Experiment N^{o} 1

1 Introduction to Assembly Language Programming

Introduction:

This experiment introduces the student to assembly language programming. In order to illustrate the basic concepts of assembly language programming a new environment (MASM) is used as a tool to develop assembly programs.

The following tools will be used:

- Microsoft Macro Assembler (MASM)
- CodeView (CV).
- Programmer's Work Bench (PWB)

Objectives:

In this experiment you will learn:

a. **MASM** tool and Code View (CV).
b. General structure of an assembly language program
c. Basic assembly instructions: MOV, INT
d. Use of the PWB environment

1.1 Program Writing

The process of writing an assembly language program is facilitated through the use of an **Assembler**. The Assembler allows the user to write alphanumeric instructions, or **mnemonics** called **instructions** and then generate the **machine code** from the Assembly Language instructions.

Assembly Language programming consists of the following steps:

a. Edit
b. Assemble
c. Link
d. Execute

1.1.1 Assembling

The assembler is used to convert the assembly language instructions into machine code. It is used immediately after writing the Assembly Language program. The assembler starts by checking the syntax, or validity of the structure, of each instruction in the source file. If any errors are found, the assembler displays a report on these errors along with a brief explanation of their nature. However, if the program does not contain any errors, the assembler produces an object file that has the same name as the original file but with the "obj" extension.

1.1.2 Linking

The linker is used to convert the object file to an executable file. The executable file is the final set of machine code instructions that can directly be executed by the microprocessor. It is different than the object file in the sense that it is self-contained and re-locatable. An object file may represent one segment of a long program. This segment can not operate by itself, and must be integrated with other object files representing the rest of the program, in order to produce the final self-contained executable file.

In addition to the executable file, the linker can also generate a special file called the map file. This file contains information about the start, end, and the length of the stack, code, and data segments. It also lists the entry point of the program.

1.1.3 Executing

The executable file contains the machine language code. It can be loaded in the RAM and be executed by the microprocessor simply by typing, from the DOS prompt, the name of the file followed by the Carriage Return Key (Enter Key). If the program produces an output on the screen, or a sequence of control signals to control a piece of hardware, the effect should be noticed almost instantly. However, if the program manipulates data in memory, nothing would seem to have happened as a result of executing the program.

1.2 Micro Soft Assembler

The assembler being used in this lab is called **MS-MASM.** MASM is an interactive means for assembling linking and debugging assembly language programs. Microsoft's Macro Assembler (MASM) is an integrated software package written by Microsoft Corporation for professional software developers. It consists of an editor, an assembler, a linker and a debugger (CodeView). The programmer's workbench (PWB) combines these four parts into a user-friendly programming environment with built-in on-line help.

The following table summarizes all the steps and commands used to edit, assemble link and run a program using MASM.

	Step	**Command**	**Produces**	**File Name**
1	Editing	**Edit**, use any editor	Source File	Filename.asm
2	Assembling	**Masm** Filename	Object File	Filename.obj
3	Linking	**Link** Filename	Executable File	Filename.exe (or *.com)
4	Executing	**Filename**	Program output	-

Table 1.1: Assembly Language Programming Phases

Note:
Steps 2 and 3 may be done in one single command: **ML filename.asm** to produce both Name.obj and Name.exe files.

1.2.1 MS-PWB:

The PWB is an environment that allows the user to define a project that may contain one or more files. Then, the user may select all the necessary assembling, linking, and debugging options for that project. Once these options are set, the user need not set them again for that project. The PWB allows the user to edit, assemble, run, or debug his program without leaving the PWB environment. It also allows the user to get help on any keyword by pointing to the keyword and pressing the F1 key.

1.3 CodeView

The CodeView (CV) is a useful utility that allows program tracing and monitoring the processor status while running a program. In this section we are going to learn how to use MASM and code view.

Before you start the steps below, write the program that appears in **Figure 1.1** using a text editor and save it as **prog1.asm** in the directory you created in the last lab.

```
; The following lines are just comments, they may be omitted,
; However, they are very useful.

; COE 205: Lab Exp. # 1      Program # 1
; Student Name:              Student ID:              Section:

TITLE "A simple program"
.MODEL SMALL
.STACK 32
.CODE
        MOV AX, 2000
        MOV BX, 2000H
        MOV CX, 10100110B
        MOV DX, -4567
        MOV AL, 'A'
        MOV AH, 'a'

        MOV AX, 4C00H
        INT 21H
END
```

Figure 1.1: First Program

After the program is saved, assemble it then link it. You should get and ***.exe** file in your directory.
To assemble the program, at the DOS prompt type:

D:worakrea\> MASM prog1

You should get the following output on your screen

```
Microsoft (R) MASM Compatibility Driver
Copyright (C) Microsoft Corp 1993.  All rights reserved.
 Invoking: ML.EXE /I. /Zm /c prog1.asm
Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993.  All rights reserved.
 Assembling: prog1.asm
```

Figure 1.2: Screen Output after Assembly Phase

A **prog1.obj** file is now created. Link your program now, the following appears on the screen.

```
Microsoft (R) Segmented Executable Linker  Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992.  All rights reserved.
Run File [prog1.exe]:
List File [nul.map]: prog1
Libraries [.lib]:
Definitions File [nul.def]:
LINK : warning L4038: program has no starting address
```

Figure 1.3: Screen Output after Link Phase

A **prog1.exe** file is now created. Notice, that because a name was given in front of the map file, a **prog1.map** file has also been created.

1.3.1 The MAP file

The Map file contains useful information about the executable program, the prog1.exe file. More details will be given concerning the MAP file in the coming labs.

```
Start  Stop   Length Name                   Class
00000H 00014H 00015H _TEXT                 CODE
00016H 00016H 00000H _DATA                 DATA
00020H 0003FH 00020H STACK                 STACK
Origin   Group
0001:0   DGROUP
```

Figure 1.4: MAP File

1.3.2 Using CodeView

You can run your program by typing the following at the DOS prompt:

```
D:worakrea\> MASM prog1
```

Since the program does not give any output, nothing is displayed on the screen. To debug our program and see its effect while running, another tool is used, the code view.

1. From the start menu, select Programs> Masm 611> Masm Prompt. You will get to the DOS screen.
2. Change the directory to the directory where your program resides

Z:\> cd COE205\LAB2

3. Use the following command to run code view: **cv prog1.exe** then press Enter.
4. You will get the following screen (Figure 1.2).

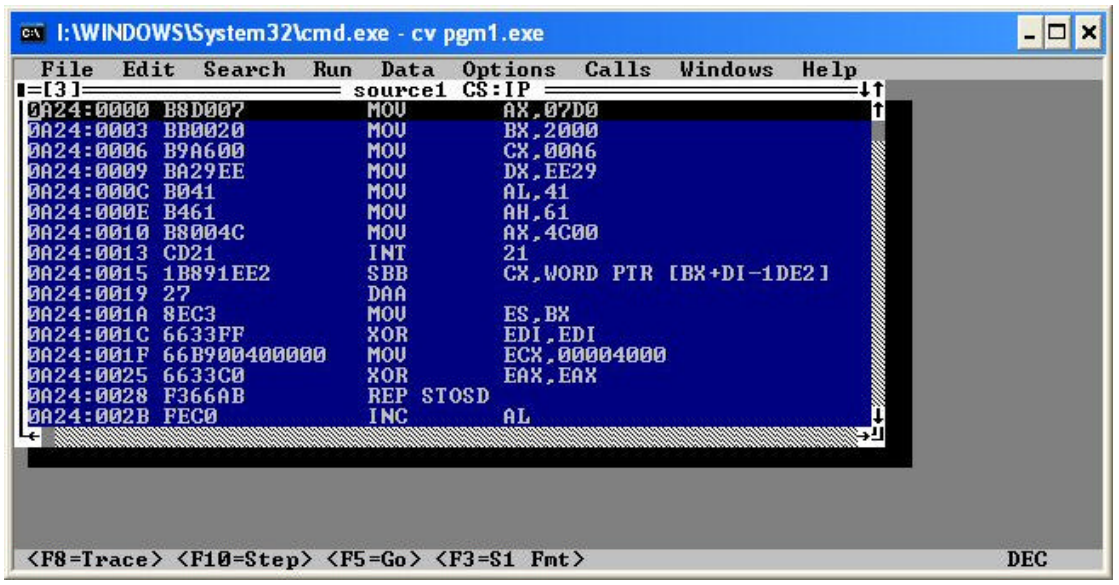

Figure 1.2: CodeView Main Screen.

Note:
The hexadecimal values and addresses in the code view windows may change when you run CV on different machines; this is due to the operating system memory management.

1.3.3 Instruction Address

The first two columns in the CV window show the address of each instruction of the program. The address is divided into two parts, *a segment number* and *an offset*. The whole address has the following format:

Segment Number: Offset

The segment number in this case represents the *code segment* (CS).

The instruction

0A24:0009 BA29EE MOV DX, EE29

is saved in the code segment number **0A24** at offset, or address, **0009**. The offset indicates a specific location within the segment.

Memory is divided into segments of size 64KByte each. Therefore, the number of bits required to address any location in memory is 16-bit (2 bytes).

The above instruction starts at address 0A24:0009 and occupies **3** bytes. Hence, it ends at address 0A24:000B. The following instruction starts at address 0A24:000C. All the instructions of a program are within the same segment.

1.3.4 Machine Code

The next column shows the machine code of each instruction. This is how the effectively instructions are saved inside the system memory. Generally, the first few bits or byte of the instruction correspond to the ***opcode***, which indicates the operation type (e.g. MOV, ADD) and ***the addressing mode***.

Example:

Machine Code	**Assembly Code**
B8D007	MOV AX, 07D0
B8004C	MOV AX, 4C00

Since both instructions have the effect of moving an immediate value into a word register, the high bytes in both instructions are identical (B8). The lower bytes carry the value to be loaded into the register. Notice that the values to be moved are written in reverse order.

Different instructions may have different lengths. To see that, compare the following two instructions.

Machine Code	**Assembly Code**	**Instruction size (Bytes)**
B8D007	MOV AX, 07D0	3
CD21	INT 21	2

1.3.5 Source Code

The last column on the main window shows the assembly language program source. Each instruction appears on one line. The instruction that is black-highlighted is the instruction that is going to be executed next. In figure 1.1 the instruction "MOV AX, 07D0" is the next instruction to be executed.

Recall from the first experiment, that an assembly language instruction may have zero, one or two operands. In case of a two-operand instruction, the right-hand operand is the **source** and the left-hand operand is the **destination**. In the instruction

MOV AX, 07D0

the value 07D0 is the source, the register AX is the destination.

1.3.6 Register Window

The registers window may be viewed from the option Windows and Registers, or by simultaneously clicking the keys **Alt** and **7.** This window shows the current values of the 14 internal registers. The default view is the 16-bit option for the 8086. To switch to the 32-bit register view, click the **Options** menu and select the **32-bit Registers** option.

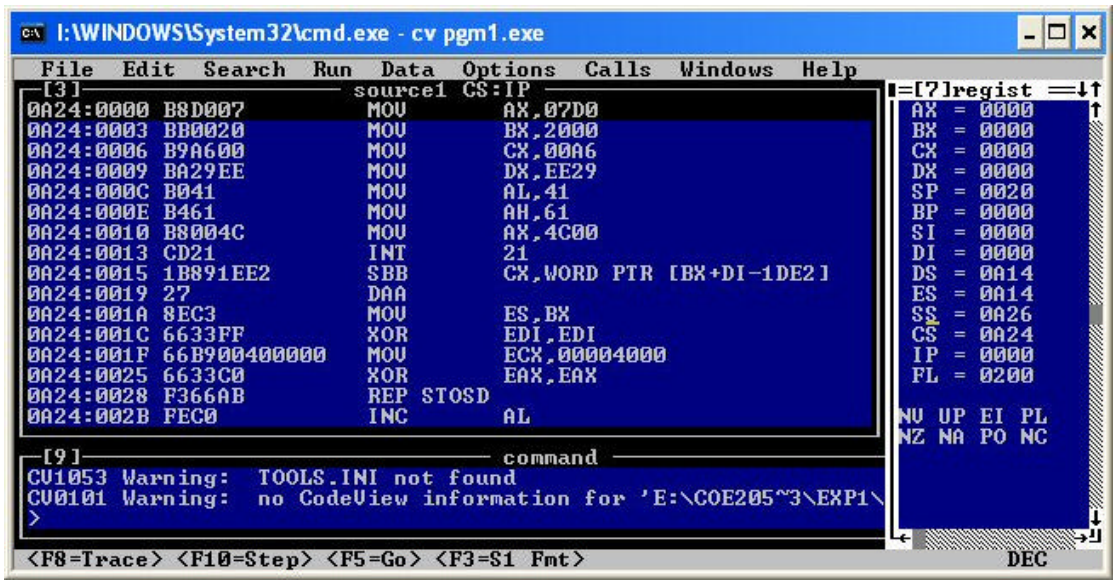

Figure 1.3: Registers Window

1.3.7 Flag Register Window

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 6.

Flag	Value	
	0	**1**
Overflow	**NV:** no overflow	**OV:** overflow
Direction	**UP:** up	**DN:** down
Interrupt	**EI**: Enable interrupt	**DI:** Disable Interrupts
Sign	**NG:** Negative	**PL:** positive
Zero	**ZR:** Zero	**NZ:** No zero
Auxiliary	**AC:** Auxiliary carry	**NA:** No Auxiliary carry
Parity	**PE:** Parity even	**PO:** Parity odd
Carry	**CY:** Carry	**NC:** No carry

Table 1.2: Flags and Their Values

1.4 Program Debugging

A program may be run as explained in section 1.1.3. The same can be done at this level using Code View. However, code view, in a way similar to the DOS debugger, can run the program in as a whole or one instruction at a time. This latter mode is used for debugging since it allows the user to see registers and memory contents while the program is running.

1.4.1 Running the Program

The whole program may be run as a whole by pressing **F5** (the option GO). However, the program should have a sequence of instructions to force it to stop, such as the one shown below, or a **breakpoint** (next section) inserted at a given instruction to force the program halt execution.

Note on the Exit function:
The following sequence of instructions has the effect of terminating the program and exiting to DOS.

```
MOV AH, 4CH
INT 21H
```

They should appear at the end of most, if not all, programs you will write.

1.4.2 Single Step through the Program

In order to run the program step by step both keys F8 and F10 may be used. To execute the next instruction press F8 or F10. The cursor will move one position down. Any change in register content is highlighted on the register window.

In the program above, the first instruction moves the value 07D0h into the register AX. After executing this instruction, the value of AX, initially equal to 0000h will change to 07D0h. It can be noticed that any value that changes is highlighted. After executing the previous instruction, notice that besides **AX** other registers have also been highlighted, namely **IP** and **FL**.

1.4.3 Breakpoints

A breakpoint is a point in the program, inserted by the programmer at debug time, to force the processor halt execution. To insert a break point, do the following:

1. Move the mouse at the point where you the breakpoint is to be inserted
2. From the Data menu select **Set Breakpoint**
3. or double click the instruction that you would like to stop at
4. Press F5 to run the program until the breakpoint

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

1.5 Pre Lab Work

1. Review the material related to introduction to assembly language programming and data representation
2. Write the source code given in figure 1.1 using the notepad or wordpad, then assemble and link the program using the MASM and Link commands.
3. Write the attached programs and bring them to the lab. Use the DOS editor or the Windows notepad. If you use a word processor, make sure that you chose the option *Save As Text* while saving

1.6 Lab Work

1.6.1 Part I

1- Assemble, Link and Run program 1.
2- Use code view to run your program.
3- Notice the values given by the assembler to the numbers you used in your program. Draw a table and write each value with its corresponding representation. What do you conclude?
4. Dress a table and write the values assigned by the assembler to the values you wrote in the editor. Write your conclusions.
5. Single step through the program and notice all the registers that are highlighted in the register window, meaning that the contents of those registers have changed. Pay more attention to registers **IP** and **FL**. What do you think the cause for that?

1.6.2 Part I

1. Now write a program that moves two 16-bit values in AX and BX registers. Chose the values so that their most significant bit is **1**. Add the two registers and make the destination the AX register.
2. Link and Run your program.
3. What value do you find in AX register?
4. What are the values of the Carry flag, Overflow flag and sign flag. Write your conclusions.
   Use table 1.2 to see the different values of the flags.

; The following lines are just comments, they may be omitted,
; However, they are very useful.

; COE 205: Lab Exp. # 1 Program # 1
; Student Name: Student ID: Section:

```
TITLE “A simple program”
.MODEL SMALL
.STACK 32
.CODE
       MOV AX, 2000
       MOV BX, 2000H
       MOV CX, 10100110B
       MOV DX, -4567
       MOV AL, 'A'
       MOV AH, 'a'

       MOV AX, 4C00H
       INT 21H
END
```

; COE 205: Lab Exp. # 1 Program # 2
; Student Name: Student ID: Section:

```
TITLE "Our second program"
.MODEL SMALL
.STACK 32
.DATA
      MULT1     EQU  25
      MULT2     DW    5
.CODE
      MOV AX, @DATA
      MOV DS, AX

      MOV AX, 00
      MOV BX, MULT1
      MOV CX, MULT2
MULT: ADD AX, BX
      DEC CX
      JNZ MULT
      MOV DX, AX

      MOV AX, 4C00H
      INT 21H
END
```