

Introdução à Software Básico: Introdução a Assembly IA-32

Departamento de Ciência da Computação
Instituto de Ciências Exatas
Universidade de Brasília

Sumário

- 1 Funções e Procedimentos em Assembly
- 2 Instrução Call e Ret
- 3 Passando Parâmetros

Funções em Assembly

- Assembly não implementa explicitamente passagem de parâmetros nem retorno de valores
- Blocos de Código chamados e executados são então procedimentos
- Instruções de chamada de procedimentos:
 - *call proc_name*: transfere a execução do programa ao bloco de instruções chamado *proc_name*
 - *ret*: retorna a execução do programa a instrução seguinte do ponto em que o bloco foi chamado.

```
%include "io.mac"
.DATA
prompt_msg1 db "Please input the first number: ",0
prompt_msg2 db "Please input the second number: ",0
sum_msg     db "The sum is ",0
.CODE
.STARTUP
PutStr prompt_msg1 ; request first number
GetInt CX          ; CX = first number
PutStr prompt_msg2 ; request second number
GetInt DX          ; DX = second number
call sum           ; returns sum in AX
PutStr sum_msg     ; display sum
PutInt AX
nwln
done:
.EXIT
;Procedure sum receives two integers in CX and DX.
;The sum of the two integers is returned in AX.
sum:
    mov AX,CX      ; sum = first number
    add AX,DX      ; sum = sum + second number
    ret
```

Instrução CALL

- A instrução call consiste nos seguintes passos:
 - $ESP = ESP - 4$; push return address onto the stack
 - $[SS:ESP] = EIP$
 - $EIP = EIP + \text{relative displacement}$; update EIP to point to the procedure
- A pilha do sistema é utilizada para guardar o endereço da instrução seguinte ao call
 - Esse endereço é necessário para o correto retorno da execução

Instrução RET

- A instrução de retorno é utilizada para recuperar o ponto de execução antes da chamada ao procedimento
- Consiste nos seguintes passos:
 - $EIP = [SS:ESP]$; pop return address at Top Of Stack into IP
 - $ESP = ESP + 4$; update TOS by adding 4 to ESP
 - Pode levar um operando opcional para adicionar esse valor para o ESP

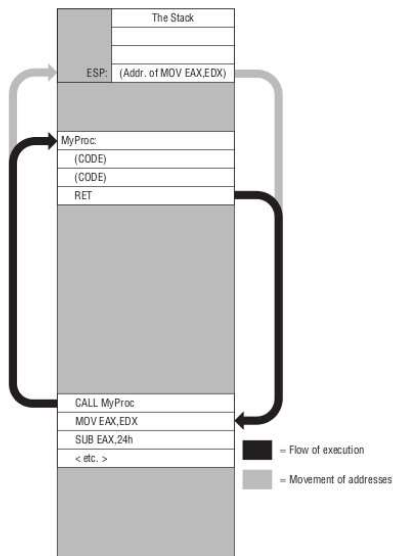


Figura: CALL e RET

Passando Parâmetros

- Os parâmetros são passados do programa que chama ao procedimento por meio de um local de acesso comum:
 - Registradores
 - Pilha
- O retorno de valores também ocorre das duas maneiras acima

Usando os Registradores

- Os parâmetros e os valores de retorno são passados via registradores de uso geral (EAX, EBX, ECX, EDX)
- Vantagem: Método rápido, conveniente para poucos parâmetros
- Desvantagem: Limita o número de parâmetros. Como os registradores são empregados durante a execução do procedimento, pode ser necessário salvar os valores dos parâmetros, geralmente na pilha

```
%include "io.mac"
.DATA
prompt_msg1 db "Please input the first number: ",0
prompt_msg2 db "Please input the second number: ",0
sum_msg      db "The sum is ",0
.CODE
.STARTUP
PutStr prompt_msg1 ; request first number
GetInt CX          ; CX = first number
PutStr prompt_msg2 ; request second number
GetInt DX          ; DX = second number
call sum           ; returns sum in AX
PutStr sum_msg     ; display sum
PutInt AX
nwln
done:
.EXIT
;Procedure sum receives two integers in CX and DX.
;The sum of the two integers is returned in AX.
sum:
    mov  AX,CX      ; sum = first number
    add  AX,DX      ; sum = sum + second number
    ret
```

Usando a Pilha

- Os parâmetros e os valores de retorno são passados via pilha do sistema:
 - Parâmetros são colocados na pilha antes de chamar o procedimento
 - Os parâmetros são utilizados pelo procedimento
 - Ao retornar, os valores ainda na pilha são desprezados, e valores de retorno são colocados na pilha
 - Após a chamada da função, os valores de retorno são retirados da pilha

Usando a Pilha

- Memória usada para pilha:
 - Registradores SS (segmento) e ESP (offset)
 - Somente words (16 bits) e double words (32 bits) podem ser colocadas na pilha
- Offset (ESP) aumenta do maior valor para o menor
 - Acrescentar um valor na pilha causa subtração no valor de ESP
 - *Stack underflow*: retirar algo da pilha vazia
 - *Stack overflow*: inserir algo na pilha cheia (não tem mais memória reservada para a pilha, o OFFSET gera um endereço menor que o SS)

| Parameter | What it controls | Recommended minimum value |
|---------------|---|---------------------------|
| maxdsiz | Maximum size of the data segment for 32 -bit processes | 1073741824 (1GB) |
| maxdsiz_64bit | Maximum size of the data segment for 64 -bit processes | 1073741824 (1GB) |
| maxssiz | Maximum size of the stack segment for 32 -bit processes | 8388608 (8MB) |
| maxssiz_64bit | Maximum size of the stack segment for 64 -bit processes | 8388608 (8MB) |

Figura: Setup do Kernel

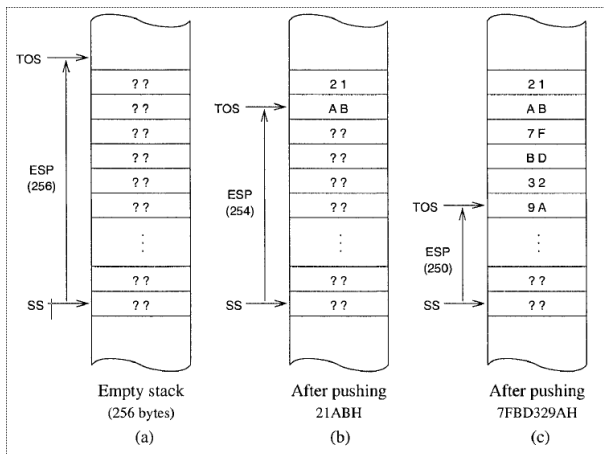


Figura: Exemplo de uso de Pilha (endereços menores abaixo)

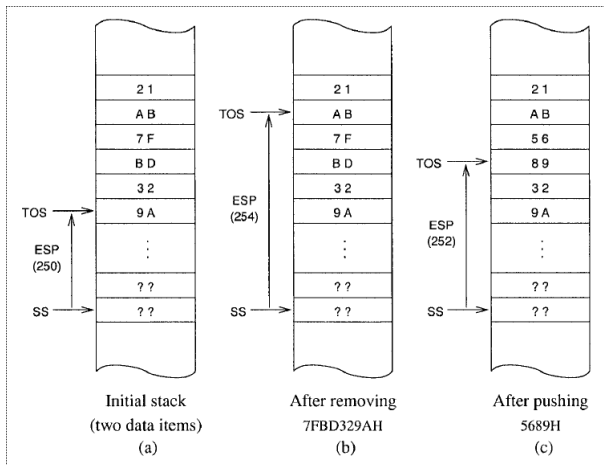


Figura: Exemplo de uso de Pilha (endereços menores abaixo)

Table 11.1 Stack operations on 16- and 32-bit data

| | | | |
|------|----------|--|---|
| push | source16 | $ESP = ESP - 2$ $SS:ESP = source16$ | ESP is first decremented by 2 to modify TOS. Then the 16-bit data from <code>source16</code> is copied onto the stack at the new TOS. The stack expands by 2 bytes. |
| push | source32 | $ESP = ESP - 4$ $SS:ESP = source32$ | ESP is first decremented by 4 to modify TOS. Then the 32-bit data from <code>source32</code> is copied onto the stack at the new TOS. The stack expands by 4 bytes. |
| pop | dest16 | $dest16 = SS:ESP$ $ESP = ESP + 2$ | The data item located at TOS is copied to <code>dest16</code> . Then ESP is incremented by 2 to update TOS. The stack shrinks by 2 bytes. |
| pop | dest32 | $dest32 = SS:ESP$ $ESP = ESP + 4$ | The data item located at TOS is copied to <code>dest32</code> . Then ESP is incremented by 4 to update TOS. The stack shrinks by 4 bytes. |

Figura: Exemplo de uso de Pilha

Instruções Especiais

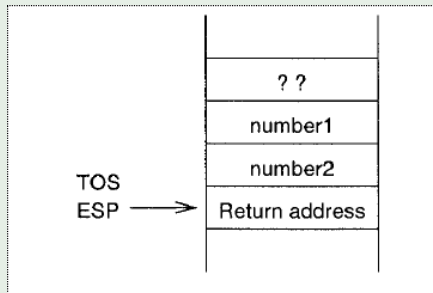
- Registrador de Status (flags):
 - *pushfd*
 - *popfd*
- Todos os registradores de uso geral:
 - *pusha* (EAX, ECX, EDX, EBX, ESP, EBP, ESI e EDI, nessa ordem)
 - *popa*

Parâmetros na pilha antes de chamar o procedimento

```
%include "io.mac"
.DATA
prompt_msg1 db "Please input the first number: ",0
prompt_msg2 db "Please input the second number: ",0
sum_msg db "The sum is ",0
.UDATA
number1 resw 1
number2 resw 1
result resw 1
.CODE
.STARTUP
PutStr prompt_msg1
GetInt number1
PutStr prompt_msg2
GetInt number2
push WORD [number1]
push WORD [number2]
call sum
PutStr sum_msg
PutInt [result]
nwn
```

Parâmetros na pilha antes de chamar o procedimento

```
%include "io.mac"
.DATA
prompt_msg1 db "Please input the first number: ",0
prompt_msg2 db "Please input the second number: ",0
sum_msg db "The sum is ",0
.UDATA
number1 resw 1
number2 resw 1
result resw 1
.CODE
.STARTUP
PutStr prompt_msg1
GetInt number1
PutStr prompt_msg2
GetInt number2
push WORD [number1]
push WORD [number2]
call sum
PutStr sum_msg
PutInt [result]
nwl
```



Acessando os parâmetros na pilha

sum:

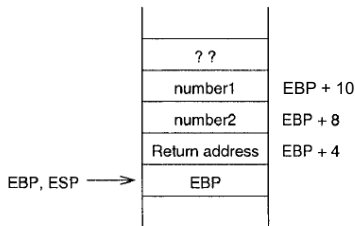
```
push EBP      ; save EBP value
mov EBP,ESP   ; copy TOS into EBP
push AX
mov AX,[EBP+10] ;sum = first number
add AX,[EBP+8] ;sum = sum+second number
mov [result],AX
pop AX
pop EBP
ret 4
```

Figura: Exemplo de uso de Pilha

Acessando os parâmetros na pilha

sum:

```
push EBP      ; save EBP
mov EBP,ESP   ; copy ESP to EBP
push AX
mov AX,[EBP+10] ;sum = sum + number1
add AX,[EBP+8] ;sum = sum + number2
mov [result],AX
pop AX
pop EBP
ret 4
```



(a) Stack after saving EBP

Figura: Exemplo de uso de Pilha

Acessando os parâmetros na pilha

sum:

```
push EBP      ; save EBP value
mov EBP,ESP   ; copy TOS into EBP
push AX
mov AX,[EBP+10] ;sum = first number
add AX,[EBP+8] ;sum = sum+second number
mov [result],AX
pop AX
pop EBP
ret 4
```

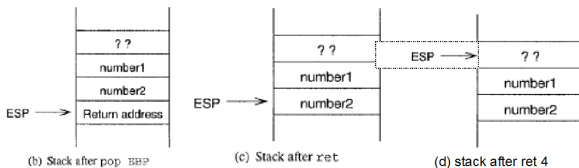


Figura: Exemplo de uso de Pilha

Retornando valor via Pilha

```
.CODE
.STARTUP
PutStr  prompt_msg1      ; request first number
GetInt  [number1]
PutStr  prompt_msg2      ; request second number
GetInt  [number2]
push    AX                ; save space for return
push    WORD [number1]
push    WORD [number2]
call    sum
pop      WORD [result]     ; copy return value
PutStr  sum_msg          ; display sum
PutInt  [result]
nwln
done:   .EXIT
sum:
push    EBP              ; save EBP value
mov     EBP,ESP          ; copy TOS into EBP
push    AX
mov     AX,[EBP+10]       ; sum = first number
add     AX,[EBP+8]        ; sum = sum + second number
mov     [EBP+12],AX       ; write return value
pop     AX
pop     EBP
ret     4
```

Figura: Exemplo de uso de Pilha

Instruções ENTER e LEAVE

- Instruções que preparam a pilha para a chamada ao procedimento
 - enter 0,0
 - Criam um FRAME simples de pilha:
 - push EBP ; save EBP value
 - mov EBP,ESP ; copy TOS into EBP
 - leave
 - Apaga o FRAME de pilha:
 - Mov ESP, EBP
 - pop EBP

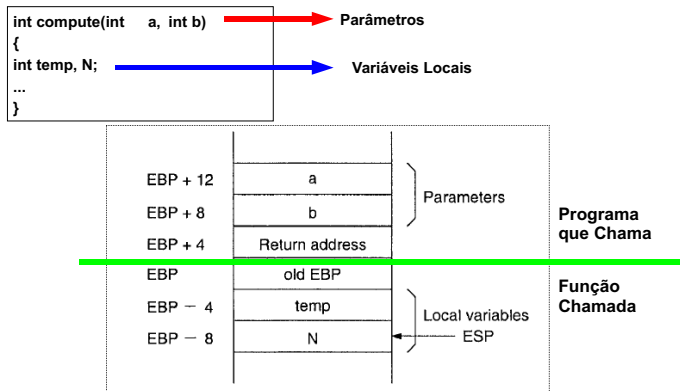
Exemplo

```
;PROC3.ASM
#include "io.mac"
.DATA
prompt_msg1 db "Please input the first number: ",0
prompt_msg2 db "Please input the second number: ",0
sum_msg db "The sum is ",0
.CODE
.STARTUP
PutStr prompt_msg1 ; request first number
GetInt CX ; CX = first number
PutStr prompt_msg2 ; request second number
GetInt DX ; DX = second number
push CX ; place first number on stack
push DX ; place second number on stack
call sum ; returns sum in AX
PutStr sum_msg ; display sum
PutInt AX
nWln
done:
.EXIT
;-----
;Procedure sum receives two integers via the stack.
;The sum of the two integers is returned in AX.
;-----
sum:
    enter 0,0 ; save EBP
    mov AX,[EBP+10] ; sum = first number
    add AX,[EBP+8] ; sum = sum + second number
    leave ; restore EBP
    ret 4 ; return and clear parameters
```

Variáveis Locais?

- Em assembly todas as variáveis declaradas a principio são GLOBAIS
- Porém, é possível usar variáveis locais a um procedimento mediante um registrador:
 - Prático, mas impossível no caso de recursão
- A opção é utilizar a pilha:
 - A instrução *enter* reserva espaço na pilha

Como usar a pilha para variáveis locais



Exemplo: Programa que chama parâmetro em EDX, retorno em EAX

```
; procfib.asm
%include "io.mac"
.DATA
prompt_msg db "Please input a positive number (>1): ",0
output_msg1 db "The largest Fibonacci number less than "
            db "or equal to ",0
output_msg2 db " is ",0
.CODE
.STARTUP
PutStr prompt_msg ; request input number
GetLint EDX      ; EDX = input number
call fibonacci
PutStr output_msg1 ; print Fibonacci number
PutLint EDX
PutStr output_msg2
PutLint EAX
nwnl
done:
.EXIT
```

Como usar a pilha para variáveis locais

Exemplo: Programa que chama parâmetro em EDX, retorno em EAX

```
;-----  
;Procedure fibonacci receives an integer in EDX and computes  
;the largest Fibonacci number that is less than the input  
;number. The Fibonacci number is returned in EAX.  
;-----  
%define FIB_LO  dword [EBP-4]  
%define FIB_HI  dword [EBP-8]  
fibonacci:  
    enter  8,0      ; space for two local variables  
    push  EBX  
    ; FIB_LO maintains the smaller of the last two Fibonacci  
    ; numbers computed; FIB_HI maintains the larger one.  
    mov   FIB_LO,1   ; initialize FIB_LO and FIB_HI to  
    mov   FIB_HI,1   ; first two Fibonacci numbers  
fib_loop:  
    mov   EAX,FIB_HI  ; compute next Fibonacci number  
    mov   EBX,FIB_LO  
    add   EBX,EAX  
    mov   FIB_LO,EAX  
    mov   FIB_HI,EBX  
    cmp   EBX,EDX     ; compare with input number in EDX  
    jle   fib_loop    ; if not greater, find next number  
    ; EAX contains the required Fibonacci number  
    pop   EBX  
    leave          ; clears local variable space  
    ret
```

Como usar a pilha para variáveis locais

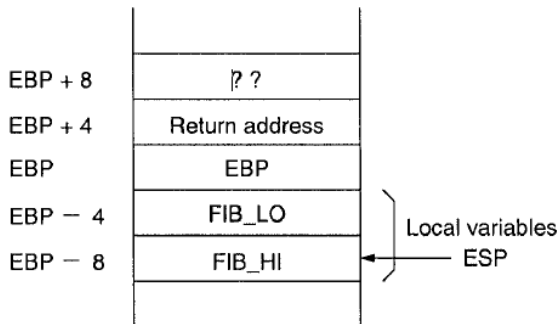
Exemplo: Programa que chama parâmetro em EDX, retorno em EAX

```
;-----  
;Procedure fibonacci receives an integer in EDX and computes  
;the largest Fibonacci number that is less than the input  
;number. The Fibonacci number is returned in EAX.  
;-----  
%define FIB_LO dword [EBP-4]  
%define FIB_HI dword [EBP-8]  
fibonacci:  
    enter 8,0  
    push EBX  
    ; FIB_LO maintains the smaller of the last two Fibonacci  
    ; numbers computed; FIB_HI maintains the larger one.  
    mov FIB_LO,1 ; initialize FIB_LO and FIB_HI to  
    mov FIB_HI,1 ; first two Fibonacci numbers  
fib_loop:  
    mov EAX,FIB_HI ; compute next Fibonacci number  
    mov EBX,FIB_LO  
    add EBX,EAX  
    mov FIB_LO,EAX  
    mov FIB_HI,EBX  
    cmp EBX,EDX ; compare with input number in EDX  
    jle fib_loop ; if not greater, find next number  
    ; EAX contains the required Fibonacci number  
    pop EBX  
    leave ; clears local variable space  
    ret
```

Reserva espaço para duas
dwords na pilha

Como usar a pilha para variáveis locais

Exemplo: Programa que chama parâmetro em EDX, retorno em EAX



Como usar a pilha para variáveis locais

Exemplo: Programa que chama parâmetro em EDX, retorno em EAX

```
;-----  
;Procedure fibonacci receives an integer in EDX and computes  
;the largest Fibonacci number that is less than the input  
;number. The Fibonacci number is returned in EAX.  
;-----
```

```
%define FIB_LO  dword [EBP-4]  
%define FIB_HI  dword [EBP-8]
```

**Permite Acessar a pilha com
nome de variáveis locais**

```
fibonacci:  
    enter  8,0      ; space for two local variables  
    push  EBX  
    ; FIB_LO maintains the smaller of the last two Fibonacci  
    ; numbers computed; FIB_HI maintains the larger one.  
    mov   FIB_LO,1  ; initialize FIB_LO and FIB_HI to  
    mov   FIB_HI,1  ; first two Fibonacci numbers  
  
fib_loop:  
    mov   EAX,FIB_HI ; compute next Fibonacci number  
    mov   EBX,FIB_LO  
    add   EBX,EAX  
    mov   FIB_LO,EAX  
    mov   FIB_HI,EBX  
    cmp   EBX,EDX    ; compare with input number in EDX  
    jle   fib_loop   ; if not greater, find next number  
    ; EAX contains the required Fibonacci number  
    pop   EBX  
    leave          ; clears local variable space  
    ret
```


Como usar a pilha para variáveis locais

Exemplo: Programa que chama parâmetro em EDX, retorno em EAX

```
;-----  
;Procedure fibonacci receives an integer in EDX and computes  
;the largest Fibonacci number that is less than the input  
;number. The Fibonacci number is returned in EAX.  
;-----  
%define FIB_LO  dword [EBP-4]  
%define FIB_HI  dword [EBP-8]  
fibonacci:  
    enter  8,0      ; space for two local variables  
    push  EBX  
    ; FIB_LO maintains the smaller of the last two Fibonacci  
    ; numbers computed; FIB_HI maintains the larger one.  
    mov   FIB_LO,1   ; initialize FIB_LO and FIB_HI to  
    mov   FIB_HI,1   ; first two Fibonacci numbers  
fib_loop:  
    mov   EAX,FIB_HI  ; compute next Fibonacci number  
    mov   EBX,FIB_LO  
    add   EBX,EAX  
    mov   FIB_LO,EAX  
    mov   FIB_HI,EBX  
    cmp   EBX,EDX     ; compare with input number in EDX  
    jle   fib_loop    ; if not greater, find next number  
    ; EAX contains the required Fibonacci number  
    pop   EBX  
    leave  
    ret
```

Limpa espaço utilizado na pilha

Próxima Aula

C e Assembly