

# Assembly Language for Intel-Based Computers, 5<sup>th</sup> Edition

Kip R. Irvine

## Capítulo 5: Procedimientos (Procedure)

*Slides prepared by the author*

*Revision date: June 4, 2006*

(c) Pearson Education, 2002. All rights reserved. You may modify and copy this slide show for your personal use, or for use in the classroom, as long as this copyright statement, the author's name, and the title are not changed.

# Índice

- **Linking de uma biblioteca externa**
- Biblioteca do autor deste livro
- Operações de pilha
- Definindo e usando Procedimentos
- Projeto de programas usando Procedimentos

# Linking de uma biblioteca externa

- Biblioteca de Linking
- Chamando um procedimento de biblioteca
- Fazendo o *Linking* para uma biblioteca
- Procedimentos de biblioteca
- Seis Exemplos

# Biblioteca de Linking

- É um arquivo contendo procedimentos que foi compilado para o código de máquina
  - construído de um ou mais arquivos OBJ
- Para construir uma biblioteca, . . .
  - começa com um ou mais arquivos fonte ASM
  - converte cada um em arquivo OBJ
  - cria um arquivo biblioteca vazio (extensão .LIB)
  - adiciona os arquivos OBJ no arquivo biblioteca, usando o utilitário LIB da Microsoft

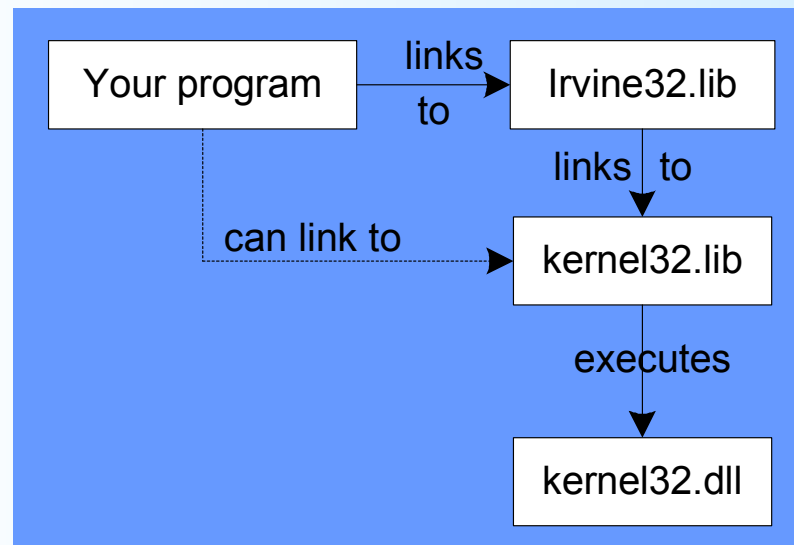
# Chamando um procedimento de biblioteca

- Chama-se um procedimento de biblioteca usando instrução CALL. Alguns procedimentos requerem argumentos de entrada. A diretiva INCLUDE copia os protótipos de procedimentos (declarações).
- O seguinte exemplo mostra “00001234” na tela:

```
INCLUDE Irvine32.inc
.code
    mov eax,1234h          ; input argument
    call WriteHex          ; show hex number
    call Crlf              ; end of line
```

# Fazendo o Linking para uma biblioteca

- É feito o link p/ a biblioteca **Irvine32.lib**
- Nota-se que existem dois arquivos LIB no exemplo:
  - Irvine32.lib;
  - kernel32.lib: parte do Microsoft Win32 Software Development Kit (SDK)



# Próxima seção

- **Linking de uma biblioteca externa**
- **Biblioteca do autor deste livro**
- Operações de pilha
- Definindo e usando Procedimentos
- Projeto de programas usando Procedimentos

# Procedimentos de biblioteca (1 de 4)

**CloseFile** – Fecha um arquivo aberto de disco

**Clrscr** – Limpa a tela e coloca o cursor no canto esquerdo superior

**CreateOutputFile** – Cria um novo arquivo de disco para escrever no modo output

**Crlf** – Escreve a seqüência de fim de linha na saída padrão

**Delay** - Pausa a execução de programa por um intervalo de *n* milisegundos

**DumpMem** - Escreve um bloco de memória em hex na saída padrão  
esi – endereço, ecx – numero de elementos, ebx - tipo

**DumpRegs** – Mostra os registradores de uso geral e flags (hex)

**GetCommandtail** – Copia os argumentos da linha de comando em array de bytes

**GetMaxXY** – Obtem o número de colunas, linhas do buffer de window da tela

**GetMseconds** – Retorna os milisegundos passados a partir da meia-noite



# Procedimentos de biblioteca (2 de 4)

**GetTextColor** – Retorna as cores ativas de texto (frente e fundo) da tela

**Gotoxy** – Localiza o cursor na linha e coluna da tela

**IsDigit** – Aciona o flag Zero se AL contem código ASCII de dígito decimal (0–9)

**MsgBox, MsgBoxAsk** – Mostra caixa de mensagem popup

**OpenInputFile** – Abre um arquivo existente para entrada

**ParseDecimal32** – Converte uma cadeia de inteiros sem sinal para binários

**ParseInteger32** – Converte uma cadeia de inteiros com sinal para binários

**Random32** - Gera inteiro pseudoaleatório de 32-bits no intervalo de 0 a FFFFFFFFh

**Randomize** – Semeia o gerador de número aleatório

**RandomRange** – Gera um inteiro pseudoaleatório dentro de um intervalo especificado

**ReadChar** – Lê um caractere da entrada padrão

# Procedimentos de biblioteca (3 de 4)

**ReadFromFile** – Lê um arquivo de entrada de disco num buffer

**ReadDec** – Lê um inteiro em decimal sem sinal de 32-bits do teclado

**ReadHex** – Lê um inteiro em hexadecimal de 32-bits do teclado

**ReadInt** – Lê um inteiro em decimal com sinal de 32-bits do teclado

**ReadKey** – Lê caractere do buffer de entrada do teclado

**ReadString** – Lê cadeia da entrada padrão, terminada por [Enter]

**SetTextColor** – Aciona as cores de frente e de fundo para todas as saídas subsequentes para a tela

**StrLength** – Retorna o comprimento de uma cadeia

**WaitMsg** – Mostra mensagem e espera pela tecla Enter ser pressionada

**WriteBin** – Escreve um inteiro sem sinal de 32-bits em formato ASCII binário.  
eax deve conter o número a ser escrito

**WriteBinB** – Escreve inteiro em binário no formato byte, word ou doubleword  
eax deve conter o número a ser escrito  
ebx deve conter o tipo (1,2, ou 4)

# Procedimentos de biblioteca (4 de 4)

**WriteChar** – Escreve um caractere na saída padrão  
al deve conter o caractere a ser escrito

**WriteDec** – Escreve um inteiro sem sinal de 32-bits em formato decimal  
eax deve conter o número inteiro a ser escrito

**WriteHex** – Escreve um inteiro sem sinal de 32-bits em formato hexadecimal  
eax deve conter o número inteiro a ser escrito

**WriteHexB** – Escreve um número de 32 bits em byte, word ou doubleword em hexadecimal  
eax deve conter o número inteiro , ebx deve conter o tipo (1,2 ou 4)

**WriteInt** – Escreve um inteiro com sinal de 32-bits em formato decimal  
eax deve conter o número inteiro a ser escrito

**WriteString** – Escreve uma cadeia terminada por zero na tela  
edx deve conter o endereço da cadeia

**WriteToFile** – Escreve buffer no arquivo de saída

**WriteWindowsMsg** – Mostra a mensagem de erro mais recente gerada pelo MS-Windows

# Exemplo 1

Limpar a tela, atrasar o programa por 500 milisegundos e mostrar os registradores e flags.

```
.code  
    call Cclrscr  
    mov  eax,500  
    call Delay  
    call DumpRegs
```

saída:

```
EAX=00000613 EBX=00000000 ECX=000000FF EDX=00000000  
ESI=00000000 EDI=00000100 EBP=0000091E ESP=000000F6  
EIP=00401026 EFL=00000286 CF=0 SF=1 ZF=0 OF=0
```

## Exemplo 1-b

Limpar a tela e mostrar os dados da memória em Hexadecimal.

```
.data
    mywords WORD 1,2,3,4
.code
    call Cclrscr
    mov esi, offset mywords
    mov ecx, 4
    mov ebx, 2
    call DumpMem
```

saída:

```
Dump of offset 00405000
```

```
-----
```

```
0001 0002 0003 0004
```

## Exemplo 2

Mostrar uma cadeia terminada por zero e mover o cursor ao início da linha seguinte.

```
.data
str1 BYTE "Assembly language is easy!",0

.code
    mov  edx,OFFSET str1
    call WriteString
    call Crlf
```

## Exemplo 2a

Mostrar uma cadeia terminada por zero e mover o cursor ao início da linha seguinte (usar CR/LF incorporado)

```
.data
str1 BYTE "Assembly language is easy!", 0Dh, 0Ah, 0

.code
    mov     edx, OFFSET str1
    call    WriteString
```

## Exemplo 3

Mostrar um inteiro sem sinal em binário, decimal e hexadecimal, em linhas separadas.

```
IntVal = 35
.code
    mov    eax,IntVal
    call   WriteBin           ; display binary
    call   Crlf
    call   WriteDec           ; display decimal
    call   Crlf
    call   WriteHex           ; display hexadecimal
    call   Crlf
```

saída:

```
0000 0000 0000 0000 0000 0000 0010 0011
35
00000023
```



## Exemplo 4

Entrar com uma cadeia digitada pelo usuário. EDX aponta para a cadeia e ECX especifica o número máximo de caracteres que o usuário pode entrar.

```
.data
fileName BYTE 80 DUP(0)
          BYTE 0
.code
    mov edx,OFFSET fileName
    mov ecx,SIZEOF fileName
    call ReadString
```

Um byte zero automaticamente termina o string.

## Exemplo 5

Gerar e mostrar dez inteiros pseudoaleatórios, com sinal, no intervalo 0 – 99. Mostrar cada inteiro na tela com WriteInt em linha separada.

```
.code
    mov ecx,10                ; loop counter

L1: mov  eax,100              ; ceiling value
    call RandomRange          ; generate random int
    call WriteInt             ; display signed int
    call Crlf                 ; goto next display line
    loop L1                   ; repeat loop
```

## Exemplo 6

Mostrar uma cadeia terminada com zero com caracteres amarelos sobre fundo azul.

```
.data
str1 BYTE "Color output is easy!",0

.code
    mov  eax,yellow + (blue * 16)
    call SetTextColor
    mov  edx,OFFSET str1
    call WriteString
    call Crlf
```

A cor de fundo é multiplicada por 16 antes de ser adicionada à cor de frente.

# Próxima seção

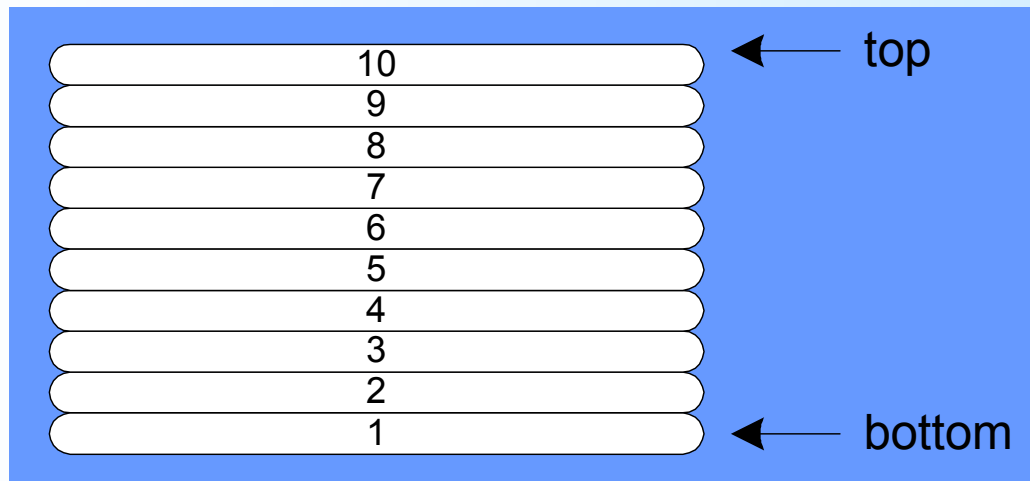
- **Linking de uma biblioteca externa**
- Biblioteca do autor deste livro
- **Operações de pilha**
- Definindo e usando Procedimentos
- Projeto de programas usando Procedimentos

# Operações de pilha

- Pilha de tempo de execução
- Operação PUSH
- Operação POP
- Instruções PUSH e POP
- Usando PUSH e POP
- Exemplo: Revertendo uma cadeia
- Instruções relacionadas

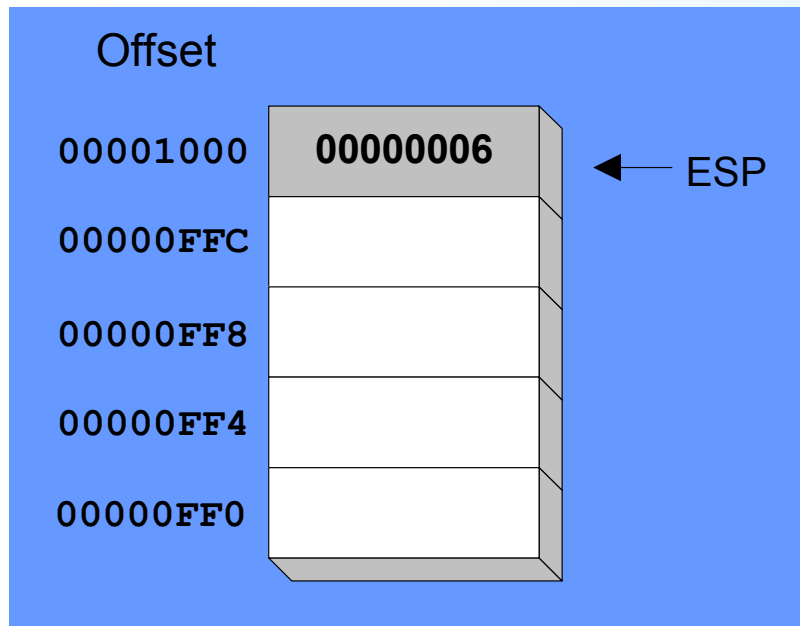
# Pilha de tempo de execução

- Imagine uma pilha de pratos . . .
  - Pratos são somente adicionados no topo
  - Pratos são somente removidos do topo
  - Estrutura LIFO



# Pilha de tempo de execução

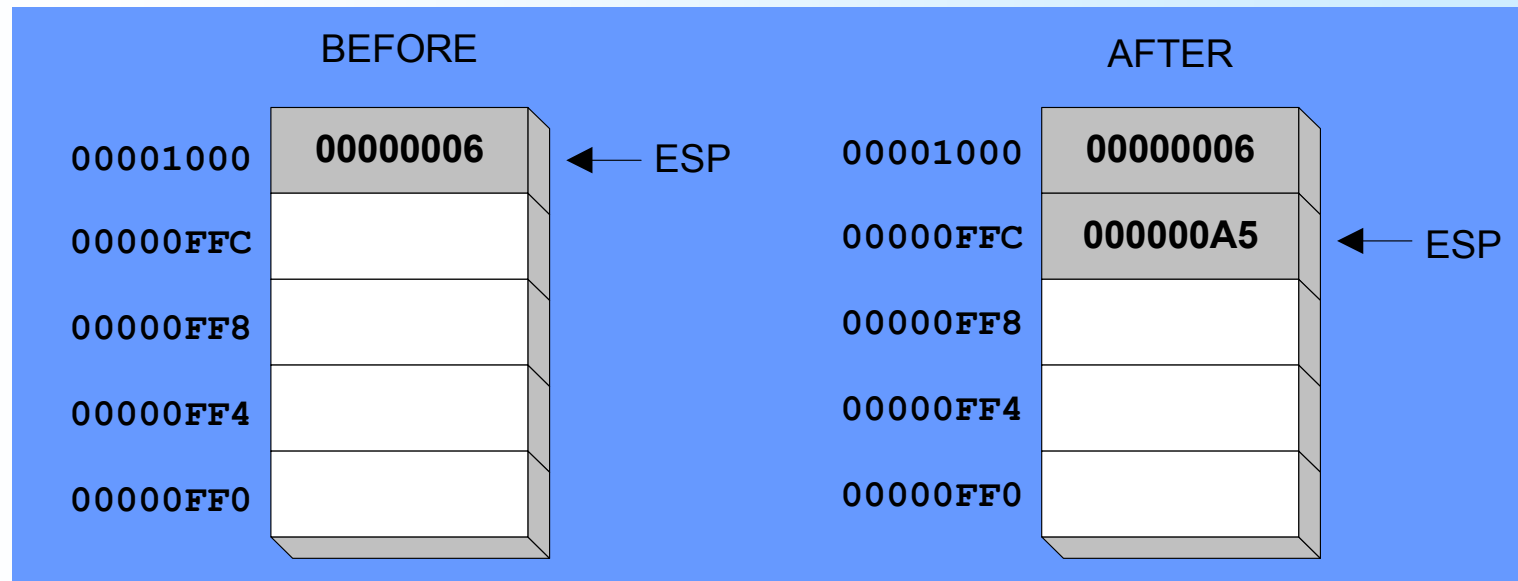
- Gerenciado pelo CPU, usando dois registradores
  - SS (stack segment)
  - ESP (stack pointer)



OBS: Em termos de endereços de memória, a pilha cresce p/ baixo, ou seja, novos elementos são empilhados em endereços menores. Mas em termos lógicos, é uma pilha!

# Operação PUSH (1 de 2)

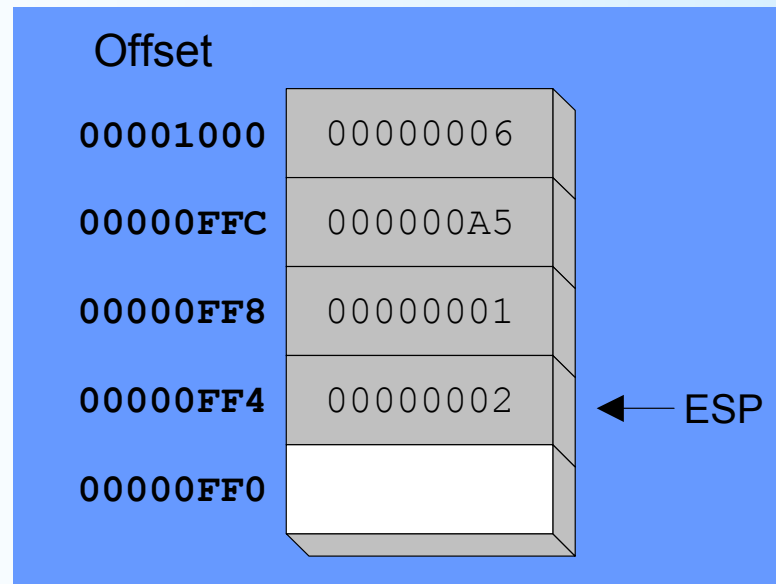
- Uma operação push de 32-bits decrementa o ponteiro de pilha de 4 e copia um valor na posição apontada pelo ponteiro de pilha.





## Operação PUSH (2 de 2)

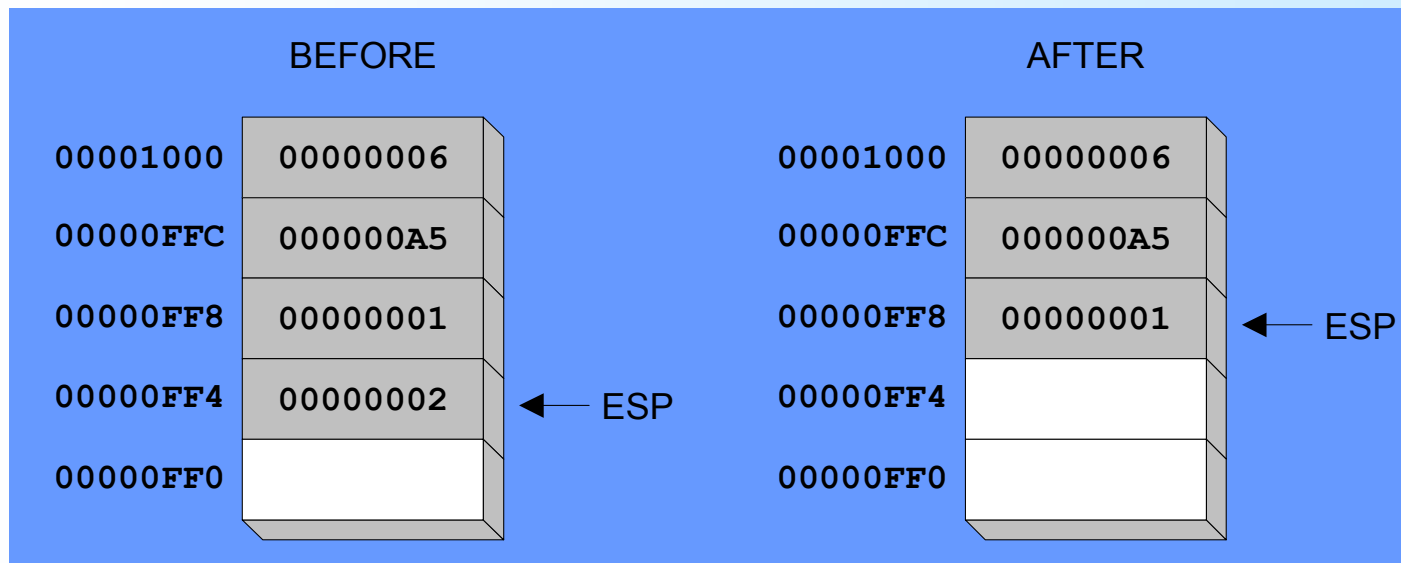
- A mesma pilha após carregar mais dois inteiros:



A pilha cresce para baixo. A área abaixo de ESP está sempre disponível (a menos que ocorra overflow de pilha).

# Operação POP

- Copia o valor da pilha [ESP] num registrador ou variável.
- Adiciona  $n$  a ESP, onde  $n$  é 2 ou 4.
  - valor de  $n$  depende do atributo do operando que recebe o dado (WORD ou DWORD).



# Instruções PUSH e POP

- Sintaxe do PUSH:
  - PUSH *r/m16*
  - PUSH *r/m32*
  - PUSH *imm32*
- Sintaxe do POP:
  - POP *r/m16*
  - POP *r/m32*

# Usando PUSH e POP

Salvar e recuperar registradores quando eles contem valores importantes. **As instruções PUSH e POP devem ser executadas em ordem oposta, com relação aos registradores.**

```
push esi                ; push registers
push ecx
push ebx

mov esi,OFFSET dwordVal ; display some memory
mov ecx,LENGTHOF dwordVal
mov ebx,TYPE dwordVal
call DumpMem

pop ebx                 ; restore registers
pop ecx
pop esi
```

# Exemplo: loop aninhado

Quando se cria um loop aninhado, faz o push do contador do loop externo antes de entrar no loop interno:

```
    mov ecx,100          ; set outer loop count
L1:  ; begin the outer loop
    push ecx             ; save outer loop count

    mov ecx,20           ; set inner loop count
L2:  ; begin the inner loop
    ;
    ;
    loop L2              ; repeat the inner loop

    pop ecx              ; restore outer loop count
    loop L1              ; repeat the outer loop
```

# Exercício: Invertendo um string

- Dicas:
- Usar um loop com endereçamento indexado
- Fazer o Push de cada caracter na pilha
- Começando pelo início da cadeia, fazer o pop da pilha em ordem reversa, inserir cada caracter de volta à cadeia
- Guardar cada caracter em EAX antes de ser guardado na pilha
  - por que ?

# Exercício: Invertendo um string

- Dicas:
- Usar um loop com endereçamento indexado
- Fazer o Push de cada caracter na pilha
- Começando pelo início da cadeia, fazer o pop da pilha em ordem reversa, inserir cada caracter de volta à cadeia
- Guardar cada caracter em EAX antes de ser guardado na pilha
  - por que ?

Porque somente word (16-bit) ou doubleword (32-bit) podem ser carregados na pilha.

# Exercício: Invertendo um string

```
.data
name BYTE "Frase a ser invertida",0
nameSize = ($ - nome) -1

.code

    mov ecx, nameSize
    mov esi, 0
L1:  movzx eax, name[esi]
     push  eax
     inc   esi
     loop L1

     mov ecx, nameSize
     mov esi, 0
L2:  pop  eax
     mov name[esi], al
     inc esi
     loop L2

    mov edx, OFFSET name
    call WriteString
```



# Sua vez . . .

- Usando o programa de inversão de cadeia:
  - #1: Modificar o programa tal que o usuário possa introduzir uma cadeia contendo de 1 a 50 caracteres.
  - #2: Modificar o programa tal que seja introduzida uma lista de inteiros de 32 bits pelo usuário, e mostre os inteiros em ordem reversa.

# Instruções relacionadas

- PUSHFD e POPFD
  - push e pop do registrador EFLAGS
- PUSHAD faz o push dos registradores de uso geral de 32-bit na pilha
  - ordem: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
- POPAD faz o pop dos mesmos registradores da pilha em ordem reversa
  - PUSHA e POPA faz o mesmo com registradores de 16 bits

## Sua vez . . .

- Escrever um programa que faz o seguinte:
  - Atribui valores inteiros a EAX, EBX, ECX, EDX, ESI e EDI
  - Usa PUSHAD para carregar os registradores de uso geral na pilha
  - Usando um loop, o programa deve fazer o pop de cada inteiro da pilha e mostrá-lo na tela

# Próxima seção

- **Linking de uma biblioteca externa**
- Biblioteca do autor deste livro
- Operações de pilha
- **Definindo e usando Procedimentos**
- Projeto de programas usando Procedimentos



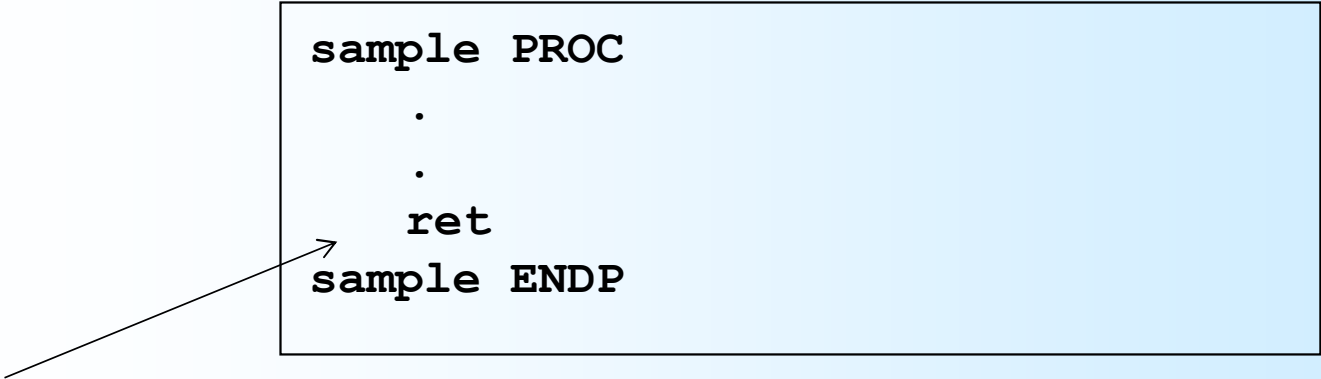
*Intervalo ?*

# Definindo e usando procedimentos

- Criando Procedimentos
- Documentando Procedimentos
- Exemplo: procedimento SumOf
- Instruções CALL e RET
- Chamadas de procedimentos aninhados
- Labels Local e Global
- Parâmetros de Procedimentos
- Símbolos de Flowchart
- Operador USES

# Criando Procedimentos

- Problemas grandes podem ser divididos em pequenas tarefas para torná-los mais fáceis
- Um procedimento no ASM é equivalente a uma função no Java ou C++
- Segue um exemplo de procedimento em assembly denominado **sample**:



```
sample PROC  
    .  
    .  
    ret  
sample ENDP
```

**Nunca esqueça do ret !**

# Documentando Procedimentos

Documentação sugerida para cada procedimento:

- Uma descrição de todas as tarefas realizadas pelo procedimento.
- **Recebe:** uma lista de parâmetros de entrada e seu uso .
- **Retorna:** uma descrição dos valores retornado pelo procedimento.
- **Requer:** lista opcional de requisitos chamados de pré-condições que devem ser satisfeitas antes do procedimento ser chamado.

Se um procedimento é chamado sem terem satisfeitas as pré-condições provavelmente não produz a saída desejada.

# Exemplo: procedimento SumOf

```
;-----  
SumOf PROC  
;  
; Calcula e retorna a soma de três inteiros de 32-bits.  
; Recebe: EAX, EBX, ECX, os três inteiros. Pode ser  
; com sinal ou sem sinal.  
; Retorna: EAX = soma e os flags de status (Carry,  
; Overflow, etc.) são alterados.  
; Requer: nada  
;-----  
    add eax,ebx  
    add eax,ecx  
    ret  
SumOf ENDP
```



# Instruções CALL e RET

- A instrução CALL chama um procedimento
  - Faz o push do offset da instrução seguinte na pilha
  - Copia o endereço do procedimento chamado no EIP
- A instrução RET retorna de um procedimento
  - Faz o pop do topo da pilha no EIP

# Exemplo de CALL-RET (1 de 2)

0000025 é o offset da  
instrução imediatamente  
seguinte à instrução CALL

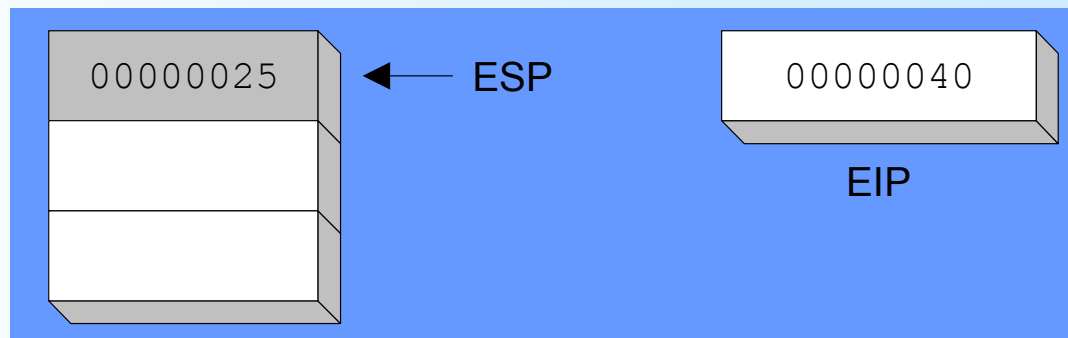
00000040 é o offset da  
primeira instrução dentro  
de MySub

```
main PROC
    00000020 call MySub
    00000025 mov  eax,ebx
    .
    .
main ENDP

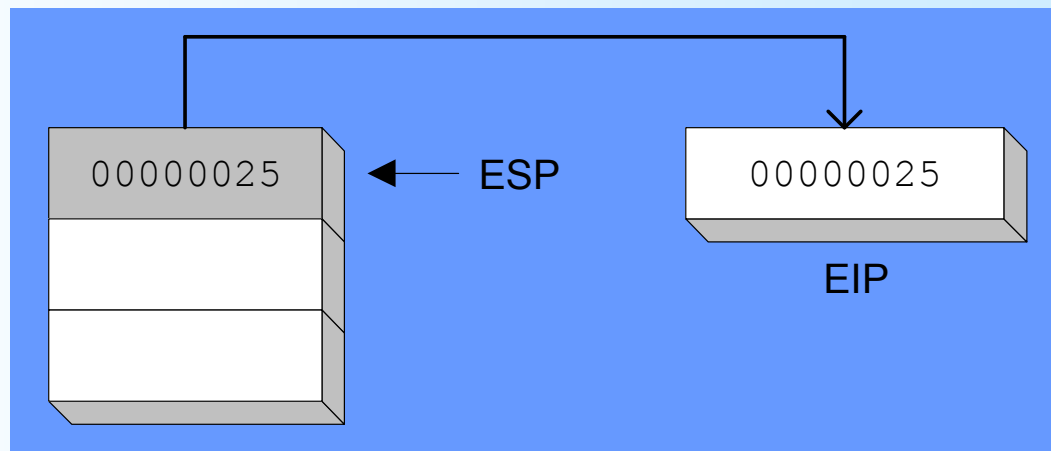
MySub PROC
    00000040 mov  eax,edx
    .
    .
    ret
MySub ENDP
```

## Exemplo de CALL-RET (2 de 2)

A instrução CALL faz o push de 00000025 na pilha e carrega 00000040 em EIP



A instrução RET faz o pop de 00000025 da pilha para EIP



(pilha mostrada ao executar RET)

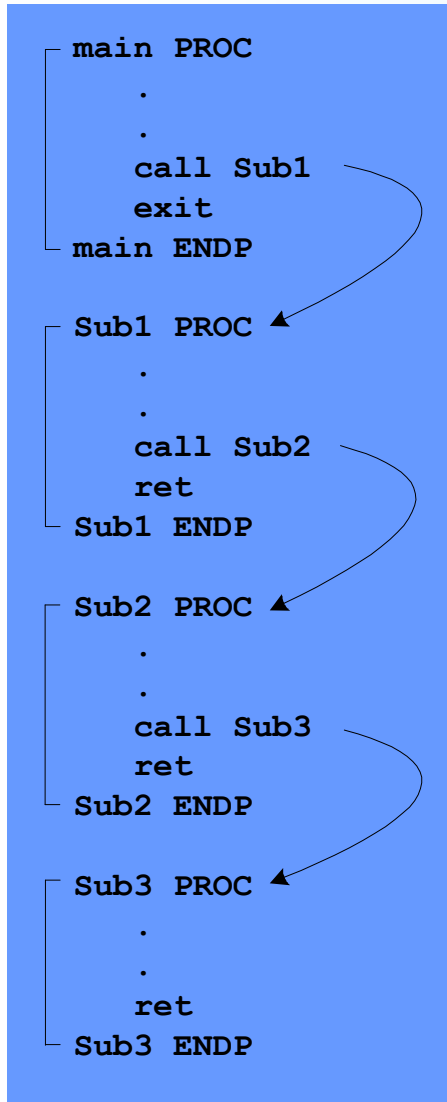
# Chamada de procedimentos aninhados

```
main PROC
    .
    .
    call Sub1
    exit
main ENDP

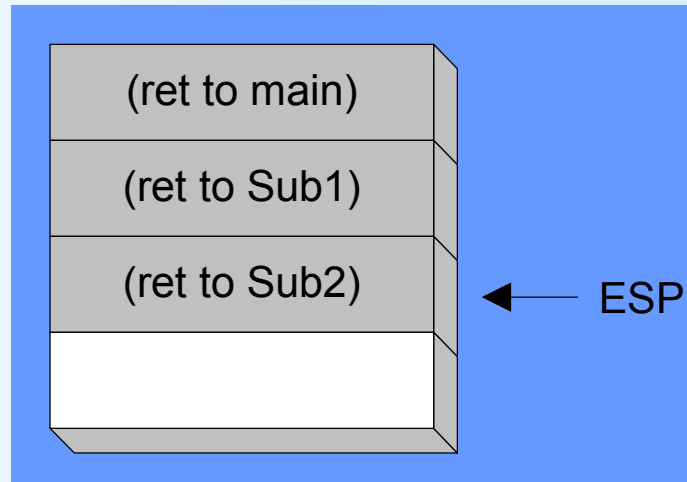
Sub1 PROC
    .
    .
    call Sub2
    ret
Sub1 ENDP

Sub2 PROC
    .
    .
    call Sub3
    ret
Sub2 ENDP

Sub3 PROC
    .
    .
    ret
Sub3 ENDP
```



Quando Sub3 é chamado a pilha contém todos os três endereços de retorno:



# Labels Local e Global

Um label local é visível somente para as instruções dentro do mesmo procedimento. Um label global é visível em todo o programa.

```
main PROC
    jmp L2                ; erro: jump p/ label local
L1::                     ; label global
    exit
main ENDP

sub2 PROC
L2:                      ; label local
    jmp L1                ; ok: jump p/ label global
    ret
sub2 ENDP
```

# Parâmetros de procedimentos (1 de 3)

- Um bom procedimento deve ser útil em muitos programas diferentes
  - Mas não se refere a nomes de variáveis específicas
- Parâmetros servem para tornar os procedimentos flexíveis pois os valores de parâmetros podem mudar no tempo de execução

## Parâmetros de Procedimento (2 de 3)

O procedimento ArraySum calcula a soma de um vetor. Ele faz duas referências para nomes de variáveis específicas:

```
ArraySum PROC
    mov esi,0                ; array index
    mov eax,0                ; set the sum to zero
    mov ecx,LENGTHOF myarray ; set number of elements

L1: add eax,myArray[esi]     ; add each integer to sum
    add esi,4                ; point to next integer
    loop L1                  ; repeat for array size

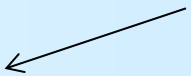
    mov theSum,eax           ; store the sum
    ret
ArraySum ENDP
```

E se quisesse calcular a soma de dois ou três vetores dentro de um mesmo programa?

## Parâmetro de Procedimentos (3 de 3)

Essa versão de ArraySum retorna a soma de qualquer vetor de doubleword 's cujo endereço é contido em ESI. A soma é retornado em EAX:

main PROC  
Vetor DWORD 1,2,3,4  
....  
mov ecx, lengthof Vetor  
Mov esi offset Vetor  
Call Arraysum  
.....



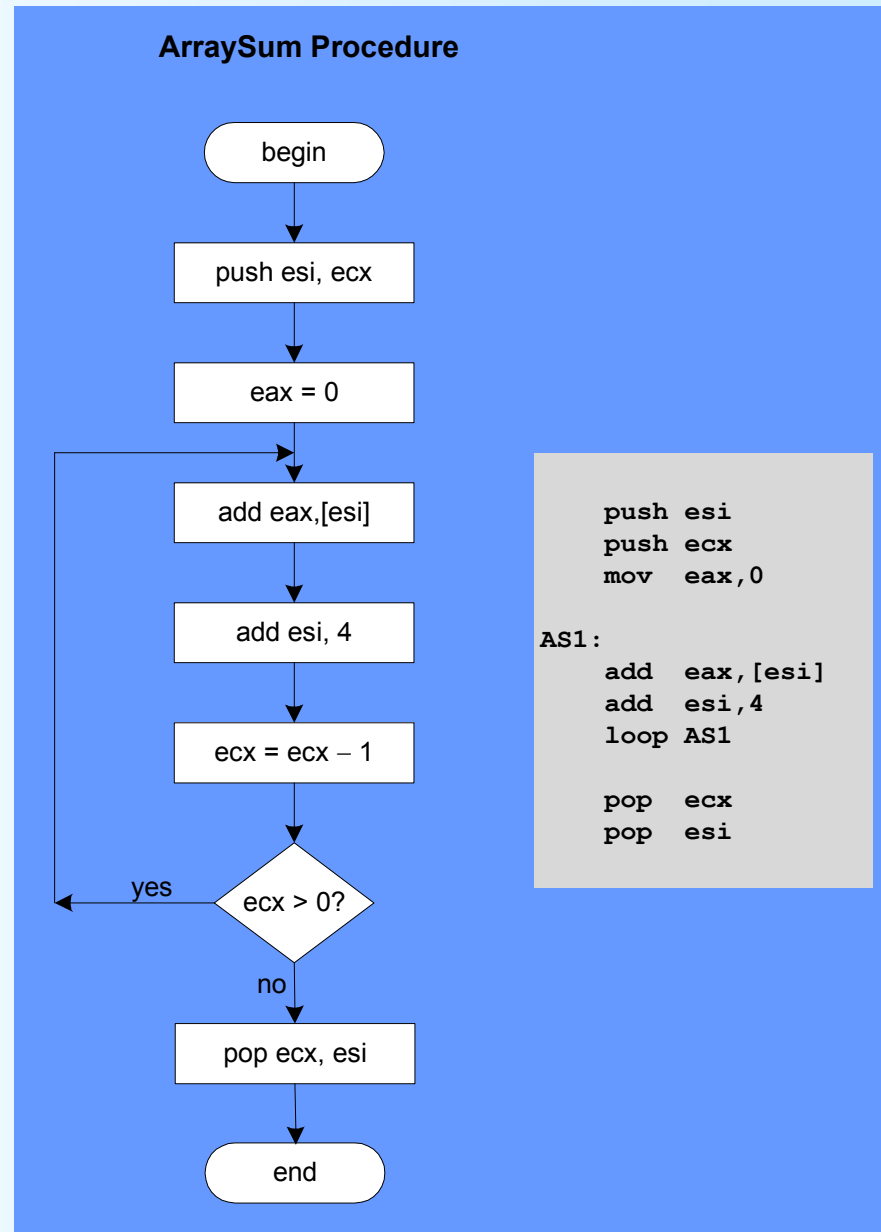
```
ArraySum PROC
; Recebe: ESI aponta a um vetor de doublewords,
;   ECX = número de elementos do vetor.
; Retorna: EAX = sum
;-----
    mov eax,0                ; set the sum to zero

L1: add eax,[esi]             ; add each integer to sum
    add esi,4                ; point to next integer
    loop L1                  ; repeat for array size

    ret
ArraySum ENDP
```



# Flowchart para o procedimento ArraySum



## Sua vez . . .

- Modificar o flowchart do slide anterior para que o usuário possa continuar a entrada de scores até que -1 seja introduzido

# Operador USES

- Lista os registradores que serão preservados

```
ArraySum PROC USES esi ecx
    mov eax,0                ; set the sum to zero
    etc.
```

MASM gera o código abaixo, em **vermelho**:

```
ArraySum PROC
    push esi
    push ecx
    .
    .
    pop ecx
    pop esi
    ret
ArraySum ENDP
```

# Próxima seção

- Linking de uma biblioteca externa
- Biblioteca do autor deste livro
- Operações de pilha
- Definindo e usando Procedimentos
- **Projeto de programas usando Procedimentos**

# Projeto de programas usando procedimentos

- Projeto Top-Down (decomposição funcional) envolve o seguinte:
  - Projetar o seu programa antes de iniciar a codificação
  - Quebrar tarefas grandes em tarefas pequenas
  - Usar uma estrutura hierárquica baseada em chamadas de procedimentos
  - Testar procedimentos individuais separadamente

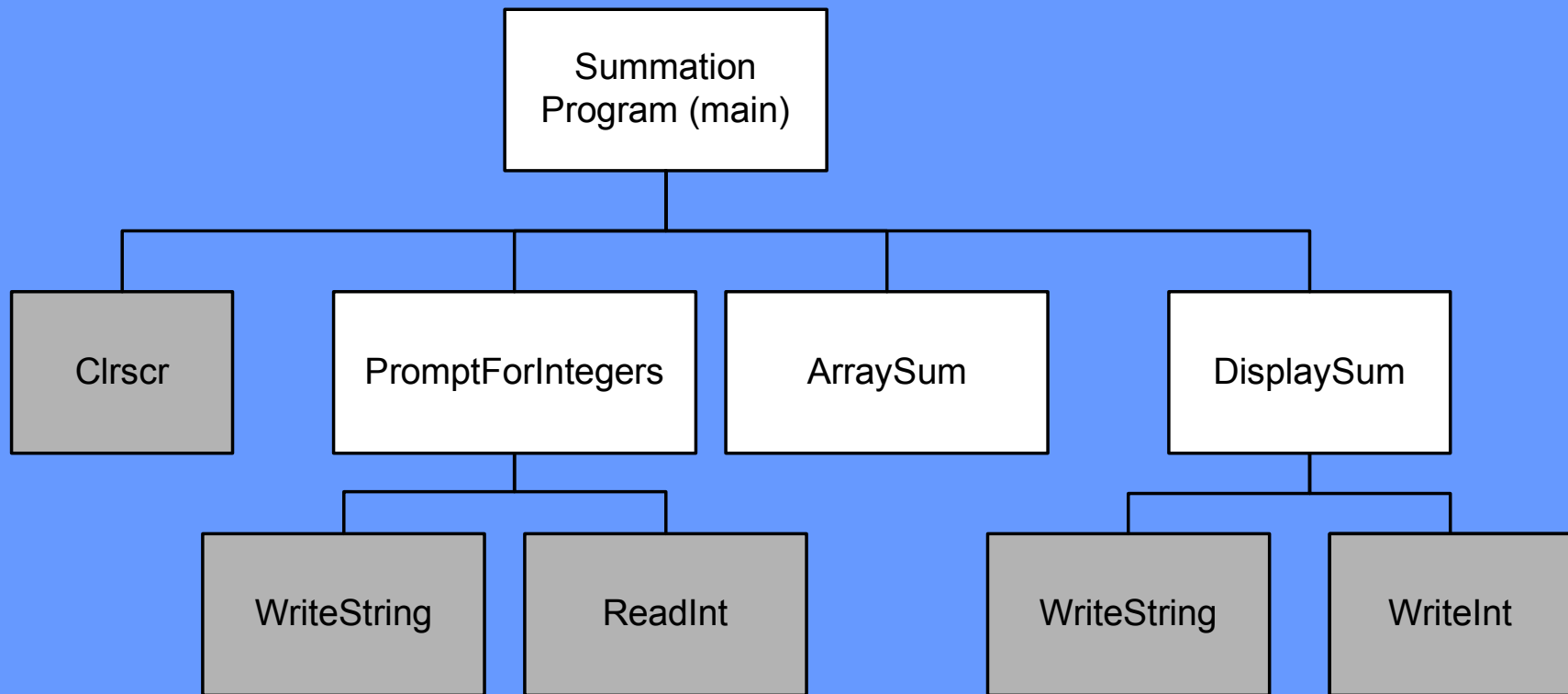
# Programa de soma de inteiros (1-2)

*Descrição:* Escrever um programa que solicita ao usuário digitar vários inteiros de 32 bits , armazena-os num vetor, calcula a soma do vetor, e mostra o resultado na tela.

Passos principais:

- Solicita (**prompt**) , ao usuário, digitar vários inteiros
- Calcula a soma do vetor
- Mostra a soma
- **Próximos Slides: Removidos, vocês já sabem isto...**

# Programa de soma de inteiros (2-2)



cinza: função  
de biblioteca

# The End

