COE 205: Computer Organization & Assembly Language
Introductory Experiment-C

By
Louai Al-Awami

Using the CodeView

The CodeView (CV) is a useful utility that allows you to trace your program and watch the status of your computer system while running your program. You will use this tool to debug your program whenever needed.

```
TITLE "programB1"
.MODEL SMALL
.STACK 100
.DATA
        A       DB      2
        B       DB      3
        C       DB      4
.CODE
        MOV     AX,@DATA        ; 1
        MOV     DS,AX           ; 2

        MOV     AL,B            ; 3
        MOV     BL,A            ; 4
        ADD     AL,BL           ; 5
        ADD     C,AL            ; 6

        MOV     AX,4C00H        ; 7
        INT     21H             ; 8
END
```

Fig 1: progB1

Before you start the steps below, write **programB1** that appears in Fig 1 using a text editor and save it as **progB1.asm** in the directory you created in the last lab. Also, notice the number in front of each instruction because it will be used as a reference in the coming paragraphs.

After you save the program, assemble it and link it as you have learned in the previous experiment. You should have and .exe file in your directory corresponding to the same program.

Running the CodeView

- Go to the Start> Programs> Masm 611> Masm Prompt.
- You will have the command prompt coming up, then change the directory to the directory containing your program.
- Then, write

Z:>COE205>LAB2> cv progB1

and press Enter.

- You will get the following screen

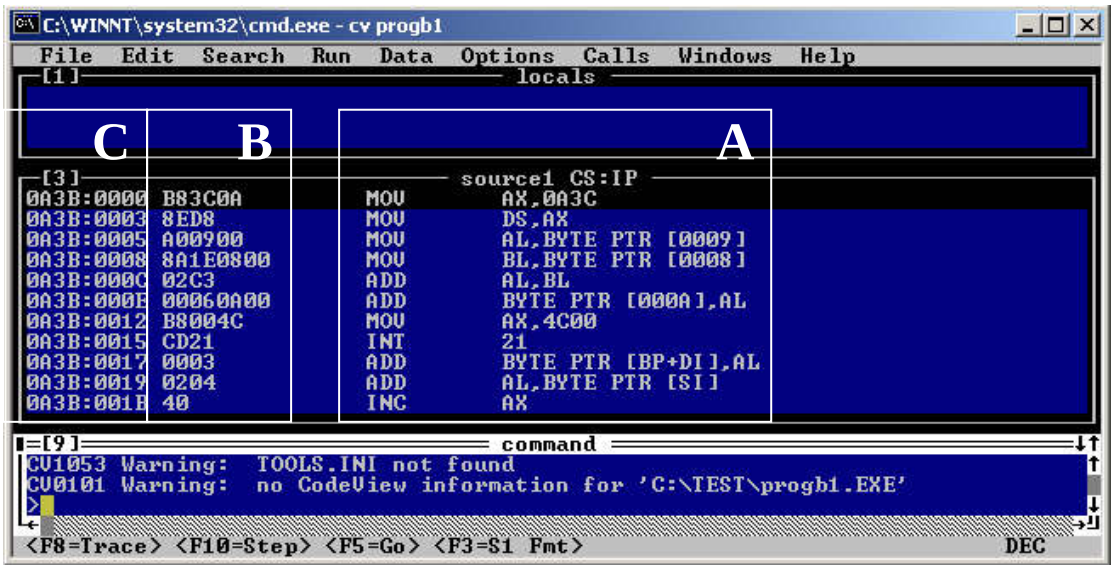

Fig 2: CodeView Main Screen.

Below, we will discuss the three parts that appear in the source window as in Fig 2.

NOTE: the values and addresses given below may change when you run the CV on different machines; this is due to the operating system memory management. Also, note that all the values given are in hexadecimal.

A: Source Code

This part shows the assembly language program. Each instruction appears in one line. The instruction that is black-highlighted is the instruction that is going to be executed next. For instance, in Fig 2 the instruction "MOV AX,0A3C" is the next instruction to be executed.

Type of Instruction	Example
No Operands	CLD
One Operand	INC AX
Two Operands	MOV AX, BX

Table 1: Type of Instructions and Examples

An Assembly language instruction may have zero, one or two operands. Example of each can be show in the Table 1. In case of two-operand instructions, we call the right-hand operand the **source** left-hand operand the **destination**. For

example, the MOV instruction copies the value of the destination to the source. In instruction#2, the value of the register AX which is the source is copied into the register DX which is the destination.

Note that some instructions are viewed in a different from than the source code. For example, the instruction “MOV AX, @Data“ has been replaced with “MOV AX,0A3C”. In reality, there is no difference; the value 0A3C is nothing but the data segment address for this program after its loaded by the MS-DOS operating system. Another Example is the third instruction “MOV AL, B”, which has been replaced by “MOV AL, BYTE PTR[0009]”. Where BYTE PTR[0009] is nothing but a pointer to the offset of the variable “B”. In this case, variable “B” has been stored in the location with offset 0009H in the data segment.

B: Machine Code

An instruction represented as a machine code is shown in part B. This shows how exactly the instruction is represented inside the memory. Generally, the first byte of the instruction is called the *opcode* and it indicates operation (MOV, ADD, .etc) and the addressing mode. For example, see the two instructions below.

	Machine Code	Assembly Code
1	B83C0A	MOV AX, 0A3C
2	B8004C	MOV AX, 4C00

Notice, that the higher bytes are the same (B8) since both instructions involves moving an immediate value into a word register. The lower byte however, carries the value to be loaded into the register in reveres order.

Different instructions have different length depending on their type. To see that, compare the following two instructions taken from the code in Fig 1.

	Machine Code	Assembly Code	Length (Bytes)
1	B83C0A	MOV AX, 0A3C	3
2	8ED8	MOV DS,AX	2

This different is made to get better efficiency in performance and space making the instructions that are used more often shorter in length. More details about the instruction formatting can be found in the textbook.

C: Instruction Address

Part C shows the address of the instruction in the data segment. Because instructions differ in lengths, they occupy different parts of memory. The address is divided into two parts, *a segment address* and *an offset*. The address looks like

Segment Address : Offset

The segment address in this case represents the *code segment* (CS). For example, instruction#6 “ADD C,AL” has the address 0A3B:000E. Which means it is stored in segment 0A3B of the code segment. The offset indicates a specific byte inside the segment.

The 8086 divides the memory into segments each is 64K Byte. If you calculate the number of bits required to address such a memory you will find it 16-bit (2 Byte) and this is why we have the offset as 2 bytes long.

Think of the memory as a country. Then, the segments correspond to cities inside the country. A segment address corresponds to a postal code of a city. As each city contains many houses, each segment is divided into bytes. Like we use the P.O. Box to address a house or an address in a city, we use the offset address a byte inside a segment.

Let us continue with our example. We saw that instruction#5 starts at address 0A3B:000E. Since the instruction is 4 bytes long, it ends at address 0A3B:0011. Similarly, instruction#6 starts at address 0A3B:0012. Note, that all instructions have the same segment address.

How to step your program

In order to execute the next instruction press F10. The curser will move one position down. You can also put a break point and make the program runs till this break point. To do this:

1- Highlight the instruction that you would like to stop at.

2- Go to **Data** menu and chose **Set Breakpoint**.

3- Press F5.

In order to reset the program, go to **Run** menu and select **Restart**.

Viewing the Registers Window

Another important window is the registers window. You can view it by Clicking Alt+7.

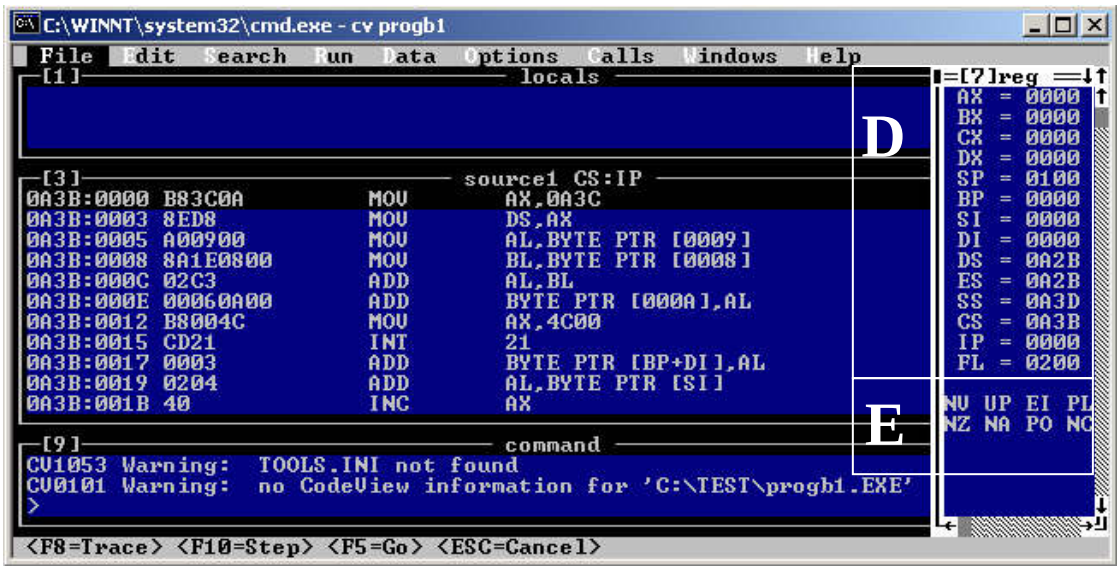

Fig 3: CV with Registers Window

D: Registers

This part shows the current values of the registers, there are 14 registers shown. The default view is 16-bit registers for the 8086. You can view the 32-bit registers by going to the **Options** menu and selecting **32-bit Registers**.

Consider the first instruction. It moves the value “0A3C” into the register AX. Try to execute this instruction as you have learned. After executing this instruction, the value of AX will change to “0A3C” instead of “0000”. You can notice that any value that changes is highlighted.

After executing the previous instruction, notice that besides AX, another registers also have been highlighted, namely IP and FL. What do you think the cause for this?

Overflow	Direction	Interrupt	Sign
NV: no overflow	**UP:** up	**EI**: Enable interrupt	**NG:** Negative
OV: overflow	**DN:** down	**DI:** Disable Interrupts	**PL:** positive
Zero	**Auxiliary**	**Parity**	**Carry**
ZR: Zero	**AC:** Auxiliary carry	**PE:** Parity even	**CY:** Carry
NZ: No zero	**NA:** No Auxiliary carry	**PO:** Parity odd	**NC:** No carry

Fig 4: Flags and Their Values

E: Flags

The lower part of the window contains the flags that indicate the status of the CPU after executing the last instruction. They are arranged as shown in Fig 4.

Viewing the Data Segment Window

To view the *data segment* (DS) windows, press Alt+5. The following screen will appear.

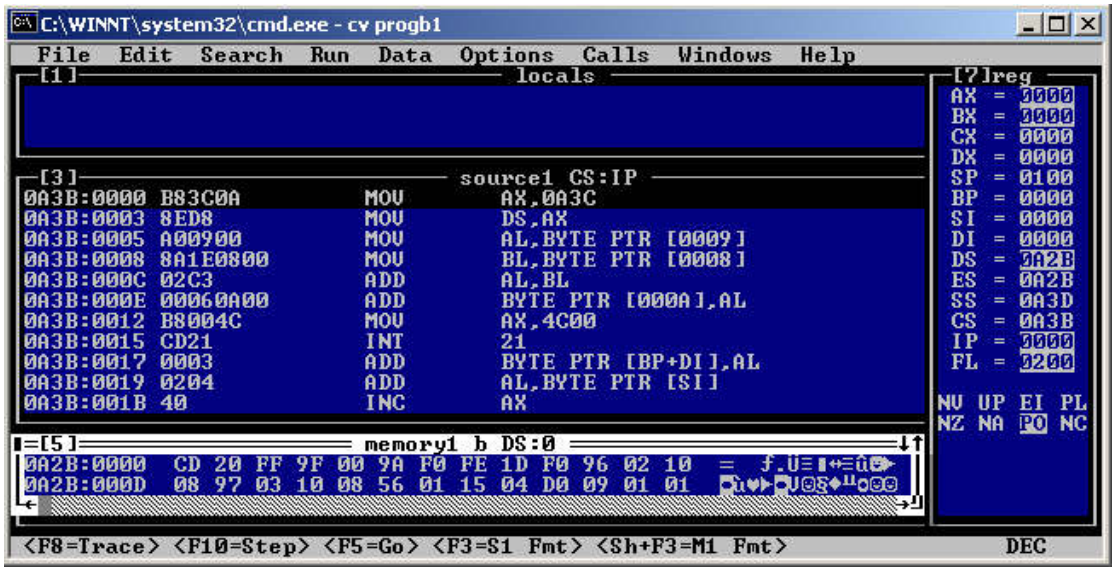

Fig 5: CV with Data Segment Windows

Notice that in Fig 5, each line contains 13 bytes. The address given at the beginning of the line is **Data Segment Address: Offset** of the first byte. This is why the next line starts from 000DH.

Be careful to one important point here. At the beginning of the program, the DS register was loaded with the value “0A3C”, which means that the data of our program is in the segment 0A3C. In the data segment register (DS) however, the segment shown initially is “0A2B” which is a different segment. To show the correct segment, put the cursor on the segment value in the data segment window and write the new value (0A3C). The new screen will look like Fig 6.

Now, let us see how to locate the values of the variables. Notice that the variable B has been replaced by BYTE PTR [0009]. This means that the offset of the address of B is 0009 inside the data segment. If you try to located the value B in the data segment as 0A3C:0009, you will find it 02.

As an exercise, try to find the values of A & C.

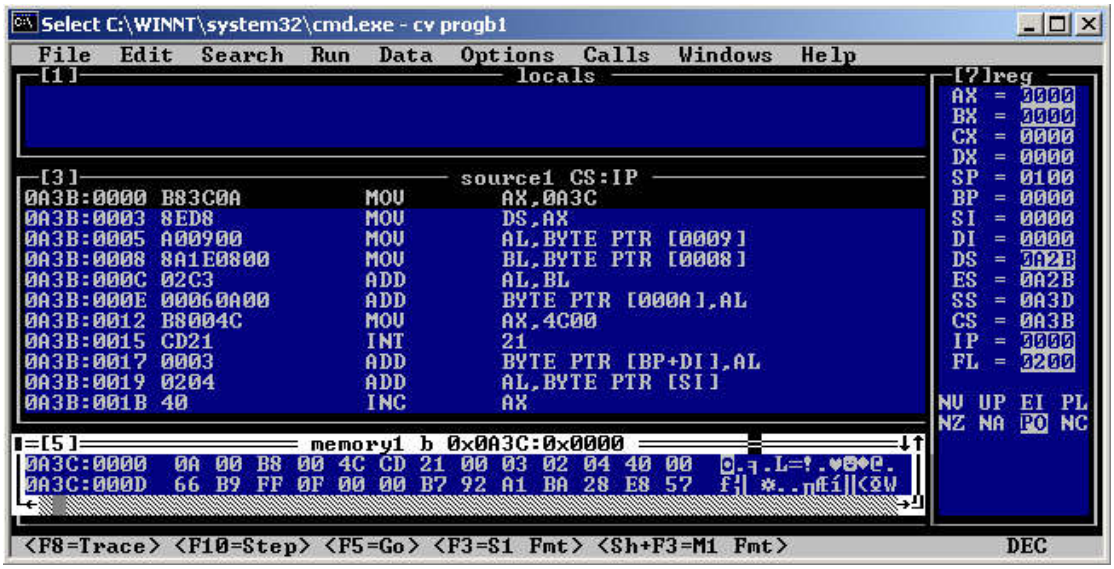

Fig 6: Data Segment with Correct Value.

Student Name: **ID#**

1. Write the following program and use the CV to answer the following questions?

```
TITLE "programB2"
.MODEL SMALL
.STACK 100
.DATA
        NUM1    DB      9
        NUM2    DB      8
        X       DB      'A'
.CODE
        MOV     AX,@DATA        ; 1
        MOV     DS,AX           ; 2

        MOV     AL,NUM1         ; 3
        ADD     NUM2,AL         ; 4
        MOV     BL,X            ; 5

        MOV     AX,4C00H        ; 6
        INT     21H             ; 7
END
```

a- What is the **starting address** of the memory where the code of this program is stored?

b- What is the **starting address** of the **data segment**?

c- What is the equivalent **binary** code for the instruction#3? What is its size?

d- How much **memory** is required to store the program?

Student Name: **ID#**

e- By looking at the binary code, what type of instruction do you think the opcode "B8" refers to?

f- What is the **address** of the location storing the variables NUM1 & NUM2?

g- What is the value stored in the memory location of NUM2 before and after executing instruction#4?

Before:

After:

h- Write the status of the flags and their meanings after executing instruction#4?

Overflow	**Direction**	**Interrupt**	**Sign**
Zero	**Auxiliary**	**Parity**	**Carry**

i- Run the program step-by-step and write the values of the *source* and *destination* before and after each instruction.

Instruction	**Source**		**Destination**	
	Before	**After**	**Before**	**After**
MOV AX,@DATA				
MOV DS,AX				
MOV AL,NUM1				
ADD NUM2,AL				
MOV BL,C				
MOV AX,4C00H				
INT 21H				