

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 9

## SALTOS, DECISÕES E REPETIÇÕES

Anteriormente foram apresentados superficialmente alguns desvios, laços e saltos em exemplos de programas aplicados. Este capítulo explica mais detalhadamente instrução de salto incondicional e condicional, tomada de decisões (simples e composta), instruções lógicas, laços de repetição e utilização de rotinas do tipo macro, além de apresentação de valores numéricos em notação decimal com o uso da pilha.

### 9.1 - Salto incondicional

Os saltos incondicionais estão associados ao desvio do fluxo de programa de um determinado ponto do código para outro ponto do mesmo código. Isso é conseguido com o comando de salto **JMP** (*jump*) utilizado e demonstrado anteriormente em várias ocasiões, similar ao comando GOTO encontrada em várias linguagens de programação de computadores de alto nível.

Esse tipo de salto deve ser usado com muita cautela, pois o uso excessivo pode tornar o código de programa confuso. Procure usá-lo o mínimo possível. Dê preferência à utilização de procedimentos. Assuma uma postura estruturada de programação. A título de ilustração considere o código de programa seguinte com o uso de saltos com o comando **JMP**:

```
;*****
;* Programa: MENSAGEM4.ASM *
;*****

org 100h

.DATA
mensagem1 DB 'Mensagem 1', 0Dh, 0Ah, 24h
mensagem2 DB 'Mensagem 2', 0Dh, 0Ah, 24h
mensagem3 DB 'Mensagem 3', 0Dh, 0Ah, 24h

.CODE
JMP salto3

salto1:
LEA DX, mensagem1
CALL escreva
JMP saida

salto2:
LEA DX, mensagem2
CALL escreva
JMP salto1
```

```

salto3:
    LEA    DX, mensagem3
    CALL   escreva
    JMP    salto2

saída:
    INT   20h

escreva PROC NEAR
    MOV   AH, 09h
    INT   21h
    RET
escreva ENDP

```

Execute no programa **emu8086** o comando de menu **file/new/com template**, aione 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 **MENSAGEM4**, de forma que fique semelhante à imagem da Figura 9.1.

```

edit: C:\Users\Augusto Manzano\Documents\MENSAGEM.asm
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about
01 :***** Programa: MENSAGEM4.ASM *****
02 ;*****
03 ;*****
04
05 org 100h
06
07 .DATA
08 mensagem1 DB 'Mensagem 1', 0Dh, 0Ah, 24h
09 mensagem2 DB 'Mensagem 2', 0Dh, 0Ah, 24h
10 mensagem3 DB 'Mensagem 3', 0Dh, 0Ah, 24h
11
12 .CODE
13 JMP salto3
14
15 salto1:
16     LEA    DX, mensagem1
17     CALL   escreva
18     JMP    saída
19
20 salto2:
21     LEA    DX, mensagem2
22     CALL   escreva
23     JMP    salto1
24
25 salto3:
26     LEA    DX, mensagem3
27     CALL   escreva
28     JMP    salto2
29

```

Figura 9.1 - Programa MENSAGEM4 na ferramenta emu8086.

Ao executar o programa, serão apresentadas as mensagens **Mensagem 3**, **Mensagem 2** e **Mensagem 1**, como mostra a Figura 9.2. Se forem omitidas as linhas com o comando **JMP**, a saída será **Mensagem 1**, **Mensagem2** e **Mensagem 3** (faça o teste).

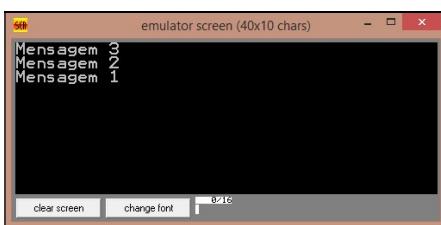


Figura 9.2 - Tela de saída do programa MENSAGEM4 na ferramenta emu8086.

O comando **JMP** desvia a execução do programa sem nenhum motivo aparente, apenas pela simples questão de ser esta a sua finalidade e de a instrução estar ali definida. Por esta razão o comando **JMP** é denominado *desvio incondicional*.

onal. Essa instrução é executada após a atualização do registrador IP em relação ao acesso à instrução seguinte e pode ser utilizada de duas maneiras distintas.

Observe que na linha **05** do programa está sendo usada a diretiva de compilação **org 100h** (poderia também ter sido usado **org 0x100**) que define um arquivo de tamanho pequeno, de até 64 KBytes, similar aos arquivos de programas criados com o programa **Enhanced DEBUG**.

Nas linhas de **08** até **10** ao lado da definição das variáveis **mensagem1**, **mensagem2** e **mensagem3**, além do *string* a ser apresentado, são definidos após a vírgula de cada mensagem os valores **0Dh**, **0Ah** e **24h**, que correspondem à ação de mudança de linha do cursor após a escrita da mensagem.

O valor **0Dh** corresponde ao código de uso da tecla **<Enter>**, o valor **0Ah** corresponde ao código de mudança de linha e o código **24h** corresponde ao uso do caractere \$, necessário para identificar o final de uma sequência de caracteres. Note que quando se faz a entrada de um dado alfabético no teclado de um computador e ação-se a tecla **<Enter>**, ocorre internamente a ação dos três códigos aqui apresentados.

Na linha **13** há o comando **JMP salto3** que envia o fluxo de execução do programa para a linha **25** (identificada com o rótulo **salto3:**) e a partir desse ponto executam-se o envio do conteúdo da **mensagem3** para o registrador **DX** na linha **26** e a linha **27** que faz a chamada do procedimento **escreva** por meio da instrução **CALL escreva** que se encontra definida a partir da linha **33**.

O procedimento **escreva** faz a definição do código de apresentação de caracteres **09h** para o registrador **AH** que será utilizado pela interrupção **21h**. Após a execução das linhas de código de **33** até **36**, ou seja, após a apresentação do conteúdo da variável **mensagem3**, a linha **36** por meio da instrução **RET** faz o retorno do procedimento para a primeira linha após sua chamada. Como a primeira chamada do procedimento ocorreu pela linha **27**, o retorno é feito na linha **28**.

A linha **28 (JMP salto2)** faz um desvio para a linha **20** (identificada com o rótulo **salto2:**) e a partir dessa posição ocorre na linha **21 (LEA DX, mensagem2)** o apontamento para o conteúdo da variável **mensagem2** para o registrador **DX**. Na linha **22** é feita nova chamada ao procedimento **escreva** que após sua execução retorna o fluxo de execução do programa para a linha **23**, que faz o desvio do programa para a linha **15**.

A partir da linha **15** ocorre na linha **16** o apontamento do conteúdo da variável **mensagem3** para o registrador **DX**. Em seguida na linha **17** é feita nova chamada do procedimento **escreva** que, após sua operacionalização, devolve o fluxo do programa à linha **18** que desvia o fluxo para a linha **30** em que o encerramento do programa é então processado.

## 9.2 - Salto condicional

O salto condicional é diferente de um salto incondicional representado pelo comando **JMP**, pois para ser executado, necessita de um comando condicional sobre a qual uma decisão é tomada. Normalmente uma condição é estabelecida com o uso de um dos comandos: **CMP (Compare)**, **AND (Logical and)**, **OR (Logical or)**, **NOT (Logical not)** e **XOR (Exclusive or)**, dos quais os comandos **CMP** e **AND** já foram apresentados.

Para auxiliar o processo de ação de desvios condicionais, são utilizadas os comandos de saltos **JA**, **JAE**, **JB**, **JBE**, **JC**, **JCXZ**, **JE**, **JG**, **JGE**, **JL**, **JLE**, **JNA**, **JNAE**, **JNB**, **JNBE**, **JNC**, **JNE**, **JNG**, **JNGE**, **JNL**, **JNS**, **JNO**, **JNP**, **JNLE**, **JNO**, **JNP**, **JNS**, **JNZ**, **JO**, **JP**, **JPE**, **JPO**, **JS**, **JZ**, dos quais os comandos **JG**, **JL**, **JLE**, **JNZ** já foram apresentadas e demonstradas. As instruções de desvios condicionais efetuam seus desvios numa escala entre -127 e 128 bytes.

A Tabela 9.1 exibe o conjunto completo de comandos de saltos a serem efetivadas pela linguagem de programação Assembly 8086/8088, após a utilização de uma instrução condicional.

Tabela 9.1 - Instruções de salto condicional

Comando	Significado	Descrição
JA	jump on above	salte se acima de
JAE	jump on above or equal	salte se acima ou igual a
JB	jump on below	salte se abaixo de
JBE	jump on below or equal	salte se abaixo ou igual a
JC	jump on carry	salte se flag carry igual a 1

Comando	Significado	Descrição
JCXZ	jump if cx register zero	salte se CX registra zero
JE	jump on equal	salte se igual a
JG	jump on greater	salte se maior que
JGE	jump on greater or equal	salte se maior ou igual a
JL	jump on less	salte se menor que
JLE	jump on less or equal	salte se menor ou igual a
JNA	jump on not above	salte se não acima de
JNAE	jump on not above or equal	salte se não acima ou igual a
JNB	jump on not below	salte se não abaixo de
JNBE	jump on not below or equal	salte se não abaixo ou igual a
JNC	jump on not carry	salte se flag carry igual a 0
JNE	jump on not equal	salte se não igual a
JNG	jump on not greater	salte se não maior que
JNGE	jump on not greater or equal	salte se não maior ou igual a
JNL	jump on not less	salte se não menor que
JNLE	jump on not less or equal	salte se não menor ou igual a
JNO	jump or not overlay	salte se não ocorreu <i>overlay</i>
JNP	jump on not parity	salte se não for par
JNS	jump on not sign	salte se positivo
JNZ	jump on not zero	salte se não for zero
JO	jump on overflow	salte se ocorreu <i>overflow</i>
JP	jump on parity	salte se for par
JPE	jump on parity equal	salte se for par
JPO	jump on parity odd	salte se for ímpar
JS	jump on sign	salte se for negativo
JZ	jump on zero	salte se for zero

É importante considerar que alguns comandos de saltos condicionais possuem aparentemente certa ambiguidade operacional com outros comandos condicionais, pois executam a mesma ação lógica de processamento, mas são grafadas de forma diferente. São pares de ação as instruções da Tabela 9.2.

Tabela 9.2 - Instruções de salto opostas

Comando	Comando Oposto
JA	JNBE
JAE	JNB
JB	JNAE
JBE	JNA
JE	JZ
JG	JNLE

Comando	Comando Oposto
JGE	JNL
JL	JNGE
JLE	JNG
JNP	JPO
JP	JPE

Esses comandos condicionais formam alguns pares de comandos. Um determinado comando pode ser utilizado perfeitamente no lugar do outro comando, dependendo da escolha ou necessidade do desenvolvedor ou do programa em desenvolvimento.

É importante considerar que para um comando de saldo condicional operar ela necessita do auxílio e definição anteriores de um dos comandos de comparação **CMP**, **AND**, **OR**, **NOT** e **XOR**.

Para a realização de desvios condicionais baseados na estrutura relacional, os comandos de comparação devem ser usados com os comandos de saltos condicionais da Tabela 9.3.

**Tabela 9.3 - Instruções de salto relacionais**

Comando	Descrição	Equivalência
JE	salte se igual a	=
JG	salte se maior que	>
JGE	salte se maior ou igual a	$\geq$
JL	salte se menor que	<
JLE	salte se menor ou igual a	$\leq$
JNE	salte se não igual a (diferente de)	$\neq$

Para a realização de desvios condicionais baseados na verificação dos registradores de estado, os comandos de comparação devem ser usados com os comandos de saltos condicionais indicados na Tabela 9.4.

**Tabela 9.4 - Instruções de salto baseados em registradores**

Comando	Descrição
JC	salta se registrador CF for igual a 1
JNC	salta se registrador CF for igual a 0
JNO	salta se registrador OF for igual a 0
JNS	salta se registrador SF for igual a 1
JNZ	salta se registrador ZF for igual a 0
JO	salta se registrador OF for igual a 1
JPE	salta se registrador PF for igual a 1 (paridade par)
JPO	salta se registrador PF for igual a 0 (paridade ímpar)
JS	salta se registrador SF for igual a 1
JZ	salta se registrador ZF for igual a 1

Todos os comandos de saltos condicionais são iniciados com o caractere "J" de *jump*. Qualquer um dos comandos de salto condicional deve estar associado a um rótulo de identificação para determinar o local de desvio no programa.

Os saltos condicionais, diferentemente dos saltos incondicionais, são feitos de forma relativa quanto à posição de sua definição. Não é possível realizar saltos condicionais muito longos. O tamanho máximo de salto está na faixa de -128 bytes a 127 bytes a partir da posição em que se encontra o comando de salto.

Além dos comandos condicionais, existem comandos de desvios condicionais de uso geral, indicados na Tabela 9.5.

**Tabela 9.5 - Instruções de salto gerais**

Comando	Descrição
JA	salte se acima de
JAE	salte se acima ou igual a
JB	salte se abaixo de
JBE	salte se abaixo ou igual a
JCXZ	salte se CX registra zero
JNA	salte se não acima de
JNAE	salte se não acima ou igual a
JNB	salte se não abaixo de
JNBE	salte se não abaixo ou igual a
JNG	salte se não maior que

Comando	Descrição
JNGE	salte se não maior ou igual a
JNL	salte se não menor que
JNLE	salte se não menor ou igual a
JNP	salte se não for par
JP	salte se for par

No universo de saltos, os comandos podem ser divididos em três grupos operacionais: o primeiro grupo são os que operam com apenas um registrador de estado, o segundo grupo são os que operam com valores sinalizados (valores negativos) e o terceiro são os que operam com valores não sinalizados (valores positivos), representadas na Tabela 9.6.

Tabela 9.6 - Instruções de salto separadas por grupos operacionais

Saltos com um Registrador de Estado		
Comando	Condição	Comando Oposto
JZ, JE	ZF = 1	JNZ, JNE
JC, JB, JNAE	CF = 1	JNC, JNB, JAE
JS	SF = 1	JNS
JO	OF = 1	JNO
JPE, JP	PF = 1	JPO
JNZ, JNE	ZF = 0	JZ, JE
JNC, JNB, JAE	CF = 0	JC, JB, JNAE
JNS	SF = 0	JS
JNO	OF = 0	JO
JPO, JNP	PF = 0	JPE, JP
Saltos com Valores Sinalizados (Negativos)		
Comando	Condição	Comando Oposto
JE, JZ	ZF = 1	JNE, JNZ
JNE, JNZ	ZF = 0	JE, JZ
JG, JNLE	ZF = 0 e SF = OF	JNG, JLE
JL, JNGE	SF <> OF	JNL, JGE
JGE, JNL	SF = OF	JNGE, JL
JLE, JNG	ZF = 1 ou SF <> OF	JNLE, JG
Saltos com Valores Não Sinalizados (Positivos)		
Comando	Condição	Comando Oposto
JE, JZ	ZF = 1	JNE, JNZ
JNE, JNZ	ZF = 0	JE, JZ
JA, JNBE	CF = 0 e ZF = 0	JNA, JBE
JB, JNAE, JC	CF = 1	JNB, JAE, JNC
JAE, JNB, JNC	CF = 0	JNAE, JB
JBE, JNA	CF = 1 ou ZF = 1	JNBE, JA

Os comandos de saltos **JA**, **JAE**, **JBE**, **JE**, **JE**, **JG**, **JGE**, **JL**, **JLE**, **JNE**, **JNE** e **JB**, bem como seus equivalentes, podem e devem ser utilizados após alguma ação aritmética ou lógica.

## 9.3 - Desvios condicionais

---

Com a finalidade de explorar o uso de alguns comandos de saltos condicionais, são exibidos em seguida exemplos simples de situações nas quais é necessário utilizar esses recursos (vale ressaltar que alguns recursos já haviam sido usados), como é o caso da tomada de decisão: simples, composta, sequencial, encadeada e por seleção.

É necessário primeiramente entender e conhecer o mecanismo de funcionamento das principais instruções de saltos condicionais, para depois melhorar a forma de estabelecer o processo entrada e saída de dados. Infelizmente por questão de espaço torna-se difícil exemplificar e utilizar todas as instruções de saltos condicionais. No entanto, muitas delas serão usadas de forma automática em outras situações a serem estabelecidas.

### 9.3.1 - Desvio condicional simples

A decisão simples ocorre quando uma ação necessita ser realizada caso uma determinada condição seja verdadeira. Se a condição for falsa, não deve ser executada a ação especificada como verdadeira, passando o controle operacional do programa para outro ponto, que normalmente também será executado após a ação considerada verdadeira.

Imagine a necessidade de verificar se o conteúdo do registrador **AL** e o conteúdo do registrador **BL** são de fato iguais. Se forem iguais, é necessário somar o valor **1** ao registrador geral **AL**. Após a execução da decisão sobre a condição ser verdadeira ou falsa, o conteúdo do registrador geral **BL** deve ser diminuído em **1**. O programa deve apresentar os valores dos registradores **AL** e **BL**.

Na linguagem Assembly 8086/8088 de programação de computadores o programa deve ser codificado como:

```
;*****  
;*      Programa: CONDIC1.ASM      *  
;*****  
  
.MODEL small  
.STACK 512d  
  
.DATA  
    a DB 6d  
    b DB 6d  
  
.CODE  
    MOV AX, @DATA  
    MOV DS, AX  
  
    MOV AL, a  
    MOV BL, b  
  
    CMP AL, BL  
    JE entao  
    JMP fimse  
  
entao:  
    INC AL  
    CALL apoio  
    MOV DL, AL  
    CALL escreva
```

```

fimse:
DEC BL
CALL apoio
MOV DL, BL
CALL escreva

```

**INT** 20h

```
escreva PROC NEAR
```

```

ADD DL, 30h
CMP DL, 39h
JLE valor
ADD DL, 07h
valor:
INT 21h
RET

```

```
escreva ENDP
```

```
apoio PROC NEAR
```

```

MOV AH, 02h
MOV CL, 04h
SHR DL, CL
RET

```

```
apoio 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 então escreva o programa anterior, gravando-o por meio dos comandos de menu **file/save** com o nome **CONDIC1**, de forma que fique semelhante à imagem da Figura 9.3.

The screenshot shows the emu8086 assembly editor interface. The menu bar includes file, edit, bookmarks, assembler, emulator, math, ascii codes, and help. The toolbar has buttons for new, open, examples, save, compile, emulate, calculator, converter, options, help, and about. The main window displays the assembly code:

```

01 ;***** Programa: CONDIC1.ASM *****
02 ;
03 ;
04
05 .MODEL small
06 .STACK 512d
07
08 .DATA
09 a DB 6d
10 b DB 6d
11
12 .CODE
13 MOV AX, @DATA
14 MOV DS, AX
15
16 MOV AL, a
17 MOV BL, b
18
19 CMP AL, BL
20 JE entao
21 JMP fimse
22
23 entao:
24 INC AL
25 CALL apoio
26 MOV DL, AL
27 CALL escreva
28
29 fimse:
30 DEC BL

```

The status bar at the bottom indicates line: 53 and col: 1.

Figura 9.3 - Programa CONDIC1 na ferramenta emu8086.

Observe o trecho de código entre as linhas 19 e 21, o qual representa algo como **SE (AL = BL) ENTÃO**. No lugar da instrução **JE entao** (linha 20) pode ser utilizada sem problema a instrução **JZ entao**.

Os registradores **AL** e **BL**, respectivamente, possuem o valor **6** (decimal) das variáveis **a** e **b** (linhas **16** e **17**), e sendo a condição verdadeira, ela é desviada pela instrução **JE** para o ponto de execução **entao** definido entre as linhas **23** e **27**.

No trecho de programa assinalado como **entao** ocorre o incremento de valor **01h** ao valor existente no registrador **AL** por meio da instrução **INC** (linha **24**). Na sequência é chamado o procedimento **apoio** (linha **25**) que prepara a memória para a apresentação do valor calculado do registrador **AL**. Em seguida é transferido o valor do registrador **AL** para o registrador **DL** (linha **26**) e na linha de código **27** é chamado o procedimento **escreva** que apresenta o valor armazenado em **AL**.

Após apresentar o valor do registrador **AL** (que estará com o valor **07h**), o programa executa o trecho identificado como **fimse** (linha **29**), no qual faz o decremento de **01h** sobre o registrador **BL** por meio da instrução **DEC** (linha **30**). Assim sendo, o registrador **BL** passa a possuir o valor **05h**.

Em seguida na linha **31** é efetuada a chamada ao procedimento **apoio** que após sua execução retorna seu processamento para a linha **32**. O conteúdo do registrador **BL** é movimentado para o registrador **DL**, depois na linha **33** ocorre a chamada ao procedimento **escreva**.

O procedimento denominado **apoio** definido entre as linhas **47** e **52** faz a preparação do ambiente para a apresentação de um caractere por vez do valor a ser mostrado na tela do monitor de vídeo. Já o procedimento **escreva** faz a apresentação de um caractere por vez.

Experimente trocar os valores das variáveis **a** e **b** e executar o programa para verificar a forma de funcionamento com valores diferentes.

### 9.3.2 - Desvio condicional composto

A decisão composta ocorre quando uma ação precisa ser realizada caso uma determinada condição seja verdadeira ou falsa. Se a condição é verdadeira, será executada uma determinada ação especificada para esta finalidade. Se a condição por falsa, será executada outra ação.

Imagine a necessidade de verificar se o conteúdo do registrador **AL** é maior que o conteúdo do registrador **BL**. Se for maior, é necessário somar o valor **1** ao registrador **AL** e apresentar seu valor; caso contrário, o conteúdo do registrador **BL** deve ser diminuído em **1** para então ser apresentado seu resultado.

Na linguagem Assembly 8086/8088 de programação de computadores o programa deve ser codificado como:

```
;*****  
;*  Programa: CONDIC2.ASM      *  
;*****  
  
.MODEL small  
.STACK 512d  
  
.DATA  
a DB 4d  
b DB 9d  
  
.CODE  
MOV AX, @DATA  
MOV DS, AX  
  
MOV AL, a  
MOV BL, b  
  
CMP AL, BL  
JG entao  
JLE senao
```

```

entao:
INC AL
CALL apoio
MOV DL, AL
CALL escreva
JMP fimse

senao:
DEC BL
CALL apoio
MOV DL, BL
CALL escreva
JMP fimse

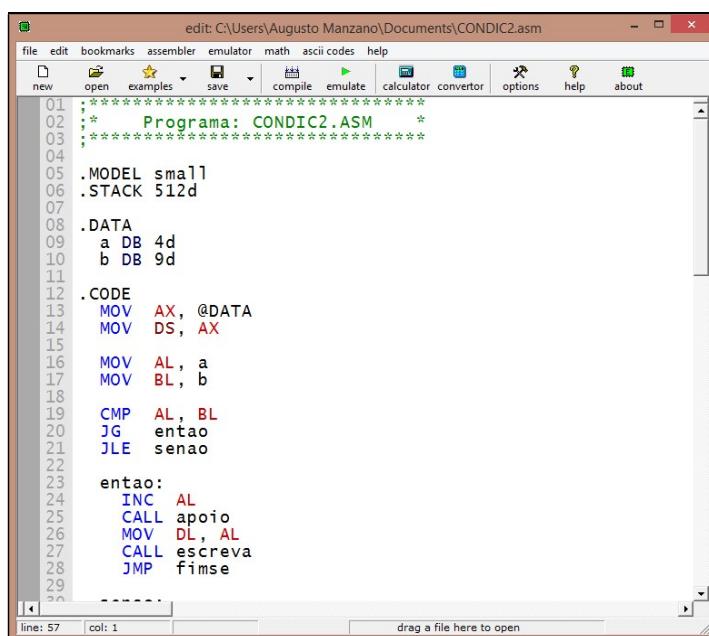
fimse:
MOV AH, 4Ch
INT 21h

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

apoio PROC NEAR
MOV AH, 02h
MOV CL, 04h
SHR DL, CL
RET
apoio ENDP

```

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 **CONDIC2**, de forma que fique semelhante à imagem da Figura 9.4.



The screenshot shows the emu8086 assembly editor window. The title bar reads "edit: C:\Users\Augusto Manzano\Documents\CONDIC2.asm". The menu bar includes file, edit, bookmarks, assembler, emulator, math, ascii codes, help, and a toolbar with various icons. The main code area contains the following assembly code:

```

01 ;***** Programa: CONDIC2.ASM *****
02
03
04 .MODEL small
05 .STACK 512d
06
07
08 .DATA
09 a DB 4d
10 b DB 9d
11
12 .CODE
13 MOV AX, @DATA
14 MOV DS, AX
15
16 MOV AL, a
17 MOV BL, b
18
19 CMP AL, BL
20 JG entao
21 JLE senao
22
23 entao:
24 INC AL
25 CALL apoio
26 MOV DL, AL
27 CALL escreva
28 JMP fimse
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57

```

Figura 9.4 - Programa CONDIC2 na ferramenta emu8086.

Atente para os detalhes de código entre as linhas **19** e **21**. Note que estão sendo utilizadas após a instrução **CMP** duas instruções de desvio condicional. Uma é a instrução **JG** (condição maior que) na linha **20** e a outra **JLE** (condição menor que) na linha **21**.

Em linhas gerais o programa **CONDIC2** é muito semelhante ao programa **CONDIC1**, tendo como diferença principal as instruções definidas entre as linhas **19** e **21**.

As duas instruções de desvio condicional permitem a representação da ação da tomada de decisões composta. O funcionamento da instrução **JG** e da instrução **JLE** é proporcionalmente inverso.

Experimente trocar os valores das variáveis **a** e **b** e executar o programa para verificar a forma de funcionamento com valores diferentes.

## 9.4 - Operações lógicas

O uso de operações lógicas em um programa são elementos importantes quando é necessário trabalhar com mais de uma condição (quando da utilização dos comandos operadores **AND**, **OR** e **XOR**) ou fazer a negação de uma condição (comando operador **NOT**) para a tomada de outra condição.

É importante ressaltar que as ações lógicas operam o conteúdo armazenado na memória no nível binário. Se estiver em análise um dado do tipo *byte* ou *word*, a instrução lógica em uso fará uma varredura do *bit* mais baixo para o *bit* mais alto, verificando o dado da direita para a esquerda. Se todos os *bits* corresponderem à condição estabelecida, ela será considerada *verdadeira*; caso contrário, será considerada falsa.

De forma operacional as instruções lógicas devem ser usadas seguindo esta sintaxe:

```
AND [destino], [origem]
OR  [destino], [origem]
XOR [destino], [origem]
NOT [destino]
```

Os parâmetros **destino** e **origem** são obrigatórios e o resultado lógico da avaliação da instrução lógica em uso será sempre implicado no parâmetro **destino**.

O comando **AND** estabelece a conjunção de dois valores em nível binário: **origem** e **destino**. O resultado de saída de **destino** será verdadeiro (valor **1**) se ambos os *bits* da posição em análise forem **1**; caso contrário, será considerado falso (valor **0**). Veja a tabela-verdade para o comando lógico **AND** indicado na Tabela 9.7.

Tabela 9.7 – Operador AND

Entrada ( <i>bits</i> )		Saída ( <i>bit</i> )
Origem	Destino	Destino
0	0	0
0	1	0
1	0	0
1	1	1

O comando **OR** estabelece a disjunção de dois valores em nível binário: **origem** e **destino**. O resultado de saída de **destino** será verdadeiro (valor **1**) se pelo menos um dos *bits* da posição em análise for **1**; caso contrário, será considerado falso (valor **0**). Veja a tabela-verdade para o comando lógico **OR** indicado na Tabela 9.8.

Tabela 9.8 – Operador OR

Entrada ( <i>bits</i> )		Saída ( <i>bit</i> )
Origem	Destino	Destino
0	0	0
0	1	1
1	0	1
1	1	1

O comando **XOR** estabelece a disjunção exclusiva de dois valores em nível binário: **origem** e **destino**. O resultado de saída de **destino** será verdadeiro (valor 1) quando os *bits* da posição em análise forem diferentes. Caso os valores binários sejam iguais, o resultado de **destino** será falso (valor 0). Veja ao lado a tabela-verdade para o comando lógico **XOR**. Veja a tabela-verdade para o comando lógico **XOR** indicado na Tabela 9.9.

Tabela 9.9 – Operador XOR

Entrada (bits)		Saída (bit)
Origem	Destino	Destino
1	0	0
0	1	1
1	0	1
1	1	0

O comando **NOT** estabelece a negação do valor em nível binário. O resultado de saída de **destino** será verdadeiro (valor 1) quando o valor de destino da entrada for falso (valor 0). Será considerado falso (valor 0) quando o valor de destino da entrada for verdadeiro (valor 1). Veja a tabela-verdade para o comando lógico **NOT** indicado na Tabela 9.10.

Tabela 9.10 – Operador NOT

Entrada (bits)		Saída (bit)
Destino	Destino	Destino
0	1	1
1	0	0

A título de ilustração geral considere a Tabela 9.11 com a avaliação lógica pelos comandos **AND**, **OR**, **XOR** e **NOT** de alguns valores binários.

Tabela 9.11 – Comaparação entre operadores lógicos

AL	BL	AND AL, BL	OR AI, BL	XOR AL, BL	NOT AL
		AL	AL	AL	AL
1111 1111	1111 1111	1111 1111	1111 1111	0000 0000	0000 0000
1111 0000	0000 1111	0000 0000	1111 1111	1111 1111	0000 1111
1100 0011	1010 1010	0000 0010	1110 1011	0110 1001	0011 1100
0110 0011	0000 0011	0000 0011	0110 0011	0110 0000	1001 1100

Para fazer uma verificação rápida dos resultados lógicos obtidos em relação aos valores binários de 1 byte, representados pelos registradores **AL** e **BL** (tabela anterior), utilize a calculadora do Windows em modo de cálculo binário e use as operações **AND**, **OR**, **XOR** e **NOT**.

Um detalhe importante a ser considerado é que os comando lógicos **AND**, **OR**, **XOR** e **NOT** devem ser utilizados de forma semelhante ao comando **CMP**. Normalmente após a definição de um desses comandos há necessidade de utilizar um comando de desvio condicional.

Para exemplificar o uso do comando **AND**, considere um programa que faça a leitura de um valor numérico positivo entre 0 e 9 de apenas um dígito, e apresente uma mensagem informando se o número é par ou ímpar. Qualquer caractere que não seja um valor entre 0 e 9 deve ser recusado. Observe o código seguinte:

```
;*****
;*      Programa: CONDIC3.ASM      *
;*****
```

**org 100h**

```

.DATA
msg1 DB 'Entre um valor numerico positivo (de 0 ate 9): ', 24h
msg2 DB 0Dh, 0Ah, 'Valor impar', 24h
msg3 DB 0Dh, 0Ah, 'Valor par', 24h
msg4 DB 0Dh, 0Ah, 'Caractere invalido', 24h

.CODE
LEA DX, msg1
CALL escreva

MOV AH, 01h
INT 21h

CMP AL, 30h
JL erro

CMP AL, 39h
JG erro

SUB AL, 30h
AND AL, 01h
JPE par
JPO impar

par:
LEA DX, msg3
CALL escreva
JMP saida

impar:
LEA DX, msg2
CALL escreva
JMP saida

erro:
LEA DX, msg4
CALL escreva
JMP saida

saida:
INT 20h

escreva PROC NEAR
MOV AH, 09h
INT 21h
RET
escreva ENDP

```

Execute no programa **emu8086** o comando de menu **file/new/com template**, aione 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 **CONDIC3**, de forma que fique semelhante à imagem da Figura 9.5.

O programa possui após a definição da diretiva **.DATA** a definição das variáveis com as mensagens que serão apresentadas. Em especial, atente para as variáveis **msg2**, **msg3** e **msg4** (linhas 09 a 11) que, além de utilizarem o código **24h** (que representa o caractere de fim de string \$), utilizam o código **0Dh** (que executa o efeito de retorno de carro, tecla **<Enter>**) e o **0Ah** (que executa o efeito de mudança de linha, *line feed*).

Quando no programa for feita a chamada dos trechos em que se encontram as variáveis **msg2**, **msg3** e **msg4**, as suas mensagens serão sempre escritas após a apresentação da mensagem identificada na variável **msg1**.

```

01 ;***** Programa: CONDIC3.ASM *****
02 ;*
03 ;*****
04
05 org 100h
06
07 .DATA
08 msg1 DB 'Entre um valor numerico positivo (de 0 ate 9):'
09 msg2 DB 0Dh, 0Ah, 'Valor impar', 24h
10 msg3 DB 0Dh, 0Ah, 'Valor par', 24h
11 msg4 DB 0Dh, 0Ah, 'Caractere invalido', 24h
12
13 .CODE
14 LEA DX, msg1
15 CALL escreva
16
17 MOV AH, 01h
18 INT 21h
19
20 CMP AL, 30h
21 JL erro
22
23 CMP AL, 39h
24 JG erro
25
26 SUB AL, 30h
27 AND AL, 01h
28 JPE par
29 JPO impar
30

```

line: 54 col: 1 drag a file here to open

Figura 9.5 - Programa CONDIC3 na ferramenta emu8086.

O programa executa algumas verificações após a entrada do valor pelo teclado. É definido na linha 17 o código **01h** para a entrada de um caractere pelo teclado que será armazenado no registrador **AH**, sendo essa ação controlada pela instrução **INT 21h** da linha 18.

Entre as linhas 20 e 21 está definido o desvio de detecção de erro, caso o valor fornecido seja menor que o valor **30h** (código ASCII para a definição do valor **0d**). Note que o programa por meio da instrução **JL erro** efetua o desvio para a linha 41 que apresenta a mensagem armazenada na variável **msg4**.

Entre as linhas 23 e 24 está definido outro desvio de detecção de erro, caso o valor fornecido seja maior ou igual ao valor **39h** (código ASCII para o valor **9h**). Note que o programa por meio da instrução **JG erro** efetua o desvio para a linha 41 que apresenta a mensagem armazenada na variável **msg4**.

O trecho da linha 20 até a linha 24 verifica a validade da entrada. Qualquer caractere que resulte um valor abaixo de **0** ou acima de **9** será considerado inválido.

O trecho mais importante do programa concentra-se nas linhas 26 até 29, pois nesse ponto é realizada a verificação lógica com o comando **AND**. A linha 26 efetua apenas a subtração do valor **30h** do código ASCII para deixar apenas no registrador menos significativo **AL** o valor numérico propriamente dito.

Na linha 27 está em uso a instrução **AND AL, 01h**. O valor **01h** equivale ao valor **00000001b**. Ao ser feita a comparação com o valor armazenado **AL**, o comando **AND** altera ou não o valor do registrador de estado **PF**. Neste caso, o registrador de estado **PF** estará com o valor **1** caso o comando **AND** seja verdadeiro. Se o comando **AND** estiver com valor falso, o registrador de estado **PF** estará com valor **0**.

A Tabela 9.12 exemplifica o que ocorre internamente no programa quando utilizados os valores **7, 8, 2 e 5**.

Tabela 9.12 – Demonstração do estado de execução do programa exemplo

AND		Resultado	
AL	01h	AL	PF
0000 0111	0000 0001	0000 0001	0
0000 1000	0000 0001	0000 0000	1
0000 0010	0000 0001	0000 0000	1
0000 0101	0000 0001	0000 0001	0

Observe também o uso dos comandos de desvio condicional **JPE** (linha 28) e **JPO** (linha 29), as quais desviam a execução do programa para o trecho pertinente à validade da condição que utilizam. O comando **JPE** é executado quando o comando **AND** resultar um valor lógico verdadeiro, enquanto que o comando **JPO** é executado quando o comando **AND** resultar um valor lógico falso.

Este exemplo merece um detalhamento quanto à sua execução passo a passo. Acione a tecla de função <F5>, em seguida a tecla de função <F8> e solicite a visualização da janela **flags** (acione o comando **view/flags** na janela **CONDIC3.com**). Ajuste a distribuição das janelas na tela de seu monitor da forma mais conveniente possível. A Figura 9.6 mostra esse momento de ação (primeira etapa da tecla de função <F8>) com as janelas solicitadas exibidas de forma distribuída manualmente. Acione a tecla <F8> mais onze vezes.

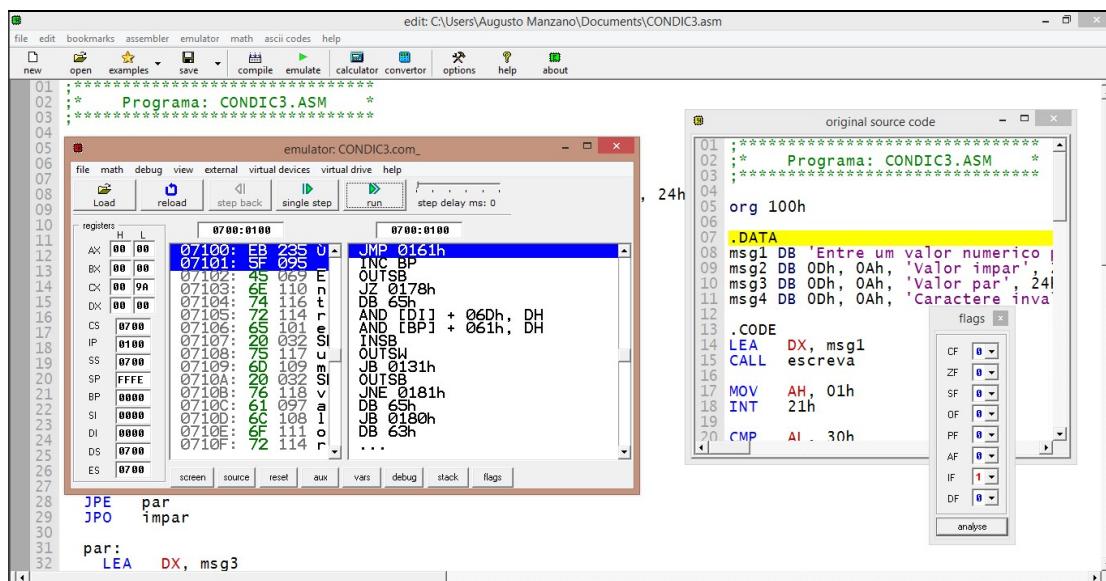


Figura 9.6 - Programa CONDIC3 em execução - início.

Para um teste de execução do programa informe o valor **8** quando a mensagem solicitando um valor numérico positivo for apresentada, acione em seguida a tecla <F8> mais seis vezes e observe a mudança da informação do registrador menos significativo **AL** com o valor **08**. Execute a partir da tela **emulator: CONDIC3.com\_** o comando de menu **view/extended value viewer**. Será então mostrada a tela do recurso **extended value viewer**, como mostra a Figura 9.8.

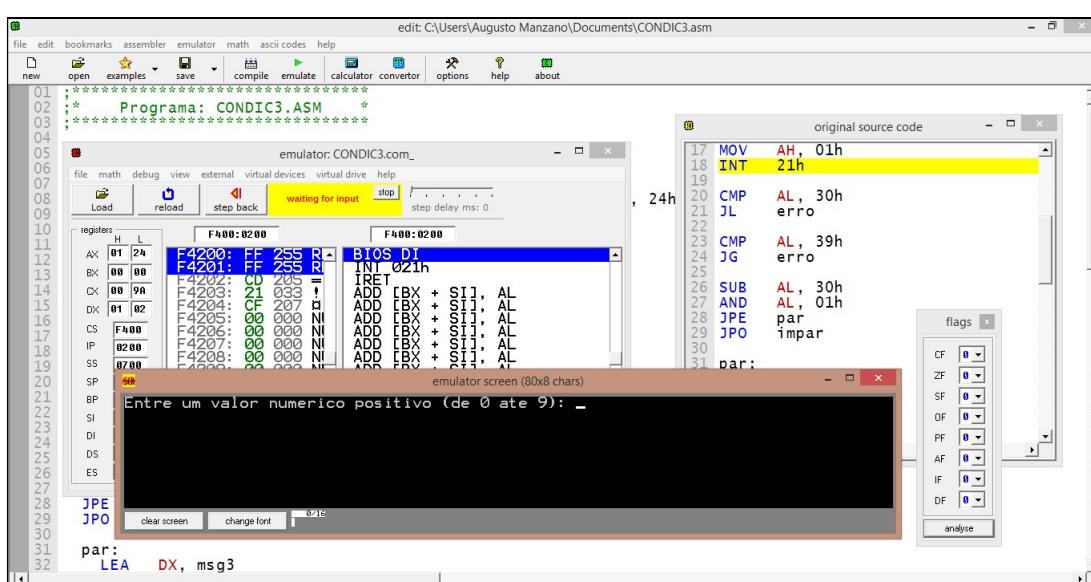


Figura 9.7 - Programa CONDIC3 em execução - entrada.

Observe as informações apresentadas na janela **extended value viewer**. O registrador menos significativo **AL** mostra o valor **08** (valor anteriormente informado). Na janela **original source code** a apresentação da barra de seleção amarela sobre a instrução **AND AL, 01h** mostra o comando que será executado no próximo passo. A Figura 9.9 apresenta a ocorrência.

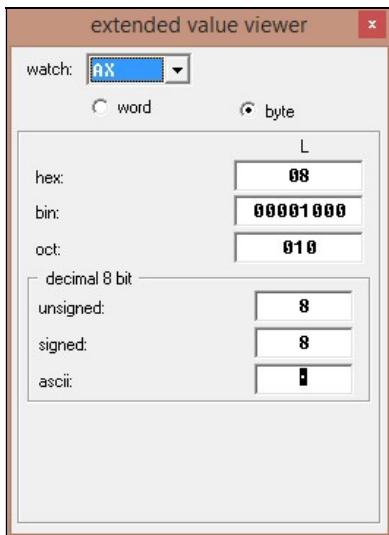


Figura 9.8 - Janela Extended value viewer.

```

17 MOV AH, 01h
18 INT 21h
19
20 CMP AL, 30h
21 JL erro
22
23 CMP AL, 39h
24 JG erro
25
26 SUB AL, 30h
27 AND AL, 01h
28 JPE par
29 JPO impar
30
31 par:
32 LEA DX, msg3
33 CALL escreva
34 JMP saida
35
36 impar:

```

The instruction at address 27, **AND AL, 01h**, is highlighted with a yellow selection bar.

Figura 9.9 - Janela Original source code.

Acione a tecla de função <F8> e observe na janela **flags** a mudança do valor do registrador **PF** para **1** indicando a ocorrência de paridade par para o valor fornecido. Assim que a tecla de função <F8> é acionada ocorre a operação lógica do comando **AND** sobre o valor armazenado no registrador menos significativo **AL**, a qual altera o valor do registrador menos significativo **AL** para **00000000b**, como pode ser verificado na Figura 9.10. Como o valor do registrador **AL** passa a ser **00000000b** com o registrador **PF** sinalizado como **1** a instrução **AND** resulta em valor verdadeiro.

Na sequência do programa, acione a tecla de função <F8> uma vez e observe que a janela **Original source code** aponta para a instrução **JPE par**. Acione a tecla de função <F8> seis vezes e será apresentada a mensagem **Valor par**. Se houvesse ocorrido a entrada de um valor ímpar, a instrução **JPE par** seria saltada, e a barra de seleção amarela seria posicionada sobre a instrução **JPO impar** para que fosse então apresentada a mensagem **Valor ímpar**. Para concluir a execução do programa, acione a tecla <F8> mais cinco vezes. A Figura 9.11 apresenta a janela **emulator screen** com a mensagem de saída do programa.

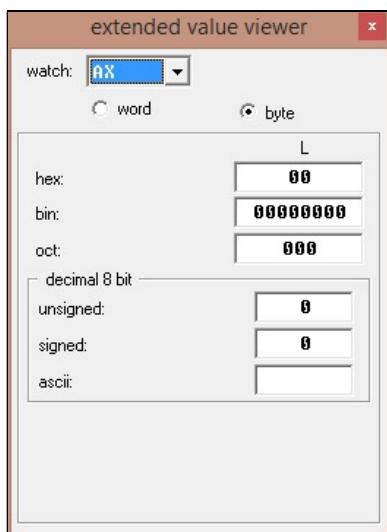


Figura 9.10 - Janela Extended value viewer com valor de AL alterado.

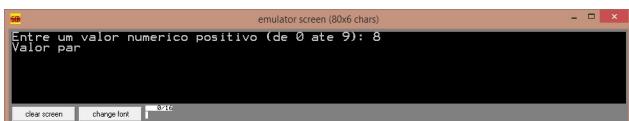


Figura 9.11 - Janela Emulator Screen.

Na caixa de diálogo **message** acione o botão **OK** e na janela **emulator: CONDIC3.com\_** acione o comando de menu **file/close emulator** para retornar à tela do editor do programa.

#### Observação

Além da instrução **AND** utilizada anteriormente há a instrução **TEST** que pode ser usada em seu lugar. A diferença entre as duas está no fato de a instrução **TEST** não efetuar a alteração do valor no registrador de destino, mas mantém a alteração do registrador de estado **CF**.

Os comandos de operações lógicas **AND**, **OR**, **XOR** e **NOT** não devem ser confundidos com os operadores lógicos em linguagens de alto nível. Os comando lógicos no *Assembly* têm a capacidade de manipular os dados em nível de *bit*.

O programa seguinte tem a finalidade de solicitar a entrada de dois valores numéricos decimais positivos de um dígito, somar os valores e apresentar o resultado da operação como sendo um valor decimal. O programa exemplifica o uso do comando **OR**. Acompanhe o código seguinte:

```
;*****  
;*      Programa: CONDIC4.ASM      *  
;*****  
  
.org 100h  
  
.DATA  
msg1 DB 'Entre valor decimal 1 (de 0 ate 9): ', 024h  
msg2 DB 0Dh, 0Ah, 'Entre valor decimal 2 (de 0 ate 9): ', 24h  
  
msg3 DB 0Dh, 0Ah, 'Soma = ', 24h  
msg4 DB 0Dh, 0Ah, 'Caractere invalido', 24h  
  
.CODE  
LEA DX, msg1  
CALL escreva  
CALL leia  
MOV BH, AL  
LEA DX, msg2  
CALL escreva  
CALL leia  
MOV BL, AL  
  
LEA DX, msg3  
CALL escreva  
XCHG AX, BX  
  
ADD AL, AH  
SUB AH, AH  
AAA  
MOV DX, AX  
MOV AH, 0Eh  
CMP DH, 0h  
JE nao_zero  
OR DH, 30h  
MOV AL, DH  
INT 10h  
nao_zero:  
OR DL, 30h  
MOV AL, DL  
INT 10h  
INT 20h
```

```

escreva PROC NEAR
    MOV AH, 09h
    INT 21h
    RET
escreva ENDP

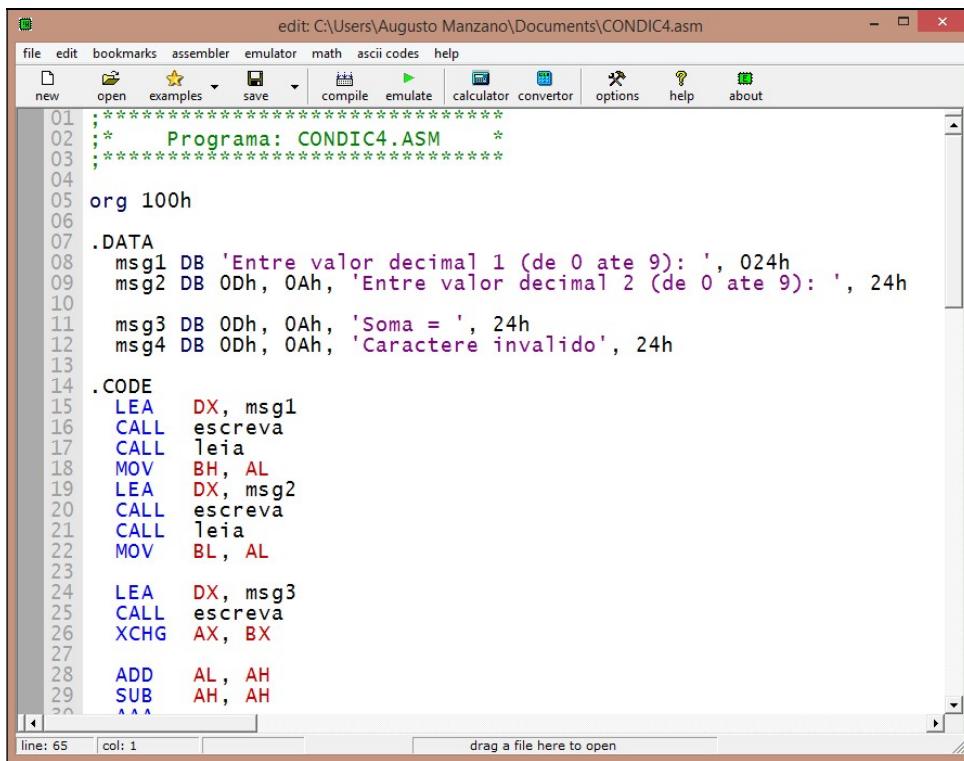
```

```

leia PROC NEAR
    MOV AH, 01h
    INT 21h
    CMP AL, 30h
    JL erro
    CMP AL, 3Ah
    JGE erro
    SUB AL, 30h
    RET
erro:
    LEA DX, msg4
    CALL escreva
    INT 20h
    RET
leia ENDP

```

Execute no programa **emu8086** o comando de menu **file/new/com template**, aione 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 **CONDIC4**, de forma que fique semelhante à imagem da Figura 9.12.



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

```

edit: C:\Users\Augusto Manzano\Documents\CONDIC4.asm
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator converter options help about
01 ; **** Programa: CONDIC4.ASM ****
02
03
04
05 org 100h
06
07 .DATA
08     msg1 DB 'Entre valor decimal 1 (de 0 ate 9): ', 024h
09     msg2 DB 0Dh, 0Ah, 'Entre valor decimal 2 (de 0 ate 9): ', 24h
10
11     msg3 DB 0Dh, 0Ah, 'Soma = ', 24h
12     msg4 DB 0Dh, 0Ah, 'Caractere invalido', 24h
13
14 .CODE
15     LEA DX, msg1
16     CALL escreva
17     CALL leia
18     MOV BH, AL
19     LEA DX, msg2
20     CALL escreva
21     CALL leia
22     MOV BL, AL
23
24     LEA DX, msg3
25     CALL escreva
26     XCHG AX, BX
27
28     ADD AL, AH
29     SUB AH, AH
30

```

The window has a menu bar with file, edit, bookmarks, assembler, emulator, math, ascii codes, help, and a toolbar with new, open, examples, save, compile, emulate, calculator, converter, options, help, and about buttons. The status bar at the bottom shows line: 65 col: 1 drag a file here to open.

Figura 9.12 - Programa CONDIC4 na ferramenta emu8086.

Apesar do programa **CONDIC4** fazer uso de recursos já apresentados e conhecidos, acrescenta-se os comandos **XCHG**, **AAA** e a instrução **INT 10**.

Primeiramente atente para a definição dos procedimentos **escreva** (idêntico ao programa **CONDIC3**) existente entre as linhas **44** e **48** e o procedimento **leia** (com alterações em relação ao programa **CONDIC3**) existente entre as linhas **50** e **64**. O procedimento **leia** é responsável pela recepção dos valores fornecidos no teclado.

O trecho de programa entre as linhas **14** e **17** apresenta a primeira mensagem solicitando o primeiro valor (linhas **14** e **15**), faz a entrada do valor (linha **16**) e com a linha **17** faz a movimentação do conteúdo do registrador **AL** (primeiro valor informado) para o registrador **BH** no sentido de esse valor ficar temporariamente armazenado para o cálculo e liberar o registrador **AL** para a nova entrada.

O trecho de programa entre as linhas **19** e **22** apresenta a segunda mensagem solicitando o segundo valor (linha **21**) e com a linha **22** faz a movimentação do conteúdo do registrador **AL** (segundo valor informado), para o registrador **BL** no sentido de esse valor ficar temporariamente armazenado para o cálculo.

A partir desse ponto o registrador **BX** conterá armazenados os dois valores fornecidos. O primeiro valor está armazenado na parte mais significativa (**BH**) e o segundo valor armazenado na parte menos significativa (**BL**).

Em seguida o trecho de linhas entre **24** e **26** mostra a terceira mensagem para a apresentação do resultado do cálculo processado. Na linha **26** há a definição de uso do comando **XCHG** (*exchange*) que faz a troca dos conteúdos entre os registradores **AX** e **BX**. Essa troca é necessária uma vez que a operação de adição entre os valores será feita com as partes mais (**AH**) e menos (**AL**) significativas do registrador **AH**. Desta forma, os valores reservados no registrador **BX** são posicionados no registrador **AX**.

O comando **XCHG** não afeta nenhum dos registradores de estado (*flags*), exigindo que ambos os operandos sejam de mesmo tamanho e não estejam relacionados à memória. Essa instrução não opera com o registrador IP, bem como com os registradores de segmento.

Na sequência o trecho de programa da linha **28** até a linha **42** é o mais importante, pois é responsável pela apresentação do valor decimal da resposta da operação de adição dos valores fornecidos.

A linha **28** efetua a soma dos dois valores e armazena o resultado dessa operação na parte menos significativa do registrador **AX**, ou seja, no registrador **AL**. Depois com a linha **29** o programa limpa o registrador **AH**.

Na linha **30** encontra-se o comando **AAA** (*Ascii Adjust for Addition*) que tem por finalidade alterar o valor do registrador **AL**, tornando-o um valor decimal válido. O comando **AAA** trabalha com o ajuste de valor baseando-se no uso de valores em BCD.

A instrução **INT 10** tem por finalidade efetuar uma chamada a interrupção da BIOS que trata o modo de acesso ao monitor de vídeo. Este comando é usado após a definição de uma função de operação junto ao registrador **AL**. No programa a instrução **MOV AL, DH** faz com que o valor definido no registrador **DH** seja passado ao registrador **AL** no sentido de apresentar o resultado da soma quando este resultado for maior ou igual a 10 (limitado a 15).

A linha **31** faz a transferência do valor armazenado no registrador geral **AX** para o registrador geral **DX**. O registrador **AX** fica livre para ser usado nas operações de apresentação em tela dos caracteres armazenados na memória.

O programa na linha **32** usa o valor **0Eh** que é um código de função utilizado para a apresentação de dados na tela do monitor de vídeo por meio da interrupção **010h** (**INT 010h** existente nas linhas **37** e **41**) que é responsável pelo acesso aos recursos da placa de vídeo conectada no computador.

Esse recurso substitui o uso da interrupção **21h**. O comportamento da interrupção **10h** é na maioria das vezes melhor que o comportamento da interrupção **21h**, pois é mais rápido.

Nas linhas **33** e **34** existe uma comparação verificando se o valor do registrador **DL** é zero, e se for, o programa é desviado para a linha **38**. Esse recurso evita que numa operação de adição de unidades seja apresentada à frente da unidade o valor **0**.

As linhas **35** e **39** utilizam a instrução **OR** para subtrair, respectivamente, dos valores existentes nos registradores **DH** e **DL** o valor **30h**. Desta forma converte-se o valor existente no seu código ASCII correspondente e para fazer a apresentação dos caracteres que formam o valor decimal, são executadas as ações das linhas **36:37** e **40:41**.

## 9.5 - Repetições

Outra estrutura de programação muito utilizada e importante são os laços que proporcionam a repetição de certos trechos de um programa (também conhecidos como *ciclos*, *loopings* ou *malhas de repetição*) cuja característica operacional é a capacidade de executar um trecho de programa por determinado número de vezes. Normalmente um laço é estabelecido em Assembly com os comandos **LOOP**, **LOOPE**, **LOOPNE**, **LOOPNZ** e **LOOPZ**. O comando **LOOP** já fora usada anteriormente.

A Tabela 9.13 exibe as instruções de laços de repetição existentes na linguagem de programação Assembly 8086/8088.

Tabela 9.13 – Instruções para execução de laços

Instrução	Significado	Descrição
LOOP	Loop	laço iterativo
LOOPE	loop while equal	enquanto laço igual a
LOOPNE	loop while not equal	enquanto laço não for igual a
LOOPNZ	loop while not zero	enquanto laço não for zero
LOOPZ	loop while zero	enquanto laço for zero

O laço baseado no comando **LOOP** é do tipo *iterativo*, ou seja, executa a ação do laço de repetição um determinado número de vezes, semelhante ao laço de repetição **FOR** encontrado em linguagens de alto nível.

Os laços **LOOPE**, **LOOPZ**, **LOOPNE** e **LOOPNZ** são do tipo *interativo*, ou seja, são laços condicionais, pois para operarem dependem do valor sinalizado no registrador de estado **ZF**. Essa forma é semelhante ao laço de repetição **WHILE** encontrado em linguagens de alto nível.

Os comandos **LOOPE** e **LOOPZ** são executados enquanto o registrador de estado **ZF** for igual a **1** (um) e o registrador geral **CX** for diferente de **0** (zero). Já os comandos **LOOPNE** e **LOOPNZ** são executados enquanto o registrador de estado **ZF** for igual a **0** (zero) e o registrador geral **CX** for diferente de **0** (zero).

Os comandos **LOOPE** / **LOOPZ** e **LOOPNE** / **LOOPNZ** são escritos de forma diferente, mas possuem a mesma estrutura operacional. Não é necessário apresentar em separado um exemplo de cada comando de laço. Os próximos programas contemplam as instruções **LOOP**, **LOOPE** e **LOOPNE**.

#### Observação

É bom relembrar que as instruções de laço (sejam elas quais forem) usam o valor que estiver armazenado no registrador geral **CX** para a operação do contador de passos. O valor é sempre decrementado do registrador geral **CX**.

Para exemplificar operações com laços , considere um programa que apresenta cinco vezes na tela do monitor de vídeo a mensagem **Alô Mundo!**, conforme indicado a seguir:

```
;*****  
;*      Programa: LAC01.ASM      *  
;*****  
  
.org 100h  
  
.DATA  
msg DB 'Alo Mundo!', 13d, 120, 24h  
  
.CODE  
LEA DX, msg  
MOV CX, 5d  
MOV AH, 09h  
laco:  
    INT 21h  
    LOOP laco  
INT 20h
```

Execute no programa **emu8086** o comando de menu **file/new/com template**, aione 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 **LAC01**, de forma que fique semelhante à imagem da Figura 9.13.

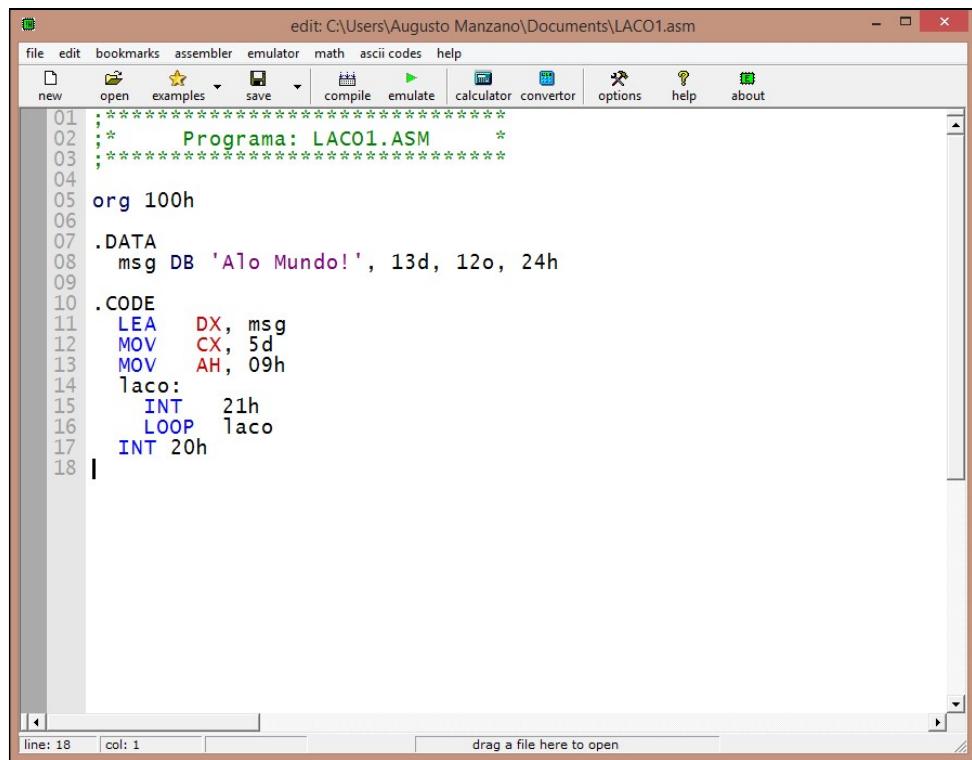


Figura 9.13 - Programa LACO1 na ferramenta emu8086

Observe na linha 08 o uso dos valores 13d (equivalente a 0Dh), 12o (equivalente a 0Ah) e 24h após a definição da mensagem Alô Mundo!, que são responsáveis pela definição dos códigos de controle para apresentação de um *string*. A definição na linha 12 da instrução **MOV CX, 5d** estabelece para o registrador geral **CX** o valor decimal 5 (valor que será usado para a efetivação da contagem dos laços de repetição), que será automaticamente decrementado em 1 toda vez que o comando **LOOP** (linha 16) for executado.

As demais linhas dos programas possuem recursos já conhecidos e que foram explanados em exemplos anteriores.

Considere em seguida um segundo exemplo de programa que leia um valor numérico positivo de um dígito entre os valores 0 e 8 e apresente como resultado o valor da fatorial do valor fornecido. Valores iguais ou superiores a 9 não são aceitos por gerarem um resultado inteiro acima da capacidade numérica de trabalho dos registradores gerais (considerando o uso de um microprocessador padrão 16 bits, sendo a solução para esta questão algo que foge do escopo deste trabalho). Observe detalhadamente o código a seguir:

```

;*****
;*      Programa: LACO2.ASM      *
;*****

org 100h

.DATA
msg1 DB 'Entre valor decimal positivo (de 0 ate 8): ', 24h
msg2 DB 0Dh, 0Ah, 'Fatorial de ', 24h
msg3 DB ' equivale a ', 24h
msg4 DB 0Dh, 0Ah, 'Valor invalido', 24h

.CODE
LEA    DX, msg1
CALL  mensagem
CALL  entrada
PUSH AX

```

```

LEA    DX, msg2
CALL   mensagem
POP    AX
MOV    DL, AL
MOV    AH, 0Eh
INT    10h
SUB    AL, 30h
MOV    CL, AL
LEA    DX, msg3
CALL   mensagem
CALL   fatorial
CALL   valor

fim:
INT    20h

mensagem PROC NEAR
MOV    AH, 09h
INT    021h
RET
mensagem ENDP

entrada PROC NEAR
MOV    AH, 01h
INT    021h
CMP    AL, 030h
JL    erro
CMP    AL, 039h
JGE   erro
JMP   fim_validacao
erro:
LEA    DX, msg4
CALL   mensagem
JMP   fim
fim_validacao:
RET
entrada ENDP

fatorial PROC NEAR
MOV    AX, 01h
CMP    CX, 0h
JE    fim_laco
repita1:
MUL    CX
LOOPNE repita1
fim_laco:
RET
fatorial ENDP

valor PROC NEAR
MOV    BX, 0Ah
SUB    CX, CX
repita2:
SUB    DX, DX
DIV    BX
PUSH   DX
INC    CX
CMP    AX, 0h
JNZ   repita2
saida:
POP    AX

```

```

ADD    AL, 30h
MOV    DL, AL
MOV    AH, 0Eh
INT    10h
DEC    CX
JNBE   saída
POP    DX
RET
valor ENDP

```

```

edit: C:\Users\Augusto Manzano\Documents\LACO2.asm
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator converter options help about
01 ;***** Programa: LACO2.ASM *****
02 ;*****
03 ;*****
04
05 org 100h
06
07 .DATA
08 msg1 DB 'Entre valor decimal positivo (de 0 ate 8): ', 24h
09 msg2 DB 0Dh, 0Ah, 'Fatorial de ', 24h
10 msg3 DB ' equivale a ', 24h
11 msg4 DB 0Dh, 0Ah, 'Valor invalido', 24h
12
13 .CODE
14 LEA    DX, msg1
15 CALL   mensagem
16 CALL   entrada
17 PUSH   AX
18
19 LEA    DX, msg2
20 CALL   mensagem
21 POP    AX
22 MOV    DL, AL
23 MOV    AH, 0Eh
24 INT    10h
25 SUB    AL, 30h
26 MOV    CL, AL
27 LEA    DX, msg3
28 CALL   mensagem
29 CALL   fatorial
30

```

line: 89 | col: 1 | drag a file here to open

Figura 9.14 - Programa LACO2 na ferramenta emu8086 - principal.

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 **LACO2**, de forma que fique semelhante às imagens das Figuras 9.14 (trecho do programa principal), 9.15 e 9.16 (trecho das sub-rotinas do programa).

No trecho de programa principal representado na Figura 9.14 encontram-se as definições da área de dados e da área de código, como de costume utilizadas até o momento.

O trecho de código de programa entre as linhas 14 e 17 apresenta a mensagem de entrada definida para a variável **msg1** pela chamada do procedimento **mensagem** (**CALL mensagem** na linha 15) e do procedimento **entrada** (**CALL entrada** na linha 16). A linha 17 exibe o armazenamento na pilha do valor atual do registrador geral **AX** que é o valor fornecido no teclado pela instrução **PUSH AX**.

O armazenamento do valor do registrador **AX** na pilha é necessário porque as próximas operações realizadas afetam o valor desse registrador. É necessário preservar esse valor para posterior resgate pelo comando **POP AX** definida na linha 21. Na sequência são encontradas as linhas 19 e 27 que apresentam as mensagens definidas para as variáveis **msg2** e **msg3** e a apresentação na tela do valor informado para que em seguida ocorra o cálculo do resultado da fatorial e a apresentação do resultado calculado.

Em especial observe o código da linha 21 (instrução **POP AX**) que resgata da pilha o valor de entrada anteriormente preservado (quando da execução da linha 17). Em seguida (linha 22) é feita a movimentação do valor (trazido por **POP AX** na lin há 21) do registrador **AL** para o registrador **DL** (nesse registrador está sendo colocado o valor que será apresentado). Depois o programa faz a apresentação do valor pelas linhas de programa 23 e 24. Na sequência a linha 25

faz a conversão do valor do código ASCII do registrador **AL** no seu valor numérico decimal correspondente e a linha **26** movimenta o valor agora decimal do registrador **AL** para o registrador **CL**.

O trecho de código situado entre as linhas 28 e 30 faz a apresentação do resultado da fatorial pelas chamadas dos procedimentos **fatorial** (CALL **fatorial** definido na linha 29) e **valor** (CALL **valor** definido na linha 30). Após a conclusão da operação o programa é finalizado com a instrução INT 20h da linha 33.

No trecho de programa representado na Figura 9.15 encontram-se as definições dos procedimentos **mensagem** (que apresenta as mensagens definidas para as variáveis **msg1**, **msg2**, **msg3** e **msg4**), **entrada** (que captura no teclado o valor numérico de um dígito) e **fatorial** (que calcula e armazena na memória o valor da fatorial).

edit: C:\Users\Augusto Manzano\Documents\LACO2.asm

file edit bookmarks assembler emulator math ascii codes help

new open examples save compile emulate calculator convertor options help about

```
34 mensagem PROC NEAR
35     MOV AH, 09h
36     INT 021h
37     RET
38 mensagem ENDP
39
40 entrada PROC NEAR
41     MOV AH, 01h
42     INT 021h
43     CMP AL, 030h
44     JL erro
45     CMP AL, 039h
46     JGE erro
47     JMP fim_validacao
48
49 erro:
50     LEA DX, msg4
51     CALL mensagem
52     JMP fim
53
54 fim_validacao:
55     RET
56 entrada ENDP
57
58 fatorial PROC NEAR
59     MOV AX, 01h
60     CMP CX, 0h
61     JE fim_laco
62     repital:
63         MUL CX
64         LOOPNE repital
65
66 fim_laco:
67
68 fim:
69
```

**Figura 9.15 - Programa LACO2 na ferramenta emu8086 - procedimentos.**

Os procedimentos **mensagem** e **entrada** de certa forma já são conhecidos e não requerem explicações. Atente para o procedimento **fatorial** situado entre as linhas 57 e 66.

Na linha **58** é feita a movimentação do valor **01h** para o registrador geral **AX**, sendo este o menor resultado do cálculo de um fatorial válido para a entrada dos valores **0** (zero) e **1** (um). As instruções **CMP CX, 0h** (linha **59**) e **JE fim\_laço** (linha **60**) verificam se o valor do registrador geral **CX** é zero, e ser for, desvia para o trecho de programa identificado como **fim\_laço** definido após a linha **64**. Esse trecho será executado quando o valor do registrador **CX** for zero; caso contrário, serão executadas as linhas de código **MUL CX** (linha **62**) e **LOOPNE repita1** (linha **63**) que calculam o resultado da fatorial.

Imagine que o valor informado para a obtenção do cálculo da fatorial seja 5. Nesse caso o procedimento (sub-rotina) **fatorial** realiza as seguintes operações:

- ◆ Lembre-se de que nesse momento, na execução do programa (linha 59) o registrador **CL** está com o valor **0005h** recebido do registrador **AL** pela execução da linha de código 26. Neste caso, a linha de código 58 faz a movimentação do valor **0001h** para o registrador geral **AX** e as linhas 59 e 60 verificam se o valor do registrador geral **CX** é zero. Se for, o procedimento é encerrado e se não for (que é o caso descrito), o programa será desviado para a linha 61.
  - ◆ A linha de código 62 multiplica o valor do registrador geral **CX** (valor **0005h**) pelo valor do registrador geral **AX** (valor **0001h**) e armazena o resultado da operação no par de registradores gerais **DX:AX**. Nesse momento o valor do registrador geral **AX** que é **0001h** é multiplicado pelo valor do registrador geral **CX** e passa a conter o valor **0005h**. O registrador geral **DX** possui neste momento o valor **0h**.

- ◆ Após o processamento da linha **62**, o programa, por meio da linha **63 (LOOPNE repita1)**, faz um retorno para a linha **61** enquanto o valor do registrador **CX** não for igual a zero.
- ◆ Ao retornar para a linha **62**, o registrador geral **CX** estará com o valor **0004h**, pois toda vez que uma instrução de laço é executada, o registrador geral **CX** é diminuído em **1h**.
- ◆ De volta à linha **62** ocorre nova multiplicação do valor atual do registrador geral **CX** (possui nesse momento **0004h**) pelo valor atual do registrador geral **AX** (possui nesse momento **0005h**). O registrador geral **AX** passa a ter o valor **0014h (20 em decimal)**.
- ◆ Na execução da linha **63** ocorre novo retorno para a linha **61** e na linha **62** por ser o valor do registrador geral **CX** diferente de **0000h**. O registrador geral **CX** estará com o valor **0003h**.
- ◆ De volta à linha **62** ocorre nova multiplicação do valor atual do registrador geral **CX** (possui nesse momento **03h**) pelo valor atual do registrador geral **AX** (possui nesse momento **0014h**). O registrador geral **AX** passa a ter valor **003Ch (60 em decimal)**.
- ◆ Na execução da linha **63** ocorre novo retorno para a linha **61** e na linha **62** por ser o valor do registrador geral **CX** diferente de **0000h**. O registrador geral **CX** estará com o valor **0002h**.
- ◆ De volta à linha **61** ocorre nova multiplicação na linha **61** do valor atual do registrador geral **CX** (possui nesse momento **0002h**) pelo valor atual do registrador geral **AX** (possui nesse momento **003Ch**). O registrador geral **AX** passa a ter o valor **0078h (120 em decimal)**.
- ◆ Na execução da linha **63** ocorre novo retorno para a linha **61** e na linha **62** por ser o valor do registrador geral **CX** diferente de **0000h**. O registrador geral **CX** estará com o valor **0001h**.
- ◆ De volta à linha **61** ocorre nova multiplicação do valor atual do registrador geral **CX** (possui nesse momento **0001h**) na linha **62** pelo valor atual do registrador geral **AX** (possui nesse momento **0078h**). O registrador geral **AX** passa a ter o valor **0078h (120 em decimal)**.
- ◆ Na execução da linha **63** ocorre novo retorno para a linha **61**. Neste momento o registrador geral **CX** estará com o valor **0000h**. O programa é transferido então para a linha **65** que encerra o procedimento **fatorial**.

No trecho de programa representado na Figura 9.16 encontra-se a definição do procedimento **valor** que apresenta o resultado da fatorial em notação decimal. Nessa etapa o trecho de programa situado na faixa da linha **72** até a linha **77** armazena o valor decimal na pilha (isso sempre ocorre de forma invertida) e o trecho das linhas **79** a **86** retira os valores da pilha, apresentando-os na forma decimal. Esse procedimento executa as seguintes ações:

- ◆ Na linha **69** é carregado o registrador geral **BX** com o valor **0Ah (10 em decimal)**. Esse registrador será usado para auxiliar o processo de conversão do valor em notação decimal.
- ◆ Na linha **70** o registrador geral **CX** é zerado e o mesmo ocorre com o registrador geral **DX** na linha **72**.
- ◆ Na linha **73** ocorre a divisão do valor atual do registrador geral **AX** (valor **0078h**) pelo valor atual do registrador geral **BX** (valor **00Ah**). O registrador geral **AX** passa a ter o valor do quociente da divisão **000Ch** e o registrador geral **DX** passa a possuir o valor do resto da divisão **0h**.
- ◆ Na linha **74** ocorre o armazenamento na pilha do valor armazenado no registrador geral **DX**.
- ◆ Na linha **75** ocorre um incremento de **0001h** no registrador geral **CX**. Este passa então a ter o valor **0001h**.
- ◆ Na linha **76** ocorre a comparação do valor atual do registrador **AX** (valor **000Ch**) com o valor **0h**. O valor do registrador geral **AX** foi diminuído, pois é o valor do quociente da divisão anterior.
- ◆ Na linha **77** ocorre o desvio para a linha de código **79** caso o valor do registrador geral **AX** não seja zero. Caso contrário o programa volta para a linha **71**.
- ◆ No retorno para a linha **71**, a linha **72** faz a limpeza do registrador geral **DX**. Na sequência (linha **73**) o programa efetua nova divisão do valor atual do registrador geral **AX (000Ch)** pelo valor do registrador geral **BX (000Ah)**. Nesse momento o registrador geral **AX** passa a ter o valor do quociente da divisão **0001h** e o registrador geral **DX** passa a ter o valor do resto da divisão **0002h**.
- ◆ Quando se executa a linha **74**, o valor do registrador geral **DX** é armazenado na pilha sobre o seu valor anterior, ou seja, nesse momento a pilha possui os valores **0000h** e **0002h**.
- ◆ As linhas **75, 76 e 77** efetuam, respectivamente, o ajuste do valor do registrador geral **CX** para **0002h** e transferem a execução do programa novamente para a linha **71**.

```

67 valor PROC NEAR
68     MOV    BX, 0Ah
69     SUB    CX, CX
70     repita2:
71         SUB    DX, DX
72         DIV    BX
73         PUSH   DX
74         INC    CX
75         CMP    AX, 0h
76         JNZ    repita2
77         saída:
78             POP    AX
79             ADD    AL, 30h
80             MOV    DL, AL
81             MOV    AH, 0Eh
82             INT    10h
83             DEC    CX
84             JNBE   saída
85             POP    DX
86             RET
87 valor ENDP
88

```

Figura 9.16 - Programa LACO1 na ferramenta emu8086 - procedimentos.

- ◆ De volta à linha 71 o programa limpa o registrador geral **DX**. Na sequência (linha 72) o programa efetua nova divisão do valor atual do registrador geral **AX (0001h)** pelo valor do registrador geral **BX (000Ah)**. Nesse momento o registrador geral **AX** passa a ter o valor do quociente da divisão **0000h** e o registrador geral **DX** passa a possuir o valor do resto da divisão **0001h**.
- ◆ Quando se executa a linha 74, o valor do registrador geral **DX** é armazenado na pilha sobre o seu valor anterior, ou seja, a pilha possui os valores **0000h**, **0002h** e **0001h**. Nesse momento a pilha possui de forma invertida os valores que formarão a imagem decimal do valor **120d**.
- ◆ As linhas 75, 76 e 77 ajustam, respectivamente, o valor do registrador geral **CX** para **0003h** e transferem a execução do programa para a linha 79, uma vez que o registrador geral **AX** passou a ter o valor **0h**.
- ◆ Na linha 79 ocorre a retirada da pilha do valor armazenado para o registrador geral **AX**. Depois na linha 80 ocorre a transformação do valor para o seu código ASCII equivalente.
- ◆ Na linha 81 ocorre a movimentação do valor do registrador menos significativo **AL** para o registrador menos significativo **DL** que armazena o código a ser apresentado na tela do monitor de vídeo. Em seguida com as linhas 82 e 83 ocorre a apresentação do primeiro valor puxado da pilha (neste caso o valor 1).
- ◆ Na linha 84 há a subtração (decremento) de **01h** do registrador geral **CX** (que está com o valor **0003h** e passa a possuir o valor **0002h**). Na sequência a linha 85 verifica, por meio da instrução **JNBE saída**, se o registrador geral **CX** está com o seu valor abaixo ou igual a zero. Caso não esteja, retorna o fluxo de execução do programa para a linha 78 que envia o processamento para a linha 79.
- ◆ A partir desse retorno para a linha 79 o programa retira o valor 2 da pilha. Em seguida da linha 80 até a linha 83 é feita a apresentação do valor retirado da pilha. A linha 84 ajusta o valor do registrador geral **CX** de **0002h** para **0001h** e executa o novo retorno à linha 78 que envia o processamento para a linha 79.
- ◆ Nessa última etapa de execução da linha 79 o programa retira o valor 1 da pilha. Em seguida entre as linhas 80 e 83 é feita a apresentação do valor retirado da pilha. A linha 84 ajusta o valor do registrador geral **CX** de **0001h** para **0000h**. Isso fará com que o programa seja desviado para a linha de código 86 que remove da pilha o valor 0 (último valor a ser retirado, que foi o primeiro a ser inserido).

A partir daí o programa é encerrado e o valor do cálculo do resultado da fatorial é apresentado.

## 9.6 - Utilização de macros

Em vários exemplos de programas anteriores foi empregada a programação estruturada baseada em **procedimentos**.

Os **procedimentos** utilizados na linguagem de programação Assembly 8086/8088 são similares às sub-rotinas usadas em algumas linguagens de alto nível.

Cabe apresentar ainda uma segunda forma de trabalhar com a programação estruturada. Nesse caso utilizando **macros**, que têm certa similaridade com funções em algumas linguagens de alto nível. Uma rotina de **macro** pode ser definida com e sem parâmetros. Ela tem a seguinte estrutura de sintaxe:

```
nome MACRO [parâmetro1, parâmetro2, ...]
    corpo da macro
ENDM
```

Uma rotina de macro deve ter um nome definido antes da diretiva **MACRO** e entre as diretivas **MACRO** e **ENDM** deve ser colocado o código a ser executado. A definição de parâmetros é opcional.

Basicamente tudo que é definido com procedimento também pode ser definido com macro. A diferença está no fato de que um procedimento é chamado com o comando **CALL** e uma rotina de macro é chamada como se fosse um comando da própria linguagem. Outra diferença entre esses dois mecanismos é que uma rotina de macro utiliza mais memória. Por essa razão deve ser usada com parcimônia.

Com base no código do programa **LAC02** serão feitas algumas mudanças para transformar o procedimento **mensagem** na macro **msg**. Observe a seguir os pontos marcados em negrito no código do programa:

```
;*****
;*      Programa: LAC03.ASM      *
;*****
```

```
org 100h

.DATA
    msg1 DB 'Entre valor decimal positivo (de 0 ate 8): ', 24h
    msg2 DB 0Dh, 0Ah, 'Fatorial de ', 24h
    msg3 DB ' equivale a ', 24h
    msg4 DB 0Dh, 0Ah, 'Valor invalido', 24h

.CODE
    LEA DX, msg1
msg
    CALL entrada
    PUSH AX

    LEA DX, msg2
msg
    POP AX
    MOV DL, AL
    MOV AH, 0Eh
    INT 10h
    SUB AL, 30h
    MOV CL, AL

    LEA DX, msg3
msg
    CALL factorial
    CALL valor

fim:
    INT 20h
```

```

msg MACRO
    MOV AH, 09h
    INT 21h
ENDM

entrada PROC NEAR
    MOV AH, 01h
    INT 21h
    CMP AL, 30h
    JL erro
    CMP AL, 39h
    JGE erro
    JMP fim_validacao
erro:
    LEA DX, msg4
    msg
    JMP fim
fim_validacao:
    RET
entrada ENDP

fatorial PROC NEAR
    MOV AX, 01h
    CMP CX, 0h
    JE fim_laco
repita1:
    MUL CX
    LOOPNE repita1
fim_laco:
    RET
fatorial ENDP

valor PROC NEAR
    PUSH AX
    MOV BX, 0Ah
    SUB CX, CX
repita2:
    SUB DX, DX
    DIV BX
    PUSH DX
    INC CX
    CMP AX, 0h
    JNZ repita2
saida:
    POP AX
    ADD AL, 30h
    MOV DL, AL
    MOV AH, 0Eh
    INT 10h
    DEC CX
    JNBE saida
    POP DX
    RET
valor 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 por meio dos comandos de menu **file/save** com o nome **LACO3**.

Nos pontos em que existia a instrução **CALL mensagem** agora existe apenas a chamada da rotina de macro **msg**. Atente também para a mudança do código da rotina de procedimento **mensagem** para o código da rotina de macro **msg** exemplificado a seguir:

```

msg MACRO
    MOV AH, 09h
    INT 21h
ENDM

```

A rotina de macro é de certa forma semelhante à de um procedimento. Houve a supressão do comando **RET**, pois macros não a utilizam porque fazem o retorno automaticamente.

O exemplo anterior apresentou a rotina de *macro* sem parâmetro. No entanto, o uso de parâmetro pode ser muito vantajoso. Tome por base o código seguinte, observando as linhas grafadas em negrito:

```

;*****  

;* Programa: LAC04.ASM *  

;*****  

org 100h  

.DATA  

msg1 DB 'Entre valor decimal positivo (de 0 ate 8): ', 24h  

msg2 DB 0Dh, 0Ah, 'Fatorial de ', 24h  

msg3 DB ' equivale a ', 24h  

msg4 DB 0Dh, 0Ah, 'Valor invalido', 24h  

.CODE  

msg msg1  

CALL entrada  

PUSH AX  

msg msg2  

POP AX  

MOV DL, AL  

MOV AH, 0Eh  

INT 010h  

SUB AL, 30h  

MOV CL, AL  

msg msg3  

CALL fatorial  

CALL valor  

fim:  

INT 20h  

msg MACRO mensagem  

    LEA DX, mensagem  

    MOV AH, 09h  

    INT 21h  

ENDM  

entrada PROC NEAR  

    MOV AH, 01h  

    INT 021h  

    CMP AL, 30h  

    JL erro  

    CMP AL, 39h  

    JGE erro  

    JMP fim_validacao  

erro:  

msg msg4  

JMP fim

```

```

fim_validacao:
RET
entrada ENDP

fatorial PROC NEAR
MOV AX, 01h
CMP CX, 0h
JE fim_laco
repita1:
MUL CX
LOOPNE repita1
fim_laco:
RET
fatorial ENDP

valor PROC NEAR
PUSH AX
MOV BX, 0Ah
SUB CX, CX
repita2:
SUB DX, DX
DIV BX
PUSH DX
INC CX
CMP AX, 0h
JNZ repita2
saida:
POP AX
ADD AL, 30h
MOV DL, AL
MOV AH, 0Eh
INT 010h
DEC CX
JNBE saida
POP DX
RET
valor 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 por meio dos comandos de menu **file/save** com o nome **LACO4**.

O código da rotina de *macro msg* tem agora a definição de um parâmetro denominado **mensagem** que receberá o conteúdo da mensagem a ser apresentada.