Assembly genérico

Comando geral:

```
[label] mnemonic [operands] [/comment]
```

Constante:

```
CR EQU ODH
```

mesma coisa

```
CR EQU 13
```

Alocação de dados

Em C:

```
char     response  // aloca 1 byte
int      value     // aloca 4 bytes
float    total     // aloca 4 bytes
double   temp      // aloca  8 bytes
```

Formato geral dos comandos de alocação

[variable-name] define-directive initial-value [,initial-value],•• •

```
DB Define Byte       ; aloca 1 byte
DW Define Word       ; aloca 2 bytes
DD Define Doubleword ; aloca 4 bytes
DQ Define Quadword   ; aloca 8 bytes
DT Define Ten Bytes  ; aloca 10 bytes
```

Mesma coisa

```
sorted DB 'y '
```
ou
```
sorted DB 79H
```
ou
```
sorted DB 1111001b
```

Dados não inicializados

```
RESB Reserve a Byte
RESW Reserve a Word
RESD Reserve a Doubleword
RESQ Reserve a Quadword
REST Reserve Ten Bytes
```

Exemplos:

```
response RESB 1    ;   char response
buffer RESW 100    ;   int buffer[100]
total RESD 1       ;   float total
```

Dados multiplos inicializados:

```
sort  DB 'Y'       ;ASCII of y = 79H
value DW 25159     ;25159D = 6247H
total DD 542803535 ;542803535D = 205A864FH
```

Mesma coisa

```
message       DB      'W'
              DB      'E'
              DB      'L'
              DB      'C'
              DB      'O'
              DB      'M'
              DB      'E'
              DB      '!'
```

ou

```
message DB  'W' , 'E ' ,'L' ,' C , '0' , 'M' , 'E ' , ' ! '
```

ou

```
message DB  'WELCOME!'
```

Vetor de zeros

```
marks TIMES 8 DW 0  ; marks[0]=0, marks[1]=0, .., marks[7]=0
```

Modo de endereçamento

```
mov destination,source
mov EAX,EBX
```

Imediato

```
mov EAX,75
```

Endereçamento direto

```
response DB ' Y'        ; allocates a byte, initializes to Y
tablel TIMES 20 DW 0    ; allocates 40 bytes, initializes to 0
namel DB 'Jim Ray'      ;7 bytes are initialized to Jim Ray
```

Alguns exemplos da instrução mov:

```
mov AL,[response]   ; copia Y no registrador AL
mov [response],'N'  ; N é copia para response
mov [namel],'K'     ; name1[0]='K'
mov [tablel],56     ;table1[0]=56
```

Endereçamento indireto

```
mov EBX,tablel    ;copia o endereço de tablel para EBX
mov [EBX],100      ;tablel[0]=100
add EBX, 2         ;EBX = EBX + 2
mov [EBX],99       ;tablel[l] = 99
```

`lea EBX, [tablel]` pode ser usado no lugar de
`mov EBX,tablel`

A diferença é que lea calcula os valores de deslocamento em tempo de execução, enquanto a versão mov resolve o valor de deslocamento no tempo de montagem. Por esta razão, vamos tentar usar o último, sempre que possível. No entanto, lea oferece mais flexibilidade quanto aos tipos de operandos fonte.

Por exemplo:

```
lea EBX,[array+ESI ]
```

Enquanto que

```
mov EBX, [array+ESI] ; ilegal
```

dá pau

Mais exemplos de mov

```
mov AL,[response]
mov DX, [tablel]
mov [response],'N'
mov [namel+4],'K'
```

Ambiguidade no mov

Mover valores imediatos para a memória algumas vezes causa ambiguidade sobre o tipo de operando.
Por exemplo, os comandos:

```
mov EBX,[table1]
mov ESI,[name1]
mov [EBX],100
mov [ESI],100
```

Não fica claro se temos que salvar 100 em um byte ou numa word.

Podemos ser mais específicos:

```
mov WORD [EBX],100
mov BYTE [ESI],100
```

Podemos também escrever assim:

```
mov [EBX],WORD 100
mov [ESI],BYTE 100
```

Tipo	Bytes endereçados
BYTE	1
WORD	2
DWORD	4
TBYTE	10

Aritmética simples

```
inc EBX  ; increment 32-bit register EBX=EBX+1
dec DL    ; decrement 8-bit register  DL=DL-1
```

Mais ambiguidade

```
.DATA
count    DW    0
value    DB    25

.CODE
      inc    [count]       ;unambiguous
      dec    [value]       ;unambiguous
      move   EBX,count
      inc    [EBX]         ;ambiguous
      mov    ESI,value
      dec    [ESI]         ;ambiguous
```

As duas primeiras operações aritméticas são claras, o compilador sabe o tipo que são count e value. Mas as operações com EBX e ESI não. Então vamos ser claros:

```
inc WORD [EBX]
dec BYTE [ESI]
```

Por que não ADD X,1 e SUB X,1?

ADD e SUB gastam mais memória que INC e DEC. E queremos escovar bits.

jump incodicional

O jump incodicional diz ao processador para executar a próxima instrução localizada num determinado label.

```
        mov     EAX,1
inc_again:
        inc     EAX
        jmp     inc_again
        mov     EBX,EAX
            . . .
```

Alguns tipos de jumps:

`je`	jump if equal
`jg`	jump if greater
`jl`	jump if less
`jge`	jump if greater or equal
`jle`	jump if less than or equal
`jne`	jump if not equal

Conditional jumps can also test the values of flags. Some examples are

`jz`	jump if zero (i.e., if ZF = 1)
`jnz`	jump if not zero (i.e., if ZF = 0)
`jc`	jump if carry (i.e., if CF = 1)
`jnc`	jump if not carry (i.e., if CF = 0)

Uso do for com assembly:

```
      mov    CL,50
repeat1:
      <loop body>
      dec    CL
      jnz    repeat1  ;jumps back to repeat1 if CL is not 0
         . . .
         . . .
```

Instruções lógicas

```
AND  AX,BX
OR   AX,DX
XOR  CX,10 ; ou exclusivo
NOT  BX    ; inverte os bits
```

INTERESSANTE:
xor reg,reg ; é o mesmo que mov reg,0 só que mais rápido e usa menos memória

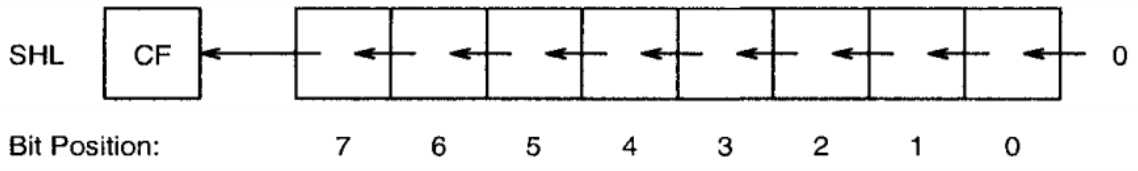
Instrução xch
Troca os valores dos registradores. Não necessita de um terceiro registrador ou variável.

```
xchg    EAX,EDX
xchg    [response],CL
xchg    [total],DX
```

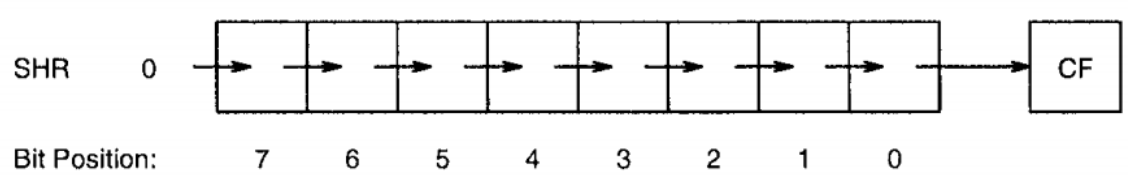
mas

```
xchg [response] , [namel] ; illegal
```

Instruções de deslocamento - SH



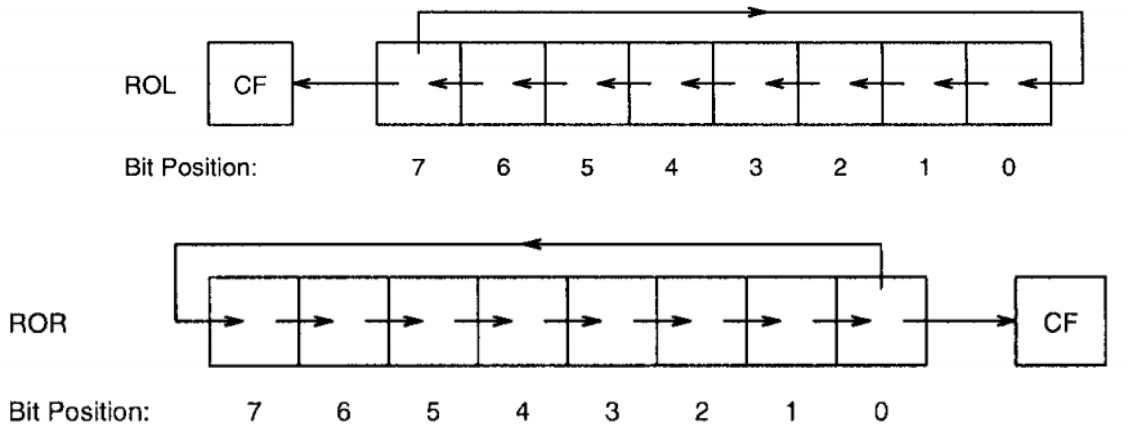
Desloca os bits para a esquerda, aqueles que passarem da posição 7. O flag CF armazena o valor do último bit deslocado. ***Deslocar um bit para a esquerda é o mesmo que multiplicar o valor desse local por 2***.



Desloca os bits para a direita CF guarda o último bit. ***Deslocar um bit para direita equivale a dividir por 2.***

```
shl  destination,count      shr  destination,count
shl  destination,CL         shr  destination,CL
```

Rotação de bits.
A instrução RO e RO rotaciona os bits. Diferente de SH, não perde os bits.



Corpo de uma macro

```
%macro     macro_name   para_count
           <macro body>
%endmacro
```

Passagem de parâmetros no procedimento

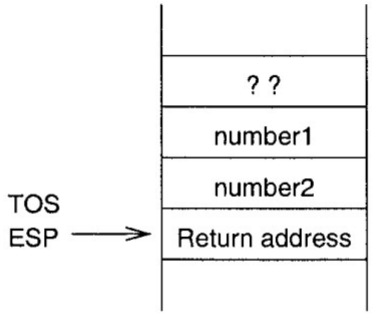


Figure 11.7 Stack state after the `sum` procedure call: Return address is the EIP value pushed onto the stack as part of executing the call instruction.

Passando dois parâmetros para o procedimento sum

```
push number1
push number2
call sum
```

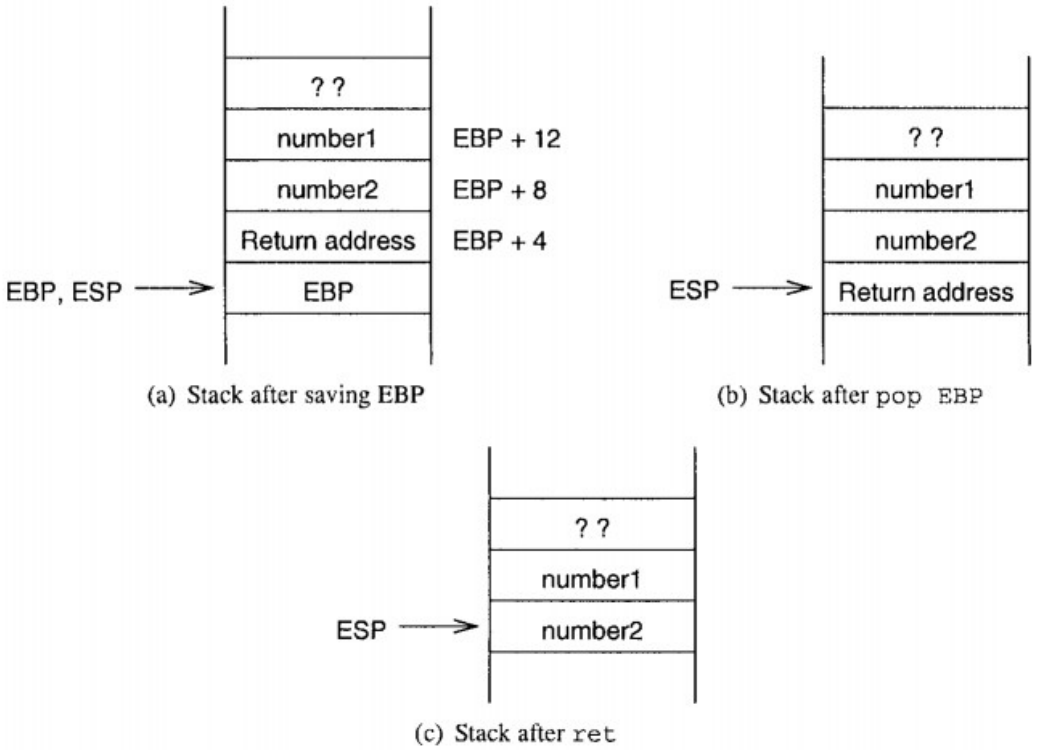


Figure 11.8 Changes in stack state during a procedure execution.

```
push num1
push num2
call soma
pop num1
pop num2


soma:
   push ebp
   mov ebp,esp
   mov eax,[ebp+8]
   mov ebx,[ebp+12]
   add eax,ebx
ret
```

MUL val
Multiplica EAX por val. O resultado fica em EDX:EAX

DIV val
Divide EAX por val. O resultado fica em EAX, e a sobra fica em EDX