Endereçamento em Assembly

- Registrador : O operando é um registrador
  – mov ax,bx
- Imediato : O operando faz parte da instrução, geralmente logo após o código da operação.
  – mov al,5
- Direto : O endereço do operando faz parte da instrução
  – or al,[2000h]
    - Realiza uma operação lógica or com o conteúdo da posição de memória 2000h e o registrador al. Os colchetes ao redor de um número ou expressão indicam que se trata de um endereço e não de um dado.

Endereçamento em Assembly

- Indireto: O endereço do operando está em um registrador *base* ou de índice
  - mov cx,[bx]
- Índice : soma de um registrador de índice e um *deslocamento. O deslocamento* faz parte da instrução.
  - *mov al,[100h+si]*

- *Deslocamento* : um valor **imediato** de 8,16,ou 32 bits, logo após um código de operação.

- Índice de base com deslocamento:
  - mov ax,[bx+si+5]

- *Base* : é o conteúdo de qualquer registrador de uso geral. Geralmente **bp** e **bx**. O registrador de base geralmente fica fixo.
- *Índice* : é o conteúdo de qualquer registrador de uso geral. Geralmente **si** ou **di**. O registrador de índice geralmente é incrementado.

Tamanho de endereço e operando

- Podemos trabalhar com operandos de 8, 16 ou 32 bits.
- Em C
  - 8 bits → char
  - 16 bits → int
  - 32 bits → float, long int
- Usamos descritores de tipos
  - byte
  - word
  - dword

Diretivas ou pseudo operações

- Elas identificam procedimentos e atribuem locais da memória para dados fixos, instruções e áreas de armazenamento.
- Diretivas de definiçao de dados
  - **db** : define byte
  - **dd** : define dupla palavra
  - **dq** : define quadrupla palavra
  - **dt** : define 10 bytes (para uso com numeros de ponto flutuante)
  - **dw** : define palavra

Criando macros

```
;--------------------------------------------------
;Chamada: _printf Buffer, Tamanho
;--------------------------------------------------
%macro _printf 2 ; 2 é o número de parâmetros
    mov rax,1             ;escrita
    mov rdi,1             ;fd=1, na tela
    mov rsi,%1            ;1o param. buffer a ser gravado
    mov rdx,%2            ;2o param. tamanho a ser gravado
    sycall
%endmacro
```

Criando macros

```
%macro _printNum 1
      push rax
      mov eax,[%1]
      xor ecx,ecx
      %%_divide:                ;LABELS EM MACRO DEVEM VIR PRECEDIDOS DE %%
         mov bx,10
         div bx     ; nao se permite div 10, div 3, div 20, etc
         cmp eax,0  ; eax quociente, edx é o resto
         add edx,'0'
         push rdx   ;empilha o resto
         inc ecx    ;conta elementos na pilha
         cmp eax,0
      jnz %%_divide
      %%_imprimeTopo:
          pop rdx
          mov [aux],edx
          _putc aux ; macro para imprimir 1 caractere
          dec cx
          cmp cx,0
       jnz %%_imprimeTopo
       pop rax
%endmacro
```

Uma macro ordena ao compilador para copiar o código correspon-dente ao nome no local

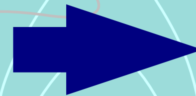
```
;-------------------------------------------------
;Chamada: _printf Buffer, Tamanho
;-------------------------------------------------
%macro _printf 2 ; 2 é o número de parâmetros
    mov eax,4           ;escrita
    mov ebx,1           ;fd=1, na tela
    mov ecx,%1          ;1o param. buffer a ser gravado
    mov edx,%2          ;2o param. tamanho a ser gravado
    int 0x80
%endmacro
```

Passagem de parâmetros em função

```
funcSoma:        ;label da funcao funcSoma
    push ebp     ;salva registro base da pilha
    mov ebp,esp  ;base recebe topo da pilha
    mov eax,[ebp+8]
    mov ebx,[ebp+12]
    add eax,ebx
    pop ebp
ret              ;o ret é que identifica a função
```

EBP, EAX, EBX, ESP
Tem 4 bytes

```
;CHAMANDO funcSoma
push num1
push num2
call funcSoma
pop num1
pop num2
;resultado em eax
```



Pilha

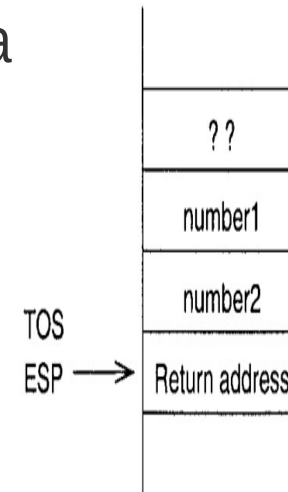


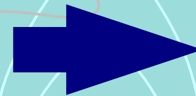
Figure 11.7 Stack state after the sum procedure call: Return address is the EIP value pushed onto the stack as part of executing the call instruction.

Passagem de parâmetros em função

```
funcSoma:        ;label da funcao funcSoma
    push ebp        ;salva registro base da pilha
    mov ebp,esp  ;base recebe topo da pilha
    mov eax,[ebp+8]
    mov ebx,[ebp+12]
    add eax,ebx
    pop ebp
ret                    ;o ret é que identifica a função
```

EBP, EAX, EBX, ESP
Tem 4 bytes

```
;CHAMANDO funcSoma
push num1
push num2
call funcSoma
pop num1
pop num2
;resultado em eax
```



Pilha

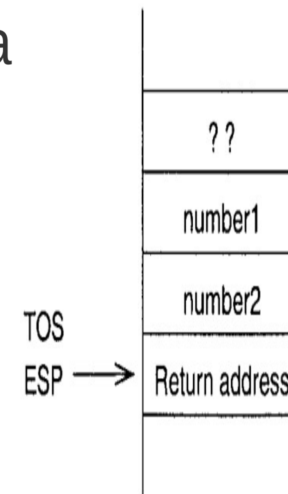


Figure 11.7 Stack state after the sum procedure call: Return address is the EIP value pushed onto the stack as part of executing the call instruction.

Passagem de parâmetros

```
funcSoma:        ;label da funcao funcSoma
    push ebp
    mov ebp,esp
    mov eax,[ebp+8]
    mov ebx,[ebp+12]
    add eax,ebx
    pop ebp
ret                        ;retorna
```

EBP, EAX, EBX, ESP
Tem 4 bytes

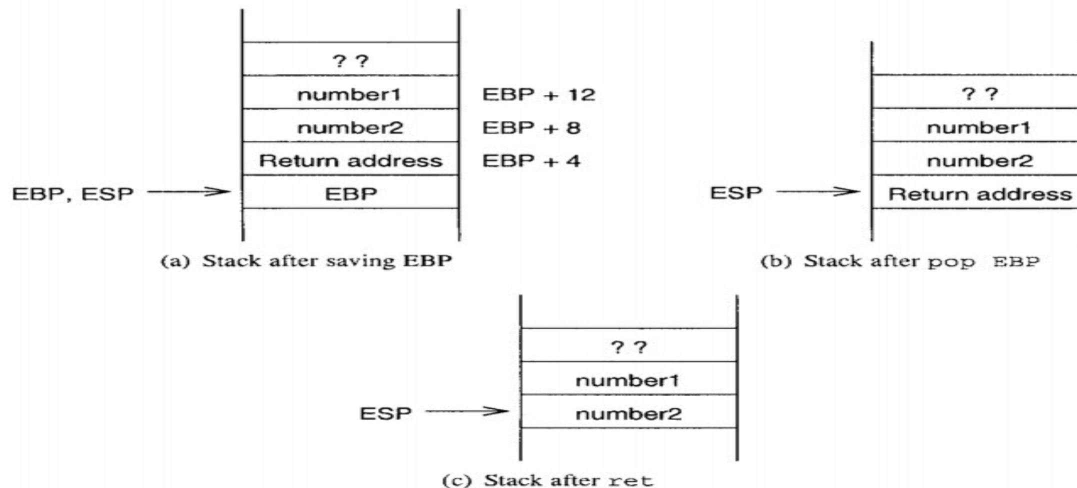


Figure 11.8 Changes in stack state during a procedure execution.

- **lods** – LoaD String
- **cmps** – CoMPare String
- **movs** – MOVe String
- **scas** – SCAn String
- **stos** – STOre String

- Cada mneumônico pode ser seguido por um sufixo **b**,**w**, ou **d**, indicando se a operacao opera sobre bytes, palavras ou palavras duplas.

Manipulando strings

- Uma instrução de string processa um único byte, palavra ou palavra dupla. Para processar uma série de bytes precisamos dos comandos de repetição.
  - **rep** : repetir enquanto cx diferente de zero
  - **repnz** ou **repne** : repetir enquanto **cx** diferente de zero e flag **zf** não ativo
  - **repz** ou **repe** : repetir enquanto **cx** diferente de zero e flag **zf** não inativo

- **std** – seta flag de direção (DF = 1)
- **cld** – reseta flag de direção (DF = 0)

Registradores	**Usado por**	**Significado**
DS:SI	LOADS CMPS MOVS	Contém o endereço do operando string de origem
ES:DI	CMPS MOVS SCAS STOS	Contém o endereço do operando string de destino
CX	REP REPE REPNE	Contém o número de itens a serem carregados, comparados, movidos, pesquisados ou armazenados
AL, AX ou EAX	LODS SCAS STOS	Recebe dados ou contém dados a serem pesquisados ou armazenados

- Obs : DF = 0 soma 1 a SI e DI e subtrai 1 em caso contrário

Manipulando strings

- Copiar uma string de **tam** bytes para de **str1** para **str2**

```
mov si, str1
mov di, str2
mov cx, tam
cld
rep movsb
```

Manipulando strings

- Copiar uma string mais rapidamente com movsw (16 bits= 2 bytes por vez). Se **tam** é impar o byte isolado remanescente pode ser copiado com uma instrução movsb final

```
mov si,  str1
mov di, str2
mov cx, tam
cld
shr cx,1        ; ?
rep movsw
rcl cx,1
rep movsb
```

- Comparando strings

```
mov si, str1
mov di,str2
mov cx, tam
cld
rep cmpsb
```

- *Após a comparação ZF = 1, se identicos, ZF = 0 se diferentes*
- *Se ZF = 0, SF = 1 se str1 < str2 , SF=0 em caso contrário*

Manipulando strings

- Inicializando o vetor com **tam** cópias de **ch**

```
mov di, str2
mov cx, tam
mov al,ch
cld
rep stosb
```

Manipulando strings

- Localizando primeira ocorrência de **ch**

```
mov di,str2
mov cx, tam
mov al,ch
cld
repnz scasb
```

- *ZF = 0 se não foi encontrado*
- *ES:DI aponta para a posição um byte além do caractere*

Manipulando strings

- Tamanho de uma string (NINJA TRICK)

```
mov di, str1
xor al,al    ;procura nulo = '\0'
mov cx, -1
cld                                            ;    0  1   2   3
repnz scasb ; cx = - tamanho –2        ;  | c | o | m | p | 0 |
            ;                                   -2  -3  -4  -5  -6      ==> -6d=1111 1010b
not cx       ; cx = tamanho + 1                                         ==> 0000 0101b=5d
dec cx       ; cx = tamanho
```

Para encontrar – 5d em binário
5d = 0101b. Faz 0101b = 1010b
Adiciona 1

```
1010b          (-6d)
 +   1b
---------
 1011b          (-5d)
```

Tomando 1011b = 0100b = 4d