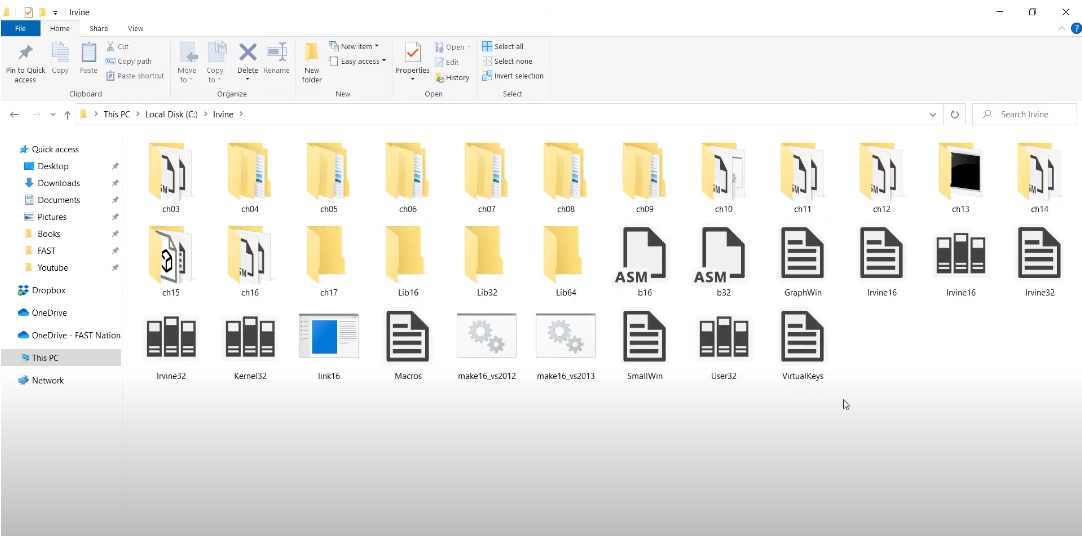
**Lab 09**

**Topic:**

1. Irvine setup
2. Procedure
3. Stack operations

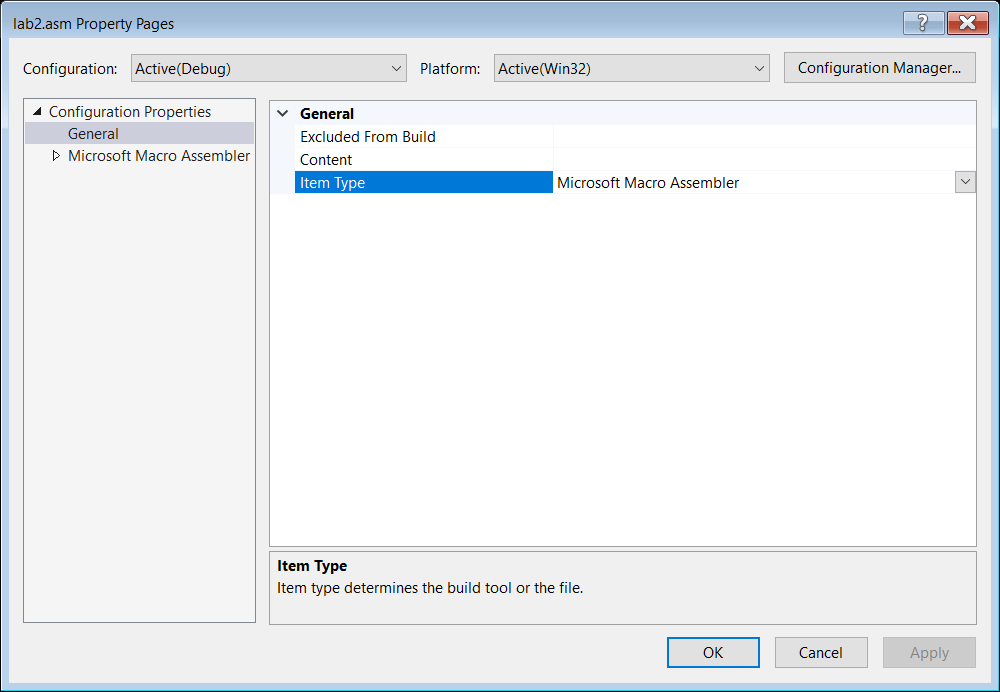
**Irvine setup:**

First download Irvine it from your Google Classroom unzip it into c directory folder

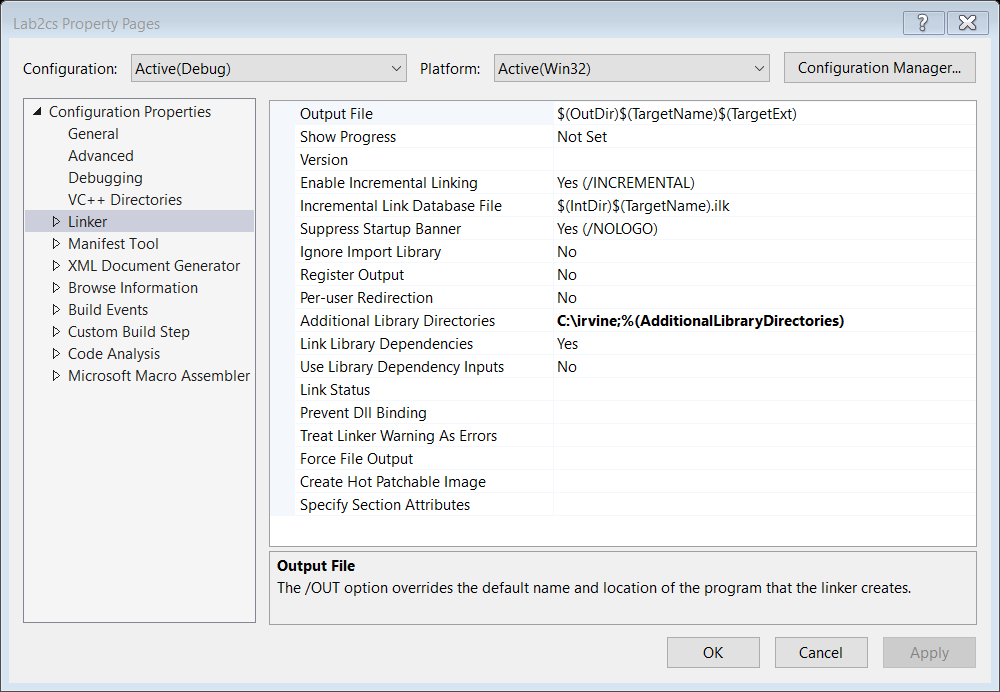


After that copy the file path and includes it into your project path into multiple following steps.

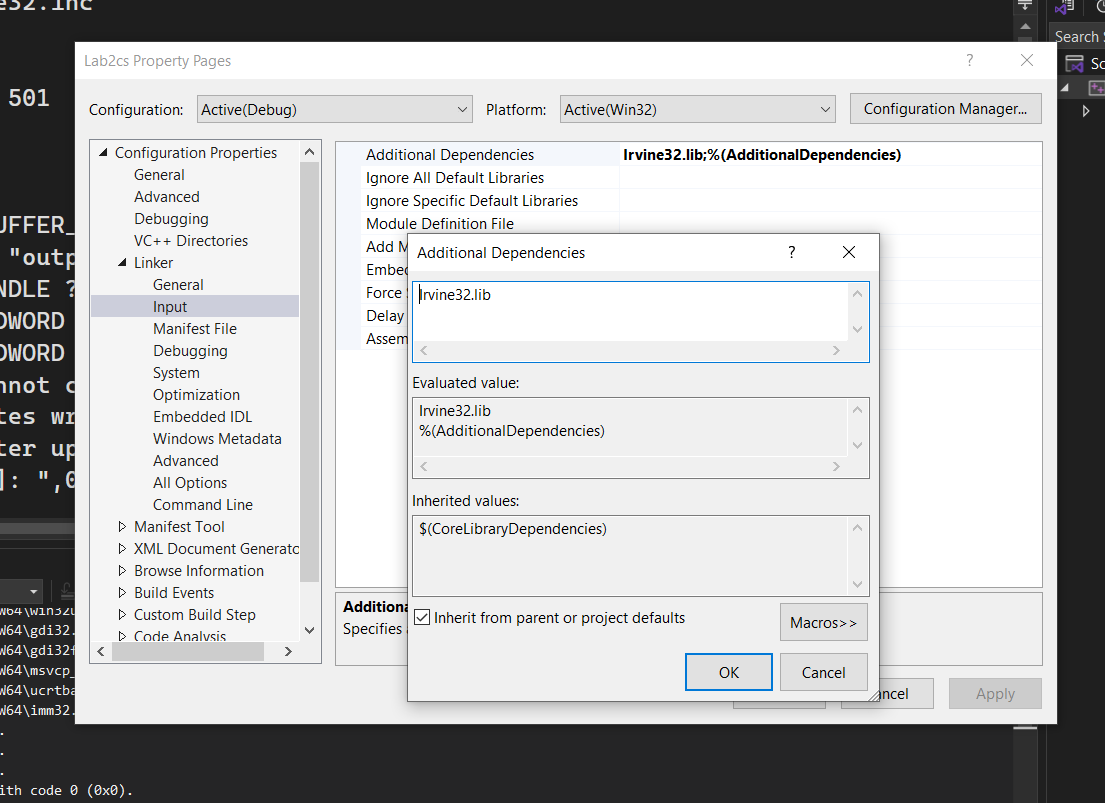
1. Right click you .asm file and go to **properties** section click new window is pop out from there you need to go **Item Type** just select **Microsoft Macro Assembler** then apply and ok.



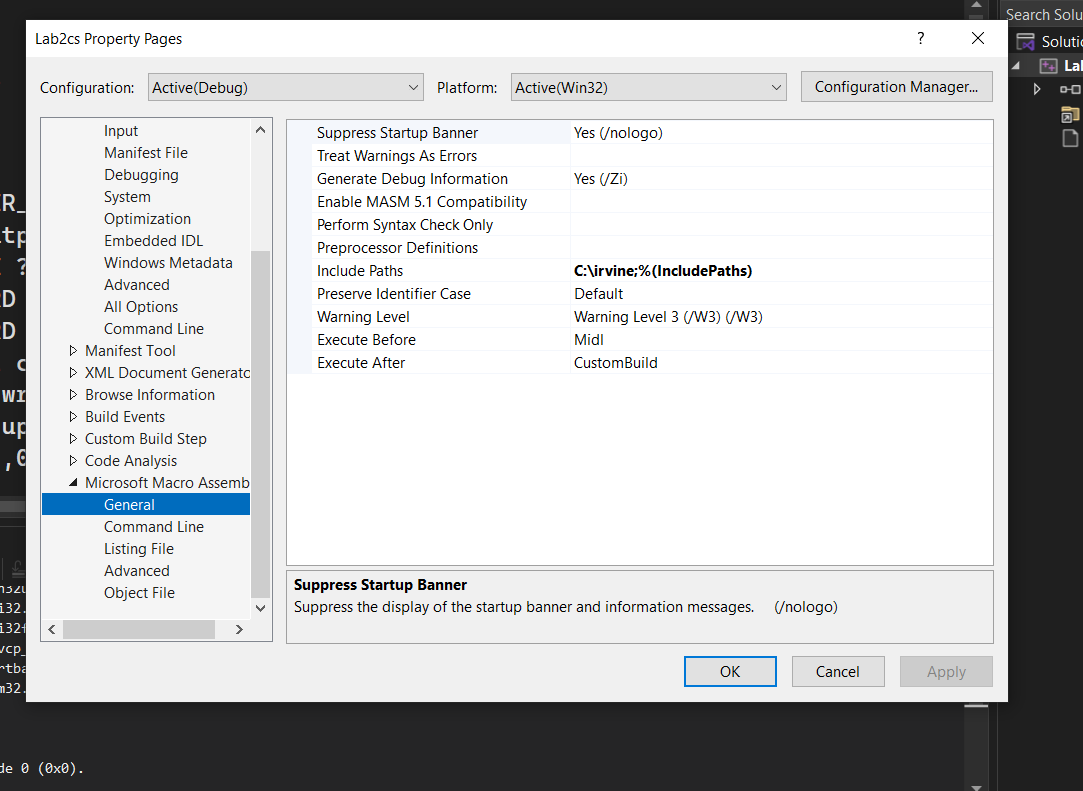
Now copy the path of Irvine folder and right click of your project included go to **properties** section from that select **Linker** option select **Additional Library Directories** includes you need to paste the path of Irvine folder that is present in your C drive.



Feom Link drop down go to input section and select the **Additional Dependencies** just write **Irvine32.lib** and ok.



Now go to the **Microsoft Macro Assembler** then select **Inculde path** paste the path then click Ok than Apply and ok



Now your Irvine setup is included just paste this code and test it.

INCLUDE Irvine32.inc ; Include the Irvine32 library

.data ; Data segment

myName BYTE "Kanwal", 0 ; Define the name and end with null terminator

.code ; Code segment

main PROC

mov edx, OFFSET myName ; Move the address of the string into EDX

call WriteString ; Call the WriteString procedure to print the string

call CrLf ; Move to the next line after printing

exit ; Exit the program

main ENDP

END main

**Procedure/Functions**

### Procedure Declaration and Definition in Assembly

In assembly language, a **procedure** (also called a subroutine or function) is a block of code that performs a specific task. Procedures are useful for modularizing code and avoiding repetition. The Irvine32 library includes built-in procedures like WriteString, but you can also define your own.

#### Structure of a Procedure

1. **Declaration**: A procedure is declared by giving it a name followed by the keyword PROC.
2. **Definition**: The actual code of the procedure is written between the declaration (PROC) and the termination (ENDP).
3. **Return**: The procedure typically ends with a RET instruction, which tells the CPU to return to the calling code.

#### Key Elements

* **PROC**: Marks the beginning of the procedure.
* **ENDP**: Marks the end of the procedure.
* **RET**: Returns control to the calling function.
* **CALL**: The instruction to invoke the procedure.

#### Simple Example

This example demonstrates a procedure called DisplayMessage that prints a message to the console using the Irvine32 library.

### Code Example (main.asm)

INCLUDE Irvine32.inc ; Include the Irvine32 library

.data

message BYTE "Hello from the procedure!", 0 ; Define a message string

.code

main PROC

call DisplayMessage ; Call the DisplayMessage procedure

exit ; Exit the program

main ENDP

; Define the DisplayMessage procedure

DisplayMessage PROC

mov edx, OFFSET message ; Load the address of the message into EDX

call WriteString ; Print the string using Irvine's WriteString procedure

call CrLf ; Move to the next line

ret ; Return to the calling function

DisplayMessage ENDP ; End of the procedure

END main ; End of the program

**Stack Applications**

There are several important uses of runtime stacks in programs:

* 1. A stack makes a convenient temporary save area for registers when they are used for more than one purpose. After they are modified, they can be restored to their original values.
  2. When the CALL instruction executes, the CPU saves the current subroutine’s return address on the stack.
  3. When calling a subroutine, you pass input values called arguments by pushing them on the stack.
  4. The stack provides temporary storage for local variables inside subroutines.

**PUSH Instruction:**

The PUSH instruction first decrements ESP and then copies a source operand into the stack.

A 16-bit operand causes ESP to be decremented by 2. A 32-bit operand causes ESP to be decremented by 4. There are three instruction formats:

PUSH reg/mem16

PUSH reg/mem32

PUSH imm32

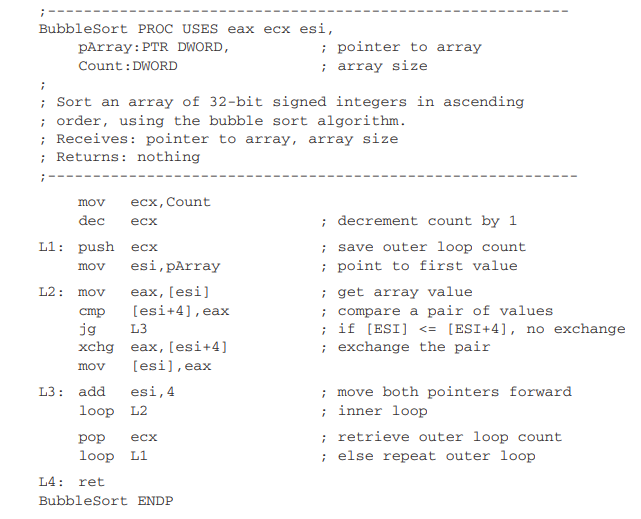
**POP Instruction:**

The POP instruction first copies the contents of the stack element pointed to by ESP into a 16- or

32-bit destination operand and then increments ESP. If the operand is 16 bits, ESP is incremented by 2; if the operand is 32 bits, ESP is incremented by 4:

POP reg/mem16

POP reg/mem32



**Tasks:**

### 1. **Checking Balanced Parentheses**

* **Problem Statement**: Write an assembly program to check if a string of parentheses is balanced using a stack. For each opening parenthesis (, push it onto the stack, and for each closing parenthesis ), pop from the stack and check if it matches.
* **Input**: "((())())"
* **Output**: "Balanced"

### 2. **Palindrome Checker**

* **Problem Statement**: Write an assembly program to check whether a given string is a palindrome. Use the stack to reverse the string and compare it with the original.
* **Input**: "madam"
* **Output**: "Palindrome"

### 3. **Evaluating a Postfix Expression**

* **Problem Statement**: Implement a postfix expression evaluator using stack operations. Push operands onto the stack and pop them when encountering an operator to perform calculations.
* **Input**: "5 6 + 2 \*" (Postfix notation for (5 + 6) \* 2)
* **Output**: 22

### 4. **Sorting an Array using Stack (Bubble Sort)**

* **Problem Statement**: Use stack operations (PUSH and POP) to simulate swapping in a Bubble Sort algorithm to sort an array.
* **Input**: [8, 4, 3, 7, 6]
* **Output**: [3, 4, 6, 7, 8]

### 5. **Infix to Postfix Conversion**

* **Problem Statement**: Convert an infix expression (e.g., 3 + 4 \* 2) to postfix notation (e.g., 3 4 2 \* +) using stack operations.
* **Input**: "3 + 4 \* 2"
* **Output**: "3 4 2 \* +"