Sub-rotinas

João Canas Ferreira

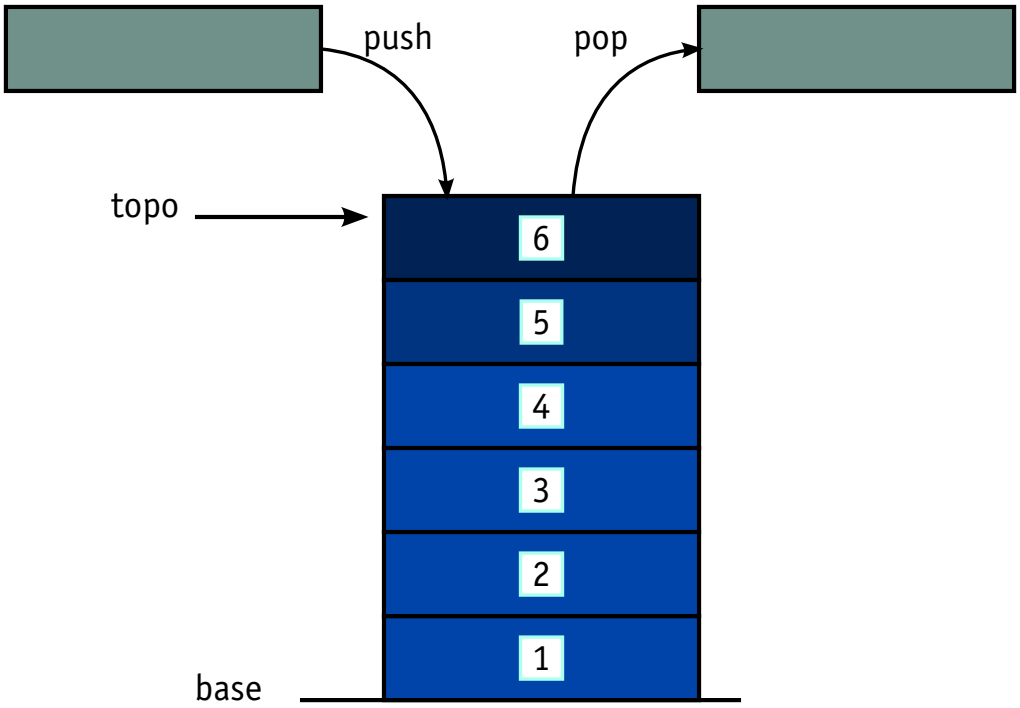
Março 2016

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Assuntos

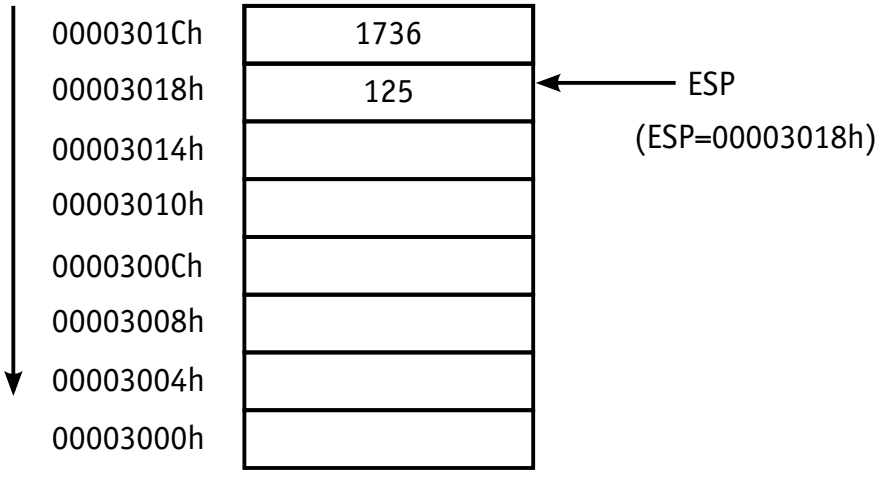
Pilha

- Durante a execução, os programas mantêm uma pilha de dados



Gestão da pilha

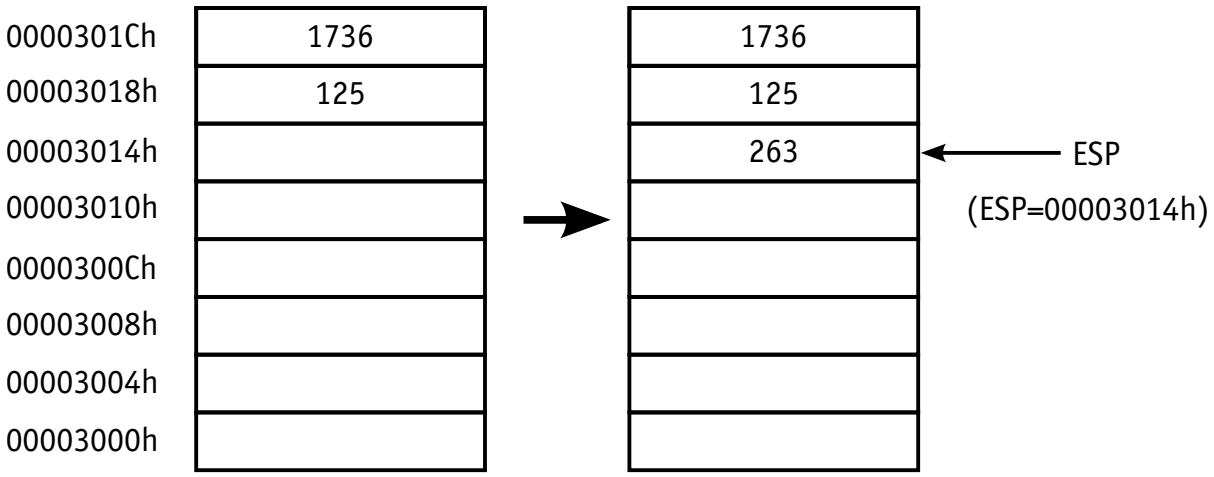
- Pilha gerida através de um apontador para o topo da pilha: ESP
- Pilha no segmento especificado pelo registo SS



- O que está na posição 00003014h ?

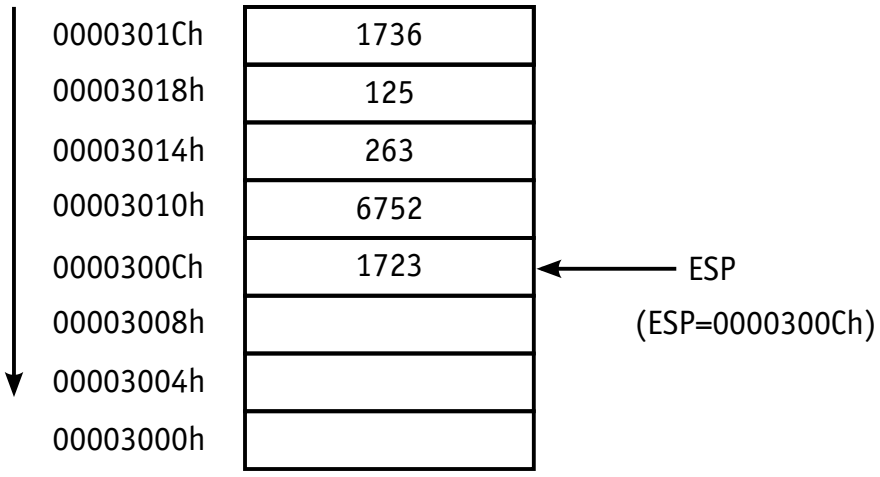
Colocar dados na pilha (1/2)

- Instrução `PUSH` : colocar um valor na pilha
- A pilha "cresce" de endereços maiores para menores
- Operando de 32 bits: decrementar ESP de 4 e copiar valor para o local apontado



Colocar dados na pilha (2/2)

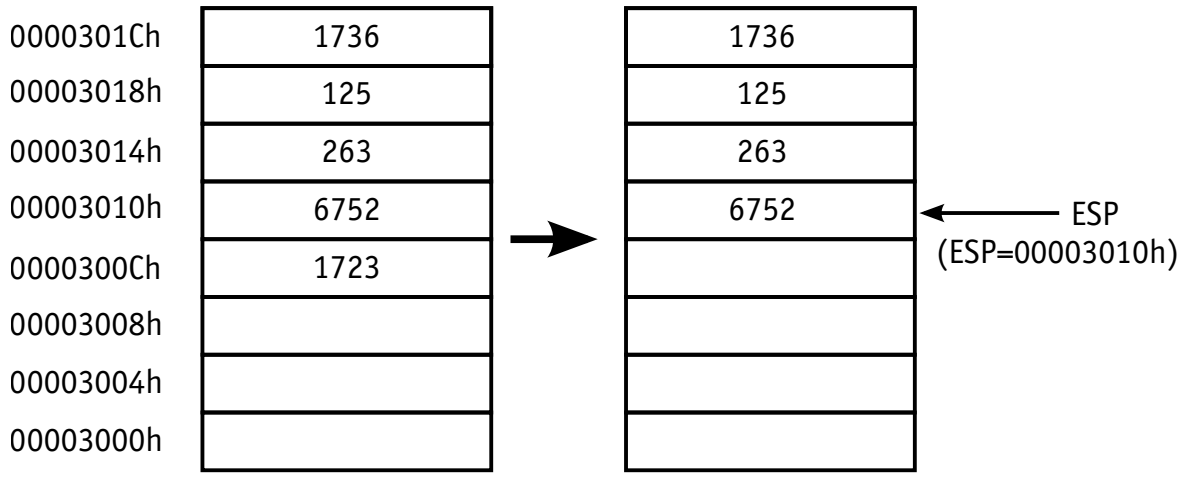
- Estado após colocar mais 2 inteiros na pilha:



- Memória abaixo de ESP considerada "vazia"

Retirar dados da pilha

- Instrução `POP`: retirar valor da pilha
- Copia valor apontado por ESP para registo ou variável
- Adiciona tamanho do item retirado (em bytes) a ESP



Instruções PUSH e POP

- Formato de PUSH
  - `PUSH reg/mem16`
  - `PUSH reg/mem32`
  - `PUSH imm32`
- Formato de POP
  - `POP reg/mem16`
  - `POP reg/mem32`
- A pilha pode ser usada para preservar valores temporariamente

```
push   eax
push   ecx
push   edx
; altera valores dos registos eax, ecx, edx
invoke printf, ...
pop    edx
pop    ecx
pop    eax
```

- Ordem de POP é a inversa de PUSH

Exemplo: ciclos aninhados

- Técnica: usar pilha para preservar o contador do ciclo exterior

```
    mov   ecx, 100     ; contador do ciclo exterior
L1:                    ; início do ciclo exterior
    push  ecx          ; guardar valor do contador
    mov   ecx,20       ; contador do ciclo interior
L2:                    ; início do ciclo interior
    ...
    loop  L2           ; repetir ciclo interior
    pop   ecx          ; repor contador do ciclo exterior
    loop  L1           ; repetir ciclo exterior
```


Outras instruções para gerir a pilha

- PUSHFD: copia EFLAGS para pilha
- POPFD: copia topo da pilha para EFLAGS
- PUSHAD: copia todos os registos para a pilha
  - ordem: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
- POPAD: copia valores da pilha para os registos
  - ordem inversa da anterior
- Versões de 16 bits:
  - PUSHF, POPF, PUSHAW, POPAW
- As instruções PUSHA e POPA podem ser interpretadas como instruções de 16 ou de 32 bits (depende da configuração).
  - Na configuração usada nas aulas, correspondem a instruções de 32 bits (i.e., PUSHA = PUSHAD, POPA=POPAD).

Assuntos

Decomposição funcional

- Programar também é gerir complexidade (da especificação e da implementação)
- A decomposição funcional envolve:
  - projetar programa antes de iniciar a codificação
  - decompor tarefas maiores em tarefas mais pequenas (sub-rotinas)
  - criar uma estrutura hierárquica de sub-rotinas
  - testar sub-rotinas individualmente
- A utilização de sub-rotinas é uma forma de *reutilização de código*
- Sub-rotinas podem ser:
  - procedimentos: a sua invocação não produz um valor
  - funções: a sua invocação produz um valor
- Em *assembly* não existe distinção formal entre procedimentos e funções: a designação usada é *procedure* (procedimento)
- CPU suporta sub-rotinas através de duas instruções: CALL e RET

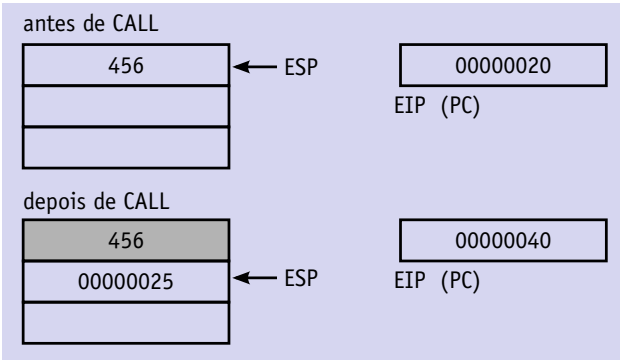
CALL e RET

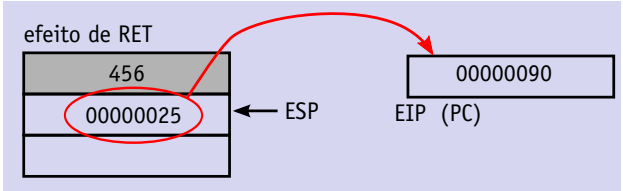
- Instrução CALL: invoca uma sub-rotina
  - coloca endereço da instrução a seguir à de CALL na pilha
  - salta para o início da sub-rotina (põe endereço de sub-rotina em EIP)
  - CALL é uma combinação especial de PUSH e JMP
- Instrução RET *retorna* de uma sub-rotina
  - o valor do topo da pilha é retirado para EIP
  - esse valor é o endereço de retorno colocado lá por CALL
  - tem efeito equivalente a saltar para a instrução a seguir ao CALL
- Formatos:
  - `CALL destino`
  - RET
- Declaração de uma sub-rotina

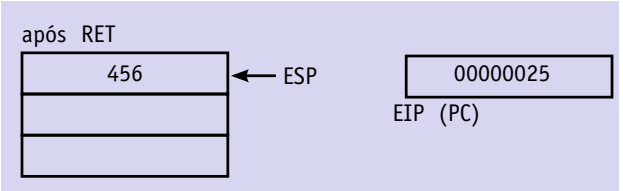
```
rotina    PROC
          ...
          ret
rotina    ENDP
```


Exemplo de CALL e RET

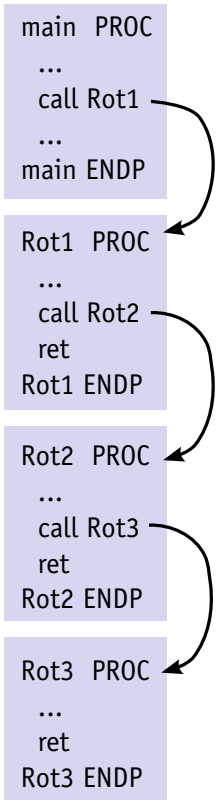
	endereço	*código*
	00000020	CALL rotina
	00000025	mov eax, ebx
		...
rotina		PROC
	00000040	mov ecx, 10
		...
	00000090	RET
rotina		ENDP



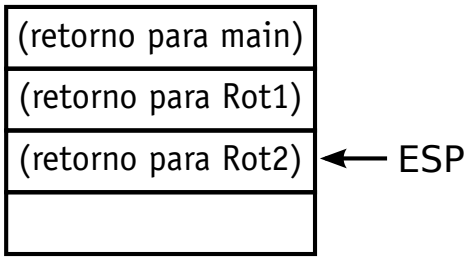




Invocações aninhadas



Pilha durante a execução de `Rot3`:



- A pilha contém três endereços de retorno
- Os endereços de retorno estão por ordem de invocação
- As rotinas terminam pela ordem inversa da invocação: `Rot3`, `Rot2`, `Rot1`

O que se segue...

1. Declaração de rotinas com parâmetros
2. Convenções de invocação
3. Passagem do resultado
4. Passagem de argumentos para uma sub-rotina
5. Acesso aos argumentos
6. Gestão dos registos (quais podem ser modificados?)
7. Variáveis locais (reserva de espaço e acesso)

Assuntos

Sub-rotinas com parâmetros

- Definição de sub-rotina com lista de parâmetros (separados por vírgula):

```
rotina PROC lista de parâmetros
...
rotina ENDP
```

- Parâmetro: `nome : tipo`
  - *nome:* identificador
  - *tipo:* tipo de dado (BYTE, WORD, etc.) ou apontador (PTR BYTE, etc.)
- Na sub-rotina, parâmetros são etiquetas de posições de memória:

```
dupl    PROC    x:  DWORD
        mov     eax,x
        add     eax,x
        ...
dupl    ENDP
```

- A diretiva OFFSET não pode ser aplicada a parâmetros.

Invocação de uma sub-rotina

- Para invocar uma sub-rotina usar a diretiva `invoke`
- Formato: `invoke nome_rotina, arg1, ..., argN`
- Diretiva verifica se número e tipo dos argumentos estão corretos
- Esta diretiva gera código para:
  1. colocar os parâmetros nas posições de memória corretas
  2. invocar a sub-rotina com `call nome_rotina`
- Uma sub-rotina deve ser *declarada antes de ser usada*:
  - pela definição
  - (ou) por um *protótipo* (declaração sem definição):

```
rot   PROTO arg1: tipo, ...  argN:tipo
```


Convenções de invocação de rotinas

- Sub-rotinas devem satisfazer requisitos de *interoperabilidade*
  - Sub-rotinas de diferentes origens usadas no mesmo programa
- Uma convenção de invocação de sub-rotinas define:
  1. localização dos argumentos
  2. tamanho dos argumentos
  3. maneira de retornar o resultado
  4. como é "libertada" a memória usada pelos argumentos
  5. formato do nome da sub-rotina
- Duas convenções usadas em Windows são:

  stdcall Usada para invocar sub-rotinas do sistema (p.ex., ExitProcess).

  C Usada em programas de linguagem C (p.ex., printf).

Convenção de invocação de rotinas "stdcall"

- Regras para convenções usadas pela Microsoft (`http://bit.ly/ZqbIlc`):
  - Argumentos são colocados na pilha da direita para a esquerda
  - Todos os argumentos são alargados para 32 bits (pelo menos)
  - No final da execução da sub-rotina, o resultado deve ficar num registo:
    - `EAX` resultados com 1-4 bytes
    - `EDX:EAX` resultados com 5-8 bytes
    - Resultados maiores são devolvidos ao invocador por outros métodos (apontador para zona "escondida")
  - A sub-rotina preserva os registos EBX, EBP, ESI e EDI
  - Após execução da sub-rotina, a pilha deve estar como antes da invocação.
  - A sub-rotina é responsável por remover os argumentos da pilha.

Exemplo

- Por omissão, as sub-rotinas usam a convenção *stdcall*.

```
; declaração de sub-rotina
dupl PROTO   x:DWORD

     .data
val  DWORD   100
msg1 BYTE    "Resultado: .code

main:
     invoke  dupl, val
     invoke  printf, offset msg1,
                     eax
     invoke  _getch
     invoke  ExitProcess, 0
```

```
; definição da rotina
dupl PROC    num:DWORD
     mov     eax, num
     add     eax, eax
     ; resultado em EAX
     ret
dupl ENDP
end main
```

invoke dupl, val ⇒

```
push val
call dupl
```

Convenção de invocação de rotinas "C"

- A convenção *C* é muito parecida com *stdcall*
  - *stdcall:* a sub-rotina é responsável por remover os argumentos da pilha. Argumentos são colocados na pilha da direita para a esquerda
  - *stdcall* não suporta sub-rotinas com número variável de argumentos (como printf). (Porquê?)
  - convenção *C*: o código que invoca a sub-rotina é responsável for "libertar" a memória usada pelos argumentos da sub-rotina invocada.
  - Nos restantes aspetos, as duas convenções são semelhantes.
- Definição de uma sub-rotina que usa a convenção *C*:

```
rotina PROC C arg1:DWORD, arg2:DWORD
```

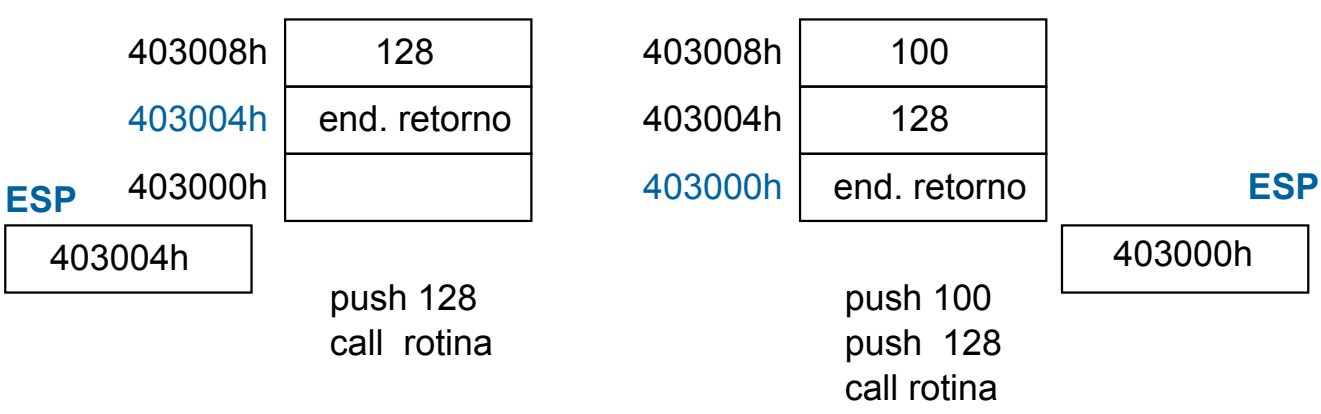
- A declaração é semelhante:

```
rotina PROTO C arg1:DWORD, arg2:DWORD
```

- Pode usar-se `rotina2 PROC STDCALL ...` para indicar explicitamente a convenção *stdcall*.

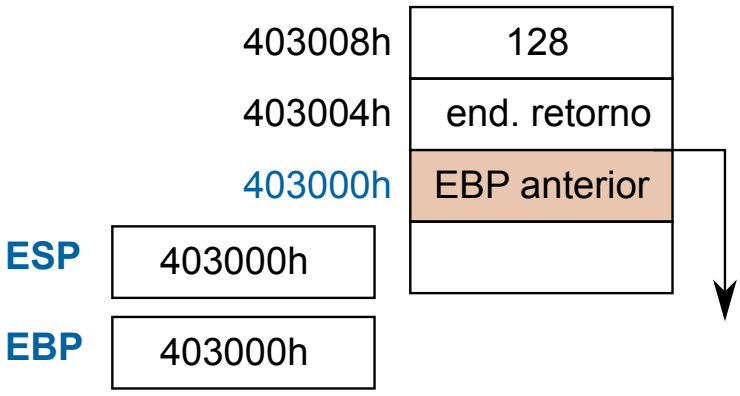
Acesso aos argumentos: o problema

- A posição dos argumentos em memória é determinada em *tempo de execução*.
- Para diferentes invocações da mesma função, argumentos podem estar em posições de memória *diferentes*.



Acesso aos argumentos: a solução

- Para aceder aos argumentos: acesso relativo a partir do topo da pilha.
- Problema: não é conveniente usar o ESP! (Porquê?)
- Convenção: Usar o registo EBP



Chamada:
push 128
call rotina

No início da rotina:
push EBP
mov EBP,ESP

No fim da rotina:
mov ESP, EBP
pop EBP

Exemplo de sub-rotina (stdcall)

```
dif   PROC    a:DWORD, b:DWORD
      mov     eax, a
      sub     eax, b
      ret
dif   ENDP
```

(azul) Prólogo

(magenta) Epílogo

```
dif:
      push    ebp
      mov     ebp, esp
      mov     eax, [ebp+8]
      sub     eax, [ebp+12]
      leave
      ret     8
dif   ENDP
```

- LEAVE = `mov esp, ebp` + `pop ebp`
- RET Num :
  - executa retorno
  - soma Num a ESP: ESP ← ESP+Num
  - Efeito: "liberta" espaço ocupado por argumentos (neste caso, 2 × 4 bytes)
- Convenção *stdcall:* rotina é responsável por "limpar" a pilha

Exemplo de sub-rotina (C)

```
dif   PROC C a:dword, b:dword
      mov      eax, a
      sub      eax, b
      ret
dif   ENDP
```

(azul) Prólogo

(magenta) Epílogo

```
dif:
      push     ebp
      mov      ebp, esp
      mov      eax, [ebp+8]
      add      eax, [ebp+12]
      leave
      ret
dif   ENDP
```

➠ Convenção *C:* código que invoca rotina é responsável por "limpar" a pilha

```
invoke dif, edi, esi
```

⇒

```
push esi
push edi
call dif
add  esp, 8
```


Preservação de registos

- Convenções de invocação exigem a preservação de alguns registos:
  - EBX, ESI, EDI e EBP
- Técnica mais comum: guardar os registos na pilha

```
rotX PROC USES EDI ESI a:DWORD
     xor      EDI, EDI
     mov      ESI, 0
     ...
     ret
rotX ENDP
```

```
rotX:
      push     ebp
      mov      ebp, esp
      push     edi
      push     esi
      ...
      pop      esi
      pop      edi
      leave
      ret      4
```

Assuntos

Moldura de uma invocação

- A moldura de uma invocação (registo de ativação) é uma área da pilha usada para:
  - argumentos da invocação
  - endereço de retorno
  - registos preservados
  - variáveis locais
- Cada invocação de uma sub-rotina cria uma moldura (“stack frame”)
- A moldura é “destruída” no fim da execução da invocação da sub-rotina
- Num dado instante, a pilha contém as molduras de todas as invocações de sub-rotina em curso
- Apenas a execução correspondente à moldura situada mais perto do topo da pilha está ativa
  - As demais execuções estão “suspensas”
- EBP é designado por *frame pointer*

Alinhamento da pilha

- Instruções PUSH e POP só suportam operandos de 16 e 32 bits
  - 16 bits: 2 bytes
  - 32 bits: 4 bytes
- Windows API *(Application Programming Interface)* requer que todos os elementos da pilha comecem em endereços que são múltiplos de 4
- Conclusão: colocar na pilha apenas dados cujo tamanho (em bytes) seja múltiplo de 4.

```
dif  PROTO a:BYTE, b:BYTE
     .data
valA byte 10
valB byte 7
     .code
     invoke dif, valA, valB
```

⇒

```
mov  al, valB
push eax
mov  al, valA
push eax
call dif
```


Variáveis locais: Definição e utilização

- Diretiva `LOCAL`, usada apenas no início de sub-rotinas
- Formato: `LOCAL lista_de_variáveis`
- Exemplo 1:

```
rotina PROC
  LOCAL var1:  BYTE, var2:  SWORD, var3:  SDWORD
```

- Exemplo 2:

```
rotina PROC
  LOCAL valores[10]:  SDWORD ; vector de 10 elementos
  LOCAL tmp:  SDWORD
```

- Utilização:

```
mov eax, tmp
mov ebx, valores[8]
mov ebx, valores[edx]
mov ebx, valores[edx+4]
```

- Endereços não podem ser obtidos com OFFSET (Porquê?)

Acesso a variáveis locais

- As variáveis locais ficam na moldura, abaixo da posição onde foi guardado o valor de EBP.
- O espaço é "reservado" subtraindo ao valor de ESP o número de bytes total das variáveis locais.
- O acesso é feito por deslocamentos *negativos* em relação a EBP.

```
rotx PROC a:DWORD
LOCAL valores[20]:DWORD
LOCAL tmp:DWORD
  mov   valores[0], 10
  mov   eax, a
  ...
```

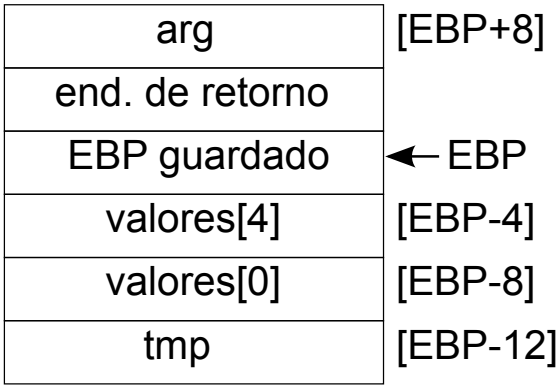
⇒

```
rotx:
  push  ebp
  mov   ebp, esp
  sub   esp, 84
  mov   dword ptr [ebp-80],10
  mov   eax, [ebp+8]
  ...
```


Moldura com variáveis locais

Moldura com 12 bytes de variáveis locais

```
rotx PROC arg:DWORD
LOCAL valores[2]:DWORD
LOCAL tmp:DWORD
     ...
```



- Variáveis locais podem ter tamanhos diferentes
- Atribuição de posição evitando sobreposição de acordo com tamanho:
  - 1 byte: próximo byte livre
  - 2 bytes: próxima posição de endereço par
  - 4 bytes: próxima posição de endereço múltiplo de 4

Moldura completa

Moldura com argumentos, preservação de registos e variáveis locais

```
roty PROC USES ESI EDI arg:DWORD
LOCAL valores[2]:DWORD
     ...
     ret
roty ENDP
```

```
push ebp
mov ebp, esp
add esp, -8
push esi
push edi
...
pop edi
pop esi
leave
ret 4
```

Na sub-rotina	Código gerado
`mov eax, arg`	`mov eax, [ebp+8]`
`mov valores[4], eax`	`mov [ebp-4], eax`

Como obter o endereço de variáveis locais?

- Endereço de variáveis locais e de argumentos só pode ser obtido em *tempo de execução*
- A instrução LEA produz endereço (deslocamento) de operandos diretos e indiretos
  - `LEA = load effective address`
- Formato: `LEA reg, mem`
- O operando `reg` fica com o endereço de `mem`, não o conteúdo

Exemplo 1: variável local em [ebp-8]

- Como inicializar `ESI` com o endereço da variável local?
- Solução: `lea esi, [ebp-8]`
- Errado: `mov esi, OFFSET [ebp-8]`

Exemplo 2

```
rotina PROC
    LOCAL vector[20]:DWORD
    lea esi, vector[0]
```