Introduction to Assembly Programming

Objectives.

- Setting up a template for writing assembly programs.
- Examine a piece of code that adds two variables together.
- Implement a simple assembly program covering the following topics:
  - Defining different types of variables.
  - Performing integer arithmetic: addition, subtraction.

Aim

The aim of this session is to learn how to write simple assembly programs

Introduction

Assembly language is a low-level programming language for a particular computer architecture. Each processor family has its own instruction set (assembly instructions). In contrast to most high-level programming languages, which are generally portable across multiple systems, assembly is not. Assembly language is converted into executable code (binary) through an assembler. Keep in mind that a processor runs only binary code. Assembly language is the lowest-level programming language that we can write programs. Most of the assembly commands (but not all) have an 1-to-1 correspondence to machine code (binary instructions). The assembly programs have a file extension .asm.

Creating a template for assembly programs.

In this task you will set up a template for your future assembly code. You would need Visual Studio 2019 for this task. Please follow the instructions below to create the template.

1. Start visual studio, and create a new project (you will see a window like below). Then click empty project (C++).

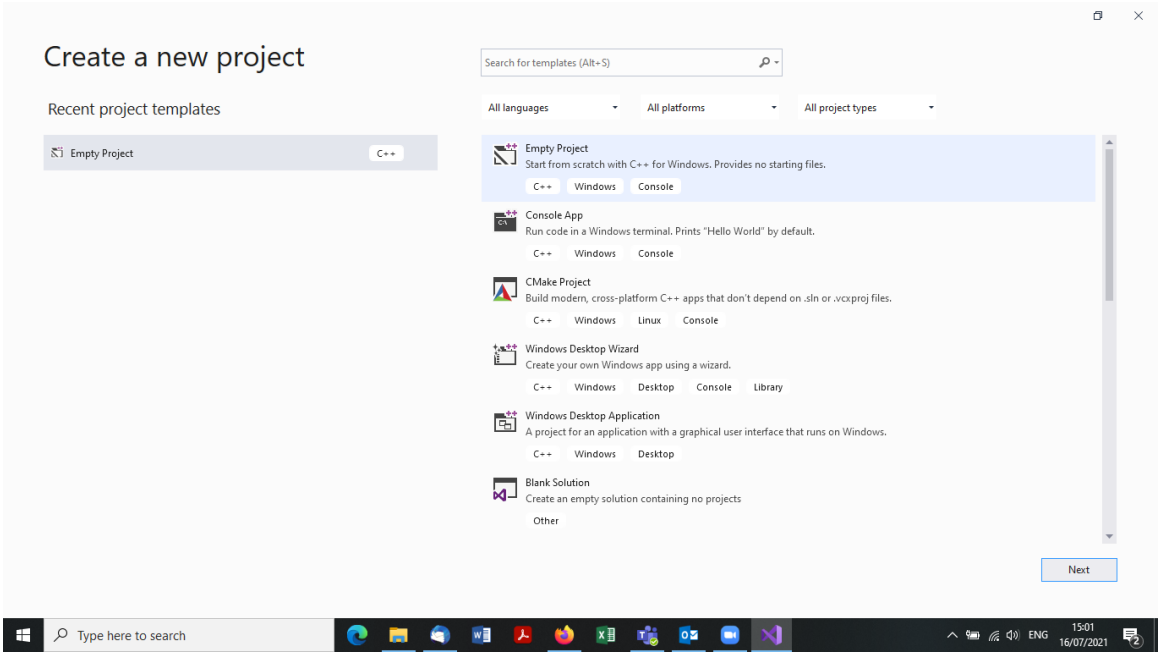

Fig.1 How to create a template for assembly programs

2. Name your project as MASM, and click OK.

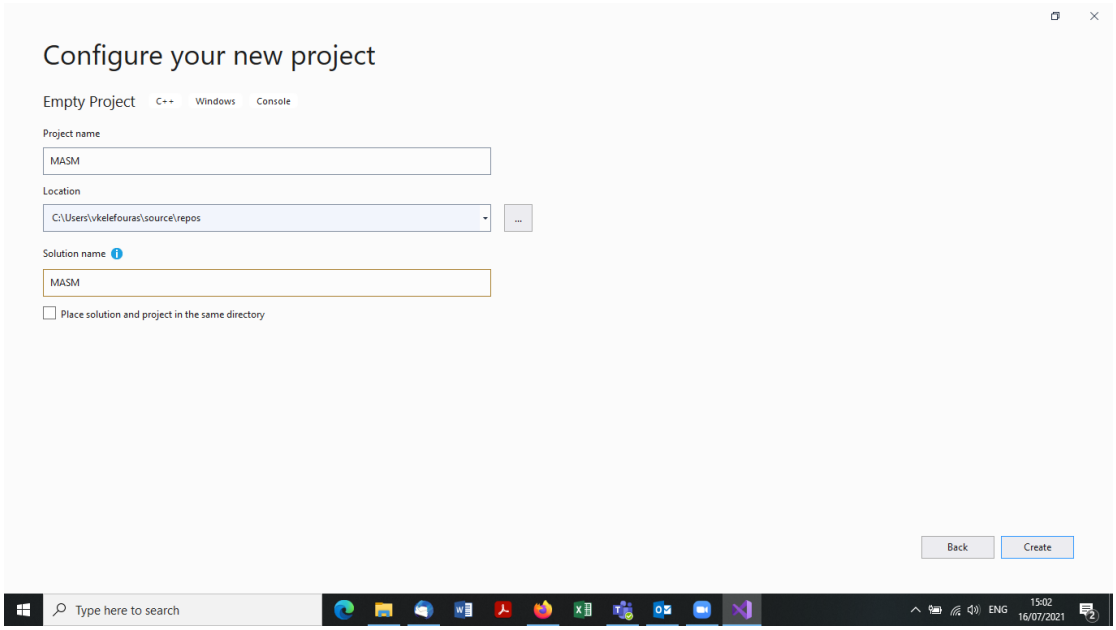

Fig.2 How to create a template for assembly programs

3. You now have a project called MASM. We will delete the following directories (highlighted in blue in the figure below): Header Files, Resource Files and Source Files.

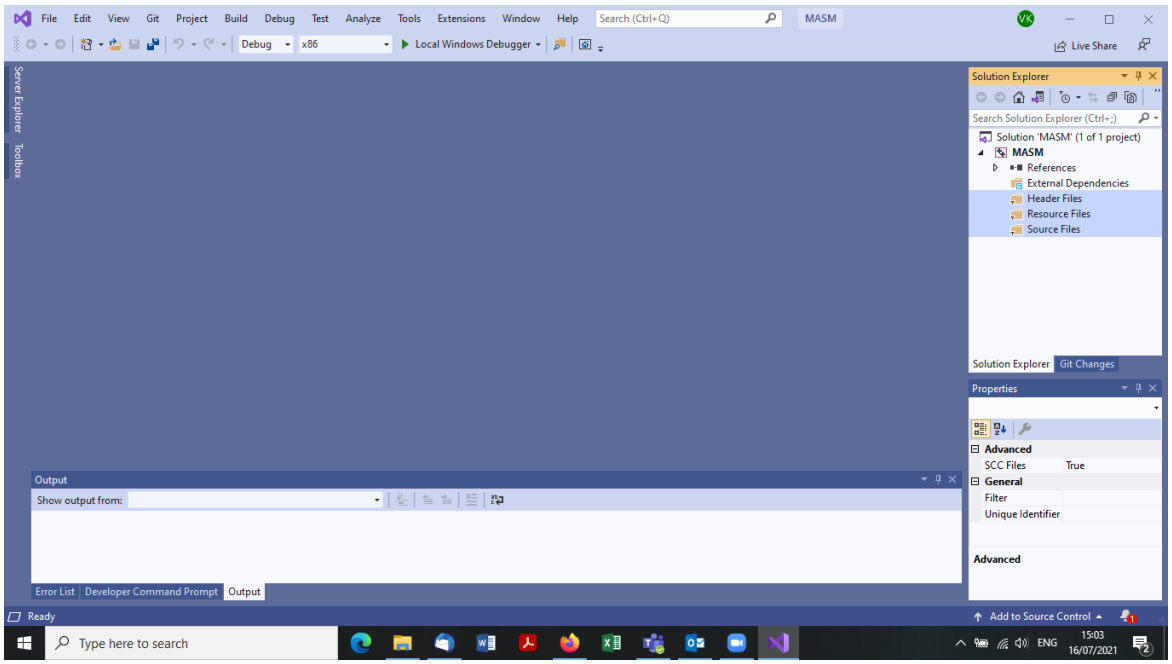

Fig.3 How to create a template for assembly programs

4. Setting up build dependencies: right click on MASM, then go to **Build Dependencies > Build Customizations**. Check the box that **masm** and click OK.

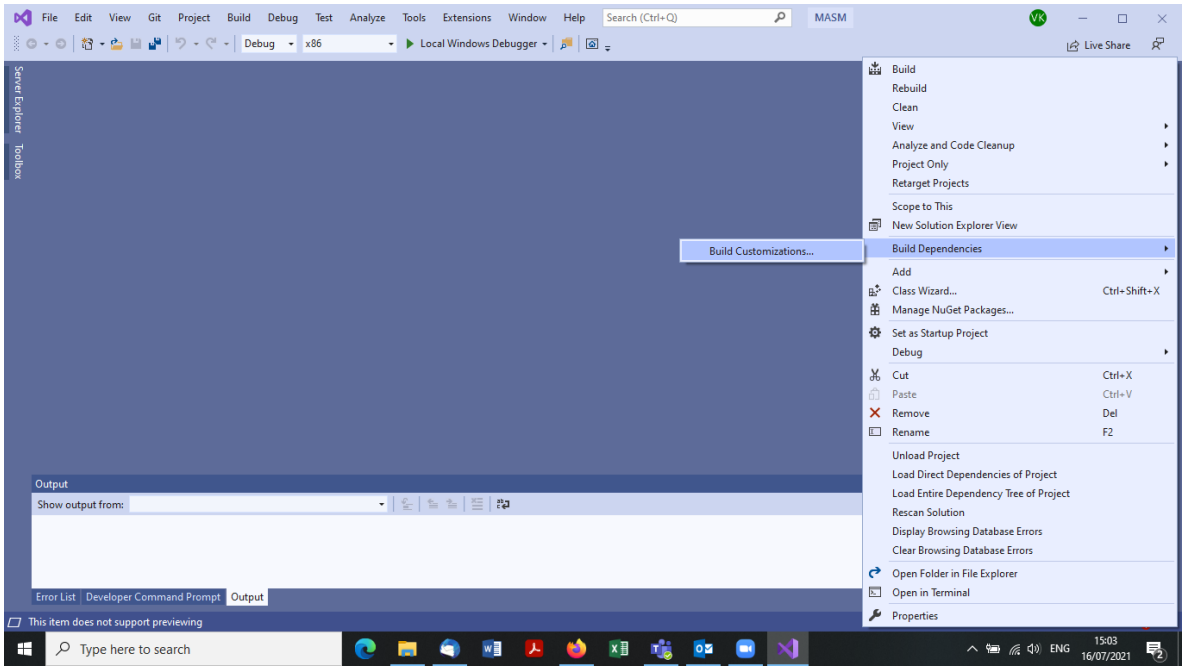

Fig.4 How to create a template for assembly programs

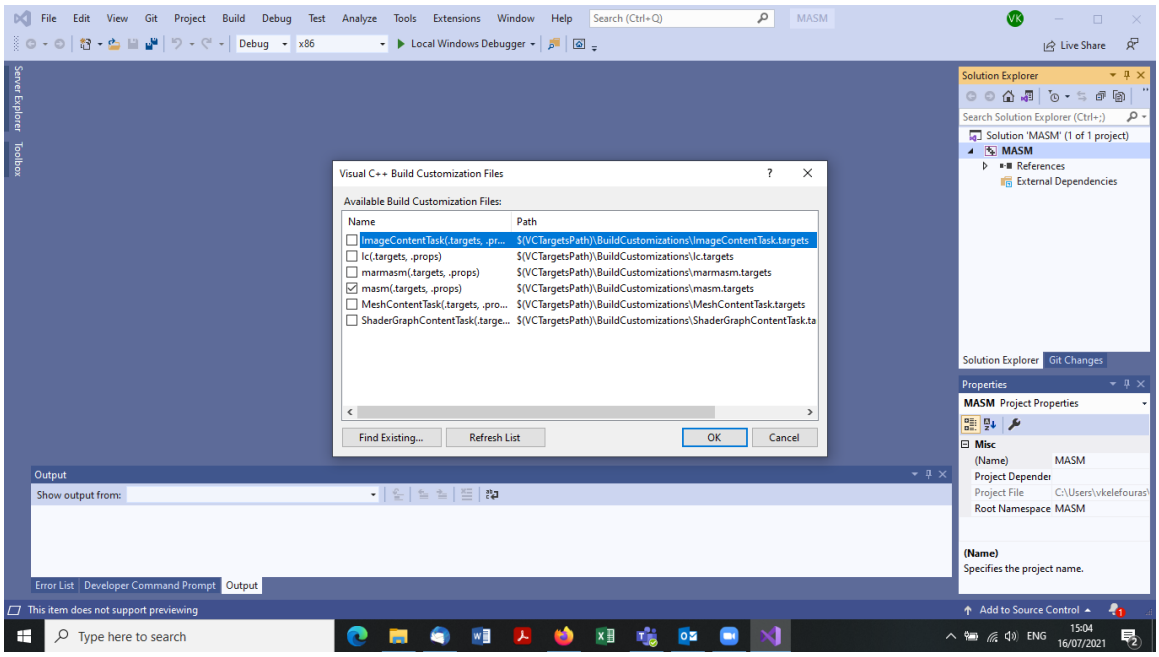

Fig.5 How to create a template for assembly programs

5. Set up the linker subsystem: right click on MASM and then click properties.

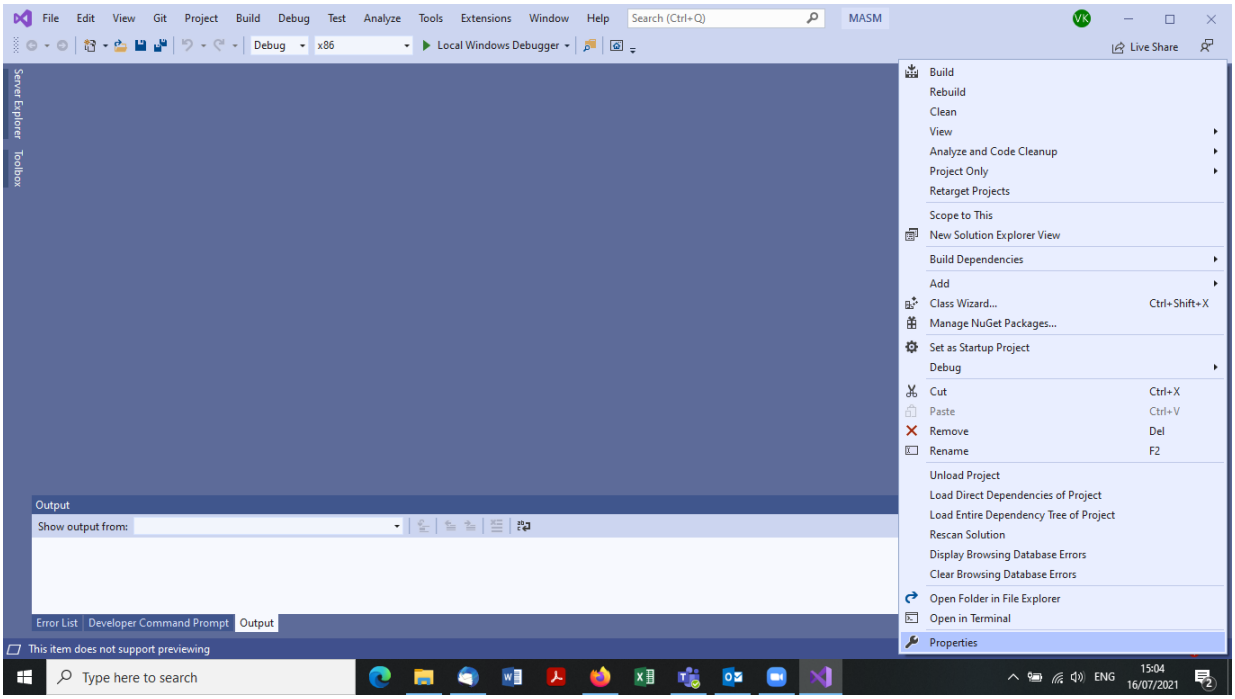

Fig.6 How to create a template for assembly programs

6. Then go to **Linker > System > SubSystem** and select **Windows (\SUBSYSTEM:WINDOWS).** Press OK.

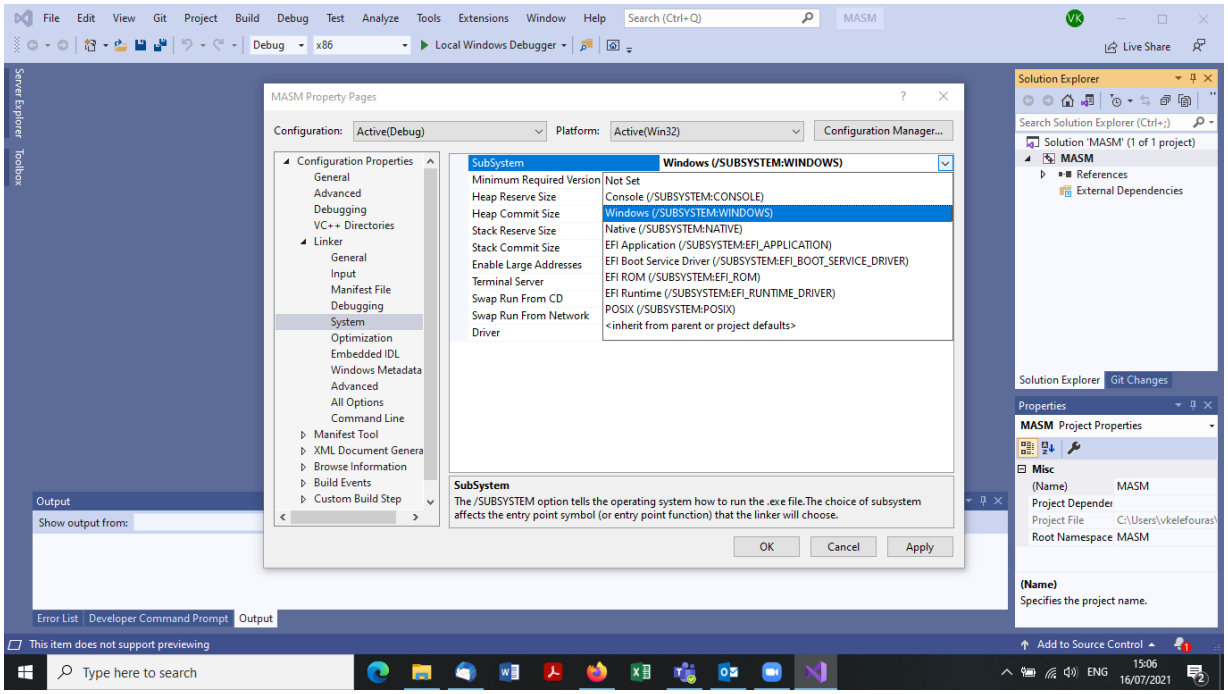

Fig.7 How to create a template for assembly programs

7. Create a new item: Right click on MASM and go to **Add > New Item**. Select C++ file and then name the file as **main.asm**. Press Add.

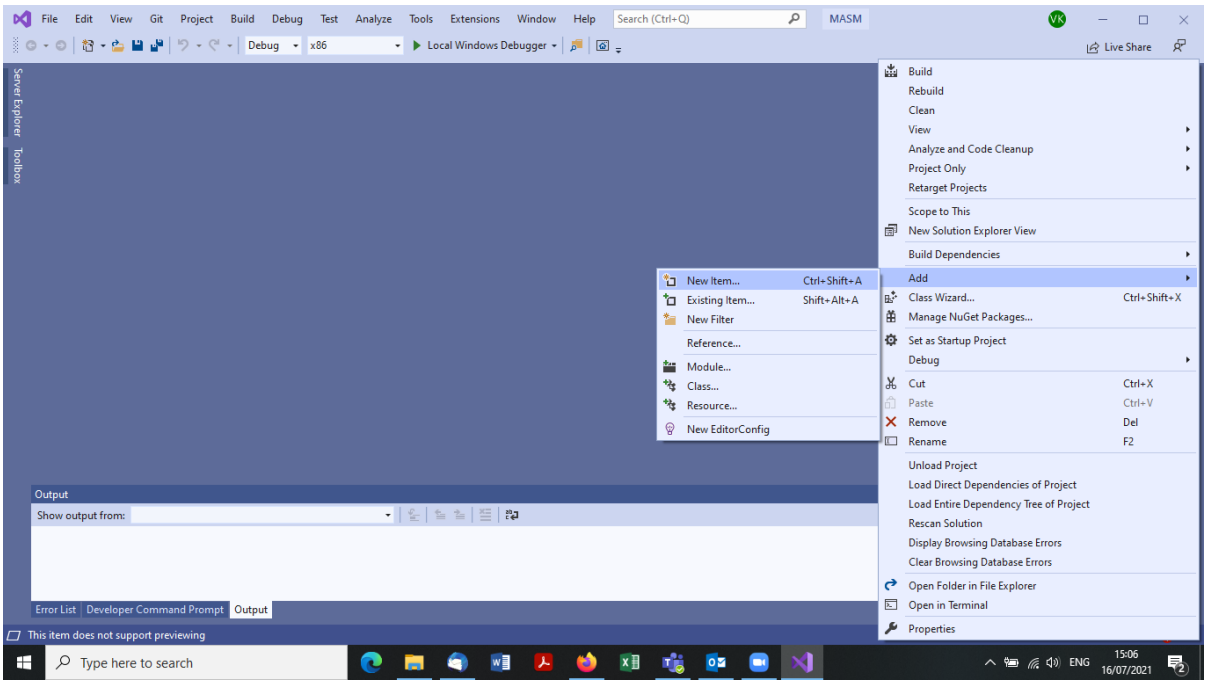

Fig.8 How to create a template for assembly programs

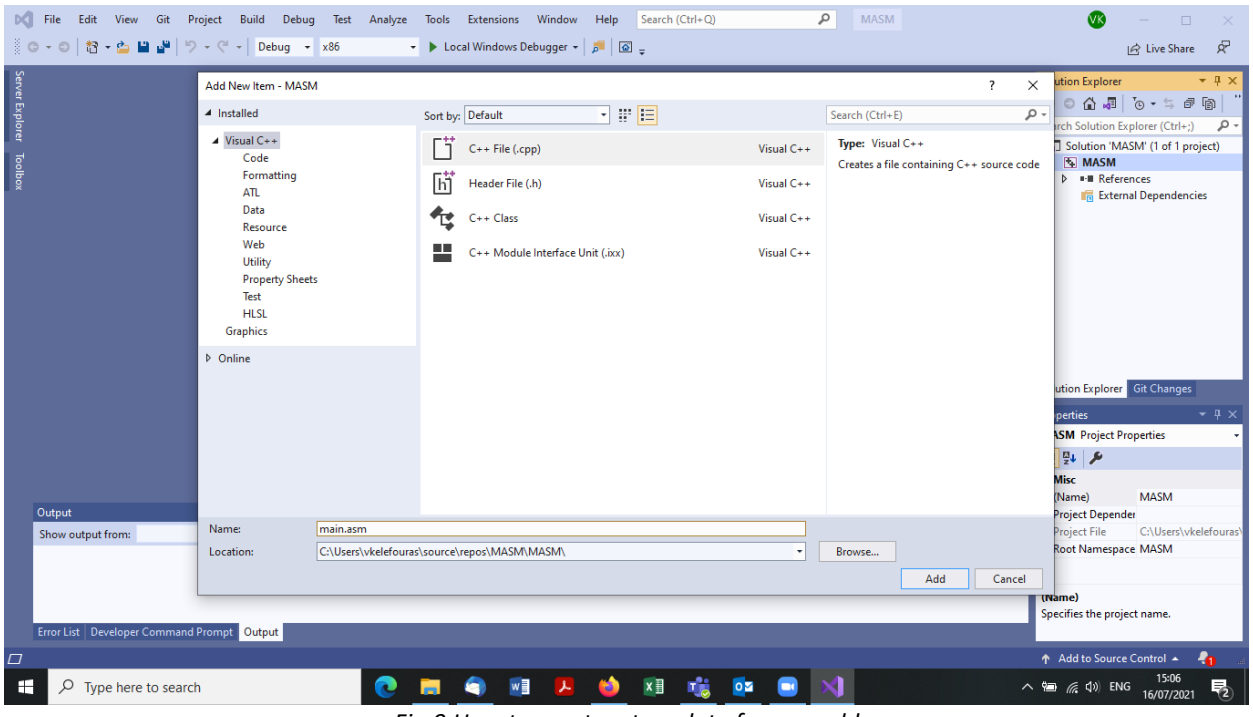

Fig.9 How to create a template for assembly programs

8. Copy the contents of **main.asm** on the DLE into your newly created **main.asm**, and save your project.

9. Now clean and build the project to ensure that everything works properly.

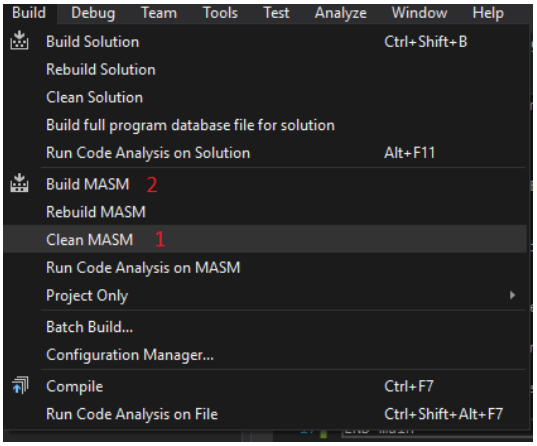

Fig.10 How to create a template for assembly programs

10. Exporting template. Go to Project menu, and then press Export Template. You may give it a description as shown in the figure below. Press Finish. You have now successfully created a template. Close the solution.

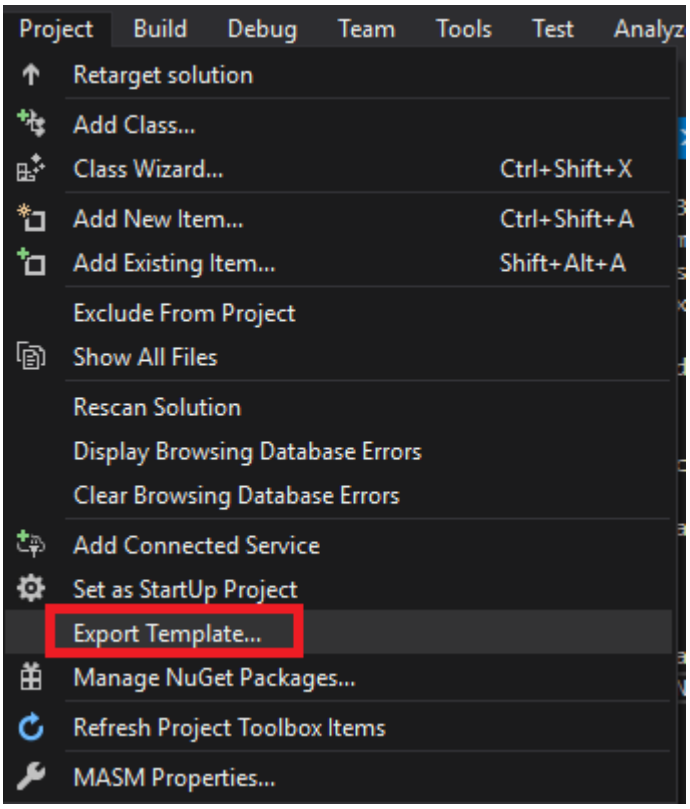

Fig.11 How to create a template for assembly programs

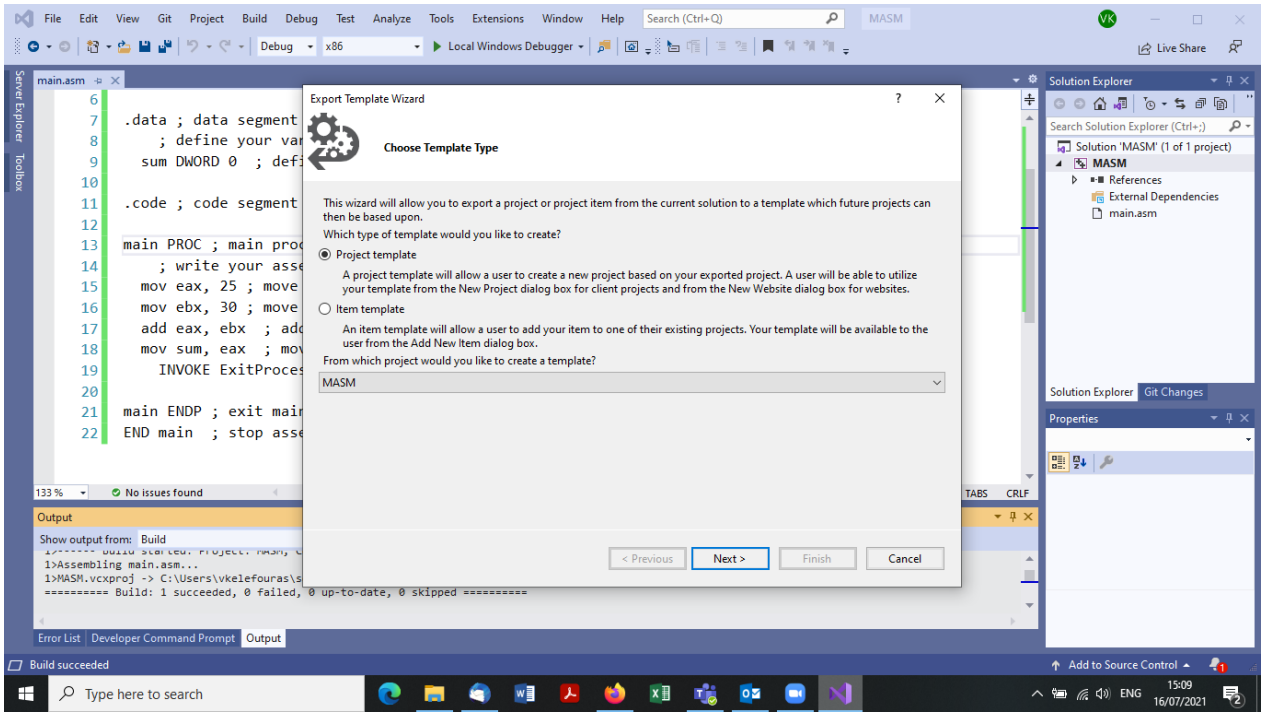

Fig.12 How to create a template for assembly programs

Creating a new project using the previous template.

In this task, you will create a new project with the newly created template.

1. Just like before, create a new project using File menu. This time, you should have an additional option: MASM (as depicted below). Select MASM and name your project **additionExample. Press OK. Then define your project's directory and project name. Do not use the default directory path.**

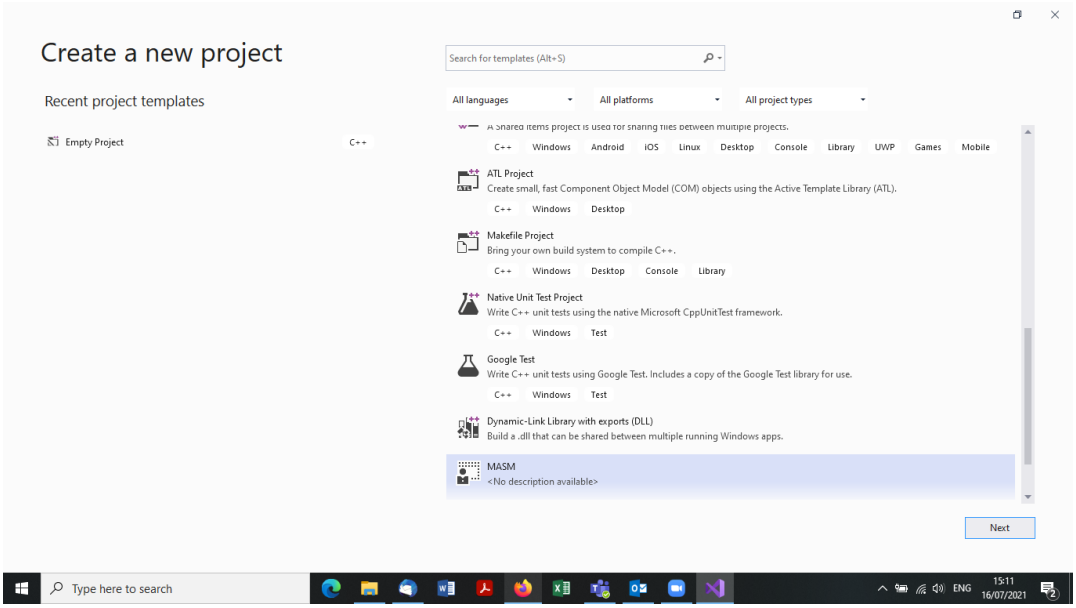

Fig.13 How to create a template for assembly programs

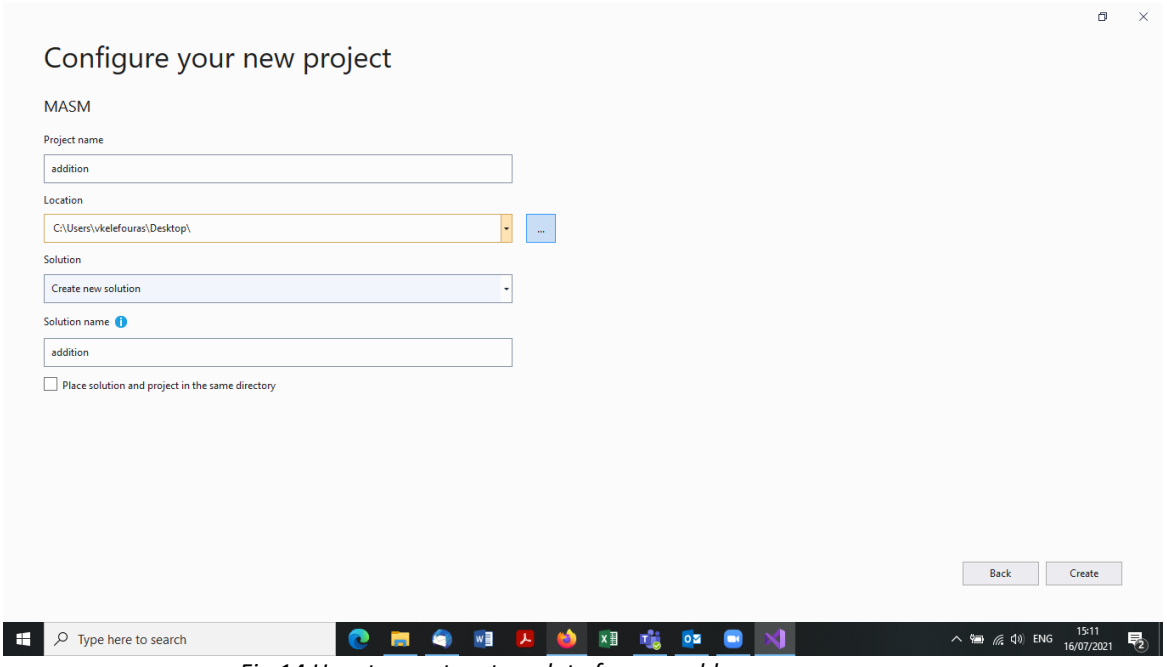

Fig.14 How to create a template for assembly programs

2. In your **main.asm** for this project, copy and paste the contents from **additionExample.asm.** You can change the name main.asm to **additionExample.asm,** if you want (but not needed). **Go into the build tab and click build.** This will generate the binary file of you program.
3. The next step is to run the program. This program does not print anything on the screen, so there is no point in running the code. However, we will run the program under debug mode.

This means that we will run the program instruction by instruction and check the status of the registers and memory in every step. To do so, we will use a breakpoint; hoover your mouse in the grey area, on the left of line 15 (our first instruction) and left click. You should see a red dot. This a breakpoint. Now go under the debug tab and click 'start debugging'. This option will run the program and it will stop into line 15, our breakpoint. You should see the following view now.

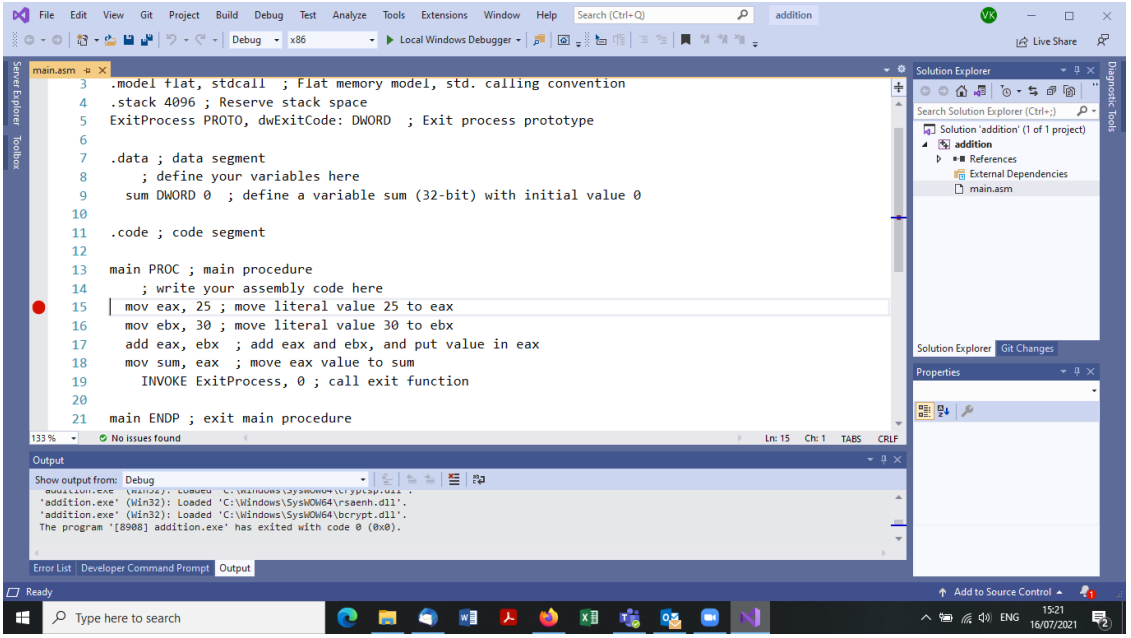

Fig.15 How to create a template for assembly programs

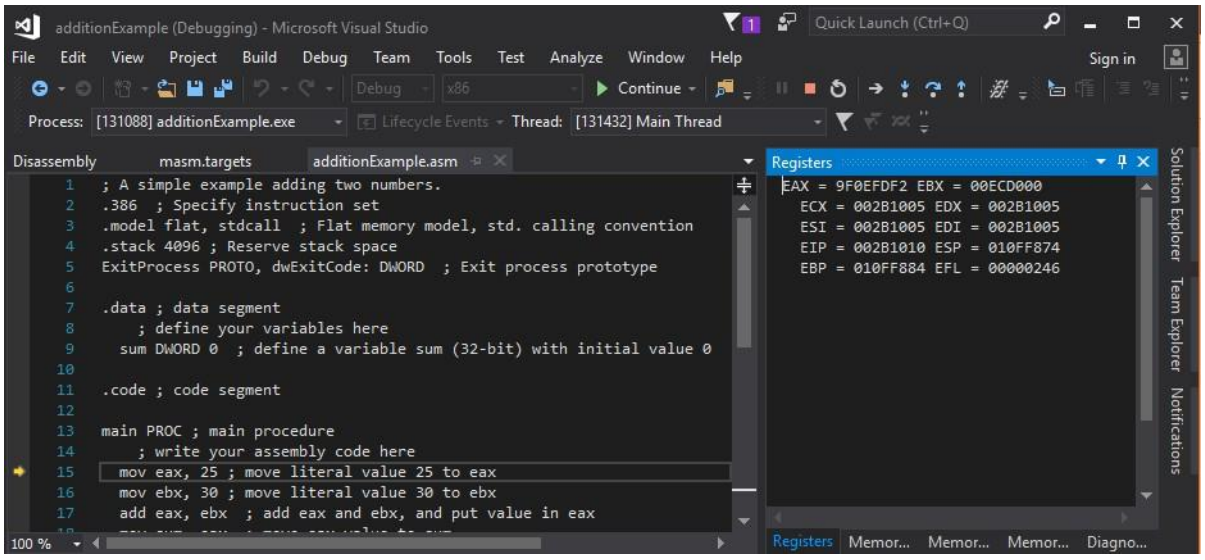

Fig.16 How to create a template for assembly programs

If you cannot see the registers' window, then use the following menu options to enable the view: Debug > Windows > Registers. If you have not clicked on the 'start debugging' option, then this option is not visible.

4. Now you can step through the program to see how the register values change and familiarize yourself with the components of the code. Proceed with the 1st example below which explains the process.

Assembly Example from the Lecture - Addition of two values

The '*additionExample.asm'* (see GitHub repository) is an assembly program that adds two numbers. In a high-level language this can be done by just using the following command

Sum = a + b

However, in assembly programming more instructions are needed to perform the above operation. It is important to note that when a developer writes a line of code like above in a high-level language, then the compiler is responsible for generating the assembly code. The compiler or the assembler will decide where the '*sum, a, b'* will be stored into memory and which instructions to choose.

In our first assembly program above (additionExample.asm) we have used several registers, such as eax and ebx. **A register is a very small and very fast dedicated memory, with a unique name**, e.g., eax. For now, assume that all the registers are 32-bits (4 bytes). The registers are being located in the CPU and therefore the arithmetic logic units can access them directly. **The CPU cannot execute any instruction unless their operands are stored into the registers (or their memory address is provided)**. Thus, in order to add two values, we must store them into the CPU's registers. In the next session you will learn more about them.

In the addition example code above, we define 'sum' variable. **A variable is not a register.** A variable is a high-level language construct to ease the development process. **Each variable is stored somewhere in memory** (the assembler decides this). We can define as many variables we want. On the contrary, the number of registers is limited. If we need to load the datum of a variable (e.g., by using *mov eax, sum instruction*), then the 'sum' variable will be loaded from memory, which is a slow operation. On the contrary, registers are accessed directly (there is no overhead).

The program's data (variables or arrays) are always defined after the *'.data'* command and the program's code is written after the *'.code'* command. The program's code is the following:

```
    mov eax, 25 ; move literal value 25 to 32-bit register eax
    mov ebx, 30 ; move literal value 30 to 32-bit register ebx
    add eax, ebx  ; add eax and ebx, and put value in eax
    mov sum, eax  ; move the eax register value to sum variable. The sum variable is stored into memory
```

Note that everything that follows ';' is a comment. All the x86 instructions can be found in this link https://www.felixcloutier.com/x86/ .

Let's debug the program, instruction by instruction. Put a breakpoint in line 15 and click 'start debugging'. If you do not know how to do that, read step 3 in '*Creating a new project using the previous template*' above. The registers' view shows the registers' contents in hexadecimal format. Now, go under the debug tab and click 'Step Into' or (F11). You will realize that the 'eax' value became red. This means that its value has just changed. Remember that the yellow arrow shows the next instruction to be executed, not the current one. Eax register has the value 19 in hex which is the 25 in decimal. Go into the 'watch' window and in the 'add item to watch' type eax and then enter. You will see the value of eax in decimal. Include the sum variable in the watch window too. Now click F11 again and check the status of the registers and sum variable.

Example #1 – Define variables

Write an assembly program that contains the following:

- A sum variable of the appropriate size to hold the initializer 0x10000
- A variable that does not contain an initial value and is able to hold the values 0-255.

Solution

All the data variables are defined after the *'.data'* command. We will choose the size of the variable (or register to use) according to their contents size. '0x10000' is a hexadecimal number because it starts with '0x'. You can use the windows calculator (programmer view) to see its decimal and binary equivalent. This number requires 17 bits (each hex digit needs up to 4 bits). The available data types are 8bit, 16bit, 32bit and 64 bit. Therefore, we need the 32 bit data type for holding this value.

Regarding the second bullet, its maximum value (255) needs 8bits (BYTE data type) and thus one byte is enough to store it. To let the assembler know that the variable is not initialized we can specify the '*?*' character. The code follows

```
aVariable BYTE ?
```

Example #2 - Addition and Subtraction.

Create an assembly program (from the template you created earlier) that calculates the following expression: answer = (A + B) – (C + D). See below a few hints.

- The values for your data (A, B, C, D) must be stored in registers (e.g., eax, ebx, ecx, edx), not in variables. You will learn more about the registers in the next session.
- You must supply the initial values for the data (A, B, C, D).
- Comment each line of code, as demonstrated in the examples you have seen so far.

All the x86 instructions can be found here https://www.felixcloutier.com/x86/ .

Further Reading

1. Chapter 1 and Chapter 2 in 'Modern X86 Assembly Language Programming' , available at https://www.pdfdrive.com/download.pdf?id=185772000&h=3dfb070c1742f50b500f07a63a30c86a&u=cache&ext=pdf
2. x86 and amd64 instruction reference Available at All the x86 instructions can be found here https://www.felixcloutier.com/x86/ .