

```

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

```

# Capítulo 12

## COMO EM ALTO NÍVEL

Ao longo da apresentação dos capítulos anteriores foram mostradas diversas técnicas simples de trabalho para que se pudesse produzir alguns programas em linguagem de programação Assembly. Neste sentido, foram focadas ações básicas de entrada, processamento e saída. Este capítulo apresenta alguns exemplos de programas complementares, utilizando-se a visão da programação de alto nível para produzir códigos em baixo nível equivalentes para programadores acostumados com o modo de programação em alto nível. São aqui apresentados pequenos programas com enfoque contextualizado a algumas ações básicas para efetuar algumas ações de processamento a partir dos recursos internos existente no ambiente de programação emu8086. Os exemplos de programas apresentados caracterizam-se por serem uma solução possível, mas não necessariamente são as melhores soluções para os problemas apresentados, são as soluções mais didáticas possíveis dentro do escopo deste trabalho.

### 12.1 - Tomada de decisão

A linguagem de programação Assembly não possui de forma explícita, como ocorre nas linguagens de alto nível, instruções de ação direta para a efetivação de tomadas de decisão. Para que este procedimento ocorra é necessário fazer uso de duas ou mais instruções para tal finalidade. Isto ocorre com o uso dos conceitos de operadores relacionais, operadores lógicos, decisões simples, decisões compostas e decisões seletivas.

A tomada de decisão é efetivada com o uso da instrução **CMP** seguida de uma instrução de desvio condicional que fará o direcionamento do fluxo do programa para o ponto desejado. Além da instrução **CMP** pode-se também fazer uso das instruções de operações lógicas **AND** (conjunção), **OR** (disjunção inclusiva), **NOT** (negação) e **XOR** (disjunção exclusiva) e **NOT** (negação).

Os desvios efetivados após o uso de uma das instruções **CMP**, **AND**, **OR**, **NOT** e **XOR** são feitos com o uso de instruções de saltos que efetuam ações similares as ações do uso dos operadores relacionais. A Tabela 12.1 apresenta um paralelo entre os operadores relacionais de uma linguagem de alto nível com as instruções de saltos da linguagem de programação Assembly dos microprocessadores padrão 8086/8088.

Tabela 12.1 - Tabela Comparativa de Uso das Relações Lógicas

Operador Relacional	Instrução Assembly
=	JE
>	JG
>=	JGE
<	JL
<=	JLE
<>	JNE

Além da existência de instruções para efetivação de ações similares ao uso de operadores relacionais, existem as instruções de ação para efetivação de operadores lógicos. A título de revisão apresenta-se a seguir as tabelas verdadeas

para os operadores lógicos a serem utilizados com a definição de duas condições. Veja as tabelas 12.2, 12.3, 12.4 e 12.5.

Tabela 12.2 - Operador lógico AND

Tabela verdade do operador lógico de conjunção – AND		
Condição 1	Condição 2	Resultado lógico
Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Falso
Falso	Verdadeiro	Falso
Falso	Falso	Falso

Tabela 12.3 - Operador lógico OR

Tabela verdade do operador lógico de disjunção inclusiva - OR		
Condição 1	Condição 2	Resultado lógico
Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Verdadeiro
Falso	Verdadeiro	Verdadeiro
Falso	Falso	Falso

Tabela 12.4 - Operador lógico XOR

Tabela verdade do operador lógico de disjunção exclusiva – XOR		
Condição 1	Condição 2	Resultado lógico
Verdadeiro	Verdadeiro	Falso
Verdadeiro	Falso	Verdadeiro
Falso	Verdadeiro	Verdadeiro
Falso	Falso	Falso

Tabela 12.5 - Operador lógico NOT

Tabela verdade do operador lógico de negação - NOT	
Condição	Resultado lógico
Verdadeiro	Falso
Falso	Verdadeiro

As ações para os operadores **AND**, **OR** e **XOR** possuem instruções próprias em linguagem Assembly. No entanto, para uso do operador lógico **NOT** deve-se usar as instruções de salto que possuem na composição de seus nomes o segundo caractere definido como **N** para as instruções **JNx** ou **JNy**, onde **x** pode ser **A, B, E, G, L, O, P, S e Z** e **yy** pode ser **AE, BE, GE e LE**. No caso das ações **AND** e **OR** estas serão executadas sem o uso desses operadores por meio de tomadas de decisão e para o uso do conceito **XOR** será usado a respectiva instrução.

Por uma questão de legibilidade os exemplos a seguir fazem um paralelo entre a forma escrita em alto nível e sua equivalência em baixo nível, e no sentido de manter uma visão didática apresentam em baixo nível alguns detalhes escritos que poderão com o tempo ser removidos

## 12.1.1 - Decisão simples

A seguir é indicada a forma de representação de uma tomada de decisão simples em pseudocódigo português estruturado, similar a forma existente nas linguagens de programação de alto nível e sua equivalência em linguagem de programação Assembly.

### Tomada de Decisão Simples em Português Estruturado

```
se ( <elemento 1> <operador relacional> <elemento 2> ) então
    [instruções executadas após condição ser verdadeira]
fim_se
    [instruções executadas após condição ser falsa ou ser verdadeira]
```

## Tomada de Decisão Simples em Assembly

```
se:  
    CMP <elemento 1>, <elemento 2>  
    <instrução condicional de salto> fim_se  
entao:  
    [instruções executadas após condição ser verdadeira]  
fim_se:  
    [instruções executadas após condição ser falsa ou ser verdadeira]
```

A indicação **elemento 1** e **elemento 2** caracterizam-se por ser a definição do uso de variáveis e/ou constantes na formação de uma condição. Esta relação pode ser composta com a partir de: variáveis versus variáveis ou variáveis versus constantes.

Na definição da condição em português estruturado encontra-se a indicação de uso de um **operador relacional** que está paralelamente assinalado no corpo do código em assembly como sendo uma **instrução condicional de salto**.

Após o rótulo **entao:** são definidas as instruções de ação para a condição verdadeira. Caso a condição não seja satisfeita a **instrução condicional de salto** desviará o fluxo de execução do programa para o trecho de instruções definidos após o rótulo **senao:**.

Para exemplificar o uso de tomada de decisão simples considere um programa que efetue a entrada de um valor numérico e mostre a mensagem "**Valor acima de 10**" se o valor informado for maior que dez. Caso contrário o programa não deverá apresentar nenhuma mensagem.

```
;*****  
;*  Programa: DECISA01.ASM  *  
;*****  
  
INCLUDE 'emu8086.inc'  
  
.org 100h  
  
.DATA  
    valor DW 0  
    msg1 DB 'Entre um valor numerico: ', 0  
    msg2 DB 'Valor acima de 10.', 0  
  
.CODE  
    LEA SI, msg1  
    CALL PRINT_STRING  
    CALL SCAN_NUM  
    MOV valor, CX  
    PUTC 13d  
    PUTC 10d  
  
se:  
    CMP valor, 10d  
    JLE fim_se  
entao:  
    LEA SI, msg2  
    CALL PRINT_STRING  
fim_se:  
  
INT 20h  
DEFINE_PRINT_STRING  
DEFINE_SCAN_NUM  
  
END
```

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 **DECISA01**.

```

edit: C:\Users\Augusto Manzano\Documents\DECISA01.asm
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about
01;***** Programa: DECISA01.ASM ****
02;
03;
04;
05 INCLUDE 'emu8086.inc'
06
07 org 100h
08
09 .DATA
10 valor DW 0
11 msg1 DB 'Entre um valor numerico: ', 0
12 msg2 DB 'Valor acima de 10.', 0
13
14 .CODE
15 LEA SI, msg1
16 CALL PRINT_STRING
17 CALL SCAN_NUM
18 MOV valor, CX
19 PUTC 13d
20 PUTC 10d
21
22 se:
23 CMP valor, 10d
24 JLE fim_se
25 entao:
26 LEA SI, msg2
27 CALL PRINT_STRING
28 fim_se:
29
30 INT 20h
31 DEFINE_PRINT_STRING
32 DEFINE_SCAN_NUM
33
34 END
35

```

Figura 12.1 - Programa DECISA01 na ferramenta emu8086.

Observe na linha 05 a instrução **INCLUDE 'emu8086.inc'** que faz a chamada da biblioteca **emu8086.inc** que possui as funções **PRINT\_STRING / DEFINE\_PRINT\_STRING** e **SCAN\_NUM / DEFINE\_SCAN\_NUM**. Desta forma, aproveita-se os recursos comuns que podem ser usados na escrita de qualquer programa e direciona-se o foco de trabalho para as ações mais incomuns.

Na linha 10 (**valor DW 0**) está sendo definida uma variável de nome **valor** com valor inicial **0** (zero). Caso queira definir uma variável em memória sem um valor inicial, basta no lugar do valor usar o caractere **?** (interrogação) com a sintaxe **<variável> DW ?**. Esta situação será exposta mais adiante.

A linha 15 (**LEA SI, msg1**) coloca, aponta, no registrador de ponteiro **SI** o endereço de memória onde se encontra a variável **msg1**. Esta forma está sendo utilizada devido ao fato do procedimento **PRINT\_STRING** da biblioteca **emu8086.inc** indicado na linha 16 fazer uso do registrador de ponteiro **SI** para acessar o conteúdo *string* da variável indicada. Neste caso, variável **msg1**.

O procedimento **SCAN\_NUM** na linha 17 efetua a leitura de um número inteiro fornecido via teclado e o armazena no registrador geral **CX**, devido a esta ocorrência é que a linha 18 (**MOV valor, CX**) transfere o valor do registrador geral **CX** para a variável **valor**.

O ponto do programa que mais interessa é o trecho de código escrito entre as linhas 22 e 28 que estabelecem as instruções de uma tomada de decisão simples. A linha 23 (**CMP valor, 10d**) efetua a comparação do conteúdo da variável **valor** com o valor decimal **10** e em conjunto com a linha 24 (**JLE fim\_se**) que faz a vez a tomada de decisão e verifica se a condição estabelecida é falsa ou verdadeira. Na linha 24 está ocorrendo o uso da instrução **JLE** que faz o papel de uso de um operador relacional. Assim sendo, a condição considerada para a tomada da decisão é **valor > 10** a qual executará as instruções das linhas de 25 até 27 caso a condição seja verdadeira e apresentará a mensagem “**Valor acima de 10.**”. No entanto, se a linha 24 detectar que o conteúdo da variável **valor** é menor ou igual a **10** efetua o seu desvio para a linha 28 identificada com o rótulo **fim\_se**: e desta forma não apresentará a mensagem.

Aos olhos de alguns programadores a condição estabelecida nas linhas 23 e 24 pode parecer estar em sentido contrário, pois a instrução **JLE** (*Jump on Less or Equal*) efetua um desvio quando a condição verificada com a instrução **CMP** for menor ou igual ao valor estabelecido. Mas tudo é uma questão de ponto de vista, ou seja, de ótica condicional, pois se o pensamento estabelecido para a condição é **valor > 10** e sendo esta condição verdadeira ocorrerá automaticamente a execução das linhas de instrução de 25 até 27 após a avaliação da instrução da linha 24 e quando o resultado da condição **valor > 10** for falso, ou seja, quando o conteúdo da variável **valor** for menor ou igual a **10** será então des-

viado o fluxo de execução do programa para a linha 28. Neste ponto é bom ter o máximo de cuidado para não criar na mente conceitos equivocados.

## 12.1.2 - Decisão Composta

A seguir é indicada a forma de representação de uma tomada de decisão composta em pseudocódigo português estruturado, similar a forma existente nas linguagens de programação de alto nível e sua equivalência em linguagem de programação Assembly.

### Tomada de Decisão Composta em Português Estruturado

```
se ( <elemento 1> <operador relacional> <elemento 2> ) então
    [instruções executadas após condição ser verdadeira]
senão
    [instruções executadas após condição ser falsa]
fim_se
[instruções executadas após condição ser falsa ou ser verdadeira]
```

### Tomada de Decisão Composta em Assembly

```
se:
    CMP <elemento 1>, <elemento 2>
    <instrução condicional de salto> senao
entao:
    [instruções executadas após condição ser verdadeira]
    JMP fim_se
senão:
    [instruções executadas após condição ser falsa]
fim_se:
[instruções executadas após condição ser falsa ou ser verdadeira]]
```

A indicação **elemento 1** e **elemento 2** caracterizam-se por ser a definição do uso de variáveis e/ou constantes na formação de uma condição. Esta relação pode ser composta com a partir de: variáveis versus variáveis ou variáveis versus constantes.

Na definição da condição em português estruturado encontra-se a indicação de uso de um **operador relacional** que está paralelamente assinalado no corpo do código em assembly como sendo uma **instrução condicional de salto**.

Após o rótulo **entao:** são definidas as instruções de ação para a condição verdadeira e após o término da execução dessas instruções o programa por meio da instrução **JMP fim\_se** efetua um salto para o rótulo **fim\_se:** e encerra a execução do bloco de tomada de decisão. Caso a condição não seja satisfeita a **instrução condicional de salto** desviará o fluxo de execução do programa para o trecho de instruções definidos após o rótulo **senao:**.

Para exemplificar o uso de tomada de decisão simples considere um programa que efetue a entrada de um valor numérico e mostre a mensagem "**Valor igual ou acima de 10**" se o valor informado for maior ou igual a dez. Caso contrário o programa deverá apresentar a mensagem "**Valor abaixo de 10**".

```
;*****
;*  Programa: DECISA02.ASM  *
;*****
```

```
INCLUDE 'emu8086.inc'

org 100h
```

```

.DATA
valor DW 0
msg1 DB 'Entre um valor numerico: ', 0
msg2 DB 'Valor igual ou acima de 10.', 0
msg3 DB 'Valor abaixo de 10.', 0

.CODE
LEA SI, msg1
CALL PRINT_STRING
CALL SCAN_NUM
MOV valor, CX
PUTC 13d
PUTC 10d

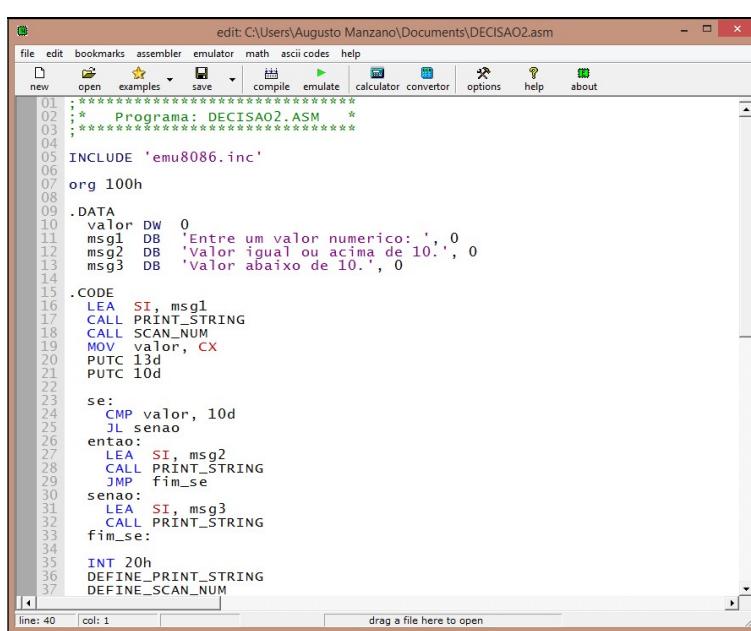
se:
CMP valor, 10d
JL senao
entao:
LEA SI, msg2
CALL PRINT_STRING
JMP fim_se
senao:
LEA SI, msg3
CALL PRINT_STRING
fim_se:

INT 20h
DEFINE_PRINT_STRING
DEFINE_SCAN_NUM

END

```

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 **DECISA02**, de forma que fique semelhante à imagem da Figura 12.2.



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

```

01 ;*****
02 * Programa: DECISA02.ASM *
03 ;*****
04
05 INCLUDE 'emu8086.inc'
06
07 org 100h
08
09 .DATA
10 valor DW 0
11 msg1 DB 'Entre um valor numerico: ', 0
12 msg2 DB 'Valor igual ou acima de 10.', 0
13 msg3 DB 'Valor abaixo de 10.', 0
14
15 .CODE
16 LEA SI, msg1
17 CALL PRINT_STRING
18 CALL SCAN_NUM
19 MOV valor, CX
20 PUTC 13d
21 PUTC 10d
22
23 se:
24 CMP valor, 10d
25 JL senao
26 entao:
27 LEA SI, msg2
28 CALL PRINT_STRING
29 JMP fim_se
30 senao:
31 LEA SI, msg3
32 CALL PRINT_STRING
33 fim_se:
34
35 INT 20h
36 DEFINE_PRINT_STRING
37 DEFINE_SCAN_NUM

```

The code implements a decision-making structure where it asks for a numeric value, compares it with 10, and prints different messages based on the result. It also includes standard assembly language definitions like .DATA and .CODE sections, and system calls for input/output.

Figura 12.2 - Programa DECISA02 na ferramenta emu8086.

O ponto do programa que mais interessa é o trecho de código escrito entre as linhas 23 e 33 que estabelecem as instruções de uma tomada de decisão composta. A linha 24 (**CMP valor, 10d**) efetua a comparação do conteúdo da variável

vel **valor** com o valor decimal **10** e em conjunto com a linha **25 (JL fim\_se)** que faz a vez a tomada de decisão e verifica se a condição estabelecida é falsa ou verdadeira. Na linha **25** está ocorrendo o uso da instrução **JL** que faz o papel de uso de um operador relacional. Assim sendo, a condição considerada para a tomada da decisão é **valor >= 10** a qual executará as instruções das linhas de **27** até **29** caso a condição seja verdadeira e apresentará a mensagem “**Valor igual ao acima de 10.**”. No entanto, se a linha **25** detectar que o conteúdo da variável **valor** é menor que **10** efetua o seu desvio para a linha **30** identificada com o rótulo **senão:** e executa a apresentação da mensagem “**Valor abaixo de 10.**”.

### 12.1.3 - Decisões com Duas Condições

Neste tópico serão apresentados exemplos que farão uso dos conceitos de aplicação dos operadores lógicos **AND**, **OR** e **XOR**. Todos os exemplos aqui apresentados estarão baseados na estrutura de tomada de decisão composta com base no seguinte estilo.

Para a efetivação de uma ação de tomada de decisão com base no uso do conceito do operador lógico de conjunção **AND** considere a estrutura de código seguinte:

#### Tomada de Decisão com Operador AND em Português Estruturado

```
se (condição1) .e. (condição2) então
    [instruções executadas após condição1 e condição2 serem verdadeiras]
senão
    [instruções executadas caso uma ou ambas as condições sejam falsas]
fim_se
```

#### Tomada de Decisão com Operador AND em Assembly

```
se:
    CMP <condição1>
    <instrução condicional de salto> senao
e:
    CMP <condição2>
    <instrução condicional de salto> senao
entao:
    [instruções executadas após condição1 e condição2 serem verdadeiras]
    JMP fim_se
senão:
    [instruções executadas caso uma ou ambas as condições sejam falsas]
fim_se:
```

O programa seguinte efetua a entrada de um valor numérico inteiro em uma variável chamada **valor** e apresenta a mensagem “**Valor esta na faixa de 10 a 90**” caso o valor fornecido seja maior ou igual a **10** e menor ou igual a **90**. Caso esta condição não seja satisfeita o programa apresentará a mensagem “**Valor esta fora da faixa de 10 a 90**”. Note que este programa faz uso do operador lógico de conjunção com a condição **valor >= 10 e valor <= 90**.

```
;*****
;*  Programa: DECISA03.ASM   *
;*****
```

```
INCLUDE 'emu8086.inc'

org 100h
```

```

.DATA
valor DW 0
msg1 DB 'Entre um valor numerico: ', 0
msg2 DB 'Valor esta na faixa de 10 a 90.', 0
msg3 DB 'Valor esta fora da faixa de 10 a 90.', 0

.CODE
LEA SI, msg1
CALL PRINT_STRING
CALL SCAN_NUM
MOV valor, CX
PUTC 13d
PUTC 10d

se:
CMP valor, 10d
JL senao

e:
CMP valor, 90d
JG senao

entao:
LEA SI, msg2
CALL PRINT_STRING
JMP fim_se

senao:
LEA SI, msg3
CALL PRINT_STRING

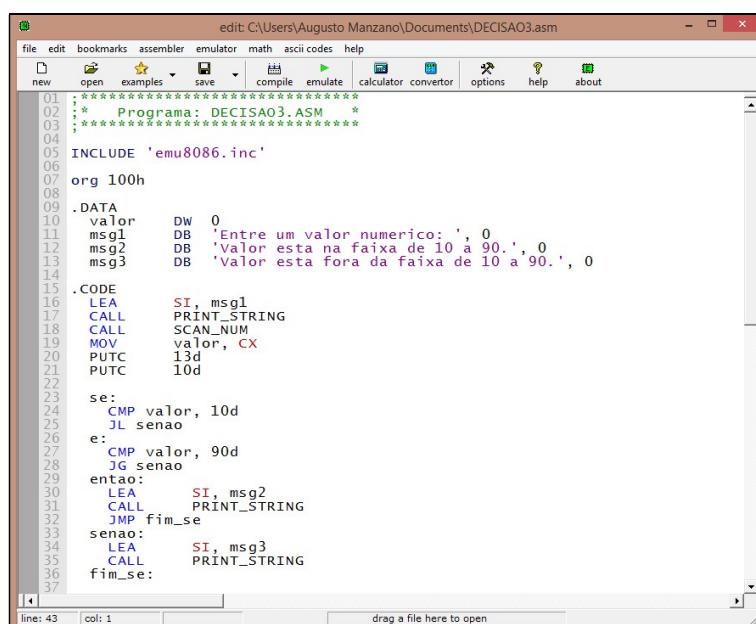
fim_se:

INT 20h
DEFINE_PRINT_STRING
DEFINE_SCAN_NUM

END

```

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 **DECISA03**, de forma que fique semelhante à imagem da Figura 12.3.



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

```

01 ;***** Programa: DECISA03.ASM *****
02 ;*****
03 ;
04 ;
05 INCLUDE 'emu8086.inc'
06
07 org 100h
08
09 .DATA
10 valor DW 0
11 msg1 DB 'Entre um valor numerico: ', 0
12 msg2 DB 'Valor esta na faixa de 10 a 90.', 0
13 msg3 DB 'Valor esta fora da faixa de 10 a 90.', 0
14
15 .CODE
16 LEA SI, msg1
17 CALL PRINT_STRING
18 CALL SCAN_NUM
19 MOV valor, CX
20 PUTC 13d
21 PUTC 10d
22
23 se:
24 CMP valor, 10d
25 JL senao
26
27 e:
28 CMP valor, 90d
29 entao:
30 LEA SI, msg2
31 CALL PRINT_STRING
32 JMP fim_se
33
34 senao:
35 LEA SI, msg3
36 CALL PRINT_STRING
37

```

Figura 12.3 - Programa DECISA03 na ferramenta emu8086.

O ponto do programa que mais interessa é o trecho de código entre as linhas **23** e **36** que estabelecem as instruções de uma tomada de decisão composta com o uso da ação de um operador lógico de conjunção **AND** (e em português). A linha **24 (CMP valor, 10d)** efetua a comparação do conteúdo da variável **valor** com o valor decimal **10** (primeira condição) e a linha **25 (JL senao)** faz o desvio para a linha **33** (identificada pelo rótulo **senao:**) caso o conteúdo da variável **valor** não seja maior ou igual a **10** apresentando a mensagem “**Valor esta fora da faixa de 10 a 90.**” e encerra a tomada da decisão. Se a condição estabelecida nas linhas **24** e **25** for maior ou igual a **10** o fluxo de execução do programa avança automaticamente para as linhas de **26** até **28** e efetua nas linhas **27 (CMP valor, 90d)** e **28 (JG senao)** a verificação da segunda condição que caso não seja menor ou igual a **90** desvia o fluxo de execução do programa para a linha **33** e apresenta a mensagem “**Valor esta fora da faixa de 10 a 90.**” e encerra a tomada da decisão. Caso o resultado da condição das linhas **27** e **28** não seja verdadeiro o fluxo do programa segue automaticamente a sequência estabelecida entre as linhas **29** e **32** apresentando a mensagem “**Valor esta na faixa de 10 a 90.**” E desviando o fluxo do programa para a linha **36** onde ocorrer o encerramento da decisão.

Para a efetivação de uma ação de tomada de decisão com base no uso do conceito do operador lógico de disjunção inclusiva **OR** considere a estrutura de código seguinte:

### Tomada de Decisão Composta com Operador OR em Português Estruturado

```
se (condição1) .ou. (condição2) então
    [instruções executadas quando pelo menos uma das condições for verdadeira]
senão
    [instruções executadas caso ambas as condições sejam falsas]
fim_se
```

### Tomada de Decisão Composta com Operador OR em Assembly

```
se:
    CMP <condição1>
    <instrução condicional de salto> senao
ou:
    CMP <condição2>
    <instrução condicional de salto> senao
entao:
    [instruções executadas quando pelo menos uma das condições for verdadeira]
    JMP fim_se
senão:
    [instruções executadas caso ambas as condições sejam falsas]
fim_se:
```

O programa a seguir solicita a sexo de uma pessoa e apresenta uma mensagem informando se o sexo fornecido é ou não válido. Será permitida a entrada dos valores **1** para sexo masculino e **2** para sexo feminino, qualquer outra entrada resultará na apresentação de uma mensagem de erro. As mensagens a serem apresentadas são: “**Sexo valido**” para informações de sexo como **1** ou **2** e “**Sexo invalido**” para qualquer outra entrada diferente de **1** ou **2**. Note que este programa faz uso do operador lógico de disjunção inclusiva com a condição **sexo = 1 ou sexo = 2**.

```
;*****
;*  Programa: DECISA04.ASM   *
;*****
```

```
INCLUDE 'emu8086.inc'

org 100h
```

```

.DATA
sexo DW 0
msg1 DB 'Informe sexo - [1] Masculino ou [2] Feminino: ', 0
msg2 DB 'Sexo invalido.', 0
msg3 DB 'Sexo valido.', 0

.CODE
LEA SI, msg1
CALL PRINT_STRING
CALL SCAN_NUM
MOV sexo, CX
PUTC 13d
PUTC 10d

.se:
CMP sexo, 1d
JE senao

.ou:
CMP sexo, 2d
JE senao

.entao:
LEA SI, msg2
CALL PRINT_STRING
JMP fim_se

.senao:
LEA SI, msg3
CALL PRINT_STRING

fim_se:

INT 20h
DEFINE_PRINT_STRING
DEFINE_SCAN_NUM

END

```

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 **DECISA04**, de forma que fique semelhante à imagem da Figura 12.4.

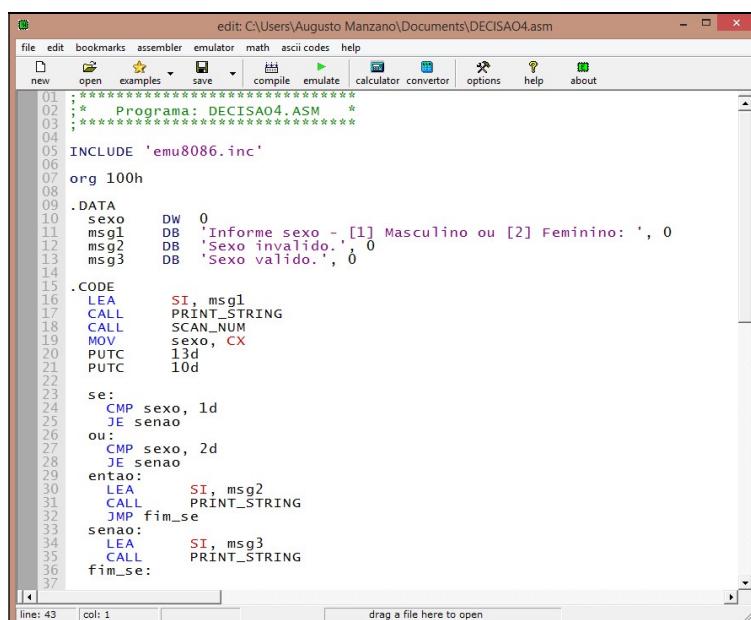


Figura 12.4 - Programa DECISA04 na ferramenta emu8086.

O ponto do programa que interessa é o trecho de código entre as linhas **23** e **36** que estabelecem as instruções de uma tomada de decisão composta com o uso da ação de um operador lógico de disjunção inclusiva **OR** (ou em português). A linha **24 (CMP sexo, 1d)** efetua a comparação do conteúdo da variável **sexo** com o valor decimal **1** (primeira condição) e a linha **25 (JE senao)** faz o desvio para a linha **33** (identificada pelo rótulo **senao:**) caso o conteúdo da variável **sexo** seja igual a **1** e neste caso será apresentada a mensagem “**Sexo válido**” e faz-se o encerramento da tomada de decisão. Caso o sexo informado não seja **1** o programa continua o seu fluxo de execução a partir da linha **26** onde encontra a segunda condição: linha **27 (CMP sexo, 2d)** que se verificada com resposta verdadeira pela linha **28 (JE senao)** faz o desvio para a linha **33** e apresenta a mensagem “**Sexo valido.**” direcionando para o encerramento da tomada de decisão. Caso as condições das linhas **24 e 25** e das linhas **27 e 28** não sejam verdadeiras ocorrerá a execução das instruções das linhas de **29** até **32**. Neste caso, será apresentada a mensagem “**Sexo invalido.**” e a instrução da linha **32 (JMP fim\_se)** direcionará o fluxo para o encerramento da tomada de decisão.

Para a efetivação de uma ação de tomada de decisão com uso do operador lógico de disjunção exclusiva **XOR** considere a estrutura de código seguinte:

### Tomada de Decisão Composta com Operador XOR em Português Estruturado

```
se (condição1) .xou. (condição2) então
    [instruções executadas quando condição1 verdadeira e condição2 falsa ou quando
     [condição1 falsa e condição2 verdadeira]
senão
    [instruções executadas caso as condições sejam falsas ou sejam verdadeiras]
fim_se
```

### Tomada de Decisão Composta com Operador XOR em Assembly

```
se:
    MOV <registrarador>, <variável condição1>
    XOR <registrarador>, <variável condição2>
    <instrução condicional de salto> senao
entao:
    [instruções executadas quando condição1 verdadeira e condição2 falsa ou quando
     [condição1 falsa e condição2 verdadeira]
    JMP fim_se
senão:
    [instruções executadas caso as condições sejam falsas ou sejam verdadeiras]
fim_se:
```

A indicação **variável condição1** e **variável condição2** caracterizam-se por ser a definição dos valores condicionais a serem avaliados.

O próximo programa será usado para definir a composição de um par de dança entre dois participantes, sendo permitido apenas a formação de pares de sexos diferentes. Pares de mesmo sexo não serão permitidos. Assim sendo, o programa apresentará a mensagem “**O 1º participante dança com o 2º participante.**” se o sexo informado para o primeiro participante for masculino e para o segundo participante for feminino ou vice-versa. Caso sejam informados sexos feminino ou masculino para ambos os participantes o programa deverá apresentar a mensagem “**O 1º participante não dança com o 2º participante.**”

```

;*****
;* Programa: DECISA05.ASM *
;*****

INCLUDE 'emu8086.inc'

org 100h

.DATA
sexo1 DW 0
sexo2 DW 0
msg1 DB 'Informe sexo participante 1 - [1] Masc. ou [2] Fem. : ', 0
msg2 DB 'Informe sexo participante 2 - [1] Masc. ou [2] Fem. : ', 0
msg3 DB 'O 1o participante danca com o 2o participante.', 0
msg4 DB 'O 1o participante nao danca com o 2o participante.', 0

.CODE
LEA    SI, msg1
CALL   PRINT_STRING
CALL   SCAN_NUM
MOV    sexo1, CX
PUTC  13d
PUTC  10d

LEA    SI, msg2
CALL   PRINT_STRING
CALL   SCAN_NUM
MOV    sexo2, CX
PUTC  13d
PUTC  10d

se:
MOV AX, sexo1
XOR AX, sexo2
JE senao
entao:
LEA    SI, msg3
CALL   PRINT_STRING
JMP fim_se
senao:
LEA    SI, msg4
CALL   PRINT_STRING
fim_se:

INT    20h
DEFINE_PRINT_STRING
DEFINE_SCAN_NUM

END

```

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 **DECISA05**, de forma que fique semelhante à imagem da Figura 12.5.

```

01 ;***** Programa: DECISAO5.ASM *****
02 ;***** Programa: DECISAO5.ASM *****
03 ;***** Programa: DECISAO5.ASM *****
04
05 INCLUDE 'emu8086.inc'
06
07 org 100h
08
09 .DATA
10 sexo1 DW 0
11 sexo2 DW 0
12 msg1 DB 'Informe sexo participante 1 - [1] Masc. ou [2] Fem. : ',0
13 msg2 DB 'Informe sexo participante 2 - [1] Masc. ou [2] Fem. : ',0
14 msg3 DB 'O 1o participante danca com o 2o participante.',0
15 msg4 DB 'O 1o participante nao danca com o 2o participante.',0
16
17 .CODE
18 LEA SI, msg1
19 CALL PRINT_STRING
20 CALL SCAN_NUM
21 MOV sexo1, CX
22 PUTC 13d
23 PUTC 10d
24
25 LEA SI, msg2
26 CALL PRINT_STRING
27 CALL SCAN_NUM
28 MOV sexo2, CX
29 PUTC 13d
30 PUTC 10d
31
32 se:
33 MOV AX, sexo1
34 XOR AX, sexo2
35 JE senao
36 entao:
37 LEA SI, msg3

```

Figura 12.5 - Programa DECISAO5 na ferramenta emu8086.

O ponto do programa que interessa é o trecho de código entre as linhas 32 e 43 que estabelecem as instruções de uma tomada de decisão composta com o uso do operador lógico de disjunção exclusiva **XOR** (xou em português estruturado). A linha 33 (**MOV AX, sexo1**) movimenta para o registrador geral **AX** o valor armazenado na variável **sexo1**, em seguida a linha 34 (**XOR AX, sexo**) efetua a comparação do conteúdo da variável **sexo1** armazenado no registrador geral **AX** com o valor armazenado na variável **sexo2** (a instrução **XOR** não opera sua ação direta sobre duas variáveis). A linha 35 (**JE senao**) faz o desvio para a linha 40 (identificada pelo rótulo **senao:**) caso o conteúdo avaliado pela instrução **XOR** seja 0 (zero). O valor 0 (zero) obtido pela instrução **XOR** acontece quando os valores avaliados são iguais, ou seja, é considerado falso. Caso os valores avaliados pela instrução **XOR** sejam diferentes o retorno será um valor 1 (um) que equivale a verdadeiro. Sendo a condição verdadeira será apresentada a mensagem “**O 1o participante danca com o 2o participante.**”, caso contrário ocorre a apresentação da mensagem “**O 1o participante nao danca com o 2o participante.**”.

## 12.1.4 - Decisões Sequências

A aplicação de decisões sequenciais se caracterizam quando se faz uso de várias tomadas de decisão simples ou compostas uma após a outra. Para a efetivação de uma ação de tomada de decisão sequencial considere a estrutura de código seguinte:

### Tomada de Decisão Sequencial em Português Estruturado

```

se (condição1) então
  [instruções executadas após condição1 ser verdadeira]
fim_se

se (condição2) então
  [instruções executadas após condição2 ser verdadeira]
fim_se

```

## Tomada de Decisão Sequencial em Assembly

```
se1:  
    CMP <condição1>  
    <instrução condicional de salto> fim_se1  
entao1:  
    [instruções executadas após condição1 ser verdadeira]  
fim_se1:  
  
se2:  
    CMP <condição2>  
    <instrução condicional de salto> fim_se2  
entao2:  
    [instruções executadas após condição2 ser verdadeira]  
fim_se2:
```

No sentido de exemplificar este tipo de ocorrência considere um programa que solicite a entrada de um valor numérico (variável N) e apresente uma das seguintes mensagens: "Você entrou o valor 1." se for dada a entrada do valor numérico 1; "Você entrou o valor 2." se for dada a entrada do valor numérico 2; "Você entrou um valor muito baixo" se for dada a entrada de um valor numérico menor que 1 ou "Você entrou um valor muito alto" se for dada a entrada de um valor numérico maior que 2.

```
;*****  
;* Programa: DECISA06.ASM *  
;*****  
  
INCLUDE 'emu8086.inc'  
  
.org 100h  
  
.DATA  
N DW 0  
msg1 DB 'Voce entrou o valor 1.', 0  
msg2 DB 'Voce entrou o valor 2.', 0  
msg3 DB 'Voce entrou um valor muito baixo.', 0  
msg4 DB 'Voce entrou um valor muito alto.', 0  
msg5 DB 'Entre um valor numerico: ', 0  
  
.CODE  
LEA SI, msg5  
CALL PRINT_STRING  
CALL SCAN_NUM  
MOV N, CX  
PUTC 13d  
PUTC 10d  
  
se1:  
    CMP N, 1d  
    JNE fim_se1  
entao1:  
    LEA SI, msg1  
    CALL PRINT_STRING  
fim_se1:  
  
se2:  
    CMP N, 2d  
    JNE fim_se2  
entao2:  
    LEA SI, msg2
```

```

    CALL PRINT_STRING
fim_se2:

se3:
    CMP N, 1d
    JGE fim_se3
entao3:
    LEA SI, msg3
    CALL PRINT_STRING
fim_se3:

se4:
    CMP N, 1d
    JLE fim_se4
entao4:
    LEA SI, msg4
    CALL PRINT_STRING
fim_se4:

INT      20h
DEFINE_PRINT_STRING
DEFINE_SCAN_NUM

```

END

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 **DECISAO6**, de forma que fique semelhante à imagem da Figura 12.6.

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

```

edit: C:\Users\Augusto Manzano\Documents\DECISAO6.asm
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about
01 ; *****
02 ; * Programa: DECISAO6.ASM *
03 ; *****
04
05 INCLUDE 'emu8086.inc'
06
07 org 100h
08
09 .DATA
10 N DW 0
11 msg1 DB 'Voce entrou o valor 1.', 0
12 msg2 DB 'Voce entrou o valor 2.', 0
13 msg3 DB 'Voce entrou um valor muito baixo.', 0
14 msg4 DB 'Voce entrou um valor muito alto.', 0
15 msg5 DB 'Entre um valor numerico: ', 0
16
17 .CODE
18 LEA SI, msg5
19 CALL PRINT_STRING
20 CALL SCAN_NUM
21 MOV N, CX
22 PUTC 13d
23 PUTC 10d
24
25 sel:
26 CMP N, 1d
27 JNE fim_sel
28 entao1:
29 LEA SI, msg1
30 CALL PRINT_STRING
31 fim_sel:
32
33 se2:
34 CMP N, 2d
35 JNE fim_se2
36 entao2:
37 LEA SI, msg2

```

Figura 12.6 - Programa DECISAO6 na ferramenta emu8086.

Observe no programa o uso da sequência de tomadas de decisão uma após a outra sinalizadas com os rótulos **se1...entao1...fim\_se1** até **se4...entao4...fim\_se4**. Quando um nome de rótulo é usado para uma ação este não pode ser usado para outra ação. Desta forma, torna-se necessário fazer uso de nomes diferentes.

## 12.1.5 - Decisão Seletiva

A tomada de decisão seletiva é uma alternativa ao uso de tomadas de decisão sequenciais. Essa estrutura lógica de condição é útil e pode ser usada em situações em que se possui um grande número de verificações lógicas a serem

realizadas. A desvantagem desta estrutura de decisão está no fato desta ser usada apenas para condições que usem o operador relacional de igualdade. Para a efetivação de uma ação de tomada de decisão seletiva considere a estrutura de código seguinte:

### Tomada de Decisão Seletiva em Português Estruturado

```
caso <variável>
  seja <opção 1> faça
    [ação para condição1 verdadeira]
  seja <opção 2> faça
    [ação para condição2 verdadeira]
  seja <opção 3> faça
    [ação para condição3 verdadeira]
senão
  [ação para nenhuma condição satisfeita]
fim_caso
```

### Tomada de Decisão Seletiva em Assembly

```
caso:
  seja01:
    CMP <condição1>
    JNE seja02
    JE faca01
  faca01:
    [ação para condição1 verdadeira]
    JMP fim_caso
  seja02:
    CMP <condição2>
    JNE seja03
    JE faca02
  faca02:
    [ação para condição2 verdadeira]
    JMP fim_caso
  seja03:
    CMP <condição3>
    JNE senao
    JE faca03
  faca03:
    [ação para condição3 verdadeira]
    JMP fim_caso
senao:
  [ação para nenhuma condição satisfeita]
fim_caso:
```

Desenvolver um programa de computador que leia um valor numérico entre os valores 1 e 12 e apresente por extenso o nome do mês correspondente ao valor entrado. Caso sejam fornecidos valores menores que 1 e maiores que 12, o programa deve apresentar a mensagem "Valor invalido".

```

;*****
;*  Programa: DECISA07.ASM  *
;*****

INCLUDE 'emu8086.inc'

org 100h

.DATA
N DW 0
msg00 DB 'Entre um valor numerico: ', 0
msg01 DB 'Janeiro', 0
msg02 DB 'Fevereiro', 0
msg03 DB 'Marco', 0
msg04 DB 'Abril', 0
msg05 DB 'Maio', 0
msg06 DB 'Junho', 0
msg07 DB 'Julho', 0
msg08 DB 'Agosto', 0
msg09 DB 'Setembro', 0
msg10 DB 'Outubro', 0
msg11 DB 'Novembro', 0
msg12 DB 'Dezembro', 0
msg13 DB 'Valor invalido', 0

.CODE
LEA SI, msg00
CALL PRINT_STRING
CALL SCAN_NUM
MOV N, CX
PUTC 13d
PUTC 10d

caso:
seja01:
CMP N, 01d
JNE seja02
JE faca01
faca01:
LEA SI, msg01
CALL PRINT_STRING
JMP fim_caso
seja02:
CMP N, 02d
JNE seja03
JE faca02
faca02:
LEA SI, msg02
CALL PRINT_STRING
JMP fim_caso
seja03:
CMP N, 03d
JNE seja04
JE faca03
faca03:
LEA SI, msg03
CALL PRINT_STRING
JMP fim_caso
seja04:
CMP N, 04d
JNE seja05
JE faca04
faca04:

```

```

LEA SI, msg04
CALL PRINT_STRING
JMP fim_caso
seja05:
CMP N, 05d
JNE seja06
JE faca05
faca05:
LEA SI, msg05
CALL PRINT_STRING
JMP fim_caso
seja06:
CMP N, 06d
JNE seja07
JE faca06
faca06:
LEA SI, msg06
CALL PRINT_STRING
JMP fim_caso
seja07:
CMP N, 07d
JNE seja08
JE faca07
faca07:
LEA SI, msg07
CALL PRINT_STRING
JMP fim_caso
seja08:
CMP N, 08d
JNE seja09
JE faca08
faca08:
LEA SI, msg08
CALL PRINT_STRING
JMP fim_caso
seja09:
CMP N, 09d
JNE seja10
JE faca09
faca09:
LEA SI, msg09
CALL PRINT_STRING
JMP fim_caso
seja10:
CMP N, 10d
JNE seja11
JE faca10
faca10:
LEA SI, msg10
CALL PRINT_STRING
JMP fim_caso
seja11:
CMP N, 11d
JNE seja12
JE faca11
faca11:
LEA SI, msg11
CALL PRINT_STRING
JMP fim_caso
seja12:
CMP N, 12d
JNE senao
JE faca12

```

```

faca12:
    LEA SI, msg12
    CALL PRINT_STRING
    JMP fim_caso
senao:
    LEA SI, msg13
    CALL PRINT_STRING
fim_caso:
    INT     20h
    DEFINE_PRINT_STRING
    DEFINE_SCAN_NUM
END

```

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 **DECISAO7**, de forma que fique semelhante à imagem da Figura 12.7.

The screenshot shows the emu8086 assembly editor window. The file path is C:\Users\Augusto Manzano\Documents\DECISAO7.asm. The menu bar includes file, edit, bookmarks, assembler, emulator, math, ascii codes, help, new, open, examples, save, compile, emulate, calculator, convertor, options, help, and about. The code area contains the following assembly code:

```

001 ; ****
002 ; * Programa: DECISAO7.ASM *
003 ; ****
004
005 INCLUDE 'emu8086.inc'
006
007 org 100h
008
009 .DATA
010 N DW 0
011 msg00 DB 'Entre um valor numerico: ', 0
012 msg01 DB 'Janeiro', 0
013 msg02 DB 'Fevereiro', 0
014 msg03 DB 'Marco', 0
015 msg04 DB 'Abril', 0
016 msg05 DB 'Maio', 0
017 msg06 DB 'Junho', 0
018 msg07 DB 'Julho', 0
019 msg08 DB 'Agosto', 0
020 msg09 DB 'Setembro', 0
021 msg10 DB 'Outubro', 0
022 msg11 DB 'Novembro', 0
023 msg12 DB 'Dezembro', 0
024 msg13 DB 'Valor invalido', 0
025
026 .CODE
027 LEA    SI, msg00
028 CALL   PRINT_STRING
029 CALL   SCAN_NUM
030 MOV    N, CX
031 PUTC  13d
032 PUTC  10d
033
034 caso:
035 seja01:
036 CMP   N, 01d
037 JNE   seja02

```

Figura 12.7 - Programa DECISAO7 na ferramenta emu8086.

Observe no programa que para executar a ação de uma estrutura **caso...seja...faça...fim\_caso** usa-se uma sequência de operações de comparação **CMP** com auxílio das instruções de salto **JNE**, **JE** e **JMP**.

## 12.1.6 - Decisão Encadeada

A tomada de decisão encadeada ocorre quando se utilizam tomadas de decisão simples ou compostas uma dentro de outra, onde para a tomada de decisão mais interna ser efetivada depende da verificação da tomada de decisão mais externa. Para uso de uma ação de tomada de decisão encadeada considere a estrutura de código seguinte:

## Tomada de Decisão Encadeada em Português Estruturado

```
se (<condição1>) então
  se (<condição2>) então
    [ação para condição 1 e condição 2 verdadeiras]
  senão
    [ação para condição 1 verdadeira e condição 2 falsa]
  fim_se
senão
  [ação para condição 1 falsa]
fim_se
```

## Tomada de Decisão Encadeada em Assembly

```
se1:
  CMP <condição1>
  <instrução condicional de salto> senao1
entao1:
se2:
  CMP <condição2>
  <instrução condicional de salto> senao2
entao2:
  [instruções executadas após condição ser verdadeira]
  JMP fim_se2
senao2:
  [ação para condição 1 verdadeira e condição 2 falsa]
fim_se2:
  JMP fim_se1
senao1:
  [ação para condição 1 falsa]
fim_se1:
```

Para fazer uso deste recurso considere um programa que informe ao usuário se certo valor fornecido está abaixo de 10, entre 10 e 50 ou acima de 50.

```
;*****
;*  Programa: DECISA08.ASM  *
;*****
```

```
INCLUDE 'emu8086.inc'

org 100h

.DATA
N DW 0
msg1 DB 'Entre um valor: ', 0
msg2 DB 'Valor entre 10 e 50.', 0
msg3 DB 'Valor acima de 50.', 0
msg4 DB 'Valor abaixo de 10.', 0

.CODE
LEA SI, msg1
CALL PRINT_STRING
CALL SCAN_NUM
MOV N, CX
PUTC 13d
PUTC 10d
```

```

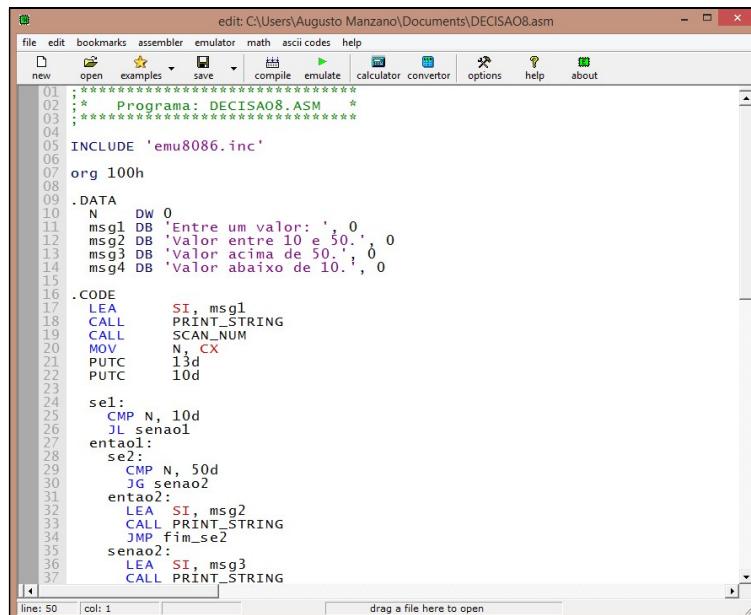
se1:
    CMP N, 10d
    JL senao1
entao1:
    se2:
        CMP N, 50d
        JG senao2
entao2:
    LEA SI, msg2
    CALL PRINT_STRING
    JMP fim_se2
senao2:
    LEA SI, msg3
    CALL PRINT_STRING
fim_se2:
    JMP fim_se1
senao1:
    LEA SI, msg4
    CALL PRINT_STRING
fim_se1:

INT      20h
DEFINE_PRINT_STRING
DEFINE_SCAN_NUM

```

END

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 **DECISAO8**, de forma que fique semelhante à imagem da Figura 12.8.



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

```

edit:C:\Users\Augusto Manzano\Documents\DECISAO8.asm
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator converter options help about
01 ; ****
02 ; * Programa: DECISAO8.ASM *
03 ;
04 ;
05 INCLUDE 'emu8086.inc'
06
07 org 100h
08
09 .DATA
10 N DW 0
11 msg1 DB 'Entre um valor: ', 0
12 msg2 DB 'Valor entre 10 e 50.', 0
13 msg3 DB 'Valor acima de 50.', 0
14 msg4 DB 'Valor abaixo de 10.', 0
15
16 .CODE
17 LEA SI, msg1
18 CALL PRINT_STRING
19 CALL SCAN_NUM
20 MOV N, CX
21 PUTC 13d
22 PUTC 10d
23
24 se1:
25     CMP N, 10d
26     JL senao1
27 entao1:
28     se2:
29         CMP N, 50d
30         JG senao2
31 entao2:
32     LEA SI, msg2
33     CALL PRINT_STRING
34     JMP fim_se2
35 senao2:
36     LEA SI, msg3
37     CALL PRINT_STRING

```

The code implements a decision-making process based on user input. It prints four messages: 'Entre um valor:', 'Valor entre 10 e 50.', 'Valor acima de 50.', and 'Valor abaixo de 10.'. It then compares the input value (N) with 10 and 50 using CMP and JG/JL instructions to determine which message to print next.

Figura 12.8 - Programa DECISAO8 na ferramenta emu8086.

## 12.2 - Laços de Repetição

A linguagem de programação Assembly não possui de forma explícita, como ocorre nas linguagens de alto nível, instruções de ação direta para a efetivação de laços. Os laços de repetição são uma estrutura de programação que repete determinado trecho de código certo número de vezes, podendo-se classificá-los como *laços de repetição interativa* ou *laços de repetição iterativa*. São *interativos* quando ocorre a intervenção de um usuário do programa para repetir a próxima ação, são laços *iterativos* quando executam as repetições previstas de forma automática um número de vezes conhecido. Os exemplos de laços apresentados a seguir são laços do tipo iterativo, quanto aos laços interativos ficam a cargo do leitor implementá-los.

As estruturas de laços de repetição escritas em linguagem de programação Assembly são mais simples de serem implementadas do que são as estruturas de tomadas de decisão. Assim sendo, os exemplos de uso das formas de laços existentes a seguir partem de um programa exemplo que apresenta no monitor de vídeo a mensagem “**Linguagem Assembly**” por 5 vezes, iniciando a contagem da variável I de 1 até 5 com incremento 1. Serão apresentados três laços, sendo: *controle condicional pré-teste*, *controle condicional pós-teste* e *controle condicional seletivo*.

### 12.2.1 - Laço Condisional Pré-Teste

O laço de repetição condicional pré-teste executa as instruções subordinadas de um bloco adjacente no período em que o resultado lógico da condição permanece verdadeiro. No momento em que o resultado lógico da condição se tornar falso, a execução do laço é automaticamente encerrada. Para uso de uma ação de laço condicional pré-teste verdadeiro considere a estrutura de código seguinte:

#### Laço Condisional Pré-teste em Português Estruturado

```
I = <valor inicial>
enquanto (I <= <valor final>) faça
    [ação executada enquanto condição é verdadeira]
    I = I + 1
fim_enquanto
```

#### Laço Condisional Pré-teste em Assembly

```
MOV I, <valor inicial>
enquanto:
    CMP I, <valor final>
    JG fim_enquanto
faca:
    [ação executada enquanto condição é verdadeira]
    INC I
    JMP enquanto:
fim_enquanto
```

Para fazer uso deste tipo de laço de repetição considere o programa seguinte:

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

```
INCLUDE 'emu8086.inc'

org 100h
```

```
.DATA
    I    DW 0
    msg1 DB 'Linguagem Assembly', 0
```

.CODE

```
MOV I, 1d
enquanto:
    CMP I, 5d
    JG fim_enquanto
faca:
    LEA SI, msg1
    CALL PRINT_STRING
    PUTC 13d
    PUTC 10d

    INC I
    JMP enquanto
fim_enquanto:

INT 20h
DEFINE_PRINT_STRING
```

END

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 **LACO3**, de forma que fique semelhante à imagem da Figura 12.9.

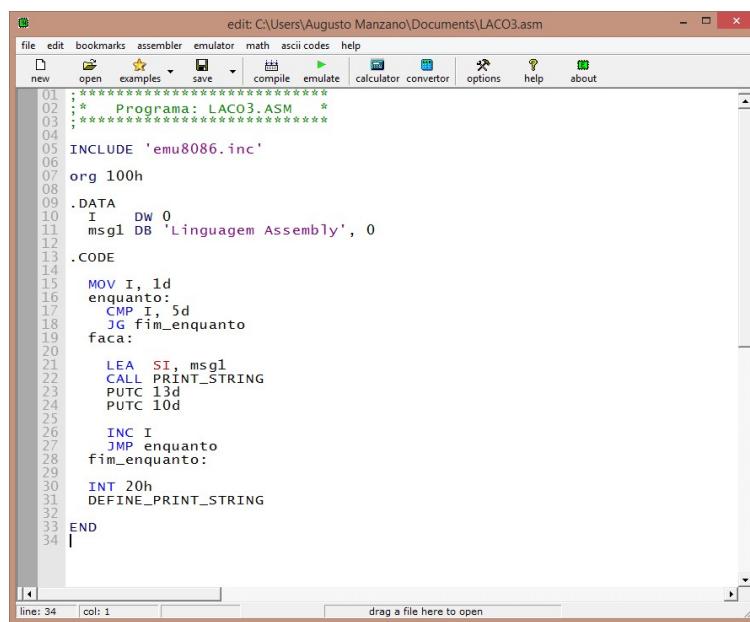


Figura 12.9 - Programa LACO3 na ferramenta emu8086.

A partir da estrutura de laço condicional pré-teste considere o desenvolvimento de um programa que apresente valores de **0** (zero) a **9** (nove) sendo um valor por linha. Assim sendo, observe o código seguinte.

```
;*****
;*  Programa: APLICASM1.ASM   *
;*****
```

```
org 100h
```

```

.DATA
I DW 0

.CODE

MOV DL, '0' ; 30h
MOV I, 1d
enquanto:
    CMP I, 10d
    JG fim_enquanto
faca:
    MOV AH, 02h
    INT 21h
    PUSH AX
    MOV AL, 10d
    MOV AH, 0Eh
    INT 10h
    POP AX
    PUSH AX
    MOV AL, 13d
    MOV AH, 0Eh
    INT 10h
    POP AX
    INC DL
    INC I
    JMP enquanto
fim_enquanto:

```

```
INT 20h
```

```
END
```

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 **APLICASM1**, de forma que fique semelhante à imagem da Figura 12.10.

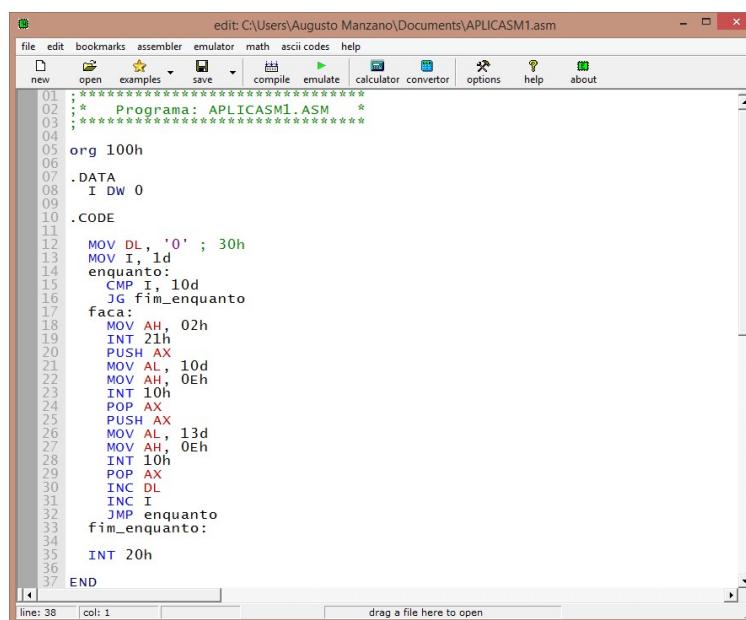


Figura 12.10 - Programa APLICASM1 na ferramenta emu8086.

O programa anterior não está fazendo usos dos recursos existentes na biblioteca **emu8086.ini** por esta razão os efeitos usados anteriormente pela macro **PUTC** para movimentar o cursor para a próxima linha do monitor de vídeo com os

códigos **10d** e **13d** estão sendo descritos entre as linhas **20** e **24** para execução do código **10d** e entre as linhas **25** e **29** para execução do código **13d**. Os códigos **10d** e **13d** podem ser usados na ordem **13d** e **10d** e efetuam o retorno de carro, ou seja, o cursor volta para a posição inicial da linha em uso quando se usa o código **13d** e a movimentação do cursor para a próxima linha. Quando se faz uso da tecla <Enter> esta tecla gera internamente a execução dos códigos **10d** e **13d**.

A linha de código **12 (MOV DL, '0' ; 30h)** carrega para a parte baixa do registrador **DX** o caractere **0** (zero) entre aspas simples, o qual pode ser substituído pelo valor **30h** (representação do valor zero na tabela ASCII) e que será impresso pela execução das linhas **18 (MOV AH, 02h)** e **19 (INT 21h)**. Lembre-se de que o valor **02h** carregado na parte alta do registrador **AX** é usado para efetuar a apresentação do caractere armazenado na parte baixa do registrador **DX** por meio da instrução da linha **19**.

As linhas de código **13, 14-17** e **31-33** são responsáveis pela execução de dez passos do laço condicional pré-teste como orientado. Atenção maior é em relação a linha de código **30 (INC DL)** dentro do laço que efetua a soma de mais **1** sobre o valor armazenado na parte baixa do registrador **DX** e assim pega o próximo caractere numérico.

## 12.2.2 - Laço Condicional Pós-Teste

O laço de repetição condicional pós-teste executa no mínimo uma vez as instruções subordinadas de um bloco adjacente e mantém a execução no período em que o resultado lógico da condição permanece falso. No momento em que o resultado lógico da condição se torna verdadeiro, a execução do laço é automaticamente encerrada. Para uso de uma ação de laço condicional pré-teste verdadeiro considere a estrutura de código seguinte:

### Laço Condicional Pós-teste em Português Estruturado

```
I = <valor inicial>
repita:
    [ação executada até que a condição torne-se verdadeira]
    I = I + 1
até_que (I > <valor final>)
```

### Laço Condicional Pós-teste em Assembly

```
MOV I, <valor inicial>
repita:
    [ação executada até que a condição torne-se verdadeira]
    INC I
ate_que:
    CMP I, <valor final>
    JLE repita
```

Para fazer uso deste tipo de repetição considere o programa seguinte:

```
;*****
;*  Programa: LAC04.ASM  *
;*****
```

```
INCLUDE 'emu8086.inc'

org 100h
```

```
.DATA
I DW 0
msg1 DB 'Linguagem Assembly', 0
```

```
.CODE
```

```
MOV I, 1d
repita:
    LEA SI, msg1
    CALL PRINT_STRING
    PUTC 13d
    PUTC 10d
```

```
    INC I
ate_que:
    CMP I, 5d
    JLE repita
```

```
INT 20h
DEFINE_PRINT_STRING
```

```
END
```

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**, de forma que fique semelhante à imagem da Figura 12.11.

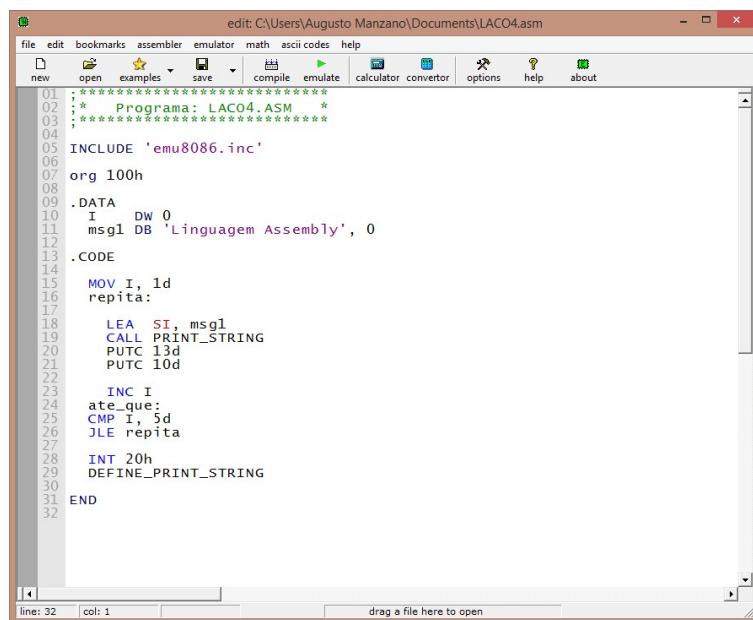


Figura 12.11 - Programa LACO4 na ferramenta emu8086.

A partir da estrutura de laço condicional pós-teste considere o desenvolvimento de um programa que apresente numa linha as letras do alfabeto em formato maiúsculo e numa segunda linha as letras do alfabeto em formato minúsculo. Assim sendo, observe o código seguinte.

```

;*****
;*  Programa: APPLICASM2.ASM   *
;*****



INCLUDE 'emu8086.inc'

org 100h

.DATA
I DW 0

.CODE

MOV DL, 'A' ; 41h
MOV I, 1d
repita1:
MOV AH, 02h
INT 21h
INC DL
INC I
ate_que1:
CMP I, 26d
JLE repita1

PUTC 13d
PUTC 10d

MOV DL, 'a' ; 61h
MOV I, 1d
repita2:
MOV AH, 02h
INT 21h
INC DL
INC I
ate_que2:
CMP I, 26d
JLE repita2

INT 20h

END

```

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 **APPLICASM2**, de forma que fique semelhante à imagem da Figura 12.12.

Este programa é de certa forma semelhante ao programa anterior, exceto a estrutura de laço de repetição em uso. O trecho situado entre as linhas **14** e **23** apresenta o alfabeto em caracteres maiúsculos e o trecho de linhas entre **28** e **37** apresenta o alfabeto em caracteres minúsculos.

The screenshot shows the emu8086 software window with the assembly code for APPLICASM2.asm. The code includes directives like INCLUDE, ORG, and various assembly instructions (MOV, INT, INC) to print the character 'a' 26 times. The software has a menu bar with File, Edit, Bookmarks, Assembler, Emulator, Math, ASCII Codes, Help, and a toolbar with New, Open, Save, Compile, Emulate, Calculator, Converter, Options, Help, and About.

```

01; *****
02; * Programa: APPLICASM2.ASM *
03; *****
04
05INCLUDE 'emu8086.inc'
06
07org 100h
08
09.DATA
10    I DW 0
11
12.CODE
13
14    MOV DL, 'A' ; 41h
15    MOV I, 1d
16    repital:
17        MOV AH, 02h
18        INT 21h
19        INC DL
20        INC I
21    ate_quel:
22        CMP I, 26d
23        JLE repital
24
25    PUTC 13d
26    PUTC 10d
27
28    MOV DL, 'a' ; 61h
29    MOV I, 1d
30    repita2:
31        MOV AH, 02h
32        INT 21h
33        INC DL
34        INC I
35    ate_quer2:
36        CMP I, 26d
37        JLE repita2

```

Figura 12.12 - Programa APPLICASM2 na ferramenta emu8086.

## 12.2.3 - Laço Incondicional

O laço de repetição incondicional executa um bloco adjacente certo número de vezes definido. Para uso de uma ação de laço condicional pré-teste verdadeiro considere a estrutura de código seguinte:

### Laço Incondicional em Português Estruturado

```

para I de [<inicio>] até [<fim>] passo 1 faça
    [<instruções executadas durante o ciclo de contagem da variável de controle>]
fim_para

```

### Laço Incondicional em Assembly

```

MOV I, <valor inicial>
para:
    CMP I, <valor final>
    JG fim_para
faca:
    [ação executada até que a condição torne-se verdadeira]
    INC I
    JMP para
fim_para:

```

Para fazer uso deste tipo de laço de repetição considere o programa seguinte:

```

; *****
;* Programa: LAC05.ASM *
;*****

```

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

```
org 100h
```

```

.DATA
    I     DW 0
    msg1 DB 'Linguagem Assembly', 0

.CODE

MOV I, 1d
para:
    CMP I, 5d
    JG fim_para
faca:

    LEA SI, msg1
    CALL PRINT_STRING
    PUTC 13d
    PUTC 10d

    INC I
    JMP para
fim_para:

INT 20h
DEFINE_PRINT_STRING

```

END

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 **LACO5**, de forma que fique semelhante à imagem da Figura 12.13.

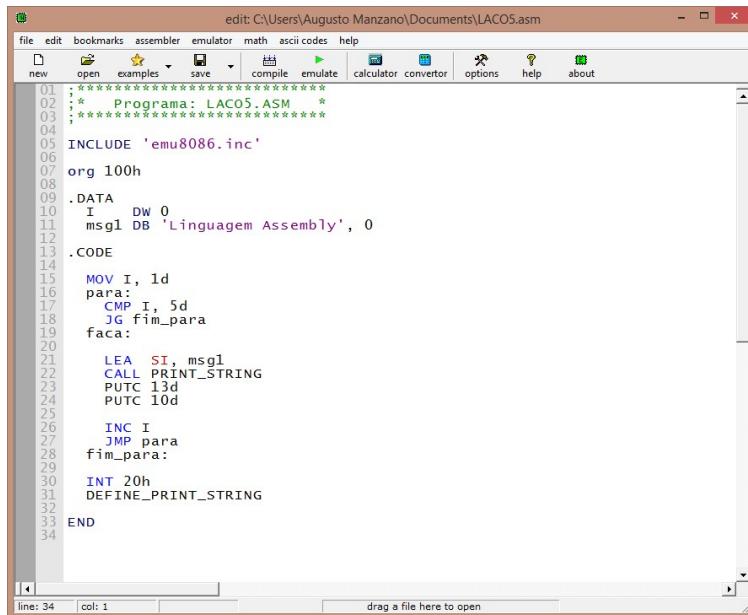


Figura 12.13 - Programa LACO5 na ferramenta emu8086.

A partir da estrutura de laço incondicional considere o desenvolvimento de um programa que apresente linearmente todos os caracteres da tabela ASCII situados na faixa de valores de **0Eh** até **7Fh**. Assim sendo, observe o código seguinte.

```

;*****
;*  Programa: APPLICASM3.ASM   *
;*****

```

```

.org 100h

.DATA
I DW 0

.CODE

MOV DL, 0Eh
MOV I, 1d
para:
CMP I, 114d
JG fim_para
faca:
MOV AH, 02h
INT 21h
INC DL
INC I
JMP para
fim_para:
INT 20h

```

END

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 **APLICASM3**, de forma que fique semelhante à imagem da Figura 12.14.

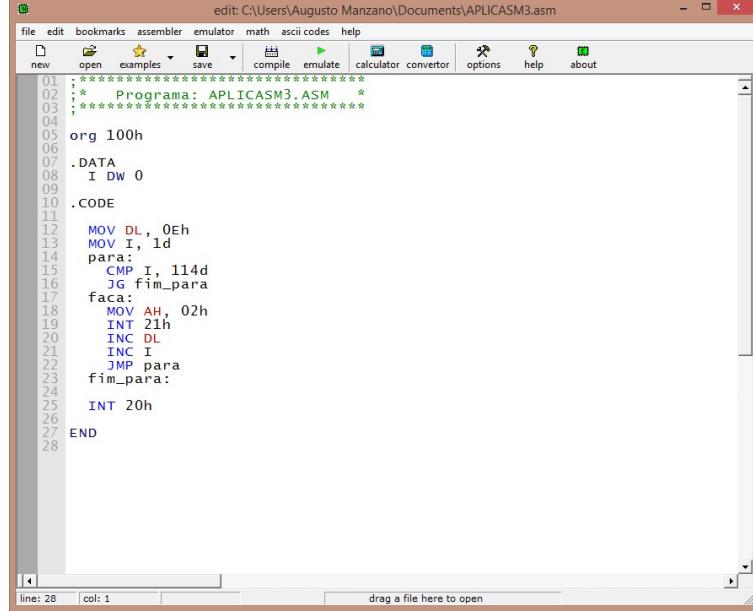


Figura 12.14 - Programa APLICASM3 na ferramenta emu8086.

Observe que o programa inicia a apresentação dos caracteres ASCII a partir do código **0Eh** e estende por 114 caracteres a apresentação dos caracteres da tabela.

## 12.3 - Demonstrações Assembly

Este tópico tem por objetivo a partir do exposto neste e nos demais capítulos desta obra apresentar como demonstração alguns exemplos de programas escritos em linguagem de baixo nível Assembly 8086/8088. Os exemplos apresentados baseiam-se numa situação problema simples. Deguste mentalmente cada uma das situações expostas e observe

os detalhes apresentados. As explicações principais sobre a ação de cada programa estão definidas dentro do próprio código como linhas de comentários precedidas do caractere ponto e vírgula.

Como sugestão complementar de estudo abra o arquivo de biblioteca **emu8086.inc** existente no diretório (pasta) de trabalho **C:\emu\binaries\inc** e observe os procedimentos definidos para o tratamento das operações de entrada e saída. Estude também os programas apresentados no diretório **C:\emu\binaries\examples**, além do modo de ajuda do programa. São ricas fontes de informação existente dentro da própria ferramenta de trabalho.

## ORDENAÇÃO DE VALORES

*Desenvolver um programa de computador que efetue a leitura de três valores numéricos e apresente estes valores dispostos em ordem crescente.*

```
;*****
;*  Programa: ORDENA.ASM  *
;*****  
  
INCLUDE 'emu8086.inc'  
  
.org 100h  
  
.DATA  
  
A DW ? ; Definição de variáveis para entrada de dados  
B DW ?  
C DW ?  
  
msg1 DB      'Entre o 1o. valor: ', '$'  
msg2 DB 0Dh, 0Ah, 'Entre o 2o. valor: ', '$'  
msg3 DB 0Dh, 0Ah, 'Entre o 3o. valor: ', '$'  
  
msg4 DB 0Dh, 0Ah, 0Dh, 0Ah, 'Valor 1: ', '$'  
msg5 DB 0Dh, 0Ah,           'Valor 2: ', '$'  
msg6 DB 0Dh, 0Ah,           'Valor 3: ', '$'  
  
.CODE  
  
; ////////////  
; // Entrada de dados //  
; ////////////  
  
; Efetua a entrada do 1o. valor (Bloco de ação 1)  
  
LEA DX, msg1 ; Pega a mensagem da variável msg1  
MOV AH, 09h ; Põe 09h na parte baixa de AX  
INT 21h ; Imprime msg1 de DX por meio do código 09h  
CALL SCAN_NUM ; Efetua a leitura do valor e põe em CX  
MOV A, CX ; Transfere o valor de CX para a variável A  
  
; Efetua a entrada do 2o. valor (Bloco de ação 2)  
  
LEA DX, msg2 ; Pega a mensagem da variável msg2  
MOV AH, 09h ; Ações similares ao bloco 1  
INT 21h ;  
CALL SCAN_NUM ;  
MOV B, CX ; Transfere o valor de CX para a variável B  
  
; Efetua a entrada do 3o. valor (Bloco de ação 3)  
  
LEA DX, msg3 ; Pega a mensagem da variável msg3  
MOV AH, 09h ; Ações similares ao bloco 1  
INT 21h ;
```

```

CALL SCAN_NUM ;  

MOV C, CX ; Transfere o valor de CX para a variável C  

;  

; ////////////////  

; // Ordenação crescente dos valores //  

; ////////////////  

;  

; se (A > B) então  

; troca os valores de A com B e de B com A  

;  

MOV BX, A ; Armazena valor da variável A em BX  

MOV CX, B ; Armazena valor da variável B em CX  

se1:  

    ;  

    CMP BX, CX ; compara BX e CX  

    JLE fim_se1 ; se BX > CX segue após então1  

entao1: ; e efetua a troca de valores  

    XCHG BX, CX ; entre os registradores BX e CX  

fim_se1:  

    ;  

MOV A, BX ; Armazena valor do registrador BX em A  

MOV B, CX ; Armazena valor do registrador CX em B  

;  

; se (A > C) então  

; troca os valores de A com C e de C com A  

;  

MOV BX, A ; Armazena valor da variável A em BX  

MOV CX, C ; Armazena valor da variável C em CX  

se2:  

    ;  

    CMP BX, CX ; compara BX e CX  

    JLE fim_se2 ; se BX > CX segue após então2  

entao2: ; e efetua a troca de valores  

    XCHG BX, CX ; entre os registradores BX e CX  

fim_se2:  

    ;  

MOV A, BX ; Armazena valor do registrador BX em A  

MOV C, CX ; Armazena valor do registrador CX em C  

;  

; se (B > C) então  

; troca os valores de B com C e de C com B  

;  

MOV BX, B ; Armazena valor da variável B em BX  

MOV CX, C ; Armazena valor da variável C em CX  

se3:  

    ;  

    CMP BX, CX ; compara BX e CX  

    JLE fim_se3 ; se BX > CX segue após então3  

entao3: ; e efetua a troca de valores  

    XCHG BX, CX ; entre os registradores BX e CX  

fim_se3:  

    ;  

MOV B, BX ; Armazena valor do registrador BX em B  

MOV C, CX ; Armazena valor do registrador CX em C  

;  

; ////////////////  

; // Saída de dados //  

; ////////////////  

;  

LEA DX, msg4 ; Pega a mensagem da variável msg3  

PUSH AX  

MOV AH, 09h  

INT 21h  

POP AX  

MOV AX, A ; Transfere para AX da variável A  

CALL print_num ; Escreve o conteúdo de AX.  

;  

LEA DX, msg5 ; Pega a mensagem da variável msg5  

PUSH AX

```

```

MOV AH, 09h
INT 21h
POP AX
MOV AX, B      ; Transfere para AX da variável B
CALL print_num ; Escreve o conteúdo de AX.

LEA DX, msg6   ; Pega a mensagem da variável msg6
PUSH AX
MOV AH, 09h
INT 21h
POP AX
MOV AX, C      ; Transfere para AX da variável C
CALL print_num ; Escreve o conteúdo de AX.

INT 20h

DEFINE_SCAN_NUM
DEFINE_PRINT_NUM_UNS
DEFINE_PRINT_NUM

```

END

### CHECA TRIÂNGULO

*Elaborar um programa que leia três valores numéricos para os lados de um triângulo, considerando lados como A, B e C. Verificar se os lados fornecidos formam um triângulo, e se for esta condição verdadeira, deve ser indicado o tipo de triângulo formado: isósceles, escaleno ou equilátero. Leve em consideração que triângulo é uma forma geométrica (polígono) composta de três lados, e o valor de cada lado deve ser menor que a soma dos outros dois lados. Assim sendo, é um triângulo quando  $A < B + C$ , quando  $B < A + C$  e quando  $C < A + B$ , considerando como lados as variáveis A, B e C. Tendo certeza de que os valores informados para os três lados formam um triângulo, deve-se então analisar os valores fornecidos para estabelecer o tipo de triângulo que será formado: isósceles, escaleno ou equilátero. Um triângulo é isósceles quando possui dois lados iguais e um diferente, sendo  $A=B$  ou  $A=C$  ou  $B=C$ ; é escaleno quando possui todos os lados diferentes, sendo  $A > B$  e  $B > C$  e é equilátero quando possui todos os lados iguais, sendo  $A=B$  e  $B=C$ .*

```

;*****
;*  Programa: TRIANGULO.ASM  *
;*****

INCLUDE 'emu8086.inc'

org 100h

.DATA

A DW ? ; Definição de variáveis para entrada de dados
B DW ?
C DW ?

msg1 DB      'Entre o valor do lado [A]: ', '$' ; 0Dh e 0Ah
msg2 DB 0Ah, 0Ah, 'Entre o valor do lado [B]: ', '$' ; são usados
msg3 DB 0Ah, 0Ah, 'Entre o valor do lado [C]: ', '$' ; para pular
                                ; linhas
msg4 DB 0Ah, 0Ah, 0Ah, 0Ah, 'Medidas nao formam um triangulo.', 0
msg5 DB 0Ah, 0Ah, 0Ah, 0Ah, 'Triangulo isosceles.', 0
msg6 DB 0Ah, 0Ah, 0Ah, 0Ah, 'Triangulo equilatero.', 0
msg7 DB 0Ah, 0Ah, 0Ah, 0Ah, 'Triangulo escaleno.', 0

.CODE

```

```

; /////////////////////////
; // Entrada de dados //
; /////////////////////////

; Efetua a entrada do 1o. valor

LEA DX, msg1 ; Pega a mensagem da variável msg1
MOV AH, 09h ; Põe 09h na parte baixa de AX
INT 21h ; Imprime msg1 de DX por meio do código 09h
CALL SCAN_NUM ; Efetua a leitura do valor e põe em CX
MOV A, CX ; Transfere o valor de CX para a variável A

; Efetua a entrada do 2o. valor

LEA DX, msg2 ; Pega a mensagem da variável msg2
MOV AH, 09h
INT 21h
CALL SCAN_NUM
MOV B, CX ; Transfere o valor de CX para a variável B

; Efetua a entrada do 3o. valor

LEA DX, msg3 ; Pega a mensagem da variável msg3
MOV AH, 09h
INT 21h
CALL SCAN_NUM
MOV C, CX ; Transfere o valor de CX para a variável C

; /////////////////////////
; // Verificação se medidas formam um triângulo //
; // e efetivação da apresentação das mensagens //
; // de saída //
; /////////////////////////

```

**se1:**

```

MOV AX, A ;
MOV AX, B ;
MOV BX, C ;
ADD BX, AX ; A < B + C
CMP AX, BX ;
JNL senao1 ;

```

**e1\_1:**

```

MOV AX, B ;
MOV AX, A ;
MOV BX, C ;
ADD BX, AX ; B < A + C
CMP AX, BX ;
JNL senao1 ;

```

**e1\_2:**

```

MOV AX, C ;
MOV AX, A ;
MOV BX, B ;
ADD BX, AX ; C < A + B
CMP AX, BX ;
JNL senao1 ;

```

**entao1:**

**se2:**

```

MOV AX, A ;
MOV BX, B ;
CMP AX, BX ; A <> B
JNE senao2 ;

```

**e2\_1:**

```

MOV AX, B ;

```

```

MOV BX, C ;
CMP AX, BX ; B <> C
JNE senao2 ;
entao2:
    LEA SI, msg6 ; Triangulo equilatero.
    CALL PRINT_STRING
    JMP fim_se2
senao2:
    se3:
        MOV AX, A ;
        MOV BX, B ;
        CMP AX, BX ; A = B
        JE senao3 ;
    ou3_1:
        MOV AX, A ;
        MOV BX, C ;
        CMP AX, BX ; A = C
        JE senao3 ;
    ou3_2:
        MOV AX, B ;
        MOV BX, C ;
        CMP AX, BX ; C = B
        JE senao3 ;
    entao3:
        LEA SI, msg7 ; Triangulo escaleno.
        CALL PRINT_STRING
        JMP fim_se3
    senao3:
        LEA SI, msg5 ; Triangulo isosceles.
        CALL PRINT_STRING
    fim_se3:
    fim_se2:
    JMP fim_se1
senao1:
    LEA SI, msg4 ; Medidas nao formam um triangulo.
    CALL PRINT_STRING
fim_se1:
    INT 20h

    DEFINE_SCAN_NUM
    DEFINE_PRINT_STRING

END

```

### TABUADA DE UM NÚMERO QUALQUER

*Elaborar um programa que leia um valor inteiro qualquer entre 1 e 10 e apresente como resultado sua tabuada desse número. Caso o usuário informe um valor fora da faixa de uso do programa o programa deve apresentar como resultado a mensagem de advertência “Entre valores entre 1 e 10.”.*

```

;*****
;*  Programa: TABUADA.ASM  *
;*****

INCLUDE 'emu8086.inc'

org 100h

.DATA

N DW ? ; Definição de variáveis para entrada de dados
I DW 0
R DW 0

msg1 DB 'Entre o valor de tabuada (1-10): ', 24h
msg2 DB ' X ', 0
msg3 DB ' = ', 0
msg4 DB 'Entre valores entre 1 e 10.', 0
espaco DB ' ', 0

.CODE

; /////////////////
; // Entrada de dados //
; /////////////////

LEA DX, msg1
MOV AH, 09h
INT 21h
CALL SCAN_NUM
MOV N, CX
PUTC 13d
PUTC 10d

; /////////////////
; // Processamento e apresentação da tabuada //
; /////////////////

PUTC 13d
PUTC 10d
se1:
    CMP N, 1d
    JL senao1
e1:
    CMP N, 10d
    JG senao1
entao1:
    MOV I, 1d
    para:
        CMP I, 10d
        JG fim_para
    faca:

    MOV AX, N ;
    MOV BX, I ; Efetua o cálculo da
    MUL BX ; tabuada ==> R = N * I
    MOV R, AX ;

    se2:           ;
        CMP N, 10d ; Coloca um espaço em branco
        JGE fim_se2 ; antes do valor da variável N caso a
    entao2:         ; unidade para tabular o valor
        LEA SI, espaco ; com uma dezena seja < 10.

```

```

        CALL PRINT_STRING ; Em C = "%2d" e PASCAL = N:2.
fim_se2:          ;

MOV AX, N
CALL PRINT_NUM      ; Escreve o valor N

LEA SI, msg2
CALL PRINT_STRING   ; Escreve o simbolo X

se3:              ;
    CMP I, 10d      ; Coloca um espaço em branco
    JGE fim_se3     ; antes do valor da variável I caso a
entao3:           ; unidade para tabular o valor
    LEA SI, espaco  ; com uma dezena seja < 10.
    CALL PRINT_STRING ; Em C = "%2d" e PASCAL = N:2.
fim_se3:          ;

MOV AX, I
CALL PRINT_NUM      ; Escreve o valor I

LEA SI, msg3
CALL PRINT_STRING   ; Escreve o simbolo =

se4:              ;
    CMP R, 100d     ; Coloca um espaço em branco
    JGE fim_se4     ; antes do valor da variável R caso o
se5:              ; valor a ser apresentado seja < que
    CMP R, 10d      ; uma centena.
    JGE fim_se5     ; Coloca mais um espaço em branco antes
    LEA SI, espaco  ; do valor da variável R caso o valor.
    CALL PRINT_STRING ; a ser apresentado seja < que uma dezena.
fim_se5:          ;
entao4:           ;
    LEA SI, espaco  ;
    CALL PRINT_STRING ; Em C = "%3d" e PASCAL = N:3.
fim_se4:          ;

MOV AX, R
CALL PRINT_NUM      ; Escreve o valor R

PUTC 13d
PUTC 10d
INC I
JMP para

fim_para:
JMP fim_se1
senao1:
MOV AH, 02h
MOV DL, 07h ; toca bip
int 21h
LEA SI, msg4
CALL PRINT_STRING
fim_se1:

INT 20h

DEFINE_SCAN_NUM
DEFINE_PRINT_STRING
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS

END

```

## 12.4 - Linguagem Assembly versus Código de Máquina

É muito comum escutar as pessoas dizerem que programar em linguagem Assembly é difícil, que quem programa em linguagem Assembly é louco, que programação em linguagem Assembly é isso ou aquilo. Enfim muito é dito e pouco é falado.

Após o discorrer sobre os exemplos apresentados nesta obra, o próprio leitor poderá responder as essas perguntas com muita facilidade, pois programar em linguagem Assembly não é difícil e não tão pouco para loucos. Programar em linguagem Assembly é um pouco diferente de programar em linguagens de alto nível. Mas o fato de ser diferente, de aplicar os preceitos de lógica de programação com outra ótica não faz em absoluto desta linguagem ser mais difícil ou mais fácil que programar em linguagens como: C, PASCAL, BASIC, COBOL, entre tantas outras.

Qual o motivo que muitas pessoas confundem e afirmam ser a linguagem Assembly mais difícil de aprender? A ideia errônea vem do fato de confundir a linguagem Assembly com a linguagem de código de máquina. Perceba que são duas as linguagens de programação em baixo nível: a linguagem em código de máquina e a linguagem Assembly. A linguagem de máquina (código de máquina) que foi levemente apresentada nos primeiros capítulos deste livro é baseada em instruções e dados escritos em valores hexadecimais que são armazenados nos endereços de memória que também são definidos em valores hexadecimais. Assim sendo, pode-se até dizer que está linguagem de comunicação com a máquina seja de difícil aprendizagem. Mas tudo é uma questão de ponto de vista, pois talvez seria mais difícil escrever um programa em código de máquina utilizando-se valores binários. Note que cada valor hexadecimal equivale a quatro valores binários. Escrever um programa diretamente em código de máquina baseado em valores hexadecimais é mais rápido e mais seguro que escrever o mesmo programa com códigos binários, pois a simples troca de um valor 1 (um) com um valor 0 (zero) muda tudo. Assim sendo, se for perguntado para uma pessoa que programa equipamentos computacionais em linguagem de máquina baseada em valores binários, se essa tarefa é difícil com certeza ele dirá que não, pois tudo é uma questão de costume e de um ponto de vista mais leigo. Na medida em que se avança em certo conhecimento, aquilo que parecia ser difícil torna-se fácil. Isto chamasse evolução técnica e intelectual.

Melhor que falar é demonstrar. Tome por base o programa mais simples que se pode escrever em qualquer linguagem de programação, ou seja, o programa: "alô mundo". Assim sendo, observe o código de programa definido a partir do endereço de memória 0100 para as versões escritas nas linguagens: assembly, máquina em hexadecimal e máquina em binário.

org 100h		
MOV AH, 09h	0100: B4 09	0000000100000000: 10110100 00001001
LEA DX, msg	0102: BA 09 01	0000000100000010: 10111010 00001001 00000001
INT 21h	0105: CD 21	0000000100000101: 11001101 00100001
INT 20h	0107: CD 20	0000000100000111: 11001101 00100000
msg DB 'Alo mundo.', 24h	0109: 41 6C 6F 20 6D 75 6E 64 6F 2E 24	0000000100001001: 01000001 01101100 01101111 00100000 01101101 01110101 01101110 01100100 01101111 00101110 00100100
RET	0114: C3	0000000100010100: 11000011

Agora tenha em mente o seguinte, tudo que é fácil resolva hoje, o que é difícil deixe para ser resolvido amanhã, pois amanhã será fácil devido a experiência acumulada do hoje. Deixe para depois de amanhã tudo aquilo que hoje seja considerado impossível. Dentro do exposto neste trabalho, meramente introdutório, espero ter auxiliado a você leitor ou educando a introduzir este tema em sua vida profissional e a partir daqui iniciar um estudo mais aprofundado. Ao profissional da educação, espero sinceramente ter lhe proporcionado um material que venha auxiliar uma introdução didática e que também facilite o início de seu trabalho.

Como última lição desta obra carregue o programa TABUADA.asm no ambiente de programação emu8086 e acione o botão emulate quando surgir a caixa de diálogo emulator: Tabuada.asm\_ acione o botão aux e escolha a opção listing que apresentará a tela do programa Bloco de notas com o código do programa escrito em linguagem Assembly e em código de máquina em hexadecimal.