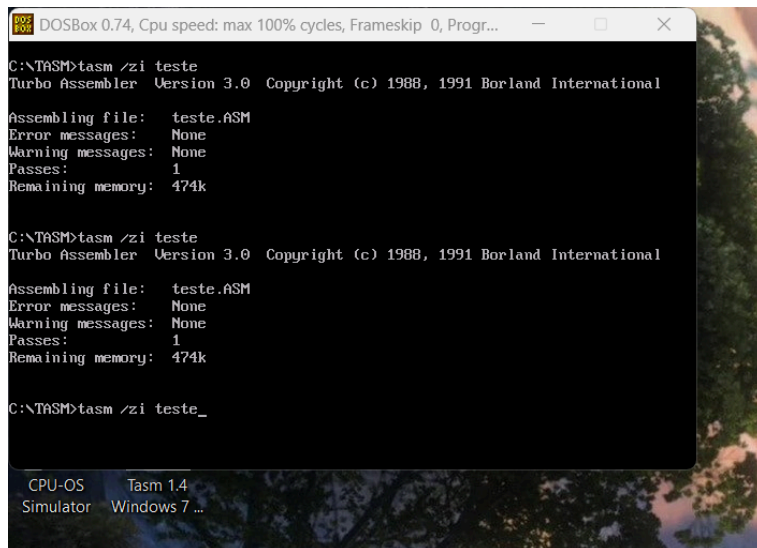


Arquitetura de Computadores
Professor: Leandro Coelho
Aluna: Ana Paula Santos Pinheiro

EXERCÍCIO 01:

Criar um arquivo texto chamado primeiro, com a extensão .asm (primeiro.asm) e que contenha o código abaixo. Salve em um local onde possa manipulá-lo com facilidade. Compile o programa primeiro.asm e execute o arquivo binário. Verifique que o programa não faz absolutamente nada, apenas executa e finaliza corretamente. Muito bem. Feito isto, execute o Turbo Debugger para se familiarizar com a interface. Treine as possibilidades descritas pelo guia de forma satisfatória. (Não esqueça de utilizar as opções /zi e /v no TASM e no TLINK, respectivamente).



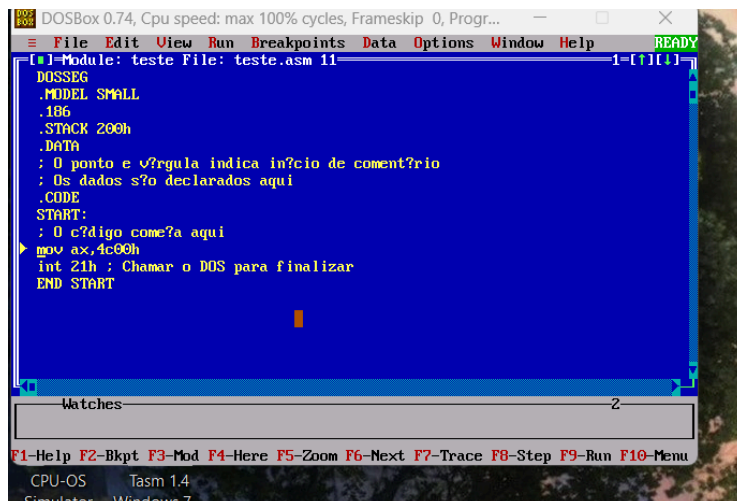
```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Progr...
C:\TASM>tasm /zi teste
Turbo Assembler Version 3.0 Copyright (c) 1988, 1991 Borland International

Assembling file: teste.ASM
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 474k

C:\TASM>tasm /zi teste
Turbo Assembler Version 3.0 Copyright (c) 1988, 1991 Borland International

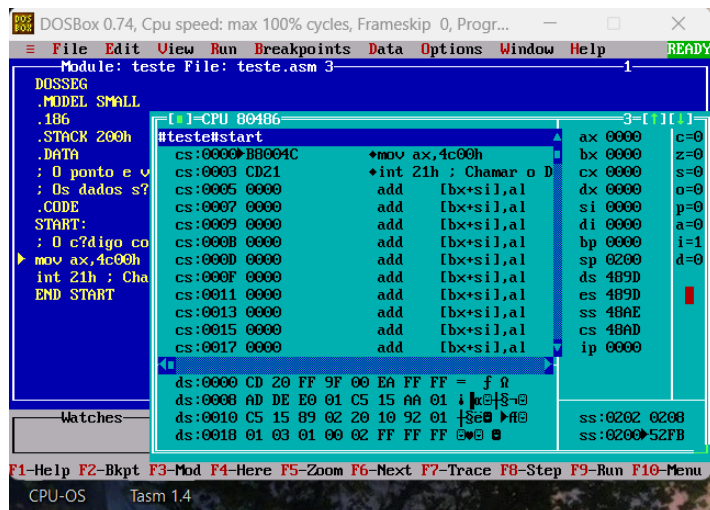
Assembling file: teste.ASM
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 474k

C:\TASM>tasm /zi teste_
```



```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Progr...
File Edit View Run Breakpoints Data Options Window Help
Module: teste File: teste.asm 11
DOSSEG
.MODEL SMALL
.186
.STACK 200h
.DATA
; 0 ponto e v?rgula indica in?cio de coment?rio
; Os dados s?o declarados aqui
.CODE
START:
; 0 c?digo come?a aqui
mov ax,4c00h
int 21h ; Chamar o DOS para finalizar
END START

Watches
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
CPU-OS Tasm 1.4
Simulator Windows 7...
```



EXERCÍCIO 02:

Utilizando o método experimental, que consiste em codificar e testar, experimentalmente, compilando o programa assembly e executando-o em seguida, verifique cada uma das instruções abaixo e sinalize qual ou quais são válidas para o repertório original do microprocessador 8086.

1) `mov ax, bl;`

Invalida, o 8086 não permite mover diretamente um valor de um registrador de 8 bits (`bl`) para um registrador de 16 bits (`ax`). Ambos os operandos devem ter o mesmo tamanho.

2) `mov ds, cs;`

Inválida, não é permitido mover diretamente o valor de um registrador de segmento para outro. Para alterar o valor de `ds`, é necessário primeiro mover o valor de `cs` para um registrador geral e, em seguida, para `ds`.

3) `mov cs, ax;`

Invalida, o registrador de segmento de código (`cs`) não pode ser modificado diretamente com a instrução `mov`. Alterações em `cs` devem ser feitas através de instruções de controle de fluxo, como `jmp` ou `call`.

4) `mov al, [ax];`

Invalida, o 8086 não permite o uso de `ax` como registrador de índice para endereçamento de memória. Registradores válidos para esse propósito são `bx`, `si` e `di`.

5) `mov ax, [si][di];`

Invalida, a sintaxe está incorreta. O correto seria `mov ax, [si + di]`.

6) `mov ax, [si+di+0];`

Válida, essa instrução é aceita no 8086. O processador permite o uso de registradores de índice, como `si` e `di`, combinados com um deslocamento constante (no caso, `+0`). Embora o `+0` seja redundante, ele não invalida a instrução.

7) `mov cx, 100[di+bp];`

Valida, O 8086 suporta o acesso à memória utilizando combinações de registradores como `BP` e `DI` com deslocamentos imediatos. A instrução `mov cx, 100[di+bp];` é válida, pois o processador pode calcular o endereço de memória somando os valores de `DI`, `BP` e o deslocamento imediato

8) `mov ax, 100[bx+1+si+1+1]`

Invalida, o 8086 permite endereçamento de memória com registradores e deslocamentos, mas não é permitido somar múltiplos deslocamentos diretamente na sintaxe de uma única instrução. Ou seja, você não pode fazer algo como `bx+1+si+1+1` diretamente dessa maneira.

A forma correta seria somar os valores dos deslocamentos primeiro ou fazer a soma corretamente, como em `100[bx+si+3]` (onde `3` é a soma de todos os deslocamentos).

EXERCÍCIO 03:

Baseado no código original do arquivo primeiro.asm, insira as instruções do pdf, depois da etiqueta `START`, compile e link o programa. Inicie o depurador e execute o programa passo a passo. A cada iteração você deverá escrever os valores que irá assumir. A cada iteração você deverá escrever os valores que irá assumir o registrador `AL` e os diferentes flags envolvidos no processo.

Código corrigido:

```
DOSSEG
.MODEL SMALL
.186
.STACK 200h
.DATA
; O ponto e vírgula indica início de comentário
; Os dados são declarados aqui
.CODE

START:
; O código começa aqui
mov al,10
lab0: cmp al,-1
jg lab1
mov ax,0
```

```
jz lab2
```

```
lab1:  
dec al
```

```
lab2:  
jc lab0
```

```
int 21h ; Chamar o DOS para finalizar  
END START
```

Que condição, relacionada com o registro AL, testa a instrução “ja lab1”?

A instrução **ja** (Jump Above) verifica se o valor no registrador AL é maior que o operando comparado, tratando os valores como números sem sinal. A condição que precisa ser atendida é que as flags Carry (CF) e Zero (ZF) estejam limpas (CF = 0 e ZF = 0), o que indica que o valor de AL é maior que o operando, considerando números sem sinal.

O problema principal do programa está no uso do comando **ja** (Jump Above), que é adequado para comparações de números sem sinal, mas o programa utiliza o registrador AL, que pode conter valores negativos devido ao decremento. O **ja** não lida corretamente com números negativos representados em complemento de dois. Para corrigir isso, é necessário usar o comando **jg** (Jump Greater), que faz comparações levando em consideração o sinal dos números. Dessa forma, o programa funcionará corretamente mesmo quando o valor de AL for negativo após o decremento.

EXERCÍCIO 04:

Existe uma segunda codificação para a instrução do exemplo anterior que também copia BX em CX! Qual é esta codificação?

R= Pesquisando, encontrei a instrução **XCHG** (87D9), que realiza a troca de valores entre dois registradores. No caso, o comando **XCHG BX, CX** troca os valores de **BX** e **CX**. Isso significa que o valor que estava em **BX** será transferido para **CX**, e o valor que estava em **CX** será transferido para **BX**. Assim, em vez de usar a instrução **MOV** para copiar o valor de um registrador para o outro, o **XCHG** faz a troca direta entre eles.

Código para essa codificação:

```
DOSSEG  
.MODEL SMALL
```

```
.186
```

```
.STACK 200h
```

```
.DATA
```

```
.CODE
START:
```

```
; O código começa aqui
    mov bx, 10    ; Atribui o valor 10 ao registrador BX

    mov cx, 9     ; Atribui o valor 9 ao registrador CX

    XCHG BX, CX   ; Troca os valores de BX e CX

    int 21h ; Chamar o DOS para finalizar
END START
```

Encontre a codificação para as seguintes instruções:

MOV BX, [BX] : 8B1F

MOV AX, BX: 8BC3



[] CPU 80486	
#programa#start	
cs:0000 BB0A00	♦ mov bx, 10 ; Atribui
cs:0003 B90900	♦ mov cx, 9 ; Atribui
cs:0006 87D9	♦ XCHG BX, CX ; Troca
cs:0008 8B1F	♦ MOV BX, [BX]
cs:000A 8BC3	♦ MOV AX, BX

EXERCÍCIO 05:

Existem erros semânticos (i.e., instruções válidas para o compilador, mas que não realizam as atividades esperadas).

Descreva os erros informando o que está errado: O código usa `mov ax, array1[bx]` para acessar os elementos de `array1`, mas isso não leva em consideração o tamanho de cada elemento. Como estamos lidando com um vetor de palavras de 2 bytes (usando o comando `dw`), o código tenta acessar os elementos de forma incorreta.

O que deveria acontecer: Quando o código acessa o vetor, o índice precisa ser ajustado para refletir o tamanho de cada elemento. Como cada palavra ocupa 2 bytes, o índice `bx` deve ser multiplicado por 2 para obter o endereço correto de cada elemento no array. Ou seja, ao acessar o vetor `array1`, o código deveria ser algo como `mov ax, array1[bx*2]` para garantir que o endereço correto do elemento seja acessado.

Código corrigido:

```
DOSSEG
.MODEL Small
.186
.STACK 100h
```

```

.DATA
Tam dw 10 ; int Tam=10;
array1 dw 0,1,2,3,4,5,6,7,8,9 ; int array1[10] = {0,1,2,3,4,5,6,7,8,9}
array2 dw 10 dup(?) ; int array2[10]
.CODE
MAIN PROC
mov ax,@DATA
mov es,ax ;inicializar registro de segmento de dados
mov bx, Tam
dec bx ; bx = 9, aponta para o último elemento do array

moveLoop:
mov si, bx ; Armazenar o valor de bx
shl si, 1 ; Multiplica si (índice) por 2
mov ax, [array1+si] ; Acessa o valor correto de array1
mov [array2+si], ax ; Copia o valor para array2

sub bx, 1
jg moveLoop ; Continua enquanto bx >= 0

mov ah,4ch
int 21h
MAIN ENDP
END MAIN

```

EXERCÍCIO 06:

Qual a função de cada uma das linhas comentadas de 1 a 4. Explique a função “lea” Crie um pequeno programa que declare um vetor de 5 posições e na terceira posição armazene o valor 40. (considere somente o escopo principal)

a) mov bx, i

Esta instrução move o valor de **i** (um valor armazenado em algum lugar, pode ser uma variável ou constante) para o registrador **BX**. O valor de **i** será agora armazenado em **BX**.

b) add bx, bx

Soma o valor de **BX** com ele mesmo.

c) lea ax, Quadrados

lea significa "Load Effective Address" (Carregar Endereço Efetivo). Ele carrega o endereço de memória da variável ou rótulo **Quadrados** no registrador **AX**.

d) add bx, ax

Esta instrução adiciona o valor de **AX** (o endereço de **Quadrados**) ao valor de **BX** (que é o índice modificado) e armazena o resultado em **BX**.

e) `mov ax, [bx]`

Carrega o valor armazenado no endereço de memória apontado por **BX** no registrador **AX**.

Crie um pequeno programa que declare um vetor de 5 posições e na terceira posição armazene o valor 40. (considere somente o escopo principal)

```
.MODEL SMALL  
.STACK 200h
```

```
.DATA  
Quadrados dw 1, 4, 9, 16, 25 ; Array com 5 valores
```

```
.CODE  
START:  
    mov ax, @data  
    mov ds, ax
```

```
; Acessando Quadrados[2] e colocando o valor em AX  
    mov bx, 2  
    shl bx, 1
```

```
; Modificando Quadrados[2] para 100  
    mov [Quadrados + bx], 40
```

```
    mov ax, [Quadrados + bx]
```

```
; Fim do programa  
    mov ah, 4ch ; Interrupção para terminar o programa  
    int 21h
```

```
END START
```

EXERCÍCIO 07:

O programa acima manipula uma matriz 10 x 10. Sabendo-se que a instrução “MUL”, funciona conforme o enunciado abaixo, responda: O que faz o programa. Se necessário utilize o TD.

R= O programa manipula uma matriz 10x10 e apaga um valor específico nela. Ele calcula qual posição da matriz deve ser alterada usando a linha e a coluna informadas. Para fazer isso, o programa usa a instrução **MUL**, que multiplica a linha pelo número de colunas (10) para encontrar o índice da célula desejada. Depois, ele calcula o endereço exato dessa célula, acessa ela e coloca o valor zero nela, apagando o conteúdo.

Por exemplo, para acessar o elemento na posição **linha = 3** e **coluna = 2**, o programa calcula:

$3 * 10 + 2 = 32$, ou seja, o 32º elemento da matriz (considerando um índice começando do 0).