National University of computer and emerging sciences

Computer Organization and Assembly Language (EL-2003)

Semester: Fall 2025
Section: BCS-3B
Course Instructor: Muhammad Owais

LAB # 02

FUNDAMENTAL OF ASSEMBLY LANGUAGE

Lab Objectives:

By the end of this lab, students will be able to:

1. Understand the concept of debugging in Assembly Language and its necessity for error detection and correction.
2. Use breakpoints effectively to pause execution and analyze program flow.
3. Perform code stepping and single stepping to trace instructions one at a time.
4. Identify the basic elements of Assembly Language, including identifiers, mnemonics, instructions, directives, and comments.
5. Define and initialize data correctly in the data segment.
6. Distinguish between different intrinsic data types and understand their sizes and ranges.
7. Apply data definition statements (DB, DW, DD, etc.) to reserve memory and store values in Assembly programs.

DEBUGGING

We have already seen how to set up Visual Studio 2022 for Assembly Language and tested it with a sample program. In that program, the result appeared in the console window. However, it is often more useful to watch the program run step by step, line by line, using breakpoints. This makes it easier to understand what is happening inside the code.

What is Debugging in Assembly Language?

Debugging in Assembly Language is the process of finding, analyzing, and fixing mistakes (bugs) in a program written in assembly.

Since assembly language is low-level and directly controls the CPU and memory, even a small mistake (like using the wrong register or forgetting a flag) can make the program crash or produce wrong results. Debugging helps you check how each instruction affects the registers, memory, and flags so you can correct errors.

let's go over some important debugging terms in Visual Studio:

DEBUGGER:

The Visual Studio Debugger allows us to observe how a program behaves while it is running and helps us find errors. With the debugger, we can pause the program, inspect and even change variables, view register values, check the assembly instructions generated from our code, and look at the memory being used.

BREAKPOINT:

Breakpoint is a marker you place in your code to tell the debugger where to pause execution. When the program stops at a breakpoint, it enters what is called *break mode*. From there, you can carefully check what's happening at that exact point in the program.

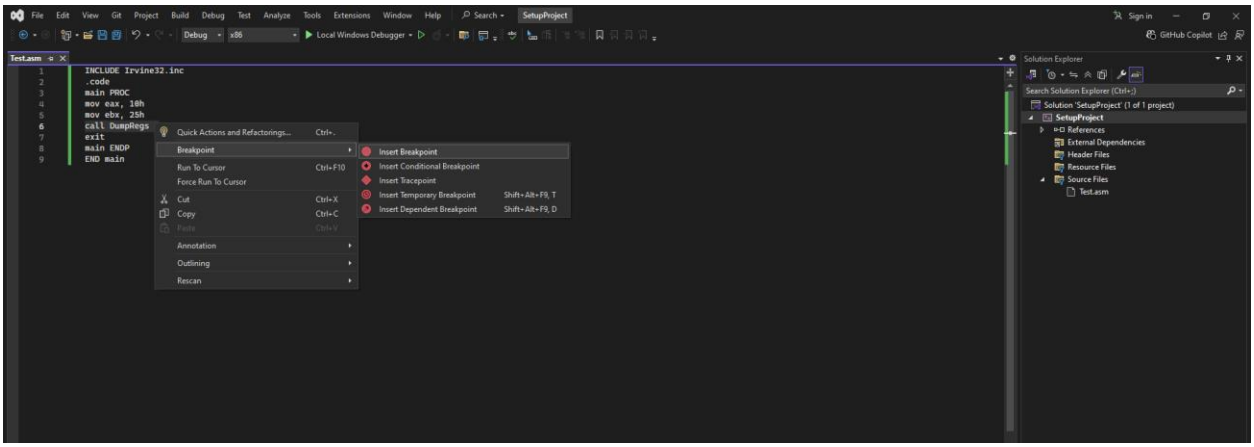

Figure 01: Break Points

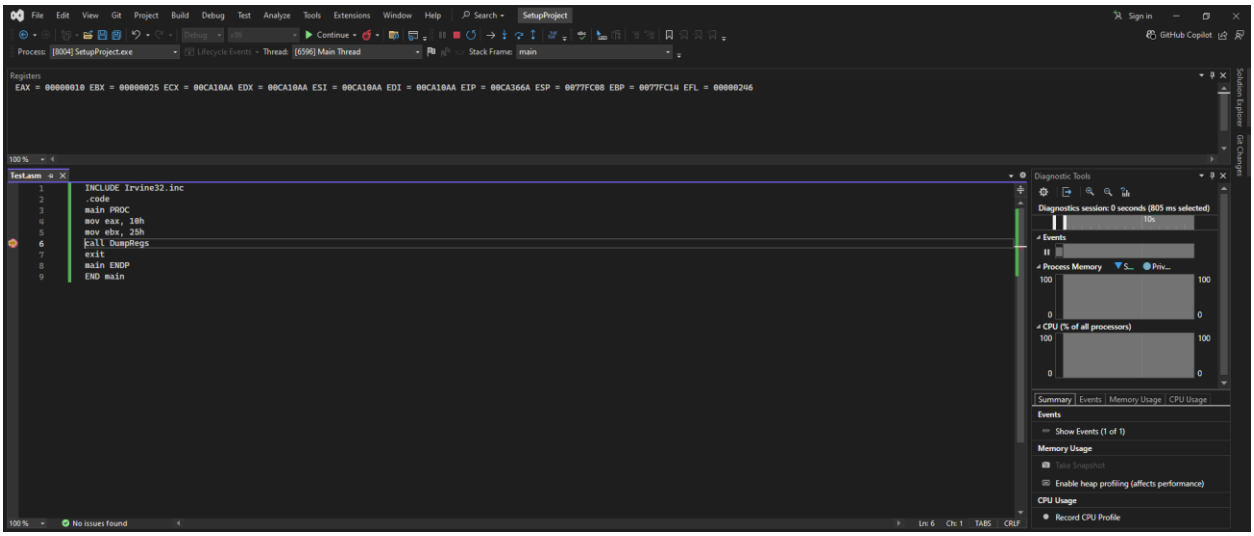

Figure 02: Observing Register Values

CODE STEPPING:

Stepping means running your program one line at a time. This is one of the most common debugging techniques. Visual Studio provides three useful commands for stepping:

- Step Into (F11): Runs the current line and goes inside any function or procedure call.
- Step Over (F10): Runs the current line but skips over function or procedure details.
- Step Out (Shift + F11): Runs the rest of the current function and pauses after it returns.

SINGLE STEPPING:

Single Stepping means running your program one instruction (or one line of code) at a time instead of running the whole program at once.

- Each time you "step," the debugger executes just the next instruction and then pauses again.
- This allows you to carefully watch how variables, registers, and memory values change after each step.
- It's especially useful in Assembly Language because you can see exactly how the CPU processes every instruction.

BASIC ELEMENT OF ASSEMBLY LANGUAGE

DIRECTIVES

Directives are command embedded in the source code that are executed by the assembler during the **compile time**. It is used to declare code, data section, select memory model, declare procedure, etc.

In MASM, directives are case insensitive. For example, it recognizes .data, .DATA and .Data as equivalent.

Example:

.data directive identifies the area of a program containing variables.

Anything inside .DATA is stored in the data segment.

.code directive identifies the area of a program containing executable instructions.

Marks the beginning of the code segment**.**

Note: Both are used with the example*(figure)* below.

INTEGER CONSTANTS

An Integer Constant is just a number that is written directly in the program and never changes. An Integer Constant consists of an optional sign (+ or –), followed by one or more digits, and may include an optional radix character to specify the number system.

Syntax: [+ , -] digits [radix]

Common radix characters:

- d – decimal
- h – hexadecimal
- b – binary
- o – octal

Examples:

40d , 6Ah , 52 , 1101b , +27

Hexadecimal beginning with letter: 0A5h

```
INCLUDE Irvine32.inc
.code
main PROC

mov eax, 25
mov ebx, -10
mov ecx, 101b

exit
main ENDP
END main
```

Figure 03: Integer Constants

IDENTIFIERS

An identifier is a programmer-defined name of a variable, procedure or code label.

- They may contain between 1 and 247 characters.
- They are not case sensitive.
- Must start with a letter, _, @, ? ,or $
- Can include letters (A-Z , a-z), digits, and some symbols like _.
- Cannot be the same as reserved words (like MOV, ADD, DB).

Examples: myVar , val_1

```
.data
num1 DWORD 10          ; identifier for first variable
num2 DWORD 20          ; identifier for second variable
sum  DWORD ?           ; identifier for result
```

Figure 04: Identifiers

CHARACTER & STRING CONSTANTS

Character constants are made up of a single character enclosed in either single or double quotes.

Example: 'A' "a"

A String of characters enclosed in either single or double quotes.

Example: "Hello World!"

INSTRUCTION

Instructions are command that are executed by the CPU during the runtime.

An instruction contains.

Label	(Optional)
Mnemonics	(Required)
Operand	(depend on the instructions)
Comment	(Optional)

Syntax: [label:] mnemonic [operand] [;comment]

COMMENTS

Comments are notes written by the programmer to explain code. They start with a semicolon `;`. The assembler ignores them.

Comments can be specified in two ways:

• **Single-line comments**: beginning with a semicolon character (;). All characters following the semicolon on the same line are ignored by the assembler.

Syntax: MOV AX, 5 **; Load 5 into AX register**

• **Block (multi line) Comments:** beginning with the COMMENT directive and a user-specified symbol.

Syntax:

COMMENT !

This line is a comment.

This line is also a comment.

!

Defining Data

Defining data means reserving space in memory for variables and assigning initial values if needed. It is done in the data segment using data definition statements.

Example:

```
myByte db 10        ; Define a byte with value 10
myWord dw 300       ; Define a word (2 bytes) with value 300
```

INTRINSIC DATA TYPES

In high-level languages (like C/Java), we have data types such as int, float, char, string. Same in Assembly language we have Intrinsic Data Types.

As we know that the CPU only understands Binary Data stored in memory (0s and 1s). To work with this data easily, Assembly provides Intrinsic Data Types.

The intrinsic data types in assembly language primarily define the size and interpretation of memory allocations. These types are essential for reserving storage space for variables and ensuring the correct handling of data by the processor.

- These are the basic data sizes that the assembler/CPU works with.
- They define how much memory a value needs.

Common Intrinsic Data Types are:

- **BYTE (or DB):**
  - Size: 1 byte = 8 bits
  - Can store small numbers (0–255) or a single character (like 'A').

- **WORD (or DW):**
  - Size: 2 bytes = 16 bits
  - Can store bigger numbers than BYTE (0–65,535).

- **DWORD (or DD):**
  - Size: 4 bytes = 32 bits
  - Can store large numbers (0–4,294,967,295).
  - Also used for 32-bit addresses or single-precision (normal) floating-point numbers.

- **QWORD (or DQ):**
  - Size: 8 bytes = 64 bits
  - Can store very large numbers, or double-precision floating-point numbers.

- **TBYTE (or DT):**
  - Represents an 80-bit value.
  - Often used for extended-precision floating-point numbers.

- **REAL4/REAL8/REAL10:**

  These are just friendly names for floating-point numbers of different sizes:

  - REAL4 4 bytes → single-precision.
  - REAL8 8 bytes → double-precision.
  - REAL10 10 bytes → extended-precision.

Signed Byte:

If we want to store signed numbers in only 1 byte. (i.e. + , -)

It is useful in math operations where both positive and negative values are possible.

- BYTE = 0 to 255 (unsigned)
- SBYTE = –128 to +127 (signed)

eg: SBYTE, SWORD etc.

DATA DEFINITION STATEMENT

In Assembly, before using variables, you must reserve space in memory. Data Definition Statements are assembler instructions that tell the assembler to allocate memory and optionally store an initial value.

Syntax:

```
.data

; Integer values

num1 db 10              ; 1 byte variable, value = 10

num2 dw 200             ; 2 bytes (word), value = 200

num3 dd 12345           ; 4 bytes, value = 12345

num4 dq 123456789   ; 8 bytes, value (large number)

; String example

msg db "Hello, World!", 0   ; string ending with 0 (null-terminated)
```