uma função recursiva de N ⟶ N
  - ✓ É preciso
    - ✓ Verificar se n>1
    - ✓ Se sim o resultado é 1
    - ✓ Se não:
      - ✓ Calcular fib(n-1)
      - ✓ Calcular fib(n-2)
      - ✓ Somar os dois e colocar no resultado

```
fib:
```

```
fib:        push ebp
            mov ebp, esp
```

```
endfib:     pop ebp
            ret
```

```
fib:        push ebp
            mov ebp, esp
            mov ebx, [ebp+8]
```

Ler o argumento: n

```
endfib:     pop ebp
            ret
```

```
fib:        push ebp
            mov ebp, esp
            mov ebx, [ebp+8]
            cmp ebx, 1
            jle oneOrLess
```

Verificar se n<=1

```
oneOrLess:  mov eax, 1
endfib:     pop ebp
            ret
```

```
fib:        push ebp
            mov ebp, esp
            mov ebx, [ebp+8]
            cmp ebx, 1
            jle oneOrLess
            dec ebx
            push ebx
            call fib
```

Calcular fib(n-1)

```
oneOrLess:  mov eax, 1
endfib:     pop ebp
            ret
```

```
fib:        push ebp
            mov ebp, esp
            mov ebx, [ebp+8]
            cmp ebx, 1
            jle oneOrLess
            dec ebx
            push ebx
            call fib
            add esp, 4

oneOrLess:  mov eax, 1
endfib:     pop ebp
            ret
```

Repor a pilha

```
fib:        push ebp
            mov ebp, esp
            mov ebx, [ebp+8]
            cmp ebx, 1
            jle oneOrLess
            dec ebx
            push ebx
            call fib
            add esp, 4
            push eax

oneOrLess:  mov eax, 1
endfib:     pop ebp
            ret
```

Guardar o resultado

```
fib:        push ebp
            mov ebp, esp
            mov ebx, [ebp+8]
            cmp ebx, 1
            jle oneOrLess
            dec ebx
            push ebx
            call fib
            add esp, 4
            push eax
            mov ebx, [ebp+8]
            sub ebx, 2
            push ebx
            call fib


oneOrLess:  mov eax, 1
endfib:     pop ebp
            ret
```

Calcular fib(n-2)

```
fib:        push ebp
            mov ebp, esp
            mov ebx, [ebp+8]
            cmp ebx, 1
            jle oneOrLess
            dec ebx
            push ebx
            call fib
            add esp, 4
            push eax
            mov ebx, [ebp+8]
            sub ebx, 2
            push ebx
            call fib
            add esp, 4

oneOrLess:  mov eax, 1
endfib:     pop ebp
            ret
```

Repor a pilha

```
fib:        push ebp
            mov ebp, esp
            mov ebx, [ebp+8]
            cmp ebx, 1
            jle oneOrLess
            dec ebx
            push ebx
            call fib
            add esp, 4
            push eax
            mov ebx, [ebp+8]
            sub ebx, 2
            push ebx
            call fib
            add esp, 4
            pop ebx


oneOrLess:  mov eax, 1
endfib:     pop ebp
            ret
```

Retirar da pilha o resultado guardado anteriormente. De notar que se usou o **pop** e não **add esp,4**. Isto porque queremos manipular o valor retirado.

```
fib:        push ebp
            mov ebp, esp
            mov ebx, [ebp+8]
            cmp ebx, 1
            jle oneOrLess
            dec ebx
            push ebx
            call fib
            add esp, 4
            push eax
            mov ebx, [ebp+8]
            sub ebx, 2
            push ebx
            call fib
            add esp, 4
            pop ebx
            add eax, ebx

oneOrLess:  mov eax, 1
endfib:     pop ebp
            ret
```

Calcular o resultado, à custa do valor retirado da pilha (ebx)

```
fib:        push ebp
            mov ebp, esp
            mov ebx, [ebp+8]
            cmp ebx, 1
            jle oneOrLess
            dec ebx
            push ebx
            call fib
            add esp, 4
            push eax
            mov ebx, [ebp+8]
            sub ebx, 2
            push ebx
            call fib
            add esp, 4
            pop ebx
            add eax, ebx
            jmp endfib
oneOrLess:  mov eax, 1
endfib:     pop ebp
            ret
```

Terminar

Programas Assembly com Vários Módulos

- Como em Java ou C, um programa assembly pode ser dividido em vários módulos
- No assembly NASM um módulo é um ficheiros fonte

- Vantagens:
  - ✓ Só é necessário assemblar os módulos cujo código foi modificado
    - ✓ Note que é sempre preciso *linkar* todos os módulos para obter o executável
  - ✓ Modularidade e facilidade de desenvolvimento
    - ✓ Trabalho em equipa
    - ✓ Ficheiros mais pequenos são mais fáceis de modificar
    - ✓ …

Programas Assembly com Vários Módulos

- O NASM oferece duas directivas para desenvolver aplicações com vários módulos
  - ✓ **`global`** → indica que uma etiqueta é pública, visível para outros módulos
    - ✓ Todo o tipo de etiquetas pode ser tornado público
      - ✓ Variáveis
      - ✓ Subrotinas
      - ✓ Etiquetas para saltos
    - ✓ No TASM/MASM a directiva tem o nome **`public`** sendo, portanto, o nome que aparece no livro
  - ✓ **`extern`** → indica que a etiqueta especificada não está definida no módulo corrente

Programas Assembly com Vários Módulos
Exemplo 1

Ficheiro main.asm

```
SYS_EXIT equ 1
LINUX_SYSCALL equ 0x80
global _start
extern C

section .data
  A:    dd 24
  B:    dd 1

section .text
_start: mov eax, [A]
        add eax, [B]
        mov [C], eax

        mov eax, SYS_EXIT
        mov ebx, 0
        int LINUX_SYSCALL
```

Ficheiro c.asm

```
global C

section .bss
C:      resd 1
```

Programas Assembly com Vários Módulos
Exemplo 2

Ficheiro main.asm

```
SYS_EXIT equ 1
LINUX_SYSCALL equ 0x80
global _start
extern C, somaAB

section .data
  A:    dd 24
  B:    dd 1
section .text
_start: push dword [A]
        push dword [B]
        call somaAB
        add esp, 8
        mov [C], eax
        mov eax, SYS_EXIT
        mov ebx, 0
        int LINUX_SYSCALL
```

Ficheiro c.asm

```
global C

section .bss
C:      resd 1
```

Ficheiro soma.asm

```
global somaAB

section .text
somaAB:
    push ebp
    mov ebp, esp
    mov eax, [ebp+8]
    add eax, [ebp+12]
    pop ebp
    ret
```

Erros de Compilação e Ligação (*Linkagem*)

- Erro de compilação/assemblagem
  - ✓ Ocorre quando se compila (linguagem de alto-nível) ou se assembla (linguagem assembly) o código fonte para código máquina
  - ✓ Exemplos:
    - ✓ Instrução não existe
    - ✓ Variável não existe
    - ✓ Em linguagens de alto-nível
      - ✓ A função/método é chamada com argumentos de tipo errado ou com o número de argumentos errado
- Erro de ligação (*linkagem*)
  - ✓ Ocorre aquando do processo de ligação
  - ✓ Um símbolo que um dado módulo precisa não é fornecido por nenhum dos outros módulos

Programas Assembly com Vários Módulos
Erro de Compilação/Assemblagem

Ficheiro main.asm

```
SYS_EXIT equ 1
LINUX_SYSCALL equ 0x80
global _start
extern C, somaAB

section .data
  A:    dd 24
  B:    dd 1
section .text
_start: push dword [AA]
        push dword [B]
        call somaAB
        add esp,8
        mov [C], eax
        move eax, SYS_EXIT
        mov ebx, 0
        int LINUX_SYSCALL
```

Erro de compilação/ assemblagem
A etiqueta AA não está declarada

Erro de compilação/ assemblagem
A instrução **move** não existe

Ficheiro c.asm

```
global C

section .bss
C:      resd 1
```

Ficheiro soma.asm

```
global somaAB

section .text
somaAB:
    push ebp
    mov ebp, esp
    mov eax, [ebp+8]
    add eax, [ebp+12]
    pop ebp
    ret
```

Programas Assembly com Vários Módulos
Erro de Ligação (*Linkagem*)

Ficheiro main.asm

```
SYS_EXIT equ 1
LINUX_SYSCALL equ 0x80
global _start
extern D, somaAB

section .data
  A:    dd 24
  B:    dd 1
section .text
_start: push dword [A]
        push dword [B]
        call somaAB
        add esp, 8
        mov [D], eax
        mov eax, SYS_EXIT
        mov ebx, 0
        int LINUX_SYSCALL
```

Erro de linkagem.
A etiqueta D não existe em nenhum dos módulos

Ficheiro c.asm

```
global C

section .bss
C:      resd 1
```

Ficheiro soma.asm

```
global somaAB

section .text
somaAB:
    push ebp
    mov ebp, esp
    mov eax, [ebp+8]
    add eax, [ebp+12]
    pop ebp
    ret
```