Assembly Language LAB

Islamic University – Gaza
Engineering Faculty
Department of Computer Engineering
2013
ECOM 2125: Assembly Language LAB
Created by: Eng. Ahmed M. Ayash
Modified and Presented By: Eihab S. El-Radie

Lab # 3

Data Transfer & Arithmetic

Objective:

To be familiar with Data Transfer & Arithmetic in Assembly.

Introduction:

1. Data Transfer

❖ MOV Instruction

Move from source to destination.

Syntax:

MOV destination, source

MOV is very flexible in its use of operands, as long as the following rules are observed:

- ✓ Both operands must be the same size.
- ✓ Both operands cannot be memory operands.
- ✓ CS, EIP, and IP cannot be destination operands.
- ✓ An immediate value cannot be moved to a segment register.

Here is a list of the general variants of MOV, excluding segment registers:

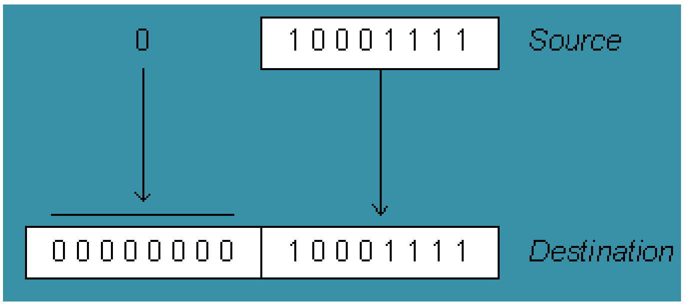
1. MOV *reg,reg*
2. MOV *mem,reg*
3. MOV *reg,mem*
4. MOV *mem,imm*
5. MOV *reg,imm*

❖ MOVZX

When you copy a **smaller value** into a **larger destination**, the MOVZX instruction (*move with zero-extend*) fills (extends) the upper half of the destination with zeros.

Syntax:

MOVZX destination, source



This instruction is only used with unsigned integers. There are three variants:

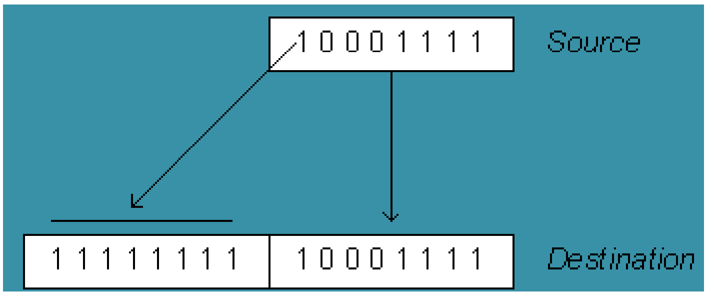
1. MOVZX *reg32,reg/mem8*
2. MOVZX *reg32,reg/mem16*
3. MOVZX *reg16,reg/mem8*

❖ MOVSX

The MOVSX instruction (*move with sign-extend*) fills the upper half of the destination with a copy of the source operand's **sign bit**.

Syntax:

MOVSX destination, source



This instruction is only used with signed integers. There are three variants:

1. MOVSX *reg32,reg/mem8*
2. MOVSX *reg32,reg/mem16*
3. MOVSX *reg16,reg/mem8*

Note:
The MOV instruction never affects the flags.

❖ XCHG

The XCHG (exchange data) instruction exchanges the contents of two operands. At least one operand must be a register. There are three variants:

1. XCHG *reg,reg*
2. XCHG *reg,mem*
3. XCHG *mem,reg*

The rules for operands in the XCHG instruction are the same as those for the MOV instruction **except** that XCHG does not accept immediate operands.

❖ Direct-Offset Operands

You can add a displacement to the name of a variable, creating a direct-offset operand. This lets you access memory locations that may not have explicit labels. Let's begin with an array of bytes named **arrayB**:

```
arrayB BYTE 10h,20h,30h,40h,50h
```

If we use MOV with **arrayB** as the source operand, we automatically move the first byte in the array:

- **mov al , arrayB   ; AL = 10h**

We can access the second byte in the array by adding 1 to the offset of **arrayB**:

- **mov al , [arrayB+1] ; AL = 20h**

The third byte is accessed by adding 2:

- **mov al , [arrayB+2] ; AL = 30h**

An expression such as **arrayB+1** produces what is called an ***effective address*** by adding a constant to the variable's offset. Surrounding an effective address with **brackets** indicates the expression is dereferenced to obtain the **contents** of memory at the address. The brackets are not required by MASM, so the following statements are equivalent:

- **mov al,[arrayB+1]**
- **mov al,arrayB+1**

Word and Doubleword Arrays:

In an array of 16-bit words, the offset of each array element is **2** bytes beyond the previous one. That is why we add 2 to **ArrayW** in the next example to reach the second element:

```
.data
arrayW WORD 100h,200h,300h
.code
mov ax,arrayW        ; AX = 100h
mov ax,[arrayW+2]    ; AX = 200h
mov ax,[arrayW+4]    ; AX = 300h
```

Similarly, the second element in a doubleword array is 4 bytes beyond the first one:

```
.data
arrayD DWORD 10000h,20000h
.code
mov eax,arrayD        ; EAX = 10000h
mov eax,[arrayD+4]    ; EAX = 20000h
```

2. Arithmetic (Adding and Subtracting Numbers)

❖ INC destination

destination ← destination + 1

Add 1 from destination operand, operand may be register or memory.

Syntax:

```
INC  reg/mem
```

Example:

INC ax

❖ DEC destination

destination ← destination – 1

Subtract 1 from destination operand, operand may be register or memory.

Syntax:

```
DEC  reg/mem
```

Example:

DEC ax

- INC and DEC affect five status flags
  - Overflow, Sign, Zero, Auxiliary Carry, and Parity
  - Carry flag is NOT modified

For Example

```
.DATA
        B SBYTE -1      ; 0FFh
        A SBYTE 127     ; 7Fh
.CODE
        inc B           ; B= 0      OF=0  SF=0  ZF=1  AF=1  PF=1
        dec B           ; B= -1     OF=0  SF=1  ZF=0  AF=1  PF=1
        inc A           ; A = 128   OF=1  SF=1  ZF=0  AF=1  PF=0
```

❖ ADD instruction

The ADD instruction adds a source operand to a destination operand of the same size. The form of the ADD instruction is:

```
ADD destination, source    ; destination operand = destination operand + source operand
```

The destination operand can be a register or in memory. The source operand can be a register, in memory or immediate.

Flags:
ZF = 1 if dest + src = 0 ; Zero flag
SF = 1 if dest + src < 0 ; Sign flag
CF = 1 if dest + src generated carry out of most significant bit
OF = 1 if dest + src resulted in signed overflow
AF = 1 if when an operation produces a carry out from bit 3 to bit 4
PF = 1 if an instruction generates an even number of 1 bits in the low byte of the destination operand.

❖ SUB instruction

The SUB instruction subtracts a source operand from a destination operand.The form of the SUB instruction is:

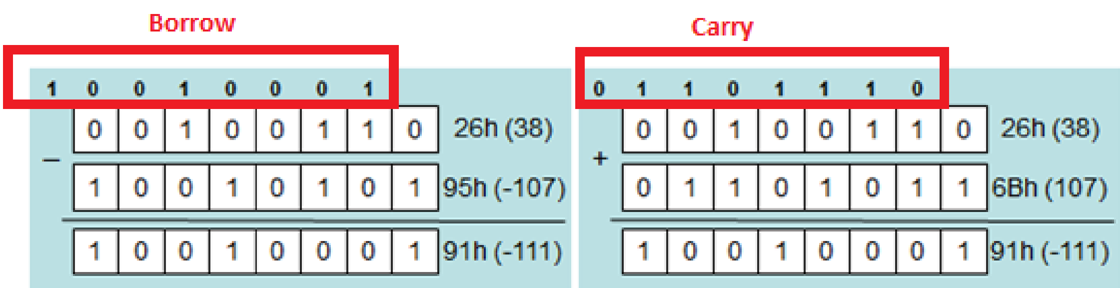
SUB destination, source ;destination operand = destination operand - source operand

The destination operand can be a register or in memory. The source operand can be a register, in memory or immediate.

Flags:
ZF = 1 if dest - src = 0 ; Zero flag
SF = 1 if dest - src < 0 ; Sign flag
CF = 1 if required a borrow at most-significant-bit
OF = 1 if resulted in signed overflow
AF = 1 if resulted in borrow into the low-ordered four bit of 8-, 16-, or 32-bit operands.
PF = 1 if an instruction generates an even number of 1 bits in the low byte of the destination operand.

Internally, the CPU can implement subtraction as a combination of negation and addition. Two's-complement notation is used for negative numbers.
For example:

```
mov al,26h    ;al=26h
sub al,95h    ;al=91h
```



For this example the flags values are:

ZF = 0; SF = 1; CF = 1; OF = 1; AF = 0; PF = 0

Note:

- **Take ZF, PF and SF values from subtraction or addition.**
- **Take OV values from addition.**
- **Take CF and AF values from subtraction.**

❖ NEG (negate) Instruction

It reverses the sign of an operand (Like 2's Complement). Operand can be a register or memory operand.

Syntax:

`NEG` ***reg/mem***

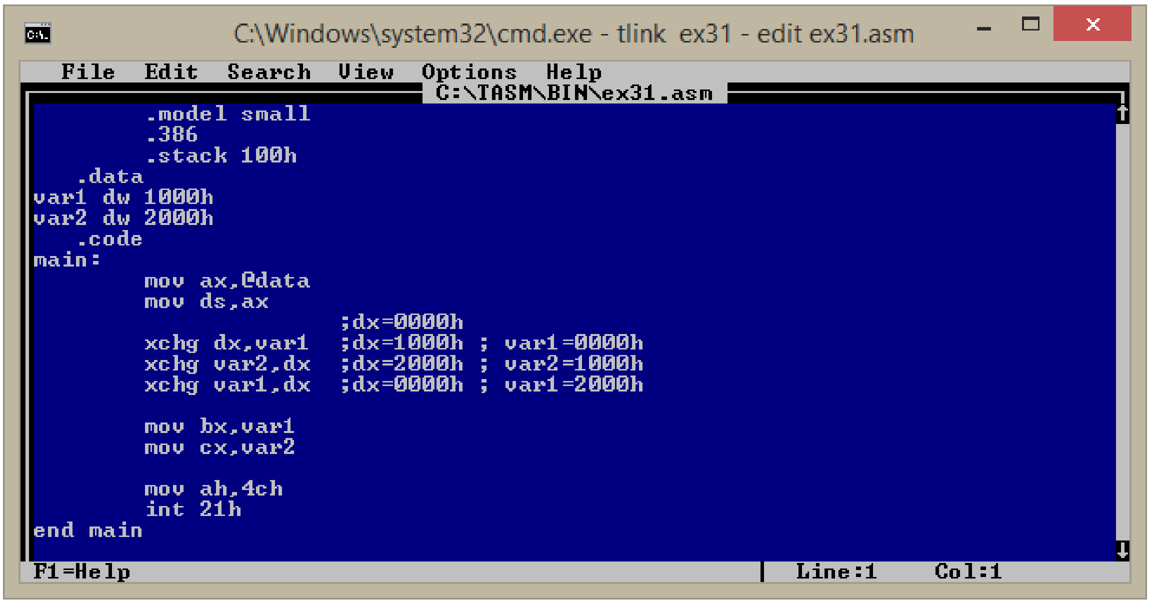
(Recall that the two's complement of a number can be found by reversing all the bits in the destination operand and adding 1.)

- NEG affects all the six status flags
  - ✧ Any nonzero operand causes the carry flag to be set

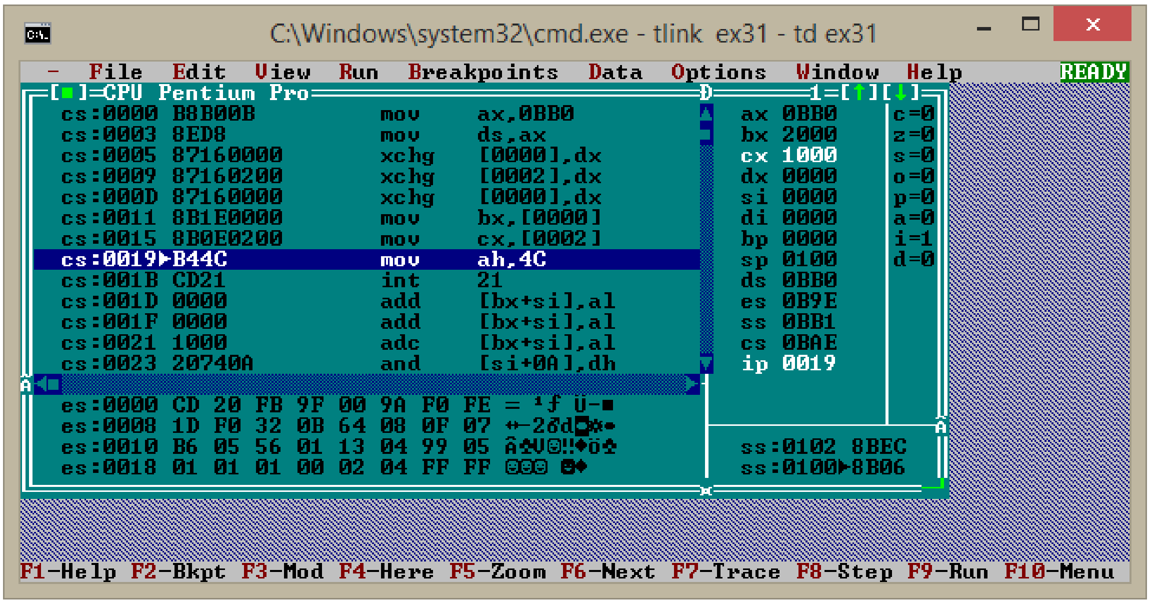
Lab work:

Excercise1:

Exchange the content of the following variables
Var1 dw 1000h
Var2 dw 2000h



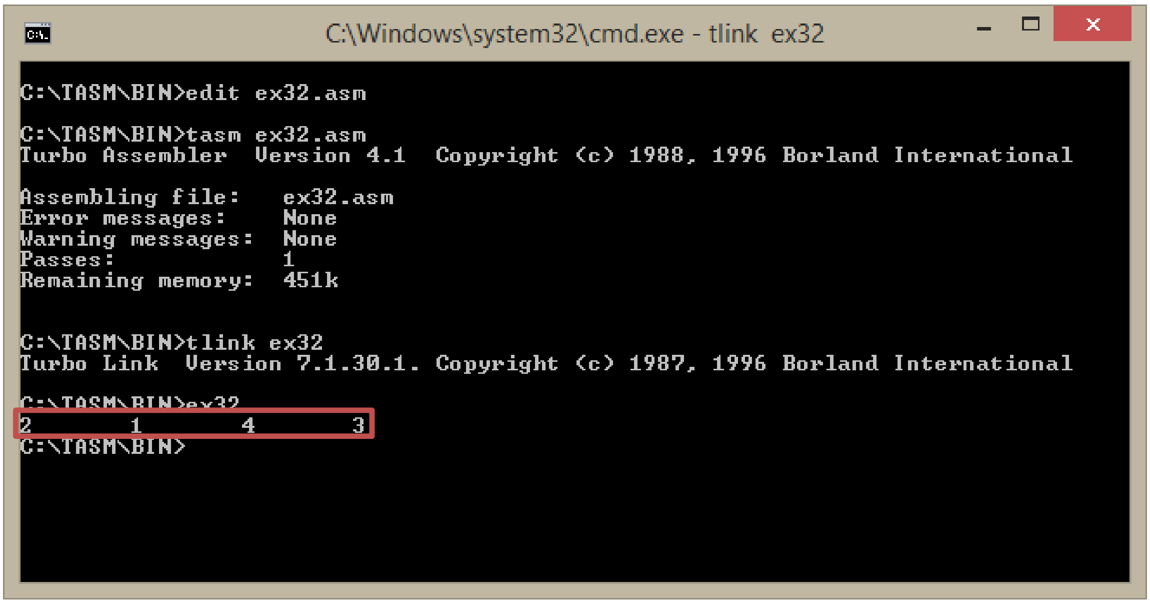

```
        .model small
        .386
        .stack 100h
    .data
var1 dw 1000h
var2 dw 2000h
    .code
main:
        mov ax,@data
        mov ds,ax
                        ;dx=0000h
        xchg dx,var1    ;dx=1000h ; var1=0000h
        xchg var2,dx    ;dx=2000h ; var2=1000h
        xchg var1,dx    ;dx=0000h ; var1=2000h

        mov bx,var1
        mov cx,var2

        mov ah,4ch
        int 21h
end main
```



Excercise2:

Write a code to increment the odd elements in the array numbers and decrement the even elements on the same array:

Numbers db 1,2,3,4



```
        .model small
        .386
        .stack 100h
      .data

numbers db 1,2,3,4

     .code
main:
      mov ax,@data
      mov ds,ax

     inc [numbers]
     dec [numbers+1]
     inc [numbers+2]
     dec [numbers+3]

     mov ah,02h
     mov dl,[numbers]
     add dl,'0'              ;30h decimal to ascii
     int 21h

     mov dl,09h     ;tab
     int 21h

     mov dl,[numbers+1]
     add dl,'0'
     int 21h

     mov dl,09h
     int 21h

     mov dl,[numbers+2]
     add dl,'0'
     int 21h

     mov dl,09h
     int 21h

     mov dl,[numbers+3]
     add dl,'0'
     int 21h

      mov ah,4ch
      int 21h
end main
```

Note:

If we define the array like this:

Numbers dw 1,2,3,4

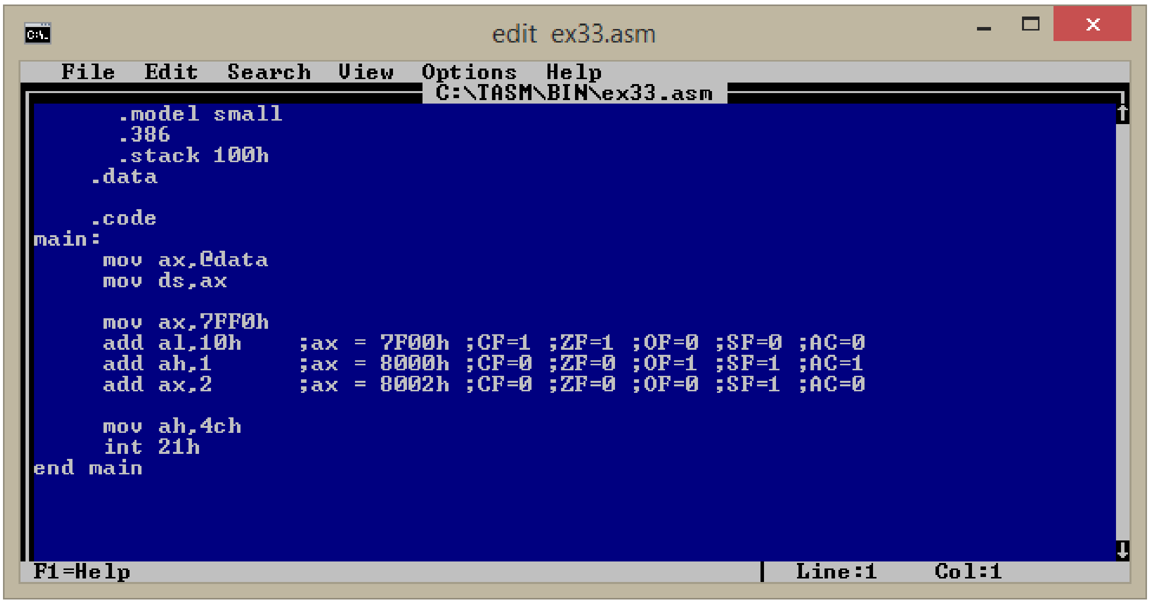
Then the offsets will be as follows:

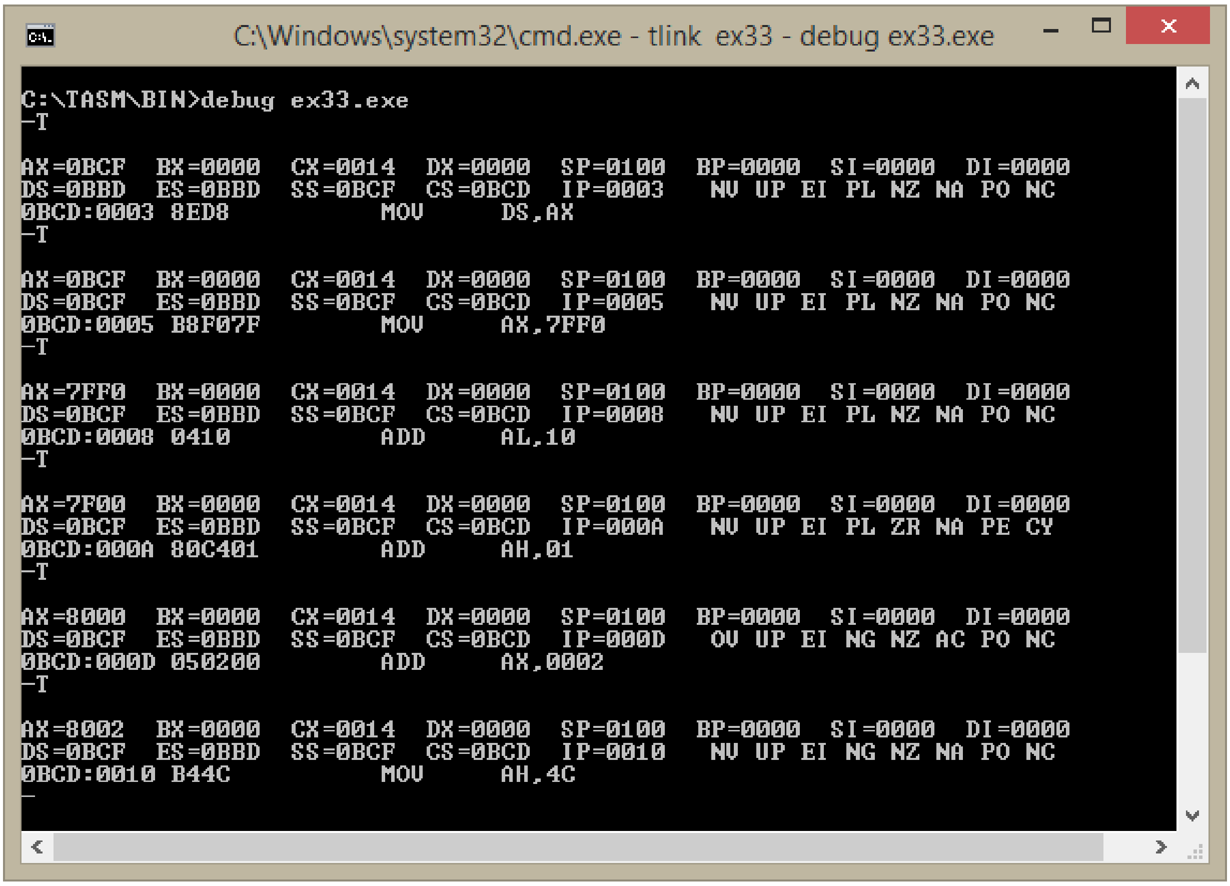
```
inc [numbers]
dec [numbers+2]
inc [numbers+4]
dec [numbers+6]
```

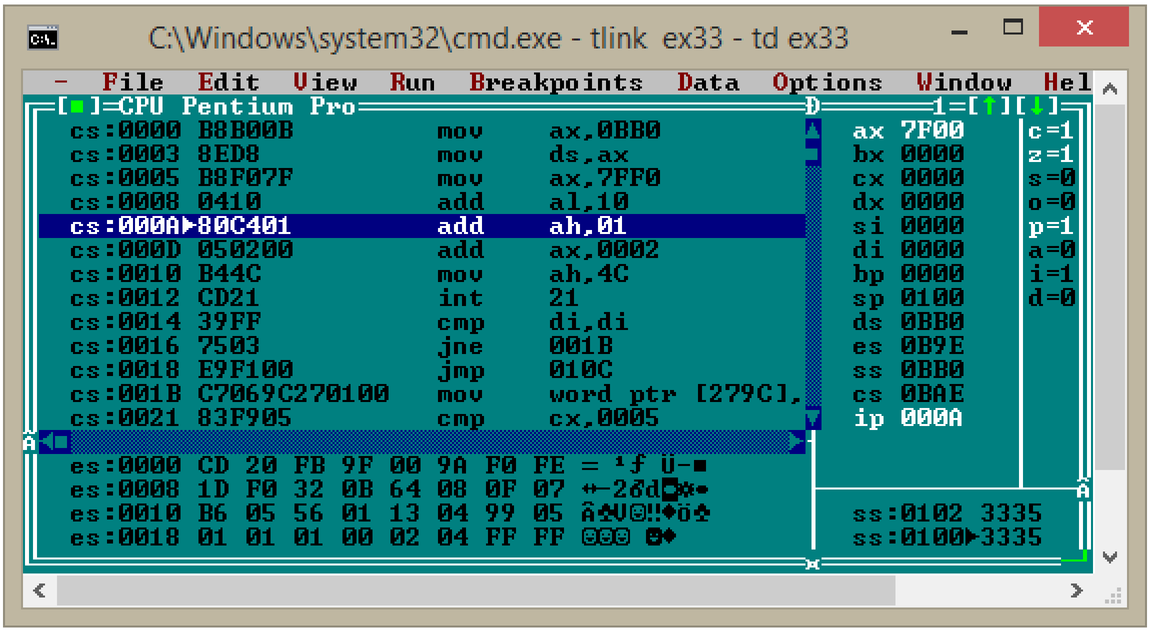
Excercise3:

Debug the following code to find which flags will be affected after each instruction from these flags (CF, ZF, OF, SF, AC)

```
mov ax,7FF0h
add al,10h
add ah,1
add ax,2
```







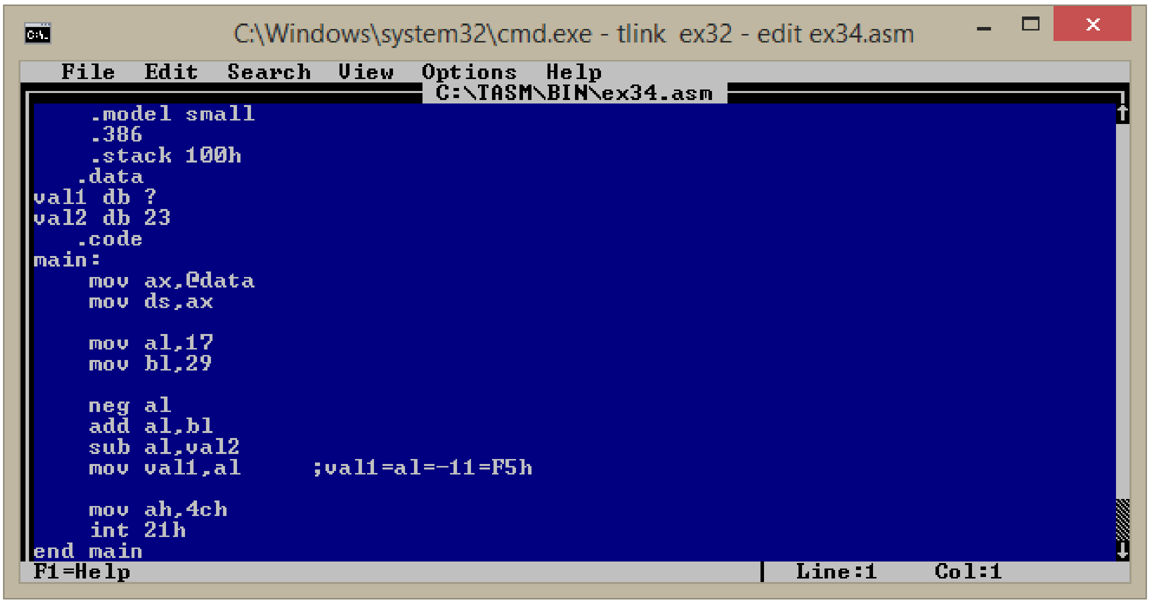
Excercise4:

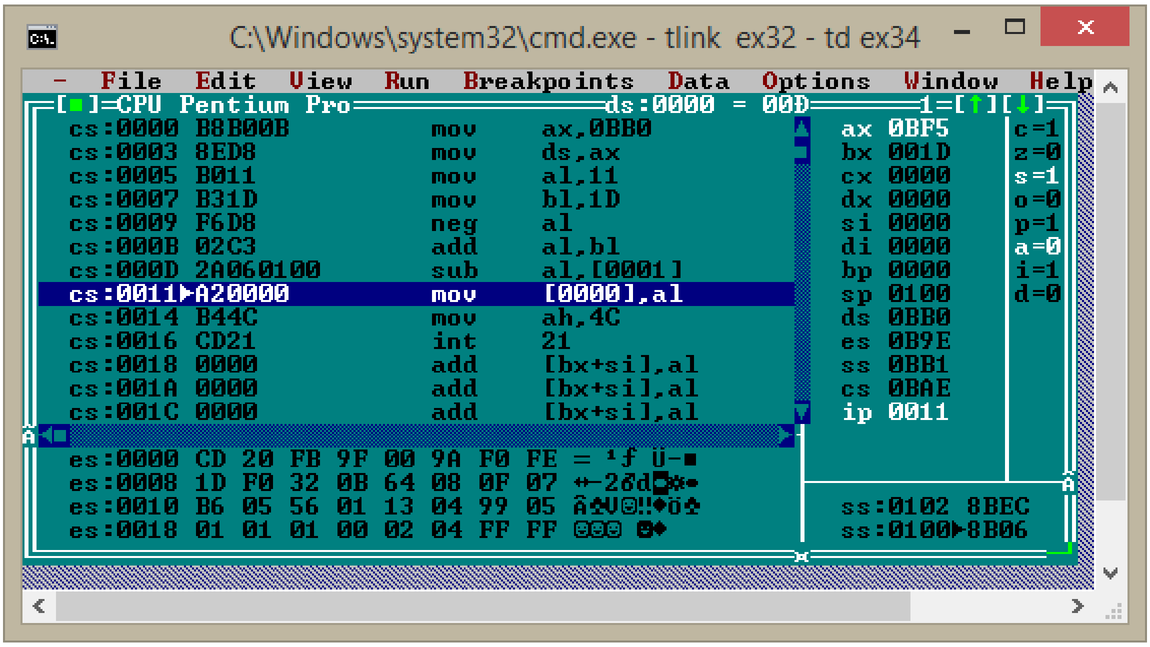
Write an assembly code that perform the following addition

val1= (-al+ bl) - va12

Consider the following initialization

```
val1 db ?
val2 db 23
mov al,17
mov bl,29
```





Homework:

1. Write an assembly code to put the byte array ***your name*** in the double word array ***New_yourname*** in the opposite direction:

eihab db 'E','I','H','A','B'
New_eihab dd 5 dup(?),'$'

Then print the content of array ***New_yourname***

2. Use the array odd to find the square of the numbers between 1 and 5 and put the square of each number in the array square:

odd db 1,3,5,7,9
square db 5 dup(?)

Quiz Next Week in LAB3

☺