

PARTE 4

ESTUDO DE CASO - INTEL FAMÍLIA X86

Depois de entendermos a organização dos computadores, entrarmos em contato com os principais comandos em linguagem de montagem e compreendermos como ocorre a execução de um programa na unidade central de processamento; chegou a hora de focarmos o estudo em uma arquitetura de mercado e assim, aprendermos a programar nesta determinada arquitetura.

Para um primeiro estudo, escolhemos a arquitetura Intel da família x86. Além da penetração de mercado, o principal motivador para a escolha dessa família de processadores foi a facilidade do estudo e aprendizagem desta arquitetura e de seus comandos básicos.

Bons Estudos!

CAPÍTULO 13 – O PROCESSADOR INTEL® X86

Objetivos do Capítulo

Após a leitura deste capítulo o leitor estará apto a:

- trabalhar com processadores, utilizando sua linguagem de montagem;*
- desenvolver rotinas otimizadas e incluí-las em código escrito em linguagem de alto nível;*
- colocar em prática os conceitos vistos nos capítulos anteriores;*
- conhecer a família INTEL x86.*

FUNDAMENTOS

13.1 A Família INTEL® x86

Como vimos na Parte 3, podemos definir um microprocessador como um circuito integrado onde estão implementadas as funções de uma Unidade Central de Processamento. São exemplos de microprocessadores os da família INTEL® x86, os da família AMD, os da família PowerPC, entre outros.

Utilizaremos como objeto de estudo de caso, um dos processadores da família INTEL.

A família x86, da INTEL®, são microprocessadores, chamados apenas de processadores, utilizados na maioria dos computadores pessoais, desde a década de 1980. Os processadores 8086, 8088, 80186, 80188, 80286, 80386, 80486, Pentium®, pertencem a esta família.

Os processadores 8086 e 8088 foram os primeiros microcomputadores de 16 bits introduzidos no final da década de 1970 pela INTEL®. A diferença entre estes dois processadores estava na largura do formato de dados, enquanto o 8086 trabalhava com dados de 16 bits, o 8088 trabalhava com dados de 8 bits, apesar de internamente os dois serem iguais.

Os processadores 80186 e 80188 são versões melhoradas dos anteriores, com a vantagem de, além de suportarem outras funções, podiam executar um conjunto maior de instruções, chamada de *"extended instruction set"*.

O processador 80286, de 1982, é também um processador de 16 bits, mais rápidos que os anteriores e com a vantagem de ter dois modos de operação: modo de endereçamento real, como os anteriores e o modo de endereçamento virtual ou endereçamento protegido, que dá suporte a processamento multitarefa (várias tarefas sendo executadas simultaneamente), protegendo a memória utilizada por um programa da ação de outro. No modo protegido é possível endereçar um espaço maior de memória, que o processador 8086.

Os processadores 80386 e o 80386SX, de 1985, foram os primeiros processadores de 32 bits da INTEL®, são mais rápidos que os anteriores, pois trabalham com barramento de 32 bits e com um *clock* de frequência maior, possibilitando que a execução de uma instrução seja feita mais rápida. Estes processadores também trabalham no modo real ou protegido. No modo protegido pode emular o 80286. Tem também o modo 8086 virtual, projetado para executar múltiplas aplicações compatíveis com o 8086. O 80386SX tem a mesma estrutura do processador 386, só que com barramento de dados de 16 bits.

Os processadores 80486 e 80486SX, de 1989, também são processadores de 32 bits, e como era de se esperar, com maior capacidade de processamento que os anteriores. O 386 pode operar junto com um co-processador aritmético (80387), para operações de ponto flutuante, enquanto que no 486, as funções de ponto flutuante são implementadas no próprio processador. O 486SX tem a mesma estrutura do processador 486, mais sem as funções do processador de ponto flutuante.

Os processadores Pentium® (*pente* em grego significa quinto), de 1993, pertencem geração que sucedeu os 486, tanto que inicialmente seriam chamados de 586. A escolha do nome foi apenas uma questão de possibilidade de registro. como números não podem ser usados para registrar a marca, eles utilizaram o nome Pentium®. Apesar de serem

processadores CISC, são processadores superescalares, ou seja, processadores com duplicidades nos caminhos de execução, permitindo que em um ciclo de *clock*, tenha terminado a execução mais de uma instrução. O Pentium® é um processador de 32 bits, mas suporta um barramento de dados externo de 64 bits, permitindo com que a leitura da memória seja feita com maior rapidez. Outra característica importante é o conjunto de instruções MMX (*Extensão Multimídia - MultiMedia eXtension*), que são instruções que permitem trabalhar com múltiplos dados (*SIMD - Single Instruction, Multiple Data*), imprescindível para aplicações multimídia. O processador Pentium® Pro, de 1995, foi o primeiro processador da INTEL® a ter um núcleo RISC (*Reduced Instruction Set Computer*), fazendo com que melhorasse seu desempenho, se comparado com os processadores Pentium® tradicionais. Estes processadores tinham a memória cache nível 2 integrado no próprio processador.

Os processadores INTEL® atuais são multicore que significa ter mais de um núcleo processador integrado em um mesmo circuito integrado. O primeiro processador multicore INTEL®, surgiu em 2005, foi o Pentium® D, que é um Dual-Core (2 cores em um único circuito), que deu o início dos chamados processadores Intel® Core™2 Duo. Com o avanço da tecnologia multicore surgiram os processadores de quatro núcleos, os chamados Intel® Core™2 Quad em 2007.

Os processadores Celerom® são processadores de baixo custo, baseados nos Pentium® e nos processadores Core™2 Duo. São processadores de menor custo e conseqüentemente de menor desempenho, pois o tamanho da *cache* nível 2 é menor, trabalha numa frequência de clock interna menor.

Os processadores Intel® Xeon® são processadores de alto desempenho específicos para servidores. Eles têm mais níveis *cache* e suportam o multiprocessamento (várias execuções de processos de maneira simultânea), através de projetos que utilizam mais de um processador na placa-mãe.

Os processadores Intel® Atom™ são processadores projetados para utilização em dispositivos portáteis que necessitam ter um baixo consumo de energia, tais como computadores *netbooks*, *smartphones* entre outros.

13.2 Arquitetura do processador 8086

O processador 8086 é um processador com uma estrutura e um conjunto de instruções simples, facilitando o aprendizado da programação em linguagem de montagem. Este aprendizado será a base, não só para os processadores da família x86, como também para outros processadores. Como vimos anteriormente, para podermos programar em linguagem de montagem, temos que conhecer, não só o conjunto de instruções do processador como também sua organização. A figura 13.1 mostra a organização do processador 8086.

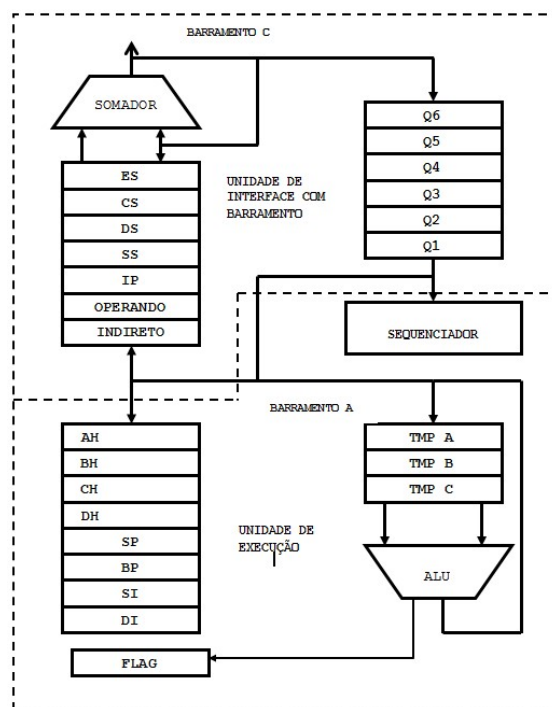


Figura 13.1 - Organização do processador 8086

(adaptado do *Intel Microprocessors Documentation*. Internet: <http://intel.com>).

Unidades do processador 8086

A figura 13.1 mostra que o processador 8086 divide-se internamente em duas unidades: a Unidade de Execução e a Unidade de Interface com o Barramento.

- **A Unidade de Execução (*Execution Unit - EU*)**

Esta unidade é responsável pela execução das instruções. Nela encontramos a Unidade Lógica e Aritmética - ULA, que executa as operações aritméticas (soma, subtração, divisão e multiplicação) e as operações lógicas (AND, OR, NOT e XOR).

- **Unidade de Interface com o Barramento (*BUS Interface Unit - BIU*)**

Esta unidade é responsável pela comunicação de dados entre a Unidade de Execução e o meio externo (memória, unidade de E/S). Controla a transmissão de sinais de endereços, dados e controle, e com isso a sequência de busca e execução de instruções. No processador 8086, em vez de fazer de fazer o *fetch* de uma instrução somente após a execução da anterior, ele lê até 6 instruções, armazenando-as na fila de instruções (*instruction queue*). Chamamos isto de *pre-fetch* e ele tem o objetivo de aumentar a velocidade de execução de um programa.

- **Registadores do processador 8086**

Os registradores do processador 8086, são divididos em registradores de propósito geral ou de dados e registradores específicos, como vimos no capítulo 11 deste livro. No caso do 8086, os registradores específico armazenam endereços (segmentos, apontadores e índices) e sinalizadores de estado e controle (FLAGS).

- **Registradores de propósito geral (ou de dados).**

O 8086 tem 4 registradores de 16 bits que também podem ser usados como registradores de 8 bits. São eles os registradores AX, BX, CX e DX, que são utilizados nas operações aritméticas e lógicas. Quando utilizados como registradores de 8 bits, os registradores passam a ter em seu nome as letras L, para identificar o byte menos significativo ou inferior, e H para o byte mais significativo ou superior. Em outras palavras se:

- AX = 4B3Ch → AH = 4Bh e AL = 3Ch
- BX = 3456h → BH = 34h e BL = 56h
- CX = 1267h → CH = 12h e CL = 67h
- DX = ABFFh → DH = ABh e DL = FFh

Apesar de serem registradores de propósito geral, eles também têm algumas funções especiais, como registradores de dados, definidas pelas instruções:

- Registrador AX (Acumulador): é utilizado como acumulador em operações aritméticas e lógicas, em instruções de E/S e em instruções de operações BCD para ajuste decimal.
 - Registrador BX (base): é usado como registrador base para referenciar posições de memória, ou seja, armazena o endereço base de uma tabela ou vetor de dados, a partir do qual as posições são obtidas adicionando-se um valor de deslocamento (*offset*).
 - Registrador CX (contador): é utilizado em operações iterativas e repetitivas para "contar" bits, bytes ou palavras, podendo ser
-

incrementado ou decrementado. A parte menos significativa de CX, o CL, funciona como um contador de 8 bits.

- **Registrador DX (dados):** é utilizado em operações de multiplicação para armazenar parte de um produto de 32 bits, ou em operações de divisão de 32 bits, para armazenar o resto. Também utilizado em operações de E/S para especificar o endereço de uma porta de E/S.

Quando tratarmos das instruções, especificamente, mostraremos exemplo de utilização dos registradores nestas funções.

- **Registradores com funções específicas.**

Registradores de Segmentos

Como o 8086 trabalha com blocos de 64K bytes de memória, ele necessita de registradores para armazenar o endereço base destes segmentos. Estes registradores são chamados de registradores de segmentos. São 4 os tipos de segmentos: de código, de dados, de pilha e extra, que é um segundo segmento de dados.

Os registradores de segmento do 8086, todos de 16 bits são:

- **Registrador de Segmento de Código - CS (*Code Segment*);**
- **Registrador de Segmento de Dados - DS (*Data Segment*);**
- **Registrador de Segmento de Pilha - SS (*Stack Segment*);**
- **Registrador de Segmento Extra de Dados - ES (*Extra Segment*).**

O endereçamento no 8086 é diferenciado para o código de programa (instruções), para os dados armazenados na memória principal e para os dados armazenados na pilha. Cada um destes tipos são alocados em segmentos (bloco de memória de 64 Kbytes, para o 8086) endereçáveis. Durante a execução de um programa no 8086, há 4 segmentos ativos:

- segmento de código endereçado por CS;
- segmento de dados endereçado por DS;
- segmento de pilha endereçado por SS;
- segmento extra (de dados) endereçado por ES;

Como estes registradores armazenam o endereço base de seu respectivo segmento, necessitamos de registradores para armazenar o deslocamento ("offset") dentro deste segmento (figura 13.2).

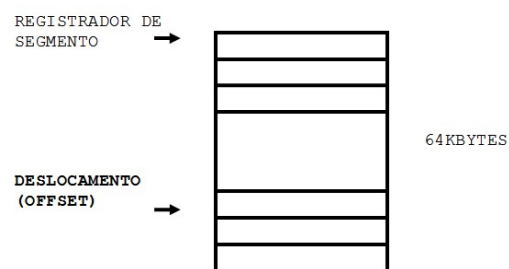


Figura 13.2 - Segmento: Endereço Base e deslocamento (offset)

Esses registradores, também de 16 bits são:

- **Registrador IP** (*Instruction Pointer*) é o Program Counter - PC do INTEL x86. É utilizado, em conjunto com CS, para localizar a posição, dentro do segmento de código corrente, da próxima instrução a ser executada. Ele é automaticamente incrementado em função do número de bytes da instrução executada, após o *fetch* da instrução que está sendo executada.
 - **O registrador SP** (*stack pointer*) é utilizado, em conjunto com registrador SS, para acessar a área de pilha na memória. Ele sempre aponta para o topo da pilha.
-

- **O registrador BP** (*base pointer*) é um ponteiro que, em conjunto com SS, permite acesso de dados dentro do segmento de pilha, como se fosse um vetor de dados.
- **O registrador SI** (*source index*) usado como registrador de índice em alguns modos de endereçamento indireto, em conjunto com DS. Por exemplo, em instruções específicas de manipulação de *string* ou em acesso a vetores armazenados no segmento de dados. Pode também ser usado como registrador de dados de 16 bits.
- **O registrador DI** (*destination index*) similar ao SI, atuando em conjunto com ES, quando utilizado em instruções de manipulação de *string*, ou com DS em acesso a vetores armazenados no segmento dados. Como o SI, pode também ser usado como registrador geral de dados de 16 bits.
- **O Registrador de sinalizadores (FLAGS)** é um registrador de 16 bits, que tem a função de indicar o estado do processador após a execução de uma instrução. Os bits subdividem-se em dois grupos, 6 bits para os *flags* da estado (*status*) e 3 bits para os *flags* de controle. Os demais bits não são utilizados e servem para "futuras implementações". A seguir faremos um estudo detalhado da função de cada bit de estado e como utilizá-los

Status do Processador e os Sinalizadores (FLAGS)

Uma importante característica que distingue um computador de outras máquinas é a habilidade do computador de tomar decisões. Os circuitos da UCP podem realizar decisões simples, baseadas no estado atual do processador. Para o processador 8086, seu estado é verificado através de 9 bits individuais chamados de sinalizadores ou FLAGS. Cada decisão tomada pelo 8086 é baseada nos valores desses FLAGS.

Os **FLAGS** estão localizados no registrador sinalizador e eles estão classificados em **FLAGS DE STATUS** e **FLAGS DE CONTROLE**.

- Os **FLAGS DE STATUS** refletem o resultado de uma operação aritmética ou lógica.
- Os **FLAGS DE CONTROLE** são utilizados para habilitar e desabilitar certas operações do processador.

Na figura 13.3 temos o registrador de **FLAGS**, com sua organização:

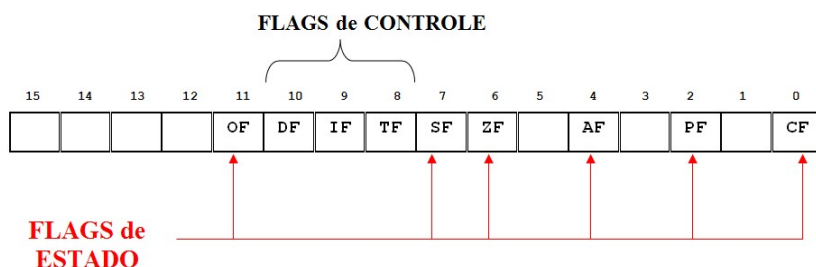


Figura 13.3 - O registrador **FLAG**

A tabela 13.1, mostra os **FLAGS** e seus significados.

FLAGS DE ESTADO	
CF - Flag de Carry	Indica a ocorrência de <i>carry out</i> após a execução de uma instrução aritmética. Se CF = 1 ocorreu o "vai um" (adição) ou o "empréstimo" (subtração). Se CF = 0 não ocorreu.
PF - Flag de paridade	Caso o byte inferior do resultado de alguma operação aritmética ou lógica apresentar um número par de "1's", PF = 1 (paridade par), caso contrário, PF = 0 (paridade impar).

AF - <i>Flag</i> de <i>Carry Auxiliar</i> - Operações BCD	Se AF = 1 existiu o "vai um" do bit 3 para o bit 4 de uma adição de números BCD ou caso exista "empréstimo" do bit 4 para o bit 3 numa subtração de números BCD. Se AF = 0 não ocorreu.
ZF - <i>Flag</i> Zero	caso o resultado da última operação aritmética ou lógica seja igual a zero, ZF = 1. Caso contrário ZF = 0.
SF - <i>Flag</i> de Sinal	Utilizado para indicar se o resultado é positivo, SF = 1, ou negativo, SF = 0. Usado também para indicar a ocorrência de <i>overflow</i> em operações em Complemento de 2.
OF - <i>Flag</i> de <i>overflow</i>	Para indicar a ocorrência, OF = 1, ou não, OF = 0, de <i>overflow</i> em operações de números sinalizados.
FLAGS DE CONTROLE	
TF - <i>Flag</i> de <i>Trap</i>	Quando forcamos TF = 1, após a execução da próxima instrução, ocorrerá uma interrupção na execução do programa. A própria interrupção faz TF = 0. Se TF=0, a execução será normal. Utilizada nos <i>debuggers</i> (programas que auxiliam a correção de erros).
IF - <i>Flag</i> de Interrupção	Habilita a ocorrência de interrupções, quando IF = 1, se IF = 0, desabilita a ocorrência de interrupções.
DF - <i>Flag</i> de Direção	Usado para indicar a direção em que as operações com <i>strings</i> são realizadas. Se DF = 1 o endereço de memória é decrementado e se DF = 0 é incrementado.

Tabela 13. 1- FLAGS do processador 8086

OVERFLOW

Para a verificação da ocorrência de erro de *overflow*, temos que saber se a operação foi com número sinalizado ou não sinalizado:

- **Overflow decorrente de operações com números não-sinalizados**

Nas operações de adição, um *overflow* não-sinalizado ocorre quando há vai 1 no MSB. Isto significa que a resposta (resultado) é maior do que o tamanho do número, ou seja, maior que FFFFh para uma palavra e maior que FFh para um byte.

Nas operações de subtração, um *overflow* não-sinalizado ocorre quando há um "empréstimo" no MSB. Isto significa que a resposta (resultado) é menor que 0 (zero). Em ambos os casos, basta verificar o FLAG CF.

- **Overflow decorrente de operações com números sinalizados**

Na adição de números com o mesmo sinal, um *overflow* sinalizado ocorre quando o resultado da soma tem um sinal diferente. Isto acontece quando estamos somando, por exemplo, dois números positivos e o resultado é um número negativo (ex: 7Fh + 7Fh = FEh)

Observação:

A subtração de números com sinais diferentes é como a adição de números com o mesmo sinal. Por exemplo:

$$A - (-B) = A + B \quad \text{e} \quad -A - (+B) = -A + (-B)$$

Na adição de números com sinais diferentes, *overflow* é impossível, pois a soma torna-se uma subtração, por exemplo, $A + (-B) = A - B$. Como A e B podem ser representados pelo número de bits dos registradores, o resultado da diferença entre eles também poderá ser. Analogamente, subtração de números com o mesmo sinal não podem resultar em *overflow*. Por isto, para verificarmos a ocorrência de *overflow* em operações sinalizadas, temos que verificar os *flags* CF e OF. O FLAG CF indica se

há um vai-um na posição do MSB e o FLAG OF verifica o vem-um que chega e o vai-um gerado no MSB. Se iguais (0 e 0 ou 1 e 1), teremos OF = 0 se diferentes, teremos OF = 1.

Exemplo 1:

ADD AL,BL ; AL \leftarrow AL + BL e AL contem FFh e BL contem 01h

		representação não-sinalizada	representação sinalizada
FFh	1111 1111b	255	-1
01h	+ 0000 0001b	+ 1	+1
	<u>1 0000 0000b</u>	256 (fora da faixa)	0 (OK)

Logo após a execução da instrução:

CF = 1 e OF = 0, pois no MSB o "vem-um" é igual ao "vai-um" (ambos 1). Neste caso temos um overflow sinalizado e não temos um overflow não sinalizado.

Exemplo 2:

Suponha que a instrução abaixo faça a soma de AL com BL (veremso as instruções aritméticas no próximo capítulo)

ADD AL,BL ; AL \leftarrow AL + BL e ambos AL e BL contém 7Fh

		representação não-sinalizada	representação sinalizada
7Fh	0111 1111b	127	+127
7Fh	+ 0111 1111b	+ 127	+127
	<u>0 1111 1110b</u>	254 (OK)	254 (fora)

Logo após a execução da instrução:

CF = 0 e OF = 1 , pois no MSB o "vem-um" é diferente do "vai-um".

13.2.3 Resumo dos registradores do processador 8086

Na tabela 13.2 encontramos um resumo dos registradores do processador 8086.

REGISTRADORES DE DADOS			
AH	AL	→	AX
BH	BL	→	BX
CH	CL	→	CX
DH	DL	→	DX
Registradores de segmentos			
CS			
DS			
SS			
ES			
Registradores índices e apontadores			
SI			
DI			
SP			
BP			
IP			
Registrador de sinalizadores			
FLAGS			

Tabela 13.2 - Resumo dos Registradores do processador 8086

Barramentos do processador 8086

Como vimos no capítulo 11 deste livro, os barramentos são vias que permitem a movimentação de informações entre diversas unidades. O processador 8086 tem um único barramento que trabalha como barramento de endereços e de dados, ou seja, tem um barramento multiplexado. Sua

largura é de 20 bits, por isso acessa até 1M Byte de memória ($2^{20} = 1.048.576 = 1M$). Quando utilizado como barramento de dados, só utilizam 16 bits dos 20 bits disponíveis. O barramento de controle é independente e possui 16 bits.

13.3 Organização de Memória

Como acabamos de ver, o processador 8086 possui um barramento que quando utilizado para endereços, usa 20 bits para acessar posições de memória física, ou seja, pode acessar memória de até 1MB, do endereço 00000h até FFFFFh.

Apesar de ter um barramento de 20 bits, o 8086 trabalha com palavras de endereço de 16 bits. Então como gerar endereços com 20 bits se trabalha com palavras de 16 bits? A solução é simples, basta utilizar a idéia de segmentação de memória. Um segmento de memória, aqui, é um bloco de 64 KB ($2^{16} = 2^6 \times 2^{10} = 65.536 = 64K$) de posições de memória consecutivas e identificadas por um endereço de segmento (endereço base) e um deslocamento (*offset*) em relação ao início do segmento. Em outras palavras, pode utilizar 64K segmentos de memória ("memória lógica") mapeados em uma memória de 1MB ("memória física"). O par *segment:offset* é o endereços lógico que nos dá a posição que queremos acessar dentro do segmento apontado por *segment*, com um deslocamento dado por *offset* (figura 13.2).

Por exemplo, dado o endereço lógico: 8350h:0420h, temos o endereço situado no segmento de 64KB que se inicia em 8350h com deslocamento de 0420h.

O endereço lógico, para ser acessado na memória física tem que ser transformado em endereço físico. Como todo início de segmento, os 4 bits menos significativos do endereço físico são iguais a zero, para transformar um endereço lógico em endereço físico, basta acrescentar 0h (0000b) no final do *segment* (deslocamento de 4 bits para a esquerda) e somar com o *offset*. Por exemplo, o endereço lógico 8350h:0420h corresponde à localização na memória física 83920h, pois:

83500h -> desloca-se 1 casa hexadecimal (4 casas binárias)
+ 0420h -> soma-se o deslocamento
83920h -> endereço físico resultante (20 bits)

O identificador de segmento (*segment*) é o endereço base, neste exemplo 8350h, e aponta para uma região da memória enquanto que o *offset* aponta para um local dentro deste segmento, neste exemplo 0420h.

Utilizando a segmentação, temos que endereços lógicos diferentes podem representar o mesmo endereço físico. Por exemplo, os endereços lógicos 028Ch:0003h e 0287h:0053h pois ambos, se calcularmos o endereço real, resulta em 028C3h.

Na programação do 8086, veremos que sempre o *segment* tem que estar armazenado do registrador de segmento (CS, DS, SS ou ES) e o *offset* nos registradores de propósito geral, ou específicos (IP, SP ou BP) ou indicado pelos rótulos. Se por exemplo quisermos acessar uma posição no segmento de código, o CS tem que ser iniciado com o endereço base do segmento e o IP com o respectivo offset, ou seja, se CS = 0F0Fh e IP = 000Fh, o endereço real será 0F0FFh.

Observação:

A segmentação é um esquema muito útil para gerar códigos relocáveis, ou seja, códigos executáveis que podem ser alocados em qualquer posição de memória.

13.4 Interrupções

Uma interrupção é um evento ocasional que deve ser prontamente atendida, durante a execução de um processamento pelo computador. O

atendimento desta interrupção causa a suspensão do processamento em curso.

No 8086 temos os seguintes tipos de interrupções:

- Causadas pela ocorrência de eventos "catastróficos": falta de energia, erro de memória, erro de paridade em comunicações, etc. Este tipo de interrupção não pode ser inibido.
- Causadas pela ação de dispositivos externos (periféricos): podem ser habilitadas ou inibidas.
- Causadas pelo próprio programa em curso: erro de divisão, erro de transbordamento (*overflow*). Este tipo de interrupção é chamado de exceção.

13.5 Portas de Entrada e Saída

O processador 8086 usa até 64KB para endereçar as portas de E/S. A tabela 13.3 nos mostra as portas de E/S mais utilizadas, que podem ser endereçadas por meio de instruções de entrada e de saída ou por chamadas do sistema operacional.

Endereço da Porta	Descrição
20h-21h	Controlador de interrupção
60h-63h	Controlador de teclado
2F8h-2FFh	Porta serial - COM 2
320h-32Fh	Disco rígido
378h-37Fh	Porta Paralela
3F8h-3FFh	Porta serial - COM 2

Tabela 13.3 - Portas de E/S

CAPÍTULO 14 INTRODUÇÃO À PROGRAMAÇÃO EM LIGUAGEM DE MONTAGEM do INTEL 8086.

Objetivos do Capítulo

Ao final da leitura desse capítulo o leitor estará apto a:

- utilizar o programa DEBUG para execução de pequenos trechos de código assembly*
- entender a estrutura dos programas em linguagem assembly*
- conhecer e utilizar o formato dos dados, variáveis e constantes na linguagem assembly*
- conhecer e utilizar os comandos de entrada e saída e sinalizadores (flags)*
- criar e executar programas em linguagem assembly*

FUNDAMENTOS

14.1 Programando com o programa DEBUG

Antes de iniciarmos, é importante explicar o objetivo de começarmos os conceitos básicos da linguagem de montagem com um programa de depuração de código (DEBUG). Trata-se de uma estratégia de ensino, prática e visual, de como as informações e os comandos são armazenadas na memória do computador, a ordem como são agrupadas e executadas/lidas. Após entender e visualizar a apresentação das informações e comandos utilizando o programa DEBUG, os estudantes não apresentam maiores dificuldades na abstração de comandos e operações mais complexas. Isso

vem facilitando muito o entendimento em nossas aulas práticas nas disciplinas iniciais de programação *assembly*. Vamos ao DEBUG!

Para executar o programa, em computadores com o Sistema Operacional Windows instalado, basta executar o "Prompt de Comando" que fica geralmente no Botão Iniciar - Acessórios. A figura 14.1 ilustra um possível local de encontrar o "prompt de comando" no Sistema Operacional Windows 7.

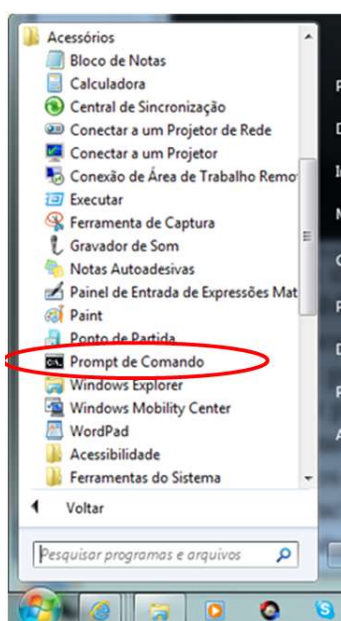


Figura 14.1 - Localização do "Prompt de Comando" no sistema Windows 7

Uma vez executado o "Prompt de Comando", digite na linha de comando o seguinte comando, finalizando com a tecla "enter" (<enter>).

```
C:\> DEBUG <Enter>
```

A Figura 14.2 ilustra o resultado desse comando.

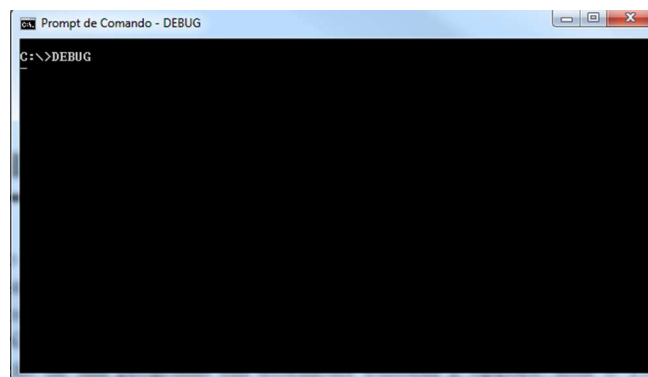


Figura 14.2 – Resultado da Execução do Comando DEBUG no “Prompt de Comando”

Se você foi atento, percebeu que abaixo da linha de comando `C:\>DEBUG` foi mostrado um “hífen”. Isso significa que você executou com êxito o programa DEBUG. Para sair do DEBUG, basta informar na linha de comando (após o hífen) a letra “Q” (quit) e ele retorna a linha de comando do “Prompt de Comando”.

Dentro do programa DEBUG, você pode verificar todos os comandos que podem ser executados. Basta digitar “?” seguido de <enter>.

Após digitar ? será mostrada Lista de Comandos do Programa DEBUG como ilustrado na Figura 14.3 a seguir.

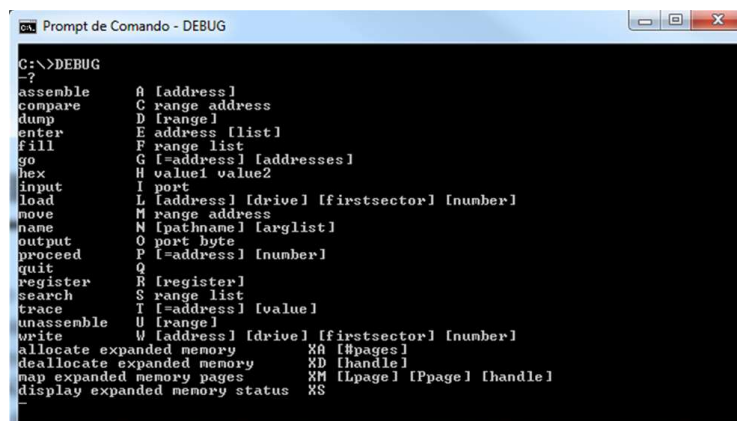


Figura 14.3 - Lista de Comandos do Programa DEBUG

Para iniciar, vamos começar com algo bem simples, e aproveitar para fazer uma revisão de aritmética binária!

O Comando H, faz cálculos de Adição e Subtração em Hexadecimal (hex).

Para Adição e Subtração: Comando H

Supondo dois número em hex:0002 e 0001, escreva no programa DEBUG o seguinte comando, terminando com a tecla <enter>:

```
H 0002 0001 <Enter>
```

Após a execução, o resultado será:

```
-H 0002 0001 <Enter>
```

```
0003 0001
```

Sendo o primeiro o valor o resultado da Adição dos dois números fornecidos como parâmetro do comando H, e o segundo o resultado da Subtração desses mesmos dois números.

Para os valores: 0009 e 0001

Resultado:

000A 0008

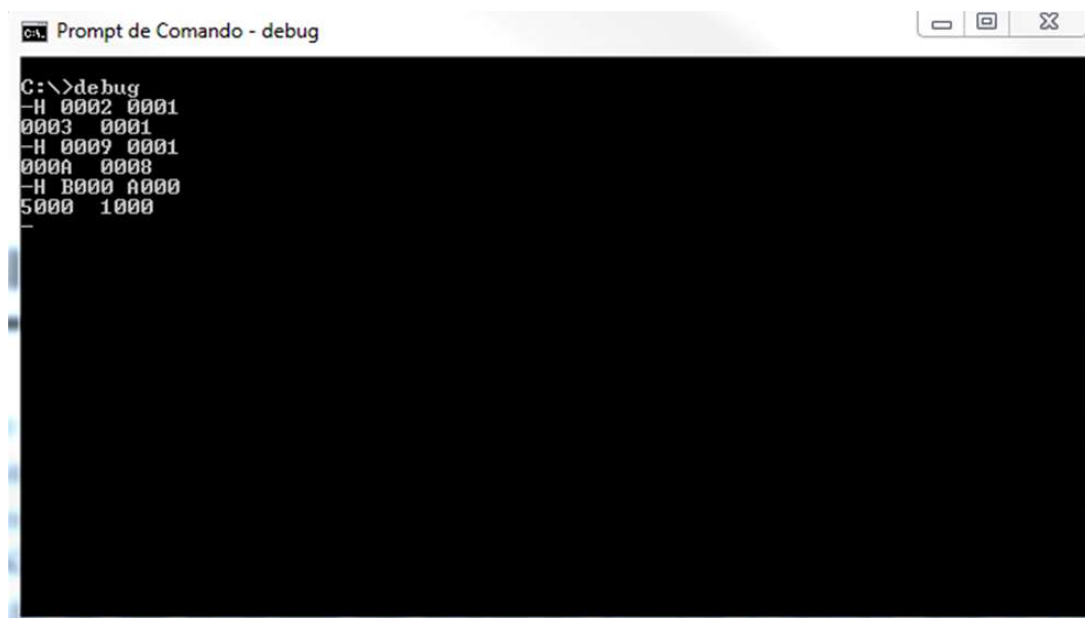
Note que o resultado é apresentado em Hexadecimal. Por isso a resposta com que estamos acostumados para a soma de 9+1 é 10. Em hexadecimal A é igual a 10 em decimal.

Outros valores: **B000** e **A000**

Resultado:

5000 1000

Note que aqui também tem alguma coisa errada. Agora pensando em aritmética hexadecimal, o primeiro valor, resultado da adição dos dois valores, deveria ser 15000. O DEBUG só apresenta os quatro valores à esquerda. A figura 14.4 apresenta os comandos e resultados descritos.



```
C:\>debug
-H 0002 0001
0003 0001
-H 0009 0001
000A 0008
-H B000 A000
5000 1000
_
```

Figura 14.4 - Comandos de adição e subtração em hexadecimal

Não sei se você já tentou, mas no nosso primeiro comando utilizando o comando H, fizemos a adição e subtração dos números 0002 e 0001 (nessa ordem). E se invertessemos os números. O que poderia acontecer? Vamos tentar?

```
-H 0001 0002 <Enter>
```

Resultado:

```
0003 FFFF
```

O segundo valor não deveria ser: -0001 ???

Lembre-se que estamos trabalhando com números hexadecimais... e FFFF corresponde a -0001 em complemento de 2.

Tente mais esse comando:

```
-H 0001 FFFF
```

Resultado:

```
0000 0002
```

Existe uma explicação bem fácil para isso... na realidade os dois valores (0001 e FFFF) são somados. O resultado é $0001 + FFFF = 10002$, ocorrendo assim um overflow, sendo o número 1, desconsiderado.

Vamos partir para cálculos e a utilização de códigos de máquina da linguagem Assembly. O primeiro passo é aprendermos a verificar os registradores e os valores que neles estão armazenados.

Você pode verificar os registradores com o comando R.

Na linha de comando do programa DEBUG digite

```
-R
```

O resultado será algo semelhante às seguintes informações:

```
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B19  ES=0B19  SS=0B19  CS=0B19  IP=0100  NV UP EI PL NZ NA PO NC
0B19:0100 7438          JZ          013A
```

Podemos ver que os registradores AX, BX, CX, DX estão zerados.

Para visualizarmos o conteúdo de um registrador específico, podemos utilizar o mesmo comando R, seguido do nome do registrador. Por exemplo, se quiséssemos saber o conteúdo do registrador AX, utilizaríamos o comando RAX. Assim:

```
-RAX <enter>
```

O resultado seria:

```
AX 0000
```

```
:
```

Note que após apresentar o conteúdo do registrador AX, foi exibido na linha seguinte "dois pontos" (":").

Se quisermos inserir um valor diferente, basta digitar em seguida dos dois pontos o valor desejado e a tecla <enter>. Se não quisermos alterar, basta teclar <enter>. Vamos alterar o valor para 0001. Digite após os dois pontos 0001 e tecle <enter>.

Agora se quisermos verificar se o novo valor foi inserido no registrador AX, podemos digitar o comando R. Assim, teríamos o seguinte resultado:

```
AX=0001 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B19 ES=0B19 SS=0B19 CS=0B19 IP=0100 NV UP EI PL NZ NA PO NC
0B19:0100 7438 JZ 013A
```

Note que agora o registrador AX apresenta o valor 0001, como queríamos.

Poderíamos montar instruções e sequências de comando utilizando esse método, com o auxílio dos códigos de operação (*opcode* ou *operate code*). Entretanto, fazer um programa assembly seria muito trabalhoso e complexo.

Para agilizar um pouco mais o processo, podemos digitar os comandos diretamente na memória, em posições sequenciais, informando apenas o endereço de início do código.

Para tanto, temos que conhecer mais um comando do programa DEBUG. O comando A (*address* ou endereço). Digitamos **A** seguido do valor em hexadecimal, geralmente 0100h, locais onde os programas .COM devem iniciar. No caso de omissão, o endereço inicial é o especificado pelos registradores CS:IP.

Vamos digitar um pequeno programa que colocará o valor 0002 no registrador AX e o valor 0003 no registrador BX, em seguida, os valores serão somados e o resultado colocado em AX.

O programa em linguagem assembly é a seguinte:

```
MOV AX,0002
```

```
MOV BX,0003
ADD AX,BX
```

A sequência de comandos necessários para registrarmos esse pequeno programa na memória é a seguinte:

```
A 100 <enter>
MOV AX,0002 <enter>
MOV BX,0003 <enter>
ADD AX,BX <enter>
NOP <enter> <enter>
```

A figura 14.5 a seguir, ilustra a entrada de dados desse pequeno programa.

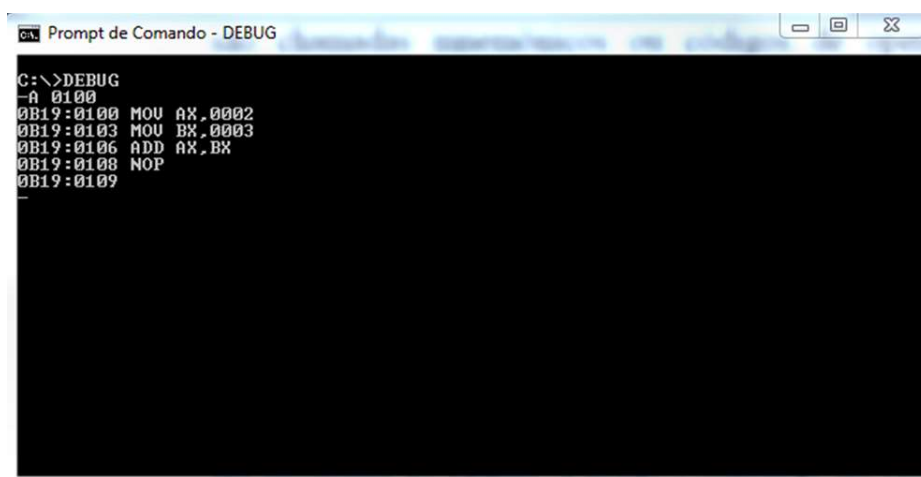


Figura 14.5 - Digitação de um pequeno programa utilizando o DEBUG

O último comando: NOP (*No Operation* ou nenhuma operação) é um comando que indica o final de uma sequência de comandos, ou seja, o final de um programa.

Para executar esse pequeno programa utilizaremos outro comando do DEBUG: o comando **t** (*trace*). Ele executa, comando a comando, mostrando o estado dos registradores ao final de cada execução.

Vamos lá!

-t (executa o comando **MOV AX,0002**)

```
AX=0002 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B19 ES=0B19 SS=0B19 CS=0B19 IP=0103 NV UP EI PL NZ NA PO NC
0B19:0103 BB0300 MOV BX,0003
```

-t (executa o comando **MOV BX,0003**)

```
AX=0002 BX=0003 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B19 ES=0B19 SS=0B19 CS=0B19 IP=0106 NV UP EI PL NZ NA PO NC
0B19:0106 01D8 ADD AX,BX
```

-t (executa o comando **ADD AX,BX**)

```
AX=0005 BX=0003 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B19 ES=0B19 SS=0B19 CS=0B19 IP=0108 NV UP EI PL NZ NA PE NC
0B19:0108 90 NOP
```

-t (executa o comando **NOP**)

```
AX=0005 BX=0003 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B19 ES=0B19 SS=0B19 CS=0B19 IP=0109 NV UP EI PL NZ NA PE NC
0B19:0109 96 XCHG SI,AX
```

-

Com esta sequência, o programa foi executado. O resultado da adição encontra-se no registrador AX.

Fazer programas em linguagem de montagem já é bem complicado. Utilizar o DEBUG para isso fica mais difícil ainda. Mesmo utilizando o modo de introduzir comandos diretamente na memória.

Como dissemos no início desse capítulo, a utilização do programa DEBUG é uma estratégia didática para que o leitor possa perceber de forma prática e visual como são armazenadas as informações e comandos na memória e assim possa fazer processos de abstração mais complexos de forma mais fácil.

14.2 Programando com o montador TASM

A maneira com que faremos a programação em Assembly será a seguinte, ilustrada na Figura 14.6:

[inserir figura 14.6 (Fig14-06.tif)]

Figura 14.6 - Processo de Construção de programas Assembly

Editaremos a sequência de instruções em um editor de texto (sem códigos de controle, como o bloco de notas). Salvaremos o arquivo com um determinado nome, com extensão **.ASM**.

Depois faremos a compilação, utilizando o TASM (Turbo Assembler - Borland). Se algum erro for evidenciado, editaremos novamente o arquivo e faremos as devidas correções.

Ao chegarmos a um processo de compilação sem erros, o TASM gerará um arquivo objeto (com extensão **.OBJ**). Depois disso, faremos a Ligação com o TLINK (Turbo Linker - Borland) e se tudo der certo, teremos o arquivo executável (**.EXE** ou **.COM** - conforme configuração).

Para exemplificar tal processo, vamos escrever um programa simples, que escreve uma mensagem na tela "Linguagem de montagem é fácil!".

Abra um editor de texto, por exemplo, o BLOCO DE NOTAS, e digite o seguinte programa:

```
TITLE PGM1: MENSAGEM  
.MODEL SMALL  
.STACK 100H  
.DATA  
mensagem DB 'Linguagem de montagem eh facil!$'  
.CODE  
MAIN PROC  
    MOV AX,@DATA  
    MOV DS, AX  
    LEA DX,mensagem  
    MOV AH,9  
    INT 21H  
    MOV AH,4CH  
    INT 21H  
MAIN ENDP  
END MAIN
```

Grave o arquivo como **MENSAGEM.ASM**

Na linha de comando do DOS (no mesmo diretório onde você salvou o arquivo) digite:

```
TASM MENSAGEM.ASM <enter>
```

Ele apresentará a seguinte mensagem:

```
C:\Tasm>edit mensagem.asm

C:\Tasm>TASM MENSAGEM.ASM
Turbo Assembler  Version 4.1  Copyright (c) 1988, 1996 Borland
International

Assembling file:   MENSAGEM.ASM
**Error** MENSAGEM.ASM(5) Extra characters on line
Error messages:    1
Warning messages:  None
Passes:            1
Remaining memory:  452k

C:\Tasm>
```

Observe que no exemplo acima ocorreu um erro no processo de compilação, indicando que o erro está na linha 5.

Basta editar o arquivo .ASM, corrigir o erro (colocando um ` no fim da linha) e compilar novamente o código.

Se tudo der certo, você terá obtido as seguintes mensagens:

```
C:\Tasm>TASM MENSAGEM.ASM~
Turbo Assembler  Version 4.1  Copyright (c) 1988, 1996 Borland
International

Assembling file:   MENSAGEM.ASM
Error messages:    None
Warning messages:  None
```

```
Passes:          1
Remaining memory: 452k

C:\Tasm>
```

Agora basta fazer a ligação com o TLINK.

```
TLINK MENSAGEM.OBJ <enter>
```

Se tudo der certo, você terá a seguinte mensagem:

```
C:\Tasm>TLINK MENSAGEM.OBJ
Turbo Link  Version 7.1.30.1. Copyright (c) 1987, 1996 Borland
International

C:\TASM>
```

Para executar o programa basta digitar o nome na linha de *prompt*.

```
C:\TASM>MENSAGEM
Linguagem de montagem eh facil!

C:\TASM>
```

Estes são os passos para gerar, montar e executar um programa em linguagem de montagem, utilizando as ferramentas da Borland.

Não se importando ainda com a estrutura e os comandos da linguagem Assembly, vamos digitar o programa abaixo que carregará dois registradores, AX e BX, com os valores 000B e 00A1 e fará a soma desses dois valores, e armazenará o resultado em AX. Logo em seguida, carregará o valor 0005 em BX e fará a subtração com o resultado obtido na operação anterior, armazenando o resultado em AX. E por fim, o programa encerrará sua execução, devolvendo o controle ao Sistema Operacional.

Programa 14.1

```
TITLE PRM141:ADICAO E SUBTRACAO
;Programa para ilustração do programa DEBUG
.MODEL SMALL
.STACK 100H
.CODE
MAIN PROC
    MOV AX,000BH    ; carrega o primeiro valor em AX
    MOV BX,00A1H    ; carrega o segundo valor em BX
    ADD AX,BX       ; soma os dois valores e armazena
                    ; o resultado em AX
    MOV BX,0005H    ; carrega o terceiro valor em BX
    SUB AX,BX       ; subtrai o terceiro valor do resultado
                    ; da operação anterior
    MOV AH,4CH      ; código para devolver o controle p/ DOS
    INT 21H         ; interrupção que executa a função em AH
MAIN ENDP
    END MAIN
```

Vamos gerar o código executável, fazendo as mesmas etapas vistas no exemplo anterior.

Montagem do programa para geração do .OBJ

```
C:\TASM>TASM PRM141.ASM <enter>
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland
International

Assembling file:      PRM141.ASM
Error messages:      None
Warning messages:    None
Passes:              1
Remaining memory:    413k
```

Ligação para geração do .EXE

```
C:\TASM>TLINK PRM141.OBJ <enter>
Turbo Link Version 7.1.30.1. Copyright (c) 1987, 1996 Borland
International
```

Execução do programa

```
C:\TASM>PRM141 <enter>

C:\TASM>_
```

Como não mandamos escrever o resultado, não sabemos se as operações foram executadas corretamente. Uma das maneiras de verificar se a execução foi correta é através da utilização do programa DEBUG.

Par usar o DEBUG, vamos digitar na linha de comando logo após a execução o seguinte:

```
C:\TASM>DEBUG PRM141.EXE
-
```

Utilizando os comandos do programa DEBUG que já vimos anteriormente, vamos verificar a situação dos registradores:

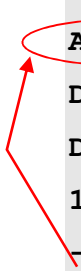
Tecla R <enter>

```
-R
AX=0000  BX=0000  CX=0011  DX=0000  SP=0100  BP=0000  SI=0000
DI=0000
DS=152B  ES=152B  SS=153D  CS=153B  IP=0000  NV UP EI PL NZ NA PO NC
153B:0000 B80B00          MOV     AX,000B
-
```

Como podemos notar, os registradores de dados estão “zerados” e a próxima instrução a ser executada é a instrução MOV AX,000B, que é a primeira instrução do programa PRM141.

Ao executar a instrução corrente com o comando **T** (*trace*), temos:

```
-T
AX=000B  BX=0000  CX=0011  DX=0000  SP=0100  BP=0000  SI=0000
DI=0000
DS=152B  ES=152B  SS=153D  CS=153B  IP=0003  NV UP EI PL NZ NA PO NC
153B:0003 BBA100          MOV     BX,00A1
-
```



Note que após a execução da instrução corrente, o registrador AX agora armazena o valor 000B como queríamos. Veja que a instrução corrente agora é MOV BX,00A1, que é a segunda instrução do programa PRM141.

E assim, executando instrução a instrução, ou executando um determinado trecho de programa (até posição final, com o comando G) podemos verificar o programa está sendo executado adequadamente.

A título de ilustração, abaixo segue a execução do programa utilizando os comandos **T** e **G**.

```
-T
AX=000B  BX=00A1  CX=0011  DX=0000  SP=0100  BP=0000  SI=0000
DI=0000
DS=152B  ES=152B  SS=153D  CS=153B  IP=0006  NV UP EI PL NZ NA PO NC
153B:0006 03C3          ADD     AX,BX
-T
AX=00AC  BX=00A1  CX=0011  DX=0000  SP=0100  BP=0000  SI=0000
DI=0000
DS=152B  ES=152B  SS=153D  CS=153B  IP=0008  NV UP EI PL NZ NA PE NC
153B:0008 BB0500      MOV     BX,0005
-T
AX=00AC  BX=0005  CX=0011  DX=0000  SP=0100  BP=0000  SI=0000
DI=0000
DS=152B  ES=152B  SS=153D  CS=153B  IP=000B  NV UP EI PL NZ NA PE NC
153B:000B 2BC3      SUB     AX,BX
-T
AX=00A7  BX=0005  CX=0011  DX=0000  SP=0100  BP=0000  SI=0000
DI=0000
DS=152B  ES=152B  SS=153D  CS=153B  IP=000D  NV UP EI PL NZ NA PO NC
153B:000D B44C      MOV     AH,4C
-G

Program terminated normally
-
```

Ainda assim podendo verificar o que está ocorrendo no processador a cada instrução com o DEBUG, essa forma de trabalho não é muito utilizada (pela visibilidade limitada).

Existe outro programa, chamado TURBO DEBUGGER da Borland, que permite a mesma verificação, em um ambiente mais confortável.

Entretanto, para fazê-lo, no processo de Montagem-Ligação, temos que informar ao montador e ao ligador (*linker*) que carregue, juntamente com o código, informações para que o programa TURBO DEBUGGER possa controlar a execução do programa.

Para isto, no processo de compilação, vamos utilizar a diretiva /zi, e no processo de ligação, a diretiva /v

Dessa forma, a montagem e ligação do programa PRM141.ASM ficaria da seguinte forma:

```
C:\TASM>TASM /zi PRM141.ASM <enter>
```

```
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland  
International
```

```
Assembling file:    PRM141.ASM
```

```
Error messages:    None
```

```
Warning messages:  None
```

```
Passes:            1
```

```
Remaining memory:  411k
```

```
C:\TASM>TLINK /v PRM141.OBJ <enter>
```

```
Turbo Link Version 7.1.30.1. Copyright (c) 1987, 1996 Borland  
International
```

```
C:\TASM>
```

O arquivo PRM141.EXE possui, então, as informações que possibilitarão ao TD (Turbo Debugger) gerenciar a execução do código.

Para executar o programa com o Turbo Debugger, digite na linha de comando do DOS:

TD <nome do programa.exe> <enter>

No nosso caso:

TD PRM141.EXE <enter>

Aparecerá na tela:

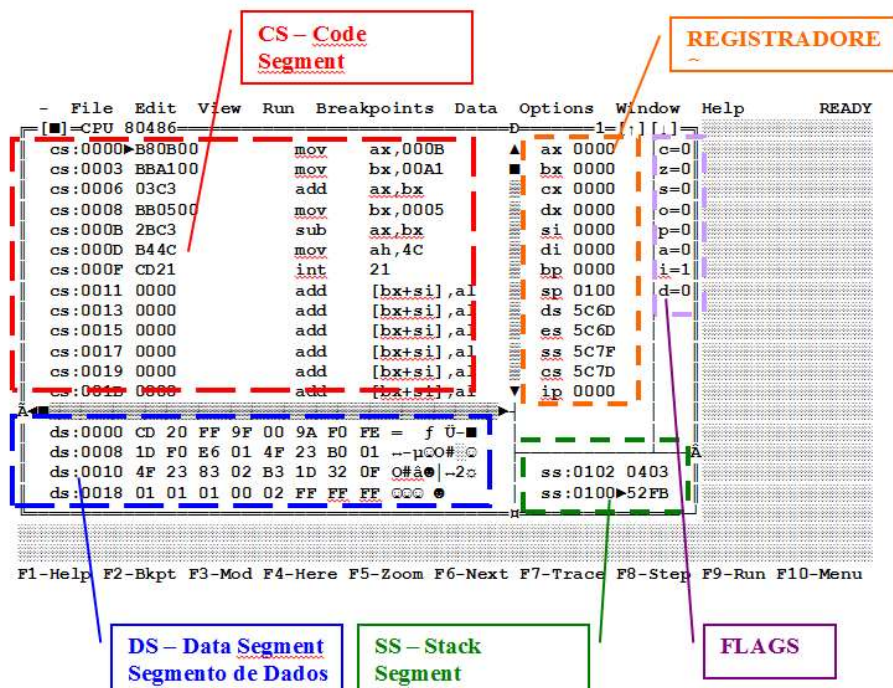
```
- File Edit View Run Breakpoints Data Options Window Help    READY
[■]=CPU 80486 1=[:][]
cs:0000 B80B00      mov     ax,000B      ▲ ax 0000    c=0
cs:0003 BBA100      mov     bx,00A1      ■ bx 0000    z=0
cs:0006 03C3        add     ax,bx         cx 0000    s=0
cs:0008 BB0500      mov     bx,0005      dx 0000    o=0
cs:000B 2BC3        sub     ax,bx         si 0000    p=0
cs:000D B44C        mov     ah,4C         di 0000    a=0
cs:000F CD21        int     21           bp 0000    i=1
cs:0011 0000        add     [bx+si],al    sp 0100    d=0
cs:0013 0000        add     [bx+si],al    ds 5C6D
cs:0015 0000        add     [bx+si],al    es 5C6D
cs:0017 0000        add     [bx+si],al    ss 5C7F
cs:0019 0000        add     [bx+si],al    cs 5C7D
cs:001B 0000        add     [bx+si],al    ip 0000
ds:0000 CD 20 FF 9F 00 9A F0 FE = f Ü-■
ds:0008 1D F0 E6 01 4F 23 B0 01 --pO#
ds:0010 4F 23 83 02 B3 1D 32 0F C#â@|~2
ds:0018 01 01 01 00 02 FF FF FF 0000
ss:0102 0403
ss:0100 52FB

F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Temos na primeira linha uma série de opções de comandos.

Na última linha as teclas de atalho para os comandos mais frequentes, tais como: F8 - Step (executa o próximo comando - um passo do programa), F9 - Run (executa o programa até encontrar um final).

Além disso, podemos ver que existem áreas de exibição:



Observe que no segmento do código, o ponteiro (que no segmento de registradores é identificado por IP - Instruction Pointer) aponta para a instrução MOV AX,000B, ou seja, a primeira instrução, que está pronta para ser executada.

Ao executar, pressionando F8, podemos observar:

```

- File Edit View Run Breakpoints Data Options Window Help
[■]=CPU 80486 1=[:][]
cs:0000 B80B00 mov ax,000B ax 000B c=0
cs:0003 BBA100 mov bx,00A1 bx 0000 z=0
cs:0006 03C3 add ax,bx cx 0000 s=0
cs:0008 BB0500 mov bx,0005 dx 0000 o=0
cs:000B 2BC3 sub ax,bx si 0000 p=0
cs:000D B44C mov ah,4C di 0000 a=0
cs:000F CD21 int 21 bp 0000 i=1
cs:0011 0000 add [bx+si],al sp 0100 d=0
cs:0013 0000 add [bx+si],al ds 5C6D
cs:0015 0000 add [bx+si],al es 5C6D
cs:0017 0000 add [bx+si],al ss 5C7F
cs:0019 0000 add [bx+si],al cs 5C7D
cs:001B 0000 add [bx+si],al ip 0003
ds:0000 CD 20 FF 9F 00 9A F0 FE = f Ü-
ds:0008 1D F0 E6 01 4F 23 B0 01 -µ00#
ds:0010 4F 23 83 02 B3 1D 32 0F 0#â|~2
ds:0018 01 01 01 00 02 FF FF FF 0000
ss:0102 0403
ss:0100 52FB
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```

Na opção RUN, no menu, temos também alguns comandos úteis:

- **Run - Run** (Executa todo o código)
- **Run - Go to cursor** (Reset o código (Run - Program Reset).
Selecione a 4 linha de instrução com o ponteiro do mouse. Execute novamente o código, com a opção Run - Go to cursor. Observe que agora ele parou onde você tinha selecionado.
- **Run - Animated** - Reset o código. Execute o programa com Run - Animated. Coloque um tempo igual a 20 e tecle Ok.
- **Breakpoints - Toggle** (estabeleça um ponto de parada no endereço 000B. Depois Reset o programa (Run - Program Reset). E logo em seguida execute-o com Run - Run. Veja que agora o programa é executado até o ponto de parada estabelecido.

14.3 ALGUNS COMANDOS BÁSICOS E A SINTAXE DA LINGUAGEM MONTAGEM

Você já deve ter notado que para podermos escrever um programa que o montador entenda, devemos seguir algumas regras. A seguir mostraremos estas regras:

a) Linhas de Comando (statements):

Uma linha de comando da linguagem de montagem, deve ser formada como:

nome	operação	operando(s)	;comentário
------	----------	-------------	-------------

Exemplo:

START:	MOV	CX,5	;inicializa o contador
--------	-----	------	------------------------

Esses campos devem ser separados por espaço(s) ou tab. O símbolo START é um rótulo.

b) Campo Nome

O campo nome pode ser um rótulo de instrução, um nome de sub-rotina, um nome de variável, podendo conter de **1 a 31 caracteres**, deve iniciar com uma letra e conter somente letras, números e os caracteres `? . @ _ : $ %`.

Exemplos:

<i>nomes válidos</i>	<i>nomes inválidos</i>
LOOP1:	DOIS BITS
.TEST	2abc
@character	A42.25
SOMA_TOTAL4	#33
\$100	

Observação:

O Montador traduz os nomes para endereços de memória.

c) Dados do Programa

Os dados dos programas podem ser passados por variáveis ou constantes.

As variáveis devem ser declaradas como mostramos a seguir:

- **Variáveis do tipo Byte**

```
Nome DB valor_inicial
```

Exemplo:

```
ALPHA DB 4  
TESTE DB ?
```

- **Variáveis do tipo Word**

```
Nome DW valor_inicial
```

Exemplo:

```
WRD DW -2  
TESTE DW ?
```

As constantes são declaradas como:

EQU (Equates)

```
Nome EQU constante
```

Exemplo:

LF	EQU	0AH
-----------	------------	------------

Isto associa a LF (constante) o valor em hexa 0A, que corresponde ao código ASCII do character *Line Feed*.

Observação:

Os números, para mostrar em que base de numeração eles estão, devem ter uma letra para esta distinção. Se não colocarmos nenhuma, o montador assume que estamos trabalhando no sistema decimal. São os seguintes os identificadores de base:

- **B ou b - binário.**

Exemplo: 1110101b ou 1110101B

- **D ou d decimal.**

Exemplo: 64223 ou 64223d ou 64223D, 1110101 é considerado decimal (ausência do B), -2184D (número negativo)

- **H ou h - hexadecimal.**

Exemplo: 64223h ou 64223H, 0FFFFh (começa com um decimal e termina com h), 1B4Dh

Estrutura de um programa Linguagem de Montagem

Segue abaixo a estrutura geral de um programa em linguagem de montagem.

```

TITLE NOME:DESCRIÇÃO

.MODEL SMALL

.STACK tamanho em hexa

.DATA

;os dados vão aqui

.CODE

MAIN PROC

    ;as instruções vão aqui

MAIN ENDP

;outros procedimentos vão aqui

END MAIN

```

Exemplos de Programa:

1. Nosso primeiro programa irá mostrar o caractere "*" na tela.

```

TITLE EX01:MOSTRA UM ASTERISCO

;Primeiro programa exemplo. Mostra um asterisco na tela

.MODEL SMALL

.STACK 100H

.CODE

MAIN PROC

    ; Exibição do caractere

    MOV AH,2          ; função para exibição de caractere
    MOV DL,'*'        ; caractere a ser exibido
    INT 21H           ; interrupção que executa a função armaz.
                        ; em AH (exibir caractere na tela)

                        ; Retorna para o DOS

    MOV AH,4CH        ;função para retorno ao DOS
    INT 21H           ;interrupção que executa a função em AH

MAIN ENDP

```

END MAIN

Digite este programa em um editor de texto, salvando-o com um nome significativo e com a extensão **.ASM**. Por exemplo: EX01.ASM

Faça a montagem (TASM) e a ligação (TLINK). Em seguida, execute o arquivo e verifique se ele mostra o caractere "*" na tela.

Neste programa foi apresentado o comando INT 21H. Este comando pode ser utilizado para realizar uma série de funções do DOS. A função que será executada tem o seu código (número) armazenado em AH (indicando o que deve ser feito).

Dependendo da função a ser executada, outros registradores devem ser utilizados para guardar as informações de entrada e saída. No nosso exemplo, está sendo executada a função de escrita no dispositivo de saída padrão (código 2 em AH), que no nosso caso é o vídeo. Em DL estará o que se quer exibir.

Em Resumo:

02H - Função para exibição de um caractere.	
Entradas	AH = 2
	DL = código ASCII do caractere a ser exibido
Saídas	AL = código ASCII do caractere exibido

2. Vamos montar um segundo programa, que agora irá solicitar a digitação de um caractere. Depois de digitada uma determinada tecla, o caractere correspondente será exibido na tela. Em seguida, vamos

exibir novamente este mesmo caractere, separado por um hífen do primeiro.

Programa para Entrada de Caractere:

```
TITLE EX02:RECEBE E MOSTRA UM CARACTERE
;Segundo programa exemplo. Recebe um caractere e o exhibe
.MODEL SMALL
.STACK 100H
.CODE
MAIN PROC
; Entrada de um Caractere
MOV AH,1      ; função para entrada de caractere
INT 21H       ; executa a função em AH
;
; Depois de pressionada uma tecla, seu código ascii é armazenado
; em AL. Se for digitada uma tecla não-caractere, AL recebe 0
; Salva o caractere digitado em BL
;
MOV BL,AL     ; salva o caractere digitado em BL
;
; Exibe o Hifen
MOV AH,2      ; função para exibição de caractere
MOV DL,'-'    ; caractere a ser exibido
INT 21H       ; interrupção que executa a função armazenado
               ; em AH (exibir caractere na tela)
; Recupera e Exibe o caractere digitado
MOV DL,BL
MOV AH,2      ; função para exibição de caractere
INT 21H       ; interrupção que executa a função armazenado
               ; em AH (exibir caractere na tela)
; Retorna para o DOS
MOV AH,4CH    ; função para retorno ao DOS
```

```

    INT 21H          ; interrupção que executa a função em AH
MAIN ENDP
END MAIN

```

Apresentamos neste programa mais uma função que pode ser utilizada: entrada de caractere.

01H - Função para Leitura (entrada) de uma tecla.	
Entradas	AH = 1
Saídas	AL = código ASCII da tecla pressionada = 0 se for tecla de um não-caractere

Usamos nos dois programas anteriores uma terceira função, que é o retorno do controle ao Sistema Operacional.

4CH - Função para retorno ao S.O. (DOS)	
Entradas	AH = 4CH AL = código de retorno
Saídas	Nenhuma

3 - Vamos agora, montar um programa para exibir uma sequência de caracteres (conhecida como *string*).

Para fazê-lo, devemos armazenar esta *string* em um local de memória. Digamos em uma variável chamada MSG:

```

MSG      DB      'HELLO!$'

```

Note que o último caractere da mensagem é "\$". Este caractere indica o final da *string* para o a função de saída. Ele não é exibido.

Devemos utilizar uma função que faz a saída de uma *string*:

09H - Função para exibição de um <i>String</i>	
Entradas	AH = 9 DX = endereço de offset da <i>String</i>
Saídas	Nenhuma

Esta função necessita carregar o offset de MSG em DX. Uma das formas de se fazer isso é utilizando a instrução LEA (**L**oad **E**ffective **A**ddress).

LEA destino, fonte

No nosso exemplo, onde colocamos a nossa mensagem na variável MSG, teríamos o seguinte comando:

LEA DX, MSG

Quando um programa é carregado na memória, o DOS cria e faz uso de um segmento de memória de 256 bytes que contem informações sobre o programa (este segmento é conhecido como PSP - *program segment prefix*).

Com isso, o DOS coloca o número deste segmento nos registradores DS e ES antes de executar o programa.

O resultado disso é que, no início do programa, DS não contém o endereço do segmento de dados.

Dessa forma, devemos colocar em DS o endereço correto do segmento de dados corrente. Para tanto, usamos:

MOV AX, @DATA
MOV DS, AX

onde **@DATA** é o nome do segmento de dados definido em **.DATA**. O montador traduz **@DATA** para o endereço base do segmento. Outro detalhe é que não podemos mover este número diretamente para **DS**. Por isso temos que utilizar duas instruções.

Então, o programa para exibir uma mensagem fica assim:

```
TITLE MSG01:MOSTRA UMA MENSAGEM
;Terceiro programa exemplo. Mostra uma mensagem na tela
.MODEL SMALL
.STACK 100H
.DATA
MSG DB 'HELLO!$'
.CODE
MAIN PROC
; Inicializa DS
MOV AX,@DATA
MOV DS,AX      ; inicializa DS
;
;exibe mensagem
LEA DX,MSG      ; carrega o endereço da mensagem
MOV AH,9        ; função para exibição de string
INT 21H         ; executa a função em AH
;
; Retorna para o DOS
MOV AH,4CH      ; função para retorno ao DOS
INT 21H         ; interrupção que executa a função em AH
MAIN ENDP
END MAIN
```

CAPÍTULO 15 - INSTRUÇÕES DE MOVIMENTAÇÃO DE DADOS, LÓGICAS, DE DESLOCAMENTO, DE ROTAÇÃO E INSTRUÇÕES ARITMÉTICAS

Objetivos do Capítulo

Ao final desse capítulo o leitor estará apto a...

- Compreender o mecanismo das instruções de movimentação de dados e das instruções que utilizam tem a ULA.
- Conhecer e utilizar as principais instruções em linguagem de montagem para armazenamentos de dados em registradores e na memória, para as operações lógicas e aritméticas.

FUNDAMENTOS

15.1 Instruções de movimentação de dados

Como já sabemos, as instruções de movimentação de dados nos permitirão trazer as informações para dentro do processador, armazenando-as em um registrador para que possa ser processada. Além disto, poderemos também escrever uma informação na memória e copiar em outro registrador.

Os processadores da família x86, só não permitem que uma informação seja transferida de uma posição da memória para outra, executando apenas uma instrução de movimentação de dados. Será necessário, para isto, ler da memória e armazenar em um registrador e depois escrever o que foi armazenado, na memória novamente.

Vamos então apresentar o elenco das instruções de movimentação de dados, sua sintaxe e seu funcionamento.

O mnemônico para estas instruções é o símbolo **MOV** (move), e a sintaxe é:

MOV destino, fonte

A tabela 15.1 mostra as possibilidades de movimentação de dados. É permitida a transferência de dados entre dois registradores, entre um registrador e uma posição de memória e a movimentação de um número (constante) diretamente para um registrador ou para uma posição de memória.

OPERANDO FONTE	OPERANDO DESTINO		
	REGISTRADOR DE DADOS	REGISTRADOR DE SEGMENTO	MEMÓRIA
REGISTRADOR DE DADOS	SIM	SIM	SIM
REGISTRADOR DE SEGMENTO	SIM	NÃO	SIM
MEMÓRIA	SIM	SIM	NÃO
CONSTANTE	SIM	NÃO	SIM

Tabela 15.1 - Possibilidades de movimentação de dados - instrução MOV

Abaixo temos exemplos de instruções de movimentação de dados válidas:

```

MOV AX,WORD1    ; MOVIMENTA O CONTEÚDO DA POSIÇÃO DE
                  ; MEMÓRIA WORD1 PARA O REGISTRADOR AX

MOV AH,'A'       ; transfere o caracter ASCII 'A' para AH

MOV AH,41h       ; idem anterior: 41h corresponde ao caracter A

MOV AH,BL        ; move o conteúdo do byte baixo de BX
                  ; o byte alto de AX

MOV AX,CS        ; transfere cópia do conteúdo de CS para AX

```

Agora suponha a execução da instrução:

```
MOV AX,WORD1
```

A Tabela 15.2 nos mostra o conteúdo de AX antes e depois da execução da instrução.

	ANTES	DEPOIS
AX	0006H	8FFFh
WORD1	8FFFH	8FFFH

Tabela 15.2 - Execução de MOV AX,WORD1

Abaixo temos um exemplo de instruções de movimentação de dados inválidas e a solução para contornar o problema:

```
MOV WORD1,WORD2 ;instrução inválida  
  
; esta restrição é contornada como segue  
  
MOV AX,WORD2 ; primeiro o conteúdo de WORD2 vai para AX  
MOV WORD1,AX ; depois, o conteúdo de AX é movido para a  
; a posição de memória WORD1
```

Outra instrução de movimentação de dados permite a troca do conteúdo de duas localizações. O mnemônico desta instrução é **XCHG** (exchange) e sua sintaxe é:

```
XCHG destino,fonte
```

A tabela 15.3 nos mostra as possibilidades de movimentação de dados. É permitida a transferência de dados entre dois registradores de dados, entre um registrador e uma posição de memória.

	OPERANDO DESTINO
--	------------------

OPERANDO FONTE	REGISTRADOR DE DADOS	MEMÓRIA
REGISTRADOR DE DADOS	SIM	SIM
MEMÓRIA	SIM	NÃO

Tabela 15.3 – Possibilidades de movimentação de dados – instrução XCHG

Abaixo temos exemplos de instruções válidas, para a troca de informações.

```
XCHG AX, WORD1 ; troca o conteúdo da posição de memória
                  ; WORD1 com o do registrador AX
XCHG AH, BL      ; troca o conteúdo do byte baixo de BX com o
                  ; do byte alto de AX
```

Agora suponha a execução da instrução:

XCHG AX,BX

A Tabela 15.4 nos mostra o conteúdo de AX e BX antes e depois da execução da instrução.

	ANTES	DEPOIS
AX	0006h	8FFFh
BX	8FFFh	0006h

Tabela 15.4 – Execução de XCHG AX,BX

Como exercício, vamos traduzir um comando de atribuição de uma linguagem de alto nível:

```
b = a ; ( b recebe uma cópia do valor armazenado em a)
```

A tradução para a linguagem assembly seria:

```
MOV AX,a ;transfere o conteúdo da posição de memória a para AX  
MOV b,AX ;transfere AX para a posição de memória b
```

Outra instrução de movimentação de dados é a instrução **LEA** (*Load Effective Address*), que permite que se carregue em um registrador, o offset de um endereço lógico.

```
LEA destino,fonte
```

Por exemplo, considere o seguinte trecho de programa:

```
. DATA  
    MENSAGEM DB 'Adoro assembly!$'  
  
...  
  
.CODE  
    LEA DX,MENSAGEM ; DX carregado com o offset de MENSAGEM
```

Após a execução da instrução **LEA**, **DX** conterà o *offset* do endereço lógico formado por **DS:MENSAGEM**.

15.2 Instruções de adição e subtração

As instruções de adição e subtração nos permitirão a realização destas operações, tão importante nos programas que desenvolvemos.

A operação de adição usa o mnemônico **ADD** e sua sintaxe é:

```
ADD destino,fonte
```

A operação de subtração usa o mnemônico **SUB** e sua sintaxe é:

SUB destino, fonte

A tabela 15.5 nos mostra as possibilidades de adição e subtração de dados. É permitida a soma e subtração de dados entre dois registradores de dados, entre um registrador e uma posição de memória, de uma constante com um registrador e de uma constante com uma posição de memória.

OPERANDO FONTE	OPERANDO DESTINO	
	REGISTRADOR DE DADOS	MEMÓRIA
REGISTRADOR DE DADOS	SIM	SIM
MEMÓRIA	SIM	NÃO
CONSTANTE	SIM	SIM

Tabela 15.5 - Possibilidades de adição e subtração de dados - instruções ADD e SUB

Abaixo temos exemplos de instruções válidas, para a soma e subtração de operandos.

ADD AX,BX	; soma o conteúdo de AX com BX, resultado em AX
ADD AX,WORD1	; soma o conteúdo da posição de AX com o da a
	; memória WORD1, guardando em AX
ADD AL,5	; soma o conteúdo de AL com 5, resultado em AL
SUB AX,BX	; subtrai o conteúdo de o conteúdo de AX com BX
SUB AX,WORD1	; subtrai o conteúdo da posição de AX com o da a
	; memória WORD1, guardando em AX
SUB AL,5	; subtrai o conteúdo de AL com 5, resultado em AL

Agora suponha a execução da instrução:

ADD AX,BX

A Tabela 15.6 nos mostra o conteúdo de AX e BX antes e depois da execução da instrução.

	ANTES	DEPOIS
AX	0006h	8FF6h
BX	8FF0h	8FF0h

Tabela 15.4 - Execução de ADD AX,BX

Agora suponha a execução da instrução:

SUB AX,BX

A Tabela 15.7 nos mostra o conteúdo de AX e BX antes e depois da execução da instrução.

	ANTES	DEPOIS
AX	0006h	0001h
BX	0005h	0005h

Tabela 15.7 - Execução de ADD AX,BX

Abaixo temos um exemplo de instruções de adição de dados inválidas e a solução para contornar o problema (o mesmo pode ser aplicado à subtração):

```
ADD BYTE1,BYTE2      ; instrução inválida

; esta restrição é contornada como segue

MOV AL,BYTE2          ; primeiro o conteúdo de BYTE2 vai para AL
MOV BYTE1,AL          ; depois, o conteúdo de AL é subtraído de
                      ; BYTE1 e armazenado na posição de memória ;
                      ; BYTE1
```

Existem outras instruções aritméticas, de incrementar, decrementar e calcular o complemento de 2.

A operação de incremento (somar 1 ao operando) utiliza o mnemônico **INC** e sua sintaxe é:

INC destino

A operação de decremento usa o mnemônico **DEC** e sua sintaxe é

DEC destino

Estas instruções utilizam como fonte e destino, um mesmo operando, que pode ser um registrador ou uma posição de memória.

```
INC CX      ; incrementa o conteúdo de CX
INC WORD1   ; incrementa conteúdo posição memória WORD1
DEC BYTE2   ; decrementa conteúdo posição de memória BYTE2
DEC CL      ; decrementa o conteúdo de CL (byte baixo de CX)
```

Vamos ver como ficaria o conteúdo do operando após a execução da instrução:

INC BYTE1

A Tabela 15.8 nos mostra o conteúdo de BYTE1 antes e depois da execução da instrução.

Antes	Depois
BYTE1	BYTE1
0006h	0007h

Tabela 15.8 - Execução de INC BYTE1

Outra instrução aritmética muito bastante usada é a que calcula o complemento de 2 de um número. O mnemônico para esta instrução é **NEG**, e o operando pode ser um registrador ou uma posição de memória. A sintaxe desta instrução é:

NEG destino

As instruções abaixo são instruções válidas para o NEG:

```
NEG BX      ; gera o complemento de 2 do conteúdo de BX
NEG WORD1   ; idem, no conteúdo da posição de memória WORD1
```

Vamos ver como ficaria o conteúdo do operando após a execução da instrução:

NEG BX

A Tabela 15.9 nos mostra o conteúdo de BYTE1 antes e depois da execução da instrução.

Antes	Depois
BX	BX
0002h	FFFEh

Tabela 15.9 - Execução de NEG BX

Agora vamos, através de alguns exemplos, mostrar o uso destas instruções em tradução de sentenças escritas em linguagem de alto nível.

Exemplo 1:

Seja a sentença abaixo em Linguagem de Alto Nível:

```
A = 5 - A
```

Esta sentença significa que a variável **A** recebe o resultado de 5 menos o valor anterior de **A**

A tradução em linguagem assembly 8086 é:

```
MOV AX,5
SUB AX,A
MOV A,AX
```

Exemplo 2:

Seja a sentença abaixo em Linguagem de Alto Nível:

```
A = B - 2 x A
```

Esta sentença significa que a variável **A** recebe o resultado de B menos duas vezes o valor anterior de **A**

A tradução em linguagem assembly 8086 é:

```
MOV AX,B
SUB AX,A
```

```
SUB AX,A
MOV A,AX
```

15.3 As instruções de movimentação de dados, de adição e de subtração, e os FLAGS

Como já vimos, os FLAGS são responsáveis por mostrar o estado da UCP após a execução das instruções. Na tabela 15, abaixo, temos quais FLAGS são afetados, pelas instruções vistas neste capítulo.

INSTRUÇÃO	FLAGS ALTERADOS
MOV	NENHUM
XCHG	NENHUM
LEA	NENHUM
ADD/SUB	TODOS
INC/DEC	TODOS, EXCETO CF
NEG	TODOS (CF = 1, A NÃO SER QUE O RESULTADO SEJA IGUAL A ZERO, OF = 1 SE O OPERANDO FOR A PALAVRA 8000h OU UM BYTE 80h)

Tabela 15.10 - Os FLAGS e as instruções de movimentação de dados e aritméticas.

15.4 Instruções lógicas

As instruções lógicas, além de executar as respectivas funções lógicas, são utilizadas para:

- zerar (*reset*) ou limpar (*clear*) um bit, ou seja forçar ele a ter o valor 0;
- forçar ao valor 1 (*set*) um bit;
- verificar valores de bits, através de operações lógicas com máscaras específicas.

As instruções lógicas válidas para o 8086 seguem as funções constantes na tabela 15.11, que na verdade representa as tabelas verdades das funções **and** (e lógico), **or** (ou lógico), **not** (inversão) e **xor** (ou-exclusivo), como visto na parte 2 deste livro.

A	B	A AND B	A OR B	A XOR B	NOT A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Tabela 15.11 - Tabelas verdade das funções lógicas constantes do Conjunto de Instrução do 8086

As instruções lógicas AND, OR e XOR

As instruções lógicas **AND**, **OR** e **XOR** têm a mesma sintaxe, apenas o mnemônico é diferente. Portanto temos:

Sintaxe da instrução AND:

AND destino, fonte

Sintaxe da instrução OR:

OR destino, fonte

Sintaxe da instrução XOR:

XOR destino, fonte

A tabela 15.12 nos mostra as possibilidades de operações lógicas de dados. É permitida a operação lógica de dados entre dois registradores de dados, entre um registrador e uma posição de memória, de uma constante com um registrador e de uma constante com uma posição de memória.

OPERANDO FONTE	OPERANDO DESTINO	
	REGISTRADOR DE DADOS	MEMÓRIA
REGISTRADOR DE DADOS	SIM	SIM
MEMÓRIA	SIM	NÃO
CONSTANTE	SIM	SIM

Tabela 15.12 - Possibilidades de adição e subtração de dados - instruções lógicas

Abaixo temos exemplos de instruções válidas, para a soma e subtração de operandos.

Estas instruções alteram os FLAGS SF, ZF, PF, de acordo com o valor armazenado no operando destino. O FLAG AF não é alterado e os FLAGS CF e OF são zerados.

As instruções abaixo são exemplos de instruções válidas:

```

XOR AX,BX      ; operador XOR aplicado aos conteúdos de AX e BX,
                ; resultado em AX

AND CH,01h     ; operador AND aplicado ao conteúdo de CH, tendo
                ; como fonte o valor imediato 01h = 0000 0001b

OR WORD1,BX    ; operador OR entre conteúdos da posição de
                ; memória WORD1 e de BX, resultado armazenado em
                ; WORD1

```

Vamos supor inicialmente que AL tenha 0Ah (1010h) e BL 0Eh (1110h). Depois da execução das instruções:

AND BL,AL

OR BL,AL

XOR BL,AL

Teremos:

Valores após a execução	BL	AL
AND BL,AL	1010h	1110h
OR BL,AL	1110h	1110h
XOR BL,AL	0000h	1110h

Tabela 15.13 - Valores após a execução de instruções lógicas

Criação de uma máscara

As máscaras são números binários, projetados especificamente para uma função de manipular bits por meio de operações lógicas. Para o projeto destas máscaras, temos que levar em conta os seguintes princípios:

- a função AND pode ser utilizada para zerar (*clear* ou *reset*) bits específicos, para isto basta ter um 0 na posição que se deseja este efeito.
 - A função OR pode ser utilizada para forçar ao valor 1 (*set*) bits específicos, para isto basta ter um 1 na posição em que se deseja este efeito.
 - A função XOR pode ser utilizada para complementar (*invert*) bits específicos para isto basta ter um 1 na posição em que se deseja este efeito.
-

A seguir vamos exemplificar a definição de máscaras para um fim específico.

Exemplo 1: Para forçar ao valor 1 os bits MSB e LSB do registrador AX, dado AX = 7444h, executaremos a instrução:

OR AX,8001h

A palavra 8001h é a máscara. Ela foi definida porque queremos fazer com que os bits MSB e LSB sejam forçados ao valor 1, por isso a função é o OR e a máscara deve ter 1 no MSB e 1 no LSB. Os demais bits têm que ser zero para manter os bits originais de AX:

AX (antes)	→	0111 0100 0100 0100b	→	7444h
		1000 0000 0000 0001b	→	8001h
OR		<hr/>		
AX (depois)	→	1111 0100 0100 0101b	→	F445h

Exemplo 2: Converter o código ASCII de um dígito numérico em seu valor binário:

AND AL,0Fh ;em substituição a: SUB AL,30h

AL (antes)	→	0011 0111b	→	37h = "7" = 55d
		0000 1111b	→	0Fh
AND		<hr/>		
AL (depois)	→	0000 0111b	→	07h = 7d (valor sete)

Exemplo 2: Converter uma letra minúscula em maiúscula, supondo o caractere em AL:

AND AL,0DFh

AL (antes)	→	0110 0001b	→	61 h = "a"
		1101 1111b	→	DFh
AND				
AL (depois)	→	0100 0001b	→	41h = "A"

Obs: Para esta conversão, tem-se apenas que zerar o bit 5 de AL.

Exemplo 3: Limpando (zerando) um registrador:

XOR AX, AX

AX (antes)	→	0111 0100 0100 0100b	→	7444h
AX (antes)	→	0111 0100 0100 0100b	→	7444h
XOR				
AX (depois)	→	0000 0000 0000 0000b	→	0000h = 0

Observação:

Esta forma é mais rápida de executar do que as opções MOV AX,0000h e SUB AX,AX

Exemplo 3: Testando se o conteúdo de algum registrador é zero:

OR CX, CX

CX (antes)	→	0111 0100 0100 0100b	→	7444h
CX (antes)	→	0111 0100 0100 0100b	→	7444h
OR				
CX (depois)	→	0111 0100 0100 0100b	→	7444h (não é 0)

Observação:

Esta operação deixa o registrador CX inalterado, modifica o FLAG ZF somente quando o conteúdo de CX é realmente zero. É mais rápido do que `CMP CX,0000h`

A instrução lógica NOT

A instrução lógica **NOT** a seguinte sintaxe:

NOT destino

É usada para aplicar o operador lógico NOT em todos os bits de um registrador e de uma posição de memória. O resultado é a complementação (inversão) de todos os bits. Nenhum FLAG é alterado.

Abaixo temos exemplos válidos da utilização da instrução NOT.

```
NOT AX      ; inverte todos os bits de AX
NOT AL      ; inverte todos os bits de AL
NOT BYTE1   ; inverte todos os bits do conteúdo da posição de
              ; memória definida pelo nome BYTE1
```

Se tivermos, por exemplo, a palavra 81h armazenada na posição de memória WORD1, e executarmos:

NOT WORD1

Teremos:

	WORD1
Antes	81h = 1000 0001b
Depois	7Eh = 0111 1110b

Tabela 15.14 - Execução da instrução NOT WORD1

A instrução lógica **TEST**

A instrução lógica **TEST** a seguinte sintaxe:

TEST destino, fonte

Esta instrução é usada para aplicar o operador lógico AND em dois operandos, como mostra a tabela 15.15.

OPERANDO FONTE	OPERANDO DESTINO	
	REGISTRADOR DE DADOS	MEMÓRIA
REGISTRADOR DE DADOS	SIM	SIM
MEMÓRIA	SIM	NÃO
CONSTANTE	SIM	SIM

Tabela 15.15 – Possibilidade de combinação de operandos para a instrução **TEST**

A diferença desta instrução para a instrução **AND**, é que ela não altera o destino. Apenas os **FLAGS** **SF**, **ZF**, **PF** são alterados de acordo com o resultado, o **FLAG** **AF** não é afetado e **CF** e **OF** ficam zerados.

Alguns exemplos de utilização válida para a instruções **TEST**, estão a seguir:

```
TEST AX,BX      ; operação AND entre AX e BX, não há resultado,  
                  ; mas apenas alteração dos FLAGS ZF, SF e PF  
TEST AL,01h    ; operação AND entre AL e o valor imediato 01h
```

Exemplo 4 - Suponha, por exemplo, que queremos testar se um número, armazenado em AX e inicialmente 44h, é par, sem alterá-lo. A instrução abaixo faz isto:

```
TEST AX,0001h
```

A máscara 0001h serve para testar se o conteúdo de AX é PAR (todo número binário PAR possui um zero no LSB). O número 4444h é PAR, pois o seu LSB vale zero (0100010**0**b). A operação AND entre 4444h e 0001h produz como resultado 0000h, alterando ZF para 1. Portanto se testarmos ZF teremos a informação de o número em AX é par ou ímpar. O resultado não é armazenado em AX, somente ZF é modificado por TEST.

Exemplo 5 - Vamos escrever um trecho de programa que salte para o rótulo PONTO2 se o conteúdo de CL for negativo:

```
....  
TEST CL,80h      ; 80h é a máscara 10000000b  
JNZ PT2  
....  
(o programa prossegue, pois o número é positivo)  
....  
PONTO2: ....  
(o programa opera aqui com o número negativo)  
....
```

15.5 Instruções de Deslocamento e Rotação

As Instruções de deslocamento (*shift*) além de permitir a alteração de posição de bits, ainda possibilitam:

- Multiplicar um número por dois a cada deslocamento para a esquerda, de uma casa binária
- Dividir um número por dois a cada deslocamento para a direita, de uma casa binária

Observação:

Os bits deslocados para fora da palavra ou do byte são perdidos

As Instruções de rotação (*rotate*) permitem deslocar de forma circular (em anel) para a esquerda ou para a direita, sem que nenhum bit seja perdido.

Instruções de Deslocamento

Temos dois tipos de deslocamento: o deslocamento lógico e o deslocamento aritmético. A diferença entre os dois é que o aritmético leva em conta o sinal do número e o lógico não. Na prática, os deslocamentos lógicos e aritméticos para a esquerda funcionam da mesma maneira, ou seja, o bit mais a esquerda (MSB) sai da palavra, todos os outros são deslocados, uma posição, para a direita e colocamos um zero no lugar LSB. No deslocamento lógico à esquerda, deslocamos, uma posição, todos os bits para a esquerda e colocamos um zero no MSB. No deslocamento aritmético para esquerda, procedemos da mesma maneira, com exceção do bit colocado no MSB, que tem que ter o mesmo valor que o anterior, isto é, estamos mantendo o sinal do número.

As instruções de deslocamentos obedecem à seguinte sintaxe:

```
Sxx destino, 1  
Sxx destino, CL
```

Estas instruções são usadas para deslocar, para a esquerda ou para a direita, de 1 bit ou tantos quantos CL indicar. Eles podem deslocar o conteúdo de um registrador ou de uma posição de memória.

A generalização do mnemônico **Sxx** corresponde a:

- **SHL** - Deslocamento para a esquerda (*Shift Left*).

A figura 15.1 mostra o esquema de funcionamento da instrução SHL.

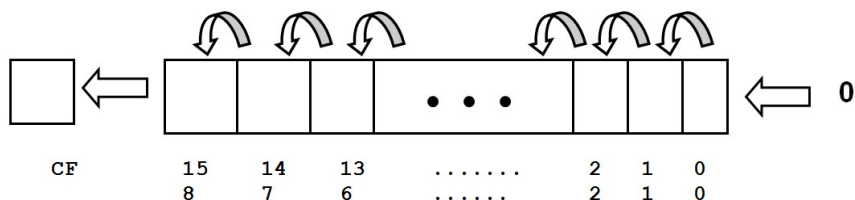


Figura 15.1 - Esquema de funcionamento da instrução SHL

- **SAL** - Deslocamento aritmético para a esquerda (*Shift Arithmetic Left*).

A figura 15.2 mostra o esquema de funcionamento da instrução SAL.

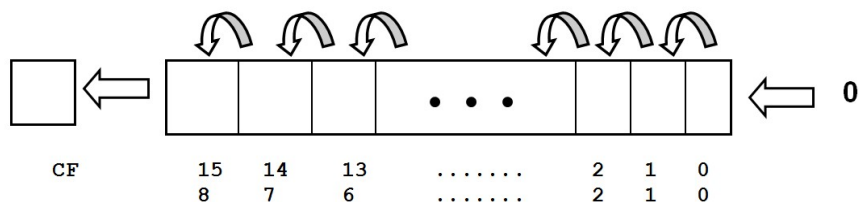


Figura 15.2 - Esquema de funcionamento da instrução SAL

- **SHR** - Deslocamento para a direita (*Shift Right*).

A figura 15.3 mostra o esquema de funcionamento da instrução SHR.

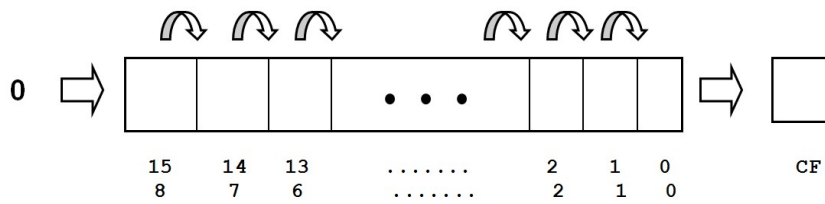


Figura 15.3 - Esquema de funcionamento da instrução SHR

- **SAR** - Deslocamento aritmético para a direita (*Shift Arithmetic Right*).

A figura 15.4 mostra o esquema de funcionamento da instrução SAR

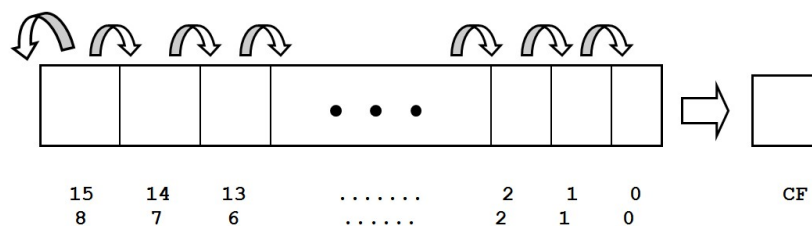


Figura 15.4 - Esquema de funcionamento da instrução SAR

Estas instruções afetam os FLAGS SF, ZF, PF, de acordo com o resultado do último deslocamento, o FLAG AF não é afetado, o CF contém o último bit deslocado para fora e OF será 1 se ocorrer troca de sinal após o último deslocamento.

A seguir, alguns exemplos de utilização válida destas instruções:

```
SHL AX,1      ; desloca os bits de AX para a esquerda
               ; 1 casa binária, sendo o LSB igual a zero
SAL BL,CL     ; desloca os bits de BL para a esquerda
               ; tantas casas binárias quantas CL
```

```

; indicar, os bits menos significativos são
; zero (mesmo efeito de SHL)
SAR DH,1    ; desloca os bits de DH para a direita 1
; casa binária, sendo que o MSB mantém
; o sinal

```

Instruções de rotação

Diferentemente das instruções de deslocamento, nas instruções de rotação, nenhum bit é perdido, uma vez que o bit que sai de uma extremidade entra pela outra. Temos aqui também depois tipos de rotação uma através do FLAG CF e outra sem ser através do CF. Na rotação, sem ser através do CF, para a esquerda, o bit que sai na posição MSB entra na posição LSB, e à direita o contrário. Na rotação através do CF, à esquerda, o bit que sai do MSB entra no CF e o que estava no CF entra no LSB, e à direita o contrário.

As instruções de rotação obedecem à seguinte sintaxe:

```

Rxx destino, 1
Rxx destino, CL

```

Estas instruções são usadas para deslocar, em anel, para a esquerda ou para a direita, de 1 bit ou tantos quantos CL indicar. Eles podem rotacionar o conteúdo de um registrador ou de uma posição de memória.

A generalização do mnemônico Rxx corresponde a:

- **ROL** - Rotacionar para a esquerda (*Rotate Left*).

A figura 15.5 mostra o esquema de funcionamento da instrução ROL

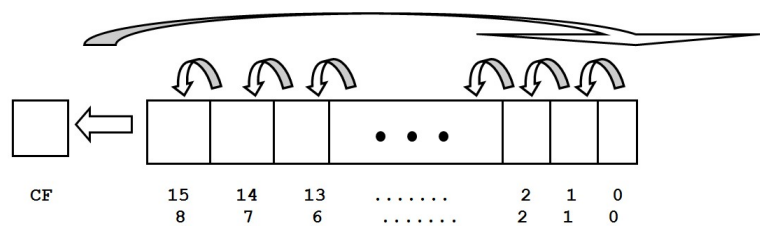


Figura 15.5 - Esquema de funcionamento da instrução ROL

- **ROR** - Rotacionar para a direita (*Rotate Right*).

A figura 15.6 mostra o esquema de funcionamento da instrução ROR

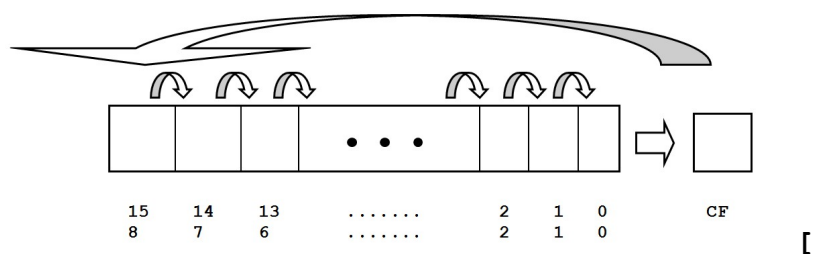


Figura 15.6 - Esquema de funcionamento da instrução ROR

- **RCL** - Rotacionar para a esquerda através do FLAG CF (*Rotate Carry Left*).

A figura 15.7 mostra o esquema de funcionamento da instrução RCL

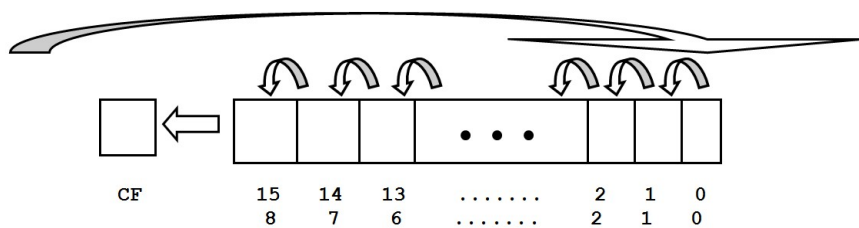


Figura 15.7 - Esquema de funcionamento da instrução RCL

- **RCR** - Rotacionar para a direita através do FLAG CF (*Rotate Carry Right*).

A figura 15.8 mostra o esquema de funcionamento da instrução RCL

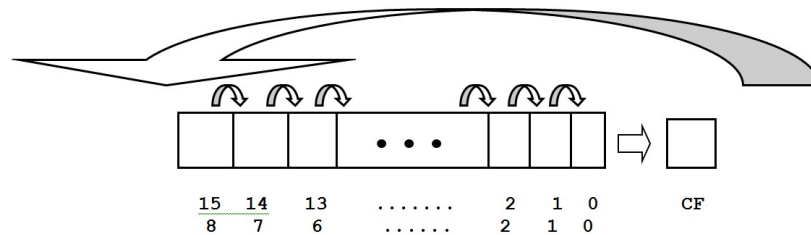


Figura 15.8 - Esquema de funcionamento da instrução RCL

Estas instruções afetam os FLAGS SF, ZF, PF, de acordo com o resultado da última rotação, o FLAG AF não é afetado, o CF contém o último bit deslocado para fora e OF será 1 se ocorrer troca de sinal após a última rotação.

A seguir, alguns exemplos de utilização válida destas instruções:

```
ROL AX,1      ; desloca os bits de AX para a esquerda 1
               ; casa binária, sendo o MSB é reinserido na
               ; posição LSB

ROR BL,CL     ; desloca os bits de BL para a direita tantas
               ; casas binárias quantas CL indicar, os bits
               ; menos significativos são reinseridos um-a-um no
               ; MSB

RCR DH,1      ; desloca os bits de DH para a direita 1 casa
               ; binária, sendo que o MSB recebe CF e o LSB é
               ; salvo em CF
```

Exemplos

1 - Fazer um programa que permita a entrada de números binários, com a seguinte especificação:

- *String* de caracteres "0's" e "1's" fornecidos pelo teclado;
- CR é o marcador de fim de *string*;
- BX é assumido como registrador de armazenamento;
- Número com no máximo de 16 bits.

Algoritmo básico em pseudocódigo:

```
Limpa BX
Entra um caractere "0" ou "1"
WHILE caractere diferente de CR DO
  Converte caractere para valor binário
  Desloca BX 1 casa para a esquerda
  Insere o valor binário lido no LSB de BX
  Entra novo caractere
END_WHILE
```

Trecho de programa implementado em linguagem de montagem:

```
...
MOV CX,16      ; inicializa contador de dígitos
MOV AH,1h      ; função DOS para entrada pelo teclado
XOR BX,BX      ; zera BX -> terá o resultado
INT 21h        ; entra, caractere está no AL
; while
PT:  CMP AL,0Dh ; é CR?
     JE  FIM    ; se sim, termina o WHILE
     AND AL,0Fh ; se não, elimina 30h do caractere
     SHL BX,1   ; abre espaço para o novo dígito
     OR  BL,AL  ; insere o dígito no LSB de BL
     INT 21h    ; entra novo caractere
```

```

        LOOP PT      ; controla o máximo de 16 dígitos
; end_while
FIM:
...

```

2 - Fazer um programa que permita a saída de números binários, com a seguinte especificação:

- BX é assumido como registrador de armazenamento;
- Total de 16 bits de saída;
- *String* de caracteres "0's" e "1's" é exibido no monitor de vídeo.

Algoritmo básico em pseudocódigo:

```

FOR 16 vezes DO
  rotação de BX à esquerda 1 casa binária (MSB vai para o CF)
  IF CF = 1
  THEN  exibir no monitor caractere "1"
  ELSE  exibir no monitor caractere "0"
  END_IF
END_FOR

```

Trecho de programa implementado em linguagem de montagem:

```

...
    MOV CX,16      ;inicializa contador de bits
    MOV AH,02h     ;prepara para exibir no monitor
;for 16 vezes do
PT1: ROL BX,1      ;desloca BX 1 casa à esquerda
;if CF = 1
    JNC PT2        ;salta se CF = 0
;then

```

```

        MOV DL, 31h      ;como CF = 1
        INT 21h          ;exibe na tela "1" = 31h
;else
PT2: MOV DL, 30h        ;como CF = 0
        INT 21h          ;exibe na tela "0" = 30h
;end_if
        LOOP PT1         ;repete 16 vezes
;end_for
Entrada de números hexadecimais

```

3 - Fazer um programa que permita a entrada de números hexadecimais, com a seguinte especificação:

- Entrada de números hexadecimais:
- BX é assumido como registrador de armazenamento;
- *String* de caracteres "0" a "9" ou de "A" a "F", digitado no teclado;
- Máximo de 16 bits de entrada ou máximo de 4 dígitos hexadecimais.

Algoritmo básico em pseudocódigo:

```

Inicializa BX
Entra um caracter hexa
WHILE  caracter diferente de CR  DO
  Converte caracter para binário
  Desloca BX 4 casas para a esquerda
  Insere valor binário nos 4 bits inferiores de BX
  Entra novo caracter
END_WHILE

```

Trecho de programa implementado em linguagem de montagem:

```
...
XOR BX,BX      ;inicializa BX com zero
MOV CL,4       ;inicializa contador com 4
MOV AH,1h      ;prepara entrada pelo teclado
INT 21h        ;entra o primeiro caractere
;while
PT:  CMP AL,0Dh  ;é o CR ?
     JE  FIM
     CMP AL, 39h ;caractere número ou letra?
     JG  LTR     ;caractere já está na faixa ASCII
     AND AL, 0Fh ;número: retira 30h do ASCII
     JMP DSL
LTR: SUB AL,37h  ;converte letra para binário
DSL: SHL BX,CL   ;desloca BX 4 casas à esquerda
     OR BL,AL    ;insere valor nos bits 0 a 3 de BX
     INT 21h     ;entra novo caractere
     JMP PT      ;faz o laço até que haja CR
;end_while
FIM:  ...
```

4 - Fazer um programa que permita a saída de números hexadecimais, com a seguinte especificação:

- BX é assumido como registrador de armazenamento;
 - Total de 16 bits de saída;
 - *String* de caracteres HEXA é exibida no monitor de vídeo.
-

Algoritmo básico em pseudocódigo:

```
FOR 4 vezes DO
Mover BH para DL
Deslocar DL 4 casas para a direita
IF DL < 10
THEN converte para caractere na faixa 0 a 9
ELSE converte para caractere na faixa A a F
END_IF
Exibição do caractere no monitor de vídeo
Rodar BX 4 casas à esquerda
END_FOR
```

Trecho de programa implementado em linguagem de montagem:

```
...
;BX já contem número binário
    MOV CH,4 ;CH contador de caracteres hexa
    MOV CL,4 ;CL contador de deslocamentos
    MOV AH,2h ;prepara exibição no monitor
;for 4 vezes do
PT:  MOV DL,BH ;captura em DL os oito bits mais ;
        ;significativos de BX
    SHR DL,CL ;resta em DL os 4 bits mais significativos
        ;de BX
;if DL , 10
    CMP DL, 0Ah ;testa se é letra ou número
    JAE LTR
;then
    ADD DL,30h ;é número: soma-se 30h
    JMP PT1
;else
LTR: ADD DL,37h ;ao valor soma-se 37h -> ASCII
```

```
;end_if
PT1: INT 21h      ;exibe
      ROL BX,CL    ;roda BX 4 casas para a direita
      DEC CH
      JNZ PT       ;faz o FOR 4 vezes
;end_for
```

15.6 Instruções de multiplicação e divisão

Instruções de multiplicação

Temos dois tipos de multiplicação, a sinalizada e não sinalizada. Para cada uma delas temos um mnemônico: o **MUL** (*multiply*), para multiplicação de números não sinalizados e **IMUL** (*integer multiply*) para multiplicação de números sinalizados. As sintaxes destas duas instruções são:

MUL fonte

IMUL fonte

Para multiplicação com números em formato *byte*, o operando fonte deverá ser um registrador de 8 bits ou uma variável de tipo DB. Neste caso, o segundo operando da multiplicação, obrigatoriamente estará em AL e o resultado, um número de 16 bits, será guardado em AX.

Quando a multiplicação for com números em formato *Word*, o operando fonte deverá ser um registrador de 16 bits ou uma variável do tipo DW. Aqui, o segundo operando é estará, obrigatoriamente em AX e o resultado de 32 bits (tamanho *doubleword*) será armazenado no par de registradores DX:AX, em DX os 16 bits mais significativos (*high word*) e em AX os 16 bits menos significativos (*low word*).

As instruções de multiplicação alteram os FLAGS desta forma:

- SF, ZF, AF, PF ficarão indefinidos;
- Após MUL, CF/OF (ambos) será 0, se a metade superior do resultado for 0 e será 1, caso contrário
- Após IMUL, CF/OF (ambos) será 0, se a metade superior do resultado for a extensão do sinal da metade inferior e será 1, caso contrário

Exemplos

1 - Vamos supor inicialmente que AX contenha 0001h BX contenha FFFFh.
Para a instrução:

MUL BX

Como em AX = 1 (número não sinalizado) e BX=65535, teríamos um novo valor em AX que seria 0000FFFFh (65535d), que é o resultado de uma multiplicação não sinalizada de AX por BX.

Para a instrução:

IMUL BX

Como em AX = +1 (número sinalizado) e BX = -1, teríamos um novo valor em AX que seria 0000FFFFh (-1), que é o resultado de uma multiplicação sinalizada de AX por BX.

O quadro abaixo mostra o resultado completo da execução das instruções acima.

Instrução	Resultado decimal	Resultado hexadecimal	DX	AX	CF/OF
MUL BX	65535	0000FFFFh	0000h	FFFFh	0
IMUL BX	-1	FFFFFFFFh	FFFFh	FFFFh	0

2 - Suponha agora que AX contenha 0FFF h:

No quadro abaixo, temos a execução das instruções:

MUL AX

e

IMUL AX

Instrução	Resultado decimal	Resultado hexadecimal	DX	AX	CF/OF
MUL AX	16769025	00FFE001h	00FFh	E001h	1
IMUL AX	+16769025	00FFE001h	00FFh	E001h	1

3 - Agora, trabalhando com operações de 8 bits, supor que AL contenha 80h (= 10000000b = 128d ou -128d, se sinalizado) e BL tenha FFh (= 11111111b = 255d ou -1, se sinalizado):

No quadro abaixo, temos a execução das instruções:

MUL BL

e

IMUL BL

Instrução	Resultado decimal	Resultado hexadecimal	AH	AL	CF/OF
MUL BL	32640	7F80h	7Fh	80h	1
IMUL BL	+128	0080h	00h	80h	1

Instruções de divisão

Analogamente multiplicação, a divisão também tem dois tipos de operação a sinalizada e a não sinalizada. Para a divisão de números não sinalizados temos o mnemônico **DIV** (*divide*) e **IDIV** (*integer divide*) para a divisão de números sinalizados. As sintaxes destas duas instruções são:

DIV fonte

IDIV fonte

Onde o operando fonte é o divisor de 8 bits ou de 16 bits, se a operação for de 8 ou 16 bits respectivamente. Ele não pode ser uma constante. Para uma operação de 8 bits, onde o divisor é um registrador de 8 bits ou uma posição de memória de 8 bits (tipo DB), o dividendo tem que ser uma palavra de 16 bits, armazenado em AX. Após a execução da divisão, o quociente (de 8 bits) estará em AL e o resto (também de 8 bits) estará em AH. Se a divisão for de 16 bits, onde o divisor é uma palavra de 16 bits, armazenado em AX ou em uma posição de memória de 16 bits (tipo DW), o dividendo será de 32 bits, armazenado no par DX:AX. O quociente desta divisão estará em AX (16 bits) e o resto em DX (16 bits). Nestas operações, todos os FLAGS são afetados e ficam indefinidos. O resto, em divisão com números sinalizados, tem o mesmo sinal do dividendo.

Exemplos:

1 - Suponha que DX e AX contenham, respectivamente 0000h e 0005h (5d ou +5d, se sinalizado), e BX contenha FFFEH (=65534d ou -2d, se sinalizado).

O quadro abaixo mostra o resultado da execução das instruções:

DIV BX**IDIV BX**

Instrução	Quociente decimal	Resto decimal	AX	DX
DIV BX	0	5	0000h	0005h
IDIV BX	-2	1	FFFEh	0001h

2 - Suponha agora que AX contenha 0005h (= 5d ou +5d, se sinalizado) e BL contenha FFh (= 256d ou -1d, se sinalizado).

O quadro abaixo mostra o resultado da execução das instruções:

DIV BX**IDIV BX**

Instrução	Quociente decimal	Resto decimal	AL	AH
DIV BL	0	5	00h	05h
IDIV BL	-5	0	FBh	00h

3 - Suponha que AX contenha 00FBh (= 251d ou +251d, se sinalizado) e BL contenha FFh (= 255d ou -1, se sinalizado):

Instrução	Quociente decimal	Resto decimal	AL	AH
DIV BL	0	251	00h	FBh
IDIV BL	-251 *	-	-	-

Observação:

Como -251 não cabe em AL (8 bits), ocorre o que chamamos de "estouro de divisão" (*DIVIDE OVERFLOW*), que é um erro que faz com que o programa termine.

Extensão do sinal do dividendo

Nas operações de divisão sinalizadas, existe o cuidado de estender o valor do sinal para a parte superior da palavra, *byte* ou *word*, caso o dividendo de 16 bits ocupe apenas AL ou o de 32 bits ocupe apenas AX. Para esclarecer melhor vamos verificar as seguintes situações:

▪ Em operações em formato word:

Caso o dividendo de uma divisão (composto de **DX:AX**) ocupe apenas o registrador AX, o registrador DX deve ser preparado, pois é sempre considerado na execução da operação. Se a instrução for **DIV**, o DX deve ser zerado, pois estamos tratando de operação de números não sinalizados. Se for **IDIV**, o DX deve ter a extensão de sinal de AX, pois se trata de operações de números sinalizados. A instrução **CWD**, faz a conversão de um palavra *word* para *doubleword*, estendendo o sinal de AX para DX. A sintaxe é:

CWD

Note que é uma instrução sem operandos, e deve ser usada antes da operação IDIV, caso o dividendo só ocupe AX e o divisor seja uma palavra de 16 bits.

▪ Em operações em formato byte

Analogamente, Caso o dividendo de uma divisão (composto por **AX**) ocupe apenas AL, AH deve ser preparado, pois é sempre considerado

na execução da operação. Se a instrução for **DIV**, o registrador AH deve ser zerado e se for a instrução **IDIV**, AH deve ter a extensão de sinal de AL. A instrução CBW faz a conversão **byte** para **word** e estende o sinal de AL para AH. A sintaxe é:

CBW

Também é uma instrução sem operandos e deve ser usada antes de IDIV quando o dividendo só ocupar AL e o divisor ser uma palavra de 8 bits. As instruções CWD e CBW não afetam os FLAGS.

Exemplos:

1 - Vamos criar um trecho de programa que divida -1250 por 7.

```
...  
MOV  AX, -1250      ;AX recebe o dividendo  
CWD                      ;estende o sinal de AX para DX  
MOV  BX, 7          ;BX recebe o divisor  
IDIV BX              ;executa a divisão  
                        ;após a execução, AX recebe o  
                        ;quociente e DX recebe o resto  
...
```

2 - O trecho de programa que divide a variável sinalizada XBYTE por -7, fica:

```
...  
MOV  AL, XBYTE       ;AL recebe o dividendo  
CBW                      ;estende o sinal (eventual) de  
                        ;AL para AH  
MOV  BL, -7          ;BL recebe o divisor
```

```
IDIV BL          ;executa a divisão
                  ;após a execução, AL recebe o
                  ;quociente e AH recebe o resto
...

```

Exemplos

1 - Entrada e Saída de números decimais

Já vimos anteriormente, quando estudamos as instruções lógicas e de deslocamento, como ficariam as rotinas de entrada e saída de números hexadecimais e binários. Vamos agora mostrar as rotinas de entrada e saída de números decimais.

Entrada de números decimais.

Vamos desenvolver uma rotina de entrada de números decimais, com a seguinte especificação:

- *String* de caracteres números de 0 a 9, fornecidos pelo teclado;
- CR é o marcador de fim de *string*;
- AX é assumido como registrador de armazenamento;
- Valores decimais permitidos na faixa de - 32768 a + 32767, ou seja, 16 bits sinalizados;
- Sinal negativo deve ser apresentado, se existir.

O Algoritmo básico em pseudo-linguagem para esta rotina ficaria:

```
total = 0
negativo = FALSO
ler um caracter
CASE  caracter  OF
    ' - '      :  negativo = VERDADEIRO   e   ler um caracter
    ' + '      :  ler um caracter

```

```

END_CASE
REPEAT
    converter caracter em valor binário
    total  = 10 x total + valor binário
    ler um caracter
UNTIL    caracter é um carriage return (CR)
IF  negativo = VERDADEIRO
    THEN  total = - (total)
END_IF

```

O procedimento, em assembly do 8086, para esta rotina é:

```

ENTDEC      PROC
; Le um número decimal da faixa de -32768 a +32767
; variáveis de entrada: nenhuma (entrada de dígitos pelo
; teclado)
; variáveis de saída: AX (valor binário equivalente do número;
; decimal)

    PUSH BX
    PUSH CX
    PUSH DX          ; salvando registradores que serão
                     ; usados
    XOR BX,BX        ; BX acumula o total, valor inicial 0
    XOR CX,CX        ; CX indicador de sinal (negativo = 1),
                     ; inicial = 0

    MOV AH,1h
    INT 21h          ; le caractere no AL
    CMP AL, '-'      ; sinal negativo?
    JE MENOS
    CMP AL, '+'      ; sinal positivo?
    JE MAIS
    JMP NUM          ; se não é sinal, então vá processar o
                     ; caractere

```

```

MENOS:    MOV CX,1          ; negativo = verdadeiro
MAIS:     INT 21h          ; le um outro caractere
NUM:      AND AX,000Fh     ; junta AH a AL, converte caractere
                                ; para binário
                                ;
                                ; salva AX (valor binário) na pilha
                                ;
                                ; prepara constante 10
                                ;
                                ; AX = 10 x total, total está em BX
                                ;
                                ; retira da pilha o valor salvo, vai
                                ; para BX
                                ;
                                ; total = total x 10 + valor binário
                                ;
                                ;
                                ; le um caractere
                                ;
                                ; é o CR ?
                                ;
                                ; se não, vai processar outro dígito em
                                ; NUM
                                ;
                                ; se é CR, então coloca o total
                                ; calculado em AX
                                ;
                                ; o numero é negativo?
                                ;
                                ; não
                                ;
                                ; sim, faz-se seu complemento de 2
SAIDA:    POP DX
                                ;
                                ;
                                ; restaura os conteúdos originais
                                ;
                                ; retorna a rotina que chamou
ENTDEC    ENDP

```

Saída de números decimais:

A rotina de saída de números decimais teria a seguinte especificação:

- AX é assumido como registrador de armazenamento;
 - Valores decimais na faixa de - 32768 a + 32767;
 - Exibe sinal negativo, se o conteúdo de AX for negativo;
-

- *String* de caracteres números de 0 a 9, exibidos no monitor de vídeo.

O algoritmo em pseudo-linguagem seria:

```

IF AX < 0 THEN      exibe um sinal de menos
                    substitui-se AX pelo seu complemento de 2
END_IF
contador = 0
REPEAT
    dividir quociente por 10
    colocar o resto na pilha
    contador = contador + 1
UNTIL quociente = 0
FOR contador vezes DO
    retirar um resto (número) da pilha
    converter para caractere ASCII
    exibir o caractere no monitor
END_FOR

```

A ideia da técnica de decomposição decimal do número em AX, utilizando a pilha, pode ser vista na figura 15.9.

Passo			Pilha	
5	2	dividido por com resto 2 -> 10 = 0	0002h	<- Topo
4	24	dividido por com resto 4 -> 10 = 2	0004h	
3	246	dividido por com resto 6 -> 10 = 24	0006h	
2	2461	dividido por com resto 1 -> 10 = 246	0001h	
1	24618	dividido por com resto 8 -> 10 = 2461	0008h	

Figura 15.9- Técnica de decomposição decimal do número usando a pilha.

A técnica é simples, divide-se o número por 10, até quociente ser 0, empilhando todos os restos. Os restos serão os algarismos que formam o número, empilhados do mais significativo (topo) para o menos significativo. Com isto o número fica decomposto.

O procedimento, em assembly 8086, referente à rotina de saída de números decimais é:

```
SAIDEC    PROC
; exibe o conteúdo de AX como decimal inteiro com sinal
; variáveis de entrada:  AX (valor binário equivalente do número
; decimal
; variáveis de saída: nenhuma (exibição de dígitos, direto no
; monitor de video)
        PUSH AX
        PUSH BX
        PUSH CX
        PUSH DX        ;salva na pilha os registradores usados
        OR AX,AX        ;prepara comparação de sinal
        JGE PT1        ;se AX maior ou igual a 0, vai para PT1
        PUSH AX        ;como AX menor que 0, salva o número na
                        ;pilha
        MOV DL,' - '    ;prepara o caractere ' - ' para sair
        MOV AH,2h       ;prepara exibição
        INT 21h         ;exibe ' - '
        POP AX          ;recupera o número
        NEG AX          ;troca o sinal de AX (AX = - AX)
                        ;obtendo dígitos decimais e salvando-os
                        ;temporariamente na pilha
PT1:     XOR CX,CX       ;inicializa CX como contador de dígitos
        MOV BX,10       ;BX possui o divisor
```

```

PT2:      XOR DX,DX      ;inicializa o byte alto do dividendo em
                        ;0 restante é AX

          DIV BX         ;após a execução, AX = quociente; DX =
                        ;resto

          PUSH DX        ;salva o primeiro dígito decimal na
                        ;pilha (1o. resto)

          INC CX         ;contador = contador + 1

          OR AX,AX       ;quociente = 0 ? (teste de parada)

          JNE PT2        ;não, continuamos a repetir o laço
                        ;exibindo os dígitos decimais (restos)
                        ;no monitor, na ordem inversa

          MOV AH,2h      ;sim, termina o processo, prepara
                        ;exibição dos restos

PT3:      POP DX         ;recupera dígito da pilha colocando-o
                        ;em DL (DH = 0)

          ADD DL,30h     ;converte valor binário do dígito para
                        ;caractere ASCII

          INT 21h        ;exibe caractere

          LOOP PT3       ;realiza o loop ate que CX = 0

          POP DX         ;restaura o conteúdo dos registros

          POP CX

          POP BX

          POP AX         ;restaura os conteúdos dos
                        ;registradores

          RET            ;retorna à rotina que chamou

SAIDEC   ENDP

```

Segue um exemplo de um programa que utiliza os procedimentos ENTDEC e SAIDEC.

```
TITLE PROGRAMA DE E/S DECIMAL

.MODEL SMALL

.STACK 100h

.DATA

MSG1 DB    'Entre um numero decimal na faixa de -32768 a 32767:$'
MSG2 DB    0Dh,0Ah,'Confirmando, você digitou:$'

.CODE

INCLUDE: C:\<diretorio_de_trabalho>\ENTDEC.ASM
INCLUDE: C:\<diretorio_de_trabalho>\SAIDEC.ASM

PRINCIPAL PROC

    MOV AX,@DATA
    MOV DS,AX
    MOV AH,9h
    LEA DX,MSG1
    INT 21h

; entrada do número
    CALL ENTDEC    ; chama procedimento ENTDEC
    PUSH AX        ; salva temporariamente o número na
                   ; pilha

; exibindo a segunda mensagem
    MOV AH,9h
    LEA DX,MSG2
    INT 21h        ; exhibe a segunda mensagem

; exibindo o número lido
    POP AX         ; recupera o número na pilha
    CALL SAIDEC    ; chama procedimento SAIDEC

; saída para o DOS
    MOV AH,4Ch
    INT 21h

PRNCIPAL ENDP

    END PRINCIPAL
```

CAPÍTULO 16 - INSTRUÇÕES DE CONTROLE DE FLUXO

Objetivos do Capítulo

Ao final desse capítulo o leitor estará apto a:

- Compreender a importância dos sinalizadores e sua utilização no controle do fluxo dos programas
- Conhecer e utilizar os principais comandos em linguagem de montagem para controle de fluxos

FUNDAMENTOS

16.1 Para que instruções de Salto?

Para programas em linguagem de montagem executarem tarefas complexas, eles devem ter uma forma de tomar decisões e repetir determinadas seções do código. Com este intuito, vamos mostrar como isso pode ser feito utilizando instruções de Salto.

As instruções de salto transferem o controle para outra parte do programa. Esta transferência pode ser incondicional ou pode depender de uma combinação particular dos FLAGS de estado.

As instruções de decisão e laço implementadas em linguagem de alto nível podem ser escritas em linguagem de montagem, como mostra o exemplo a seguir.

Um Exemplo de instruções de salto:

```

TITLE EXIBICAO DE CARACTERES ASCII
.MODEL SMALL
.STACK 100H
.CODE
MAIN PROC
;inicialização de alguns registradores
    MOV AH,2      ;função DOS para exibição de caractere
    MOV CX,256    ;contador com o numero total de caracteres
    MOV DL,00H    ;DL inicializado com o primeiro ASCII
;
;definição de um processo repetitivo de 256 vezes
PRINT_LOOP:
    INT 21H        ;exibir caractere na tela
    INC DL        ;incrementa o caractere ASCII
    DEC CX        ;decrementa o contador
    JNZ PRINT_LOOP ;continua exibindo enquanto CX não for 0
                        ;quando CX = 0, o programa quebra a
                        ;sequencia do loop
;
;saída para o DOS
;
    MOV AH,4CH
    INT 21H
MAIN ENDP
END MAIN

```

Após a montagem, ligação e execução do código acima, teríamos o seguinte resultado:

Existem três categorias de SALTOS CONDICIONAIS:

- Saltos Sinalizados
- Saltos Não-Sinalizados
- Saltos de FLAG único

A tabela 16.1 nos mostra as instruções de saltos sinalizados, sua descrição e o FLAGS responsáveis pela execução do salto.

Saltos sinalizados		
Mnemônico	Descrição	Condições
JG ou JNLE	Salto se maior do que OU salto se não menor do que ou igual a	ZF = 0 E SF = OF
JGE ou JNL	Salto se maior do que ou igual a OU salto se não menor do que	SF = OF
JL ou JNGE	Salto se menor do que OU salto se não maior do que ou igual a	SF ≠ OF
JLE ou JNG	Salto se menor do que ou igual a OU salto se não maior do que	ZF = 1 OU SF ≠ OF

Tabela 16.1 - Saltos Sinalizados.

A tabela 16.2 nos mostra as instruções de saltos não-sinalizados, sua descrição e o FLAGS responsáveis pela execução do salto.

Saltos sinalizados		
Mnemônico	Descrição	Condições
JA ou JNBE	Salto se acima do que OU salto se não abaixo do que ou igual a	ZF = 0 E CF = 0
JAE ou JNB	Salto se acima do que ou igual a OU salto se não abaixo do que	CF = 0
JB ou JNAE	Salto se abaixo do que OU salto se não acima do que ou igual a	CF = 1
JBE ou JNA	Salto se abaixo do que ou igual a OU salto se não acima do que	ZF = 1 OU CF = 1

Tabela 16.2 - Saltos Não-Sinalizados.

A tabela 16.3 nos mostra as instruções de saltos de FLAG único, sua descrição e o FLAGS responsáveis pela execução do salto.

Saltos de FLAG único		
Mnemônico	Descrição	Condições
JE ou JZ	Salto se igual OU salto se igual a zero	ZF = 1
JNE ou JNZ	Salto se não igual OU salto se não igual a zero	ZF = 0
JC	Salto se há VAI-UM (<i>carry</i>)	CF = 1
JNC	Salto se não há VAI-UM (<i>not carry</i>)	CF = 0
JO	Salto se há <i>overflow</i>	OF = 1
JNO	Salto se não há <i>overflow</i>	OF = 0
JS	Salto se o sinal é negativo	SF = 1
JNS	Salto se o sinal é não-negativo (+)	SF = 0
JP ou JPE	Salto se a paridade é PAR (<i>even</i>)	PF = 1
JNP ou JPO	Salto se a paridade é IMPAR (<i>odd</i>)	PF = 0

Tabela 16.3 - Saltos de FLAG único.

16.3 A Instrução de Comparação - CMP

Além das instruções aritméticas, as condições de salto são frequentemente geradas por comparação entre dois operandos. Para isto temos uma instrução cujo mnemônico é **CMP** (*compare*). A sintaxe desta instrução é:

CMP destino, fonte

Esta instrução compara o destino com a fonte, realizando uma subtração entre o destino e a fonte, sendo que o resultado não é armazenado, apenas os FLAGS são alterados.

A tabela 16.4 nos mostra as combinações de operando do para destino-fonte.

OPERANDO FONTE	OPERANDO DESTINO	
	REGISTRADOR DE DADOS	MEMÓRIA
REGISTRADOR DE DADOS	SIM	SIM
MEMÓRIA	SIM	NÃO
CONSTANTE	SIM	SIM

Tabela 16.4 - Possibilidades de comparação de dados - instrução CMP.

Por exemplo, vamos supor que AX=7FFFh, BX=0001h e o seguinte trecho de código:

```
CMP AX,BX
JG BELOW
```

O resultado de CMP AX,BX é 7FFFh-0001h = 7FFEh. O comando JG BELOW, é um comando de salto (salto de maior do que - *Jump if greater than*). Para tanto, ele verifica os FLAGS ZF, SF e OF. Se todos forem iguais a ZERO (0) sua condição é verdadeira e ele redireciona o ponteiro de instrução para BELOW. No caso deste exemplo, ZF = SF = OF = 0, portanto, a próxima instrução a ser executada deverá estar em BELOW, ou seja, o salto será feito.

Na verdade o que temos é que se AX é maior que BX será feito o salto e o fluxo do programa é transferido para BELOW, caso contrário, não.

Neste outro exemplo, vamos supor que AX e BX contenham os mesmos números do exemplo anterior. O que o código abaixo fará?

```
MOV CX,AX      ;coloca o valor de AX em CX
CMP BX,CX      ;BX é maior?
JLE NEXT       ;se não, vá para NEXT
MOV CX,BX      ;senão, coloque BX em CX!!
NEXT:...
```

Evidentemente como a instrução JLE considera ambos AX e BX, como números sinalizados, a execução deste trecho fará com que maior dos números seja colocado em CX. No exemplo, CX será conterá o valor 7FFFh.

16.4 Instrução de Salto Incondicional - JMP

Além dos saltos condicionais, também temos um instrução que, se executada faz com que um salto seja feito. Chamamos de salto incondicional, cujo mnemônico é **JMP** e a sintaxe é:

JMP destino

onde destino é um rótulo no mesmo segmento da instrução JMP. Não existe limitação de faixa de endereçamento, dentro do segmento, como no salto condicional.

Exemplo:

```
TOP:
;corpo do loop
DEC CX
JZ EXIT
JMP TOP
EXIT:
MOV AX,BX
```

16.5 Estruturas de Linguagens de Alto Nível

Utilizando as instruções vistas até agora, vamos traduzir algumas estruturas complexas, de uma linguagem de alto nível.

Estrutura IF-THEN

Em pseudocódigo (próximo a uma linguagem de alto nível) teríamos a seguinte estrutura para este comando:

```
IF condição é verdadeira
  THEN
    Execute um conjunto de instruções
  END_IF
```

Esta estrutura nos diz que se condição for verdadeira, um conjunto de instruções será executado, caso contrário não.

A figura 16.1 nos mostra o fluxo do comando IF-THEN

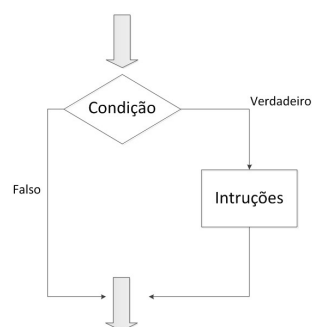


Figura 16.1 - Comando IF-THEN

Vamos fazer um trecho de programa, em linguagem de montagem do 8086, que substitua o número que está em AX pelo seu valor absoluto. Se fôssemos escrever um algoritmo em um pseudocódigo, teríamos:

```
IF AX < 0
  THEN
    Replace AX by -AX
  END_IF
```

Traduzindo para a linguagem de montagem:

```
;se AX < 0
    CMP AX,0      ;AX < 0?
    JNL END_IF    ;não, saia do IF
;then
    NEG AX        ;sim, mude o sinal de AX
END_IF:
```

Estrutura IF-THEN-ELSE

O trecho em pseudocódigo que representa este comando é:

```
IF condição é verdadeira
  THEN
    Execute um conjunto de instruções (verdade)
  ELSE
    Execute outro conjunto de instruções (falso)
  END_IF
```

Se a condição for verdadeira, um conjunto de instruções será executado, caso contrário, se for falsa, outro conjunto de instruções será executado.

A figura 16.2 nos mostra o fluxo do comando IF-THEN-ELSE

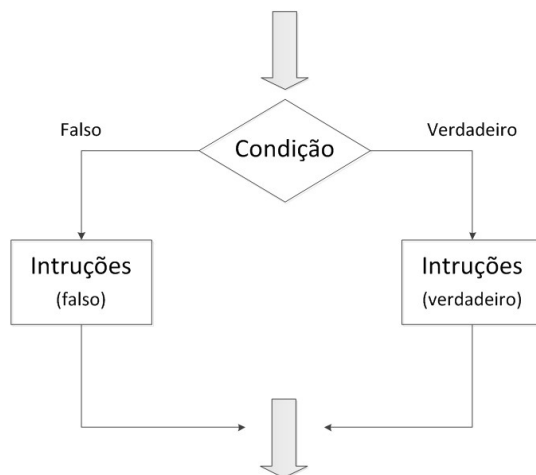


Figura 16.2 - Comando IF-THEN-ELSE

Vamos supor, por exemplo, que AL e BL contêm caracteres ASCII. Queremos escrever um trecho de programa que mostrar o caractere que menor em ordem alfabética.

O algoritmo em pseudocódigo ficaria:

```
IF AL <= BL  
  THEN  
    Mostre o caractere em AL  
  ELSE  
    Mostre o caractere em BL  
END_IF
```

A tradução em linguagem de montagem:

```

        MOV AH,2      ;preparando para exibir caractere
;se AL <= BL
        CMP AL,BL     ;AL <= BL?
        JNBE ELSE_    ;não, mostre caractere em BL
;then
        MOV DL,AL     ;move caractere a ser exibido
        JMP DISPLAY   ;vai para exibição
;else
        MOV DL,BL
ELSE_:
DISPLAY:
        INT 21h       ;exibe o caractere em DL
END_IF:
...

```

Observação:

O rótulo ELSE_ é usado, pois ELSE é uma palavra reservada.

Estrutura CASE

O comando CASE pode ser representado, em pseudocódigo, como:

```

CASE expressão
    Valor_1: conjunto1 de instruções
    Valor_2: conjunto2 de instruções
    Valor_3: conjunto3 de instruções
    ...
    Valor_N: conjuntoN de instruções
END_CASE

```

Nesta estrutura, expressão é testada. Se o seu valor faz parte de um universo de valores (Valor_1, Valor_2, ... , Valor_N), então o respectivo conjunto de instruções será executado. Os outros conjuntos de instruções serão ignorados.

A figura 16.3 nos mostra o fluxo do comando CASE

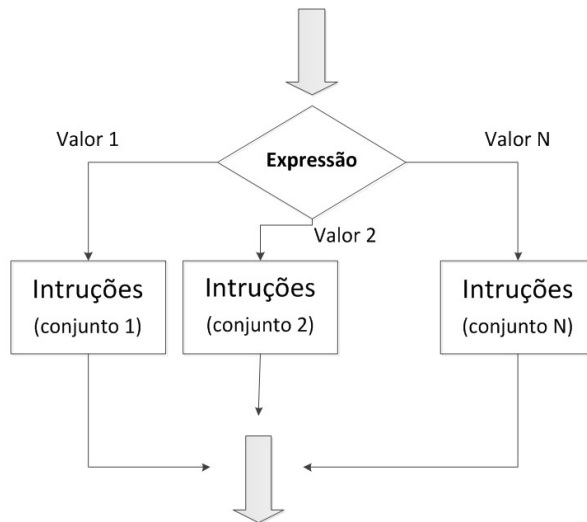


Figura 16.3 - Comando CASE

Por exemplo, vamos supor que queremos colocar -1 em BX se AX tiver um valor negativo, colocar zero em BX, se AX tiver um valor igual a zero, e colocar 1 em BX se AX tiver um valor positivo.

Para este problema podemos escrever o seguinte algoritmo:

```
CASE AX  
  <0:BX = -1  
  =0:BX = 0  
  >0:BX = 1  
END_CASE
```

O trecho de programa em linguagem de montagem seria:

```

;case AX
    CMP AX,0      ;testa AX
    JL NEGATIVO   ;AX < 0
    JE ZERO       ;AX = 0
    JG POSITIVO   ;AX > 0
NEGATIVO:
    MOV BX,-1     ;coloca -1 em BX pois AX negativo
    JMP END_CASE  ;e sai...
ZERO:
    MOV BX,0      ;coloca 0 em BX pois AX = 0
    JMP END_CASE  ;e sai...
POSITIVO:
    MOV BX,1      ;coloca 1 em BX pois AX positivo
END_CASE:
...

```

Observação:

Observe que apenas um comando CMP é necessário, pois as instruções de salto (JXXX) não alteram os FLAGS.

Estrutura de Laço (LOOP)

O laço em pseudocódigo pode ser escrito como:

```

FOR n vezes DO
    Conjunto de instruções
END_FOR

```

Para a tradução para a linguagem de montagem, vamos introduzir a instrução **LOOP**, cuja sintaxe é:

```

LOOP rótulo_de_destino

```

Esta instrução tem como contador implícito o registrador CX, que deve ser inicializado antes do laço. Dessa forma, a instrução fará o salto para rótulo_de_destino, enquanto CX não for zero. Quando CX = 0, a próxima instrução após LOOP será executada.

Observação:

O registrador CX é decrementado automaticamente quando a instrução LOOP é executada.

Por exemplo, vamos escrever um trecho de código em linguagem de montagem para mostrar em uma linha com 80 asteriscos ("*").

Em algoritmo, utilizando pseudocódigo, ficaria:

```
FOR I=1 TO 80 DO
    DISPLAY '*'
END_FOR
```

ou

```
FOR 80 TIMES DO
    DISPLAY '*'
END_FOR
```

A tradução para a linguagem de montagem é:

```
    MOV CX,80          ;número de asteriscos
    MOV AH,2           ;prepara para mostrar um caractere
    MOV DL,'*'         ;caractere a ser mostrado
TOPO:
    INT 21h            ;mostra um asterisco
    LOOP TOPO          ;repete 80 vezes (vide CX)
...

```

Eventualmente, o valor de CX pode sofrer alguma alteração. Caso o valor de CX mude para zero e após isso o laço seja executado, após o primeiro comando o valor de CX será igual a FFFFh = 65535, fazendo com que o laço seja executado mais essa quantidade de vezes.

Para prevenir tal fato, deve-se utilizar a instrução **JCXZ** (*jump if CX is zero*) antes do início do laço. Esta instrução testa o valor de CX e se for zero, salta para o rótulo_destino.

```
JCXZ SKIP
TOPO:
    ;corpo do laço
    LOOP TOPO
SKIP:
...
```

Estrutura de Laço WHILE

O comando WHILE pode ser representado, em pseudocódigo, como:

```
WHILE condição DO
    Conjunto de instruções
END_WHILE
```

A condição é verificada no início do laço. Se verdadeira, o conjunto de instruções é executado. Se falsa, o programa vai para o que tiver abaixo do conjunto de instruções. O laço é executado enquanto a condição for verdadeira.

A figura 16.4 nos mostra o fluxo do comando WHILE

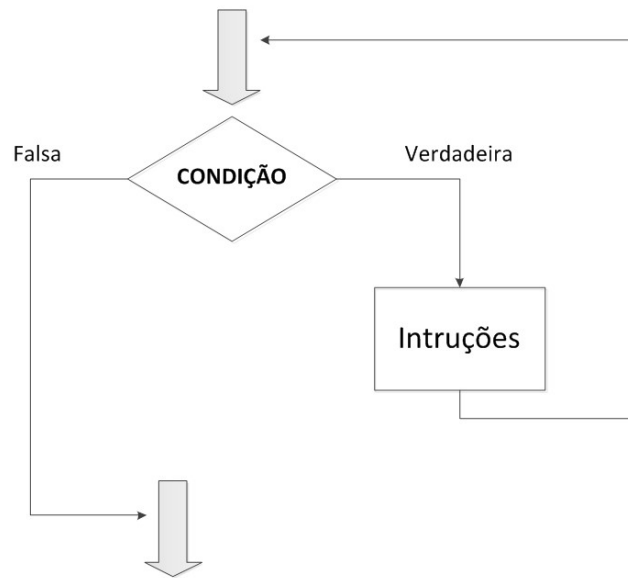


Figura 16.4 - Comando WHILE

Por exemplo, vamos escrever um código em linguagem de montagem para contar o número de caracteres digitados em uma linha, sendo que a condição de para da é a entrada da tecla <enter> (CR - carriage return)

O algoritmo deste programa pode ser representado assim:

```
Inicialize o contador com 0
Leia um caractere
WHILE (caractere <> CR) DO
    Armazenar caractere lido
    Contador = contador + 1
    Leia um caractere
END_WHILE
```

Em linguagem de montagem o programa ficaria:

```

...
        MOV DX,0h                ;inicialização
        MOV AH,1h
        INT 21h
;while
LOOP_:   CMP AL,0Dh              ;é o caractere CR?
        JE    FIM                ;salto quando caractere é
igual a CR
        MOV AL,(algun lugar)    ;salvando o caractere lido
        INC DX                    ;conta número de caracteres
        INT 21h                  ;lê outro caractere
        JMP LOOP_                ;fecha o loop WHILE
;end_while
FIM:

```

Estrutura de Laço REPEAT

O comando REPEAT pode ser escrito em pseudocódigo como:

```

REPEAT
Conjunto de instruções
UNTIL condição

```

A condição é verificada no final do laço. Se falsa, o conjunto de instruções é executado novamente. Se verdadeira, o programa continua executando os comandos abaixo da verificação da instrução. O laço é executado enquanto a condição for falsa.

A figura 16.5 nos mostra o fluxo do comando REPEAT

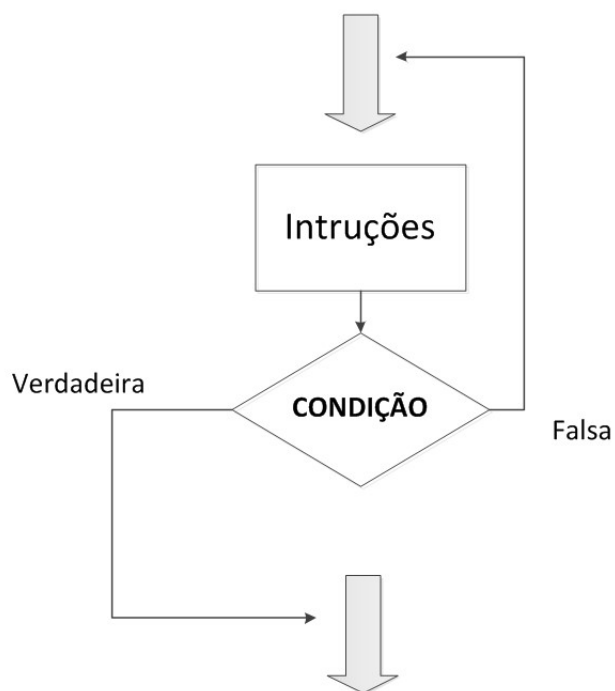


Figura 16.5 - Comando REPEAT

Para exemplificar a tradução do comando REPEAT, vamos escrever um código em linguagem de montagem para contar o número de caracteres digitados em uma linha, até que a tecla <enter> seja digitada (CR - *carriage return*)

Em pseudocódigo fica:

```
Inicialize o contador com 0
REPEAT
    Leia um caractere
    Armazenar caractere lido
    Contador = contador + 1
UNTIL (caractere = CR)
```

Em linguagem de montagem:

```

        MOV DX,0h                ;inicialização
        MOV AH,1h
;repeat
LOOP_:
        INT 21h
        MOV AL,(algun lugar)    ;salvando o caractere lido
        INC DX                  ;conta número de caracteres
        CMP AL,0Dh              ;é o caractere CR?
        JNE LOOP_               ;fecha o loop REPEAT
;until
...

```

Estrutura WHILE x Estrutura REPEAT

Em muitos casos onde uma estrutura de repetição é necessária, a utilização de WHILE ou REPEAT é apenas uma questão de preferência pessoal.

A vantagem da estrutura WHILE é que o laço pode ser transposto se a condição de terminação for inicialmente falsa. Em contrapartida, caso isso ocorra na estrutura REPEAT, o conjunto de instruções será executada ao menos uma vez.

Em contraposição, a estrutura REPEAT é geralmente menor que a estrutura WHILE.

Estruturas com condições compostas

- **Utilizando o operador AND**

É uma estrutura onde um determinado conjunto de instruções será executado, se e somente se, duas condições forem verdadeiras.

Condição1 AND condição2

Por exemplo, se quisermos escrever um caractere, somente se pertencer ao alfabeto e se estiver em CAIXA-ALTA (maiúscula).

O algoritmo ficaria:

```
Leia um caractere (para AL)
IF ("A" <= caractere) AND (caractere <= "Z")
    THEN
        Mostre-o
    END_IF
```

O trecho de programa em linguagem de montagem:

```
;ler um caractere
    MOV AH,1          ;prepara para leitura
    INT 21h           ;lê e coloca em AL
;se ('A' <= char) E (char <= 'Z')
    CMP AL,'A'        ;char >= 'A'
    JNGE END_IF       ;não, saia!
    CMP AL,'Z'        ;char <= 'Z'
    JNLE END_IF       ;não, saia!
;then mostrar o caractere
    MOV DL,AL         ;pegue o caractere
    MOV AH,2          ;prepare para mostrar um caractere
    INT 21h           ;mostra o caractere
END_IF:
...
```

- **Utilizando o operador OR**

É uma estrutura onde um determinado conjunto de instruções será executado, se pelo menos uma das duas condições forem verdadeiras.

Condição1 OR condição2

Por exemplo, vamos escrever um programa que leia um caractere. Se ele for igual a "y" ou "Y", mostre-o. Do contrário, saia do programa.

Em algoritmo ficaria:

```
Leia um caractere (para AL)
IF (caractere = "y") OR (caractere = "Y")
  THEN
    Mostre-o
  ELSE
    Termine o programa
END_IF
```

A tradução para a linguagem de montagem seria:

```
;ler um caractere
    MOV AH,1                ;prepara para leitura
    INT 21h                 ;lê e coloca em AL
;se (char = 'y') OU (char = 'Y')
    CMP AL,'y'              ;char = 'y'
    JE THEN                 ;sim, mostre-o!
    CMP AL,'Y'              ;char = 'Y'
    JE THEN                 ;sim, mostre-o!
    JMP ELSE_               ;não, termine o programa
THEN:
    MOV DL,AL               ;pegue o caractere
    MOV AH,2                ;prepare para mostrar um caractere
    INT 21h                 ;mostra o caractere
    JMP END_IF              ;e saia do IF
ELSE_:
    MOV AH,4Ch              ;sair para o DOS
    INT 21h
END_IF:
...
```

CAPÍTULO 17 - PILHA, PROCEDIMENTOS, MACROS, VETORES, MATRIZES E MODOS DE ENDEREÇAMENTO.

Objetivos do Capítulo

Após a leitura deste capítulo o leitor estará apto a:

- manipular a estrutura de pilha disponível na linguagem de montagem do processador 8086.*
- modular os programas de forma eficiente e elegante, através dos procedimentos.*
- definir estrutura de dados mais complexas, como os vetores e matrizes.*

FUNDAMENTOS

Como já vimos, a pilha é uma estrutura de dados unidimensional, onde as informações podem ser inseridas e retiradas utilizando a ordem do "último que entrou é o primeiro a ser retirado - *last-in first-out*". O topo da pilha é a localização do último elemento inserido nela. A linguagem de montagem do processador 8086 propicia a utilização de uma pilha de dados de maneira fácil, através de instruções específicas de armazenamento e retirada de dados desta pilha. Esta pilha, durante a execução do programa, é utilizada para o armazenamento temporário de dados e endereço, principalmente no chama e retorno de um procedimento.

Os procedimentos (sub-rotinas) são trechos de programas que executam uma determinada função e são definidos fora do programa principal. A chamada para que este procedimento seja executado é no programa principal, em outro procedimento ou no próprio procedimento (chamadas recursivas). A programação utilizando procedimentos é de extrema importância, não só em programação em alto nível como também em programação linguagem de montagem, por permitir a modulação de programas. A modulação por sua vez faz com que tenhamos uma programação

elegante, um código de menor tamanho e ainda facilita a verificação e correção de erros.

17.1 Organização da pilha

Antes de estudarmos as instruções de manipulação de pilha, vamos ver como podemos utilizá-la com o montador TASM. Para isto utilizaremos a diretiva **.STACK** que permite que um bloco de memória seja alocado para a pilha. A declaração que o programa deve conter é:

```
.STACK 100h ;aloca um bloco de memória para a pilha
```

Esta declaração faz com que o SS (*stack segment*), tenha, após a montagem do programa, o endereço base do segmento de pilha. O registrador SP (*stack pointer*) será usado pelas instruções de manipulação de pilha, para indicar o offset deste segmento, onde será o topo da pilha. Portanto o endereço lógico do topo da pilha será dado pelo par SS:SP. Importante salientar que a pilha "cresce" de cima para baixo. Nos exemplos de utilização das instruções isto ficará mais claro.

17.2 Instruções de manipulação da pilha

Temos dois tipos de instruções específicas de manipulação da pilha. Uma que permite que um item seja armazenado na pilha, representado pelo mnemônico PUSH e outra que permite que um item seja retirado da pilha, representado pelo mnemônico POP.

Instruções de armazenamento na pilha

As instruções de armazenamento de um item na pilha são as instruções **PUSH** e **PUSHF**.

As instruções do tipo **PUSH** tem a seguinte sintaxe:

PUSH fonte

Nesta instrução, fonte é um operando de 16 bits, podendo apenas ser um registrador ou uma palavra de memória do tipo DW.

A execução do PUSH apresenta as seguintes ações:

- O registrador SP é decrementado de 2, uma vez que a pilha armazena palavras de 16 bits, ou seja, 2 bytes.
- Uma cópia do conteúdo do operando fonte é armazenado na pilha de forma que a posição SS:SP, armazene o byte menos significativo do operando fonte e a posição SS:(SP + 1) armazena o byte mais significativo.
- O conteúdo da fonte não é alterado e não altera nenhum dos FLAGS.

Outra instrução para armazenado de uma informação na pilha é a instrução **PUSHF**, que tem a seguinte sintaxe:

PUSHF

A instrução PUSHF que não possui operando explícito. Ela é usada para armazenar, no topo da pilha, o conteúdo do registrador de FLAGS. Sua operação é análoga à da instrução PUSH, com a exceção que aqui o que vai ser armazenado na pilha é o registrador FLAG.

Por exemplo, suponha que AX contenha o valor 1234h e que o registrador de FLAG tenha armazenado 5678h. Quais valores serão armazenados na pilha após a execução de cada uma das instruções abaixo?

PUSH AX	; armazena AX no topo da pilha
PUSHF	; armazena o FLAG no topo da pilha

A Figura 17.1 nos mostra a solução. O que podemos observar é que inicialmente a pilha está vazia e SP aponta para o offset 0100h. Após a execução da primeira instrução, temos o valor 1234h armazenado e o valor de SP atualizado para o novo topo. Como vimos no capítulo 12, o SP tem que ser atualizado ($SP = SP - 2$) antes de o dado ser armazenado. Como a organização da memória é em bytes, a palavra de 16 bits é armazenada da seguinte maneira: o byte menos significativo na posição de maior endereço e o mais significativo na posição de menor endereço. Após a execução da segunda instrução, temos o conteúdo do FLAG armazenado no novo topo da pilha.

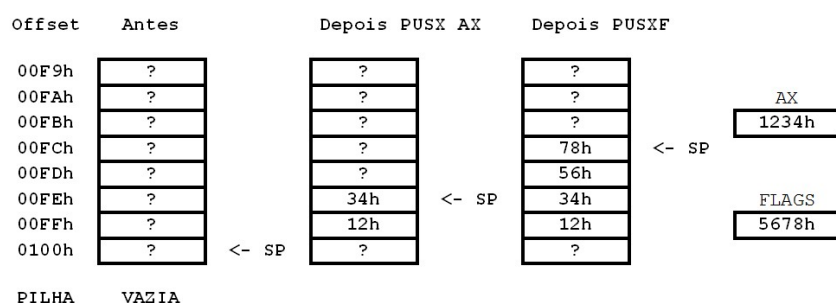


Figura 17.1 - Passos da execução de instruções PUSH e PUSHF.

Instruções de retirada na pilha

As instruções de retirada de um item da pilha são as instruções **POP** e **POPF**.

As instruções do tipo **POP** tem a seguinte sintaxe:

POP destino

Análogo à instrução PUSH, nesta instrução, destino é um operando de 16 bits, podendo apenas ser um registrador ou uma palavra de memória do tipo DW.

A execução de POP apresenta as seguintes ações:

- O conteúdo das posições SS:SP (byte menos significativo) e SS:(SP + 1) (byte mais significativo) é movido para o destino.

- O registrador SP (*stack pointer*) é incrementado de 2, determinado o novo topo da pilha.
- Nenhum dos bits de FLAG é alterado.

Outra instrução para retirada de um item da pilha é a instrução **POPF**, que tem a seguinte sintaxe:

POPF

A instrução POPF, como a instrução PUSHF, não possui operando explícito. Ela é usada para retirar do topo da pilha a palavra armazenada e colocá-la no registrador de FLAGS. Sua operação é análoga à da instrução POP, com a exceção que aqui o destino será o registrador FLAG.

Por exemplo, suponha que a pilha tenha armazenado duas palavras 1234H e 5678h, sendo está última, no seu topo. Suponha ainda que tenha em AX o número F0D3h e no FLAG o número 006Ah. Quais valores seriam armazenados nos registradores após a execução de cada uma das instruções abaixo e como ficaria a pilha?

POPF	; retira do topo da pilha e armazena no FLAG
POP AX	; retira do topo da pilha e armazena no AX

A figura 17.2 nos mostra o que acontece. O inicialmente o SP aponta para o offset 00FCh, que é o topo da pilha e tem armazenado o número 5678h. Após a execução da primeira instrução, temos este valor é armazenado no FLAG e o valor de SP atualizado para o novo topo ($SP = SP + 2$). Após a execução da segunda instrução, temos o conteúdo do novo topo armazenado em AX e SP apontando para a última posição da pilha, que é vazia. Portanto, apesar dos valores estarem na memória, até que algo seja escrito em cima, a pilha é considerada vazia, uma vez que não temos como desempilhá-los.

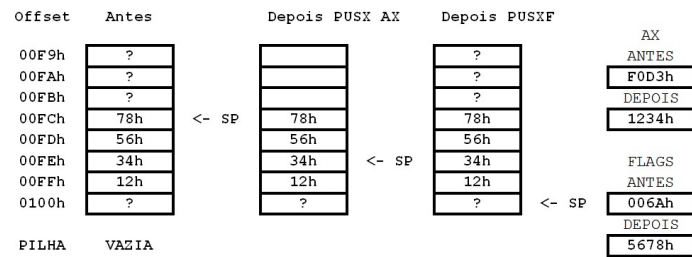


Figura 17.2- Passos da execução de instruções POP e POPF.

Este outro exemplo é de um programa que utiliza a pilha para escrever uma string de forma invertida da sua leitura.

```

TITLE ENTRADA INVERTIDA
.MODEL SMALL
.STACK 100h
.CODE

        MOV AH,2          ; exibe o prompt para o usuário
        MOV DL,'?'        ; caracter '?' para a tela
        INT 21h           ; exibe
        XOR CX,CX          ; inicializando contador de
                           ; caracteres em zero

        MOV AH,1          ;prepara para ler um caracter do
                           ;teclado

        INT 21h           ;caracter em AL

; while caracter não é <CR> do

INICIO:  CMP AL,0DH        ; e' o caracter <CR>?
        JE PT1            ; sim, _ntão saindo do loop

; salvando o caracter na pilha e incrementando o contador

        PUSH AX           ; AX vai para a pilha (interessa

```

```

                                ; somente AL)
                                INC CX          ; contador = contador + 1
                                               ; lendo um novo caracter
                                INT 21h         ; novo caracter em AL
                                JMP INICIO      ; retorna para o inicio do loop
; end_while

PT1:      MOV AH,2              ; prepara para exibir
          MOV DL,0DH           ; <CR>
          INT 21h              ; exibindo
          MOV DL,0AH           ; <LF>
          INT 21h              ; exibindo: mudança de linha
          JCXZ FIM              ; saindo se nenhum caracter foi
                                ; digitado
; for contador vezes do

TOPO:     POP DX                ; retira o primeiro caracter
                                ; da pilha
          INT 21h              ; exibindo este caracter
          LOOP TOPO            ; em loop até CX = 0
; end_for

FIM:      MOV AH,4CH            ;preparando para sair para o DOS
          INT 21H
          END

```

17.3 Definição e Utilização de Procedimentos

Um procedimento é definido pelas diretivas **PROC** e **ENDP**. A diretiva **PROC** serve para definir o nome do procedimento e o seu início, enquanto que a **ENDP** para definir o fim do procedimento.

A definição de um procedimento segue a seguinte sintaxe:


```
nome PROC tipo

; instruções do procedimento

RET          ; instrução de retorno, que transfere o controle de
              ; volta para a rotina principal
nome ENDP
```

O **nome** do procedimento é qualquer símbolo válido que não seja palavra reservada. O **tipo** define se o procedimento está definido no mesmo segmento de código do local de chamada, tipo **FAR**, ou em segmento distintos, tipo **NEAR**.

17.4 Instruções de chamada e retorno de procedimentos

Instrução de chamada de procedimento

Para que um procedimento seja executado, ele tem que ser invocado por uma instrução. A instrução **CALL** faz com que o controle de execução do programa passe para o procedimento.

A sintaxe desta instrução é

```
CALL nome
```

Onde **nome**, como já vimos, é qualquer símbolo válido que não seja palavra reservada, que foi definido pela diretiva **PROC**. Quando da execução desta instrução, o registrador IP, que contem o *offset* do endereço da próxima instrução à chamada, ou seja, da instrução depois do **CALL**, é armazenado na pilha. Depois o IP recebe o *offset* do endereço da primeira instrução do procedimento chamado para que a rotina seja executada. Esta instrução não afeta nenhum dos **FLAGS**.

Instrução de retorno de procedimentos

A última instrução de um procedimento deve ser a instrução de retorno de procedimento, pois o controle de execução do programa deve retornar a quem chamou o procedimento. A instrução **RET** faz com que este retorno seja feito. A sintaxe desta instrução é:

RET

O retorno para quem chamou significa que o IP deverá ter o endereço da instrução depois do CALL. Vamos lembrar que o *offset* é empilhado pela instrução CALL. Portanto o RET deverá desempilhar o *offset* e carregá-lo em IP. Esta instrução não afeta nenhum dos FLAGS.

A Figura 17.3 mostra o mecanismo de chamada e a Figura 17.4 mostra o mecanismo de retorno de um procedimento. Podemos notar que durante a execução do CALL, a pilha está vazia e o IP aponta para a próxima instrução (*offset* 1012h) depois da instrução CALL. Após a execução do CALL, o endereço de retorno (*offset* 1200h) é empilhado no topo da pilha e IP aponta para a primeira instrução do procedimento (*offset* 1200h). Isto fará com que a próxima instrução a ser executada seja a primeira instrução do procedimento.

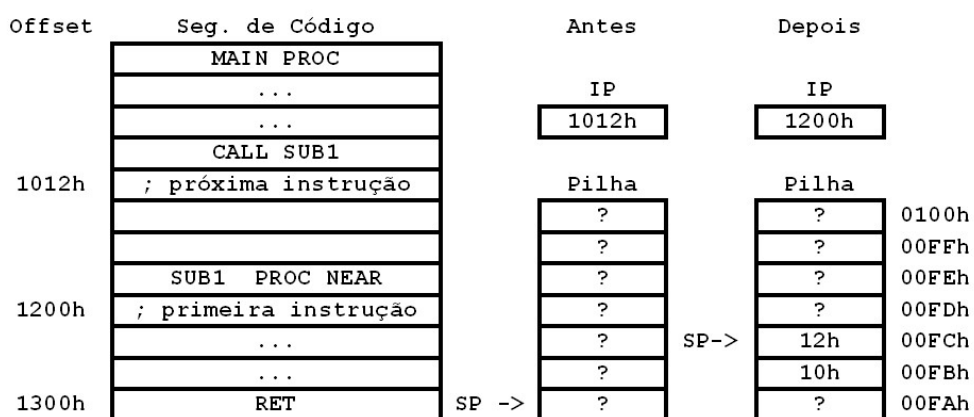


Figura 17.3 - Mecanismo de chamada de um procedimento

A Figura 17.4 nos mostra o mecanismo de retorno de um procedimento. Antes da execução da instrução RET, temos no registrador IP o *offset* desta instrução e no topo da pilha o endereço de retorno (1012h - *offset* da primeira instrução depois da instrução CALL). Após a execução do RET, o conteúdo do topo da pilha é carregado em IP (*offset* 1012h), devolvendo o controle de execução "para quem chamou". A pilha fica vazia.

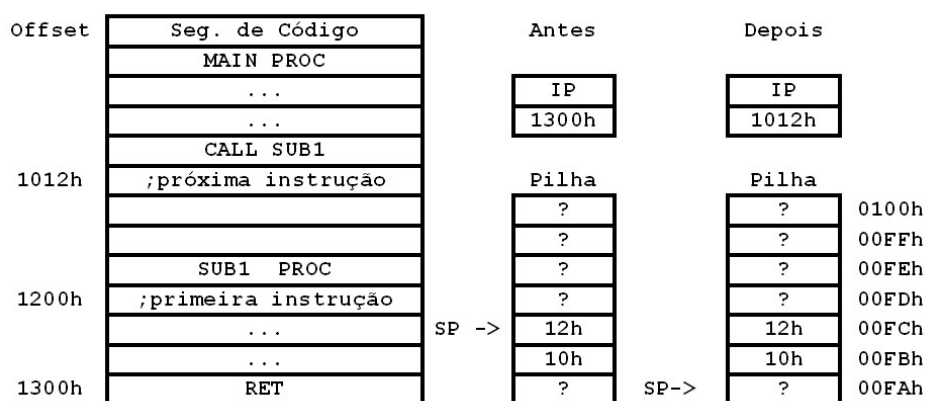


Figura 17.4 - Mecanismo de retorno de um procedimento

17.5 Passagem de Parâmetros em Procedimentos

Como fazer a passagem de parâmetros, isto é, como podemos fornecer os dados necessários para a execução do procedimento e enviar os dados calculados por este procedimento, para "quem" o chamou? Na linguagem de montagem isto é feito utilizando os registradores de propósito geral ou até mesmo pela pilha.

O trecho de programa abaixo exemplifica a utilização de um procedimento com o programa principal. Trata-se de um procedimento que calcula a multiplicação de dois números, utilizando operações de soma e deslocamento.

```

TITLE    MULTIPLICACAO POR SOMA E DESLOCAMENTO

.MODEL   SMALL
.STACK   100h
.CODE

PRINCIPAL    PROC

    ...                ; supondo a entrada de dados
    CALL      MULTIPLICA
    ...                ; supondo a exibição do resultado
    MOV AH,4Ch        ; retorno ao Sistema Operacional
    INT 21h

PRINCIPAL    ENDP

MULTIPLICA    PROC      NEAR

;multiplica dois números A e B por soma e deslocamento
;entradas:      AX = A, BX = B, números na faixa 00h - FFh
;saída:         DX = A*B (produto)

    PUSH AX
    PUSH BX          ; salva os conteúdos de AX e BX
    AND DX,0         ; inicializa DX em 0

; repeat
; if   B e' impar
TOPO:    TEST BX,1    ; B e' impar?
        JZ PT1       ; não, B e' par (LSB = 0)
        ;then
        ADD DX,AX     ; sim, então produto = produto + A
        ;end_if
PT1:     SHL AX,1      ; desloca A para a esquerda 1 bit
        SHR BX,1      ; desloca B para a direita 1 bit

;until

```

```
        JNZ TOPO          ; fecha o loop repeat
        POP BX
        POP AX            ; restaura os conteudos de BX e AX
        RET               ; retorno para o ponto de chamada
MULTIPLICA      ENDP
END  PRINCIPAL
```

17.6 Macros

Outra técnica utilizada na modulação de programas são as MACROS. Como os procedimentos, as macros também são trechos de programa que realizam uma determinada função. Mais especificamente, em linguagem de montagem, podemos definir uma macro como um bloco de texto que recebe um nome especial e que contém instruções, diretivas, comentários ou referências à outras macros.

A macro é chamada no momento da montagem e é expandida, ou seja, o montador copia o bloco texto na posição de cada chamada de macro. As expansões podem ser vistas no arquivo .LST, gerada pelo TASM.

As macros são utilizadas para “criar novas instruções” que operacionalizem tarefas freqüentes e repetitivas. A tabela 17.1 mostra a comparação do uso macro com o uso do procedimento. Utilizando macros, o tempo de montagem é maior, pois a cada chamada de macro, o montador substitui pelas respectivas instruções enquanto que com a chamada de procedimentos isto não é feito. A consequência disto é que o código gerado com macros é maior que o com procedimentos, mas em compensação, com procedimentos teremos mais acessos na pilha (memória) tornando a execução é mais lenta.

	Macros	Procedimentos
Tempo para montagem do programa	Maior	Menor
Tamanho de código de máquina (.EXE)*	Maior	Menor
Tempo de execução menor	Menor	Maior
Utilização	Pequenas funções	Grandes funções

Observação: * Na maioria das vezes para funções análogas.

Tabela 17.1 - Comparação Macros e Procedimentos

A sintaxe da definição de uma macro é:

```
nome_da_macro  MACRO      p1,p2,p3,...pn
instruções
ENDM
```

Onde p1,p2,p3..pn são argumentos da macro e são opcionais.

Por exemplo, suponha a criação de uma macro que mova uma palavra de uma posição de memória para outra. Já sabemos que não temos uma instrução que faça isto.

A macro seria MOVW definida como:

```
MOVW MACRO      WORD1, WORD2
    PUSH        WORD1
    POP         WORD2
ENDM
```

A chamada desta macro seria:

```
MOVW      PALAVRA1, PALAVRA2
```

Onde PALAVRA1, PALAVRA2 são offset no segmento de dados do tipo DW.

Podemos observar que poderíamos também utilizar esta macro para mover uma palavra de um registrador de 16 bits para outro.

Por exemplo:

```
MOVW    AX, BX
```

O que acontece é que WORD1 e WORD2 são substituídos pelos argumentos da chamada. No primeiro caso, duas posições de memória e no segundo dois registradores. Em ambas chamadas os argumentos precisam ser de 16 bits por causa das instruções da macro, PUSH e POP.

Diretiva INCLUDE.

Para incluir outro arquivo no arquivo do programa que será montado, usamos a diretiva:

```
INCLUDE caminho\nome_do_arquivo_texto.
```

17.7 Vetores e matrizes (arrays)

Vetores unidimensionais

Podemos definir um vetor unidimensional como uma lista ordenada de elementos do mesmo tipo. Por ordem entendemos que existe um primeiro, um segundo, um terceiro elemento, e assim por diante até um último elemento. Por exemplo, já nos deparamos nas aulas de matemática com um vetor de n elementos, onde os elementos são identificados por:

$$A[1], A[2], A[3], A[4], A[5].. A[n];$$

Onde A é o nome do vetor e os números dentro dos colchetes são os índices. Portanto se quisermos acessar o segundo elemento do vetor A , é só utilizarmos $A[2]$. Na linguagem de alto nível C, em um vetor de n posições, o primeiro elemento teria o índice 0 e o último o índice **($n-1$)**.

Na linguagem linguagem de montagem, os vetores têm que ser definidos na área de dados. A seguir mostraremos as diversas definições deste tipo de estrutura:

- **Definição de uma cadeia de caracteres (string).**

```
MSG      DB      'abcde'      ; vetor composto por um string de 5
                                ;caracteres ASCII
```

Onde MSG é o offset dentro do segmento de dados (supondo ser 00FFh), e o armazenamento desta *string* na memória ficaria:

	Offset de endereço	Endereço simbólico	Conteúdo
MSG ->	00FFh	MSG	10h
	0100h	MSG+1	10h
	0101h	MSG+2	20h
	0102h	MSG+3	10h
	0103h	MSG+4	30h

Figura 17.5 - Conteúdo de memória referente à *string* MSG

- **Definição de um vetor de elementos do tipo *word*.**

```
W DW      1010h, 1020h, 1030h ; vetor de 3 valores de 16 bits
```

Onde W é o *offset* (supondo 0200h) dentro do segmento de dados e o armazenamento deste vetor na memória ficaria:

	Offset de endereço	Endereço simbólico	Conteúdo
W ->	0200h	W	10h
	0201h		10h
	0202h	W+2	20h
	0203h		10h
	0204h	W+4	30h
	0205h		10h

Figura 17.6 - Conteúdo de memória referente ao vetor W

Operador DUP

O operador **DUP** pode ser utilizado na definição de vetores, quando queremos repetir algumas sub-estruturas. Os exemplos abaixo mostram a correta utilização deste operador.

```

GAMA      DB    100 DUP (0)      ; cria um vetor de 100 bytes, cada byte
                                   ; inicializado com o valor zero, a
                                   ; partir do offset GAMA

BETA      DW    200 DUP (?)      ; cria um vetor de 200 words (16 bits),
                                   ; não inicializados, a partir do offset
                                   ;BETA

; operadores DUP encadeados
LINHA     DB    5, 4, 3 DUP (2, 3 DUP (0), 1)
; equivalente a definição de LINHA dada abaixo:
LINHA     DB    5, 4, 2, 0, 0, 0, 1, 2, 0, 0, 0, 1, 2, 0, 0, 0, 1

```

- **Matrizes ou vetores bidimensionais**

Uma matriz ou um vetor bidimensional é uma estrutura que, logicamente, é organizado em linhas e colunas, apesar de fisicamente ser considerado como um vetor unidimensional.

Para uma determinada matriz **A**, o acesso aos elementos organizados em linha e coluna é dado por: **A[i,j]** onde **A** é o nome da matriz, **i** representa a linha e **j** a coluna do elemento a ser acessado.

Por exemplo, suponha uma matriz 3 X 4, a Figura 17.7 mostra como seria:

MATRIZ	A[1,1]	A[3,2]	A[1,3]	A[1,A]
A	A[2,1]	A[2,2]	A[2,3]	A[2,4]
	A[3,1]	A[3,2]	A[3,3]	A[3,4]

Figura 17.7 - Matriz A - representação.

A definição das matrizes em linguagem de montagem é feita no segmento de dados, conforme mostraremos no exemplo a seguir.

Suponha a matriz A (3 x 4) abaixo, inicializado com valores mostrados na Figura 17.8.

	1	2	3	4
1	10	20	30	40
2	50	60	70	80
3	90	100	110	120

Figura 17.8 - Matriz A

A definição no linguagem de montagem do 8086, se a organização for por linha é:

```

...
.DATA
A      DW      10,20,30,40
        DW      50,60,70,80
        DW      90,100,110,120
...

```

Se a organização for por coluna:

```

...
.DATA
      A      DW      10,50,90
        DW      20,60,100
        DW      30,70,110
        DW      40,80,120
...

```

17.8 Modos de endereçamento

A forma em que um operando é especificado numa instrução é conhecido como Modo de Endereçamento, entre outras palavras, mostra o que se deve fazer para acessá-lo. Temos 9 modos de endereçamento, que estudaremos a seguir:

Modo de Endereçamento por Registrador.

Neste modo, os operandos são um dos registradores da UCP. Por exemplo:

```
ADD AX,BX
```

Neste caso, os dados estão armazenados nos registradores AX e BX. Para acessá-los, basta o processador ler os dois registradores.

Modo de Endereçamento utilizando um Imediato.

Aqui o operando é uma constante que está explicitamente, definida na instrução, ou seja, disponibilizada imediatamente. Por exemplo:

```
MOV AX,5
```

O valor 5 é definido na própria instrução. Uma vez a feito o *fetch* da instrução, este valor estará disponível para a execução da instrução.

Modo de Endereçamento Direto

Neste modo, um dos operandos é uma variável declarada, ou seja uma posição de memória com endereço determinado.

```
DADO    DB      5
.....
MOV AL, DADO
```

A variável DADO (*offset* dentro do segmento de dados) é inicializada como 5. Quando a instrução MOV é executada, o conteúdo do endereço físico dado por DS:DADO é armazenado em AL, ou seja, o valor 5 é armazenado em AL.

Modo de Endereçamento Indireto por Registrador.

Neste modo de endereçamento, o *offset* do operando está em um registrador e, portanto, este registrador **atua** como ponteiro para a posição de memória

Os únicos registradores que podem ser utilizados para armazenar o *offset* neste modo de endereçamento são:

- BX, SI, DI, juntamente com o registrador de segmento DS, formando o endereço físico com o par DS:[registrador]. Lembre-se que quando utilizamos os colchetes, estamos querendo dizer que o conteúdo do operando dentro dos colchetes é um apontador (*offset*) para a memória.
- BP, juntamente com o registrador de segmento SS, cujo endereço físico é formado por SS:[BP]

Os exemplos a seguir nos mostram como utilizar este modo de endereçamento:

Exemplo 1

Vamos supor que o conteúdo de SI seja 0100h e que a palavra contida na posição de memória de *offset* 0100h seja 1234h. Os comentários mostram o que acontece com o registrador SI após a execução das respectivas instruções

```
MOV AX,SI           ;AX recebe 0100h
```

Neste caso, temos o endereçamento por registrador e, portanto, o conteúdo de SI será armazenado em AX.

```
MOV AX,[SI]         ;AX recebe 1234h
```

Aqui, como temos os colchetes, o modo é o indireto por registrador, ou seja, o conteúdo de SI é o *offset* do dado a ser armazenado em AX.

Exemplo 2

Escreva um trecho de programa que acumule em AX a soma dos 10 elementos do vetor LISTA, pré-definido.

```
LISTA      DW      10,20,30,40,50,60,70,80,90,100
            ...
            XOR AX,AX      ;inicializa AX com zero
            LEA SI,LISTA   ;SI recebe o offset de end. de LISTA
            MOV CX, 10     ;contador inicializado no. de elementos
SOMA:      ADD AX,[SI]     ;acumula AX com o elemento de LISTA
            ;apontado por SI
            ADD SI,2       ;movimenta o ponteiro para o próximo
            ;elemento de LISTA (que é do tipo DW)
            LOOP SOMA      ;faz o laço até CX = 0
```

Modo de Endereçamento por Base

No modo de endereçamento por base, o *offset* do operando é obtido adicionando um deslocamento ao conteúdo de um registrador base, BX ou BP.

O deslocamento pode ser um *offset* definido por uma variável, uma constante (positiva ou negativa), ou um *offset* definido por uma variável mais ou menos uma constante. Abaixo temos os formatos possíveis:

- [registrador + deslocamento] ou [deslocamento + registrador]
 - [registrador] + deslocamento ou deslocamento + [registrador]
 - deslocamento [registrador]
-

Os registradores que podem ser utilizados neste modo de endereçamento são:

- BX (*base register*) juntamente com o registrador de segmento DS.
- BP (*base pointer*) juntamente com o registrador de segmento SS.

Abaixo exemplos de utilização deste modo de endereçamento:

Exemplo1

Supondo que BX tenha o valor 4d e que o vetor lista é definido como:

LISTA	DW	10h,20h,30h,40h,50h,60h,70h,80h,90h,100h
--------------	-----------	---

A execução da instrução:

MOV AX, [LISTA + BX]	;resulta que AX = 30h
-----------------------------	------------------------------

Ela faz com que o conteúdo do endereço físico dado por DS e (LISTA + 4) seja armazenado em AX, ou seja, o número 30h. Note que ao somarmos 4 à LISTA, estamos acessando o quinto byte e o sexto byte (word), visto que a memória é organizada em bytes, conforme mostra a Figura 17.9.

	Endereço simbólico	Conteúdo
LISTA ->	LISTA+0	10h
		00h
	LISTA+2	20h
		00h
	LISTA+4	30h
		00h
	LISTA+6	40h
		00h
	LISTA+8	50h
		00h
	LISTA+10	60h
		00h
	LISTA+12	70h
		00h
	LISTA+14	80h
		00h
	LISTA+16	90h
		00h
	LISTA+18	00h
		01h

Figura 17.9 - Vetor LISTA na memória

Exemplo 2

Escreva um trecho de programa que acumule em AX a soma dos 10 elementos do vetor LISTA definido previamente, usando endereçamento por Base.

```

LISTA      DW      10h,20h,30h,40h,50h,60h,70h,80h,90h,100h
            ...
            XOR AX,AX          ; inicializa AX com zero
            XOR BX,BX          ; limpa o registrador base
            MOV CX, 10          ; contador inicializado no. de
                                ; elementos
SOMA:      ADD AX,LISTA+[BX]    ; acumula AX com o elemento de LISTA
                                ; apontado por offset de LISTA + BX
            ADD BX,2            ; incrementa base
            LOOP SOMA           ; faz o laço até CX = 0
            ...

```


Exemplo 3

O acesso á pilha sem alteremos o valor do SP, ou seja sem empilhar ou desempilhar qualquer elemento, pode ser feito utilizando o modo de endereçamento por base e o registrador BP. Vamos escrever um trecho de programa que carregue AX, BX e CX com as três palavras mais superiores da pilha, sem modificá-la.

```
...  
MOV BP,SP          ;BP aponta para o topo da pilha  
MOV AX, [BP]       ;coloca o conteúdo do topo em AX  
MOV BX, [BP+2]     ;coloca a 2a. palavra (abaixo do topo) em BX  
MOV CX, [BP+4]     ;coloca o 3a. palavra em CX  
...
```

Modo de Endereçamento Indexado

Analogamente ao endereçamento por base, neste caso o *offset* do do operando é obtido adicionando um deslocamento ao conteúdo de um registrador indexador, SI ou DI

O deslocamento pode ser o *offset* definido por uma variável, uma constante (positiva ou negativa), ou o *offset* definido por uma variável mais ou menos uma constante. Abaixo temos os formatos possíveis:

- [registrador + deslocamento] ou [deslocamento + registrador]
- [registrador] + deslocamento ou deslocamento + [registrador]
- deslocamento [registrador]

Os registradores utilizados só podem ser o SI e o DI, juntamente com o registrador de segmento DS

Segue alguns exemplos de utilização deste modo de endereçamento:

Exemplo 1

Supondo que SI contenha o *offset* de LISTA, definido abaixo:

LISTA	DW	10,20,30,40,50,60,70,80,90,100
-------	----	--------------------------------

Para acessar um elemento de LISTA, podemos:

LEA SI,LISTA	;SI recebe o <i>offset</i> de LISTA
MOV AX, [SI + 12]	;resulta que AX = 70h

Neste caso, na primeira instrução, SI recebe o *offset*, indicado por LISTA, dentro do segmento de dados (similar à instrução MOV SI, offset LISTA). A segunda instrução pegará a palavra armazenada em (LISTA+12) (0070h - Figura 17.3) e guardará em AX.

Exemplo 2

Fazer uma rotina que converta os caracteres da *string* definida em MSG, para maiúsculos.

MSG	DB	'isto e uma mensagem'
		...
	MOV CX,19	;inicializa no. de caracteres de MSG
	XOR SI,SI	;SI = 0
TOPO:	CMP MSG[SI], ' '	;compara caracter com branco
	JE PULA	;igual, não converte
	AND MSG[SI],0DFh	;diferente, converte para maiúscula
PULA:	INC SI	;incrementa indexador
	LOOP TOPO	;faz o laço até CX = 0
		...

Modo de Endereçamento por Base Indexado

O modo de endereçamento por Base Indexado (*Based Indexed Mode*) é onde o *offset* do operando é obtido somando-se o conteúdo de um registrador de base (BX ou BP) com o conteúdo de um registrador índice (SI ou DI) e opcionalmente com o *offset* representado por uma variável ou uma constante (positiva ou negativa). Portanto os possíveis formatos são:

- variável [reg_de_base] [reg_índice]
- variável [reg_de_base + reg_índice + constante]
- constante [reg_de_base + reg_índice + variável]
- [reg_de_base + reg_índice + variável + constante]

Neste modo, podemos apenas utilizar os registradores: SI, DI e BX juntamente com o registrador de segmento DS e SI, DI e o registrador BP juntamente com o registrador de segmento SS.

A seguir alguns exemplos de utilização:

Exemplo1

Supondo a variável LISTA definida abaixo e que SI tenha 14 e BX tenha 2:

LISTA	DW	10,20,30,40,50,60,70,80,90,100
-------	----	--------------------------------

MOV AX, LISTA [BX][SI]	;resulta que AX = LISTA+2+14 = 90
------------------------	-----------------------------------

A execução da instrução acima faz com que o conteúdo do segmento de dados apontado pelo offset LISTA + 2 + 14 seja armazenado em AX.

Exemplo 2

Suponha que A seja uma matriz 3X4 com elementos de tipo DW. Escreva um trecho de programa que zere os elementos da segunda linha de A.

```

...
MOV BX,8           ;BX aponta para o 1o. elemento da
                   ;segunda linha
MOV CX,4           ;CX contem o número de colunas
XOR SI,SI          ;inicializa o indexador de coluna
LIMPA: MOV A [BX][SI], 0 ;carrega zero no operando calculado
ADD SI,2           ;incrementa 2 em SI pois é word
LOOP LIMPA         ;faz o laço até que CX seja zero
...

```

Observações

a. Acesso de dados em outro segmento

Para acessar dados de um segmento em outro (*Segment Override*), basta que utilizemos a notação completa de endereço, especificando, o endereço base pelo registrador de segmento. Por exemplo:

```
MOV AX,ES:[SI]
```

Pode-se utilizar *segment override* nos modos de endereçamento indireto por registrador, por base e indexado.

b. Diretiva PTR

A instrução `MOV [BX],1` é ilegal, pois o montador não pode determinar, por si só, se `[BX]` aponta para uma informação na memória do tipo byte ou do tipo word, uma vez que o imediato pode ser representado com 8 ou 16 bits. Para resolver isto, utilizamos a diretiva `PTR`, da seguinte maneira:

```
MOV BYTE PTR [BX],1 ;define o destino como byte
MOV WORD PTR [BX],1 ;define o destino como word
```

d. Diretiva LABEL

LABEL é uma diretiva que serve para alterar o tipo de variáveis, por exemplo:

TEMPO	LABEL	WORD
HORAS	DB	10
MINUTOS	DB	20

Esta estrutura, declarada no segmento de dados, permite que:

- TEMPO e HORAS recebam o mesmo endereço pelo montador;
- TEMPO (16 bits) engloba HORAS e MINUTOS (8 + 8 bits);

Portanto, são legais as seguintes instruções:

```
MOV AH,HORAS
MOV AL,MINUTOS
MOV AX,TEMPO      ;produz o mesmo efeito das acima
```

e. A instrução XLAT

É uma instrução, sem operando, que converte um valor (tipo *byte*) em outro valor (tipo *byte*), utilizando uma tabela de conversão. O valor a ser convertido tem que estar em AL, o *offset* do endereço base da tabela de conversão tem que estar em BX. A execução da instrução XLAT fará com que o conteúdo de AL seja somado ao *offset* dado por BX, resultando em uma posição na tabela. O conteúdo desta posição será armazenado em AL.

Por exemplo, converter o conteúdo de AL, supondo um número binário, no seu correspondente hexadecimal, representado por um caracter.

```

.DATA
Tabela 16.      DB      30h,31h,32h,33h,34h,35h,36h,37h,38h,39h
                  DB      41h,42h,43h,44h,45h,46h

...

.CODE
...
MOV AL,0Ch      ; exemplo, converter 0Ch para caracter ASCII
                  ; 'C'

LEA BX,TABELA 16.  ; BX recebe o offset de TABELA 16.
XLAT             ; é feita a conversão e AL recebe 43h = 'C'
...

```

Exercícios

Parte 4

- 1 - Com relação à família INTEL fale um pouco da história, mostre a sequência de lançamento dos processadores, suas principais características e aplicações.
 - 2 - Defina segmentação de memória.
 - 3 - Quais os registradores que o 8086 possui e quais as suas funções?
 - 4 - Que endereço real representa o par 3EEAh:00FFh?
 - 5 - A partir da figura 13.1, explique o funcionamento do **processador** 8086 em um ciclo de execução de uma instrução.
 - 6 - O que é interrupção e exceção. Quais o 8086 suporta?
-

7 - Crie um programa em linguagem de montagem do 8086 para somar dois valores (A001h e 00C1h). A este resultado, deve ser subtraído o valor de 00FFh. Em seguida, coloque o resultado nos registradores BX e CX. Termine o programa, devolvendo o controle para o DOS. Depois disso, compile, ligue e execute o programa. Em seguida, faça novamente a compilação e a ligação com as opções para execução do programa com o Turbo Debugger. Execute o programa, passo-a-passo, e verifique se as operações são executadas.

8 - Quais nomes abaixo são ilegais para a linguagem Assembly no IBM-PC?

- a) TWO_WORDS
- b) ?1
- c) Two words
- d) .@?
- e) \$145
- f) LET'S_GO
- g) T = .

9 - Quais dos números abaixo são ilegais? Para os que são legais, diga se serão tratados como binário, decimal ou hexadecimal.

- a) 246
 - b) 246h
 - c) 1001
 - d) 1,101
 - e) 2A3h
 - f) FFFEh
 - g) 0Ah
 - h) Bh
 - i) 1110b
-

10 - Suponha que os dados abaixo estão armazenados, iniciando no offset (deslocamento) 0000h:

A	DB	7
B	DW	1ABCh
C	DB	'HELLO'

- a) Dê o endereço de offset associado às variáveis A,B,C
- b) Dê o conteúdo do byte no offset 0002h
- c) Dê o conteúdo do byte no offset 0004h
- d) Dê o endereço de offset do caractere "O" in "HELLO."

11 - Faça um programa para exibir um caractere "☺" e logo em seguida o programa deve soar um bip.

Dica: Código ASCII de ☺ = 01h e Bip = 07h

12 - Faça um programa para receber uma tecla (character) e mostrá-lo na linha seguinte entre asterisco. Por exemplo: Se você teclou "A", ele exibirá:

```
Ex07 <enter>
A
*A*
```

Dica: O character "line feed" = 0Ah e o character "carriage return" = 0Dh.

13 - Usando apenas MOV, ADD, SUB, INC e DEC, traduza as expressões abaixo em linguagem de alto-nível para linguagem de montagem.

- a) $A = B - A$
 - b) $C = A + B$
 - c) $B = 3 \times B + 7$
-

14 - Escreva um programa para:

- (a) mostrar um caractere "?";
- (b) ler dois dígitos decimais cuja soma seja < 10 ;
- (c) mostre-os e o resultado de sua soma na linha seguinte, com uma mensagem apropriada, por exemplo:

?27

A SOMA DE 2 E 7 EH 9

15 - Escreva um programa para apresentar uma caixa de 10x10 asteriscos.

Dica: Declare uma *string* com 10 '*' no segmento de dados

16 - Suponha uma operação de adição de AL e BL (ADD AL,BL), onde ambos registradores contenham 80h. Quais os estados dos FLAGS após esta operação?

- SF=
- PF=
- ZF=
- CF=
- OF=

17 - Suponha uma operação de subtração de AX, BX (SUB AX,BX) onde AX=8000h e BX=0001h. Quais os estados dos FLAGS após esta operação?

- SF=
 - PF=
 - ZF=
 - CF=
 - OF=
-

18 - Suponha uma operação de incremento em AL (INC AL) onde AL = FFh. Quais os estados dos FLAGS após esta operação?

- SF=
- PF=
- ZF=
- CF=
- OF=

19 - Suponha a operação MOV AX,-5. Quais os estados dos FLAGS após esta operação?

- SF=
- PF=
- ZF=
- CF=
- OF=

20 - Suponha uma operação NEG AX, AX = 8000h. Quais os estados dos FLAGS após esta operação?

- SF=
- PF=
- ZF=
- CF=
- OF=

21 - Para cada uma das instruções abaixo, indique os novos conteúdos dos FLAGS CF, SF, ZF, PF e OF, supondo que todos os FLAGS estão inicializados com 0, no início de cada instrução.

a) ADD AX, BX ; AX=7FFFh e BX=0001h

b) SUB AL, BL ; AL=01h E BL=FFh

- c) DEC AL ;AL=00h
- d) NEG AL ;AL=7Fh
- e) XCHG AX,BX ;AX=1ABCh e BX=712Ah
- f) ADD AL,BL ;AL=80h e BL=FFh
- g) SUB AX,BX ;AX=0000h e BX=8000h
- h) NEG AX ;AX=0001h

22 - Suponha que no registrador AL exista o valor A5h e no registrador BL o valor 69h. Qual o estado dos FLAGS OF e CF depois das operações abaixo?

- a) ADD AL,BL ; $AL \leftarrow AL + BL$
- b) SUB AL,BL ; $AL \leftarrow AL - BL$

23 - Podemos usar o programa abaixo para verificar as alterações nos FLAGS. Edite o programa, monte-o e faça a ligação com as opções para executar o programa com o TD. Após isto, execute o programa com o TD, verificando passo-a-passo, os valores dos FLAGS.

```
TITLE PRM81:FLAGS  
;Programa para ilustração de alterações nos flags de estado  
.MODEL SMALL  
.STACK 100H  
.CODE  
MAIN PROC  
; PARTE 1 Os efeitos no (ZF) ZERO FLAG  
; =====
```

```

        sub     ax,ax           ; Limpa AX - ZF = 1
        add     ax,1           ; ZF fica com 0
        add     ax,0FFFFh     ; ZF fica com 1
        add     ax,1           ; ZF fica com 0 novamente
; PARTE 2 Os efeitos no (SF) SIGN FLAG
; =====
        sub     ax,ax           ; Limpa AX - SF fica com 0
        sub     ax,1           ; SF fica com 1
        add     ax,2           ; SF fica com 0
; PARTE 3 Os efeitos no (CF) CARRY FLAG
; =====
        sub     ax,ax           ; Limpa AX - CF deve ficar 0
        mov     ax,0FFFFh     ; CF não é afetado
        add     ax,1           ; CF fica com 1
        add     ax,0           ; CF fica com 0
        sub     ax,1           ; CF fica com 1
; PARTE 4 Os efeitos no (OF) OVERFLOW FLAG
; =====
        sub     ax,ax           ; Limpa AX - OF fica com 0
; Adicao
        mov     al,128         ; OF não é afetado
        add     al,128         ; OF fica com 1
; Subtracao
        sub     ax,ax           ; Limpa AX - OF fica com 0
        add     ax,8000h       ; OF não é afetado
        sub     ax,7FFFh       ; OF fica com 1
; PARTE 5 Os efeitos no (PF) PARITY FLAG
; =====
        sub     ax,ax           ; Limpa AX - PF fica com 1
        add     ax,1           ; PF fica com 0
        add     ax,10h         ; PF fica com 1
        add     ax,0100h       ; a paridade não muda.
; PARTE 6 Terminando o Programa

```

```

; =====
        mov     ax,4C00h      ; AH - número da função DOS,
        int     21h          ; chamada da função 4C
MAIN ENDP
        END MAIN

```

24 - Suponha as condições iniciais AL = 11001011b e CF = 1. Dê o novo conteúdo de AL após cada uma das seguintes instruções, sempre com base nas condições iniciais acima:

- a) SHL AL,1
- b) SHR AL,1
- c) ROL AL,CL ;CL contendo 2
- d) ROR AL,CL ;CL contendo 3
- e) SAR AL,CL ;CL contendo 2
- f) RCL AL,1
- g) RCR AL,CL ;CL contendo 3

25 - Escreva um programa que peça ao usuário para entrar um caractere ASCII, na próxima linha exiba no monitor uma mensagem apresentando o valor binário deste código e numa segunda linha exiba outra mensagem que apresente o número de bits "1" existentes no código ASCII.

26 - Escreva um programa que peça ao usuário para entrar um caractere ASCII, na próxima linha exiba no monitor o caractere lido e numa segunda linha exiba outra mensagem que apresente o valor hexadecimal correspondente ao código ASCII. Repita este procedimento até que o usuário entre com *carriage-return* CR.

Por exemplo:

```

Digite um caractere: Z
codigo ASCII de Z em hexadecimal vale: 5Ah

```

Digite um caractere: ...

27 - Escreva um programa que peça ao usuário para entrar um número hexadecimal de 4 dígitos ou menos, terminado com CR, e exiba na próxima linha do monitor o mesmo número expandido em binário. Na leitura, faça com que apenas as letras maiúsculas de A a F sejam aceitas. Se houver um caractere ilegal, o programa deve emitir uma mensagem instruindo o usuário a tentar novamente.

28 - Escreva um programa que peça ao usuário para entrar um número binário de 16 dígitos ou menos, terminado com CR, e exiba na próxima linha do monitor o mesmo número compactado em hexadecimal. Se houver um caractere ilegal durante a entrada do número binário, o programa deve emitir uma mensagem instruindo o usuário a tentar novamente.

29 - Escreva um programa que peça ao usuário para entrar dois números binários, cada um com 8 dígitos ou menos, terminado com CR, e exiba na próxima linha do monitor o valor binário da soma destes números. Se houver um caractere ilegal durante a entrada dos números binários, o programa deve emitir uma mensagem instruindo o usuário a tentar novamente. Considere representação não sinalizada para os números. Considere que o 9º bit (vai um), se houver, estará em CF e poderá ser exibido, tal como no exemplo abaixo:

```
Entre o primeiro número binário (até 8 bits):      11001010  
Entre o segundo número binário (até 8 bits):       10011100  
A soma binária vale: 101100110 -> resultado de 9 bits
```

30 - Escreva um trecho de programa que faça a divisão inteira, indicando o quociente e o resto. Coloque o quociente em BH e o resto em BL, sem destruir o dividendo DH. Exemplo:

DH (original)		0001 0010 = 18d
após a divisão por 4:		
quociente	BH =	0000 0100 = 4d
com resto	BL =	0000 0010 = 2d

31 - De os conteúdos de DX, AX e CF/OF após a execução de cada uma das seguintes instruções (operandos de 16 bits com resultado em 32 bits):

- a) MUL BX ;se AX = 0008h e BX = 0003h
- b) IMUL CX ;se AX = 0005h e CX = FFFFh

32 - De os conteúdos de AX e CF/OF após a execução de cada uma das seguintes instruções (operandos de 8 bits com resultado em 16 bits):

- a) MUL BL ;se AL = ABh e BL = 10h
- b) IMUL BYTE1 ;se AL = 02h e BYTE1 = FBh

33 - De os conteúdos de AX e DX após a execução de cada uma das seguintes instruções (dividendo de 32 bits, divisor de 16 bits, resultando em quociente e resto ambos de 16 bits):

- a) DIV BX ;se DX = 0000h, AX = 0007h e BX = 0002h
- b) IDIV BX ;se DX = FFFFh, AX = FFFCh e BX = 0003h

34 - De os conteúdos de AL e AH após a execução de cada uma das seguintes instruções (dividendo de 16 bits, divisor de 8 bits, resultando em quociente e resto ambos de 8 bits):

- a) DIV BL ;se AX = 000Dh e BL = 03h
 - b) IDIV BL ;se AX = FFFBh e BL = FEh
-

35 - O que ocorre após a execução de:

- a) CWD ;se AX = 8ABCh
- b) CBW ;se AL = 5Fh

36 - Modifique o procedimento ENTDEC para que a mesma verifique se o dígito decimal que entra pelo teclado está na faixa de 0 a 9, e se o número final produziu ou não **overflow**.

Dica: No algoritmo apresentado, pode ocorrer **overflow** em dois momentos:

- quando AX = 10 X total
- quando total = 10 X total + valor binário

37 - Escreva um programa que leia uma quantidade de tempo, expressa em segundos, menor ou igual a 65535d, e apresente esta quantidade em horas, minutos e segundos (formato HH:MM:SS). Apresente mensagens adequadas e utilize os procedimentos ENTDEC e SAIDEC para realizar a entrada e saída de dígitos decimais (será necessário adaptá-las).

38 - Escreva um procedimento que calcule X^n , onde as variáveis **X** e **n** estão respectivamente em AX e BX. O resultado retorna para a rotina que chamou em dois registradores: DX (16 bits mais significativos) e AX (16 bits menos significativos). Inclua um teste de **overflow** caso o resultado não caiba em 32 bits.

39 - Escreva um programa completo que peça ao usuário, mediante mensagens adequadas, para entrar um número X, entrar um expoente n e apresentar a enésima potência de X (X^n). Utilize o procedimento desenvolvida no exercício 8 para o cálculo de X^n . Utilize os procedimentos ENTDEC e SAIDEC para E/S de números decimais. Apresente uma mensagem em caso de **overflow**.

40 - Supondo AX=FFFFh e BX=0001. Observe o trecho de código abaixo. O Fluxo será redirecionado para OUTRO_LUGAR?

CMP BX,AX

JG OUTRO_LUGAR

41 - Supondo que AX e BX contenham números sinalizados, escreva um pequeno trecho em linguagem de montagem para colocar o menor (dentre esses dois) em CX.

42 - Escreva um trecho de código em linguagem de montagem para mostrar os caracteres "i" ou "p", conforme o valor de AL. Se AL tiver valores 1 ou 3, mostre "i"; se AL tiver valores 2 ou 4, mostre "p"

43 - Escreva o trecho de código em linguagem de montagem para ler um caractere e se o caractere lido for igual a "n" ou "N", soar um BIP. Caso contrário, mostrar o caractere 3 vezes.

44 - Escreva um trecho de código assembly para ler caracteres até que um espaço em branco seja lido.

45 - Faça um programa que mostre a seguinte mensagem:

Digite uma linha de texto em caixa alta:

Em seguida, o programa deverá ir para a linha seguinte para receber a linha de texto (até que CR seja digitado). Depois deverá pular para a próxima linha e mostrar o menor e o maior caractere em caixa alta digitado:

Menor Letra: X - Maior Letra: Y

Caso nenhuma letra em caixa alta seja digitada, mostrar a mensagem:

"Nenhuma letra em caixa alta foi digitada!"

Exemplo de Execução:

```
EX45 <enter>
Digite uma linha de texto em caixa alta:
EU DIGITEI ESTE TEXTO PARA TESTAR! <enter>
Menor Letra: A - Maior Letra: X
—
```

46 - Escreva o código em linguagem de montagem, para cada uma das estruturas de decisão abaixo:

```
a) IF AX < 0
    THEN
        Coloque -1 em BX
    END_IF
```

```
b) IF AL < 0
    THEN
        Coloque FFh em AH
    ELSE
        Coloque 0 em AH
    END_IF
```

```
c) Supondo que DL contem um código ASCII de um caracter
IF (DL >= "A") AND (DL <= "Z")
    THEN
        Mostre DL
    END_IF
```

```
d) IF AX < BX
    THEN
        IF BX < CX
            THEN
                Coloque 0 em AX
            ELSE
                Coloque 0 em BX
            END_IF
        END_IF
```

```
e) IF (AX < BX) OR (BX < CX)
    THEN
        Coloque 0 em DX
    ELSE
        Coloque 1 em DX
    END_IF
```

```
f) IF AX < BX
    THEN
        Coloque 0 em AX
    ELSE
        IF BX < CX
            THEN
                Coloque 0 em BX
            ELSE
                Coloque 0 em CX
            END_IF
        END_IF
```

47 - Escreva a sequência de instruções para cada um dos itens abaixo que:

- a) Coloque a soma $1+4+7+\dots+148$ em AX
- b) Coloque a soma $100+95+90+\dots+5$ em AX

48- Implemente instruções de laço que:

- a) Coloque a soma dos 50 primeiros termos de uma sequência aritmética 1,5,9,13... em DX
- b) Leia um caractere e mostre-o 80 vezes na linha seguinte
- c) Leia uma senha de cinco caracteres e a sobrescreva usando cinco **X**'s usando o *carriage return*. Você não precisa armazenar esses caracteres.

49 - Escreva um programa que apresente um caractere '?', leia em seguida duas letras maiúsculas e exiba-as na próxima linha, em ordem alfabética.

50 - Modifique o programa de exibição de caracteres ASCII de forma a exibir 10 caracteres por linha separados por espaços em branco.

51 - Escreva um programa que pergunte ao usuário para teclar um dígito hexadecimal, exiba na próxima linha o seu valor decimal e pergunte ao usuário se deseja continuar a utilizar o programa: se for digitado **S** (sim), o programa se repete desde o começo; se for digitado outro caractere, o programa termina. Teste se o dígito hexadecimal está na faixa de valores correta. Se não estiver, exiba uma mensagem para o usuário tentar de novo.

Exemplo de execução:

```
Entre com um digito hexa: 9
Em decimal ele eh = 9
Você quer continuar? S
Entre com um digito hexa: c
Caractere ilegal - entre 0..9 ou A..F: C
Em decimal ele eh = 12
```

Você quer continuar? N

—

52 - Crie um trecho de código modificando o programa do exercício acima, tal que se o usuário falhar em entrar com um dígito hexadecimal na faixa correta mais do que três tentativas, o programa exibe uma mensagem adequada e termina.

53 - Escreva um programa que leia uma *string* de letras em caixa alta, terminadas com *carriage return* (CR) e mostre a maior sequência de caracteres alfabéticos consecutivos crescentes.

Exemplo de execução:

Entre com uma string em caixa alta:

FGHAEFGHC <enter>

A maior sequência de caracteres crescentes eh:

DEFGH

—

54 - Supondo que o segmento da pilha foi declarado como se segue:

.STACK 100h

Responda:

- Qual o valor em hexadecimal de SP quando o programa inicia?
- Qual o número máximo de palavras que podem ser armazenadas neste segmento de pilha?

55 - Supondo que AX=1234h, BX=5678h, CX=9ABCh, e SP=100h. Qual o conteúdo de AX, BX, CX e SP após a execução do trecho de código a seguir:

```
PUSH AX  
PUSH BX  
XCHG AX,CX  
POP CX  
PUSH AX  
POP BX
```

56 - O que acontece quando tentamos armazenar um valor na pilha quando $SP = 0h$? O que acontece com o programa?

57 - Suponha que um programa possua o trecho de código a seguir:

```
CALL PROC1  
MOV AX,BX
```

Supondo também:

- A instrução `MOV AX,BX` está armazenada no endereço: `08FD:0203h`,
- `PROC1` é um procedimento do tipo NEAR e começa em `08FD:300h`, e
- $SP=010Ah$.

Quais são os conteúdos de IP e SP no instante após a execução da instrução `CALL PROC1`? O que existe no topo da pilha?

58 - Escreva um programa que possibilite a entrada pelo usuário de um texto, contendo palavras, separadas por espaço e terminadas por "CR". O programa deve apresentar o texto na outra linha na ordem de entrada, entretanto substituindo o espaço por underline (`_`).

Exemplo de Execução:

```
este eh um teste  
Este_eh_um_teste
```

59 - O método abaixo pode ser usado para gerar números aleatórios na faixa de valores entre 1 e 32767.

Comece com um número nesta faixa de valores
Faça um deslocamento para esquerda uma vez
Substitua o bit 0 pelo XOR dos bits 14 e 15
Limpe o bit 15

60 - Escreva os procedimentos a seguir:

- a. Um procedimento READ que permite que o usuário entre com um número binário e armazene em AX.
- b. Um procedimento RANDOM que recebe um número em AX e retorna um número aleatório em AX
- c. Um procedimento WRITE que mostra o número binário em AX.

Em seguida, escreva um programa que apresente um "?", chame o procedimento READ para ler um número binário e chame os procedimentos RANDOM e WRITE para calcular e mostrar 10 números aleatórios. Os números devem ser mostrados 2 por linha, separados por quatro espaços em branco.

61 - Crie uma macro para salvar e outra restaurar todos os registradores na pilha.

62- Suponha que a variável CLASSE armazene na memória o nome e as quatro notas (notas inteiras entre 0 e 100), referentes a quatro alunos de um curso, sendo definida como:

CLASSE	DB	'Antonio Carlos	',50,49,19,25
	DB	'Carlos Roberto	',35,74,100,86

DB	'Maria Carla	',45,57,63,67
DB	'Valeria Cristina	',35,0,34,23

Suponha que o campo de nome contenha 20 *bytes* reservados, ou seja, a primeira nota está armazenada somente no *byte* 21 de cada linha. Escreva um trecho de programa, baseado na definição da variável CLASSE acima, que exiba no monitor de vídeo o nome do aluno, seguido de sua média final (que será truncada para o inteiro inferior à média real), seguido da palavra aprovado se a média for maior ou igual a 50 e reprovado, se menor que 50.

63 - Supondo um vetor de 30 palavras chamado W. Escreva a sequência de comandos que faça a troca do 10° com o 25° elemento deste vetor.

64 - Escreva o trecho de código em linguagem de montagem para somar em AX os 10 elementos do array W definido por:

W	DW	10,20,30,40,50,60,70,80,90,100
---	----	--------------------------------

65 - Escreva um procedimento INVERSO, que inverterá um vetor de N palavras. Isto significa que a n-ésima palavra se tornará a primeira e a primeira se tornará a última, e assim por diante.

O procedimento receberá como parâmetros: SI o *offset* do vetor e BX o número de elementos.

66 - Converta cada letra minúscula na *string* a seguir em letra maiúscula (equivalente). Use modo de endereçamento indexado.

MSG	DB	'esta eh uma mensagem'
-----	----	------------------------

67 - Na mesma string do exercício 5, troque os caracteres "m" por "M".

- a) usando modo indireto e
- b) modo indexado.

68 - Supondo os dados declarados abaixo:

```
.DATA  
A DW 1234h  
B LABEL BYTE  
    DW 5678h  
C LABEL WORD  
C1 DB 9Ah  
C2 DB 0Bch
```

Quais das instruções abaixo são legais? Se forem legais, informe o valor que foi movido.

- a) MOV AX,B
 - b) MOV AH,B
 - c) MOV CX,C
 - d) MOV BX, WORD PTR B
 - e) MOV DL, WORD PTR C
 - f) MOV AX, WORD PTR C1
-