

A LINGUAGEM ASSEMBLY DO 8086

- ◎ Para introduzir a linguagem Assembly do 8086 vamos utilizar o programa exemplo.asm.
- Trata-se de um programa completo, embora muito simples, que tem como objectivo multiplicar dois números de 16 bits, dando como resultado um número de 32 bits que é guardado em memória.

EXEMPLO.ASM

```
PILHA SEGMENT PARA STACK 'STACK'
    DB      1024      DUP('PILHA')
PILHA ENDS
DADOS SEGMENT PARA 'DATA'
    MULT1   DW 204Ah
    MULT2   DW 382Ah
    PRODUTO DW 2 DUP(0)
DADOS ENDS
CODIGO SEGMENT PARA 'CODE'
    ASSUME CS:CODIGO, DS:DADOS, SS:PILHA
INICIO:
    MOV AX, DADOS
    MOV DS, AX
    MOV AX, MULT1
    MUL MULT2
    MOV PRODUTO, AX
    MOV PRODUTO+2, DX
    MOV AH, 4CH
    INT 21H
CODIGO ENDS
END INICIO
```

INSTRUÇÕES DE INICIAÇÃO

- ◎ Como na maioria das linguagens de programação existem na linguagem Assembly um conjunto de instruções que é necessário incluir antes de se começar a escrever o programa propriamente dito, de forma a permitir a sua correcta execução.
- ◎ Essas instruções são habitualmente designadas de Instruções de Iniciação.

SEGMENTOS DE UM PROGRAMA

- ◎ Um programa em Assembly possui 3 zonas principais de memória, às quais correspondem 3 segmentos:
 - Segmento de dados (Data Segment)
 - Zona onde são colocados os dados usados no programa.
 - Segmento de código (Code Segment)
 - Zona destinada ao código/instruções do programa.
 - Segmento de pilha (Stack Segment)
 - Zona destinada a:
 - armazenamento dos endereços de retorno aquando da chamada de procedimentos;
 - armazenamento temporário de dados.

AS DIRECTIVAS SEGMENT, ENDS E END

- ◎ As directivas SEGMENT e ENDS são usadas para identificar o grupo de dados ou de instruções que se pretende que façam parte do mesmo segmento.
- ◎ A directiva END indica o términos do programa. Quaisquer instruções depois desta directiva são ignoradas.

A DIRECTIVA SEGMENT

- ◎ A directiva segment permite a especificação de um parâmetro opcional (PARA) que possibilita a definição do alinhamento dos dados em memória.
- ◎ Esse alinhamento pode ser um dos seguintes: byte, word, dword, para ou page.
 - Estas palavras-chave instruem o Assembler, o linker e o DOS a efectuar o carregamento de segmentos em múltiplos de bytes (8 bits), words (16 bits), double words (32 bits), parágrafos (16 bytes) ou páginas (256 bytes).

A DIRECTIVA SEGMENT

- ◎ Caso nenhuma das palavras-chave mencionadas surja como parâmetro da directiva segment, o alinhamento utilizado por default é o parágrafo.
- Os registos de segmento do 8086 apontam sempre para endereços de parágrafos, como tal fica facilitado o acesso caso o alinhamento especificado seja o parágrafo.

A DIRECTIVA SEGMENT

- ◎ Um programa em linguagem Assembly pode ter vários segmentos de dados e vários segmentos de código, mas apenas pode aceder a um de cada tipo em cada momento.
- ◎ O operando final da directiva segment é um tipo de classe. Este tipo especifica o ordenamento dos segmentos.
 - Esse operando consiste num símbolo delimitado por apóstrofes. Geralmente utilizam-se os seguintes nomes: CODE, DATA e STACK.

A DIRECTIVA SEGMENT

- ◎ O ordenamento é feito da seguinte forma:
 - O Assembler localiza o primeiro segmento no ficheiro.
 - Concatena todos os outros segmentos que são do mesmo tipo, no final desse segmento.
 - Após processar todos os segmentos do mesmo tipo, é pesquisado novamente o ficheiro fonte, no que respeita aos restantes segmentos, repetindo todo o processo.

A DIRECTIVA ASSUME

- ◎ A directiva ASSUME indica ao assembler quais os valores a assumir para um conjunto de registos de segmento, ou seja, quais dos segmentos lógicos correspondem a cada um dos segmentos mencionados.

INICIAÇÃO DOS REGISTOS DE SEGMENTO

- ◎ Uma das iniciações que necessita de ser sempre feita é a dos registos de segmento.
 - Estes registos precisam de ser carregados com os endereços de memória onde se pretende que os segmentos comecem.
 - Esta iniciação é feita automaticamente para os registos CS e SS.

INICIAÇÃO DOS REGISTOS DE SEGMENTO

- A iniciação do registo DS é efectuada conforme mostrado no programa [exemplo.asm](#), através das instruções
 - MOV AX, DADOS
 - MOV DS, AX.
- As variáveis globais são declaradas no segmento de dados.
- A declaração de uma variável é efectuada através da atribuição de um nome e indicação do seu tipo.

NOMES PARA OS DADOS

- ◎ Após a declaração de uma variável, é possível a sua referência pelo seu nome em vez de pela sua localização no programa.
- ◎ A 1^a variável colocada no segmento de dados obtém armazenamento colocado na localização DS:0.
- O MASM tem o cuidado de alocar variáveis de tal forma que elas não se sobreponham.

TIPOS DOS DADOS

- ◎ Simples ou escalares
 - Byte
 - Word
 - Dword
 - Fword, Qword e Tbyte
 - Real4, Real8 e Real10
- ◎ Compostos
 - Arrays
 - Estruturas

DECLARAÇÃO DE VARIÁVEIS - BYTE

- ◎ Existem 3 formatos:
 - Identificador db ?
 - Identificador byte ? → Para variáveis sem sinal
 - Identificador $\in [0, 255]$
 - Identificador sbyte ? → Para variáveis com sinal
 - Identificador $\in [-128, 127]$

DECLARAÇÃO DE VARIÁVEIS - BYTE

Ex:

- NoSignedByte db ?
- UnSignedByte byte ?
- SignedByte sbyte ?
- :

○ O ‘?’ indica que a variável não é iniciada. Para que uma variável seja iniciada dever-se-á usar o formato exemplificado a seguir:

- NoSignedByte db 0
- UnSignedByte byte -5 Atenção!
- SignedByte sbyte -1

○ É da responsabilidade do programador assegurar o uso correcto das variáveis.

DECLARAÇÃO DE VARIÁVEIS - WORD

- ◎ Existem 3 formatos:
 - Identificador dw ?
 - Identificador word ? → Para variáveis sem sinal
 - Identificador $\in [0, 65\,535]$
 - Identificador sword ? → Para variáveis com sinal
 - Identificador $\in [-32\,768, 32\,767]$

DECLARAÇÃO DE VARIÁVEIS - WORD

- Ex:

- NoSignedWord dw ?
- UnSignedWord word ?
- SignedWord sword ?

- O ‘?’ indica que a variável não é inicializada. Para que uma variável seja inicializada dever-se-á usar o formato exemplificado a seguir:

- NoSignedWord dw 65535
- UnSignedWord word 0
- SignedWord sword -1

DECLARAÇÃO DE VARIÁVEIS - DWORD

- ◎ Existem 3 formatos:
 - Identificador dd ?
 - Identificador dword ? → Para variáveis sem sinal
 - Identificador $\in [0, 4\ 294\ 967\ 295]$
 - Identificador sdword ? → Para variáveis com sinal
 - Identificador $\in [-2\ 147\ 483\ 648, 2\ 147\ 483\ 647]$

DECLARAÇÃO DE VARIÁVEIS - DWORD

- Ex:

- NoSignedDWord dd ?
- UnsignedDword dword ?
- SignedDWord sdword ?

- O ‘?’ indica que a variável não é inicializada. Para que uma variável seja inicializada dever-se-á usar o formato exemplificado a seguir:

- NoSignedDWord dd 4000000000
- UnSignedDWord dword 255
- SignedDWord sdword -1

DECLARAÇÃO DE VARIÁVEIS

- ◎ Para trabalhar com variáveis de maiores dimensões usam-se os formatos:
 - df/fword (6 bytes)
 - dq/qword (8 bytes)
 - dt/tbyte (10 bytes)

NOMES PARA OS DADOS

- ◎ No programa exemplo.asm as instruções:
 - MULT1 DW 204AH declara uma variável do tipo word e inicia-a com o valor 204AH.
 - MULT2 DW 382AH declara uma variável também do tipo word e inicia-a com o valor 382AH.
 - PRODUTO DW 2 DUP (0) guarda espaço para duas words em memória, dá ao endereço do início da primeira word o nome PRODUTO, e inicia as duas words com o valor zero.

DECLARAÇÃO DE VARIÁVEIS DE VÍRGULA FLUTUANTE

- ◎ REAL4 (4 bytes)
- ◎ REAL8 (8 bytes)
- ◎ REAL10 (10 bytes)

- Ex.:

○	X	REAL4	1.0
○	Y	REAL8	1.0e-25
○	Z	REAL10	-1.2594e+10

- ◎ Atenção: O operando tem de conter uma constante de vírgula flutuante válida usando a notação científica ou decimal. Variáveis do tipo REAL não aceitam inteiros puros.

DEFINIÇÃO DE NOMES PARA OS TIPOS DE VARIÁVEIS COM TYPEDEF

- ◎ Suponhamos que não gostamos dos nomes usados para declarar as variáveis e que preferimos as convenções de nomes usadas em C ou Pascal como integer, float, double, char,...
- ◎ O MASM possui a declaração typedef que permite criar aliases para os nomes.

DEFINIÇÃO DE NOMES PARA OS TIPOS DE VARIÁVEIS COM TYPEDEF

Ex.:

- int **typedef** sword
- char **typedef** byte
- float **typedef** real4
- :
- i int ?
- ch char ?
- x float ?

TIPOS DE DADOS PONTEIROS

- ◎ Ponteiro: localização de memória cujo valor é o endereço (índice) de outra localização de memória.
- ◎ A família 80x86 suporta 2 tipos de ponteiros:
 - Near: valor de 16 bits que fornece um offset dentro de um segmento (normalmente o data segment).
 - Far: valor de 32 bits para um Segment:Offset de um objecto. O offset é carregado para BX, BP, SI, ou DI; o Segment é carregado para um registo de segmento (normalmente o ES).

PONTEIROS NEAR

Ex:

MOV P,1000H

MOV BX,P

MOV AX,[BX]

↔

MOV AL,[1000H]

MOV AH,[1001H]

Desvantagem: endereçamento limitado a 64k!

PONTEIROS FAR

- ◎ Os dados são acedidos utilizando os modos de endereçamento indirecto por registo.
- ◎ Ex: Para armazenar o valor de AL no byte apontado por P
 - LES BX, P
 - MOV ES:[BX], AL

TIPOS DE DADOS PONTEIROS

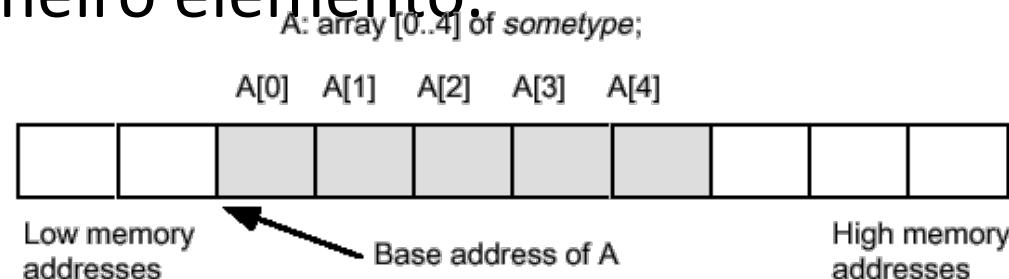
- ◎ Uma vez que os ponteiros near são de 16 bits e os ponteiros far são de 32 bits, podem-se usar dw/word e dd/dword para declarar ponteiros near e far, respectivamente.
 - Não se deve usar sword nem sdword porque os ponteiros são sem sinal.
- ◎ Formato mais usual para declarar ponteiros:
 - `typename typedef near ptr basetype`
 - `typename typedef far ptr basetype`
 - Typename representa o nome do novo tipo que se pretende criar.
 - Basetype é o nome do tipo para o qual se pretende criar o ponteiro.

TIPOS DE DADOS PONTEIROS

 Exs.:

- nbytptr `typedef near ptr byte`
 - fbytptr `typedef far ptr byte`
 - wptr `typedef near ptr word`
 - `:`
 - bytestr `nbytptr ?`
 - bytestr2 `fbytptr ?`
 - Mystring `db'Tecnologia da Informática'`
 - Strstr `nbytptr Mystring`

- ◎ Arrays são tipos de dados agregados cujos elementos são todos de um determinado tipo. Os seus elementos são acedidos por intermédio de índices inteiros.
- ◎ O endereço base de um array é o endereço do primeiro elemento.



ARRAYS UNIDIMENSIONAIS

- ◎ Formato (todos os elementos iniciados com o mesmo valor):
 - **arrayname basetype n dup (x)**
 - arrayname: nome do array
 - basetype: tipo de cada elemento do array
 - n dup (x) : duplicação do objecto dentro de parêntesis n vezes
- ◎ Exs.:
 - CharArray byte 128 dup (1)
 - ;array [0..127] of byte
 - IntArray word 64 dup (1)
 - ;array [0..63] of word
 - BytArray byte 10 dup (?)
 - ;array [0..9] of byte
 - RealArray real4 8 dup (1.0)
 - ;array [0..7] of real4

ARRAYS UNIDIMENSIONAIS

- ◎ Formato (elementos iniciados com valores diferentes):
 - arrayname basetype v1, v2, v3,..., vn

- ◎ Exs.:
 - Squares byte 0, 1, 4, 9, 16, 25
 - Squares1 byte 0, 1, 4, 9, 16, 25, 36, 49
 byte 64, 81, 100, 121, 144
 - Squares2 byte 0, 1, 4, 9, 16, 25, 36, 64,\
 81, 100, 121, 144, 160, 180,\
 196, 210, 220, 225
 - BigArray word 256 dup (0,1,2,3)
 - ;array[0..1023] of word

ARRAYS MULTIDIMENSIONAIS

- ◎ O seu armazenamento é efectuado através de um mapeamento para um array unidimensional.
- ◎ Existem duas técnicas para manipulação de elementos de um array multidimensional:
 - Row Major Ordering
 - Column Major Ordering

ARRAYS MULTIDIMENSIONAIS

◎ Row Major Ordering

A:array [0..3,0..3] of char;

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Memory

15	A[3,3]
14	A[3,2]
13	A[3,1]
12	A[3,0]
11	A[2,3]
10	A[2,2]
9	A[2,1]
8	A[2,0]
7	A[1,3]
6	A[1,2]
5	A[1,1]
4	A[1,0]
3	A[0,3]
2	A[0,2]
1	A[0,1]
0	A[0,0]

◎ Column Major Ordering

A:array [0..3,0..3] of char;

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Memory

15	A[3,3]
14	A[2,3]
13	A[1,3]
12	A[0,3]
11	A[3,2]
10	A[2,2]
9	A[1,2]
8	A[0,2]
7	A[3,1]
6	A[2,1]
5	A[1,1]
4	A[0,1]
3	A[3,0]
2	A[2,0]
1	A[1,0]
0	A[0,0]

ARRAYS MULTIDIMENSIONAIS

Ex.:

- A1 byte 4 dup (4 dup (?))
- A1 byte 16 dup (?)
 - ;array [0..3, 0..3] of byte
- A2 word 3 dup (4 dup (5 dup (?)))
 - array [0..2, 0..3, 0..4] of word
- A3 word 3 dup (4 dup (5 dup (1)))
 - array [0..2, 0..3, 0..4] of word
- A4 byte 0, 1, 2, 3
 - byte 5, 6, 7, 8
 - byte 9, 6, 3, 1
- array [0..2, 0..3] of byte

CONJUNTO DE INSTRUÇÕES DO 8086

- ◎ Classes de instruções:
 - Instruções de Movimentação de dados:
 - MOV, XCHG, LEA, LDS, LES, LSS, PUSH, POP, PUSHF, POPF
 - Instruções de Conversões de dados:
 - CBW, CWD, XLAT
 - Instruções Aritméticas:
 - ADD, ADC, INC, SUB, SBB, DEC, CMP, NEG, MUL, IMUL, DIV, IDIV

CONJUNTO DE INSTRUÇÕES DO 8086

- Instruções de Deslocamentos:
 - SHL, SAL, SHR, SAR
- Instruções de Rotações:
 - RCL, RCR, ROL, ROR
- Instruções Lógicas:
 - AND, OR, XOR, NOT
- Instruções para manipulação de bits e de flags:
 - TEST, CLC, STC, CMC, CLD, STD, CLI, STI

CONJUNTO DE INSTRUÇÕES DO 8086

- Instruções de I/O:
 - IN, OUT
- Instruções de manipulação de “Strings”:
 - MOVS, LODS, STOS, SCAS, CMPS
- Instruções de Repetição:
 - REP, REPZ, REPE, REPNZ, REPNE
- Instruções de Controlo de Fluxo do programa:
 - JMP, CALL, RET e saltos condicionais

FLAGS

- ◎ FLAGS: registam o modo de operação actual da CPU e algumas informações sobre o estado das instruções.
- ◎ O estado (0 ou 1) das FLAGS Carry, Parity, Zero, Sign e Overflow pode ser verificado através das instruções de salto condicional e das instruções setcc (“Set on Condition”).

FLAGS

- ◎ Algumas instruções aritméticas, lógicas e outras afectam a flag Overflow.
 - Ex: Após uma operação aritmética esta flag contém o valor 1 se o resultado não couber no operando destino (com sinal), caso contrário conterá o valor 0.
- ◎ As instruções lógicas aplicam-se geralmente a números sem sinal, pelo que colocam a flag Overflow a “0”.

- ◎ As instruções de “Strings” usam a flag Direction.
 - Quando a “0”, a CPU processa os elementos da string do endereço mais baixo para o mais alto.
 - Quando a “1”, a CPU processa os elementos da string do endereço mais alto para o mais baixo.

- ◎ A flag Interrupt enable/disable controla a capacidade do processador responder ou não a eventos externos chamados interrupções.

- ◎ A flag Sign é activada quando o resultado de um cálculo é negativo.

- ◎ A flag Trace activa ou desactiva o modo de trace do processador.
 - Alguns Debuggers usam este bit para activar ou desactivar a operação de trace/ passo-a-passo.
 - Não existem instruções que alterem directamente o valor desta flag.

FLAGS

- ◎ A flag Zero é activada quando o resultado de algumas instruções é zero.
 - Muito utilizada para testar a igualdade de dois valores.
- ◎ A flag Auxiliary Carry serve de apoio a operações em BCD.
- ◎ A flag Parity é activada quando o resultado de uma operação contém um número par de 1's.

- ◎ A flag Carry tem diversos propósitos:
 - ◎ Sinalizar um “overflow” sem sinal (tal como a flag overflow detecta um overflow com sinal);
 - ◎ De grande utilidade para várias operações booleanas, uma vez que facilmente se pode testar, activar, desactivar ou inverter.

INSTRUÇÕES DE MOVIMENTAÇÃO - MOV

- ◎ Copia o valor presente em ORIG para DEST.
- ◎ Formato geral:
 - MOV DEST, ORIG
- ◎ Formatos específicos:
 - MOV REG, REG
 - MOV MEM, REG
 - MOV REG, MEM

INSTRUÇÕES DE MOVIMENTAÇÃO - MOV

- MOV MEM, IMMEDIATE_DATA
- MOV REG, IMMEDIATE_DATA
- MOV AX/AL, MEM
- MOV MEM, AX/AL
- MOV SEGREG, MEM16
- MOV SEGREG, REG16
- MOV MEM16, SEGREG
- MOV REG16, SEGREG

INSTRUÇÕES DE MOVIMENTAÇÃO - MOV

◎ NOTAS:

- Não existe a operação MOV MEM, MEM;
- Não é possível mover dados directamente para registos de segmento.
- Se nenhum dos operandos envolver dados imediatos, ambos os operandos têm de possuir o mesmo tamanho.

INSTRUÇÕES DE MOVIMENTAÇÃO - MOV

NOTAS:

- Se o operando destino for um dado imediato, qual a quantidade de bytes armazenada nas seguintes situações:
 - MOV var,0
 - Depende do tamanho de var!
 - MOV [BX],0?
 - Erro!
 - Solução: MOV byte ptr [BX], 0 ou MOV word ptr [BX], 0.
- A instrução MOV não afecta as flags.

INSTRUÇÕES DE MOVIMENTAÇÃO - XCHG

- ◎ Troca os valores entre OPERAND1 e OPERAND2.
- ◎ Formato geral:
 - XCHG OPERAND1, OPERAND2
- ◎ Formatos específicos:
 - XCHG REG, MEM
 - XCHG REG, REG
 - XCHG AX, REG16
- ◎ Ambos os operandos têm de ter o mesmo tamanho.
- ◎ A instrução XCHG não afecta as flags.

INSTRUÇÕES DE MOVIMENTAÇÃO - LEA

- ◎ “Load Effective Address”: carrega um registo com o endereço efectivo de uma localização da memória.
- ◎ Formato geral:
 - LEA DEST, ORIG
- ◎ Formato específico:
 - LEA REG16, MEM
- ◎ A instrução LEA não afecta as flags.

INSTRUÇÕES DE MOVIMENTAÇÃO - LEA

○ Ex1:

○ LEA AX, DS: [8B67h]

AX

08B67h

○ Ex2:

○ LEA AX, 3 [BX]

↔

○ MOV AX, BX

○ ADD AX, 3

INSTRUÇÕES DE MOVIMENTAÇÃO - LDS, LES, LSS

- ◎ Permitem carregar o par SEGREG:REG16 com uma única instrução.
- ◎ Formato geral ($x \in \{D, E, S\}$):
 - $LxS \quad DEST, ORIG$
 - $ORIG$ é uma variável ponteiro do tipo far.
- ◎ Formatos específicos:
 - $LxS \quad REG16, MEM32$
 \Updownarrow
 - $REG16:=[MEM32], \quad xS:=[MEM32+2]$

INSTRUÇÕES DE MOVIMENTAÇÃO LDS, LES E LSS

- ◎ Estas instruções carregam a double word presente no endereço especificado por MEM32 para o par de registos constituído por REG16 (L. O. word) e um dos registos de segmento DS, ES ou SS (H. O. word).
- ◎ Dada a importância dos registos (DS, ES ou SS) envolvidos nestas instruções, há que haver precaução aquando da sua utilização.
- ◎ As instruções LxS não afectam as flags.

INSTRUÇÕES DE MOVIMENTAÇÃO PUSH, POP, PUSHF E POPF

- ◎ Manipulam dados da pilha.
- ◎ Formatos específicos:
 - PUSH REG16
 - POP REG16
 - PUSH SEGREG
 - POP SEGREG;excepto CS
 - PUSH MEM
 - POP MEM
 - PUSHF
 - POPF

INSTRUÇÕES DE MOVIMENTAÇÃO PUSH, POP, PUSHF E POPF

◎ Pilha:

- Localização: SS.
- Cresce no sentido dos endereços decrescentes.
- SS:SP contém o endereço do elemento do topo da pilha (o último valor colocado na pilha).
- É utilizada essencialmente para:
 - Guardar temporariamente registos e variáveis.
 - Passar parâmetros para um procedimento.
 - Armazenar variáveis locais.

INSTRUÇÕES DE MOVIMENTAÇÃO PUSH, POP, PUSHF E POPF

- ◎ **PUSH OPERANDO16**
 - SP:=SP-2
 - [SS:SP]:=OPERANDO16
- ◎ **POP OPERANDO16**
 - OPERANDO16:=[SS:SP]
 - SP:=SP+2
- ◎ OPERANDO16 pode ser um registo ou memória.
- ◎ As instruções PUSH e POP não afectam as flags, com a excepção óvia de PUSHF e POPF.

INSTRUÇÕES DE CONVERSÕES - CBW E CWD

- ◎ CBW (“Convert Byte to Word”)
 - Faz a extensão com sinal do valor de 8 bits em AL para AX.
 - Formato:
 - CBW
- ◎ CWD (“Convert Word to Double Word”)
 - Faz a extensão com sinal do valor de 16 bits em AX para DX:AX.
 - Formato:
 - CWD
- ◎ As instruções CBW e CWD não afectam as flags.

INSTRUÇÕES DE CONVERSÕES - XLAT

- ◎ Traduz o valor presente em AL, baseando-se numa tabela de endereço BX situada no segmento de dados.
- ◎ Formato:
 - ◎ XLAT
 - temp:= AL+BX
 - AL:=DS:[temp]
 - BX aponta para uma tabela no segmento de dados actual.
 - XLAT substitui o valor em AL pelo byte/elemento de índice AL dentro da tabela apontada por DS:BX.
 - ◎ A instrução XLAT não afecta as flags.

INSTRUÇÕES ARITMÉTICAS - ADD E ADC

- ◎ Adiciona o conteúdo do operando ORIG ao operando DEST.
 - $\text{DEST} := \text{DEST} + \text{ORIG}$
- ◎ Formato geral:
 - ADD DEST, ORIG
 - Ambos os operandos têm de possuir o mesmo tamanho.
- ◎ Formatos específicos:
 - ADD REG, REG
 - ADD REG, MEM
 - ADD MEM, REG

INSTRUÇÕES ARITMÉTICAS - ADD E ADC

- ADD REG, IMMEDIATE_DATA
- ADD MEM, IMMEDIATE_DATA
- ADD AX/AL, IMMEDIATE_DATA
- Os formatos de ADC são idênticos aos de ADD.
- ADC adiciona o conteúdo do operando ORIG ao operando DEST adicionando ainda o valor presente na flag de Carry.
- $\text{DEST} := \text{DEST} + \text{ORIG} + C$

INSTRUÇÕES ARITMÉTICAS - ADD E ADC

- ◎ Ambas as instruções afectam as flags de forma idêntica, activando as seguintes flags:
 - Overflow, significando um overflow aritmético com sinal.
 - Carry, significando um overflow aritmético sem sinal.
 - Sign, significando um resultado negativo.
 - Zero, significando que o resultado da adição é zero.
 - Auxiliary Carry, significando que ocorreu um overflow em BCD no nibble menos significativo.
 - Parity, significando que existe um nº par de bits a “1” no resultado – paridade par.

INSTRUÇÕES ARITMÉTICAS - INC

- ◎ Incrementa o valor de OPERAND de uma unidade.
- ◎ Formato geral:
 - INC OPERAND
- ◎ Formatos específicos:
 - INC MEM
 - INC REG
- ◎ À excepção da flag de Carry que não é afectada, INC activa as flags da mesma forma que a instrução equivalente
 - ADD OPERAND,1

INSTRUÇÕES ARITMÉTICAS - SUB E SBB

- ◎ Subtrai o conteúdo do operando ORIG ao operando DEST.
 - $\text{DEST} := \text{DEST} - \text{ORIG}$
- ◎ Formato geral:
 - SUB DEST, ORIG
 - Ambos os operandos têm de possuir o mesmo tamanho.
- ◎ Formatos específicos:
 - SUB REG, REG
 - SUB REG, MEM
 - SUB MEM, REG

INSTRUÇÕES ARITMÉTICAS - SUB E SBB

- SUB REG, IMMEDIATE_DATA
- SUB MEM, IMMEDIATE_DATA
- SUB AX/AL, IMMEDIATE_DATA
- Os formatos de SBB são idênticos aos de SUB.
- SBB subtrai o conteúdo do operando ORIG ao operando DEST subtraindo ainda o valor presente na flag de Carry.
- $DEST := DEST - ORIG - C$

INSTRUÇÕES ARITMÉTICAS - SUB E SBB

- ◎ Ambas as instruções afectam as flags de forma idêntica, activando as seguintes flags:
 - Overflow, significando um overflow ou underflow aritmético com sinal.
 - Carry, significando um overflow aritmético sem sinal.
 - Sign, significando um resultado negativo.
 - Zero, significando que o resultado é zero.
 - Auxiliary Carry, significando que ocorreu um overflow em BCD no nibble menos significativo.
 - Parity, significando que existe um nº par de bits a “1” no resultado – paridade par.

INSTRUÇÕES ARITMÉTICAS - DEC

- ◎ Decrementa o valor de OPERAND de uma unidade.
- ◎ Formato geral:
 - DEC OPERAND
- ◎ Formatos específicos:
 - DEC MEM
 - DEC REG
 - À excepção da flag de Carry que não é afectada, DEC activa as flags da mesma forma que a instrução equivalente: SUB OPERAND,1

INSTRUÇÕES ARITMÉTICAS - CMP

- ◎ Formato geral:
 - **CMP OPERAND1, OPERAND2**
 - Ambos os operandos têm de possuir o mesmo tamanho.
- ◎ Idêntica à instrução SUB mas com uma diferença crucial – não armazena o resultado da subtração em DEST.
- ◎ Formatos específicos:
 - **CMP REG, REG**
 - **CMP REG, MEM**
 - **CMP MEM, REG**

INSTRUÇÕES ARITMÉTICAS - CMP

- CMP REG, IMMEDIATE_DATA
- CMP MEM, IMMEDIATE_DATA
- CMP AX/AL, IMMEDIATE_DATA
- ◎ Após o cálculo da subtracção entre os operandos, as flags são afectadas em função desse resultado, activando as seguintes flags:
 - Zero, significando que ambos os operandos são iguais.
 - Sign, significando um resultado negativo.

INSTRUÇÕES ARITMÉTICAS - CMP

- Overflow, significando que a diferença entre os operandos produziu um overflow ou underflow aritmético com sinal.
- Carry, significando que a subtracção entre os operandos teve necessidade de um empréstimo.
- Apesar de CMP também afectar as flags de parity e auxiliary carry, estas flags raramente são testadas após uma instrução de comparação.

INSTRUÇÕES ARITMÉTICAS - NEG

- ◎ Obtém o complemento de 2 do valor presente em OPERAND
- ◎ Formato geral:
 - NEG OPERAND
 - OPERAND := 0 – OPERAND
- ◎ Formatos específicos:
 - NEG MEM
 - NEG REG

INSTRUÇÕES ARITMÉTICAS - NEG

- Se OPERAND for 0, o seu sinal não é alterado, apesar da flag de carry ser posta a “0”.
- Negar qualquer outro valor coloca a flag de carry a “1”.
- As flags de Overflow, Auxiliary Carry, Sign, Parity e Zero são afectadas da mesma forma que se utilizasse a instrução SUB.

INSTRUÇÕES ARITMÉTICAS - (I)MUL

- ◎ Multiplica operandos de 8 ou de 16 bits.
 - Se OPERAND for de 8 bits AX:=
AL*OPERAND
 - Se OPERAND for de 16 bits
DX:AX:=AX*OPERAND
 - DX contém a word mais significativa do resultado;
 - AX contém a word menos significativa do resultado.

INSTRUÇÕES ARITMÉTICAS - (I)MUL

◎ Formato geral:

○ (I)MUL OPERAND

- Existem 2 formatos consoante OPERAND seja um valor com sinal (IMUL) ou sem sinal (MUL).
- Assume-se que o acumulador (AL ou AX) é o operando destino.

◎ Formatos específicos:

○ (I)MUL REG

○ (I)MUL MEM

INSTRUÇÕES ARITMÉTICAS - (I)MUL

- ◎ As flags auxiliary carry, parity, sign e zero ficam com valores indefinidos após as instruções MUL e IMUL.
- ◎ As flags carry e overflow são activadas se a metade superior do resultado contiver dígitos significativos do resultado, caso contrário são postas a “0”.

INSTRUÇÕES ARITMÉTICAS - (I)DIV

- ◎ Divide operandos de 8 ou de 16 bits.
 - Se OPERAND for de 8 bits divide AX por OPERAND, ficando o quociente em AL e o resto em AH.
 - Se OPERAND for de 16 bits divide DX:AX por OPERAND, ficando o quociente em AX e o resto em DX.

INSTRUÇÕES ARITMÉTICAS - (I)DIV

- ◎ Formato geral:
 - (I)DIV OPERAND
 - Existem 2 formatos consoante OPERAND seja um valor com sinal (IDIV) ou sem sinal (DIV).
- ◎ Formatos específicos:
 - (I)DIV REG
 - (I)DIV MEM

INSTRUÇÕES ARITMÉTICAS - (I)DIV

- ◎ As flags de overflow, carry, auxiliary carry, parity, sign e zero ficam com valores indefinidos após a operação de divisão.
- ◎ Notar que não existe qualquer forma de fazer uma divisão por um valor imediato.

INSTRUÇÕES LÓGICAS - AND E OR

- ◎ Formato geral:

- AND DEST, ORIG
- DEST:= DEST AND ORIG

- OR DEST, ORIG
- DEST:= DEST OR ORIG

AND Lógico		
DEST	ORIG	Result
0	0	0
0	1	0
1	0	0
1	1	1

OR Lógico		
DEST	ORIG	Result
0	0	0
0	1	1
1	0	1
1	1	1

INSTRUÇÕES LÓGICAS - XOR E NOT

- XOR DEST, ORIG
- DEST:= DEST XOR ORIG

XOR Lógico		
DEST	ORIG	Result
0	0	0
0	1	1
1	0	1
1	1	0

- NOT DEST
- DEST:= NOT DEST

NOT Lógico	
DEST	Result
0	1
1	0

INSTRUÇÕES LÓGICAS

◎ Formatos específicos:

- AND REG, REG
- AND REG, MEM
- AND MEM, REG
- AND REG, IMMEDIATE_DATA
- AND MEM, IMMEDIATE_DATA
- AND AX/AL, IMMEDIATE_DATA
- NOT REG
- NOT MEM

INSTRUÇÕES LÓGICAS

- ◎ As instruções OR e XOR usam o mesmo formato da instrução AND.
- ◎ À excepção da instrução NOT que não afecta as flags as instruções AND, OR e XOR afectam as flags da seguinte forma:
 - Colocam a flag carry a “0”.
 - Colocam a flag overflow a “0”.
 - Activam a flag zero, se o resultado for zero e desactivam-na em situação contrária.

INSTRUÇÕES LÓGICAS

- Copiam o bit mais significativo do resultado para a flag sign.
- Activam a flag parity de acordo com a paridade (número de bits a “1”) do resultado.
- A flag auxiliary carry fica com um valor indefinido.
- Ambos os operandos das instruções AND, OR e XOR têm de possuir o mesmo tamanho.

INSTRUÇÕES DE DESLOCAMENTO-SHL/SAL

- ◎ Movem cada bit no operando DEST uma posição à esquerda, o número de vezes especificado pelo operador COUNT.
 - As posições menos significativas são preenchidas com zeros.
 - O bit mais significativo vai para a flag carry.



- ◎ Formato geral:
 - SHL/SAL DEST, COUNT

INSTRUÇÕES DE DESLOCAMENTO-SHL/SAL

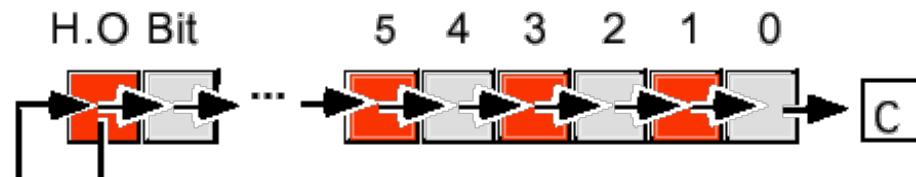
- ◎ As instruções SHL/SAL apresentam o seguinte comportamento:
 - Se COUNT for “0”, as flags não são afectadas.
 - A flag carry contém o último bit deslocado da posição mais significativa do operando.
 - A flag overflow contém “1” se os 2 bits mais significativos forem diferentes antes da realização do deslocamento. Fica com um valor indefinido se COUNT não for “1”.
 - A flag zero ficará com “1” se o deslocamento produzir um resultado zero.

INSTRUÇÕES DE DESLOCAMENTO-SHL/SAL

- A flag sign conterá o bit mais significativo do resultado.
 - A flag parity conterá “1” se houver um número par de bits a “1” no byte menos significativo do resultado.
 - A flag auxiliary carry fica com um valor indefinido.
- ◎ Nota:
- Deslocar um valor inteiro (com ou sem sinal) à esquerda n vezes equivale a multiplicar esse número por 2^n .

INSTRUÇÕES DE DESLOCAMENTO - SAR

- ◎ Desloca todos os bits no operando DEST uma posição à direita, o número de vezes especificado pelo operador COUNT.
- As posições mais significativas são réplicas do bit mais significativo, que se mantém.
- O bit menos significativo vai para a flag carry.



INSTRUÇÕES DE DESLOCAMENTO - SAR

- ◎ Formato geral:
 - SAR DEST, COUNT
- ◎ Esta instrução apresenta o seguinte comportamento:
 - Se COUNT for “0”, as flags não são afectadas.
 - A flag carry contém o último bit deslocado, da posição menos significativa do operando.

INSTRUÇÕES DE DESLOCAMENTO - SAR

- A flag sign conterá o bit mais significativo do resultado.
- A flag overflow contém “0” se COUNT for “1”. Caso contrário fica com um valor indefinido ;
- A flag zero ficará com “1” se o deslocamento produzir um resultado zero.
- A flag parity conterá “1” se houver um número par de bits a “1” no byte menos significativo do resultado.
- A flag auxiliary carry fica com um valor indefinido.

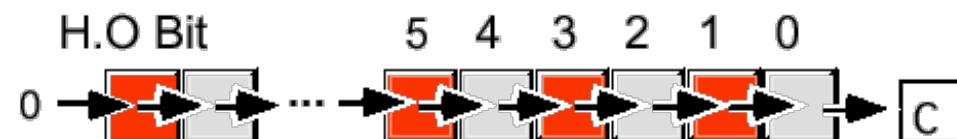
INSTRUÇÕES DE DESLOCAMENTO - SAR

◎ Notas:

- Deslocar um valor inteiro à direita n vezes equivale a dividir esse número por 2^n .
- Esta instrução é equivalente a IDIV com excepção das “truncagens” que faz!
 - EX: MOV AX, -15
CWD
MOV BX, 2
IDIV BX ;Produz -7
;IDIV faz a truncagem em direcção a “0”
 - EX: MOV AX, -15
SAR AX, 1 ; Produz -8
;SAR faz a truncagem em direcção ao menor resultado

INSTRUÇÕES DE DESLOCAMENTO - SHR

- ◎ Desloca todos os bits no operando DEST uma posição à direita, o número de vezes especificado pelo operador COUNT.
 - O bit mais significativo é colocado a zero.
 - O bit menos significativo vai para a flag carry.



INSTRUÇÕES DE DESLOCAMENTO - SHR

- ◎ Formato geral:
 - SHR DEST, COUNT
- ◎ A instrução SHR apresenta o seguinte comportamento:
 - Se COUNT for “0”, não afecta quaisquer flags.
 - A flag carry contém o último bit deslocado, da posição menos significativa do operando.
 - Se COUNT for “1” a flag overflow contém o valor do bit mais significativo do operando, antes do deslocamento. Caso contrário fica com um valor indefinido.

INSTRUÇÕES DE DESLOCAMENTO - SHR

- A flag zero ficará com “1” se o deslocamento produzir um resultado zero.
- A flag sign conterá o bit mais significativo do resultado, que é sempre zero.
- A flag parity conterá “1” se houver um número par de bits a “1” no byte menos significativo do resultado.
- A flag auxiliary carry fica com um valor indefinido.

INSTRUÇÕES DE DESLOCAMENTO - SHR

◎ Notas:

- Deslocar um valor inteiro à direita n vezes equivale a dividir esse número por 2^n . Porém esta instrução apenas funciona correctamente como divisão se ambos os operandos forem sem sinal.

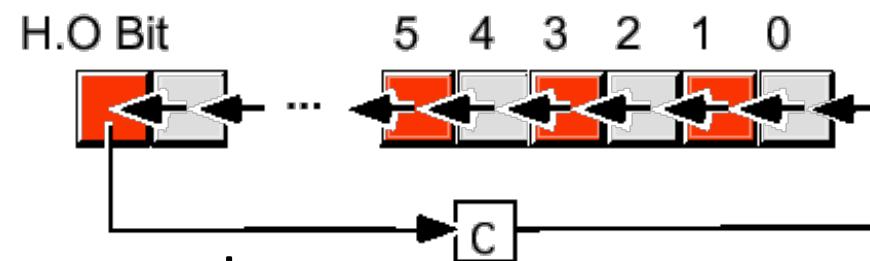
○ EX: MOV AX, -2

SHR AX,1

AX=32767 (7FFFh) e não -1 como esperado.

INSTRUÇÕES DE ROTAÇÃO - RCL

- ◎ Roda os bits no operando DEST uma posição à esquerda, o número de vezes especificado pelo operador COUNT, passando pela flag Carry.



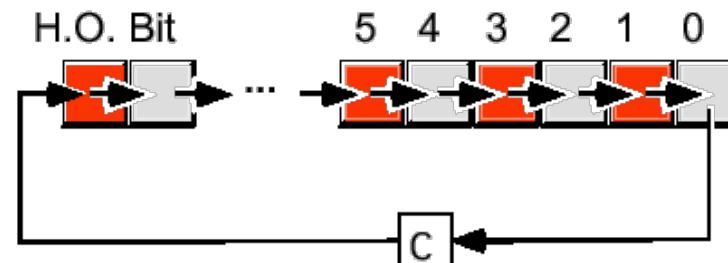
- ◎ Formato geral:
 - **RCL DEST, COUNT**

INSTRUÇÕES DE ROTAÇÃO - RCL

- ◎ A instrução RCL apresenta o seguinte comportamento:
 - A flag carry contém o último bit deslocado do bit mais significativo do operando.
 - Se COUNT é 1, RCL activa a flag overflow se o sinal mudar em resultado de uma rotação, caso contrário fica com um valor indefinido.
 - Não modifica as flags zero, sign, parity e auxiliary carry.

INSTRUÇÕES DE ROTAÇÃO - RCR

- ◎ Roda os bits no operando DEST uma posição à direita, o número de vezes especificado pelo operador COUNT, passando pela flag carry.



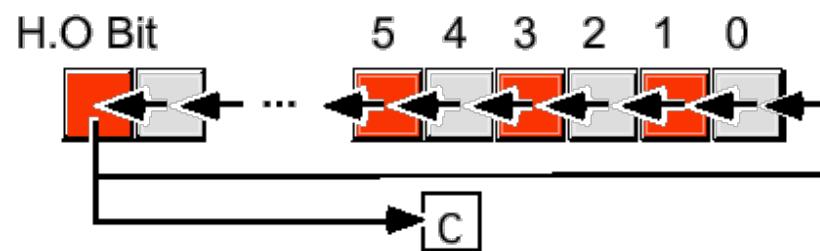
- ◎ Formato geral:
 - ◎ **RCR DEST, COUNT**

INSTRUÇÕES DE ROTAÇÃO - RCR

- ◎ A instrução RCR apresenta o seguinte comportamento:
 - A flag carry contém o último bit deslocado do bit menos significativo do operando.
 - Se COUNT é 1, RCR activa a flag overflow se o sinal mudar em resultado de uma rotação, caso contrário fica com um valor indefinido.
 - Não modifica as flags zero, sign, parity e auxiliary carry.

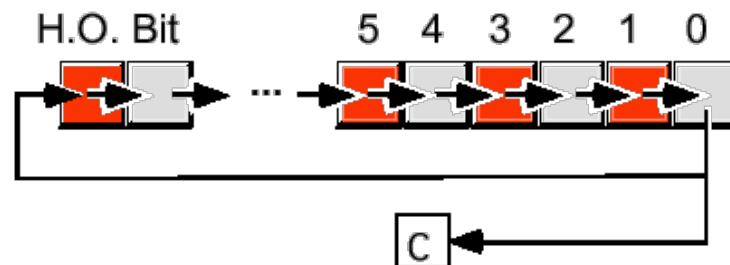
INSTRUÇÕES DE ROTAÇÃO - ROL

- ➌ É semelhante à instrução RCL, rodando o seu operando à esquerda um determinado número de bits. A principal diferença é que ROL desloca o seu bit mais significativo do operando e não a flag carry para o bit menos significativo. No entanto, ROL também reproduz o bit mais significativo para a flag de carry.



INSTRUÇÕES DE ROTAÇÃO - ROR

- ➌ É semelhante à instrução RCR, rodando o seu operando à direita um determinado número de bits. A principal diferença é que ROR desloca o seu bit menos significativo do operando e não a flag carry para o bit mais significativo. No entanto ROL também reproduz o bit menos significativo para a flag de carry.



INSTRUÇÕES DE DESLOCAMENTO E ROTAÇÃO

- ◎ Todas as instruções de deslocamento e rotação referidas podem apresentar os seguintes formatos específicos:
 - instrução reg, 1
 - instrução mem, 1
 - instrução reg, cl
 - instrução mem, cl

INSTRUÇÕES PARA TESTE DE BITS - TEST

- ◎ Realiza o AND lógico, bit a bit, entre os seus operandos activando as flags correspondentes mas não guarda o resultado da operação.
- ◎ Normalmente utilizada para testar se um determinado bit tem o valor “1”.
 - Ex: TEST AL,16
 - Testa o estado do 5º bit.

INSTRUÇÕES PARA TESTE DE BITS - TEST

- ◎ Formato geral:
 - TEST OPERAND1, OPERAND2
- ◎ Formatos específicos:
 - TEST REG, REG
 - TEST MEM, REG
 - TEST REG, MEM*
 - TEST MEM, IMMEDIATE_DATA
 - TEST REG, IMMEDIATE_DATA
 - TEST AX/AL, IMMEDIATE_DATA

INSTRUÇÕES PARA TESTE DE BITS - TEST

- ◎ Activa as seguintes flags de forma idêntica à instrução AND:
 - Põe a flag carry a “0”.
 - Põe a flag overflow a “0”.
 - Activa a flag zero, se o resultado for zero e desactiva-a em situação contrária.
 - Copia o bit mais significativo do resultado para a flag sign.
 - Activa a flag parity de acordo com a paridade (número de bits a “1”) do resultado.
 - A flag auxiliary carry fica com um valor indefinido.

INSTRUÇÕES DE MANIPULAÇÃO DE FLAGS

- ◎ CLC: Coloca a “0” a Flag de Carry
- ◎ STC: Coloca a “1” a Flag de Carry
- ◎ CMC: Complementa a Flag de Carry
- ◎ CLD: Coloca a “0” a Flag Direction
- ◎ STD: Coloca a “1” a Flag Direction
- ◎ CLI: Coloca a “0” a Flag Interrupt Enable/Disable
- ◎ STI: Coloca a “1” a Flag Interrupt Enable/Disable

INSTRUÇÕES DE MANIPULAÇÃO DE STRINGS

- ◎ As instruções MOVS, LODS, STOS, SCAS e CMPS podem operar sobre Strings de bytes ou de words. Para especificar o tamanho do objecto acrescenta-se à mnemónica da instrução a letra b ou w:
 - movsb, movsw, lodsb, lodsw, cmpsb, etc.
- ◎ MOVS, STOS, SCAS, CMPS podem ser usadas para manipular um elemento (byte ou word) numa String, ou para processar uma String completa.

INSTRUÇÕES DE MANIPULAÇÃO DE STRINGS MOVS

- ◎ Reproduz uma String entre localizações de memória
- ◎ **movs{b, w}:**

```
es:[di] := ds:[si]
if direction_flag = 0 then
    si := si + size      ; size= 1 (byte), 2 (word)
    di:= di + size
else
    si := si - size
    di:= di - size
endif
```

INSTRUÇÕES DE MANIPULAÇÃO DE STRINGS CMPS

- ◎ Compara duas Strings.

- ◎ **cmps{b, w}:**

cmp ds:[si], es:[di]

if direction_flag = 0 then

 si := si + size ; size= 1 (byte), 2 (word)

 di:= di + size

else

 si := si - size

 di:= di - size

endif

INSTRUÇÕES DE MANIPULAÇÃO DE STRINGS LODS

- ◎ Carrega um elemento da String, apontada por SI, para o acumulador.

- ◎ **lod{b, w}:**

ax/al := ds:[si]

if direction_flag = 0 then

 si := si + size ; size= 1 (byte), 2 (word)

else

 si := si - size

endif

INSTRUÇÕES DE MANIPULAÇÃO DE STRINGS STOS

- ◎ Armazena o valor presente no acumulador numa posição da String, determinada por DI.

- ◎ **stos{b, w}:**

es:[di] := ax/al

if direction_flag = 0 then

 di := di + size ; size= 1 (byte), 2 (word)

else

 di := di - size

endif

INSTRUÇÕES DE MANIPULAÇÃO DE STRINGS SCAS

- ◎ Verifica se o valor presente no acumulador se encontra numa determinada posição da String, determinada por DI.
- ◎ **scas{b, w}:**

cmp ax/al, es:[di]

if direction_flag = 0 then

 di := di + size ; size= 1 (byte), 2 (word)

else

 di := di - size

endif

INSTRUÇÕES DE MANIPULAÇÃO DE STRINGS

- ◎ As instruções MOVS, LODS, STOS e suas derivadas não alteram as flags.
- ◎ As instruções CMPS, SCAS e suas derivadas alteram as flags overflow, sign, zero, auxiliary carry, parity e carry da mesma forma que a instrução CMP.

INSTRUÇÕES DE REPETIÇÃO

- ◎ As instruções de manipulação de strings, só por si, não operam sobre strings, mas em apenas um byte ou word de cada vez.
- ◎ Se se utilizar um prefixo de repetição associado a essas instruções é possível realizar uma operação sobre uma string envolvendo múltiplos bytes ou words.
- ◎ Quando se especifica o prefixo de repetição antes de uma instrução de manipulação de strings, essa instrução é repetida CX vezes.

INSTRUÇÕES DE REPETIÇÃO

Tipos:

- REP (repete uma operação CX vezes)
- REPZ (repete enquanto 0)
- REPE (repete enquanto igual)
- REPNZ (repete enquanto não zero)
- REPNE (repete enquanto diferente)

INSTRUÇÕES DE REPETIÇÃO

- ◎ Para MOVS:
 - REP MOVS{b,w}

- ◎ Para CMPS:
 - REPE CMPS{b,w}
 - REPZ CMPS{b,w}
 - REPNE CMPS{b,w}
 - REPNZ CMPS{b,w}

INSTRUÇÕES DE REPETIÇÃO

- ◎ Para SCAS:
 - REPE SCAS{b,w}
 - REPZ SCAS{b,w}
 - REPNE SCAS{b,w}
 - REPNZ SCAS{b,w}
- ◎ Para STOS:
 - REP STOS{b,w}
- ◎ Normalmente não se utilizam prefixos de repetição com a instrução LODS.

INSTRUÇÕES DE CONTROLO

- ◎ Existem 3 tipos de instruções que permitem o controlo do fluxo de um programa:
 - saltos incondicionais
 - saltos condicionais
 - chamada e retorno de procedimentos

INSTRUÇÕES DE CONTROLO

- ◎ O registo CS (Code Segment) contém os 16 bits mais significativos do endereço físico onde começa o segmento de código.
 - Esse segmento contém os códigos binários das instruções que constituem o programa que está a ser executado.

INSTRUÇÕES DE CONTROLO

- ➊ O registo IP (Instruction Pointer) referencia, dentro do segmento de código, a próxima instrução a ser executada, ou seja, o deslocamento em relação ao CS que tem a próxima instrução.

INSTRUÇÕES DE CONTROLO

- ◎ Se o programa consistir numa sequência de instruções simples (que não alterem o fluxo do programa) o endereço da próxima instrução a executar é sempre o imediatamente a seguir ao da instrução actual.
- ◎ Porém, existem instruções que alteram a sequência normal de um programa (instruções que permitem efectuar saltos, ex: jmp).

INSTRUÇÕES DE CONTROLO

- ◎ Existem essencialmente dois tipos de saltos:
 - Saltos incondicionais
 - Ocorrem sempre, sem estarem sujeitos a qualquer condição.
 - Saltos condicionais
 - Ocorrem apenas se uma determinada condição se verificar.

INSTRUÇÕES DE CONTROLO

- ◎ O 8086 calcula o endereço físico da próxima instrução a executar adicionando o deslocamento contido no registo IP ao endereço de base representado pelo número contido no registo CS.
- ◎ Quando o 8086 executa uma instrução de salto, carrega um novo número (deslocamento) no registo IP e por vezes também um novo número no registo CS.

INSTRUÇÃO DE SALTO INCONDICIONAL JMP

- ◎ Os saltos incondicionais transferem o controlo do fluxo para outro ponto do programa, sem estarem dependentes de qualquer condição.
- ◎ Existem saltos:
 - Near ou Intrasegmento
 - Far ou Intersegmento
 - Directos
 - Indirectos

INSTRUÇÃO DE SALTO INCONDICIONAL JMP

- ◎ Saltos Near ou Intrasegmento
 - Transferem o controlo para uma instrução localizada no segmento de código actual.
- ◎ Saltos Far ou Intersegmento
 - Transferem o controlo para uma instrução localizada num segmento de código diferente do actual.
 - Neste caso, é necessário alterar não só o valor do registo IP, mas também o valor do registo CS.

INSTRUÇÃO DE SALTO INCONDICIONAL JMP

◎ Saltos Directos

- São directos se o destino for um número, o qual referencia um deslocamento em relação à localização actual. Nesse caso o novo valor do IP ficará a ser $IP + <\text{destino}>$.

◎ Saltos Indirectos

- São indirectos se o destino for um registo ou localização de memória. Nesse caso, o 8086 terá de ir buscar o conteúdo do registo ou localização de memória, conteúdo esse que passará a ser o novo valor do registo IP.

INSTRUÇÃO DE SALTO INCONDICIONAL JMP

◎ Formato geral:

JMP <destino>

◎ Formato específico:

- JMP DISP8; Intrasegmento directo de 8 bits
- JMP DISP16; Intrasegmento directo de 16 bits
- JMP ADRS32; Intersegmento directo de 32 bits
- JMP MEM16; Intrasegmento indirecto de 16 bits
- JMP REG16; Intrasegmento indirecto por registo
- JMP MEM32; Intersegmento indirecto de 32 bits

INSTRUÇÃO DE SALTO INCONDICIONAL JMP

- ◎ Inrasegmento Directo:
 - $IP := IP + disp$
- ◎ Intersegmento Directo:
 - $CS:IP := addr32$
- ◎ Inrasegmento Indirecto:
 - $IP := mem16$
- ◎ Inrasegmento Indirecto por Registo:
 - $IP := Reg$
- ◎ Intersegmento Indirecto:
 - $CS:IP := mem32$

INSTRUÇÕES DE SALTO CONDICIONAL

- ◎ Os saltos condicionais ocorrem apenas se uma determinada condição se verificar.
- ◎ As instruções que especificam saltos condicionais analisam o conteúdo de determinadas flags ou registos para decidir se o “salto” se realiza.

INSTRUÇÕES DE SALTO CONDICIONAL

- ◎ Estas instruções aparecem normalmente depois de instruções de operações aritméticas e lógicas (que alteram os valores das flags) e quase sempre após as instruções CMP ou TEST.
- ◎ Em todos os saltos condicionais o destino tem de especificar um endereço dentro do mesmo segmento da instrução de salto.

INSTRUÇÕES DE SALTO CONDICIONAL

◎ Instruções que testam as flags

Instruction	Description	Condition	Aliases	Opposite
JC	Jump if carry	Carry = 1	JB, JNAE	JNC
JNC	Jump if no carry	Carry = 0	JNB, JAE	JC
JZ	Jump if zero	Zero = 1	JE	JNZ
JNZ	Jump if not zero	Zero = 0	JNE	JZ
JS	Jump if sign	Sign = 1		JNS
JNS	Jump if no sign	Sign = 0		JS
JO	Jump if overflow	Ovrlw=1		JNO
JNO	Jump if no Ovrlw	Ovrlw=0		JO
JP	Jump if parity	Parity = 1	JPE	JNP
JPE	Jump if parity even	Parity = 1	JP	JPO
JNP	Jump if no parity	Parity = 0	JPO	JP
JPO	Jump if parity odd	Parity = 0	JNP	JPE

INSTRUÇÕES DE SALTO CONDICIONAL

◎ Instruções para comparações sem sinal

Instruction	Description	Condition	Aliases	Opposite
JA	Jump if above ($>$)	Carry=0, Zero=0	JNBE	JNA
JNBE	Jump if not below or equal (not \leq)	Carry=0, Zero=0	JA	JBE
JAE	Jump if above or equal (\geq)	Carry = 0	JNC, JNB	JNAE
JNB	Jump if not below (not $<$)	Carry = 0	JNC, JAE	JB
JB	Jump if below (<)	Carry = 1	JC, JNAE	JNB
JNAE	Jump if not above or equal (not \geq)	Carry = 1	JC, JB	JAE
JBE	Jump if below or equal (\leq)	Carry = 1 or Zero = 1	JNA	JNBE
JNA	Jump if not above (not $>$)	Carry = 1 or Zero = 1	JBE	JA
JE	Jump if equal (=)	Zero = 1	JZ	JNE
JNE	Jump if not equal (\neq)	Zero = 0	JNZ	JE

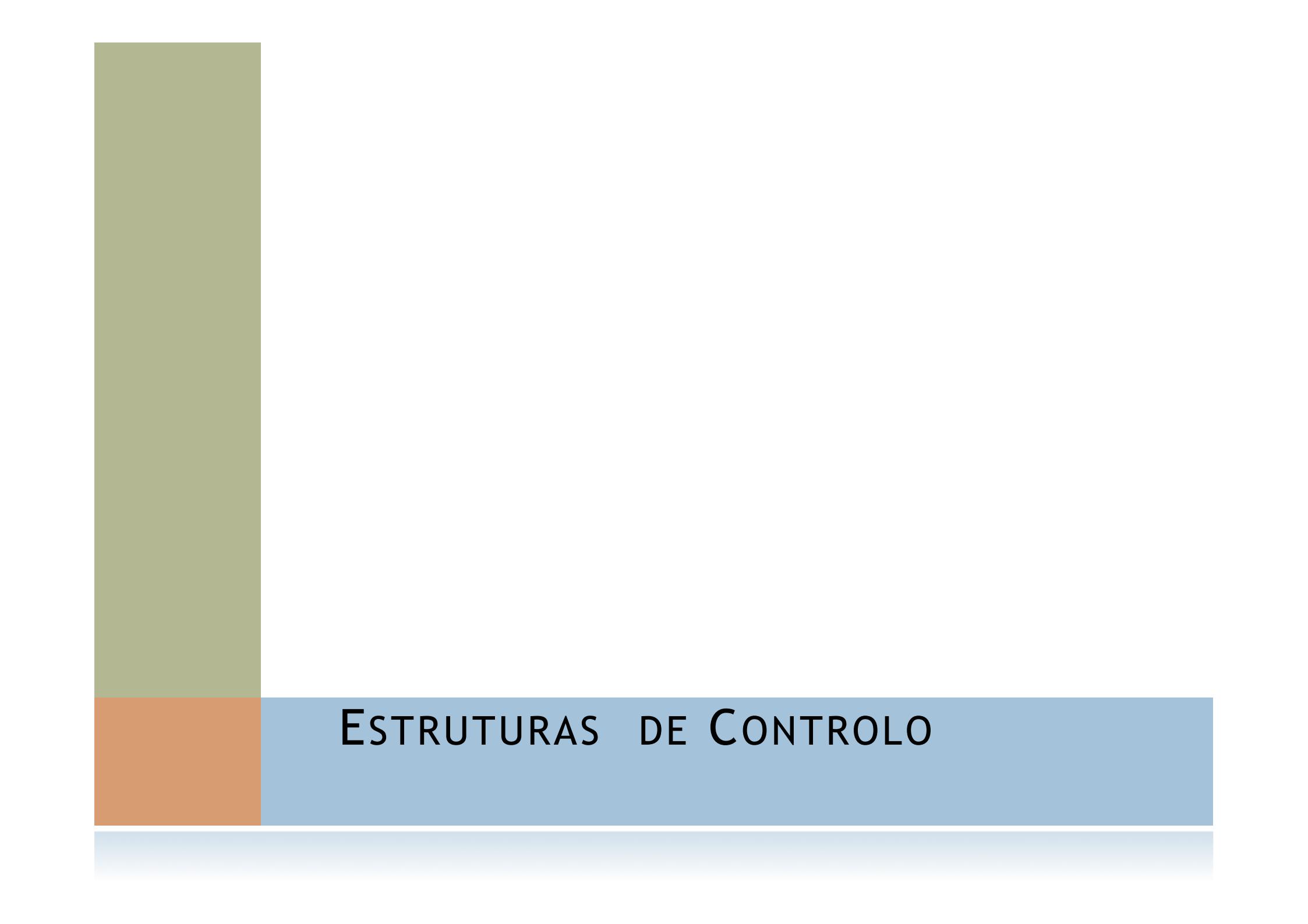
INSTRUÇÕES DE SALTO CONDICIONAL

◎ Instruções para comparações com sinal.

Instruction	Description	Condition	Aliases	Opposite
JG	Jump if greater (>)	Sign = Ovrlfw or Zero=0	JNLE	JNG
JNLE	Jump if not less than or equal (not <=)	Sign = Ovrlfw or Zero=0	JG	JLE
JGE	Jump if greater than or equal (>=)	Sign = Ovrlfw	JNL	JGE
JNL	Jump if not less than (not <)	Sign = Ovrlfw	JGE	JL
JL	Jump if less than (<)	Sign ≠ Ovrlfw	JNGE	JNL
JNGE	Jump if not greater or equal (not >=)	Sign ≠ Ovrlfw	JL	JGE
JLE	Jump if less than or equal (<=)	Sign ≠ Ovrlfw or Zero = 1	JNG	JNLE
JNG	Jump if not greater than (not >)	Sign ≠ Ovrlfw or Zero = 1	JLE	JG
JE	Jump if equal (=)	Zero = 1	JZ	JNE
JNE	Jump if not equal (≠)	Zero = 0	JNZ	JE

CHAMADAS E RETORNO DE PROCEDIMENTOS

- ◎ A chamada a procedimentos é realizada através da instrução CALL.
 - É esta instrução que indica ao processador qual o endereço da primeira instrução do procedimento.
- ◎ O retorno de procedimentos é realizado através da instrução RET surge normalmente no final do procedimento.
- ◎ Consultar ficha explicativa de Procedimentos em Assembly.



ESTRUTURAS DE CONTROLO

EXPRESSÕES CONDICIONAIS-IF THEN

⑤ if <condicao> then

accão

accão

...

accão

endif

EXPRESSÕES CONDICIONAIS-IF THEN

Ex:

```
if temp=37 then  
    febre=1
```

Em Assembly:

```
CMP temp, 37  
JNE FIMSE  
MOV febre, 1
```

FIMSE:

EXPRESSÕES CONDICIONAIS-IF THEN ELSE

```
④ if <condicao> then  
    accao  
    accao  
    ...  
    accao  
else  
    accao  
    accao  
    ...  
    accao  
endif
```

EXPRESSÕES CONDICIONAIS-IF THEN ELSE

Ex:

if a>b then

 max:=a

else

 max:=b

Em Assembly:

```
MOV    AX,  a  
CMP    AX,  b  
JBE    ElseBlk  
MOV    AX,  a  
jmp    EndOfIf  
ElseBlk: MOV    AX,  b  
EndOfIf: MOV    max, AX
```

EXPRESSÕES CONDICIONAIS-IF THEN ELSE

Ex:

```
if a=1 then
    igual := igual + 1
else
    if a<1 then
        menor := menor + 1
    else
        maior := maior + 1
```

Em Assembly:

```
cmp    a,1
jne   dif
inc   igual
jmp   fim

dif:  cmp   a,1
      jae  maior
      inc   menor
      jmp   fim

maior: inc   maior
fim:
```

CICLOS - WHILE Do

```
⑤ while <condicao> do  
    accao  
    accao  
    ...  
endwhile
```

CICLOS - WHILE DO

◎ Ex:

```
i:=0  
while i<100 do  
    i:=i+1
```

◎ Em Assembly:

```
        MOV    i, 0  
WhileIni: CMP    i, 100  
           JGE    WhileDone  
           INC    i  
           JMP    WhileIni
```

WhileDone:

CICLOS - Do WHILE

do

acao

acao

...

while <condicao>

endDoWhile

CICLOS - Do WHILE

◎ Ex:

```
i := 1000;  
do {  
    i := i-1;  
} while i>0;
```

◎ Em Assembly:

```
mov cx, 1000  
ciclo:  
    loop ciclo
```

Notar que o ciclo é executado pelo menos uma vez mesmo se cx=0.
Isto porque só efectua o teste no final e não no inicio!!

CICLOS - For To Do

⑤ for count = m to n do

 accão

 accão

 ...

endfor

CICLOS - For To Do

◎ EX:

FOR i := 1 TO 10 DO k := k + i - j

◎ Em Assembly:

```
        MOV    i, 1
InitFor: CMP    i, 10
          JA     FIM
          MOV    AX, k
          ADD    AX, i
          SUB    AX, j
          MOV    k, AX
          INC    i
          JMP    InitFor
FIM:
```

INSTRUÇÃO LOOP

- Formato:

LOOP <label>

- Decrementa o registo CX saltando para a localização destino caso CX seja diferente de zero.
- Útil para o caso em que se pretende repetir uma sequência de instruções um determinado número de vezes.

INSTRUÇÃO LOOP

- ◎ De cada vez que a instrução LOOP é executada o conteúdo do registo CX é decrementado um valor.
- ◎ Se depois de decrementado o conteúdo do registo CX o seu valor for diferente de zero, a instrução LOOP implementa um jump para o endereço referenciado por <label>.
- ◎ O <label> referencia um deslocamento em relação ao registo CS onde está a primeira instrução do ciclo.
- ◎ As instruções Loop não afectam as flags.

INSTRUÇÃO LOOPE/LOOPZ

- ◎ Repete uma sequência de instruções enquanto um determinado valor for igual a outro, CX for diferente de zero e a flag zero estiver a “1”.
 - De cada vez que esta instrução é executada o conteúdo do registo CX é decrementado de uma unidade.
 - Se depois de decrementado o conteúdo do registo CX, o seu valor for diferente de zero e a flag zero estiver activada a instrução implementa um jump para o endereço referenciado por <label>.

INSTRUÇÃO LOOPNE/LOOPNZ

- ◎ Repete uma sequência de instruções enquanto um determinado valor for diferente de outro, CX for diferente de zero e a flag zero estiver a “0”.
 - De cada vez que esta instrução é executada o conteúdo do registo CX é decrementado um valor.
 - Se depois de decrementado o conteúdo do registo CX, o seu valor for diferente de zero e a flag zero estiver desactivada a instrução implementa um jump para o endereço referenciado por <label>.