

OD82:0100	B402	MOV	AH, 02
OD82:0102	B241	MOV	DL, 41
OD82:0104	CD21	INT	21
OD82:0106	CD20	INT	20
OD82:0108	69	DB	69

Capítulo 10

DETALHES COMPLEMENTARES

Este capítulo mostra alguns detalhes técnicos ainda não comentados, como, por exemplo, endereçamento de memória pela manipulação dos registradores de segmento e registradores de deslocamento, programas executáveis, bibliotecas externas, manipulações básicas (teclado e cursor) e valores negativos.

10.1 - Mais sobre segmentos e deslocamentos

No capítulo 2 o tema sobre segmentos e deslocamentos foram comentados superficialmente para situar o leitor nos parâmetros básicos do funcionamento da linguagem de programação de computadores Assembly 8086/8088. Nos demais capítulos o assunto não foi aprofundado. É propósito, neste momento, relembrar alguns pontos apresentados e introduzir novos detalhes técnicos ainda desconhecidos:

- ◆ O microprocessador Intel 8086/8088¹ foi concebido com a capacidade de endereçar fisicamente no máximo uma memória de até 1 MB, ou seja, 2^{20} (1.048.576), ou seja, de **0000h** até **FFFFh**.
- ◆ A memória de 1 MB é dividida em 16 blocos (segmentos de memória) de 64 KB endereçados da posição **0h** (primeiro segmento de memória) até a posição **Fh** (décimo sexto segmento de memória). Cada segmento de memória trabalha com até 64 KBytes (65.536) de dados, ou seja, cada segmento está dividido em deslocamentos (*offsets*) que variam de **0000h** até **FFFFh**. Logicamente a memória está organizada em quatro segmentos de trabalho, sendo: *code*, *data*, *stack* e *extra*.
- ◆ Cada deslocamento permite manipular um dado de até 16 bits, ou seja, um dado do tipo *word* (manipulado pelos registradores de segmento).
- ◆ Cada *word* (posição de memória) é representado por meio do endereço cartesiano **xxxxh:yyyyh** (segmento:deslocamento), em que **xxxxh** é o segmento e **yyyyh** é o deslocamento.

Para melhor entendimento da estrutura organizacional interna do microprocessador 8086/8088, observe todos os detalhes apresentados na Figura 10.1.

Para entender o motivo que levou a empresa Intel a adotar essa estrutura de endereçamento, é necessário observar alguns pontos históricos, apontados por Hyde (2003), Norton (1993) e também pela empresa Intel (2004):

- ◆ O microprocessador 8086/8088 foi lançado no ano de 1978, ocasião em que o custo de produção de memórias era alto para os padrões de mercado. Nesse período os microcomputadores em uso eram máquinas de 8 bits com capacidade de trabalhar com 48 KB de memória.

¹ A partir dessa limitação os demais microprocessadores da família Intel lançados a partir do modelo 8086/8088 possuem essa mesma característica, tendo como diferença a possibilidade de trabalhar com maior capacidade de memória, dependendo do modelo em uso.

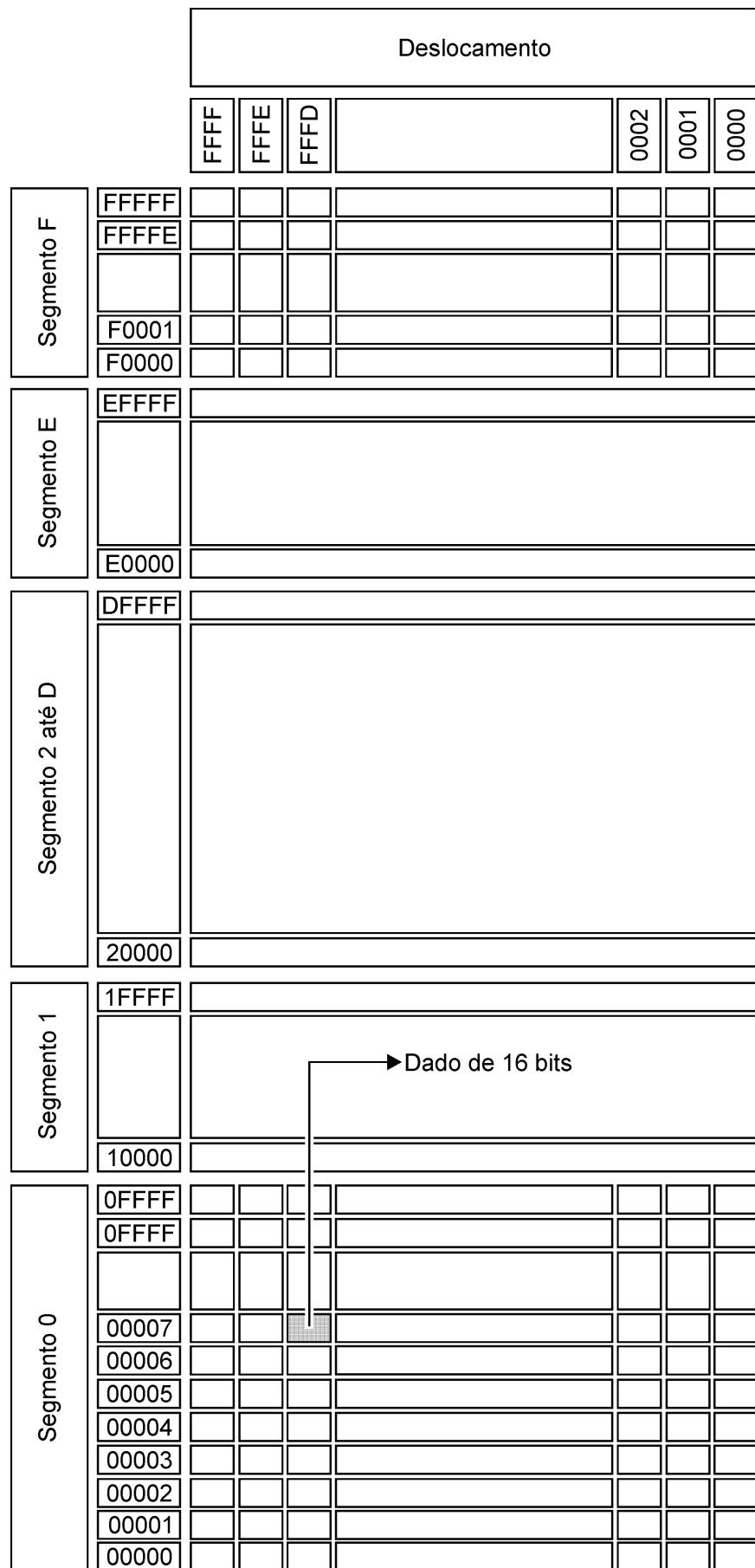


Figura 10.1 - Mapeamento de memória de 1 MB do processador 8086/8088.

- ◆ No ano de 1981, a empresa IBM lança seu microcomputador de uso pessoal (PC) trabalhando com 640 KB (dotado com o microprocessador Intel 80882, uma versão do processador 8086/8088 lançado em 1979) de memória com a capacidade de manipular dados de 16 bits. Nesse período os microcomputadores vendidos no mercado norte-americano possuíam a capacidade de memória de 64 KB, alguns chegavam a 128 KB, operando dados em processamento a 8 bits.
- ◆ Nessa ocasião poder usar uma memória de 1 MB era algo fantástico. A Intel não acreditava que o processador 8086/8088 duraria tanto, apesar de sua grande capacidade para a época, pois os modelos anteriores de microprocessadores tiveram um ciclo de vida em torno de cinco anos. Assim, não se preocupou em trabalhar em cima do processador 8086/8088, pois tinha planos para o desenvolvimento de outros processadores e a linha 8086/8088 não fazia parte desses planos.
- ◆ Devido ao sucesso estrondoso de vendas dos microcomputadores IBM-PC e da grande quantidade de softwares desenvolvidos, a Intel acabou sendo pega de surpresa e não pôde mais abandonar o projeto do processador 8086/8088, gerando uma família de processadores x86.
- ◆ Em 1982, esbarrou-se no limite de 1 MB do processador 8086/8088, foi quando a Intel lançou o processador 80286 e acabou ficando presa à estrutura interna do processador 8086/8088, pois todos os softwares escritos para os computadores da família IBM-PC estavam calcados na estrutura anterior utilizada no processador 8086/8088, ou seja, conseguiam chegar até 1 MB de memória e os 16 MB total de memória que o processador 80286 disponibilizava ficavam resumidos aos míseros 1 MB, com uma perda de 15 MB.
- ◆ O problema não era a memória endereçável, mas o processador 8086/8088 ser de 16 bits, com registradores de 16 bits e endereços de 16 bits, limitando a capacidade de endereçamento de memória em 64 KB.
- ◆ A forma que a Intel encontrou para solucionar o problema da capacidade de endereçamento da memória, criando a segmentação com 64 KB, foi muito inteligente. No entanto, a solução era viável para endereçar até 1 MB.
- ◆ O problema era como aumentar a capacidade de 64 KB para endereçar um volume maior de memória. Isso exigiria um esforço muito maior, o que foi de certa forma resolvido pelos engenheiros da Intel ao criarem a segmentação de blocos de memória.
- ◆ A segmentação não é ruim. O problema está na maneira como a implementação foi feita pela Intel em 1978, com o lançamento do processador 8086/8088 e que é de certa forma usada até hoje.
- ◆ Não se pode culpar a Intel por isso, pois ela não tinha grandes perspectivas para o processador 8086/8088 e se viu obrigada a manter a estrutura original mesmo em novo processador, como foi o caso do 80286. É importante considerar que a Intel resolveu o problema com o lançamento do processador 80386 em 1985.
- ◆ Devido ao grande volume de softwares desenvolvidos para os IBM-PCs durante os anos de 1980, incluindo-se o sistema operacional MS-DOS, a forma de segmentação adotada originariamente pela Intel não foi abandonada.
- ◆ Sistemas operacionais como LINUX e versões do Windows a partir da edição 95 não sofrem dos mesmos males que o MS-DOS, mas os processadores mais novos até os últimos lançamentos da linha Pentium, bem como do concorrente AMD, ainda por questões de compatibilidade utilizam segmentação.

A partir dos pontos apresentados fica mais fácil entender os motivos que levaram a empresa Intel a adotar a segmentação da memória e deslocamento de endereço, os quais acabaram sendo largamente utilizados na época.

O processador 8086/8088 utiliza quatro registradores de segmentos de memória: **CS** (code segment), **DS** (data segment), **SS** (stack segment) e **ES** (extra segment), desses quatro os mais utilizados são **CS**, **DS** e **SS**; cinco registradores de deslocamentos: **SI** (source index), **DI** (destination index), **SP** (stack pointer), **BP** (base pointer) e **IP** (instruction pointer). São sobre esses registradores que as operações de **segmento:deslocamento** são efetivadas. As operações de endereçamento de **segmento** são usadas nos registradores **CS**, **DS** e **SS** e as operações de endereçamento **deslocamento** são usadas nos registradores **SI**, **DI**, **SP**, **BP** ou **IP**, podendo-se também utilizar o registrador **BX** para a indicação de um endereço de deslocamento.

A Tabela 10.1 mostra as principais operações executadas nas áreas de segmentos de código, pilha, dados e extra e a Figura 10.2 apresenta esquema organização da memória de um microprocessador 8086/8088.

² Os microprocessadores 8086 e 8088 são idênticos, ou seja, ambos são processadores de 16 bits, exceto pela quantidade de dados que podiam receber e enviar de uma só vez quando da necessidade de comunicação com periféricos internos (unidade de disco) e externos (impressororas). O processador 8086 fazia essa comunicação a 16 bits, enquanto o processador 8088 fazia a 8 bits apesar de processar os dados internos a 16 bits. Isso tudo porque a maioria dos dispositivos e circuitos disponíveis na ocasião era de 8 bits (NORTON, 1993, p. 13).

Tabela 10.1 - Operações das áreas de segmento

Referência à memória	Identificador de Segmento	Identificador de Deslocamento
Segmento de código	CS	IP
Segmento de pilha	SS	SP, BP
Segmento de dados	DS	BX, SI, DI
Segmento extra	ES	DI

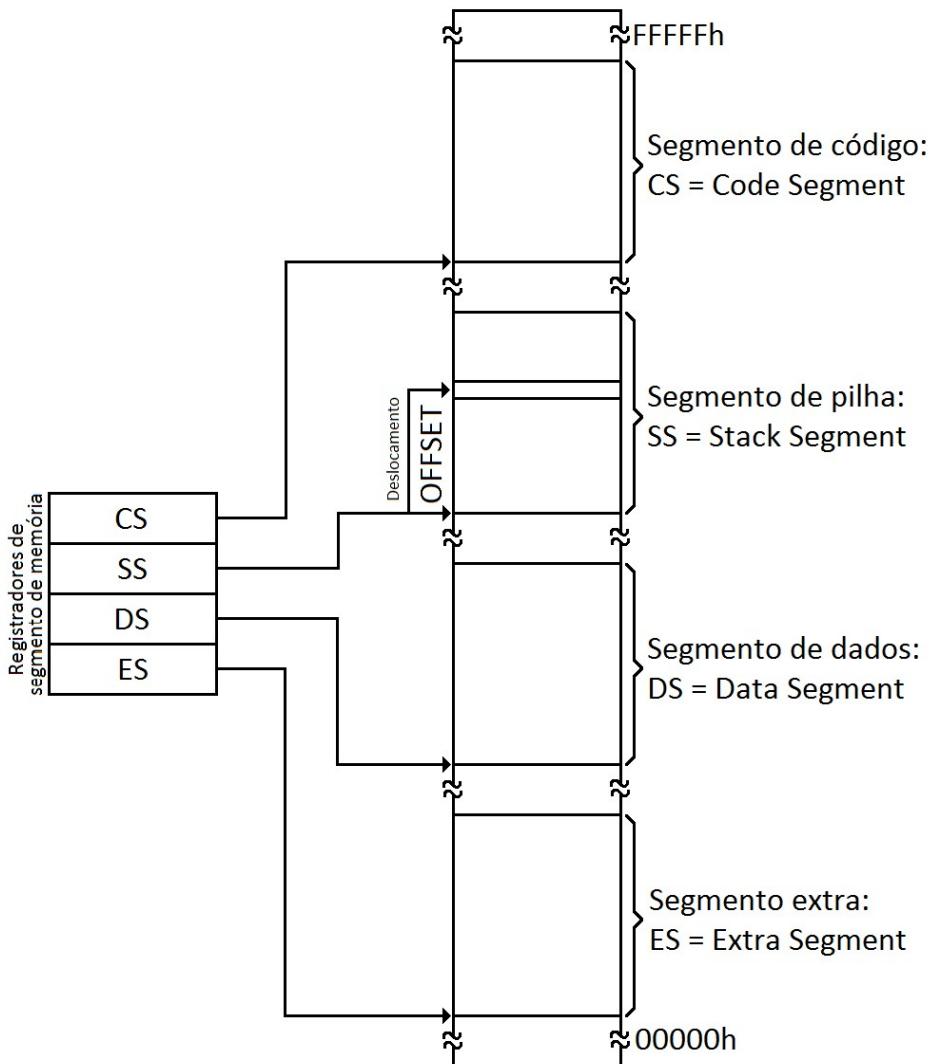


Figura 10.2 – Organização da memória de um microprocessador 8086/8088. Adaptado de YADAV, 2008.

Os registradores de segmento: **CS** tem por finalidade armazenar o endereço de início do código do programa a ser executado no segmento de memória, **DS** tem por finalidade armazenar o endereço onde se localizam os dados (variáveis) a serem usadas no programa, **SS** tem por finalidade armazenar os endereços da pilha e dos dados e **ES** tem por finalidade armazenar o endereço adicional de mais dados que venham a ser necessários ao programa se assim necessitar.

O microprocessador 8086/8088 opera com dois tipos de endereçamento de memória: *lógico* e *físico*. O endereço lógico é utilizado pelos programadores, enquanto o endereço físico é utilizado pelo *hardware* para acessar posições de memória. No entanto, é o próprio programador que direciona para o *hardware* o endereço físico que será usado a partir da informação do endereço lógico.

Os endereços físicos, para serem utilizados, ocupam um total de 20 bits. Internamente o microprocessador 8086/8088 opera com registradores de segmento com no máximo de 16 bits, o que gera a necessidade de utilizar mais 4 bits emprestados de um registrador de deslocamento.

O endereço do registrador de segmento é multiplicado por **10h** obtendo-se um endereço de 20 bits. A partir desta operação ocorre um deslocamento de 4 bits para a esquerda do valor e insere-se o valor obtido na parte direita do segmento, para então somar a ele o valor do endereço do registrador de deslocamento.

Os 4 bits excedentes que são a parte menos significativa do registro de 20 bits são sempre marcados com quatro bits setados a zero e os 16 bits mais significativos ficam definidos no par de registradores de base e de ponteiro, como ocorre, por exemplo, no caso do formato **CS:IP** que é uma das referências de endereço lógico mais usadas.

Vale relembrar que os registradores na sua forma geral possuem a capacidade de armazenar 16 bits de dados, ou seja, 2 bytes. Cada byte é subdividido em 2 nibbles, cada nibble é um conjunto de 4 bits e representa uma unidade de valor hexadecimal entre **0h** e **Fh**. Assim sendo, observe a Figura 10.3 que representa o cálculo do endereço físico **B2213h** a partir do segmento **A705h** e de deslocamento **B1C3h**, ou seja, **A705:B1C3**.

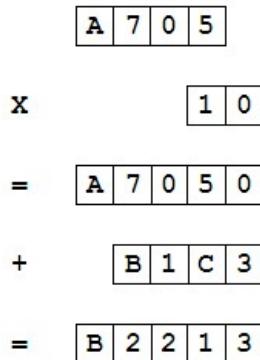


Figura 10.3 - Demonstração do cálculo do endereço físico de memória.

Um endereço de memória, segundo Hyde (2003), é aquele que "contém um componente *segmento* e outro componente *deslocamento*". Por esta razão é que normalmente se faz referência à posição lógica de memória utilizando a nomenclatura **segmento:deslocamento**, sendo esses dois valores constantes de 16 bits³.

A partir das coordenadas de *segmento* e *deslocamento* e de acordo com o exposto anteriormente, torna-se possível descobrir o endereço físico real a ser utilizado na memória. Isso é conseguido com a fórmula: **ENDEREÇO FÍSICO = SEGMENTO X 10h + DESLOCAMENTO**, em que, **SEGMENTO** é a informação de endereço existente no registrador de segmento **CS** (*code segment*) e **DESLOCAMENTO** é a informação do offset existente no registrador de deslocamento **IP** (*instruction pointer*). O valor **10h** (equivalente a 16 em notação decimal) é a capacidade em bits de armazenamento. O microprocessador utiliza as instruções de um programa com base no endereço de **segmento:deslocamento**, ou seja, com base nas informações dos registradores **CS:IP**.

Imagine obter o endereço físico (endereço real) da memória com base no endereço de segmento e deslocamento **2425:5121** (**segmento:deslocamento**). Basta substituir os valores na fórmula:

$$\text{ENDEREÇO FÍSICO} = 2425\text{h} \times 10\text{h} + 5121\text{h}$$

Após fazer o cálculo do valor **2425h** multiplicado pelo valor **10h** (valor 16 em decimal - 16 bits), obter-se-á o valor **24250h** que somado ao valor **5121h** resulta no endereço físico **29371h**. Apesar de esse trabalho ser realizado automaticamente pelo microprocessador, é interessante saber como funciona.

O programa **emu8086** pode ser usado para conferir o cálculo da geração de endereço físico de 20 bits a partir da definição de valores junto aos registradores **CS:IP**. Para proceder a este teste encerre o programa **emu8086** e carregue-o novamente. Em seguida execute no programa **emu8086** o comando de menu **file/new/com template**, acione as teclas de atalho **<Ctrl> + <A>** e sem escrever nenhum código acione o botão **emulate** ou execute a tecla de função **<F5>**. Ao ser apresentada a caixa de diálogo **8086 microprocessor emulate** indicada na Figura 10.4 entre os valores de segmento e deslocamento nos campos dos registradores **CS** e **IP** assinalados.

³ No processador 8086/8088 com deslocamentos de 16 bits, um segmento não pode ser maior do que 64 KB; pode ser menor (e a maioria é), mas nunca maior. Os processadores 80386 e posteriores permitem deslocamentos de 32 bits (o dobro do processadores 8086/8088) com segmentos de até 4 gigabytes (HYDE, 2003).

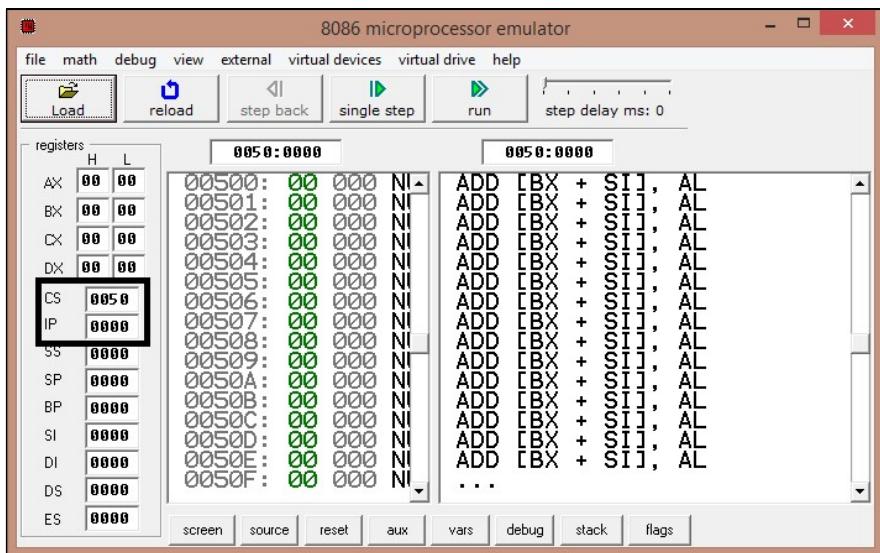


Figura 10.4 - Caixa de diálogo 8086 microprocessor emulate.

Junto aos campos dos registradores **CS** e **IP** informe respectivamente os valores **A705h** e **B1C3h** e observe a indicação automática do valor **B2213h** como mostrado junto a Figura 10.5 para a área que indica os valores do programa em opcodes.

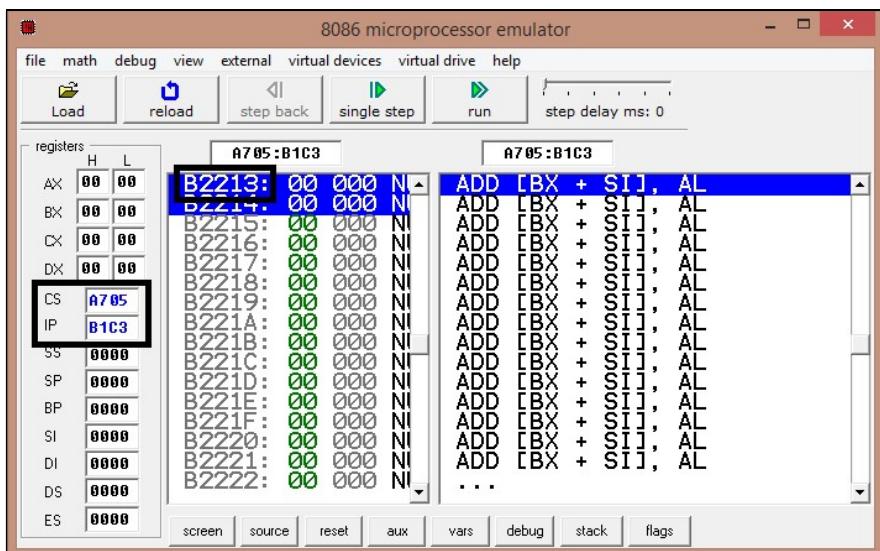


Figura 10.5 - Caixa de diálogo 8086 microprocessor emulate com cálculo de endereço físico.

Os valores dos registradores de segmento possuem inicialmente o mesmo valor quando o programa em desenvolvimento é montado como um arquivo do tipo **.COM**, normalmente definidos a partir do endereço de deslocamento **0100h** (**org 100h**). Para confirmar este fato, considere o seguinte programa:

```
;*****  
;* Programa: SEGM1.ASM *  
;*****
```

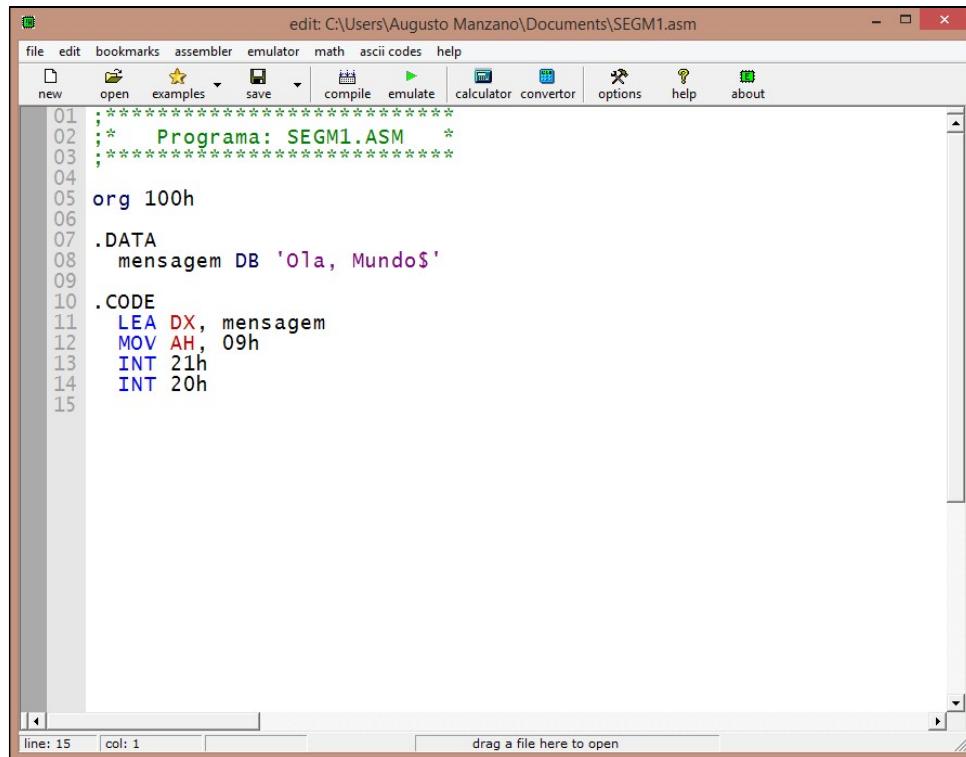
```
org 100h  
  
.DATA  
mensagem DB 'Olá, Mundo$'  
  
.CODE  
LEA DX, mensagem
```

```

MOV AH, 09h
INT 21h
INT 20h

```

Execute no programa **emu8086** o comando de menu **file/new/com template**, ação as teclas de atalho **<Ctrl> + <A>** do editor de texto e escreva o programa anterior, gravando-o por meio dos comandos de menu **file/save** com o nome **SEGM1**, de forma que fique semelhante à imagem da Figura 10.6.



```

edit: C:\Users\Augusto Manzano\Documents\SEGM1.asm
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator converter options help about
01: ****
02: ;* Programa: SEGM1.ASM *
03: ****
04:
05 org 100h
06
07 .DATA
08 mensagem DB 'Olá, Mundo$'
09
10 .CODE
11 LEA DX, mensagem
12 MOV AH, 09h
13 INT 21h
14 INT 20h
15

```

Figura 10.6 - Programa SEGM1 na ferramenta emu8086.

Em seguida, execute o comando de menu **assembler/compile and load in the emulator** ou ação a tecla de função **<F5>**. Na janela **Emulator: SEGM1.com_** são mostrados os valores dos registradores de segmento **CS**, **IP**, **DS**, **SS** e **ES**, como indica a Figura 10.7.

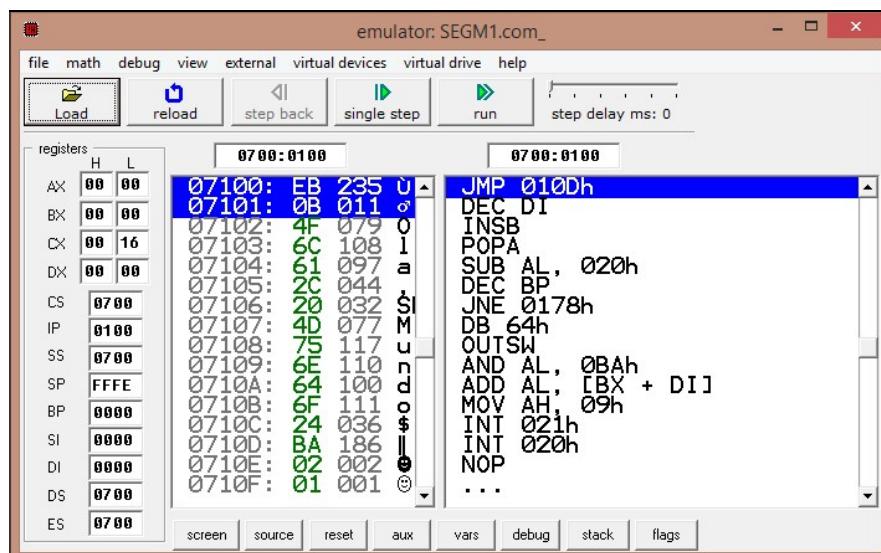


Figura 10.7 - Janela Emulator: SEGM1.com_ indicando o endereço de segmento 0700:0100.

Note em especial os registradores **CS:IP** que indicam o uso do endereço lógico de memória **0700:0100**. Além do conjunto de registradores **CS:IP** mostrarem o valor de endereço lógico de memória correspondente ao endereço físico de memória **07100h** (**0700h x 10h + 0100h**), os registradores **DS**, **SS** e **ES** também indicam o valor de endereço de segmento **0700**. Perceba que na área **Register** os registradores **CS** e **IP** mostram, respectivamente, os valores **0700h** e **0100h** que são o endereço lógico e na área **Memory** (quadro central do lado direito da área *register*) é indicado o valor **07100h** na primeira linha, informando assim o endereço físico na memória.

Ao ser executado o programa passo a passo, o valor do registrador **CS** permanece inalterado enquanto estiver no mesmo endereço de segmento de memória, enquanto o valor do registrador de deslocamento **IP** é alterado constantemente. Os demais valores dos registradores de segmento permanecem fixos.

Na Figura 10.7 note a indicação da marca de seleção sobre a instrução **JMP 010Dh**. Esta instrução faz um salto do fluxo de execução do programa para a linha de código **010Dh**. Observe o valor do registrador de segmento **IP** indicando **0100h**. Acelere a tecla de função **<F8>** pela primeira vez para iniciar o processo de execução passo a passo do programa, como apresenta a Figura 10.8. Note o valor **010Dh** definido como endereço do registrador de segmento **IP**.

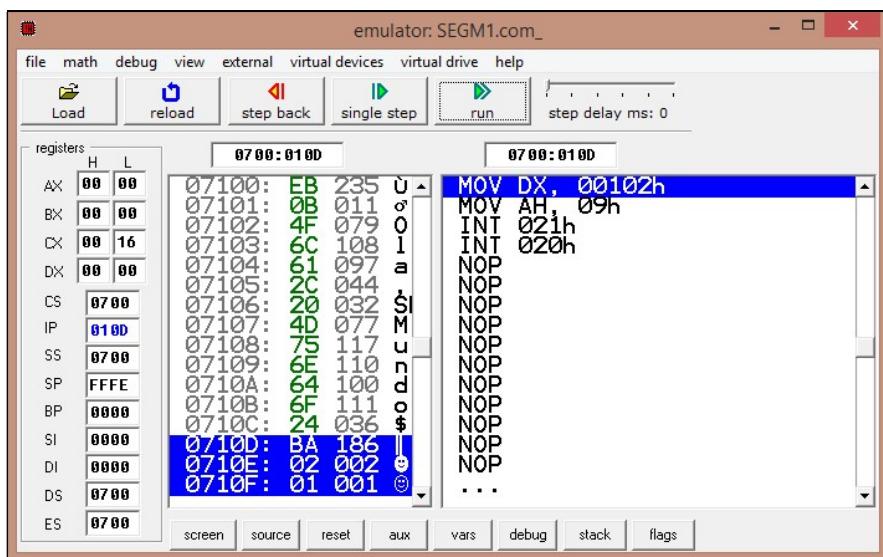


Figura 10.8 - Janela Emulator: SEGMENT1.com_ com ação da tecla <F8>.

Acelere a tecla de função **<F8>** mais três vezes e aparece a indicação do valor do registrador de segmento **CS** apontando para a região de memória em que se encontra o código interno de execução da instrução **INT 21h**. Ao ser executada a linha de instrução **INT 21h**, o programa é desviado para uma sub-rotina interna que faz a interrupção. Ocorre a alteração do registrador de segmento **CS** e do registrador de deslocamento **IP** a fim de indicar para o programa em que posição da memória ele deve ir para executar a ação desejada.

A Figura 10.9 mostra essa ocorrência em que os registradores **CS** e **IP** possuem, respectivamente, os valores **F400h** e **0200h** (**CS:IP** como endereço de segmento **F400:0200**, ou seja, endereço físico **F4200h**). Note também que o registrador **SP** possui seu valor **FFFFEh** alterado para o valor **FFF8h**.

Quando o registrador de deslocamento **SP** é alterado para **FFF8h** este indica que a pilha foi acionada. Esse efeito ocorreu devido à alteração do valor do registrador de segmento **CS**. Antes de o registrador de segmento **CS** ser alterado, o seu endereço **0700** foi armazenado na pilha, como pode ser conferido na Figura 10.10 (para ver execute o comando **view/stack** na janela **SEGMENT1.com**).

A Figura 10.10 mostra acima da barra de seleção atual (**0700:FFF8**) a linha definida com o endereço **0700:FFFF** em que se encontra o valor **0700**, endereço de memória onde se encontra o código do programa. Assim que a ação da instrução **INT 21h** for executada, o valor armazenado na pilha (que está na posição de memória **0700:FFF8**) retorna ao registrador de segmento **CS**.

Ao executar a tecla de função **<F8>** mais duas vezes, é possível ver a apresentação da mensagem e o retorno do valor **0700** para o registrador de segmento **CS**.

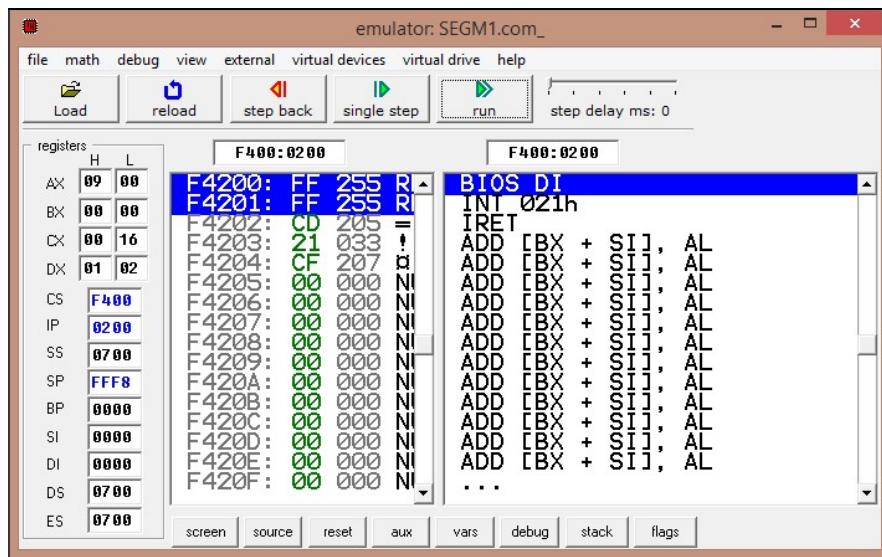


Figura 10.9 - Janela Emulator: SEGMENT1.com_ com alteração do registrador CS, IP e SP.

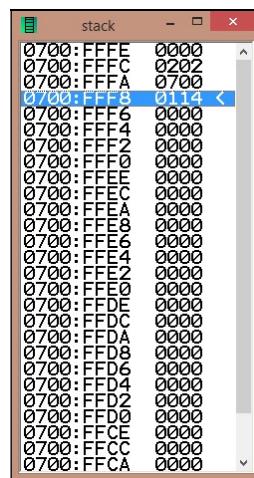


Figura 10.10 - Janela Stack com valor do registrador CS armazenado.

Na sequência execute a tecla de função <F8> mais duas vezes, acione o botão **OK** da caixa de diálogo **message** para encerrar o programa e saia do modo de execução do programa retornando à tela do editor de textos.

10.2 - Programas executáveis

Até esse momento os programas desenvolvidos em linguagem de programação Assembly 8086/8088 utilizaram a diretiva **org 100h** ou as diretivas **.MODEL small** e **.STACK 512d**. Os programas que usaram as diretivas **.MODEL small** e **.STACK 512d** realizaram operações com valores numéricos e os programas que usaram a diretiva **org 100h**, além de usar valores numéricos, fizeram apresentação de sequências de caracteres (*string*).

Os programas que usam as diretivas **.MODEL small** e **.STACK 512d**, quando compilados, possuem a extensão **.EXE** e os programas que usam a diretiva **org 100h**, quando compilados, possuem a extensão **.COM**.

Os programas com extensão **.EXE** são definidos com algumas diferenças em relação aos programas com extensão **.COM**, pois os registradores de segmento **CS** e **SS** são iniciados com valores diferentes dos registradores **DS** e **ES**.

Os programas compilados com extensão **.COM**, como já explanado, são programas com estrutura simples que possuem seu código em linguagem de máquina muito pequeno, ocupando até 64 KBytes. Já os programas compilados com extensão **.EXE** são programas com estrutura avançada, pois é possível trabalhar com cabeçalhos, realocação de recursos, entre outras possibilidades.

Para trabalhar com arquivos de programas com extensão .EXE, a partir do programa emu8086 é necessário acionar os comandos de menu file/new/exe template. Será apresentado, na sequência, o código de programa seguinte:

```
01 ; multi-segment executable file template..
02
03 data segment
04     ; add your data here!
05     pkey db "press any key...$"
06 ends
07
08 stack segment
09     dw 128 dup(0)
10 ends
11
12 code segment
13 start:
14 ; set segment registers:
15     mov ax, data
16     mov ds, ax
17     mov es, ax
18
19     ; add your code here
20
21     lea dx, pkey
22     mov ah, 9
23     int 21h      ; output string at ds:dx
24
25     ; wait for any key....
26     mov ah, 1
27     int 21h
28
29     mov ax, 4C00h ; exit to operating system.
30     int 21h
31 ends
32
33 end start ; set entry point and stop the assembler.
```

Se o programa for executado, neste momento. É apresentada a mensagem “**press any key...**”. Ao olhar para o modelo de programa indicado, percebe-se que este possui uma estrutura sintática um pouco mais complexa que as utilizadas nos exemplos anteriores deste livro. Num programa .EXE, é necessário se preocupar com a definição dos segmentos de dados (linhas de **03** até **06**), pilha (linhas de **08** até **10**) e código (linhas de **12** até **33**) de forma mais explícita que em um programa .COM. Não é necessário se preocupar com a posição de endereçamento da memória, a menos que estivesse sendo utilizada a ferramenta **DEBUG** para a geração do código.

As linhas iniciadas com o caractere ponto e vírgula se configuram por serem linhas de comentários, normalmente usadas para se estabelecer a identificação de ações do programa com a finalidade de orientar de forma mais clara o código do programa na documentação interna.

Nas linhas **03**, **08** e **12** são definidos os trechos de códigos identificados pelas diretivas **SEGMENT** após a identificação das áreas **data**, **stack** e **code** e também a diretiva **ENDS** para encerramento da diretiva **SEGMENT**.

Os segmentos com a utilização das diretivas **SEGMENT** e **ENDS** obedecem à seguinte estrutura sintática:

```
[nome] SEGMENT
        [corpo do seguimento]
[nome] ENDS
```

O rótulo *nome* é obrigatório para identificar o segmento em uso, podendo-se definir qualquer nome escolhido pelo programador.

O trecho para a área de dados, existente entre as linhas de código **03** e **06**, deve conter os dados que o programa irá manipular.

O trecho para a pilha, existente entre as linhas de código **08** e **10**, estabelece o valor de 128 bytes de espaço para a manipulação da pilha com dados do tipo **DW**. O operador **dup(0)** definido na linha **09** estabelece que o valor **0** (zero) será repetido um determinado número de vezes na memória. Na linha **09** encontra-se a instrução **dw 128 dup(0)**, que define que serão alocados na memória 128 bytes com valor zero. O operador **dup(?)** permite alocar um tamanho limite na memória sem a definição de valores.

O trecho para a definição da área de código, existente na linha **19**, deve ser utilizado para se inserir nessa posição o código de programa. Note que o trecho de linhas entre **12** e **33** possui a definição de parte do código de programa de forma automática (característica da ferramenta **emu8086**).

Para demonstrar o uso de programas **.EXE**, acompanhe as seguintes instruções de preenchimento do modelo de programa na ferramenta **emu8086** e grave o programa com o nome **SEGM2**:

- ◆ Vá até a linha **05** e sobre escreva a linha existente pela instrução **mensagem db "Olá, Mundo\$"** e ao final acione a tecla <Enter> e na linha **29** substitua a linha existente pela instrução **lead x, mensagem**. Remova as linhas de código **31, 32 e 33**.
- ◆ A partir da linha **19** retire a linha de comentário e acrescente o código:

```
MOV      DX, OFFSET mensagem  
MOV      AH, 09h  
INT      21h  
MOV      AH, 4Ch  
INT      21h
```

O trecho de código com as instruções **MOV AH, 4Ch** e **INT 21h** procede com o encerramento do programa e a devolução do controle de execução do programa ao sistema operacional. Esta é uma forma alternativa ao modo **INT 20h**. As instruções **MOV AH, 4Ch** e **INT 21h** carregam o código de retorno para o registrador geral **AL**, sendo então considerado retorno normal quando o valor de **AL** for **0** (zero), sendo diferente de zero há algum erro no retorno do programa. O retorno do programa ao sistema operacional ocorre com a execução da interrupção com o código de função **4Ch** em **AH**.

Usar a forma **MOV AH, 4Ch** e **INT 21h** caracteriza a maneira adequada de retorno ao sistema operacional, mas não adequada para uso junto ao programa **Enhanced DEBUG** que faz uso da instrução **INT 20h**.

A seguir é apresentado o código completo do programa. Atente para as linhas grafadas em negrito contendo as inserções anteriormente solicitadas:

```
; multi-segment executable file template.  
  
data segment  
    ; add your data here!  
    mensagem db "Olá, Mundo$"  
ends  
  
stack segment  
    dw 128 dup(0)  
ends  
  
code segment  
start:  
; set segment registers:  
    mov ax, data  
    mov ds, ax  
    mov es, ax  
  
    MOV      DX, OFFSET mensagem  
    MOV      AH, 09h  
    INT      21h
```

```

MOV      AH, 4Ch
INT      21h

lea dx, mensagem
mov ah, 9
int 21h      ; output string at ds:dx

mov ax, 4C00h ; exit to operating system.
int 21h
ends

end start ; set entry point and stop the assembler.

```

Observação

A definição de strings pode ser realizada tanto com apóstrofos (aspas simples) como com aspas (aspas inglesas).

Aparentemente não existirá diferença do ponto de vista externo do programa, mas internamente sim. Os programas com estenção .EXE são executados um pouco mais devagar e são maiores que os programas com estenção .COM.

Apesar de a ferramenta emu8086 ser muito prática e agradável de trabalhar, em alguns casos, como na definição de programas do tipo .EXE, ela torna o código um pouco complexo. Não por culpa dela, mas da própria estrutura interna desse tipo de programa que é mais complexa do que a de programas .COM.

Execute o comando de menu **file/new**, escolha qualquer uma das opções de *template* apresentadas, acione as teclas <Ctrl> + <A> para selecionar todo o texto existente e em seguida acione a tecla para limpar toda a área de texto. Na sequência escreva o código do programa a seguir:

```

TITLE    Teste de Segmento 3

#MAKE_EXE#

DADOS  SEGMENT 'DATA'
       mensagem DB "Olá, Mundo$"
DADOS  ENDS

PILHA  SEGMENT STACK 'STACK'
       DW 0100h DUP(?)
PILHA  ENDS

CODIGO SEGMENT 'CODE'
INICIO PROC FAR
  MOV     AX, DADOS
  MOV     DS, AX
  MOV     ES, AX

  MOV     DX, OFFSET mensagem

  MOV     AH, 09h
  INT     21h

  MOV     AH, 4Ch
  INT     21h
  RET

INICIO ENDP
CODIGO ENDS
END INICIO

```

A partir deste ponto com os comandos de menu **file/save** grave o programa anterior com o nome **SEGM3**, de forma que fique semelhante à Figura 10.11.

```

01 TITLE Teste de Segmento 3
02
03 #MAKE_EXE#
04
05 DADOS SEGMENT 'DATA'
06     mensagem DB "Olá, Mundo$"
07 DADOS ENDS
08
09 PILHA SEGMENT STACK 'STACK'
10    DW 0100h DUP(?)
11 PILHA ENDS
12
13 CODIGO SEGMENT 'CODE'
14     INICIO PROC FAR
15         MOV AX, DADOS
16         MOV DS, AX
17         MOV ES, AX
18
19         MOV DX, OFFSET mensagem
20
21         MOV AH, 09h
22         INT 21h
23
24         MOV AH, 4Ch
25         INT 21h
26         RET
27     INICIO ENDP
28 CODIGO ENDS
29 END INICIO

```

The screenshot shows the emu8086 assembly editor window with the file 'SEGM3.asm' open. The code is written in Intel Assembly language. It defines three segments: 'DADOS' (data), 'PILHA' (stack), and 'CODIGO' (code). The 'CODIGO' segment contains a procedure named 'INICIO' which prints the string 'Olá, Mundo\$' to the screen and then exits. The editor interface includes a menu bar with 'file', 'edit', 'bookmarks', 'Assembler', 'Emulator', 'math', 'ascii codes', and 'help'. Below the menu is a toolbar with icons for 'new', 'open', 'examples', 'save', 'compile', 'emulate', 'calculator', 'converter', 'options', 'help', and 'about'. The status bar at the bottom shows 'line: 19 col: 1' and 'drag a file here to open'.

Figura 10.11 - Programa SEGM3 na ferramenta emu8086.

O programa anterior está sendo escrito de forma mais simples que a versão obtida por meio do *template*. Nesta versão está se utilizando o estilo de escrita de códigos de programas em *Assembly* semelhante à estrutura utilizada pelos *assemblers* **TASM** (Turbo Assembler - Borland) e **MASM** (MS-Assembler - Microsoft).

A linha **01** do programa utiliza a diretiva **TITLE** para indicar a identificação de um nome interno para o programa. Essa diretiva não é exclusiva para programas com extensão **.EXE**, e pode também ser utilizada em programas com extensão **.COM**. Aliás, é uma forma elegante de identificar os programas na primeira linha de código.

Na linha **03** encontra-se a definição do identificador de tipo de arquivo a ser gerado. Neste caso, está sendo definido o identificador **#MAKE_EXE#**. Em programas com extensão do tipo **.COM** pode-se fazer uso do identificador **#MAKE_COM#** em conjunto com a diretiva **org 100h** para indicar o local onde os dados devem ser manipulados na memória. O identificador **#MAKE_EXE#**, para ser usado, exige que seja determinada a área de dados por meio das diretivas **SEGMENT** e **ENDS** com a definição do parâmetro '**DATA**'.

O programa **SEGM3** usa a definição das diretivas **SEGMENT** e **ENDS** de uma forma um pouco diferente da forma utilizada no programa **SEGM2**. Para tanto, observe a seguinte sintaxe:

```
[nome] SEGMENT <STACK> <parâmetro>
        [corpo do seguimento]
[nome] ENDS
```

O rótulo *nome* é obrigatório e o identificador **STACK** após **SEGMENT** é utilizado após a diretiva **SEGMENT** quando da definição da pilha. Após a definição do nome de identificação do segmento e da diretiva **SEGMENT**, torna-se necessário identificar o segmento em uso com um parâmetro de reconhecimento de seu tipo de operação entre os símbolos de apóstrofos, que pode ser:

- ◆ **DATA** - identifica a área de definição de dados do programa.
- ◆ **STACK** - define o tamanho da área de pilha.
- ◆ **CODE** - define o trecho de código do programa.

Os rótulos de definição e identificação dos segmentos de dados, pilha e código nas linhas **06-08**, **10-12** e **14-30** estão grafados em português. Quanto aos parâmetros entre os símbolos de apóstrofos, são de certa forma, opcionais, devendo ser mantidos na sua forma original para maior legibilidade do código.

Para verificar detalhes existentes, execute o programa com o comando de menu **assembler/compile and load in the emulator**. Em seguida na janela **emulator: SEGM3.exe** acione o comando de menu **view/stack**, de forma que a área de trabalho fique semelhante à Figura 10.12. Se necessitar, faça a distribuição das janelas e ajuste as suas barras de rolagem.

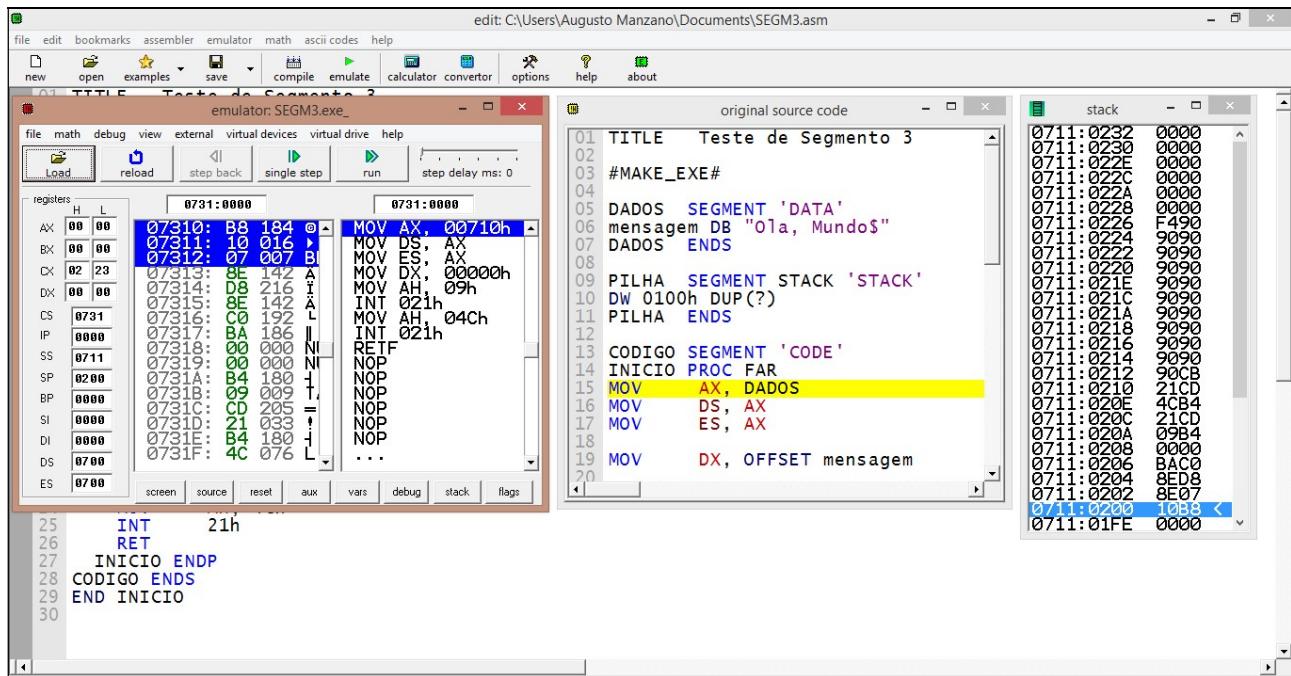


Figura 10.12 - Definição da área de trabalho para o programa SEGM3.

Antes de proceder à execução passo a passo do programa, é necessário levar em consideração algumas linhas de código que aparentemente, ao ser solicitada a execução do programa, não são executadas:

- ◆ O endereço do segmento de dados é definido na memória pelo trecho de código situado entre as linhas **06** e **08**. Os registradores de segmento **DS** e **ES** apontam o endereço dessa área, que neste caso marca o segmento **0700h**.
- ◆ O endereço do segmento de pilha é definido na memória pelo trecho de código situado entre as linhas **10** e **12**. O registrador de segmento **SS** aponta o endereço dessa área, que neste caso marca o segmento **0711h**, a partir do deslocamento **0200h**, como pode ser constatado no registrador de segmento **SP** (**SS:SP = 0711:0200**). A indicação **SS:SP** mostra o endereço de posição atual da pilha.
- ◆ O endereço de segmento de código é definido na memória pelo trecho de código situado entre as linhas **14** e **30**. O registrador de segmento **CS** aponta o endereço dessa área, que neste caso marca o endereço de segmento **0731**, a partir do endereço de deslocamento **0000h**, como pode ser constatado no registrador de segmento **IP** (**CS:IP = 0731:0000**).

É importante ressaltar mais uma vez que possivelmente no computador do leitor os valores de endereço de memória aqui apresentados sejam diferentes. É importante estar atento a esse detalhe.

À medida que o programa for executado, esses valores podem sofrer algumas alterações. Nesse momento inicial as áreas **memory** (área de apresentação do código na parte central da tela ao lado direito da área **registers**) e **disassembly** (área de apresentação do código na parte direita da tela) apresentam, respectivamente, o conteúdo existente na memória e a linha de código a ser executada.

Na tela **original source code** a linha de código do programa **MOV AX, DADOS** (referente à linha de código **15**) é definida para a área **disassembly** como **MOV AX, 00710h**, que está armazenado a partir do endereço de memória **0731:0000h**. O endereço **00710h** é o local onde se encontra a definição da área de dados do programa que está entre as linhas **06** e **08**. Mais adiante este valor será associado aos registradores de segmento **DS** e **ES**. Observe esses dados na Figura 10.13.

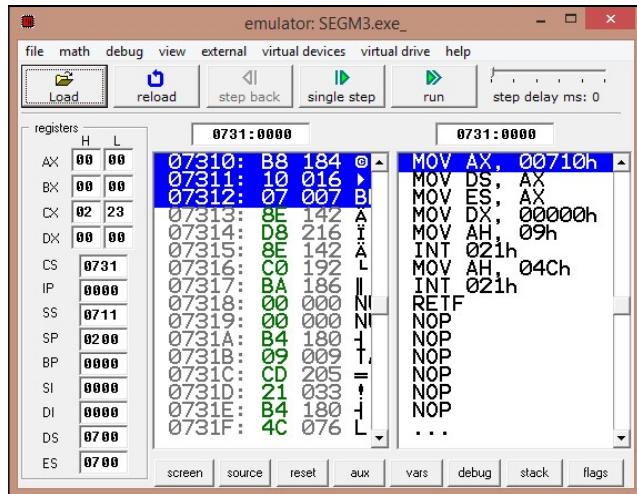


Figura 10.13 - Valores das posições de memória.

Acione a tecla de função <F8> (primeira vez), note que o registrador **AX** é armazenado com o valor **0710h** e o registrador de deslocamento **IP** passa a ter o valor **0003h**, como pode ser constatado na Figura 10.14. Observe que o valor de **AX** nesse momento é o que estava associado à instrução **MOV AX, 00710h** (**CS:IP = 0731:0003**) na área **disassemble**.

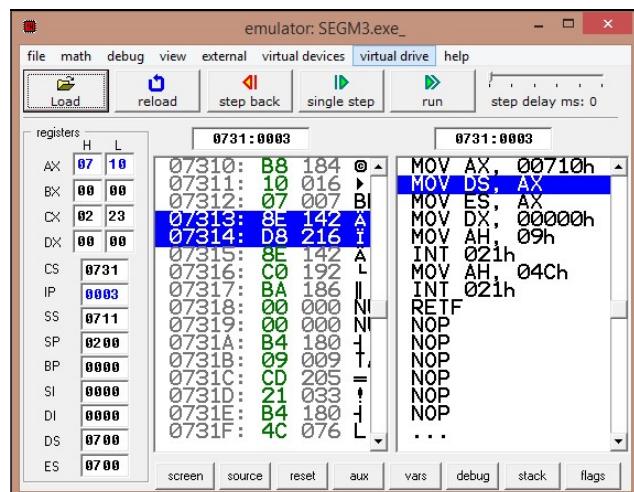


Figura 10.14 - Valores das posições de memória após <F8> - primeira vez.

É importante ressaltar que um registrador de segmento, como é o **DS**, não pode receber um endereço de forma direta, como ocorre com os registradores gerais. Por esta razão a próxima instrução a ser executada (**MOV DS, AX**, referente à linha 16) movimenta o valor do registrador **AX** para o registrador **DS** de forma indireta.

Se observar a área **memory** da Figura 10.14, verá que as instruções marcadas são **07313 8E 142 Ä** e **07314 D8 216 Í**. Os valores **07313h** e **07314h** correspondem ao endereço físico de memória. Os códigos **8E** e **D8** correspondem ao opcode de execução da instrução **MOV DS, AX**. Os valores **142** e **216** que aparecem do lado direito são a representação decimal dos valores **8E** e **D8** e os caracteres **Ä** e **Í** apresentados na quarta coluna da área **memory** correspondem aos caracteres ASCII referentes ao valor indicado na terceira e quarta colunas.

A área **memory** (parte central da janela) mostra a listagem do programa em linguagem de máquina e a área **disassemble** (lado direito) mostra o programa em linguagem Assembly. Nesse momento, se for acionado o botão **debug** (sexto botão na parte inferior da tela **emulator: SEGMENT3.exe_**), será apresentada a janela **Debug** como indicado na Figura 10.15.

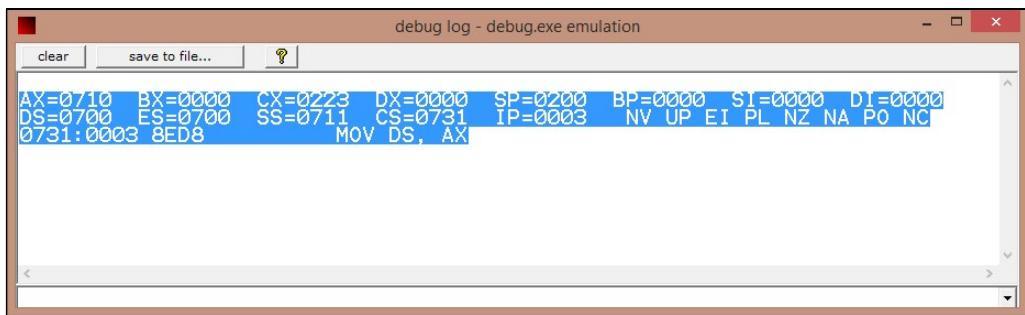


Figura 10.15 - Janela Debug.

A Figura 10.15 indica na terceira linha o local de memória onde se encontra a instrução, sendo o endereço de memória **0731:0003**, o código *opcode* **8ED8** e no extremo direito dessa linha é indicado a instrução em Assembly **MOV DS, AX**. Vale lembrar que o código de máquina (*opcode*) **8ED8** é a instrução equivalente em linguagem Assembly **MOV DS, AX**.

Olhe novamente a Figura 10.14 e observe que o próximo passo fará com que o valor do registrador IP que aponta o endereço **0003h** indique o endereço **0005h**. Haja vista a indicação do valor do endereço de memória **07315** estar logo abaixo da parte selecionada do código na área **memory**.

Para verificar a ocorrência de **MOV DS, AX**, acione pela segunda vez a tecla de função **<F8>** e observe as alterações no registrador de segmento **IP** de **0003h** que aponta para a próxima linha do programa, endereço de segmento **0005h**, e do registrador de deslocamento **DS** com o valor **0710** em que se encontra o início da área de segmento de dados, como pode ser verificado na Figura 10.16.

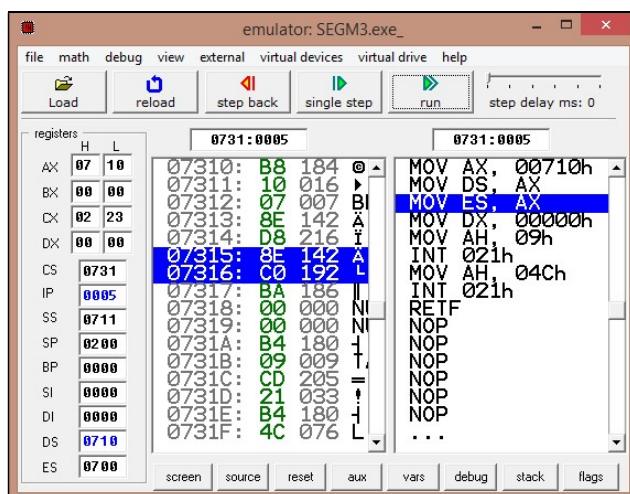


Figura 10.16 - Valores das posições de memória após **<F8>** - segunda vez.

É sabido que os registradores de segmento **DS** e **ES** em programas com extensão **.EXE** necessitam indicar o mesmo ponto de endereço. Os registradores em questão são diferentes, e por esta razão é necessário também movimentar o valor armazenado no registrador geral **AX** para o registrador de segmento **ES** pela linha de código **MOV ES, AX**.

Para certificar essa nova ação, acione a tecla de função **<F8>** pela terceira vez e observe a alteração dos valores do registrador de deslocamento **IP** para o valor **0007h** e do registrador de segmento **ES** com o valor **0710h**, como indica a Figura 10.17.

Os registradores de segmento **DS** e **ES**, após a terceira etapa de execução do programa, possuem o mesmo valor, ou seja, **0710h**, e assim devem estar. A título de curiosidade acione o botão **aux** e selecione na lista a opção **memory**. No campo ao lado esquerdo do botão **update** entre o valor **0710:0000** e acione com o ponteiro do **mouse** o botão **update**. Note que a mensagem **Ola, Mundo** está armazenada a partir do deslocamento **0000h** do segmento **0710h**. A Figura 10.18 mostra a ocorrência.

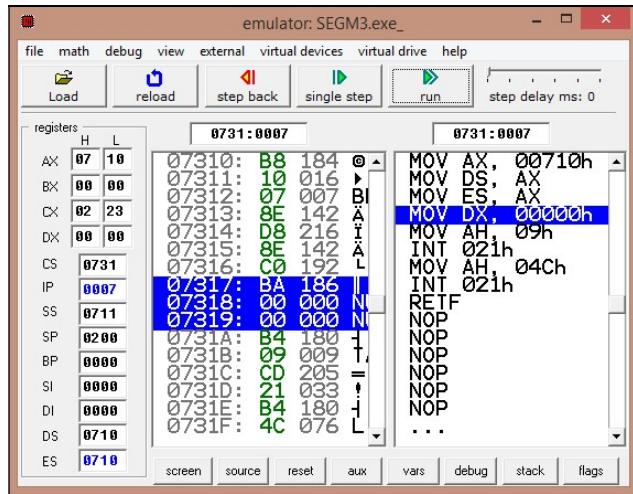


Figura 10.17 - Valores das posições de memória após <F8> - terceira vez.

A janela **random access memory** possibilita dois modos de visualização do estado da memória. Os dados podem ser vistos como tabela, semelhante à Figura 10.18 por meio da opção **table**, ou podem ser vistos como uma listagem por meio da opção **list** como mostra a Figura 10.19. Para fechar a janela, basta acionar o botão X no lado direito de sua barra de título.

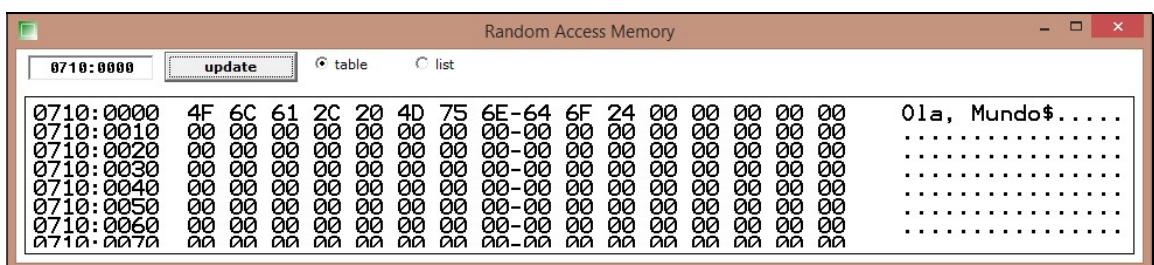


Figura 10.18 - Janela Ramdom Access Memory com a indicação do endereço de segmento de dados (tabela).

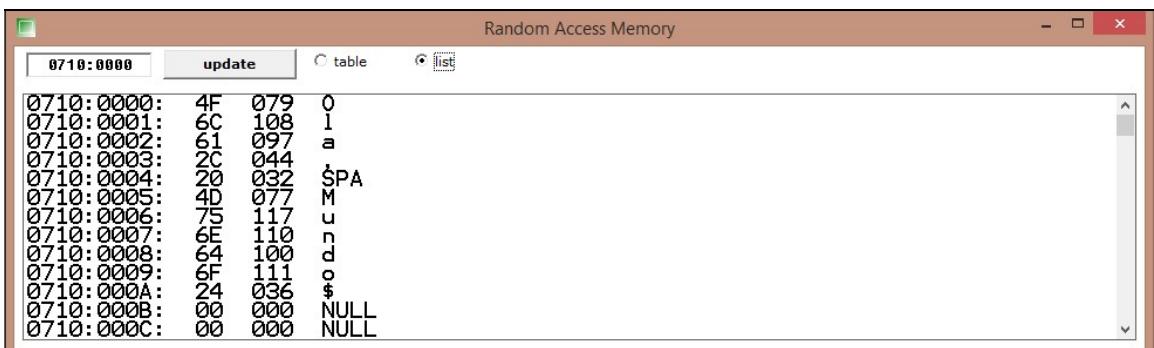


Figura 10.19 - Janela Ramdom Access Memory com a indicação do endereço de segmento de dados (lista).

A linha de código do programa **MOV DX, OFFSET mensagem** (linha 19) ou **MOV DX, 00000h** na área **disassembler** define o valor de deslocamento (offset) do início da área de dados (que é **00000h**) para o registrador geral **DX**. Isso faz com que o programa visualize o conteúdo na área de dados da memória. Na sequência acione a tecla de função <F8> pela quarta vez e observe que a única alteração ocorrida é com relação ao valor do registrador de deslocamento **IP**, que passa a ter o valor de deslocamento **000Ah**, como mostra a Figura 10.20.

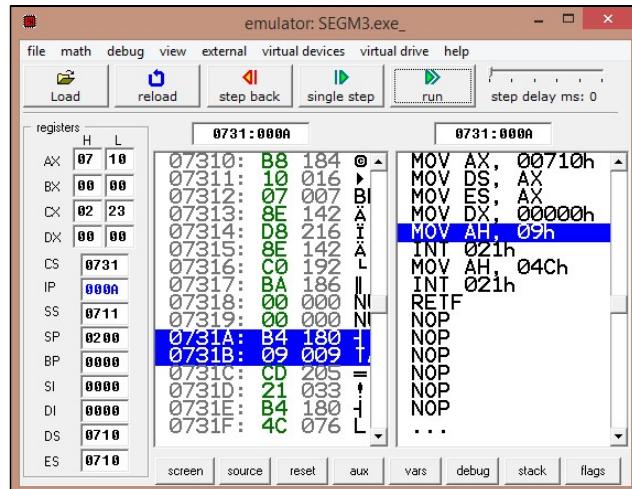


Figura 10.20 - Valores das posições de memória após <F8> - quarta vez.

A partir desse ponto o programa fará a apresentação da mensagem na tela do monitor de vídeo. Será executada a instrução **MOV AH, 09h** (linha 21) que movimenta para o registrador **AH** o valor de código **09h**, responsável por apresentar um *string* no monitor de vídeo quando da execução da instrução **INT 21h** (linha 22). Em seguida será efetuado o encerramento da execução do programa e o controle é devolvido para o sistema operacional. Isso é processado pelas linhas de instrução **MOV AH, 4Ch** (linha 24) e **INT 21h** (linha 25). Acesse a tecla <F8> até que sejam apresentadas a mensagem na tela e a caixa de mensagem **message**, quando então deve ser acionado o botão **OK**, em seguida encerre também a execução do modo de emulação (comando de menu **file/close the emulator** da janela **emulator: SEGMENT3.exe_**).

Em relação ao modo de finalização do programa e retorno para o sistema operacional, é possível usar as instruções **MOV AX, 4Ch** e **INT 21h** ou a instrução **INT 20h**. No entanto, isso é válido para programas com extensão **.COM**. Os programas com extensão **.EXE** não devem ser finalizados com a instrução **INT 20h**, pois a organização dos segmentos entre os dois tipos de programa é diferente e necessita ser tratada diferentemente (NORTON & SOSSA, 1988, p. 91). O modo **INT 20h** não funciona em programas com extensão **.EXE** pelo fato de as áreas de segmentos não serem contíguas.

A título de ilustração, a Figura 10.21 mostra um exemplo de esboço da memória de um processador 8086/8088 quando usada por programas com extensão **.COM** e com extensão **.EXE**.

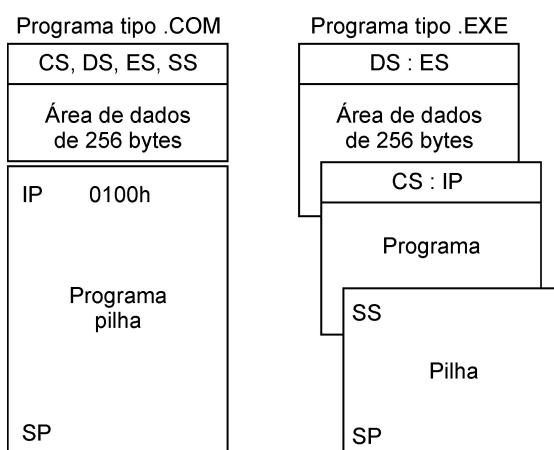


Figura 10.21 - Esboço de memória (adaptado de NORTON & SOSSA, 1988).

Dependendo do tipo de ferramenta *Assembler* em uso (que não é o caso da ferramenta **emu8086**), é necessário também utilizar a diretiva **ASSUME**, como nas ferramentas **TASM** e **MASM**. A diretiva **ASSUME** informa para o programa assemblador de forma explícita como os segmentos de memória devem ser utilizados pelo programa. Norton & Sossa (1988) afirmam que toda vez que ocorre a definição de um rótulo ou variável na memória, o programa assemblador lida com várias informações, tais como definição do nome, tipo, endereço do nome e segmento de memória no qual o nome está sendo definido, e a diretiva **ASSUME** está associada a essa última informação.

Apesar de não surtir o efeito na ferramenta **emu8086** (pois a diretiva **ASSUME** não é processada), é possível manter a definição da diretiva **ASSUME** no sentido de manter compatibilidade com os programas assembla-dores **TASM** e **MASM**.

Execute o comando de menu **file/new**, escolha qualquer uma das opções de *template* apresentadas, aione as teclas **<Ctrl> + <A>** para selecionar todo o texto existente e em seguida aione a tecla **** para limpar toda a área de texto. Na sequência escreva o código do programa a seguir:

```
TITLE Teste de Segmento 4

#MAKE_EXE#

DADOS SEGMENT 'DATA'
    mensagem DB "Ola, Mundo$"
DADOS ENDS

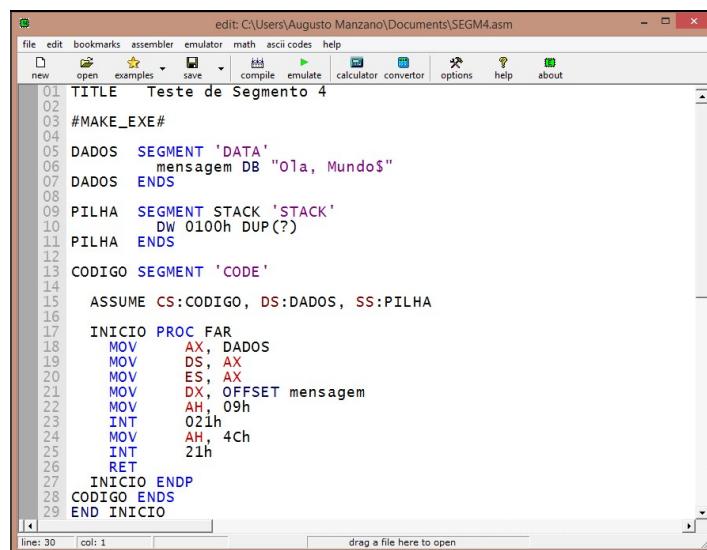
PILHA SEGMENT STACK 'STACK'
    DW 0100h DUP(?)
PILHA ENDS

CODIGO SEGMENT 'CODE'

ASSUME CS:CODIGO, DS:DADOS, SS:PILHA

INICIO PROC FAR
    MOV AX, DADOS
    MOV DS, AX
    MOV ES, AX
    MOV DX, OFFSET mensagem
    MOV AH, 09h
    INT 021h
    MOV AH, 4Ch
    INT 21h
    RET
INICIO ENDP
CODIGO ENDS
END INICIO
```

A partir deste ponto, com os comandos de menu **file/save**, grave o programa anterior com o nome **SEGM4** de forma que fique semelhante à Figura 10.22.



The screenshot shows the emu8086 assembly editor window with the following details:

- Title bar: edit: C:\Users\Augusto Manzano\Documents\SEGM4.asm
- Menu bar: file, edit, bookmarks, assembler, emulator, math, ascii codes, help
- Toolbar: new, open, examples, save, compile, emulate, calculator, converter, options, help, about
- Code area:

```
01 TITLE Teste de Segmento 4
02
03 #MAKE_EXE#
04
05 DADOS SEGMENT 'DATA'
06     mensagem DB "Ola, Mundo$"
07 DADOS ENDS
08
09 PILHA SEGMENT STACK 'STACK'
10     DW 0100h DUP(?)
11 PILHA ENDS
12
13 CODIGO SEGMENT 'CODE'
14
15 ASSUME CS:CODIGO, DS:DADOS, SS:PILHA
16
17 INICIO PROC FAR
18     MOV AX, DADOS
19     MOV DS, AX
20     MOV ES, AX
21     MOV DX, OFFSET mensagem
22     MOV AH, 09h
23     INT 021h
24     MOV AH, 4Ch
25     INT 21h
26     RET
27 INICIO ENDP
28 CODIGO ENDS
29 END INICIO
```
- Status bar: line: 30 | col: 1

Figura 10.22 - Programa SEGM4 na ferramenta emu8086.

Observação

Para maiores detalhes sobre compatibilidade da ferramenta **emu8086** com as ferramentas **TASM** e **MASM**, consulte a documentação *on-line* da ferramenta. Acesse **Help/MASM/TASM compatibility**.

Atente para a linha grafada em negrito (que será a linha 15) com o uso da diretiva **ASSUME**. Quando utilizada a diretiva **ASSUME**, ela permite associar de forma explícita cada rótulo definido ao seu respectivo segmento de memória.

10.3 - Bibliotecas em Assembly

Muitos recursos usados ou desenvolvidos para alguns programas tornam-se úteis para outros programas. O fato de copiar um determinado trecho de programa para ser usado em outro, apesar de válido, pode se tornar algo cansativo e propenso a erros.

Um recurso de programação bastante adequado é o desenvolvimento de uma biblioteca de recursos genéricos que possam ser utilizados em vários outros programas. Com essa atitude se economiza tempo de desenvolvimento e evita-se a ocorrência de muitos erros de escrita ou de lógica, pois se faz uso da filosofia de reaproveitamento de código.

A biblioteca em *Assembly* é um arquivo em formato texto, o qual possui uma sequência de rotinas de programa (podem ser procedimentos ou macros) que serão anexadas ao programa em execução. Para demonstrar esse recurso considere o seguinte código:

```
ESCREVA MACRO
PUSH AX
MOV AH, 9h
INT 21h
POP AX
ENDM
```

Execute no programa **emu8086** o comando de menu **file/new**, escolha qualquer uma das opções, acione simultaneamente as teclas **<Ctrl> + <A>** para selecionar o código existente e acione em seguida a tecla ****. Informe o código anterior, gravando-o com o nome **BIBLIO.inc**, de forma que fique semelhante à Figura 10.23.

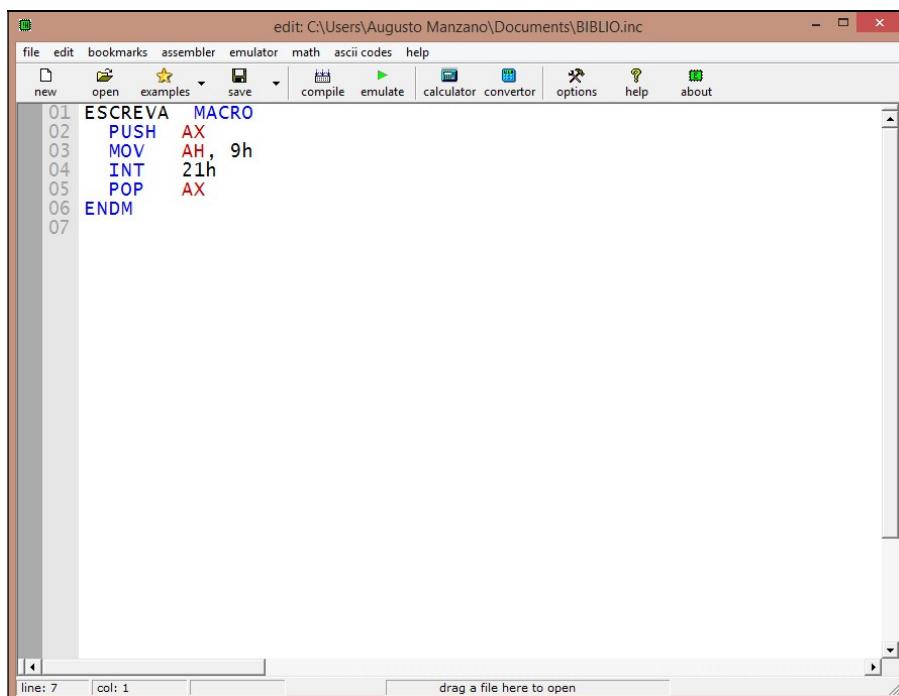


Figura 10.23 - Biblioteca BIBLIO na ferramenta emu8086.

Observação

A extensão .inc é apenas uma sugestão, pois o arquivo de biblioteca pode ter qualquer nome e extensão, desde que respeitados os limites da máquina.

Após a montagem da biblioteca, acesse **file/new**, escolha qualquer uma das opções, acione simultaneamente as teclas **<Ctrl> + <A>** para selecionar o código existente e acione em seguida a tecla ****. Na sequência informe este código:

TITLE Teste de Segmento 5

#MAKE_EXE#

INCLUDE 'biblio.inc'

DADOS SEGMENT 'DATA'
mensagem DB "Ola, Mundo\$"

DADOS ENDS

PILHA SEGMENT STACK 'STACK'
DW 0100h DUP(?)

PILHA ENDS

CODIGO SEGMENT 'CODE'
ASSUME CS:CODIGO, DS:DADOS, SS:PILHA
INICIO PROC FAR

MOV AX, DADOS
MOV DS, AX
MOV ES, AX
MOV DX, OFFSET mensagem

ESCREVA

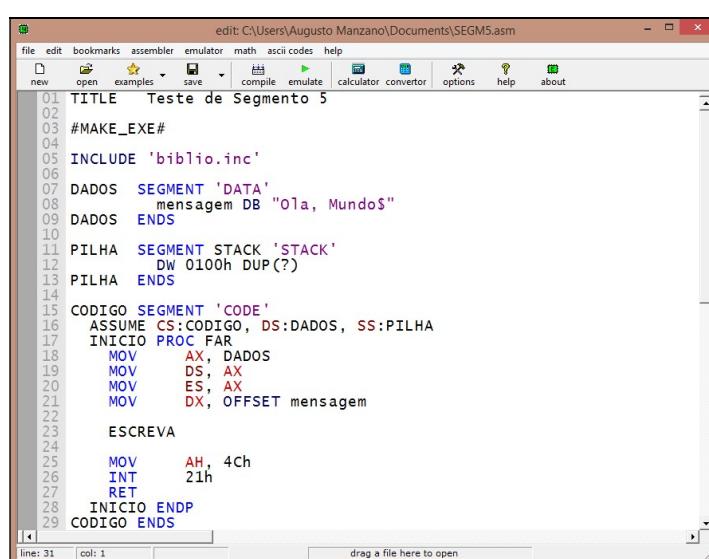
MOV AH, 4Ch
INT 21h
RET

INICIO ENDP

CODIGO ENDS

END INICIO

Execute os comandos de menu **file/save**, depois grave o programa anterior com o nome **SEGM5**, de forma que fique semelhante à Figura 10.24.



The screenshot shows the emu8086 assembly editor window with the following menu bar: file, edit, bookmarks, assembler, emulator, math, ascii codes, help. The toolbar includes new, open, example, save, compile, emulate, calculator, convertor, options, help, and about. The code area contains the assembly code for the program SEGM5, which includes segments for DATA, STACK, and CODE, along with procedures for outputting text and exiting via INT 21h. The status bar at the bottom shows 'line: 31 col: 1' and 'drag a file here to open'.

Figura 10.24 - Programa SEGM5 na ferramenta emu8086.

Atente para a linha **05** do programa na qual se encontra a diretiva **INCLUDE** seguida do nome do arquivo de biblioteca entre os símbolos de apóstrofo. A diretiva **INCLUDE** é responsável por vincular ao programa um arquivo de biblioteca externo, possibilitando ao programa ter acesso aos recursos definidos na biblioteca, como é demonstrado na linha **23** quando da chamada da macro **ESCREVA**, como se essa instrução fosse da própria linguagem de programação **Assembly**.

O recurso de biblioteca aumenta a eficácia e eficiência de um programador de computadores. Para ampliar um pouco mais a biblioteca **biblio.inc**, serão acrescidos alguns recursos como limpar tela, posicionar cursor em um determinado ponto da tela, colocar o teclado em espera e manipular o cursor.

Para abrir o arquivo de biblioteca na ferramenta **emu8086**, proceda à execução do comando **file/open**. Certifique-se de que está em aberto a pasta que foi usada para a gravação da biblioteca. Em seguida, forneça para o campo **Nome do arquivo** o nome **biblio.inc** e aione o botão **Abrir**.

Escreva a rotina de macro a seguir para o arquivo de biblioteca **BIBLIO.inc**. O código em questão deve ser informado a partir da linha de código **08**, logo abaixo da rotina **ESCREVA**, deixando uma linha em branco:

```
POSICAO MACRO linha, coluna
    PUSH AX
    PUSH DX
    MOV AH, linha
    MOV AL, coluna
    DEC AH
    DEC AL
    MOV DX, AX
    MOV AH, 02h
    INT 10h
    POP DX
    POP AX
ENDM
```

A macro **POSICAO** coloca o cursor em uma coordenada de tela fornecida como parâmetro de linha e de coluna. A tela do monitor de vídeo em modo texto é mapeada com 80 colunas (numeradas de **0d** até **79d**) e 24 linhas (numeradas de **0d** até **23d**). O parâmetro **linha** externamente aceita valores entre **1d** e **24d** e o parâmetro **coluna** externamente aceita valores entre **1d** e **80d**. Internamente os valores fornecidos pelos parâmetros **linha** e **coluna** são diminuídos em 1 pela ação da instrução **DEC**.

Na macro **POSICAO**, antes de executar qualquer ação, está sendo utilizada a instrução **PUSH** para guardar na pilha os valores atuais dos registradores gerais **AX** e **DX**. Isso se faz necessário porque a macro **POSICAO** é um elemento externo ao programa em operação. Essa macro utiliza os registradores gerais **AX** e **DX** para sua ação. Assim sendo, é melhor garantir que se há algum valor importante nos registradores gerais, ele fica armazenado na pilha e pode ser recuperado pela instrução **POP**.

A instrução **PUSH** tem por finalidade armazenar (escrever) na pilha um dado do tipo *word*. Essa ocorrência é utilizada quando se deseja armazenar na pilha temporariamente um dado a ser utilizado em seguida, como se esse efeito fosse o de passagem de parâmetro encontrado em linguagens de alto nível. As operações de escrita na pilha não podem ser realizadas com a definição de valores imediatos. A instrução **PUSH**, quando em operação, efetua o decremento de **02h** sobre o registrador **SP**, além de colocar o operando **MSB** (mais significativo) na posição **SS:[SP+1]** e o operando **LSB** (menos significativo) na posição **SS:[SP]**.

A instrução **POP** retira um dado do tipo *word* da pilha. As características aplicadas à instrução **POP** são as mesmas aplicadas à instrução **PUSH**. A instrução **POP**, quando em operação, efetua o incremento de **02h** sobre o registrador **SP**, além de receber para o operando **MSB** (mais significativo) o conteúdo da posição **SS:[SP+1]** e para o operando **LSB** (menos significativo) receber o conteúdo da posição **SS:[SP]**.

Após armazenar os valores dos registradores gerais **AX** e **DX**, a rotina movimenta os valores dos parâmetros **linha** e **coluna**, respectivamente, para os registradores **AH** (mais significativo) e **AL** (menos significativo). Depois diminuem em **1** os valores dos registradores **AH** e **AL**, ajustando o valor real da posição do cursor, e movimenta o conjunto de valores do registrador geral **AX** para o registrador geral **DX**.

É necessário que o registrador geral **DX** tenha na sua parte mais significativa (**DH**) o valor do parâmetro **linha** e na sua parte menos significativa (**DL**) o valor do parâmetro **coluna**. Depois o valor da função **02h** (responsável por posicionar o

cursor) é armazenado na parte mais significativa (**AH**) do registrador geral **AX** e executa-se a ação da instrução **INT 10h** para que o cursor seja propriamente posicionado.

Informe a rotina de macro a seguir para o arquivo de biblioteca **BIBLIO.inc**. O código em questão deve ser informado a partir da linha de código **22**, logo abaixo da rotina **POSICAO**, deixando uma linha em branco:

```
TECLE MACRO
    PUSH AX
    MOV AH, 00h
    INT 16h
    POP AX
ENDM
```

A macro **TECLE** coloca o teclado em modo de espera e aguarda até que alguma tecla seja acionada. No momento em que alguma tecla é acionada, a rotina é encerrada. Na macro **TECLE**, antes de executar qualquer ação, está sendo utilizada a instrução **PUSH** para guardar na pilha os valores atuais do registrador geral **AX**. Valor que será recuperado pela instrução **POP**.

Após armazenar o valor do registrador geral **AX** na pilha, o programa armazena o valor da função **00h** na parte mais significativa (**AH**) do registrador geral **AX** e executa a ação da instrução **INT 16h** para que o teclado entre em modo de espera. No momento em que uma tecla é acionada, o seu valor ASCII é armazenado no registrador **AH** e a rotina é encerrada.

Informe a rotina de macro a seguir para o arquivo de biblioteca **BIBLIO.inc**. O código em questão deve ser informado a partir da linha de código **29**, logo abaixo da rotina **TECLE**, deixando uma linha em branco:

```
CURSORG MACRO
    PUSH AX
    PUSH CX
    MOV AX, 0100h
    MOV CX, 000Ah
    INT 10h
    POP CX
    POP AX
ENDM
```

Na macro **CURSORG**, antes de executar qualquer ação, está sendo utilizada a instrução **PUSH** para guardar na pilha os valores atuais dos registradores gerais **AX** e **CX**, os quais podem, posteriormente, ser recuperados pela instrução **POP**.

Após armazenar os valores dos registradores gerais **AX** e **CX**, a rotina movimenta os valores **0100h** e **000Ah** para os registradores gerais **AX** e **CX** e executa-se a ação da instrução **INT 10h** para que o cursor seja apresentado em um formato maior que o tradicional. O valor **0100h** permite manipular o cursor e o valor **000Ah** permite mudar seu tamanho para grande após a chamada da interrupção **10h**.

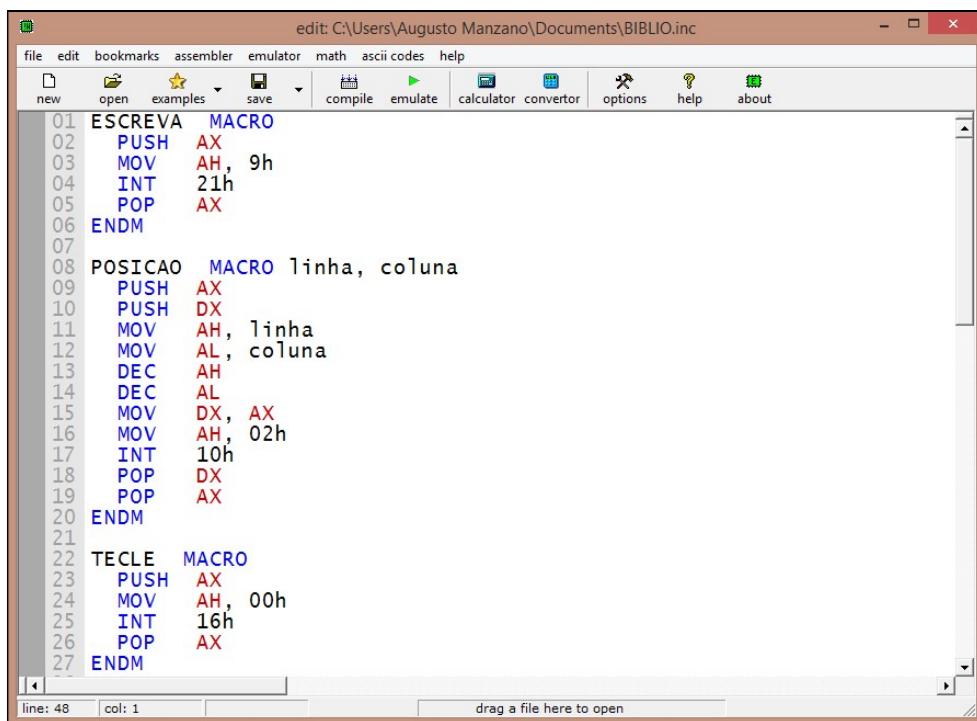
Informe a rotina de macro a seguir para o arquivo de biblioteca **BIBLIO.inc**. O código em questão deve ser informado a partir da linha de código **39**, logo abaixo da rotina **CURSORG**, deixando uma linha em branco:

```
CURSORP MACRO
    PUSH AX
    PUSH CX
    MOV AX, 0100h
    MOV CX, 0506h
    INT 10h
    POP AX
    POP CX
ENDM
```

Na macro **CURSORP**, antes de executar qualquer ação, está sendo utilizada a instrução **PUSH** para guardar na pilha os valores atuais dos registradores gerais **AX** e **CX**, os quais podem, posteriormente, ser recuperados pela instrução **POP**.

Após armazenar os valores dos registradores gerais **AX** e **CX**, a rotina movimenta os valores **0100h** e **0506h** para os registradores gerais **AX** e **CX** e executa-se a ação da instrução **INT 010h** para que o cursor seja apresentado em seu formato tradicional. O valor **0100h** permite manipular o cursor e o valor **0506h** permite mudar seu tamanho para pequeno após a chamada da interrupção **10h**.

Defina todas as macros anteriores e grave o arquivo **BIBLIO.inc**. As Figuras 10.25 e 10.26 apresentam, respectivamente, o trecho de rotinas de macro a partir da linha **08** e a partir da linha **39**

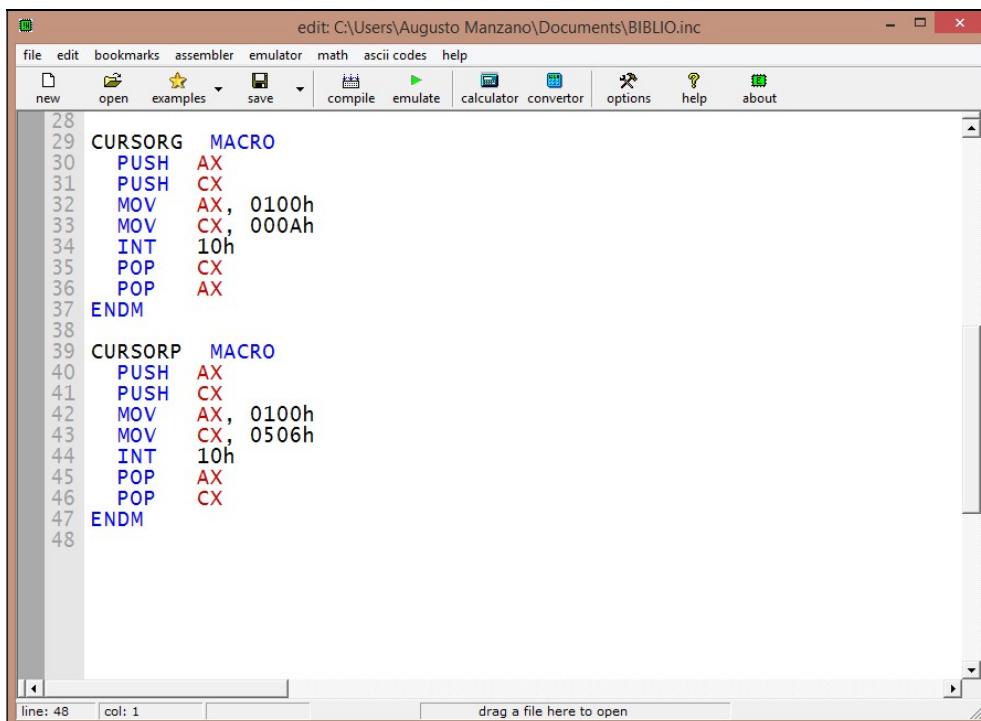


```

edit: C:\Users\Augusto Manzano\Documents\BIBLIO.inc
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about
01 ESCREVA MACRO
02 PUSH AX
03 MOV AH, 9h
04 INT 21h
05 POP AX
06 ENDM
07
08 POSICAO MACRO linha, coluna
09 PUSH AX
10 PUSH DX
11 MOV AH, linha
12 MOV AL, coluna
13 DEC AH
14 DEC AL
15 MOV DX, AX
16 MOV AH, 02h
17 INT 10h
18 POP DX
19 POP AX
20 ENDM
21
22 TECLE MACRO
23 PUSH AX
24 MOV AH, 00h
25 INT 16h
26 POP AX
27 ENDM
line: 48 col: 1 drag a file here to open

```

Figura 10.25 - Biblioteca BIBLIO na ferramenta emu8086 - a partir linha 08.



```

edit: C:\Users\Augusto Manzano\Documents\BIBLIO.inc
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about
28 CURSORG MACRO
29 PUSH AX
30 PUSH CX
31 MOV AX, 0100h
32 MOV CX, 000Ah
33 INT 10h
34 POP CX
35 POP AX
36 ENDM
37
38 CURSОРР MACRO
39 PUSH AX
40 PUSH CX
41 MOV AX, 0100h
42 MOV CX, 0506h
43 INT 10h
44 POP AX
45 POP CX
46 ENDM
47
48
line: 48 col: 1 drag a file here to open

```

Figura 10.26 - Biblioteca BIBLIO na ferramenta emu8086 - a partir linha 39.

Para fazer um teste de execução do conjunto de macros armazenadas em uma biblioteca, execute o comando de menu **file/new** e escolha qualquer uma das opções. Acione simultaneamente as teclas **<Ctrl> + <A>** para selecionar o texto atual, em seguida acione a tecla **** para remover o texto existente e na área de edição em branco e codifique o programa seguinte:

```
TITLE Teste de Segmento 6

#MAKE_EXE#

INCLUDE 'biblio.inc'

DADOS SEGMENT 'DATA'
    msg0 DB "Tecle algo para prosseguir", 0Dh, 0Ah, 24h
    msg1 DB "Ola, Mundo 1", 0Dh, 0Ah, 24h
    msg2 DB "Ola, Mundo 2", 0Dh, 0Ah, 24h
    msg3 DB "Ola, Mundo 3", 0Dh, 0Ah, 24h
DADOS ENDS

PILHA SEGMENT STACK 'STACK'
    DW 0100h DUP(?)
PILHA ENDS

CODIGO SEGMENT 'CODE'
ASSUME CS:CODIGO, DS:DADOS, SS:PILHA
INICIO PROC FAR
    MOV AX, DADOS
    MOV DS, AX
    MOV ES, AX

    POSICAO 03, 05
    MOV DX, OFFSET msg1
    ESCREVA

    POSICAO 20, 01
    MOV DX, OFFSET msg0
    ESCREVA
    TECLE

    POSICAO 07, 10
    MOV DX, OFFSET msg2
    ESCREVA

    POSICAO 22, 01
    MOV DX, OFFSET msg0
    ESCREVA
    TECLE

    POSICAO 11, 40
    MOV DX, OFFSET msg3
    ESCREVA

    POSICAO 24, 01
    MOV DX, OFFSET msg0
    ESCREVA
    CURSORG
    TECLE
    CURSORP

    MOV AH, 4Ch
    INT 21h
    RET
```

```

INICIO ENDP
CODIGO ENDS
END INICIO

```

Execute os comandos de menu **file/save** e grave o programa com o nome **SEGM6.asm**. A Figura 10.27 apresenta a imagem do resultado da execução do programa.



Figura 10.27 - Resultado da execução do programa.

10.4 - Apresentação de Valores Negativos

Todos os programas utilizados anteriormente manipularam apenas valores positivos. Quanto aos valores negativos, são estes calculados e armazenados na memória com o complemento por dois.

Neste tópico, utilizando o comando **NEG** (Negate), é possível transformar valores negativos em positivos e vice-versa. O comando **NEG** afeta o comportamento de alguns registradores de estado, destacando-se entre eles o **SF**.

Quando uma operação de subtração ocorre entre dois valores (comando **SUB**), o registrador de estado **SF** é afetado. Se for executada uma operação de **9 – 7**, o resultado obtido na memória será **2** e o registrador de estado **SF** será sinalizado com valor **0** (sem efeito). Se for realizada a operação **7 – 9**, o resultado obtido na memória será **FEh** (tipo byte) ou **FFFFEh** (tipo word) e o registrador de estado **SF** será sinalizado com o valor **1**, indicando que o valor armazenado na memória é negativo com base no complemento por dois.

O valor **FEh** pode tanto ser **-2d** como o valor **254d** (algo semelhante a esse efeito havia sido apresentado em capítulos anteriores). A forma de interpretação do valor vai depender da análise do registrador de estado **SF**. Se quiser interpretar o valor **FEh** como **-2d**, é necessário considerar o valor **1** do registrador de estado **SF**. Caso contrário, se não levar em conta o registrador de estado **SF** estar com **1** ou com **0**, o valor **FEh** pode ser interpretado como **254d**.

Tome como base um programa que faça a subtração de dois valores numéricos de apenas um dígito lidos no teclado. Observe o código do programa seguinte:

```

TITLE Teste de Segmento 7

#MAKE_EXE#
DADOS SEGMENT 'DATA'
msg1 DB 'Entre valor 1 (de 0 a 9) .....: ', 24h
msg2 DB 0Dh, 0Ah, 'Entre valor 2 (de 0 a 9) .....: ', 24h
msg3 DB 0Dh, 0Ah, 'Resultado .....: ', 24h
msg4 DB 0Dh, 0Ah, 'Valor invalido', 24h
DADOS ENDS

```

```

PILHA SEGMENT STACK 'STACK'
DW 0100h DUP(?)
PILHA ENDS

CODIGO SEGMENT 'CODE'
ASSUME CS:CODIGO, DS:DADOS, SS:PILHA
INICIO PROC FAR
    MOV AX, DADOS
    MOV DS, AX
    MOV ES, AX

    MOV DX, OFFSET msg1
    MSG
    CALL entrada
    MOV BH, AL

    MOV DX, OFFSET msg2
    MSG
    CALL entrada
    MOV BL, AL

    MOV DX, OFFSET msg3
    MSG
    SUB BH, BL
    JS negativo
    JGE positivo
negativo:
    NEG BH
    MOV AL, 2Dh
    MOV AH, 0Eh
    INT 10h
    JMP mostra

positivo:
    JMP mostra

mostra:
    MOV AL, BH
    MOV DL, AL
    ADD AL, 30h
    MOV AH, 0Eh
    INT 10h
    FIM
    RET
INICIO ENDP
CODIGO ENDS

fim MACRO
    MOV AH, 4Ch
    INT 21h
ENDM

msg MACRO
    MOV AH, 09h
    INT 21h
ENDM

entrada PROC NEAR
    MOV AH, 01h
    INT 21h

```

```

    CMP     AL, 30h
    JL      erro
    CMP     AL, 40h
    JGE    erro
    JMP    fim_validacao
erro:
    LEA     DX, msg4
    MSG
    FIM
fim_validacao:
    SUB    AL, 30h
    RET
entrada ENDP

```

END INICIO

Acione simultaneamente as teclas **<Ctrl> + <A>** para selecionar o texto atual, em seguida acione a tecla **** para remover o texto existente na área de edição em branco. Codifique o programa anterior e com os comandos de menu **file/save** grave-o com o nome **SEGM7**. As Figuras 10.28 (resposta positiva) e 10.29 (resposta negativa) apresentam, respectivamente, o resultado para os valores **9 e 2; 2 e 9**.

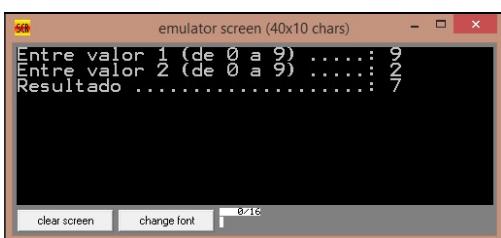


Figura 10.28 - Resultado positivo.

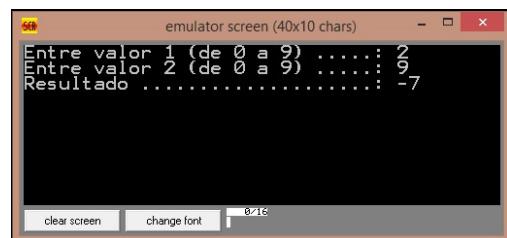


Figura 10.29 - Resultado negativo.

Como vários detalhes do programa são conhecidos e foram anteriormente explicados, eles não necessitam ser revistos, mas é pertinente considerar os detalhes ainda desconhecidos. As Figuras 10.30, 10.31 e 10.32 apresentam a imagem do programa completo.

```

01 TITLE Teste de Segmento 7
02
03 #MAKE_EXE#
04
05 DADOS SEGMENT 'DATA'
06 msg1 DB 'Entre valor 1 (de 0 a 9) .....: ', 24h
07 msg2 DB 0Dh, 0Ah, 'Entre valor 2 (de 0 a 9) .....: ', 24h
08 msg3 DB 0Dh, 0Ah, 'Resultado .....: ', 24h
09 msg4 DB 0Dh, 0Ah, 'Valor invalido', 24h
10 DADOS ENDS
11
12 PILHA SEGMENT STACK 'STACK'
13 DW 0100h DUP(?)
14 PILHA ENDS
15
16 CODIGO SEGMENT 'CODE'
17 ASSUME CS:CODIGO, DS:DADOS, SS:PILHA
18 INICIO PROC FAR
19     MOV AX, DADOS
20     MOV DS, AX
21     MOV ES, AX
22
23     MOV DX, OFFSET msg1
24     MSG
25     CALL entrada
26     MOV BH, AL
27
28     MOV DX, OFFSET msg2
29     MSG
30     CALL entrada
31     MOV BL, AL

```

Figura 10.30 - Programa SEGM7 na ferramenta emu8086 - parte 1 (linhas 01 até 31).

```

32      MOV      DX, OFFSET msg3
33      MSG
34      SUB      BH, BL
35      JS       negativo
36      JGE     positivo
37      positivo:
38      negativo:
39      NEG      BH
40      MOV      AL, 2Dh
41      MOV      AH, 0Eh
42      INT     10h
43      JMP     mostra
44
45      positivo:
46      JMP     mostra
47
48      mostra:
49      MOV      AL, BH
50      MOV      DL, AL
51      ADD      AL, 30h
52      MOV      AH, 0Eh
53      INT     10h
54      FIM
55      RET
56      INICIO ENDP
57      CODIGO ENDS
58
59      fim MACRO
60      MOV      AH, 4Ch
61      INT     21h
62      ENDM

```

Figura 10.31 - Programa SEGM7 na ferramenta emu8086 - parte 2 (linhas 32 até 62).

```

63      msg MACRO
64      MOV      AH, 09h
65      INT     21h
66      ENDM
67
68      entrada PROC NEAR
69      MOV      AH, 01h
70      INT     21h
71      CMP      AL, 30h
72      JL      erro
73      CMP      AL, 40h
74      JL      erro
75      JGE     fim_validacao
76      JMP     fim_validacao
77      erro:
78      LEA      DX, msg4
79      MSG
80      FIM
81      fim_validacao:
82      SUB      AL, 30h
83      RET
84      entrada ENDP
85
86      END INICIO
87

```

Figura 10.32 - Programa SEGM7 na ferramenta emu8086 - parte 3 (linhas 63 até 87).

Na Figura 10.31 considere a explanação da ação de algumas linhas do programa. Na linha 35 ocorre a instrução **SUB BH, BL**. Nesse caso, se o valor do registrador mais significativo **BH** for menor que o valor do registrador menos significativo **BL**, ocorre a alteração do registrador de estado **SF** para 1, indicando que o valor armazenado no registrador mais significativo é negativo.

Na linha 36 na instrução **JS negativo**, verifica-se se o valor do registrador de estado **SF** é 1. Nesse caso, se **SF** for 1, o programa desvia para o trecho **negativo** definido a partir da linha 39.

Na linha **37** na instrução **JGE positivo**, verifica-se se o valor do registrador mais significativo **BH** é igual ou maior que o registrador menos significativo **BL**. Caso a instrução **JGE** resulte um valor lógico verdadeiro, o programa desvia para o trecho **positivo** definido a partir da linha **46**.

O trecho de código a partir da linha **46** apresenta o valor armazenado em memória. Para estar nesse trecho, a instrução **JS negativo** da linha **36** foi executada, pois o registrador de estado **SF** está com o valor **1**. Primeiramente na linha **40** é utilizada a instrução **NEG BH**, que transforma o valor negativo (complemento de dois) em seu equivalente positivo. A instrução **NEG** muda o valor de positivo para negativo e de negativo para positivo quando for necessário, alterando o valor armazenado na memória, utilizando o complemento por dois. Nesse caso, o valor do registrador mais significativo **BH** é **F9h**. A instrução **NEG BH** da linha **40** transforma o valor **F9h** no valor **7**.

A linha **41** movimenta o valor **2Dh** para o registrador menos significativo **AL** que é o código ASCII do símbolo de sinal negativo, em seguida as linhas **41** e **43** apresentam o sinal na tela do monitor de vídeo. Na sequência (linha **44**), o programa chama o trecho identificado pelo rótulo **mostra** e apresenta o valor numérico armazenado na memória, que nesse caso será **7**.

10.5 - Biblioteca de Funções Externas

O programa **emu8086** oferece uma biblioteca de funções externas genéricas comuns e muito utilizadas em vários programas a serem desenvolvidos em linguagem de programação *Assembly 80886*.

A biblioteca padrão do programa denominada **emu8086.inc** é um arquivo de programa em formato texto com a definição interna de *macros* e *procedimentos*. Para o efetivo uso da biblioteca dentro de um programa em desenvolvimento, é necessário usar a diretiva **INCLUDE** (como demonstrado anteriormente).

O arquivo de biblioteca pode ter qualquer tipo de extensão. No caso do programa **emu8086**, é aconselhável que o arquivo de biblioteca possua a extensão **.inc** e de preferência esteja gravado na pasta **Inc** dentro do diretório do programa **C:\emu\binaries\inc**. Se não estiver no diretório **inc**, o arquivo de biblioteca deve estar na mesma pasta em que o programa em desenvolvimento está gravado.

Neste tópico são estudadas as funções existentes na biblioteca **emu8086.inc**. Segundo o próprio Alexander Popov, desenvolvedor do produto, é possível que o usuário da ferramenta (aluno) não entenda todo o conteúdo e recursos disponibilizados pela biblioteca. No entanto, é importante saber o que a biblioteca faz.

O autor da ferramenta acrescenta que para utilizar qualquer uma das funções disponibilizadas é necessário colocar no início do programa-fonte em desenvolvimento a linha **INCLUDE 'emu8086.inc'**.

O arquivo de biblioteca **emu8086.inc** do programa **emu8086** tem funções agrupadas em duas categorias operacionais, sendo funções de macro e funções de procedimento.

As funções de macro são as seguintes:

- ◆ **PUTC caractere** - macro com a definição de um parâmetro, que apresenta como saída o caractere informado como parâmetro na posição atual do cursor.
- ◆ **PRINT mensagem** - macro com a definição de um parâmetro, que apresenta uma mensagem na tela, mantendo o cursor na mesma linha de apresentação.
- ◆ **PRINTN mensagem** - macro com a definição de um parâmetro, que mostra uma mensagem na tela. Após a apresentação o cursor é posicionado na próxima linha.
- ◆ **CURSOROFF** - macro sem a definição de parâmetros, que desabilita a apresentação do cursor na tela.
- ◆ **CURSORON** - macro sem a definição de parâmetros, que habilita a apresentação do cursor na tela.
- ◆ **GOTOXY coluna, linha** - macro com a definição de dois parâmetros, que coloca o cursor em uma determinada coordenada de tela.

As funções de procedimento são as seguintes:

- ◆ **SCAN_NUM** - obtém um valor numérico com mais de um dígito, podendo ser positivo ou negativo. Antes da diretiva **END** é necessário utilizar a declaração **DEFINE_SCAN_NUM**.
- ◆ **PRINT_STRING** - procedimento usado para imprimir mensagem terminada com caractere nulo na posição atual do cursor. Antes da diretiva **END** é necessário utilizar a declaração **DEFINE_PRINT_STRING**.

- ◆ **PTHIS** - utilizado para imprimir mensagem terminada com caractere nulo na posição atual do cursor, semelhante ao procedimento **PRINT_STRING**, mas com a diferença de receber o endereço da sequência de caracteres que formam a mensagem a partir da pilha. Para sequências de caracteres terminadas em zero, é necessário que a definição da mensagem ocorra logo após a chamada do procedimento. Antes da diretiva **END** é necessário utilizar a declaração **DEFINE_PTHIS**.
- ◆ **GET_STRING** - lê do teclado uma sequência de caracteres terminada em nulo. Esse procedimento é interrompido quando utilizada a tecla <Enter>. Antes da diretiva **END** é necessário utilizar a declaração **DEFINE_GET_STRING**.
- ◆ **CLEAR_SCREEN** - limpa a tela e posiciona o cursor no seu topo esquerdo. Antes da diretiva **END** é necessário utilizar a declaração **DEFINE_CLEAR_SCREEN**.
- ◆ **PRINT_NUM** - apresenta um valor numérico negativo. Antes da diretiva **END** é necessário utilizar as declarações **DEFINE_PRINT_NUM** e **DEFINE_PRINT_NUM_UNS**.
- ◆ **PRINT_NUM_UNS** - apresenta um valor numérico positivo. Antes da diretiva **END** é necessário utilizar as declarações **DEFINE_PRINT_NUM** e **DEFINE_PRINT_NUM_UNS**.

Ao fazer a compilação de um programa-fonte com a chamada de uma determinada biblioteca, a ferramenta **emu8086** localiza a macro ou procedimento na biblioteca e a executa no programa-fonte. As estruturas de rotinas macros e procedimentos possuem vantagens e desvantagens.

No caso de macros ocorre a substituição no programa-fonte do código da macro pelo código real. Assim sendo, o programa-fonte torna-se maior. Caso o uso de uma macro seja muito frequente, o programa pode também ficar enorme. Nesse caso é aconselhável utilizar procedimentos. No entanto, as macros possibilitam o uso de parâmetros externos e os procedimentos não.

No caso de usar procedimentos, o compilador processa as declarações sem copiá-las para o código-fonte. Nesse caso, o compilador, quando encontra a instrução **CALL**, substitui o nome de procedimento com o seu endereço de memória, onde o procedimento foi declarado, efetua a sua ação e retorna após sua chamada quando encontra a instrução **RET**.

Para exemplificar o uso da biblioteca **emu8086.inc**, considere o programa seguinte:

```

;*****
;*      Programa: BIBLIOT1.ASM      *
;*****

INCLUDE 'emu8086.inc'

#MAKE_EXE#

.DATA
msg1 DB 'Entrada e Apresentacao', 0d
msg2 DB 'Tecle algo para continuar...', 0d
msg3 DB 'Entre um valor numerico ...:', 0d

.CODE

; Desabilita cursor
CURSOROFF

; Poe cursor na Coluna = 30 / Linha = 12
; armazena msg1 em SI
; apresenta o conteúdo de msg1 na tela

GOTOXY    29d, 11d
PRINT     'Programa para Teste'

; Poe cursor na Coluna = 01 / Linha = 24
; armazena msg2 em SI
; apresenta o conteúdo de msg2 na tela

GOTOXY    00d, 23d
LEA       SI, msg2

```

```

CALL      print_string
; Aguarda que algo seja teclado

MOV      AH, 00h
INT      16h

; Habilita cursor

CURSORON

; Limpa a tela

CALL      clear_screen

; Poe cursor na Coluna = 29 / Linha = 01
; armazena msg1 em SI
; apresenta o conteudo de msg1 na tela

GOTOXY    28d, 00d
LEA      SI, msg1
CALL      print_string

; Poe cursor na Coluna = 01 / Linha = 05
; armazena msg3 em SI
; apresenta o conteudo de msg3 na tela

GOTOXY    00d, 04d
LEA      SI, msg3
CALL      print_string

; Efetua a entrada de um valor numerico em CX

CALL      scan_num

; Transfere o valor de CX para AX

MOV      AX, CX

; Apresenta mensagem de saida

CALL      pthis
        DB 0Dh, 0Ah, 'Foi fornecido o valor .....: ', 0d

; Apresenta o valor armazenado em AX
CALL      print_num

; Finaliza programa

INT      20h

; Rotulos de definicao das operacoes da biblioteca
DEFINE_SCAN_NUM
DEFINE_PRINT_STRING
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS
DEFINE_PTHIS
DEFINE_CLEAR_SCREEN

```

END

Execute no programa **emu8086** o comando de menu **file/new/com template** e a partir da linha **06** do editor de texto escreva o programa anterior, gravando-o com o nome **BIBLIOT1.asm**.

Execute no programa **emu8086** o comando de menu **file/new/com template**, acione as teclas de atalho **<Ctrl> + <A>** do editor de texto e escreva o programa anterior, gravando-o por meio dos comandos de menu **File/Save** com o nome **BIBLIOT1**.

Execute o programa com o comando de menu **assembler/compile and load in the emulator**. Acione o botão **run** ou a tecla de função **<F9>** para executar o programa e observar todas as ocorrências de funcionalidade inseridas pela biblioteca **emu8086.inc**.

A seguir apresenta-se uma rápida descrição da funcionalidade de cada trecho do programa. Explicações mais detalhadas sobre a funcionalidade de cada recurso da biblioteca encontram-se no próximo tópico.

```
INCLUDE 'emu8086.inc'
```

Neste trecho o código faz referência ao uso da biblioteca **emu8086.inc**. Se essa linha for omitida, ocorrem erros de compilação em todos os trechos em que houver a menção de qualquer um dos recursos externos em uso.

```
CURSOROFF
```

O trecho de programa com a chamada da macro anterior desabilita o cursor. Desta forma, o cursor não será apresentado.

```
; Poe cursor na Coluna = 30 / Linha = 12
; armazena msg1 em SI
; apresenta o conteúdo de msg1 na tela

GOTOXY    29d, 11d
PRINT      'Programa para Teste'
```

As duas linhas indicam o uso das macros que posicionam o cursor na tela do monitor de vídeo e apresentam a mensagem **Programa para Teste**. Observe que é usado na linha **GOTOXY** um valor menor do que o desejado para posicionamento.

```
; Poe cursor na Coluna = 01 / Linha = 24
; armazena msg2 em SI
; apresenta o conteúdo de msg2 na tela

GOTOXY    00d, 23d
LEA        SI, msg2
CALL       print_string
```

O trecho anterior na linha **LEA SI, msg2** transfere o valor de endereçamento da área de dados em que se encontra definida a variável **msg2** para o registrador de apontamento **SI** (*source index*). Na sequência a instrução **CALL** chama o procedimento **print_string** existente na biblioteca **emu8086.inc**.

```
; Aguarda que algo seja teclado

MOV        AH, 00h
INT        016h
```

O trecho anterior já havia sido usado anteriormente e tem por finalidade fazer uma pausa no teclado e colocá-lo em modo de espera até que alguma tecla seja acionada. Com isso aparece uma tela de saudação do programa.

```
; Habilita cursor
```

```
CURSORON
```

Após a apresentação da tela de saudação, o cursor é novamente habilitado.

```
; Limpa a tela  
CALL      clear_screen
```

Na sequência da execução de eventos o programa limpa a tela de saudação e deixa-a em branco para nova apresentação.

```
; Poe cursor na Coluna = 29 / Linha = 01  
; armazena msg1 em SI  
; apresenta o conteudo de msg1 na tela
```

```
GOTOXY    28d, 00d  
LEA       SI, msg1  
CALL      print_string
```

```
; Poe cursor na Coluna = 01 / Linha = 05  
; armazena msg3 em SI  
; apresenta o conteudo de msg3 na tela
```

```
GOTOXY    00d, 04d  
LEA       SI, msg3  
CALL      print_string
```

O trecho anterior faz a apresentação das mensagens armazenadas nas variáveis **msg1** **msg3** de acordo com o endereço de apontamento do registrador **SI**. A mensagem da variável **ms3** indica a entrada de um valor numérico.

```
; Efetua a entrada de um valor numerico em CX
```

```
CALL      scan_num
```

A instrução **CALL scan_num** executa a chamada do procedimento que faz o tratamento da entrada de um valor numérico que pode ser positivo ou negativo.

```
; Transfere o valor de CX para AX
```

```
MOV      AX, CX
```

Após fazer a entrada do valor numérico e proceder à sua ação de tratamento, por meio da linha anterior o programa transfere o conteúdo do registrador geral **CX** para o registrador geral **AX**. O registrador geral **CX** foi usado como uma área de armazenamento temporário do valor informado anteriormente.

```
; Apresenta mensagem de saida
```

```
CALL      pthis  
DB 0Dh, 0Ah, 'Foi fornecido o valor .....: ', 0d
```

```
; Apresenta o valor armazenado em AX
```

```
CALL      print_num
```

Por fim o programa chama o procedimento **pthis** que apresenta a mensagem **Foi fornecido o valor**, e depois apresenta o valor propriamente dito por meio da chamada do procedimento **print_num**.

```
; Finaliza programa
```

```
INT      020h
```

O trecho anterior faz o término de um programa **.COM**, como descrito anteriormente em outros capítulos.

```

; Rotulos de definicao das operacoes da biblioteca
DEFINE_SCAN_NUM
DEFINE_PRINT_STRING
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS
DEFINE_PTHIS
DEFINE_CLEAR_SCREEN
END

```

Inicialmente, antes da diretiva **END** define-se o fim das diretivas do código da biblioteca **emu8086.inc**.

10.6 - Manipulação de Cadeias

Na programação Assembly encontra-se alguns comandos exclusivos para a manipulação de cadeias (*strings*) de caracteres, tais como: **MOVSB**, **MOVSW**, **CMPSB**, **CMPSW**, **SCASB** e **SCASW**. Esses comandos para serem operacionalizados fazem uso dos registradores de índice **SI** e **DI** que dependendo do estado da definição do valor do registrador de direção **DF** podem incrementar ou decrementar o deslocamento dos comandos de manipulação de cadeias na memória. O incremento ou decremento ocorre a partir do formato da instrução, tamanho do operando ou do estado do *flag* de direção representado pelo registrador **DF**. O valor do incremento e decremento será de **1** para a manipulação de um *byte* e será de **2** para a manipulação de um *word* (OLIVEIRA, 2013 & LITERÁK, 2013). Se o valor do registrador **DF** for **0** ocorrerá incremento, se o valor for **1** ocorrerá decremento. O registrador **DF** pode ser configurado com os comandos **STD** para definir valor **1** (sentido da direita para à esquerda) e **CLD** para definir valor **0** (sentido da esquerda para à direita).

Os comandos **MOVSB** e **MOVSW** movimentam um *byte* (**MOVSB**) ou um *word* (**MOVSW**) da fonte endereçada pelos registradores de origem **SI:DI** para o destino endereçado pelos registradores **ES:DI** sem afetar nenhum registrador de estado (*flags*) e atualiza os registradores de índice **SI** e **DI** a partir do tamanho do operando ou instrução em uso. Os registradores de índice **SI** e **DI** são incrementados quando o sinalizador de direção é limpo e decrementado quando o sinalizador de direção está definido (LITERÁK, 2013). A operação de movimentação copia os dados da origem para o destino. Os comandos **MOVSB** e **MOVSW** são referenciados como comandos **MOVS** (**M**OV**E** **S**trings) e os dados manipulados são definidos em variáveis.

Para demonstrar a ação de cópia de conteúdo de uma posição de memória para outra posição considere um programa que peça a leitura de uma primeira sequência de caracteres, armazene-a em uma variável e mostre-a. Depois peça a leitura da segunda sequência de caracteres, armazene-a em outra variável e mostre-a. O programa deve copiar o conteúdo da variável da primeira entrada na variável da segunda entrada e mostrar o resultado da ocorrência. Assim sendo, execute o comando de menu **file/new** e escolha qualquer uma das opções. Acione simultaneamente as teclas **<Ctrl> + <A>** para selecionar o texto atual, em seguida acione a tecla **** para remover o texto existente e na área de edição em branco e codifique o programa seguinte, gravando-o com o nome **MANIPCAD01** e atente para o trecho em negrito:

```

;*****
;* Programa: MANIPCAD01.ASM *
;*****

INCLUDE 'emu8086.inc'

org 100h

.DATA

msg1e DB 'Entre a 1a. cadeia de caracteres .....: ', 0
msg2e DB 'Entre a 2a. cadeia de caracteres .....: ', 0
msg1s DB '1a. cadeia de caracteres .....: ', 0
msg2s DB '2a. cadeia de caracteres .....: ', 0
msg3s DB 'Agora 2a. cadeia de caracteres possui ...: ', 0

```

```

cadeia1 DB 30d DUP ('x')
cadeia2 DB 30d DUP ('x')

.CODE

; Ajuste do acesso a memoria

MOV AX, @DATA
MOV DS, AX
MOV ES, AX

; Entrada da primeira sequencia de caracteres

LEA    SI, msg1e
CALL   PRINT_STRING
LEA    DI, cadeia1
MOV    DX, 30d
CALL   GET_STRING
PUTC  13d
PUTC  10d

; Saida da primeira sequencia de caracteres em cadeia1

LEA    SI, msg1s
CALL   PRINT_STRING
MOV    SI, DI
CALL   PRINT_STRING
PUTC  13d
PUTC  10d

; Entrada da segunda sequencia de caracteres

LEA    SI, msg2e
CALL   PRINT_STRING
LEA    DI, cadeia2
MOV    DX, 30d
CALL   GET_STRING
PUTC  13d
PUTC  10d

; Saida da segunda sequencia de caracteres em cadeia2

LEA    SI, msg2s
CALL   PRINT_STRING
MOV    SI, DI
CALL   PRINT_STRING
PUTC  13d
PUTC  10d

; Operacao de copia da fonte para o destino

LEA SI, cadeia1 ; fonte
LEA DI, cadeia2 ; destino
CLD             ; DF = 0
MOV CX, 30d     ; tamanho da cadeia
REP MOVSB       ; repete cada caractere ate 30

```

```

; Saída da primeira sequencia de caracteres em cadeia2

LEA    SI, msg3s
CALL   PRINT_STRING
LEA    DI, cadeia2
MOV    DX, 30d
MOV    SI, DI
CALL   PRINT_STRING
PUTC  13d
PUTC  10d

INT    20h

DEFINE_PRINT_STRING
DEFINE_GET_STRING
END

```

Os detalhes gerais do programa anterior já são conhecidos. Assim, atente para o trecho em negrito e observe a transferência dos conteúdos das variáveis **cadeia1** e **cadeia2** respectivamente para os registradores de índice **SI** e **DI** por meio da instrução **LEA**. Após configurar o acesso as variáveis é executada a instrução **CLD** que coloca o registrador **DF** em **0** e estabelece que a leitura dos caracteres nas variáveis serão procedidos da esquerda para a direita. A linha de instrução **MOV CX, 30d** transfere para o registrador geral **CX** o valor decimal **30** que informa a quantidade máxima de caracteres das variáveis em uso e a instrução **REF MOVSB** faz a repetição (**REF**) do comando **MOVSB** por trinta vezes até percorrer toda a extensão do tamanho das variáveis em uso.

Os comandos **CMPSB** e **CMPSW** são usados para comparar um *byte* (CMPSB) ou um *word* (CMPSW) efetuando uma subtração entre o *byte* ou *word* endereçado no registrador **ES:DI** de destino em relação a origem endereçada no registrador **DS:SI** dentro do segmento de dados, sem devolver o resultado da subtração efetivada, mas afetando os registradores de estado **AF**, **CF**, **OF**, **PF**, **SF** e **ZF** (OLIVEIRA, 2013). O incremento e decremento para **CMPSB** ocorrerá em 1 e o incremento e decremento para **CMPSW** ocorrerá em 2 (LITERÁK, 2013). Os comandos **CMPSB** e **CMPSW** são referenciados como instruções **CMPS** (*CoMPare Strings*) e os dados manipulados são definidos em variáveis.

Para demonstrar a ação de comparação de conteúdo de uma posição de memória para outra posição considere um programa que peça a leitura de uma primeira sequência de caracteres e armazene-a em uma variável. Depois peça a leitura da segunda sequência de caracteres e armazene-a. O programa deve comparar os conteúdos informados e apresentar mensagem informando se são iguais ou diferentes. Assim sendo, execute o comando de menu **file/new** e escolha qualquer uma das opções. Acione simultaneamente as teclas **<Ctrl> + <A>** para selecionar o texto atual, em seguida acione a tecla **** para remover o texto existente e na área de edição em branco e codifique o programa seguinte, gravando-o com o nome **MANIPCAD02** e atente para o trecho em negrito:

```

;*****
;* Programa: MANIPCAD02.ASM *
;*****

INCLUDE 'emu8086.inc'

org 100h

.DATA

msg1e DB  'Entre a 1a. cadeia de caracteres .....: ', 0
msg2e DB  'Entre a 2a. cadeia de caracteres .....: ', 0
msg1s DB  '1a. cadeia de caracteres .....: ', 0
msg2s DB  '2a. cadeia de caracteres .....: ', 0
msg3s DB  'As cadeias sao iguais.', 0
msg4s DB  'As cadeias sao diferentes.', 0

cadeia1 DB 30d DUP ('x')
cadeia2 DB 30d DUP ('x')

```

.CODE

; Ajuste do acesso a memoria

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

; Entrada da primeira sequencia de caracteres

```
LEA    SI, msg1e  
CALL   PRINT_STRING  
LEA    DI, cadeia1  
MOV    DX, 30d  
CALL   GET_STRING  
PUTC  13d  
PUTC  10d
```

; Saida da primeira sequencia de caracteres em cadeia1

```
LEA    SI, msg1s  
CALL   PRINT_STRING  
MOV    SI, DI  
CALL   PRINT_STRING  
PUTC  13d  
PUTC  10d
```

; Entrada da segunda sequencia de caracteres

```
LEA    SI, msg2e  
CALL   PRINT_STRING  
LEA    DI, cadeia2  
MOV    DX, 30d  
CALL   GET_STRING  
PUTC  13d  
PUTC  10d
```

; Saida da segunda sequencia de caracteres em cadeia2

```
LEA    SI, msg2s  
CALL   PRINT_STRING  
MOV    SI, DI  
CALL   PRINT_STRING  
PUTC  13d  
PUTC  10d
```

; Operacao de comparacao da fonte com o destino

```
LEA SI, cadeia1 ; fonte  
LEA DI, cadeia2 ; destino  
CLD             ; DF = 0  
MOV CX, 30d      ; tamanho da cadeia  
REPE CMPSB       ; repete cada caractere ate 30
```

JL difere

```
LEA SI, msg3s  
CALL PRINT_STRING ; escreve mensagem para condicao verdadeira  
JMP fim  
difere:  
LEA SI, msg4s  
CALL PRINT_STRING ; escreve mensagem para condicao falsa
```

```

fim:
PUTC 13d
PUTC 10d

INT 20h

DEFINE_PRINT_STRING
DEFINE_GET_STRING
END

```

Note que este programa no trecho em negrito muito se assemelha ao trecho em negrito do programa anterior. Observe que são trechos muito semelhantes, exceto pelo uso do comando **CMPSB**. Perceba que está se fazendo uso das instruções de salta no tratamento da condição para apresentar a mensagem devida a condição ocorrida.

Os comandos **SCASB** e **SCASW** são usados para comparar um *byte* (SCASB) ou um *word* (SCASW) de um caractere que esteja indicado no registrador de origem **AL** (para SCASB) ou **AX** (para SCASW) com o dado apontado no endereço de destino dos registradores **ES:DI**, afetando os registradores de estado **AF**, **CF**, **OF**, **PF**, **SF** e **ZF**. Essas instruções não efetuam a subtração, mas afetam os *flags* como se tivesse realizada a subtração. (LITERÁK, 2013).

O programa seguinte solicita a entrada de uma cadeia e a entrada de um caractere e apresenta mensagens informando se o caractere informado existe ou não na cadeia informada. Assim sendo, execute o comando de menu **file/new** e escolha qualquer uma das opções. Acione simultaneamente as teclas **<Ctrl> + <A>** para selecionar o texto atual, em seguida acione a tecla **** para remover o texto existente e na área de edição em branco e codifique o programa seguinte, gravando-o com o nome **MANIPCAD03** e atente para o trecho em negrito:

```

;*****
;* Programa: MANIPCAD03.ASM *
;*****

INCLUDE 'emu8086.inc'

org 100h

.DATA

msg1e DB 'Entre uma cadeia de caracteres ....: ', 0
msg2e DB 'Entre apenas um caractere .....: ', 0
msg3s DB 'A cadeia possui o caractere informado.', 0
msg4s DB 'O caractere nao existe na cadeia.', 0

cadeia1 DB 30d DUP ('x')
cadeia2 DB 01d DUP ('x')

.CODE

; Ajuste do acesso a memoria

MOV AX, @DATA
MOV ES, AX

; Entrada da primeira sequencia de caracteres

LEA SI, msg1e
CALL PRINT_STRING
LEA DI, cadeia1
MOV DX, 30d
CALL GET_STRING
PUTC 13d
PUTC 10d

```

```

; Entrada apenas do caractere

LEA      SI, msg2e
CALL    PRINT_STRING
LEA      DI, cadeia2
MOV      DX, 30d
CALL    GET_STRING
PUTC    13d
PUTC    10d

; Operacao de comparacao da fonte com o destino

LEA DI, cadeia1 ; fonte
MOV AL, cadeia2 ; destino
CLD             ; DF = 0
MOV CX, 30d     ; tamanho da cadeia
REPE SCASB      ; repete cada caractere ate 30

JGE difere
LEA SI, msg3s
CALL PRINT_STRING ; escreve mensagem para condicao verdadeira
JMP fim
difere:
LEA SI, msg4s
CALL PRINT_STRING ; escreve mensagem para condicao falsa
fim:
PUTC 13d
PUTC 10d

INT 20h

DEFINE_PRINT_STRING
DEFINE_GET_STRING
END

```

Além das instruções de manipulação de cadeias apresentadas existem outras instruções que não são tratadas aqui. No entanto, o leitor poderá a partir desta base ampliar seu conhecimento pesquisando a respeito de outras instruções com esta finalidade.