CONJUNTO DE INSTRUÇÕES DO 8086 / 88

As instruções do 8086 / 88 podem ser agrupadas em classes, de acordo com o tipo de trabalho que executam.
Esses grupos podem ser classificados em:

1) Instruções para movimentação de dados
2) Instruções aritméticas
3) Instruções lógicas
4) Instruções de desvio
5) Instruções de controle.

Outras classificações podem ser encontradas, mas iremos adotar a acima citada.

1) Instruções para movimentação de dados

São instruções do tipo MOV, PUSH, POP, IN, OUT, XCHG, LOAD (LEA, LDS, LES, LAHF, SAHF) e XLAT.

1.1 MOV op1, op2

Copia do operando fonte (OP2) para operando destino (OP1). Altera apenas o operando destino. Não altera reg de flags.

a) MOV mem, acc

```
Ex:   MOV [1000H], AX
      MOV [16], AL
```

b) MOV acc, mem

```
Ex:   MOV AX, [32],
      MOV AL, ES:[1000H] (é uma instrução de 4 bytes porque tem um
```

byte para especificar ES como segmento)

c) MOV seg, mem ou MOV seg, reg

Neste caso, o destino é um registrador de segmento e a fonte pode ser um registrador de 16 bits (AX, BX, CX, DX, SI, DI, SP, BP) ou uma posição de memória (16 bits). **CS não pode ser o operando destino em nenhuma instrução de transferência de dados.**

O tamanho do código da instrução (tam) é de 2 bytes no caso de endereçamento de registrador, 2 bytes mais o número de bytes usados para especificar o deslocamento (0, 1 ou 2 bytes) no caso do endereçamento indireto e 4 bytes no caso do endereçamento direto.

```
Ex.:  MOV ES, BX              ; tam = 2.
      MOV DS, [BX]            ; tam = 2 + 0 = 2.
      MOV SS, [SI]            ; tam = 2 + 0 = 2.
      MOV DS, [BX+DI-100]     ; tam = 2 + 1 = 3.
      MOV ES, [BP+10]         ; tam = 2 + 1 = 3.
      MOV DS, [10]            ; tam = 4.
```

d) MOV mem,seg ou MOV reg,seg

Neste caso a fonte é um registrador de segmento, e o destino pode ser um registrador de 16 bits (AX, BX, CX, DX, SI, DI, SP, BP) ou uma posição de memória (16 bits).

Observa-se que o campo seg pode assumir qualquer valor, portanto o registrador CS também pode se utilizado como fonte de uma transferência de dados (não pode ser destino apenas).

```
Ex.:   MOV AX, DS     ; tam = 2.
       MOV [SI], CS   ; tam = 2.
       MOV [500], DS  ; tam = 4.
```

e) MOV reg, reg ou MOV reg, mem ou MOV mem, reg: Neste caso a fonte e o destino podem ser um registrador de 8 ou 16 bits (AX, BX, CX, DX, SI, DI, SP, BP, AH, AL, BH, BL, CH, CL, DH, DL) ou uma posição de memória de 8 ou 16 bits. Observa-se contudo que fonte e destino não podem ser ambos posições de memória em uma mesma instrução, ou seja **não existe MOV mem, mem**.

```
Ex.:   MOV [BX+SI], AX        ; tam = 2.
       MOV AX, [BX+SI]        ; tam = 2.
       MOV AX, CX             ; tam = 2.
       MOV BL, [BP+DI-128]    ; tam = 3.
```

f) MOV reg, val: Neste caso, o destino é um registrador e a fonte é um valor imediato. Um valor imediato é um número codificado na própria istrução e que é carregado no operando de destino diretamente.

```
Ex.:   MOV AL, 16     ; carrega o valor 10H em AL. tam = 2 bytes.
       MOV BX, -1     ; carrega o valor FFFFH em BX. tam = 3 bytes.
```

g) MOV reg/mem, val: Carrega um valor imediato em um registrador ou em uma posição de memória.

Ex: MOV [1000H], 25H

1.2 **PUSH** OP "Empilha OP"

Esta instrução permite copiar um operando de 16 bits na pilha. O funcionamento desta instrução é o seguinte:

i) O registrador SP é decrementado duas vezes (SP ← SP - 2).

ii) OP é copiado para posição de memória (16 bits) de endereço SS:[SP], ou seja o byte menos significativo de OP é escrito no endereço SS:[SP] enquanto o byte mais significativo de OP é escrito no endereço SS:[SP+1].

Como exemplo, sejam SS = 1000H e SP = 0000H. A instrução PUSH AX irá alterar SP = FFFEH e escrever AH na memória no endereço linear 1FFFFH e AL no endereço linear 1FFFEH. Se em seguida for executada PUSH BX, BH será escrito no endereço 1FFFDH e BL no endereço 1FFFCH enquanto SP é alterado para SP = FFFCH. A figura 3.13 ilustra este exemplo. Nota-se na figura 3.13 que SP sempre aponta para o último dado empilhado, daí o seu nome "Stack Pointer". Observa-se também que a pilha no 8086 (8088) cresce "para baixo", ou seja, na direção dos endereços menores.

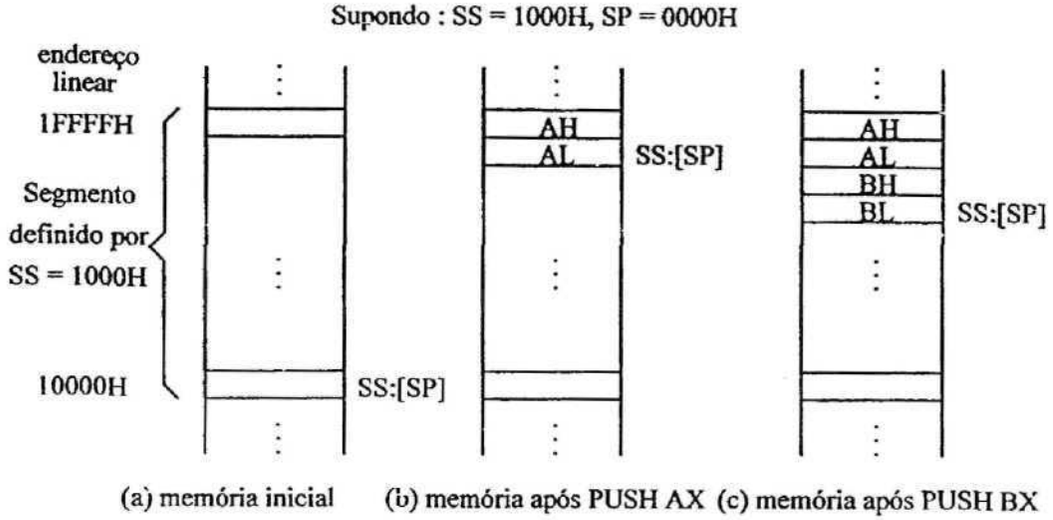

Figura 3.13 - Funcionamento da pilha do 8086 (8088).

Existem tres códigos para PUSH:

a) PUSH reg: Neste caso o operando é um registrador de 16 bits (AX, BX, CX, DX, SI, DI, SP, BP).

b) PUSH seg: Neste caso o operando é um registrador de segmento

c) PUSH reg/mem: Neste caso o operando é um registrador de 16 bits ou uma posição de memória de 16 bits.

Observando-se as instruções PUSH nota-se que:

i) Não é possível colocar na pilha um registrador ou posição de memória de 8 bits.
ii) A instrução PUSH permite, ao contrário de MOV, a transferência de uma posição de memória para a outra. Isso acontece porque as instruções do 8086 (8088) possuem no máximo 1 byte de endereçamento com os campos mod e r/m. Por isso não é possível MOV mem, mem pois seriam necessários dois bytes de endereçamento para especificar as duas posições de memória. Já na instrução PUSH, o destino na memória é determinado por SS:[SP], e portanto apenas um byte de endereçamento é usado para especificar a fonte, permitindo assim codificar PUSH mem que é uma transferência de memória para memória.

```
Ex.:   PUSH AX        ; tam = 1.
       PUSH CS        ; tam = 1.
       PUSH [SI + 10] ; tam = 3.
```

1.3 PUSHF "Empilha o registrador de flags"

Esta instrução coloca na pilha o conteúdo do registrador de flags. O registrador de flags FR é o único registrador do 8086 (8088) que é endereçável bit a bit. Os bits do FR tem funções distintas e são usados para controlar o modo de execução de algumas instruções do microprocessador. O registrador de flags está ilustrado na figura 3.17.

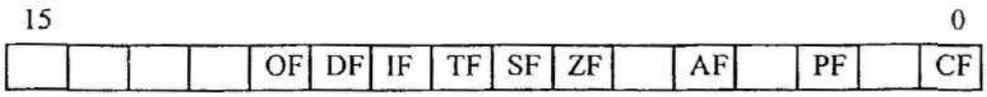

Figura 3.17 - O registrador de flags.

O significado dos bits do registrador de flags é o seguinte:

CF "Carry Flag" - Indica quando ocorre um transporte do bit mais significativo do resultado após a execução de uma instrução aritmética.
PF "Parity Flag" - Indica a paridade do resultado da última operação (par - PF = 1, ímpar - PF = 0).
AF "Auxiliary Flag" - Indica quando ocorre um transporte do bit 4 do resultado após a execução de uma instrução aritmética.
ZF "Zero Flag" - Indica que o resultado da última operação foi zero (ZF = 1 indica resultado nulo).
SF "Sign Flag" - Indica o sinal do resultado (SF = 0 indica resultado positivo).
TF "Trap Flag" - Quando setado, habilita a operação "passo a passo".
IF "Interrupt Flag"- Quando IF = 1 as interrupções de hardware via pino INTR são habilitadas.
DF "Direction Flag" - Controla o sentido da transferência em instruções de manipulação de strings.
OF "Overflow Flag" - Quando igual a 1, indica que o sinal do resultado está errado.

1.4 POP OP "Desempilha OP"

Esta instrução permite mover para um operando de 16 bits o conteúdo no topo da pilha. O funcionamento desta instrução é o seguinte:

i) O conteúdo da posição de memória (16 bits) de endereço SS:[SP] é copiada para OP, ou seja o byte no endereço SS:[SP] é escrito no byte menos significativo de OP enquanto o byte no endereço SS:[SP+1] é escrito sobre o byte mais significativo de OP.

ii) O registrador SP é incrementado duas vezes (SP ← SP + 2).

Existem 3 códigos para POP:

a) POP reg: Neste caso o operando é um registrador de 16 bits (AX, BX, CX, DX, SI, DI, SP, BP).

b) POP seg: Neste caso o operando é um registrador de segmento. **O registrador CS não pode ser o operando de POP.**

c) POP reg/mem: Neste caso o operando é um registrador de 16 bits ou uma posição de memória de 16 bits.

```
Ex.:   POP AX        ; tam = 1.
       POP DS        ; tam = 1.
       POP [SI]      ; tam = 2.
```

1.5 POPF "Desempilha o registrador de flags"

Esta instrução transfere o conteúdo do topo da pilha, word em SS:[SP], para o registrador de flags FR e em seguida incrementa SP de duas unidades. Obviamente esta instrução altera o conteúdo de todos os bits do registrador de flags e por isso POPF é a única instrução de transferência que afeta os flags.

1.6 XCHG OP1, OP2 "Trocar OP1 com OP2"

Esta instrução permite trocar o conteúdo de um registrador ou posição de memória com um outro registrador. Existem dois tipos:

a) XCHG AX, reg ou XCHG reg, AX: Neste caso o conteúdo do AX é trocado com o conteúdo de um registrador de 16 bits.

b) XCHG mem/reg, reg ou XCHG reg, mem/reg: Neste caso o conteúdo de um registrador de 8 ou 16 bits é trocado com o conteúdo de outro registrador ou posição de memória.

```
Ex.:   XCHG AL, CH        ; tam = 2.
       XCHG AX, CX        ; tam = 1.
       XCHG BP, [BX+10]   ; tam = 3.
```

1.7 **IN** OP1, OP2 "Lê um dado do espaço de IO"

As instruções de movimentação de dados em geral são capazes de endereçar dados em registradores ou no espaço de endereços de memória. Para acessar o espaço de endereços de IO, o 8086 (8088) possui apenas duas instruções: IN para ler um dado em IO e OUT para escrever. A instrução IN possui dois códigos:

a) IN acc, port: Neste caso, o destino é o acumulador (AL ou AX) e a fonte é o dado no endereço "port" no espaço de IO (port está no intervalo [0, 255]).

Esta instrução faz endereçamento direto no espaço de IO, e por possuir apenas um byte de endereço está limitada aos 256 endereços mais baixos deste espaço.

```
Ex.:   IN AL, 16      ; Lê o byte no endereço 10H de IO e grava em AL.
       IN AX, 16      ; Lê o byte no endereço 10H de IO e grava em AL e lê o byte
                      ; no endereço 11H de IO e grava em AH.
```

b) IN acc, DX: Neste caso o destino é o acumulador (AL ou AX) e a fonte é o dado no endereço de IO especificado pelo conteúdo de DX. Como DX é um registrador de 16 bits, esta instrução permite acessar os 65536 endereços disponíveis no espaço de IO.

Esta instrução faz endereçamento indireto no espaço de IO.

```
Ex.:   MOV DX, 16     ; Carrega DX com o valor 10H.
       IN AL, DX      ; Lê o byte no endereço 10H de IO e grava em AL.
```

1.8 **OUT** OP1, OP2 "Escreve um dado no espaço de IO"

Esta instrução escreve dados no epaço de IO. A instrução OUT possui dois códigos:

a) OUT port, acc: Neste caso, a fonte é o acumulador (AL ou AX) e o destino é o dado no endereço "port" no espaço de IO (port está no intervalo [0, 255]).

Esta instrução faz endereçamento direto no espaço de IO, e por possuir apenas um byte de endereço está limitada aos 256 endereços mais baixos deste espaço.

```
Ex.:   OUT 16, AL     ; Escreve o conteúdo de AL no byte de endereço 10H.
       OUT 16, AX     ; Escreve AL no endereço 10H de IO e AH no endereço 11H.
```

b) OUT DX, acc: Neste caso a fonte é o acumulador (AL ou AX) e o destino é o dado no endereço de IO especificado pelo conteúdo de DX. Como DX é um registrador de 16 bits, esta instrução permite acessar os 65536 endereços disponíveis no espaço de IO.

Esta instrução faz endereçamento indireto no espaço de IO.

```
Ex.:   MOV DX, 16     ; Carrega DX com o valor 10H.
       OUT DX, AL     ; Escreve o conteúdo de AL no byte de endereço 10H.
```

1.9 **LEA** reg, EA "Carrega endereço efetivo em reg"

Esta instrução calcula o endereço efetivo de um operando na memória e armazena o resultado em um registrador.

```
Ex.:   LEA BX, [SI+BP-12]   ; BX recebe o resultado da soma SI+BP-12.
```

O seguinte código é equivalente à instrução acima:

```
MOV BX, SI          ;tam = 2.
ADD BX, BP          ; soma BP à BX. tam = 2.
ADD BX, -12         ; soma -12 à BX. tam = 4.
```

1.10 LDS reg, mem "Transfere 4 bytes consecutivos da memória para reg e DS"

Esta instrução carrega um registrador de 16 bits e o registrador de segmento DS com valores armazenados na memória.

```
Ex.:  LDS BX, [SI-10]   ; Transfere a palavra no endereço DS:[SI-10] para BX.
                        ; Em seguida transfere a palavra em DS:[SI-8] para DS.
                        ;tam = 3.
```

O seguinte código é equivalente à LDS BX, [SI-10]:

```
MOV BX, [SI-10]     ; tam = 3.
MOV DS, [SI-8]      ; tam = 3.
                    ; total tam = 6.
```

1.11 LES reg, mem "Transfere 4 bytes consecutivos da memória para reg e ES"

Esta instrução, similar à LDS, carrega um registrador de 16 bits e o registrador de segmento ES com valores armazenados na memória

```
Ex.:  LES BX, [SI-10]   ; Transfere a palavra no endereço DS:[SI-10] para BX.
                        ; Em seguida transfere a palavra em DS:[SI-8] para ES.
```

1.12 XLAT "Converte o conteúdo de AL usando uma tabela em DS:[BX]"

Usada para conversão de código. O valor de AL é usado para indexar uma tabela. O byte na memória cujo endereço é DS:[BX + AL] é transferido para AL.

Ex.: Supondo-se que AL = 02H, BX = 0000H, e o conteúdo da memória em DS:0000, DS:0001 ... etc. é respectivamente: 00H, 02H, 04H, 06H, ... então após a execução de XLAT AL = 04H.

1.13 LAHF "Transfere para AH os bits de flag SF, ZF, AF, PF e CF"

Esta instrução foi criada para facilitar a tradução de programas de 8080 (8085) para 8086 (8088).

```
Ex.:  LAHF      ; Neste ponto AH = xSZAxPxC, onde S= SF, Z = ZF, A = AF
                ; P = PF e C = CF.
```

1.14 SAHF "Transfere para os bits de flag SF, ZF, AF, PF e CF o conteúdo de AH"

Esta instrução foi criada para facilitar a tradução de programas de 8080 (8085) para 8086 (8088).

2) Instruções aritméticas

Nestas instruções, um dado não é simplesmente transferido de um operando para outro como nas instruções de movimentação de dados, mas é alterado durante a transferência. Estas instruções afetam uma série de bits no registrador de flags, em geral aqueles associados ao resultado de operações aritméticas como o CF, o AF, o SF, o PF, o ZF e o OF. Contudo nem todos são afetados da mesma forma por todas as instruções aritméticas, sendo o efeito sobre os flags indicado em cada instrução a seguir.

2.1 ADD OP1, OP2 "Soma OP2 a OP1 e armazena o resultado em OP1"

Esta instrução executa o algoritmo da soma de inteiros não sinalizados sobre OP1 e OP2 (Observa-se que é possível portanto utilizá-la para somar dois inteiros sinalizados usando a representação em complemento a dois). Os bits de flags afetados são: AF, CF, OF, PF, SF e ZF. Existem 3 códigos para ADD:

a) ADD reg, reg ou ADD mem, reg ou ADD reg, mem

```
Ex.:    ADD AX, BX          ; Se por exemplo AX = FFFEH e BX = 0001H antes da
                            ; execução desta instrução, então AX = FFFFH
                            ; após a execução. Nota-se que o resultado está correto se
                            ; for considerada a representação sem sinal
                            ; (65534 + 1 = 65535) ou com sinal em
                            ; complemento a dois (-2 + 1 = -1).
        ADD [SI], CX
        ADD DI, [BX + DI - 10]
```

b) ADD reg/mem, val: Neste caso um valor imediato val é somado ao conteúdo de um registrador ou posição de memória.

```
Ex.:    ADD BX, 10
        ADD byte ptr [SI], 20
```

c) ADD acc, val: Neste caso um valor imediato val é somado ao conteúdo do acumulador (AX ou AL).

```
Ex.:    ADD AX, 10
        ADD AL, 10
```

2.2 ADC OP1, OP2 "Soma OP2 a OP1 considerando CF e armazena o resultado em OP1"

Esta instrução executa o algoritmo da soma de inteiros não sinalizados sobre OP1 e OP2 e em seguida soma o flag de carry CF ao resultado. Esta instrução permite executar somas de inteiros com ou sem sinal de precisão maior do que 16 bits. Os bits de flags afetados são: AF, CF, OF, PF, SF e ZF. Existem 3 códigos para ADC:

a) ADC reg, reg ou ADC mem, reg ou ADC reg, mem

Ex.: Deseja-se somar dois números de 32 bits. Cada um dos números está em 4 bytes consecutivos na memória. O endereço do primeiro número é DS:BX e o endereço do segundo número é DS:SI. O seguinte código pode ser utilizado para somar os dois números e armazenar o resultado em DS:BX:

```
MOV     AX, [BX]    ;Carrega em AX os dois bytes menos significativos do 1º número.
ADD     AX, [SI]    ;Soma a parte menos significativa dos dois números em AX.
MOV     [BX], AX    ;Atualiza os dois bytes menos significativos do 1º número com
                    ;o resultado da soma.
```

```
MOV   AX, [BX + 2]   ;Carrega em AX os dois bytes mais significativos do 1º número.
ADC   AX, [SI + 2]   ;Soma a parte mais significativa dos dois números em AX, porém
                     ;considerando o flag de transporte (vai-um ou "carry") CF.
MOV   [BX + 2], AX   ;Atualiza os dois bytes mais significativos do 1º número com
                     ;o resultado da soma.
```

OBS.: **É importante observar que o código acima funciona porque as duas instruções MOV executadas entre a instrução ADD e a instrução ADC não afetam o CF.** Isto é importante porque para que a soma esteja correta o CF considerado na execução de ADC tem que ser o "vai-um" gerado pela instrução ADD. Assim deve-se sempre observar que flags são afetados pelas instruções usadas em um programa, principalmente quando uma determinada instrução depende de algum bit de flag como é o caso de ADC.

Para ilustrar, o exemplo abaixo é uma tentativa de somar os dois números de 32 bits mas que **não funciona**.

```
MOV   AX, [BX]       ;Carrega em AX os dois bytes menos significativos do 1º número.
ADD   AX, [SI]       ;Soma a parte menos significativa dos dois números em AX.
MOV   [BX], AX       ;Atualiza os dois bytes menos significativos do 1º número com
                     ;o resultado da soma.
ADD   BX, 2          ;Faz BX apontar a parte mais significativa do 1º número.
ADD   SI, 2          ;Faz SI apontar a parte mais significativa do 2º número.
MOV   AX, [BX]       ;Carrega em AX os dois bytes mais significativos do 1º número.
ADC   AX, [SI]       ;Soma a parte mais significativa dos dois números em AX, porém
                     ;considerando o flag de transporte (vai-um ou "carry") CF.
                     ;Observa-se contudo que o CF contém o resultado da operação
                     ;ADD SI, 2 e não de ADD AX, [SI] que era o desejado.
MOV   [BX], AX       ;Atualiza os dois bytes mais significativos do 1º número com
                     ;o resultado da soma (errado).
```

b) ADC reg/mem, val: Neste caso um valor imediato val é somado ao conteúdo de um registrador ou posição de memória. O CF é adicionado ao resultado.

```
Ex.:  ADC BX, 10
      ADC byte ptr [SI], 20
```

c) ADC acc, val: Neste caso um valor imediato val é somado ao conteúdo do acumulador (AX ou AL) mais o valor de CF.

```
Ex.:  ADC AX, 10
      ADC AL, 10
```

2.3 SUB OP1, OP2 "Subtrai OP2 de OP1 e armazena o resultado em OP1"

Esta instrução executa o algoritmo da subtração de inteiros não sinalizados sobre OP1 e OP2 (Observa-se que é possível portanto utilizá-la para subtrair dois inteiros sinalizados usando a representação em complemento a dois). Os bits de flags afetados são: AF, CF, OF, PF, SF e ZF. O CF neste caso é interpretado como empréstimo ou "borrow". Existem 3 códigos para SUB:

a) SUB reg, reg ou SUB mem, reg ou SUB reg, mem:

```
Ex.:  SUB AX, BX          ; Se por exemplo AX = FFFFH e BX = 0001H antes da
                          ; execução desta instrução, então AX = FFFEH
                          ; após a execução. Nota-se que o resultado está correto se
                          ; for considerada a representação sem sinal
                          ; (65535 - 1 = 65534) ou com sinal em
                          ; complemento a dois (-1 - 1 = -2).
      SUB [SI], CX
      SUB DI, [BX + DI - 10]
```

b) SUB reg/mem, val: Neste caso um valor imediato val é subtraído do conteúdo de um registrador ou posição de memória.

```
Ex.:  SUB BX, 10
      SUB byte ptr [SI], 20
```

c) SUB acc, val: Neste caso um valor imediato val é subtraído do conteúdo do acumulador (AX ou AL).

```
Ex.:  SUB AX, 10
      SUB AL, 10
```

2.4 SBB OP1, OP2 "Subtrai (OP2 + CF) de OP1 e armazena o resultado em OP1"

Esta instrução executa o algoritmo da subtração de inteiros não sinalizados sobre OP1 e OP2 (Observa-se que é possível portanto utilizá-la para subtrair dois inteiros sinalizados usando a representação em complemento a dois). A diferença com relação à SUB é que o valor do bit CF é subtraído do resultado. Analogamente à ADC, pode ser usada para efetuar subtrações de inteiros com precisão maior do que 16 bits. Os bits de flags afetados são: AF, CF, OF, PF, SF e ZF. O CF neste caso é interpretado como empréstimo ou "borrow". Existem 3 códigos para SBB:

a) SBB reg, reg ou SBB mem, reg ou SBB reg, mem:

Ex.: Deseja-se subtrair dois números de 32 bits. Cada um dos números está em 4 bytes consecutivos na memória. O endereço do primeiro número é DS:BX e o endereço do segundo número é DS:SI. O seguinte código pode ser utilizado para subtrair os dois números e armazenar o resultado em DS:BX:

```
MOV   AX, [BX]      ;Carrega em AX os dois bytes menos significativos do 1º número.
SUB   AX, [SI]      ;Subtrai a parte menos significativa dos dois números em AX.
MOV   [BX], AX      ;Atualiza os dois bytes menos significativos do 1º número com
                    ;o resultado da subtração.
MOV   AX, [BX + 2]  ;Carrega em AX os dois bytes mais significativos do 1º número.
SBB   AX, [SI + 2]  ;Subtrai a parte mais significativa dos dois números em AX,
                    ;porém considerando o flag de transporte (vai-um ou "carry") CF.
MOV   [BX + 2], AX  ;Atualiza os dois bytes mais significativos do 1º número com
                    ;o resultado da subtração.
```

b) SBB reg/mem, val: Neste caso um valor imediato val é subtraído do conteúdo de um registrador ou posição de memória.

```
Ex.:  SBB BX, 10
      SBB byte ptr [SI], 20
```

c) SBB acc, val: Neste caso um valor imediato val é subtraído do conteúdo do acumulador (AX ou AL).

```
Ex.:  SBB AX, 10
      SBB AL, 10
```

2.5 DAA "Ajuste decimal para adição"

Esta instrução é usada para corrigir no AL o resultado da soma de dois números BCD compactados efetuada usando a instrução ADD.

Quando o algoritmo da soma de inteiros (instrução ADD) é usado para somar números BCD, o resultado pode estar incorreto de duas formas:

i)
```
  00101001    (29)
+ 00010100    (14)
  00111101    (3?)
```

Neste caso, os 4 bits menos significativos do resultado não correspondem a um número BCD (obviamente os 4 bits mais significativos também poderiam estar errados).

Isto ocorre porque deveria ter ocorrido um transporte (vai um) para o bit 4 quando o valor dos 4 bits menos significativos ultrapassou 9. No entanto, usando o algoritmo da soma de inteiros da instrução ADD, o transporte só ira ocorrer quando este valor ultrapassar 15.

Para corrigir o resultado basta adiantar o transporte somando 6 ao resultado, ou seja:

```
  00101001    (29)
+ 00010100    (14)
  00111101    (3?)    - →   incorreto.
+ 00000110    (06)
  01000011    (43)    →     correto.
```

Se o dígito errado fosse o mais significativo bastaria somar 60 e se fossem ambos bastaria somar 66.

ii)
```
  00101001    (29)
+ 00011000    (18)
  00111101    (41)
```

Neste caso os dois dígitos correspondem a representações BCD válidas, no entanto o resultado está incorreto. Isto ocorre porque deveria ter ocorrido um transporte (vai um) para o bit 4 quando o valor dos 4 bits menos significativos ultrapassou 9. No entanto, por ter sido usado o algoritmo da soma de inteiros da instrução ADD, o transporte só ocorreu quando este valor ultrapassou 15. A ocorrência do transporte do bit 4 é indicada pelo bit de flag AF = 1. Para corrigir o resultado basta adiantar o transporte somando 6 ao resultado, ou seja:

```
  00101001    (29)
+ 00011000    (18)
  00111101    (41)    →     incorreto (indicado por AF = 1).
+ 00000110    (06)
  01000111    (47)    →     correto.
```

Para efetuar a correção do resultado, aplica-se DAA logo após o uso de ADD.

```
Ex.:  MOV  BL, 29H    ; BL contém a representação BCD do número decimal 29.
      MOV  AL, 14H    ; AL contém a representação BCD do número decimal 14.
      ADD  AL, BL     ; AL contém 3DH, que é inválido como BCD.
      DAA             ; AL contém a 43H que é a representação BCD
                      ; correta para o resultado.
```

Os flags afetados são: AF, CF, PF, SF, ZF.

2.6 DAS "Ajuste decimal para a subtração"

Esta instrução é análoga à DAA porém corrige o resultado (em AL) da subtração de dois números BCD compactados.

Os flags afetados são: AF, CF, PF, SF, ZF.

2.7 AAA "Ajuste ASCII para a soma"

Esta instrução é análoga à DAA porém corrige o resultado (em AL) da soma de dois números BCD não compactados.

Os flags afetados são: AF, CF, PF, SF, ZF.

2.8 AAS "Ajuste ASCII para a subtração"

Os flags afetados são: AF, CF, PF, SF, ZF.

2.9 MUL OP "Multiplica o acumulador por OP"

Esta instrução multiplica o conteúdo do acumulador (AL ou AX) pelo operando OP que pode ser um registrador ou uma posição de memória. **O algoritmo utilizado é o algoritmo da multiplicação de inteiros sem sinal.** Observa-se que esta instrução **não pode** portanto ser usada para multiplicar inteiros sinalizados usando a representação em complemento a dois. Por causa da natureza da operação, a precisão do resultado é o dobro da precisão dos operandos. Por exemplo, MUL é usada para multiplicar AL por BL (MUL BL), o resultado estará em AX. Se por outro lado MUL é usada para multiplicar AX por BX (MUL BX), o resultado, de 32 bits, será armazenado no par de registradores DXAX (a parte mais significativa do resultado em DX).

Obs.: No caso das instruções ADD, ADC, SUB e SBB, não é necessário dobrar a precisão do resultado porque, qualquer que sejam os operandos, o resultado completo sempre caberá em um registrador com a mesma precisão dos operandos aumentada de 1 bit (o CF). Isto já não é verdade no caso de MUL.

Esta instrução afeta os flags: CF, OF.

Os flags AF, PF, SF e ZF são alterados de forma imprevisível, sendo portanto indefinidos.

```
Ex.:    MUL BX                  ; Multiplica AX por BX. Resultado em DXAX.
        MUL byte ptr [SI]       ; Multiplica AL por byte na memória. Resultado em AX.
```

2.10 IMUL OP "Multiplica o acumulador por OP"

Esta instrução multiplica o conteúdo do acumulador (AL ou AX) pelo operando OP que pode ser um registrador ou uma posição de memória. **O algoritmo utilizado é o algoritmo da multiplicação de inteiros sinalizados em complemento a dois.** Observa-se que esta instrução **não pode** portanto ser usada para multiplicar inteiros não sinalizados. Por causa da natureza da operação, a precisão do resultado é o dobro da precisão dos operandos. Por exemplo se IMUL é usada para multiplicar AL por BL (IMUL BL), o resultado estará em AX. Se por outro lado IMUL é usada para multiplicar AX por BX (IMUL BX), o resultado, de 32 bits, será armazenado no par de registradores DXAX (a parte mais significativa do resultado em DX).

Esta instrução afeta os flags: CF, OF.

Os flags AF, PF, SF e ZF são alterados de forma imprevisível, sendo portanto indefinidos.

```
Ex.:    IMUL BX                 ; Multiplica AX por BX. Resultado em DXAX.
        IMUL byte ptr [SI]      ; Multiplica AL por byte na memória. Resultado em AX.
```

2.11 AAM "Ajuste ASCII para a multiplicação"

Esta instrução é usada para corrigir o resultado, presente no acumulador, da multiplicação de dois números BCD não compactados efetuada usando a instrução MUL. O resultado estará em AX (as dezenas em AH e as unidades em AL). Observa-se que apesar do nome da instrução fazer referência à ASCII, os números multiplicados usando MUL antes do ajuste não podem estar na representação ASCII (ou seja 0 = 30H, 1 = 32H etc.). Para funcionar corretamente os números multiplicados devem ter os 4 bits mais significativos zerados (ou seja 0 = 00H, 1 = 01H, etc.).

Esta instrução afeta: PF, SF, ZF.

Ficam indefinidos: AF, CF, OF.

```
Ex.:    MOV AL, 09H      ; AL contém a representação BCD do dígito 9.
        MOV BL, 08H      ; AL contém a representação BCD do dígito 8.
        MUL BL           ; AX contém 0048H.
        AAM              ; AX contém 0702H.
```

Obs.: Se AL = 39H (ASCII para 9) e BL = 38H (ASCII para 8), o seguinte código multiplica os dois dígitos ASCII:

```
        SUB AL, 30H      ; AL contém 09H.
        SUB BL, 30H      ; BL contém 08H.
        MUL BL           ; AX contém 0048H.
        AAM              ; AX contém 0702H.
        ADD AX, 3030H    ; AX contém 3732H, que é a representação ASCII do
                         ; resultado.
```

2.12 DIV OP "Divide o acumulador por OP"

Esta instrução divide o conteúdo do acumulador (AX ou DXAX) pelo operando OP que pode ser um registrador ou uma posição de memória. **O algoritmo utilizado é o algoritmo da divisão de inteiros sem sinal**. Observa-se que esta instrução **não pode** portanto ser usada para dividir inteiros sinalizados usando a representação em complemento a dois. A instrução de divisão foi projetada para desfazer o que é feito pela instrução de multiplicação. Por exemplo se DIV é usada para dividir AX por BL (DIV BL), o resultado estará em AL. Se por outro lado DIV for usada para dividir DXAX por BX (DIV BX), o resultado será armazenado em AX. No entanto, a instrução de divisão pode ser usada sem o uso prévio da instrução de multiplicação, ou seja pode haver um resto da divisão. Por isso, quando o divisor (OP) é de 8 bits, DIV divide AX por OP e coloca o resultado em AL e o resto em AH. Quando o divisor (OP) é de 16 bits, DIV divide DXAX por OP e coloca o resultado em AX e o resto em DX.

Os flags AF, CF, OF, PF, SF e ZF são alterados de forma imprevisível, sendo portanto indefinidos.

```
Ex.:    DIV BX           ; Divide DXAX por BX. Resultado em AX, resto em DX.
        DIV byte ptr [SI] ; Divide AX por byte na memória. Resultado em AL e
                         ; resto em AH.
```

2.13 IDIV OP "Divide o acumulador por OP"

Esta instrução divide o conteúdo do acumulador (AX ou DXAX) pelo operando OP que pode ser um registrador ou uma posição de memória. **O algoritmo utilizado é o algoritmo da divisão de inteiros sinalizados em complemento a dois**. Observa-se que esta instrução **não pode** portanto ser usada para dividir inteiros não sinalizados. O procedimento de divisão é o mesmo da isntrução DIV OP. O sinal do resto é sempre igual ao sinal do quociente.

Os flags AF, CF, OF, PF, SF e ZF são alterados de forma imprevisível, sendo portanto indefinidos.

```
Ex.:    IDIV BX          ; Divide DXAX por BX. Resultado em AX, resto em DX.
        IDIV byte ptr [SI] ; Divide AX por byte na memória. Resultado em AL e
                         ; resto em AH.
```

2.14 AAD "Ajuste ASCII para a divisão"

Esta instrução é usada para corrigir o dividendo, presente no acumulador, imediatamente **antes** de efetuar a divisão de dois números BCD não compactados usando a instrução DIV. O resultado estará em AX (as dezenas em AH e as unidades em AL). Observa-se que apesar do nome da instrução fazer referência à ASCII, os números em AX não podem estar na representação ASCII (ou seja 0 = 30H, 1 = 32H etc.). Para funcionar corretamente os números devem ter os 4 bits mais significativos zerados (ou seja 0 = 00H, 1 = 01H, etc.).

Esta instrução afeta: PF, SF, ZF, de acordo com o resultado em AL

Ficam indefinidos: AF, CF, OF.

```
Ex.:    MOV AX, 0702H       ; AX contém a representação BCD dos dígitos 72.
        MOV BL, 08H         ; BL contém a representação BCD do dígito 8.
        AAD                 ; AX contém 0048H.
        DIV BL              ; AL contém 09H (dígito 9 BCD) e
                            ; AH contém 00H (dígito 0 BCD).
```

2.15 CBW "Converte de byte para word"

Aumenta a precisão de um número em AL (8 bits) para 16 bits em AX. Se o bit mais significativo de AL é 0, AH recebe 00H e se o bit mais significativo de AL é 1, AH recebe FFH. Esta instrução é útil por exemplo para dividir um número sinalizado em AL pelo conteúdo de BL. Primeiro a precisão do número em AL é aumentada para 16 bits em AX. Em seguida pode-se utilizar IDIV BL para efetuar a divisão.

Nenhum flag é afetado.

```
Ex.:    MOV AL, 11          ; AL contém 0BH.
        MOV BL, 2           ; BL contém 02H.
        CBW                 ; AX contém 000BH.
        IDIV BL             ; AL contém 05H (5) e AH contém 01H (1).
```

2.16 CWD "Converte de word para double-word"

Aumenta a precisão de um número em AX (16 bits) para 32 bits em DXAX. Se o bit mais significativo de AX é 0, DX recebe 0000H e se o bit mais significativo de AX é 1, DX recebe FFFFH. Esta instrução é útil por exemplo para dividir um número sinalizado em AX pelo conteúdo de BX. Primeiro a precisão do número em AX é aumentada para 32 bits em DXAX. Em seguida pode-se utilizar IDIV BX para efetuar a divisão.

Nenhum flag é afetado.

```
Ex.:    MOV AX, 11          ; AX contém 000BH.
        MOV BX, 2           ; BX contém 0002H.
        CBW                 ; DXAX contém 0000000BH.
        IDIV BX             ; AX contém 0005H (5) e DX contém 0001H (1).
```

2.17 INC OP "Incrementa OP"

Esta instrução soma 1 ao conteúdo de OP. Os flags afetados são AF, OF, PF, SF e ZF. (não afeta o CF). Existem 2 códigos para INC:

a) INC reg/mem: Neste caso o operando é uma posição de memória ou um registrador.

```
Ex.:    INC AL
        INC word ptr [SI]
```

b) INC reg: Neste caso o operando é um registrador de 16 bits.

Ex.: INC AX
INC SI

2.18 DEC OP "Decrementa OP"

Esta instrução subtrai 1 do conteúdo de OP. Os flags afetados são AF, OF, PF, SF e ZF. (não afeta o CF). Existem 2 códigos para DEC:

a) DEC reg/mem: Neste caso o operando é uma posição de memória ou um registrador.

Ex.: DEC AL
DEC word ptr [SI]

b) DEC reg: Neste caso o operando é um registrador de 16 bits.

Ex.: DEC AX
DEC SI

2.19 CMP OP1, OP2 "Compara OP1 com OP2"

Esta é considerada uma instrução aritmética porque compara a magnitude do número em OP1 com a magnitude do número em OP2. A comparação é feita subtraindo OP2 de OP1. A instrução é idêntica à SUB no que diz respeito ao registrador de flags (Os bits de flags afetados são: AF, CF, OF, PF, SF e ZF). No entanto OP1 não é atualizado com o resultado da operação (OP1 - OP2).

Para ilustrar, seja a instrução CMP AX, BX. Esta instrução subtrai BX de AX mas não atualiza AX com o resultado, afetando apenas o registrador de flags. Após a execução desta instrução é possível portanto tirar conclusões a respeito da magnitude relativa de AX e BX apenas observando o FR. Por exemplo, se ZF = 1 pode-se concluir que o conteúdo de AX é igual ao conteúdo de BX. Se AX e BX continham números inteiros sem sinal, se CF = 1 pode-se concluir que o conteúdo de AX é menor que o conteúdo de BX. Ainda considerando os números inteiros sem sinal, se CF = 0 conclui-se que o conteúdo de AX é maior ou igual ao conteúdo de BX; se CF = 0 e ZF = 0 pode-se concluir que AX é maior que BX. É importante observar que a interpretação dos flags depende da interpretação dada aos números comparados, se são sinalizados ou não. O flag CF = 1 por exemplo, não indica que AX é menor do que BX se o conteúdo dos registradores é interpretado como inteiros sinalizados em complemento a dois. De fato a condição dos flags que indica que AX é menor do que BX neste caso é SF ⊕ OF = 1.

Existem 3 códigos para CMP:

a) CMP reg, reg ou CMP mem, reg ou CMP reg, mem:

```
Ex.:  CMP AX, BX          ; Se por exemplo AX = FFFFH e BX = 0001H antes da
                          ; execução desta instrução, então ZF = 0, CF = 0, SF = 1
                          ; OF = 0 após a execução.
                          ; Se for considerada a representação sem sinal, CF = 0 e
                          ; ZF = 0 indicam AX > BX (65535 > 1).
                          ; Se for considerada a representação sinalizada, SF = 1 e
                          ; OF = 0 indicam AX < BX (-1 < +1).
      CMP [SI], CX
      CMP DI, [BX + DI - 10]
```

b) CMP reg/mem, val: Neste caso um valor imediato val é subtraído do conteúdo de um registrador ou posição de memória.

Ex.: CMP BX, 10
CMP byte ptr [SI], 20

c) CMP acc, val: Neste caso um valor imediato val é subtraído do conteúdo do acumulador (AX ou AL).

Ex.: CMP AX, 10
CMP AL, 10

2.20 NEG OP "Muda o sinal de OP"

Esta instrução calcula o complemento a dois de OP.

Ex.: NEG AX
NEG byte ptr [BX]

Os bits de flags afetados são: AF, CF, OF, PF, SF e ZF

3) Instruções lógicas

Estas instruções manipulam o conteúdo dos operandos sem considerar o seu valor numérico. De fato diversas instruções lógicas manipulam os operandos bit a bit (o resultado da operação em um dos bits independe dos demais).

3.1 AND OP1, OP2 "Faz o AND bit a bit de OP1 e OP2"

Esta instrução executa a operação lógica AND sobre cada bit de OP1 e OP2. Os bits de flags afetados são: PF, SF e ZF. Os bits AF, CF e OF são indefinidos. Existem 3 códigos para AND:

a) AND reg, reg ou AND mem, reg ou AND reg, mem:

```
Ex.:  AND AX, BX          ; Se por exemplo AX = 000FH e BX = FFF1H antes da
                          ; execução desta instrução, então AX = 0001H
                          ; após a execução.
      AND [SI], CX
      AND DI, [BX + DI - 10]
```

b) AND reg/mem, val: Neste caso é feito o and bit a bit de um valor imediato val com o conteúdo de um registrador ou posição de memória.

Esta instrução é freqüentemente utilizada para resetar bits individuais em registradores ou posições de memória.

```
Ex.:  AND BX, 7FFFH; Tem o efeito de resetar o bit mais significativo de BX.
      AND BX, AFFFH       ; Tem o efeito de resetar os bits 12 e 14 de BX.
      AND byte ptr [SI], F7H ; Tem o efeito de resetar o bit 3 do byte na memória cujo
                          ; endereço é dado por DS:SI.
```

c) AND acc, val: Neste caso é feito o and bit a bit de um valor imediato val com o conteúdo do acumulador (AX ou AL).

Ex.: AND AX, 10
AND AL, 10

3.2 OR OP1, OP2 "Faz o OR bit a bit de OP1 e OP2"

Esta instrução executa a operação lógica OR sobre cada bit de OP1 e OP2. Os bits de flags afetados são: PF, SF e ZF. Os bits AF, CF e OF são indefinidos. Existem 3 códigos para OR:

a) OR reg, reg ou OR mem, reg ou OR reg, mem:

```
Ex.:  OR AX, BX           ; Se por exemplo AX = 000FH e BX = 1001H antes da
```

```
                                    ; execução desta instrução, então AX = 100FH
                                    ; após a execução.
        OR [SI], CX
        OR DI, [BX + DI - 10]
```

b) OR reg/mem, val: Neste caso é feito o OR bit a bit de um valor imediato val com o conteúdo de um registrador ou posição de memória.

Esta instrução é freqüentemente utilizada para setar bits individuais em registradores e posições de memória.

```
Ex.:    OR BX, 1000H            ; Tem o efeito de setar o bit mais significativo de BX.
        OR BX, 5000H            ; Tem o efeito de setar os bits 12 e 14 de BX.
        OR byte ptr [SI], 02H   ; Tem o efeito de setar o bit 1 no byte na memória
                                ; cujo endereço é DS:SI.
```

c) OR acc, val: Neste caso é feito o OR bit a bit de um valor imediato val com o conteúdo do acumulador (AX ou AL).

```
Ex.:    OR AX, 10
        OR AL, 10
```

3.3 XOR OP1, OP2 "Faz o XOR bit a bit de OP1 e OP2"

Esta instrução executa a operação lógica XOR sobre cada bit de OP1 e OP2. Os bits de flags afetados são: PF, SF e ZF. Os bits AF, CF e OF são indefinidos. Existem 3 códigos para XOR:

a) XOR reg, reg ou XOR mem, reg ou XOR reg, mem: O código para esta instrução está ilustrado na figura 3.72.

```
Ex.:    XOR AX, BX              ; Se por exemplo AX = 0001H e BX = 0001H antes da
                                ; execução desta instrução, então AX = 0000H
                                ; após a execução.
        XOR [SI], CX
        XOR DI, [BX + DI - 10]
```

b) XOR reg/mem, val: Neste caso é feito o and bit a bit de um valor imediato val com o conteúdo de um registrador ou posição de memória.

Esta instrução é freqüentemente utilizada para complementar bits individuais em registradores ou posições de memória.

```
Ex.:    XOR BX, 1000H           ; Tem o efeito de complementar o bit mais significativo
                                ; de BX.
        XOR BX, 5000H           ; Tem o efeito de complementar os bits 12 e 14 de BX.
        XOR byte ptr [SI], 08H  ; Tem o efeito de complementar o bit 3 do byte na
                                ; memória cujo endereço é dado por DS:SI.
```

c) XOR acc, val: Neste caso é feito o and bit a bit de um valor imediato val com o conteúdo do acumulador (AX ou AL).

```
Ex.:    XOR AX, 10
        XOR AL, 10
```

3.4 NOT OP "Complementa OP bit a bit"

Esta instrução inverte o estado lógico de cada bit do operando que pode ser um registrador ou uma posição de memória. A instrução NOT não afeta os flags.

```
Ex.:    NOT AX          ; Se AX contém 0001H antes da execução, então AX irá
                        ; conter FFFEH após a execução da instrução.
```

3.5 SHL OP1, OP2 "Deslocamento lógico (aritmético) à esquerda sobre OP1"
(**SAL** OP1, OP2)

Esta(s) instrução(ões) transfere(m) o conteúdo de cada bit do operando OP1 para o seu bit imediatamente à esquerda. O bit menos significativo é preenchido com zero e o bit mais significativo é transferido para o CF. Este procedimento é repetido um número de vezes especificado por OP2. OP1 pode ser um registrador ou uma posição de memória. OP2 só pode ser o valor imediato 1 ou então o conteúdo do registrador CL. O funcionamento desta instrução está ilustrado na figura 3.76.

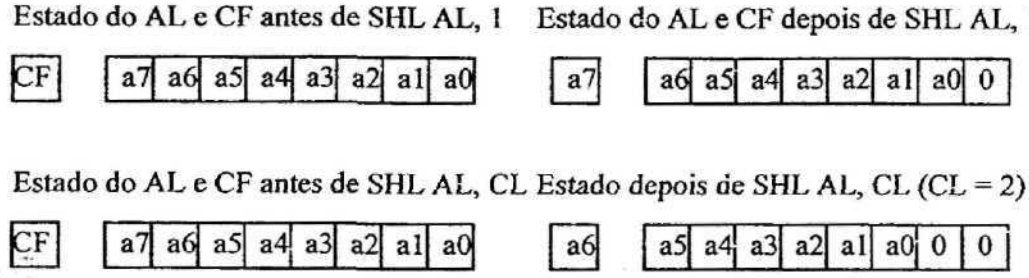

Figura 3.76 - Funcionamento de SHL e SAL.

Observa-se que SHL(SAL) AL, 1 produz o efeito de **multiplicar** o valor de AL por 2, SHL(SAL) AL, CL com CL = 2 o efeito de multiplicar o valor de AL por 4 e assim por diante.
OBS: Não existe diferença de operação entre SHL e SAL.

Esta instrução afeta os flags CF, OF, PF, SF e ZF. O flag AF é indefinido.

```
Ex.:    SAL AX, 1       ; Se AX conitém 0001H (1 decimal) antes da execução,
                        ; Ax conterá 0002H após a execução (2 decimal).
                        ; Se AX contém FFFFH (-1 decimal) antes da execução,
                        ; Ax conterá FFFEH após a execução (-2 decimal).
        SHL BX, CL
        SAL byte ptr [SI], 1
```

3.6 SHR OP1, OP2 "Deslocamento lógico à direita sobre OP1"

Esta instrução transfere o conteúdo de cada bit do operando OP1 para o seu bit imediatamente à direita. O bit mais significativo é preenchido com zero e o bit menos significativo é transferido para o CF. Este procedimento é repetido um número de vezes especificado por OP2. OP1 pode ser um registrador ou uma posição de memória. OP2 só pode ser o valor imediato 1 ou então o conteúdo do registrador CL. O funcionamento desta instrução está ilustrado na figura 3.78.

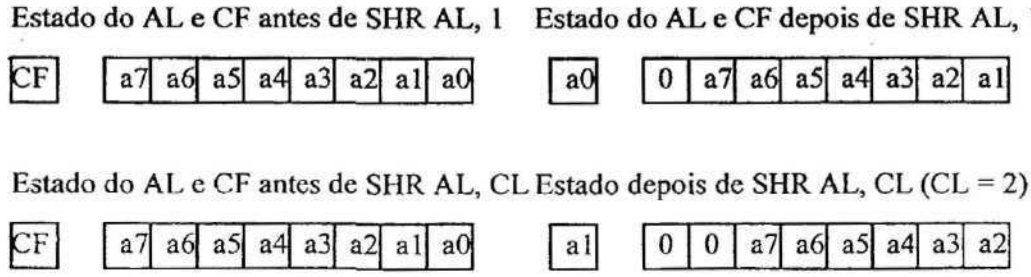

Figura 3.78 - Funcionamento de SHR.

Observa-se que SHR AL, 1 produz o efeito de **dividir** o valor de AL por 2, SHR AL, CL com CL = 2 o efeito de dividir o valor de AL por 4 e assim por diante, **desde que o conteúdo de AL seja um inteiro não sinalizado.**

Esta instrução afeta os flags CF, OF, PF, SF e ZF. O flag AF é indefinido.

```
Ex.:    SHR AX, 1       ; Se AX contém FFFFH (65535 decimal) antes da execução,
                        ; Ax conterá 7FFFH após a execução (32767 decimal).
        SHR BX, CL
        SHR byte ptr [SI], 1
```

3.7 SAR OP1, OP2 "Deslocamento aritmético à direita sobre OP1"

Esta instrução transfere o conteúdo de cada bit do operando OP1 para o seu bit imediatamente à direita. O bit mais significativo é mantido inalterado e o bit menos significativo é transferido para o CF. Este procedimento é repetido um número de vezes especificado por OP2. OP1 pode ser um registrador ou uma posição de memória. OP2 só pode ser o valor imediato 1 ou então o conteúdo do registrador CL. O funcionamento desta instrução está ilustrado na figura 3.80.

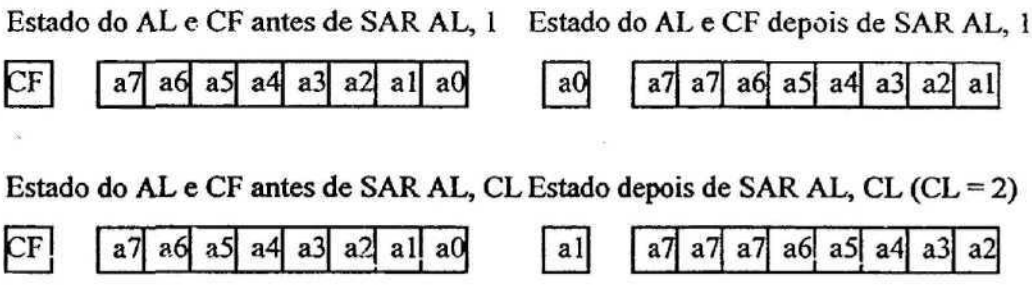

Figura 3.80 - Funcionamento de SAR.

Observa-se que SAR AL, 1 produz o efeito de **dividir** o valor de AL por 2, SHR AL, CL com CL = 2 o efeito de dividir o valor de AL por 4 e assim por diante, **desde que o conteúdo de AL seja um inteiro sinalizado em complemento a dois.**

Esta instrução afeta os flags CF, OF, PF, SF e ZF. O flag AF é indefinido.

```
Ex.:    SAR AX, 1       ; Se AX contém 0003H (3 decimal) antes da execução,
                        ; Ax conterá 0001H após a execução (1 decimal).
                        ; Se AX contém FFFDH (-3 decimal) antes da execução,
                        ; Ax conterá FFFEH após a execução (-2 decimal).
        SAR BX, CL
        SAR byte ptr [SI], 1
```

3.8 ROL OP1, OP2 "Deslocamento circular à esquerda sobre OP1"

Esta instrução transfere o conteúdo de cada bit do operando OP1 para o seu bit imediatamente à esquerda. O bit mais significativo é transferido para o bit menos significativo e também é copiado para o CF. Este procedimento é repetido um número de vezes especificado por OP2. OP1 pode ser um registrador ou uma posição de memória. OP2 só pode ser o valor imediato 1 ou então o conteúdo do registrador CL. O funcionamento desta instrução está ilustrado na figura 3.82.

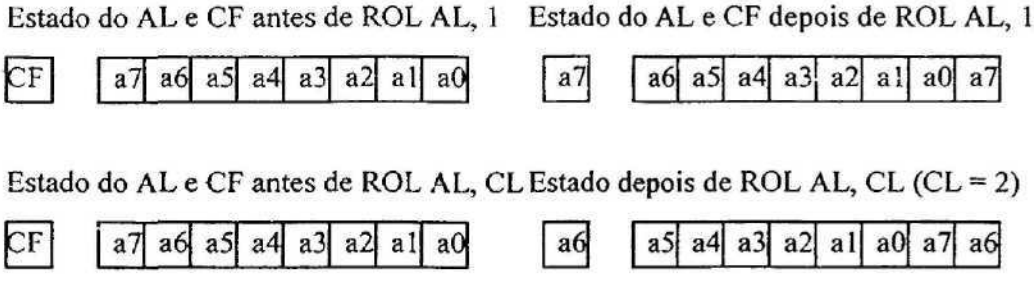

Figura 3.81 - Funcionamento de ROL.

Esta instrução afeta os flags CF, OF, PF, SF e ZF. O flag AF é indefinido.

```
Ex.:    ROL AX, 1       ; Se AX contém 8003H antes da execução,
                        ; Ax conterá 0007H após a execução.
        ROL BX, CL
        ROL byte ptr [SI], 1
```

3.9 ROR OP1, OP2 "Deslocamento circular à direita sobre OP1"

Esta instrução transfere o conteúdo de cada bit do operando OP1 para o seu bit imediatamente à direita. O bit menos significativo é transferido para o bit mais significativo e também é copiado para o CF. Este procedimento é repetido um número de vezes especificado por OP2. OP1 pode ser um registrador ou uma posição de memória. OP2 só pode ser o valor imediato 1 ou então o conteúdo do registrador CL. O funcionamento desta instrução está ilustrado na figura 3.83.

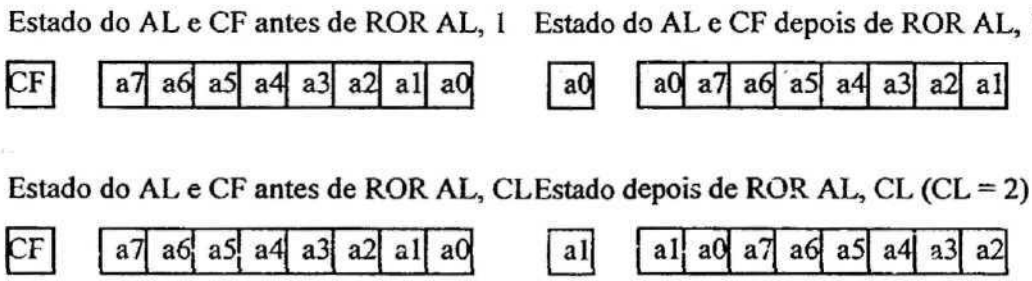

Figura 3.83 - Funcionamento de ROR.

Esta instrução afeta os flags CF, OF, PF, SF e ZF. O flag AF é indefinido.

```
Ex.:    ROR AX, 1       ; Se AX contém 0003H antes da execução,
                        ; Ax conterá 8001H após a execução.
        ROR BX, CL
        ROR byte ptr [SI], 1
```

3.10 RCL OP1, OP2 "Deslocamento circular à esquerda através do CF sobre OP1"

Esta instrução transfere o conteúdo de cada bit do operando OP1 para o seu bit imediatamente à esquerda. O bit mais significativo é transferido para o CF e o CF para o bit menos significativo. Este procedimento é repetido um número de vezes especificado por OP2. OP1 pode ser um registrador ou uma posição de memória. OP2 só pode ser o valor imediato 1 ou então o conteúdo do registrador CL. O funcionamento desta instrução está ilustrado na figura 3.85.

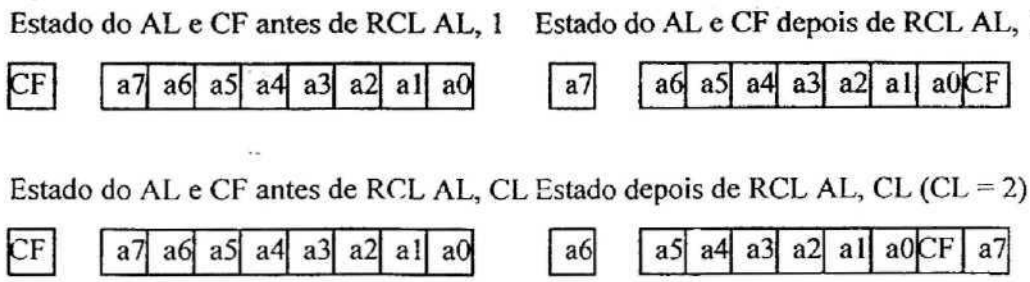

Figura 3.85 - Funcionamento de RCL.

Esta instrução afeta os flags CF, OF, PF, SF e ZF. O flag AF é indefinido.

```
Ex.:    RCL AX, 1       ; Se AX contém 8003H e CF = 0 antes da execução,
                        ; Ax conterá 0006H e CF = 1 após a execução.
        RCL BX, CL
        RCL byte ptr [SI], 1
```

3.11 RCR OP1, OP2 "Deslocamento circular à direita através do CF sobre OP1"

Esta instrução transfere o conteúdo de cada bit do operando OP1 para o seu bit imediatamente à direita. O bit menos significativo é transferido para o CF e o CF para o bit mais significativo. Este procedimento é repetido um número de vezes especificado por OP2. OP1 pode ser um registrador ou uma posição de memória. OP2 só pode ser o valor imediato 1 ou então o conteúdo do registrador CL. O funcionamento desta instrução está ilustrado na figura 3.87.

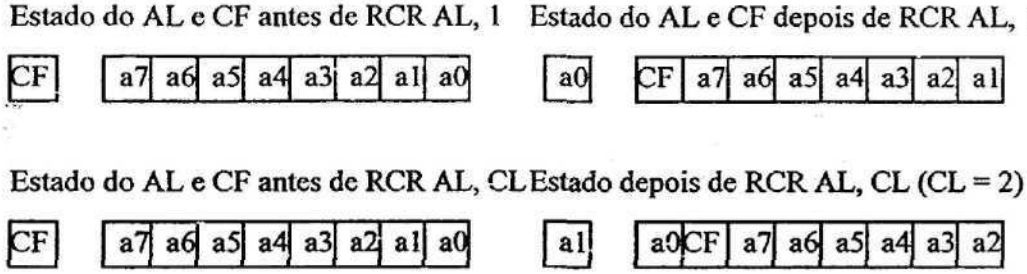

Figura 3.87 - Funcionamento de RCR.

Esta instrução afeta os flags CF, OF, PF, SF e ZF. O flag AF é indefinido.

```
Ex.:    RCR AX, 1       ; Se AX contém 0003H e CF = 0 antes da execução,
                        ; Ax conterá 0001H e CF = 1 após a execução.
        RCR BX, CL
        RCR byte ptr [SI], 1
```

3.12 TEST OP1, OP2 "Compara logicamente OP1 com OP2"

Esta instrução compara bit a bit OP1 com OP2. A comparação é feita efetuando o AND bit a bit de OP2 com OP1. A instrução é idêntica à AND no que diz respeito ao registrador de flags (Os bits de flags afetados são: PF, SF e ZF, sendo AF, CF e OF indefinidos). No entanto OP1 não é atualizado com o resultado da operação (OP1 AND OP2). Existem 3 códigos para TEST:

a) TEST reg, reg ou TEST mem, reg ou TEST reg, mem:

```
Ex.:    TEST AX, BX     ; Se por exemplo AX = FFFEH e BX = 0001H antes da
                        ; execução desta instrução, então ZF = 1, PF = 0, SF = 0
                        ; após a execução. ZF = 1 neste caso indica que todos os
                        ; bits de AX são diferentes dos correspondentes em BX.
        TEST [SI], CX
        TEST DI, [BX + DI - 10]
```

b) TEST reg/mem, val: Neste caso é feito o AND de um valor imediato val com o conteúdo de um registrador ou posição de memória.

Esta instrução é útil para testar bits individuais de registradores ou posições de memória.

```
Ex.:    TEST BX, 0100H  ; O resultado dependerá apenas do valor do bit 8 de BX.
                        ; Se este bit for 1, então ZF = 0. Se este bit for 0 então
                        ; ZF = 1.
        CMP byte ptr [SI], 20H
```

c) TEST acc, val: Neste caso é feito o AND de um valor imediato val com o conteúdo do acumulador (AX ou AL).

Ex.: TEST AX, 0110H
TEST AL, 10H

4) Instruções de desvio

As instruções de desvio permitem alterar a seqüência normal de execução de um programa. Normalmente as instruções são lidas da memória em endereços consecutivos pela unidade de interface de barramento e colocadas na fila de instruções para serem executadas. Através das instruções de desvio é possível transferir a execução do programa para qualquer ponto no espaço de memória. Por isso as instruções de desvio são capazes de alterar o conteúdo dos registradores CS e IP (de fato são as únicas instruções com esta capacidade).

4.1 **JMP** OP "Desvio incondicional"

A instrução JMP ("Jump") transfere a execução do programa para o endereço especificado por OP. O efeito do JMP é esvaziar a fila de instruções e em seguida atualizar CS e IP de acordo com o endereço especificado em OP. Deste modo a próxima instrução a ser executada será lida da memória a partir deste novo endereço, o que caracteriza o desvio. O JMP não afeta os flags. Existem 4 tipos de JMP, em função da forma de especificação de OP:

a) JMP direto intrasegmento: Neste caso o destino do desvio está contido no mesmo segmento corrente (CS não é alterado). Por causa disso basta especificar IP e portanto OP é um número de 16 bits. No entanto, OP não é simplesmente carregado em IP mas sim somado ao conteúdo do mesmo. Assim, o valor em OP especifica um **desvio relativo**, medido em relação ao endereço de início da instrução JMP. OP é neste caso um número sinalizado em complemento a dois permitindo desvios de -32768 até +32767 bytes em relação ao endereço do JMP (o que permite alcançar todo o segmento). O efeito deste tipo de desvio está ilustrado na figura 3.92.

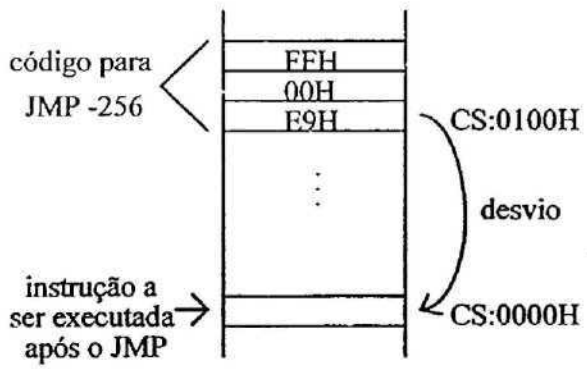

Figura 3.92 - Efeito do JMP direto intrasegmento.

```
Ex.:    JMP -0100H      ; Desvia para a instrução localizada 256 bytes antes do JMP no
                        ; programa.
```

Obs.: Os programas montadores permitem declarar labels que facilitam a escrita do programa por liberar o programador da tarefa de calcular o valor do deslocamento. O seguinte exemplo ilustra o uso de um label:

```
        .
LB1:    MOV   CX, 10    ; O label é definido pelo nome (LB1 neste caso) seguido de ":".
        .
        .
        .
        JMP LB1         ; O montador calcula a distância em bytes até o label LB1 e usa
                        ; como valor do deslocamento.
```

b) JMP direto curto intrasegmento: Semelhante ao anterior, exceto que o deslocamento está entre -128 e +127.

c) JMP indireto intrasegmento: Neste caso o destino está no mesmo segmento e portanto CS não muda. O valor de IP será substituído pelo conteúdo de um registrador ou de uma posição de memória. Nota-se que neste caso o desvio não é relativo (o valor não é somado ao IP, ele substitui IP).

```
Ex.:    JMP CX                  ; Após a execução do JMP, a próxima instrução executada
                                ; está no endereço CS:CX.
        JMP word ptr [SI]       ; O offset do endereço da próxima instrução está na
                                ; palavra de memória cujo endereço é DS:SI.
```

d) JMP direto intersegmentos: Neste caso o destino do salto está em outro segmento. Portanto é necessário especificar o endereço completo (CS e IP). Os novos valores de CS e IP são codificados em 4 bytes imediatos na instrução. O efeito deste tipo de desvio está ilustrado na figura 3.96.

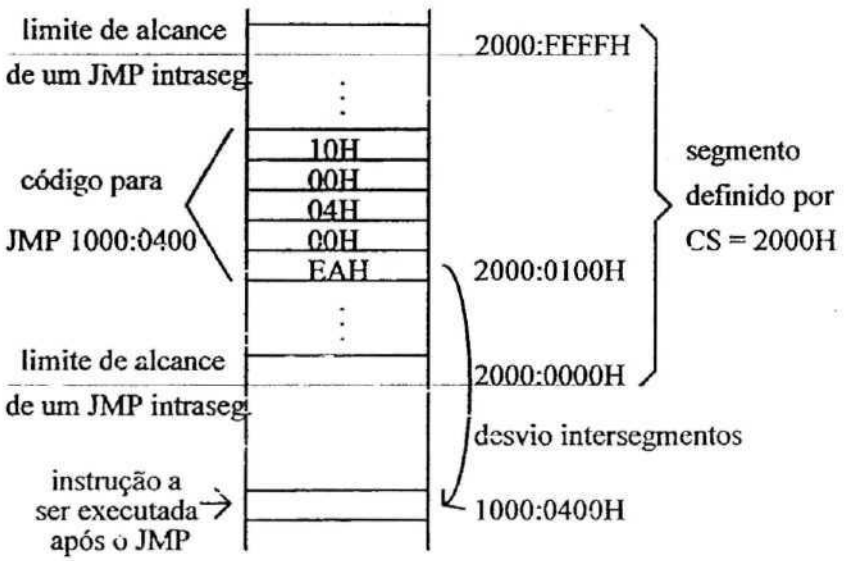

Figura 3.96 - Efeito do JMP direto intersegmentos.

```
Ex.:    JMP LB_distante         ; No programa fonte, o label LB_distante está definido
                                ; em outro segmento.
```

e) JMP indireto intersegmentos: O destino do JMP está em outro segmento que não o corrente e os novos valores de CS e IP estão em uma "double word" (quatro bytes consecutivos) na memória.

```
Ex.:    JMP dword ptr [SI]      ; O endereço de destino está nos quatro bytes consecutivos
                                ; na memória a partir do endereço DS:SI. Os dois bytes
                                ; menos significativos contém o novo valor de IP e os dois
                                ; bytes mais significativos contém o novo valor do
                                ; segmento.
```

4.2 J(condição) OP "Desvio condicional"

Esta é uma instrução de desvio que permite implementar tomadas de decisão no programa. Os desvios condicionais são sempre de curto alcance e por isso OP é um byte que especifica um deslocamento relativo sinalizado a ser somado (ou não) ao IP. Se a condição especificada na instrução é verdadeira, o desvio é executado (ou seja OP é somado ao IP). Se a condição é falsa, o desvio não é executado e a execução prossegue a partir da instrução seguinte ao J(condição). Nenhum flag é afetado por esta instrução. Existem diversas possibilidades para condição, quase todas envolvendo um ou mais

bits do registrador de flags. As possibilidades estão resumidas na tabela 3.4 (nesta tabela "+" indica ou e "⊕" indica ou-exclusivo).

Instrução	Condição	Desvia se ...
JE ou JZ	ZF = 1	"igual" ou "zero"
JL ou JNGE	(SF ⊕ OF) = 1	"menor" ou "não maior ou igual"
JLE ou JNG	((SF ⊕ OF) + ZF) = 1	"menor ou igual" ou "não maior"
JB ou JNAE ou JC	CF = 1	"abaixo" ou "não acima ou igual" ou "carry"
JBE ou JNA	(CF + ZF) = 1	"abaixo ou igual" ou "não acima"
JP ou JPE	PF = 1	"paridade" ou "paridade par"
JO	OF = 1	"overflow"
JS	SF = 1	"negativo"
JNE ou JNZ	ZF = 0	"não igual" ou "não zero"
JNL ou JGE	(SF ⊕ OF) = 0	"não menor" ou "maior ou igual"
JNLE ou JG	((SF ⊕ OF) + ZF) = 0	"não menor ou igual" ou "maior"
JNB ou JAE ou JNC	CF = 0	"não abaixo" ou "acima ou igual" ou "não carry"
JNBE ou JA	(CF + ZF) = 0	"não abaixo ou igual" ou "acima"
JNP ou JPO	PF = 0	"não paridade" ou "paridade ímpar"
JNO	OF = 0	"não overflow"
JNS	SF = 0	"não sinal"

Tabela 3.4 - Condições para desvio condicional.

Cada uma das instruções da tabela 3.4 possui um código distinto, levando a 16 tipos de desvios condicionais.

Os valores de c3, c2, c1 e c0 especificam a condição do desvio de acordo com a tabela 3.5.

Condição	JO	JNO	JB JNA JC	JNB JAE JNC	JE JZ	JNE JNZ	JBE JNA	JNBE JA
c3c2c1c0	0000	0001	0010	0011	0100	0101	0110	0111

Condição	JS	JNS	JP JPE	JNP JPO	JL JNGE	JNL JGE	JLE JNG	JNLE JG
c3c2c1c0	1000	1001	1010	1011	1100	1101	1110	1111

Tabela 3.5 - Valores de c3c2c1c0 nas instruções de desvio condicional.

Obs.: As condições que envolvem "Lesser" e "Greater" (L e G) são usadas para decidir sobre a magnitude relativa de números sinalizados em complemento a dois após uma instrução de comparação CMP. Por outro lado, as condições "Below" e "Above" são usadas para decidir sobre a magnitude relativa de números sem sinal, após uma instrução de comparação. Por exemplo, o seguinte código não faz sentido:

```
CMP AX, 0       ; Subtrai 0 de AX e corrige os flags.
JB negativo     ; Desvia se CF = 1 (nunca vai ocorrer).
```

No exemplo acima o uso de JB é impróprio pois o resultado da comparação anterior (que subtrai 0 de AX) nunca produzirá CF = 1 e portanto o desvio nunca será executado. O problema ocorre porque JB deve ser usado apenas com números sem sinal (e se AX contém um inteiro sem sinal, ele jamais será menor do que zero). No caso acima, o correto é utilizar JL do seguinte modo:

```
CMP AX, 0       ; Subtrai 0 de AX e corrige os flags.
JL negativo     ; Desvia se (SF ⊕ OF) = 1 (pode ocorrer se AX contiver um
                ; número negativo representado em complemento a dois).
```

Algumas vezes pode ser necessário executar um desvio condicional para um endereço mais afastado do que -128 ou +127 bytes da instrução de desvio. Quando isso ocorre é possível aumentar o alcance do desvio condicional associando-se o mesmo a um desvio incondicional. Por exemplo sejam os códigos:

```
i)      CMP AX, 10          ii)     CMP AX, 10
        JL LB1                      JGE LB2
        .                           JMP LB1:
        .                   LB2:    .
        .                           .
LB1:                        LB1:    .
```

Os dois códigos acima são equivalentes, no entanto no segundo caso (ii) o salto para LB1 pode ter qualquer tamanho (pode inclusive ser intersegmentos) o que não ocorre no primeiro caso (i).

4.3 LOOP OP "Laço de repetição"

Esta instrução é usada para repetir um trecho de código um determinado número de vezes. Funciona do seguinte modo: Primeiramente o registrador CX é decrementado. Em seguida o valor de CX é comparado com zero. Se CX for diferente de zero desvia para o endereço especificado por OP. OP é um deslocamento relativo na faixa de -128 até + 127. Nenhum flag é afetado.

```
Ex.:        MOV CX, 10      ; Inicializa CX com o número de repetições.
      LB1:  .
            .               ; Código a ser repetido.
            .
            LOOP LB1        ; Decrementa CX. Retorna a LB1 enquanto CX ≠ 0.
```

4.4 LOOPZ OP (LOOPE OP) "Laço de repetição"

Esta instrução é usada para repetir um trecho de código um determinado número de vezes, mas com a possibilidade de interromper o laço antes de concluir o número de repetições preestabelecido. Funciona do seguinte modo: Primeiramente o registrador CX é decrementado. Em seguida o valor de CX é comparado com zero. Se CX for diferente de zero e ZF = 1 desvia para o endereço especificado por OP. OP é um deslocamento relativo na faixa de -128 até + 127. A condição ZF = 0 portanto permite interromper o laço. LOOPZ não afeta os flags.

Ex.: Um programa utiliza uma rotina de comunicação através da porta serial. O protocolo de comunicação transmite um pacote através da interface serial e depois espera uma resposta (ACK) antes de enviar o próximo pacote. Se o pacote não chegar, o pacote previamente transmitido é repetido. A interface serial usada é um 8251 mapeado nos endereços 378H (regs. Tx e Rx) e 379H (regs. modo, comando e status). O seguinte trecho de código implementa o laço de espera do ACK:

```
        MOV CX, 1000    ; CX é preparado para 1000 repetições.
        MOV DX, 379H    ; DX recebe o endereço do registrador de status do 8251.
Espera: IN AL, DX       ; Lê o status do 8251 para AL.
        TEST AL, 02H    ; Verifica se o bit 1 (RxRDY) do reg. de status está setado.
        LOOPZ Espera    ; Se RxRDY = 1 então ZF = 0 e o laço é interrompido.
        JZ Time_Out     ; Se neste ponto ZF = 1, o laço chegou ao limite sem que
                        ; ocorresse a recepção do ACK. Portanto desvia para o
                        ; código de tratamento de falha (Time_Out).
```

4.5 LOOPNZ OP (LOOPNE OP) "Laço de repetição"

Esta instrução é usada para repetir um trecho de código um determinado número de vezes, mas com a possibilidade de interromper o laço antes de concluir o número de repetições preestabelecido. Funciona do seguinte modo: Primeiramente o registrador CX é decrementado. Em seguida o valor de CX é comparado com zero. Se CX for diferente de zero e ZF = 0 desvia para o endereço especificado

por OP. OP é um deslocamento relativo na faixa de -128 até + 127. A condição ZF = 1 portanto permite interromper o laço. LOOPNZ não afeta os flags.

4.6 JCXZ OP "Desvia se CX = 0"

Esta instrução é um tipo de desvio condicional mas que não depende dos flags. Se CX for igual a zero desvia para o endereço especificado por OP. OP é um deslocamento relativo na faixa de -128 até + 127. Nenhum flag é afetado.

4.7 CALL OP "Desvia para subrotina"

Esta instrução é semelhante ao desvio incondicional JMP OP, mas é utilizada quando se pretende retornar do desvio executado. Por causa disso esta instrução salva o endereço da instrução seguinte à ela antes de executar o desvio. Este endereço é salvo na pilha. Analogamente ao JMP, CALL pode ser direto ou indireto, intrasegmento ou intersegmentos. Esta instrução não afeta os flags. CALL OP funciona da seguinte maneira:

i) PUSH CS (apenas nos tipos de CALL intersegmentos).
ii) PUSH IP.
iii) JMP OP.

Existem 4 códigos para CALL:

a) CALL direto intrasegmento: Neste caso a subrotina chamada está contida no mesmo segmento corrente (CS não é alterado). Por causa disso basta especificar IP e portanto OP é um número de 16 bits. No entanto, OP não é simplesmente carregado em IP mas sim somado ao conteúdo do mesmo. Assim, o valor em OP especifica um **desvio relativo**, medido em relação ao endereço de início da instrução CALL (Ver JMP). OP é neste caso um número sinalizado em complemento a dois permitindo desvios de -32768 até +32767 bytes em relação ao endereço do CALL (o que permite alcançar todo o segmento).

```
Ex.:    CALL Produto_Escalar    ; Desvia para a rotina Produto_Escalar.
```

b) CALL indireto intrasegmento: Neste caso a subrotina também está no mesmo segmento e portanto CS não muda. O valor de IP será substituído pelo conteúdo de um registrador ou de uma posição de memória. Nota-se que neste caso o desvio não é relativo (o valor não é somado ao IP, ele substitui IP).

```
Ex.:    CALL CX                 ; Desvia para a subrotina no endereço CS:CX.
        CALL word ptr [SI]      ; O offset do endereço da subrotina está na
                                ; palavra de memória cujo endereço é DS:SI.
```

c) CALL direto intersegmentos: Neste caso a subrotina está em outro segmento. Portanto é necessário especificar o endereço completo (CS e IP). Os novos valores de CS e IP são codificados em 4 bytes imediatos na instrução.

```
Ex.:    CALL Subrotina_Dist     ; No programa fonte, a rotina Subrotina_dist está definida
                                ; em outro segmento.
```

d) CALL indireto intersegmentos: A subrotina chamada está em outro segmento que não o corrente e os novos valores de CS e IP estão em uma "double word" (quatro bytes consecutivos) na memória.

```
Ex.:    CALL dword ptr [SI]     ; O endereço da subrotina está nos quatro bytes
                                ; consecutivos na memória a partir do endereço DS:SI.
                                ; Os dois bytes menos significativos contém o novo
                                ; valor de IP e os dois bytes mais significativos contém
                                ; o novo valor do segmento.
```

Um aspecto importante associado à chamada de subrotinas é a passagem de parâmetros. Considere-se, para ilustração, uma linha de programa em uma linguagem de alto nível como C envolvendo a chamada de uma subrotina:

```
int Maximo(int num1, int num2);   /* Protótipo da função Maximo. Esta função recebe como
.                                    parâmetros dois números inteiros e devolve um
.                                    inteiro */
.
int a, b, c;          /* Declaração das variáveis inteiras a, b, c */
.
.
.
c = Maximo(a, b);          /* Uso da função Maximo. Envolve a passagem dos parâmetros
.                             a e b, e o retorno do valor c. */
.
.
```

Quando for traduzido para instruções de 8086, o programa acima envolverá uma chamada à subrotina Maximo feita utilizando-se a instrução CALL Maximo. No entanto, a instrução CALL não fornece nenhum suporte para a passagem dos parâmetros a e b, bem como para retorno do valor c. Por isso adota-se algumas convenções para passagem de parâmetros, a saber:

i) Passagem de parâmetros por registradores. Neste caso usa-se um ou mais registradores da CPU para armazenar os parâmetros (no exemplo AX poderia ser carregado com o valor a e BX com o valor b por exemplo). Pode-se usar também um registrador para armazenar o resultado (por exemplo CX poderia armazenar o resultado e neste caso c = CX). Usando-se esta convenção, o código assembly correspondente à linha "c = Maximo(a, b);" do exemplo seria (Essa **não** é a convenção adotada pelos compiladores C):

```
        MOV AX, a       ; a é uma posição de memória, ou uma constante, ou até um
                        ; outro registrador.
MOV BX, b       ; b é uma posição de memória, ou uma constante, ou até um
                        ; outro registrador.
CALL Maximo     ; Após esta instrução, CX contém o resultado.
```

O código da subrotina tem de ser escrito de acordo com a convenção adotada. Por exemplo, a subrotina Maximo possuiria o seguinte código:

```
Maximo PROC             ; Esta linha é utilizada pelo programa montador apenas para identificar a
                        ; subrotina e não gera nenhum código (não é uma instrução do 8086).
        PUSHF           ; É comum iniciar uma subrotina salvando o contexto que será alterado
                        ; por ela (como contexto entende-se o FR e todos os registradores usados,
                        ; menos os utilizados para passagem de parâmetros e retorno de resultados).
        CMP AX, BX      ; Compara para escolher o maior.
        JL BX_e_Maior   ; Desvia se AX < BX. Como AX e BX são sinalizados (int), usa JL.
        MOV CX, AX      ; CX recebe o maior entre AX e BX, AX neste caso.
        POPF            ; Restaura o contexto.
        RET             ; Retorna da subrotina (ver RET).

BX_e_Maior:
        MOV CX, BX      ; CX recebe o maior entre AX e BX, BX neste caso.
        POPF            ; Restaura o contexto.
        RET             ; Retorna da subrotina (ver RET).

Maximo ENDP             ; Esta linha é utilizada pelo programa montador apenas para identificar o
                        ; fim da subrotina e não gera nenhum código (não é uma instrução do
                        ; 8086).
```

ii) Passagem de parâmetros por posições de memória e pela pilha: Outra convenção de passagem de parâmetros utiliza posições de memória ao invés de registradores. É análoga à convenção anterior, mas admite a passagem de um número bem maior de parâmetros (no caso anterior, o pequeno número de registradores disponíveis limita o número de parâmetros a serem passados). Um caso

4.8 RET OP "retorna de subrotina"

Esta instrução é usada para retornar de uma subrotina para a rotina principal. RET funciona da seguinte maneira:

i) POP IP
ii) POP CS (Apenas nos tipos de RET intersegmentos).
iii) OP, um número de 16 bits, é somado ao SP (opcional).

O parâmetro OP é opcionalmente utilizado para descartar parâmetros na pilha quando se utiliza a convenção de passagem de parâmetros pela pilha. No exemplo da função Maximo apresentado na descrição do uso de CALL, usou-se RET 2 para retirar automaticamente da pilha os dois parâmetros a e b (ver figura 3.108) ao retornar da subrotina. O efeito de RET 2 é, ao retornar, somar 2 ao SP, eliminando assim os dois parâmetros da pilha.

Uma subrotina que utiliza um RET intrasegmentos deve ser chamada utilizando-se uma instrução CALL intrasegmentos. Analogamente rotinas que utilizam RET intersegmentos devem ser chamadas usando-se CALL intersegmentos. Por isso, é uma propriedade intrínseca da subrotina poder ser chamada por rotinas de outros segmentos ou não. Uma subrotina que pode ser chamada de qualquer segmento (que usa portanto RET intersegmentos) é classificada como "FAR". Uma subrotina que só pode ser chamada por outras no mesmo segmento é classificada como "NEAR". A maioria dos programas montadores admitem que se use os atributos FAR e NEAR ao especificar uma subrotina, forçando deste modo o uso respectivamente de RET intersegmentos ou intrasegmento. Os 4 tipos de códigos para RET são:

a) RET intrasegmento:
b) RET intrasegmento com retirada de parâmetros da pilha:
c) RET intersegmentos:
d) RET intrasegmento com retirada de parâmetros da pilha:

4.9 INT OP "Interrupção"

Inicia uma interrupção por software. O operando OP é um byte que especifica uma das 256 possíveis rotinas de interrupção. Esta instrução afeta os flags TF e IF. O funcionamento de INT é o seguinte:

i) PUSHF
ii) reseta os bits IF e TF no registrador de flags.
iii) PUSH CS
iv) "PUSH IP"
v) Lê da memória 4 bytes consecutivos a partir do endereço 0000: (4×OP). Os 2 bytes menos significativos lidos são carregados em CS e Os 2 bytes mais significativos lidos são carregados em IP. Após INT portanto a execução prossegue a partir do endereço do início da rotina de interrupção especificado na tabela de endereços de interrupções.

Existem 2 tipos de código para INT:

a) INT val, val ≠ 3:

```
Ex.:    INT 8       ; Inicia a interrupção tipo 8.
                    ; O endereço da rotina de tratamento de interrupção é
                    ; lido da memória a partir do endereço 0000:0020H.
        INT 10H     ; Inicia a interrupção tipo 16.
```

c) INT 3:

Esta instrução de interrupção especial de 1 byte existe para facilitar a implementação de programas de depuração (debugger). Especificamente é utilizada para instalação de "breakpoints". O [illegible] é instalado em um determinado ponto no programa a ser depurado substituindo-se a instrução original por uma interrupção tipo 3. Basta então colocar como endereço da interrupção tipo 3

na tabela de endereços de interrupções o endereço da subrotina de tratamento de breakpoint do debugger. Como existem instruções de 1 byte, é necessário que a interrupção usada para breakpoint seja de um byte para poder substituir qualquer instrução sem mutilar a instrução seguinte (Pode parecer desnecessário, já que em princípio a instrução imediatamente seguinte ao breakpoint não será executada e portanto poderia ser mutilada. No entanto isso não é sempre verdadeiro por causa dos desvios; ou seja a instrução imediatamente seguinte ao breakpoint pode ser executada **sem** passar pelo breakpoint por causa das instruções de desvio).

Assim como o caso especial da interrupção tipo 3, existem outras que, apesar de não possuírem um código diferenciado, são ditas reservadas. Estas interrupções são geradas sempre que ocorrem eventos específicos no microprocessador. São elas:

INT 0: É automaticamente iniciada ao ocorrer uma divisão por zero. Por exemplo se BL contém 0, DIV BL iniciará uma INT 0.

INT 1: Se TF = 1, Executa uma instrução e em seguida inicia uma INT1. É usada para implementar a execução passo a passo em programas de depuração (single-step).

INT 2: NMI. É a interrupção iniciada pelo pino de hardware NMI.

INT 3: usada para breakpoint em depuradores.

INT 4: Ao executar a instrução INTO, será iniciada a INT 4 se OF = 1.

INT 5...31: Reservadas para uso em processadores mais avançados (386, 486, pentium ...).

Uma vantagem do uso da instrução INT para chamar uma subrotina é que, ao contrário de CALL, não é necessário especificar o endereço da subrotina, mas apenas o seu tipo. Por este motivo vários serviços de BIOS ("Basic Input Output System") e do sistema operacional DOS (Disc Operating System) usado em PCs são acionados através de interrupções. Isso permite que sejam feitas atualizações (mudanças de versão e "upgrades") no software de BIOS e do sistema operacional sem que seja preciso modificar o software de programas de aplicações que utilizam os serviços disponíveis no BIOS e no DOS. Nota-se que uma subrotina escrita para ser chamada com CALL não pode ser chamada com INT (INT empilha os flags mas CALL não) e vice-versa. Assim uma subrotina que pode ser chamada com INT (ou iniciada por um sinal de hardware) é uma subrotina de tratamento de interrupção e diferencia-se de uma subrotina comum por utilizar um tipo diferente de instrução de retorno, a IRET.

4.10 IRET "Retorno de interrupção"

Diferencia-se de RET porque desempilha o registrador de flags. Obviamente todos os flags são afetados. É usada para retornar de subrotinas de tratamento de interrupção. Não é comum o uso de passagem de parâmetros pela pilha para rotinas de tratamento de interrupção e por isso IRET não possui parâmetros.

4.11 INTO "Interrompe se overflow"

Ao executar INTO, se OF = 1 inicia uma interrupção tipo 4. Afeta TF e IF.

5) Instruções de controle

São usadas para modificar a resposta do processador a diversos eventos de software e de hardware. Entre estas instruções estão instruções de acesso aos bits de flags, sincronização de coprocessador, etc.

5.1 CLC "zera flag de carry"

Faz CF = 0.

5.2 STC "seta flag de carry"

Faz CF = 1.

5.3 CMC "complementa flag de carry"

Faz CF = /CF.

5.4 CLI "Desabilita interrupções"

Faz IF = 0. Desabilita as interrupções de hardware ativadas pelo pino INTR.

5.5 STI "Habilita interrupções"

Faz IF = 1. Habilita as interrupções de hardware ativadas pelo pino INTR.

5.6 CLD "zera flag de direção"

Faz DF = 0.

5.7 STD "seta flag de direção"

Faz DF = 1.

5.8 HLT "Halt"

Após a execução de HLT, a unidade de execução do 8086 (8088) para de ler instruções da fila de instruções e a unidade de controle do barramento para de fazer a busca de novas instruções. Os únicos modos de reiniciar a operação normal são através de um RESET externo ou de um pedido de interrupção externo. Esta instrução faz o microprocessador interromper a execução do programa e esperar até que ocorra uma interrupção de hardware ou um reset.

5.9 ESC OP1, OP2 "Escape"

Esta instrução é usada para sincronização de coprocessador. OP1 é um número entre 0 e 63 que permite especificar uma dentre 64 instruções diferentes de coprocessador. OP2 é um endereço de operando na memória. Quando o 8086 executa uma instrução ESC ele calcula o endereço do operando especificado na instrução e executa externamente um ciclo de barramento de leitura, porém o dado lido é descartado. O coprocessador, 8087 por exemplo, está fisicamente conectado ao barramento local do 8086 e captura o operando lido (Caso o operando esteja incompleto, ou se a operação de coprocessador

envolver uma escrita, o coprocessador toma o barramento e lê o restante do operando na memória). A instrução ESC não afeta os flags.

```
Ex.:    ESC 4, [SI]     ; Coloca o conteúdo da posição de memória em DS:SI no
                        ; barramento local. Informa ao coprocessador que ele deve
                        ; executar a sua instrução de código 4.
```

5.10 WAIT "Espera coprocessador"

Esta instrução verifica o estado do pino TEST#. Se TEST# = 1, o 8086 (8088) aguarda até que TEST# = 0 para então prosseguir na execução da instrução seguinte. Em um sistema com coprocessador aritmético 8087, o pino TEST# do 8086 (8088) estará conectado ao pino BUSY do 8087 (BUSY = 1 quando o coprocessador não está pronto para receber uma nova instrução). A instrução WAIT permite o atendimento de um pedido de interrupção de hardware enquanto estiver no estado de espera. WAIT não afeta os flags.

Uma forma de utilizar esta instrução é colocar uma WAIT precedendo cada ESC.

```
Ex.:    WAIT            ; Aguarda o coprocessador desocupar.
        ESC 4, [BX]     ; Instrui o coprocessador a executar a instrução 4.
        .               ; Instruções de 8086 (8088). Neste ponto, o processador
        .               ; principal e o coprocessador executam ao mesmo tempo,
        .               ; operando em paralelo.
        WAIT            ; Aguarda o coprocessador terminar ESC 4 anterior.
        ESC 2, [SI +1]  ; Instrui o coprocessador a executar a instrução 2.
```

5.11 LOCK "Ativa o pino LOCK#"

Esta instrução faz o sinal de hardware LOCK# = 0 durante a execução da instrução seguinte. Em um sistema com múltiplos processadores pode ser usada para impedir a tomada do barramento por outro processador durante a próxima instrução. Perde o efeito se usada com as instruções de manipulação de strings associadas ao prefixo REP, se a operação de string for interrompida por um sinal de interrupção de hardware. Não afeta os flags.

5.12 NOP "No OPeration"

Não faz nada. É usada para gastar tempo. Não afeta flags.

6) Instruções de manuseio de strings

Estas instruções permitem fazer operações com dados em bloco.

6.1 MOVS "Move string"

Copia o conteúdo de uma posição de memória para outra, porém com endereçamento mais restrito que o da instrução MOV (observa-se que não existe MOV mem, mem). O operando fonte é sempre a posição de memória (pode ser de 8 ou 16 bits) no endereço DS:SI (SI significa "source index") e o destino a posição de memória no endereço ES:DI (DI significa "destination index"). Além de copiar o dado, MOVS atualiza SI e DI de modo a apontar a próxima posição das strings fonte e destino. O funcionamento de MOVS é o seguinte:

i) Copia o dado (byte ou word) de DS:SI para ES:DI.
ii) Há 4 possibilidades: Se DF = 0 e o dado é um byte, incrementa SI e DI.
Se DF = 1 e o dado é um byte, decrementa SI e DI.

Se DF = 0 e o dado é uma word, incrementa SI e DI duas vezes.
Se DF = 1 e o dado é uma word, decrementa SI e DI duas vezes.

Observa-se que o bit de flag DF controla o sentido da transferência, sendo do menor para o maior endereço das strings se DF = 0 e do maior para o menor endereço se DF = 1. Por isso este flag é chamado "Direction Flag". Usam-se 2 mnemônicos diferentes para distinguir quando o operando é de 8 bits (MOVSB) ou de 16 bits (MOVSW).

6.2 REP "Repetição"

Esta instrução (na verdade um prefixo para as demais instruções de string), quando associada as demais instruções de string, permite manusear blocos de bytes ou words. É usada imediatamente antes de uma instrução de string. Quando o microprocessador executa uma instrução REP ele decrementa o registrador CX (sem afetar os flags) e repete a instrução de string subsequente até que CX = 0. Há duas variações: REPZ (REPE) que repete enquanto ZF = 1 (e CX ≠ 0) e REPNZ (REPNE) que repete enquanto ZF = 0 (e CX ≠ 0). O REP não afeta os flags.

```
Ex.:  CLD          ; Prepara transferência do menor para o maior endereço.
      MOV SI, 10   ; Bloco de dados fonte começa em DS:000AH.
      MOV DI, 0    ; Bloco de dados destino começa em ES:0000H.
      MOV CX, 10   ; Tamanho do bloco é 10.
      REP MOVSB    ; Copia o bloco em DS:000AH a DS:0014H
                   ; para ES:0000H a ES:000AH.
```

6.3 CMPS "Compara string"

Compara DS:SI com ES:DI (subtrai ES:DI de DS:SI mas só afeta os flags). atualiza SI e DI como em MOVS. Afeta os flags AF, CF, OF, PF, SF e ZF. Existem CMPSB (8 bits) e CMPSW (16 bits).

Ex.: Rotina para comparar 2 strings em DS:0000H e DS:000AH, ambas com 10 bytes.

```
MOV CX, 10     ; CX recebe tamanho das strings
MOV DI, 10     ; Endereço de destino.
SUB SI, SI     ; SI = 0 (endereço de origem).
PUSH DS
POP ES         ; Faz ES apontar o mesmo segmento de DS (assim destino é
               ; dado por DS:DI).
REPE CMPSB     ; Compara as duas strings até encontrar um caracter diferente.
```

6.4 SCAS "Compara caracter da string com AL"

Compara o acumulador com ES:DI (subtrai ES:DI de AX ou AL mas só afeta os flags). Atualiza DI como em MOVS. Afeta os flags AF, CF, OF, PF, SF e ZF. Existem SCASB (8 bits) e SCASW (16 bits).

Ex.: Rotina para procurar o caracter "." na string em DS:0000H de tamanho 10 bytes.

```
MOV CX, 10     ; CX recebe tamanho da string
MOV AL, '.'    ; AL recebe o ASCII de ".".
SUB DI, DI     ; DI = 0 (endereço da string).
PUSH DS
POP ES         ; Faz ES apontar o mesmo segmento de DS (assim o endereço
               ; da string é dado por DS:DI).
REPNE SCASB    ; Busca o caracter na string.
```

6.5 LODS "Carrega o acumulador com string"

Copia o conteúdo da posição de memória DS:SI para AL ou AX. Atualiza SI como em MOVS. Existem LODSB (8 bits) e LODSW (16 bits).

6.6 STOS "Armazena o acumulador em string"

Copia o conteúdo da posição de AL ou AX para memória ES:DI. Atualiza DI como em MOVS. Existem STOSB (8 bits) e STOSW (16 bits).