Arquitetura de Computadores

MIEI – 2021/22

DI-FCT/UNL

IA-32: Modos de Endereçamento e Subrotinas

Sumário

- Programação em Assembly:
  - Modos de endereçamento (continuação)
  - A pilha
  - Subrotinas
- Ligação C – Assembly

Bibliografia:

Bryant, O'Hallaron Computer Systems: A Programmer's Perspective, 2nd ou 3rd Ed, secções 3.7 a 3.9 e 7.1

João M. Fernandes, Essentials of computing systems, Coleção Educação | Ciências, Engenharia e Tecnologia, Cap 2. secções 4.1 e 4.2.7 a 4.2.8 (https://doi.org/10.21814/uminho.ed.33)

Modos de indicar um operando

- Diferentes modos de indicar os operandos:
  - **imediato** – o operando está na própria instrução (após o fetch já se encontra no CPU)
  - **registo** – o valor encontra-se num registo do CPU; na instrução encontra-se o código (endereço) do registo
  - **endereçamento de memória** – o operando está em memória …

    Visto: direto e por registo

    Mas há outros modos de endereçar operandos em memória …

Modos de endereçar a memória

- Endereçamento **direto** *(visto antes)*
  - o **endereço efetivo** está na instrução.

  Exemplos:

```
mov (100), %eax
mov var1, %eax
```

- **Indireto por registo** *(visto antes)*
  - um registo contém o endereço efetivo onde o operando está na memória.

  Exemplo: `mov (%ebx), %eax`

Modos de endereçar a memória

- Endereçamento **indireto** sempre que:
  - O CPU calcula o endereço efetivo durante a execução da instrução com base em informação codificada na instrução
  - Variantes: envolvendo mais de um registo, deslocamentos e fatores de escala

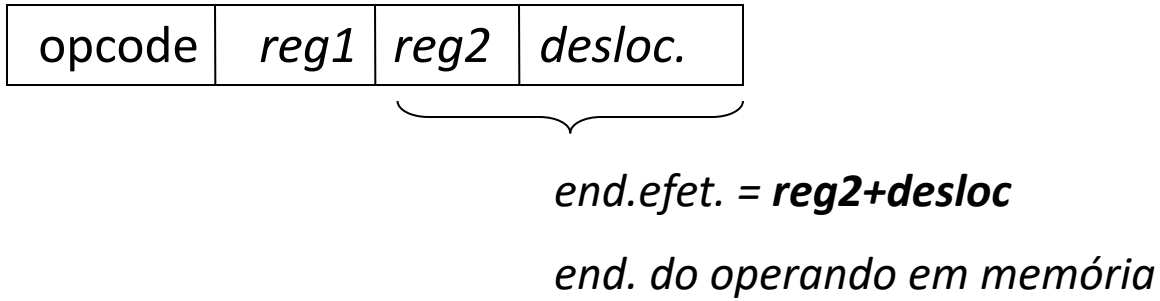
- *Endereçamento **indireto por memória:***
  - *endereço efetivo encontra-se na memória, cujo endereço é o indicado na instrução*

  Não suportado na arquitetura IA-32!

Endereçamento baseado

- Indireto com **base e deslocamento**
  - um registo contém um endereço (base) e na instrução é indicado um deslocamento
  - Ou, a base é dada como constante e o registo contém o deslocamento (índice)

    Exemplo: **mov** 8(%ebx), %eax

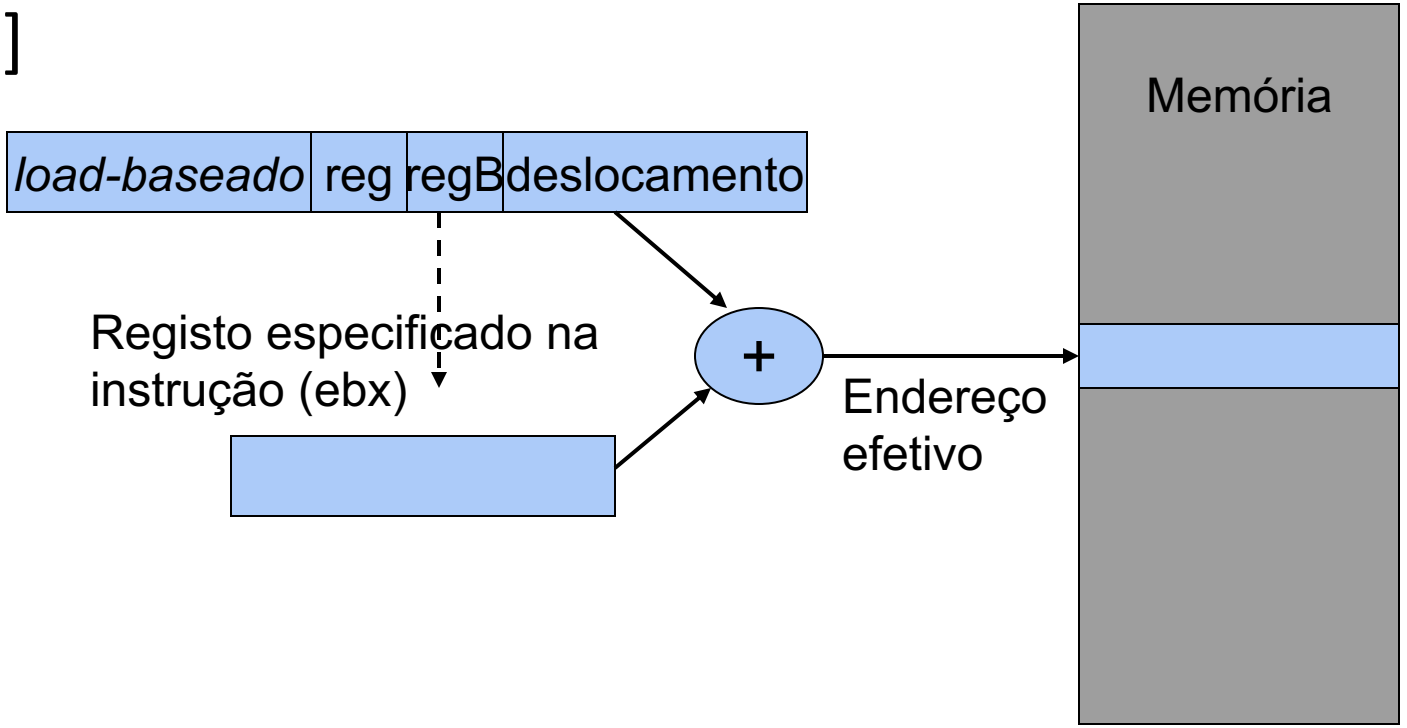


Endereçamento baseado ou indexado

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

  Exemplo: **`mov`** `8(%ebx), %eax`

  EAX ← Mem[EBX+8]



Endereçamento indireto de memória

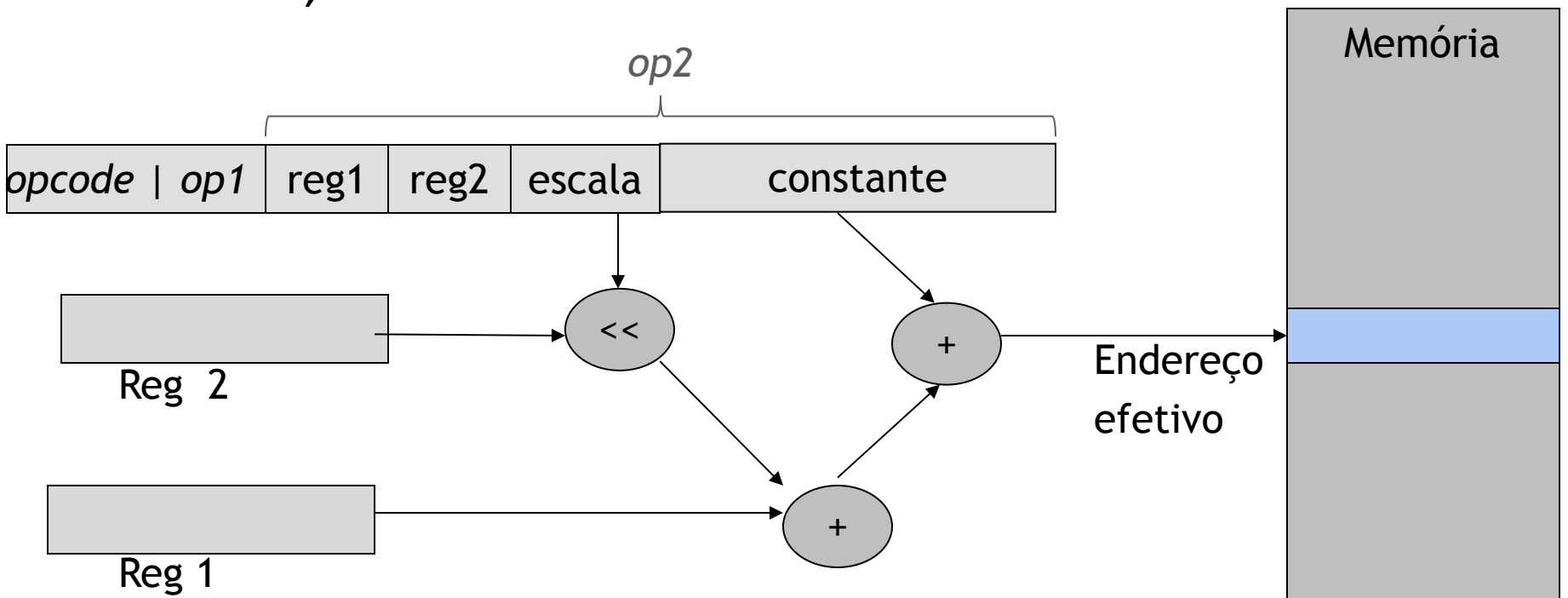
- Forma geral em *assembly* para IA-32:
  - **D(Rb, Ri, S)** End. Efetivo = D + Rb + Ri*S
    - D: Constante (deslocamento)
    - Rb: Registo Base: Qualquer dos 8 registos gerais
    - Ri: Registo índice: Qualquer, excepto ESP
    - S: Escala: 1, 2, 4 ou 8

Exemplo: `movl $5, 100(%ebx, %ecx, 2)`

End Efetivo = 100 + EBX + 2 x ECX

Endereçamento indireto de memória

- O endereço efetivo é obtido somando o conteúdo do registo1 (registo base) ao resultado de aplicar ao registo 2 (registo índice) um factor de escala, e somando a constante



No IA32, o factor de escala pode ser 1, 2, 4 ou 8 (2^0, 2^1, 2^2 ou 2^3)

Endereçamento de memória

- Alguns casos particulares (c/exemplos):

D ou (D)	**`var`** ou **`(var)`**	- direto
(Rb)	**`(%eax)`**	- indireto p/registo
D(Rb)	**`vet(%ebx)`**	- ind baseado
(Rb, Ri, S)	**`(%ebx, %ecx, 2)`**	- ind baseado e indexado
D(, Ri, S)	**`vet( ,%ecx, 4)`**	- ind indexado

Exemplos de cálculo do endereço efectivo

edx	**0xf000**
ecx	**0x100**

End. assembly	Cálculo	Endereço efetivo
`0x8(%edx)`	`0xf000 + 0x8`	?
`(%edx, %ecx)`	`0xf000 + 0x100`	?
`(%edx,%ecx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80( ,%ecx,2)`	`0x80 + 2*0x100`	`0x280`

Exemplos de cálculo do endereço efectivo

edx	0xf000
ecx	0x100

End. assembly	Cálculo	Endereço efetivo
0x8(%edx)	0xf000 + 0x8	?
(%edx, %ecx)	0xf000 + 0x100	?
(%edx,%ecx,4)	0xf000 + 4*0x100	?
0x80(,%ecx,2)	0x80 + 2*0x100	?

Endereçamento baseado e indexado

- Conveniente para endereçar vetores

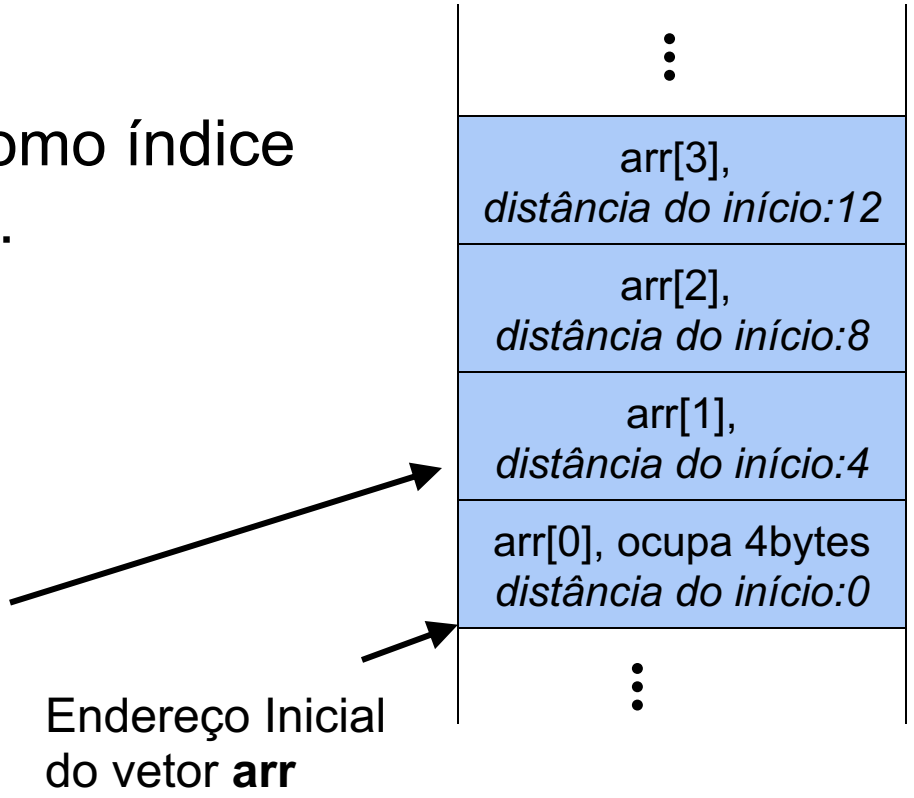
```
int arr[100];
```

Sabendo o endereço de `arr`,

Um registo pode funcionar como índice tomando os valores 0, 1, 2 …

Exemplo:

```
mov $arr, %eax
mov $1, %ecx
mov (%eax, %ecx, 4), %ebx
```



Endereçamento indexado

- Conveniente para endereçar vetores

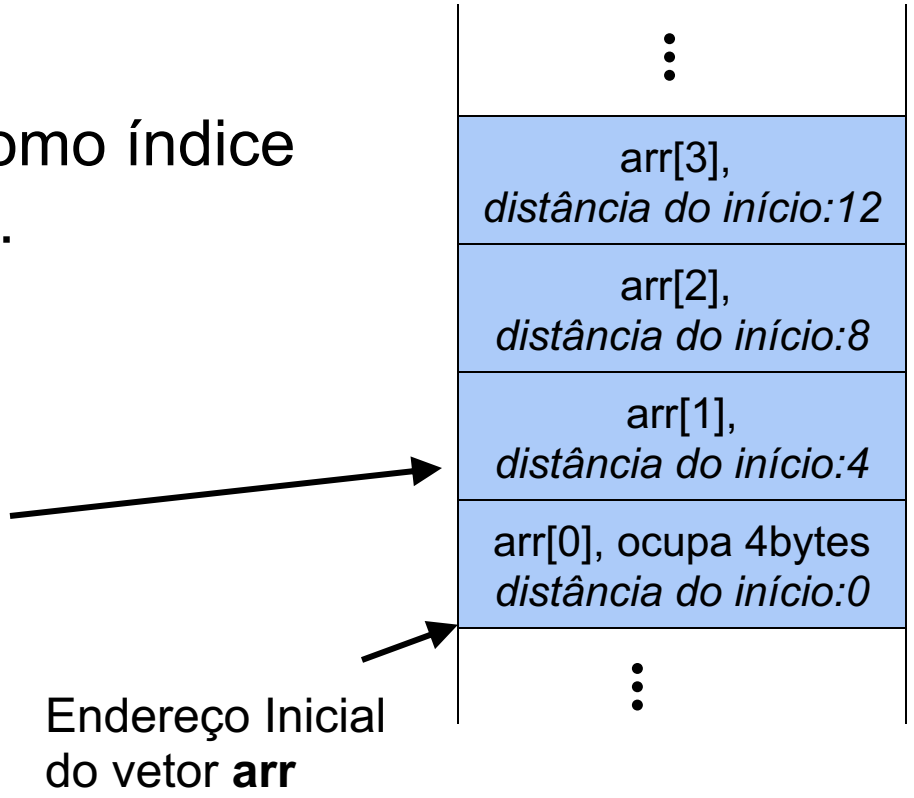
```
int arr[100];
```

Sabendo o endereço de `arr`,

Um registo pode funcionar como índice tomando os valores 0, 1, 2 …

Exemplo:

```
mov $1, %ecx
mov arr( , %ecx, 4), %ebx
```



Incrementar todos os ints de vet

```
.data
vet:    .int 0, 3, 5, 10, 2

.text
        . . .
        mov $5, %eax
        mov $vet, %ebx
ciclo:  incl (%ebx)
        add $4, %ebx
        dec %eax
        jnz ciclo
        . . .
```

```
.data
vet: .  int 0, 3, 5, 10, 2

.text
        . . .
        mov $5, %eax
        mov $0, %ebx
ciclo:  incl vet(%ebx)
        add $4, %ebx
        dec %eax
        jnz ciclo
        . . .
```

Incrementar todos os ints de vet

```
.data
vet:    .int 0, 3, 5, 10, 2

.text
        . . .
        mov $5, %eax
        mov $0, %ebx
ciclo:  incl vet(%ebx)
        add $4, %ebx
        dec %eax
        jnz ciclo
        . . .
```

```
.data
vet:    .int 0, 3, 5, 10, 2

.text
        . . .
        mov $5, %ecx
        mov $0, %eax
ciclo:  incl vet( , %eax, 4)
        inc %eax
        loop ciclo
        . . .
```

Incrementar todos os ints de vet

```
.data
vet:    .int 0, 3, 5, 10, 2

.text
            . . .
            mov $4, %ecx
ciclo:      incl vet( , %ecx, 4)
            loop ciclo
            incl (vet)
            . . .
```

Reutilização de Código

- Repetir a operação para outros vetores?
- Como reexecutar código?
- Como implementar funções, etc...?

Subrotinas

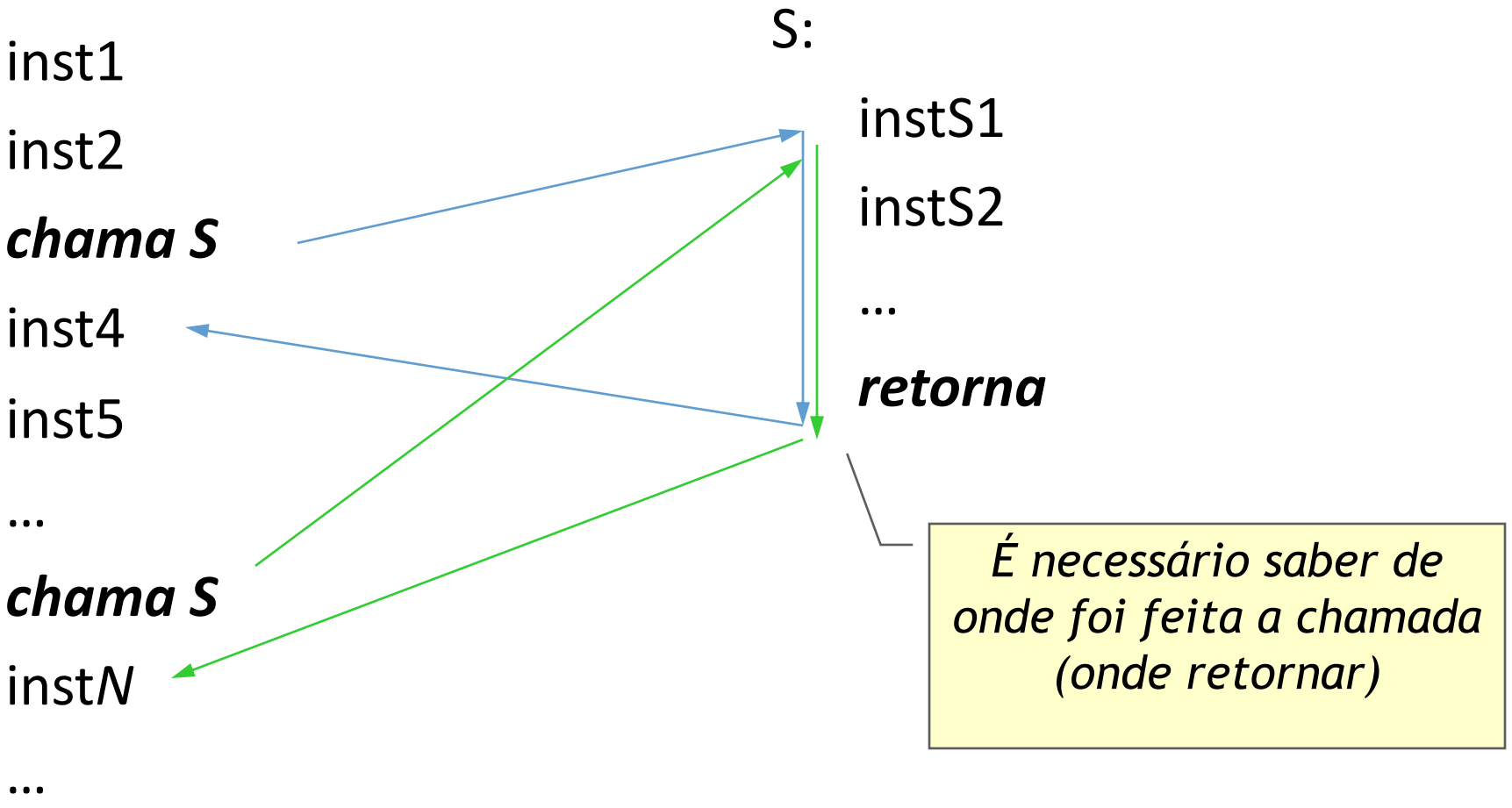
- **Subrotina**: uma sequência de instruções, que pode ser chamada múltiplas vezes, a partir de diferentes pontos de um programa

- O suporte, a nível da arquitetura do computador, sobre a qual assentam as funções e métodos de linguagens de "alto-nível".

Vantagens

- Permite implementar procedimentos e funções
  - Redução do tamanho do programa:
    - Evita a repetição de sequências muito utilizadas
  - Estruturação do programa:
    - Maior clareza e modularidade
    - Desenvolvimento e teste incrementais
    - Bibliotecas de código reutilizável
    - etc

"*Jumps*" para chamada e retorno



chamar: CALL e retornar: RET

- ***call*** *S*
  - Semelhante a **`jmp`** *S*, mas...
  - Não basta saltar! Há que saber regressar:

    Na chamada, guarda o EIP e só depois salta

    Mem? ← EIP

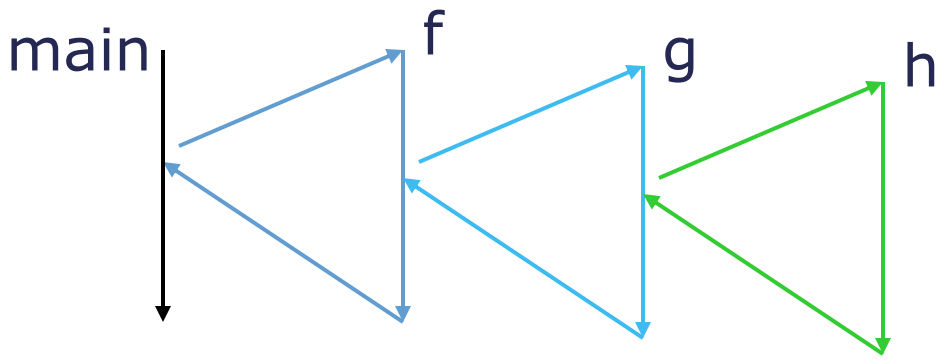
    EIP← S (tal como jmp S)
- ***ret***
  - salta para o endereço guardado

    No fim da subrotina, repõe o EIP, saltando para o endereço que “call” guardou

    EIP← *Mem?*

Salvar o endereço de retorno

- Como salvaguardar o endereço de retorno:
  - Numa célula pré-definida de memória?
  - Num registo dedicado, do CPU?
  - Numa estrutura de dados especial em memória?
- Podem existir chamadas em cascaca e/ou recursivas
  - exemplo: main() → f() → g() → h()



Salvar o endereço de retorno

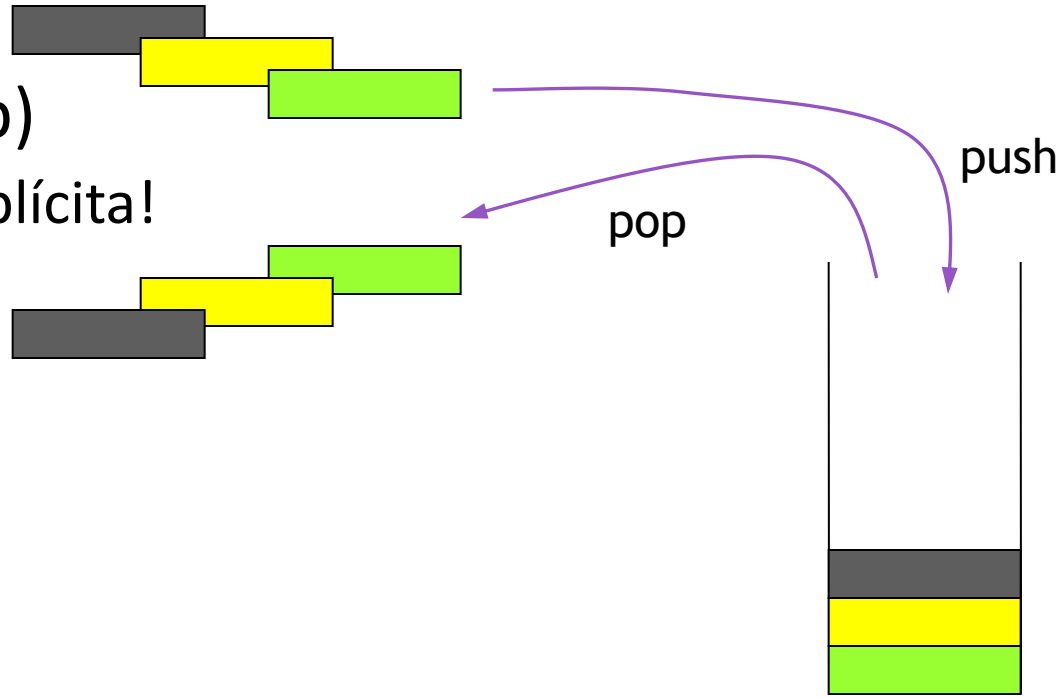
- A solução exige manter todos os endereços das chamadas activas
  - registos ou células de memória predefinidas é limitado (impraticável)

- Usa-se uma estrutura de dados em pilha
  - permite que se empilhe sempre mais um endereço, ficando no topo o último empilhado
  - retorna-se para o endereço que está no topo

Estrutura de dados: Pilha (stack)

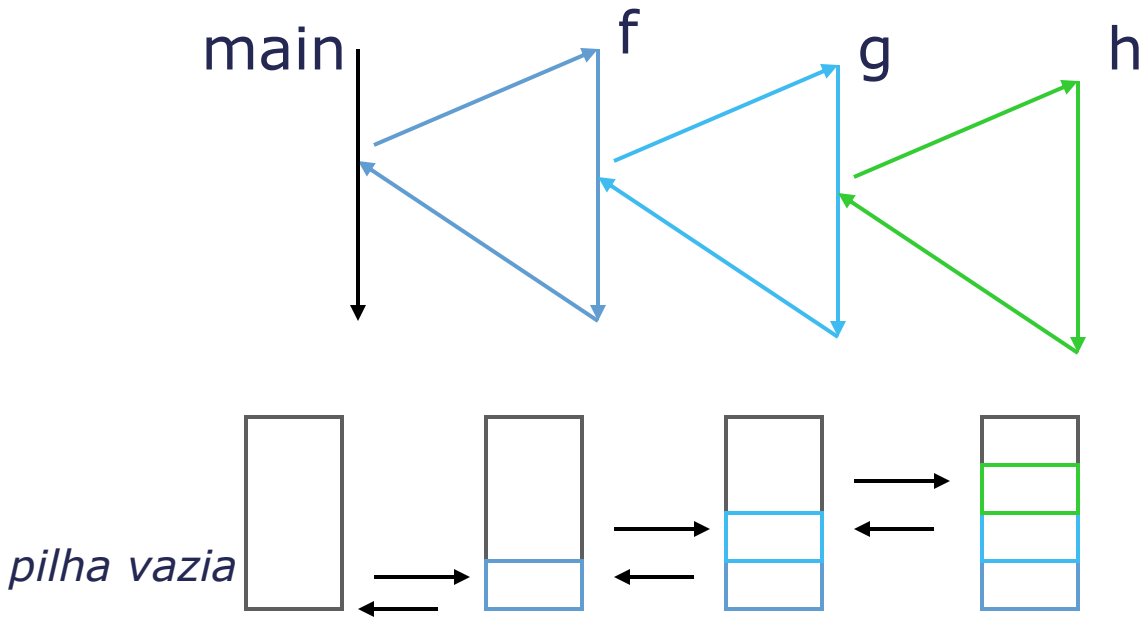
- Repositório temporário de dados seguindo o princípio LIFO (Last In – First Out)

- Duas operações: pôr (push) e tirar (pop)
  - A posição onde guardar os dados está implícita!



Salvar o endereço de retorno numa pilha

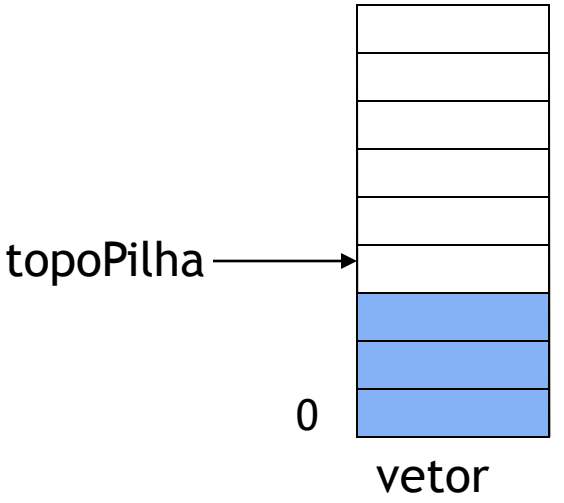
- A solução exige manter todos os endereços das chamadas activas
- Usa-se uma estrutura de dados em pilha



Implementação da Pilha

- Exemplo de uma implementação sobre um vetor (sem verificar erros) :

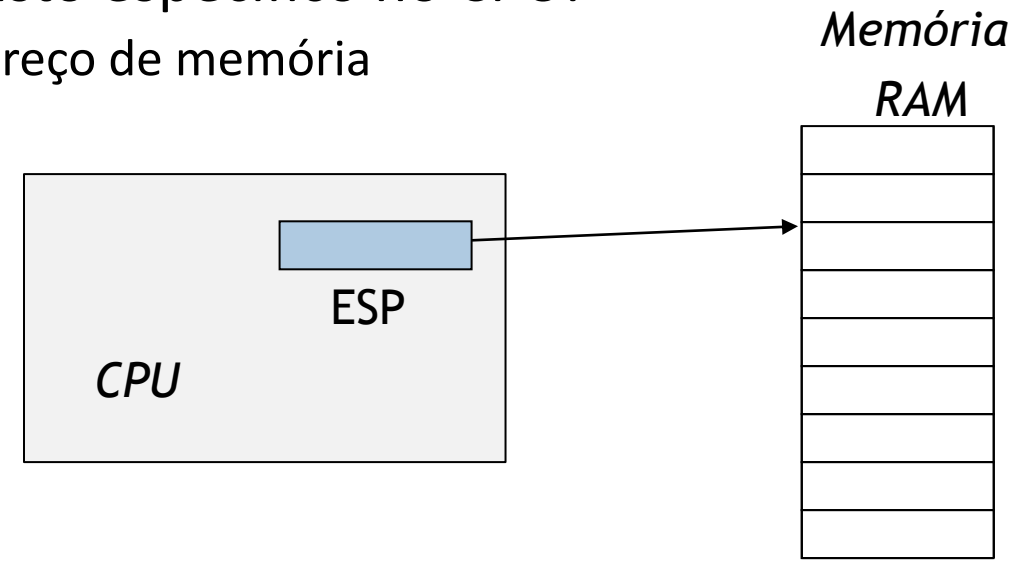
```
tipoE  vector[DimPilha];
int topoPilha = 0; // primeira posição vazia

void Por( tipoE E ){
  vector[ topoPilha ] = E;
  topoPilha = topoPilha+1;
}

tipoE Tirar (){
  topoPilha = topoPilha-1;
  return vector[ topoPilha ];
}
```



Estrutura de dados Pilha (stack)

- Esta estrutura de dados é diretamente suportada pelos CPUs!

- Na arquitectura IA-32 (Intel):
  - Pilha → sequência de bytes em memória
  - Índice do topo da pilha é um registo específico no CPU:
    - **ESP** (Stack Pointer) -- contém endereço de memória



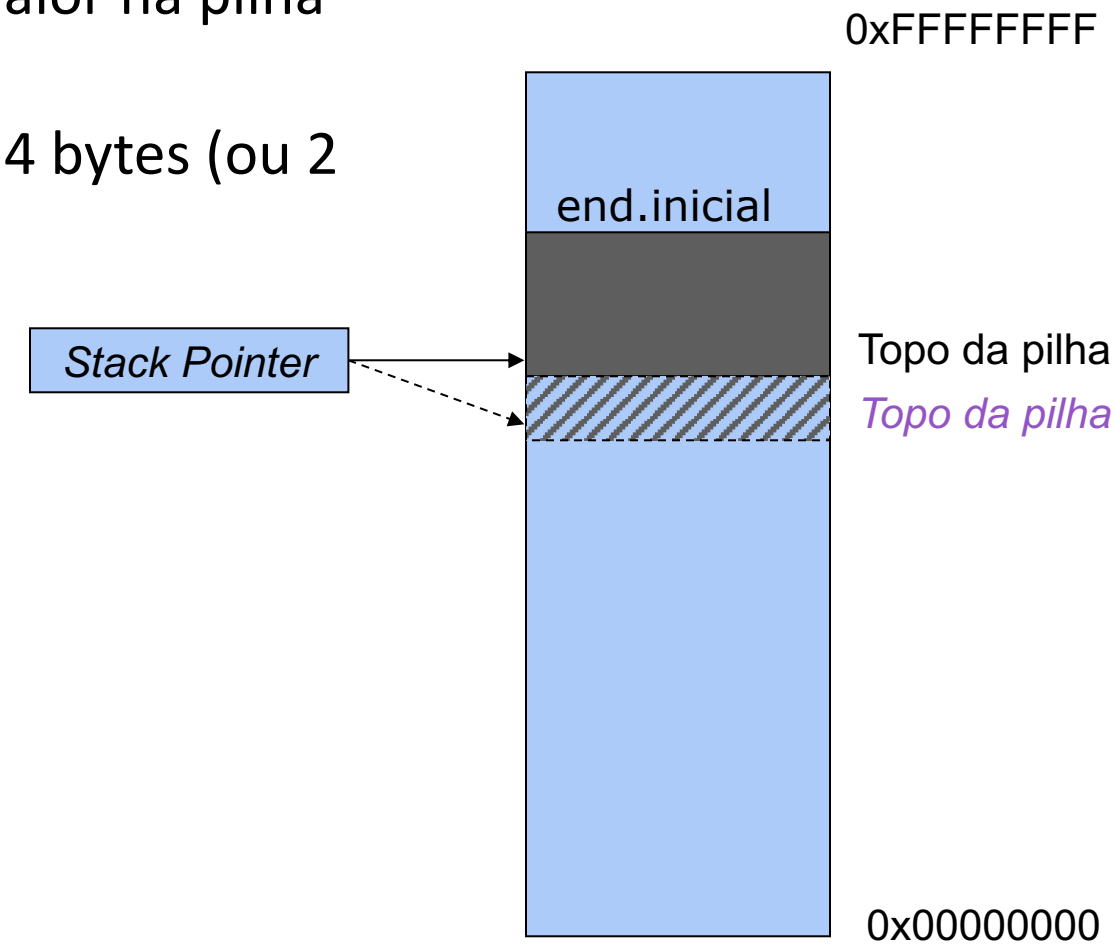
Operações sobre a pilha: Exemplo IA-32

- O ESP desce quando é inserido mais um valor na pilha
- O ESP aponta a última posição ocupada
- As operações trabalham com palavras de 4 bytes (ou 2 bytes)

- **Inicializar**: ESP ← end.inicial

Push *valor*:
  ESP ← ESP – 4 *(ou 2)*
  Mem[ESP] ← *valor*

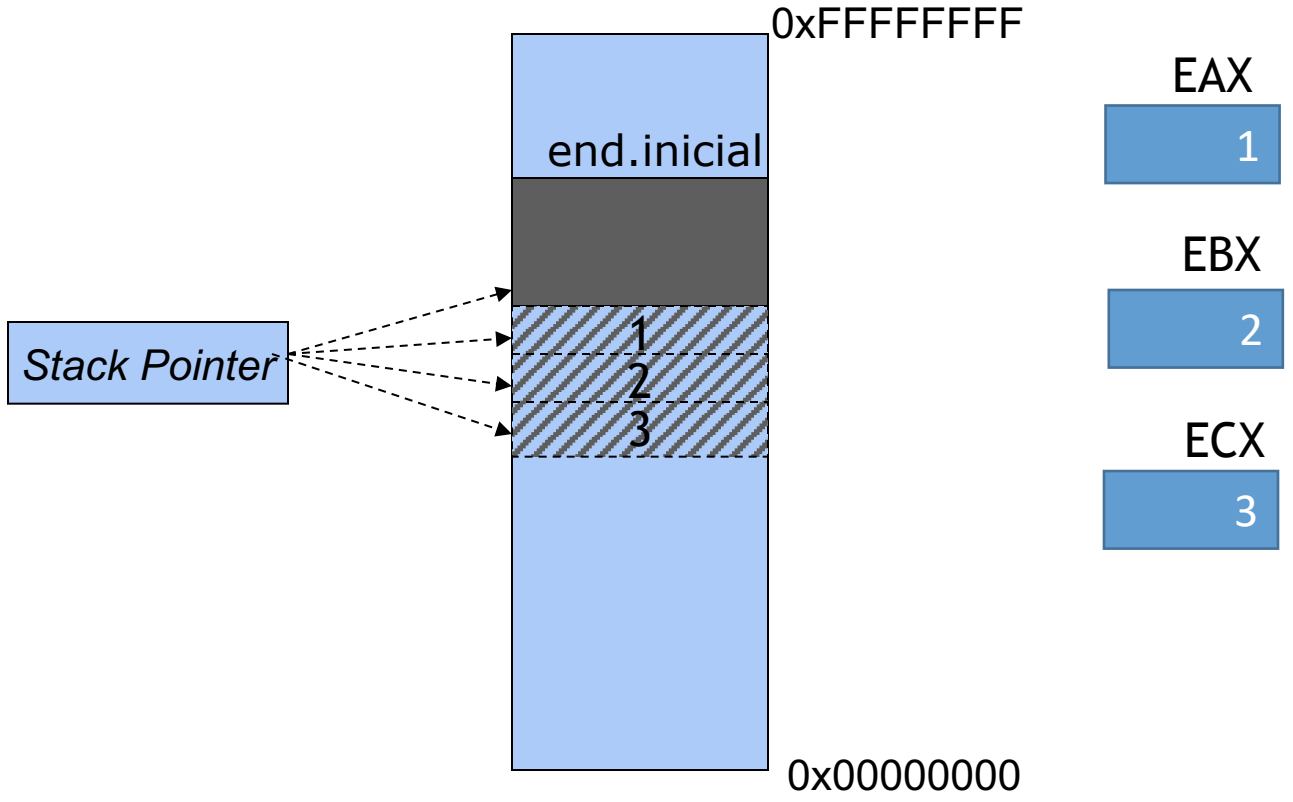
Pop *destino*:
  destino← Mem[ESP]
  ESP ← ESP + 4 *(ou 2)*



Exemplos

- Guardar (*store*) valores nos registos para memória lendo-os trocados (*load*):

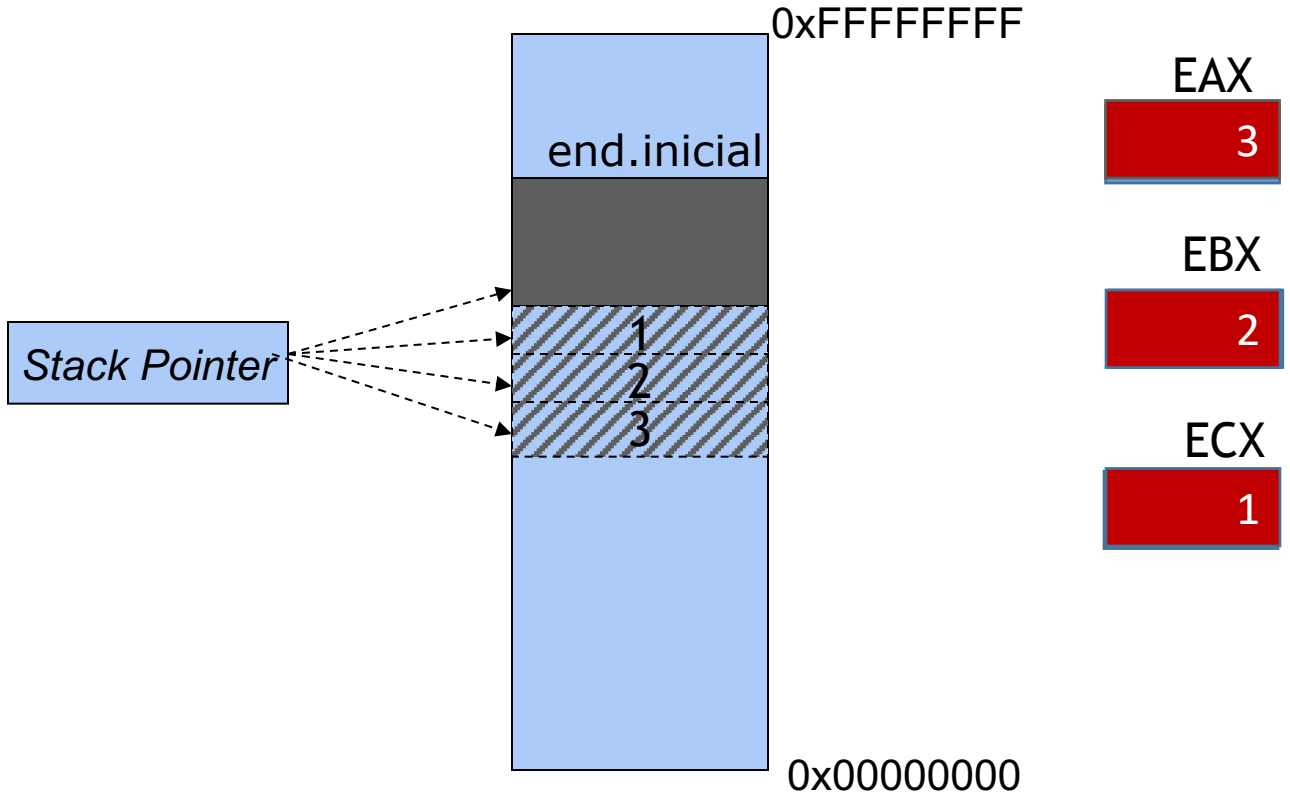
```
push %eax
push %ebx
push %ecx
```



Exemplos

- Guardar (*store*) valores nos registos para memória lendo-os trocados (*load*):

```
push %eax
push %ebx
push %ecx
pop %eax
pop %ebx
pop %ecx
```



A zona da pilha (ou stack)

- Além do código e dos dados, um programa escrito numa linguagem como o C/C++ utiliza outra zona de memória, como **pilha** (ou *stack*).
  - Esta pilha é usada para manter valores temporários, os endereços de retorno das subrotinas e outros…
  - Para além do PUSH e POP, também as instruções CALL e RET do CPU utilizam esta pilha
  - A dimensão ocupada varia ao longo da execução do programa

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

  EIP ← *S*

- **`ret`**

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

  EIP ← Mem[ESP]

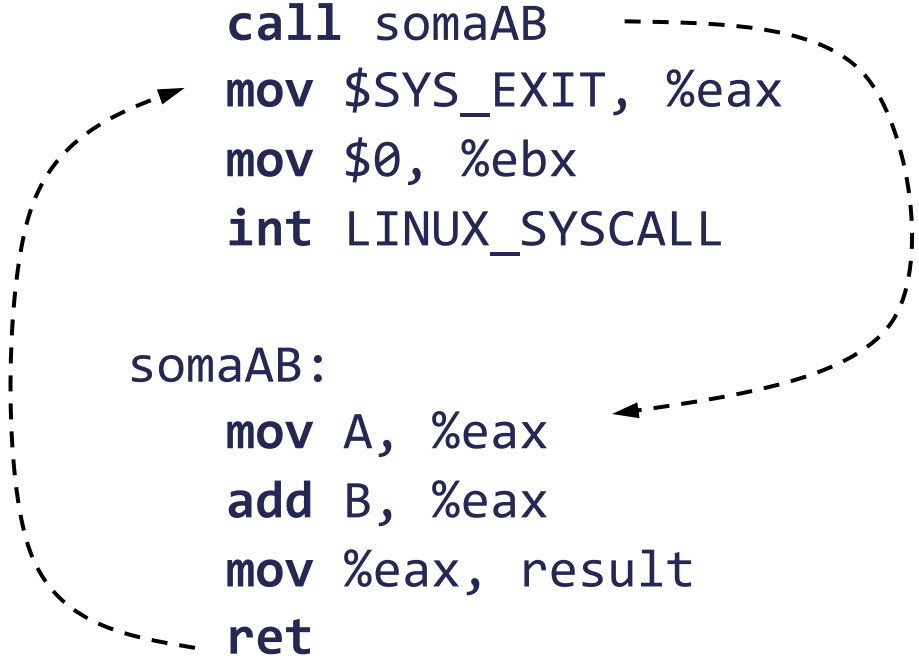
  ESP ← ESP + 4

Nota: há que garantir que o endereço de retorno está mesmo no topo da pilha

Exemplo:

```
SYS_EXIT = 1
LINUX_SYSCALL = 0x80

.global _start

.data
  A:      .int 0
  B:      .int 0
  result: .int 0
```

```
.text
_start:
    movl $13, A
    movl $4, B
    call somaAB
    mov $SYS_EXIT, %eax
    mov $0, %ebx
    int LINUX_SYSCALL

somaAB:
    mov A, %eax
    add B, %eax
    mov %eax, result
    ret
```



Passagem de parâmetros e resultado

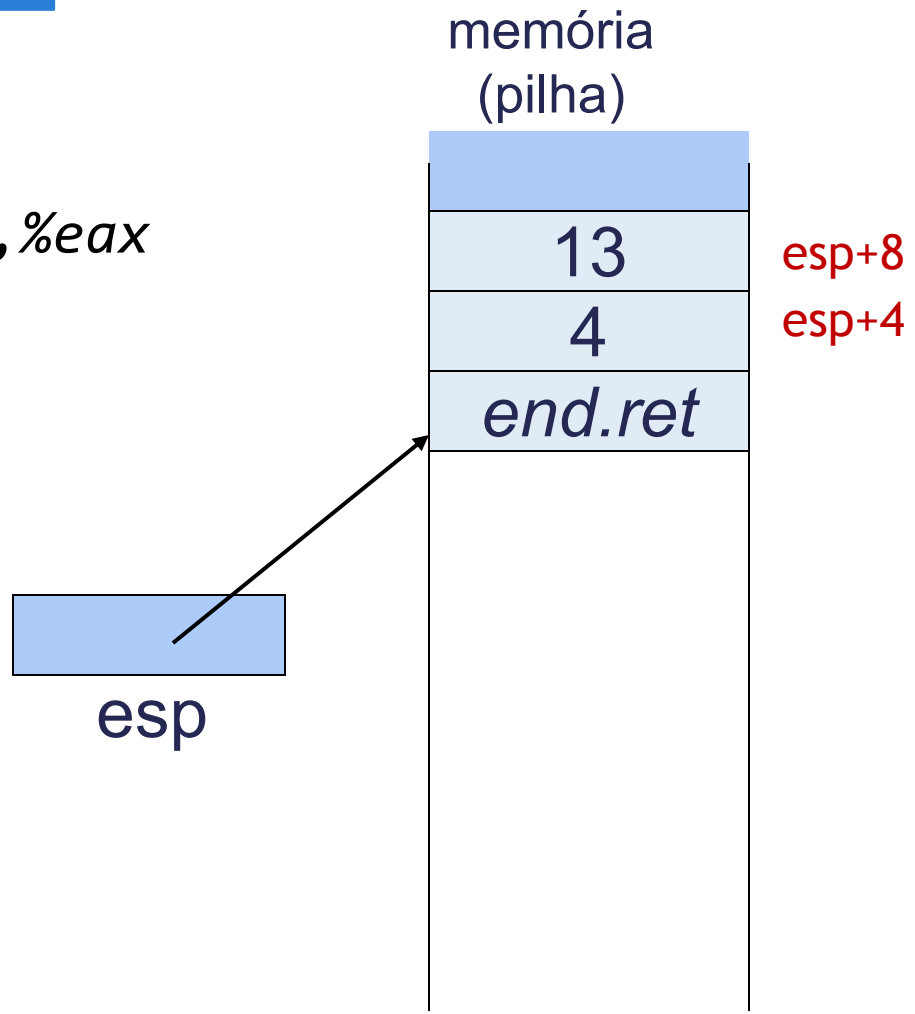
- Modo:
  - Por cópia do valor
  - Por referência ou endereço (apontador)
- Onde:
  - Em posições pré-definidas de memória?
  - Em registos do CPU?
  - Através da pilha de execução?
- E o resultado das funções?
  - Onde fica?

Parâmetros na pilha

- A solução mais flexível é usar a pilha
  - Especialmente se existirem poucos registos
- Chamada:
  - empilham-se todos os parâmetros (**push**)
  - chama-se a subrotina (**call**)
- Na subrotina:
  - acede-se às posições de memória onde ficaram os parâmetros
- Há que normalizar estas convenções:
  - Dimensão dos parâmetros, ordem destes na pilha, etc.

Exemplo

```
…                       mySoma:
pushl $13                   …
pushl $4                    mov 8(%esp),%eax
call mySoma                 …
…                           ret
```



Exemplo:

- chamar a função mySoma passando 13 e 4
- em mySoma queremos aceder ao 1º parâmetro empilhado (valor 13):

  Está em ESP+8

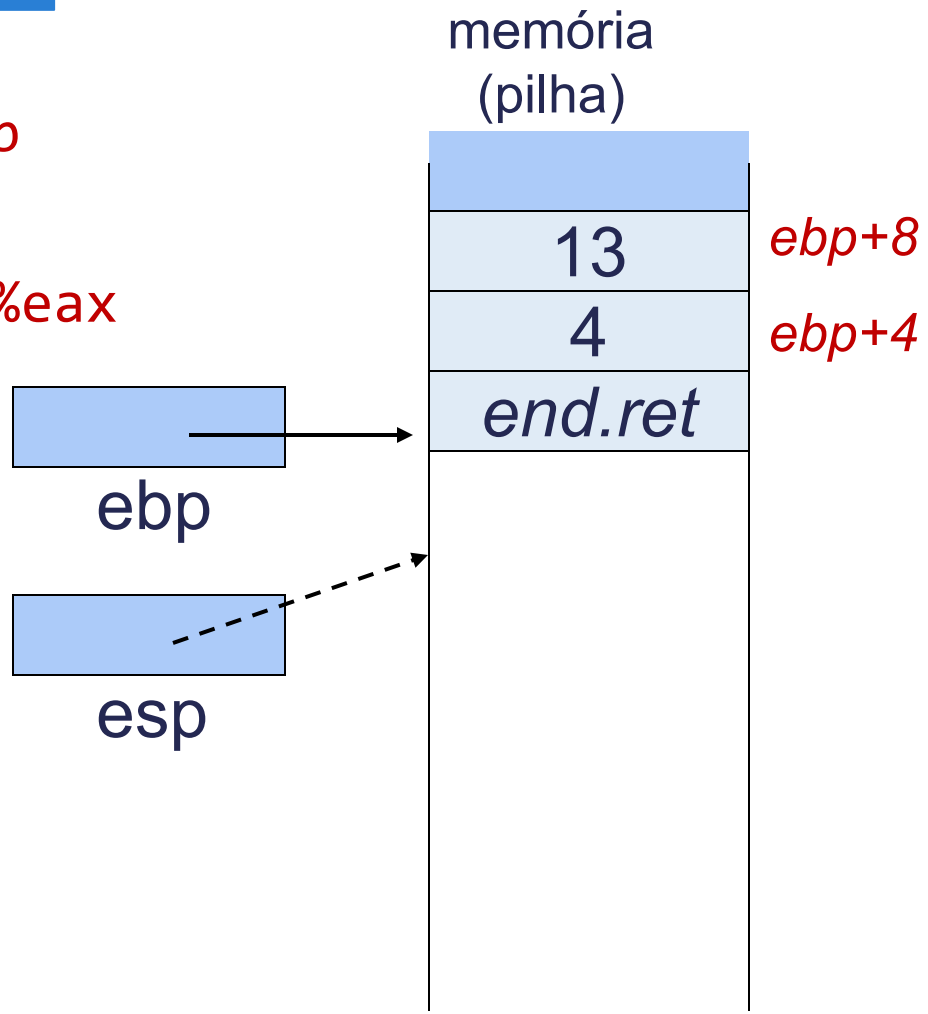
O registo “Frame Pointer”

- Usar o SP como registo base não é prático uma vez que podem ser feitos *pushs* e *pops* e assim a distância aos parâmetros variar
- Normalmente usa-se outro registo que está intrinsecamente ligado ao *stack* – o *Frame Pointer*
  - O objectivo é manter um endereço de referência durante a execução da subrotina (podendo o SP variar)
  - Este pode ser usado como registo base para acesso aos parâmetros e às variáveis locais colocadas na pilha (esta zona é o ***frame de ativação da subrotina***)
- No Pentium este registo chama-se EBP (*Extended Base Pointer*)

Exemplo – usando EBP

```
…
pushl $13
pushl $4
call mySoma
…
```

```
mySoma:
        mov %esp, %ebp
        …
        mov 8(%ebp), %eax
        …
        ret
```

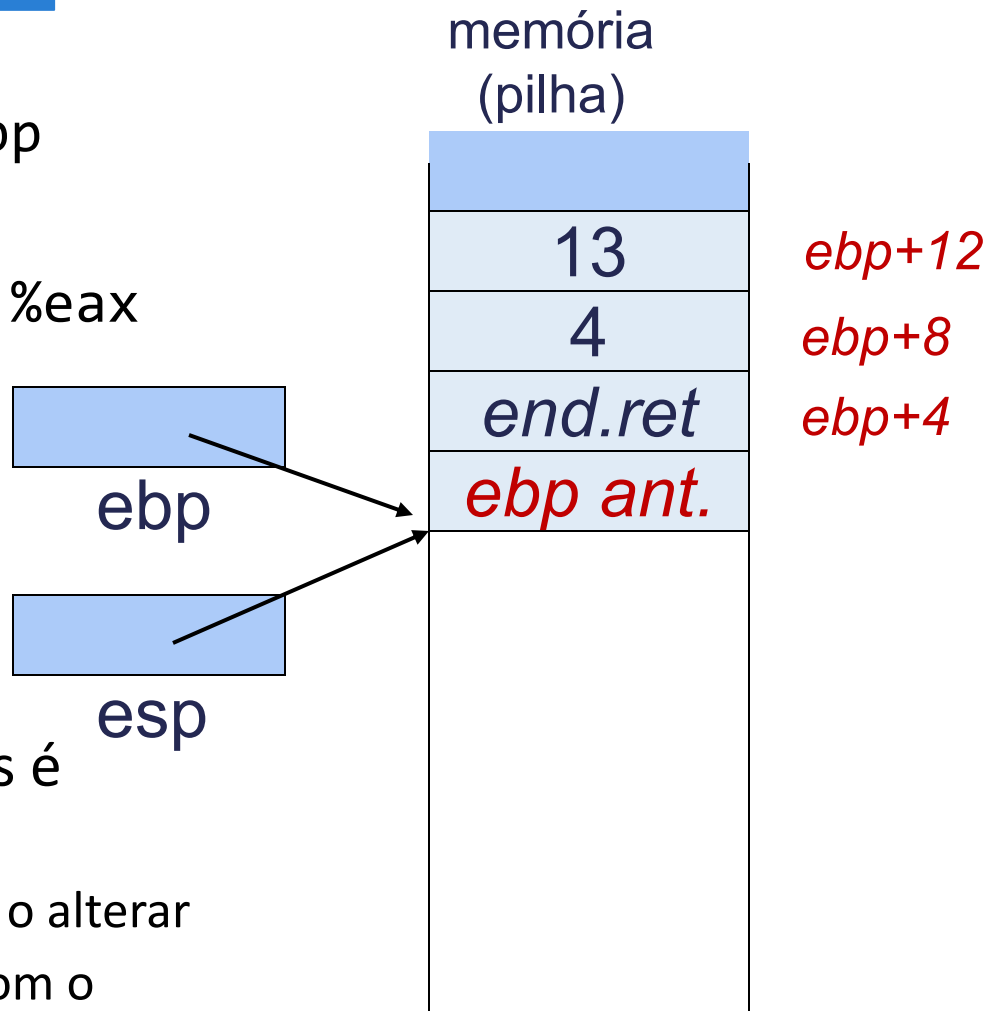


- Para continuar a usar a pilha temos de "libertar" **esp**
  - Copia-se o endereço em **esp** para **ebp**
- Se **esp** mudar, os parâmetros ficam na mesma posição em relação a **ebp**
  - os parâmetros devem ser acessíveis sempre do mesmo modo ao longo de toda a subrotina

Exemplo – guardar/repor EBP

```
…
pushl $13
pushl $4
call mySoma
…
```

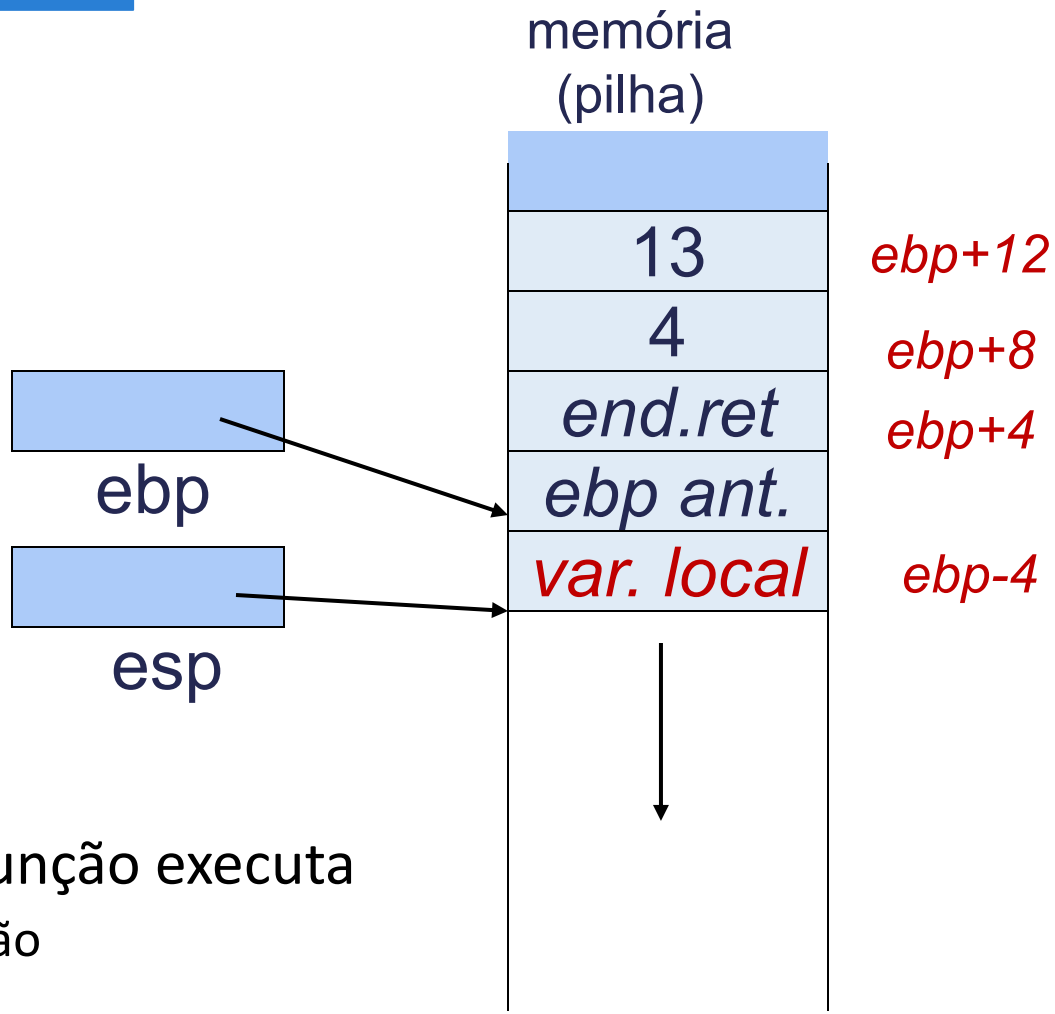
```
mySoma: push %ebp
        mov %esp, %ebp
        …
        mov 12(%ebp),%eax
        …
        pop %ebp
        ret
```



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

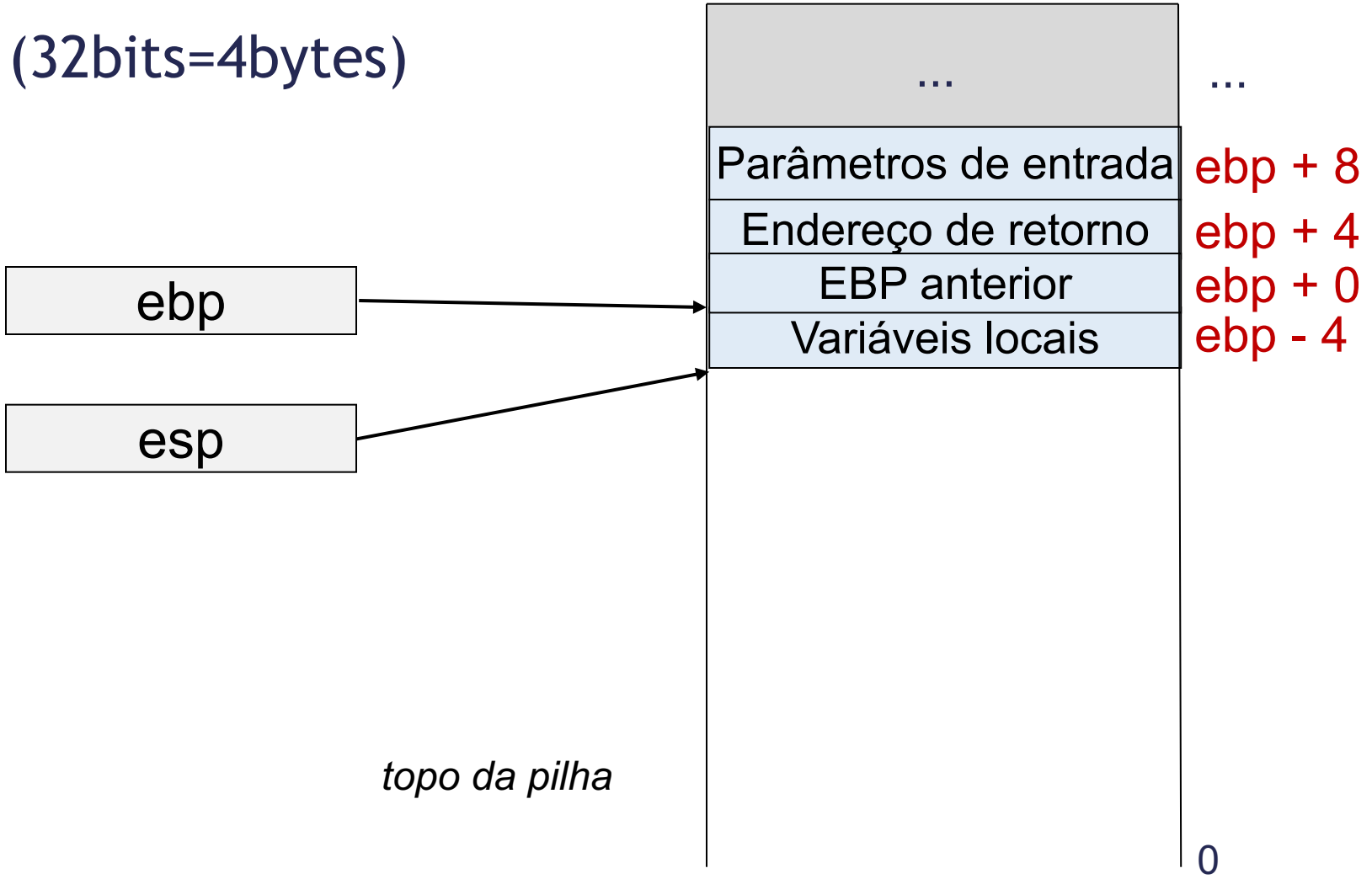
Exemplo – Variável locais

```
mySoma: push %ebp
        mov %esp, %ebp
        sub $4, %esp
        …
        mov $5, -4(%ebp)
        …
        mov %ebp, %esp
        pop %ebp
        ret
```

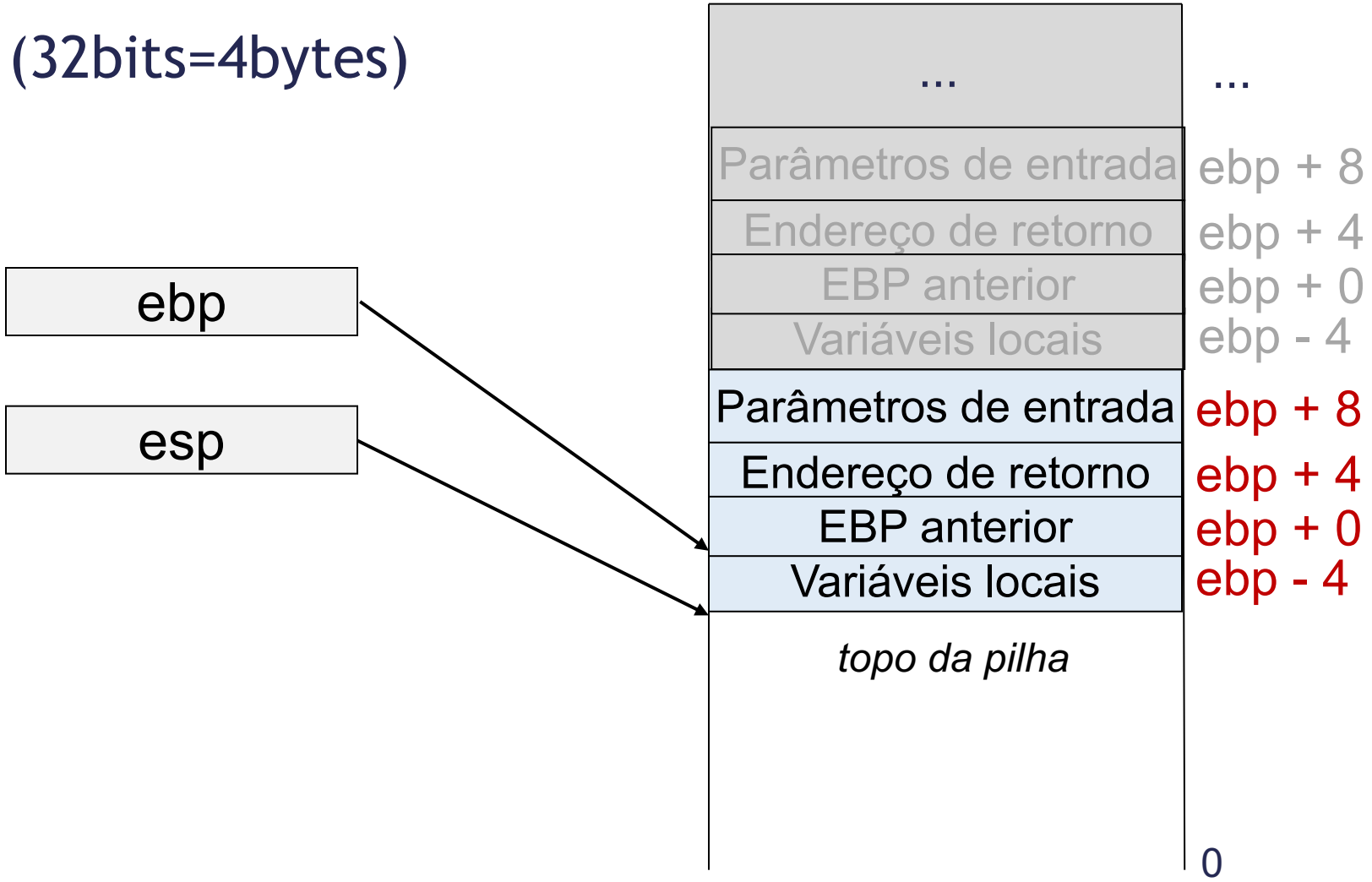


- As variáveis locais só existem enquanto a função executa
  - Var. local→zona de memória no *frame* da função
  - espaço reservado no início e libertado no fim

Frame de activação de subrotina



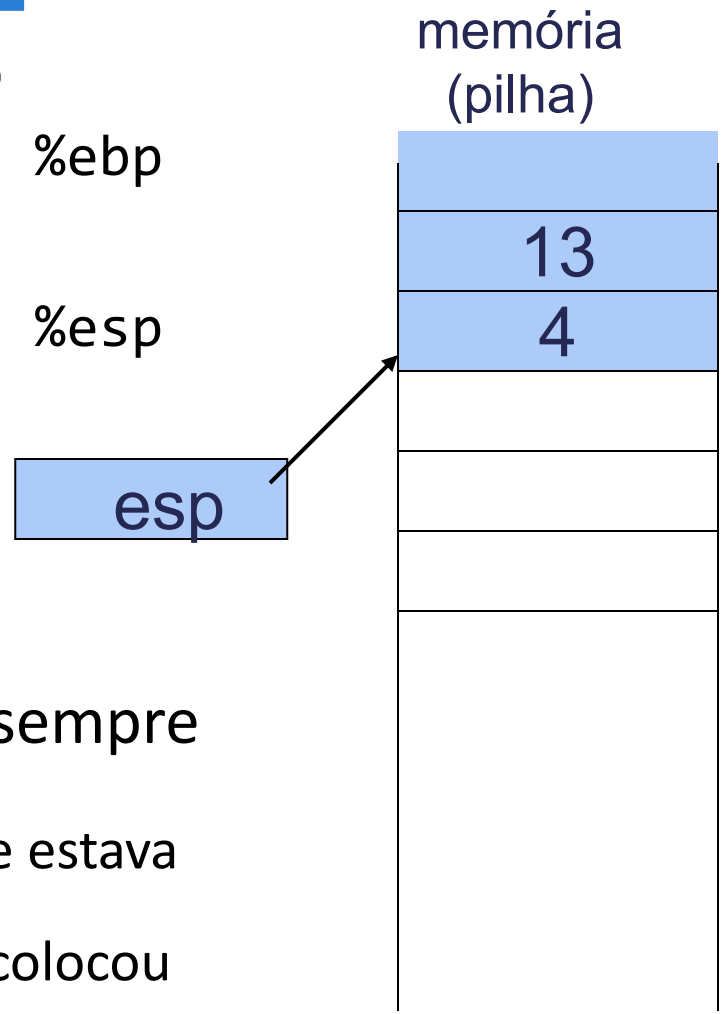
Frame de activação de subrotina



Repor a pilha

```
...
pushl $13
pushl $4
call mySoma
add $8, %esp
...
```

```
mySoma: push %ebp
        mov %esp, %ebp
        ...
        mov %ebp, %esp
        pop %ebp
        ret
```

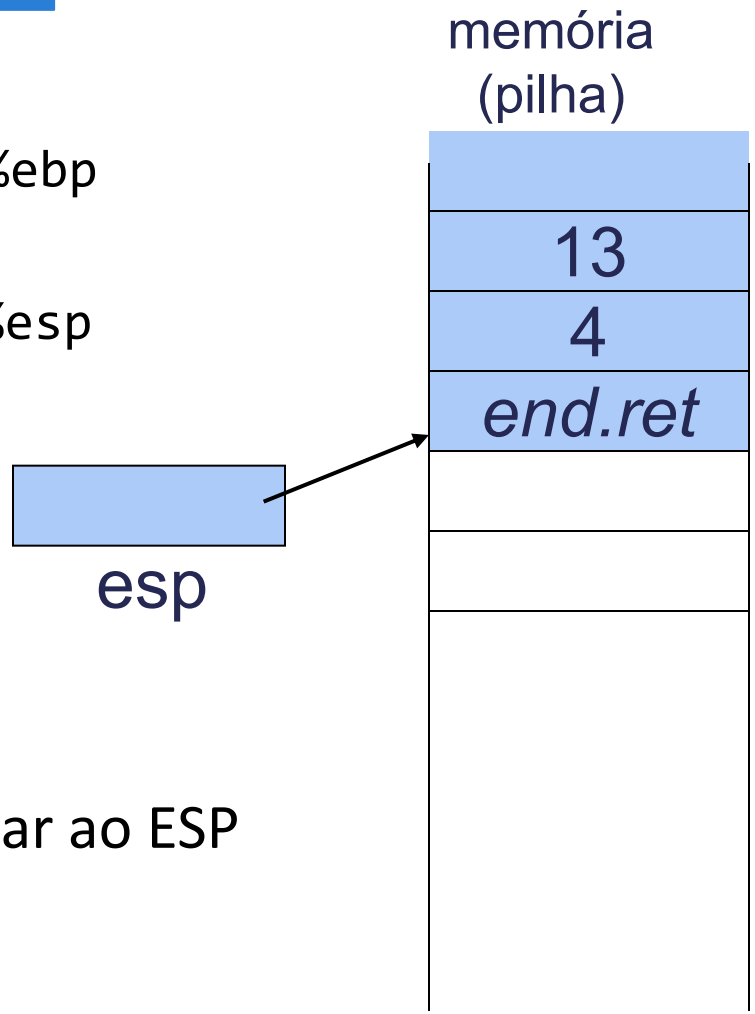


- A pilha deve ser reposta para que não vá sempre crescendo
  - A subrotina deve deixar a pilha no estado que estava aquando da chamada
  - O *chamador* deve retirar os parâmetros que colocou para a chamada

Repor a pilha (alternativa em Intel)

```
…
pushl $13
pushl $4
call mySoma
mov %eax,result
…
```

```
mySoma: push %ebp
        mov %esp, %ebp
        ...
        mov %ebp, %esp
        pop %ebp
        ret 8
```



- A pilha deve ser reposta:
  - Nos Intel, **`ret`** pode ter um valor a somar ao ESP
  - Não é usado por alguns compiladores

Retorno de resultados de funções

```
…
pushl $13
pushl $4
call mySoma
add $8, %esp
mov %eax, resultado
…
```

```
mySoma:
push %ebp
mov %esp, %ebp
mov 8(%ebp), %eax
add 12(%ebp), %eax
mov %ebp, %esp
pop %ebp
ret
```

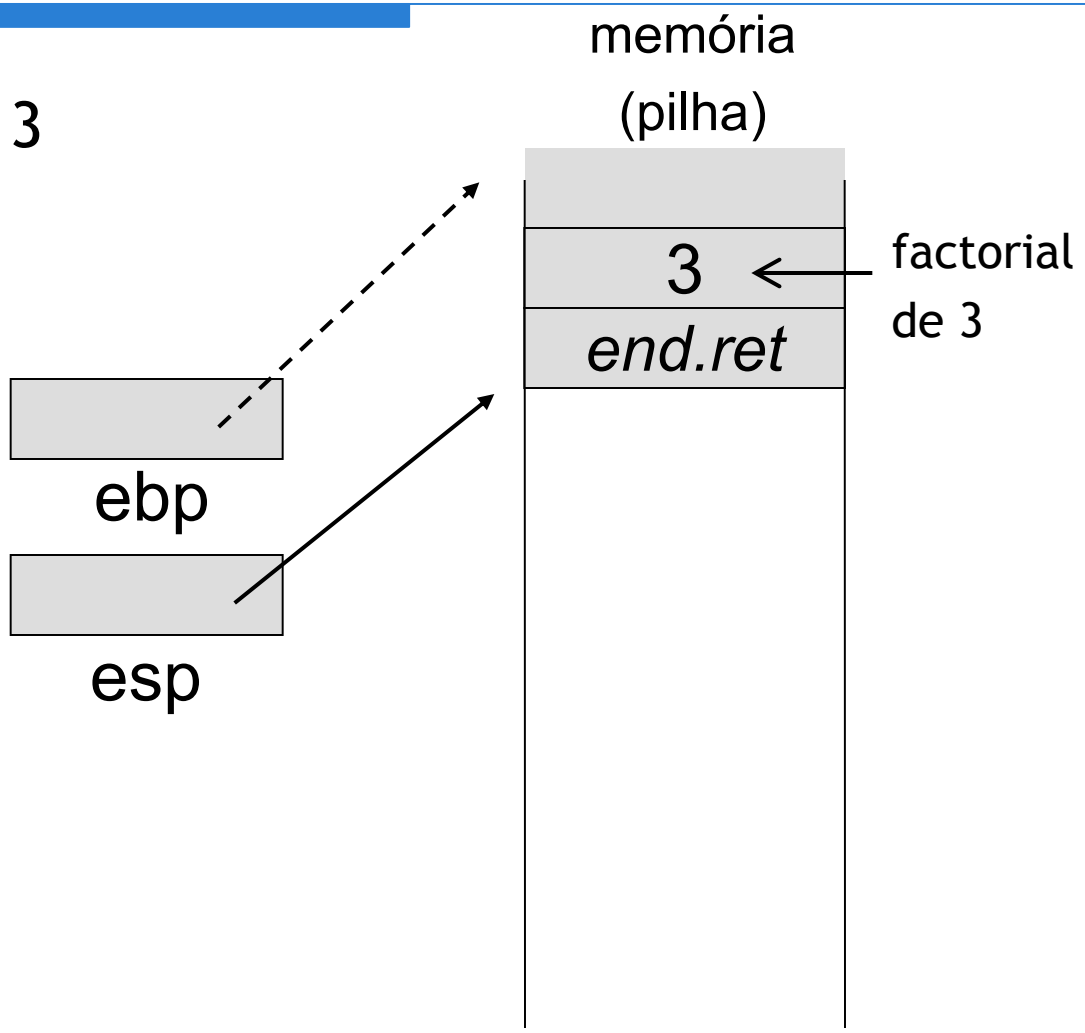
- Se é uma função retorna um resultado
  - Normalmente é usado um registo para retornar o resultado (por exemplo o EAX)

Exemplo: factorial recursivo

Vamos calcular o factorial de 3

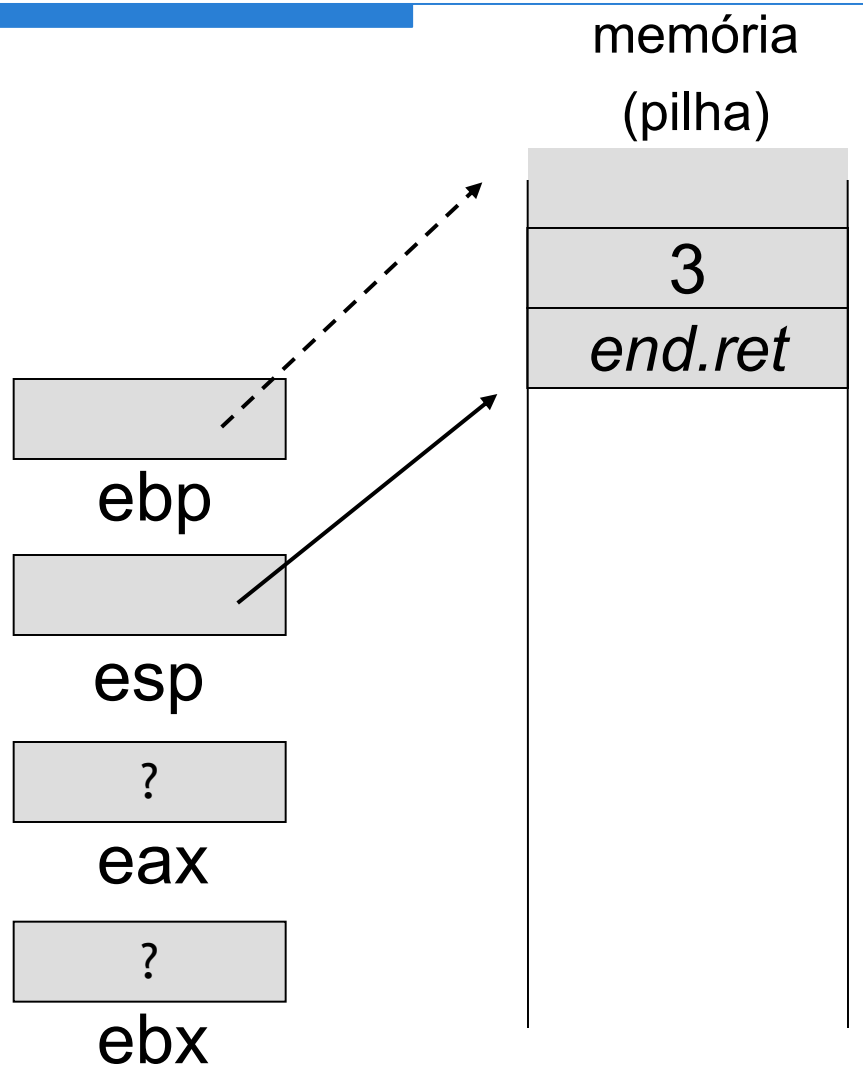
O código da chamada é:

```
...
push 3
call factorial
add $4, %esp
```



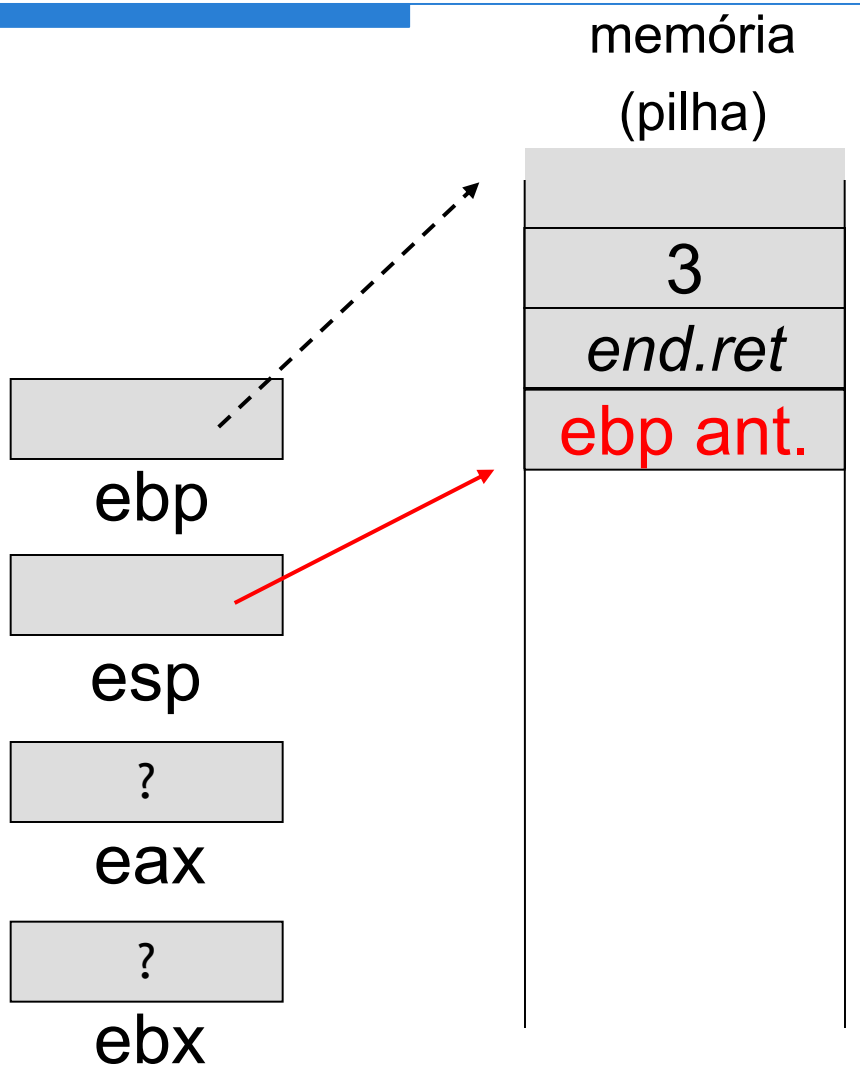
Exemplo: factorial recursivo

```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one

one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



Exemplo: factorial recursivo

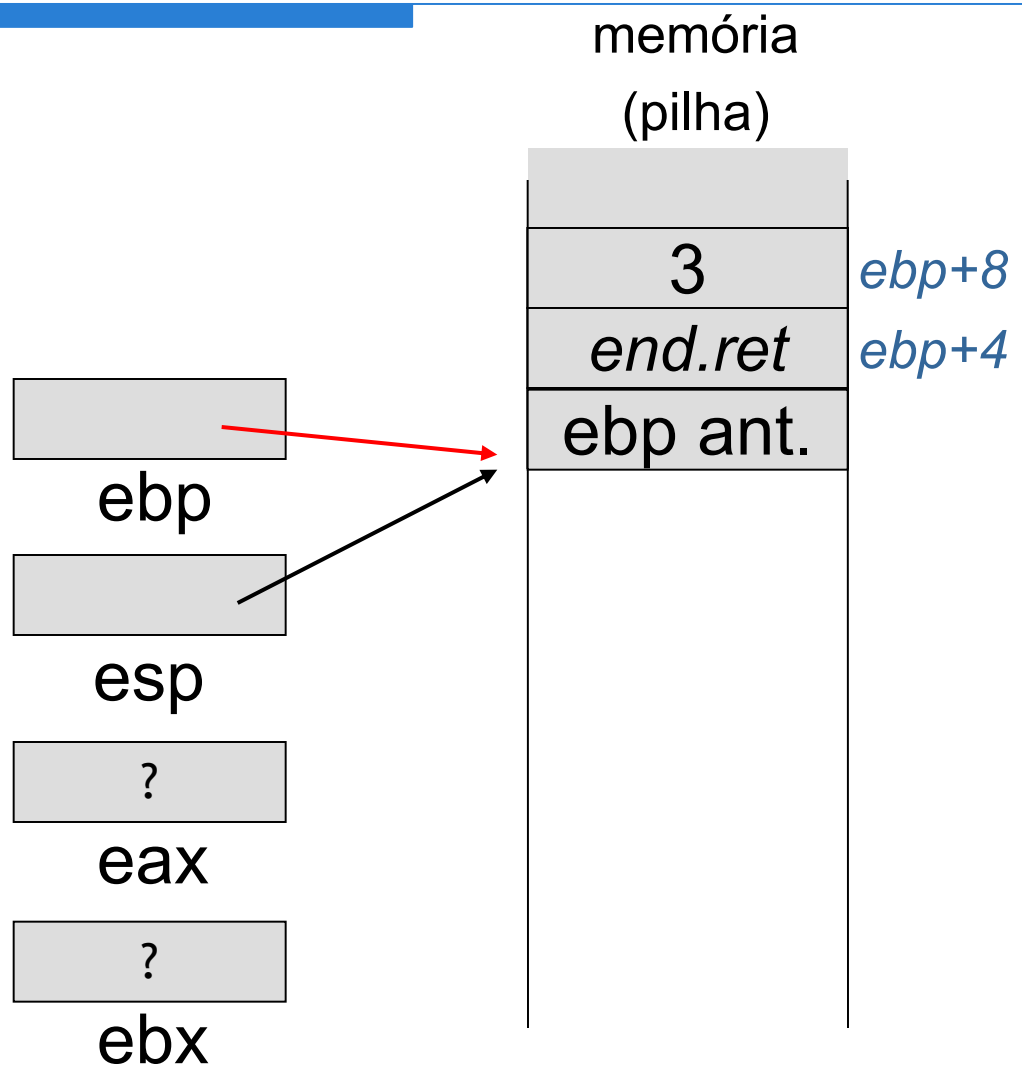
```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    
    
    
    
    
    
    
    
one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



Exemplo: factorial recursivo

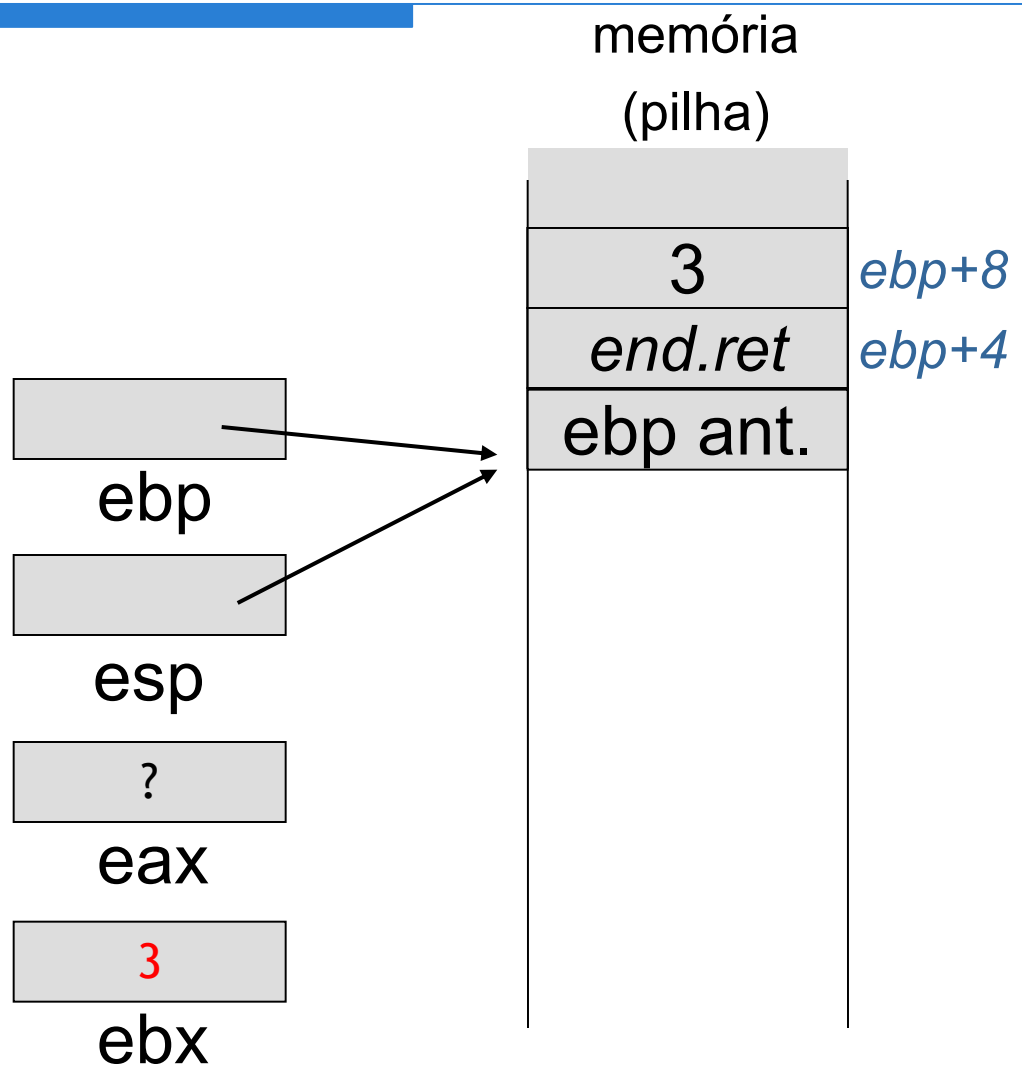
```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
```

```
one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



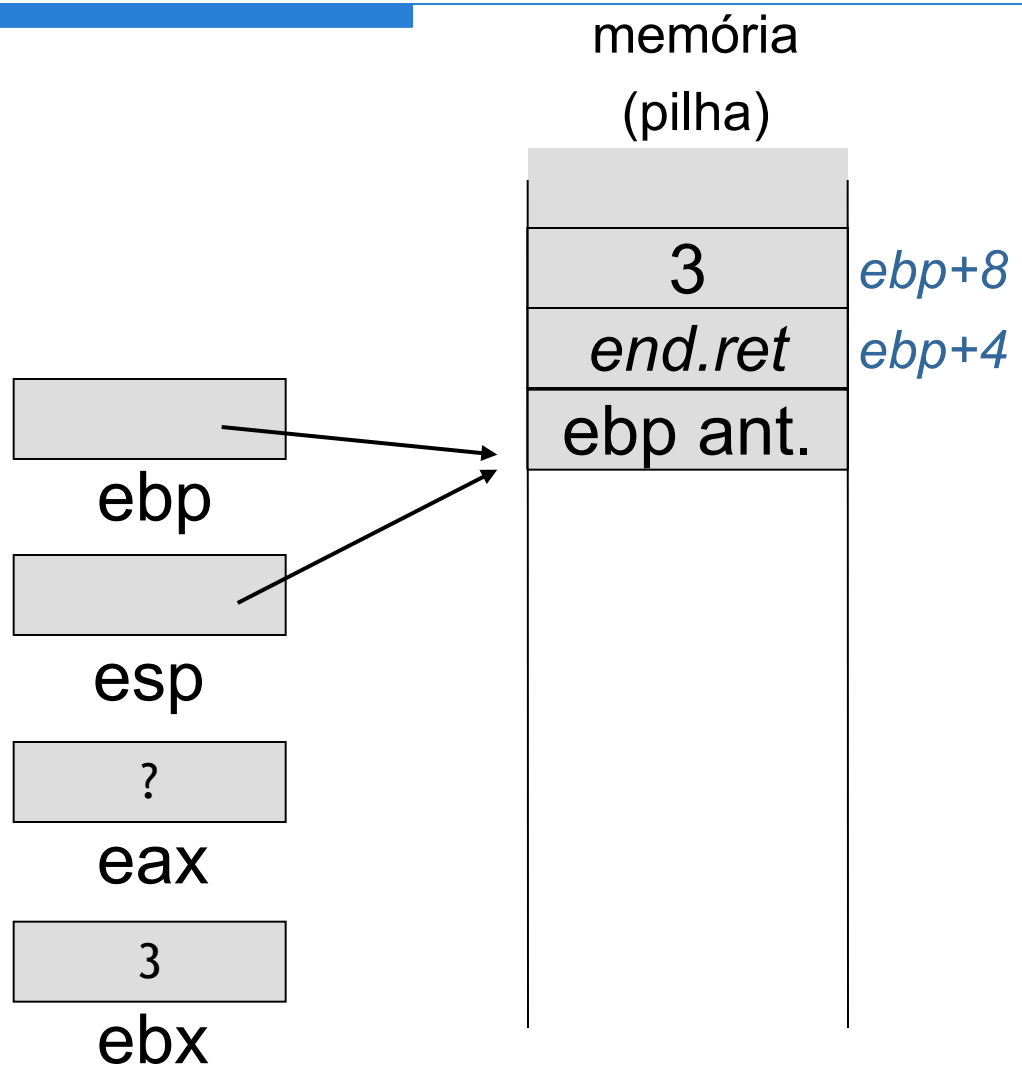
Exemplo: factorial recursivo

```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one

one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



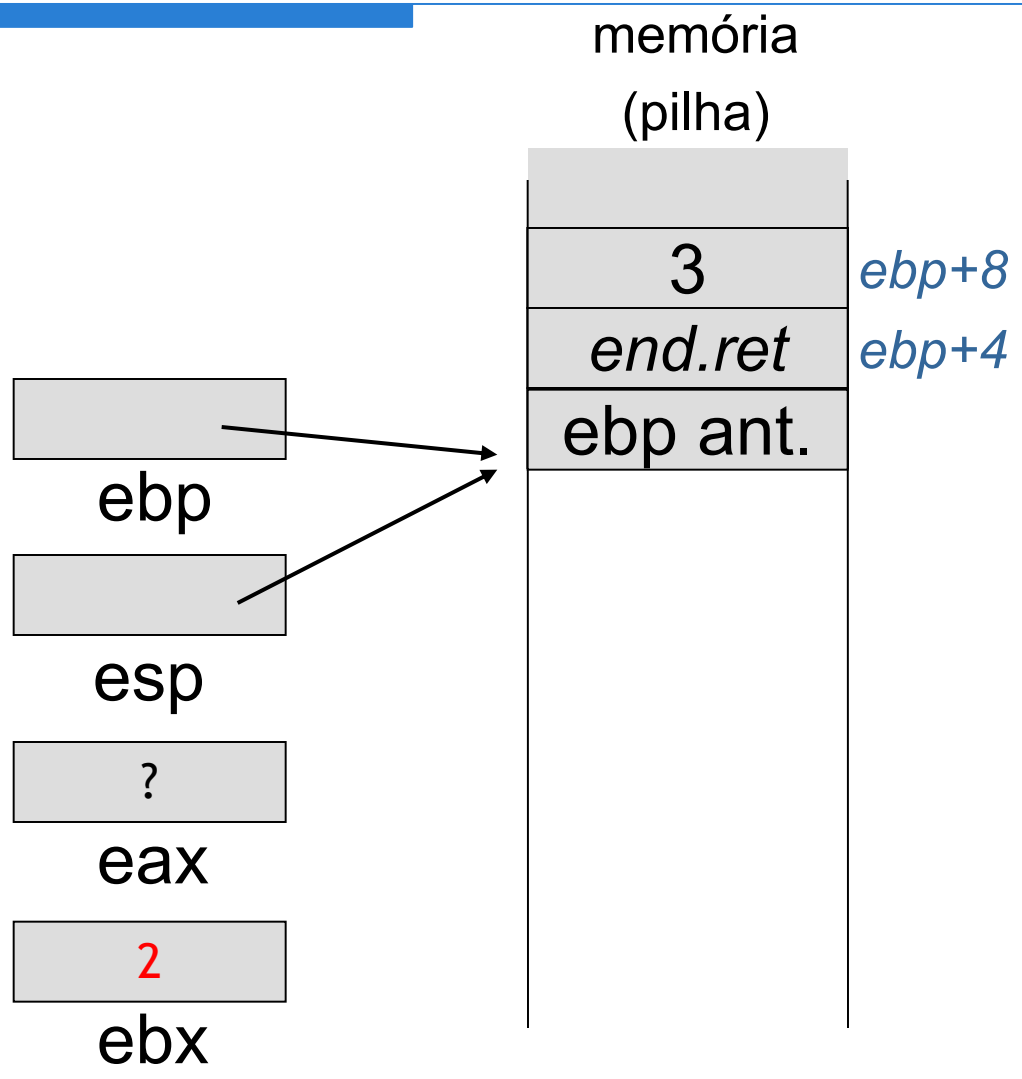
Exemplo: factorial recursivo

```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one

one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



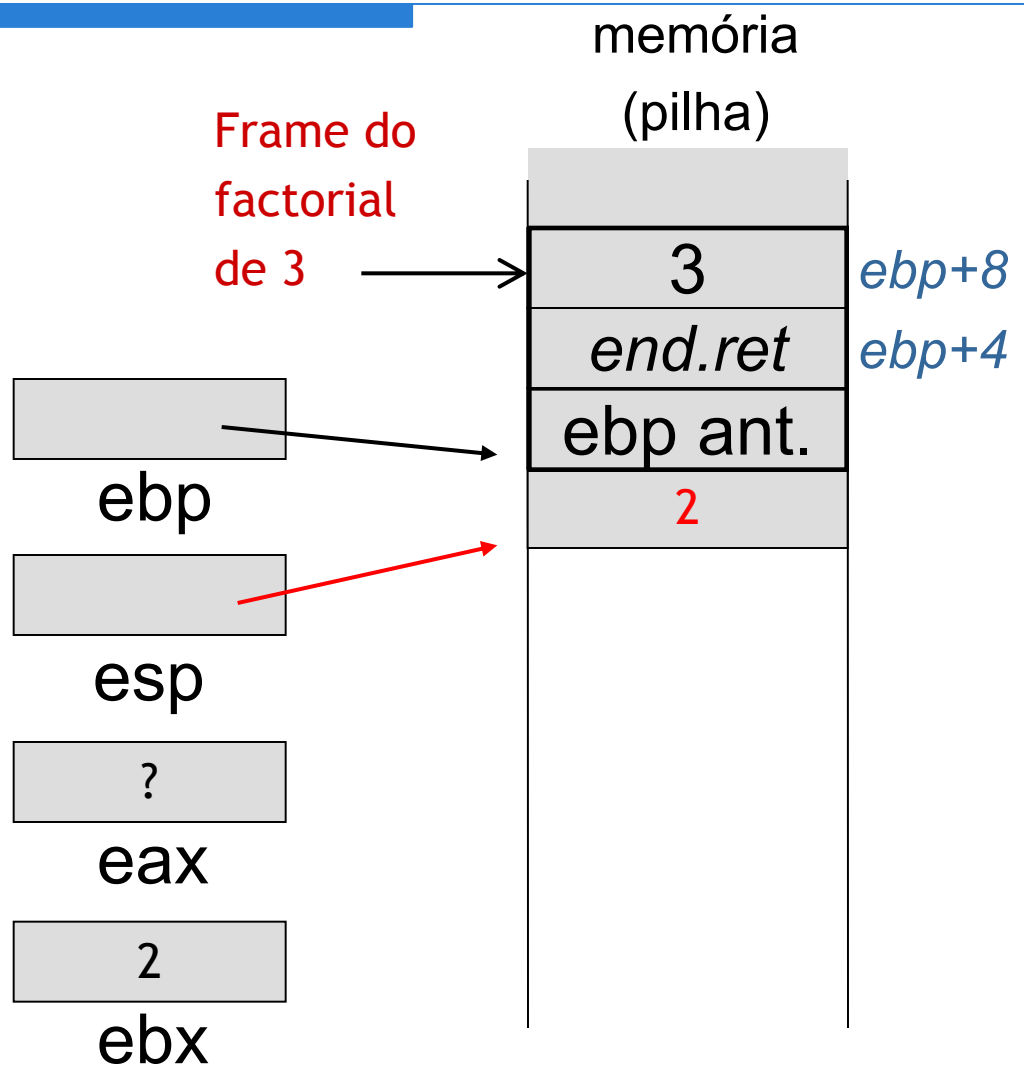
Exemplo: factorial recursivo

```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx

one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



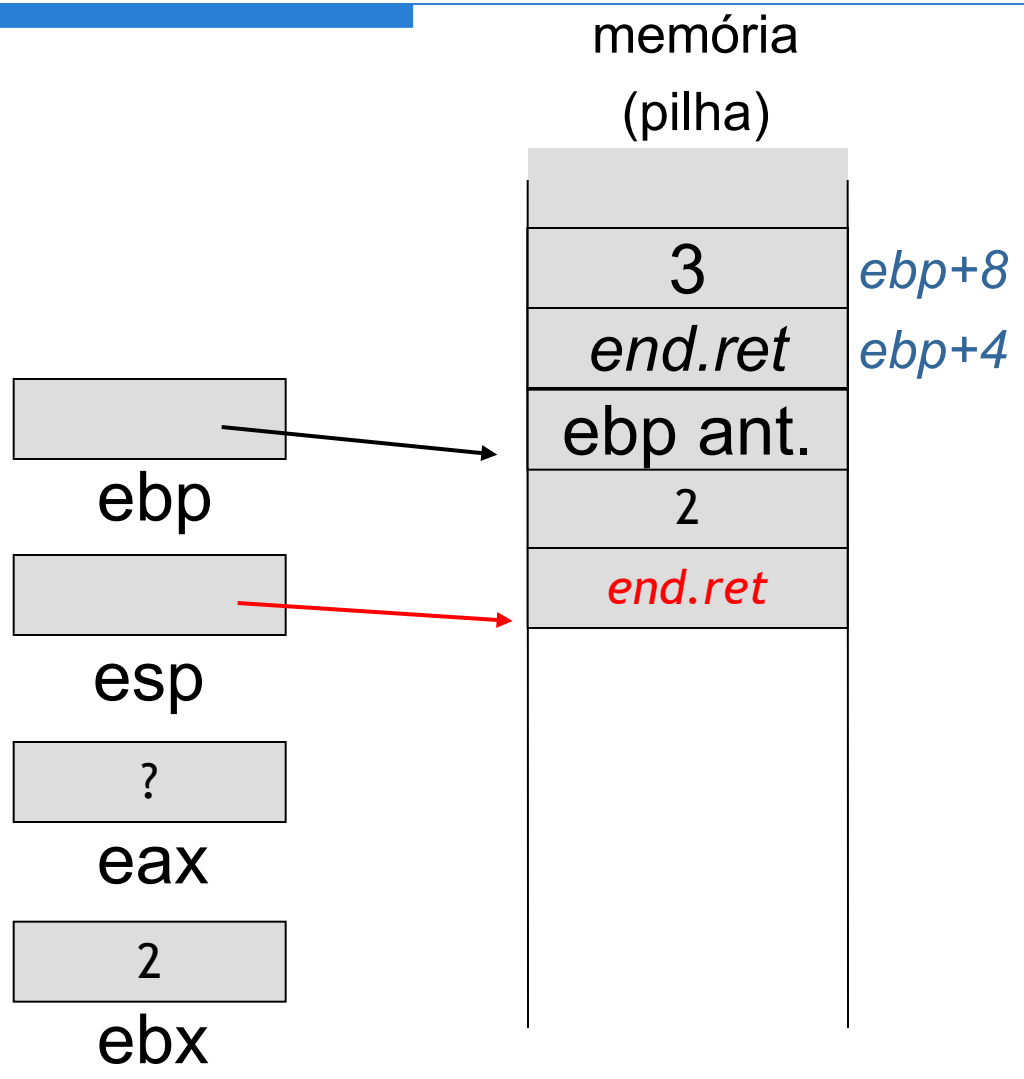
Exemplo: factorial recursivo

```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx




one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



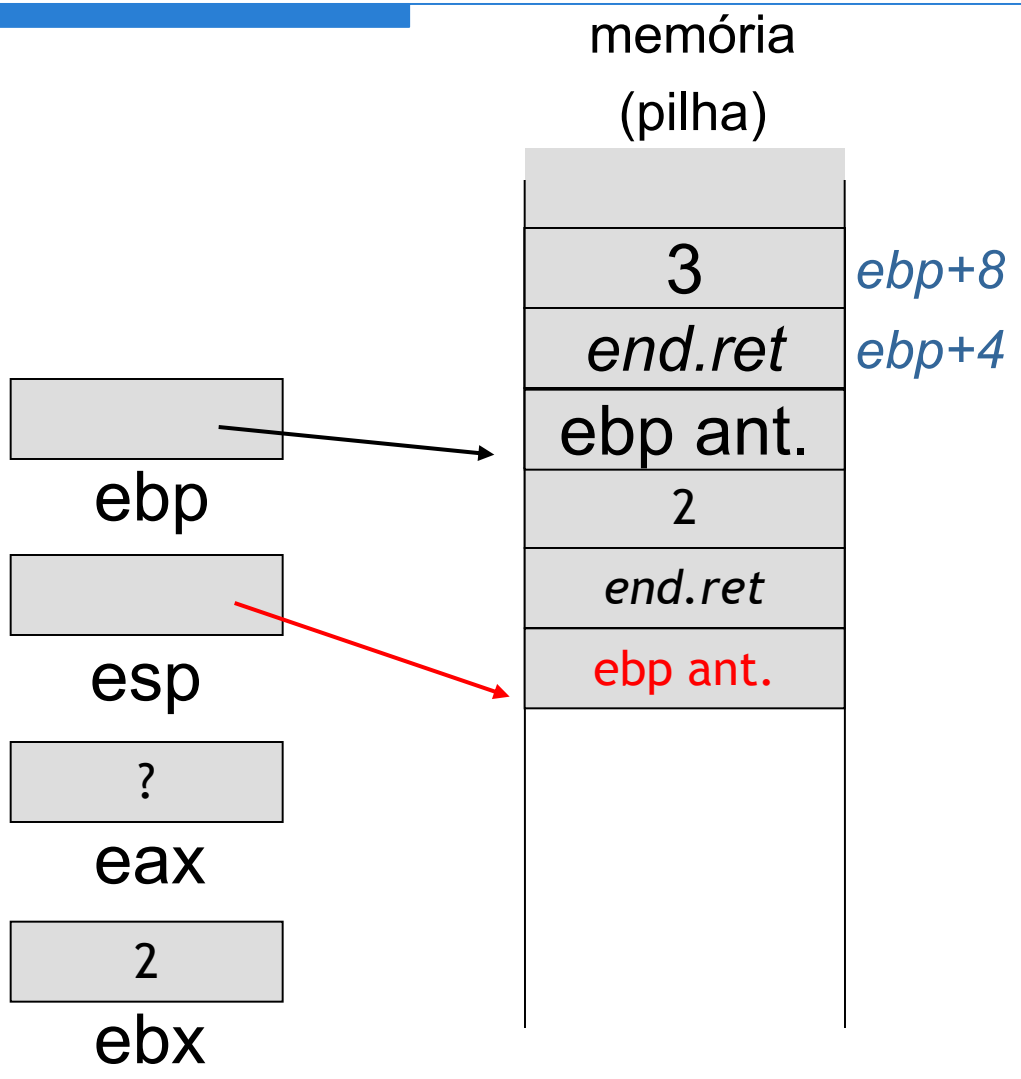
Exemplo: factorial recursivo

```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial


one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



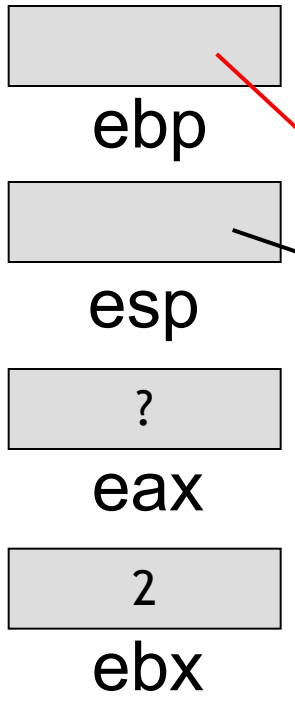
Exemplo: factorial recursivo

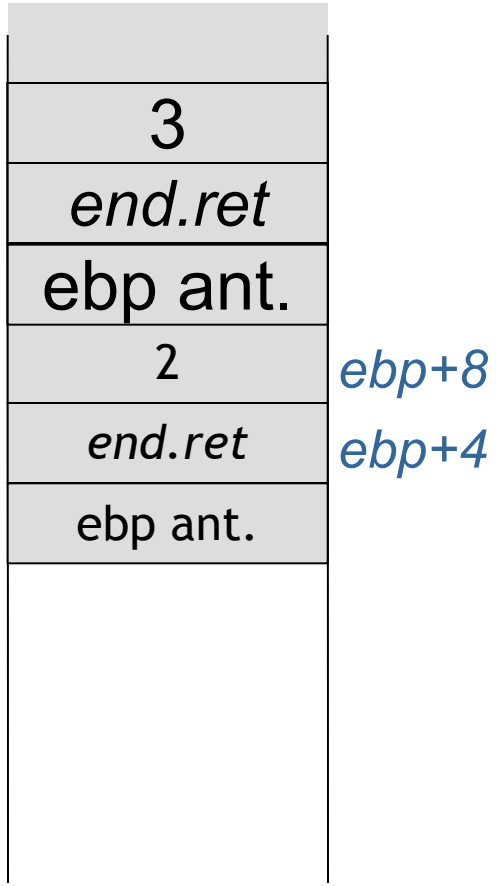
```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial


one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



Exemplo: factorial recursivo

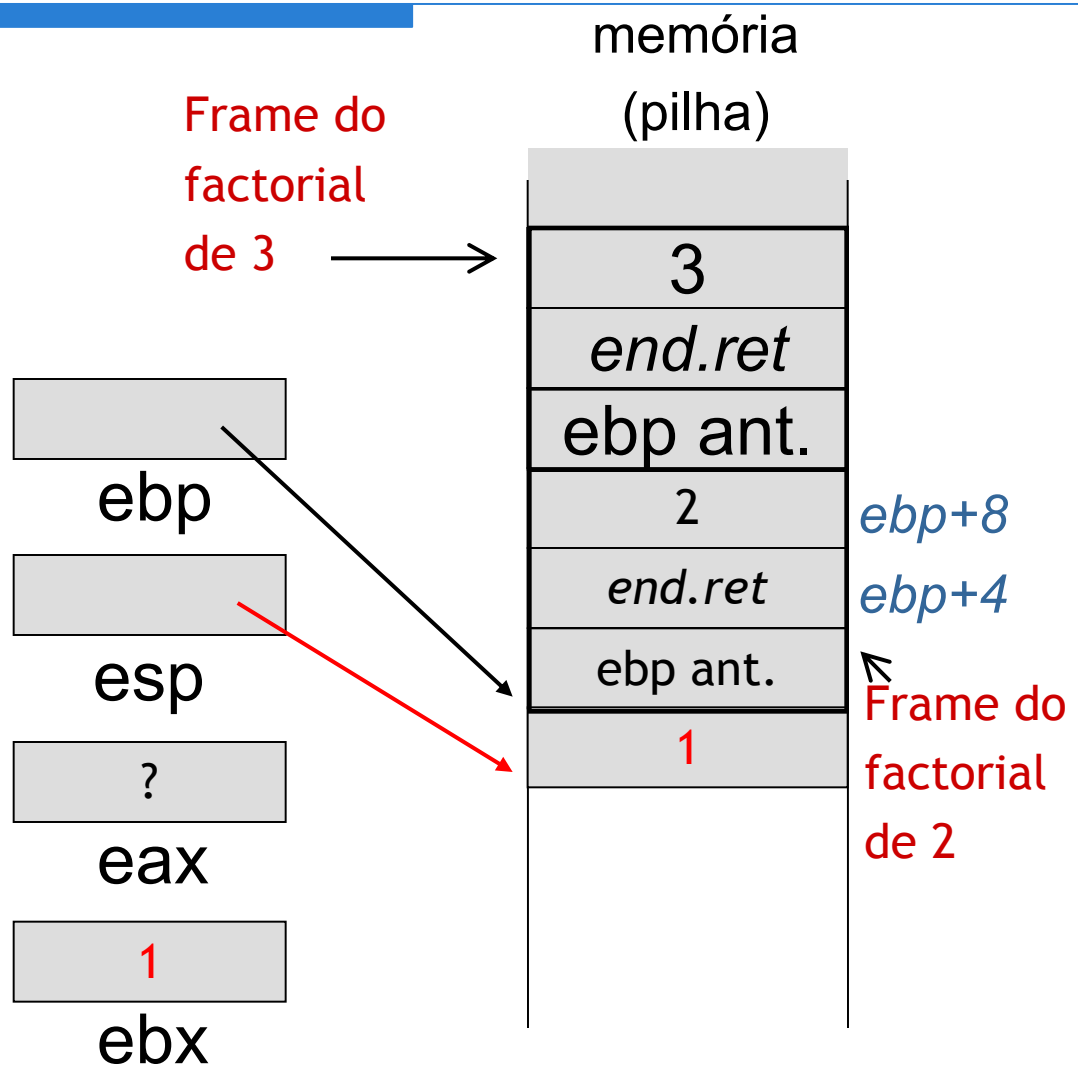
```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial


one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```





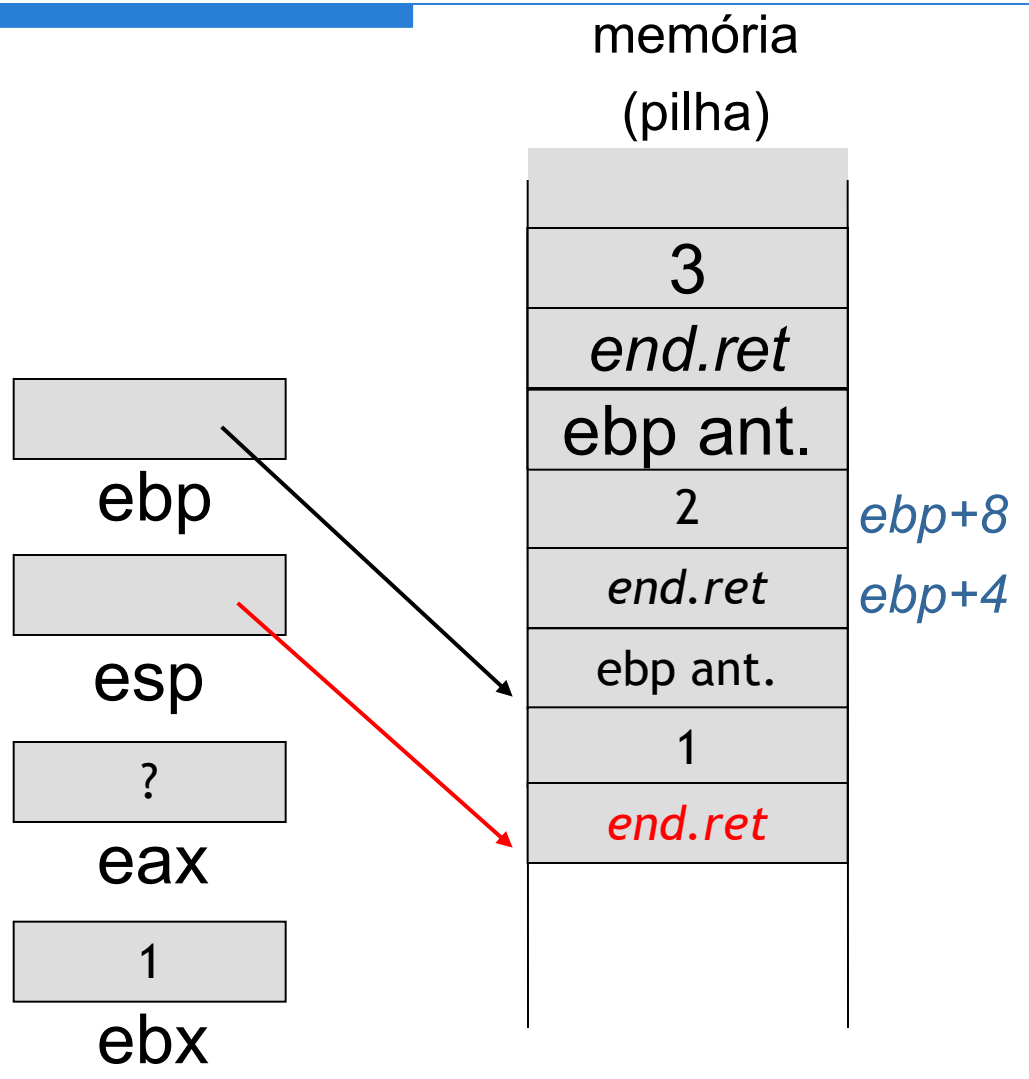
Exemplo: factorial recursivo

```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial

one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



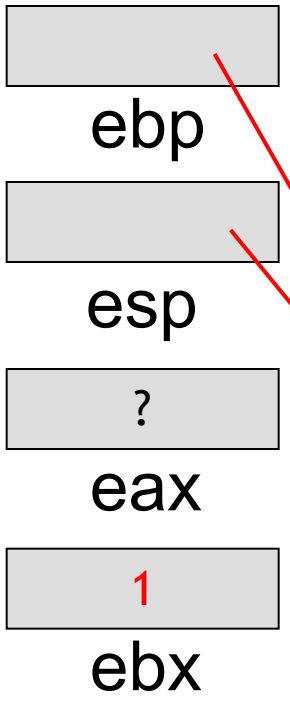
Exemplo: factorial recursivo

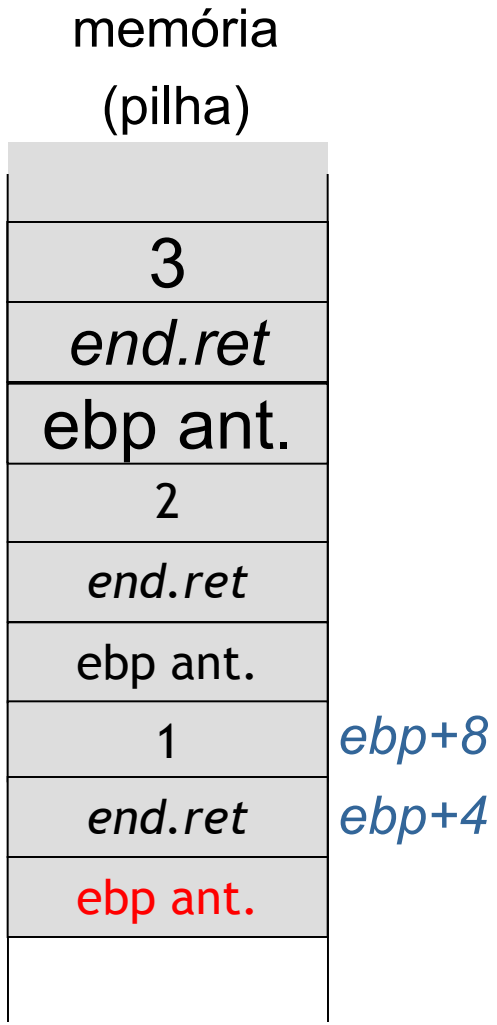
```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial

one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



Exemplo: factorial recursivo

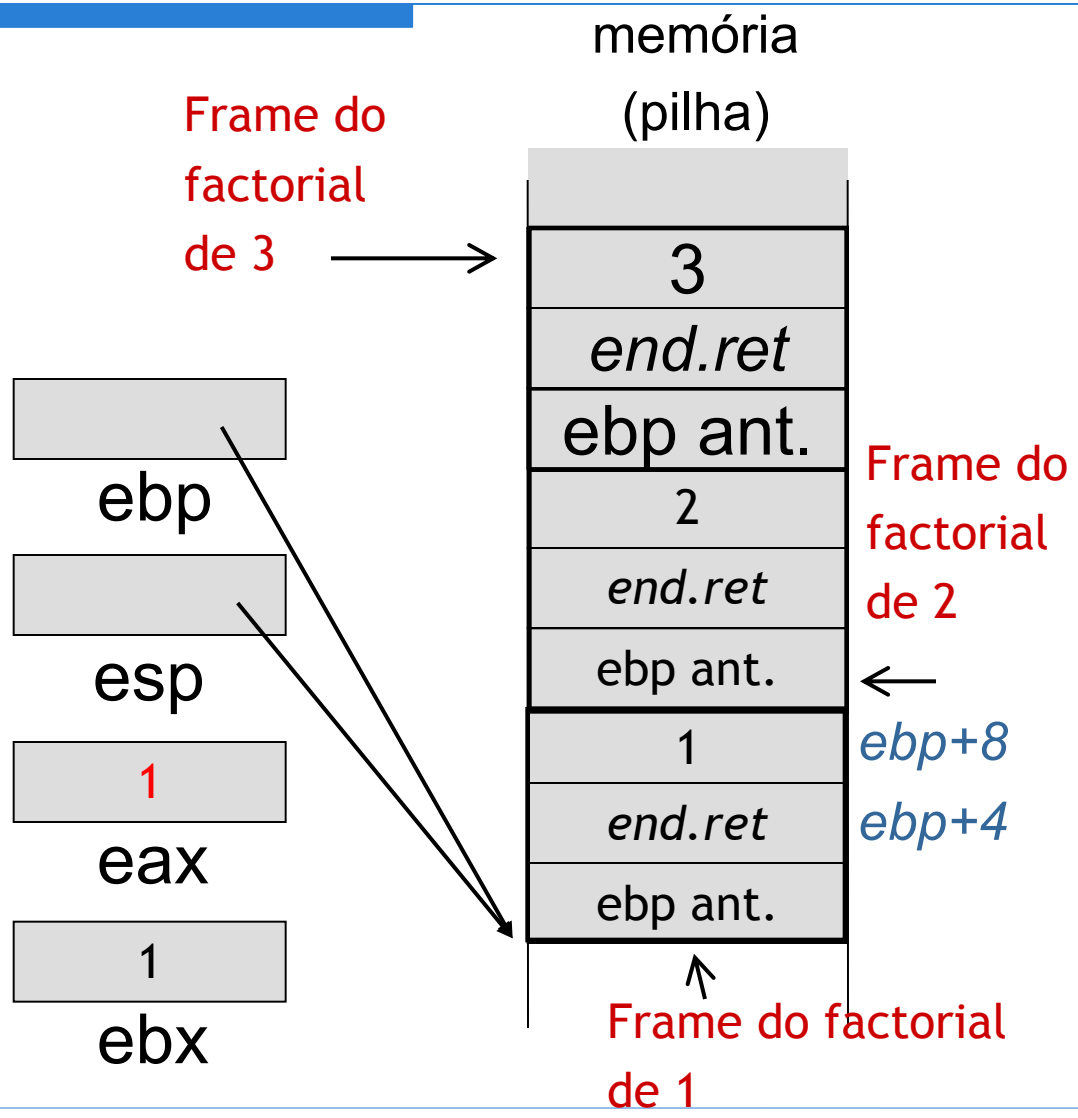
```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial

one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```





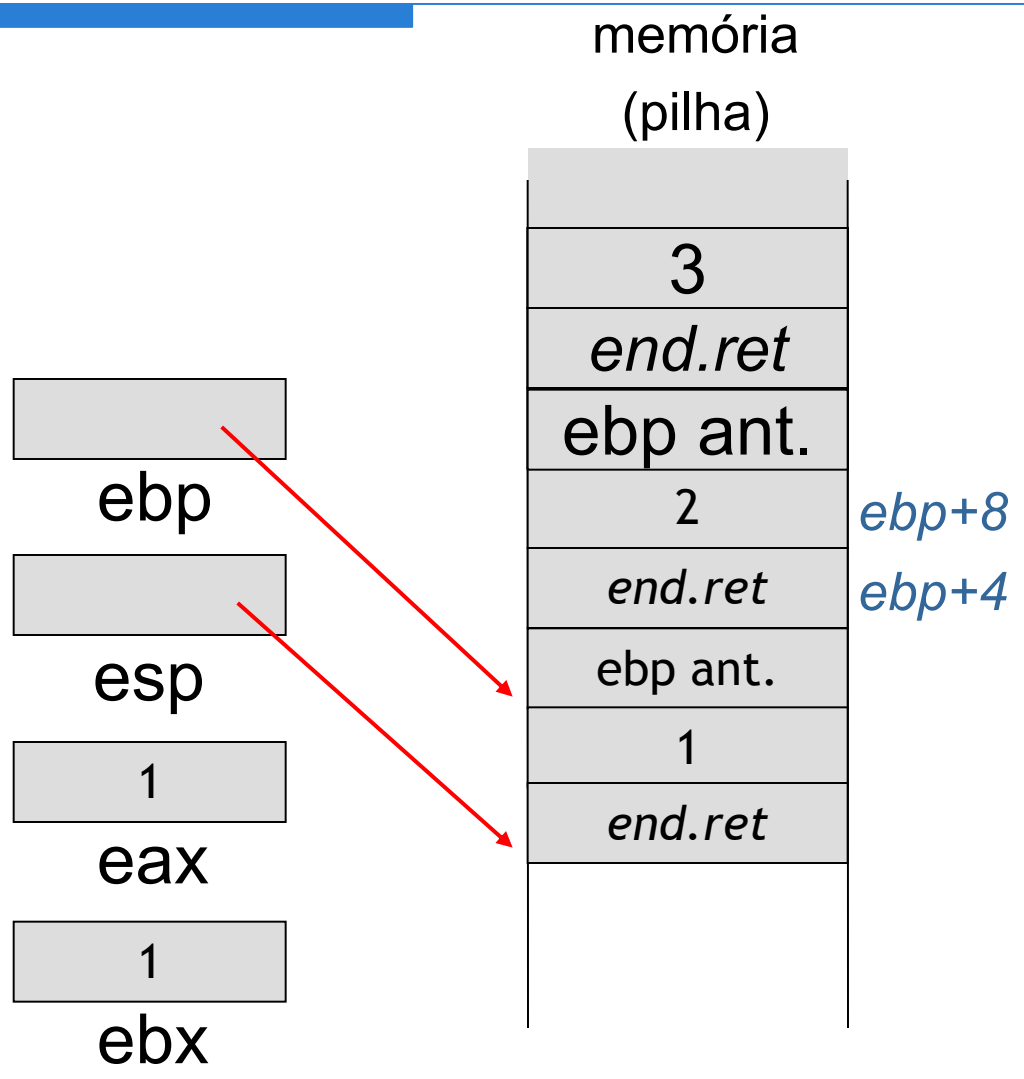
Exemplo: factorial recursivo

```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial

one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



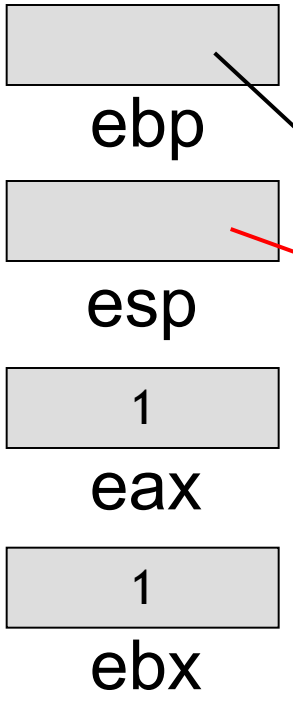
Exemplo: factorial recursivo

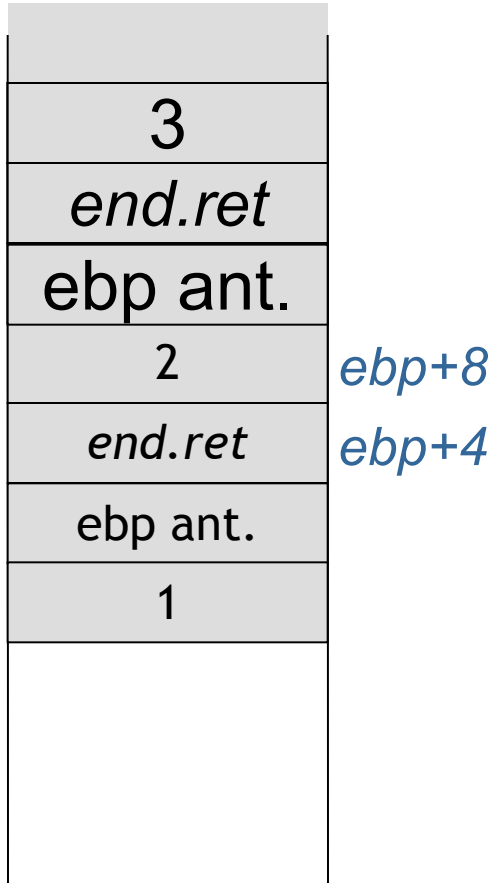
```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial

one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



Exemplo: factorial recursivo

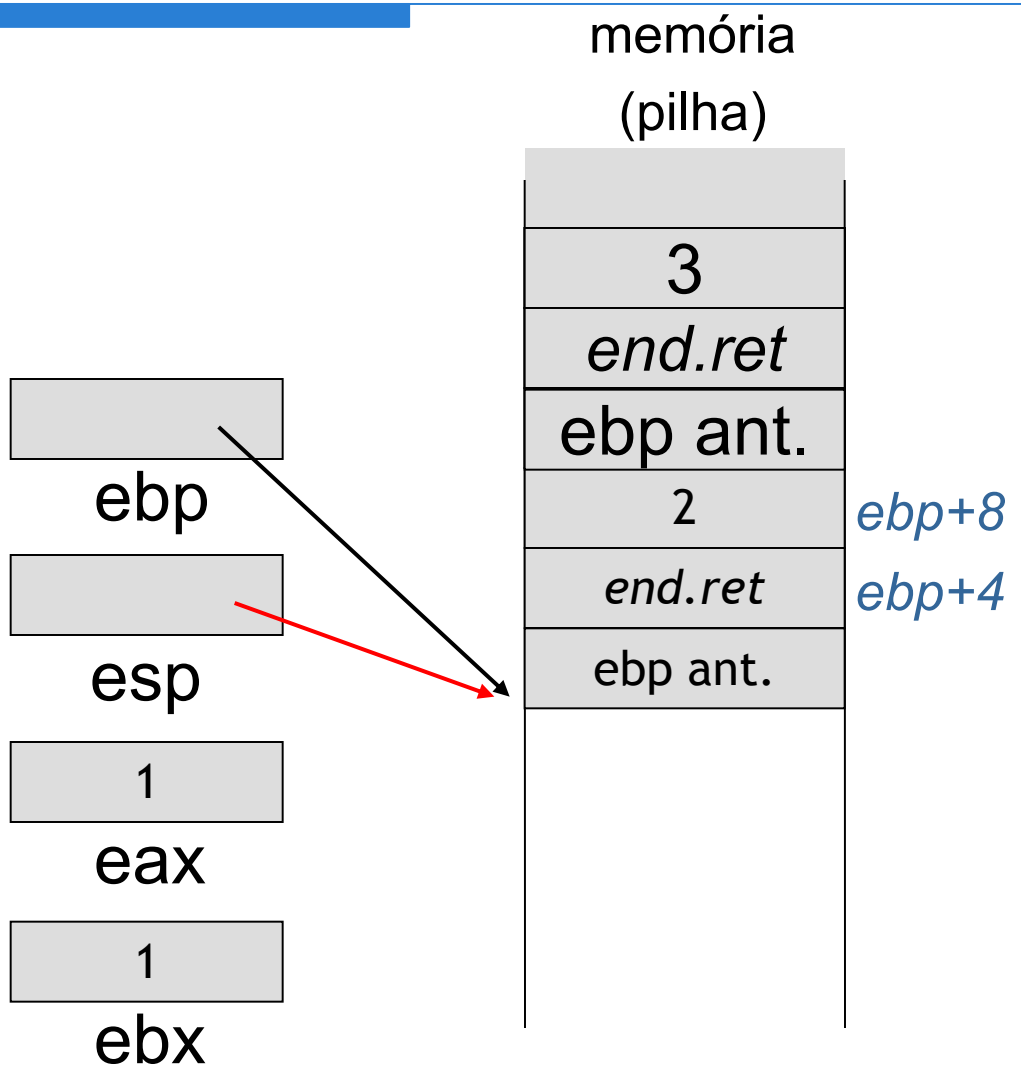
```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial
    add $4, %esp


one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```





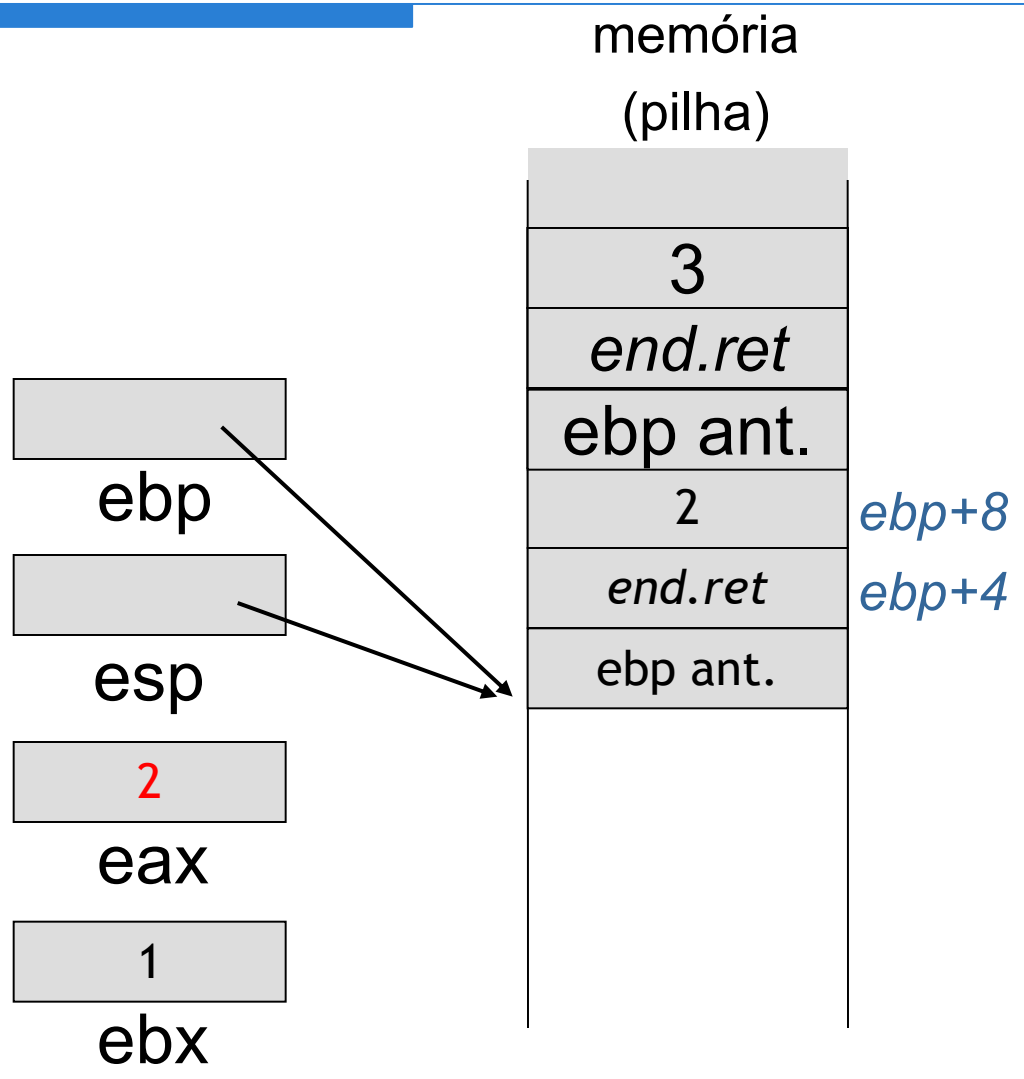
Exemplo: factorial recursivo

```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial
    add $4, %esp


one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



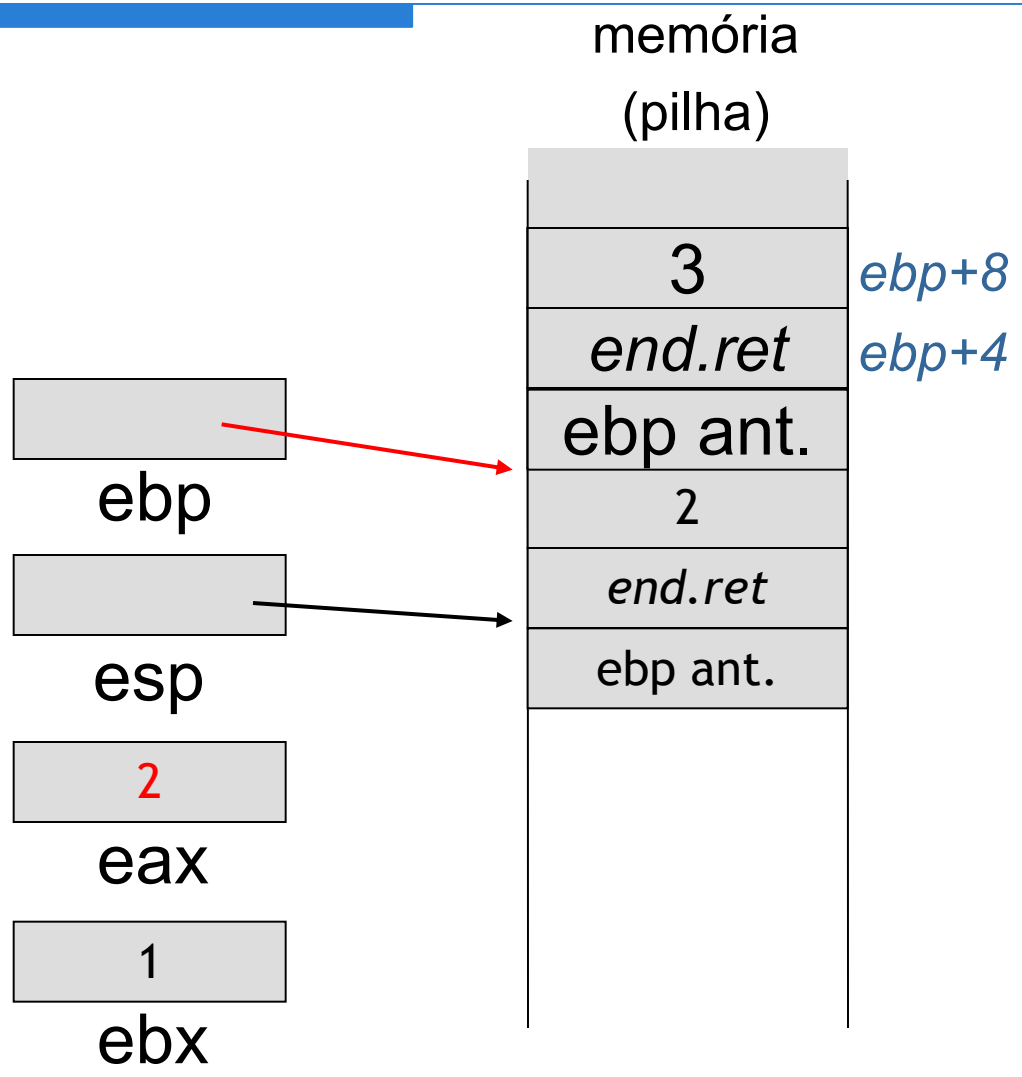
Exemplo: factorial recursivo

```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial
    add $4, %esp
    mull 8(ebp)

one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



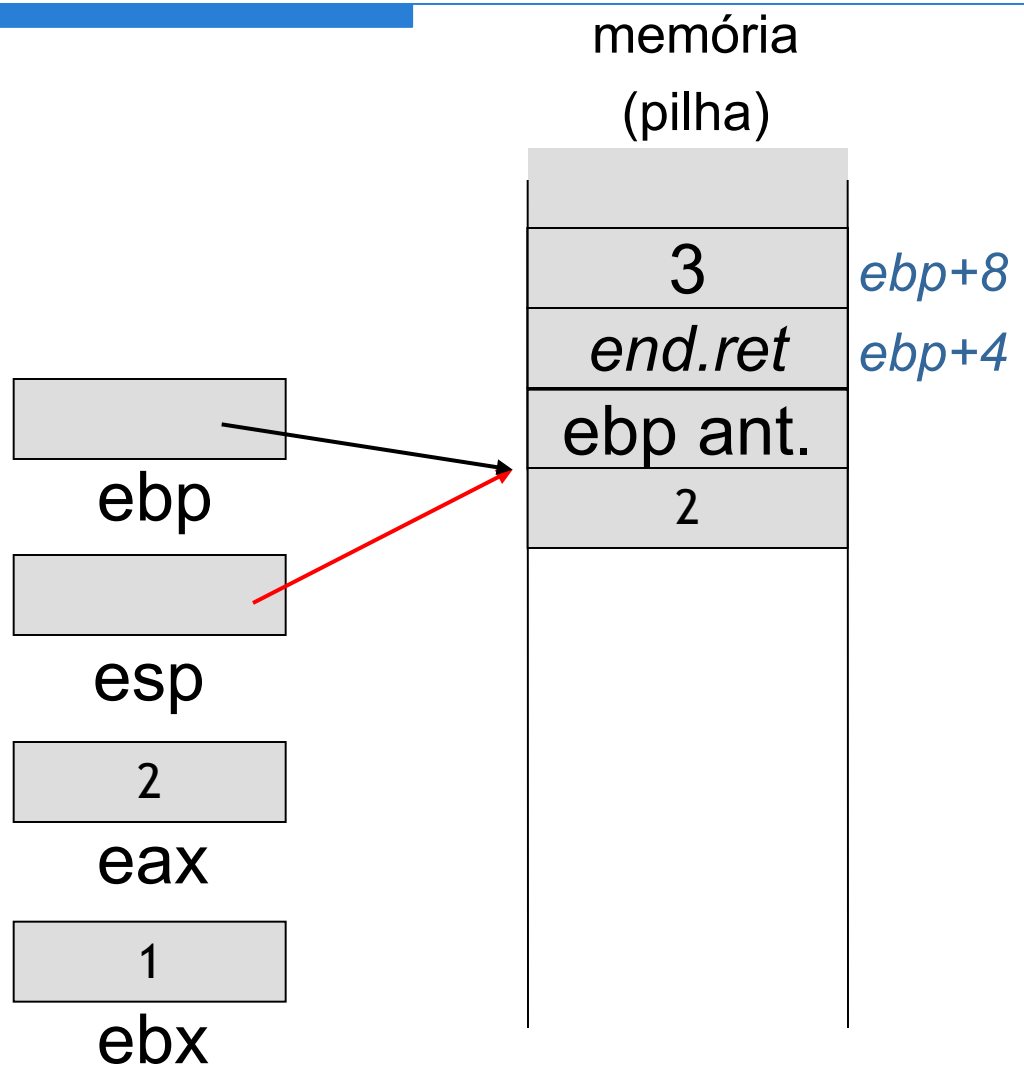
Exemplo: factorial recursivo

```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial
    add $4, %esp
    mull 8(ebp)
    jmp endfact
one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



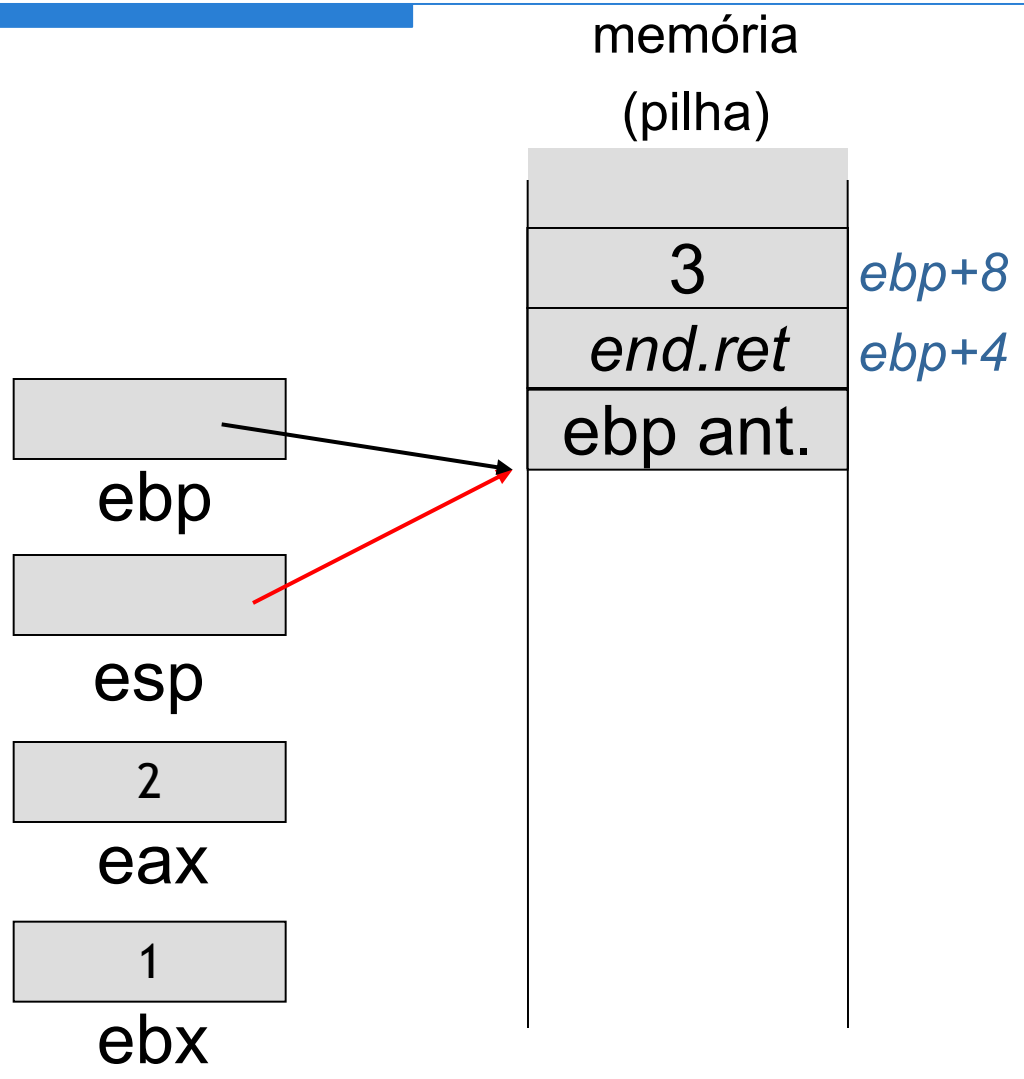
Exemplo: factorial recursivo

```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial
    add $4, %esp
    mull 8(ebp)
    jmp endfact
one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



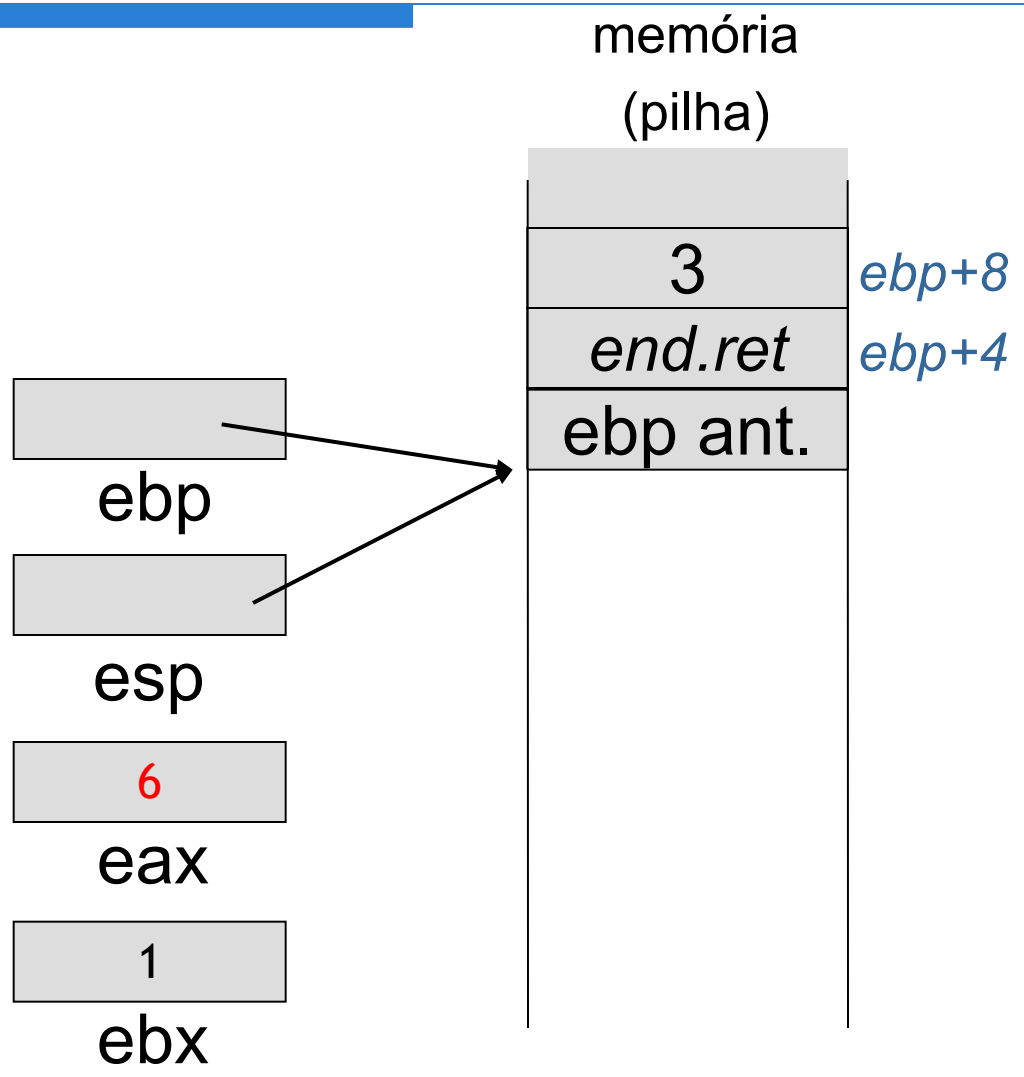
Exemplo: factorial recursivo

```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial
    add $4, %esp
    mull 8(ebp)
    jmp endfact
one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



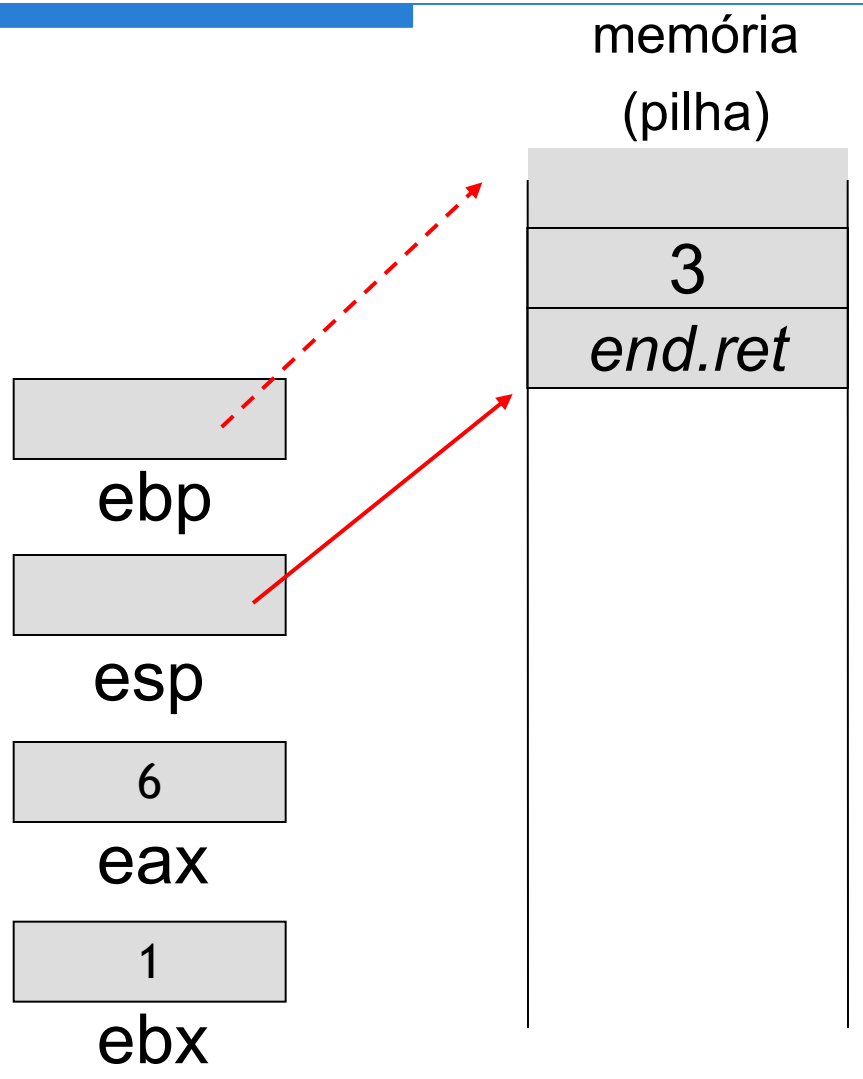
Exemplo: factorial recursivo

```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial
    add $4, %esp
    mull 8(ebp)
    jmp endfact
one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



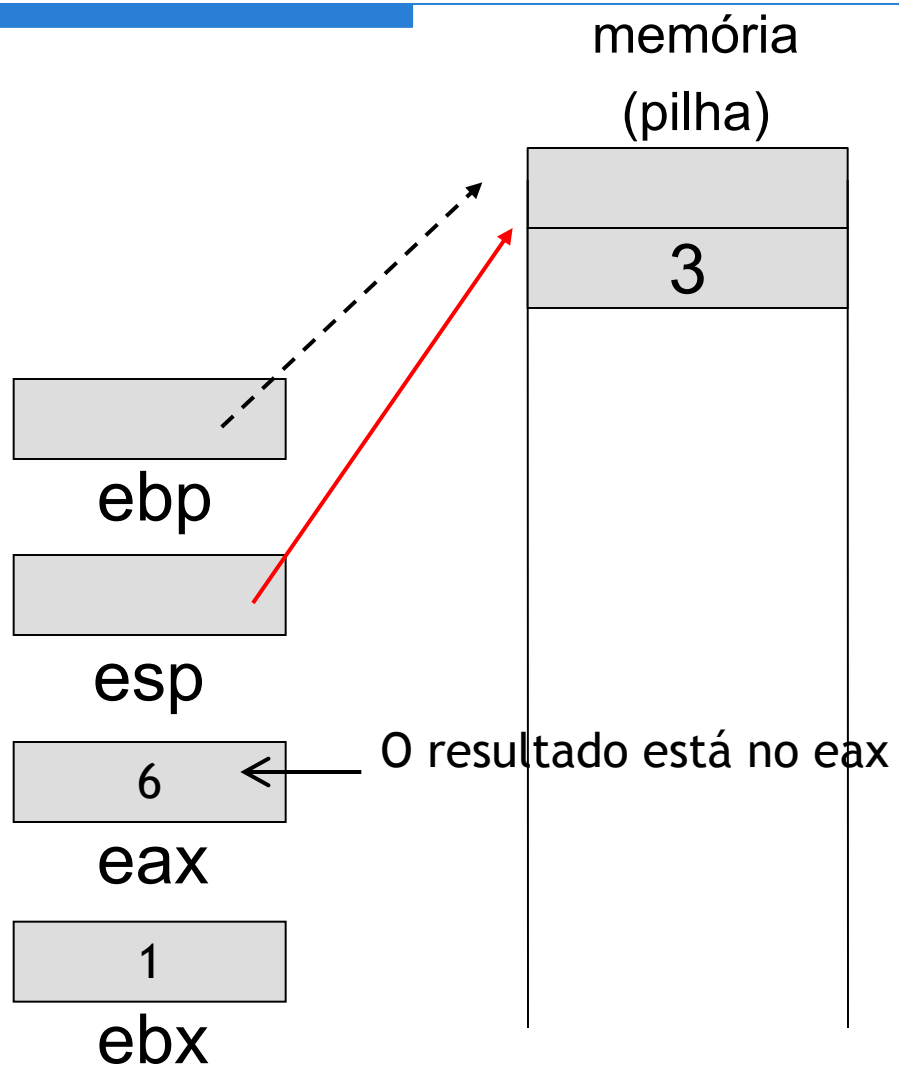
Exemplo: factorial recursivo

```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial
    add $4, %esp
    mull 8(ebp)
    jmp endfact
one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



Exemplo: factorial recursivo

```
factorial:
    push %ebp
    mov %esp, %ebp
    mov 8(ebp), %ebx
    cmp $1, %ebx
    jle one
    dec %ebx
    push %ebx
    call factorial
    add $4, %esp
    mull 8(ebp)
    jmp endfact
one:
    mov $1, %eax
endfact:
    pop %ebp
    ret
```



ABI – Application Binary Interface

- Definição de regras e protocolos binários seguidos na implementação das aplicações
  - p.e. por compiladores, ligadores, ...
- Inclui: formatos dos ficheiros objeto e bibliotecas, dimensões dos dados, uso dado aos registos, utilização da pilha, convenções de chamada de funções e retorno de resultados, convenções de chamada do SO e retorno de resultados, etc...
- Permite a interação entre código de diferentes origens (linguagens ou asm), de bibliotecas, assim como com o SO

ISA vs ABI vs API

- ISA – instruction set architectute
  - Define a interface com o hardware (e suas características)
- ABI – application binary interface
  - Define a interface entre módulos binários de software e entre estes e o Sistema de Operação
  - Permite portabilidade do binário/executável
- API – application programming interface
  - As interfaces ao nível da linguagem de programação entre módulos de software (p.e. funções exportadas por bibliotecas)
  - Permite portabilidade do código fonte por recompilação

Linux IA-32 ABI - Convenção de chamada e retorno

- Os argumentos são colocados na pilha, da “direita para a esquerda” (o primeiro argumento no C fica no topo da pilha);
- É efectuado o call;
- A função (subrotina) só pode alterar os registos gerais %eax, %ecx e %edx
  - para usar outros, tem de os salvar na pilha e repor o valor original antes do retorno (exemplo: %ebp)
- O resultado, se existe, fica em %eax.
- Depois do retorno remove-se os argumentos da pilha

Exemplo

```
int mySoma(int x,int y)  {
   int z=x+y;
   return z;
}
...
int a = mySoma(13, 4);
...
```

```
...
pushl $4
pushl $13
call mySoma
add $8, %esp
mov %eax, (a)
...
```

Chamada

```
mySoma:
   push %ebp
   mov  %esp,%ebp
   sub $4, %esp

   mov 12(%ebp),%eax
   add 8(%ebp),%eax
   mov %eax,-4(%ebp)

   mov %ebp, %esp
   pop %ebp
   ret
```

Entrada

Corpo

Saída

Passagem de referência (por valor/cópia)

- Se for empilhado o endereço da variável
  - na subrotina pode-se aceder por endereçamento indireto à própria variável
  - Nota: em C só existe passagem por valor mas este valor pode ser uma referência (um pointer)

```
void procA(int *x) {
        *x = 5;
}

int main(){
        int z = 2;
        procA(&z);
        printf("%d",z);
}
```

```
main:
        mov %esp, %ebp
        sub $4, %esp    # int z
        movl $2, -4(%ebp)
        lea -4(%ebp), %eax
        push %eax
        call procA
        add $4, %esp
        ...
```

Passagem de referência (por valor/cópia)

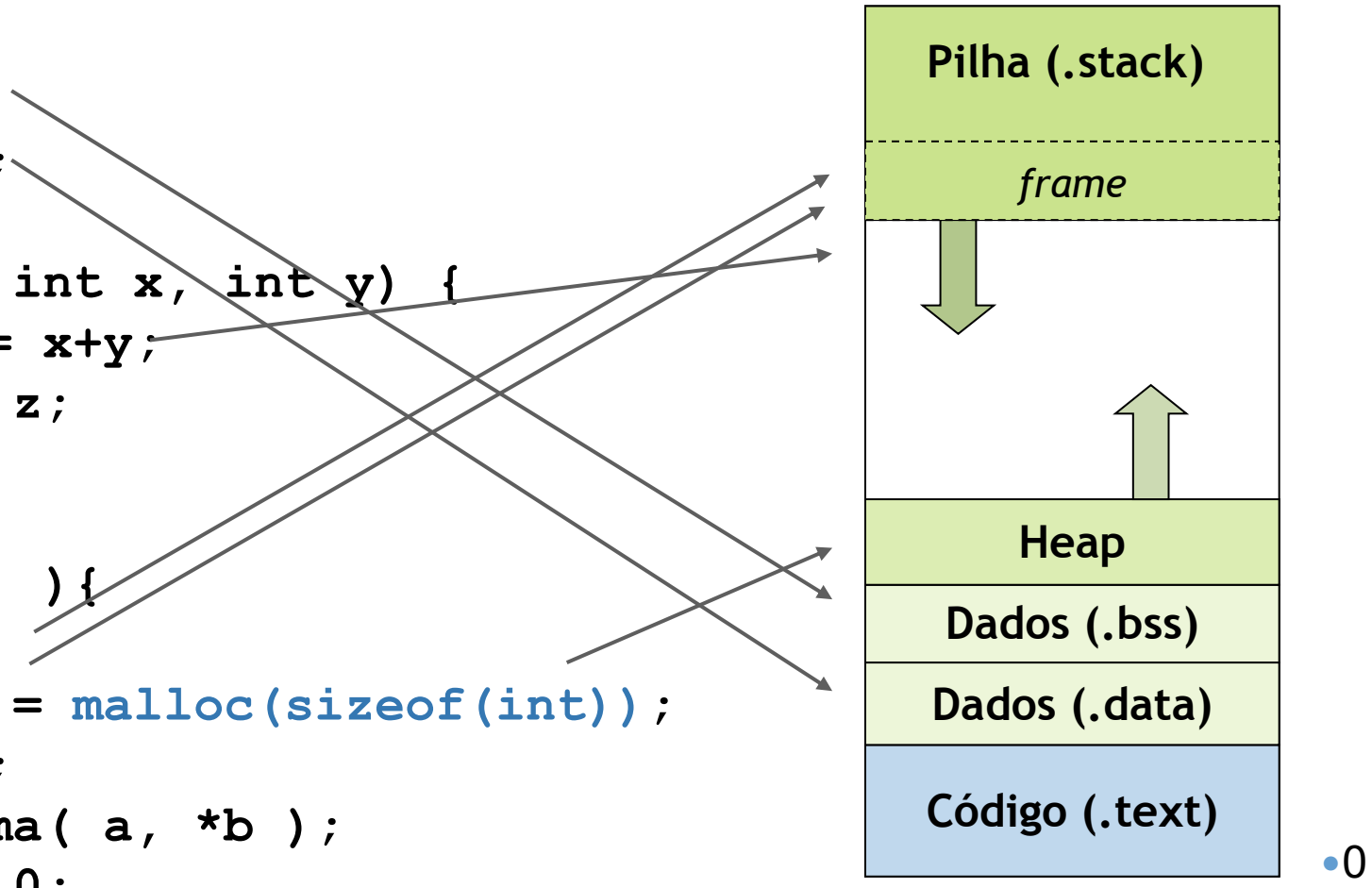
- Se for empilhado o endereço da variável
  - na subrotina pode-se aceder por endereçamento indireto à própria variável
  - Nota: em C só existe passagem por valor mas este valor pode ser uma referência (um pointer)

```
void procA(int *x) {
        *x = 5;
}

int main(){
        int z = 2;
        procA(&z);
        printf("%d",z);
}
```

```
procA:
        push %ebp
        mov %esp, %ebp
        mov 8(%ebp), %edx
        movl $5, (%edx)
        pop %ebp
        ret
```

Mapa de memória durante a execução

```
#include <stdio.h>

int x;
int y = 5;

int soma( int x, int y) {
    int z = x+y;
    return z;
}

int main(  ){
    int a;
    int *b = malloc(sizeof(int));
    *b = 4;
    a = soma( a, *b );
    return 0;
}
```



- 0

Ligação de C e Assembly

- Há que saber exatamente como é o protocolo de chamada usado pelo compilador (a ABI):
  - ordem pela qual os parâmetros são empilhados
  - dimensão de cada tipo de dados em C
  - onde fica o resultado da função para cada tipo de dados
- Os vários símbolos (identificadores) têm de ser conhecidos globalmente para que a ligação seja possível
  - Linker (ld) tem de resolver todos os símbolos

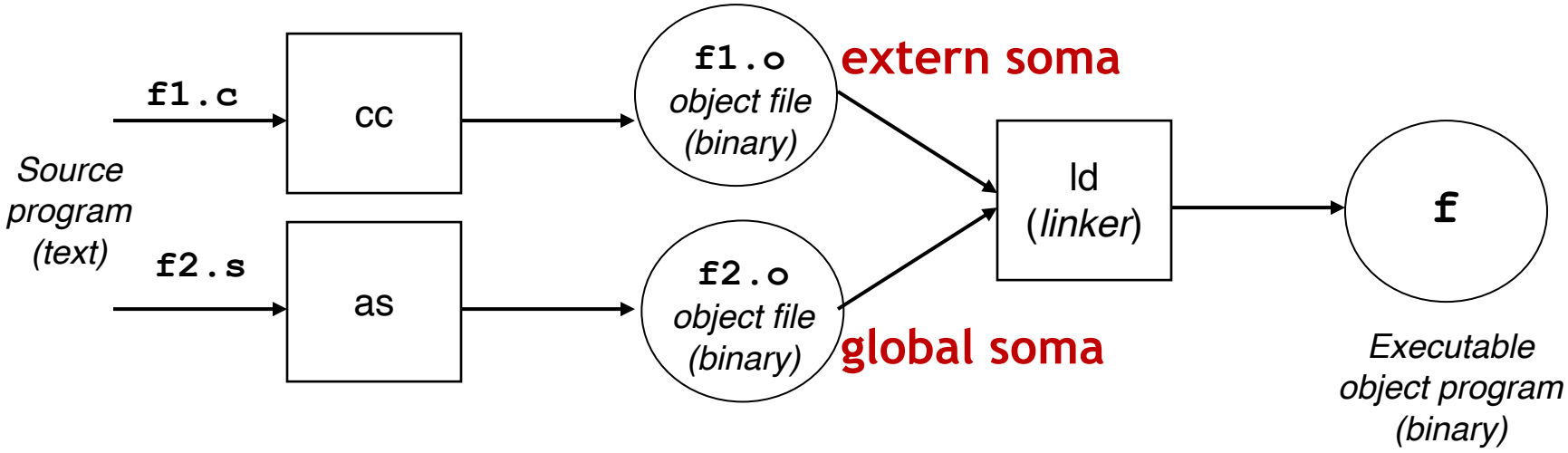
Ligação de C e Assembly – Símbolos públicos

```c
#include <stdio.h>

// implicito
extern
int soma(int a,int b);

int main(){
        int x, y;

        x = 6;
        y = soma(x,3);
        printf("%d\n", y);
        return 0;
}
```

```
.global soma
.text
soma:  push %ebp
       mov %esp, %ebp

       mov 8(%ebp),%eax
       add 12(%ebp),%eax

       pop  %ebp
       ret
```

Ligação de ficheiros “objeto”

- O ficheiro objecto (.o) tem código máquina, num formato independentemente da sua origem, C, Pascal, assembly, etc.
- Pode incluir referências a símbolos externos (usados mas não definidos)
- Indica os símbolos exportados (podem ser usados por outros)
- Na ligação (linking) todos os símbolos externos de um .o têm de ficar resolvidos pelos exportados por outros ficheiros .o



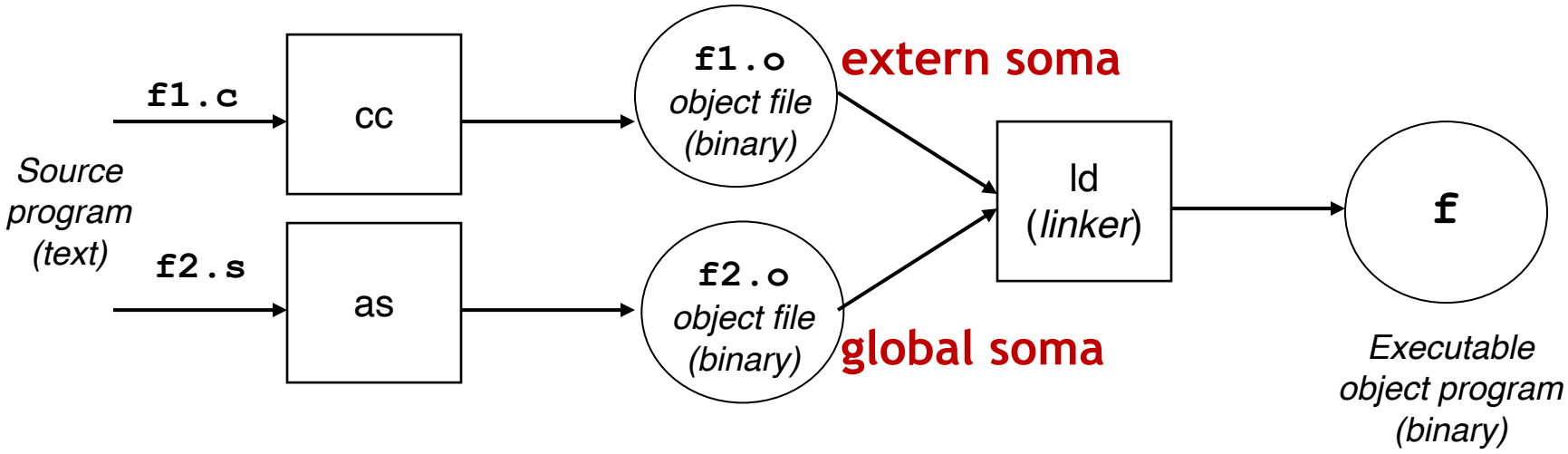
Ligação de ficheiros “objeto”

- No caso de ser só em C:

```
f1.c:
extern int myF2( float a );

int main( ) {
      int x = myF2( 3.6 );
      ...
}
```

```
f2.c:
int myF2( float a ) {
      ...
}
```

```
cc -o f  f1.c f2.c
```



Módulos em C

- Módulo f2 com header:

f1.c:

```
#include "f2.h"

int main(  ) {
      int x = myF2( 3.6 );
      ...
}
```

f2.h:

```
extern  int myF2( float a );
```

f2.c:

```
int myF2( float a ) {
      ...
}
```

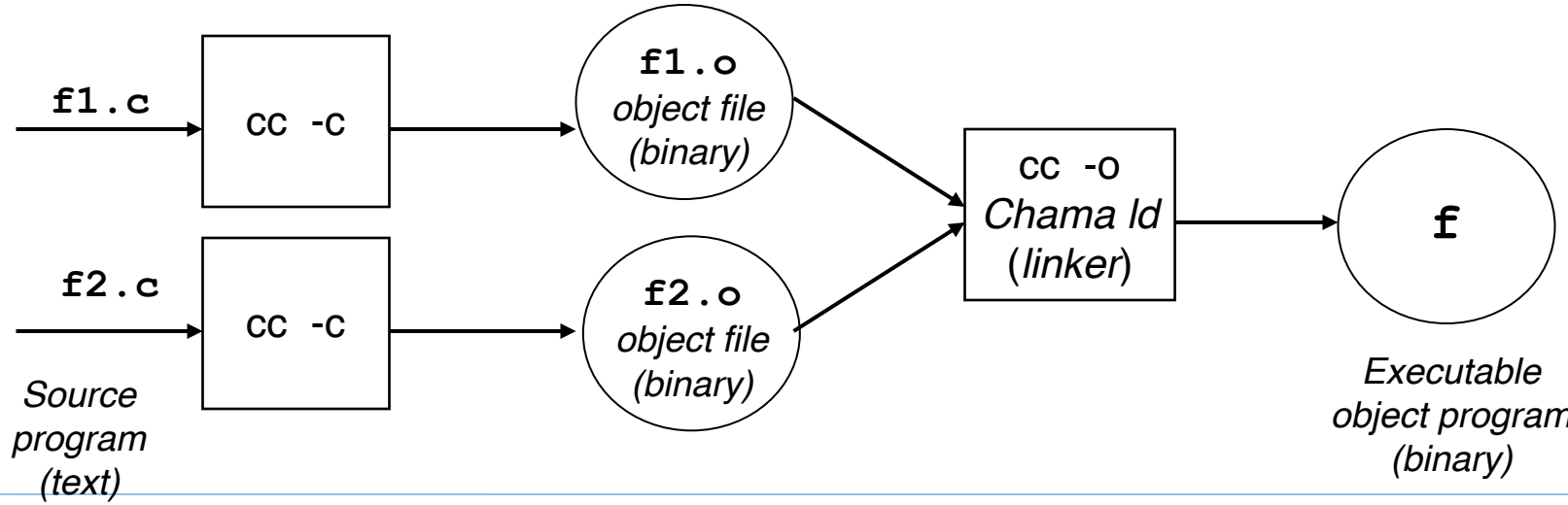
```
cc -o f  f1.c  f2.c
```

Compilação separada

- Podemos compilar em separado:

```
cc  -c   f2.c
cc  -o  f   f1.c   f2.o
```

```
cc -c   f1.c
cc -c   f2.c
cc -o  f    f1.o f2.o
```



Compilação separada C + assembly

```
as  -o  f2.o   f2.s
cc  -o f   f1.c  f2.o
```

```
cc -c   f1.c
as  -o f2.o   f2.s
cc -o  f    f1.o f2.o
```

- ou ainda: cc -o f   f1.c f2.s