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 # 01

Getting Started with Assembly Programming Language (x86) in Visual Studio 2022 using Irvine Library

Lab Objectives:

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

1. Understand the core concepts of Assembly Language and its role in computer systems.
2. Recognize the importance and applications of Assembly in various domains.
3. Successfully install and configure Visual Studio 2022 with the Irvine Library for MASM programming.
4. Create, assemble, link, and execute a basic Assembly program.
5. Apply step-by-step setup techniques to ensure a smooth programming environment.
6. Troubleshoot and resolve common setup, compilation, and linking errors.
7. Utilize Visual Studio's debugging tools, including setting breakpoints, stepping through code, and observing register/memory states during execution.

Introduction to Assembly Programming Language

Assembly language is a type of programming language that is designed to be used by developers to write programs that can run directly on a computer's central processing unit (CPU). It is a low-level language, which means it is closer to the machine code the CPU can execute, making it more powerful than other higher-level languages such as C++, Java, or Python.

In an assembly language program, each instruction represents a single operation that the computer's CPU can perform. These can include simple arithmetic and logical operations, such as adding and subtracting values, as well as more complex operations that involve manipulating data stored in the computer's memory. Assembly language programs are typically written in a text editor and then assembled using a specialized software tool called an assembler.

Importance of Assembly Programming Language

1. Close to Hardware (Low-Level Control)

   Assembly language gives programmers direct access to a computer's hardware. Unlike high-level languages, it allows you to directly manipulate CPU registers, memory addresses, and I/O ports. This is essential when developing device drivers, firmware, or embedded systems**,** where you need precise control over the hardware.

2. High Performance and Efficiency

   Since assembly instructions are closely mapped to the CPU's native machine code, programs written in assembly run faster and are more memory-efficient**.** It is often used in situations where every CPU cycle matters, such as real-time systems, gaming engines, or robotics**.**

3. Essential for Embedded Systems

   Many embedded devices (like microcontrollers in washing machines, cars, or IoT devices) have very limited processing power and memory. Assembly is often used to write small, fast, and highly optimized programs for these devices.

4. Better Understanding of Computer Architecture

   Learning assembly helps you understand how the CPU, memory, and other components interact. You gain practical knowledge about instruction execution, registers, stack operations, and memory addressing, which is also useful for higher-level programming and debugging.

5. Optimizing High-Level Code

   Even if a program is written in a high-level language like C++ or Python, certain performance-critical sections can be optimized using inline assembly. This helps speed up tasks like graphics rendering, encryption, or scientific computation**.**

How Does Assembly Work?

The syntax of assembly language varies depending on the specific machine architecture it is being used with. However, most assembly languages share a few basic features. Assembly language programs are typically made up of a series of instructions written using a combination of mnemonic codes and operands, representing the data being manipulated by the instruction.

Let's understand how assembly language works in detail.

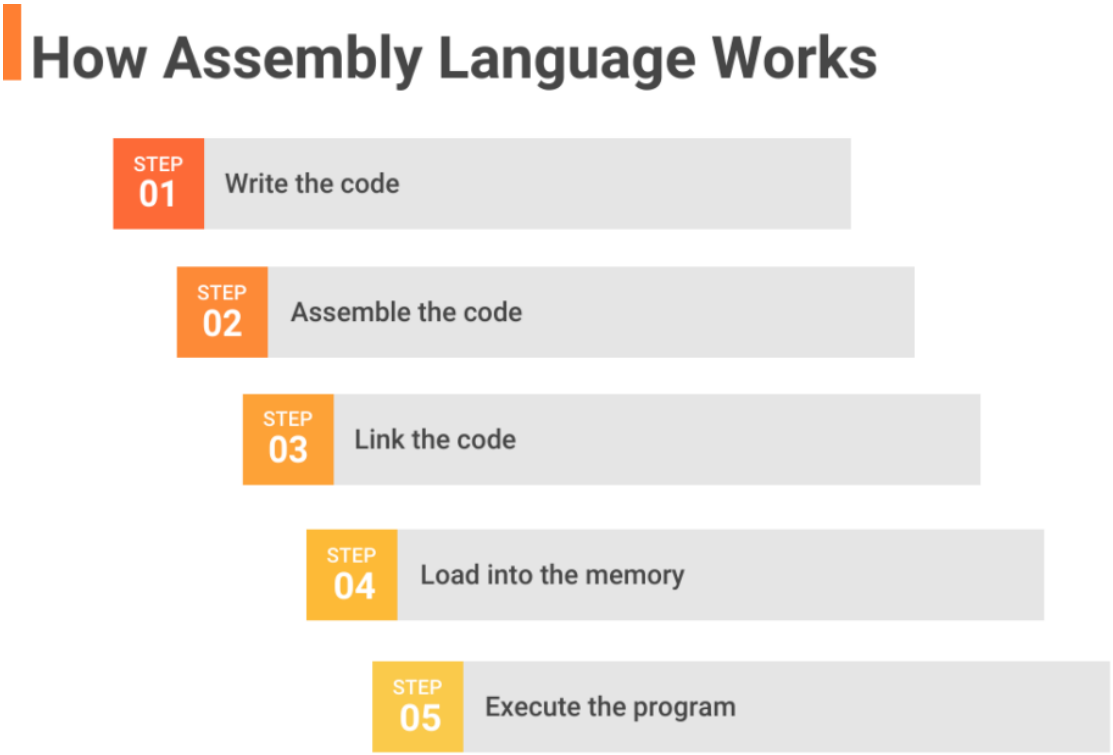

Figure 1: How Assembly Language Works

Step 1: Writing the code

The first step is to write the code in assembly language. Assembly language code consists of mnemonic instructions that correspond directly to the machine language instructions executed by the CPU.

For example, here is a code in x86 assembly language that adds two numbers:

```
mov eax, 5 ; move the value 5 into the abx register

mov ebx, 7 ; move the value 7 into the cdx register

add eax, ebx ; add the values in eax and ebx and store the result in eax
```

In this code, the first two lines set the values of the eax and ebx registers. The third line adds the values in the eax and ebx registers and stores the result in the eax register.

Step 2: Assembling the code

The next step is to assemble the code using an assembler. An assembler is a program that converts the assembly language code into machine language the computer can execute. The assembler reads the assembly code and translates it into binary machine code, a series of 0s and 1s representing the instructions and data in the program.

When the assembler converts the assembly code into machine code, it generates a file with an .obj extension containing the machine code and other information that the linker uses to create the final executable file.

Step 3: Linking the code

The next step is to link the code. Linking is the process of combining the object file generated by the assembler with any necessary system libraries to create an executable program. During the linking process, the linker resolves any external references to functions or variables and combines all the object files into a single executable file.

Step 4: Load into the memory

Once the code has been linked, it needs to be loaded into memory. The operating system accomplishes this. The executable file is loaded into a specific location in memory, and the operating system sets up the program's environment, including its stack, heap, and global variables.

Step 5: Executing the program

The final step is to execute the program. When the CPU reads the instructions, it will execute them one by one. The machine code instructions generated by the assembler correspond directly to the assembly language instructions in the original code.

Visual Studio

Visual Studio is an integrated development environment (IDE) created by Microsoft. It's basically a powerful software application that provides all the tools you need to write, edit, debug, and manage code for different types of applications.

For this course, we will use Visual Studio version 2022 to develop programs in Assembly Language. We could, however, use a stand-alone assembler like NASM or MASM to code in Assembly Language.

Steps to Install Visual Studio 2022

Step 1: Go to the Download Page

1. Open your web browser and go to: https://visualstudio.microsoft.com/downloads/
2. Scroll down to find Visual Studio 2022
3. Click Download for the Community Version.

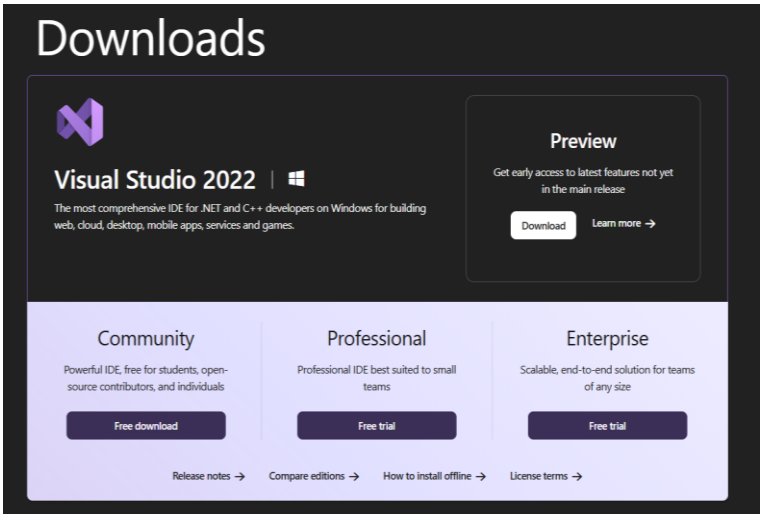

Figure 2: Downloading Page - Visual Studio

Step 2: Download the Bootstrapper File

1. Once you click Download, a small setup file (bootstrapper) will be downloaded.
2. The file name will be something like:

    - VisualStudioSetup.exe
    - or vs_community.exe

Step 3: Initiate the Installation

1. Go to your Downloads folder.
2. Double-click the downloaded bootstrapper file.
3. If prompted by User Account Control, click Yes.
4. Read the Microsoft License Terms and the Privacy Statement, then click Continue.

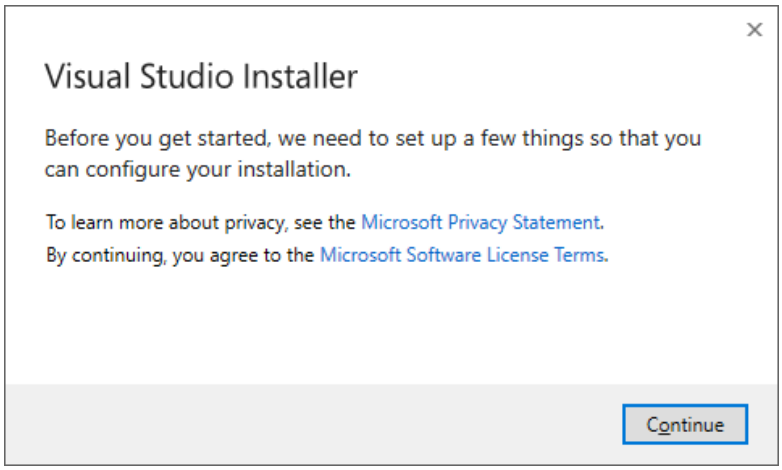

Figure 3: Visual Studio Installer

Step 4: Choose Workloads

1. The Visual Studio Installer will open.
2. From the list of workloads, select Desktop development with C++ (mandatory for C++ development).
3. Review workload summaries to confirm the features you need.
4. After selecting workloads, click Install.

Step 5: Installation Progress

1. You will see progress bars showing the installation status.
2. Wait until the process completes; this may take several minutes depending on your internet speed and selected workloads.

Step 6: Launch Visual Studio 2022

1. Once the installation is complete, click Launch from the installer.
2. Sign in with your Microsoft account (optional, but recommended).
3. Choose your preferred development settings and theme.

Visual Studio 2022 is now ready to use!

Introduction to the Irvine Library

The Irvine Library (formally called *Irvine32.inc*) is a collection of procedures, macros, and definitions created by Kip Irvine to make it easier for students and beginners to learn Assembly Language Programming (specifically x86 Assembly on Windows). It comes with the textbook *Assembly Language for x86 Processors* by Kip Irvine.

The library provides functions for:

- Displaying strings and numbers
- Reading input from the keyboard
- File handling
- Random number generation
- Simple screen formatting

Why Do We Use It?

1. Simplifies Learning
   Assembly is very low-level, and even printing "Hello World" requires many lines of code (using system calls or interrupts). Irvine library makes this simple with easy-to-use procedures. Example:

```
INCLUDE Irvine32.inc

.code
main PROC
   mov edx, OFFSET msg   ; point to string
   call WriteString      ; print string
   exit
main ENDP

.data
msg BYTE "Hello, World!", 0
END main
```

Without the library, you'd need to use int 21h (DOS) or Windows API calls, which are much harder for beginners.

2. Focus on Concepts
   Instead of spending too much time on low-level OS calls, students can concentrate on assembly instructions, registers, stack, loops, and procedures.
3. Cross-Platform Learning (within Windows)
   It hides OS-specific details (like system calls), so the same simple code runs on any Windows system with MASM + Irvine library.

Click here to download Irvine library from this
link:www.asmirvine.com/gettingStartedVS2019/Irvine.zip

Once you have downloaded the required Irvine library, install it in your computer and verify that a folder named Irvine has been created in your C:\ drive.

Configuring Irvine32 Library with Visual Studio

Follow the steps below to create and configure a Visual Studio project for Assembly Language using the Irvine Library:

Step 1: Create a New Project

1. Open Visual Studio 2022.
2. Click on Create a new project.
3. In the template search bar, type Empty Project and select it.
4. Click Next, give your project a name, and click Create.

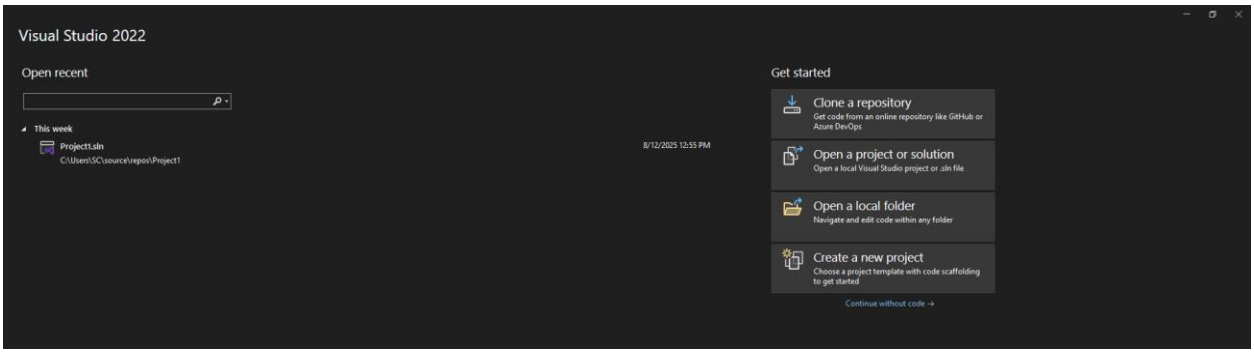

Figure 4: Creating a New Project

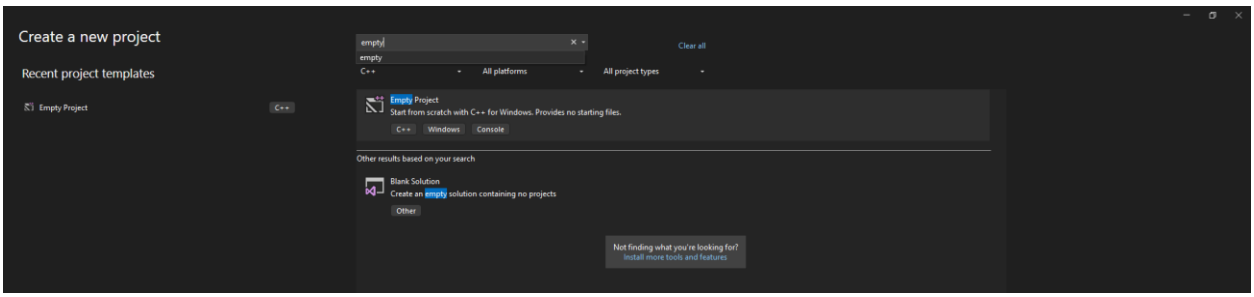

Figure 5: Template Search Bar

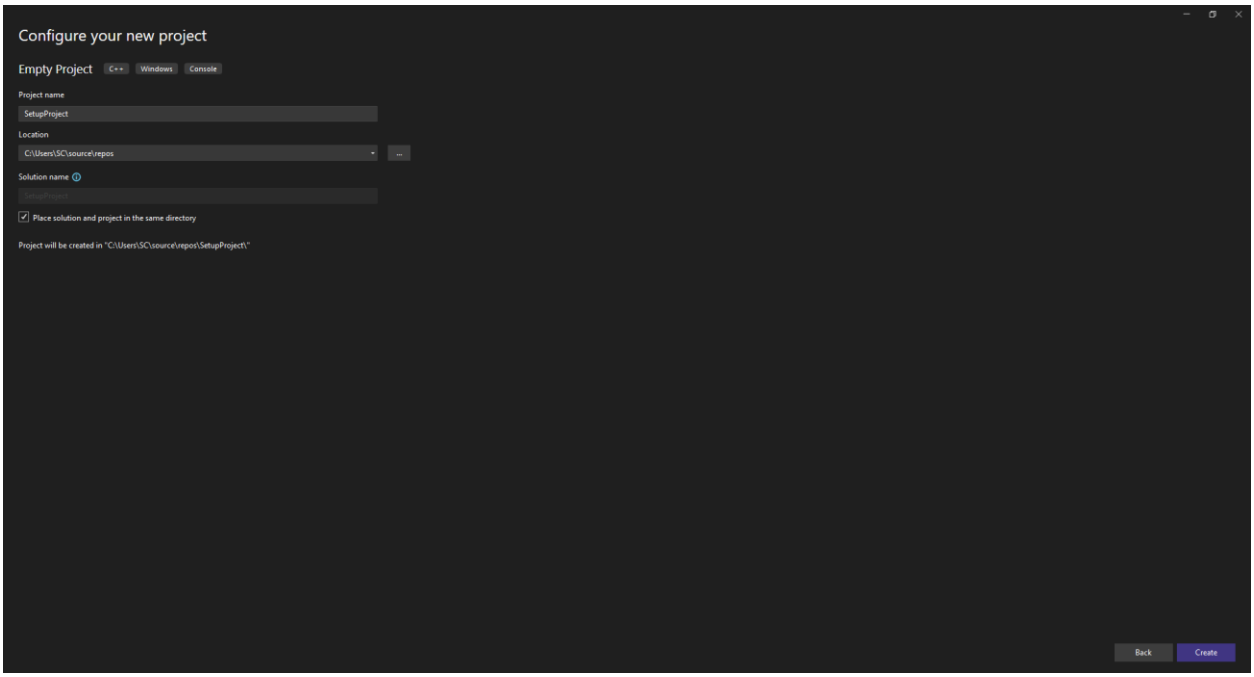

Figure 6: Project Name

Step 2: Add an Assembly File

1. In Solution Explorer, right-click on Source Files → Add → New Item.
2. Choose Text File, name it with a .asm extension (e.g., program.asm), and click Add.

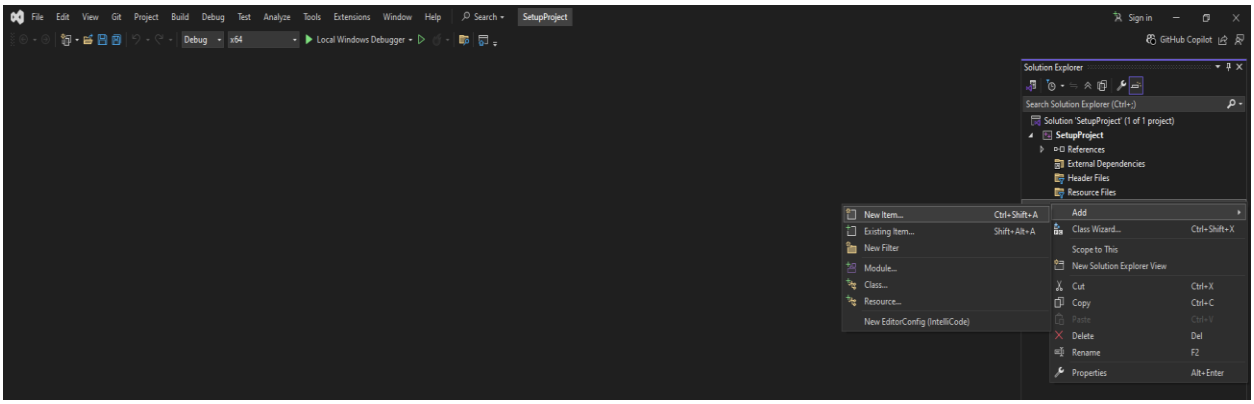

Figure 7: Inserting New Item

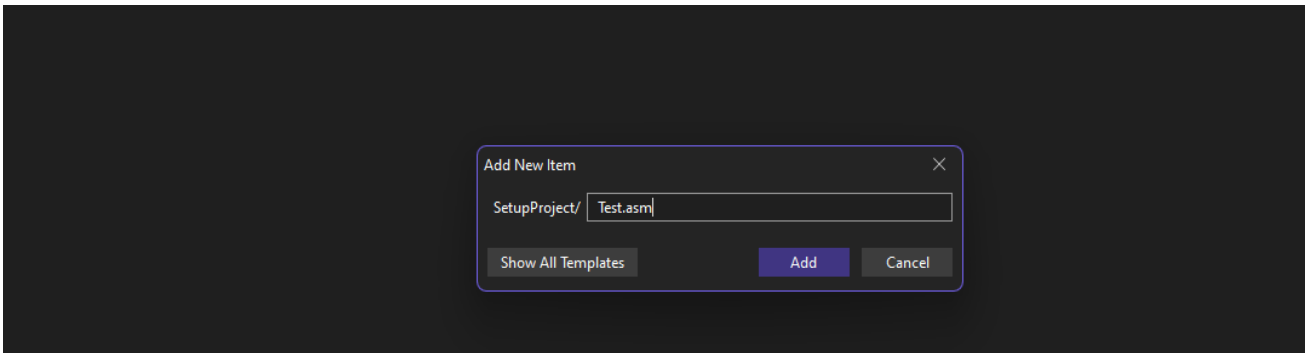

Figure 8: Renaming a File Name with .asm extension

Step 3: Enable MASM (Microsoft Macro Assembler)

1. In Solution Explorer, right-click on your Project Name.
2. Select Build Dependencies → Build Customizations.
3. Tick the checkbox for masm (.targets, .props) and click OK.

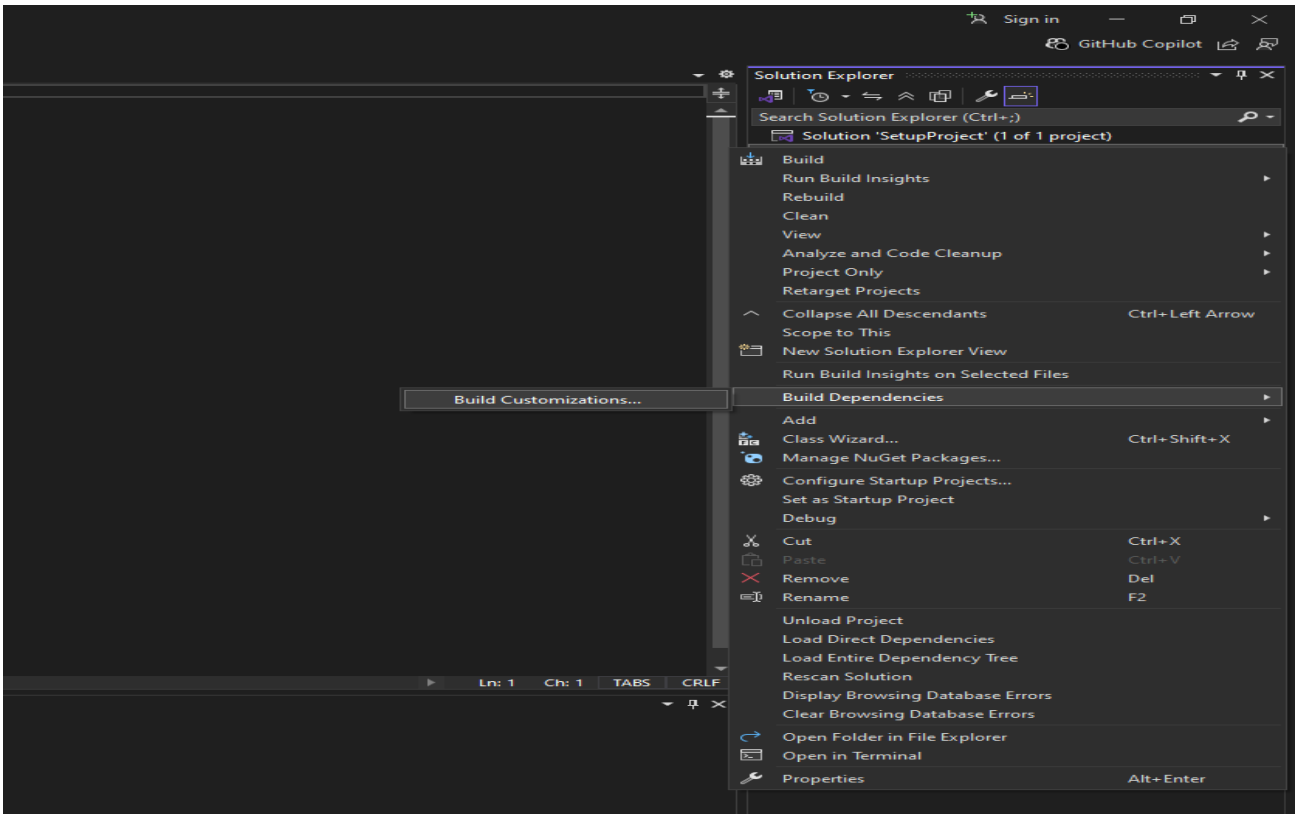

Figure 9: Build Dependencies

Figure 10: Tick MASM

Step 4: Configure Library Path

1. Copy the path where the Irvine Library is installed (e.g., `C:\Irvine`).
2. Right-click on your Project Name → select Properties.
3. Go to Linker → General → Additional Library Directories.
4. Click Edit, paste the copied Irvine Library path, and click OK.

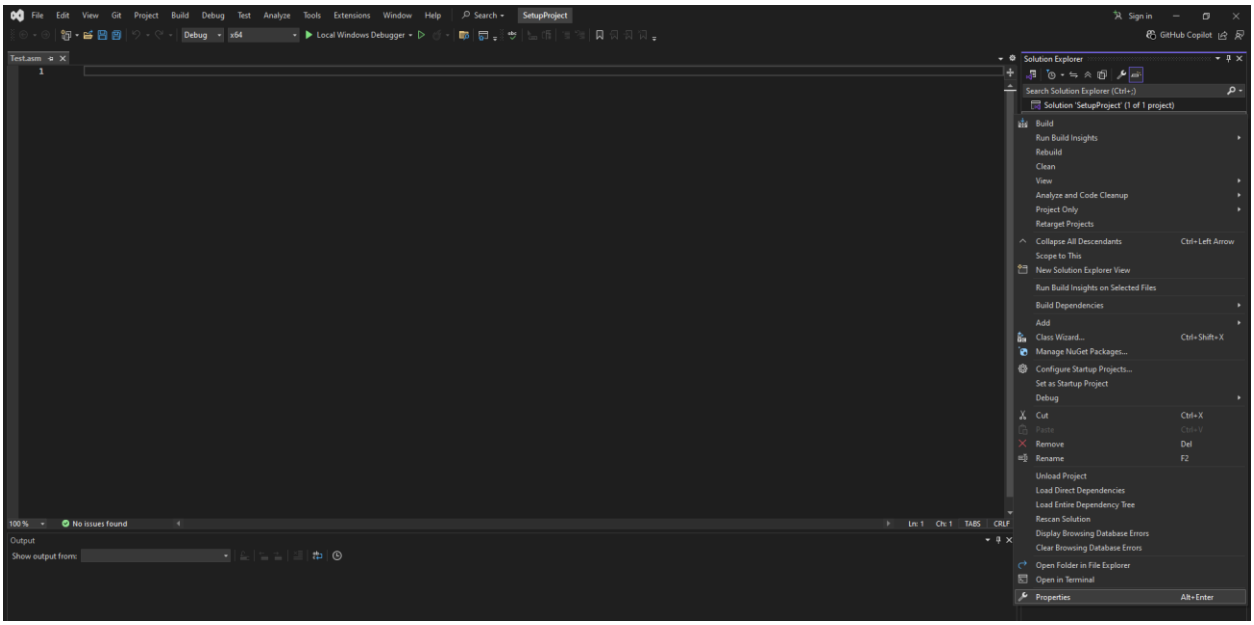

Figure 11: Go to Properties

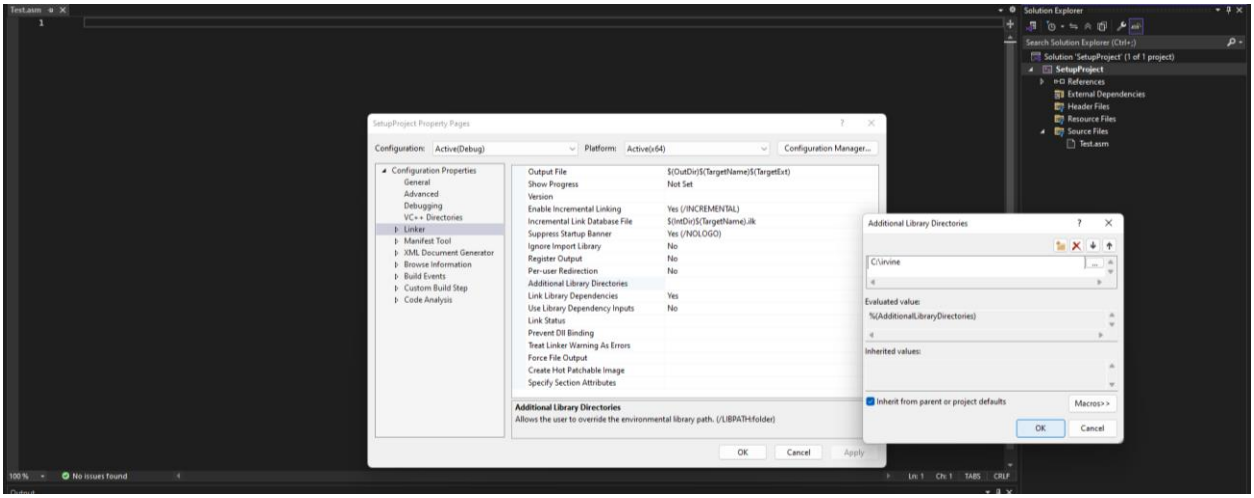

Figure 12: Copying Irvine Library Path

Step 5: Configure Linker Input

1. In the same Properties window, go to Linker → Input.
2. In Additional Dependencies, add:
3. Irvine32.lib
4. Click Apply and then OK.

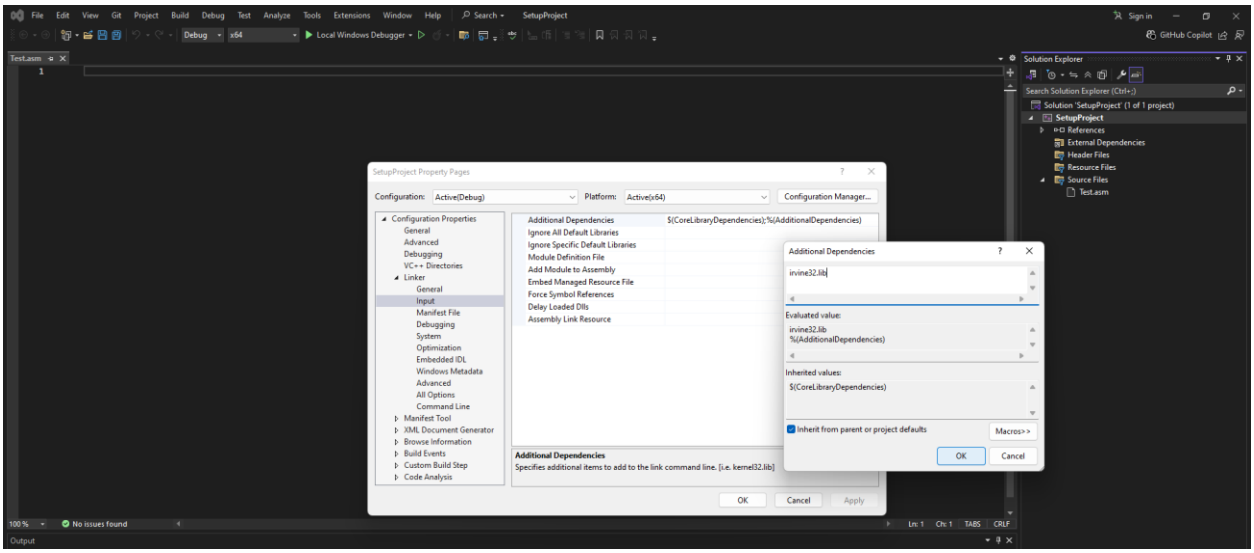

Figure 13: Configure Linker Input

Step 6: Configure ASM File Properties

1. Right-click your `.asm` file in Solution Explorer → Properties.
2. Under General, set Item Type to Microsoft Macro Assembler.
3. Apply the changes.

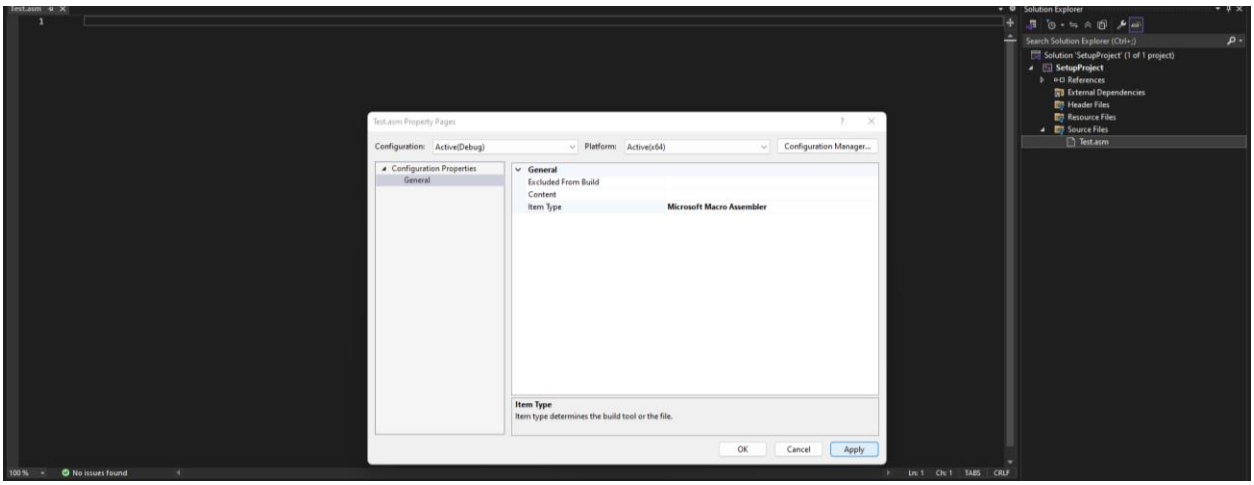

Figure 14: Configure ASM File Properties

Step 7: Set Include Path

1. In the same properties window, go to Microsoft Macro Assembler → General.
2. In Include Paths, paste the path of the Irvine library (e.g., `C:\Irvine`).
3. Click Apply and then OK.

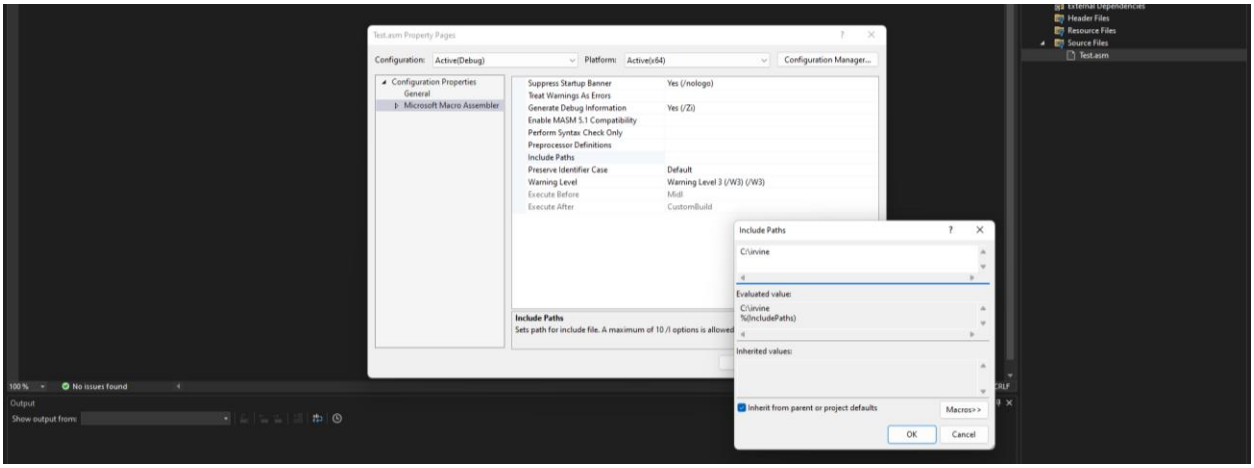

Figure 15: Set Include Path

Step 8: Write and Run a Sample Program

1. Copy a sample Irvine program (e.g., Hello World).
2. Paste it inside your `.asm` file.
3. Press F5 to build and run the program in the debugger.

Example – 01

```
TITLE MyFirstProgram (Test.asm)

INCLUDE Irvine32.inc

.code

main PROC

mov eax, 10h

mov ebx, 25h

call DumpRegs

exit

main ENDP

END main
```

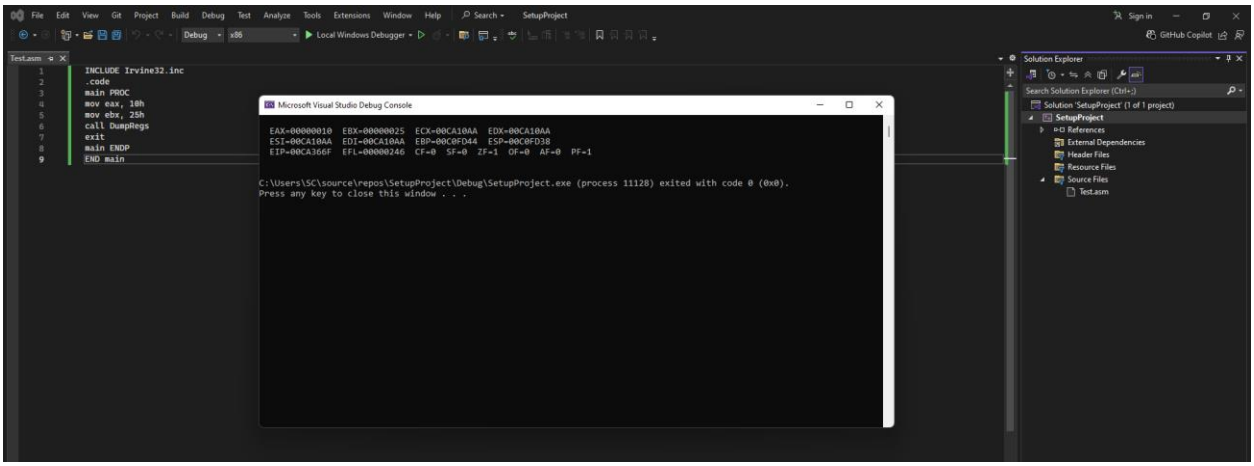

Figure 16: Example 01

Example 01 – Program Explanation

1. The first line TITLE MyFirstProgram (Test.asm) gives an optional title to our program.
2. The second line INCLUDE irvine32.inc adds a reference to the include file that links your program to the Irvine library.
3. The third line .code defines the beginning of the code segment (to be covered in detail later). The code segment is the segment of memory where all your code resides.
4. In the fourth line, a main procedure is defined.
5. The fifth and sixth lines show a mnemonic mov (to be covered in detail later) that 'moves' values 10h and 25h to eax and ebx, respectively. The radix h defines a hexadecimal constant. The lines seven and eight calls the procedure DumpRegs that outputs the current values of the registers followed by a call to windows procedure named exit that halts the program.
6. The lines nine and ten mark the end of the main procedure.

Important Note:

The Irvine Library works only with x86 (32-bit) projects, not x64.

To check this setting:

1. Go to Build → Configuration Manager.
2. Under Active Solution Platform, if it shows x64, change it to x86.

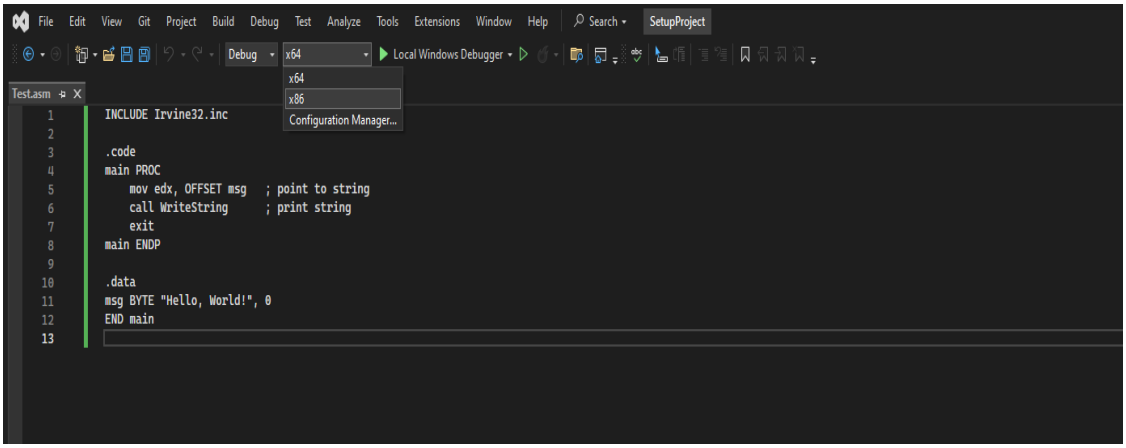

Figure 17: Configuration Manager Settings

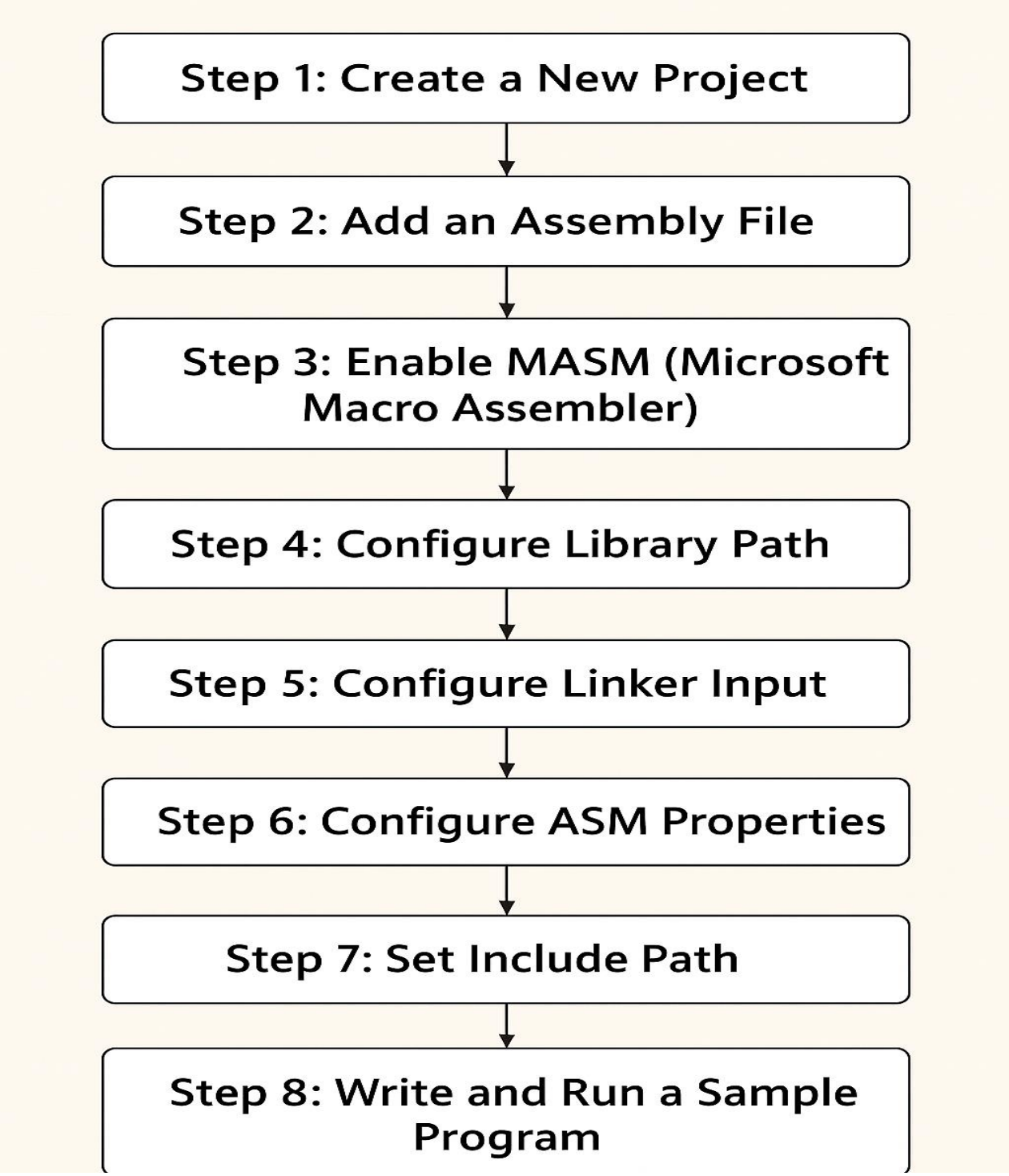

Figure 18: Assembly Project Setup Steps

Debugging our Program

We have already seen how to configure Visual Studio 2022 for Assembly Language and tested it with a sample program. The output of our sample program appeared in the console window. However, in most cases, it is more useful to watch the step-by-step execution of our program by using breakpoints.

In this section, we will first define important debugging terms in Visual Studio, and then cover an example for better understanding.

Debugger

The Visual Studio Debugger allows us to observe the run-time behavior of our program and locate problems. Using the debugger, we can:

- Pause execution at specific points.
- Inspect and edit variables.
- View processor registers.
- Analyze instructions generated from our source code.
- Examine the memory space used by our application.

Breakpoint

A breakpoint is a marker that tells the debugger to temporarily suspend execution of your program at a specified line of code.

- When the program halts at a breakpoint, it enters break mode, allowing you to carefully examine the state of your program.

Code Stepping

Stepping is one of the most common debugging techniques. It allows you to execute code one line at a time, making it easier to understand program flow and catch logical errors.

Visual Studio provides three main stepping commands under the Debug menu:

- Step Into → Executes the current line of code and enters any function calls inside it. *(Shortcut: F11)*
- Step Over → Executes the current line but skips over function calls. *(Shortcut: F10)*

- Step Out → Executes the rest of the current function and returns to the caller. *(Shortcut: Shift + F11)*

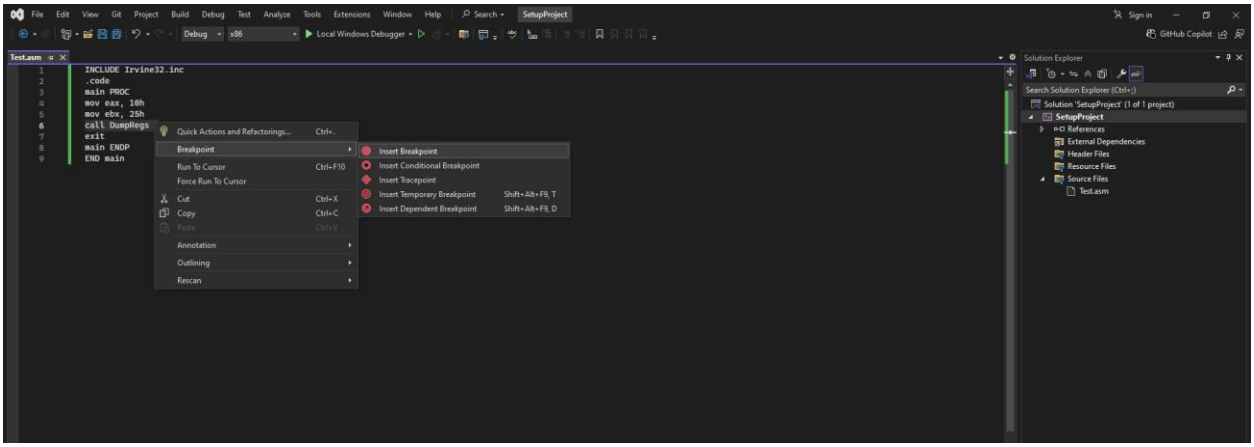

Figure 19: Break Points

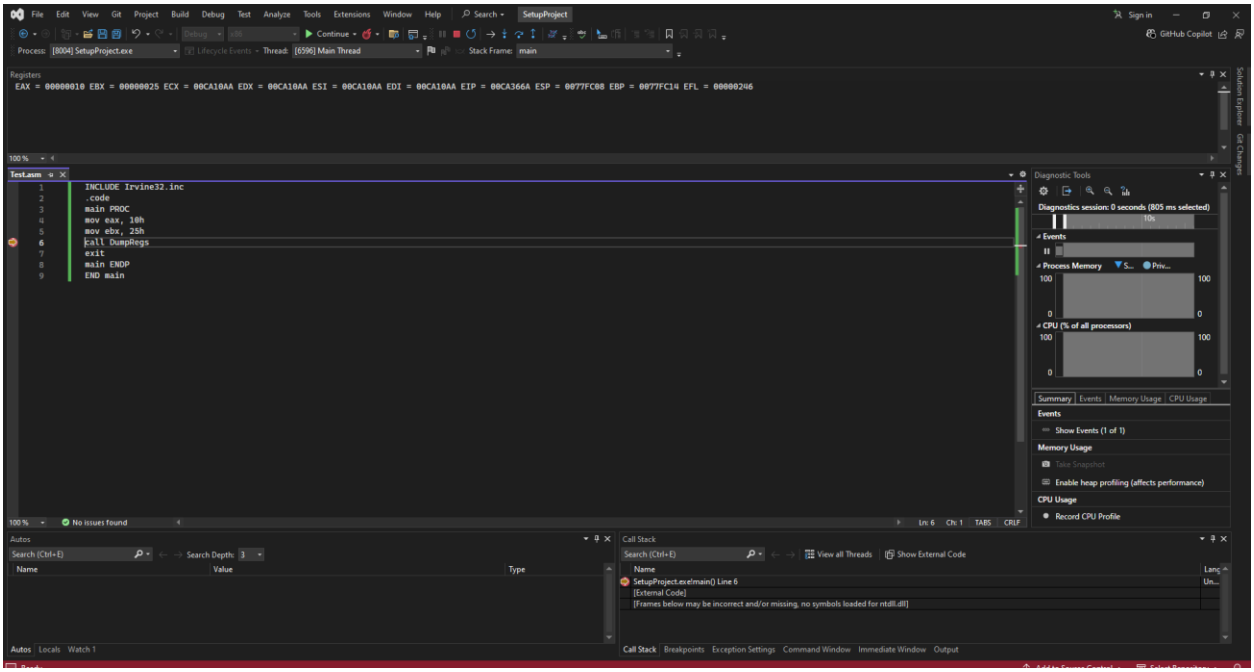

Figure 20: Break Points - Observing Register Values

Exercise

Task # 01:

Use the lab manual instructions to install Visual Studio Community Edition and set up a new Empty Project in Visual C++.

Task # 02:

- Copy the code written below and paste into the project you created by following instructions mentioned in the lab manual.
- Modify the program by replacing the 'Hello World!' message with your own Name and ID, and then execute the code.

```
INCLUDE Irvine32.inc
.code
main PROC
   mov edx, OFFSET msg
   call WriteString
   exit
main ENDP
.data
msg BYTE "Hello, World!", 0
END main
```

WHAT YOU NEED TO SUBMIT

A screenshot of the console window Displaying your Full Name and ID