PC Assembler

COM PROGRAMA-EXEMPLO
TOTALMENTE COMENTADO

Daniel G. A. QUADROS

ÍNDICE

ÍNDICE **2**

PREFÁCIO DA EDIÇÃO ELETRÔNICA **3**

1 INTRODUÇÃO **4**

1.1 O PC-IBM E O MICROPROCESSADOR 8088 4
1.2 A FAMÍLIA DE MICROPROCESSADORES 8088/86/188/186/286 4
1.3 O QUE É UM ASSEMBLER 5

2 CONCEITOS BÁSICOS **6**

2.1 COMO O 8088 ENDEREÇA MEMÓRIA 6
2.2 OS REGISTRADORES DO 8088 6
2.3 SÍMBOLOS 8
2.4 ELEMENTOS DA LINGUAGEM ASSEMBLY 9
2.5 FORMATO DE UM PROGRAMA EM LINGUAGEM ASSEMBLY 10
2.6 DEFININDO E REFERENCIANDO SEGMENTOS 11
2.7 DEFININDO VARIÁVEIS 12
2.8 DEFININDO RÓTULOS 13
2.9 DEFININDO CONSTANTES 14
2.10 ACESSANDO E ALTERANDO CARACTERÍSTICAS DE UM SÍMBOLO 14
2.11 MODOS DE ENDEREÇAMENTO 14
2.12 INTERRUPÇÕES DE HARDWARE E SOFTWARE 15

3 AS INSTRUÇÕES **17**

3.1 INTRODUÇÃO 17
3.2 INSTRUÇÕES DE MOVIMENTAÇÃO 18
3.3 INSTRUÇÕES ARITMÉTICAS 26
3.4 INSTRUÇÕES LÓGICAS 41
3.5 INSTRUÇÕES DE ENTRADA E SAIDA 52
3.6 INSTRUÇÕES DE MANIPULAÇÃO DE CADEIAS 53
3.7 INSTRUÇÕES DE DESVIO 60
3.8 INSTRUÇÕES DE CONTROLE DO PROCESSADOR 75

4 TÓPICOS AVANÇADOS **80**

4.1 MÓDULOS 80
4.2 MAIS DETALHES SOBRE SEGMENTOS 81
4.3 DIRETIVA RECORD E OS OPERADORES MASK E WIDTH 83
4.4 DIRETIVA STRUC 84

5 PROGRAMA EXEMPLO **86**

APÊNDICE - INSTRUÇÕES X CÓDIGOS **98**

PREFÁCIO DA EDIÇÃO ELETRÔNICA

Em 1983, quando eu era um engenheiro eletrônico recém formado, comprei o meu primeiro microcomputador pessoal. Era um TK-82C, uma cópia nacional do ZX-81 da Sinclair. Naquele tempo, muito anterior à internet, a principal fonte de informações eram as revistas. Após algum tempo comecei a colaborar, de forma não sistemática, com revistas como a **Micro Sistemas** e **MicroBits** e a participar de encontros de usuários. Foi em um deles que conheci o Nelson Santos.

Nelson tinha escrito um livro sobre o TK-82, estava trabalhando como editor para a **Editora Campos** e, como os cilônios, tinha um plano. O plano era um livro com vários programas comentados (*Jogos em Linguagem de Máquina para a Linha Sinclair*), cada programa escrito por um autor. Este plano acabou não indo em frente e o Nelson perguntou se eu não tinha alguma idéia para um livro. A única coisa que veio na cabeça foi escrever algo sobre aquilo que eu estava trabalhando no dia-a-dia: o PC-IBM.

Estávamos no tempo da reserva de mercado e eu trabalhava em uma empresa idealista, que decidiu fazer o seu próprio BIOS e sistema operacional (compatível com o PC-DOS). O PC ainda estava na sua infância e muita coisa era escrita na linguagem *Assembly* (se bem que o sistema operacional foi escrito na sua maior parte em Pascal e posteriormente re-escrito em C).

O Nelson gostou da idéia e assim nasceu o que viria a ser o primeiro volume do **PC Assembler**. O livro foi escrito em um Unitron (clone do Apple][) e os originais foram enviados impressos para serem re-digitados e compostos na editora.

O livro foi um relativo sucesso: foram impressos 6253 livros e vendidos 6066 exemplares, em quatro edições, do segundo semestre de 86 até o primeiro semestre de 93, quando a Editora Campus decidiu não mais reeditar a obra e me devolveu os direitos de publicação.

No final de 2007, resolvi disponibilizar o texto no meu blog (http://dqsoft.blogspot.com). Após uma pequena odisséia (descrita no blog) consegui recuperar o texto do disco original, mas em formato “txt” e sem o programa exemplo.

Quase dois anos e meio depois, um dos freqüentadores do meu blog (Cacio Gazola) me repassou o texto em um formato mais adequado para edição. Nesta “edição eletrônica” (cujo preparo se arrastou por vários meses) fiz uma pequena revisão no texto, formatei todo o conteúdo e acrescentei capa, índice e o programa exemplo.

Espero que seja útil para alguém.

Daniel Quadros

Junho/2010

1 INTRODUÇÃO

1.1 O PC-IBM E O MICROPROCESSADOR 8088

Em 1981 o panorama do mercado de microcomputadores sofreu um forte abalo: o lançamento pela IBM do seu primeiro microcomputador, o PC (*Personal Computer*).

Rapidamente o PC-IBM transformou-se no padrão de fato para os microcomputadores profissionais. Uma grande quantidade de hardware e software de boa qualidade, está atualmente disponível no mercado, permitindo o emprego do PC-IBM nos mais diversos ambientes.

No Brasil os micros compatíveis com o PC-IBM surgiram em 1983. No final de 1985 já atingiam a marca de mais de duas dúzias de modelos no mercado, com uma venda da ordem de 500 unidades por mês.

Um dos motivos básicos do sucesso do PC-IBM foi o uso de um microprocessador de 16 bits. O uso do 8088 no PC-IBM forneceu maior capacidade de processamento e de endereçamento que os micros de 8 bits então existentes, permitindo o desenvolvimento de uma nova geração de hardware e software.

Estranhamente, apesar do grande domínio do PC-IBM no mercado de microcomputadores, existe uma falta de bons livros sobre a programação do 8088.

Este livro se propõe basicamente a duas finalidades:

- Ensinar o programador que já conheça outros microprocessadores a programar o 8088 e
- Ser uma referência para o programador que já conhece o 8088.

1.2 A FAMÍLIA DE MICROPROCESSADORES 8088/86/188/186/286

O microprocessador 8086 foi desenvolvido pela INTEL Corporation, firma que já havia sido responsável pelo primeiro microprocessador , o 4004, e pelo processador de 8 bits mais difundido, o 8080.

Um dos primeiros microprocessadores de 16 bits, o 8086 se caracteriza por:

- Ter internamente registradores de 16 bits,
- Ter um conjunto completo de instruções aritméticas de 16 bits, inclusive multiplicação e divisão e
- Ter uma via de dados externa (de comunicação com memória e periféricos) também de 16 bits.

Posteriormente a INTEL lançou o 8088, que é idêntico ao 8086 exceto pela via de dados externa que é de 8 bits. Esta diferença é transparente ao programador, que utiliza as mesmas instruções e continua acessando números de 16 bits na memória. O 8088 cuida internamente de transformar os acessos a um WORD (número de 16 bits) em dois acessos a BYTEs (números de 8 bits). Do ponto de vista de hardware, o 8088 permite uma forte simplificação, sem grande impacto no desempenho.

Mais recentemente a família foi ampliada com o lançamento de três novos membros, capazes de executar o mesmo conjunto de instruções e mais algumas outras:

- 80188 / 80186 - incluem em uma única pastilha o microprocessador e circuitos de apoio como controladores de DMA e interrupção

- 80286 - amplia a capacidade de endereçamento, através da incorporação de um gerenciador de memória que fornece recursos de proteção de memória e memória virtual.

1.3 O QUE É UM ASSEMBLER

Um processador executa instruções, que são armazenadas em memória como números - na chamada "linguagem de máquina". A programação destas instruções diretamente nesta linguagem é bastante trabalhosa. Em primeiro lugar é necessário consultar uma tabela dos códigos das várias instruções. Em seguida é necessário determinar o endereço das instruções para as quais são efetuados desvios. Uma vez obtido o programa, se houver necessidade de incluir ou retirar instruções, estes endereços terão de ser re-determinados.

O ***Assembler*** (ou Montador) é um programa que automatiza estas tarefas (e outras mais). Ele recebe como entrada o programa escrito em outra forma - na "linguagem ***Assembly***[1]". Nesta linguagem as instruções são referenciadas por **MNEMÔNICOS** - abreviações de fácil memorização - e os endereços por **SÍMBOLOS** - nomes aos quais o *Assembler* associa os endereços reais.

Na linguagem *Assembly* temos também instruções para o programa *Assembler* - são as **DIRETIVAS**.

Neste livro são utilizados os mnemônicos e nomes das diretivas definidos pelo fabricante do 8088 (a INTEL), que são utilizados também pelos *assemblers* disponíveis no mercado.

O Assembler do 8088 possui algumas características especiais:

- O número de mnemônicos é bastante reduzido em comparação com o número de instruções em linguagem de máquina. A cada mnemônico corresponde uma classe de instruções correlatas, sendo a instrução definida pelos operandos. Por exemplo, o mnemônico "MOV" corresponde tanto à movimentação do conteúdo de um registrador para outro como à movimentação de um valor para uma posição da memória.

- Existe um forte conceito de TIPO nos símbolos. Por exemplo, ao se reservar uma área para o armazenamento de dados, é necessário especificar qual o tipo de dado (byte, word, etc.) que será armazenado. Esta informação é utilizada para gerar o tipo correto de instrução e para verificar a validade dos operandos.

- Em alguns casos existem duas instruções, correspondentes a um mesmo mnemônico, que executam a mesma operação, porém com diferente ocupação de memória e/ou tempo de execução. Por este motivo o Assembler tem que ter a capacidade de otimizar o código.

[1] A edição impressa utilizava a terminologia “linguagem Assembler”. Nesta nova edição estou usando “linguagem Assembly” que é a terminologia mais usual atualmente.

2 CONCEITOS BÁSICOS

2.1 COMO O 8088 ENDEREÇA MEMÓRIA

O 8088 é capaz de endereçar até 1 Mbyte de memória (1024x1024 bytes), o que corresponde a 20 bits de endereço, embora os seus registradores se limitem a 16 bits.

Para obter estes 20 bits os endereços são normalmente tratados como duas partes: o **SEGMENTO** e o **DESLOCAMENTO** ("*offset*"). O endereço final é obtido multiplicando-se o segmento por 16 e somando-se o deslocamento, ignorando o "vai um" do bit mais significativo.

Por exemplo, podemos acessar a posição de endereço "físico" 100H (hexadecimal) através do seguinte par *segmento:offset* :

```
0001H:00F0H  00010H  <-  segmento * 16
       +     00F0H   <-  deslocamento
             ------
                00100H  <-  endereço "físico"
```

A mesma posição pode ser acessada através de 0000H:0100H ou 0010H:000H ou FF00H:1100H (neste caso, o "vai um" do resultado é ignorado).

Ao armazenar um word na memória, o 8088 coloca primeiro o byte MENOS significativo e depois o MAIS significativo. Por exemplo, o valor 1234H é armazenado na posição de endereço "físico" 300H da seguinte forma:

```
300H -> 34H
301H -> 12H
```

Da mesma forma, um valor de 32 bits (um "DOUBLE WORD", como um endereço composto por segmento e deslocamento) é armazenado com o word MENOS significativo precedendo o MAIS. Por exemplo, o endereço 1234H:5678H seria armazenado em quatro bytes na seguinte ordem:

```
78H, 56H, 34H, 12H
```

2.2 OS REGISTRADORES DO 8088

O 8088 possui um conjunto de 14 registradores internos, que podemos classificar em quatro categorias: de uso geral, de base e índice, de segmento e de controle.

Registradores de uso geral

Esta categoria é composta por 4 registradores de 16 bits, referenciados pelos nomes AX, BX, CX e DX. Estes mesmos registradores podem ser acessados como 8 registradores de 8 bits: AH, AL, BH, BL, CH, CL, DH e DL. Os registradores "H" contêm a parte mais significativa dos correspondentes "X" e os "L" a parte menos significativa.

Apesar de estes registradores poderem ser utilizados indistintamente na maioria das instruções, eles têm algumas peculiaridades e usos mais comuns:

AX - é o "acumulador". Tem uso especial nas instruções de entrada e saída, multiplicação, divisão, e manipulação de cadeia. De forma geral, as movimentações entre registradores e memória e as instruções aritméticas e lógicas com operando imediato possuem variantes mais curtas (e rápidas) quando o registrador utilizado é o acumulador.

BX - é o registrador de "base", usado para acesso indireto da memória, principalmente tabelas.

CX - é o "contador". Tem uso especial nas instruções repetitivas - "LOOP" e de manipulação de cadeias.

DX - é um registrador auxiliar. Tem uso especial nas operações de entrada e saída e serve de "extensão" do acumulador nas instruções de multiplicação e divisão.

Registradores de Base e Índice

Estes registradores são normalmente utilizados para apontar para dados, isto é, armazenar um deslocamento. Ao contrário dos de uso geral, não é possível acessar independentemente as partes mais e menos significativas destes registradores.

Os registradores deste grupo são denominados:

SP - ("*stack pointer*") é o ponteiro para a pilha do processador, indicando sempre o último dado nela colocado. É utilizado pelas instruções "PUSH", "POP", "CALL", "RET", "INT" e "IRET".

BP - ("*base pointer*") é utilizado normalmente como ponteiro para acessar dados armazenados na pilha.

SI - ("source index") e

DI - ("*destination index*") podem conter o deslocamento de uma posição de memória que se deseja acessar ou um valor ("índice") a ser somado a um registrador base (BX ou BP) para se obter o deslocamento de uma posição de memória. Tem também uso especial nas instruções de manipulação de cadeias.

Registradores de Segmento

Conforme já dito, o acesso à memória é feito sempre a partir de dois valores: o segmento e o deslocamento. Os deslocamentos são obtidos a partir das instruções, dos registradores de base e índice (BX, BP, SP, SI e DI) ou de uma combinação destes elementos. Já os segmentos são sempre obtidos de um dos quatro registradores de segmento:

CS - aponta para o segmento de código, no qual são acessadas as instruções;

SS - aponta para o segmento da pilha, utilizado pelas instruções PUSH / POP / CALL / RET / INT e IRET;

DS - aponta para o segmento de dados, utilizado preferivelmente no acesso a dados na memória e

ES - aponta para o segmento extra, que é uma alternativa para o acesso a dados na memória e utilizado nas instruções de manipulação de cadeias.

Registradores de Controle

Neste grupo temos dois registradores:

IP - ponteiro de instruções, contém o deslocamento (em relação ao segmento de código apontado por CS) da próxima instrução a executar. Este registrador não é acessível diretamente, sendo manipulado pelas instruções de desvio.

"F" - este registrador ("*flags*") contém vários bits que são utilizados pelo 8088 para indicar resultados ou controlar operações:

Indicadores de resultado

AF "vai um" auxiliar, utilizado em operações sobre números no formato "BCD compactado".

CF "vai um" ("*carry*"), indica que o resultado de uma operação não coube na área destinada. Usado também nas instruções de deslocamento e rotação.

OF "*overflow*", indica que o resultado de uma operação aritmética com sinal não coube na área destinada.

PF paridade, contém "1" quando o resultado da última operação continha um número par de "1"s.

SF sinal, corresponde ao bit mais significativo do resultado da última operação.

ZF zero, indica se o resultado da última operação foi zero.

flags de controle

DF controla as operações de manipulação de cadeias, selecionando auto-incremento ("0") ou auto-decremento("1").

IF habilita ("1") ou desabilita ("0") interrupções externas

TF quando em "1" coloca o processador no modo "passo a passo", usado para depuração.

Quando os *flags* são colocados na pilha (por uma instrução PUSHF ou uma interrupção) eles assumem a seguinte forma:

```
bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Flag    X  X  X  X OF DF IF TF SF ZF  X AF  X PF  X CF
```

As posições marcadas com "X" correspondem a bits não utilizados e de conteúdo indefinido. Esta organização é tal que o byte menos significativo contém *flags* equivalentes aos do microprocessador 8080.

2.3 SÍMBOLOS

Os **SÍMBOLOS** são os nomes definidos pelo programador para indicar os endereços de instruções e variáveis. No assembler do 8088 existe uma forte distinção entre duas classes de símbolos:

- Os ***RÓTULOS*** - que correspondem a endereços de instruções, e
- As ***VARIÁVEIS*** - que correspondem a endereços de dados

Todo símbolo possui três características:

- O SEGMENTO em que ele foi definido
- O DESLOCAMENTO em relação a esse segmento
- O TIPO de símbolo

  Se rótulo:

  - ***NEAR*** neste caso os desvios para este rótulo envolvem apenas alteração de deslocamento (registrador IP) ou
  - ***FAR*** neste caso os desvios envolvem alteração de segmento e deslocamento (registradores CS e IP)

  Se variável:

  - ***BYTE*** *ocupa um byte*
  - ***WORD*** *ocupa dois bytes*
  - ***DWORD*** *ocupa quatro bytes*

O tipo de símbolo irá distinguir entre as variantes de uma mesma instrução. Por exemplo, a instrução

```
INC XIS
```

pode significar o incremento de um byte ou um word; o tipo associado a XIS é que irá definir qual das duas operações é desejada.

2.4 ELEMENTOS DA LINGUAGEM ASSEMBLY

A linguagem assembler do 8088 utiliza os seguintes elementos:

Constantes Numéricas

Uma constante numérica pode ser fornecida em binário (base 2), octal (base 8), decimal (base 10) ou hexadecimal (base 16). A base de uma constante numérica é indicada através de uma letra ao final da constante:

B - binário
O - octal
D - decimal (pode ser omitida)
H - hexadecimal, deve começar por um dígito (se necessário, colocar um "0" no inicio)

Uma constante pode ser precedida do sinal "-", para indicar um valor negativo. Vejamos alguns exemplos:

```
 10011B   ;  19 em binário
-10011B   ; -19 em binário
    23O   ;  19 em octal
   -23O   ; -19 em octal
    19D   ;  19 em decimal
    19    ;  idem
   -19D   ; -19 em decimal
    13H   ;  19 em hexadecimal
   -13H   ; -19 em hexadecimal
```

Todos os números devem ser representáveis em 16 bits (mais um bit de sinal):

```
-1111111111111111B <= cte binária <= 1111111111111111B
    -177777O       <= cte octal   <=     177777O
     -65535        <= cte decimal <=      65535
     -0FFFFH       <= cte hexa    <=     0FFFFH
```

Cadeias de Caracteres

Uma cadeia de caracteres é uma sequência de caracteres cercada por apostrofes ('), por exemplo:

```
 'Cadeia'
```

Nomes

Um **nome** é formado por uma sequência de letras e dígitos, iniciada por uma letra. Normalmente os caracteres @ _ e ? são considerados como letras e um nome pode ter até 31 caracteres. No caso de ? ser permitido, ele não pode ser usado sozinho como um nome. Alguns nomes válidos:

```
NOME_125
E_QUE_TAL_@?
UM_NOME_GRANDE_PARA_TERMINAR
```

2.5 FORMATO DE UM PROGRAMA EM LINGUAGEM ASSEMBLY

Um programa escrito em linguagem assembly é composto por:

- linhas em branco,
- linhas de comentário,
- linhas de instrução e
- linhas de diretiva

As linhas de **COMENTÁRIO** se caracterizam por terem como primeiro caractere não branco um ";". Os caracteres seguintes ao ";" são ignorados pelo Assembler.

(Observação: o termo **BRANCO** é usado aqui no sentido de englobar tanto o caractere ESPAÇO como o caractere TAB).

As linhas de INSTRUÇÃO são as que contém um mnemônico e irão gerar uma instrução de máquina. Tem o seguinte formato:

```
nome:  prefixo  mnemônico  operandos  ; comentário
```

O único campo sempre obrigatório é o do mnemônico; o campo de operandos será obrigatório ou opcional dependendo da instrução. No caso de se ter mais de um operando eles são separados por uma vírgula.

As linhas de DIRETIVA tem o seguinte formato:

```
nome  diretiva  operandos  ; comentário
```

O único campo sempre obrigatório é a diretiva; os campos de nome e operandos podem ser obrigatórios ou opcionais conforme a diretiva. Notar que (ao contrário da linha de instrução) o nome inicial NÃO é seguido de ":".

Finalmente, todo programa termina com uma diretiva END:

```
END  rótulo
```

O rótulo (que é opcional) indica a primeira instrução a ser executada.

2.6 DEFININDO E REFERENCIANDO SEGMENTOS

Um segmento é iniciado pela diretiva SEGMENT:

```
nome SEGMENT
```

e terminado pela diretiva ENDS:

```
nome ENDS
```

Todos os símbolos declarados entre estas duas diretivas terão associados a si o segmento "nome".

Para verificar se um determinado símbolo pode ser acessado e como ele deve ser acessado, o assembler necessita saber que valores o programador colocou nos registradores de segmento.

Isto é feito através da diretiva ASSUME:

```
ASSUME segreg:segnome {,segreg:segnome ...}
```

onde "segreg" corresponde ao nome de um registrador de segmento (CS, DS, ES ou SS) e "segnome" ao nome de um segmento ou a palavra NOTHING.

Vejamos agora como é escolhido o registrador de segmento a ser utilizado em um acesso à memória:

1. A execução de instruções é sempre feita no segmento indicado por CS;
2. O acesso à pilha via SP utiliza sempre o segmento indicado por SS;
3. Algumas instruções de manipulação de cadeias utilizam sempre o segmento contido em ES;

Nos demais casos, existe um registrador padrão, imposto por "hardware", que pode ser substituído por outro através da colocação de um "prefixo" na instrução (o "***SEGMENT OVERRIDE***"). O registrador padrão é DS, exceto quando é utilizado como base o registrador BP (neste caso o padrão é SS). Consideremos alguns exemplos:

```
        MOV   AX,VAR    utiliza DS
        MOV   AX,[BX]   utiliza DS
        MOV   AX,[BP]   utiliza SS
```

A utilização de um segmento diferente do padrão pode ser forçada pelo programador, precedendo o operando por "segreg:", como no exemplo abaixo:

```
        INC  CS:XIS
```

Neste caso a variável XIS será acessada através de CS ao invés de DS.

O "segment override" pode também ser gerado automaticamente pelo assembler. Isto ocorre quando o assembler sabe (pelos ASSUMEs) que o segmento desejado não está no registrador padrão mas sim em um outro. Por exemplo, suponhamos que a variável XIS foi definida no segmento X e codificou-se os seguintes comandos:

```
        ASSUME  DS:Y, CS:X
        INC     XIS
```

Neste caso o assembler se encarrega de colocar o prefixo "CS:" na instrução.

Finalmente, temos a palavra "**NOTHING**". Ela permite informar ao assembler que o conteúdo de um registrador é indefinido, só sendo possível o seu uso através da codificação explicita do "segment override".

2.7 DEFININDO VARIÁVEIS

A definição de variáveis é feita através de três diretivas: DB ("define byte"), DW ("define word") e DD ("define double-word"), que definem variáveis de 8, 16 e 32 bits.

Além de definir os símbolos e reservar espaço, estas diretivas permitem especificar o conteúdo inicial da variável:

```
XIS  DB  0
```

define a variável XIS, do tipo byte, com valor inicial 0;

```
YPS  DW  ?
```

define a variável YPS, do tipo word, com valor inicial indeterminado;

```
ZE   DB  1,2,3
```

define três bytes, com valores iniciais 1, 2 e 3. O símbolo ZE permite acessar o primeiro byte;

```
NOME DB  "123"
```

define três bytes, tendo como valores iniciais os caracteres "1", "2" e "3" (o que é diferente dos números 1, 2 e 3);

```
ZEROS DW 10 DUP(0)
```

define dez words, todos com valor inicial 0 e

```
ZEROS_E_UNS  DB  5 DUP (0,1)
```

define cinco bytes, com os valores 0, 1, 0, 1 e 0.

2.8 DEFININDO RÓTULOS

A forma mais comum de se definir um rótulo é colocando no inicio da linha o nome do rótulo, seguido de ":", como abaixo:

```
AQUI:
     INC  XIS
ALI:
     INC YPS
```

O rótulo AQUI aponta para a instrução "INC XIS", o rótulo ALI para a instrução "INC YPS". Utilizando esta forma de definição o rótulo fica sempre com o tipo "near".

No caso de rotinas, é preferível o uso das diretivas PROC e ENDP:

```
nome  PROC  tipo
  ;
  ;   codigo da rotina
  ;
nome  ENDP
```

Neste caso, "nome" aponta para a instrução seguinte à diretiva PROC e tem o tipo "NEAR" ou "FAR" nela especificado. Este tipo irá influir na chamada ao procedimento, que pode ser:

Intra-segmento - se envolve apenas a alteração de deslocamento, sendo salvo na pilha apenas IP, ou;

Inter-segmento - se envolve alteração de segmento e deslocamento, sendo salvo na pilha IP e CS.

A instrução de retorno (RET) de subrotina tem que ser "casada" com a instrução de chamada (CALL). O tipo da instrução CALL é determinado pelo tipo do rótulo da rotina. Já o tipo da instrução RET é determinado pelo tipo da rotina (delimitada por PROC / ENDP) em que ela foi usada. As instruções RET fora de rotinas são do tipo NEAR (intra-segmento).

Por exemplo, consideremos o trecho abaixo:

```
XIS  PROC FAR
     RET       ; 1

YPS  PROC NEAR
     RET       ; 2
YPS  ENDP

     RET       ; 3

XIS  ENDP
```

Os RETs 1 e 3 estão dentro da rotina XIS, que é FAR - logo eles são do tipo inter-segmento. O RET 2 está dentro da rotina YPS (por sua vez dentro de XIS) e é um RET NEAR - prevalece o tipo da rotina mais interna.

Existe ainda mais uma forma de definir símbolos, que pode ser utilizada tanto para variáveis como rótulos:

```
nome  LABEL  tipo
```

que associa o símbolo "nome" do tipo especificado à próxima posição definida. Desta forma, é possível definir um tipo alternativo para uma posição de memória:

```
A_BYTE  LABEL  BYTE
A_WORD  DW     ?
```

define um word, referenciável por A_WORD, cujo primeiro byte é acessível pelo símbolo A_BYTE.

2.9 DEFININDO CONSTANTES

Na definição de rótulos e variáveis associamos nomes a endereços de memória. É possível também associar nomes a valores, isto é definir CONSTANTES. Isto é feito através da diretiva EQU ("equate"):

```
nome  EQU  expressão
```

por exemplo:

```
TAMANHO  EQU  20
AREA     DB   TAMANHO DUP (?)
```

associa o nome TAMANHO ao valor 20 e o utiliza para definir uma área de 20 bytes.

A vantagem do uso do EQU é a melhora da legibilidade do programa e a possibilidade de alterar todas as referências a um valor alterando uma única linha.

2.10 ACESSANDO E ALTERANDO CARACTERÍSTICAS DE UM SÍMBOLO

Ao se referenciar um símbolo é possível precede-lo de um OPERADOR de forma a acessar ou alterar uma de suas características. Os operadores utilizados são:

(nos exemplos abaixo supomos definida a variável VAR do tipo word)

OFFSET símbolo — Fornece o deslocamento do símbolo

Ex:
```
MOV  BX,OFFSET VAR
```
Coloca em BX o deslocamento de VAR no segmento em que VAR foi definida

SEG símbolo — Fornece o segmento do símbolo

Ex:
```
MOV  BX,SEG VAR
```
Coloca em BX o segmento que VAR foi definida

Tipo PTR símbolo — Permite alterar o tipo de um símbolo

Ex:
```
INC  BYTE PTR VAR
```
Incrementa o primeiro byte de VAR

SHORT rótulo — Usado com a instrução JMP para indicar que o rotulo está a uma distância de -128 a +127 bytes da instrução seguinte, sendo possível um desvio curto (ver Instruções de Desvio, item 3.7).

Ex
```
JMP  NEAR DESTINO
```
Comanda a geração de um desvio curto

2.11 MODOS DE ENDEREÇAMENTO

Os operandos das instruções do 8088 podem vir de vários lugares:

- Registradores,
- Da própria instrução (valor imediato) ou
- Da memória.

No caso de operando obtido da memória, necessitamos de um segmento e de um deslocamento. Já vimos como é determinado o segmento, vejamos agora como pode ser especificado o deslocamento:

1) através de um valor direto, armazenado na própria instrução:

```
      MOV   AX,VAR
```

2) indiretamente, através de um registrador de base ou índice, opcionalmente somado a uma distância de 8 (-128 a +127) ou 16 (-32268 a +32267) bits:

```
MOV   AX,[BX]
MOV   AX,[BX+1]
MOV   AX,TABELA[BX]                ; (o deslocamento final
                                            ; é a soma de BX com o
                                            ; deslocamento do símbolo
                                            ; TABELA)
```

3) indiretamente, pela soma de um registrador base e um registrador índice, opcionalmente somado a uma distância de 8 ou 16 bits:

```
MOV   AX,[BX+SI]
MOV   AX,TABELA[BP][DI-300]        ; (deslocamento =
                                            ;  offset TABELA
                                            ;  - 300 + BP + DI
```

2.12 INTERRUPÇÕES DE HARDWARE E SOFTWARE

Uma INTERRUPÇÃO é um desvio causado por:

a) Um evento externo, normalmente uma sinalização de um dispositivo de entrada ou saída, ou

b) Um evento interno, associado à instrução sendo executada.

No primeiro caso, a interrupção será assíncrona ao programa (poderá ocorrer após a execução de qualquer instrução) e é causada por hardware externo ao processador.

No segundo caso temos quatro possibilidades:

b1) Interrupção por erro de divisão (o quociente não cabe no registrador destino),

b2) Interrupção de execução passo a passo, que é controlada pelo flag TF,

b3) Interrupção causada pela execução da instrução INTO ("interrupt on overflow") com o flag OF ativo, ou

b4) Interrupção causada pela execução da instrução INT - este tipo de interrupção é chamado de **INTERRUPÇÃO DE SOFTWARE**.

As interrupções do 8088 são ditas **VETORIZADAS**: a cada interrupção está associado um número de 0 a 255 que é utilizado para acessar uma tabela localizada a partir do endereço "físico" 0. Esta tabela contem o endereço (deslocamento e segmento) da rotina que irá tratar a interrupção. Por exemplo, o endereço da rotina que trata a interrupção 40H está armazenado a partir da posição "física" 100H (4 * 40H).

Alguns vetores têm funções pré-definidas:

0 - erro em divisão

1 - passo a passo

2 - interrupção não mascarável de hardware

4 - interrupção causada pela instrução INTO

Quando ocorre uma interrupção as seguintes ações ocorrem:

1. O registrador de flags é colocado na pilha,
2. Os flags TF e IF são limpos, desabilitando interrupções externas e de passo a passo,
3. CS e IP são colocados na pilha e
4. É feito o desvio para a rotina cujo endereço está no vetor correspondente.

Uma instrução especial (IRET) é normalmente utilizada ao final do tratamento de uma interrupção para retirar IP, CS e os flags da pilha, retornado ao trecho interrompido.

3 AS INSTRUÇÕES

3.1 INTRODUÇÃO

Neste capítulo são descritas todas as instruções da linguagem Assembly do 8088. A descrição segue sempre o seguinte formato:

- Mnemônico - nome da instrução
- Codificação
- Operação realizada
- Flags afetados
- Comentários e exemplos

Na descrição da codificação são usados os seguintes termos:

reg	um registrador de uso geral ou de base/índice (AX, BX, CX, DX, AH, AL, BH, BL, CH, CL, DH, DL, SI, DI, BP ou SP)
segreg	um registrador de segmento (CS, DS, ES ou SS)
regword	um registrador de uso geral de 16 bits ou de base/índice (AX, BX, CX, DX, SI, DI, BP ou SP)
mem	uma referência (direta ou indireta) à memória
imed	uma constante
rótulo	um rótulo qualquer

3.2 INSTRUÇÕES DE MOVIMENTAÇÃO

MOV - Movimenta dados

Codificação:

```
MOV     reg,reg
MOV     mem,reg
MOV     reg,mem
MOV     reg,imed
MOV     mem,imed
MOV     reg,segreg
MOV     segreg*,reg
MOV     segreg*,mem
MOV     mem,segreg
```

* não é permitido CS

Operação:

A instrução **MOV** movimenta dados (byte ou word) entre registradores e memória.

Flags Afetados

Nenhum

Comentários e Exemplos

A instrução **MOV** é bastante regular no que diz respeito ao uso com registradores e memória. No caso de registradores de segmento, notar que não existe a movimentação de valor imediato. Notar também que não é permitido escrever em CS.

Alguns exemplos válidos:

```
MOV     AX,VALOR_IMEDIATO
MOV     DS,AX
MOV     SI,TABELA[SI]
MOV     AX,CS
```

PUSH - Coloca Word na Pilha

Codificação:

```
        PUSH    regword
        PUSH    mem
        PUSH    segreg
```

Operação:

O registrador SP é decrementado de 2 e o operando é colocado nas posições SS:[SP] e SS:[SP+1].

Flags Afetados

Nenhum

Comentários e Exemplos

A instrução **PUSH** é normalmente utilizada para salvar temporariamente um registrador (que será recuperado através da instrução **POP**) ou passar um parâmetro para uma sub-rotina.

Cabe lembrar que SP aponta sempre para o último dado colocado na pilha e que o segmento utilizado é sempre SS.

Notar que não existe PUSH de um valor imediato. Por outro lado, PUSH é uma das poucas instruções capazes de fazer dois acessos à memória. A sequência

```
        PUSH    VARIAVEL_1
        POP     VARIAVEL_2
```

move o conteúdo de VARIAVEL_1 para VARIAVEL_2 sem uso de registradores intermediários.

Alguns exemplos:

```
        PUSH    AX
        PUSH    DS
        PUSH    SS
        PUSH    SP                ; salva SP já decrementado de 2
        PUSH    CS:[BX+SI+1]
```

POP - Retira Word da Pilha

Codificação:

```
POP     regword
POP     mem
POP     segreg*
```

(* CS não é permitido)

Operação:

O operando recebe o conteúdo das posições SS:[SP] e SS:[SP+1]; o registrador SP é incrementado de 2.

Flags Afetados

Nenhum

Comentários e Exemplos

A instrução **POP** é normalmente utilizada para recuperar um registrador armazenado na pilha através da instrução **PUSH**.

Cabe lembrar que SP aponta sempre para o último dado colocado na pilha e que o segmento utilizado é sempre SS.

Notar que "POP CS" não é permitido. Por outro lado, POP é uma das poucas instruções capazes de fazer dois acessos à memória. A sequência

```
PUSH    VARIAVEL_1
POP     VARIAVEL_2
```

move o conteúdo de VARIAVEL_1 para VARIAVEL_2 sem uso de registradores intermediários.

Alguns exemplos:

```
POP     AX
POP     DS
POP     SS
POP     SP
POP     CS:[BX+SI+1]
```

XCHG - Troca

Codificação:

```
        XCHG    reg,reg
        XCHG    reg,mem
        XCGH    mem,reg
```

Operação:

Os conteúdos dos dois operandos são trocados.

Flags Afetados

Nenhum

Comentários e Exemplos

Esta instrução é bastante poderosa, pois realiza a troca sem usar registradores ou variáveis auxiliares. Permite também ler e alterar uma posição de memória com uma única instrução.

Notar que não é permitida referência a registradores de segmento.

Alguns exemplos:

```
        XCHG    AX,BX
        XCHG    AH,AL
        XCHG    AH,BL
        XCHG    AL,VARIAVEL
```

XLAT - Converte AL

Codificação:

```
        XLAT    var
```

Operação:

A instrução **XLAT** substitui o conteúdo de AL pelo byte em [BX+AL].

Flags Afetados

Nenhum

Comentários e Exemplos

A instrução XLAT é ao mesmo tempo uma instrução poderosa (uma rotina equivalente teria pelo menos três instruções e alteraria pelo menos um registrador) e limitada (utiliza sempre AL e BX).

Seu uso típico é a conversão de um byte através de uma tabela:

```
MOV     AL,VALOR_A_CONVERTER
MOV     BX,OFFSET TABELA
XLAT    TABELA
```

É importante notar que o operando **NÃO** irá especificar o offset inicial da tabela (que é obtido de BX). Sua finalidade é permitir que o assembler determine qual o registrador de segmento a utilizar.

Exemplos:

```
        XLAT    TAB_1
        XLAT    CS:TAB_2
```

LEA - Carrega Endereço Efetivo

Codificação:

```
        LEA     regword,mem
```

Operação:

A instrução **LEA** carrega em um registrador o deslocamento do operando.

Flags Afetados

Nenhum

Comentários e Exemplos

Para entendermos a instrução LEA é interessante comparar as três instruções abaixo:

```
        MOV     BX,XIS
        MOV     BX,OFFSET XIS
        LEA     BX,XIS
```

A primeira carrega BX com o **CONTEÚDO** da variável XIS; as duas últimas carregam BX com o **DESLOCAMENTO** (*offset*) da variável XIS. A instrução LEA tem duas diferenças em relação ao "MOV OFFSET":

- Carrega um registrador com o offset correspondente a uma referência qualquer à memória, inclusive com indexação e
- Leva em conta o valor dos registradores de segmentos (especificado via ASSUME).

Por outro lado, no caso mais comum de carga do deslocamento de um símbolo em relação ao segmento onde ele foi definido, o "MOV OFFSET" é mais curto e rápido.

Alguns exemplos:

```
        LEA     AX,VAR
        LEA     SI,TABELA[BX+SI+3]
```

LDS - Carrega Ponteiro p/ DS

Codificação:

```
        LDS     regword,mem
```

Operação:

A instrução **LDS** carrega um ponteiro de 32 bits da memória para DS (segmento) e um registrador (deslocamento).

Flags Afetados

Nenhum

Comentários e Exemplos

As instruções LDS e LES são as únicas a permitir o acesso a ponteiros "completos" (segmento + deslocamento). O deslocamento pode ser carregado tanto nos registradores de base e índice como nos registradores de uso geral de 16 bits, inclusive em registradores que não possam ser usados para acesso indireto à memória.

Infelizmente, não existe uma instrução que execute a operação inversa, isto é, mova um ponteiro "completo" de registradores para a memória; isto deve ser feito através de dois "MOVs".

Exemplos:

```
        LDS     SI,PONTEIRO_ORIGEM
        LDS     AX,ENDERECO
```

LES - Carrega Ponteiro p/ ES

Codificação:

```
        LES     regword,mem
```

Operação:

A instrução **LES** carrega um ponteiro de 32 bits da memória para ES (segmento) e um registrador (deslocamento).

Flags Afetados

Nenhum

Comentários e Exemplos

As instruções **LES** e **LDS** são as únicas a permitir o acesso a ponteiros "completos" (segmento + deslocamento). O deslocamento pode ser carregado tanto nos registradores de base e índice como nos registradores de uso geral de 16 bits, inclusive em registradores que não possam ser usados para acesso indireto à memória.

Infelizmente, não existe uma instrução que execute a operação inversa, isto é, mova um ponteiro "completo" de registradores para a memória; isto deve ser feito através de dois "MOVs".

Exemplos:

```
        LES     SI,PONTEIRO_ORIGEM
        LES      AX,ENDERECO
```

LAHF - Carrega AH com Flags

Codificação:

```
        LAHF
```

Operação:

O byte menos significativo dos Flags é movido para AH.

Flags Afetados

Nenhum

Comentários e Exemplos

Os bits de AH terão os seguintes conteúdos após a execução da instrução **LAHF**:

Bit	Conteúdo
7	flag de sinal
6	flag de zero
5	Indefinido
4	flag de vai um auxiliar
3	Indefinido
2	flag de paridade
1	Indefinido
0	flag de vai um

A finalidade básica desta instrução é permitir salvar, sem o uso da pilha, o conteúdo dos flags equivalentes aos do microprocessador 8080. Ela foi criada pela INTEL para facilitar a conversão de programas assembler 8080 para 8088.

Os flags salvos em AH podem ser restaurados através da instrução **SAHF**.

Exemplo:

```
        LAHF                    ;Salva Flags em AH
        INC     BX              ;Altera Flags
        SAHF                    ;Recupera valor dos Flags antes do INC
```

SAHF - Carrega Flags com AH

Codificação:

```
SAHF
```

Operação:

O conteúdo de AH é movido para o byte menos significativo dos Flags.

Flags Afetados

SF, ZF, AF, PF, CF

Comentários e Exemplos

Os bits de AH ocuparão os seguintes flags após a execução da instrução **SAHF**:

Bit	Flag
7	Sinal
6	Zero
4	Vai um auxiliar
2	Paridade
0	Vai um

Normalmente esta instrução é utilizada para restaurar os flags salvos através da instrução **LAHF**. Ela foi criada pela INTEL para facilitar a conversão de programas assembler 8080 para 8088.

Exemplo:

```
        LAHF                    ;Salva Flags em AH
        INC     BX              ;Altera Flags
        SAHF                    ;Recupera valor dos Flags antes do INC
```

PUSHF - Coloca Flags na Pilha

Codificação:

```
        PUSHF
```

Operação:

O registrador SP é decrementado de 2 e os flags são colocados nas posições SS:[SP] e SS:[SP+1].

Flags Afetados

Nenhum

Comentários e Exemplos

Esta instrução é normalmente utilizada para salvar temporariamente o conteúdo dos flags, que serão restaurados depois através da instrução **POPF**. Um uso menos comum é colocar os flags na pilha para serem restaurados através da instrução **IRET**.

Cabe lembrar que SP aponta sempre para o último dado colocado na pilha e que o segmento utilizado é sempre SS.

Exemplo:

```
PUSHF
```

POPF - Retira Flags da Pilha

Codificação:

```
POPF
```

Operação:

Os flags recebem o conteúdo das posições SS:[SP] SS:[SP+1] e o registrador SP é incrementado de 2.

Flags Afetados

Todos

Comentários e Exemplos

Esta instrução é normalmente utilizada para restaurar o conteúdo dos flags, salvos através da instrução **PUSHF**.

Notar que somente as instruções **POPF** e **IRET** são capazes de alterar o flag de "passo a passo" ("*single step*").

Cabe lembrar que SP aponta sempre para o último dado colocado na pilha e que o segmento utilizado é sempre SS.

Exemplo:

```
POPF
```

3.3 INSTRUÇÕES ARITMÉTICAS

Estas instruções permitem efetuar operações aritméticas envolvendo registradores, variáveis e constantes. Nenhuma delas permite o acesso a registradores de segmento, sendo necessário utilizar registradores ou variáveis, movendo valores de/para os registradores de segmento.

A maioria delas afeta os Flags da maneira esperada, isto é, posicionando-os de acordo com suas definições.

ADD - Adição

Codificação:

```
ADD     reg,reg
ADD     mem,reg
ADD     reg,mem
ADD     reg,imed
ADD     mem,imed
```

Operação:

A instrução **ADD** soma os dois operandos, colocando o resultado no primeiro deles.

Flags Afetados

AF, CF, OF, PF, SF e ZF

Exemplos

```
ADD     CX,DX
ADD     MATRIZ[BX],BP
ADD     SP,VARIAVEL
ADD     SI,50
ADD     VARIAVEL,3
```

ADC - Adição com "vai um"

Codificação:

```
ADC     reg,reg
ADC     mem,reg
ADC     reg,mem
ADC     reg,imed
ADC     mem,imed
```

Operação:

A operação **ADC** soma os dois operandos e mais um se o flag de "vem um" (CF) estiver ativo. O resultado é colocado no primeiro operando.

Flags Afetados

AF, CF, OF, PF, SF e ZF

Comentários e Exemplos

A instrução ADC é normalmente utilizada para "propagar" o "vai um" quando se soma números com mais de 2 bytes, por exemplo:

```
ADD     BX,DX
ADC     AX,CX
```

soma dois números de 32 bits, um em CX (word menos significativo) e DX (word mais significativo) e o outro em BX e AX, colocando o resultado em AXBX. Notar que a primeira soma é do menos significativo e é feita por "**ADD**", pois não temos "vem um" de somas anteriores. Pode-se utilizar "**ADC**", desde que antes o flag de "vai um" seja desligado, por exemplo:

```
CLC
ADC     BX,DX
ADC     AX,CX
```

INC - Incremento

Codificação:

```
INC     reg
INC     mem
```

Operação:

A instrução **INC** soma um ao operando.

Flags Afetados

AF, OF, PF, SF e ZF

Comentários e Exemplos

A instrução **INC** permite incrementar um registrador ou variável de forma mais curta e rápida que "ADD ,1" e sem alterar o flag CF. Exemplos:

```
INC     SP
INC     [BX+DI+1]
```

SUB - Subtração

Codificação:

```
SUB     reg,reg
SUB     mem,reg
SUB     reg,mem
SUB     reg,imed
SUB     mem,imed
```

Operação:

A instrução **SUB** subtrai o segundo operando do primeiro, colocando o resultado no primeiro.

Flags Afetados

AF, CF, OF, PF, SF e ZF

Exemplos

```
SUB     CX,DX
SUB     MATRIZ[BX],BP
SUB     SP,VARIAVEL
SUB     SI,50
SUB     VARIAVEL,3
```

SBB - Subtração com "vem um"

Codificação:

```
SBB     reg,reg
SBB     mem,reg
SBB     reg,mem
```

```
        SBB     reg,imed
        SBB     mem,imed
```

Operação:

A operação **SBB** subtrai o segundo operando do primeiro e mais um se o flag de "vem um" (CF) estiver ativo. O resultado é colocado no primeiro operando.

Flags Afetados

AF, CF, OF, PF, SF e ZF

Comentários e Exemplos

A instrução SBB é normalmente utilizada para "propagar" o "vem um" quando se subtrai números com mais de 2 bytes, por exemplo:

```
        SUB     BX,DX
        SBB     AX,CX
```

subtrai o número de 32 bits em CX (word menos significativo) e DX (word mais significativo) de outro em BX e AX, colocando o resultado em AXBX. Notar que a primeira subtração é do menos significativo e é feita por "**SUB**", pois não temos "vem um" de somas anteriores. Pode-se utilizar "**SBB**", desde que antes o flag de "vai um" seja desligado, por exemplo:

```
        CLC
        SBB     BX,DX
        SBB     AX,CX
```

DEC - Decremento

Codificação:

```
        DEC     reg
        DEC     mem
```

Operação:

A instrução **DEC** subtrai um do operando.

Flags Afetados

AF, OF, PF, SF e ZF

Comentários e Exemplos

A instrução **DEC** permite decrementar um registrador ou variável de forma mais curta e rápida que "SUB ,1" e sem alterar o flag CF. Exemplos:

```
        DEC     SP
        DEC     [BX+DI+1]
```

CMP - Comparação

Codificação:

```
CMP     reg,reg
CMP     mem,reg
CMP     reg,mem
CMP     reg,imed
CMP     mem,imed
```

Operação:

A instrução **CMP** posiciona os flags de forma equivalente a uma subtração sem alterar os operandos.

Flags Afetados

AF, OF, PF, SF e ZF

Exemplos:

```
CMP     CX,DX
CMP     MATRIZ[BX],BP
CMP     SP,VARIAVEL
CMP     SI,50
CMP     VARIAVEL,3
```

MUL - Multiplicação

Codificação:

```
MUL     reg
MUL     mem
```

Operação:

A instrução **MUL** efetua a multiplicação, **sem sinal**, do acumulador pelo operando especificado:

Tipo do Operando	Multiplicando	Produto	
		+ significativo	- significativo
Byte	AL	AH	AL
Word	AX	DX	AX

Flags Afetados

CF e OF - ativados se a parte mais significativa do produto (AH ou DX, conforme o caso) for diferente de zero.

AF, PF SF e ZF - indefinidos

Comentários e Exemplos

A instrução **MUL** fornece sempre um resultado de tamanho (em bits) do dobro dos fatores (que tem o mesmo tamanho), o que garante que o resultado sempre caiba no destino.

Os registradores utilizados para o multiplicando e produto são fixos; notar que não existe a multiplicação por um valor imediato.

No que diz respeito aos flags, apenas CF e OF são posicionados coerentemente com o resultado; os demais (em particular **ZF**) têm seu valor INDEFINIDO após a multiplicação.

Finalmente, a instrução **MUL** é relativamente lenta; no caso de multiplicação por potências de dois é preferível a utilização de deslocamentos (ver instrução SHL).

Exemplos:

```
        MUL     AH              ; AX = AH * AL
        MUL     VAR_WORD        ; DXAX = AX * VAR_WORD
```

IMUL - Multiplicação com sinal

Codificação:

```
        IMUL    reg
        IMUL    mem
```

Operação:

A instrução **IMUL** efetua a multiplicação, **com sinal**, do acumulador pelo operando especificado:

Tipo do Operando	Multiplicando	Produto	
		+ significativo	- significativo
Byte	AL	AH	AL
Word	AX	DX	AX

Flags Afetados

CF e OF - ativados se a parte mais significativa do produto (AH ou DX, conforme o caso) não for uma extensão do sinal da parte menos significativa.

AF, PF SF e ZF - indefinidos

Comentários e Exemplos

A instrução **IMUL** multiplica números com sinal (na notação de complemento de 2), fornecendo sempre um resultado de tamanho (em bits) do dobro dos fatores (que tem o mesmo tamanho), o que garante que o resultado sempre caiba nos destino.

Os registradores utilizados para o multiplicando e produto são fixos; notar que não existe a multiplicação por um valor imediato.

No que diz respeito aos flags, apenas CF e OF são posicionados coerentemente com o resultado; os demais (em particular **ZF**) tem seu valor INDEFINIDO após a multiplicação.

Finalmente, a instrução **IMUL** é relativamente lenta; no caso de multiplicação por potências de dois é preferível a utilização de deslocamentos, veja a instrução SAL.

Exemplos:

```
        IMUL    AH              ; AX = AH * AL
        IMUL    VAR_WORD        ; DXAX = AX * VAR_WORD
```

DIV - Divisão

Codificação:

```
        DIV     reg
        DIV     mem
```

Operação:

A instrução **DIV** efetua a divisão, **sem sinal**, do acumulador pelo operando especificado:

Tipo do Operando	Dividendo		Quociente	Resto
	+ significativo	- significativo		
Byte	AH	AL	AL	AH
Word	DX	AX	AX	DX

Se o quociente não couber no registrador destino, uma interrupção de tipo 0 (**erro em divisão**) é gerada, com os registradores destino contendo valores indefinidos.

Flags Afetados

AF, CF, OF, PF, SF e ZF - indefinidos

Comentários e Exemplos

A instrução **DIV** divide sempre um dividendo de tamanho (em bits) do dobro do divisor e do resto (que tem o mesmo tamanho). Desta forma é possível que o resultado não caiba no registrador destino, mesmo que o divisor não seja zero.

Os registradores utilizados para o dividendo, quociente e resto são fixos; notar que não existe a divisão por um valor imediato. Um erro comum é se esquecer de zerar a parte mais significativa do dividendo quando se deseja a divisão de um byte por outro ou de um word por outro.

Finalmente, a instrução **DIV** é relativamente lenta; no caso de divisão por potências de dois é preferível a utilização de deslocamentos para obter o quociente (ver instrução SHR) e de mascaramento para obter o resto (veja a instrução AND).

Exemplos:

```
        DIV     BL              ; divide AX por BL
                                ; AL=quociente e AH=resto
        DIV     VAR_WORD        ; divide DXAX por VAR_WORD
                                ; AX=quociente, DX=resto
```

IDIV - Divisão com sinal

Codificação:

```
        IDIV    reg
        IDIV    mem
```

Operação:

A instrução **IDIV** efetua a divisão, **com sinal**, do acumulador pelo operando especificado:

Tipo do Operando	Dividendo		Quociente	Resto
	+ significativo	- significativo		
Byte	AH	AL	AL	AH
Word	DX	AX	AX	DX

Se o quociente não couber no registrador destino, uma interrupção de tipo 0 (**erro em divisão**) é gerada, com os registradores destino contendo valores indefinidos.

Flags Afetados

AF, CF, OF, PF, SF e ZF - indefinidos

Comentários e Exemplos

A instrução **IDIV** divide sempre um dividendo de tamanho (em bits) do dobro do divisor e do resto (que tem o mesmo tamanho). Desta forma é possivel que o resultado não caiba no registrador destino, mesmo que o divisor não seja zero. O resto tem sempre o mesmo sinal que o dividendo.

Os registradores utilizados para o dividendo, quociente e resto são fixos; notar que não existe a divisão por um valor imediato. Um erro comum é se esquecer de zerar a parte mais significativa do dividendo quando se deseja a divisão de um byte por outro ou de um word por outro.

Finalmente, a instrução IDIV é relativamente lenta; no caso de divisão por potências de dois é preferível a utilização de deslocamentos para obter o quociente (veja a instrução **SAR**) e de mascaramento para obter o resto (veja a instrução **AND**).

Exemplos:

```
        IDIV    BL              ; divide AX por BL
                                ; AL=quociente e AH=resto
        IDIV    VAR_WORD        ; divide DXAX por VAR_WORD
                                ; AX=quociente, DX=resto
```

NEG - Negação Aritmética

Codificação:

```
        NEG     reg
        NEG     mem
```

Operação:

A instrução **NEG** calcula o complemento de dois do operando, subtraindo o operando de 0FFH (se byte) ou de 0FFFFH (se word) e somando um ao resultado.

Flags Afetados

AF, CF, OF, PF, SF e ZF

Comentários e Exemplos

A instrução **NEG** é utilizada para mudar o sinal do seu operando (na notação complemento de dois). Não deve ser confundida com a instrução **NOT**, que efetua a negação LÓGICA (ou de complemento de um).

Exemplos:

```
        NEG     AX
        NEG     VAR[BX]
```

CBW - Converte Byte em Word

Codificação:

```
        CBW
```

Operação:

A instrução **CBW** efetua a expansão do sinal de AL em AH:

```
Se AL < 80H
   então  AH ← 0
   senão  AH ← 0FFH
```

Flags Afetados

Nenhum

Comentários

A instrução **CBW** converte um byte em word, **MANTENDO o SINAL**. A conversão sem sinal pode ser obtida zerando-se o byte mais significativo. Notar que esta instrução opera SEMPRE sobre AL e AH.

CWD - Converte Word em DoubleWord

Codificação:

```
CWD
```

Operação:

A instrução **CWD** efetua a expansão do sinal de AX em DX:

```
Se AX < 8000H
   então  DX ← 0
   senão  DX ← 0FFFFH
```

Flags Afetados

Nenhum

Comentários

A instrução **CWD** converte um word em doubleword, MANTENDO o SINAL. A conversão sem sinal pode ser obtida zerando-se o word mais significativo. Notar que esta instrução opera SEMPRE sobre AX e DX

AAA - Ajuste para Adição ASCII

Codificação:

```
AAA
```

Operação:

A instrução **AAA** ajusta o resultado (em AL) de uma soma de dois valores decimais (0 a 9) de forma a se obter (em AL) um novo dígito decimal (0 a 9) e passar-se (em AH e CF) o "vai um" correto para o dígito seguinte:

```
Se os 4 bits - signif. de AL > 9 ou AF ativo
   então   Soma 6 em AL
           Soma 1 em AH
           Zera os 4 bits + signif. de AL
           Ativa AF e CF
   senão   Desativa AF e CF
```

Flags Afetados

AF e CF	ativados se "vai um" decimal
OF, PF, SF e ZF	indefinidos

Comentários

O exemplo abaixo mostra a soma de dois números decimais não compactados de dez dígitos:

```
        MOV     SI,OFFSET NUMERO_1 + 9      ;último digito
        MOV     DI,OFFSET NUMERO_2 + 9      ;último digito
        CLC                                 ;vai um inicial = 0
        MOV     CX,10                       ;# de dígitos
SOMA:
        MOV     AL,[SI]
        ADC     AL,[DI]                         ;soma
        AAA                                     ;ajusta soma
        MOV     [DI],AL                         ;guarda dígito
        DEC     SI
        DEC     DI
        LOOP    SOMA                            ;repete p/ todos os dígitos
```

Notar que as instruções **DEC**, **MOV** e **LOOP** não alteram CF, o que simplifica a passagem do "vai um".

É importante destacar que esta instrução atua somente em AL e AH. Os quatro bits menos significativos de AL são zerados, sendo indiferente o seu conteúdo original. Isto permite a soma de números em representação ASCII (na qual 0=30H, 1=31H, ...), embora o resultado não saia com a representação correta.

AAS - Ajuste para Subtração ASCII

Codificação:

```
        AAS
```

Operação:

A instrução **AAS** ajusta o resultado (em AL) de uma subtração de dois valores decimais (0 a 9) de forma a se obter um novo dígito decimal (0 a 9, em AL) e o "vem um" correto (em AH e CF) para o dígito seguinte:

```
Se os 4 bits - signif. de AL > 9 ou AF ativo
   então  Subtrai 6 de AL
          Subtrai 1 de AH
          Zera os 4 bits + signif. de AL
          Ativa AF e CF
   senão  Desativa AF e CF
```

Flags Afetados

AF e CF	ativados se "vem um" decimal
OF, PF, SF e ZF	indefinidos

Comentários

O exemplo abaixo mostra a subtração de dois números decimais não compactados de dez dígitos:

```
        MOV     SI,OFFSET NUMERO_1 + 9      ;último digito
        MOV     DI,OFFSET NUMERO_2 + 9      ;último digito
        CLC                                 ;vai um inicial = 0
```

```
        MOV     CX,10                   ;# de dígitos
SUBT:
        MOV     AL,[SI]
        SBB     AL,[DI]                     ;subtrai
        AAS                                 ;ajusta resultado
        MOV     [DI],AL                     ;guarda dígito
        DEC     SI
        DEC     DI
        LOOP    SUBT                        ;repete p/ todos os dígitos
```

Notar que as instruções DEC, MOV e LOOP não alteram CF, o que simplifica a passagem do "vai um".

É importante destacar que esta instrução atua somente em AL e AH. Os quatro bits menos significativos de AL são zerados, sendo indiferente o seu conteúdo original. Isto permite a soma de números em representação ASCII (na qual 0=30H, 1=31H, ...), embora o resultado não saia com a representação correta.

DAA - Ajuste para Adição de Decimal Compactado

Codificação:

```
        DAA
```

Operação:

A instrução **DAA** ajusta o resultado (em AL) de uma soma de dois valores decimais compactados (00H a 99H) de forma a se obter um novo valor decimal compactado (em AL) e o "vai um" correto (em CF):

```
Se os 4 bits - signif. de AL > 9 ou AF ativo
   então  Soma 6 em AL
          Ativa AF
Se os 4 bits + signif. de AL > 9 ou CF ativo
   então  Soma 60H emAL
          Ativa CF
```

Flags Afetados

AF, CF, PF, SF e ZF

OF - indefinido

Comentários

Esta instrução é semelhante à **AAA**, porém aqui estamos armazenando dois dígitos por byte. O exemplo abaixo mostra a soma de dois números decimais compactados de dez dígitos:

```
        MOV     SI,OFFSET NUMERO_1 + 4      ;último digito
        MOV     DI,OFFSET NUMERO_2 + 4      ;último digito
        CLC                                 ;vai um inicial = 0
        MOV     CX,5                        ;# de bytes
SOMA:
```

```
        MOV     AL,[SI]
        ADC     AL,[DI]                         ;soma
        DAA                                     ;ajusta soma
        MOV     [DI],AL                         ;guarda
        DEC     SI
        DEC     DI
        LOOP    SOMA                            ;repete p/ todos os dígitos
```

Notar que as instruções **DEC**, **MOV** e LOOP não alteram CF, o que simplifica a passagem do "vai um".

É importante destacar que esta instrução atua somente em AL.

DAS - Ajuste para Subtração de Decimal Compactado

Codificação:

```
        DAS
```

Operação:

A instrução **DAS** ajusta o resultado (em AL) da subtração de dois valores decimais compactados (00H a 99H) de forma a se obter um novo valor decimal compactado (em AL) e o "vem um" correto (em CF):

```
Se os 4 bits - signif. de AL > 9 ou AF ativo
   então  Subtrai 6 de AL
          Ativa AF
Se os 4 bits + signif. de AL > 9 ou CF ativo
   então  Subtrai 60H de AL
          Ativa CF
```

Flags Afetados

AF, CF, PF, SF e ZF

OF - indefinido

Comentários

Esta instrução é semelhante à **AAS** porém aqui estamos armazenando dois dígitos por byte. O exemplo abaixo mostra a subtração de dois números decimais compactados de dez dígitos:

```
        MOV     SI,OFFSET NUMERO_1 + 4      ;último digito
        MOV     DI,OFFSET NUMERO_2 + 4      ;último digito
        CLC                                 ;vai um inicial = 0
        MOV     CX,5                        ;# de bytes
SUBT:
        MOV     AL,[SI]
        SBB     AL,[DI]                     ;subtrai
        DAS                                 ;ajusta resultado
        MOV     [DI],AL                     ;guarda
        DEC     SI
        DEC     DI
```

```
        LOOP    SUBT                        ;repete p/ todos os dígitos
```

Notar que as instruções **DEC**, **MOV** e **LOOP** não alteram CF, o que simplifica a passagem do "vai um".

É importante destacar que esta instrução atua somente em AL.

AAM - Ajuste para Multiplicação ASCII

Codificação:

```
        AAM
```

Operação:

A instrução **AAM** ajusta o resultado (em AL) da multiplicação de dois dígitos decimais não compactados (0 a 9) de forma a se obter (em AH e AL) os DOIS dígitos decimais correspondentes ao produto:

```
AH - quociente da divisão de AL por 10
AL - resto da divisão de AL por 10
```

Flags Afetados

PF, SF e ZF

AF, CF e OF - indefinidos

Comentários

O exemplo abaixo mostra a multiplicação de um número decimal de dez dígitos por um número de 0 a 9:

```
FATOR1  DB      10 DUP(?)
FATOR2  DB      ?
PRODUT  DB      11 DUP(?)

        MOV     SI,10           ;índice p/ FATOR2 e PRODUTO
        MOV     DL,0            ;vai um do produto anterior
MULT:
        DEC     SI
        MOV     AL,FATOR1[SI]
        MUL     FATOR2
        AAM                     ;AHAL = produto
        ADD     AL,DL           ;soma vai um anterior
        AAA
        MOV     DL,AH           ;novo vai um
        MOV     PRODUTO[SI+1],AL ;dígito do produto
        OR      SI,SI
        JNZ     MULT            ;repete p/ os 10
        MOV     PRODUTO[SI],DL  ;último dígito
```

Esta rotina funciona como a multiplicação feita à mão: multiplicam-se dois dígitos e soma-se o "vai um" do produto anterior; o algarismo das unidades é colocado no produto e o das dezenas é o novo "vai um".

Como as demais operações de ajuste, **AAM** opera sempre sobre AL e AH. Porém, ao contrário de **AAA** e **AAS**, os bits mais significativos dos fatores TÊM que conter zero para a obtenção do resultado correto. O 8088 NÃO possui uma instrução voltada para a multiplicação de números decimais compactados.

Finalmente, cabe notar que AAM é ligeiramente mais rápida que DIV, podendo ser usada como alternativa para a divisão por 10.

AAD - Ajuste para Divisão ASCII

Codificação:

```
AAD
```

Operação:

A instrução **AAD** prepara os conteúdos de AH e AL, normalmente dois dígitos decimais não compactados, de forma a se obter em AX um número que, dividido por um terceiro número decimal, forneça quociente e resto corretos. Isto é feito somando-se a AL o valor contido em AH multiplicado por 10 e zerando-se AH em seguida.

Flags Afetados

PF, SF e ZF

AF, CF e OF - indefinidos

Comentários

Ao contrário das demais instruções de ajuste, **AAD** deve ser executada ANTES da operação aritmética. O 8088 não possui uma instrução para divisão de números decimais compactados.

Como a instrução AAD é ligeiramente mais rápida que **MUL**, ela pode ser usada como opção para a multiplicação de AH por 10.

3.4 INSTRUÇÕES LÓGICAS

NOT - Negação Lógica

Codificação:

```
        NOT     reg
        NOT     mem
```

Operação:

A instrução **NOT** calcula o complemento de 1 do operando, subtraindo-o de 0FFH (se byte) ou 0FFFFH (se word), sem afetar os flags.

Flags Afetados

Nenhum

Comentários

A instrução **NOT** é utilizada para efetuar a negação lógica do operando. Não deve ser confundida com a instrução **NEG**, que efetua a negação aritmética (complemento de 2).

Exemplos:

```
        NOT     SI
        NOT     VAR[BX]
```

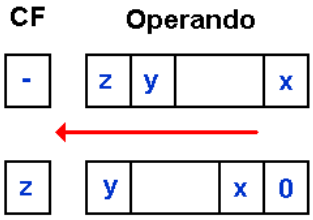
SHL - Deslocamento Lógico para a Esquerda

Codificação:

```
        SHL     reg,1
        SHL     reg,CL
        SHL     mem,1
        SHL     mem,CL
```

Operação:

A instrução **SHL** desloca o primeiro operando para a esquerda, sendo o número de posições especificado pelo segundo operando. A cada deslocamento o bit mais significativo é colocado no flag de "vai um" e um zero é colocado no bit menos significativo:



Flags Afetados

CF último bit deslocado para fora

PF, SF e ZF	correspondem ao resultado final
AF	indefinido
OF	se o segundo operando for 1, OF será ativado se o bit mais significativo do resultado for diferente do "vai um" final (isto é, se o sinal foi deslocado para fora do número). Se o segundo operando não for 1, OF fica indefinido.

Comentários e Exemplos

Como as demais instruções de deslocamento e rotação, a instrução **SHL** aceita um segundo operando constante (1) ou variável (CL); outras combinações (constantes diferentes de 1 ou outros registradores que não CL) não são permitidas.

A instrução **SHL** é idêntica à instrução **SAL**; os dois mnemônicos permitem uma maior clareza no programa.

Um uso bastante frequente de SHL é para efetuar a multiplicação por potências de dois:

```
        SHL     SI,1            ;multiplica por 2
        MOV     CL,4
        SHL     VARIAVEL,CL     ;multiplica por 16
```

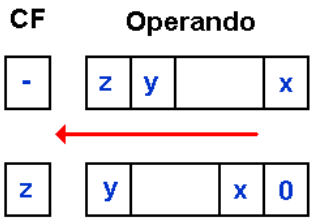
SAL - Deslocamento Aritmético para a Esquerda

Codificação:

```
        SAL     reg,1
        SAL     reg,CL
        SAL     mem,1
        SAL     mem,CL
```

Operação:

A instrução **SAL** desloca o primeiro operando para a esquerda, sendo o número de posições especificado pelo segundo operando. A cada deslocamento o bit mais significativo é colocado no flag de "vai um" e um zero é colocado no bit menos significativo:



Flags Afetados

CF	último bit deslocado para fora
PF, SF e ZF	correspondem ao resultado final
AF	indefinido

OF	se o segundo operando for 1, OF será ativado se o bit mais significativo do resultado for diferente do "vai um" final (isto é, se o sinal foi deslocado para fora do número). Se o segundo operando não for 1, OF fica indefinido.

Comentários e Exemplos

Como as demais instruções de deslocamento e rotação, a instrução SAL aceita um segundo operando constante (1) ou variável (CL); outras combinações (constantes diferentes de 1 ou outros registradores que não CL) não são permitidas.

A instrução **SAL** é idêntica à instrução **SHL**; os dois mnemônicos permitem uma maior clareza no programa.

Um uso bastante frequente de **SHL** é para efetuar a multiplicação por potências de dois:

```
        SAL     SI,1            ;multiplica por 2
        MOV     CL,4
        SAL     VARIAVEL,CL     ;multiplica por 16
```

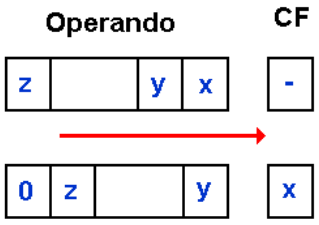
SHR - Deslocamento Lógico para a Direita

Codificação:

```
        SHR     reg,1
        SHR     reg,CL
        SHR     mem,1
        SHR     mem,CL
```

Operação:

A instrução SHR desloca o primeiro operando para a direita, sendo o número de posições especificado pelo segundo operando. A cada deslocamento o bit menos significativo é colocado no flag de "vai um" e um zero é colocado no bit mais significativo:



Flags Afetados

CF	último bit deslocado para fora
PF, SF e ZF	correspondem ao resultado final
AF	indefinido
OF	se o segundo operando for 1, OF será ativado se o bit mais significativo do resultado for diferente do original (isto é, se o sinal foi deslocado para fora do número). Se o segundo operando não for 1, OF fica indefinido.

Comentários e Exemplos

Como as demais instruções de deslocamento e rotação, a instrução **SHR** aceita um segundo operando constante (somente 1) ou variável (somente CL).

É importante notar que a instrução **SHR** é DIFERENTE de **SAR**; a primeira preenche com zero os bits mais significativos e a segunda reproduz o bit mais significativo original.

Um uso bastante frequente de **SHR** é para efetuar a divisão de números sem sinal por potências de dois:

```
        SHR     SI,1            ;divide por 2
        MOV     CL,4
        SHR     VARIAVEL,CL     ;divide por 16
```

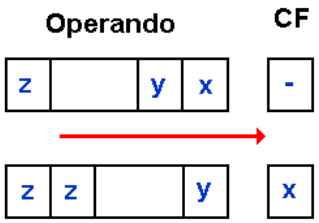
SAR - Deslocamento Aritmético para a Direita

Codificação:

```
        SAR     reg,1
        SAR     reg,CL
        SAR     mem,1
        SAR     mem,CL
```

Operação:

A instrução **SAR** desloca o primeiro operando para a direita, sendo o número de posições especificado pelo segundo operando. A cada deslocamento o bit menos significativo é colocado no flag de "vai um" e o bit mais significativo não é alterado:



Flags Afetados

CF	último bit deslocado para fora
PF, SF e ZF	correspondem ao resultado final
AF	indefinido
OF	desativado se o segundo operando for 1, indefinido no caso contrario.

Comentários e Exemplos

Como as demais instruções de deslocamento e rotação, a instrução **SAR** aceita um segundo operando constante (somente 1) ou variável (somente CL).

É importante notar que a instrução **SAR** é DIFERENTE de **SHR**; a primeira reproduz o bit mais significativo original e a segunda preenche com zero os bits mais significativos.

Um uso bastante frequente de SAR é para efetuar a divisão de números com sinal por potências de dois:

```
        SAR     SI,1            ;divide por 2
        MOV     CL,4
        SAR     VARIAVEL,CL     ;divide por 16
```

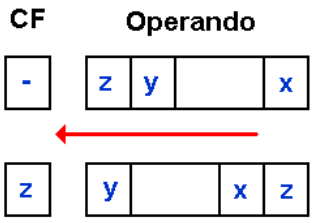
ROL - Rotação para a Esquerda

Codificação:

```
        ROL     reg,1
        ROL     reg,CL
        ROL     mem,1
        ROL     mem,CL
```

Operação:

A instrução **ROL** roda o primeiro operando para a esquerda, sendo o número de posições especificado pelo segundo operando. A cada deslocamento o bit mais significativo é colocado no flag de "vai um" e no bit menos significativo:



Flags Afetados

CF	último bit rodado do bit mais significativo para o menos.
PF, SF e ZF	correspondem ao resultado final
AF	indefinido
OF	se o segundo operando for 1, OF será ativado se o bit mais significativo do resultado for diferente do original. Se o segundo operando não for 1, OF fica indefinido.

Comentários e Exemplos

Como as demais instruções de deslocamento e rotação, a instrução **ROL** aceita um segundo operando constante (somente 1) ou variável (somente CL).

Não confundir esta instrução com **RCL**, na qual o flag de "vai um" é também rodado.

Exemplos:

```
        ROL     AX,1
        ROL     VARIAVEL,CL
```

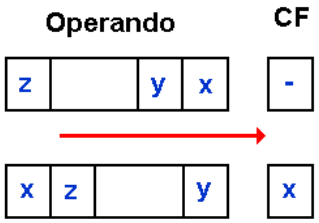
ROR - Rotação para a Direita

Codificação:

```
        ROR     reg,1
        ROR     reg,CL
        ROR     mem,1
        ROR     mem,CL
```

Operação:

A instrução **ROR** roda o primeiro operando para a direita, sendo o número de posições especificado pelo segundo operando. A cada deslocamento o bit menos significativo é colocado no flag de "vai um" e no bit mais significativo:



Flags Afetados

CF	último bit rodado do bit menos significativo para o mais.
PF, SF e ZF	correspondem ao resultado final
AF	indefinido
OF	se o segundo operando for 1, OF será ativado se o bit mais significativo do resultado for diferente do original. Se o segundo operando não for 1, OF fica indefinido.

Comentários e Exemplos

Como as demais instruções de deslocamento e rotação, a instrução **ROR** aceita um segundo operando constante (somente 1) ou variável (somente CL).

Não confundir esta instrução com **RCR**, na qual o flag de "vai um" é também rodado.

Exemplos:

```
        ROR     AX,1
        ROR     VARIAVEL,CL
```

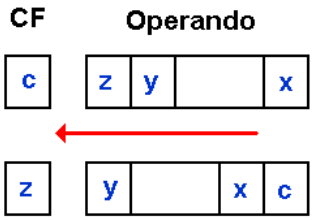
RCL - Rotação através do "Carry" para a Esquerda

Codificação:

```
        RCL     reg,1
        RCL     reg,CL
        RCL     mem,1
        RCL     mem,CL
```

Operação:

A instrução **RCL** roda o primeiro operando para a esquerda, sendo o número de posições especificado pelo segundo operando. A cada deslocamento o bit mais significativo é colocado no flag de "vai um" e o valor anterior do flag de "vai um" no bit menos significativo:



Flags Afetados

CF	último bit rodado do bit mais significativo
PF, SF e ZF	correspondem ao resultado final
AF	indefinido
OF	se o segundo operando for 1, OF será ativado se o bit mais significativo do resultado for diferente do original. Se o segundo operando não for 1, OF fica indefinido.

Comentários e Exemplos

Como as demais instruções de deslocamento e rotação, a instrução **RCL** aceita um segundo operando constante (somente 1) ou variável (somente CL).

Não confundir esta instrução com **ROL**, na qual o flag de "vai um" não é rodado.

Exemplos:

```
        RCL     AX,1
        RCL     VARIAVEL,CL
```

RCR - Rotação através do "Carry" para a Direita

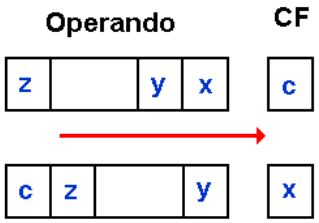
Codificação:

```
        RCR     reg,1
        RCR     reg,CL
        RCR     mem,1
```

```
        RCR     mem,CL
```

Operação:

A instrução **RCR** roda o primeiro operando para a direita sendo o número de posições especificado pelo segundo operando. A cada deslocamento o bit menos significativo é colocado no flag de "vai um" e no bit mais significativo:



Flags Afetados

CF	último bit rodado do bit menos significativo
PF, SF e ZF	correspondem ao resultado final
AF	indefinido
OF	se o segundo operando for 1, OF será ativado se o bit mais significativo do resultado for diferente do original. Se o segundo operando não for 1, OF fica indefinido.

Comentários e Exemplos

Como as demais instruções de deslocamento e rotação, a instrução **RCR** aceita um segundo operando constante (somente 1) ou variável (somente CL).

Não confundir esta instrução com ROR, na qual o flag de "vai um" não é rodado.

Exemplos:

```
        RCR     AX,1
        RCR     VARIAVEL,CL
```

AND - Operação Lógica "E"

Codificação:

```
        AND     reg,reg
        AND     mem,reg
        AND     reg,mem
        AND     reg,imed
        AND     mem,imed
```

Operação:

A instrução **AND** efetua o "E" lógico dos dois operandos (bit a bit), colocando o resultado no primeiro deles.

Flags Afetados

PF, SF e ZF	correspondem ao resultado
CF e OF	desativados
AF	indefinido

Comentários e Exemplos

A instrução **AND** é normalmente utilizada para isolar (ou zerar) determinados bits de um número, visto que um bit sera "1" no resultado somente se for "1" em ambos os operandos. Por exemplo:

```
        AND     BP,1000000000000001B
```

"isola" os bits mais e menos signficativos de BP, zerando os demais.

Um outro uso é para obter o resto da divisão de um número por uma potência de dois. Para obter o resto da divisão de um número por 2^N fazemos um **AND** com 2^N-1:

```
        AND     VALOR,7
```

coloca em VALOR o resto da divisão de VALOR por 8.

Outros exemplos:

```
        AND     CX,DX
        AND     MATRIX[BX],SP
        AND     SI,VARIAVEL
```

TEST - Operação Lógica "E" não destrutiva

Codificação:

```
        TEST    reg,reg
        TEST    mem,reg
        TEST    reg,mem
        TEST    reg,imed
        TEST    mem,imed
```

Operação:

A instrução **TEST** efetua o "E" lógico dos dois operandos (bit a bit) e posiciona os flags de acordo com o resultado, sem alterar os operandos.

Flags Afetados

PF, SF e ZF	correspondem ao resultado
CF e OF	desativados
AF	indefinido

Comentários e Exemplos

A instrução **TEST** é normalmente utilizada para testar um ou mais bits de um valor. Não se deve confundir **TEST** com **CMP**; na **TEST** verifica-se se todos os bits que são "1" no segundo operando são "0" no primeiro; na **CMP** verifica-se se todos os bits dos dois operandos são iguais:

```
        TEST    AX,3
```

deixará ativo o flag Z se os dois bits menos significativos de AX forem "0", independente dos demais bits;

```
        CMP     AX,3
```

deixará ativo o flag Z se os dois bits menos significativos de AX forem "1" e os demais "0".

Outros exemplos:

```
        TEST    CX,DX
        TEST    MATRIX[BX],SP
        TEST    SI,VARIAVEL
```

OR - Operação Lógica "OU"

Codificação:

```
        OR      reg,reg
        OR      mem,reg
        OR      reg,mem
        OR      reg,imed
        OR      mem,imed
```

Operação:

A instrução **OR** efetua o "OU" lógico dos dois operandos (bit a bit), colocando o resultado no primeiro deles.

Flags Afetados

PF, SF e ZF	correspondem ao resultado
CF e OF	desativados
AF	indefinido

Comentários e Exemplos

A instrução **OR** é normalmente utilizada para ativar determinados bits de um número, visto que um bit será "1" no resultado se for "1" em pelo menos um dos operandos. Por exemplo:

```
        OR      BP,1000000000000001B
```

ativa os bits mais e menos signficativos de BP, sem afetar os demais.

Uma vez que o "OU" de um número com ele mesmo não o altera, é comum fazer OR de um registrador com ele mesmo para posicionar os flags PF, SF e ZF de acordo com o seu conteúdo (o mesmo resultado pode ser obtido através da operação AND, o que é menos comum).

Outros exemplos:

```
        OR      CX,DX
        OR      MATRIX[BX],SP
        OR      SI,VARIAVEL
```

XOR - Operação Lógica "OU Exclusivo"

Codificação:

```
        XOR     reg,reg
        XOR     mem,reg
        XOR     reg,mem
        XOR     reg,imed
        XOR     mem,imed
```

Operação:

A instrução **XOR** efetua o "OU Exclusivo" dos dois operandos (bit a bit), colocando o resultado no primeiro deles.

Flags Afetados

PF, SF e ZF	correspondem ao resultado
CF e OF	desativados
AF	indefinido

Comentários e Exemplos

A instrução XOR é normalmente utilizada para inverter determinados bits de um número, visto que um bit será "1" no resultado somente se for "1" em apenas um dos operandos. Por exemplo:

```
        XOR     BP,1000000000000001B
```

inverte os bits mais e menos significativos de BP, sem afetar os demais.

Uma vez que o "OU exlusivo" de um número com ele mesmo resulta em zero, é comum fazer XOR de um registrador com ele mesmo para zerá-lo (o mesmo resultado pode ser obtido através da operação SUB, o que é menos comum).

Outros exemplos:

```
        XOR     CX,DX
        XOR     MATRIX[BX],SP
        XOR     SI,VARIAVEL
```

3.5 INSTRUÇÕES DE ENTRADA E SAIDA

IN - Leitura de Dispositivo

Codificação:

```
IN      AL,imed
IN      AL,DX
IN      AX,imed
IN      AX,DX
```

Operação:

A instrução **IN** lê um byte ou word de um dispositivo, colocando-o no acumulador (AL ou AX).

Flags Afetados

Nenhum

Comentários e Exemplos

Em primeiro lugar, deve-se notar que a instrução **IN** é capaz de ler um byte ou um word, o que é determinado pelo primeiro operando (AL ou AX). O uso de outros registradores não é permitido.

Em segundo lugar, o endereço do dispositivo ("port") pode ser constante (0 a 255) ou variável (em DX, 0 a 65535).

Exemplos:

```
IN      AL,30H
IN      AL,DX
IN      AX,30H
IN      AX,DX
```

OUT - Escrita em Dispositivo

Codificação:

```
OUT     imed,AL
OUT     DX,AL
OUT     imed,AX
OUT     DX,AX
```

Operação:

A instrução **OUT** escreve um byte (contido em AL) ou word (contido em AX) em um dispositivo.

Flags Afetados

Nenhum

Comentários e Exemplos

Em primeiro lugar, deve-se notar que a instrução **OUT** é capaz de escrever um byte ou um word, o que é determinado pelo segundo operando (AL ou AX). O uso de outros registradores não é permitido.

Em segundo lugar, o endereço do dispositivo ("port") pode ser constante (0 a 255) ou variável (em DX, 0 a 65535).

Exemplos:

```
        OUT     AL,30H
        OUT     AL,DX
        OUT     AX,30H
        OUT     AX,DX
```

3.6 INSTRUÇÕES DE MANIPULAÇÃO DE CADEIAS

As instruções de manipulação de cadeias (*strings*) podem ser divididas em dois grupos:

- **PRIMITIVAS**, que operam sobre uma posição da cadeia e avançam (ou recuam) o ponteiro para a posição seguinte (ou anterior).
- **PREFIXOS de REPETIÇÃO**, que permitem repetir uma primitiva enquanto o flag Z mantém um certo estado e CX é diferente de zero.

A seleção de avanço ou recuo de ponteiros pelas primitivas é feito através do flag **DF** (ver instruções **CLD** e **STD**). O erro mais comum no uso das instruções de manipulação de cadeias é o posicionamento errado deste flag quando da execução da instrução.

Os prefixos de repetição são semelhantes aos prefixos de alteração de segmento ("*segment override*"). Cuidado deve ser tomado quando do uso conjunto dos dois prefixos; em caso da ocorrência de uma interrupção um dos dois prefixos será "esquecido" pelo processador, causando funcionamento incorreto da instrução.

REP - Repete Primitiva enquanto CX <> 0

Codificação:

```
        REP     primitiva
```

Operação:

Enquanto CX for diferente de zero a primitiva é executada e CX é decrementado.

Flags Afetados

Os flags não são afetados pelo prefixo **REP**, mas sim pela primitiva sendo repetida.

Comentários

Se CX for igual a zero a primitiva não será executada.

O prefixo **REP** é utilizado em conjunto com as primitivas **MOVS**, **STOS** e **LODS**; com as primitivas **SCAS** e **CMPS** utilizam-se os prefixos **REPZ**, **REPNZ**, **REPE** e **REPNE**.

REPZ / REPE - Repete Primitiva enquanto Igual

Codificação:

```
        REPZ    primitiva
        REPE    primitiva
```

Operação:

Enquanto CX for diferente de zero CX é decrementado e a primitiva é executada. Caso o flag Z fique inativo (Z=0) após a execução da primitiva, a repetição é interrompida.

Flags Afetados

Os flags não são afetados pelos prefixos **REPE** e **REPZ**, mas sim pela primitiva sendo repetida.

Comentários e Exemplos

Se CX for igual a zero a primitiva não será executada; caso contrário ela será executada pelo menos uma vez.

Os mnemônicos **REPZ** e **REPE** correspondem ao mesmo prefixo, sendo utilizados com as primitivas **SCAS** e **CMPS**.

Uma vez que CX é decrementado **antes** do teste do flag Z e as primitivas **SCAS** e **CMPS** avançam (ou recuam) os ponteiros **após** a execução da comparação, ao final da repetição teremos:

- Se Z = 1: não foi encontrado elemento na cadeia que resultasse em Z=0; CX contém zero e os ponteiros apontam para a posição seguinte ao final das cadeias.
- Se Z = 0: foi encontrado elemento na cadeia que resultou em Z=0; CX e os ponteiros correspondem à posição seguinte a este elemento.

Exemplo:

```
        MOV     AL,0
        REPE    SCASB
        JZ      TUDO_ZERO         ;Z=1 se todos elementos contém 0
```

REPNZ / REPNE - Repete Primitiva enquanto Diferente

Codificação:

```
        REPNZ   primitiva
        REPNE   primitiva
```

Operação:

Enquanto CX for diferente de zero CX é decrementado e a primitiva é executada. Caso o flag Z fique ativo (Z=1) após a execução da primitiva, a repetição é interrompida.

Flags Afetados

Os flags não são afetados pelos prefixos **REPNE** e **REPNZ**, mas sim pela primitiva sendo repetida.

Comentários e Exemplos

Se CX for igual a zero a primitiva não será executada; caso contrário ela será executada pelo menos uma vez.

Os mnemônicos **REPNZ** e **REPNE** correspondem ao mesmo prefixo, sendo utilizados com as primitivas **SCAS** e **CMPS**.

Uma vez que CX é decrementado **antes** do teste do flag Z e as primitivas **SCAS** e **CMPS** avançam (ou recuam) os ponteiros **após** a execução da comparação, ao final da repetição teremos:

- Se Z = 0: não foi encontrado elemento na cadeia que resultasse em Z=1; CX contem zero e os ponteiros apontam para a posição seguinte ao final das cadeias.
- Se Z = 1: foi encontrado elemento na cadeia que resultou em Z=1; CX e os ponteiros correspondem à posição seguinte a este elemento.

Exemplo:

```
        MOV     AL,0
        REPNE   SCASB
        JZ      ACHOU_ZERO        ;Z=1 se achou elemento = 0
```

LODS / LODSB / LODSW - Carrega Elemento de Cadeia

Codificação:

```
        LODS    mem
        LODSB
        LODSW
```

Operação:

Se operando for BYTE

carrega em AL o byte apontado por SI

se DF = 0 soma 1 a SI

se DF = 1 subtrai 1 de SI

Se operando for WORD

carrega em AX o word apontado por SI

se DF = 0 soma 2 a SI

se DF = 1 subtrai 2 de SI

Flags Afetados

Nenhum

Comentários e Exemplos

O operando da instrução é utilizado pelo Assembler para determinar o tipo (byte ou word) e verificar se ele é acessível com os conteúdos indicados para os registradores de segmentos (via **ASSUME**). As formas **LODSB** e **LODSW** permitem especificar explicitamente o tipo do operando, que é considerado acessível.

Notar que sempre são utilizados os registradores AL (ou AX) e SI. O registrador de segmento utilizado é DS, sendo permitida a sua alteração via "*segment override*".

STOS / STOSB / STOSW - Coloca Elemento em Cadeia

Codificação:

```
STOS  mem
STOSB
STOSW
```

Operação:

Se operando for BYTE

coloca o conteúdo de AL no byte apontado por ES:DI

se DF = 0 soma 1 a DI

se DF = 1 subtrai 1 de DI

Se operando for WORD

coloca o conteudo de AX No word apontado por ES:DI

se DF = 0 soma 2 a DI

se DF = 1 subtrai 2 de DI

Flags Afetados

Nenhum

Comentários e Exemplos

O operando da instrução é utilizado pelo Assembler para determinar o tipo (byte ou word) e verificar se ele é acessível com os conteúdos indicados para ES (via **ASSUME**). As formas **STOSB** e **STOSW** permitem especificar explicitamente o tipo do operando, que é considerado acessível.

Notar que sempre são utilizados os registradores AL (ou AX) ,SI e ES. Não é permitido o uso de "*segment override*".

Por exemplo, a rotina abaixo causa o preenchimento de 80 words, a partir do endereço 50H:50H com o valor 0050H (isto é, primeiro byte com 50H e segundo com 0).

```
        CLD                     ;seleciona incremento
        MOV     AX,50H          ;valor a colocar na cadeia
        MOV     ES,AX           ;segmento da cadeia
        MOV     DI,AX           ;deslocamento inicial
        MOV     CX,AX           ;número de elementos (50H = 80D)
        REP     STOSW           ;preenche
```

MOVS / MOVSB / MOVSW - Copia Cadeia

Codificação:

```
        MOVS    mem,mem
        MOVSB
        MOVSW
```

Operação:

Se operando for BYTE

copia o byte apontado por SI para ES:DI

se DF = 0 soma 1 a DI

se DF = 1 subtrai 1 de DI

Se operando for WORD

copia o word apontado por SI para ES:DI

se DF = 0 soma 2 a DI

se DF = 1 subtrai 2 de DI

Flags Afetados

Nenhum

Comentários e Exemplos

O operando da instrução é utilizado pelo Assembler para determinar o tipo (byte ou word) e verificar se ele é acessível com os conteúdos indicados para os registradores de segmentos (via **ASSUME**). As formas **MOVSB** e **MOVSW** permitem especificar explicitamente o tipo do operando, que é considerado acessível.

Notar que sempre são utilizados os registradores SI, DI e ES. O registrador de segmento utilizado para a origem é DS, sendo permitida a sua alteração via "*segment override*"; para o destino, é sempre utilizado ES.

Por exemplo, podemos mover 20 bytes de AREA1 para AREA2 da seguinte forma (supondo DS e ES corretos):

```
        CLD                     ;seleciona incremento
        MOV     SI,OFFSET AREA1 ;deslocamento origem
        MOV     DI,OFFSET AREA2 ;deslocamento destino
        MOV     CX,20           ;número de bytes
        REP     MOVSB           ;copia
```

SCAS / SCASB / SCASW - Percorre Cadeia

Codificação:

```
SCAS    mem
SCASB
SCASW
```

Operação:

Se operando for BYTE

posiciona os flags para resultado da subtração AL - (byte apontado por ES:DI)

se DF = 0 soma 1 a DI

se DF = 1 subtrai 1 de DI

Se operando for WORD

posiciona os flags para resultado da subtração AX - (word apontado por ES:DI)

se DF = 0 soma 2 a DI

se DF = 1 subtrai 2 de DI

Flags Afetados

AF, CF, OF, PF, SF e ZF

Comentários e Exemplos

O operando da instrução é utilizado pelo Assembler para determinar o tipo (byte ou word) e verificar se ele é acessível com os conteúdos indicados para ES (via **ASSUME**). As formas **SCASB** e **SCASW** permitem especificar explicitamente o tipo do operando.

Notar que sempre são utilizados os registradores AL (ou AX), DI e ES. Não é permitido o "*segment override*".

Lembrar que ao final o flag Z (e não CX) é quem indica se foi encontrada a condição desejada (que é o contrário da codificada no **REP**); e se sim ES:DI aponta para o elemento **SEGUINTE**:

```
CLD
REPE    SCASB
```

avança DI enquanto ES:DI apontar para um byte com conteúdo igual ao de AL e CX for diferente de zero;

```
CLD
REPNE   SCASB
```

avança DI enquanto ES:DI apontar para um byte com conteúdo diferente ao de AL e CX for diferente de zero.

CMPS / CMPSB / CMPSW - Compara Cadeias

Codificação:

```
        CMPS    mem,mem
        CMPSB
        CMPSW
```

Operação:

Se operando for BYTE

posiciona os flags para resultado da subtração (byte apontado por SI) - (byte apontado por ES:DI)

se DF = 0 soma 1 a DI

se DF = 1 subtrai 1 de DI

Se operando for WORD

posiciona os flags para resultado da subtração (word apontado por SI) - (word apontado por ES:DI)

se DF = 0 soma 2 a DI

se DF = 1 subtrai 2 de DI

Flags Afetados

AF, CF, OF, PF, SF e ZF

Comentários e Exemplos

O operando da instrução é utilizado pelo Assembler para determinar o tipo (byte ou word) e verificar se ele é acessível. As formas **CMPSB** e **CMPSW** permitem especificar explicitamente o tipo do operando.

Notar que sempre são utilizados os registradores SI, DI e ES. É permitido o "segment override" para alterar o segmento da cadeia apontada por SI.

Lembrar que ao final o flag Z (e não CX) é quem indica se foi encontrada a condição desejada (que é o contrário da codificada no **REP**); e se sim ES:DI e SI apontam para os elementos SEGUINTES:

```
        CLD
REPE    CMPSB
```

avança SI e DI enquanto DS:SI ES:DI apontarem para bytes com conteúdos iguais e CX for diferente de zero;

```
        CLD
REPNE   CMPSB
```

avança SI e DI enquanto DS:SI e ES:DI apontarem para bytes com conteúdos diferentes e CX não for zero.

3.7 INSTRUÇÕES DE DESVIO

As instruções de desvio podem ser agrupadas da seguinte forma:

A) **Desvio** incondicional (JMP) ou condicional (JE/JZ, etc.),

B) **Chamada a sub-rotina** (CALL), sempre incondicional,

C) **Chamada a interrupção de software**, incondicional (INT) ou condicional (INTO) e

D) **Retorno** de sub-rotina (RET) ou de interrupção (IRET)

No grupo "C" o endereço destino é obtido do vetor de interrupção; no grupo "D" da pilha. Já nos grupos "A" e "B", podemos ter:

1) Desvios **diretos**, em que o endereço destino está na própria instrução, ocupando

   a) Um byte - desvio **CURTO** ("*short*") - armazenando a distância (-128 a +127) entre o byte seguinte ao desvio e o endereço destino, ou

   b) Dois bytes - desvio **INTRASEGMENTO** ("*near*") - armazenando a distância entre o byte seguinte ao desvio e o endereço destino, ou ainda

   c) Quatro bytes - desvio **INTERSEGMENTO** ("*far*") - armazenando o deslocamento e o segmento do endereço destino.

2) Desvios **indiretos**, em que o endereço destino é obtido de

   a) Um registrador - desvio intrasegmento, ou

   b) Uma variável do tipo word - também desvio intrasegmento, ou

   c) Uma variável do tipo dword - desvio intersegmento.

O desvio incondicional permite todas estas combinações; o desvio condicional somente o desvio curto ("short", sempre direto) e o desvio para sub-rotina todas exceto o desvio curto. É importante notar a falta de instruções condicionais de chamada e retorno.

JMP - desvio incondicional

Codificação:

```
        JMP     rótulo
        JMP     reg
        JMP     mem
```

Operação:

Antes do desvio propriamente dito, IP é atualizado para a instrução seguinte (daí as distâncias mencionadas abaixo serem em relação ao byte seguinte ao **JMP**). O desvio se processa da seguinte forma:

1 - desvio direto

a) curto - a distância é somada a IP

b) intrasegmento - a distância é somada a IP

c) intersegmento - CS e IP recebem os valores contidos na instrução

2 - desvio indireto

a) intrasegmento - IP recebe o valor contido no operando

b) intersegmento - CS e IP recebem os valores contidos no operando

Flags Afetados

Nenhum

Comentários e Exemplos

O tipo de desvio é determinado pelo tipo do operando:

Operando	desvio
Registrador	indireto intrasegmento
variável Word	indireto intrasegmento
variável dword	indireto intersegmento
rótulo "near"	direto intrasegmento/curto *
rótulo "far"	direto intersegmento

A seleção entre desvio direto intrasegmento e desvio direto curto (*) é controlada pelo programador (através do atributo SHORT) e pelo Assembler: se o desvio é feito para um rótulo anterior à instrução de desvio, o Assembler determina automaticamente se é possível utilizar um desvio curto; se o desvio é feito para um rótulo à frente, o Assembler reserva espaço para um desvio intrasegmento (3 bytes). Se posteriormente verificar que era possível o uso de desvio curto, a posição extra é preenchida com uma instrução "NOP".

Vejamos alguns exemplos:

```
END_NEAR DW      ?                ;variável tipo WORD
END_FAR  DD      ?                ;variável tipo DWORD

ROT_NEAR LABEL   NEAR             ;rótulo NEAR
ROT_FAR  LABEL   FAR              ;rótulo FAR

XIS:     JMP     ROT_NEAR         ;direto, intrasegmento
         JMP     ROT_FAR          ;direto, intersegmento

         JMP     END_NEAR         ;indireto, intrasegmento
         JMP     END_FAR          ;indireto, intersegmento

         JMP     XIS              ;curto, pois XIS está a menos de 128 bytes

         MOV     AX,OFFSET ROT_NEAR
         JMP     AX               ;indireto, intrasegmento, desvia p/ ROT_NEAR

         MOV     END_NEAR,OFFSET ROT_NEAR
```

```
        MOV     BX,OFFSET END_NEAR
        JMP     [BX]            ;"duplamente" indireto, intrasegmento,
                                ;desvia p/ ROT_NEAR
```

JE/JZ - Desvia se Zero

Codificação:

```
        JE      rótulo
        JZ      rótulo
```

Operação:

Se ZF = 1, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários

O mnemônico **JE** corresponde a "desvio se igual" e é normalmente utilizado após uma instrução **CMP**; **JZ** significa "desvio se zero" e é normalmente utilizado após uma instrução aritmética ou lógica. Os dois mnemônicos correspondem à mesma instrução.

JNE/JNZ - Desvia se não Zero

Codificação:

```
        JNE     rótulo
        JNZ     rótulo
```

Operação:

Se ZF = 0, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários

O mnemônico **JNE** corresponde a "desvio se diferente" e é normalmente utilizado após uma instrução CMP; **JNZ** significa "desvio se não zero" e é normalmente utilizado após uma instrução aritmética ou lógica. Os dois mnemônicos correspondem à mesma instrução.

JB/JNAE - Desvia se Abaixo

Codificação:

```
        JB      rótulo
        JNAE    rótulo
```

Operação:

Se CF = 1, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários

Os mnemônicos **JB** ("desvio se abaixo") e **JNAE** ("desvio se não acima ou igual") correspondem à mesma instrução, sendo normalmente utilizados após uma comparação entre números SEM sinal.

JNB/JAE - Desvia se não Abaixo

Codificação:

```
        JNB     rótulo
        JAE     rótulo
```

Operação:

Se CF = 0, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários

Os mnemônicos **JNB** ("desvio se não abaixo") e **JAE** ("desvio se acima ou igual") correspondem à mesma instrução, sendo normalmente utilizados após uma comparação entre números SEM sinal.

JA/JNBE - Desvia se Acima

Codificação:

```
        JÁ      rótulo
        JNBE    rótulo
```

Operação:

Se ZF=0 e CF=0, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários

Os mnemônicos **JA** ("desvio se acima") e **JNBE** ("desvio se não abaixo ou igual") correspondem à mesma instrução, sendo normalmente utilizados após uma comparação entre números SEM sinal.

JNA/JBE - Desvia se não Acima

Codificação:

```
        JNA     rótulo
        JBE     rótulo
```

Operação:

Se ZF=1 ou CF=1, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários

Os mnemônicos **JNA** ("desvio se não acima") e **JBE** ("desvio se abaixo ou igual") correspondem à mesma instrução, sendo normalmente utilizados após uma comparação entre números SEM sinal.

JL/JNGE - Desvia se Menor

Codificação:

```
        JL      rótulo
        JNGE    rótulo
```

Operação:

Se SF<>OF, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários

Os mnemônicos **JL** ("desvio se menor") e **JNGE** ("desvio se não maior ou igual") correspondem à mesma instrução, sendo normalmente utilizados após uma comparação entre números COM sinal.

JNL/JGE - Desvia se não Menor

Codificação:

```
        JNL     rótulo
        JGE     rótulo
```

Operação:

Se SF=OF, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários

Os mnemônicos **JNL** ("desvio se não menor") e **JGE** ("desvio se maior ou igual") correspondem à mesma instrução, sendo normalmente utilizados após uma comparação entre números COM sinal.

JG/JNLE - Desvia se Maior

Codificação:

```
        JG      rótulo
        JNLE    rótulo
```

Operação:

Se ZF=0 e SF=OF, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários

Os mnemônicos **JG** ("desvio se maior") e **JNLE** ("desvio se não menor ou igual") correspondem à mesma instrução, sendo normalmente utilizados após uma comparação entre números COM sinal.

JNG/JLE - Desvia se não Maior

Codificação:

```
        JNG     rótulo
        JLE     rótulo
```

Operação:

Se ZF=1 ou SF<>OF, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários

Os mnemônicos **JNG** ("desvio se não maior") e **JLE** ("desvio se menor ou igual") correspondem à mesma instrução, sendo normalmente utilizados após uma comparação entre números COM sinal.

JP/JPE - Desvia se Paridade Par

Codificação:

```
        JP      rótulo
        JPE     rótulo
```

Operação:

Se PF = 1, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários

Os mnemônicos **JP** e **JPE** correspondem à mesma instrução, sendo normalmente utilizados para testar se o resultado da última operação tem paridade PAR.

JNP/JPO - Desvia se Paridade Impar

Codificação:

```
        JNP     rótulo
        JPO     rótulo
```

Operação:

Se PF = 0, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários

Os mnemônicos **JNP** e **JPO** correspondem à mesma instrução sendo normalmente utilizados para testar se o resultado da última operação tem paridade IMPAR.

JS - Desvia se Negativo

Codificação:

```
        JS      rótulo
```

Operação:

Se SF = 1, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários

A instrução **JS** é normalmente utilizada para verificar se a última operação teve um resultado NEGATIVO.

JNS - Desvia se Positivo

Codificação:

```
        JNS     rótulo
```

Operação:

Se SF = 0, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários

A instrução **JNS** é normalmente usada para testar se o resultado da última operação é POSITIVO.

JO - Desvia se "Overflow"

Codificação:

```
        JO      rótulo
```

Operação:

Se OF = 1, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários

A instrução **JO** é normalmente utilizada para testar se a última operação resultou em "estouro" (*overflow*).

JNO - Desvia se não "Overflow"

Codificação:

```
        JNO     rótulo
```

Operação:

Se OF = 0, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários

A instrução **JNO** é normalmente utilizada para verificar se a última operação não resultou em "estouro" (*overflow*).

JCXZ - Desvia se CX = 0

Codificação:

```
        JCXZ    rótulo
```

Operação:

Se CX=0, soma a IP a distância (-128 a +127, contida na instrução) entre o operando e a instrução seguinte.

Flags Afetados

Nenhum

Comentários e Exemplos

A instrução **JCXZ** permite testar CX diretamente, sem a necessidade de uma operação aritmética ou lógica. É frequentemente utilizada para evitar a entrada em um laço ou repetição com contagem zero:

```
        MOV     CX,CONTAGEM
        JCXZ    CONT_ZERO
LACO:
        ; ...
        LOOP    LACO            ; se CX fosse zero repetiria 65536 vezes !
```

LOOP - Repete até CX=0

Codificação:

```
        LOOP    rótulo
```

Operação:

CX é decrementado; se o novo valor não for zero, a distância (-128 a +127, na própria instrução) entre a instrução seguinte e o rótulo destino é somada a IP.

Flags Afetados

Nenhum

Comentários e Exemplos

A instrução **LOOP** decrementa CX e efetua um desvio curto se CX não chegou a zero. É normalmente utilizada para repetir um trecho um certo número de vezes:

```
        MOV     CX,N_REPETICOES
LACO:
        ; trecho a ser repetido
        LOOP    LACO
```

Notar que a instrução LOOP não afeta os flags, apesar de envolver um decremento. Se CX contem inicialmente zero, o trecho é repetido 65536 vezes.

LOOPZ/LOOPE - Repete até CX=0 ou Z=0

Codificação:

```
        LOOPZ   rótulo
        LOOPE   rótulo
```

Operação:

CX é decrementado; se o novo valor não for zero e o flag **Z** estiver ativo, a distância (-128 a +127, na própria instrução) entre a instrução seguinte e o rótulo destino é somada a IP.

Flags Afetados

Nenhum

Comentários e Exemplos

A instrução **LOOPZ** (ou **LOOPE**) decrementa CX e efetua um desvio curto se CX não chegou a zero e o flag Z está ativo. É normalmente utilizada para repetir um trecho, no máximo um certo número de vezes, enquanto uma condição é atendida:

```
        MOV     CX,N_MAXIMO
LACO:
        ; trecho a ser repetido
        CMP     AL,1            ; teste da condição
        LOOPE   LACO            ; repete enquanto CX>0 e AL=1
        ; neste ponto AL <> 1 ou CX = 0
```

Notar que a instrução LOOPZ/LOOPE não afeta os flags, apesar de envolver um decremento. Se CX contem inicialmente zero, o trecho é repetido até 65536 vezes.

LOOPNZ/LOOPNE - Repete até CX=0 ou Z=1

Codificação:

```
        LOOPNZ  rótulo
        LOOPNE  rótulo
```

Operação:

CX é decrementado; se o novo valor não for zero e o flag **Z** estiver inativo, a distância (-128 a +127, na própria instrução) entre a instrução seguinte e o rótulo destino é somada a IP.

Flags Afetados

Nenhum

Comentários e Exemplos

A instrução **LOOPNZ** (ou **LOOPNE**) decrementa CX e efetua um desvio curto se CX não chegou a zero e o flag Z está inativo. É normalmente utilizada para repetir um trecho, no máximo um certo número de vezes, enquanto uma condição é atendida:

```
        MOV     CX,N_MAXIMO
LACO:
        ; trecho a ser repetido
        CMP     AL,1            ; teste da condição
        LOOPNE  LACO            ; repete enquanto CX>0 e AL<>1
        ; neste ponto AL = 1 ou CX = 0
```

Notar que a instrução **LOOPNZ**/**LOOPNE** não afeta os flags, apesar de envolver um decremento. Se CX contem inicialmente zero, o trecho é repetido até 65536 vezes.

CALL - Chamada a Subrotina

Codificação:

```
        CALL    rótulo
        CALL    reg
        CALL    mem
```

Operação:

Antes da execução da chamada propriamente dita, IP é atualizado para apontar para a instrução seguinte. As ações seguintes dependem do tipo de chamada:

- Chamada direta
  - intrasegmento
    - subtrai 2 de SP
    - coloca o conteúdo de IP em SS:SP
    - soma a IP a distância (contida na própria instrução) entre o destino e a instrução seguinte
  - intersegmento
    - subtrai 2 de SP
    - coloca o conteúdo de CS em SS:SP
    - subtrai 2 de SP
    - coloca o conteúdo de IP em SS:SP
    - carrega CS e IP com o endereço destino (contido na instrução)

- Chamada indireta
  - intrasegmento
    - subtrai 2 de SP
    - coloca o conteúdo de IP em SS:SP
    - carrega IP com o conteúdo do operando
  - intersegmento
    - subtrai 2 de SP
    - coloca o conteúdo de CS em SS:SP
    - subtrai 2 de SP
    - coloca o conteúdo de IP em SS:SP
    - carrega CS e IP com o conteúdo do operando

Flags Afetados

Nenhum

Comentários e Exemplos

O tipo de chamada é determinado pelo operando:

operando	Chamada
rótulo near	direta intrasegmento
rotúlo far	direta intersegmento
registrador	indireta intrasegmento
variável word	indireta intrasegmento
variável dword	indireta intersegmento

Alguns exemplos:

```
END_NEAR DW      ?                ;variável tipo WORD
END_FAR  DD      ?                ;variável tipo DWORD

ROT_NEAR LABEL   NEAR             ;rotina NEAR
ROT_FAR  LABEL   FAR              ;rotina FAR

         CALL    ROT_NEAR         ;direta, intrasegmento
         CALL    ROT_FAR          ;direta, intersegmento

         CALL    END_NEAR         ;indireta, intrasegmento
         CALL    END_FAR          ;indireta, intersegmento

         MOV     AX,OFFSET ROT_NEAR
         CALL    AX               ;indireta, intrasegmento, chama ROT_NEAR

         MOV     END_NEAR,OFFSET ROT_NEAR
         MOV     BX,OFFSET END_NEAR
         CALL    [BX]             ;"duplamente" indireta, intrasegmento,
                                  ; chama ROT_NEAR
```

RET - Retorno de Subrotina

Codificação:

```
        RET
        RET     imed
```

Operação:

Retorna de uma subrotina, através dos seguintes passos:

1. Carrega IP com SS:SP, soma 2 a SP
2. Se retorno intersegmento, carrega CS com SS:SP, soma 2 a SP
3. Se foi especificado valor imediato, soma-o a SP

Flags Afetados

Nenhum

Comentários e Exemplos

O tipo de retorno (intra ou intersegmento) é determinado pelo par mais interno de diretivas **PROC/ENDP** que cercam a instrução **RET**. Se o **RET** estiver fora de um par **PROC/ENDP** ele será um **RET** intrasegmento.

Notar que é vital o "casamento" do tipo do **CALL** com o tipo do **RET** para a correta restauração de CS, IP e SP.

A possibilidade de incluir um valor a ser somado a SP permite retirar da pilha parâmetros colocados por quem chamou a rotina:

```
        ; Programa Principal
        PUSH    PAR_1           ;coloca na pilha 1o parâmetro
        PUSH    PAR_2           ;coloca na pilha 2o parâmetro
        CALL    ROTINA          ;chama subrotina
        ; Subrotina
ROTINA  PROC    NEAR
        MOV     BP,SP           ;BP aponta p/ IP,PAR_2,PAR_1
        MOV     AX,[BP+2]       ;PAR_2
        MOV     BX,[BP+4]       ;PAR_1
        ; ...
        RET     4               ;retira PAR_1 e PAR_2 da pilha
ROTINA  ENDP
```

INT - Interrupção de Software

Codificação:

```
        INT     imed
```

Operação:

Gera uma *interrupção de software*, através dos seguintes passos:

1. Avança IP para a próxima instrução
2. Subtrai 2 de SP e coloca os flags em SS:SP
3. Desativa os flags IF e TF
4. Subtrai 2 de SP e coloca CS em SS:SP
5. Subtrai 2 de SP e coloca IP em SS:SP
6. Executa um desvio indireto intersegmento, obtendo o endereço destino do doubleword de endereço "físico" 4 * número da interrupção.

Flags Afetados

IF e TF

Comentários e Exemplos

A instrução **INT** é um tipo especial de desvio indireto intersegmento, sendo normalmente utilizado para passar o controle entre módulos (por exemplo, para um programa aplicativo requisitar um serviço ao sistema operacional).

Ao se escrever rotinas que serão chamadas através de **INT**, não se deve esquecer que a interrupção é inibida e os flags originais são colocados na pilha. Por estes motivos, normalmente estas rotinas são iniciadas por uma instrução **STI** (habilita interrupções) e finalizadas por uma instrução **IRET** (retorna restaurando os flags).

Finalmente, lembrar que existem 256 interrupções possíveis, de 0 a 0FFH.

INTO - Gera Interrupção se Overflow

Codificação:

```
        INTO
```

Operação:

Se OF estiver ativo, gera uma interrupção de software, através dos seguintes passos:

1. Avança IP para a próxima instrução
2. Subtrai 2 de SP e coloca os flags em SS:SP
3. Desativa os flags IF e TF
4. Subtrai 2 de SP e coloca CS em SS:SP
5. Subtrai 2 de SP e coloca IP em SS:SP
6. Executa um desvio indireto intersegmento, obtendo o endereço destino do doubleword de endereço "físico" 00010H

Flags Afetados

IF e TF

Comentários e Exemplos

A instrução **INTO** se destina a facilitar a detecção de erros aritméticos em programas, sendo normalmente executada apos uma operação sobre números com sinal. Caso o resultado não caiba no destino, OF será ativado e, consequentemente, chamada uma rotina. Esta rotina pode, por exemplo, fornecer o endereço de programa em que ocorreu o erro, o conteúdo dos registradores, etc.

Ao se escrever uma rotina que será chamada através de **INTO**, não se deve esquecer que a interrupção é inibida e os flags originais são colocados na pilha. Por estes motivos, normalmente esta rotina é iniciada por uma instrução **STI** (habilita interrupções) e finalizada por uma instrução **IRET** (que retorna restaurando os flags).

IRET - Retorno de Interrupção

Codificação:

```
        IRET
```

Operação:

Retorna de uma interrupção, através dos seguintes passos:

1. Carrega IP com SS:SP, soma 2 a SP
2. Carrega CS com SS:SP, soma 2 a SP
3. Carrega Flags com SS:SP, soma 2 a SP

Flags Afetados

Todos

Comentários e Exemplos

A instrução **IRET** efetua um retorno intersegmento, restaurando da pilha o conteúdo dos flags. A pilha deve conter, em ordem de endereço crescente, IP CS e flags (que é a ordem obtida através de interrupções de hardware e software).

Notar que **IRET** e **POPF** são as únicas instruções capazes de alterar TF.

3.8 INSTRUÇÕES DE CONTROLE DO PROCESSADOR

CLC - Desativa Flag de "Vai Um"

Codificação:

```
CLC
```

Operação:

Desativa o flag **CF** ("vai um").

Flags Afetados

CF

CMC - Complementa Flag de "Vai Um"

Codificação:

```
CMC
```

Operação:

Complementa o flag **CF** ("vai um"), isto é, ativa-o se estiver desativado e vice-versa.

Flags Afetados

CF

STC - Ativa Flag de "Vai Um"

Codificação:

```
STC
```

Operação:

Ativa o flag **CF** ("vai um").

Flags Afetados

CF

CLD - Desativa Flag de Direção

Codificação:

```
CLD
```

Operação:

Desativa o flag de direção (**DF**).

Flags Afetados

DF

Comentário

A instrução **CLD** seleciona o INCREMENTO de ponteiros nas instruções de manipulação de cadeias.

STD - Ativa Flag de Direção

Codificação:

```
STD
```

Operação:

Ativa o flag de direção (**DF**).

Flags Afetados

DF

Comentário

A instrução **STD** seleciona o DECREMENTO de ponteiros nas instruções de manipulação de cadeias.

CLI - Desabilita Interrupções

Codificação:

```
CLI
```

Operação:

Desativa o flag de interrupção (**IF**).

Flags Afetados

IF

Comentário

A instrução **CLI** desabilita a ocorrência de interrupções de hardware, exceto a interrupção não mascarável. Não afeta as interrupções de software.

STI - Habilita Interrupções

Codificação:

```
STI
```

Operação:

Ativa o flag de interrupção (**IF**).

Flags Afetados

IF

Comentário

A instrução **STI** habilita o tratamento de interrupções de hardware.

HLT - Para o processador

Codificação:

```
        HLT
```

Operação:

O processador fica parado até que ocorra uma interrupção de hardware, ou seja reiniciado (através de "reset"). No caso da ocorrência de uma interrupção, o processamento prossegue, após o tratamento da interrupção, com a instrução seguinte ao **HLT**.

Flags Afetados

Nenhum

Comentário

A instrução **HLT** é raramente usada; caso seja utilizada para aguardar uma interrupção, é importante não esquecer antes de habilitar as interrupções via **STI**.

ESC - Envia Instrução para Coprocessador

Codificação:

```
        ESC     imed,mem
```

Operação:

A instrução **ESC** faz com que o 8088 coloque na via de endereços o endereço referenciado na instrução, sem efetuar qualquer transferência de dados.

Flags Afetados

Nenhum

Comentário

A instrução ESC é utilizada em sistemas que utilizam um coprocessador (por exemplo, o coprocessador numérico 8087). O valor imediato especificado (de 0 a 63) é tratado pelo coprocessador como uma instrução.

WAIT - Espera

Codificação:

```
        WAIT
```

Operação:

A instrução **WAIT** coloca o processador em um estado de espera, até receber um sinal no pino "TEST".

Flags Afetados

Nenhum

Comentário

A instrução **WAIT** é utilizada para sincronizar o 8088 com um hardware externo (por exemplo, o coprocessador numérico 8087). Durante a espera, o processador continua atendendo as interrupções.

LOCK - Prefixo de Exclusividade da Via

Codificação:

```
        LOCK
```

Operação:

O prefixo **LOCK** faz com que o 8088 indique que a via de dados e endereços não pode ser compartilhada durante a execução da instrução.

Flags Afetados

Nenhum

Comentário

O prefixo **LOCK** é utilizado para garantir exclusividade de acesso em máquinas onde vários processadores tem que compartilhar recursos de hardware (por exemplo memória).

Um problema típico surge quando existe uma variável que indica se um determinado dispositivo está em uso (este tipo de variável é normalmente denominado SEMAFORO).

Vamos supor que esta variável seja 1 quando o dispositivo está em uso e zero quando está livre. Admitamos também que um pedido de uso pode ocorrer tanto no processamento normal como durante o tratamento de uma interrupção.

Consideremos primeiro o caso de um ambiente com um único processador. As instruções abaixo NÃO são adequadas:

```
        CMP     SEMAFORO,1      ; (A)
        JE      OCUPADO
        MOV     SEMAFORO,1      ; (B)
```

pois entre a instrução (A) e a instrução (B) pode ter havido uma interrupção que alterou o conteúdo do semáforo. A instrução **XCHG** permite ler e escrever em uma variável de forma não interrompível:

```
        MOV     AL,1
        XCHG    SEMAFORO,AL
        CMP     AL,1
        JE      OCUPADO           ;desvia se continha 1
                                  ; livre, passou a conter 1
```

Em um ambiente com mais de um processador, é possível que um processador assuma o controle da memória entre a leitura e escrita comandadas pela instrução **XCHG** que está sendo executada em outro. Desta forma é necessário o prefixo **LOCK**:

```
        MOV     AL,1
        LOCK    XCHG  SEMAFORO,AL
        CMP     AL,1
        JE      OCUPADO           ;desvia se continha 1
                                  ; livre, passou a conter 1
```

O prefixo LOCK pode ser usado em conjunto com o "segment override"; por outro lado, só deve ser utilizado em conjunto com os prefixos de repetição se as interrupções estiverem inibidas.

4 TÓPICOS AVANÇADOS

4.1 MÓDULOS

Um programa assembly não precisa ser composto de um único fonte. É possível codificá-lo em várias partes (**MÓDULOS**) e depois "ligar" todos os objetos para obter o programa. Esta tarefa de "ligar" os módulos é normalmente feita por um programa **LIGADOR de MODULOS** ("*linker*").

Neste item veremos algumas diretivas relacionadas ao uso de módulos.

4.1.1 DIRETIVA NAME

A diretiva NAME permite especificar um nome para o módulo sendo “assemblado”:

```
        NAME    nome
```

O nome do módulo é utilizado pelo programa LIGADOR.

4.1.2 DIRETIVA PUBLIC

A diretiva PUBLIC é utilizada para indicar quais símbolos de um módulo podem ser referenciados em outros:

```
        PUBLIC  lista de símbolos
```

Por exemplo, suponhamos que em um módulo A sejam definidas as variáveis VAR1 e VAR2 e os rótulos ROT1 e ROT2. Para que estes símbolos possam ser mencionados em um outro módulo é preciso colocar no módulo A a diretiva

```
        PUBLIC  VAR1, VAR2, ROT1, ROT2
```

A diretiva PUBLIC pode ser colocada em qualquer lugar do programa fonte.

4.1.3 DIRETIVA EXTRN

A diretiva EXTRN é utilizada para indicar que serão utilizados símbolos definidos em outros módulos e quais os seus tipos:

```
        EXTRN   nome:tipo, nome:tipo, ...
```

Os tipos podem ser BYTE, WORD, DWORD, NEAR, FAR e ABS. Este último tipo se refere a uma constante e não a um símbolo propriamente dito.

Retomando o exemplo anterior, para que um módulo B use as variáveis VAR1 (byte) e VAR2 (word) e as rotinas ROT1 (near) e ROT2 (far) definidas no módulo A, temos que incluir a diretiva

```
        EXTRN   VAR1:BYTE, VAR2:WORD
        EXTRN   ROT1:NEAR, ROT2:FAR
```

O posicionamento da diretiva EXTRN é importante: as variáveis e rotinas "near" que ela menciona serão consideradas como definidas no segmento onde a diretiva é utilizada:

```
A       SEGMENT
        EXTRN   X:BYTE
A       ENDS

B       SEGMENT
        EXTRN   Y:NEAR
B       SEGMENT
```

faz com que a variável X seja considerada como definida no segmento A e o rótulo Y no segmento B.

4.1.4 DIRETIVA END

Conforme já dito, a diretiva END marca o fim do programa fonte e permite a indicação do endereço da primeira instrução a ser executada.

Quando um programa consiste de vários módulos, somente um deles (o "programa principal") deve especificar o endereço inicial de execução.

4.2 MAIS DETALHES SOBRE SEGMENTOS

4.2.1 DIRETIVA SEGMENT

Na explicação anterior da diretiva SEGMENT, algumas opções foram propositalmente omitidas. O formato completo é:

```
Nome    SEGMENT [alinhamento] [combinação] ['classe']
```

onde os colchetes indicam que os três parâmetros são opcionais.

O **ALINHAMENTO** é utilizado para indicar onde um segmento deve ser posicionado pelo ligador, e pode ser:

BYTE o segmento pode começar em qualquer endereço

WORD o segmento deve começar em um endereço par

PARA o segmento deve começar em um endereço múltiplo de 16

PAGE o segmento deve começar em um endereço múltiplo de 256

INPAGE o segmento tem que ficar todo contido em uma página, isto é, entre dois múltiplos consecutivos de 256

Se não for especificado alinhamento, o assembler assume PARA.

A **COMBINAÇÃO** indica como o segmento deverá ser combinado com segmentos de outros módulos:

PUBLIC o segmento será agrupado com outros segmentos de mesmo nome, também públicos

COMMON o segmento compartilhará as mesmas posições que todos os outros segmentos COMMON de mesmo nome. Ou seja, todos os segmentos COMMON de mesmo nome descrevem a mesma área de memória

AT expr o segmento será localizado no endereço 16*expr. Por exemplo, "AT 12H" significa que o segmento começa no endereço "físico" 120H

STACK os segmentos STACK de mesmo nome são combinados de forma que o tamanho final do segmento seja o do maior deles e o offset do último item de todos eles seja igual. Por exemplo:

```
PILHA     SEGMENT   STACK          ;módulo 1
          DW        20H DUP (?)
FIM_1     LABEL     WORD
PILHA     ENDS

PILHA     SEGMENT   STACK          ;módulo 2
          DW        10H DUP (?)
FIM_2     LABEL     WORD
PILHA     ENDS
```

O segmento PILHA final terá 30H words e FIM_1 e FIM_2 terão offset 60H.

MEMORY semelhante a COMMON, porém este segmento é colocado ao final de todos os outros (em endereço mais alto).

Caso não seja especificada a combinação, o segmento não será agrupado a outros (mesmo que de mesmo nome).

Finalmente, **CLASSE** é um nome qualquer. Os segmentos de mesma classe são colocados juntos na memória.

4.2.2 DIRETIVA ORG

Ao se declarar pela primeira vez um segmento, as variáveis e instruções começam a ser alocadas a partir do deslocamento 0. Caso o segmento seja fechado (via ENDS) e posteriormente reaberto, a alocação continua do último deslocamento anterior.

A diretiva ORG é utilizada para especificar o deslocamento em que serão alocados os símbolos seguintes. O uso conjunto de AT e ORG permite declarar variáveis e rotinas em endereços absolutos.

Por exemplo:

```
X         SEGMENT AT 12H

A         LABEL   BYTE             ;fica em 12H:0H

          ORG     34H
B         LABEL   BYTE             ;fica em 12H:34H

X         ENDS
```

4.2.3 DIRETIVA GROUP

Até o momento sempre acessamos símbolos através de registradores de segmento contendo o segmento em que o símbolo foi definido.

Existem ocasiões em que, apesar de definirmos vários segmentos, desejamos acessar símbolos definidos neles através de um único valor nos registradores de segmento. Isto é feito informando ao assembler que estes segmentos constituem um **GRUPO**, através da diretiva GROUP:

```
nome      GROUP   seg1, seg2, ...
```

Para que isto seja possível, todos os segmentos devem ficar em um mesmo bloco de 64K bytes, o que é verificado pelo ligador de módulos.

Se, em um ASSUME, dizemos que um registrador contém um grupo, o assembler passa a informar ao ligador que os deslocamentos tem de ser ajustados em função da posição do segmento no grupo. O operador OFFSET normalmente fornece o deslocamento em relação ao segmento em que o símbolo foi definido; para obter o deslocamento de um símbolo em um grupo utiliza-se a forma

```
          OFFSET  grupo:símbolo
```

O exemplo abaixo ilustra o uso de grupos:

```
CONST     SEGMENT
UM        DB      1
CONST     ENDS

VAR       SEGMENT
VARIAVEL  DB      ?
VAR       ENDS

          ASSUME  DS:VAR
          MOV     AL,VARIAVEL       ;contém o deslocamento de VARIAVEL
                                    ; no segto VAR

DADOS     GROUP   CONST, VAR
          ASSUME  DS:DADOS
          MOV     AL,VARIAVEL       ;contém o deslocamento de VARIAVEL
                                    ; no grupo DADOS
          MOV     BX,OFFSET VARIAVEL; ERRO !
          MOV     AL,[BX]           ; DS contém DADOS mas a instrução
                                    ; contem deslocamento no segmento VAR
```

4.3 DIRETIVA RECORD E OS OPERADORES MASK E WIDTH

É muito comum armazenar-se mais de uma informação em um único byte ou word, associando-se a cada uma delas um certo número de bits. Cada um destes grupos de bits será um **CAMPO**.

A diretiva RECORD permite definir a organização de um byte ou word neste tipo de aplicação, associando nomes aos campos:

```
Nome      RECORD campo:largura [= expr] [,campo:largura ...]
```

onde

nome é o nome do "*record*", que será usado para alocar uma área com a organização descrita.

campo é o nome de um campo. Este nome pode ser usado como uma constante que indica quantos bits devem ser deslocados para encostar o campo à direita.

largura é o número de bits (1 a 16) ocupados pelo campo.

expr é o valor padrão a ser utilizado na alocação de áreas através do nome do "*record*". Sua presença não é obrigatória.

A definição de um "*record*" não aloca memória; apenas define como é a estrutura de campos. Uma vez definido um "*record*" pode-se alocar uma área com o seu formato de modo semelhante ao usado para definir bytes e words:

```
[nome]    nome do record < [iniciação] >
```

O primeiro nome (opcional) corresponde ao símbolo que será utilizado para acessar a área. A "iniciação" é uma lista de expressões separadas por vírgulas, que correspondem aos valores a serem colocados nos campos.

Caso o número total de bits seja menor ou igual a 8, é alocado um byte; caso contrário, é alocado um word. Os bits são sempre alocados de forma a que o último bit do último campo seja o bit menos significativo da área alocada.

O operador MASK permite obter um valor que isola um campo (através de um AND); o operador WIDTH fornece o número de bits em um campo.

O exemplo abaixo resume o que foi dito:

```
DESCRITOR RECORD LIVRE:1=0, USO:3, TAMANHO:4
                                   ; define uma área com 3 campos:
                                   ;  LIVRE   com 1 bit e valor padrão 0
                                   ;  USO     com 3 bits
                                   ;  TAMANHO com 4 bits
                                   ; total = 8 bits -> ocupa um byte

DESCR_1  DESCRITOR <>              ; LIVRE contém 0, demais campos indefinidos
DESCR_2  DESCRITOR <1,2,3>         ; LIVRE = 1, USO = 2 e CONTEUDO = 3

         MOV     AL,MASK USO       ; AL = 01110000B
         AND     AL,DESCR_1
         MOV     CL,USO            ; CL = 4
         SHR     AL,CL             ; AL = campo de USO, encostado à direita
```

4.4 DIRETIVA STRUC

A diretiva STRUC permite definir uma ESTRUTURA, composta por vários campos de tipo byte, word ou doubleword.

O formato da definição de uma estrutura é:

```
nome da estrutura  STRUC

[nome de campo]    DB/DW/DD  iniciação
                   .
                   .

nome da estrutura  ENDS
```

onde DB/DW/DD indica que uma destas três diretivas deve ser usada.

Como a diretiva RECORD, STRUC não aloca memória; apenas define como é a organização da estrutura. Os nomes de campos são utilizados para acessar o campo. Isto é feito colocando-se no final de uma referência à memória (que deve apontar para o primeiro campo da estrutura) a terminação ".nome do campo".

Os valores iniciais fornecidos nas diretivas DB, DW e DD serão os valores padrão quando do uso do nome da estrutura para alocação de memória. Esta alocação é feita de forma semelhante à usada com "records":

```
[nome]  nome da estrutura  <iniciação>
```

Na iniciação só é permitido alterar o valor padrão de campos que possuem um único valor.

O exemplo abaixo ilustra o uso de estruturas:

```
REGISTRO STRUC

NOME     DB      "ABCD"            ; 4 bytes, valor padrão pode ser alterado
SALARIO  DB      10 DUP (?)        ; 10 bytes, valor padrão não pode ser alterado
DEPTO    DW      1                 ; 1 word, valor padrão pode ser alterado

REGISTRO  ENDS

REG_1    REGISTRO <"JOAO",,>       ; 1o campo alterado

         MOV     AX,REG_1.DEPTO    ; AX contém o campo DEPTO de REG_1
         MOV     BX,OFFSET REG_1
         MOV     AX,[BX].DEPTO     ; idem
         MOV     BX,OFFSET REG_1.NOME  ; BX contém o offset do inicio
                                       ; do campo NOME de REG_1
```

5 PROGRAMA EXEMPLO

Neste capítulo vamos ver um programa completo, como um exemplo de uso das principais instruções.

O programa é uma pequena "calculadora" hexadecimal e decimal, que aceita comandos na seguinte forma:

```
número operador número
```

Entre os números e o operador não é permitido qualquer caractere. Os operadores permitidos são:

+ soma
- subtração
* multiplicação
/ divisão
% resto
& "e" lógico
| "ou" lógico

Os números podem ser fornecidos em hexadecimal ou em decimal; neste último caso, deve ser precedido do sinal "#". Os números podem ainda ser precedidos de um sinal positivo ("+") ou negativo ("-"), sendo armazenados em 16 bits. São impostos os seguintes limites:

+FFFF <= numero hexa <= -FFFF

+#32767 <= numero decimal <= -#32767

O resultado da operação é apresentado em hexadecimal (sem sinal) e em decimal (com sinal). Abaixo temos alguns exemplos:

```
>100+1
 0101 +00257
>-100+1
 FF01 -00255
>-#100/#33
 FFFD -00003
>-#100%#33
 FFFF -00001
```

O sinal ">" é colocado pelo programa para indicar que está pronto para receber um comando.

Nas páginas seguintes é apresentada a listagem completa e comentada deste programa exemplo. Procurou-se dar mais ênfase ao aspecto didático do programa que à eficiência; sugere-se ao leitor procurar outras soluções para o problema, exercitando assim o conhecimento adquirido no restante do livro.

```
----- Objeto ----------      ------- Programa Fonte ----------------------------------------------------

                             ;
                             ;       PROGRAMA EXEMPLO
                             ;

                             ;       DEFINICAO DE CONSTANTES

= 0014                       MAX_LIN         EQU     20              ;TAMANHO MAXIMO DE UMA LINHA DE
                                                                     ; COMANDO

= 000D                       CR              EQU     0DH             ;CARACTER "CARRIAGE RETURN"
                                                                     ; (RETORNO DO CARRO)

= 000A                       LF              EQU     0AH             ;CARACTER "LINE FEED"
                                                                     ; (AVANCO DE LINHA)


                             ;       DEFINICAO DA PILHA

0000                         STACK           SEGMENT STACK

0000     80 [                                DW      128 DUP (?)     ;128 WORDS, NAO INICIADOS
            ????
                ]

0100                         TOPO_DA_PILHA   LABEL   WORD            ;LEMBRAR QUE A PILHA CRESCE NO
                                                                     ; SENTIDO DOS ENDERECOS DECRESCENTES

0100                         STACK           ENDS


                             ;       DEFINICAO DAS VARIAVEIS

0000                         DADOS           SEGMENT


0000  14                     AREA_LEITURA    DB      MAX_LIN         ;INICIO DA AREA PARA LEITURA DE COMANDO
                                                                     ; CONTEM O NUMERO MAXIMO DE CARACTERES
0001  ??                    TAM_LINHA       DB      ?               ;NUMERO DE CARACTERES LIDOS
0002     14 [                LINHA           DB      MAX_LIN DUP (?) ;CARACTERES LIDOS
            ??
               ]


0016  ????                   NUMERO_1        DW      ?               ;PRIMEIRO OPERANDO
0018  ????                   NUMERO_2        DW      ?               ;SEGUNDO  OPERANDO
001A  ??                     OPERADOR        DB      ?               ;OPERADOR A SER USADO
```

```
----- Objeto ----------      ------- Programa Fonte ----------------------------------------------------

                             ;       DEFINICAO DE TABELAS

001B  2B 2D 2A 2F 25 26      TAB_OPERADORES  DB      '+-*/%&|'       ;OPERADORES VALIDOS
      7C

= 0007                       N_OPERADORES    EQU     7               ;NUMERO DE OPERADORES VALIDOS

0022  0067 R                 TAB_ROT         DW      OFFSET SOMA     ;TABELA COM OS ENDERECOS DAS ROTINAS
0024  0074 R                                 DW      OFFSET SUBTRACAO; QUE EXECUTAM AS OPERACOES
0026  0081 R                                 DW      OFFSET MULTIPLICACAO
0028  008E R                                 DW      OFFSET DIVISAO
002A  00A1 R                                 DW      OFFSET RESTO
002C  00B6 R                                 DW      OFFSET E_LOGICO
002E  00C1 R                                 DW      OFFSET OU_LOGICO

                             ;       DEFINICAO DE MENSAGENS


0030  0D 0A                  APRESENTACAO    DB      CR,LF
0032  43 61 6C 63 75 6C                     DB      'Calculadora Decimal/Hexadecimal'
      61 64 6F 72 61 20
      44 65 63 69 6D 61
      6C 2F 48 65 78 61
      64 65 63 69 6D 61
      6C
0051  0D 0A                                 DB      CR,LF
0053  24                                     DB      '$'

0054  0D 0A 3E 24           PEDE_CMD        DB      CR,LF,'>','$'

0058  20 2A 2A 2A 20 4C     LINHA_INVALIDA  DB      ' *** Linha invalida ***'
      69 6E 68 61 20 69
      6E 76 61 6C 69 64
      61 20 2A 2A 2A
006F  24                                     DB      '$'

0070  20 2A 2A 2A 20 52     MUITO_GRANDE    DB      ' *** Resultado muito grande ***'
      65 73 75 6C 74 61
      64 6F 20 6D 75 69
      74 6F 20 67 72 61
      6E 64 65 20 2A 2A
      2A
008F  24                                     DB      '$'
```

```
----- Objeto ----------      ------- Programa Fonte ----------------------------------------------

0090  20 2A 2A 2A 20 44      DIV_POR_ZERO    DB      ' *** Divisao por zero ***'
      69 76 69 73 61 6F
      20 70 6F 72 20 7A
      65 72 6F 20 2A 2A
      2A
00A9  24                                     DB      '$'


00AA                         DADOS           ENDS

0000                         CODIGO          SEGMENT

                                             ASSUME  CS:CODIGO

                             ;       INICIO DA EXECUCAO

0000                         INICIO:
0000  B8 ---- R                              MOV     AX,STACK                ; (NAO EXISTE "MOV" IMEDIATO
0003  8E D0                                  MOV     SS,AX                   ;  P/ REGISTRADOR DE SEGMENTO)
0005  BC 0100 R                              MOV     SP,OFFSET TOPO_DA_PILHA ;INICIA PILHA

0008  B8 ---- R                              MOV     AX,DADOS
000B  8E D8                                  MOV     DS,AX
000D  8E C0                                  MOV     ES,AX                   ;INICIA DS E ES

                                             ASSUME  DS:DADOS, ES:DADOS

000F  BA 0030 R                              MOV     DX,OFFSET APRESENTACAO
0012  E8 01CE R                              CALL    MOSTRA_MENSAGEM         ;MOSTRA APRESENTACAO

                             ;       PROGRAMA PRINCIPAL

0015                         PRINC:
0015  BA 0054 R                              MOV     DX,OFFSET PEDE_CMD
0018  E8 01CE R                              CALL    MOSTRA_MENSAGEM         ;PEDE COMANDO
001B  BA 0000 R                              MOV     DX,OFFSET AREA_LEITURA
001E  E8 01BF R                              CALL    LE_LINHA                ;LE LINHA DE COMANDO
0021  E8 0026 R                              CALL    TRATA_LINHA             ;INTERPRETA A LINHA
0024  EB EF                                  JMP     PRINC                   ;REPETE
```

```
----- Objeto ----------      ------- Programa Fonte ----------------------------------------------------

                             ;       TRATA LINHA DE COMANDO

0026                         TRATA_LINHA    PROC    NEAR

0026  FC                                    CLD                         ;AUTO-INCREMENTO
0027  32 ED                                XOR     CH,CH
0029  8A 0E 0001 R                         MOV     CL,TAM_LINHA        ;CX = # DE CARACTERES A TRATAR
002D  BE 0002 R                             MOV     SI,OFFSET LINHA     ;SI = PONTEIRO P/ PROXIMO
                                                                        ; CARACTER A TRATAR

0030  E8 00DA R                             CALL    LE_NUMERO           ;OBTEM 1o OPERANDO
0033  72 2B                                 JC      ERRO                ;DESVIA SE ERRO
0035  A3 0016 R                             MOV     NUMERO_1,AX         ;GUARDA 1o OPERANDO

0038  E3 26                                 JCXZ    ERRO                ;ERRO SE ACABOU A LINHA
003A  AC                                    LODSB                       ;PEGA PROXIMO CARACTER
003B  A2 001A R                             MOV     OPERADOR,AL         ;GUARDA OPERADOR
003E  49                                    DEC     CX                  ;MAIS UM TRATADO

003F  E8 00DA R                             CALL    LE_NUMERO           ;OBTEM 2o OPERANDO
0042  72 1C                                 JC      ERRO                ;DESVIA SE ERRO
0044  A3 0018 R                             MOV     NUMERO_2,AX         ;GUARDA 2o OPERANDO

0047  A0 001A R                             MOV     AL,OPERADOR         ;AL = OPERADOR
004A  BF 0018 R                             MOV     DI,OFFSET TAB_OPERADORES;SI = INICIO DA TABELA
004D  B9 0007                               MOV     CX,N_OPERADORES     ;CX = # DE OPERADORES
0050  F2/ AE                        REPNE  SCASB                       ;PROCURA NA TABELA
0052  75 0C                                 JNE     ERRO                ;ERRO SE NAO ACHOU


0054  81 EF 001B R                          SUB     DI,OFFSET TAB_OPERADORES;DI = INDICE DO OPERADOR
0058  4F                                    DEC     DI                  ;LEMBRAR QUE PAROU APONTANDO
                                                                        ; PARA O SEGUINTE
0059  D1 E7                                 SAL     DI,1                ;MULTIPLICA POR DOIS, PARA
                                                                        ; ACESSAR TABELA DE WORDS
005B  FF 95 0022 R                          CALL    TAB_ROT[DI]         ;CHAMA A ROTINA CORRESPONDENTE
                                                                        ; AO OPERADOR


005F  C3                                    RET                         ;LINHA TRATADA

0060                         ERRO:
0060  BA 005B R                             MOV     DX,OFFSET LINHA_INVALIDA
0063  E8 01CE R                             CALL    MOSTRA_MENSAGEM     ;INFORMA ERRO
0066  C3                                    RET                         ;LINHA TRATADA

0067                         TRATA_LINHA    ENDP
```

```
----- Objeto ----------      ------- Programa Fonte ----------------------------------------------------

                             ;       ROTINAS QUE EXECUTAM AS OPERACOES

0067                         OPERACOES       PROC    NEAR


0067                         SOMA:
0067  A1 0016 R                              MOV     AX,NUMERO_1             ;PEGA PRIMEIRO OPERANDO
006A  03 06 0018 R                           ADD     AX,NUMERO_2             ;SOMA O SEGUNDO
006E  70 5C                                  JO      ESTOURO                 ;DESVIA SE "OVERFLOW"
0070  E8 0156 R                             CALL    MOSTRA_RESULTADO        ;MOSTRA O RESULTADO
0073  C3                                    RET

0074                         SUBTRACAO:
0074  A1 0016 R                              MOV     AX,NUMERO_1             ;PEGA PRIMEIRO OPERANDO
0077  2B 06 0018 R                           SUB     AX,NUMERO_2             ;SUBTRAI O SEGUNDO
007B  70 4F                                  JO      ESTOURO                 ;DESVIA SE "OVERFLOW"
007D  E8 0156 R                             CALL    MOSTRA_RESULTADO        ;MOSTRA O RESULTADO
0080  C3                                    RET

0081                         MULTIPLICACAO:
0081  A1 0016 R                              MOV     AX,NUMERO_1             ;PEGA PRIMEIRO OPERANDO
0084  F7 2E 0018 R                           IMUL    NUMERO_2                ;MULTIPLICA PELO SEGUNDO
0088  70 42                                  JO      ESTOURO                 ;DESVIA SE "OVERFLOW"
008A  E8 0156 R                             CALL    MOSTRA_RESULTADO        ;MOSTRA O RESULTADO
008D  C3                                    RET

008E                         DIVISAO:
008E  83 3E 0018 R 00                       CMP     NUMERO_2,0              ;SE DIVISOR FOR ZERO
0093  74 3E                                 JE      ERRO_DIV                ;   ERRO !

0095  A1 0016 R                             MOV     AX,NUMERO_1             ;PEGA PRIMEIRO OPERANDO
0098  99                                    CWD                             ;"ESTENDE"
0099  F7 3E 0018 R                          IDIV    NUMERO_2                ;DIVIDE PELO SEGUNDO
009D  E8 0156 R                             CALL    MOSTRA_RESULTADO        ;MOSTRA O RESULTADO
00A0  C3                                    RET

00A1                         RESTO:
00A1  83 3E 0018 R 00                       CMP     NUMERO_2,0              ;SE DIVISOR FOR ZERO
00A6  74 2B                                 JE      ERRO_DIV                ;   ERRO !

00A8  A1 0016 R                             MOV     AX,NUMERO_1             ;PEGA PRIMEIRO OPERANDO
00AB  99                                    CWD                             ;"ESTENDE"
00AC  F7 3E 0018 R                          IDIV    NUMERO_2                ;DIVIDE PELO SEGUNDO
00B0  8B C2                                 MOV     AX,DX                   ;AX = RESTO
00B2  E8 0156 R                             CALL    MOSTRA_RESULTADO        ;MOSTRA O RESULTADO
00B5  C3                                    RET
```

```
----- Objeto ----------      ------- Programa Fonte ----------------------------------------------------

00B6                        E_LOGICO:
00B6  A1 0016 R                         MOV     AX,NUMERO_1             ;PEGA PRIMEIRO OPERANDO
00B9  23 06 0018 R                      AND     AX,NUMERO_2             ;"E" COM O SEGUNDO
00BD  E8 0156 R                         CALL    MOSTRA_RESULTADO        ;MOSTRA O RESULTADO
00C0  C3                                RET

00C1                        OU_LOGICO:
00C1  A1 0016 R                         MOV     AX,NUMERO_1             ;PEGA PRIMEIRO OPERANDO
00C4  0B 06 0018 R                      OR      AX,NUMERO_2             ;"OU" COM O SEGUNDO
00C8  E8 0156 R                         CALL    MOSTRA_RESULTADO        ;MOSTRA O RESULTADO
00CB  C3                                RET

00CC                        ESTOURO:
00CC  BA 0070 R                         MOV     DX,OFFSET MUITO_GRANDE
00CF  E8 01CE R                         CALL    MOSTRA_MENSAGEM         ;AVISA ERRO
00D2  C3                                RET

00D3                        ERRO_DIV:
00D3  BA 0090 R                         MOV     DX,OFFSET DIV_POR_ZERO
00D6  E8 01CE R                         CALL    MOSTRA_MENSAGEM         ;AVISA ERRO
00D9  C3                                RET

00DA                        OPERACOES       ENDP


                            ;       OBTEM NUMERO DA LINHA DE COMANDO
                            ;
                            ;       <E>     SI - PROXIMO CARACTER A TRATAR
                            ;               CX - # DE CARACTERES POR TRATAR
                            ;
                            ;       <S>     SI - PROXIMO CARACTER A TRATAR
                            ;               CX - # DE CARACTERES POR TRATAR
                            ;               CY = 1 SE ERRO
                            ;                    0 SE ACHOU, AX CONTEM O NUMERO

00DA                        LE_NUMERO       PROC    NEAR

00DA  E3 78                             JCXZ    NUM_INVALIDO            ;ERRO SE FIM DA LINHA

00DC  33 DB                             XOR     BX,BX                   ;BH = 0 SE NUMERO POSITIVO
                                                                        ;BL = 0 SE NAO ACHOU NUMERO

00DE  AC                                LODSB                           ;PEGA PROXIMO CARACTER
00DF  3C 2B                             CMP     AL,'+'                  ;SINAL "+" ?
00E1  74 06                             JE      SINAL_OK                ;SIM: DESVIA
00E3  3C 2D                             CMP     AL,'-'                  ;NAO: SINAL "-" ?
00E5  75 06                             JNE     SINAL_TRATADO           ;     NAO: SEM SINAL
00E7  F6 D3                             NOT     BL                      ;     SIM: NUMERO NEGATIVO
00E9                        SINAL_OK:
00E9  49                                DEC     CX                      ;TRATOU MAIS UM CARACTER
00EA  E3 68                             JCXZ    NUM_INVALIDO            ;ERRO SE FIM DA LINHA
00EC  AC                                LODSB                           ;PEGA PROXIMO CARACTER
00ED                        SINAL_TRATADO:
00ED  33 FF                             XOR     DI,DI                   ;DI = RESULTADO
00EF  3C 23                             CMP     AL,'#'                  ;COMECA COM "#" ?
00F1  74 34                             JE      LE_DECIMAL              ;SIM: E' NUMERO DECIMAL
```

```
----- Objeto ----------      ------- Programa Fonte ----------------------------------------------------

00F3                        LE_DIG_HEXA:
00F3  3C 61                             CMP     AL,'a'
00F5  72 06                             JB      NAO_MINUSCULA           ;SE MAIOR OU IGUAL QUE "a"
00F7  3C 7A                             CMP     AL,'z'                  ;   E
00F9  77 02                             JA      NAO_MINUSCULA           ;MENOR OU IGUAL A "z"
00FB  2C 20                             SUB     AL,'a'-'A'              ;CONVERTE PARA MAIUSCULA
00FD                        NAO_MINUSCULA:
00FD  2C 30                             SUB     AL,'0'
00FF  72 44                             JC      FIM_NUM                 ;FIM SE CARACTER MENOR QUE "0"
0101  3C 0A                             CMP     AL,10                   ;0 A 9 ?
0103  72 08                             JB      DIG_HEX_OK              ;SIM: PRONTO
0105  2C 07                             SUB     AL,'A'-'9'-1            ;NAO: MENOR QUE "A" ?
0107  72 3C                             JC      FIM_NUM                 ;     SIM: FIM DO NUMERO
0109  3C 0F                             CMP     AL,0FH                  ;     NAO: MAIOR QUE "F" ?
010B  77 38                             JA      FIM_NUM                 ;          SIM: FIM DO NUMERO
010D                        DIG_HEX_OK:
010D  F7 C7 F000                        TEST    DI,0F000H               ;OS 4 BITS + SIGNIF. SAO ZERO ?
0111  75 41                             JNZ     NUM_INVALIDO            ;NAO: "ESTOUROU"

0113  B7 FF                             MOV     BH,0FFH                 ;ENCONTROU UM DIGITO
0115  D1 E7                             SAL     DI,1                    ;MULTIPLICA DI POR 16
0117  D1 E7                             SAL     DI,1                    ;(PENA QUE CL ESTA' OCUPADO...)
0119  D1 E7                             SAL     DI,1
011B  D1 E7                             SAL     DI,1
011D  32 E4                             XOR     AH,AH
011F  03 F8                             ADD     DI,AX                   ;COLOCA O NOVO DIGITO

0121  AC                                LODSB                           ;PEGA O PROXIMO
0122  E2 CF                             LOOP    LE_DIG_HEXA             ;REPETE SE NAO CHEGOU AO FIM

0124  EB 1F 90                          JMP     FIM_NUM                 ;FIM DA LINHA
```

```
----- Objeto ----------      ------- Programa Fonte ----------------------------------------------------

0127                        LE_DECIMAL:
0127  4A                             DEC     DX              ;DESCONTA O "#"
0128  E3 2A                          JCXZ    NUM_INVALIDO    ;ERRO SE FIM DA LINHA
012A                        LE_DIG_DEC:
012A  AC                             LODSB                   ;PEGA PROXIMO CARACTER
012B  2C 30                          SUB     AL,'0'          ;MENOR QUE "0" ?
012D  72 16                          JC      FIM_NUM         ;SIM: FIM DO NUMERO
012F  3C 09                          CMP     AL,9            ;NAO: MAIOR QUE "9" ?
0131  77 12                          JA      FIM_NUM         ;     SIM: FIM DO NUMERO
0133  B7 FF                          MOV     BH,0FFH         ;     NAO: ACHOU DIGITO
0135  32 E4                          XOR     AH,AH           ;          AX = DIGITO
0137  97                             XCHG    AX,DI
0138  BA 000A                        MOV     DX,10
013B  F7 EA                          IMUL    DX              ;          AX = DI*10
013D  70 15                          JO      NUM_INVALIDO    ;          ERRO SE "ESTOUROU"
013F  03 F8                          ADD     DI,AX           ;          SOMA O NOVO DIGITO
0141  70 11                          JO      NUM_INVALIDO    ;          ERRO SE "ESTOUROU"
0143  E2 E5                          LOOP    LE_DIG_DEC      ;          REPETE SE NAO ACABOU
0145                        FIM_NUM:
0145  0A FF                          OR      BH,BH           ;ACHOU PELO MENOS UM DIGITO?
0147  74 0B                          JZ      NUM_INVALIDO    ;NAO: ERRO

0149  8B C7                          MOV     AX,DI           ;AX = NUMERO
014B  0A DB                          OR      BL,BL           ;NEGATIVO ?
014D  74 02                          JZ      PRONTO          ;NAO: PRONTO
014F  F7 D8                          NEG     AX              ;SIM: NEGA
0151                        PRONTO:
0151  4E                             DEC     SI              ;SI APONTA P/ O ULTIMO CARACTER
                                                             ; NAO TRATADO
0152  F8                            CLC                     ;FIM NORMAL
0153  C3                            RET

0154                        NUM_INVALIDO:
0154  F9                            STC                     ;INDICA ERRO
0155  C3                            RET

0156                        LE_NUMERO       ENDP
```

```
----- Objeto ----------      ------- Programa Fonte ----------------------------------------------------

                            ;       MOSTRA O RESULTADO EM HEXA E EM DECIMAL
                            ;
                            ;       <E>     AX = RESULTADO


0156                        MOSTRA_RESULTADO PROC   NEAR

0156  50                                     PUSH    AX              ;SALVA O RESULTADO
0157  B2 20                                  MOV     DL,' '
0159  E8 01D3 R                              CALL    MOSTRA_CARACTER ;COLOCA UM ESPACO
015C  58                                     POP     AX              ;RECUPERA RESULTADO

015D  50                                     PUSH    AX              ;SALVA O RESULTADO
015E  E8 016B R                              CALL    MOSTRA_HEXA     ;MOSTRA EM HEXADECIMAL
0161  B2 20                                  MOV     DL,' '
0163  E8 01D3 R                              CALL    MOSTRA_CARACTER ;COLOCA UM ESPACO
0166  58                                     POP     AX              ;RECUPERA RESULTADO

0167  E8 018A R                              CALL    MOSTRA_DECIMAL  ;MOSTRA EM DECIMAL

016A  C3                                     RET


016B                        MOSTRA_HEXA     PROC    NEAR

016B  B9 0004                                MOV     CX,4            ;NUMERO DE DIGITOS
016E                        MOS_DIG_HEX:
016E  D1 C0                                  ROL     AX,1            ;MOVE OS 4 BITS + SIGNIFICATIVOS
0170  D1 C0                                  ROL     AX,1            ; PARA OS 4 BITS - SIGNIFICATIVOS
0172  D1 C0                                  ROL     AX,1
0174  D1 C0                                  ROL     AX,1
0176  50                                    PUSH    AX              ;SALVA O NUMERO RODADO

0177  24 0F                                  AND     AL,0FH          ;ISOLA OS 4 BITS DE INTERESSE
0179  04 30                                  ADD     AL,'0'
017B  3C 39                                  CMP     AL,'9'          ;"0" A "9" ?
017D  76 02                                  JBE     HEX_OK          ;SIM: PRONTO
017F  04 07                                  ADD     AL,'A'-'9'-1    ;NAO: ACERTA "A" A "F"
0181                        HEX_OK:
0181  8A D0                                  MOV     DL,AL
0183  E8 01D3 R                              CALL    MOSTRA_CARACTER ;MOSTRA O DIGITO

0186  58                                     POP     AX              ;RECUPERA O NUMERO
0187  E2 E5                                  LOOP    MOS_DIG_HEX     ;REPETE P/ OS QUATRO DIGITOS

0189  C3                                     RET

018A                        MOSTRA_HEXA     ENDP
```

```
----- Objeto ----------      ------- Programa Fonte ----------------------------------------------------

018A                        MOSTRA_DECIMAL  PROC    NEAR

018A  0B C0                             OR      AX,AX           ;NUMERO POSITIVO ?
018C  79 09                             JNS     SINAL_MOSTRADO  ;SIM: NAO MOSTRA SINAL

018E  50                                PUSH    AX              ;NAO:
018F  B2 2D                             MOV     DL,'-'
0191  E8 01D3 R                         CALL    MOSTRA_CARACTER ;     MOSTRA SINAL "-"
0194  58                                POP     AX
0195  F7 D8                             NEG     AX
0197                        SINAL_MOSTRADO:
0197  B9 2710                           MOV     CX,10000        ;VAMOS DIVIDIR POR 10000,1000,100 E 10
019A                        MOS_DIG_DEC:
019A  33 D2                             XOR     DX,DX           ;ZERA "ESTENSAO" (AX E' POSITIVO)
019C  F7 F1                             DIV     CX              ;AX = QUOCIENTE, DX = RESTO
019E  52                                PUSH    DX              ;SALVA O RESTO
019F  51                                PUSH    CX              ;SALVA O DIVISOR

01A0  04 30                             ADD     AL,'0'          ;CONVERTE EM CARACTER
01A2  8A D0                             MOV     DL,AL
01A4  E8 01D3 R                         CALL    MOSTRA_CARACTER ;MOSTRA O DIGITO

01A7  58                                POP     AX
01A8  2B D2                             SUB     DX,DX
01AA  B9 000A                           MOV     CX,10
01AD  F7 F1                             DIV     CX
01AF  8B C8                             MOV     CX,AX           ;CX = CX / 10

01B1  58                                POP     AX              ;NOVO DIVIDENDO E' O RESTO
01B2  83 F9 01                          CMP     CX,1            ;JA' DIVIDIU POR 10 ?
01B5  75 E3                             JNE     MOS_DIG_DEC     ;NAO: REPETE

01B7  04 30                             ADD     AL,'0'          ;ACERTA O ULTIMO DIGITO
01B9  8A D0                             MOV     DL,AL
01BB  E8 01D3 R                         CALL    MOSTRA_CARACTER ;MOSTRA O DIGITO

01BE  C3                                RET

01BF                        MOSTRA_DECIMAL  ENDP

01BF                        MOSTRA_RESULTADO ENDP
```

```
----- Objeto ----------      ------- Programa Fonte ----------------------------------------------------

                             ;       ROTINAS DE ENTRADA / SAIDA
                             ;       (AS ROTINAS ABAIXO SAO PARA O SISTEMA OPERACIONAL MSDOS,
                             ;        E PODEM PRECISAR DE ALTERACOES EM OUTROS SISTEMAS)


                             ;       LE UMA LINHA DO CONSOLE
                             ;
                             ;       <E>     DS:DX - AREA COM O SEGUINTE FORMATO:
                             ;                       1o BYTE CONTEM O # MAXIMO DE CARACTERES A LER
                             ;                       2o BYTE CONTERA O NUMERO DE CARACTERES LIDOS
                             ;                       3o BYTE EM DIANTE CONTERA OS CARACTERES LIDOS

01BF                         LE_LINHA        PROC    NEAR

01BF  B4 0A                                  MOV     AH,0AH
01C1  CD 21                                  INT     21H             ;CHAMA A FUNCAO 0AH DO MSDOS

01C3  B2 0D                                  MOV     DL,CR
01C5  E8 01D3 R                              CALL    MOSTRA_CARACTER
01C8  B2 0A                                  MOV     DL,LF
01CA  E8 01D3 R                              CALL    MOSTRA_CARACTER ;MUDA DE LINHA
01CD  C3                                     RET

01CE                         LE_LINHA        ENDP


                             ;       MOSTRA MENSAGEM NO CONSOLE
                             ;
                             ;       <E>     DS:DX - INICIO DA MENSAGEM, QUE DEVE SER TERMINADA
                             ;                       POR UM "$"

01CE                         MOSTRA_MENSAGEM PROC    NEAR

01CE  B4 09                                  MOV     AH,09H
01D0  CD 21                                  INT     21H             ;CHAMA A FUNCAO 09H DO MSDOS
01D2  C3                                     RET

01D3                         MOSTRA_MENSAGEM ENDP


                             ;       MOSTRA CARACTER NO CONSOLE
                             ;
                             ;       <E>     DL - CARACTER A MOSTRA

01D3                         MOSTRA_CARACTER PROC    NEAR

01D3  B4 02                                  MOV     AH,02H
01D5  CD 21                                  INT     21H             ;CHAMA A FUNCAO 02H DO MSDOS
01D7  C3                                     RET

01D8                         MOSTRA_CARACTER ENDP


01D8                         CODIGO          ENDS


                                             END     INICIO
```

APÊNDICE - INSTRUÇÕES X CÓDIGOS

As instruções do 8088 correspondem a um, dois, três, quatro, cinco ou até seis bytes, organizados da seguinte forma:

Instrução	1 ou 2 bytes
Dados	1 ou 2 bytes
Endereço	1, 2 ou 4 bytes

O segundo byte da instrução, se existir, segue sempre o seguinte formato:

mod xxx r/m

onde

mod indica de onde é obtido o operando:

00	memória, deslocamento zero (*)
01	memória, deslocamento armazenado em um byte, estendido com sinal para 16 bits
10	memória, deslocamento armazenado em um word
11	registrador (r/m = reg)

r/m seleciona o modo de endereçamento ou o registrador:

se memória

000	BX + SI + deslocamento
001	BX + DI + deslocamento
010	BP + SI + deslocamento
011	BP + DI + deslocamento
100	SI + deslocamento
101	DI + deslocamento
110	BP + deslocamento (*)
111	BX + deslocamento

(*) se mod=00 e r/m=110 então o deslocamento do operando é o contido na instrução, em 2 bytes.

se registrador

16 bits	8 bits	segmento
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
000 SP	000 AH	
001 BP	001 CH	
010 SI	010 DH	
010 DI	010 BH	

xxx varia conforme a instrução

Na descrição são também utilizados os seguintes símbolos:

w 0 operação de 8 bits

1 operação de 16 bits

S se w = 1 0 dois bytes de operando imediato

1 um byte de operando imediato, estendido com sinal para 16 bits

se w = 0 não importa

Alguns exemplos:

MOV reg,reg	**1000100w**	**mod reg r/m**		
MOV AX,BX	1000100111	011 000		
MOV reg,imed	**100000sw**	**mod 000 r/m**		
MOV BX,1	10000011	11 000 011	00000001	
MOV BX,256	10000001	11 000 011	00000000	00000001
MOV reg,mem	**1000101w**	**mod reg r/m**		
MOV AL,[5]	10001010	00 000 110	00000101	00000000
MOV AL,[BX]	10001010	00 000 111		
MOV AL,[BX+5]	10001010	01 000 111	00000101	

AAA		00110111
AAD		11010101 00001010
AAM		11010100 00001010
AAS		00111111
ADC	reg,reg	000100dw 11 reg reg
	mem,reg	0001000w mod reg r/m
	reg,mem	0001001w mod reg r/m
	reg,imed	100000sw 11 010 reg
	mem,imed	100000sw mod 010 r/m
	AL,imed	00010100
	AX,imed	00010101
ADD	reg,reg	000000dw 11 reg reg
	mem,reg	0000000w mod reg r/m
	reg,mem	0000001w mod reg r/m
	reg,imed	100000sw 11 000 reg
	mem,imed	100000sw mod 000 r/m
	AL,imed	00000100
	AX,imed	00000101
AND	reg,reg	001000dw 11 reg reg
	mem,reg	0010000w mod reg r/m
	reg,mem	0010001w mod reg r/m

```
          reg,imed        100000sw    11 100 reg
          mem,imed        100000sw   mod 100 r/m
          AL,imed         00100100
          AX,imed         00100101
-------------------------------------------------------------

C   CALL  rótulo near     11101000
          rótulo far      10011010
          reg             11111111    11 010 reg
          mem word        11111111   mod 010 r/m
          mem dword       11111111   mod 011 r/m
    CBW                   10011000
    CLC                   11111000
    CLD                   11111100
    CLI                   11111010
    CMC                   11110101
    CMP   reg,reg         001110dw    11 reg reg
          mem,reg         0011100w   mod reg r/m
          reg,mem         0011101w   mod reg r/m
          reg,imed        100000sw    11 111 reg
          mem,imed        100000sw   mod 111 r/m
          AL,imed         00111100
          AX,imed         00111101
    CMPS                  1010011w
    CWD                   10011001
-------------------------------------------------------------

D   DAA                   00100111
    DAS                   00111111
    DEC   reg             1111111w    11 001 reg
          mem             1111111w   mod 001 r/m
          regword         01001reg
    DIV   reg             1111011w    11 110 reg
          mem             1111011w   mod 110 r/m

-------------------------------------------------------------

E   ESC   imed            11011iii   mod iii r/m
-------------------------------------------------------------

H   HLT                   11110100
-------------------------------------------------------------

I   IDIV  reg             1111011w    11 111 reg
          mem             1111011w   mod 111 r/m
    IMUL  reg             1111011w    11 101 reg
          mem             1111011w   mod 101 r/m
    IN    AL,imed         11100100
          AX,imed         11100101
          AL,DX           11101100
          AX,DX           11101101
    INC   reg             1111111w    11 000 reg
          mem             1111111w   mod 000 r/m
```

```
        regword         01000reg
    INT 3               11001100
        imed            11001101
    INTO                11001110
    IRET                11001111
-------------------------------------------------------------

J   JA                  01110111
    JAE                 01110011
    JB                  01110010
    JBE                 01110110
    JCXZ                11100011
    JE                  01110100
    JG                  01111111
    JGE                 01111101
    JL                  01111100
    JLE                 01111110
    JMP rótulo short    11101011
        rótulo near     11101001
        rótulo far      11101010
        reg             11111111    11 100 reg
        mem word        11111111   mod 100 r/m
        mem dword       11111111   mod 101 r/m
    JNA                 01110110
    JNAE                01110010
    JNB                 01110011
    JNBE                01110111
    JNE                 01110101
    JNG                 01111110
    JNGE                01111100
    JNL                 01111101
    JNLE                01111111
    JNO                 01110001
    JNP                 01111011
    JNS                 01111001
    JNZ                 01110101

    JO                  01110000
    JP                  01111010
    JPE                 01111010
    JPO                 01111011
    JS                  01111000
    JZ                  01110100
-------------------------------------------------------------

L   LAHF                10011111
    LDS   reg,mem       11000101   mod reg r/m
    LEA   reg,mem       10001101   mod reg r/m
    LES   reg,mem       11000100
    LOCK                11110000
    LODS                1010110w
```

```
    LOOP                11100010
    LOOPE               11100001
    LOOPNE              11100000
    LOOPNZ              11100000
    LOOPZ               11100001
-------------------------------------------------------------

M   MOV   reg,reg       100010dw    11 reg reg
          reg,mem       1000101w   mod reg r/m
          mem,reg       1000100w   mod reg r/m
          reg,imed      1100011w    11 000 reg
          mem,imed      1100011w   mod 000 r/m
          reg,imed      1011wreg
          AL,mem        10100000  (endereco direto)
          AX,mem        10100001  (endereco direto)
          segreg,reg    10001110    11 0rg reg
          segreg,mem    10001110   mod 0rg r/m
          reg,segreg    10001100    11 0rg reg
          mem,segreg    10001100   mod 0rg r/m
    MOVS                1010010w
    MUL   reg           1111011w    11 100 reg
          mem           1111011w   mod 100 r/m
-------------------------------------------------------------

N   NEG   reg           1111011w    11 011 reg
          mem           1111011w   mod 011 r/m
    NOP                 10010000
    NOT   reg           1111011w    11 010 reg
          mem           1111011w   mod 010 r/m
 -------------------------------------------------------------

O   OR    reg,reg       000010dw    11 reg reg
          mem,reg       0000100w   mod reg r/m
          reg,mem       0000101w   mod reg r/m
          reg,imed      100000sw    11 001 reg
          mem,imed      100000sw   mod 001 r/m
          AL,imed       00001100
          AX,imed       00001101
    OUT   imed,AL       11100110
          imed,AX       11100111
          DX,AL         11101110
          DX,AX         11101111

-------------------------------------------------------------

P   POP   reg           10001111    11 000 reg
          mem           10001111   mod 000 r/m
          regword       01011reg
          segreg        000rg111
    POPF                10011101
    PUSH  reg           11111111    11 110 reg
          mem           11111111   mod 110 r/m
```

```
          regword         01010reg
          segreg          000rg110
    PUSHF                 10011100
-------------------------------------------------------------

R   RCL   reg,1           1101000w    11 010 reg
          mem,1           1101000w   mod 010 r/m
          reg,CL          1101001w    11 010 reg
          mem,CL          1101001w   mod 010 r/m
    RCR   reg,1           1101000w    11 011 reg
          mem,1           1101000w   mod 011 r/m
          reg,CL          1101001w    11 011 reg
          mem,CL          1101001w   mod 011 r/m
    REP                   11110011
    REPE                  11110011
    REPNE                 11110010
    REPNZ                 11110010
    REPZ                  11110011
    RET   near            11000011
          imed near       11000010
          far             11001011
          imed far        11001010
    ROL   reg,1           1101000w    11 000 reg
          mem,1           1101000w   mod 000 r/m
          reg,CL          1101001w    11 000 reg
          mem,CL          1101001w   mod 000 r/m
    ROR   reg,1           1101000w    11 001 reg
          mem,1           1101000w   mod 001 r/m
          reg,CL          1101001w    11 001 reg
          mem,CL          1101001w   mod 001 r/m
-------------------------------------------------------------

S   SAHF                  10011110
    SAL   reg,1           1101000w    11 100 reg
          mem,1           1101000w   mod 100 r/m
          reg,CL          1101001w    11 100 reg
          mem,CL          1101001w   mod 100 r/m
    SAR   reg,1           1101000w    11 111 reg
          mem,1           1101000w   mod 111 r/m
          reg,CL          1101001w    11 111 reg
          mem,CL          1101001w   mod 111 r/m
    SBB   reg,reg         000110dw    11 reg reg
          mem,reg         0001100w   mod reg r/m
          reg,mem         0001101w   mod reg r/m
          reg,imed        100000sw    11 011 reg
          mem,imed        100000sw   mod 011 r/m
          AL,imed         00011100
          AX,imed         00011101

    SCAS                  1010111w
    SHL   reg,1           1101000w    11 100 reg
```

```
          mem,1           1101000w   mod 100 r/m
          reg,CL          1101001w    11 100 reg
          mem,CL          1101001w   mod 100 r/m
    SHR   reg,1           1101000w    11 101 reg
          mem,1           1101000w   mod 101 r/m
          reg,CL          1101001w    11 101 reg
          mem,CL          1101001w   mod 101 r/m
    STC                   11111001
    STD                   11111101
    STI                   11111011
    STOS                  1010101w
    SUB   reg,reg         001010dw    11 reg reg
          mem,reg         0010100w   mod reg r/m
          reg,mem         0010101w   mod reg r/m
          reg,imed        100000sw    11 101 reg
          mem,imed        100000sw   mod 101 r/m
          AL,imed         00101100
          AX,imed         00101101
-------------------------------------------------------------
T   TEST  reg,reg         1000010w    11 reg reg
          mem,reg         1000010w   mod reg r/m
          reg,mem         1000010w   mod reg r/m
          reg,imed        1111011w    11 000 reg
          mem,imed        1111011w   mod 000 r/m
          AL,imed         10101000
          AX,imed         10101001
-------------------------------------------------------------
W   WAIT                  10011011
-------------------------------------------------------------
X   XCHG  reg,reg         1000011w    11 reg reg
          reg,mem         1000011w   mod reg r/m
          mem,reg         1000011w   mod reg r/m
          AX,reg          100010rg
    XLAT                  11010111
    XOR   reg,reg         001100dw    11 reg reg
          mem,reg         0011000w   mod reg r/m
          reg,mem         0011001w   mod reg r/m
          reg,imed        100000sw    11 110 reg
          mem,imed        100000sw   mod 110 r/m
          AL,imed         00110100
          AX,imed         00110101
```