



Organização de Computadores 1

5.1 – Linguagem de Montagem (*Assembly*)

Sistemas Numéricos

- ★ **Sistema Decimal:** sistema **natural** do homem.
 - ✓ No assembly um número decimal **pode** terminar com um **d**.
 - × Ex: 64223 ou 64223**d** ou 64223**D**.
- ★ **Sistema Binário:** representado por **bits** (0 ou 1)
 - ✓ **Agrupamento dos bits:**
 - × **Nibble:** 4 bits
 - × **Byte:** 8 bits
 - × Palavra (**word**): 16 bits
 - × Palavra dupla (**double word**): 32 bits
 - ✓ No assembly um número binário **deve** terminar com um **b**.
 - × Ex: 1110101**b** ou 1110101**B**.
- ★ **Sistema Hexadecimal:**
 - ✓ 1 algarismo hexadecimal = 4 bits (**ex:** 0011|1101b = 3Dh)
 - ✓ No assembly um número hexadecimal **deve** começar com um **numeral decimal** e terminar com um **h**.
 - × Ex: 1Fh, 0FFAh.

Programação

- ✦ Computador executa programas criados pelos programadores.
 - ✓ Programas = conjunto de instruções.
- ✦ Instruções dizem ao computador o que fazer para resolver determinado problema.
- ✦ Elementos de uma instrução:
 - ✓ **Opcode:** define a operação a ser realizada
 - ✓ **Operandos:** dados necessários para realizar a operação desejada.

Arquitetura 8086

CPU	Registradores de 16 bits
Memória	Memória limitada a 1MB
	Dividida em segmentos 64kb
	Somente modo real
	Bytes na memória não possuem endereço único
	Organização <i>little endian</i>

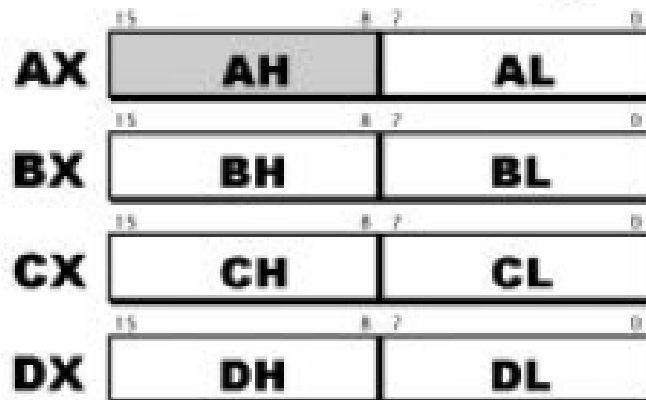
Arquitetura 8086

- Mapeamento da memória

0x00000 - 0x003FF	Tabela de interrupções (ISR)
0x00400 - 0x005FF	Área de BIOS (BDA)
0x00600 - 0x9FFFF	Área livre
0xA0000 - 0xAFFFF	Memória de vídeo EGA/VGA
0xB0000 - 0xB7FFF	Memória de texto monocromático
0xB8000 - 0xBFFFF	Memória de vídeo CGA
0xC0000 - 0xDFFFF	ROM instalada
0xE0000 - 0xFDFFF	ROM fixa
0xFE000 - 0xFFFFF	ROM da BIOS

Registadores (Processador 8088/8086)

- ★ CPU possui 14 registradores de **16 bits visíveis**.
- ★ **4 registradores de uso geral**:
 - ✓ **AX (Acumulador)**: armazena operandos e resultados dos cálculos aritméticos e lógicos.
 - ✓ **BX (Base)**: armazena endereços indiretos.
 - ✓ **CX (Contador)**: conta iterações de *loops* ou especifica o n° de caracteres de uma *string*.
 - ✓ **DX (Dados)**: armazena *overflow* e endereço de E/S.
 - ✓ Podem ser usados como **registradores de 8 bits**:
 - × Ex: **AH** e **AL** (byte alto e byte baixo de **AX**).



Registradores (Processador 8088/8086)

★ 4 registradores de segmento:

- ✓ **CS (Segmento de Código)**: contém o endereço da área com as **instruções** de máquina em execução.
- ✓ **DS (Segmento de Dados)**: contém o endereço da área com os **dados** do programa.
 - ✗ Geralmente aponta para as variáveis globais do programa.
- ✓ **SS (Segmento de Pilha)**: contém o endereço da área com a **pilha**. Que armazena informações importantes sobre o estado da máquina, variáveis locais, endereços de retorno e parâmetros de subrotinas.
- ✓ **ES (Segmento Extra)**: utilizado para ganhar acesso a alguma área da memória quando não é possível usar os outros registradores de segmento.
 - ✗ **Ex**: transferências de bloco de dados.

Registadores (Processador 8088/8086)

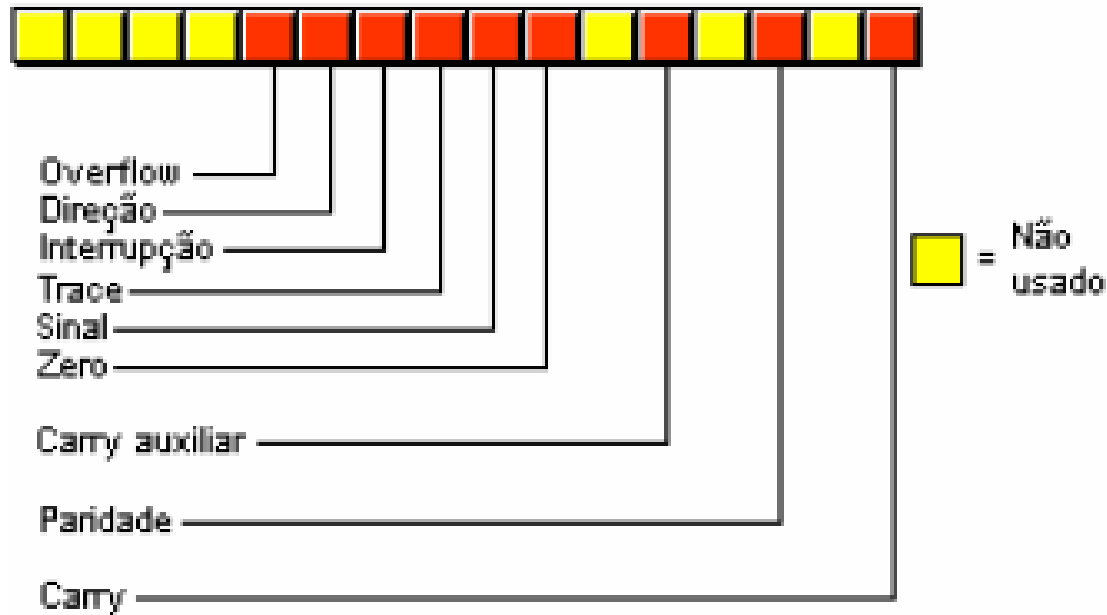
★ 5 registradores de *offset*:

- ✓ **PC** ou **IP** (*Instruction Pointer*): usado em conjunto com o **CS** para apontar a próxima instrução.
- ✓ **SI** (*source index*) e **DI** (*destiny index*): utilizados para mover blocos de bytes de um lugar (**SI**) para outro (**DI**) e como ponteiros para endereçamento (junto com os registradores CS, DS, SS e ES).
- ✓ **BP** (*Base Pointer*): usado em conjunto com o **SS** para apontar a base da pilha.
 - × Similar ao registrador **BX**.
 - × Usado para acessar parâmetros e variáveis locais.
- ✓ **SP** (*Stack Pointer*): usado em conjunto com o **SS** para apontar o topo da pilha.

Registadores (Processador 8088/8086)

★ 1 registrador de estado do processador (PSW) :

- ✓ Registrador especial composto por sinalizadores (*flags*) que ajudam a determinar o **estado atual** do processador.
 - × Coleção de **valores de 1 bit**.
- ✓ Apenas 9 bits são utilizados.
 - × **4 mais utilizados**: **ZF** - zero; **CF** - *carry* ("vai um") ou *borrow* ("vem um"); **SF** - sinal; e **OF** - *overflow* ou *underflow*.



Organização dos Registradores – Família Intel

General Registers

AX	Accumulator
BX	Base
CX	Count
DX	Data

Pointer & Index

SP	Stack Pointer
BP	Base Pointer
SI	Source Index
DI	Dest Index

Segment

CS	Code
DS	Data
SS	Stack
ES	Extra

Program Status

Instr Ptr
Flags

General Registers

EAX	AX
EBX	BX
ECX	CX
EDX	DX

ESP	SP
EBP	BP
ESI	SI
EDI	DI

Program Status

FLAGS Register
Instruction Pointer

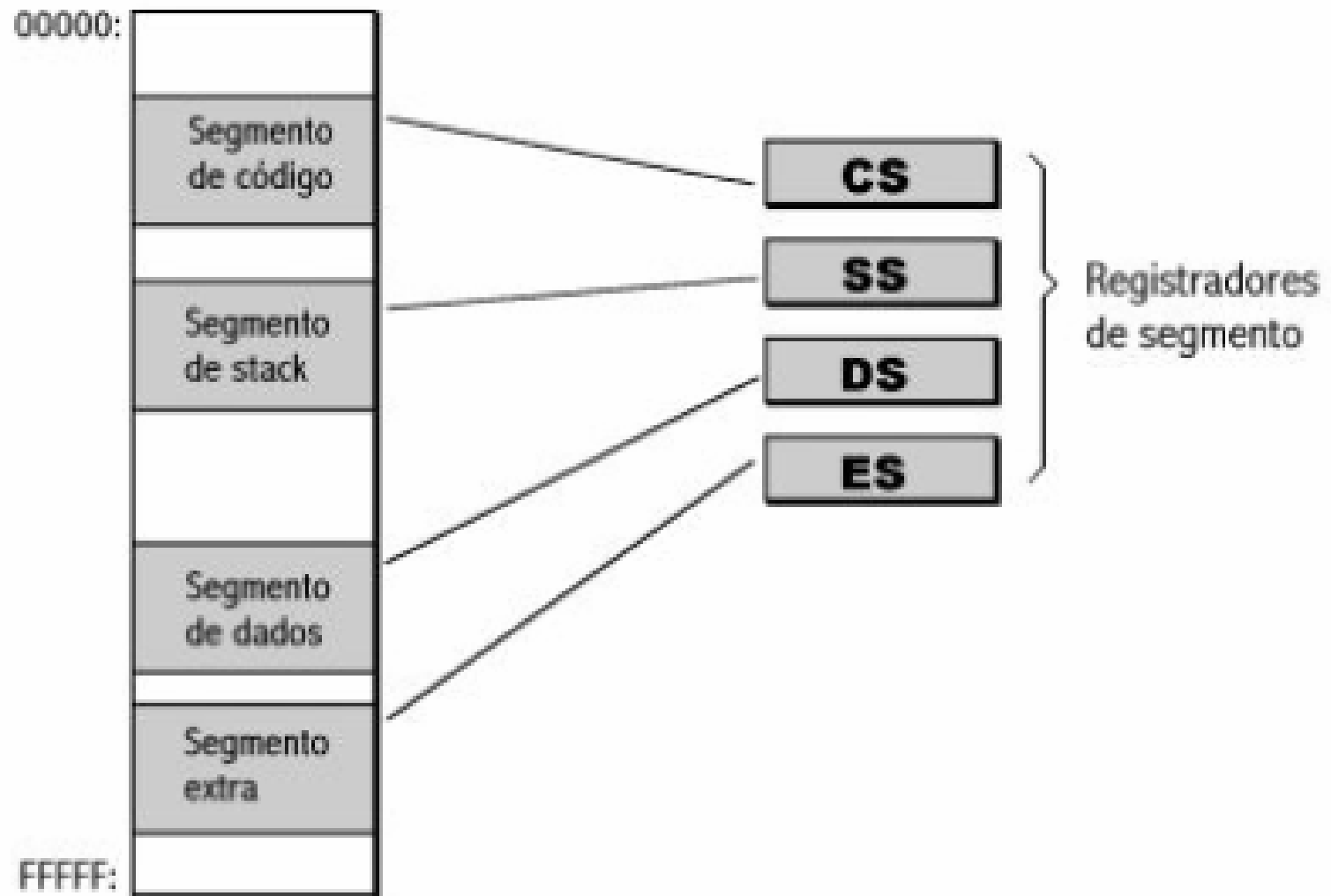
**80386 – Pentium II
(32 bits)**

**8086
(16 bits)**

Segmentação da Memória

- ★ **Ponto de vista físico:** memória é homogênea.
 - ✓ Processador 8086 endereça até 2^{20} bytes = **1MByte**.
- ★ **Ponto de vista lógico:** memória é dividida em áreas denominadas **segmentos**.
 - ✓ **Expansão** na capacidade de acesso à memória.
 - ✓ **Organização** bem mais eficiente.
- ★ Cada segmento no 8086 é uma área de memória com no mínimo **64 KB** e no máximo **1MB**.
- ★ **Registradores** de segmento indicam o **endereço inicial do segmento**.

Segmentação da Memória

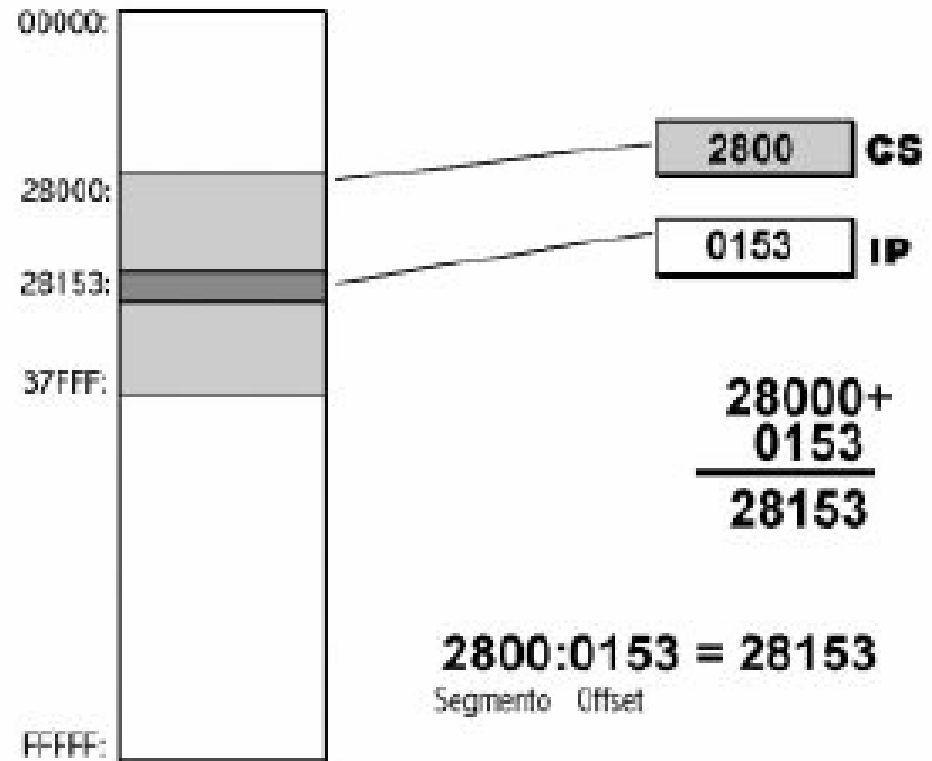


Exemplo de Utilização do Segmento

- ✦ Todos os acessos a instruções são feitas automaticamente no segmento de código.
 - ✓ Suponha que **CS** contenha o valor **2800h** e **PC** o valor **0153h**.
 - ✓ Obtenção do endereço efetivo (**EA**):

- Inclusão de um **zero** à direita do valor do CS (**endereço base**).
 - Inclusão de 4 bits.
 - Endereços possuem 20 bits.
- Soma do deslocamento (**offset**) ao endereço do segmento.

$$\underbrace{28000h}_{\text{CS} \times 16} + \underbrace{0153h}_{\text{PC}} = \underbrace{28153h}_{\text{EA}}$$



Exercícios

★ Resolva as seguintes questões:

1. Dado CS = 1E00h e IP = 152Fh, em qual posição de memória será buscada a próxima instrução?
2. Sendo DS = F055h e DI = 5FFEh, qual a posição da memória está o dado referenciado por DS:DI?

Linguagem *Assembly*

- ✦ Linguagem de montagem (***assembly***) é uma forma de **representar textualmente** o conjunto de instruções de máquina (**ISA**) do computador.
 - ✓ Cada arquitetura possui um ISA particular, portanto, pode ter uma linguagem *assembly* diferente.
- ✦ Instruções são representadas através de ***mnemônicos***, que associam o nome à sua função.
 - ✓ Nome da instrução é formada por 2, 3 ou 4 letras.
 - ✓ **Exemplos:**
 - ✦ ADD AH BH
 - **ADD**: comando a ser executado (adição).
 - **AH e BH**: operandos a serem somados.
 - ✦ MOV AL, 25
 - Move o valor **25** para o registrador **AL**.

Por que Utilizar?

- ★ Muito utilizado no desenvolvimento de aplicativos que exigem resposta em tempo real.
- ★ Tirar proveito de conjuntos de instruções específicas dos processadores.
- ★ Obter conhecimento do funcionamento do HW, visando desenvolver SW de melhor qualidade.
 - ✓ Aplicativos podem precisar de maior desempenho em partes críticas do código.
 - ✗ Nesses trechos deve-se empregar **algoritmos otimizados**, com baixa ordem de complexidade.
 - ✗ Se não atingir o tempo de resposta necessário, podemos tentar melhorar a performance utilizando **otimizações de baixo nível**.

Programa Montador (*Assembler*)

- ✦ O montador traduz diretamente uma instrução da forma textual para a forma de código binário.
 - ✓ É sob a forma binária que a instrução é carregada na memória e interpretada pelo processador.
- ✦ O montador converte o programa *assembly* para um programa em código-objeto.
- ✦ Programas complexos são estruturados em módulos.
 - ✓ Cada módulo é compilado separadamente.
 - ✓ Os módulos objeto gerados são reunidos pelo **ligador (linkeditor)**.

Criação de Programas em Assembly

★ Ferramentas necessárias:

- ✓ **Editor** para criar o programa-fonte.
 - ✗ Qualquer editor que gere texto em **ASCII** (ex: notepad, edit, etc.).
- ✓ **Montador** para transformar o código-fonte em um programa-objeto.
 - ✗ ∃ várias ferramentas no mercado (ex: masm, nasm, tasm, etc.).
- ✓ **Ligador** (linkeditor) para gerar o programa executável a partir do código-objeto.

★ Ferramentas desejáveis:

- ✓ **Depurador** para acompanhar a execução do código.
 - ✗ Importante para encontrar erros durante a programação.

TASM – Turbo Assembler

- ✦ Montador da *Borland*[®].
- ✦ Age em conjunto com o linkeditor *tlink* e o debugador *td*.
- ✦ **Criação de aplicativos:**
 - ✓ Cria-se um arquivo texto (formato ASCII) com extensão “*.asm” contendo o programas-fonte.
 - ✓ O montador *tasm* gera programas “*.obj” a partir de programas “*.asm”.
 - ✓ O ligador *tlink* transforma os arquivos “*.obj” em executáveis “*.exe”.

Montagem sem Depuração

- ★ Turbo Assembler da Borland (**TASM**):

- ✓ Programa-fonte: **exemplo.asm**
- ✓ Programa-objeto: **exemplo.obj**
- ✓ Programa executável: **exemplo.exe**

- ★ Tradução:

tasm exemplo.asm <enter>

- ★ Ligação:

tlink exemplo.obj <enter>

Montagem para Depuração

- ★ **Depuração:** acompanhamento passo a passo da execução (debug).
 - ✓ Programa **executável maior**, pois inclui **tabela de símbolos**.
- ★ Tradução:
`tasm /zi exemplo.asm <enter>`
- ★ Ligação:
`tlink /v exemplo.obj <enter>`
- ★ Depuração:
`td exemplo.exe <enter>`

Estrutura Básica de um Programa

✦ Um programa assembly deve conter as seguintes diretivas:

<code>.model small</code>	; define o modelo de memória do programa
<code>.stack 100h</code>	; reserva espaço de memória da pilha
<code>.data</code>	; define a área de declaração de constantes e
<code>...</code>	; variáveis globais (opcional)
<code>.code</code>	; define o início de um programa
<code>nm_func proc</code>	; início da execução do programa (<i>entry point</i>)
<code>...</code>	
<code>nm_func endp</code>	; final da função principal
<code>end nm_func</code>	; finaliza o programa assembly

OBS: o símbolo “;” é utilizado para incluir **comentários** no programa.

Exemplo de Programa

.MODEL SMALL ; modelo de memória com segmentos de 64K
.STACK 100 ; espaço de memória para instruções do prog. na pilha
.DATA
Msg db "Hello Assembly ! ",0dh,0ah,'\$'
.CODE ; linhas seguintes são instruções do programa
Prog **PROC** ;Sinaliza o início do programa (função principal)
mov ax,@data ; carrega o endereço inicial do segmento de dados em AX
mov ds,ax ; carrega o valor de AX em DS
lea dx,Msg ; obtem o endereço efetivo de Msg
mov ah,09h ; move valor 09h para o registrador AH (apresenta string)
int 21h ; chama a interrupção 21h (S.O.)
mov ah,4Ch ; move valor 4ch para o registrador AH (encerra programa)
int 21h ; chama a interrupção 21h (S.O.)
Prog **ENDP** ; encerra função principal
END Prog ;finaliza o código do programa

Modelos de Memória

- ✦ Definido pela diretiva “**.MODEL**”
- ✦ Os modelos de memória podem ser classificados nas seguintes categorias:
 - ✓ **Tiny**: código + dados + pilha $\leq 64k$
 - ✓ **Small**: código $\leq 64k$, dados $\leq 64k$
 - ✓ **Medium**: dados $\leq 64k$, código de qualquer tamanho
 - ✓ **Compact**: código $\leq 64k$, dados de qualquer tamanho
 - ✓ **Large**: código e dados de qualquer tamanho.
- ✦ No 8086 o tamanho máximo de memória é 1 MB.
 - ✓ $2^{20} = 1Mb$.

Declaração de Dados

- ★ Dados são sempre declarados na porção de dados do programa, e serão acessados via segmento de dados (DS) ou segmento extra (ES).
 - ✓ Feita após a diretiva **“.DATA”**.
- ★ Todas variáveis devem possuir um tipo de dado:
 - ✓ **DB** (*define byte*) - **8 bits**
 - ✓ **DW** (*define word*) - **16 bits**
 - ✓ **DD** (*define double word*) - **32 bits**
 - ✓ **DF** (*define far word*) - **48 bits**
- ★ **Constantes** podem ser declaradas pela **cláusula EQU**.
 - ✓ Exemplo: **LF EQU 0AH** ; LF = 0A (código ASCII para *Line Feed*).
- ★ Variáveis podem ou não possuir valores iniciais.

Declaração de Dados

★ Ex: .data

var1 **DW** 0019h

var2 **DB** ? ; ? Indica a não inicialização da variável

var3 **DB** 'a'

var4 **DB** 24,23,22

Msg **DB** "Entre com o numero:",0dh,0ah,'\$'

.code

Prog:

mov al, var4 ; al = 24 = 18h

mov al, var4+2 ; al = 23 = 16h

★ **Cláusula DUP** pode ser utilizada para duplicar um valor na inicialização de uma variável estruturada.

- ✓ **Ex:** myvet db 1000 **dup** (?) ; define um vetor de 1000 bytes não inicializados.

Declaração de Dados

- ★ Quando um programa é carregado na memória, o DOS cria e usa um segmento de memória de 256 bytes que contem informações sobre o programa.
 - ✓ **PSP - *Program Segment Prefix***
 - ✓ DOS coloca o endereço deste segmento nos registradores **DS** e **ES** antes de executar o programa.
- ★ **Problema: DS NÃO contém o endereço do segmento de dados** no início do programa.
- ★ **Solução:** colocar manualmente em **DS** o endereço correto do segmento de dados corrente.

```
MOV AX,@DATA
MOV DS,AX
```

 - ✓ **@DATA** é o nome do segmento de dados definido em **.DATA**.
 - ✓ Assembly traduz **@DATA** para o endereço inicial do segmento de dados.

Acesso a Dados

- ✦ **Diretiva **SEG****: obtém o endereço de segmento de uma variável.
 - ✓ Normalmente utilizada para obtenção do segmento de **variáveis externas** ao programa.
 - ✗ Não estão no segmento de dados do programa.
 - ✓ **Ex:** `MOV AX,seg MSG1 ; coloca em AX o endereço de ; segmento da variável msg1`

- ✦ **Diretiva **OFFSET****: obtém o endereço relativo (deslocamento) de uma variável no segmento.
 - ✓ **Ex:** `MOV DX,offset MSG1 ; coloca em DX o offset do ; endereço da variável msg1`

Transferência de Dados

- ✦ Qualquer programa precisa **movimentar dados entre dispositivos E/S, memória e registradores.**
- ✦ **Formas de transferência aceitas:**
 - ✓ Transmitir dados para um **dispositivo externo.**
 - ✓ Receber dados de um **dispositivo externo.**
 - ✓ Copiar os dados de um **registrador** para a **pilha.**
 - ✓ Copiar os dados da **pilha** para um **registrador.**
 - ✓ Copiar dados da **memória** para algum **registrador.**
 - ✓ Copiar dados de um **registrador** para a **memória.**
 - ✓ Copiar os dados de **registrador** para **registrador.**

Transferência de Dados

- ✦ Transferências estão sujeitas a regras e restrições:
 - ✓ **NÃO** pode mover dados diretamente entre posições de memória.
 - ✦ **Solução:** origem → registrador e registrador → destino.
 - ✓ **NÃO** pode mover uma constante diretamente para um registrador de segmento.
 - ✦ **Solução:** usar registrador de propósito geral como intermediário.
 - ✦ **Ex:** `mov AX,@data`
`mov DS, AX`

Instruções de E/S

- ✦ Para a comunicação com dispositivos externos são utilizados comandos específicos de E/S.
- ✦ **Comando saída (OUT):** envia dado à porta E/S.
 - ✓ Sintaxe: **OUT Port,Orig**
 - ✦ **Port:** endereço da porta de saída do dado (**DX**).
 - ✦ **Orig:** registrador de origem do dado (**AL, AX** ou **EAX**).
- ✦ **Comando entrada (IN):** recebe dado da porta E/S.
 - ✓ Sintaxe: **IN Dest,Port**
 - ✦ **Dest:** registrador de destino do dado (**AL, AX** ou **EAX**).
 - ✦ **Port:** endereço da porta de entrada do dado (**DX**).

Utilização da Pilha em Assembly

- ✦ Nas arquiteturas de processadores x86, os registradores **SP** e **BP** representam ponteiros da pilha.
 - ✓ **BP**: aponta para a base da pilha.
 - ✓ **SP**: aponta para o topo da pilha.
 - ✦ Seu valor é atualizado a cada operação de inserção ou remoção na pilha.
- ✦ A pilha cresce de cima para baixo na memória.
 - ✓ **SP** referencia o endereço mais elevado.
 - ✓ Qdo. a pilha cresce, os valores são inseridos nos endereços inferiores e **SP é decrementado em 2 posições**.
 - ✦ Informação gravada na pilha ocupa **16 bits** (uma palavra).

Instruções de Manipulação da Pilha

- ★ **Empilhamento (PUSH)**: coloca o conteúdo do operando no topo da pilha.
 - ✓ Sintaxe: **PUSH Op**
 - × **Op**: registrador que contém o valor a ser colocado na pilha.
 - × **Decrementa SP e $[SP] \leftarrow Op$.**
- ★ **Desempilhamento (POP)**: retira elemento do topo da pilha e o coloca no operando.
 - ✓ Sintaxe: **POP Op**
 - × **Op**: registrador que receberá o valor contido no topo da pilha.
 - × **$Op \leftarrow [SP]$ e incrementa SP.**
- ★ **Variações: PUSHF, PUSHA, POPF e POPA.**
 - ✓ **?F**: manipula o registrador de *flags*.
 - ✓ **?A**: manipula os registradores **DI, SI, BP, SP, AX, BX, CX e DX.**

Instruções de Transferência de Dados

★ **Instrução MOV**: copia dados da posição de origem para a posição de destino.

✓ Sintaxe: **MOV Dest,Orig**

× **Dest** contém o endereço de destino (memória ou registrador).

× **Orig** contém o endereço de origem (memória ou registrador).

✓ **Ex:** MOV AX,BX ; Copia BX em AX

MOV BX,1000h ; Copia 1000h em BX

MOV DX,[8000h]; Copia DS:8000h em DX

★ Existe variantes para **mover blocos de dados**:

✓ **MOVSB**: copia *n* bytes da origem para o destino.

✓ **MOVSW**: copia *n* palavras da origem para o destino.

× Adota **DS:SI** como origem e **ES:DI** como destino.

Modos de Endereçamento

★ **Imediato:** opera com valores constantes, embutidos na própria instrução.

- ✓ **Ex:** MOV AX,0 ; Carrega AX com 0
- MOV BX,1000h ; Carrega BX com 1000h
- MOV SI,3500h ; Carrega SI com 3500h

★ **Registrador:** quando envolve apenas registradores.

- ✓ **Ex:** MOV AX,BX ; Copia BX em AX
- MOV CX,SI ; Copia SI em CX
- MOV DS,AX ; Copia AX em DS

★ **Direto:** faz referência a um endereço fixo de memória.

- ✓ **Ex:** MOV DX,[8000h] ; EA = DS:8000h.
- ✗ SE DS = 8000h, ENTÃO DX ← (88000h)

Modos de Endereçamento

✦ **Indexado:** utiliza os registradores **BX**, **BP**, **SI** e **DI** como índices.

✓ Eles podem ser usados **sozinhos ou combinados**.

✦ Valor da **soma** de **BX** ou **BP** com **SI** ou **DI**, ou com uma **constante**.

✓ **Ex:** MOV CL,[BX]

MOV DL,[BP]

MOV DL,[BP+50]

MOV AL,[SI+100]

MOV AX,[BX+SI]

MOV AH,[BP+DI]

MOV DX,[BP+DI+300]


MOV AH,[BP+SI+2000]

Acesso aos Dados

- ★ Um mesmo dado pode ser acessado de **vários modos**:

Ex:

```
.data
TEXTO DB 'ABCDEF'
.code
mov ax,@data
mov ds,ax
mov si,OFFSET TEXTO + 3
mov al, [si]
mov bx,3
mov al, [TEXTO + bx]
mov bx,OFFSET TEXTO
mov al, [bx + 3]
mov si,3
mov al, [texto + si]
mov di, 3
mov al, [bx + di]
```



```
mov si,OFFSET TEXTO
mov bx, 3
mov al, [bx + si]
mov bx, OFFSET TEXTO
mov si, 2
mov al,[bx + si + 1]
mov bx, 1
mov si, 2
mov al, [texto + bx + si]
```

Acesso aos Dados

★ Quando for necessário **explicitar o tamanho do dado a transferir** deve-se utilizar as diretivas:

- ✓ **BYTE PTR** : transfere 8 bits.
- ✓ **WORD PTR**: transfere 16 bits.

★ **Exemplo:** movimentação de dados com 8 ou 16 bits

.data

nro dw 1234h ; *little endian* (00 = 34h, 01 = 12h)

.code

mov al,byte ptr [nro] ; al = 34h

mov ah,byte ptr [nro+1] ;ah = 12h

mov bx,word ptr [nro] ;bx = 1234h

mov cx,word ptr [nro+1] ;cx = 0012h

Instruções Aritméticas

- ★ **Instrução NEG**: inverte o sinal do valor aritmético especificado (utilizando o **complemento de 2**).
 - ✓ **Ex:** NEG AL
 NEG AX
 NEG byte ptr [BX+SI]
- ★ **Instruções ADD e ADC**: soma os dois operandos, colocando o resultado no primeiro operando.
 - ✓ **ADC** também soma o **bit Carry**, usado para o “vai 1”, possibilitando formar dados maiores.
 - ✓ Pode ser com 8 ou 16 bits (depende do operando).
 - ✓ **Ex:** ADD BX,SI
 ADC AH,[BP+SI+3]

Exemplo: Soma de 2 n^o Extensos

```
.model small
```

```
.stack 100h
```

```
.data
```

```
    nro1 dd 00012345h
```

```
    nro2 dd 00054321h
```

```
    soma dd 00000000h
```

```
.code
```

```
...
```

```
mov ax,@data
```

```
mov ds,ax
```

```
mov ax, word ptr[nro1]
```

```
mov dx, word ptr [nro1+ 2]
```

```
add ax,word ptr[nro2]
```

```
adc dx,word ptr [nro2+2]
```

```
mov word ptr [soma],ax
```

```
mov word ptr [soma+2],dx
```

```
...
```

```
; obtém o segmento de dados
```

```
;ax = 2345h (byte – significativo de NRO1)
```

```
;dx = 0001h (byte + significativo de NRO1)
```

```
;ax = 2345h + 4321h
```

```
;dx = 0001h + 0005h + 0000h (bit CARRY)
```

```
; guarda byte – significativo de SOMA
```

```
; guarda byte + significativo de SOMA
```


Instruções Aritméticas

- ★ **Instruções SUB e SBB**: subtrai dois operandos, colocando o resultado no primeiro operando.
 - ✓ SBB subtrai também o valor do **bit Carry**, tornando possível a operação de “vem um” (*borrow*) com n^o maiores.
 - ✓ **Ex:** SUB BX,DX
 SBB AX,[BX+DI]
- ★ **Instruções MUL e IMUL**: multiplica o acumulador (**AX** ou **AL**) por um operando na memória ou em outro registrador.
 - ✓ Escolha do ACC depende do tamanho do operando multiplicador.
 - ✓ **MUL** é usada para números sem sinal (só +).
 - ✓ **IMUL** aceita números inteiros (+ ou -).
 - ✓ Resultado é guardado em ACC maior (**AH** → **AX** → **DX** e **AX**).
 - ✓ **Ex:** MUL CL
 MUL word ptr [SI]
 IMUL DX

Instruções Aritméticas

- ✦ **Instruções **DIV** e **IDIV****: divide o acumulador (**AX** ou **DX** e **AX**) por um operando de 8 ou 16 bits.
 - ✓ **DIV** é usada para números sem sinal (só +).
 - ✓ **IDIV** aceita números inteiros (+ ou -).
 - ✓ Dividendo é definido pelo tamanho do divisor.
 - ✦ Divisor de **8 bits** \Rightarrow **AX** \div **Op** \rightarrow **AL** e o resto \rightarrow **AH**.
 - ✦ Divisor de **16 bits** \Rightarrow **DX:AX** \div **Op** \rightarrow **AX** e o resto \rightarrow **DX**.
 - ✓ Se quociente não cabe no registrador, a operação gera um estouro de divisão (***divide overflow***).
 - ✓ **Ex:** **DIV CL**
 IDIV byte ptr [BP+4]

Instruções Aritméticas

- ★ **Instruções INC e DEC:** incrementa ou decrementa de uma unidade o operando especificado.
 - ✓ **Bit Zero** é afetado, possibilitando implementar contadores.
 - ✓ **Ex:** Preencher a tela com 2000 caracteres em branco.

```
MOV DX,2000 ; N° de bytes a serem enviados
ENVIA: MOV AL, 20h ; 20h é o código ASCII do caractere " ".
CALL OUTCHAR ; Envia o caractere para o vídeo
DEC DX ; Decrementa o contador
JNZ ENVIA ; Pula se não chegou a zero
```
- ✓ Instruções **INC** e **DEC** também podem ser usadas para implementar ponteiros para posições de memória.
 - ✗ Útil quando queremos manipular dados seqüenciais.

Instruções Aritméticas

★ **Instrução CMP**: comparar dois valores.

✓ Realiza uma **subtração entre os operandos**, alterando os valores dos flags necessários

✓ **Exemplos:**

CMP AL,57H ; Compara o conteúdo de **AL** com 57h.

CMP DI,BX ; Compara os conteúdos de **DI** e **BX**.

CMP [SI],AX ; Compara uma **palavra** gravada em ; **DS:SI**, com o conteúdo de **AX**.

CMP CH,[SI+BX+3] ; Compara o conteúdo de **CH** com ; o **byte** gravado em **DS:SI+BX+3**

Instruções Lógicas

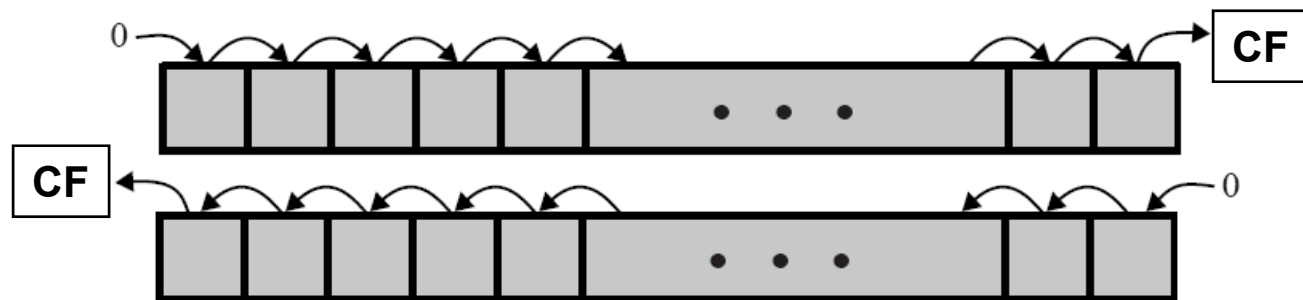
- ★ **Instrução NOT**: inverte todos os bits do dado.
 - ✓ Operação **unária** (somente um operando).
 - ✓ Bit **1** vira **0** e bit **0** vira **1**.
 - ✓ Sintaxe: **NOT Op**
- ★ **Instruções AND, OR e XOR**: realiza as operações lógicas tradicionais “**E**”, “**OU**” e “**OU exclusivo**”.
 - ✓ Operações **binárias** (possuem 2 operandos).
 - ✓ **AND** pode ser usada para separar os bits de interesse.
 - × 2º operando é a máscara de bits.
 - ✓ **OR** pode ser usada para incluir bits 1 ao operando.
 - ✓ **Ex:** `AND AL,0Fh` ; conversão *ASCII* → N° inteiro.
`OR AL, 30h` ; conversão N° inteiro → *ASCII*.

Deslocamentos e Rotações

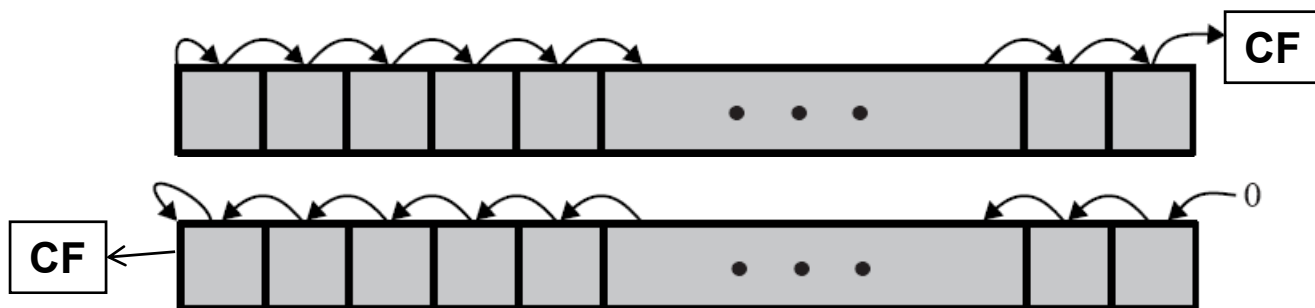
- ★ **Instruções SAL e SAR:** realiza o deslocamento aritmético N posições para a esquerda (**SAL**) ou para a direita (**SAR**).
 - ✓ Bit + a direita recebe zero, bit + a esquerda recebe o bit do sinal
- ★ **Instruções SHL e SHR:** realiza o deslocamento lógico N posições para a esquerda (**SHL**) ou direita (**SHR**).
 - ✓ Novo bit recebe zero.
- ★ **Instruções ROL e ROR:** rotaciona os bits do operando para a esquerda ou direita N posições.
 - ✓ RCL e RCR são variações que incluem o bit **Carry** na rotação.
- ★ **Aspectos Gerais:**
 - ✓ Sintaxe: **??? Op, N**
 - × **???**: instrução a ser realizada.
 - × **Op**: operando que sofrerá a operação (deslocamento ou rotação).
 - × **N**: quantidade de bits deslocados ou rotacionados (1 ou valor contido em **CL**).
 - ✓ **Bit eliminado** vai para o bit **Carry**, sobrepondo seu valor anterior.

Deslocamentos e Rotações

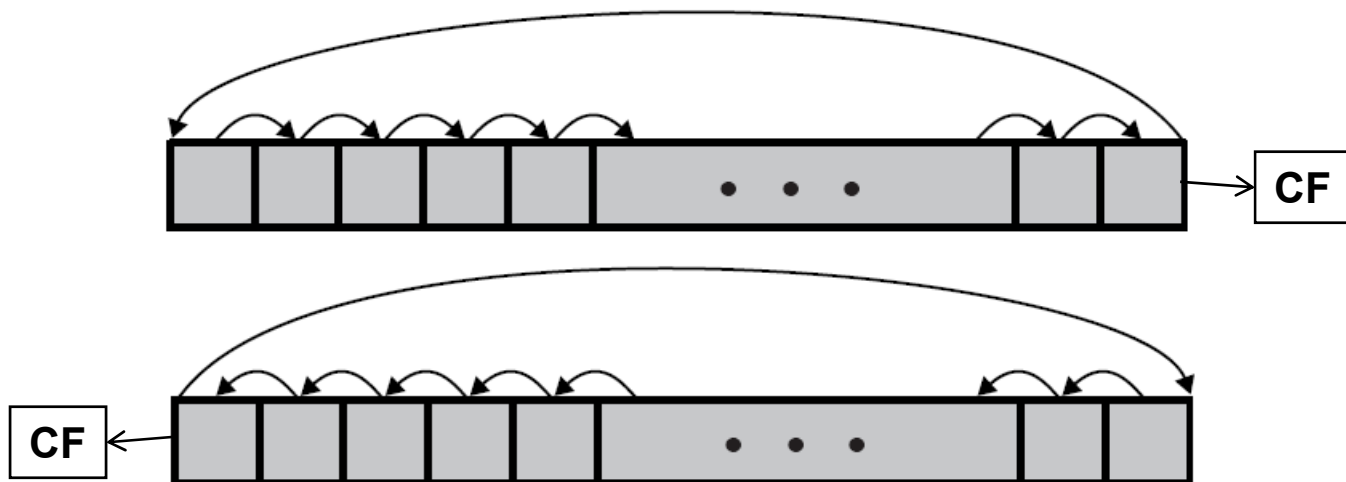
Deslocamento Lógico



Deslocamento Aritmético



Rotação Simples



Instruções de Chamada de Sub-rotinas

- ✦ **Instrução CALL:** chama uma sub-rotina, alterando o fluxo normal de execução.
 - ✓ Endereço de retorno é colocado na pilha pela instrução.
 - ✗ Quando uma instrução CALL é executada, o conteúdo de PC é armazenado na pilha (empilhado).
 - ✓ Sintaxe: **CALL Proc**
 - ✗ **Proc:** nome do procedimento (sub-rotina) a ser executado.

- ✦ **Instrução RET:** encerra uma sub-rotina, retomando a execução do programa chamador da sub-rotina.
 - ✓ Transfere o fluxo de processamento para a instrução seguinte à chamada da sub-rotina.
 - ✗ Desempilha o endereço armazenado na pilha e o coloca no registrador PC.
 - ✓ Sintaxe: **RET**

Fluxo da Chamada de Sub-rotina

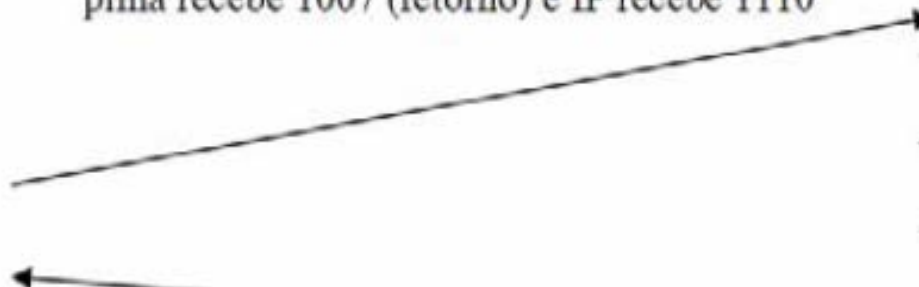
Arquitetura de Computadores (17)

End.	Instr.
1000	mov al,1
1002	mov bl,3
1004	call calculo
1007	mov ah,2
1009	int 21h

pilha recebe 1007 (retorno) e IP recebe 1110

End.	Instr.
1110	shl al,1
1112	add al,bl
1114	and al,7
1116	add al,'0'
1118	ret

IP recebe 1007 (topo da pilha)



Exemplo: Impressão de Texto

```
.model small
.stack 200h
.data
    mens1 db 'Bom dia',13, 10, 0
    mens2 db 'Boa tarde', 13, 10, 0
    mens3 db 'Boa noite', 13, 10, 0
.code
inicio PROC NEAR
    mov ax,@data
    mov ds,ax
    mov bx, OFFSET mens1
call imprime
    mov bx, OFFSET mens2
call imprime
    mov bx, OFFSET mens3
call imprime
    mov ah,4ch
```

```
→ int 21h
inicio ENDP

imprime proc near
repete:
    mov dl,[bx]
    and dl,dl
    jz fim
    inc bx
    mov ah,02h
    int 21h
    jmp repete

fim:
    ret
imprime endp
end inicio
```

Instruções de Desvio e Salto

- ✦ Desvios e saltos podem ser implementados por uma das instruções de **JUMP** ou pelo comando **LOOP**.
 - ✓ Normalmente adiciona-se rótulos (*label*) às instruções para **indicar os pontos de desvio**.
 - ✓ Instrução **JMP**: realiza o **desvio incondicional** no fluxo de execução do programa.
 - ✦ Sintaxe: **JMP Dest**
 - **Dest**: próxima instrução a ser executada.
 - ✓ Instruções **J??**: realizam **desvios condicionais** no fluxo de execução do programa de acordo com os **bits de flag**.
 - ✦ Existe várias instruções deste tipo, algumas gerais, outras específicas para n° cardinais (sem sinal) ou n° inteiros (com sinal).
 - ✦ Sintaxe: **J?? Dest**

Exemplos de Desvios Condicionais (J??)

Unsigned (Cardinal)				signed (Integer)			
JA	Jump if Above	JA Dest	(≡ JNBE)	JG	Jump if Greater	JG Dest	(≡ JNLE)
JAE	Jump if Above or Equal	JAE Dest	(≡ JNB ≡ JNC)	JGE	Jump if Greater or Equal	JGE Dest	(≡ JNL)
JB	Jump if Below	JB Dest	(≡ JNAE ≡ JC)	JL	Jump if Less	JL Dest	(≡ JNGE)
JBE	Jump if Below or Equal	JBE Dest	(≡ JNA)	JLE	Jump if Less or Equal	JLE Dest	(≡ JNG)
JNA	Jump if not Above	JNA Dest	(≡ JBE)	JNG	Jump if not Greater	JNG Dest	(≡ JLE)
JNAE	Jump if not Above or Equal	JNAE Dest	(≡ JB ≡ JC)	JNGE	Jump if not Greater or Equal	JNGE Dest	(≡ JL)
JNB	Jump if not Below	JNB Dest	(≡ JAE ≡ JNC)	JNL	Jump if not Less	JNL Dest	(≡ JGE)
JNBE	Jump if not Below or Equal	JNBE Dest	(≡ JA)	JNLE	Jump if not Less or Equal	JNLE Dest	(≡ JG)
JC	Jump if Carry	JC Dest		JO	Jump if Overflow	JO Dest	
JNC	Jump if no Carry	JNC Dest		JNO	Jump if no Overflow	JNO Dest	
				JS	Jump if Sign (= negative)	JS Dest	
				JNS	Jump if no Sign (= positive)	JNS Dest	

JUMPS (flags remain unchanged)							
Name	Comment	Code	Operation	Name	Comment	Code	Operation
JE	Jump if Equal	JE Dest	(≡ JZ)	JNE	Jump if not Equal	JNE Dest	(≡ JNZ)
JZ	Jump if Zero	JZ Dest	(≡ JE)	JNZ	Jump if not Zero	JNZ Dest	(≡ JNE)
JCXZ	Jump if CX Zero	JCXZ Dest		JECXZ	Jump if ECX Zero	JECXZ Dest	386
JP	Jump if Parity (Parity Even)	JP Dest	(≡ JPE)	JNP	Jump if no Parity (Parity Odd)	JNP Dest	(≡ JPO)
JPE	Jump if Parity Even	JPE Dest	(≡ JP)	JPO	Jump if Parity Odd	JPO Dest	(≡ JNP)

Instruções de Desvio e Salto

★ **Instrução LOOP**: realiza desvios para a construção de **laços de repetição** (iteração) no programa.

✓ Decrementa o valor de **CX**, e se **NÃO for zero**, desvia para o *label*.

✓ Sintaxe: **LOOP Dest**

✓ **Ex:** MOV CX,10 ; Contador = 10

 MOV SI,1000 ; SI aponta para endereço 1000 da memória

 MOV DI,2000 ; DI aponta para 2000

TRANSF: MOV AL,[SI] ; Pega um byte da origem

 MOV [DI],AL ; Guarda no destino

 INC SI ; Incrementa ponteiros

 INC DI

 LOOP TRANSF ; Dec **CX** e se **≠ zero** vai para **TRANSF**

★ **Variações: LOOPE, LOOPNE, LOOPZ e LOOPNZ.**

✓ Fazem um teste no **bit Zero** do registrador de flags:

× Se a condição for satisfeita, executa o LOOP.

× Caso contrário, termina a iteração.

Outras Instruções

- ★ Instruções **CBW** e **CWD**: realiza a conversão do tipo byte para word e de word para double word, respectivamente.
 - ✓ **CBW** expande o conteúdo de **AL** para **AX**.
 - ✓ **CWD** expande o conteúdo de **AX** para **DX:AX**.
 - ✓ Trabalham sobre n° inteiros (com sinal) em complemento 2.
 - ✓ Sintaxe: **C??**

- ★ Instrução **XCHG**: troca o valor dos operadores.
 - ✓ Sintaxe: **XCHG Op1,Op2**

- ★ Instrução **LEA**: obtém o endereço efetivo de uma variável ou rótulo.
 - ✓ Equivalente ao **&** na linguagem C.
 - ✓ Sintaxe: **LEA Dest,Orig**
 - × **Dest** recebe o endereço de **Orig**.

Interrupções da BIOS

- ★ **Instrução INT:** executa uma interrupção de SW.
 - ✓ Primeiros 1024 bytes da memória são reservados para o **vetor de interrupções**, com **256 elementos**.
 - × Cada elemento é composto de **4 bytes** (2 para indicar um **segmento** e 2 para indicar um **offset**).
 - × Corresponde ao **endereço de uma função do S.O.** encarregada de um determinado serviço.
- ★ **Exemplos:**
 - ✓ Interrupção **10h**: placa de vídeo.
 - ✓ Interrupção **16h**: teclado
 - ✓ Interrupção **21h**: serviços do DOS.
 - ✓ Interrupção **33h**: mouse.
- ★ **Capítulo 13 da apostila *The Art of Assembly*** descreve as chamadas de serviço da BIOS (**INT 16h** e **INT 10h**), usadas para leitura e escrita de dados fornecidos pelo usuário.

Interrupção 21h

- ✦ Utilizada no MS-DOS para várias chamadas de **funções básicas de acesso a disco e E/S**.
- ✦ **Comando: `int 21h`** ;invoca a interrupção do DOS
- ✦ Registrador **AH** indica qual é a operação desejada:
 - ✓ **Leitura de um caractere do teclado (`AH = 01h`)**
 - ✦ Saída: **AL** = caractere (*ASCII* em hexa)
 - ✓ **Escrita de um caractere na tela (`AH = 02h`)**
 - ✦ Entrada: **DL** = caractere a ser escrito
 - ✦ Saída: nenhuma
 - ✓ **Escrita de uma string na tela (`AH = 09h`)**
 - ✦ Entrada: **DX** = endereço para o início da string
 - ✦ Saída: nenhuma
 - ✦ O final da string deve ser determinado pelo caractere '\$'.
 - ✓ **Encerra o programa e retorna ao S.O. (`AH = 4Ch`)**
 - ✦ Saída: nenhuma

Exercícios

1. Escreva um programa que mostre na tela os 256 caracteres do código ASCII.
2. Escreva um programa que receba dois números entre 0 e 9 do teclado e apresente o maior deles.
3. Escreva um programa que receba um número inteiro e retorne se o número é par ou ímpar.
4. Escreva um programa que recebe uma *string* de no máximo 30 caracteres e a escreva com letras maiúsculas (obs: tecla **ENTER** encerra a *string*).
5. Escreva um programa que receba uma expressão aritmética na forma infixa (ex: $A+B*C$) e a retorne na forma pós-fixa (ex: $ABC*+$).