Arquitetura de Computadores

MiEI – 2016/17

DI-FCT/UNL

Aula 9

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 …

    Mas há muitos modos de endereçar operandos em memória …

Modos de endereçar a memória (1)

- Muitos modos diferentes de indicar o endereço do operando
  - Endereçamento **direto**:

    o endereço efetivo está na instrução. Exemplos:

    mov (100), %eax

    mov var1, %eax
  - Endereçamento **indireto**:

    é obtido pelo CPU na execução da instrução com base em várias informações codificadas na instrução (pode envolver valores na instrução e o conteúdo de registos)

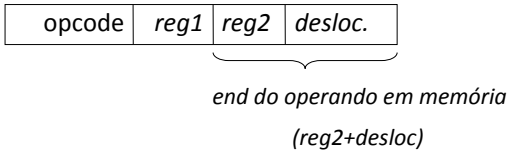
Modos de endereçar a memória (2)

- **Indireto por registo** *(visto antes)*
  - um registo contém o endereço na memória onde está o operando. Exemplo: mov (%ebx), %eax
- Outras variantes: envolvendo mais de um registo, deslocamentos e factores de escala
  - . . .
- *Endereçamento **indirecto por memória:***
  - *endereço efetivo encontra-se na memória, cujo endereço é o indicado na instrução*

  Não suportado na arquitetura IA-32

Endereçamento baseado ou indexado

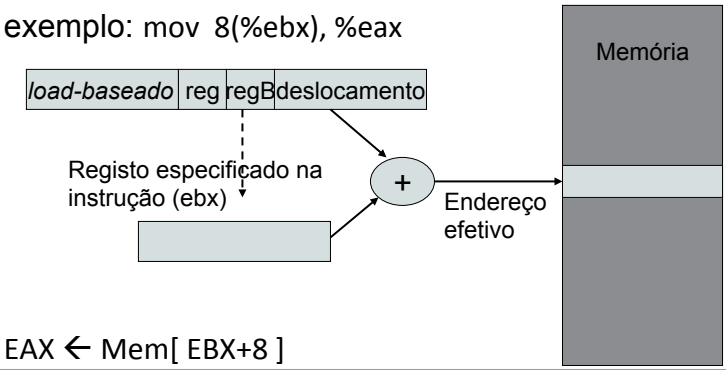
- Indireto com base e deslocamento (endereçamento baseado ou indexado)
  - um registo contém um endereço (base) e na instrução é indicado um deslocamento
  - Ou, a base é dada na instrução e o registo contém o deslocamento (índice)
  - exemplo: mov 8(%ebx), %eax



Endereçamento baseado ou indexado (2)

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

exemplo: mov 8(%ebx), %eax



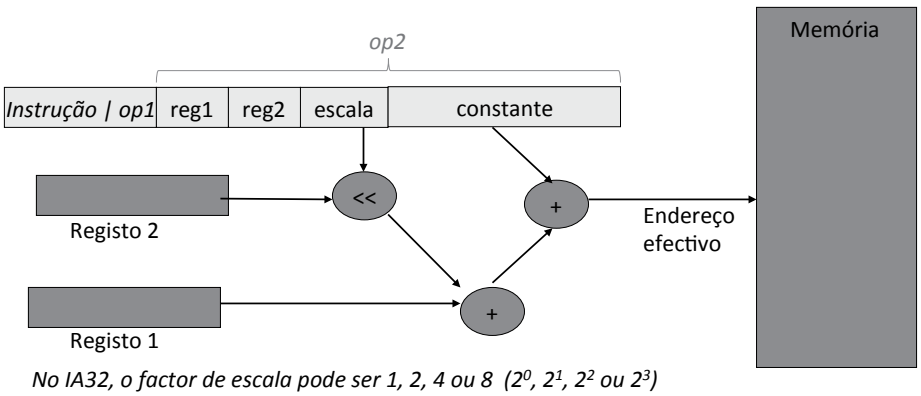
EAX ← Mem[EBX+8]

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)`

Endereçamento indireto de memória (2)

- 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/indexado
(Rb, Ri, S)	**`(%ebx, %ecx, 2)`**	
D(, Ri, S)	**`vet( ,%ecx, 4)`**	

Exemplos de cálculo do endereço efectivo

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

End. assembly	Cálculo	Endereço
`0x8(%edx)`	`0xf000 + 0x8`	`0xf008`
`(%edx, %ecx)`	`0xf000 + 0x100`	`0xf100`
`(%edx,%ecx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80( ,ecx,2)`	`0x80 + 2*0x100`	`0x280`

Endereçamento indexado

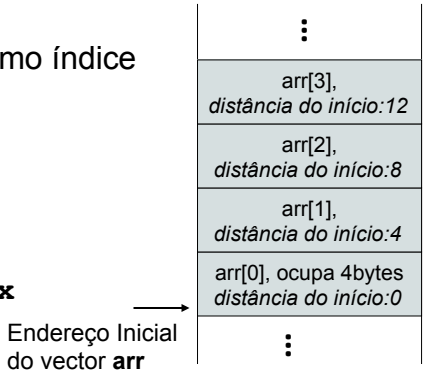
- Conveniente para endereçar vetores

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

Sabendo o endereço de **arr**,
Um registo pode funcionar como índice tomando os valores 0, 1, 2 …

Exemplo:

```
mov $arr, %eax
mov $0, %ecx
mov (%eax, %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, %cx
        mov $vet, %ebx
ciclo:  incl (%ebx)
        add $4, %ebx
        loop ciclo
        . . .
```

 # incrementar todos os ints de vet

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

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

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

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


- Como reexecutar código?
- Como implementar funções, etc...?

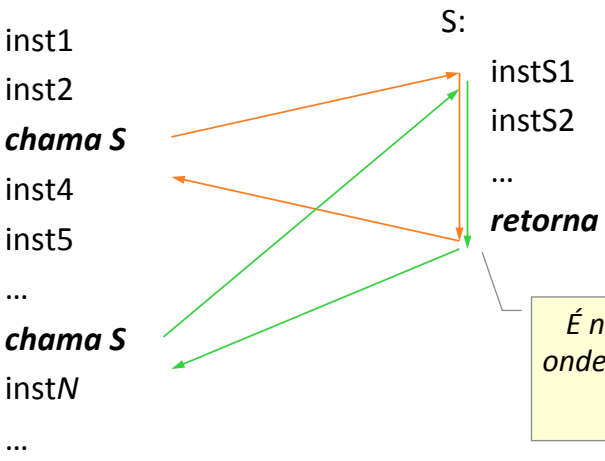
Subrotinas

- 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, da funcionalidade sobre a qual assentam as funções 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

Chamada e retorno



É necessário saber de onde foi feita a chamada (onde retornar)

chamar: CALL e retornar: RET

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

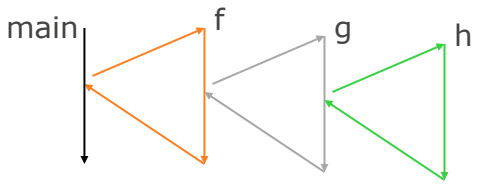
    Na chamada, guarda o EIP e depois salta

    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

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()

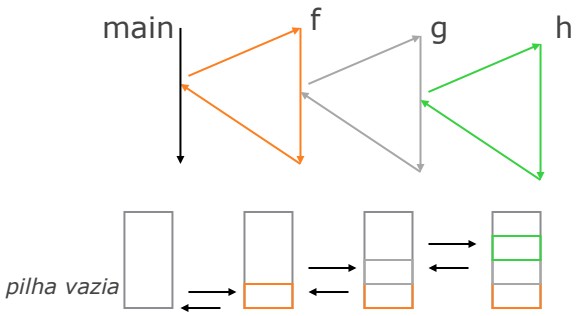


Salvar o endereço de retorno (2)

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

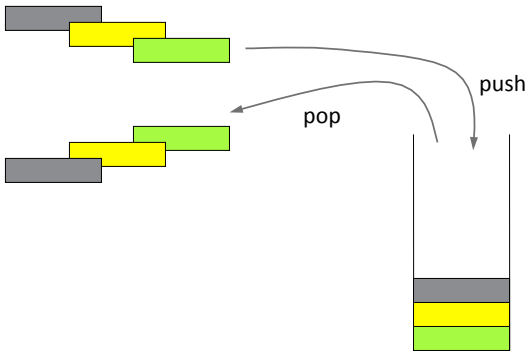
Salvar o endereço de retorno numa pilha

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



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!

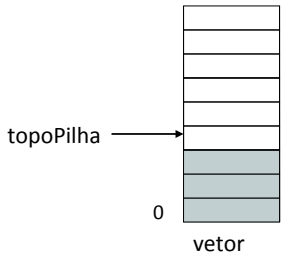


Implementação da Pilha

- Possível implementação sobre um vetor

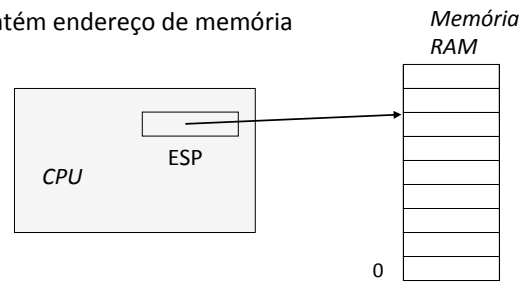
(sem verificar erros) :

```
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

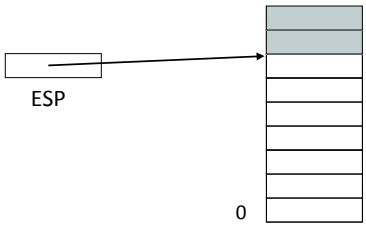


Estrutura de dados Pilha (stack)

- As instruções suportadas são: **Push** e **Pop**
  - Copiam valores entre CPU e memória como os MOV mas:
  - Usam um operando: o elemento a empilhar ou para onde desempilhar (16/32bits)
  - ESP é usado implicitamente para endereçamento indireto à memória
  - ESP é automaticamente ajustado em 2 ou 4 (depende do operando ser 2 ou 4 bytes): ESP é **decrementado** pelo push e **incrementado** pelo pop

Exemplos (32bits):

push %eax	ESP ← ESP-4
	Mem[ESP] ← EAX
pop %ebx	EBX ← Mem[ESP]
	ESP ← ESP+4



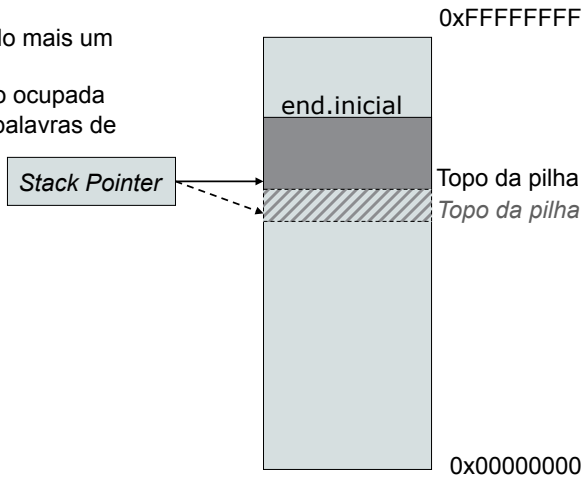
Operações sobre a pilha

Exemplo dos Intel de 32bits:
• 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*

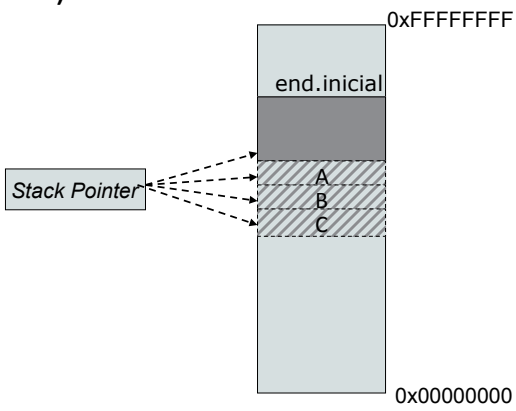
Pop *destino:*
destino← Mem[ESP]
ESP ← ESP + 4 *(ou 2)*



Exemplos

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

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



A zona da pilha (ou stack)

- Além do código e dos dados, um programa escrito numa linguagem imperativa como o C/C++ utiliza outra zona de memória, como **pilha** (ou *stack*).
  - A dimensão desta varia ao longo da execução do programa
  - 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

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

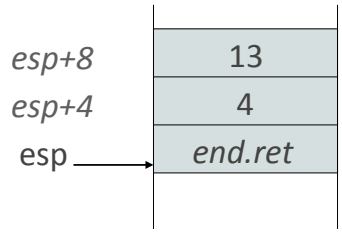
- Modo:
  - Por cópia do valor
  - Por referência ou endereço (apontador)
- Onde:
  - Em posições pré-definidas de memória?
  - Em registos do CPU?
  - Através da pilha de execução?
- E o resultado das funções?
  - Onde fica?

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
- Na chamada e na subrotina há que seguir as mesmas convenções:
  - Que parâmetros, sua dimensão, ordem destes na pilha, etc

Exemplo ilustrativo

pushl $13
pushl $4
call mySoma
…
…

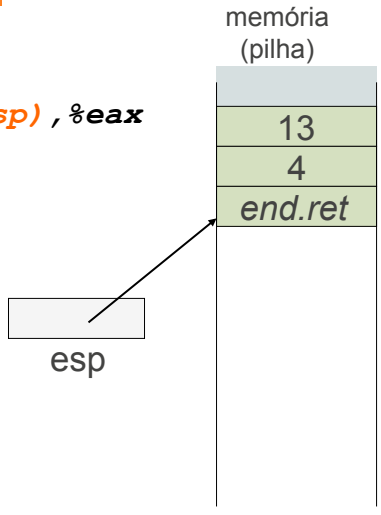


mySoma:

…
1º argumento → aceder ao endereço dado por esp+8
2º argumento → aceder ao endereço dado por esp+4 …

Exemplo

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



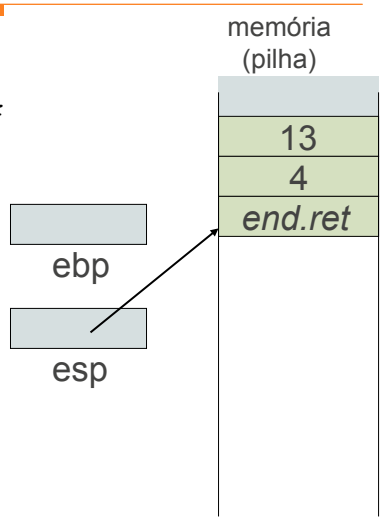
- Exemplo:
  - chamar a função mySoma passando 13 e 4
  - em mySoma queremos aceder ao parâmetro com o 13:
    Está em ESP+8

O registo "Frame Pointer"

- Usar o SP como registo base não é cómodo uma vez que podem ser feitos *pushs* e *pops* e assim a distância aos parâmetros variar
- 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 SP livre para 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 activação da subrotina***)
- No Pentium este registo chama-se EBP (Extended Base Pointer)

Exemplo

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

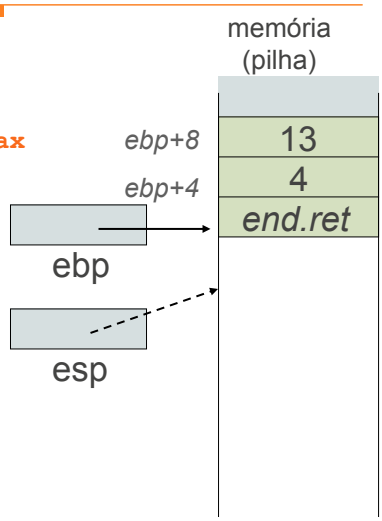


- Exemplo:
  - chamar a função mySoma passando 13 e 4
  - em mySoma queremos aceder ao parâmetro com o 13:

    Está em ESP+8

Exemplo – usar EBP

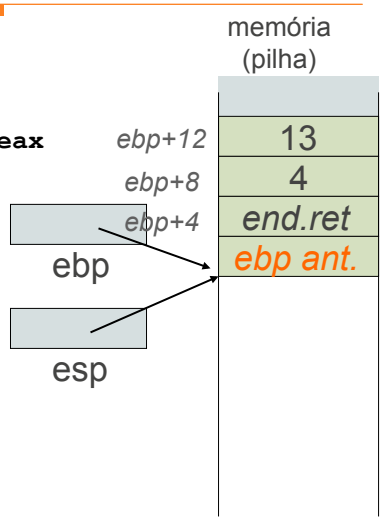
```
…                mySoma:
pushl $13        mov %esp, %ebp
pushl $4         …
call mySoma      mov 8(%ebp), %eax
…                …
                 ret
```



- 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

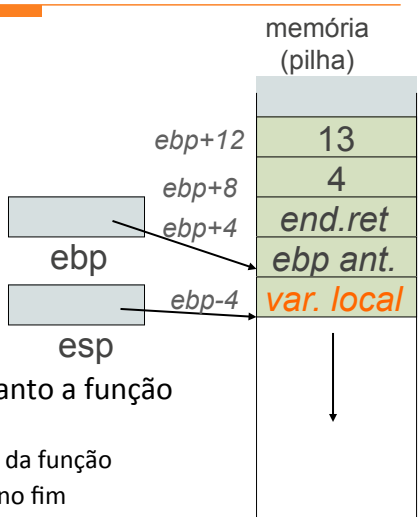
```
…                mySoma:push %ebp
pushl $13        mov %esp, %ebp
pushl $4         …
call mySoma      mov 12(%ebp),%eax
…                …
                 pop %ebp
                 ret
```



- 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 o procedimento termina, restaurar o EBP (a partir do *stack*) com o endereço anterior

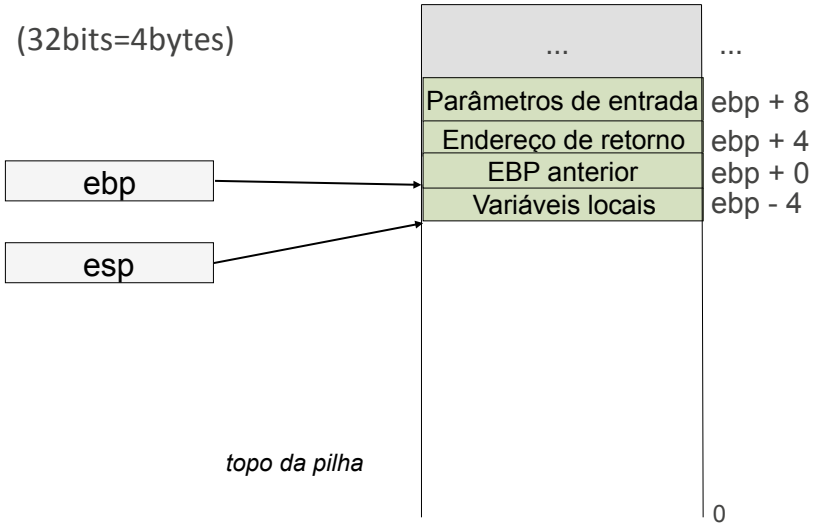
Exemplo – Variável locais

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

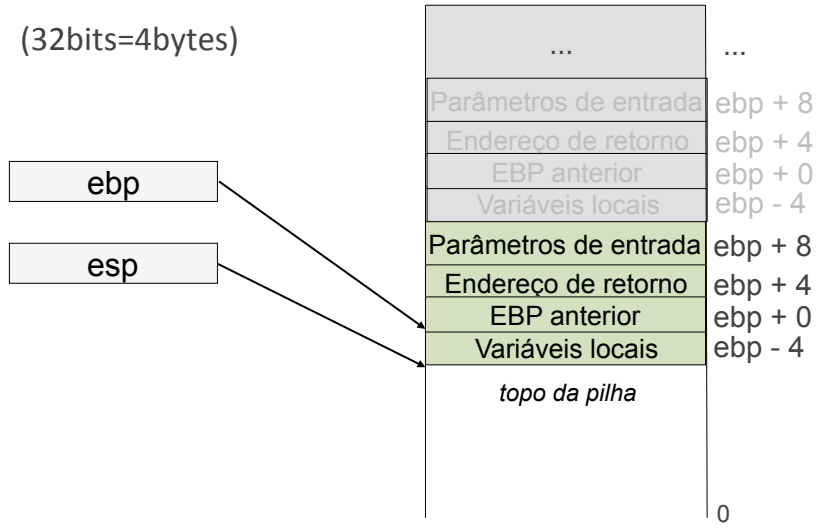


- As variáveis locais só existem enquanto a função executa
  - Var. local→zona de memória no *frame* da função
  - espaço reservado no início é libertado no fim

Frame de activação de subrotina [1]



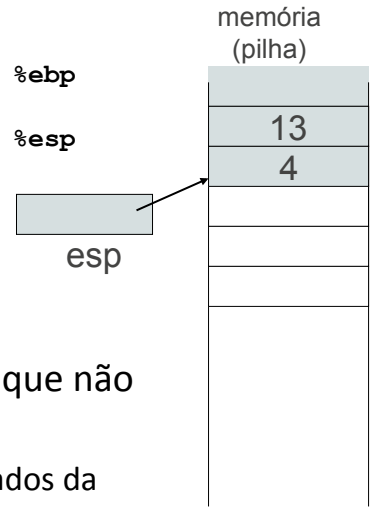
Frame de activação de subrotina [2]



Repor a pilha

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

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

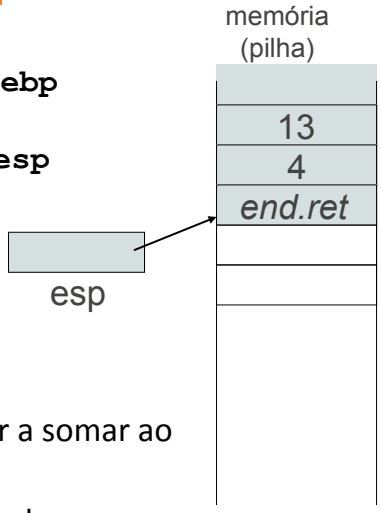


- A pilha deve ser reposta para que não vá sempre crescendo
  - os parâmetros têm de ser retirados da pilha

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 no Pentium)

Até agora:
secções relevantes do livro recomendado

- Computer Systems, a programmer's perspective, 2nd Ed. Bryant, O'Hallaron
  - [1 até 1.4] (e parte de 1.6 e 1.7)
  - [2 até 2.3.3] [2.3.6 até 2.4.3]
  - [3 até 3.5.4] [3.6 até 3.6.5] [3.7 até 3.7.4] [3.8 até 3.8.2] (unions em 3.9.2) [3.10]
  - *(parte de 4 até 4.1.3 - note que o exemplo do livro é um processador fictício semelhante ao Intel 32bits)*