```
0D82:0100 B402          MOV     AH,02
0D82:0102 B241          MOV     DL,41
0D82:0104 CD21          INT     21
0D82:0106 CD20          INT     20
0D82:0108 69            DB      69
```

Capítulo 8

PROGRAMAÇÃO BÁSICA

Este capítulo traz alguns elementos básicos complementares utilizados na programação de baixo nível na linguagem de programação Assembly 8086/8088. São abordados manipulação de registradores, manipulação de dados, tipos de dados a serem associados a variáveis, processo de cálculos matemáticos, novas instruções matemáticas e valores hexadecimais com oito dígitos.

8.1 - Manipulação de registradores e dados

Anteriormente tanto no programa **Enhanced DEBUG** como no programa **emu8086** foi bastante usado o comando **MOV** para a movimentação de valores entre registradores. A característica básica do comando **MOV** é movimentar (carregar) um determinado dado de um endereço de memória fonte (registrador) para outro endereço de memória destino ou movimentar para um registrador certo valor de forma direta. Além das formas usadas a instrução **MOV** pode ser utilizada de outras formas para movimentar os dados com o acesso de memória (DANDAMUDI, 2000 e HYDE, 2003) baseado em acesso direto, indireto por registrador, indexado, acesso de base indexada e de base indexada mais deslocamento, os quais serão apresentados mais adiante com alguns novos detalhes.

8.1.1 - Endereçamento imediato

O endereçamento imediato está relacionado à possibilidade de carregar um registrador com um valor de 8 ou 16 *bits*. Imagine a necessidade de mover um valor hexadecimal de 8 *bits* para os 8 *bits* mais significativos do registrador geral **AX**. Neste caso o valor seria movido para o registrador mais significativo **AH** com a linha de código:

```
MOV AH, FFh
```

O valor **FFh** (equivalente a **1111 1111** em notação binária, pode ser entendido como valor decimal positivo **255** ou decimal negativo **–1**) está sendo movimentado de forma direta para o registrador **AH**, sendo esta uma das formas mais comuns desse tipo de operação. Lembre-se de que a consideração em relação ao fato de o valor ser positivo ou negativo depende da forma como este estará sinalizado. Para ser visto como negativo, os registradores **CF**, **SF** e **AF** devem estar setados com valor **1**.

8.1.2 - Endereçamento por registrador

O endereçamento por registrador está relacionado com a possibilidade de carregar um determinado registrador com o valor existente em outro registrador, uma das formas utilizadas nos exemplos anteriores. Imagine a necessidade de mover um valor existente no registrador de 8 *bits* mais significativo do registrador geral **CX** para o de 8 *bits* menos significativo do registrador geral **BX**. Neste caso o valor seria movido do registrador **CH** para o **BL** com a linha de código:

```
MOV BL, CH
```

Após a transferência do valor, o registrador **CH** permanece inalterado, assim como também permanece inalte-rado o valor que porventura exista no registrador **BH**. Apenas o registrador **BH** foi alterado com o valor do registrador **CH**.

8.1.3 - Endereçamento por deslocamento (offset)

As duas formas comentadas anteriormente são os mecanismos mais rápidos e fáceis de usar, pois o endereçamento é passado de forma imediata, seja por intermédio de um dado (valor) informado a um registrador ou de transferência de valores entre registradores. Ocorre muitas vezes a necessidade de obter dados que estão armazenados na memória e não em registradores. Torna-se necessário utilizar o endereçamento por deslocamento (também conhecido como endereçamento por *offset*).

Para que seja possível obter o endereçamento em memória de um determinado dado, é necessário definir uma constante de 16 *bits* (a qual é denominada *deslocamento* ou *offset*) que somada ao conteúdo de um registrador (de 16 *bits*) fornece a real posição de memória em que se encontra aquele dado.

O endereço efetivo (deslocamento ou *offset*) pode ser obtido de diversos modos. Inicialmente basta conhecer de forma básica os endereçamentos indexado e de base indexada com deslocamento.

O endereçamento indexado, quando de sua indicação na própria instrução de movimentação, está relacionado a um valor numérico hexadecimal entre colchetes.

Imagine a necessidade de mover um valor de endereço de deslocamento de memória para o registrador **AX**, considerando que o valor em questão corresponde ao endereço de deslocamento **00100h**:

```
MOV AX, [00100h]
```

O endereço de memória **00100h** (valor hexadecimal) representa o valor de deslocamento (*offset*) do segmento em uso. O endereçamento indireto está relacionado, quando de sua indicação na própria instrução de movimentação, associado a uma posição de memória de um determinado registrador entre colchetes. Imagine a necessidade de mover para o registrador geral **AX** a posição de memória do conteúdo do registrador geral **BX**:

```
MOV AX, [BX]
```

É possível ainda determinar a posição de memória utilizando:

```
MOV BX, [00100h]
MOV AX, [BX]
```

Neste caso, primeiramente se está carregando o registrador geral **BX** de forma direta com o endereço de memória **00100h**, em seguida, de forma indireta, o registrador geral **AX** com o endereço de memória armazenado no registrador geral **BX**.

O endereçamento de base indexada está relacionado, quando de sua indicação entre colchetes, ao endereço inicial de memória somado a ele um valor constante de deslocamento. Imagine a necessidade de mover para o registrador geral **AX** o conteúdo existente no registrador geral **BX** mais a posição do endereço de memória **00100h**:

```
MOV AX, [00100h+BX]
```

Esta indicação pode ainda ser escrita:

```
MOV AX, [BX+00100h]
MOV AX, 00100h[BX]
MOV AX, [BX]+00100h
```

Esse tipo de endereçamento permite o acesso a dados que estejam armazenados em lugares diferentes da memória. Esse tipo de indicação facilita o acesso de dados existentes em matrizes (tabelas), registros ou outras estruturas de dados que ainda serão apresentadas.

Além das formas indicadas, a passagem de endereço de deslocamento pode também ser definida como:

```
MOV DX, OFFSET var
```

Esse tipo de endereçamento permite definir o endereço de deslocamento utilizado pela variável (neste caso, variável **var**) para o registrador geral em uso (neste caso **DX**).

8.1.4 - Outras formas de deslocamento

Além das formas de endereçamentos de deslocamento apresentadas é possível realizar outras operações como: endereçamento com deslocamento direto; endereçamento com deslocamento indireto sobre registrador; endereçamento com deslocamento por registrador geral; endereçamento com deslocamento por registrador ponteiro e endereçamento de deslocamento sobre registrador geral com registrador de ponteiro.

A ação de endereçamento com deslocamento direto é definida a partir da instrução **MOV AX, var**, onde **var** é a definição do endereço de uma variável na memória.

O endereçamento com deslocamento indireto sobre registrador ocorre a partir da leitura de um registrador base **BX** ou **BP** ou mesmo de um registrador de ponteiro (índice) **SI** ou **DI** por meio de instrução similar a **MOV AX, [BX]**.

Para o endereçamento com deslocamento de registrador geral o endereço é lido de um registrador base **BX** ou **BP** adicionado um valor de deslocamento a partir da instrução **MOV AX, [BX+0010h]**.

Para o endereçamento com deslocamento por registrador ponteiro usa-se sintaxe de instrução idêntica ao uso de deslocamento de registrador geral. Neste caso, levando-se em conta o uso dos registradores de ponteiro **DI** ou **SI** com a instrução **MOV AX, [DI+0010h]**.

Os modos de endereçamento indexados são obtidos a partir da soma de um registrador base (**BX** ou **BP**) com um registrador de ponteiro (**SI** ou **DI**). O registrador **BP** é usado quando se necessita trabalhar com a pilha. A sintaxe deste tipo de ação pode ser estabelecida com a instrução **MOV AX, [BX][DI]** ou **MOV AX, tabela[BX][DI]**.

8.2 - Tipos de dados em Assembly

A linguagem de programação *Assembly 8086/8088,* por meio das diretivas **DB** (*Define Byte*), **DW** (*Define Word*), **DD** (*Define Doubleword*), **DQ** (*Define Quadword*) e **DT** (*Define Ten Bytes*), permite manipular alguns tipos dados básicos associados a valores e as suas variáveis. Analogamente a outras linguagens, é possível manipular valores do tipo inteiro, real e caractere (*string*). Os tipos de dados em *Assembly* podem tratar os seguintes dados:

- A diretiva **DB** pode ser utilizada para manipular dados do tipo *string* e também valores inteiros curtos (8 *bits*), com a capacidade de manipular valores de –128 até 255 (de -2^7 até $2^8 - 1$).
- A diretiva **DW** pode ser usada para manipular valores inteiros curtos e também valores reais curtos (16 *bits*, sendo 2 *bytes* consecutivos, 1 *word*), com a capacidade de manipular valores de –32.768 até 65.535 (de -2^{15} até $2^{16} - 1$).
- A diretiva **DD** pode ser usada para manipular valores inteiros longos e também valores reais curtos (32 *bits*, sendo 4 *bytes* consecutivos, 1 *doubleword*), com a capacidade de manipular valores de –2.147.483.648 até 4.294.967.295 (de -2^{63} até $2^{64} -1$).
- A diretiva **DQ** pode ser usada para manipular valores reais longos (64 *bits*, sendo 8 *bytes* consecutivos, 1 *quadword*), com a capacidade de manipular valores de –9.223.372.036.854.775.808 até 18.446.744.073.709 .551.615 (de -2^{63} até $2^{64} - 1$).
- A diretiva **DT** pode ser usada para manipular valores que ocupem até dez *bytes* consecutivos.

Das diretivas apresentadas o programa **emu8086** aceita apenas **DB** e **DW**, as quais são mais do que suficientes para os objetivos deste trabalho. Além disso, a ferramenta **emu8086** é um instrumento de apoio básico e didático ao aprendizado das noções preliminares da linguagem de programação de computadores *Assembly 8086/8088*.

A seguir são apresentados alguns exemplos das diretivas de definição de tipos de dados. Para o uso desse recurso deve-se observar a sintaxe seguinte:

```
variável diretiva valor1 [,<valor1>], ...
```

O parâmetro **variável** é um rótulo de identificação do nome da variável para utilização dos dados referenciados no programa, o parâmetro **diretiva** é a definição de um tipo de dado válido e o parâmetro **valor** é a definição do valor associado a uma variável.

Há a possibilidade de definir o parâmetro **valor** com o operador **?** (interrogação), que tem por finalidade reservar espaço na memória de acordo com o tipo de diretiva de definição em uso, sem efetuar a inicialização da variável com algum valor.

Após a definição do valor da expressão pode-se também reservar espaço na área de memória com a utilização da diretiva **DUP**. Assim sendo, poder-se-ia utilizar esse recurso de algumas maneiras.

```
caractere1 DB 'A'
caractere2 DB 41h
caractere3 DB 01000001b
```

A definição anterior estabelece para a variável com denominação **caractere** o espaço de um *byte* simples para o armazenamento do caractere **A**, ou dos valores em hexadecimal e binário para a representação do caractere **A**.

```
valor1 DW 26987d
```

A definição de um valor numérico positivo de 16 *bits* em notação decimal é internamente convertida de forma automática no seu valor equivalente em notação hexadecimal. O valor **26987d** é considerado internamente **696Bh**.

```
valor2 DW -25476d
```

A definição de um valor numérico negativo de 16 *bits* em notação decimal é internamente convertida de forma automática no seu valor equivalente em notação hexadecimal, considerando a complementação por 2. O valor **–25476d** é considerado internamente **9C7Ch**.

Na possibilidade de definir uma variável que não possua um valor inicial, ela pode ser definida como:

```
valor3 DB ?
valor2 DW ?
```

A definição de valores do tipo *string* para variáveis com a diretiva **DB** pode ser realizada no estilo de múltiplos dados. Considere a seguir a definição da variável **palavra** com o conteúdo "**Alo mundo!**", a qual pode ser definida como:

```
palavra DB 'A', 'l', 'o', ' ', 'M', 'u', 'n', 'd', 'o', '!'

palavra DB 'A'
        DB 'l'
        DB 'o'
        DB ' '
        DB 'M'
        DB 'u'
        DB 'n'
        DB 'd'
        DB 'o'
        DB '!'
```

As duas formas anteriores equivalem à seguinte definição:

```
palavra  DB 'Alo Mundo!'
```

O mesmo conceito também pode ser utilizado com as demais diretivas (considerando o fato de a ferramenta de compilação as aceitar). Considere uma matriz de dados denominada **vetor** com a capacidade de armazenar cinco valores numéricos.

```
vetor DW 0
      DW 0
      DW 0
      DW 0
      DW 0
```

Que também pode ser definida como:

```
vetor DW 0, 0, 0, 0, 0
```

No caso anterior poder-se-ia também reservar espaço de memória utilizando a diretiva **DUP**, como mostrado a seguir:

```
vetor DW 5 DUP (0)
```

A diretiva **DUP** é largamente utilizada quando há necessidade de definir uma variável que seja do tipo vetor (matriz de uma dimensão) ou tabela (matriz de duas ou mais dimensões). São formas válidas:

```
vetor1 DB 8 DUP (?)
vetor2 DW 4 DUP ('?')
vetor3 DB 2 DUP ('Alo ')
```

A primeira definição estabelece para a variável **vetor1** a reserva de 8 *bytes* de memória com valor de inicialização desconhecido. A segunda definição estabelece para a variável **vetor2** a reserva de 4 *words* com a definição do caractere **?**. A terceira e última definição estabelece para a variável **vetor3** a definição de 2 *bytes* inicializados com **Alo Alo**.

Para os casos de definição de uma tabela, a diretiva **DUP** deve ser utilizada com a seguinte sintaxe:

```
matriz DW 5 DUP (3 DUP (0))
```

Neste caso está sendo definida uma variável **matriz**, a qual é inicializada com 15 *words* com valor zero. Essa forma seria algo similar a uma tabela de cinco linhas e três colunas.

A diretiva **DUP** pode ser usada de muitas formas para inicializar o valor de uma variável. Por exemplo, imagine que se deseja inicializar uma variável denominada **dados** com o seguinte conteúdo: **444433322espaço 444433322espaço 444433322espaço**. Neste caso, usar-se-ia a seguinte sintaxe:

```
dados DB 3 DUP (4 DUP ('4'), 3 DUP ('3'), 2 DUP ('2'), 1 DUP (' '))
```

A variável **dados** é uma matriz com três posições, e cada posição possui a inicialização de quatro caracteres **4**; três caracteres **3**; dois caracteres **2** e **1** caractere com um espaço em branco.

O uso de diretivas de definição para tipos de dados sugere o uso em conjunto das diretivas **DATA** e **CODE**, que permitem o estabelecimento de uma sequência de definições.

Por exemplo, considere o programa **MENSAGEM1** apresentado no capítulo anterior, conforme em seguida:

```
; You may customize this and other start-up templates;
; The location of this template is c:\emu8086\inc\0_com_template.txt

org 100h

MOV AH, 09h
LEA DX, mensagem
INT 21h
INT 20h

mensagem DB 41h, 6Ch, 6Fh, 20h, 6Dh, 75h, 6Eh, 64h, 6Fh, 24h

ret
```

No programa **MENSAGEM1** o código é definido a frente da definição da variável **mensagem**. Esta forma de definição permite que o programa montador crie na memória o código de apresentação da mensagem e coloque a mensagem após a última linha do código associando o endereço escolhido automaticamente para a variável **mensagem** junto a instrução **LEA DX, mensagem**. No entanto, a forma de escrita apresentada não é a forma mais comum de definição de um programa em *Assembly*, pois normalmente se define primeiro as variáveis do programa e depois seu código.

Como segundo exemplo, no programa **emu8086** execute o comando de menu **file/new/com template**, sobre a linha **07** escreva o código de programa a seguir e com o comando **file/save as...** grave o programa com o nome **MENSAGEM2**:

```
JMP inicio

mensagem DB 41h, 6Ch, 6Fh, 20h, 6Dh, 75h, 6Eh, 64h, 6Fh, 21h, 24h

inicio:
  LEA DX, mensagem
  MOV AH, 09h
  INT 21h
  INT 20h
```

A definição da variável **mensagem** está ocorrendo nessa versão antes do início das linhas de código de controle do programa. Por esta razão é necessário indicar no código do programa o uso da instrução **JMP** que salta para o rótulo indicado como **inicio**, onde estão as instruções do programa. Observe a Figura 8.1.

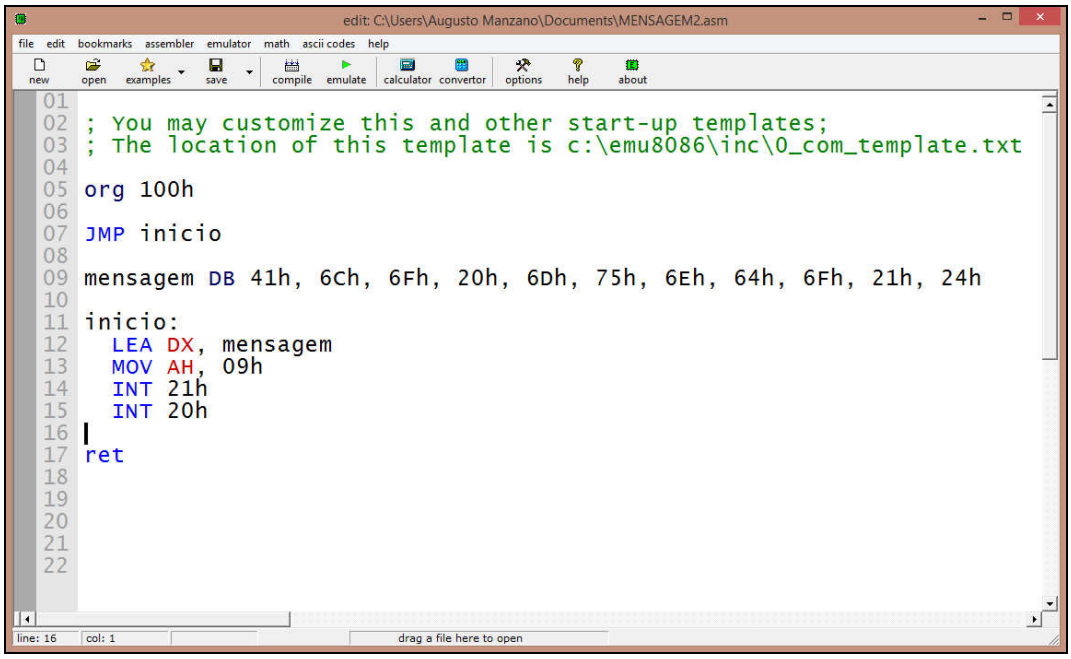

Figura 8.1 - Programa MENSAGEM2.

Se retirada a linha **JMP inicio** do código do programa, ocorrerá um erro na execução do programa. O erro ocorre quando há a tentativa de montar o programa na memória, pois o código do programa deve vir sempre à frente da definição dos dados a serem manipulados pelo programa. Por esta razão a segunda versão utiliza a instrução **JMP** que faz com que seja executado primeiramente o código do programa que buscará os dados na variável **mensagem**.

Uma forma de proceder com uma montagem de programa mais adequada é fazer uso das diretivas **.DATA** e **.CODE**. Desta forma é possível definir primeiramente os dados do programa antes da parte do código do programa.

Definir primeiramente os dados para depois definir o código é uma prática de programação considerada correta e elegante. Assim sendo, observe o trecho de código seguinte:

```
.DATA
  mensagem DB 41h, 6Ch, 6Fh, 20h, 6Dh, 75h, 6Eh, 64h, 6Fh, 21h, 24h

.CODE
  LEA DX, mensagem
  MOV AH, 09h
  INT 21h
  INT 20h
```

Execute o comando de menu **file/new/com template**, informe o código anterior a partir da linha **04** e grave o programa com o comando **file/save** com o nome **MENSAGEM3**. Observe a Figura 8.2.

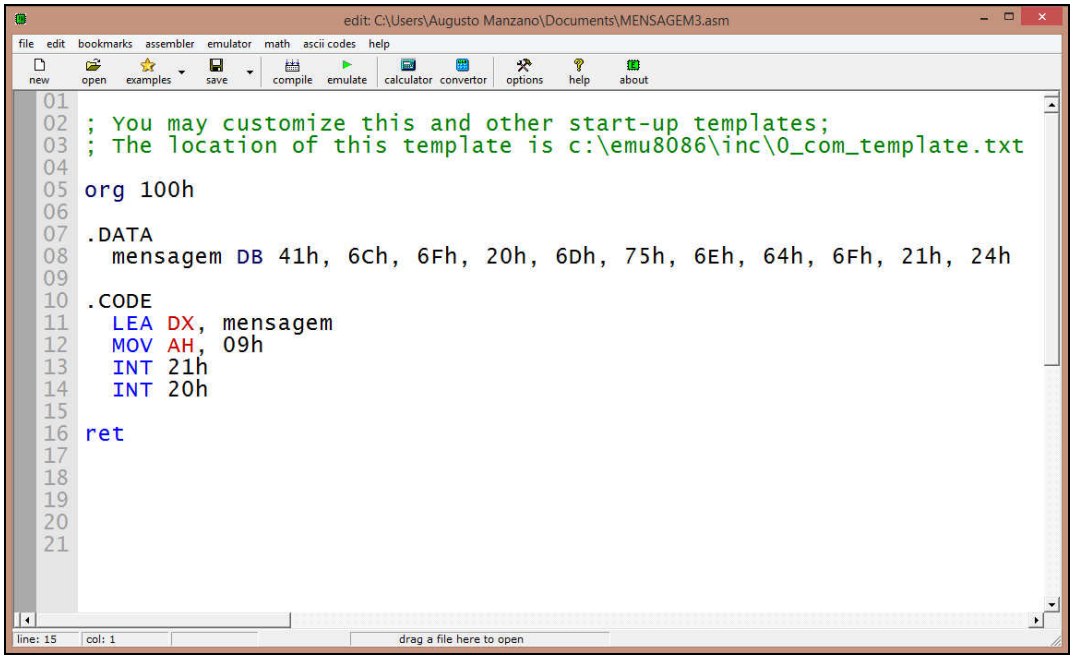

Figura 8.2 - Programa MENSAGEM3.

Veja o uso das diretivas **.DATA** e **.CODE** (atenção para os símbolos de ponto usados nas diretivas). Desta forma, definem-se duas áreas de programa: a área de dados (**.DATA**) e a área de código (**.CODE**). A partir desse instante esta será a abordagem usada para a codificação de todos os programas que, quando compilados, possuírem a extensão **.COM**.

É importante ressaltar que as ferramentas de montagem (os programas *assembler*) não fazem nenhuma questão quanto aos caracteres utilizados nos códigos serem maiúsculos ou minúsculos. É possível escrever um programa inteiro com todos os caracteres grafados no formato minúsculo ou no formato maiúsculo. No entanto, por questões de melhor visualização do código, nesta obra adotou-se o critério de grafar em caracteres maiúsculos as instruções e as diretivas. Rótulos de identificação de posição de código e de variáveis são escritos em formato minúsculo.

8.3 - Cálculos matemáticos intermediários

Anteriormente foram apresentadas algumas formas básicas para a realização de pequenos cálculos matemáticos com a ferramenta **Enhanced DEBUG**. Neste tópico este assunto volta à tona acrescido de alguns novos detalhes, como, por exemplo, levar em consideração no processo de cálculo o registrador de estado **CF** (*Carry Flag*).

A linguagem de programação de computadores *Assembly 8086/8088* tem alguns comandos reservados para cálculos aritméticos, tais como adição (**ADD** ou **ADC**), subtração (**SUB** ou **SBB**), multiplicação (**MUL** ou **IMUL**) e divisão (**DIV** ou **IDIV**). Lembre-se de que os comandos **ADD**, **SUB**, **MUL** e **DIV** já foram apresentados e serão revistos neste tópico (é sempre bom lembrar).

É conveniente considerar que os comandos de cálculos aritméticos alteram os registradores de estado (*flags*) **OF**, **SF**, **ZF**, **AF**, **PF** e **CF** existentes na memória. O registrador de estado afetado mais importante em operações aritméticas é **CF**, pois sinaliza na memória a ocorrência do efeito "vai um" quando do estouro da manipulação aritmética sobre a posição de um registrador geral.

8.3.1 - Adição

As operações de adição podem ocorrer com a utilização dos comandos **ADD** ou **ADC**. O comando **ADD** tem a mesma finalidade do operador aritmético "**+**" existente em outras linguagens de programação. Já o comando **ADC** possui um pequeno diferencial, pois além de executar a mesma operação do comando **ADD,** ele soma ao valor o valor do registrador de estado **CF** que pode ser **0** (zero) ou **1** (um).

O funcionamento lógico do comando **ADD** será **ADD DESTINO, ORIGEM** (**DESTINO ← DESTINO + ORIGEM**).

```
ADD AX, 5d
```

No exemplo apresentado o registrador geral **AX** está sendo adicionado com o valor decimal **5**. Neste contexto o valor 5 é a definição de um valor constante no registrador. Se no registrador geral **AX** existir algum valor anterior, o valor **5** será somado ao existente.

Observação

A definição de valores constantes pode ocorrer com o uso de sinalizadores da base numérico a que o número é referenciado. Por exemplo, o valor **5d** indica a definição de um valor do tipo *decimal*. Assim sendo pode-se usar como sinalização de base após um valor constante as letras: **d** (decimal), **h** (hexadecimal), **o** (octal) e **b** (binário).

O funcionamento lógico da instrução **ADC** será **ADC DESTINO, ORIGEM** (**DESTINO ← DESTINO + ORIGEM + CF**).

```
ADC AX, 5d
```

No exemplo apresentado o registrador geral **AX** está sendo adicionado com o valor decimal **5** mais o valor que estiver no registrador de estado **CF**, que pode ser **0** (zero) ou **1** (um).

Tome como base um programa que deva executar a soma de dois valores numéricos que ocupem no máximo 1 *word*. O primeiro valor deve estar associado a uma variável denominada **a**, o segundo valor a uma variável denominada **b** e definir uma terceira variável denominada **x** para armazenar o valor da soma.

Para a criação deste programa execute no programa **emu8086** o comando de menu **file/new/com template**, acione as teclas **<Ctrl> + <a>** para selecionar o texto apresentado e em seguida acione **<Del>**. Em seguida escreva o código do programa seguinte:

```
;*******************************
;*    Programa: ADICAO1.ASM    *
;*******************************

.MODEL small
.STACK 512d

.DATA
  a DW 6d
  b DW 2d
  x DW 0, '$'

.CODE
  MOV  AX, @DATA
  MOV  DS, AX

  MOV  AX, a
  ADD  AX, b
  MOV  x, AX
  ADD  x, 30h
  MOV  DX, OFFSET x

  MOV  AH, 09h
  INT  21h

  MOV  AH, 4Ch
  INT  21h
```

Grave o programa por meio dos comandos de menu **file/save** na pasta **Documentos** com o nome **ADICAO1**. A Figura 8.3 apresenta o programa definido dentro do editor de textos do ambiente de programação **emu8086**.

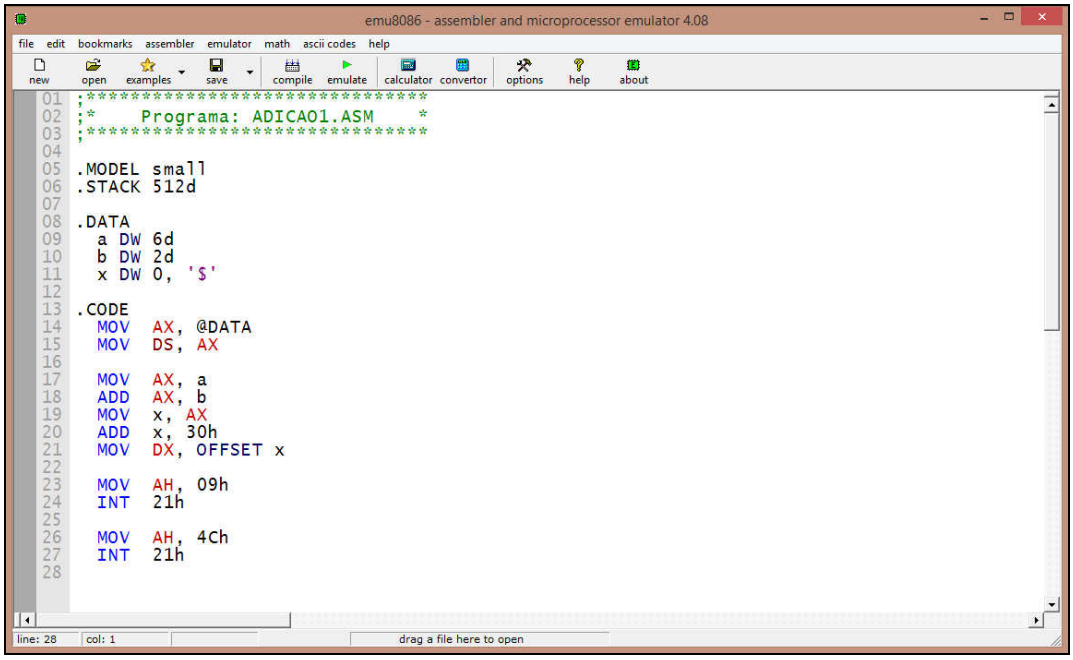

Figura 8.3 - Programa ADICAO1 na ferramenta emu8086.

Nas linhas **05** e **06** estão sendo utilizadas duas novas diretivas, sendo **.MODEL** e **.STACK**.

A diretiva **.MODEL** indica o tipo de modelo de memória que deve ser usado pelo programa. Deve ser utilizada antes de qualquer definição de segmento de memória, ou seja, deve ser a primeira linha do programa.

O parâmetro utilizado após a diretiva **.MODEL** pode ser[1] **small**, **medium**, **compact**, **large** e **huge**, os quais possuem o seguinte significado:

O modelo **small** é o modo mais adequado para a maioria das aplicações, devido à sua rapidez de carga e facilidade de depuração. Esse modelo estabelece que o código do programa está em um segmento de memória e os dados estão em outro segmento, cada um ocupando menos de 64KB de memória.

- O modelo **medium** estabelece que o código do programa pode ultrapassar a marca de 64 KB e pode existir mais de um segmento de código, enquanto os dados estarão alocados em um único segmento com até 64 KB.
- O modelo **compact** estabelece que o código do programa não pode ser maior que 64 KB, porém permite que para os dados do programa possa ser utilizado mais de um segmento de dados, ou seja, mais de 64 KB.
- O modelo **large** estabelece que tanto o código do programa como os seus dados podem utilizar mais de 64 KB. No entanto arranjos de dados (*arrays*) não podem ultrapassar 64 KB.
- O modelo **huge** estabelece que código do programa, dados e *arrays* podem utilizar mais de 64 KB.

A diretiva **.STACK** tem por finalidade estabelecer a reserva de espaço na pilha do programa. O tamanho da pilha a ser definido depende de alguns fatores com relação à chamada de sub-rotinas, registradores salvos, interrupções e passagens de parâmetros. Normalmente, utiliza-se um tamanho em torno de 512 *bytes* (valor decimal). Desta forma, você pode usar o valor 512 como padrão.

Na linha **08** define-se o segmento de dados **.DATA** com a criação de três variáveis **a**, **b** e **x**, todas do tipo **DW** com seus respectivos valores decimais **6**, **2** e **0**. O caractere "**$**" existente após a definição da variável **X** tem a mesma ação utilizada nos programas **MENSAGEM1**, **MENSAGEM2** e **MENSAGEM3**. A diretiva **.DATA** permite que seja definido o espaço do segmento de dados do programa.

A partir da linha **13** é definido o segmento **.CODE**, o qual possui a sequência de instruções do programa que soma os valores entre as variáveis **a** e **b**, faz a atribuição do valor à variável **x** e executa a apresentação do resultado na tela do monitor de vídeo. A diretiva **.CODE** permite que seja definido o espaço do segmento de código do programa.

Observação

Os programas apresentados neste capítulo têm estruturas simples e devem ser escritos como são exibidos. Não tente fazer nenhuma alteração, pois talvez não surtam os efeitos pretendidos. O programa **ADICAO1** está preparado apenas para tratar somas de unidades. Valores que utilizem dezenas, centenas ou milhares não serão apresentados. Portanto, não se preocupe ainda com esses detalhes.

As linhas **14** (**MOV AX, @DATA**) e **15** (**MOV DS, AX**) definem o acesso do segmento de código ao segmento de dados. Com a linha **17** (**MOV AX, a**) ocorre a movimentação do valor da variável **a** para o registrador **AX** e com a linha **18** (**ADD AX, b**) é efetuado o processamento da operação de adição do valor da variável **b** sobre o valor da variável **a** armazenado no registrador **AX**. A linha **19** (**MOV AX, x**) movimenta o valor somado do registrador **AX** para a variável **x**, efetuando a operação **x = a + b**.

Na linha **20** (**ADD x, 30h**) encontra-se a adição do valor **30h** à variável **x**. Lembre-se de que esse recurso de adicionar o valor **30h** a um resultado possibilita que esse resultado seja apresentado. A linha **21** (**MOV DX, OFFSET x**) efetua o calculo do tamanho que a variável **x** ocupa na memória e armazena esta quantidade no registrador **DX** para que seja usada no momento da apresentação do resultado quando da execução das linhas **23** e **24**.

As linhas **23** (**MOV AH, 09h**) e **24** (**INT 21h**) fazem a apresentação do resultado da soma que está armazenado na variável **x** apontada no registrador **DX** e as linhas **26** (**AH, 4Ch**) e **27** (**INT 21h**) fazem o encerramento do programa. O valor **4Ch** armazenado no registrador **AX** na linha **27** estabelece que ocorrerá o encerramento do programa, fazendo-se o retorno do controle operacional para sistema operacional a partir da chamada da interrupção **21h**.

Execute no programa **emu8086** o comando de menu **file/new/com template**, acione as teclas de atalho **<Ctrl> + <a>** e **<Del>** do editor de texto e escreva o programa anterior gravando-o na pasta **Documentos** com os comandos de menu **file/save** com o nome **ADICAO2**.

---

[1] *Dependendo do programa montador e versão utilizada, pode existir outros modelos de memória como **tiny** que estabelece que tanto o código como os dados deverão estar no mesmo segmento de 64KB de memória.*

Tome como base um programa que execute a equação **X ← A + B + CF**, em que a variável **a** possui o valor decimal **6**, a variável **b** o valor decimal **2** e **CF** é o valor existente no registrador de estado **CF**. Observe o código do programa a seguir:

```
;*******************************
;*    Programa: ADICAO2.ASM    *
;*******************************

.MODEL small
.STACK 512d

.DATA
  a DW 6d
  b DW 2d
  x DW 0, '$'

.CODE
  MOV  AX, @DATA
  MOV  DS, AX

  STC
  MOV  AX, a
  ADC  AX, b
  MOV  x, AX

  ADD  x, 30h
  MOV  DX, OFFSET x

  MOV  AH, 09h
  INT  21h

  MOV  AH, 4Ch
  INT  21h
```

A Figura 8.4 mostra a disposição do código dentro do ambiente de desenvolvimento do programa **emu8086**.

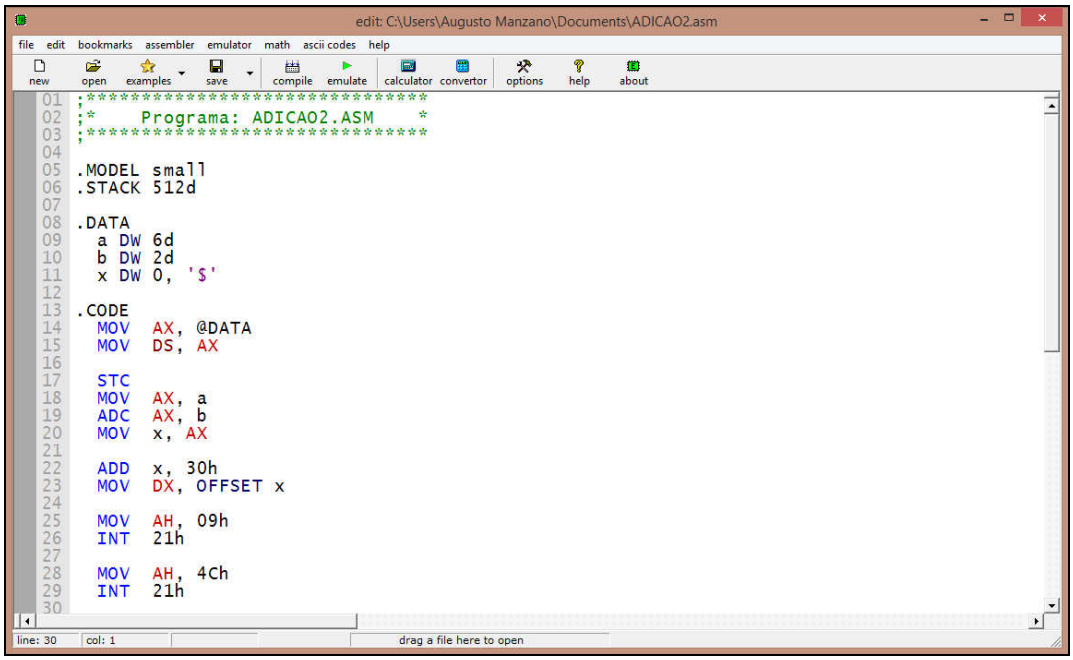

Figura 8.4 - Programa ADICAO2 na ferramenta emu8086.

A título de ilustração e de maior intimidade com a operacionalização da ferramenta **emu8086** , serão descritos os passos de ação das instruções do programa. Execute o comando de menu **assembler/compile and load in emulator**, ou acione a tecla de função **<F5>**, ou o botão **emulate** da barra de ferramentas. Assim que a ação anterior de execução do programa for

solicitada, serão apresentadas as telas de demonstração do programa, como indica a imagem da Figura 8.5, que retrata o primeiro trecho de ação.

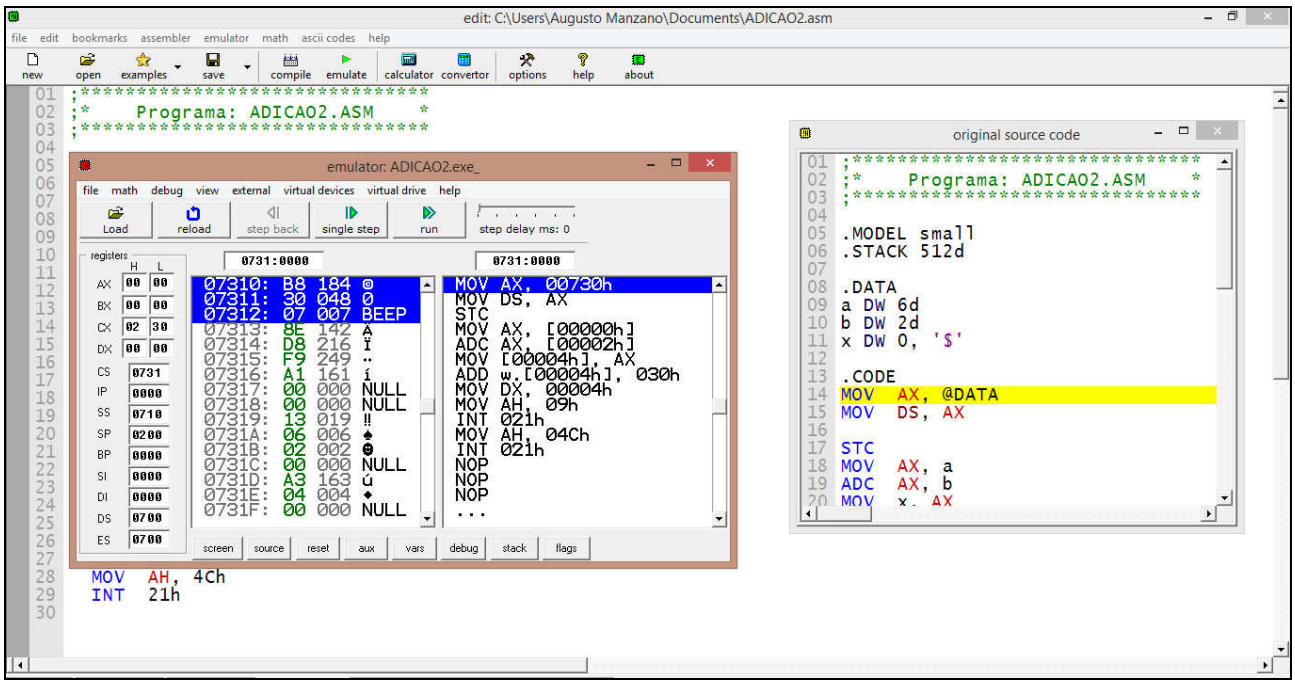

Figura 8.5 - Telas de ação com dados preliminares do programa.

A Figura 8.5 apresenta a barra de ação (barra amarela) da janela **original source code** posicionada após a diretiva **.CODE** da linha **14** do programa na linha **MOV AX. @DATA**. A janela **emulator: ADICAO2** mostra os registradores de segmento **CS** posicionado no endereço **0731h** e **IP** posicionado no endereço **0000h** (**CS:IP – 0731:0000**). Mas lembre-se de que os valores de segmento e deslocamento apresentados podem variar conforme o computador ou o momento. Assim sendo, aqui são válidos apenas como referência. A estrutura **CS:IP** é usada para indicar o endereço de memória do código sendo executado.

Pressione a tecla de função **<F8>** (primeira vez) ou acione o botão **single step** da janela **emulator: ADICAO2.exe_**. Observe que nessa primeira etapa o programa mostra a alteração do endereço dos registradores de deslocamento **IP** para o valor **0003h**, pois é o endereço de início da primeira instrução do programa (linha **15**). Basta olhar para a barra de ação amarela da janela **original source code** sobre o comando **MOV DS, AX** que transfere para o registrador geral **DS** o conteúdo de **AX**, o valor do endereço de segmento em que as variáveis e seus valores estão definidos.

Acione a tecla de função **<F8>** pela segunda vez e observe os dados apresentados em tela. Note que o programa está posicionado sobre a linha de código **STC** (linha **17**) que, ao ser executada, fará com que o registrador de estado **CF** passe do valor **0** (zero) para o valor **1** (um).

Na sequência execute o comando de menu **view/flags** na janela **emulator: ADICAO2exe_** e posicione a janela no lugar mais confortável da tela. Em seguida acione a tecla de função **<F8>** pela terceira vez e observe os dados apresentados em tela, como mostra a Figura 8.6 que indica o registrador de deslocamento **IP** com a indicação de seu valor como **0005h**, pois é o endereço de início da segunda instrução do programa.

A execução da linha de código **STC** (linha **17**) define para o registrador **CF** o valor **1**, que será perceptível a partir da terceira execução da tecla de função **<F8>**. Observe junto a Figura 8.7 este efeito que mostra também o registrador **IP** com a definição do valor **0006h** e a barra amarela mostra a linha de código **MOV AX, a** (linha **18**) que transfere o valor definido para a variável **a** para o registrador geral **AX**, que se encontra com o valor do endereço de deslocamento utilizado anteriormente para o registrador **DS**.

Nessa etapa na área no quadro direito da tela **emulator** encontra-se marcado o comando **MOV AX, [00000h]**, sendo o endereço de memória **0000h** o local onde está armazenado o valor hexadecimal **6**, como pode ser observado na Figura 8.7. Note o valor do registrador **CS** definido como **1**.

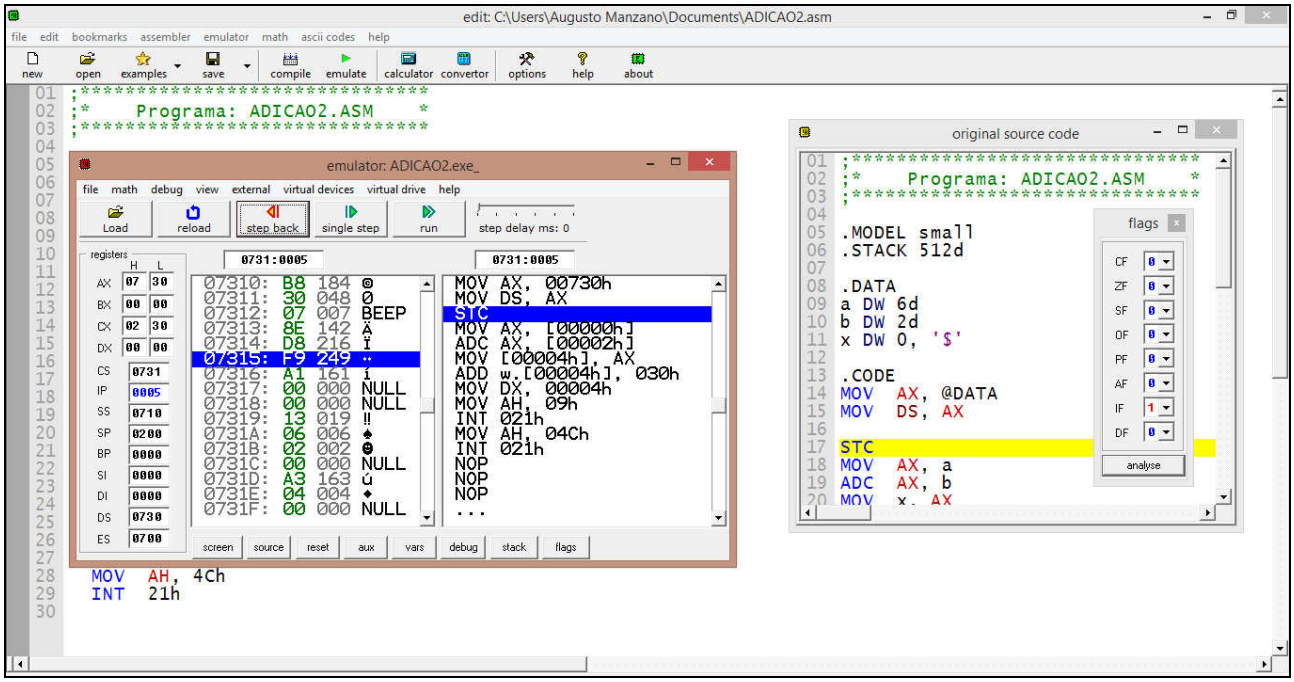

Figura 8.6 - Detalhes de ação do programa (segunda ação da tecla <F8>).

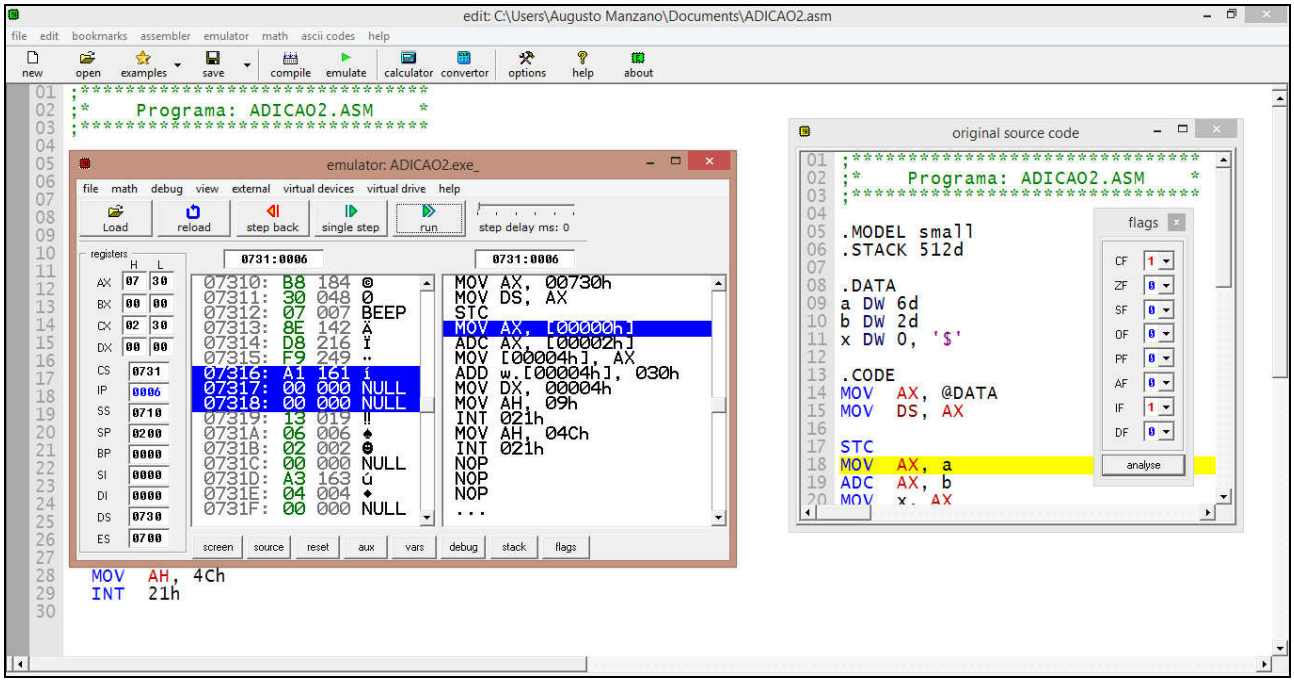

Figura 8.7 - Detalhes de ação do programa (terceira ação da tecla <F8>).

Na sequência de execução acione a tecla de função **<F8>** pela quarta vez e observe os dados apresentados em tela, como na Figura 8.8.

A instrução **ADC AX, b** (linha **19**) transfere o valor definido para a variável **b** para o registrador geral **AX**. Na área no quadro direito da tela **emulator** encontra-se marcado o comando **ADC AX, [00002h]** e o endereço **0002h** é o local em que se encontra o valor hexadecimal **2**, como pode ser observado.

Acione a tecla de função **<F8>** pela quinta vez e observe os dados apresentados em tela, Figura 8.9. Nessa etapa o registrador geral **AX** já está com a soma **8** e o valor **1** do registrador de estado **PF**, como pode ser notado na janela **ADICAO2**. O registrador **CF** volta a ter o valor **0** (zero).

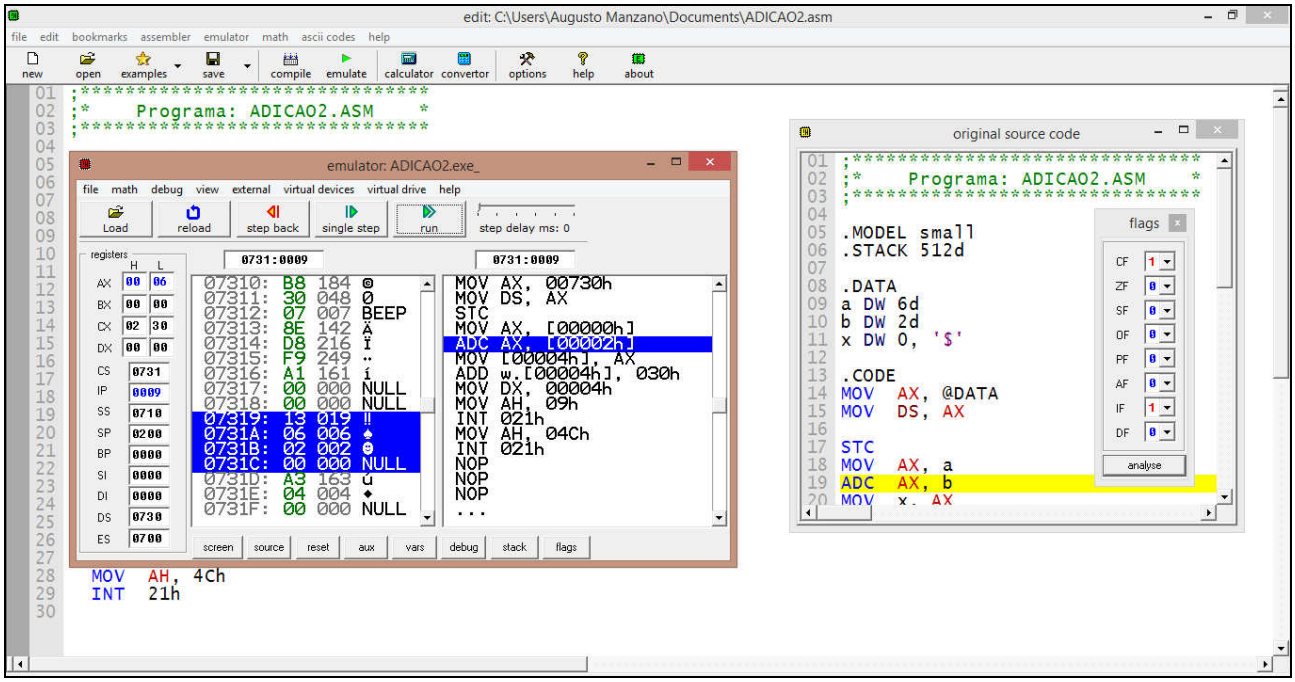

Figura 8.8 - Detalhes de ação do programa (quarta ação da tecla <F8>).

Pressione a tecla de função **<F8>** pela sexta vez e observe os dados apresentados em tela, como indica a Figura 8.10. A partir dessa etapa será adicionado o valor hexadecimal **030h** ao valor existente no registrador geral **AX**, por meio da instrução **ADD x, 030h** (linha **22**). Lembre-se de que essa estratégia é usada para obter o valor correspondente ao código ASCII do valor armazenado no registrador menos significativo do registrador geral **AX**.

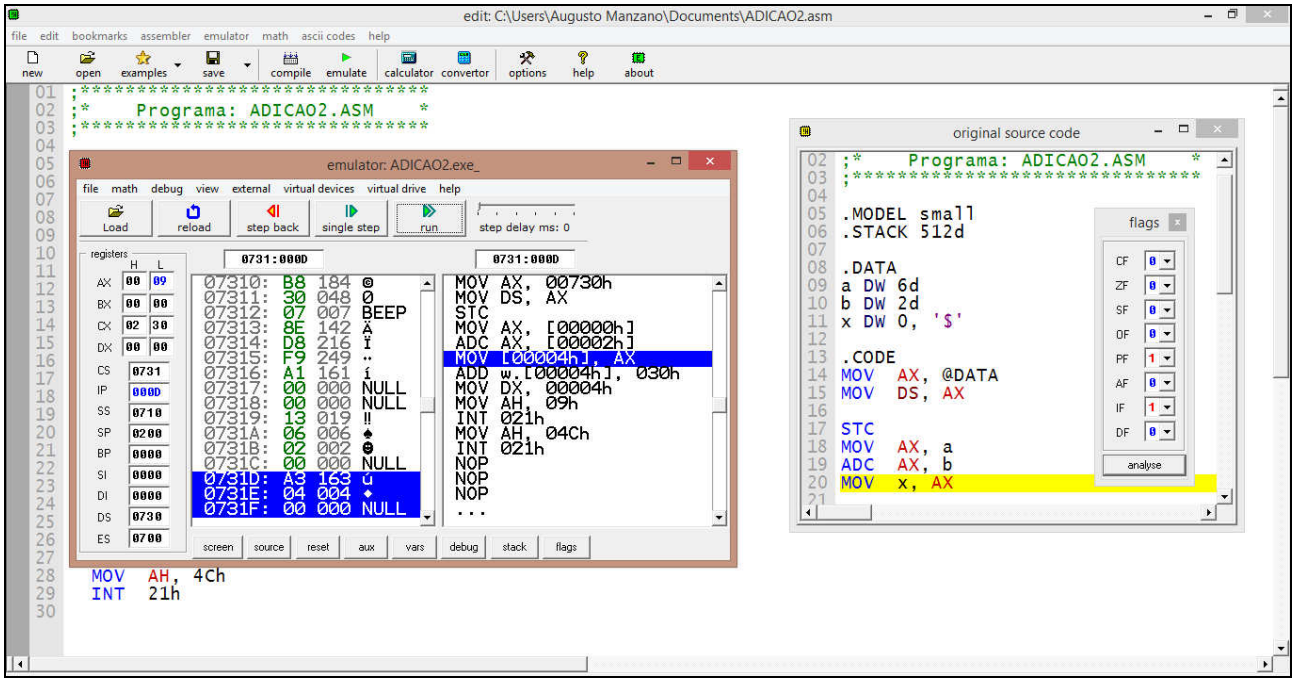

Figura 8.9 - Detalhes de ação do programa (quinta ação da tecla <F8>).

Observe também a identificação do quadro direito da tela **emulator** que apresenta marcada a linha de código **ADD w,[00004h], 030h**, demonstrando que o valor **030h** será adicionado ao conteúdo existente no endereço de deslocamento **0004h**.

Aperte a tecla de função **<F8>** pela sétima vez e observe os dados apresentados em tela, como na Figura 8.11.

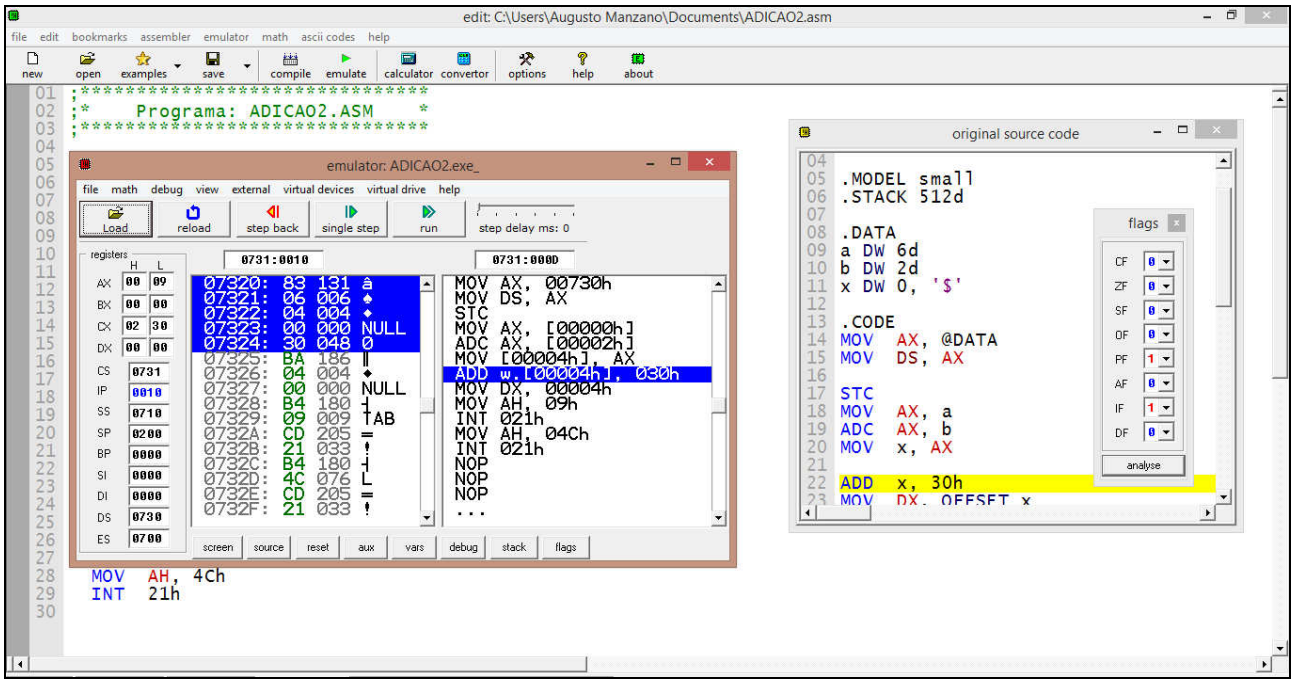

Figura 8.10 - Detalhes de ação do programa (sexta ação da tecla <F8>).

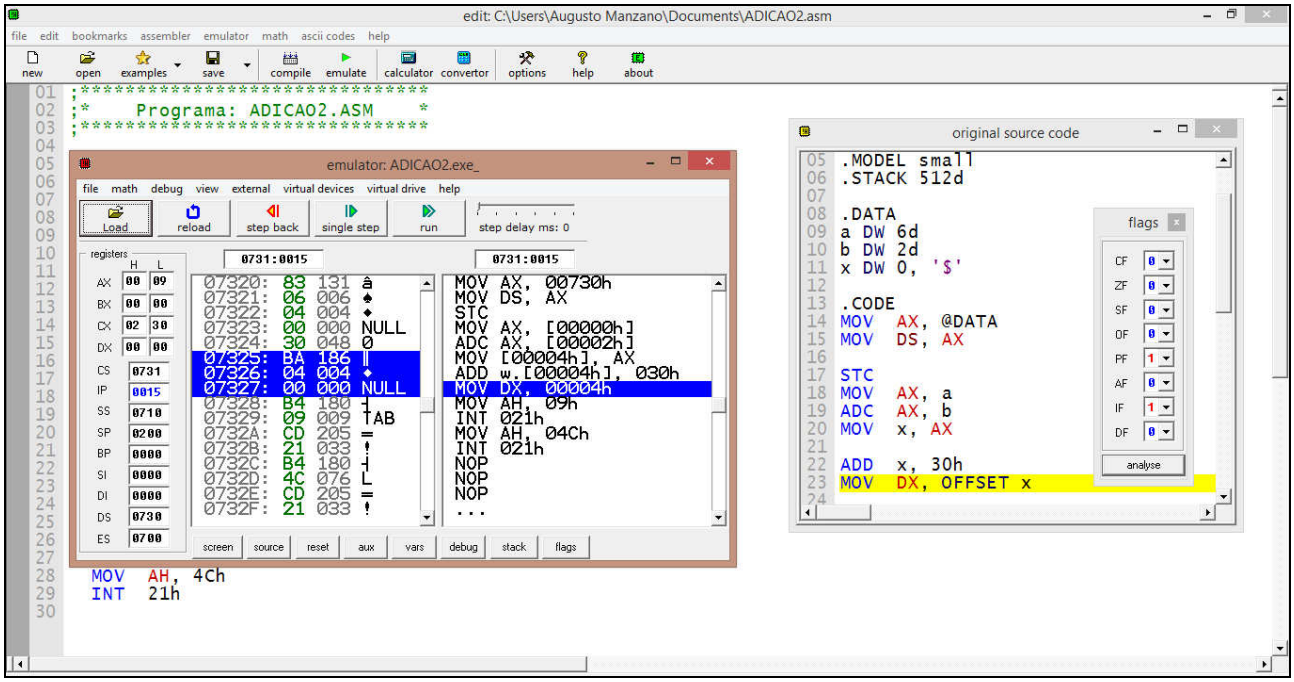

Figura 8.11 - Detalhes de ação do programa (sétima ação da tecla <F8>).

A linha de código **MOV DX, OFFSET x** (linha **23**) mostra que o endereço de deslocamento em que se encontra definida a variável **x** (**00004h**) será movimentado para o registrador geral **DX**. Veja os detalhes da área do quadro direito da tela **emulator** na Figura 8.12.

Na sequência acione a tecla de função **<F8>** pela oitava vez e observe os dados apresentados em tela, conforme a Figura 8.13.

A partir desse ponto o programa apresenta o resultado da operação na tela do monitor de vídeo, ou seja, o conteúdo existente no endereço de deslocamento **0018h**.

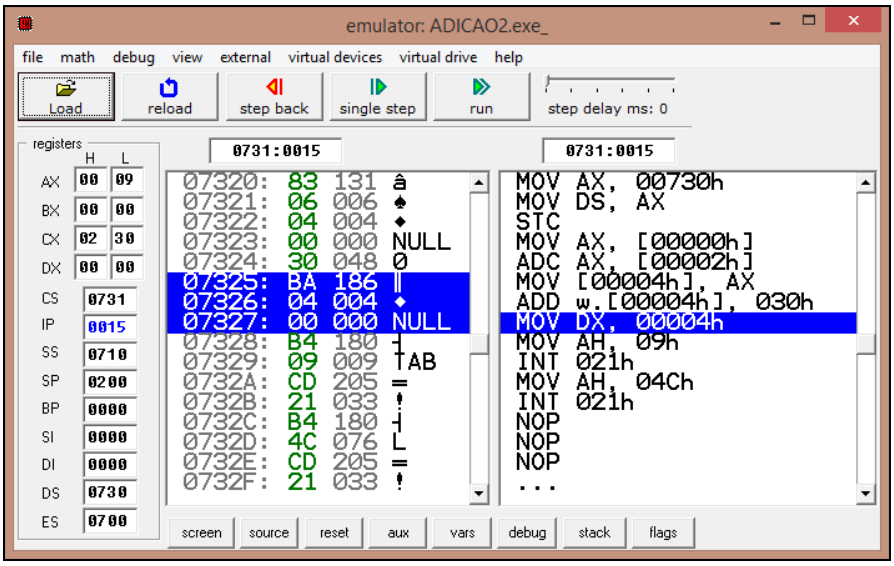

Figura 8.12 - Detalhes de ação do programa.

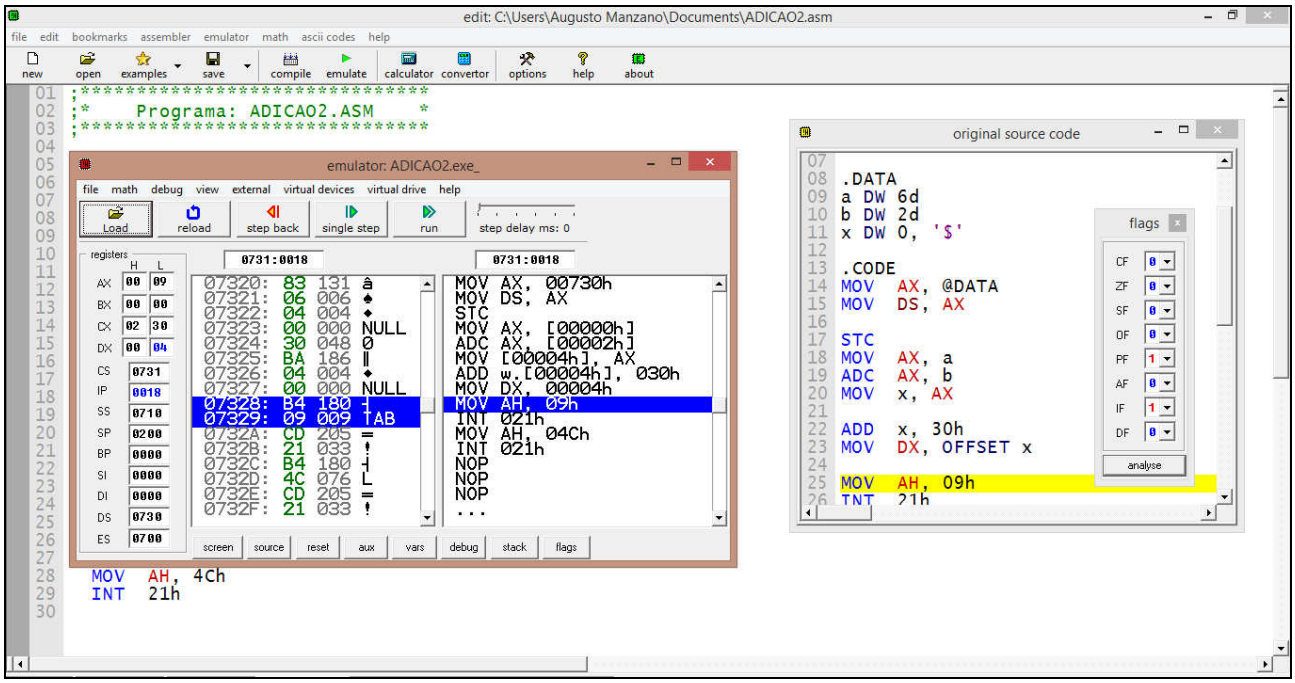

Figura 8.13 - Detalhes de ação do programa (oitava ação da tecla <F8>).

Em seguida, acione a tecla de função **<F8>** mais três vezes até que a caixa de diálogo com a mensagem de término seja apresentada, como indica a Figura 8.14. Para finalizar, pressione o botão **OK** da caixa de diálogo **message** e na janela **emulator: ADICAO2.exe_** acione o comando de menu **file/close the emulator**.

Em seguida feche todas as janelas do ambiente **emu8086**.

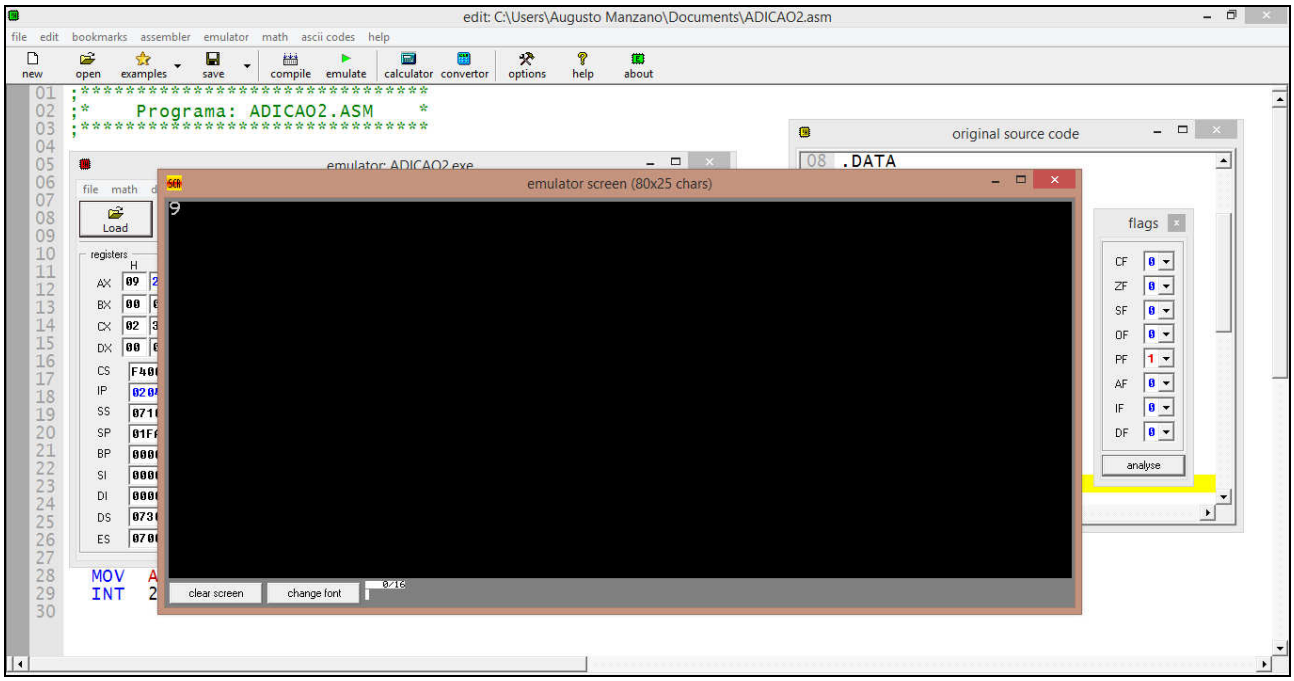

Figura 8.14 - Finalização da execução do programa.

8.3.2 - Subtração

As operações de subtração podem ocorrer com a utilização das instruções **SUB** ou **SBB**. A instrução **SUB** tem a mesma finalidade do operador aritmético "**–**" existente em outras linguagens de programação. Já a instrução **SBB** possui como diferencial a capacidade de subtrair, além dos valores existentes, o valor encontrado no registrador de estado **CF**.

O funcionamento lógico da instrução **SUB** será **SUB DESTINO, ORIGEM** (**DESTINO ← DESTINO – ORIGEM**).

```
SUB AX, 5d
```

No exemplo apresentado, o registrador geral **AX** está sendo subtraído com o valor decimal **5**. Se no registrador geral **AX** existir algum valor anterior, o valor **5** será subtraído do valor existente.

O funcionamento lógico da instrução **SBB** será **SBB DESTINO, ORIGEM** (**DESTINO ← DESTINO – ORIGEM – CF**).

```
SBB AX, 5d
```

No exemplo apresentado o registrador geral **AX** está sendo subtraído do valor decimal **5** mais o valor que estiver no registrador de estado **CF**, que pode ser **0** (zero) ou **1** (um).

Tome como base um programa que deva executar a equação **X ← A – B**, em que a variável **A** tem o valor decimal **6** e a variável **B**, o valor decimal **4**. Acompanhe o código do programa a seguir:

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 seguinte, gravando-o por meio dos comandos de menu **File/Save** com o nome **SUBTRAI1**, de forma que fique semelhante à Figura 8.15.

```
;*******************************
;*    Programa: SUBTRAI1.ASM   *
;*******************************

.MODEL small
.STACK 512d
```

```
.DATA
  a DW 6d
  b DW 4d
  x DW 0, '$'

.CODE
  MOV   AX, @DATA
  MOV   DS, AX

  MOV   AX, a
  SUB   AX, b
  MOV   x, AX

  ADD   x, 30h
  MOV   DX, OFFSET x

  MOV   AH, 09h
  INT   21h

  MOV   AH, 4Ch
  INT   21h
```

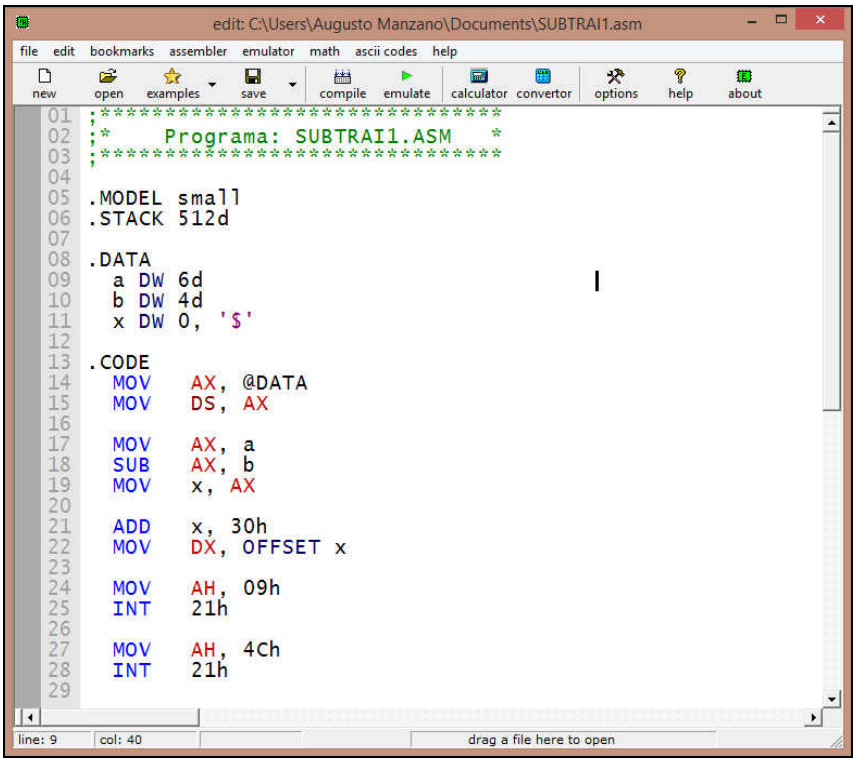

Figura 8.15 - Programa SUBTRAI1 na ferramenta emu8086.

A estrutura do programa de subtração é idêntica à do programa de adição. A execução ocorre de forma semelhante. Teste a execução passo a passo, observando os detalhes apresentados.

Em seguida, a título de ilustração de uso da instrução **SBB**, considere um programa que execute a equação **X ← A – B – CF**, em que a variável **A** tem o valor decimal **6**, a variável **B** o valor decimal **4** e **CF** é o valor existente no registrador de estado **CF**. Acompanhe o código do programa a seguir:

```
;*******************************
;*    Programa: SUBTRAI2.ASM   *
;*******************************

.MODEL small
.STACK 512d

.DATA
  a DW 6d
  b DW 4d
  x DW 0, '$'

.CODE
  MOV   AX, @DATA
  MOV   DS, AX

  STC
  MOV   AX, a
  SBB   AX, b
  MOV   x, AX

  ADD   x, 30h
  MOV   DX, OFFSET x

  MOV   AH, 09h
  INT   21h

  MOV   AH, 4Ch
  INT   21h
```

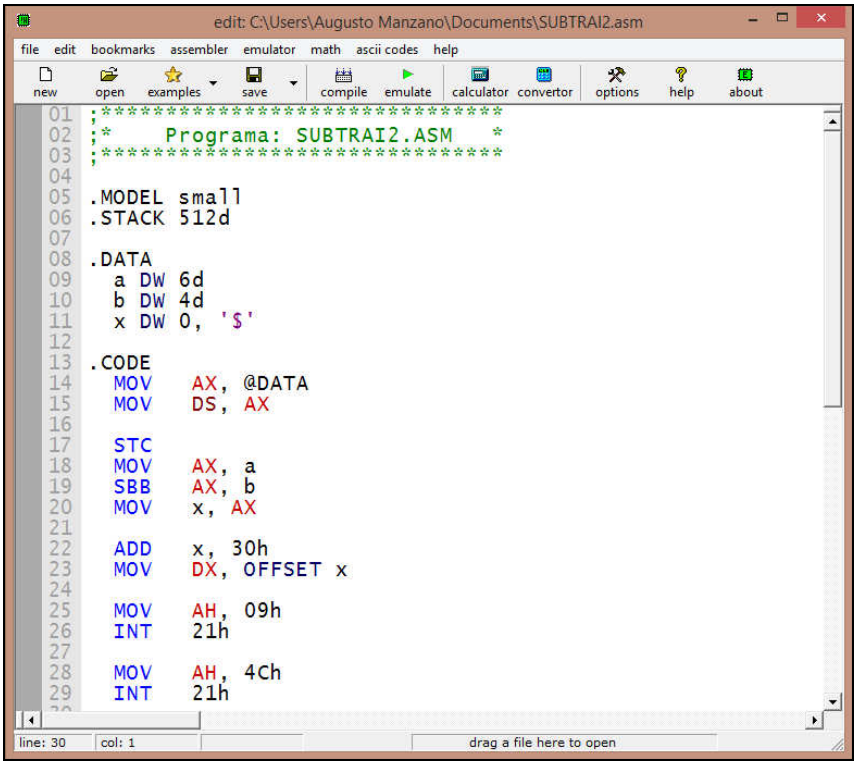

Figura 8.16 - Programa SUBTRA2 na ferramenta emu8086.

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 com os comandos de menu **file/save com** o nome **SUBTRAI2.asm**, de forma que fique semelhante à Figura 8.16.

Observe o detalhe de uso da instrução **STC** na linha **17** do programa que apresenta como resultado de cálculo o valor **1**.

8.3.3 - Divisão

As operações de divisão podem utilizar as instruções **DIV** (*divide*), a qual anteriormente já foi demonstrada, ou **IDIV** (*integer divide*). As instruções **DIV** e **IDIV** possuem a mesma finalidade do operador aritmético "**/**" existente em outras linguagens de programação.

As operações de divisão exigem que sejam tomados dois cuidados básicos:

- O primeiro cuidado é com relação à capacidade de armazenamento do número de *bits* em relação ao registrador em uso.
- O segundo é quanto ao uso do valor zero na indicação do divisor.

Observação

A operação de divisão pode ocorrer entre valores de 32 e 16 *bits* ou entre valores de 16 e 8 *bits*.

Caso um resultado obtido não caiba em um registrador ou o divisor seja zero, será gerado um erro de operação.

Após uma operação de divisão entre um dividendo de 16 *bits* (que deve estar armazenado no registrador geral **AX**) e um divisor de 8 *bits*, o valor do quociente é armazenado no registrador menos significativo **AL** do registrador geral **AX** e o resto da divisão é armazenado no registrador mais significativo **AH** do registrador geral **AX**. Assim sendo, a instrução:

```
DIV BL
```

Indica que em uma divisão de 16 *bits* por 8 *bits*, o registrador geral **AX** será armazenado com o valor de 16 *bits*, que será dividido pelo divisor armazenado no registrador menos significativo **BL** do registrador geral **BX**. O quociente estará armazenado no registrador menos significativo **AL** e o resto da divisão armazenado no registrador mais significativo **AH**. É como informar em uma linguagem de alto nível a linha de instrução **AL ← AX / BL**, em que **AH** é o resto de divisão.

Caso a divisão ocorra entre um dividendo de 32 *bits* (que deve estar armazenado no registrador geral **DX**) e um divisor de 16 *bits*, o valor do quociente será armazenado no registrador geral **AX** e o resto da divisão armazenado no registrador geral **DX**. Assim sendo, a instrução:

```
DIV BX
```

Indicaria que em uma divisão de 32 *bits* por 16 *bits*, o valor armazenado no par de registradores gerais **DX:AX** estaria com um valor de 32 *bits*, que será dividido pelo divisor armazenado no registrador geral **BX**. O quociente estaria no registrador geral **AX** e o resto da divisão armazenado no registrador geral **DX**. Seria como informar em uma linguagem de alto nível a linha de instrução **AX ← DX:AX / BX**, em que **DX** é o resto de divisão.

Observação

Quando são feitas operações matemáticas com operações de divisão, não se leva em conta o comportamento dos registradores de estado (*flags*).

Tome como base um programa que vai executar a equação **X ← A / B**, em que a variável **A** tem o valor decimal **9** de 16 *bits* (tipo **DW**) e a variável **B** o valor decimal **2** de 8 *bits* (tipo **DB**). Observe atentamente cada linha do código do programa a seguir:

```
;*******************************
;*    Programa: DIVIDE1.ASM    *
;*******************************

.MODEL small
.STACK 512d

.DATA
  a DW 9d
  b DB 2d
  x DB 0, '$'

.CODE
   MOV   AX, @DATA
   MOV   DS, AX

   MOV   AX, a
   MOV   BL, b
   DIV   BL

   MOV   x, AL
   ADD   x, 30h
   MOV   DX, OFFSET x

   MOV   AH, 09h
   INT   21h

   MOV   AH, 04Ch
   INT   21h
```

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 com os comandos de menu **file/save** com o nome **DIVIDE1** de forma que fique semelhante à Figura 8.17.

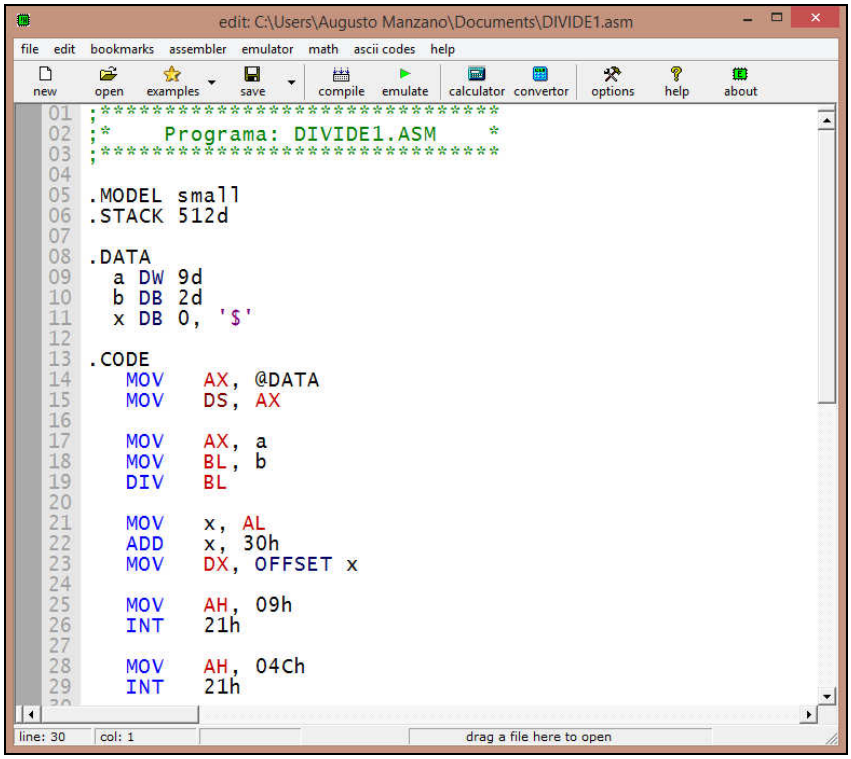

Figura 8.17 - Programa DIVIDE1 na ferramenta emu8086.

Conforme descrito anteriormente, considerando a divisão de valores de 16 *bits* (valor **9** definido para variável **a** do tipo **DW**) por um valor de 8 *bits* (valor **2** definido para a variável **b** do tipo **DB**), o valor do quociente estará no registrador menos significativo **AL** e o resto da divisão no registrador mais significativo **AH**.

Para verificar essa ocorrência, serão apresentados alguns detalhes da execução do programa passo a passo. Acione o comando de menu **file/compile and load in emulator**.

Acione a tecla de função **<F8>** por cinco vezes e observe na janela **emulate: DIVIDE1.com_** o resultado do quociente no registrador menos significativo **AL** e o resto da divisão no registrador mais significativo **AH**, como mostra a Figura 8.18.

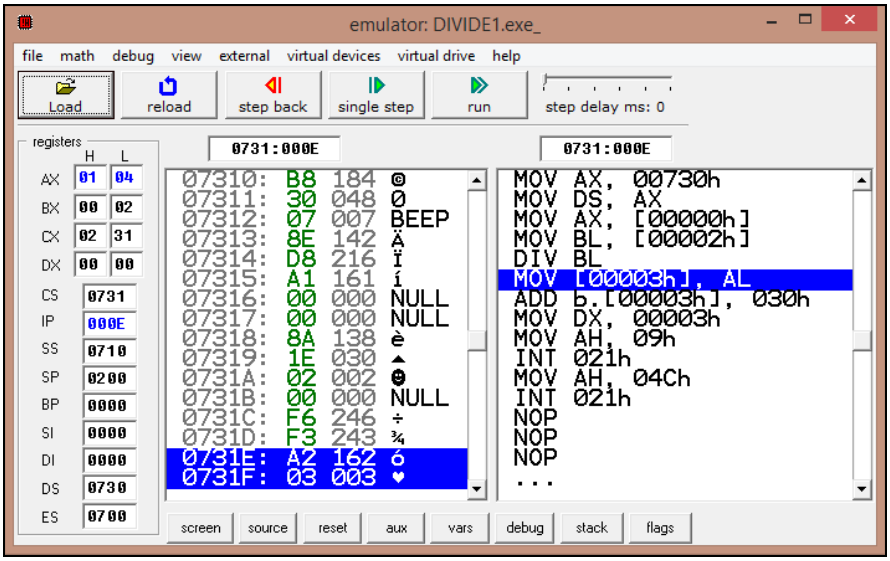

Figura 8.18 - Resultados do programa DIVIDE1.

Observe o valor **04h** do quociente no registrador menos significativo **AL**, resultado da divisão, e o valor **01h** do resto no registrador mais significativo **AH**. Em operações de divisão com valores hexadecimais, normalmente se leva em conta apenas o valor do quociente, desprezando o valor do resto. Haja vista se for executada a mesma operação no programa **Calculadora** do Windows.

Na sequência do programa acione a tecla de função **<F8>** algumas vezes ou acione o botão **run** até o programa apresentar o resultado da operação, quando deve ser acionado o botão **OK** da caixa de diálogo **message**.

Com o objetivo de demonstrar uma operação de divisão com valores do tipo *word*, considere um programa que deva executar a equação **X ← A / B**, em que a variável **A** possui o valor decimal **9** de 16 *bits* (tipo **DW**) e a variável **B** o valor decimal **2** de 16 *bits* (tipo **DW**). Acompanhe atentamente cada linha do código do programa a seguir:

```
;*******************************
;*    Programa: DIVIDE2.ASM    *
;*******************************

.MODEL small
.STACK 512d

.DATA
  a DW 9d
  b DW 2d
  x DB 0, '$'

.CODE
   MOV   AX, @DATA
   MOV   DS, AX

   MOV   AX, a
   MOV   BX, b
```

```
DIV   BX

MOV   x, AL
ADD   x, 30h
MOV   DX, OFFSET x

MOV   AH, 09h
INT   21h

MOV   AH, 04Ch
INT   21h
```

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 com os comandos de menu **file/save** com o nome **DIVIDE2**, de forma que fique semelhante à Figura 8.19.

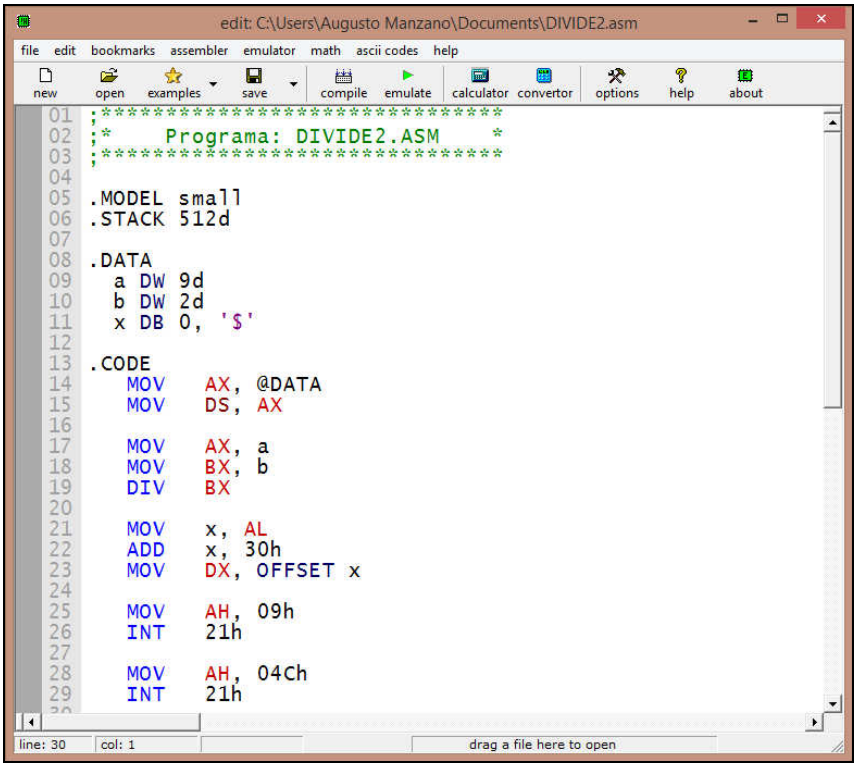

Figura 8.19 - Programa DIVIDE2 na ferramenta emu8086.

Conforme descrito anteriormente, considerando a divisão de valores de 16 *bits* (valor **9** definido para variável **a** do tipo **DW**) por um valor de 16 *bits* (valor **2** definido para a variável **b** do tipo **DW**), o valor do quociente no registrador menos significativo **AX** e o resto da divisão estará no registrador mais significativo **DX**.

Para verificar essa ocorrência, serão apresentados alguns detalhes da execução do programa passo a passo. Acione o comando de menu **assembler/ compile and load in emulator**.

Acione a tecla de função **<F8>** por cinco vezes e observe na janela **emulator: DIVIDE2.com_** o resul-tado do quociente no registrador geral **AL** (**AX**) e o resto da divisão no registrador geral **DL** (**DX**), como exibe a Figura 8.20.

Na sequência acione a tecla de função **<F8>** algumas vezes até o programa apresentar o resultado da operação, quando deve ser acionado o botão **OK** da caixa de diálogo **message**.

É importante salientar que a instrução **DIV** deve operar apenas com valores numéricos não sinalizados (valores positivos). Caso haja necessidade de efetuar operações de divisão com valores numéricos sinalizados (valores negativos), deve-se utilizar a instrução **IDIV**.

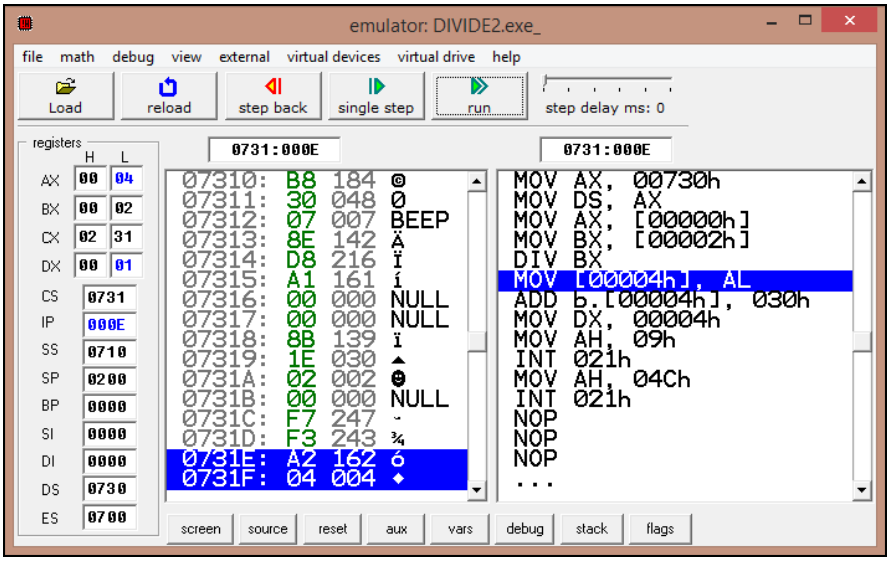

Figura 8.20 - Resultados do programa DIVIDE2.

Com o objetivo de demonstrar uma operação de divisão com a instrução **IDIV**, considere um programa que deva executar a equação **X** ← **A** / **B**, em que a variável **A** possui o valor decimal negativo **–9** de 16 *bits* (tipo **DW**) e a variável **B** o valor decimal **2** de 8 *bits* (tipo **DB**). Observe atentamente cada linha do código do programa em seguida:

```
;*******************************
;*    Programa: DIVIDE3.ASM    *
;*******************************

.MODEL small
.STACK 512d

.DATA
  a DW -9d
  b DB 2d
  x DB 0, '$'

.CODE
   MOV   AX, @DATA
   MOV   DS, AX

   MOV   AX, a
   MOV   BL, b
   IDIV  BL

   MOV   x, AL
   SUB   BX, BX
   MOV   BL, x

   MOV   AH, 02h
   MOV   DL, BL
   MOV   CL, 04h
   SHR   DL, CL
   ADD   DL, 30h
   CMP   DL, 39h
   JLE   valor1
   ADD   DL, 07h
```

```
valor1:
   INT   21h

   MOV   DL, BL
   AND   DL, 0Fh
   ADD   DL, 30h
   CMP   DL, 39h
   JLE   valor2
   ADD   DL, 07h

valor2:
   INT   21h

   MOV   AH, 4Ch
   INT   21h
```

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 com os comandos de menu **file/save** com o nome **DIVIDE3**. A Figura 8.21 mostra um trecho de como o programa deve ficar.

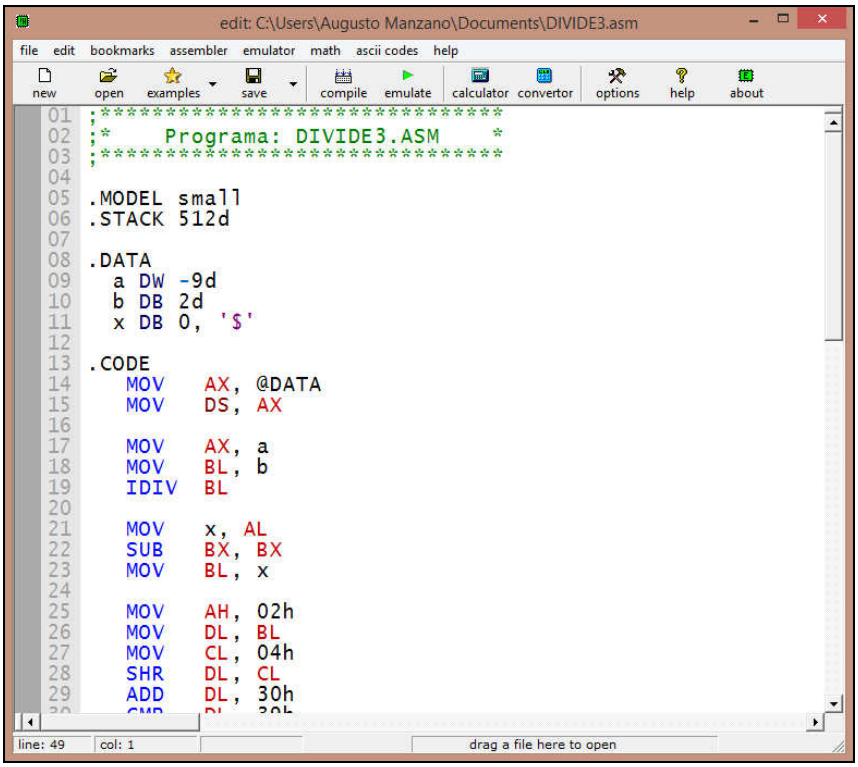

Figura 8.21 - Programa DIVIDE3 na ferramenta emu8086.

A operação de divisão de um dividendo negativo **–9** (que internamente será um valor hexadecimal) de 16 *bits* com um divisor **2** (também hexadecimal) de 8 *bits* resulta em um valor hexadecimal de duas posições, ou seja, o resultado da divisão dos valores hexadecimais **–9h / 2h** terá um quociente igual a **FCh** definido no registrador **AL** (**AX**). Para esta comprovação acione a tecla de função **<F5>** uma vez e a tecla de função **<F8>** cinco vezes. A Figura 8.22 mostra o resultado na tela da etapa de simulação **emulator: DIVIDE3.exe**_.

Na linha **17** é feita a transferência do valor armazenado na variável **a** para dentro do registrador geral **AX** por meio da instrução **MOV AX, a**. Observe que na linha **09** está sendo definido o tipo **DW** para armazenar o valor negativo (**–9d**). Ao ser processada essa linha, o registrador **AX** armazena o valor **FFF7h** (**–9** em decimal). Valores negativos para serem armazenados usam o tamanho *doubleword*.

Na linha **18** é feita a transferência do valor armazenado na variável **b** para dentro do registrador geral **BL** por meio da instrução **MOV BL, b**. Observe que na linha **10** está sendo definido o tipo **DB** para armazenar o valor **02h**. Ao ser processada essa linha, o registrador **BL** armazena então o valor **02h**, deixando o registrador **BX** com valor **0002h**.

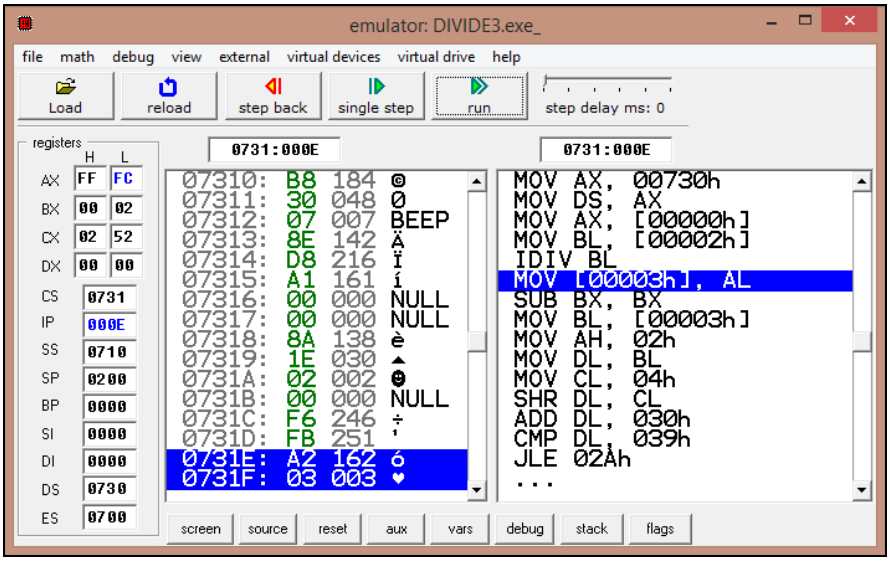

Figura 8.22 - Resultado de divisão no programa.

Na linha **19** é executada a instrução **IDIV BL**, a qual pega o valor do registrador **AX** e faz sua divisão com o valor do registrador **BL**, armazenando o quociente da operação no registrador **AL** e o resto da divisão no registrador **AH**. No caso da linha **19** o resultado obtido da divisão **FCh** é armazenado no registrador **AL**.

A instrução **IDIV** pode operar com valores do tipo *byte* ou *word*. Quando a operação de divisão envolver o uso do tipo *byte*, ocorre no registrador **AL** a atribuição da divisão do valor do registrador **AX** pelo valor do divisor que estará armazenado no registrador **BL** e se houver resto da divisão, esse valor será armazenado no registrador **AH**. Para a divisão que envolva o uso de valores do tipo *word*, ocorre no registrador **AX** a atribuição do valor da divisão que estará armazenado nos registradores **DX** e **AX** (**DX AX**) pelo valor do divisor que gera um quociente armazenado no registrador **AX** e se houver resto, será armazenado no registrador **DX**.

A partir da linha **21** são efetuadas as ações que possibilitam de certa forma apresentar o resultado na tela do monitor de vídeo.

O trecho da linha **21** até a linha **23** faz a transferência do valor do registrador menos significativo **AL** que contém o resultado da divisão efetuada na linha **19** para a variável **x** por meio de **MOV x, AL**. Depois por meio da linha **SUB BX, BX** faz a limpeza do registrador geral **BX**, ou seja, seu valor torna-se **00h** para que a sua parte menos significativa receba o valor armazenado na variável **x** (**MOV BL, x**). Neste caso o valor **FCh**, que é o resultado da operação de divisão, pois esse valor precisa ser apresentado na tela do monitor de vídeo, por esta razão o resultado está sendo preservado no registrador **BL** para seu tratamento no programa pelas linhas **26** e **37**.

Da linha **25** até a linha **34** são efetuadas as ações responsáveis pela apresentação do caractere **F** (primeiro caractere do valor **FCh**) que ocorre na linha **34** quando da execução da instrução **INT 21h** identificada após o rótulo **valor1** definido na linha **33**.

Da linha **36** até a linha **43** são efetuadas as ações responsáveis pela apresentação do caractere **C** (segundo caractere do valor **FCh**) que ocorre na linha **43** quando da execução da instrução **INT 21h** identificada após o rótulo **valor2** definido na linha **42**.

A definição do valor **02h** na linha **25** para o registrador **AH** é o estabelecimento da saída de um caractere para a interrupção **21h** executada nas linhas **34** e **43** por meio da instrução **INT 21h**.

A linha **26** faz a transferência do valor **FCh** (resultado da divisão) do registrador **BL** para o registrador **DL** por meio da instrução **MOV DL, BL**, para que o primeiro caractere da sequência numérica seja preparado para apresentação.

As linhas **27** e **28** efetuam o tratamento para apresentação de valores hexadecimais. Na linha **27** está sendo fornecido o valor **04h** para o registrador **CL** por meio da instrução **MOV CL, 04h**, isso fará com que ocorra o deslocamento de 4 *bits* (1 *nibble*) para a direita quando a linha **28** for executada. Na linha **28** a instrução **SHR DL, CL** fará o deslocamento para a direita a quantidade de *bits* informada a partir do valor armazenado no registrador **CL**, ou seja, fará o deslocamento para a direita do *nibble* representado pelo caractere **F** do valor **FCh** que se encontra definido no registrador **DL**. Assim sendo, o registrador **DL** passa a possuir o valor **0Fh** (**F** é o conteúdo extraído do valor **FCh**).

A linha **29** prepara por meio da execução da instrução **ADD DL, 30h** o conteúdo do registrador **DL** para apresentação na tela do monitor de vídeo, semelhante ao recurso já explanado no capítulo quatro para a apresentação de valores binários.

A linha **30** estabelece a comparação do valor armazenado no registrador **DL** com o valor **39h**. Se for o valor da comparação verdadeiro, o programa será desviado pela linha de código **31** para a linha **33** de forma que ocorra a apresentação do caractere **F**. Se a condição não for verdadeira, será executada antes da apresentação do caractere **F** a instrução da linha **32**.

A linha **31** por meio da instrução **JLE valor1** fará o desvio do programa para a linha marcada com o rótulo **valor1:** (linha **33**) quando a condição assinalada pela linha **30** for verdadeira; caso contrário, a ação dessa instrução não é executada e o programa continua a partir da linha **32**.

A linha **32** efetua a soma de sete *bits* para compor a representação numérica de um valor em formato hexadecimal. Essa ação é executada quando o conteúdo a ser apresentado estiver distante do valor de nove posições, por isso essa linha faz a adição do valor **07h** ao valor armazenado no registrador **DL** por meio da instrução **ADD DL, 07h**.

Após a apresentação do caractere **F** que compõe o valor **FCh** o programa fará algo idêntico para apresentar o caractere **C**. Lembre-se de que a linguagem de programação *Assembly* efetua entradas e saídas sempre um caractere por vez.

A linha **36** faz a movimentação do valor **FCh** armazenado no registrador **BL** para o registrador **DL**, como ocorreu com a instrução da linha **26**.

A linha de código **37** para preparar a apresentação do segundo caractere do valor **FCh** usa a instrução **AND DL, 0Fh**. A instrução **AND** faz a comparação em um valor *bit* a *bit* (modo binário) em dados do tipo *byte* ou *word*. Neste sentido, para extrair o segundo caractere do valor **FCh** que é sua parte direita faz-se a comparação do valor armazenado no registrador **DL** com o valor **0Fh**. Lembre-se de que esse recurso foi apresentado no capítulo quatro. Após essa ação o registrador **DL** apresenta como valor armazenado o conteúdo **0Ch**, que é o segundo valor a ser apresentado.

As linhas de **38** a **43** efetuam ação idêntica à ação já descrita para as linhas de **29** a **34**. As linhas **45** e **46** são responsáveis pelo término da execução do programa, como já foi explicado.

8.3.4 - Multiplicação

As operações de multiplicação podem ser feitas com as instruções **MUL** (*multiply*) ou **IMUL** (*integer multiply*), as quais têm a mesma finalidade do operador aritmético "*****" existente em outras linguagens de programação.

Para realizar operações de multiplicação, é necessário levar em conta o fato de trabalhar com registradores de 8 ou 16 *bits*. Elas são mais complexas que as de adição e subtração.

Se for feita a multiplicação entre dois registradores de 16 *bits,* obter-se-á um resultado de 32 *bits* (ou seja, um *double word*). Isso está acima da capacidade de trabalho do processador 8086/8088, e neste caso serão utilizados automaticamente os registradores gerais **DX** e **AX** para armazenar um valor de 32 *bits*. O registrador geral **DX** armazena o *word* mais significativo e o registrador geral **AX**, o *word* menos significativo do valor de 32 *bits* resultante após a multiplicação.

Imagine a necessidade de multiplicar o valor hexadecimal **AA1Fh** (43.551 decimal) pelo valor hexadecimal **FF2Ah** (65.322 decimal) que resultaria no valor **A990CA16h** (2.844.838.422 decimal). Neste caso a porção mais significativa (**A990h**) seria armazenada no registrador geral **DX** e a porção menos significativa (**CA16h**) no registrador geral **AX**.

É fundamental considerar também que as operações de multiplicação serão sempre imputadas sobre os registradores acumuladores **AL** ou **AX**, dependendo de os valores serem de 8 ou 16 *bits*. Desta forma a linha de instrução:

```
MUL BL
```

Indica que o registrador menos significativo **BL** está sendo multiplicado pelo valor existente no registrador menos significativo **AL** e armazenando o resultado obtido no registrador geral **AX**. Seria como informar em uma linguagem de alto nível a linha de instrução **AX ← AL * BL**, considerando que os valores trabalhados são de 8 *bits*.

Caso venha a utilizar valores de 16 *bits*, o resultado será armazenado no par de registradores gerais **DX:AX**. Assim sendo, a instrução:

```
MUL BX
```

indicaria que o registrador geral **BX** está sendo multiplicado pelo valor existente no registrador geral **AX** e armazenando o resultado obtido nos registradores **DX** e **AX**. Seria como informar em uma linguagem de alto nível a linha de instrução **DX:AX ← AX * CX**.

Observação

Quando se fazem operações de multiplicação, não se leva em conta o comportamento dos registradores de estado *(flags)*.

Tome como base um programa que deva executar a equação **X ← A * B**, em que a variável **A** possui o valor decimal **5** de 16 *bits* (tipo **DW**) e a variável **B** o valor decimal **3** de 8 *bits* (tipo **DB**). Observe atentamente cada linha do código do programa a seguir:

```
;*******************************
;*    Programa: MULTIP1.ASM    *
;*******************************

.MODEL small
.STACK 512d

.DATA
  a DW 5d
  b DB 3d
  x DB 0, '$'

.CODE
   MOV   AX, @DATA
   MOV   DS, AX

   MOV   AX, a
   MOV   BL, b
   MUL   BL

   MOV   x, AL
   SUB   BX, BX
   MOV   BL, x

   MOV   AH, 02h

   MOV   DL, BL
   MOV   CL, 04h
   SHR   DL, CL
   ADD   DL, 30h
   CMP   DL, 39h
   JLE   valor1
   ADD   DL, 07h

valor1:
   INT   021h

   MOV   DL, BL
   AND   DL, 0Fh
   ADD   DL, 30h
   CMP   DL, 39h
   JLE   valor2
   ADD   DL, 07h

valor2:
   INT   21h
```

```
MOV   AH, 04Ch
INT   21h
```

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 com os comandos de menu **file/save** com o nome **MULTIP1**. A Figura 8.23 mostra um trecho de como o programa deve ficar.

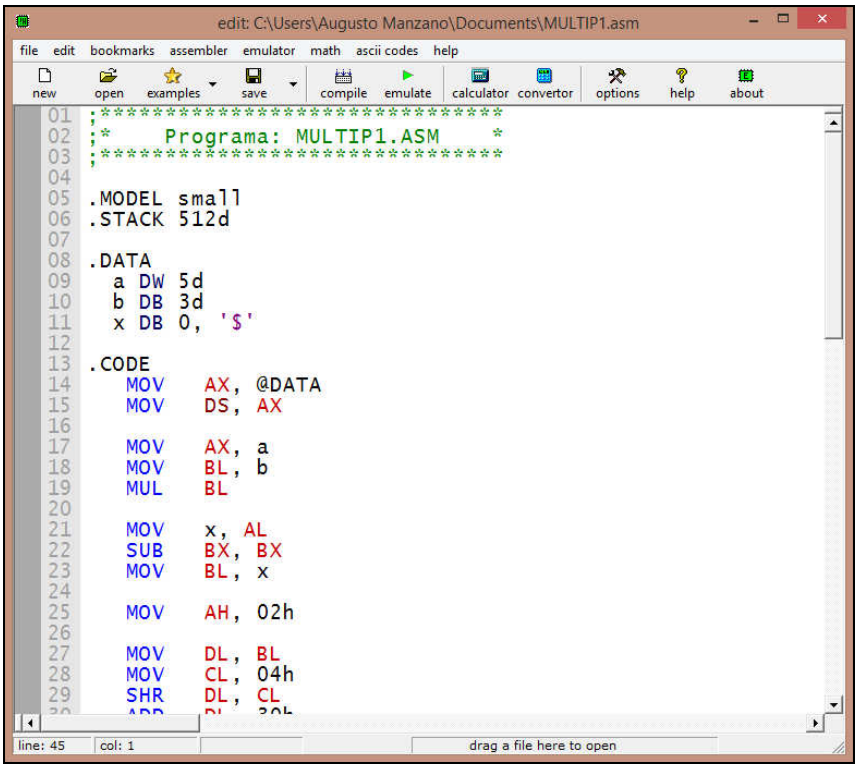

Figura 8.23 - Programa MULTIP1 na ferramenta emu8086..

O mecanismo de ação do programa **MULTIP1** é semelhante ao funcionamento do programa **DIVIDE3**, com exceção da linha de código **24** que apresenta o uso da instrução **MUL BL**.

A instrução **MUL** pode operar com valores do tipo *byte* ou *word*. Quando a operação de multiplicação envolver o uso do tipo *byte*, ocorre no registrador **AX** a atribuição da multiplicação do valor do registrador **AL** pelo valor do registrador **BL** como seu operando, como indicado nas linhas de código de **17** até **19**. Para multiplicação que envolva o uso de valores do tipo *word*, ocorre nos registradores **DX** e **AX** (**DX AX**) a atribuição do valor da multiplicação do valor armazenado no registrador **AX** pelo valor definido como operando.

A operação de multiplicação de um valor **5** (que internamente será um valor hexadecimal) de 16 *bits* com um valor **3** (também hexadecimal) de 8 *bits* resulta em um valor hexadecimal de duas posições, ou seja, o resultado do produto dos valores hexadecimais **5h * 3h** é **0Fh** (acione a tecla de função **<F5>** uma vez e a tecla de função **<F8>** cinco vezes para comprovar este resultado). A Figura 8.24 mostra o resultado obtido após a operação indicada.

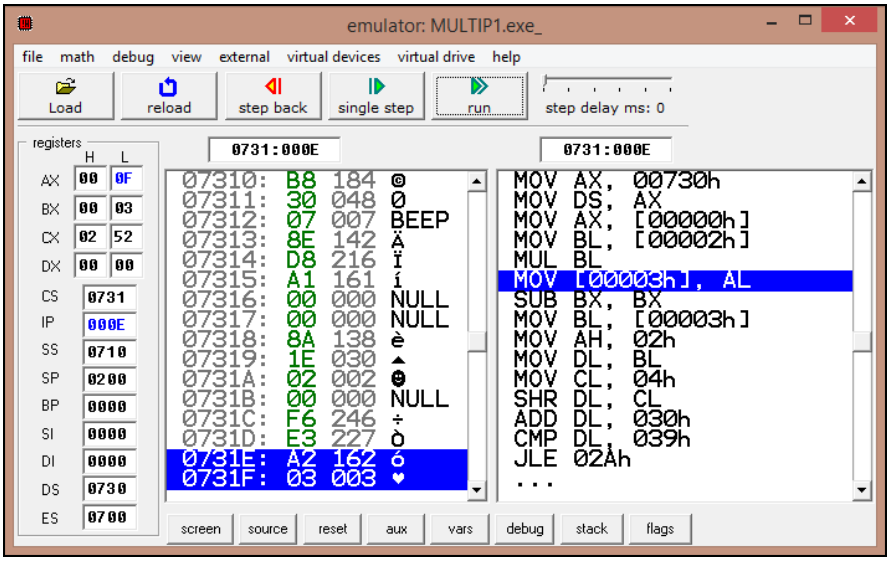

Figura 8.24 - Resultado de multiplicação no programa.

Considere como exemplo outro programa que deva apresentar o resultado da multiplicação de dois valores de 16 *bits*. Ele deve executar a equação **YX ← A * B**, em que a variável **A** possui o valor decimal **43551** (**AA1Fh**) de 16 *bits* (tipo **DW**) e a variável **B** o valor decimal **65322** (**FF2Ah**) de 16 *bits* (tipo **DW**). Observe atentamente cada linha do código do programa a seguir:

```
;*******************************
;*    Programa: MULTIP2.ASM    *
;*******************************

.MODEL small
.STACK 512d

.DATA
  a DW 43551d
  b DW 65322d
  x DW 0
  y DW 0, '$'

.CODE
   MOV   AX, @DATA
   MOV   DS, AX

   MOV   AX, a
   MOV   BX, b
   MUL   BX

   MOV   x, DX
   MOV   y, AX

   SUB   BX, BX
   MOV   BX, x
   CALL  valor1
   CALL  valor2

   SUB   BX, BX
   MOV   BX, y
   CALL  valor1
   CALL  valor2

   MOV   AH, 4Ch
   INT   21h
```

```
saida PROC NEAR
   ADD   DL, 30h
   CMP   DL, 39h
   JLE   valor
   ADD   DL, 07h
   valor:
     INT   21h
   RET

saida ENDP

valor1 PROC NEAR
   MOV   AH, 02h
   MOV   DL, BH
   MOV   CL, 04h
   SHR   DL, CL
   CALL  saida
   MOV   DL, BH
   AND   DL, 0Fh
   CALL  saida
   RET
valor1 ENDP

valor2 PROC NEAR
   MOV   AH, 02h
   MOV   DL, BL
   MOV   CL, 04h
   SHR   DL, CL
   CALL  saida
   MOV   DL, BL
   AND   DL, 0Fh
   CALL  saida
   RET
valor2 ENDP
```

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 com os comandos de menu **file/save** com o nome **MULTIP2**. A Figura 8.25 mostra um trecho de como o programa deve ficar.

Esse programa utiliza muitos dos conceitos já abordados, entre eles a movimentação de *bits*, o uso de proced-mentos e saltos condicionais. Note o trecho de código definido entre as linhas **39** e **46**, como apresentado em seguida:

```
saida PROC NEAR
   ADD   DL, 30h
   CMP   DL, 39h
   JLE   valor
   ADD   DL, 07h
   valor:
     INT   21h
   RET
saida ENDP
```

Para a definição desse procedimento (sub-rotina) estão sendo utilizadas as diretivas **PROC** e **ENDP**, que definem, respectivamente, o início (**PROC** - **PROC**edure) e o fim (**ENDP** - **END P**rocedure) de um procedimento. A diretiva **ENDP** é utilizada por duas razões: a primeira razão por ela identificar o ponto de início (primeira instrução) do procedimento após a diretiva **PROC**; a segunda razão pelo fato de sinalizar ao programa *Assembler* o fim do trecho do procedimento em operação.

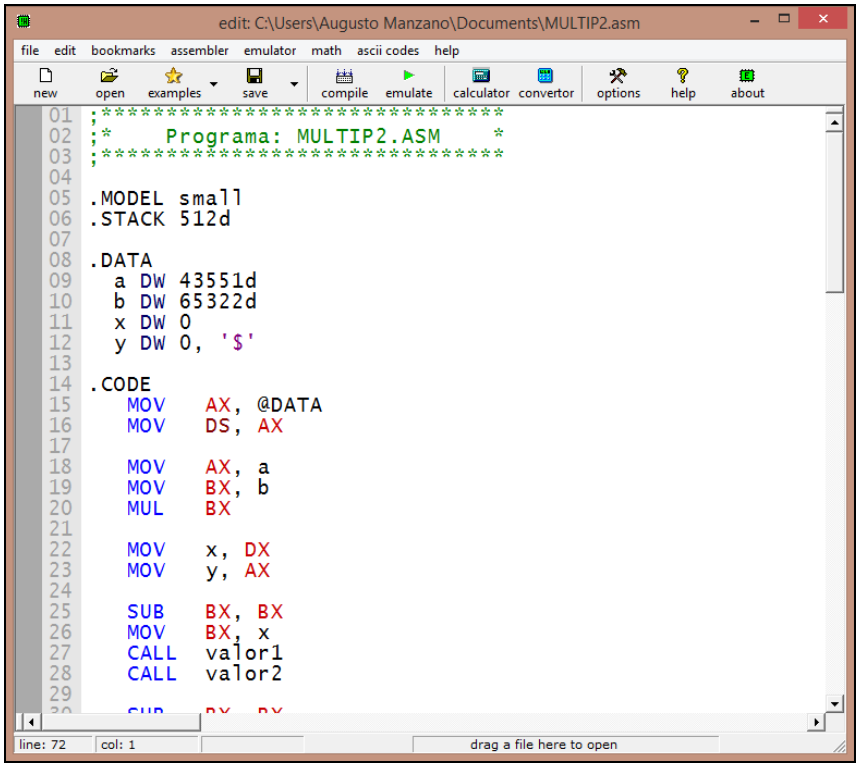

Figura 8.25 - Programa MULTIP2 na ferramenta emu8086.

O parâmetro opcional **NEAR** definido à frente da diretiva **PROC** estabelece para o programa *Assembler* (neste caso **emu8086**) informações sobre o uso e forma de acesso aos segmentos de memória pelo programa em execução. Dependendo da forma de acesso ao endereçamento de memória, pode também ser utilizado o parâmetro **FAR** após a diretiva **PROC**.

O programa *Assembler* utiliza a informação do parâmetro definido após a diretiva **PROC** para estabelecer a forma de acesso que ocorrerá na memória por meio de um procedimento. Esse acesso pode ocorrer no mesmo segmento de memória quando usada a diretiva **NEAR** que representa um segmento próximo, ou em segmentos diferentes do local em que o programa está definido por meio da diretiva **FAR** que representa um segmento distante. Mais adiante (próximo tópico) este tema será discutido com um pouco mais de detalhamento.

A rotina de programa tem por finalidade imprimir um caractere hexadecimal na tela do monitor de vídeo, levando em consideração a faixa numérica de **0** (zero hexadecimal) até **9** (nove hexadecimal) e a faixa de **A** (dez hexadecimal) até **F** (quinze hexadecimal) de acordo com o código da tabela ASCII.

Nas linhas **50** até **60** e nas linhas **62** até **72** encontram-se, respectivamente, os procedimentos **valor1** e **valor2** que são responsáveis pela obtenção do primeiro e do segundo caracteres hexadecimais que serão armazenados no registrador **DX**.

```
valor1 PROC NEAR
   MOV   AH, 02h
   MOV   DL, BH
   MOV   CL, 04h
   SHR   DL, CL
   CALL  saida
   MOV   DL, BH
   AND   DL, 0Fh
   CALL  saida
   RET
valor1 ENDP
```

```
valor2 PROC NEAR
   MOV   AH, 02h
   MOV   DL, BL
   MOV   CL, 04h
   SHR   DL, CL
   CALL  saida
   MOV   DL, BL
   AND   DL, 0Fh
   CALL  saida
   RET
valor2 ENDP
```

Com relação à parte do programa principal, ele amplifica a apresentação de valores hexadecimais de dois dígitos para oito dígitos. Observe o trecho de código seguinte:

```
.CODE
   MOV   AX, @DATA
   MOV   DS, AX

   MOV   AX, a
   MOV   BX, b
   MUL   BX

   MOV   x, DX
   MOV   y, AX

   SUB   BX, BX
   MOV   BX, x
   CALL  valor1
   CALL  valor2

   SUB   BX, BX
   MOV   BX, y
   CALL  valor1
   CALL  valor2

   MOV   AH, 4Ch
   INT   21h
```

Após a multiplicação (das linhas **19** até **21**) são transferidos para as variáveis **x** e **y,** respectivamente, os valores do resultado da multiplicação armazenados nos registradores gerais **DX** e **AX**. Isso é necessário, pois os registradores devem estar disponíveis para a operação de processamento do programa.

Na sequência são definidos dois trechos de código, um existente entre as linhas **26** e **29** e outro no trecho das linhas **31** e **34**. O trecho da linha **26** até a linha **29** manipula os valores da variável **x** (antigo conteúdo do registrador geral **DX**) e pelas chamadas de procedimento com a instrução **CALL** apresenta os primeiros quatro caracteres hexadecimais. Por meio do trecho de código da linha **31** até a linha **34** manipula os dados da variável **y** (antigo conteúdo do registrador geral **AX**).

A operação de multiplicação dos valores **43551d** e **65322d** fornece como resultado um valor decimal **2.844.838.422** (ou seu equivalente **A990CA16** em hexadecimal). Para ver esse resultado, acione a tecla de função **<F5>** e em seguida acione a tecla de função **<F9>** ou use o botão **Run**. A Figura 8.26 mostra o resultado obtido após a operação indicada.

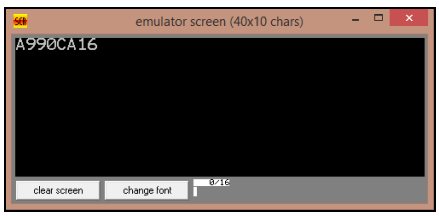

Figura 8.26 - Resultado de multiplicação no programa.

Em seguida considere como exemplo um programa que deva apresentar o resultado da multiplicação de dois valores de 16 *bits* utilizando a instrução **IMUL**. O programa deve executar a equação **YX ← A * B**, em que a variável **A** possui o valor decimal negativo **–32.700** (**8044h**) de 16 *bits* (tipo **DW**) e a variável **B** o valor decimal **25** (**19h**) de 16 *bits* (tipo **DW**), que resulta no valor decimal **–81.7500** (**FFF386A4h**). Observe atentamente cada linha do código do programa a seguir:

```
;******************************
;*    Programa: MULTIP3.ASM    *
;******************************

.MODEL small
.STACK 512d

.DATA
  a DW -32700d
  b DW 25d
  x DW 0
  y DW 0, '$'

.CODE
  MOV   AX, @DATA
  MOV   DS, AX

  MOV   AX, a
  MOV   BX, b
  IMUL  BX
  MOV   x, DX
  MOV   y, AX

  SUB   BX, BX
  MOV   BX, x
  CALL  valor1
  CALL  valor2

  SUB   BX, BX
  MOV   BX, y
  CALL  valor1
  CALL  valor2

  MOV   AH, 4Ch
  INT   021h

saida PROC NEAR
  ADD   DL, 30h
  CMP   DL, 39h
  JLE   valor
  ADD   DL, 07h
  valor:
   INT   21h
   RET
saida ENDP

valor1 PROC NEAR
  MOV   AH, 02h
  MOV   DL, BH
  MOV   CL, 04h
  SHR   DL, CL
  CALL  saida
  MOV   DL, BH
  AND   DL, 0Fh
  CALL  saida
  RET
```

```
valor1 ENDP

valor2 PROC NEAR
  MOV   AH, 02h
  MOV   DL, BL
  MOV   CL, 04h
  SHR   DL, CL
  CALL  saida
  MOV   DL, BL
  AND   DL, 0Fh
  CALL  saida
  RET
valor2 ENDP
```

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 com os comandos de menu **file/save** com o nome **MULTIP3**. A Figura 8.27 mostra um trecho de como o programa deve ficar.

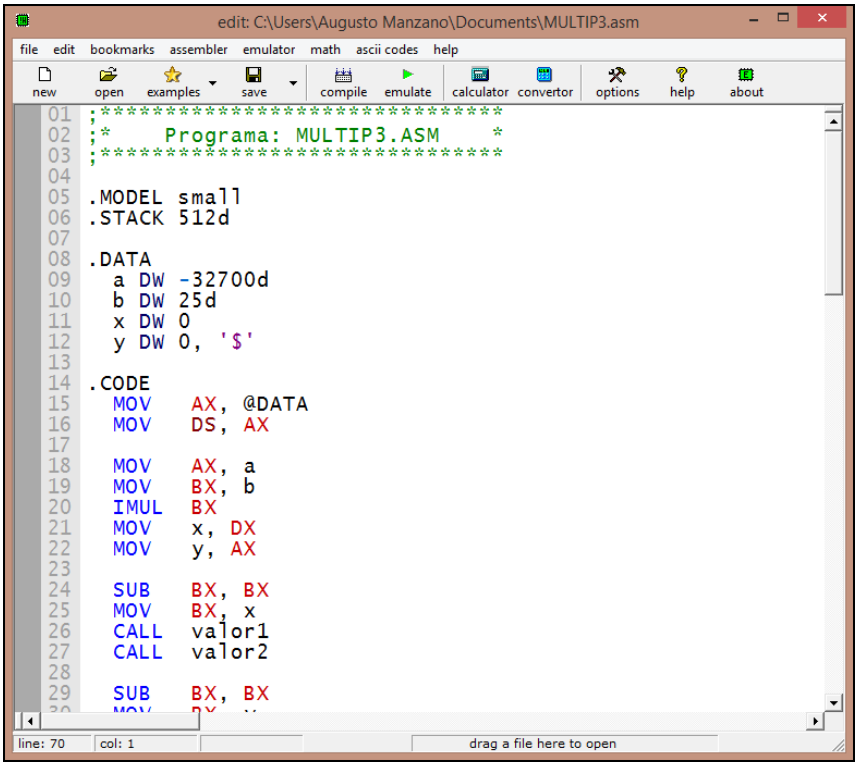

Figura 8.27 - Programa MULTIP3 na ferramenta Emu8086.

Os programas **MULTIP3** e **MULTIP2** são semelhantes. A diferença está na definição dos valores para as variáveis **a** e **b**, e também no uso do comando **IMUL** na linha **25** do programa.

A operação de multiplicação dos valores **–32700d** e **25d** fornece como resultado um valor decimal negativo **–817.500** (ou seu equivalente **FFF386A4** em hexadecimal). Para ver esse resultado, acione a tecla de função **<F5>** e em seguida acione a tecla de função **<F9>** ou use o botão **Run**. A Figura 8.28 mostra o resultado obtido após a operação indicada.

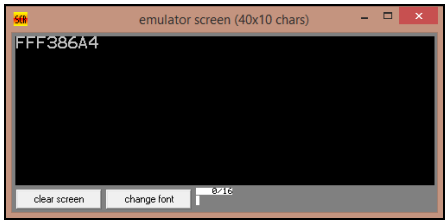

Figura 8.28 - Resultado de multiplicação no programa.

8.4 - Procedimento próximo x procedimento distante

O conceito de procedimento próximo ou distante está associado ao uso do tipo de parâmetro após a definição da diretiva **PROC** que pode ser **NEAR** (próximo) ou **FAR** (distante), dependendo de como se deseja acessar o endereçamento de memória.

O parâmetro **NEAR** (que pode ser omitido) é utilizado quando o acesso de um procedimento na memória ocorre no mesmo segmento onde se encontra o código de programa em execução (chamada direta intrassegmento), ou seja, o acesso ocorre apenas com a alteração do valor do registrador de ponteiro **IP** sem que ocorra a alteração do registrador de segmento **CS**. É importante levar em conta que cada segmento de memória está limitado a um tamanho de 64 *KBytes*, o que inviabiliza o desenvolvimento de grandes programas. Ao término de execução de um procedimento **NEAR** o retorno a parte chamadora da subrotina é efetuado com a instrução **RET**.

O parâmetro **FAR** (deve obrigatoriamente ser declarado) é utilizado quando o acesso a um procedimento na memória ocorre em um endereço de segmento diferente do endereço de segmento em que se encontra o código do programa em execução (chamada direta intersegmento), ou seja, ocorre a alteração do valor dos registradores **IP** e **CS**. Desta forma, torna-se possível ultrapassar a barreira do tamanho de 64 *KBytes* de cada segmento de memória. Ao término de execução de um procedimento **FAR** o retorno a parte chamadora da sub-rotina é efetuado com a instrução **RETF**.

A execução de um procedimento com parâmetro **NEAR** é mais simples que a execução de um procedimento com parâmetro **FAR**. Note que o parâmetro **NEAR** manipula apenas o registrador de ponteiro **IP**, enquanto o parâmetro **FAR** manipula os registradores **IP** e **CS**. Assim sendo, o parâmetro **NEAR** coloca na pilha um único *word* contendo o endereço do segmento **IP**, enquanto o parâmetro **FAR** coloca na pilha dois *words*, sendo um valor para o registrador de ponteiro **IP** e o outro valor para o registrador de segmento **CS**.

Os microprocessadores padrão 8086/8088 operam com base em uma arquitetura de memória segmentada, onde somente é possível acessar um segmento de memória de 64 KB por vez. É devido a esta característica que se tem a definição de procedimentos com parâmetro **NEAR** ou **FAR**. A Figura 8.29 exemplifica o uso conceitual de procedimentos **NEAR** e **FAR**.

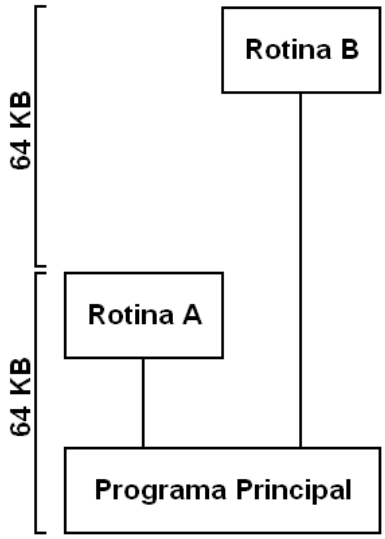

Figura 8.29 - Procedimentos NEAR e FAR.

O procedimento **Rotina A** está situado no mesmo segmento de memória em que se encontra definido o **Programa Principal**. Neste caso o procedimento **Rotina A** é um procedimento do tipo **NEAR** (próximo), pois para acessar o procedimento **Rotina A** não é necessário mudar o endereço do registrador de segmento **CS** uma vez que o registrador de segmento **CS** aponta para o mesmo segmento de memória onde está o **Programa Principal**, o que muda apenas é o endereço do registrador de ponteiro **IP**. Neste caso o valor do deslocamento muda, mas o valor do segmento permanece inalterado.

O procedimento **Rotina B** está situado em um segmento de memória diferente do segmento de memória onde se encontra o **Programa Principal**. Neste caso o procedimento **Rotina B** é um procedimento **FAR** (distante), pois para o **Programa Principal** acessar o procedimento **Rotina B** é necessário mudar o valor do endereço do registrador de segmento **CS** e o valor do registrador de ponteiro **IP**.

O parâmetro **NEAR** de procedimentos é utilizado em programas do tipo **.COM**, pois os programas desse tipo são sempre escritos no mesmo segmento de memória e o parâmetro **FAR** é utilizado em programas do tipo **.EXE**, como pode ser visto em um pequeno exemplo no capítulo nove desta obra, pois esses podem ser feitos em mais de um segmento.

Anotações