2025

Computer Organization and Assembly Language

Dr. Nasrullah Jaleel;

Dr. Adnan Ghafoor

Faculty of Information Technology & Computer Science, University of Central Punjab, Lahore

Computer Organization and Assembly Language
CSCS3543 Theory
CSCS3541 Lab

HANDOUT

Faculty of Information Technology & Computer Science
University of Central Punjab
Lahore

Table of Contents

Lab 1: Number Systems

Learning Outcomes

- Students will come to know about various number systems.
- They will be able to perform conversions between number systems.
- They will be able to perform basic arithmetic operations on binary number systems.
- They will be able to perform addition in hexadecimal and octal number systems.
- They will be able to represent signed numbers.
- They will be able to represent BCD numbers and perform their addition.

Number System

Number systems are the technique to represent numbers in the computer system architecture, every value that you are saving or getting into/from computer memory has a defined number system. Computer architecture supports the following number systems.

- Binary number system
- Octal number system
- Decimal number system
- Hexadecimal (hex) number system

Binary Number System

A Binary number system has only two digits, that are 0 and 1. Every number (value) represents **0 and 1** in this number system. The base of the binary number system is 2 because it has only two digits.

Octal Number System

The octal number system has only eight (8) digits from **0 to 7**. Every number (value) represents with 0, 1, 2, 3, 4, 5, 6 and 7 in this number system. The base of the octal number system is 8 because it has only 8 digits.

Decimal Number System

The decimal number system has only ten (10) digits from **0 to 9**. Every number (value) represents 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 in this number system. The base of the decimal number system is 10 because it has only 10 digits.

Hexadecimal Number System

A Hexadecimal number system has sixteen (16) alphanumeric values from **0 to 9 and A to F**. Every number (value) represents 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F in this number system. The base of the hexadecimal number system is 16 because it has 16 alphanumeric values. Here A is 10, B is 11, C is 12, D is 13, E is 14 and F is 15.

Number system	Base (Radix)	Used digits	Example
Binary	2	0, 1	$(11110000)_2$
Octal	8	0, 1, 2, 3, 4, 5, 6, 7	$(360)_8$
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	$(240)_{10}$
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F	$(F0)_{16}$

Conversions From Decimal to Other

Decimal Number System to Other Base

To convert a Number system from a **Decimal Number System** to **Any Other Base** is quite easy. You have to follow just two steps:

A) Divide the Number (Decimal Number) by the base of the target base system (in which you want to convert the number: Binary (2), octal (8), and Hexadecimal (16).

B) Write the remainder from Step 1 as a Least Signification Bit (LSB) to Step Last as a Most Significant Bit (MSB).

1. **Decimal To Binary**

Decimal to Binary Conversion				**Result**
Decimal Number is: **(12345)10**				
2	12345	1	**LSB**	
2	6172	0		
2	3086	0		
2	1543	1		
2	771	1		
2	385	1		Binary Number is
2	192	0		**(11000000111001)2**
2	96	0		
2	48	0		
2	24	0		
2	12	0		
2	6	0		
2	3	1		
	1	1	**MSB**	

2. Decimal To Octal

Decimal to Octal Conversion	Result
Decimal Number is: **$(12345)_{10}$** 8 | 12345 — 1 LSB 8 | 1543 — 7 8 | 192 — 0 8 | 24 — 0 3 — 3 MSB	Octal Number is **$(30071)_8$**

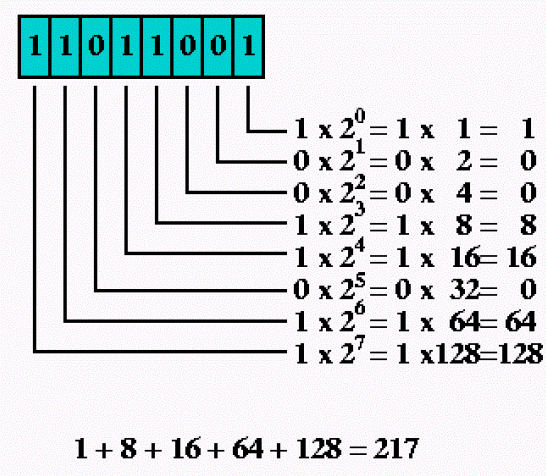
3. Decimal To Hexadecimal

Decimal to Hexadecimal Conversion	Result
Example 1 The Decimal Number is: **$(12345)_{10}$** 16 | 12345 — 9 LSB 16 | 771 — 3 16 | 48 — 0 8 | 3 — 3 MSB	Hexadecimal Number is **$(3039)_{16}$**
Example 2 The Decimal Number is: **$(725)_{10}$** 16 | 725 — 5 LSB 16 | 45 — 13 (D) 2 — 2 MSB	Hexadecimal Number is **$(2D5)_{16}$** **Convert 10, 11, 12, 13, 14, 15 to its equivalent...** **A, B, C, D, E, F**

Binary To Other

Multiply the digit with 2(with place value exponent). Eventually, the sum of all the multiplication becomes the Decimal number.

1. **Binary To Decimal**



2. **Binary To Octal**

An easy way to convert from binary to octal is to group binary digits into sets of three, starting with the least significant (rightmost) digits.

Binary: 11100101 =	11 100 101	
	011 100 101	Pad the most significant digits with zeros if necessary to complete a group of three.

Then, look up each group in a table:

Binary:	000	001	010	011	100	101	110	111
Octal:	0	1	2	3	4	5	6	7

Binary =	011	100	101	
Octal =	3	4	5	= $\mathbf{(345)_8}$

3. Binary To Hexadecimal

An equally easy way to convert from binary to hexadecimal is to group binary digits into sets of four, starting with the least significant (rightmost) digits.

Binary: 11100101 = 1110 0101

Then, look up each group in a table:

Binary:	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
Hexadecimal:	0	1	2	3	4	5	6	7	8	9

Binary:	1010	1011	1100	1101	1110	1111
Hexadecimal:	A	B	C	D	E	F

Binary =	1110	0101	
Hexadecimal =	E	5	= **$(E5)_{16}$**

Octal To Other

1. Octal To Binary

Converting from octal to binary is as easy as converting from binary to octal. Simply look up each octal digit to obtain the equivalent group of three binary digits.

Octal:	0	1	2	3	4	5	6	7
Binary:	000	001	010	011	100	101	110	111

Octal =	3	4	5	
Binary =	011	100	101	= **$(011100101)_2$**

2. Octal To Hexadecimal

When converting from octal to hexadecimal, it is often easier to first convert the octal number into binary and then from binary into hexadecimal. For example, to convert 345 octal into hex:

(from the previous example)

Octal =	3	4	5	
Binary =	011	100	101	= **$(011100101)_2$**

Drop any leading zeros or pads with leading zeros to get groups of four binary digits (bits): Binary 011100101 = 1110 0101

Then, look up the groups in a table to convert to hexadecimal digits.

Binary:	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
Hexadecimal:	0	1	2	3	4	5	6	7	8	9

Binary:	1010	1011	1100	1101	1110	1111
Hexadecimal:	A	B	C	D	E	F

Binary =	1110	0101	
Hexadecimal =	E	5	= **$(E5)_{16}$**

Therefore, through a two-step conversion process, octal 345 equals binary $(11100101)_2$ equals hexadecimal $(E5)_{16}$.

3. Octal To Decimal

The conversion can also be performed in the conventional mathematical way, by showing each digit place as an increasing power of 8.

$(345)_8 = (3 * 8^2) + (4 * 8^1) + (5 * 8^0) = (3 * 64) + (4 * 8) + (5 * 1) = 192 + 32 + 5 = 229$ decimal

Hexadecimal To Other

1. Hexadecimal To Binary

Converting from hexadecimal to binary is as easy as converting from binary to hexadecimal. Simply look up each hexadecimal digit to obtain the equivalent group of four binary digits.

Binary:	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
Hexadecimal:	0	1	2	3	4	5	6	7	8	9

Binary:	1010	1011	1100	1101	1110	1111
Hexadecimal:	A	B	C	D	E	F

Hexadecimal =	A	2	D	E	
Binary =	1010	0010	1101	1110	= 1010001011011110 binary

2. Hexadecimal To Octal

To Convert a hexadecimal number to an octal number, we need to first convert the hexadecimal number into a binary number and then from binary to Octal. Simply look up each hexadecimal digit to obtain the equivalent group of four binary digits.

Binary:	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
Hexadecimal:	0	1	2	3	4	5	6	7	8	9

Binary:	1010	1011	1100	1101	1110	1111
Hexadecimal:	A	B	C	D	E	F

Hexadecimal =	A	2	D	E			
Binary =	1010	0010	1101	1110	= 1010001011011110 binary		
Re-grouping	001	010	001	011	011	110	
Octal	1	2	1	3	3	6	=121336 Octal

Binary Arithmetic

Rules of Binary Addition

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 0, and carry 1 to the next more significant bit

For example,

`00011010 + 00001100 = 00100110`

			1	1						***Carries***
	0	0	0	1	1	0	1	0	=	$26_{(base10)}$
+	0	0	0	0	1	1	0	0	=	$12_{(base10)}$
	0	**0**	**1**	**0**	**0**	**1**	**1**	**0**	=	$38_{(base10)}$

`00010011 + 00111110 = 01010001`

		1	1	1	1	1				***Carries***
	0	0	0	1	0	0	1	1	=	$19_{(base10)}$
+	0	0	1	1	1	1	1	0	=	$62_{(base10)}$
	0	**1**	**0**	**1**	**0**	**0**	**0**	**1**	=	$81_{(base10)}$

Rules of Binary Multiplication

- 0 x 0 = 0
- 0 x 1 = 0
- 1 x 0 = 0
- 1 x 1 = 1, and no carry or borrow bits

00101001 x 00000110 = 11110110

		0	0	1	0	1	0	0	1	= **41$_{(base10)}$**
	x	0	0	0	0	0	1	1	0	= **6$_{(base10)}$**
		0	0	0	0	0	0	0	0	
	0	0	1	0	1	0	0	1		
0	0	1	0	1	0	0	1			
0	0	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	= **246$_{(base10)}$**

Binary Division

Binary division is the repeated process of subtraction, just as in decimal division.

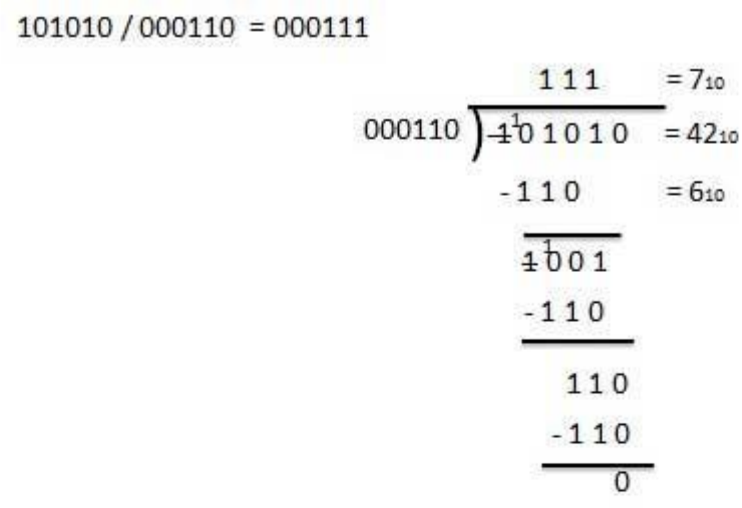
For example,

00101010 ÷ 00000110 = 00000111

		1	1	1					= **7$_{(base10)}$**
6$_{(base10)}$ =	110)	0	0	~~1~~	$^{1}0$	1	0	1	0 = **42$_{(base10)}$**
				−	1	1	0		
					~~1~~	^{1}~~0~~	$^{1}0$	1	
				−		1	1	0	
							1	1	0
						−	1	1	0
							0	0	0

10000111 ÷ 00000101 = 00011011

		1	1	0	1	1				= **27$_{(base10)}$**
5$_{(base10)}$ =	101)	1	~~1~~ 0	~~1~~ 0	^{1}0	0	1	1	1	= **135$_{(base10)}$**
		−	1	0	1					
				1	~~1~~	^{1}0				
			−	1	0	1				
						1	1			
					−		0			
						1	1	1		
					−	1	0	1		
							1	0	1	
						−	1	0	1	
							0	0	0	



Hexadecimal Addition

Following the hexadecimal addition table will help you greatly to handle Hexadecimal addition.

+ (Y \ X)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

To use this table, simply follow the directions used in this example − Add A_{16} and 5_{16}. Locate A in the X column then locate the 5 in the Y column. The point in the 'sum' area where these two columns intersect is the sum of two numbers.

$A_{16} + 5_{16} = F_{16}$

Example

$4A6_{16} + 1B3_{16} = 659_{16}$

	1	carry
	4 A 6	$= 1190_{10}$
+	1 B 3	$= 435_{10}$
	6 5 9	$= 1625_{10}$

Octal Addition

Following the octal addition table will help you to handle octal addition.

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

A (header row) — Sum (body) — B (first column)

To use this table, simply follow the directions used in this example: Add 6_8 and 5_8. Locate 6 in the A column then locate the 5 in the B column. The point in the 'sum' area where these two columns intersect is the 'sum' of two numbers.

$6_8 + 5_8 = 13_8$.

Example

$456_8 + 123_8 = 601_8$

	1 1	carry
	4 5 6	$= 302_{10}$
+	1 2 3	$= 83_{10}$
	6 0 1	$= 385_{10}$

Signed Numbers

We can make the binary numbers into the following two groups − **Unsigned numbers** and **Signed numbers**.

Unsigned Numbers

Unsigned numbers contain only the magnitude of the number. They don't have any sign. That means all unsigned binary numbers are positive. As in the decimal number system, the placing of a positive sign in front of the number is optional for representing positive numbers. Therefore, all positive numbers including zero can be treated as unsigned numbers if the positive sign is not assigned in front of the number.

Signed Numbers

Signed numbers contain both the sign and magnitude of the number. Generally, the sign is placed in front of the number. So, we have to consider the positive sign for positive numbers and the negative sign for negative numbers. Therefore, all numbers can be treated as signed numbers if the corresponding sign is assigned in front of the number.

If the sign bit is zero, it indicates the binary number is positive. Similarly, if the sign bit is one, it indicates the binary number is negative.

Representation of Un-Signed Binary Numbers

The bits present in the un-signed binary number hold the **magnitude** of a number. That means if the un-signed binary number contains **'N'** bits, then all **N** bits represent the magnitude of the number since it doesn't have any sign bit.

Example

Consider **decimal number 108**. The binary equivalent of this number is **1101100**. This is the representation of the unsigned binary number.

$$\mathbf{108_{10} = 1101100_2}$$

It has 7 bits. These 7 bits represent the magnitude of the number 108.

Representation of Signed Binary Numbers

The Most Significant Bit (MSB) of signed binary numbers is used to indicate the sign of the numbers. Hence, it is also called a sign bit. The positive sign is represented by placing '0' in the sign bit. Similarly, the negative sign is represented by placing '1' in the sign bit.

If the signed binary number contains 'N' bits, then N−1 bits only represent the magnitude of the number since the MSB (one-bit) is reserved for representing the sign of the number. There are three types of representations for signed binary numbers:

- Sign-Magnitude form
- 1's complement form
- 2's complement form

The representation of a positive number in all these three forms is the same. However, only the representation of negative numbers will differ in each form.

Example

Consider the **positive decimal number +108**. The binary equivalent of the magnitude of this number is 1101100. These 7 bits represent the magnitude of the number 108. Since it is a positive number, consider the sign bit as zero, which is placed on the leftmost side of magnitude.

$$+108_{10} = \mathbf{01101100_2}$$

Therefore, the **signed binary representation** of the positive decimal number +108 is $\mathbf{01101100_2}$. So, the same representation is valid in sign-magnitude form, 1's complement form, and 2's complement form for the positive decimal number +108.

Sign-Magnitude form.

In sign-magnitude form, the MSB is used to represent a **sign** of the number, and the remaining bits represent the **magnitude** of the number. So, just include a sign bit at the leftmost side of an unsigned binary number. This representation is similar to the signed decimal numbers representation.

Example

Consider the **negative decimal number -108**. The magnitude of this number is 108. We know the unsigned binary representation of 108 is **1101100**. It has 7 bits. All these bits represent the magnitude.

Since the given number is negative, consider the sign bit as one, which is placed on the leftmost side of magnitude.

$$-108_{10} = \mathbf{1\ 1101100_2}$$

Therefore, the sign-magnitude representation of -108 is $\mathbf{11101100_2}$.

1's complement form

The 1's complement of a number is obtained by **complementing all the bits** of a signed binary number. So, 1's complement of a positive number gives a negative number. Similarly, 1's complement of a negative number gives a positive number.

That means, if you perform two times 1's complement of a binary number including sign bit, then you will get the original signed binary number.

Example

Consider the **negative decimal number -108**. The magnitude of this number is 108. We know the signed binary representation of 108 is **01101100**.

It has 8 bits. The MSB of this number is zero, which indicates the positive number. The complement of zero is one and vice-versa. So, replace zeros by one and ones by zeros in order to get the negative number.

$$-108_{10} = \mathbf{10010011_2}$$

Therefore, the **1's complement of** $\mathbf{108_{10}}$ is $\mathbf{10010011_2}$.

2's complement form

The 2's complement of a binary number is obtained by **adding one to the 1's complement** of a signed binary number. So, 2's complement of a positive number gives a negative number. Similarly, 2's complement of a negative number gives a positive number.

That means, if you perform 2's complement twice (two times) for a binary number including sign bit, then you will get the original signed binary number.

Example

Consider the **negative decimal number -108**.

We know the 1's complement of $(\mathbf{108})_{10}$ is $(\mathbf{10010011})_2$

2's compliment of 108108_{10} *= 1's compliment of* 108108_{10} *+ 1.*

= 10010011 + 1

= 10010100

Therefore, the **2's complement of** 108_{10} is $\mathbf{10010100}_2$.

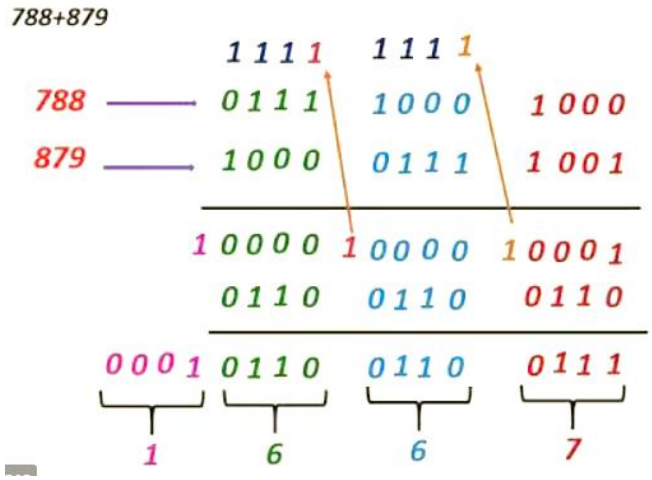
Binary Coded Decimal (BCD) Numbers

Binary-Coded Decimal (BCD) is a class of binary encodings of decimal numbers where each decimal digit is represented by a fixed number of bits, four bits. For example, decimal 396 is represented in BCD with 12 bits as 0011 1001 0110. A decimal number in BCD is the same as its equivalent binary number only when the number is between 0 and 9. A BCD number greater than 10 looks different from its equivalent binary number, even though both contain 1's and 0's. Moreover, the binary combinations 1010 through 1111 are not used and have no meaning in BCD.

4-bit Binary	Hexadecimal	BCD	Decimal Equivalent
0000	0	0000	0
0001	1	0001	1
0010	2	0010	2
0011	3	0011	3
0100	4	0100	4
0101	5	0101	5
0110	6	0110	6
0111	7	0111	7
1000	8	1000	8
1001	9	1001	9
1010	A		10
1011	B		11
1100	C		12
1101	D		13
1110	E		14
1111	F		15

BCD adders

A BCD number is added like a 4-bit binary number for each corresponding digit. Since BCD has 4 bits with the largest number being 9, and the largest 4-bit binary number is equivalent to 15, there is a difference of 6 between the binary and the BCD adder. If a sum exceeds 9, 6 is further added to it, and carry is propagated to the next digit as shown below.



CSCS3543 Computer Organization and Assembly Language

Lab-1 Exercise

Task-1

Convert the basis of the following numbers. Use direct method where applicable.

a) $(10100110)_2$ = ( )$_{10}$
b) $(11101011)_2$ = ( )$_8$
c) $(00110111)_2$ = ( )$_{16}$
d) $(3471)_8$ = ( )$_{10}$
e) $(7372)_8$ = ( )$_{16}$
f) $(1B2CA)_{16}$ = ( )$_{10}$
g) $(7E5E2)_{16}$ = ( )$_2$
h) $(853325)_{10}$ = ( )$_2$
i) $(71552)_{10}$ = ( )$_8$

Task-2

Perform the following arithmetic operations on numbers.

a) $(1010010)_2 + (1011011)_2$
b) $(1101001)_2 \times (101)_2$
c) $(1110011)_2 / (11)_2$
d) $(BCD)_{16} + (386)_{16}$
e) $(4737)_8 + (121)_8$

Task-3

Using the binary number system, add up the following signed numbers by writing them in each of the three ways to represent signed numbers. Also, check which representation method is valid for performing arithmetic operations.

a) **57** + (**-38**)
b) **100** + (**-51**)
c) (**-42**) + (**-84**)

Task-4

Add the following BCD numbers. (Show intermediate working.)

a) $(\mathbf{6178})_{BCD} + (\mathbf{4933})_{BCD}$
b) $(\mathbf{2901})_{BCD} + (\mathbf{4734})_{BCD}$

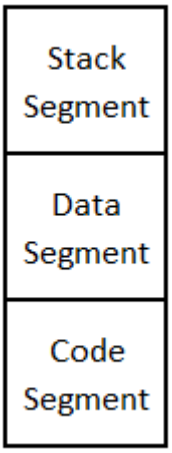
Lab 2: Introduction to Emu8086

Learning Outcomes

- Students will know the basic structure of a program.
- Students will know about various assembler directives.
- Students will come to know how addresses are translated from logical to physical.
- Students will write a basic assembly language program.
- Students will learn the basic rules of writing assembly instructions.
- Students will be able to debug and run their first program.

Structure of a program

Every program that is compiled using a compiler contains primarily two segments: Code and Data. When a program comes into execution, another segment is attached to it, called a stack segment. The structure of a program during execution is shown below.



The structure of an assembly language program is quite similar to that of a compiled program. The data segment contains global variables, and the code segment contains executable code. The code segment contains the main function as well as other user-defined functions.

In order to define various segments in an assembly language program, assembler directives are used. Directives are commands that are part of the assembler syntax but are not related to the x86 processor instruction set. All assembler directives begin with a period (.). The following table shows various assembler directives that we will use in our program.

.model	Defines the number of code and data segments a program can have. Small: For 1 code and 1 data segment Medium: for 1 data and more than 1 code segment Large: for more than 1 code and data segment. Tiny: code and data fit in a single segment. Used for Com file.
.stack <size>	Marks the beginning of the stack segment. Also, define the size of the stack.
.data	Marks the beginning of the data segment
.code	Marks the beginning of the code segment
.exit	Terminates a program.

The structure of an assembly language program is given below.

```
.model small

.stack 100h

.data
      Global variables are declared here.
.code
       Code and functions are defined in this segment.

.exit
```

Registers

Registers are the storage elements inside a processor. Their size is equal to the size of a processor. Intel 8086 processor contains a 16-bit register. Some registers also serve for special purposes in addition to being general purposes.

- AX - the accumulator register (divided into AH / AL).
- BX - the base address register (divided into BH / BL).
- CX - the count register (divided into CH / CL).
- DX - the data register (divided into DH / DL).
- SI - source index register.
- DI - destination index register.
- BP - base pointer.
- SP - stack pointer

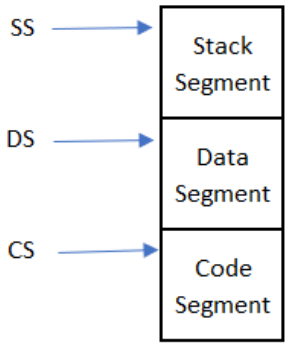
Despite the name of a register, it's the programmer who determines the usage of each general-purpose register. The main purpose of a register is to keep a number. The size of the above registers is 16-bit. It's something like: 0011000000111001b (in binary form), or 12345 in decimal (human) form.

Four general-purpose registers (AX, BX, CX, DX) are made of two separate 8-bit registers. For example, if AX = 0011000000111001b, then AH = 00110000b and AL = 00111001b. Therefore, when you modify any of the 8-bit registers, the 16-bit register is also updated, and

vice-versa. The same is true for the other 3 registers. "H" is for the high, and "L" is for the low part.

Besides general-purpose registers, there are some segment registers. They have a very special purpose as mentioned below.

- CS - points at the segment containing the current program.
- DS - generally points at the segment where variables are defined.
- ES - extra segment register, it's up to a coder to define its usage.
- SS - points at the segment containing the stack.



The 8086 processor has a 20-bit address bus that can address up to 1 MB of memory. However, all the registers inside the processor are 16-bit. Therefore, a physical address cannot be stored in any register completely and hence is converted logical address containing SEGMENT: OFFSET fields. Segment and offset are both 16-bit fields. The physical address is calculated by calculating SEGMENT x 10h + offset.

The physical address for the code segment is always formed using CS: IP. Once the instruction is executed, the IP is incremented by the size of the instructions. In 8086, the instructions vary in size.

A segment is an area of memory that includes up to 64K bytes and begins on an address evenly divisible by 16 (such as an address that ends in 0h).

Basic Instructions

1. MOV Destination operand, Source Operand

   mov instruction copies the contents of the source operand to the destination operand.

2. ADD Destination operand, Source Operand

   ADD instruction adds the contents of source and destination operands and stores results back to the destination operand.

3. SUB Destination operand, Source Operand

   SUB instruction subtracts the contents of the source operand from the destination operand and stores the result back to the destination operand.

Basic Rules:

- Both source and destination operands should be the same size.
- Both operands cannot be segment registers.
- The destination operand cannot be a CS or an IP register.
- If the destination operand is a segment register, the source operand cannot be an immediate value.

First assembly language program

The assembly language program

```
.model small

.stack 100h

.data

.code

Mov ax, 100h
Mov bx, 200h
Add ax, bx
Sub  ax, 25h


.exit
```

The description of the above program in shown in the following table.

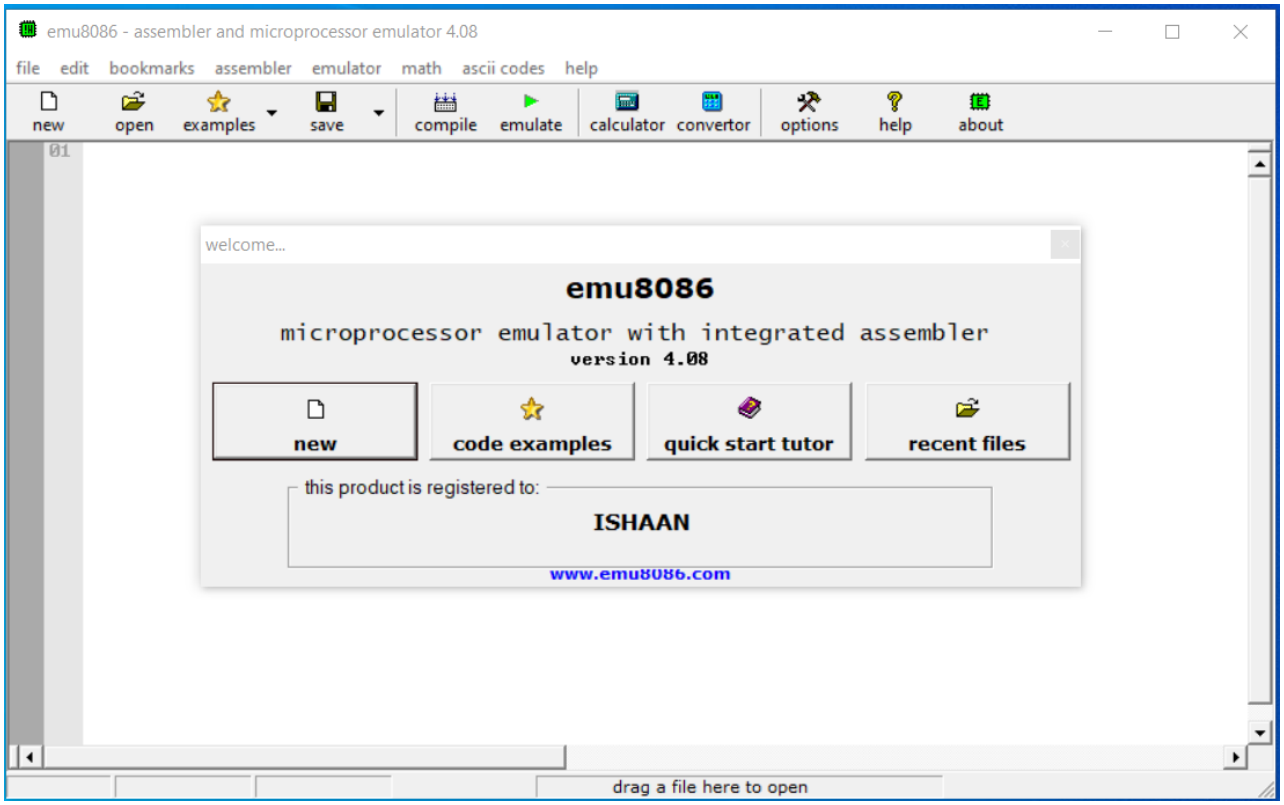
Instruction	Description
Mov ax,100h	Ax ← 100h
Mov bx,200h	Bx ← 200h
Add ax, bx	Ax ← Ax + Bx
Sub ax, 25h	Ax ← Ax – 25h

Emu8086 Tutorial Step-by-Step

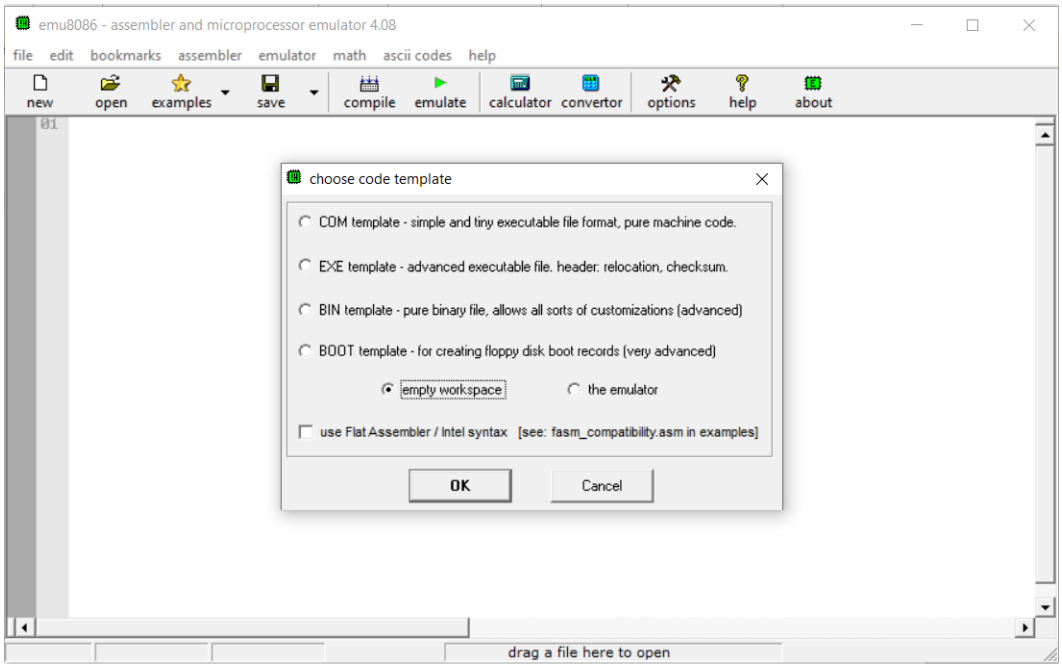
Step-1

Double-click on the icon on the desktop 

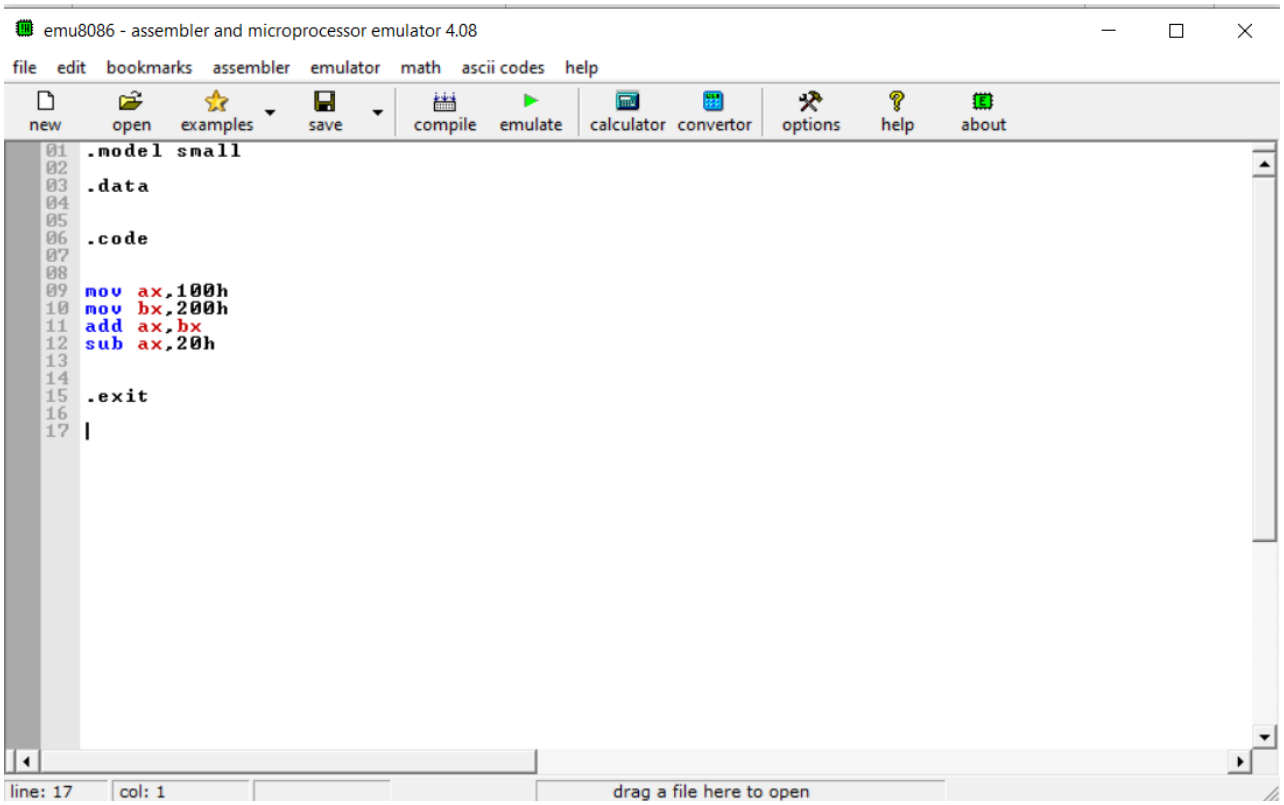
Step-2
The following window will appear. Click on "new".



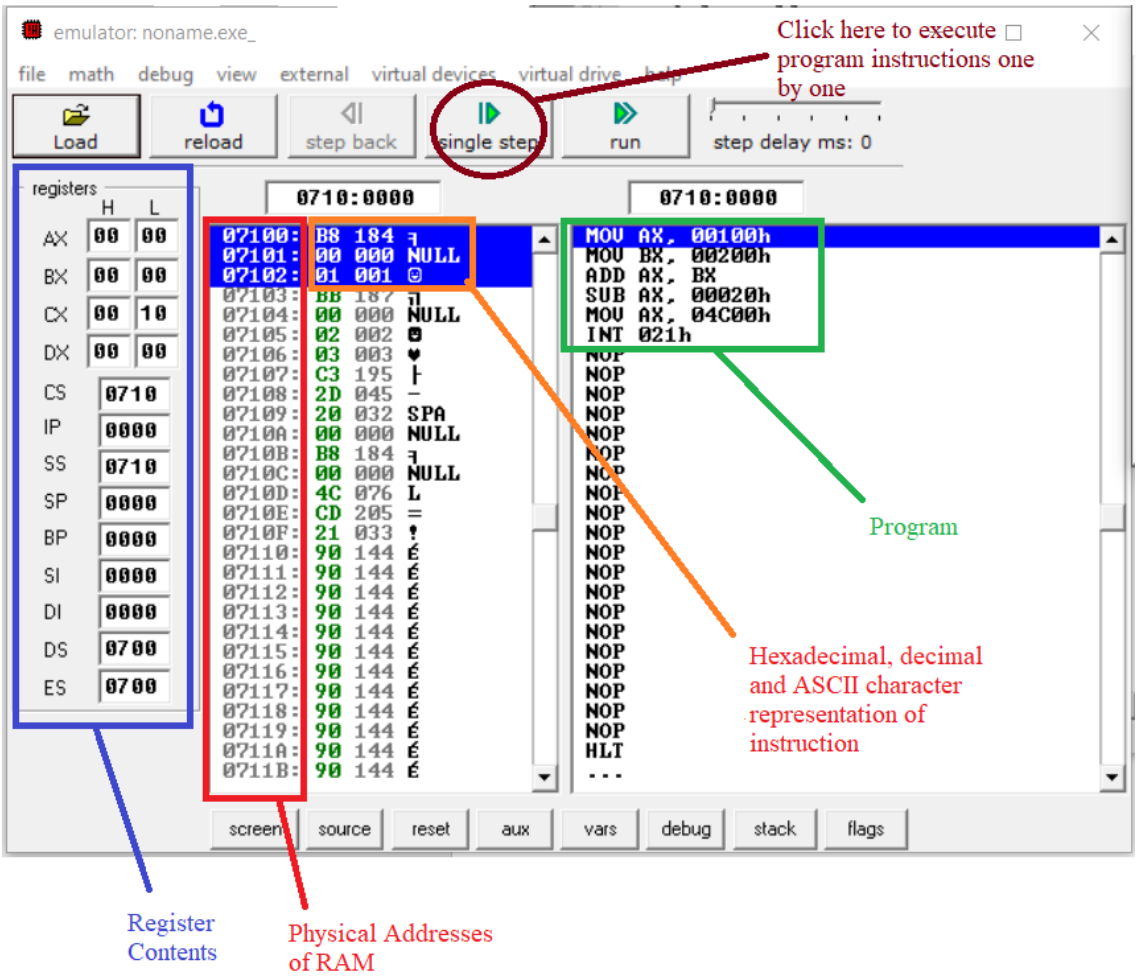
Step-3
Click on the empty workspace and press "OK".



Step-4
Type the code given above and clock on emulate.



Step-5
Keep clicking on "Single step" to execute program instructions one by one



Step-6
Run a complete program and observe the value of various registers.

Observation:

- Observe how the physical address is being calculated. i.e., **CS : IP**
- Observe what number is added to the IP register after the execution of each instruction. Is that number constant? If not, why?

Lab-2 Exercise

Task-1

Which of the following instructions is invalid? Give reasons.

Sr#	Instructions	Status (Valid/Invalid)	Reasons
1.	MOV AX, 27		
2.	MOV AL, 97Fh		
3.	MOV SI, 9516		
4.	MOV DS, BX		
5.	MOV BX, CS		
6.	MOV AX, 23FB9h		
7.	MOV DS, BH		
8.	MOV DS, 9BF2		
9.	MOV CS, 3490		
10.	MOV DS, ES		
11.	MOV ES, BX		

Task-2

Write a program in assembly language that calculates the square of six by adding six to the accumulator six times.

Task-3

Write a program to solve the following equation.

DX = AX + BH – CL + DX

Initialize the AX, BX, CX, and DX registers with 0100h, 55ABh, 0A11h and 0001h values, respectively.

NOTE: *There is no mistake in Task-3 statement, so please solve it as provided.*

Lab 3: Addressing Modes

Learning Outcomes

- Students will learn basic addressing modes.
- Students will come to know about little-endian and big-endian notations.
- Students will be able to access any physical location of memory for data.

When we run a program, the operating system locates the complete program on disk and loads (copies) it to the RAM. It also initializes the value of the CS and SS registers with the starting addresses of the code and stack segments. However, it does not initialize the DS segment. The DS value (and ES if used) must be initialized by the program to access memory for data. This is done as follows:

```
MOV AX, @DATA
MOV DS, AX
```

Here, @DATA refers to the start of the data segment and is replaced by a number. Since we cannot assign a number directly to segment registers, therefore, we must first assign it to a general-purpose register and then from that general-purpose register to a segment register.

Addressing Modes

Immediate addressing mode

In the immediate addressing mode, the source operand is constant. In immediate addressing mode, as the name implies, when the instruction is assembled, the operand comes immediately after the opcode. For this reason, this addressing mode executes quickly. However, in programming, it has limited use. Immediate addressing mode can be used to load information into general-purpose registers.

Examples:

```
MOV AX,2550H
MOV CX,625
MOV BL,40H
```

Register addressing mode.

The register addressing mode involves the use of registers to hold the data to be manipulated. Memory is not accessed when this addressing mode is executed; therefore, it is relatively fast.

Examples:

```
ADD BX, DX
MOV ES, AX
MOV AL, BH
```

Direct Addressing mode

In the direct addressing mode, the data is in some memory location(s) and the address of the data in memory comes immediately after the instruction. Note that in immediate addressing, the operand itself is provided with the instruction, whereas in direct addressing mode, the address of the operand is provided with the instruction. This address is an offset.

```
MOV DL, [2400h]          ; move contents of DS:2400H into DL
```

The physical address is calculated by DS x 10h + 2400h.

Register indirect addressing mode

In the register indirect addressing mode, the address of the memory location where the operand resides is held by a register. The registers used for this purpose are SI, DI, BX, and BP.

For example:

```
MOV AL, [BX]          ;moves the contents of the memory location into AL,
                      the memory location pointed to  DS:BX.
```

- The physical address is calculated by DS x 10h + BX. The same rules apply when using register SI or DI.
- BP register can also be used as a pointer register. However, the physical address will be calculated by SS x 10h + BP.
- AX, CX, DX, and SP cannot be used as pointer registers/ offset.

We can override the segment register while using BX, SI, DI, and BP registers as pointers by writing the preferred segment register name with it as shown below.

```
MOV AL, ES:[BX]
MOV AL, DS:[BP]
MOV AL, CS:[SI]
MOV AL, SS:[DI]
```

The size of a register in an instruction specifies how many bytes will be read or written from or to memory. If a register is not there, one byte is accessed from memory by default in Emu8086. However, to avoid confusion, one must specify the number of bytes to read or write to memory using the following:

- **byte ptr** - for byte.
- **word ptr** - for word (two bytes).

For example:

MOV byte ptr [2400h], 1

ADD word ptr [2400h], 2

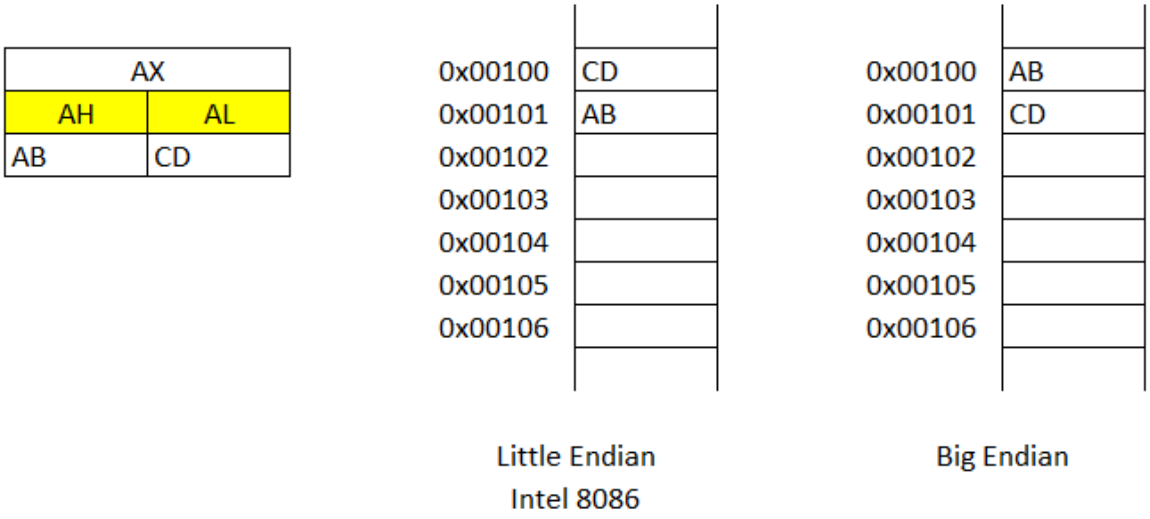
Storing multi-byte data in RAM

Little-endian and big-endian are the two ways of storing data in memory. In little-endian, the lower byte is stored at a lower address and the higher byte is stored at a higher address. However, in big-endian, the lower byte is stored at the higher address and the higher byte is stored at the lower address.

Let the AX register contain ‘0xABCD’ and the DS register contain "0000h". The following instruction will write the contents of the AX register into the memory and start writing from physical location 0x00100 onwards. Since the Intel 8086 processor is based on little-endian, it will store the lower byte ‘CD’ at 0x00100 and higher byte ‘AB’ at 0x00101 as shown in the following figure. However, the processor that is based on big-endian will store it the other way around. Memory will be read in the same way.

Instruction:

MOV [0100h], AX



Accessing a specific physical location of RAM

Using the concepts above, one can access any physical location of memory. The physical address is a 20-bit value that has to be converted into Segment: Offset in a way such that when the processor combines it to form a physical address, it should be in the same location.

To write a byte "0x12" to the physical address 0xABCDE of RAM, one needs to break the physical address into Segment and Offset parts each of 16-bit. One combination is:

Segment : Offset
0ABCDh : 000Eh.

```
.code

   mov ax,0abcdh
   mov ds,ax
   mov bx,000eh
   mov byte ptr [bx],012h
```

Similarly, to write a word with the physical address "0xABCDE", the prefix: byte ptr will be replaced with word ptr.

```
   mov ax,0abcdh
   mov ds,ax
   mov bx,000eh
   mov word ptr [bx],012h
```

Program to write 0x1234 physical memory "0xABCDE".

```
.model small

.data

.code

mov bx,0abcdh
mov es,bx
mov word ptr es:[000eh],01234h

.exit
```

The description of the above program is shown in the following table.

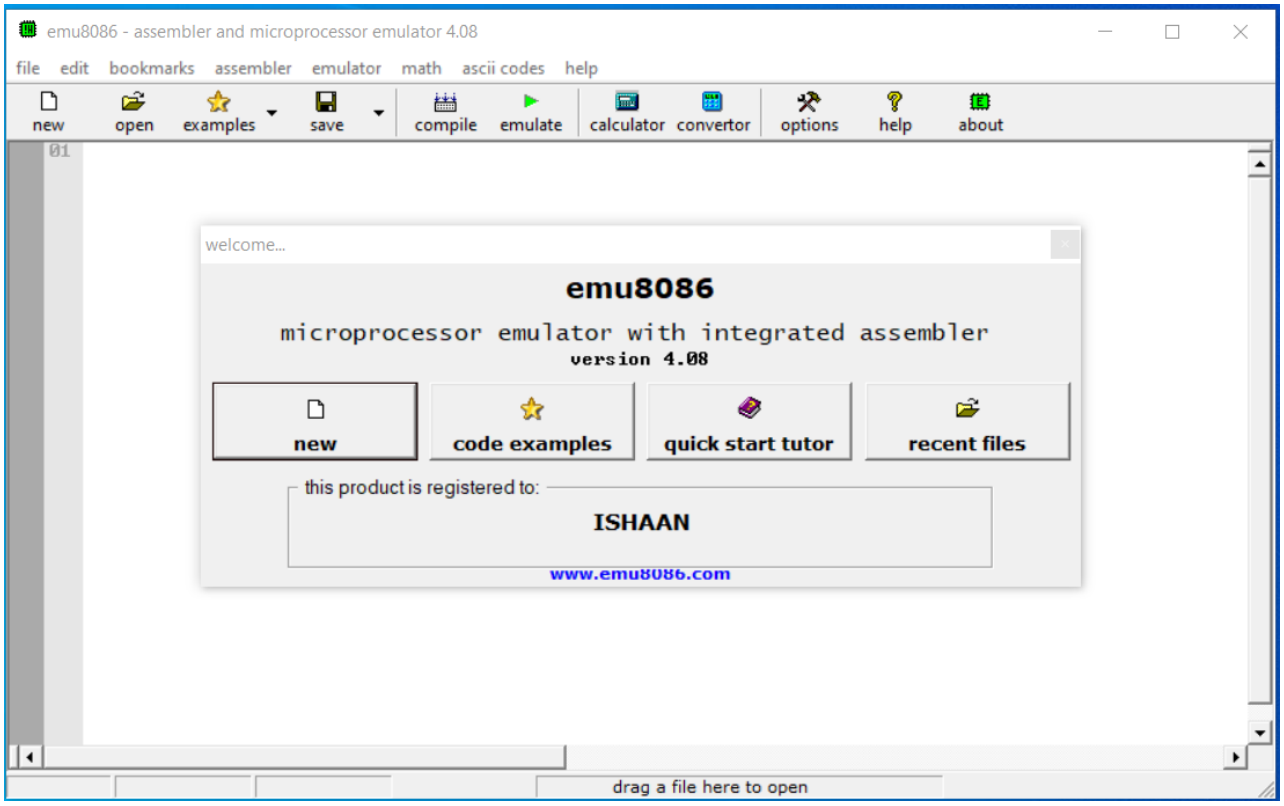
Instruction	Description
mov bx,0abcdh	Moving segment address to bx before moving it to ES
mov es,bx	Moving segment address to ES
mov word ptr es:[000eh]	Writing word “0x1234” to physical memory 0xABCDE

Emu8086 Tutorial Step-by-Step

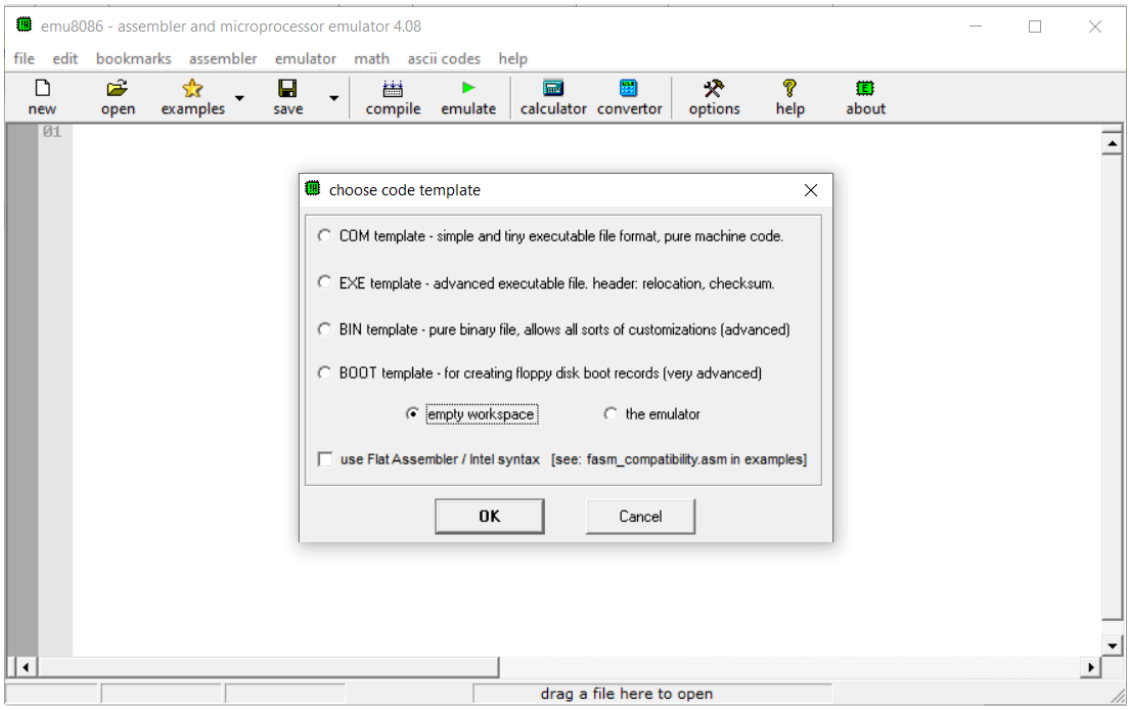
Step-1

Double-click on the icon on the desktop 

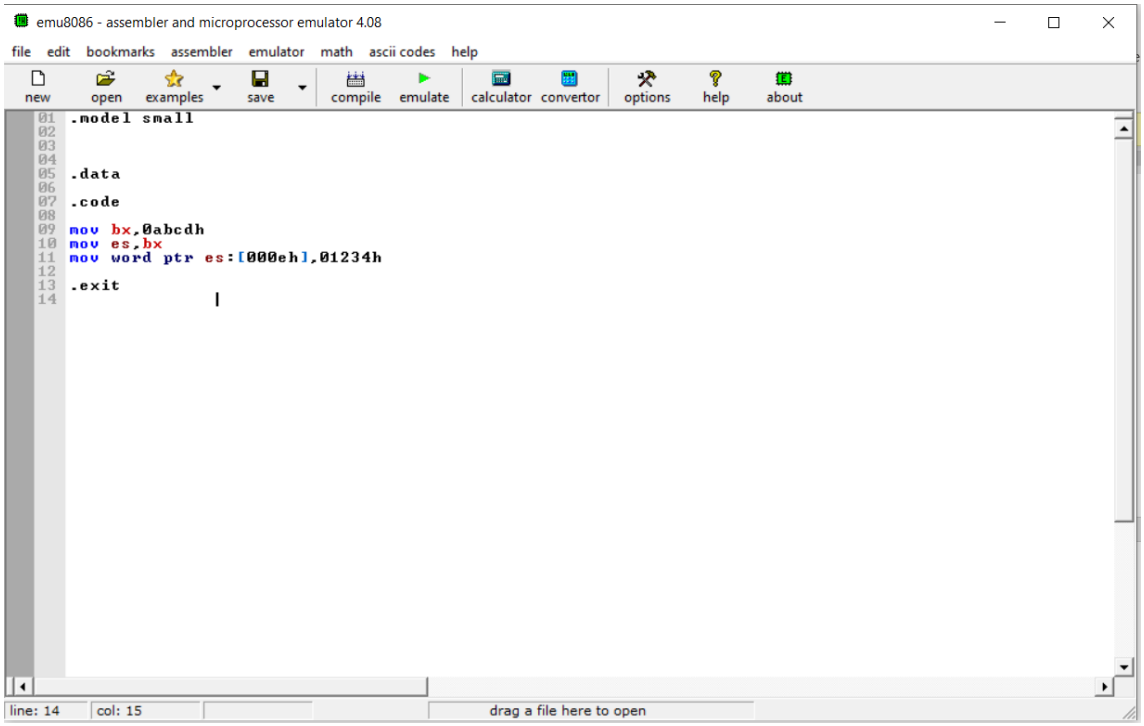
Step-2
The following window will appear. Click on “new”.



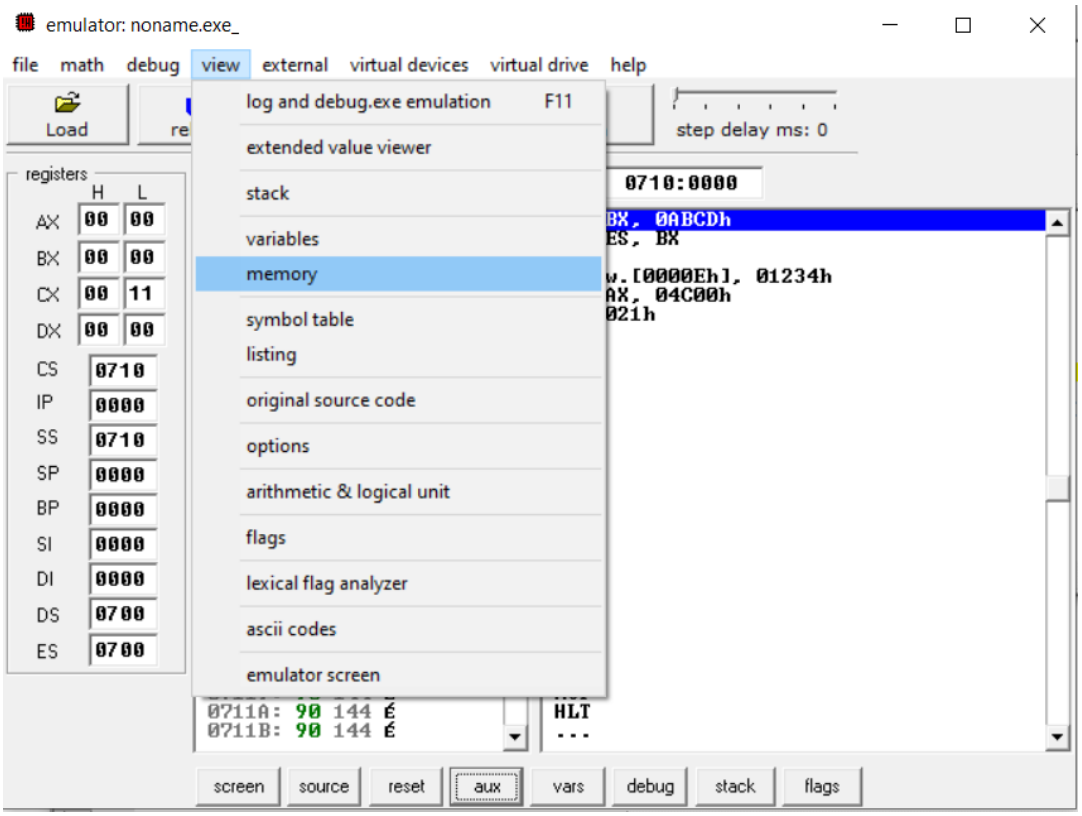
Step-3
Click on the empty workspace and press "OK".



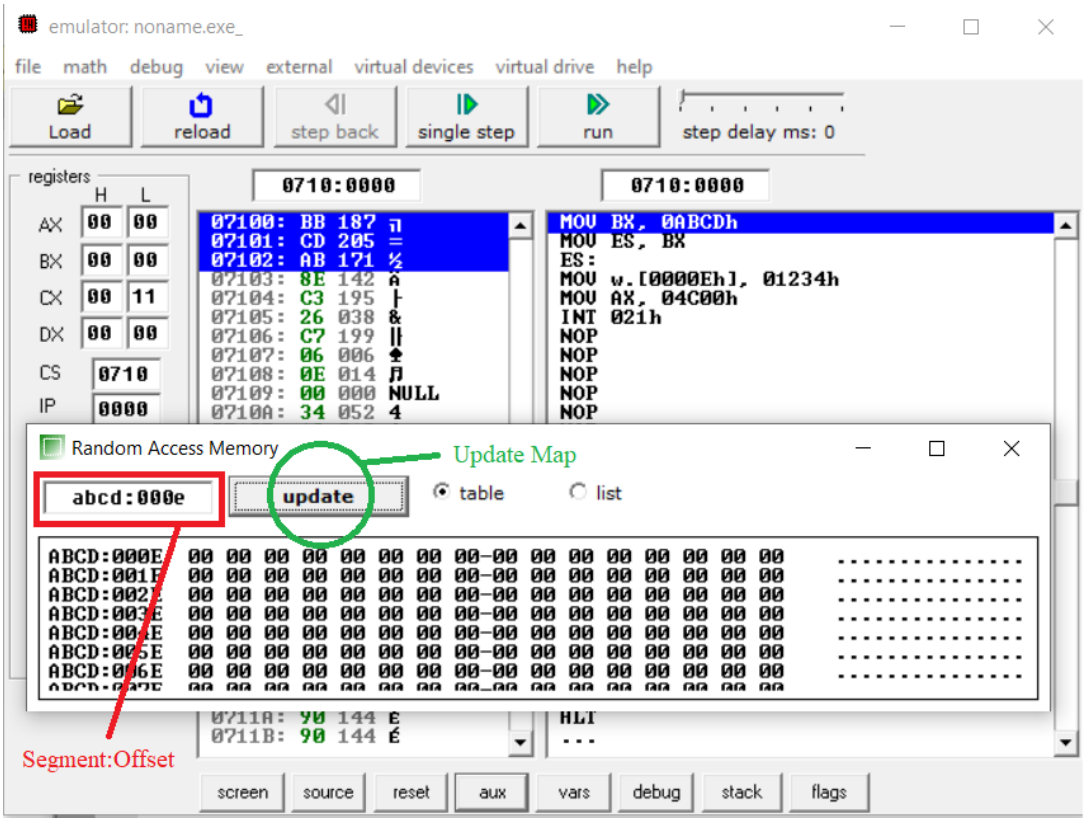
Step-4
Type the code given above and click on "emulate".



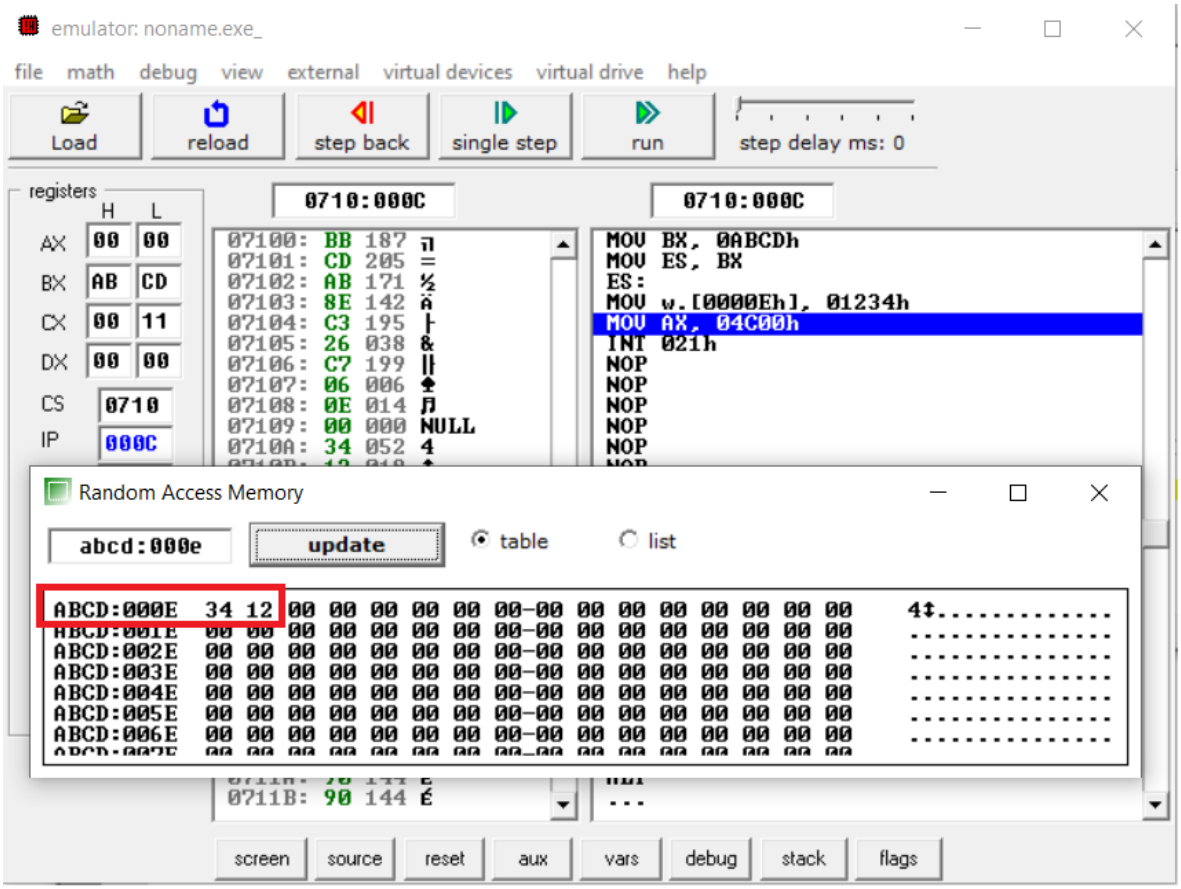
Step-5
Click on "Memory" from the view menu!



Step-6
Write the logical address of the desired part of the segment that you want to view.



Step-7
Keep clicking on "Single step" to execute program instructions one by one.
(The mov word ptr es:[000eh],01234h instruction will write two bytes to memory.)



Observation:

- Which byte is written to what address?

Lab-3 Exercise

Task-1

Write a program that stores the following numbers into the current data segment at offset: 0x1000 onwards. The program then calculates the sum of these numbers and stores it in the DX register, then stores the value of DX at the physical location: 0xCD1F3.

Numbers: 0x1F01, 0xA0EF, 0x7704, 0x34B0, 0x2250

Task-2

The "HLT" has a machine code of "0xF4".

Insert this instruction before the first instruction of your program implemented in Task-1.

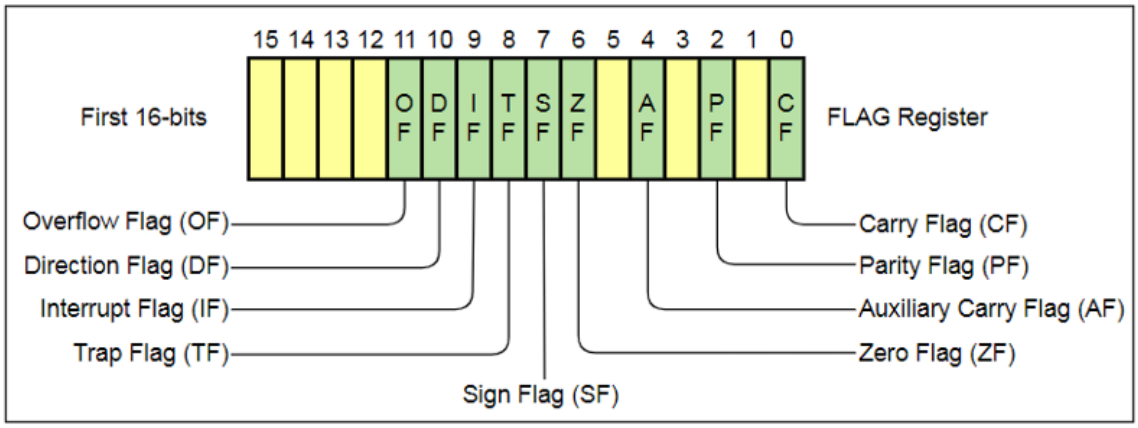
Lab 4: ALU Operations

Learning Outcomes

- Students will learn about various flags and their purposes.
- Students will learn various arithmetic and logic operations.
- Students will be able to perform arithmetic operations and logic operations on n-bit numbers.
- Students will learn what instructions affect which flags.

Flag Register

The 8086 processor contains a 16-bit flag register where nine bits are used as flags as shown in the following figure. Out of which, 6 flags indicate the status of recently executed instruction, while the remaining 3 are control flags.



Flag registers of 8086 microprocessor

Status Flag Registers:

1. Overflow Flag (OF):

The overflow Flag is set to **1** when there is a **signed overflow**. For example, when you add bytes **100 + 50** (result is not in range -128…127).

Example:

On adding bytes 100 + 50 (result is not in range -128…127), so overflow flag will set.

```
MOV AL, 100 (100 is 0110 0100 which is positive)
MOV BL, 50 (50 is 0011 0010 which is positive)
ADD AL, BL (150 is 1001 0110 which is negative)
```

Overflow flag became set as we added 2 +ve numbers and we got a -ve number.

2. Sign Flag (SF):

The sign Flag is set to **1** when the result is **negative**. When the result is **positive** it is set to **0**. This flag takes the value of the most significant bit.

3. Zero Flag (ZF):

Zero Flag (ZF) is set to **1** when the result is **zero**. For non-zero results, this flag is set to **0**.

4. Auxiliary Flag (AF):

Auxiliary Flag is set to **1** when there is an **unsigned overflow** for low nibble (4 bits).

5. Parity Flag (PF):

The parity Flag is set to **1** when there is an even number of ones in the result and set to **0** when there is an odd number of ones.

6. Carry Flag (CF):

Carry Flag is set to **1** when there is an **unsigned overflow**.

Example

When you add bytes **255 + 1** (result is not in range 0…255). When there is no overflow, this flag is set to **0**.

Control Flag Registers:
1. Direction Flag (DF):

Direction Flag is used by some instructions to process data chains, when this flag is set to **0** – the processing is done forward, when this flag is set to **1** the processing is done backward.

2. Interrupt Enable Flag (IF):

When the Interrupt Enable Flag is set to **1** CPU reacts to interrupts from external devices.

3. Trap Flag (TF):

Trap Flag is used for on-chip debugging.

Arithmetic and Logic Operations

The following table lists various arithmetic and logical operations, their descriptions, and their effects on flags.

Instructions **D – Destination Operand** **S – Source Operand**	**Description**	**Flag status:** **0 – Clear** **1 – Set** **? – Unknown** **r – Depends on the Result**
ADD D, S	**D ← D + S**	C Z S O P A r r r r r r
ADC D, S	**D ← D + S + CF**	C Z S O P A r r r r r r
SUB D, S	**D ← D – S**	C Z S O P A r r r r r r
SBB D, S	**D ← D – S – CF**	C Z S O P A r r r r r r
AND D, S	**D ← D AND S**	C Z S O P 0 r r 0 r
OR D, S	**D ← D OR S**	C Z S O P A 0 r r 0 r ?
NOT D	**D ← NOT D**	C Z S O P A Unchanged
INC D	**D ← D + 1**	Z S O P A r r r r r CF - Unchanged
DEC D	**D ← D - 1**	Z S O P A r r r r r CF - Unchanged
XOR D,S	**D ← D XOR 1**	C Z S O P A 0 r r 0 r ?
NEG D	**D ← (NOT D) + 1**	Z S O P A r r r r r CF - Unchanged

Program to add 100 and 50 to check status of various flags.

```
.model small

.data

.code

mov al, 100
add  al, 50

.exit
```

The description of the above program in shown in the following table.

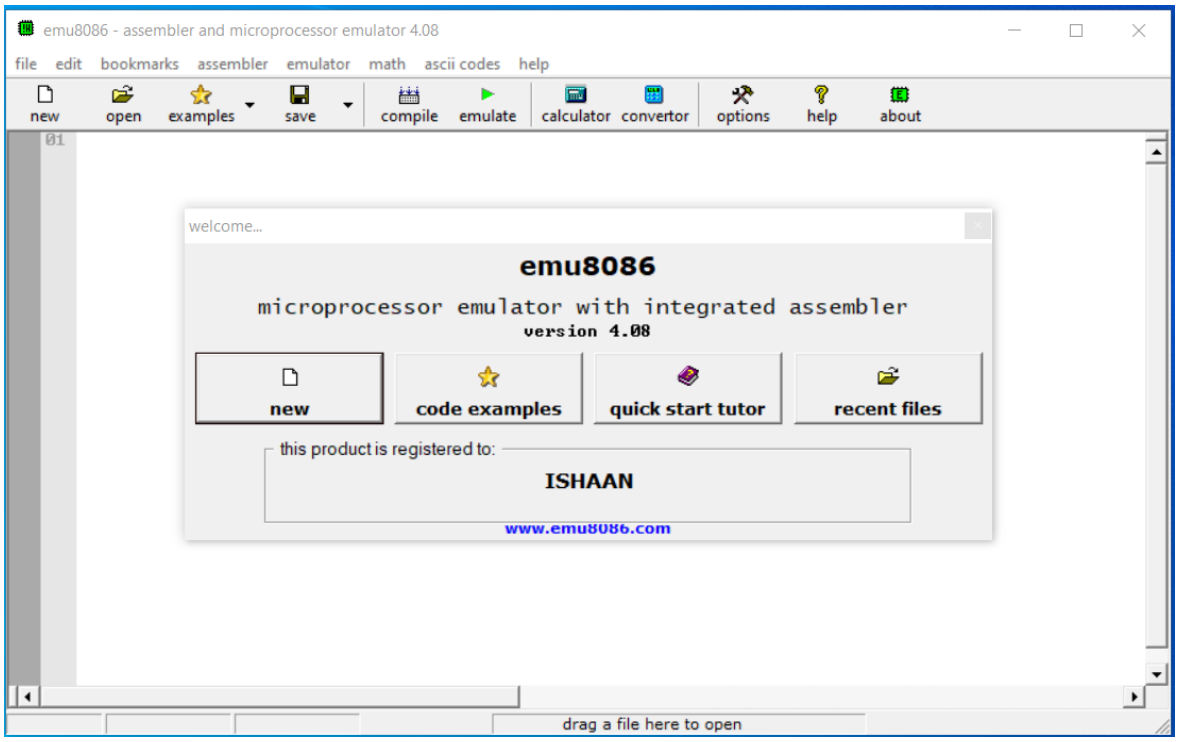
Instruction	Description
mov al,100	Moving value:100 to the AL register
add al, 50	Adding value: 50 to the contents of the AL register and storing the result back to the AL register The sum of 150 exceeds the range of 8-bit signed numbers and will set the overflow (OF) flag. The sum of 150 will not affect the carry flag (CF) as the range for 8-bit unsigned numbers is 0–255. The number of ones in 150 is even, which will set the parity flag (PF). The sign flag (SF) will also be set as the most significant bit of the result is 1.

Emu8086 Tutorial Step-by-Step

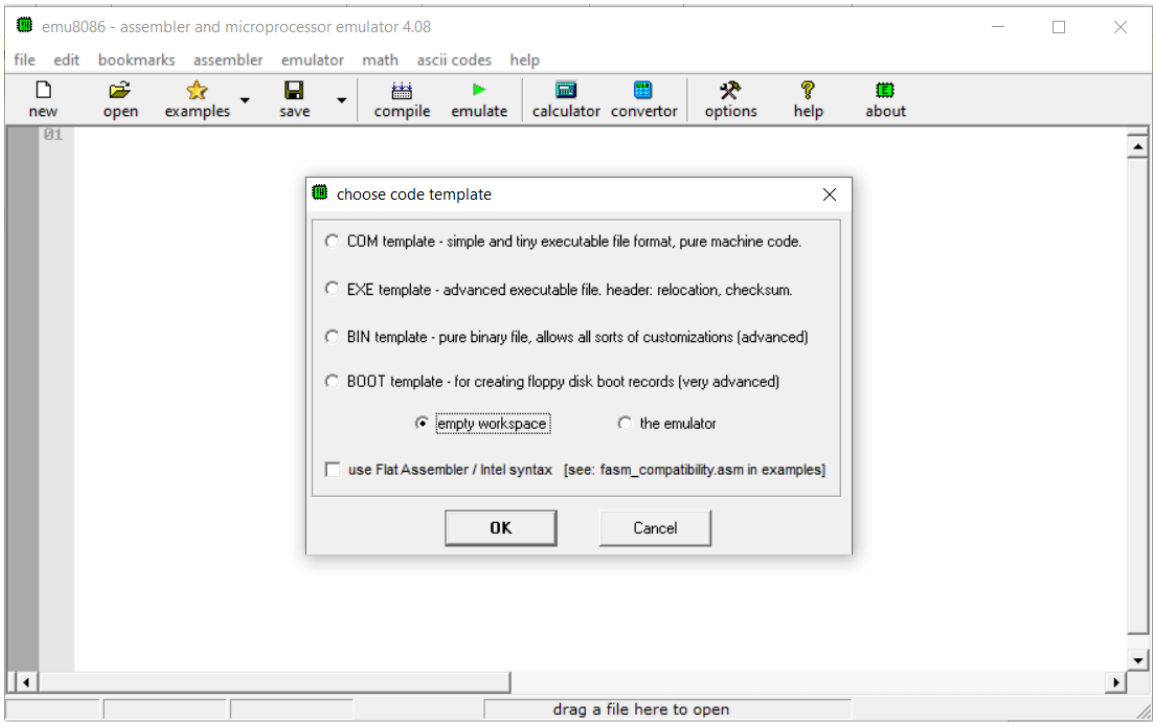
Step-1

Double-click on the icon on the desktop 

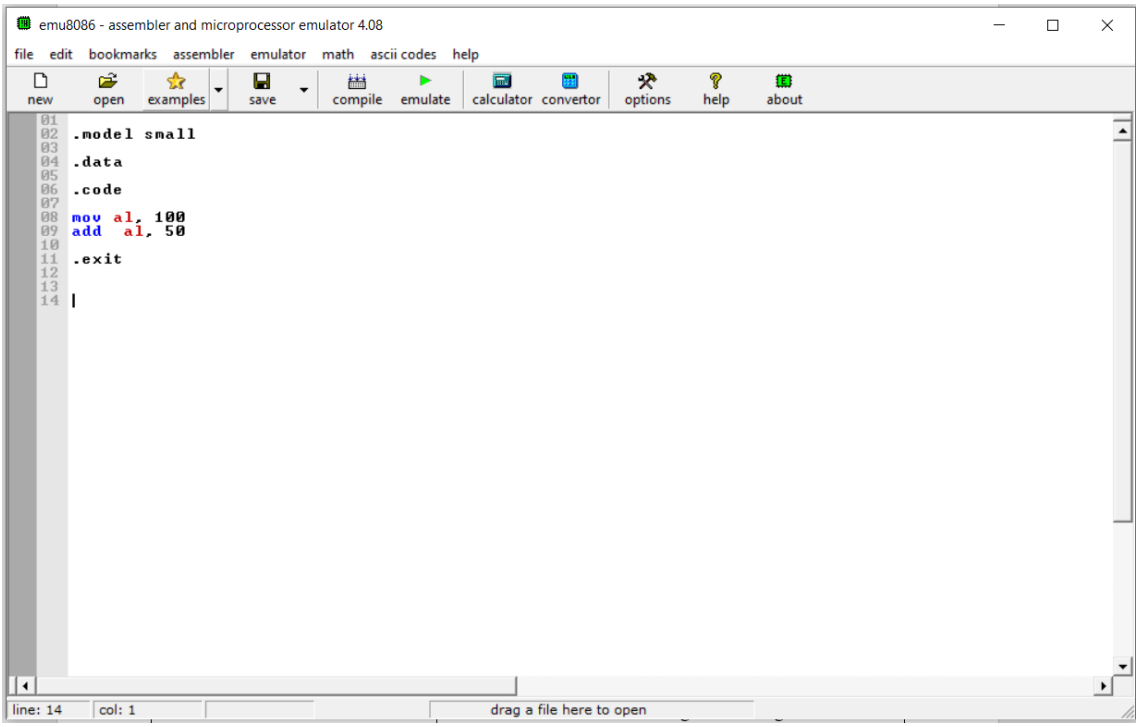
Step-2
The following window will appear. Click on “new”.



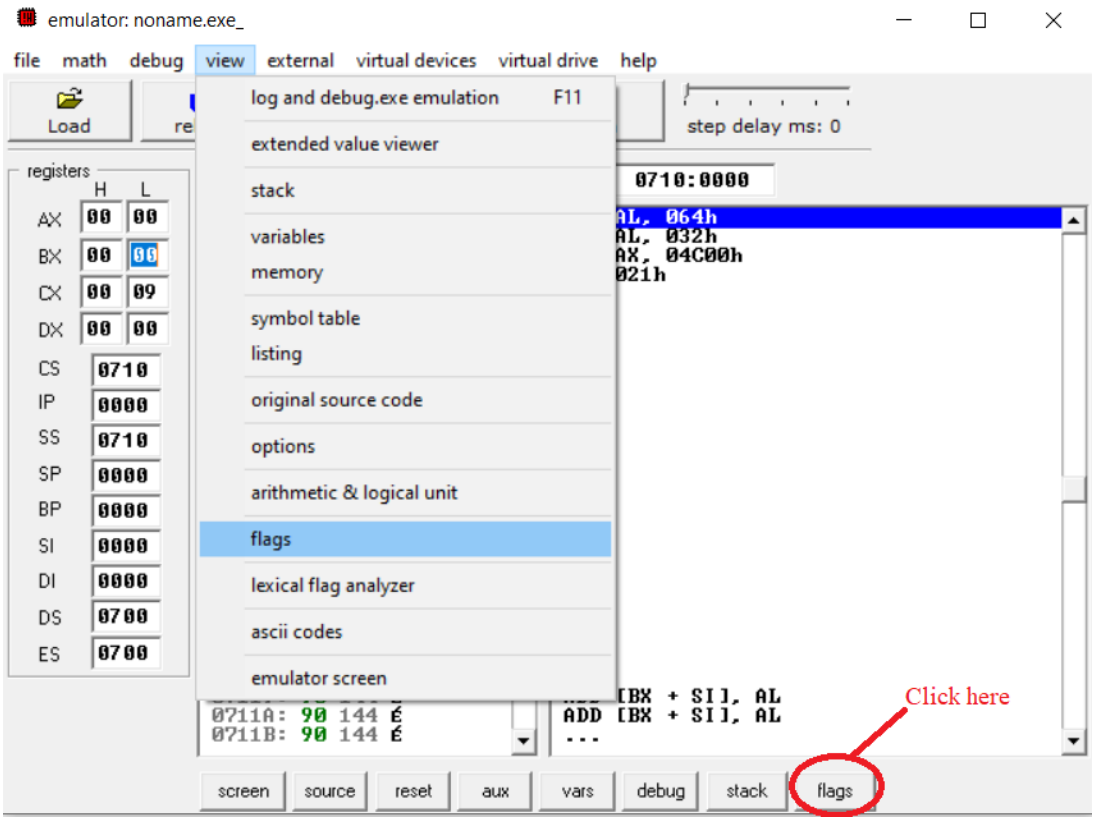
Step-3
Click on the empty workspace and press “OK”.



Step-4
Type the code given above and click on "emulate".

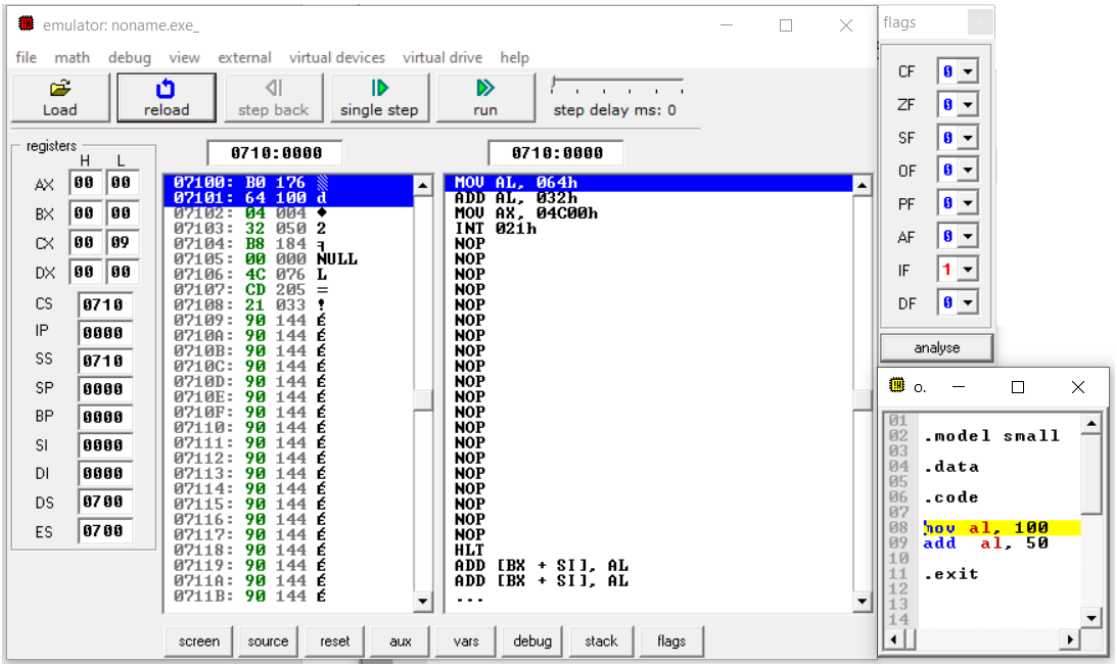


Step-5
Click on "flags" from the view menu OR click on the button "flags" at the bottom.



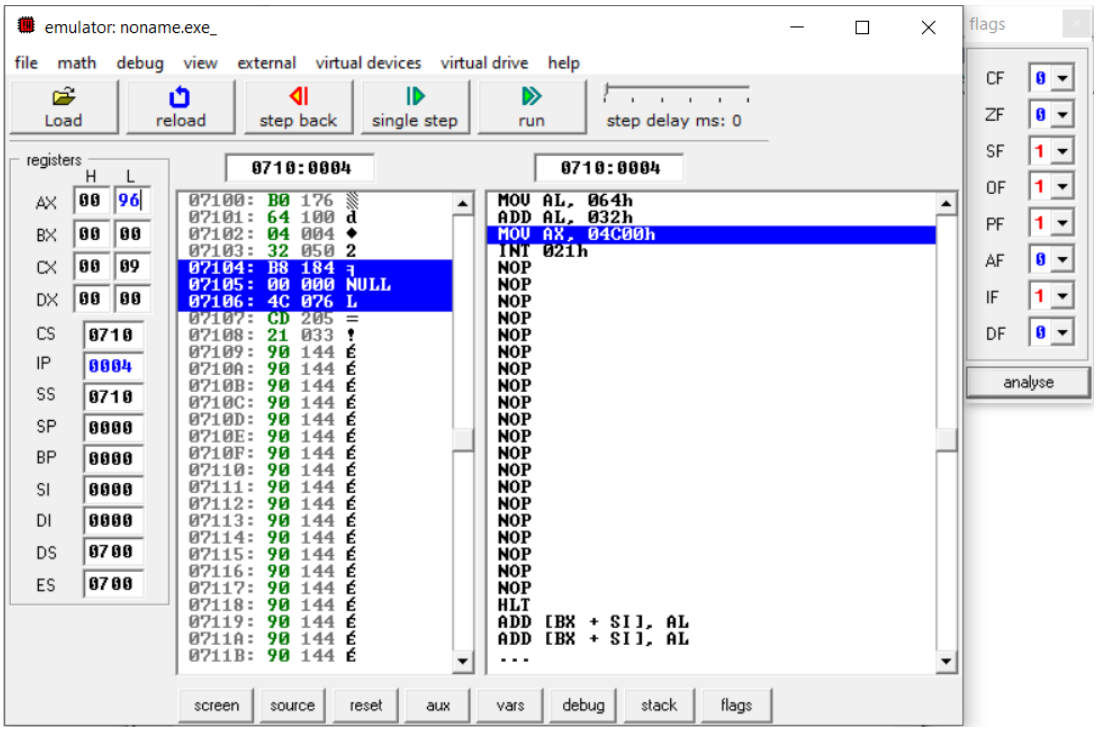
Step-6

Keep clicking on "Single step" to execute program instructions one by one. Stop clicking "Single step" just after the "Add al, 50" instruction to observe various flags.



Step-7

The add al, 50 instruction sets the SF, OF, and PF.



Observation:

- Why are the ZF and AF zero?

Lab-4 Exercise

Task-1

Write a program that stores the given two 32-bit numbers into the current data segment at offset: 0x1000 and 0x1008, respectively. The program then calculates the sum of these numbers and stores it at the offset: 0x1010.

Numbers: 0x1F540398, 0xC0A1F02E

Task-2

Write a program to implement the following equation.

X = ~0xFF12 ^ {0xABFF & (0x2113 | 0x2340)}

~	Invert all bits
^	Bitwise XOR
&	Bitwise AND
|	Bitwise OR

Task-3

Perform any ALU operation that sets CF and OF at the same time.

Lab 5: Variables, Loops, Arrays & 2d Arrays

Learning Outcomes

- Students will be able to declare and initialize variables.
- Students will be able to declare and initialize a linear array.
- Students will be able to define a constant.
- Students will be able to implement built-in loop instructions.
- Students will be able to apply base plus index addressing mode to access arrays.
- Students will be able to traverse arrays and perform various operations on them.
- Students will know how a 2D array is stored in memory.
- Students will know how a 2D array can be accessed from memory.
- Students will be able to perform various operations on the applications of 2D arrays.

Variables

A variable is a memory location. It is easier for a programmer to remember a variable name like "Var1" than an address like **5A73:235B**, especially when there are 10 or more variables. Emu8086 supports bytes and words as the DB and DW variables, respectively.

Syntax for a variable declaration:

name **DB** value

name **DW** value

DB - for Define Byte.
DW - for Define Word.

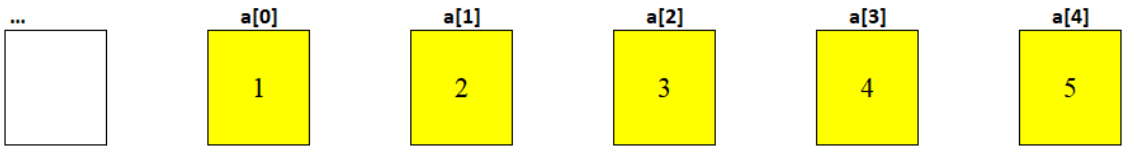
name - can be any letter or digit combination, though it should start with a letter. It's possible to declare unnamed variables by not specifying the name (this variable will have an address but no name).

value - can be any numeric value in any supported numbering system (hexadecimal, binary, or decimal), or the "?" symbol for variables that are not initialized.

Arrays

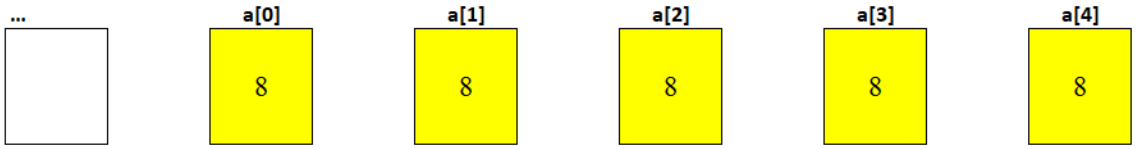
Arrays can be seen as chains of variables. It is a set of consecutive memory locations having the same data type. Declaring and initializing an array

a **db** 1,2,3,4,5



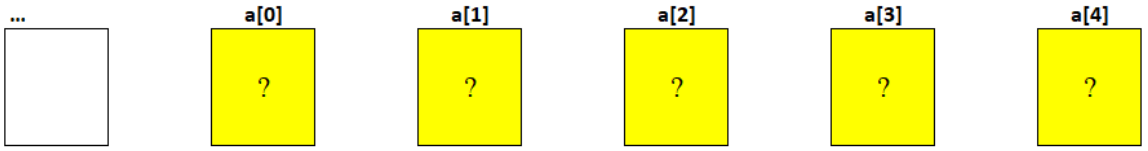
Declaring array of 5 elements and initializing it with 8.

a **db** 5 dup(8)



Declaring array of 5 elements without initializing it.

a **db** 5 dup(?)



Constant

Constants are just like variables, but they exist only until your program is assembled. After assembling, the definition of a constant is replaced with its value. Constant is defined as follows:

name EQU < any expression >

Example:

```
.data
k EQU 5
.code
MOV AX, k
```

The above example is functionally identical to code: MOV AX, 5

Base-plus-Index addressing mode

Arrays can be accessed using direct or register-indirect addressing modes. However, the base-plus-index addressing mode facilitates the programmer in accessing arrays better. In this addressing mode, as the name suggests, there is a base register that points to the base address of an array, while the index register points to the index of the array that is to be accessed. The offset part of the logical address is made by adding the contents of the base and index registers.

There are two base registers: BX and BP, and two index registers: SI and DI. One base register and one index register must be combined to form an offset. There cannot be two base registers or two index registers. If BX is used as the base register, segment addresses will be taken from DS by default. However, in the case of BP, the segment address will be taken from SS by default. However, segment registers can be overridden, as we have already seen in previous labs. The following table shows how physical addresses are calculated.

Instructions	Default Segment	Physical Address Calculation
MOV AX, [BX + SI]	DS	DS:[BX+SI]
MOV AX, [BP + SI]	SS	SS:[BP + DI]

Program 1:
Program to move elements of array to AL, AH, CL, CH and DL registers.

```
.model small

.data
Array db 1,2,3,4,5

.code
Mov ax,@data
Mov ds,ax

Mov bx, offset Array
Mov si, 0

Mov al, [bx + si]
Inc si

Mov ah, [bx + si]
Inc si

Mov cl, [bx + si]
Inc si

Mov ch, [bx + si]
Inc si

Mov dl, [bx + si]

.exit
```

- The **offset** keyword is used to get the address of a variable or an array.
- **SI** is incremented by 1 in the case of a byte array. It will be incremented by 2 for the word array.

Labels

A label is an identifier that is optional and can be placed at the beginning of an instruction. When a program is assembled, the reference to the label is replaced by the offset address.

Example:

```
L1: Mov ax,bx

    Mov bx, L1
```

Loops

Loops are used to execute a set of instructions multiple times until specific conditions are met. There are some built-in loop instructions. One of them is "Loop," which transfers the control to the label specified with it if the counter register (CX) is not zero.

Instruction	Description

Loop label	CX ← CX – 1
	
	If (CX ! = 0)
	Jump to label
	else
	Jump to next instruction

Program 2:
Program to increment AX and decrement DX register 100 times

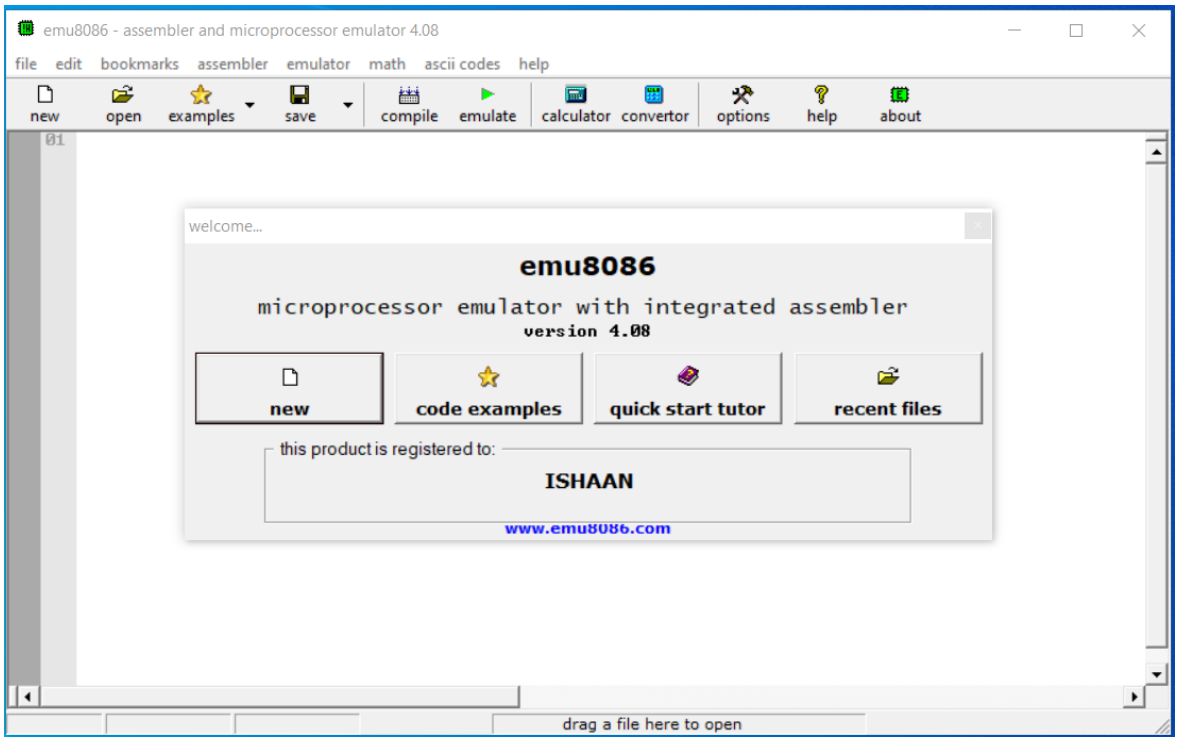
```
.model small

.data

.code
Mov DX,0xffff
Mov CX,100

L1:

Inc AX
Dec DX

Loop L1

.exit
```

Emu8086 Tutorial Step-by-Step

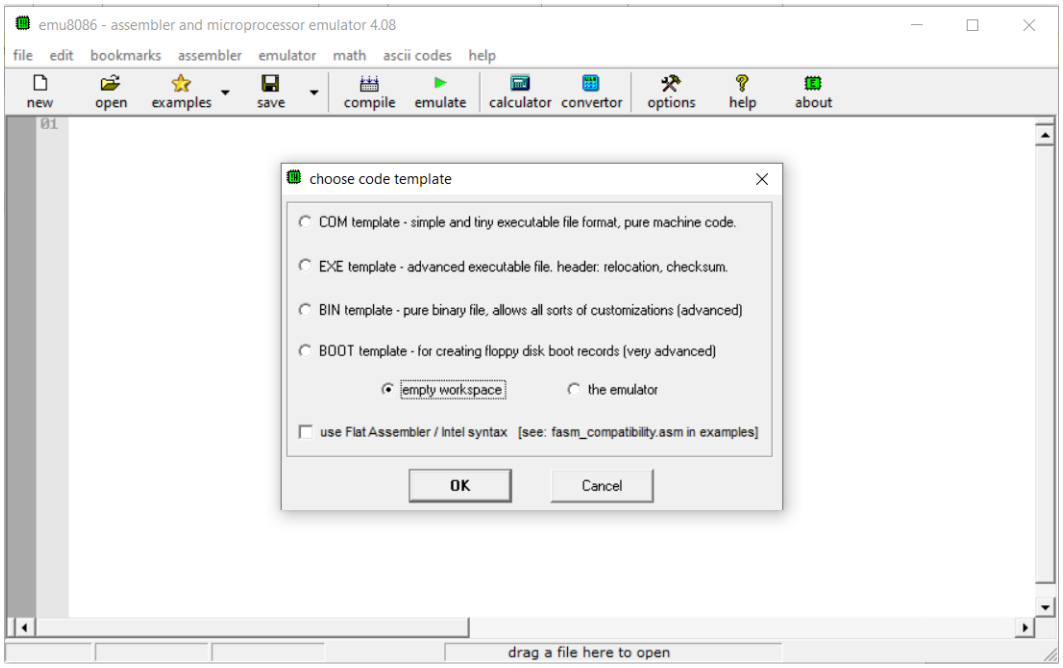
Step-1:

Double-click on the icon on the desktop 

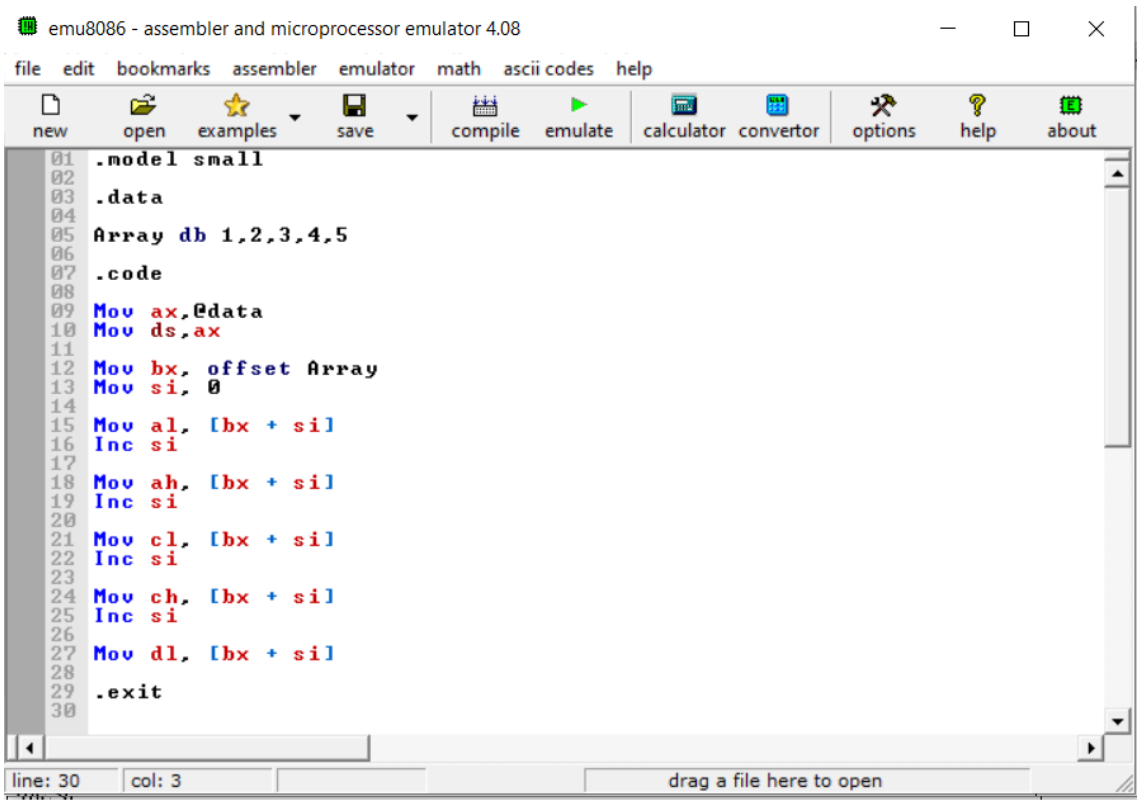
Step-2:
The following window will appear. Click on "new".



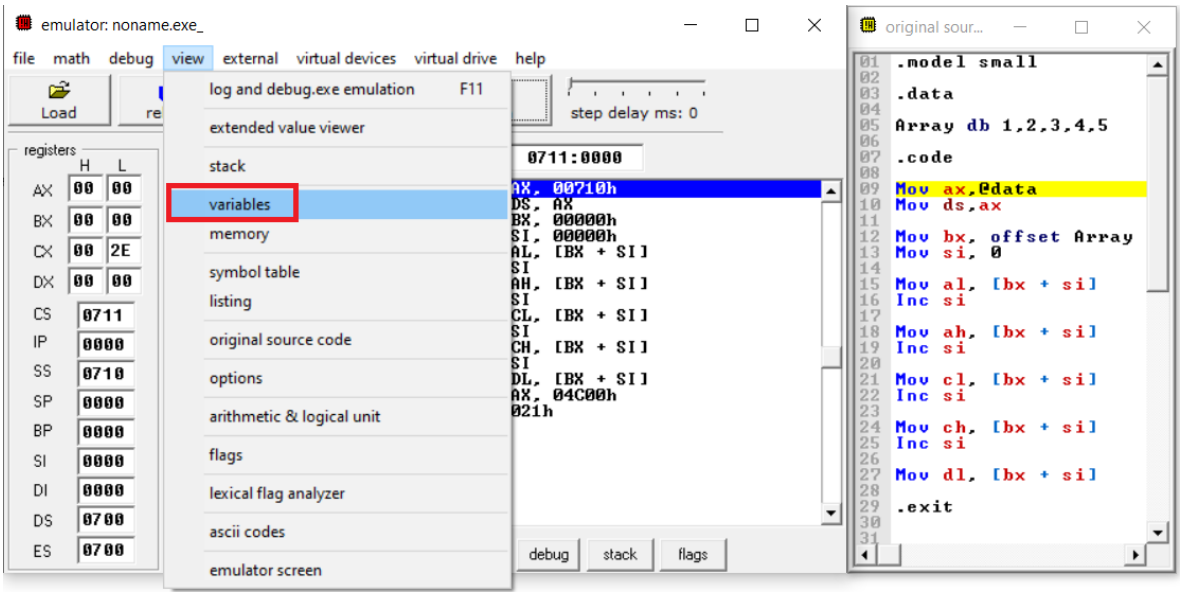
Step-3:
Click on the "empty workspace" and press "OK".



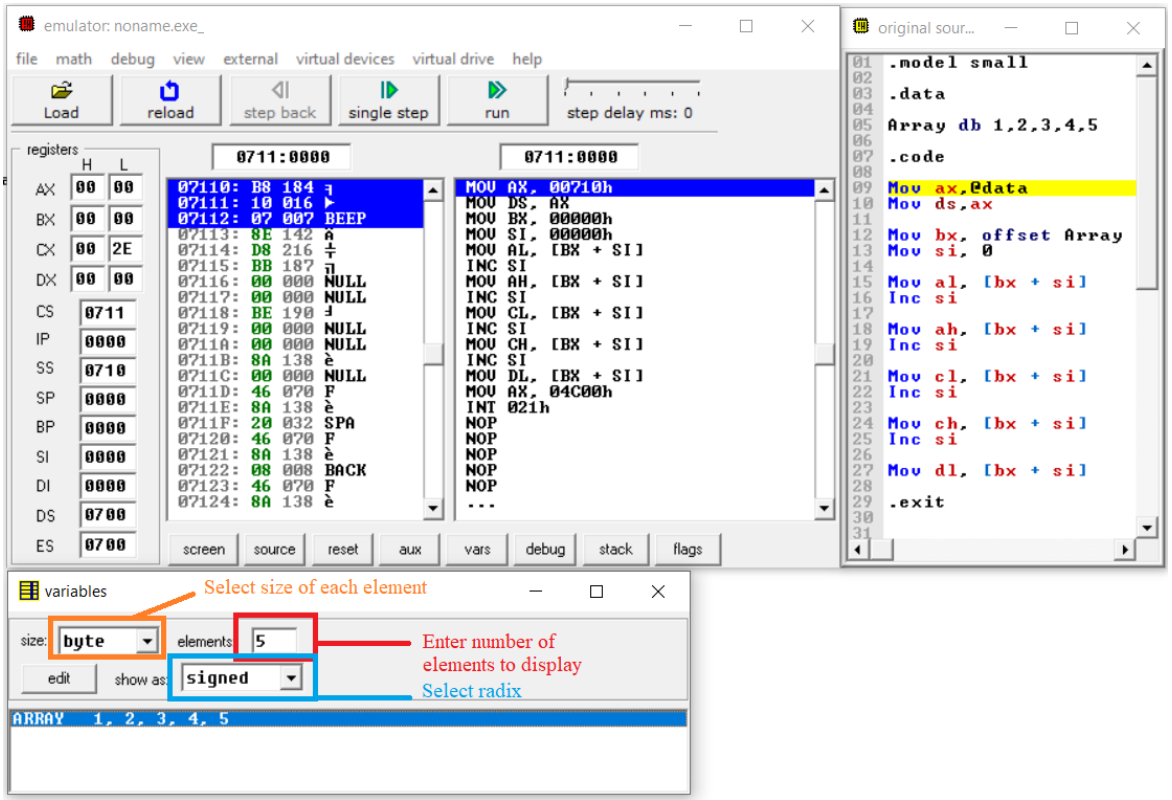
Step-4:
Type the code given in program 1 above and click on "emulate".



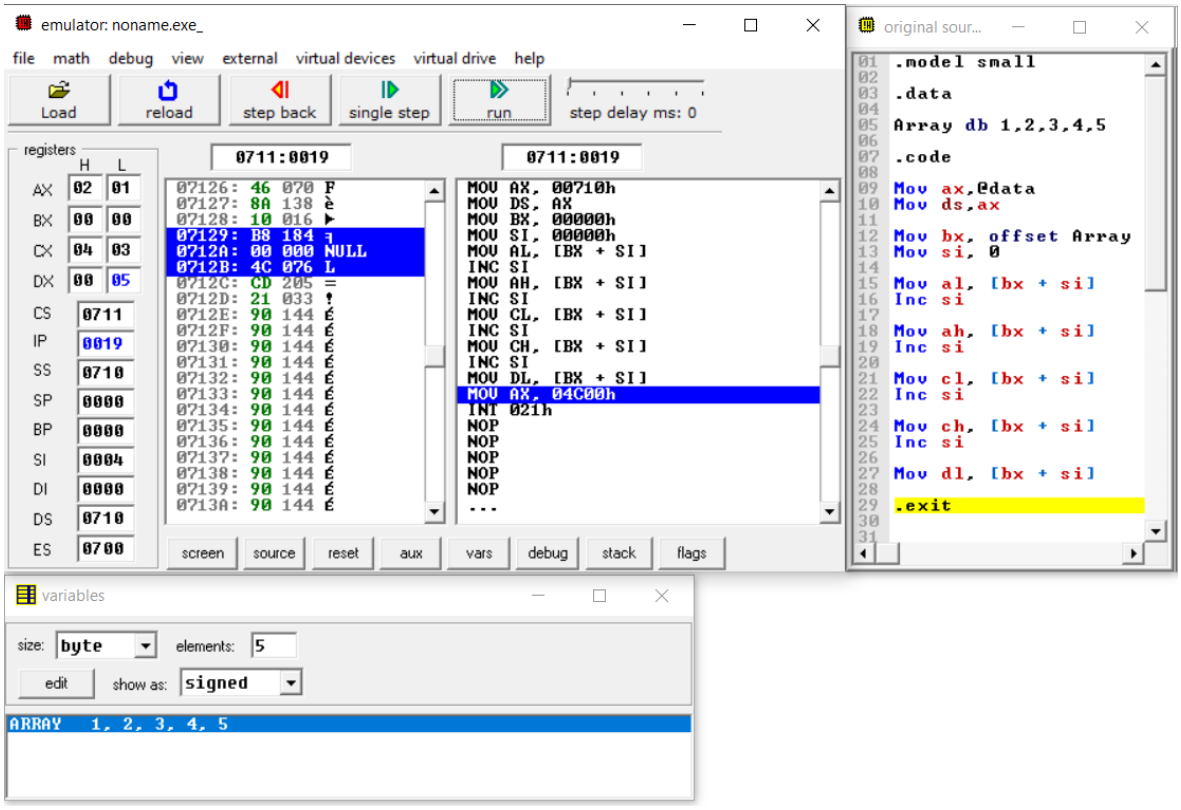
Step-5:
Click on "variables" from the view menu.



Step-6:
Set size, number of elements, and radix in the variable window.

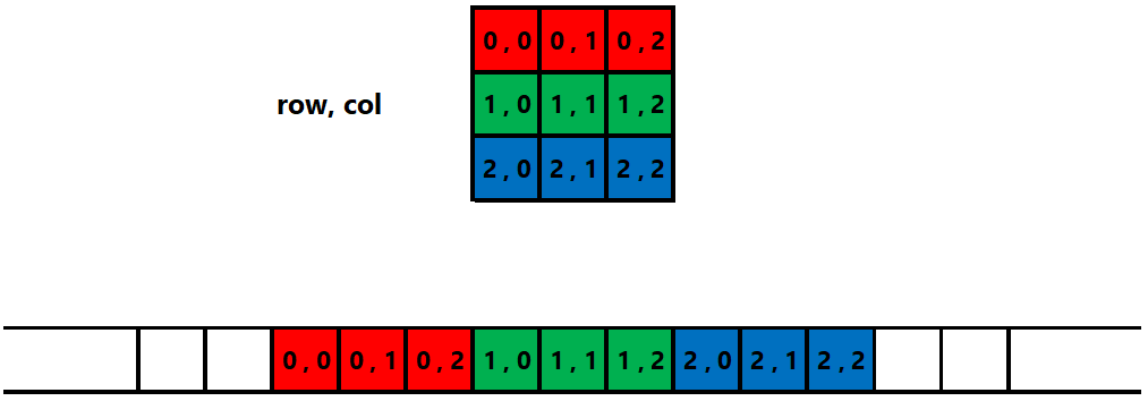


Step-7:
Keep clicking on "Single step" to execute program instructions one by one and observe the register values side by side.
Stop clicking "Single step" when the ".exit" is highlighted to observe the register values.



Storage of 2D arrays in memory

Note that a 2D matrix is stored in a memory as a linear array as shown in the image below. You can access this linear array using base plus index addressing mode. Base register (BX) will hold the base address of the array, while the index address will be incremented as you access the linear array.



Linear address calculations

linear_address = base_address + (row_index * num_columns + column_index) * element_size

For (1,2), the linear address will be calculated by the following equation. Please note that this address will be the offset part of logical address whereas the segment part will be from DS register. If

row_index = 1
num_columns = 3
column_index = 2
element_size = 1

then,

Linear_address = base_address + (1 * 3 + 2) * 1

Example#1

Program to declare and initialize 3x3 array in assembly language.

```
.model small

.data

array2d db 1,2,3
        db 3,4,5
        db 6,7,8

.code
```

Example#2

Program to read index (2,1) of a 2D array.

```
.model small

.data

array2d db 1,2,3
        db 3,4,5
        db 6,7,8

.code
mov ax,@data
mov ds,ax


mov bx,offset array2d
 mov al,3                ; number of cols
mov dl,2                 ;mov row index to dl (i,j) = (2,1)
mov dh,1                 ;mov column index to dh (i,j) = (2,1)
mul dl                   ;multiple row index with number of col
add al,dh               ; add col index to the product calculated in previous instruction
mov si,ax                ; moving linear address to si
mov al,[bx+si]           ; reading desired element that is 7 at index (2,1)

.exit
```

Example#3

Program to read index (2,1) of a 2D array.

```
.model small

.data
array2d db 1,2,3,4,5,6,7,8,9


.code
mov ax,@data
mov ds,ax

mov bx,offset array2d
mov al,3                ; number of cols
mov dl,2                ;mov row index to dl (i,j) = (2,1)
mov dh,1                ;mov column index to dh (i,j) = (2,1)
mul dl                  ;multiple row index with number of col
add al,dh               ; add col index to the product calculated in previous
instruction
mov si,ax               ; moving linear address to si
mov al,[bx+si]          ; reading desired element that is 7 at index (2,1)
.exit
```

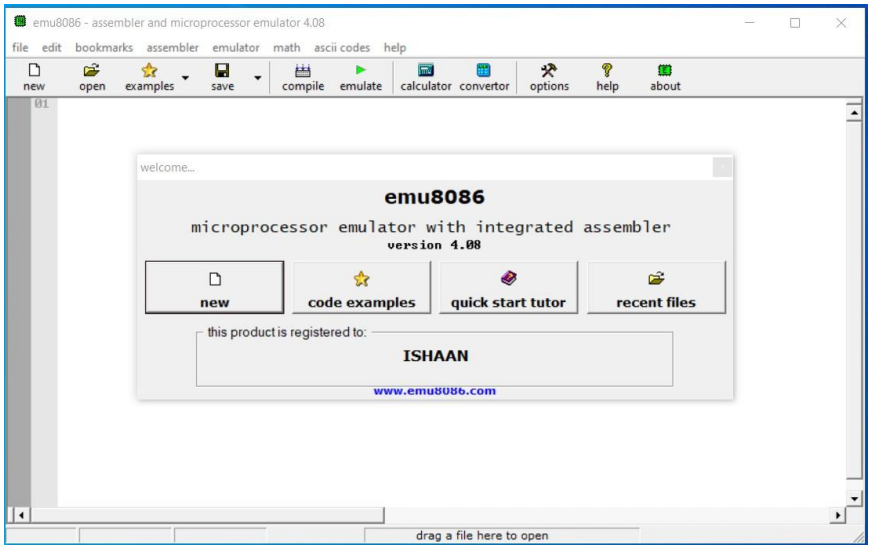
Emu8086 Tutorial Step by Step

Step-1

Double click on the icon on the desktop 

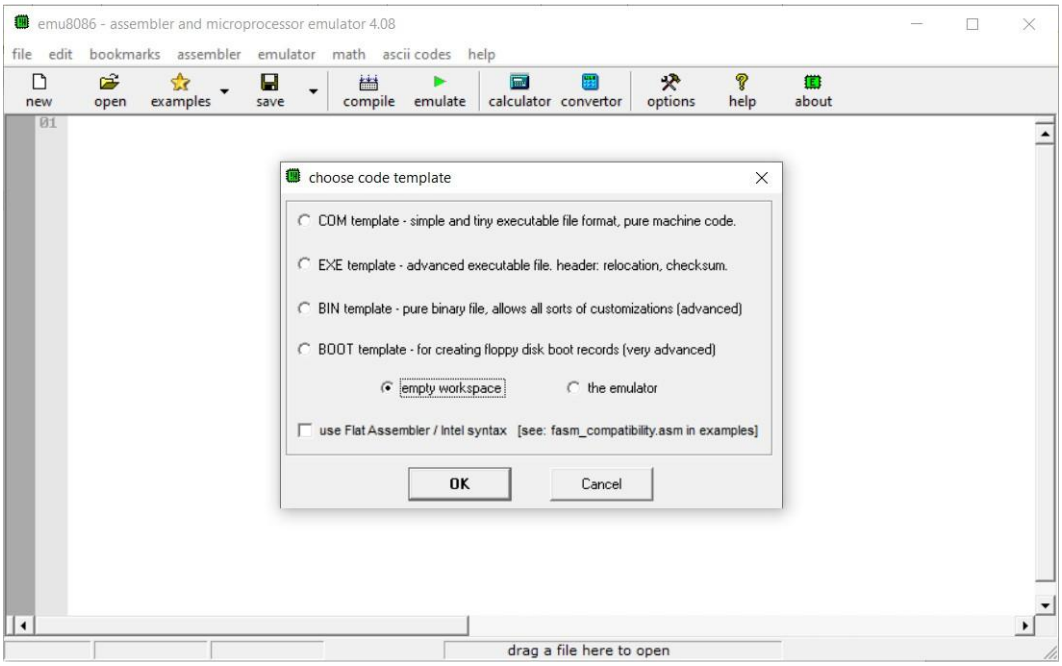
Step-2

The following window will appear. Click on “new”.



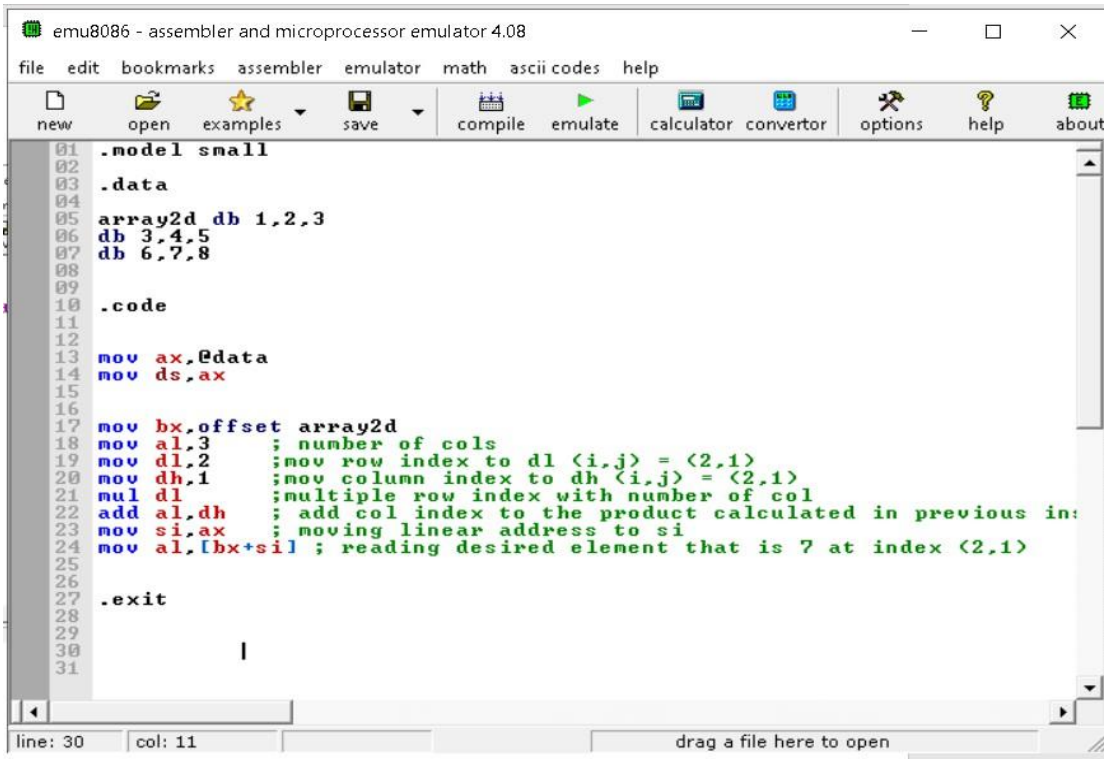
Step-3:

Click on the empty workspace and press “OK”.



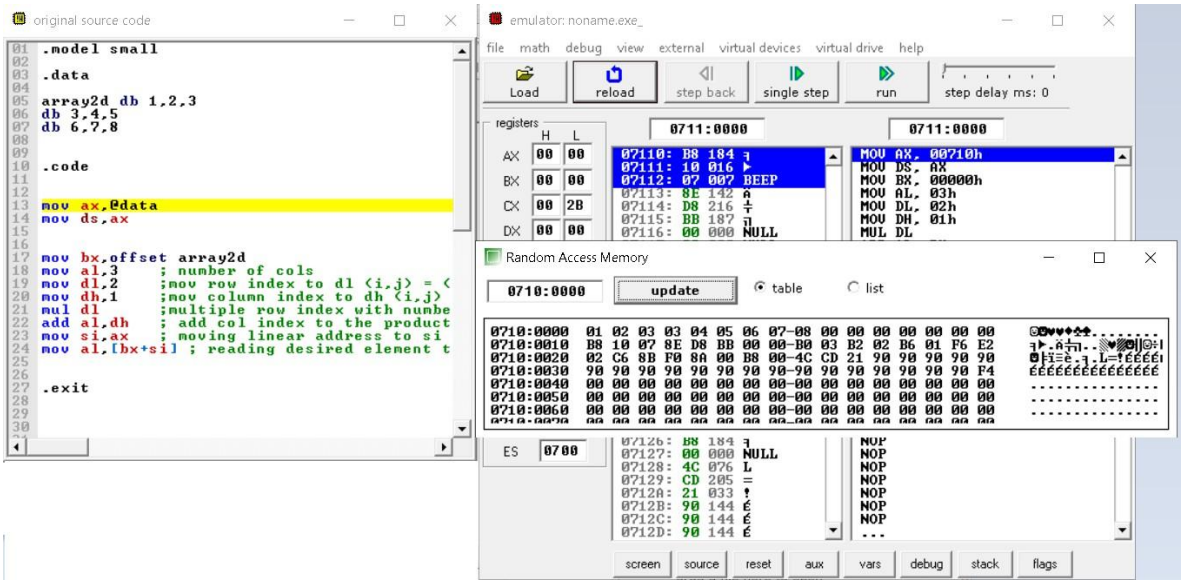
Step-4:

Type the code given in example#2 and click on “emulate”.



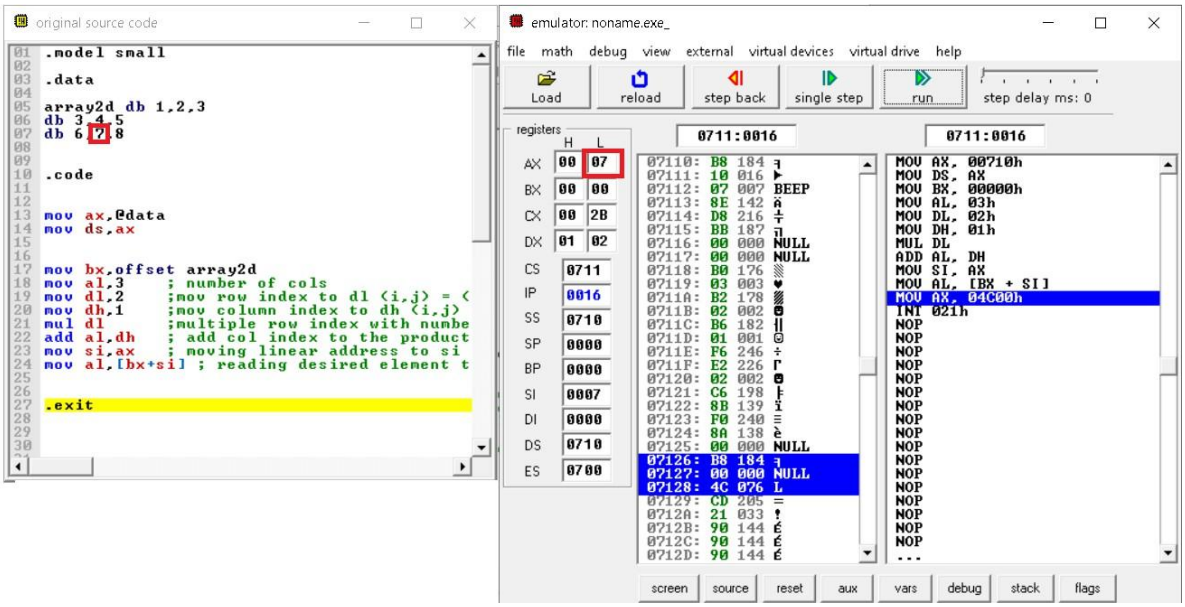
Step-5:

Click on the view → memory button to view 2D array stored in memory.



Step-6:

Keep clicking on “Single step” and observe the working of the program.



Lab-5 Exercise

Task-1

Write a program that declares and initializes an array with 10 elements, then uses a loop to find the sum of those elements and stores the result in a variable named "SUM".

Task-2

Write a program that declares and initializes two word-type arrays: A and B, each of which has 20 elements. The program then adds the corresponding elements of these two arrays and stores the result in the third array: C.

Task-3

Write a program that declares and initializes a 3x4-sized matrix M1, takes its transpose, and stores the transpose in another matrix M2.

Task-4

Write a program that initializes two 3x4-size matrices and stores their sum in another matrix.

Task-5

Write a program that initializes two 3x4 and 4x3-size matrices, named A and B, and stores the result after multiplication of A and B into a third matrix, C.

Lab 6: Program Flow Control

Learning Outcomes

- Students will be able to transfer control to any label or address unconditionally.
- Students will be able to transfer control to any label or address based on some conditions.
- Students will be able to implement customized loops instead of using built-in loops.
- Students will be able to use opposite conditional jumps to reduce the number of jump instructions inside loop.
- Students will be able to make a long conditional jump using short conditional jump instructions.

Control in a program can be unconditionally or conditionally transferred to any location based on status flags. There are instructions for transferring control.

Unconditional Jump

The basic syntax of JMP instruction:

JMP label

The "JMP" instruction transfers control to the address or label followed by it. This instruction can move control within or outside of the current code segment. In the following table, example 1 shows the jump inside the current code segment, called a "near jump," and example 2 shows the jump outside the current code segment, called a "long jump." For a short jump, the address that comes after the "JMP" instruction will be an offset. For a long jump, the address will be a logical address made up of a segment address and an offset.

Example-1: Transferring control to label within same code segment.	**Example-2:** Transferring control to address outside the current code segment.

.model small
.data
 	
.code
 	
Mov ax,@data
Mov ds,ax
 	
Jmp Label1
 	
Mov AX, BX
Mov CX, DX
 	
Lable1:
Mov AX,1
Mov BX,2
 	
.exit
	.model small
	.data
	
	.code
	
	Mov ax,@data
	Mov ds,ax
	
	Jmp 0x07C0:0x0000
	
	Mov AX, BX
	Mov CX, DX
	
	.exit

Conditional Jumps

Unlike JMP instruction that does an unconditional jump, there are instructions that do a conditional jump (jump only when some conditions are in act). These instructions are divided in three groups, first group just test single flag, second compares numbers as signed, and third compares numbers as unsigned. All these three groups use status flags to jump.

We know that the status flags are modified because of ALU instructions. Therefore, to compare two numbers, we usually perform a subtraction operation on these two numbers. We are not concerned with the result of the subtract operation but with the status flags that are updated. Therefore, instead of the "SUB" instruction, the "CMP" instruction is used, which discards the result and keeps the status of flags.

The basic syntax of CMP instruction:

CMP Destination Operand, Source Operand

Jump instructions that test single flag

The following table shows the conditional jump instructions that jump based on the value of a single flag. In the instruction column, some rows contain more than one instruction that does the same thing. They are even assembled into the same machine code. **JE** will be assembled as **JZ, JC will be** assembled as **JB,** and so on. These names are used to make programs easier to understand, code, and remember.

Instruction	Description	Condition	Opposite Instruction
JZ JE	Jump if Zero Jump if Equal	ZF = 1	JNZ JNE
JC JB JNAE	Jump if Carry Jump if Below Jump if Not Above Equal	CF = 1	JNC JNB JAE
JS	Jump if Sign.	SF = 1	JNS
JO	Jump if Overflow.	OF = 1	JNO
JPE/ JP	Jump if Parity Even.	PF = 1	JPO
JNZ JNE	Jump if Not Zero Jump if Not Equal	ZF = 0	JZ JE
JNC JNB JAE	Jump if Not Carry Jump if Not Below Jump if Above Equal	CF = 0	JC JB JNAE
JNS	Jump if Not Sign.	SF = 0	JS
JNO	Jump if Not Overflow.	OF = 0	JO
JPO JNP	Jump if Parity Odd Jump if No Parity	PF = 0	JPE JP

Jump instructions for signed numbers.

To compare two signed numbers, the instructions shown in the following table are used right after the 'CMP' instruction.

Instruction	Description	Condition	Opposite Instruction
JE JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE JNZ
JNE JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE JZ
JG JNLE	Jump if Greater (>). Jump if Not Less or Equal (not <=).	ZF = 0 and SF = OF	JNG JLE
JL JNGE	Jump if Less (<). Jump if Not Greater or Equal (not >=).	SF <> OF	JNL JGE
JGE JNL	Jump if Greater or Equal (>=). Jump if Not Less (not <).	SF = OF	JNGE JL
JLE JNG	Jump if Less or Equal (<=). Jump if Not Greater (not >).	ZF = 1 or SF <> OF	JNLE JG

Jump instructions for unsigned numbers

To compare two unsigned numbers, the instructions shown in the following table are used right after the 'CMP' instruction.

Instruction	Description	Condition	Opposite Instruction
JE JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE JNZ
JNE JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE JZ
JA JNBE	Jump if Above (>). Jump if Not Below or Equal (not <=).	CF = 0 and ZF = 0	JNA JBE
JB JNAE JC	Jump if Below (<). Jump if Not Above or Equal (not >=). Jump if Carry.	CF = 1	JNB JAE JNC
JAE JNB JNC	Jump if Above or Equal (>=). Jump if Not Below (not <). Jump if Not Carry.	CF = 0	JNAE JB
JBE JNA	Jump if Below or Equal (<=). Jump if Not Above (not >).	CF = 1 or ZF = 1	JNBE JA

Limitation of Conditional Jump instructions

- All conditional jumps have one big limitation, unlike **JMP** instructions, they are short (one-byte jumps having a range from -128 to 127 bytes). However, we can easily avoid this limitation using a cute trick:
  - Get conditional jump instruction from the tables above and make it jump to ***label_X*.**
  - Under that **label_X**, Use **JMP** instructions to jump to the desired location.
- Emu8086 uses this trick implicitly for conditional jumps.

Program 1:
To swap values of AX and BX registers if AX < BX.

1a: Program with straight conditional jumps.	**1b:** Program with opposite conditional jumps to reduce jumps.

.model small
 	
.data
 	
.code
 	
Mov ax,@data
Mov ds,ax
 	
Mov ax,5
Mov bx,10
 	
Cmp ax,bx
JS swap
Jmp exit_cmp
 	
Swap:
XCHG ax,bx
 	
exit_cmp:
 	
.exit
	.model small
	
	.data
	
	.code
	
	Mov ax,@data
	Mov ds,ax
	
	Mov ax,5
	Mov bx,10
	
	Cmp ax,bx
	JNS exit_cmp
	
	Swap:
	XCHG ax,bx
	
	exit_cmp:
	
	.exit

Program 2:
To iterate a loop while AX < BX using conditional jump for unsigned numbers.

1a: Program with straight conditional jumps.	**1b:** Program with opposite conditional jumps to reduce jumps.

.model small
 	
.data
 	
.code
 	
Mov ax,@data
Mov ds,ax
 	
Mov ax,5
Mov bx,10
 	
compare:
cmp ax,bx
JB iterate
jmp exit_loop
 	
iterate:
inc ax
jmp compare
 	
exit_loop:
 	
.exit
	.model small
	
	.data
	
	.code
	
	Mov ax,@data
	Mov ds,ax
	
	Mov ax,5
	Mov bx,10
	
	compare:
	cmp ax,bx
	JAE exit_loop
	
	iterate:
	inc ax
	jmp compare
	
	exit_loop:
	
	.exit

Program 3:
To iterate a loop while AX < BX using conditional jump for signed numbers.

1a: Program with straight conditional jumps.	**1b:** Program with opposite conditional jumps to reduce jumps.

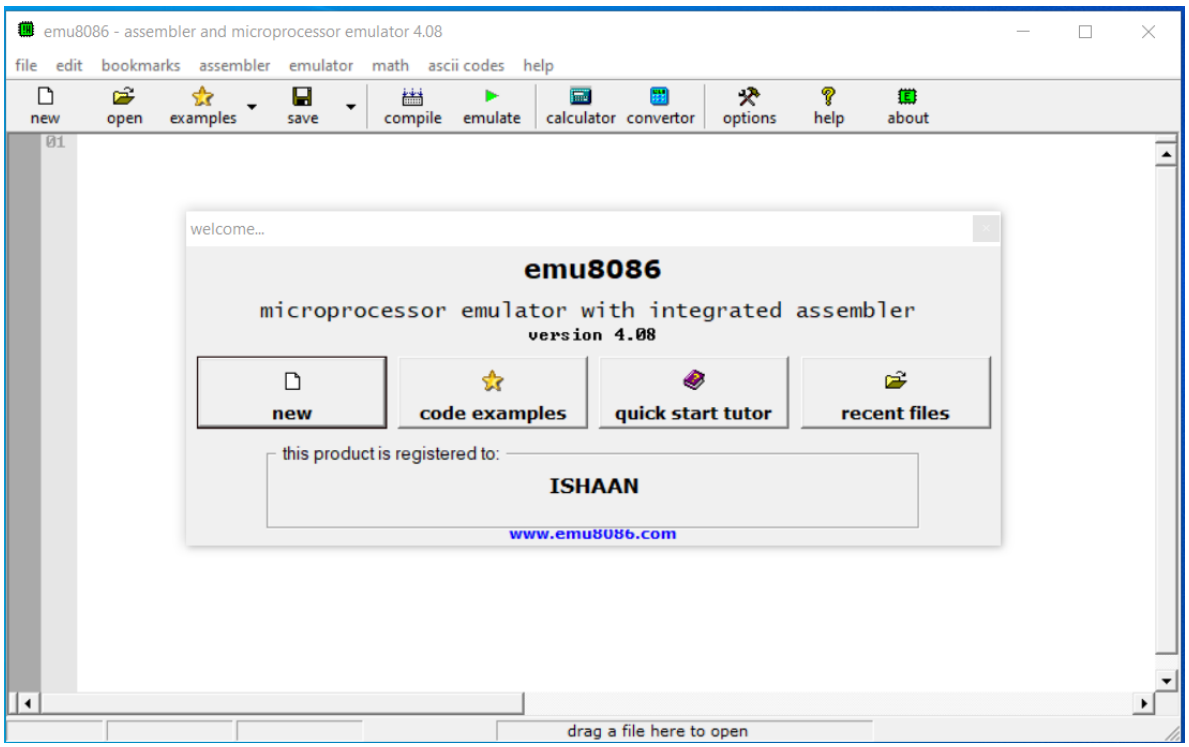
.model small
 	
.data
 	
.code
 	
Mov ax,@data
Mov ds,ax
 	
Mov ax, -5
Mov bx, 0
 	
compare:
cmp ax, bx
JL iterate
jmp exit_loop
 	
iterate:
inc ax
jmp compare
 	
exit_loop:
 	
.exit
	.model small
	
	.data
	
	.code
	
	Mov ax,@data
	Mov ds,ax
	
	Mov ax, -5
	Mov bx, 0
	
	compare:
	cmp ax, bx
	JGE exit_loop
	
	iterate:
	inc ax
	jmp compare
	
	exit_loop:
	
	.exit

Emu8086 Tutorial Step by Step

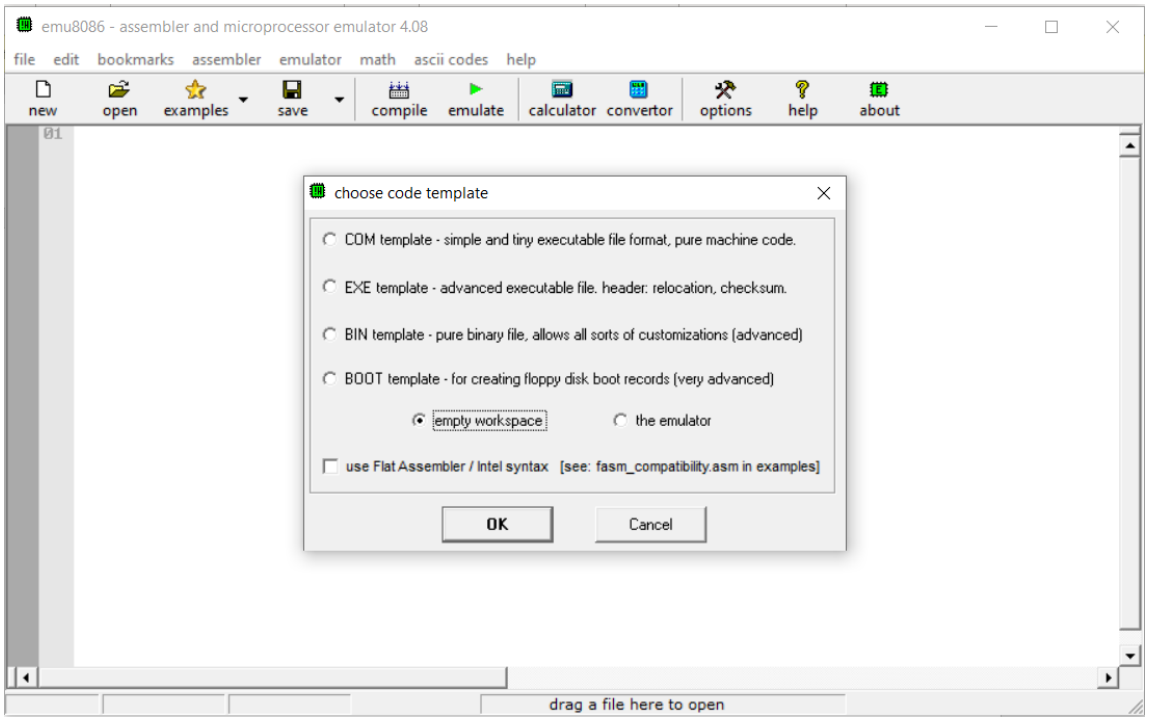
Step-1:

Double click on the icon on the desktop 

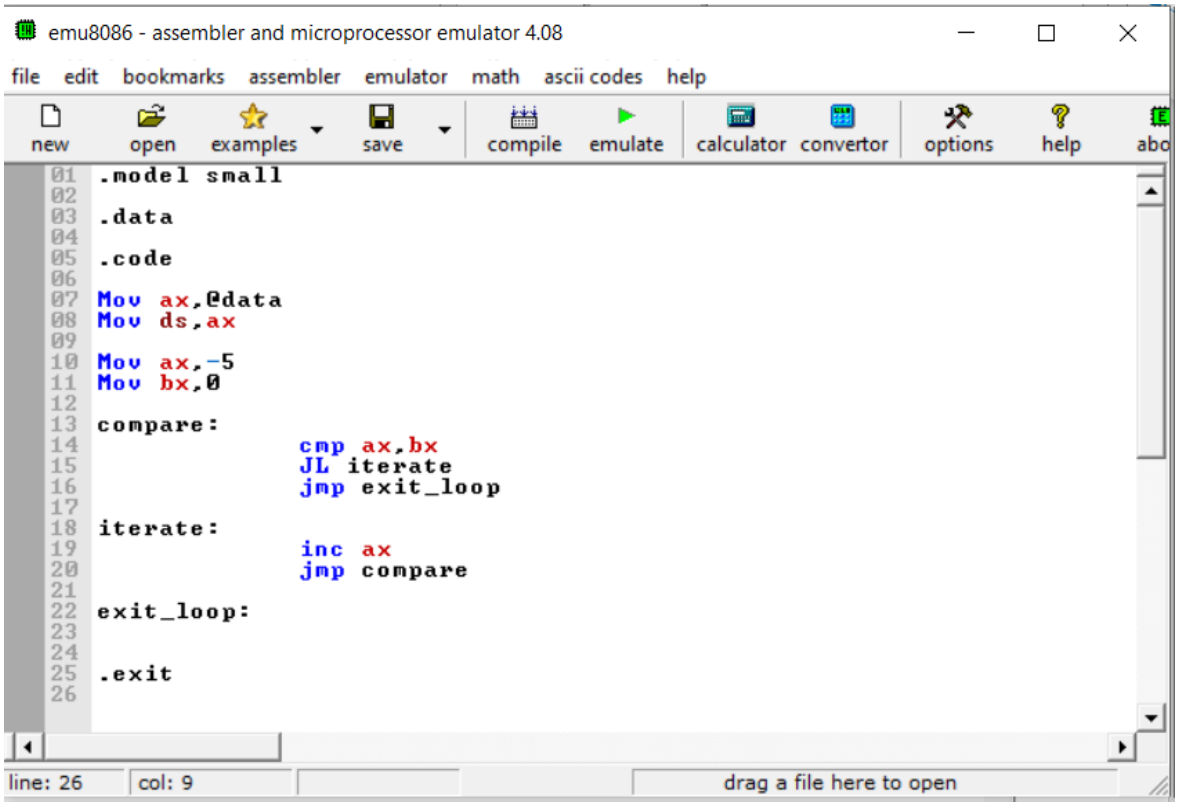
Step-2:
The following window will appear. Click on “new”.



Step-3:
Click on the “empty workspace” and press “OK”.

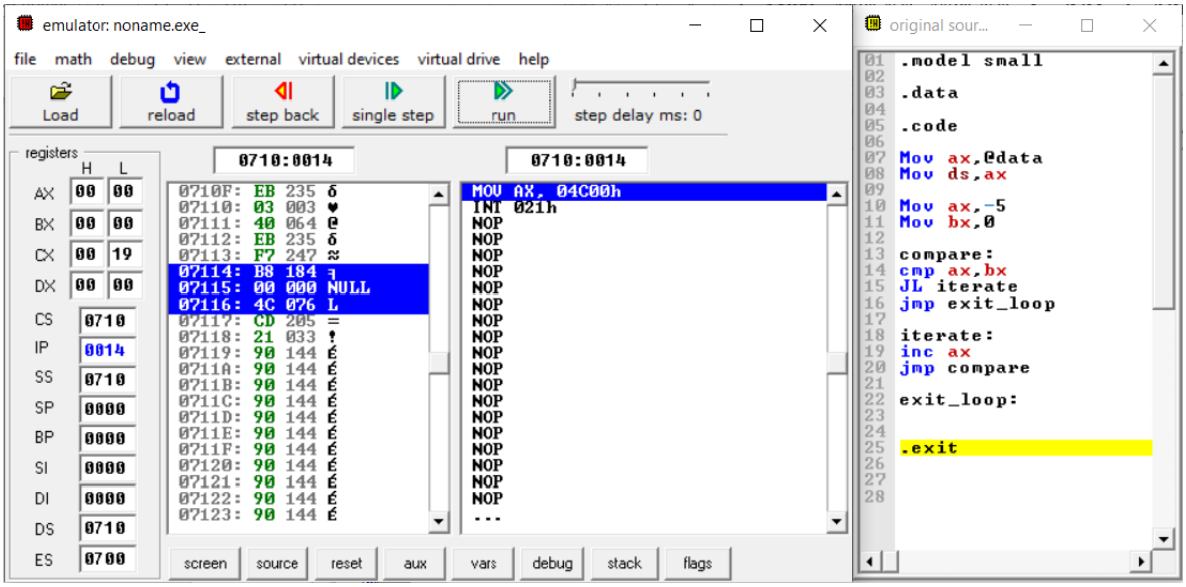


Step-4:
Type the code given in program-3a above and click on “emulate”.



Step-5:

Keep clicking on "Single Step" to execute program instructions one by one and observe the register values side by side and the number of times the loop is iterated. Stop clicking "Single Step" when the ". exit" is highlighted to observe the final value of the AX register.



Lab-6 Exercise

Task-1

Write a program that declares and initializes an array of 20 elements and then calculates the number of occurrences of a specific number in the array.

Task-2

Write a program that declares and initializes a word-type array of 20 elements and sorts it using any sorting algorithm of your choice.

Lab 7: Stack & Procedures

Learning Outcomes

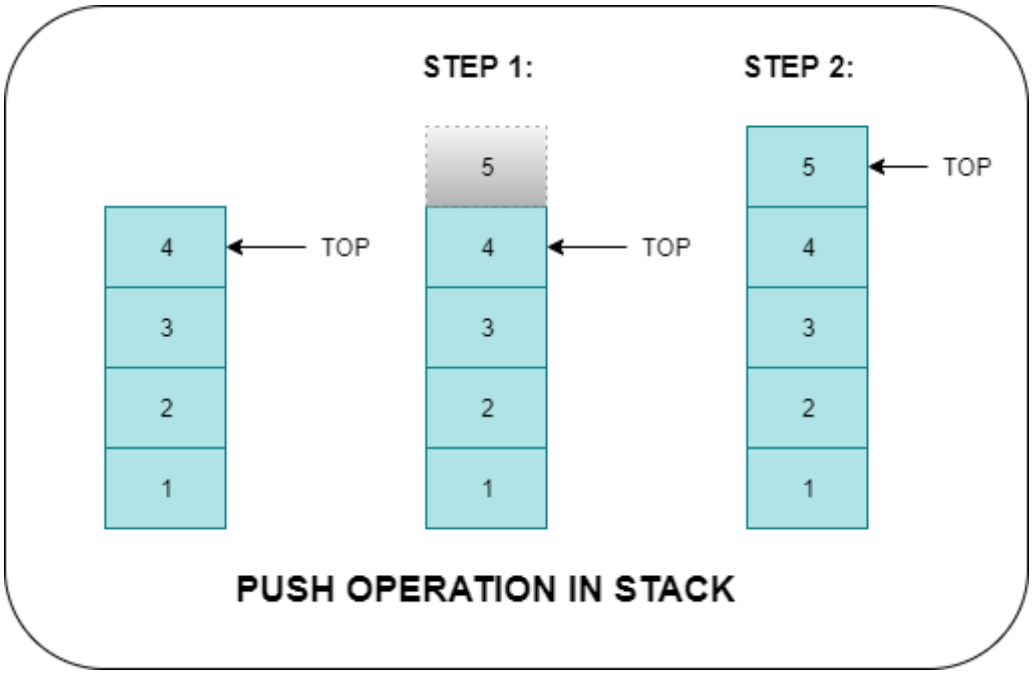
- Students will be able to use real-time Stack using push and pop instructions.
- Students will be able to access Stack directly without using push and pop instructions.
- Students will be able to use Stack for various applications.
- Students will be able to define procedures, pass parameters to them, and return values from them.

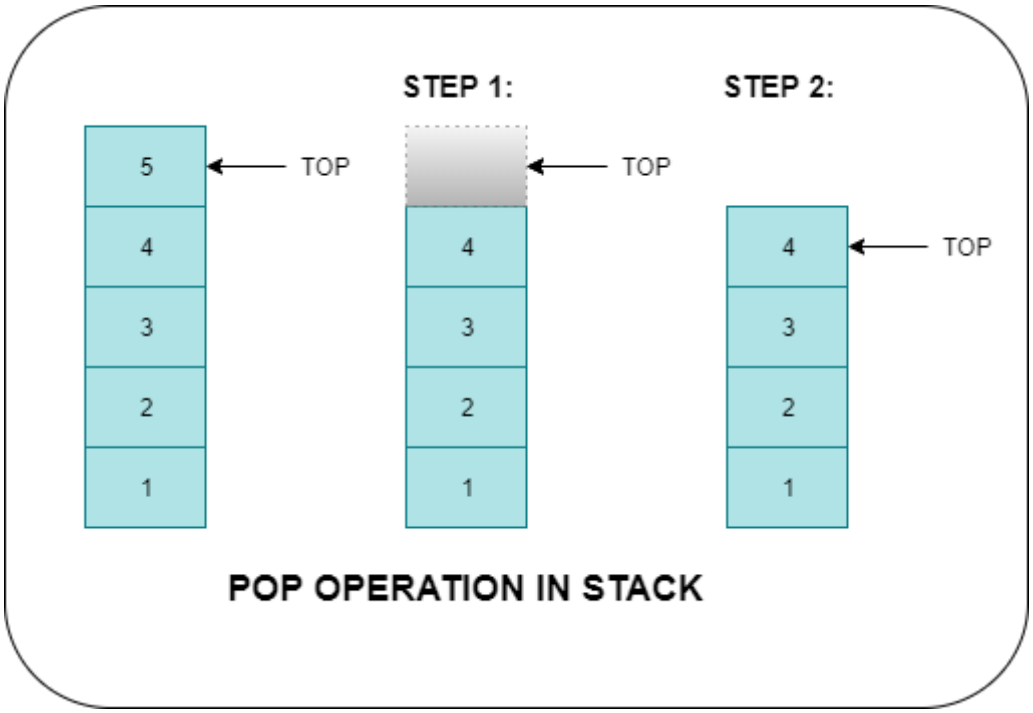
The Stack

A stack is a Last-In-First-Out (LIFO) list that is attached to a program when it is loaded into memory for execution. This stack is managed by the hardware. The Stack Segment (SS) register points to the base of the stack, and the Stack Pointer (SP) register points to the top of the stack (TOS).

In 8086, a 16-bit register, or variable is pushed on the stack using PUSH instructions. The POP instruction, on the other hand, removes the values pointed to by SP in the same register.

If we push 1, 2, 3, 4, 5 one by one into the stack, the first value that we will get on pop will be 5, then 4, 3, 2, and then 1 as shown in the following figure.





The instructions that push the operand to stack and pop it from stack are shown in the table below.

Instruction	Operation	Description
PUSH source	▪ SP ← SP – 2 ▪ SS:[SP] ← source	Source: ▪ All general-purpose registers, ▪ All segment registers. ▪ Variables ▪ Immediate Value ▪ IP register cannot be pushed
POP destination	▪ Destination ← SS:[SP] ▪ SP ← SP + 2	Destination: ▪ All general-purpose registers, ▪ All segment registers except CS & IP ▪ Word-type Variables

The following instructions push and pop flag registers into the stack.

Instruction	Operation	Description
PUSHF	▪ SP ← SP – 2 ▪ SS:[SP] ← Flag register	▪ Stores flag registers on top of the stack
POPF	▪ Flag register ← SS:[SP] ▪ SP ← SP + 2	▪ Mov value at top of the stack to flag register

Applications of Stack

Reusing registers simultaneously

There are a few registers in a processor to operate. Therefore, to use a register for another purpose, the contents of the register are temporarily pushed to the stack and popped later.

- Store the original value of the register in a stack (using PUSH).
- Use the register for any purpose.
- Restore the original value of the register from the stack (using POP).

Storing Return Address
When a function is called, the return address is pushed to the stack.

Declaring local variables
Global variables are declared and defined in data segments. However, local variables are stored on stack.

Accessing stack without using pop instructions

The Stack Segment can be accessed directly without using the POP instruction. We know that the stack segment is accessed when we use the BP register in register indirect addressing, i.e., [BP]. The contents of the SP register are moved to the BP register to gain access to the top of the stack. And then, using register indirect mode, the value at the top of the stack can be read or written.

Example:
Accessing top of the stack

```
Mov BP, SP
MOV AX, [BP]
```

Example:
Accessing the second value from the top of the stack

```
Mov BP, SP
MOV AX, [BP+2]
```

Example:
Accessing the third value from the top of the stack

```
Mov BP, SP
MOV AX, [BP+4]
```

Procedures

A procedure is a part of code that can be called from your program to perform some specific tasks. Procedures make programs easier to understand. Generally, the procedure returns to the same point from where it was called.

The syntax for a procedure declaration is:

```
name PROC

    ; here goes the code
    ; of the procedure

RET
name ENDP
```

name- is the procedure name. The same name should appear at the top and bottom. This is used to ensure that procedures are properly closed.

RET- is required at the end of the procedure to return the program control back to where it came from.

The following table shows the instructions that are used to call a procedure and return control back.

Instruction	Operation	Description
Call label/ offset (2 bytes)	Push IP IP ← Label	Calling procedure that is defined in the current code segment
Call label/ segment: offset (4 bytes)	Push CS Push IP IP ← offset CS← segment	Calling procedure that is defined outside the current code segment
Ret	Pop IP	Returns control from procedure defined in the current code segment.
Retf	POP IP POP CS	Returns control from procedure defined outside the current code segment.

So far, we have not defined any procedure in a code segment. However, there should be a procedure inside the code segment, just as there is a main function in C/C++. From now on, we will write all the code in procedures. The main function is called by an operating system when we run a program.

Program#1:
Program that defines the main function in a program.

```
.model small

.stack

.data

.code
Main proc

Mov ax,@data
Mov ds,ax

.exit
Main endp
```

Parameter and Return value to/ from Procedures.

Procedures can be given data to processes, called parameters. These parameters can be passed via registers or through the stack. However, the procedures return values through the AL or AX register.

Program#2:
Program that defines the procedure "addition" to add two numbers, passed through registers, and return their sum.

```
.model small

.stack

.data

.code
Main proc                ;Defining main procedure
Mov ax,@data
Mov ds,ax

Call addition            ;Calling procedure
.exit
Main endp                ;main procedure ends here

Addition proc            ;defining procedure
Add ax,bx                ;adding two registers

Ret                      ;return control back to calling procedure
Addition endp
```

Program#3:
Program that defines the procedure "addition" to add two numbers, passed through stack, and return their sum.

```
.model small

.stack

.data

.code
Main proc

Mov ax,@data
Mov ds,ax

Mov ax,5
Mov bx,10
Push ax                    ;pushing value of ax, which is 5 to stack
Push bx                    ;pushing value of bx, which is 10 to stack

Call addition              ; calling procedure

Pop bx                     ; removing number 10 from stack
Pop bx                     ; removing number 5 from stack

.exit
Main endp

Addition proc              ;defining procedure

Push bp                    ;storing value of BP on stack so that we can restore it
later

Mov bp,sp                  ;to access stack without pop instruction—moving TOS
to bp
Mov ax,0                   ;no need to push AX as it is safe to use AX register

Add ax,[bp+4]              ; [bp+4] contains value 10
Add ax,[bp+6]              ; [bp+6] contains value 5

Pop bp                     ; restoring value of BP from stack

Ret                        ; transferring control back to the calling procedure
Addition endp
```

Program#4:
Program that defines the procedure "addition" to sum up an array of 5 elements and return its sum. **Note:** Array is always passed by reference.

```
.model small
.data
array db 1,2,3,4,5
.code
Main proc
Mov ax,@data
Mov ds,ax

Mov bx,offset array ;base address of array
mov ax,5            ;size of array

Push bx
Push ax

Call addition       ;calling procedure

pop bx              ;removing parameters from stack
pop bx              ;removing parameters from stack

.exit
Main endp

Addition proc  ;Defining procedure

push bp     ;saving the value of BP on stack before using it
push cx     ;saving the value of CX on stack before using it
push si     ;saving the value of SI on stack before using it

mov bp,sp       ;moving top of the stack to bp
mov cx,[bp+8]   ;reading size of array
mov ax,[bp+10]  ;reading base address of array
mov bp,ax       ;moving base address to BP
mov ax,0        ; it is safe to use ax without pushing it
mov si,0        ;assigning index register with 0

l1:
Add al,ds:[bp+si]  ;accessing value 10
inc si          ;incrementing index by 1 as the array is of a byte type
loop l1
pop si          ;Restoring original value of SI from stack
pop cx          ;Restoring original value of CX from stack
pop bp          ;Restoring original value of BP from stack

ret             ;returning control back to calling procedure
Addition endp
```

Program#5:
Program that creates local variables.

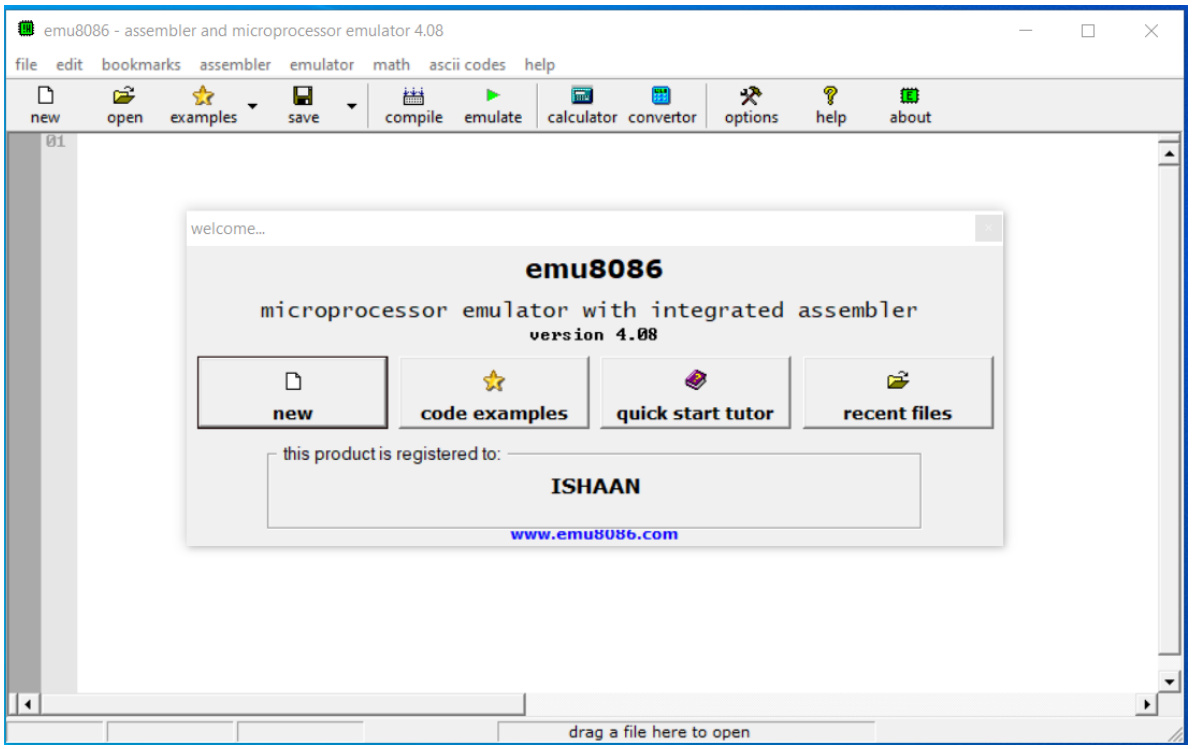
```
.model small

.data

.code

Main proc

        push    bp                      ; storing value of bp on stack
        mov     bp,sp
        sub     sp,4                    ;creating space on stack for two variables

        mov     word ptr [bp-2],5       ; assigning value to local variable
        mov     word ptr [bp-2],10      ; assigning value to local variable

       ; now you may PUSH and POP equal number of times here


        mov     sp,bp                   ;destroying local variables before returning
        pop     bp                      ; restoring value of bp from stack
        ret

.exit

Main endp
```

Emu8086 Tutorial Step by Step

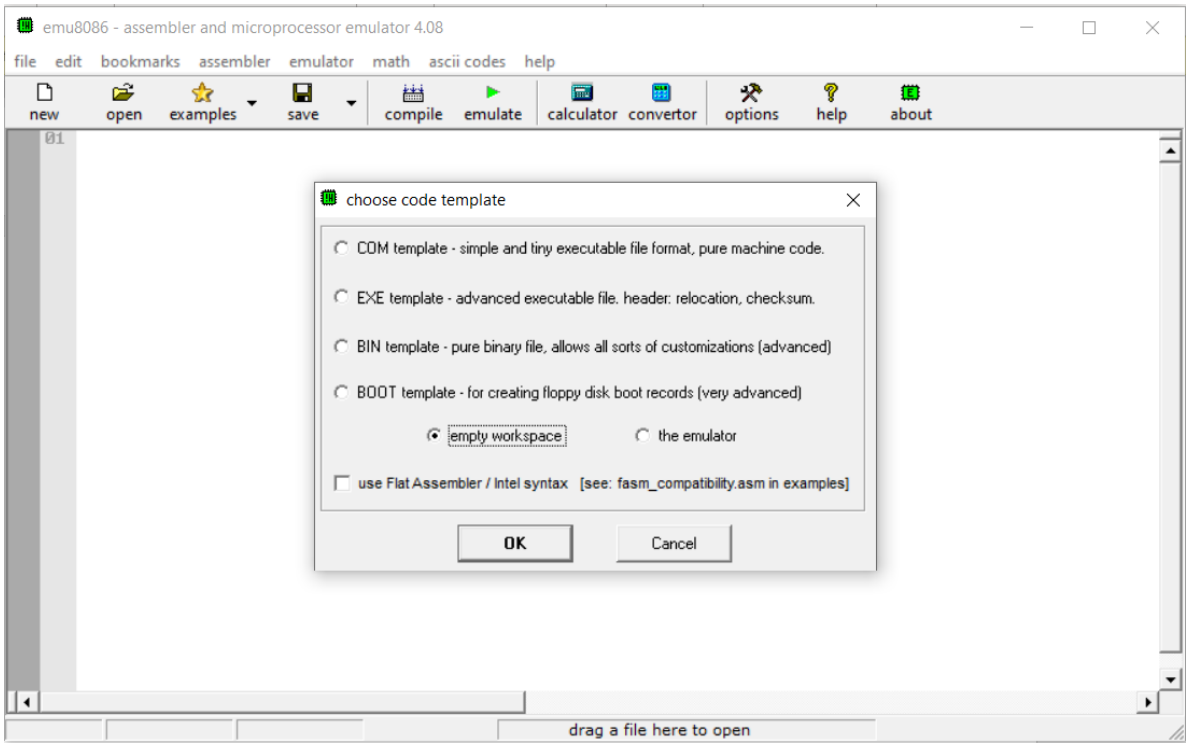
Step-1:

Double click on the icon on the desktop 

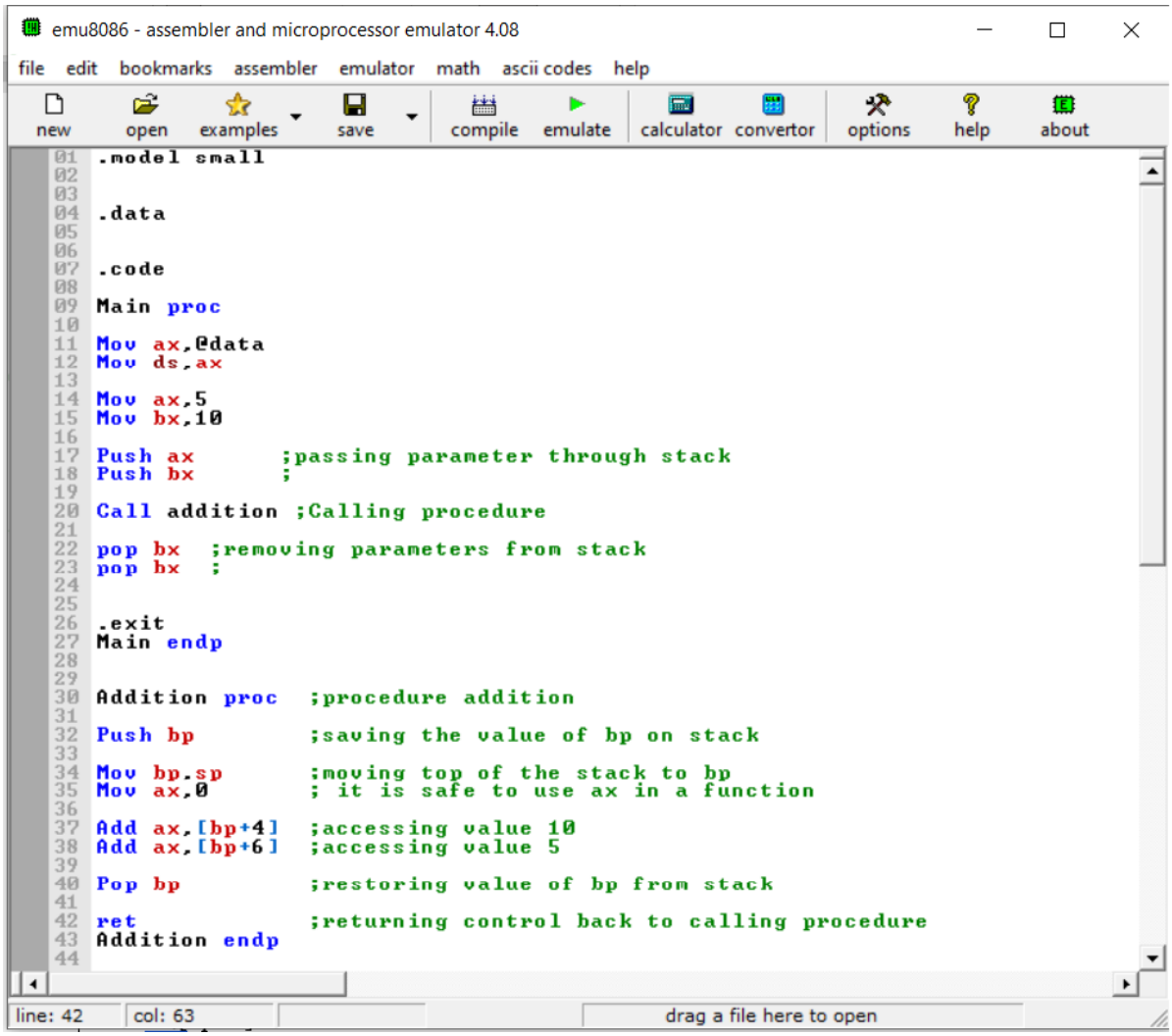
Step-2:
The following window will appear. Click on “new”.



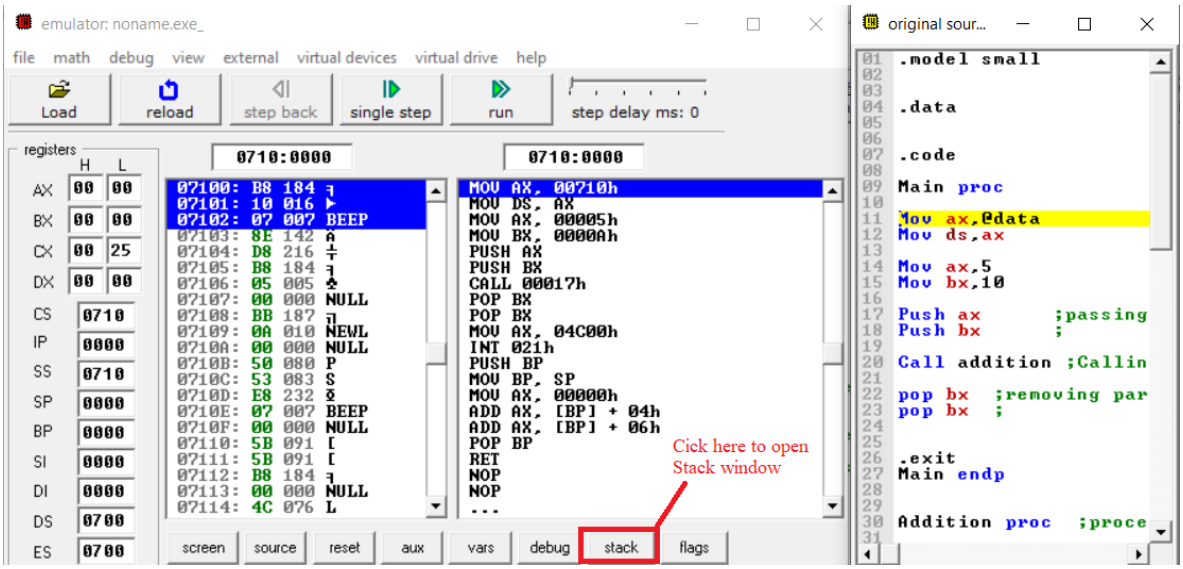
Step-3:
Click on the “empty workspace” and press “OK”.



Step-4:
Type the code given in program#3 above and click on “emulate”.

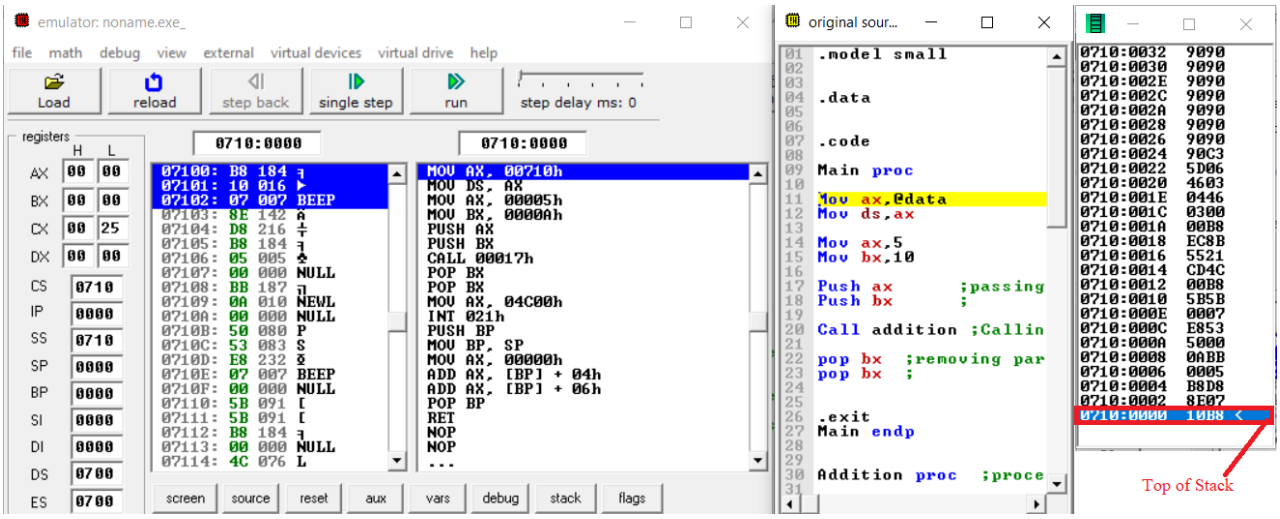


Step-5:
Click on the Stack to view stack window.



Step-6:

Keep clicking on "Single Step" to execute program instructions one by one and observe the top of the Stack side by side.



Observations

Which address is pushed to stack when a procedure is called?
At what address is control transferred when a procedure returns?
How have [BP+4] and [BP+6] been calculated for the desired locations of parameters?

Lab-7 Exercise

Task-1

Write a program that defines the procedure "SUM." The procedure receives three arrays: A, B, and C, and the sizes of the arrays through a parameter. The procedure adds corresponding elements from arrays A and B and stores the sum in array C.

Task-2

Write a program that declares a byte array and stores an English word into it. The program then checks if the array contains a palindrome. It stores 1 in the DL register in the case of palindromes and 0 otherwise.

Lab 8: Video Memory and String Instructions

Learning Outcomes

- Students will be able to write characters directly to video memory.
- Students will be able to translate screen coordinates into linear memory addresses.
- Students will be able to declare strings and display them on screen using loops.
- Students will be able to copy strings using string instructions.
- Students will know how OS displays messages on screen.

Video Memory

The text screen video memory for color monitors resides at **0xB8000**. The text screen resolution is **80 x 25**, which means that there are **25 rows**, and each row has 80 characters, therefore, there are 2000 characters. Each character requires **2 bytes** of information. One byte is for the **ASCII code of character** that is to be displayed on screen, and the other byte is for the **attributes associated with that character**. Attributes contain the foreground and background colors of the character. The **first nibble** (4 least significant bits) is reserved for **foreground (text) color**, and the **second nibble** (4 most significant bits) is **reserved for background color**. Each character's background and foreground colors can be set or obtained by reading a specific byte from memory.

Attributes

The various fields of the byte reserved for attributes are shown below. RGB are red, green, and blue colors. 'I' is the intensity of the color, and 'B' is for blink.

Background Color				Foreground Color			
B	R	G	B	I	R	G	B
Blink	Red	Green	Blue	Intensity	Red	Green	Blue

The following table shows the 4-bit color combination.

I/B	R	G	B	Color
0	0	0	0	Black
0	0	0	1	Blue
0	0	1	0	Green
0	0	1	1	Cyan
0	1	0	0	Red
0	1	0	1	Magenta
0	1	1	0	Brown
0	1	1	1	White
1	0	0	0	Gray
1	0	0	1	Light Blue
1	0	1	0	Light Green
1	0	1	1	Light Cyan
1	1	0	0	Light Red
1	1	0	1	Light Magenta
1	1	1	0	Yellow
1	1	1	1	Bright

Example#1:
Program to write "**HELLO**" on screen in red color with yellow background.

```
.model small

.stack 100h

.data

.code

Mov ax,0xb800
Mov es, ax
Mov si,0

Mov es:[si], 'H'
Inc si
Mov byte ptr es:[si], 11100100b
Inc si
Mov es:[si], 'E'
Inc si
Mov byte ptr es:[si], 11100100b
Inc si

Mov es:[si], 'L'
Inc si
Mov byte ptr es:[si], 11100100b
Inc si

Mov es:[si], 'L'
Inc si
Mov byte ptr es:[si], 11100100b
Inc si

Mov es:[si], 'O'
Inc si
Mov byte ptr es:[si], 11100100b
Inc si

.exit
```

Screen coordinates to linear address conversion

The screen is two-dimensional, having rows and columns, while the memory containing the screen's contents is linear. Each character on a screen has a row and column address, which is represented as (row, column), which can be translated to a linear memory address by using the following equations:

Equation 1 finds the linear address of the memory location where the ASCII code of the character to be shown on screen will be stored.

Equation 2 finds the linear address of the memory location where the character's screen attributes will be stored.

This linear address will be added to the base address, i.e., 0xB8000, to form the physical address.

Linear Address $_{(ASCII)}$ = 2 * (Row address * Columns per Row + Column address) **(eq. 1)**

Linear Address $_{(Attributes)}$ = 2 * (Row address * Columns per Row + Column address) + 1 **(eq. 2)**

Here,
Number of rows = 25
Columns per Row = 80
Since there are 80 columns in each row.

Example:

To display a text on the middle of the screen, we need to know its screen coordinates which are (12,40). These coordinates can be translated into physical address by using equation 1 as shown below.

Memory Address = 2 * (12 * 80 + 40) **=** 2000 **(eq. 3)**

Similarly, the offset address of the attributes of center character is

Memory Address = 2 * (12 * 80 + 40) +1 **=** 2000 + 1 = 2001 **(eq. 4)**

Example#2:
Program to write "**HELLO**" on center of the screen.

```
.model small

.stack 100h

.data

.code

Mov ax,0xb800
Mov es,ax
Mov si,2000

Mov es:[si], 'H'
add si,2

Mov es:[si], 'E'
add si,2

Mov es:[si], 'L'
add si,2

Mov es:[si], 'L'
add si,2

Mov es:[si], 'O'

.exit
```

Defining Strings

Strings are defined using a byte array as shown below.

Msg db "Hello World"

A string is terminated by a character which can be NULL or '$' so that the procedure handling string could know that the string has been ended. Since we are going to define our own code to process strings, therefore, we will place 0, i.e., NULL character at the end of a string as shown below.

Msg db "Hello World",0

Here ',' is the concatenation operator. It will concatenate strings on its both sides to form a single string.

Example#3:
Program that defines a string and displays it through loop/ jumps.

```
.model small
.stack 100h
.data

msg db "Hello World",0

.code
mov ax,@data
mov ds,ax

Mov ax,0xB800
Mov es,ax
Mov di,0

mov bx,offset msg
mov si,0

display:

mov al, [bx+si]
; mov one character from array to al
cmp al,0
; compare is that character is null
je end              ; if null, terminate loop
mov es:[di],al
; Otherwise, mov that character to video memory
add di,2
; add 2 to di, skipping the attribute part
inc si
; inc si to access the next character of array
jmp display      ; iterate loop

end:
.exit
```

```
.model small
.stack 100h
.data

msg db "Hello World"
len equ $- msg

.code
mov ax,@data
mov ds,ax
mov cx,len

Mov ax,0xB800
Mov es,ax
Mov di,0

LEA  BX, msg
; LEA stands for Load Effective Address
; same effect like this instruction (mov bx, offset msg)
mov si,0

display:

mov al, [bx+si]
; mov one character from array to al
mov es:[di],al
; mov that character to video memory

add di,2
; add 2 to di, skipping the attribute part
inc si
; inc si to access the next character of array
loop display      ; iterate loop

end:
.exit
```

String instructions

The following instructions facilitate programmers in moving strings from one memory location to another and comparing strings.

Instruction	Operands	Operation

LODSB
 	
Also see: LODSW
	No operands	• **AL = DS:[SI]**
		• if DF = 0 then
		  ○ SI = SI + 1
		else
		  ○ SI = SI – 1

STOSB
 	
Also See: STOSW
	No operands	• **ES:[DI] = AL**
		• if DF = 0 then
		  ○ DI = DI + 1
		else
		  ○ DI = DI – 1

MOVSB
 	
Also See: MOVSW
	No operands	• **ES:[DI] = DS:[SI]**
		• if DF = 0 then
		  ○ SI = SI + 1
		  ○ DI = DI + 1
		else
		  ○ SI = SI - 1
		  ○ DI = DI – 1

SCASB
 	
Also See: SCASW
	No operands	• **AL - ES:[DI]**
		• set flags according to result:
		OF, SF, ZF, AF, PF, CF
		• if DF = 0 then
		  ○ DI = DI + 1
		else
		  ○ DI = DI - 1

Example#4:
Program to write “Hello world” on screen using string movement instructions.

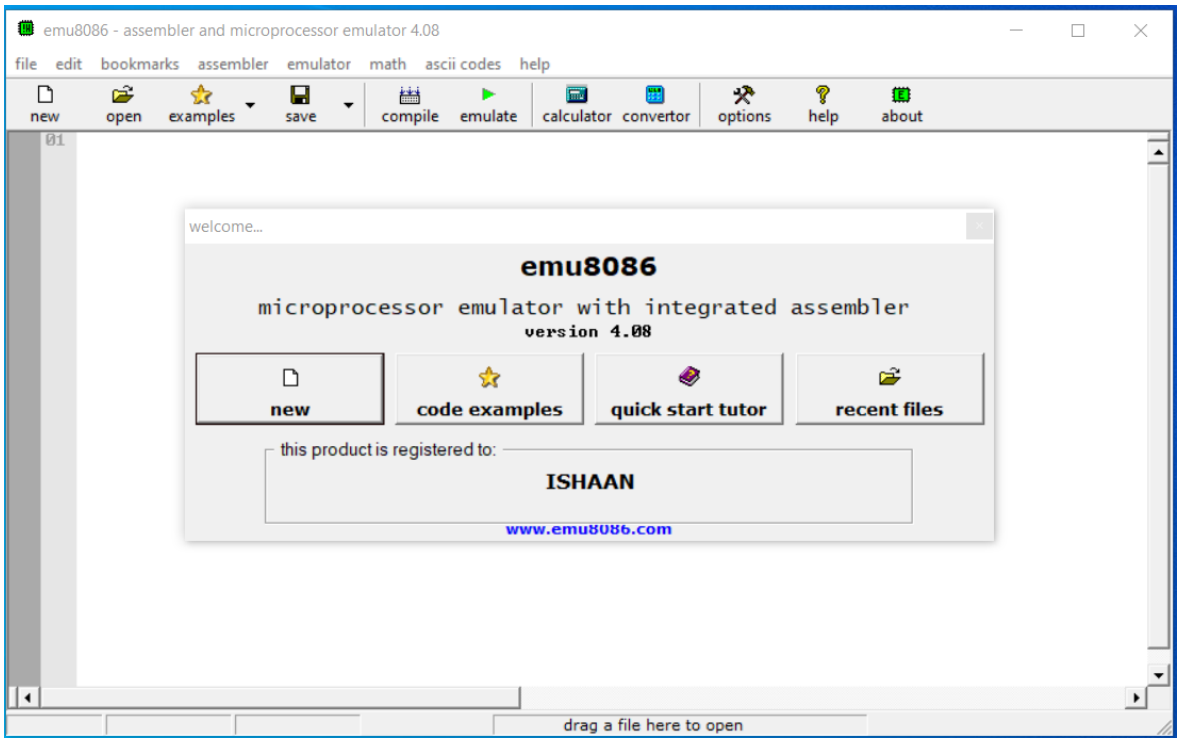
```
.model small

.stack 100h

.data

msg1 db "Hello World",0
msg2 db 12 dup(?)

.code

mov ax,@data
mov ds,ax
mov es,ax

Mov si,offset msg1        ; moving offset of msg1 to SI
Mov di,offset msg2        ; moving offset of msg2 to DI


mov cx,11                 ; initializing counter to iterate loop 11 times

copy:
movsb                     ; copying msg1 to msg2

loop copy


.exit
```

Emu8086 Tutorial Step by Step

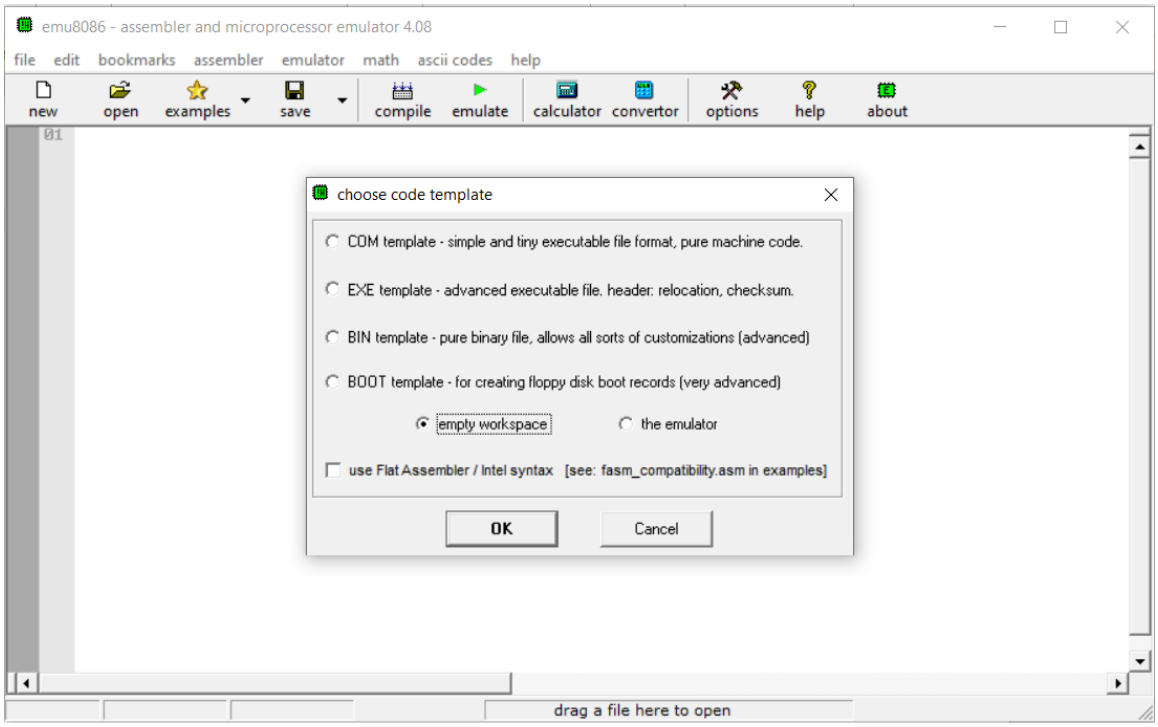
Step-1

Double click on the icon on the desktop 

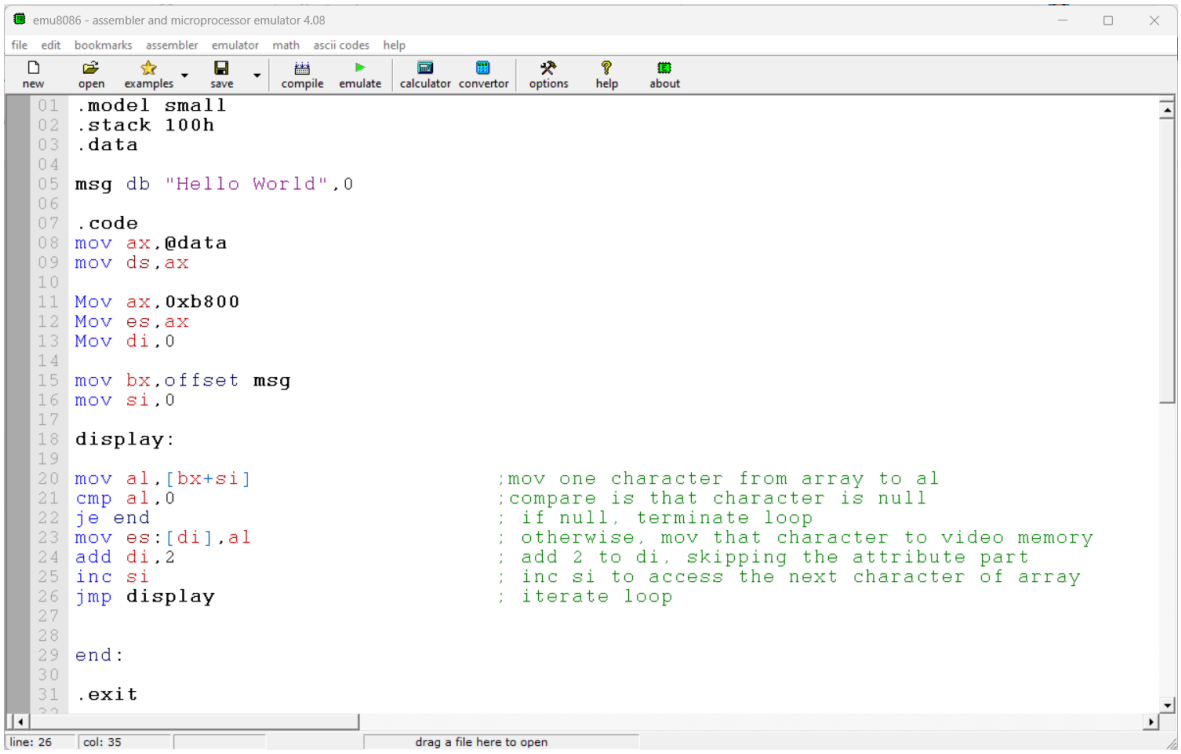
Step-2
The following window will appear. Click on “new”.



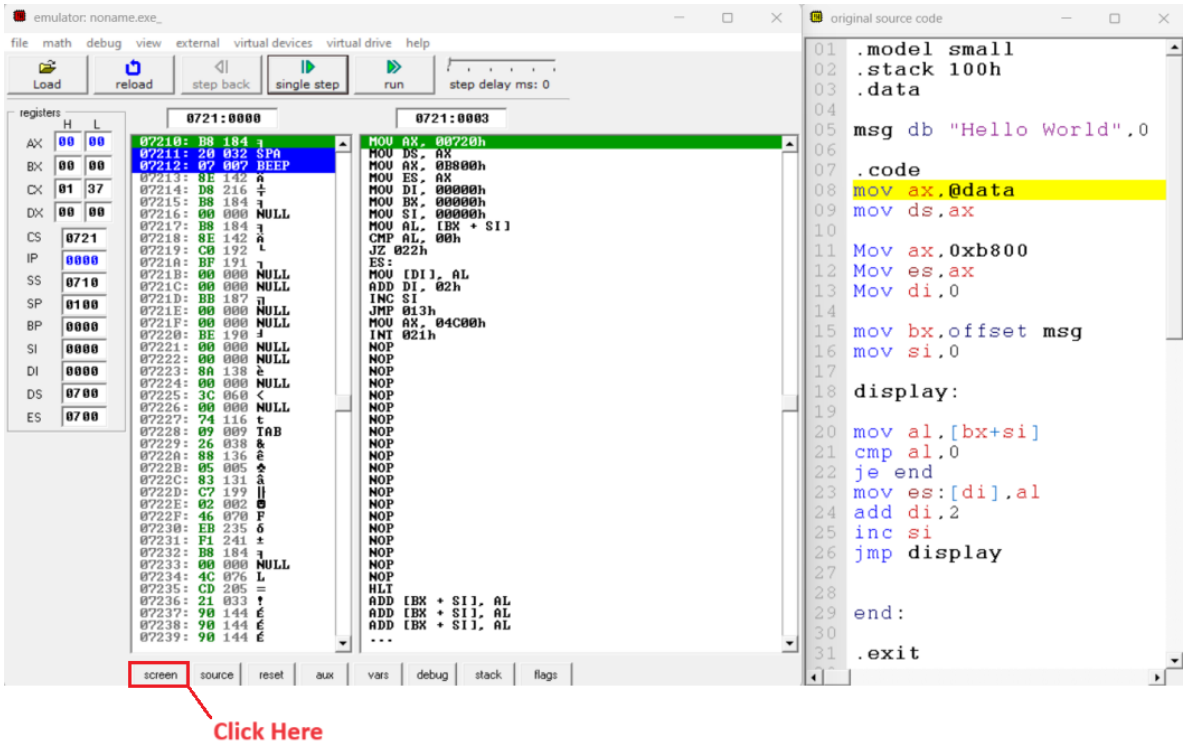
Step-3:
Click on the “empty workspace” and press “OK”.



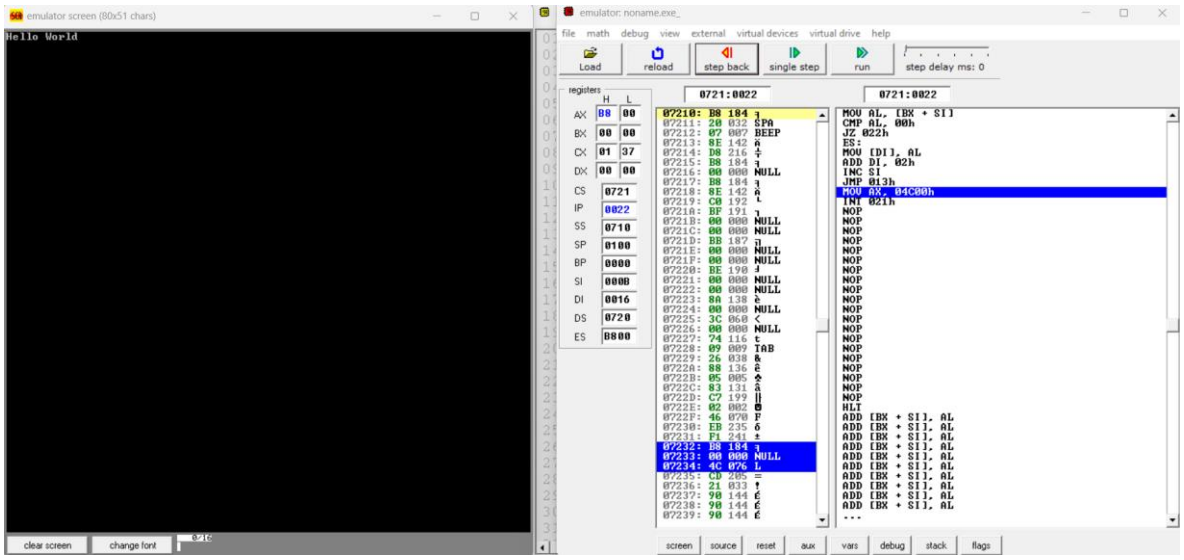
Step-4:
Type the code given in example#3 and click on “emulate”.



Step-5:
Click on the screen button.



Step-6:
Keep clicking on “Single step” and observe the working of the program.



Lab-8 Exercise

Task-1

Write a procedure named "print" that takes a null-terminated string as a parameter and displays it on the screen at the position of the cursor. The cursor increments accordingly whenever a string is displayed. (Hint: You need to keep cursor information in a global variable.)

Task-2

Newline, carriage return, and tab characters are not displayed, but they instruct the display function to perform specific operations. The newline character moves the cursor to the next row in the same column, while carriage return moves the cursor to the first column of the current row. Similarly, the tab character inserts eight spaces.

Add the functionality of newline, cret, and tab to the print procedure you created in task 1. (Hint: See ASCII code of these characters from ASCII table)

Task-3

There are 25 rows from 0 to 24 and 80 columns in a row from 0 to 79. When a column address reaches 80, it is reset to 0, and the row is incremented by 1. When the row address reaches 25, the screen moves up one line, called "scrolling up."

Incorporate the functionality of scroll up to the print procedure you created in task 1.

The C++ code for scroll up is given below:

```
for (i = 0; i < 24; i++)
    for (j = 0; j < 80; j++)
        screen[i][j] = screen[i + 1][j];

for (j = 0; j < 80; j++)
    screen[24][j] = " ";
```

Lab 9: Software Interrupts, Hooking Software Interrupts & Exceptions

Learning Outcomes

- Students will know about interrupts and their types.
- Students will know how software interrupts are invoked.
- Students will know how software interrupts are serviced.
- Students will know various DOS and BIOS interrupts.
- Students will be able to hook interrupts.
- Students will be able to handle exceptions.

Interrupts

The notion of "interrupt" was originally conceived to allow devices to interrupt the current operation of the CPU. For example, whenever a key is pressed, the 8086 must be notified to read a key code from the keyboard buffer. Such interrupts are called hardware interrupts. On the other hand, the software interrupts are generated by the programs to get a service from the operating system or the BIOS. For example, to read a file from disk or a character from the keyboard.

Another type of interrupt is an exception, which is generated automatically when an error occurs during the execution of an instruction. For example, divide by zero or overflow.

In this manual, we are going to discuss only software interrupts. The software interrupts are invoked by using an "INT" instruction, which is followed by an interrupt number. There are 256 interrupts in the system. Some are reserved for exceptions, and the rest can be used as hardware or software interrupts.

The processor uses the interrupt number to index the Interrupt Vector Table (IVT) to get the address of a special type of procedure called an Interrupt Service Routine (ISR). The ISR is executed whenever the interrupt is generated. There are 256 entries in IVT, and each entry stores the logical address of a respective ISR. IVT resides in memory at **physical address 0x00000**. Each entry stores the **logical address**, which is 4 bytes; the total size of the IVT is **256 x 4B = 1 KB**. To index the IVT, the processor multiplies the interrupt number by 4 to get the physical address of the entry against that interrupt number.

To hook an interrupt in this context means to change the ISR address of an interrupt in the Interrupt Vector Table. Before doing so, one needs to write ISR first and then add its logical address, consisting of Segment: Offset, to IVT.

ISR is a special procedure that is called whenever an interrupt is generated. Whenever an ISR is called as a result of an interrupt, the flag register is pushed onto the stack along with IP and CS. Therefore, unlike procedures, the ISRs return control by issuing "IRET" instructions instead of "RET" instructions. The IRET instruction also pops the flag register from the stack, whereas the RET instruction does not.

The control transfer to an ISR is similar to that of a procedure call. Before transferring control to the ISR, the processor first saves the address of the next instruction on the stack, called the return address. 8086 also saves the FLAG register on the stack. When the ISR returns, the FLAG register and return address are restored from the stack.

BIOS and DOS Interrupts

There are some interrupts that are provided by the **BIOS** to perform **basic I/O operations**, and some are provided by the **DOS**. There are multiple services against each interrupt number. The following table shows a few interrupts with few of its services. The service number is moved to the AH register before invoking an interrupt instruction. Depending on your needs, you may find more interrupts and services in your textbook.

Interrupt Vectoring

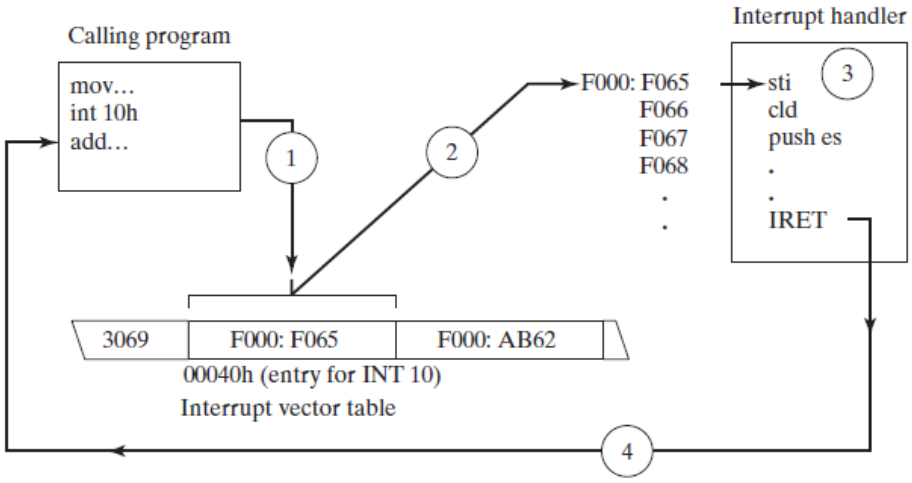
The CPU processes the INT instruction using the interrupt vector table, which, as we've mentioned, is a table of addresses in the lowest 1024 bytes of memory. Each entry in this table is a 32-bit segment-offset address that points to an interrupt handler. The actual addresses in this table vary from one machine to another. Figure 14-2 illustrates the steps taken by the CPU when the INT instruction is invoked by a program:

Step 1: The operand of the INT instruction is multiplied by 4 to locate the matching interrupt vector table entry.

Step 2: The CPU pushes the flags and a 32-bit segment/offset return address on the stack, disables hardware interrupts, and executes a far call to the address stored at location (10h * 4) in the interrupt vector table (F000:F065).

Step 3: The interrupt handler at F000:F065 executes until it reaches an IRET (interrupt return) instruction.

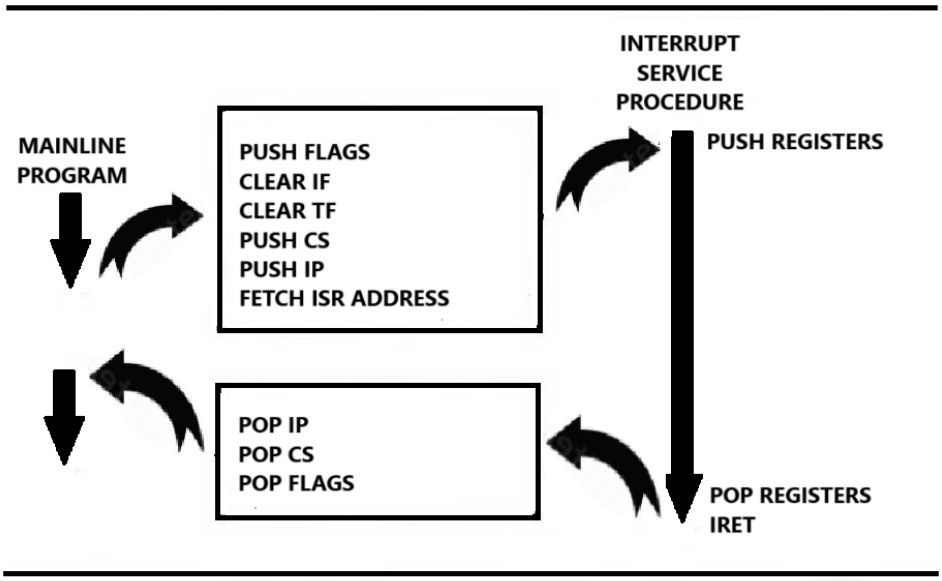
Step 4: The IRET instruction pops the flags and the return address off the stack, causing the processor to resume execution immediately following the INT 10h instruction in the calling program.



Interrupt Vectoring Process

8086 Interrupt maps

When an interrupt is generated, the following steps are performed.



8086 Interrupt Response

1. It decrements stack pointer by 2 and pushes the flag register on the stack.
2. It disables the INTR interrupt input by clearing the interrupt flag in the flag.
3. It resets the trap flag in the flag register.
4. It decrements stack pointer by 2 and pushes the current code segment register contents on the stack.
5. It decrements stack pointer by 2 and pushes the current instruction pointer contents on the stack.
6. It does an indirect jump at the start of the procedure by loading the CS and IP values for the start of the interrupt service routine (ISR).
7. An IRET instruction at the end of the interrupt service procedure returns execution to the main program.
8. The ISR should push registers to stack before using them and pop them before IRET.

Interrupt#	Service#	Description	BIOS/DOS
0x21	**0x1**	read character from standard input, with echo, result is stored in **AL**. if there is no character in the keyboard buffer, the function waits until any key is pressed.	**DOS**
0x21	**0x7**	read character from standard input, without echo, result is stored in **AL**. if there is no character in the keyboard buffer, the function waits until any key is pressed.	**DOS**
0x21	**0x2**	write character to standard output device. entry: **DL = Character** to write. After execution **AL = DL**.	**DOS**
0x21	**0x9**	Print string at **DS: DX** on screen. String must be terminated by '**$**'.	**DOS**
0x21	**0x4C**	Terminate program and return control to DOS	**DOS**
0x10	**0x2**	Set Cursor position. **DH ← ROW#** **DL ← COL #** **BH ← Page# (i.e., 0)**	**BIOS**
0x10	**0x9**	Display character & attributes on current cursor position. **AL = character to display.** **BH = page number (i.e. 0).** **BL = Color attributes.** **CX = number of times to write character.**	**BIOS**

Example#1:
Program to write "Hello" on screen and terminate itself using interrupts.

```
.model small

.stack 100h

.data
Msg db "Hello$"
.code

Mov ax,@data
Mov ds,ax

Mov ah,0x9                    ;display string on screen
Mov dx,offset msg
Int 0x21

Mov ah,0x4c                   ;terminating program
Int 0x21
```

Example#2:
Program to read a single-digit number as character from keyboard and convert it to a number and store in variable.

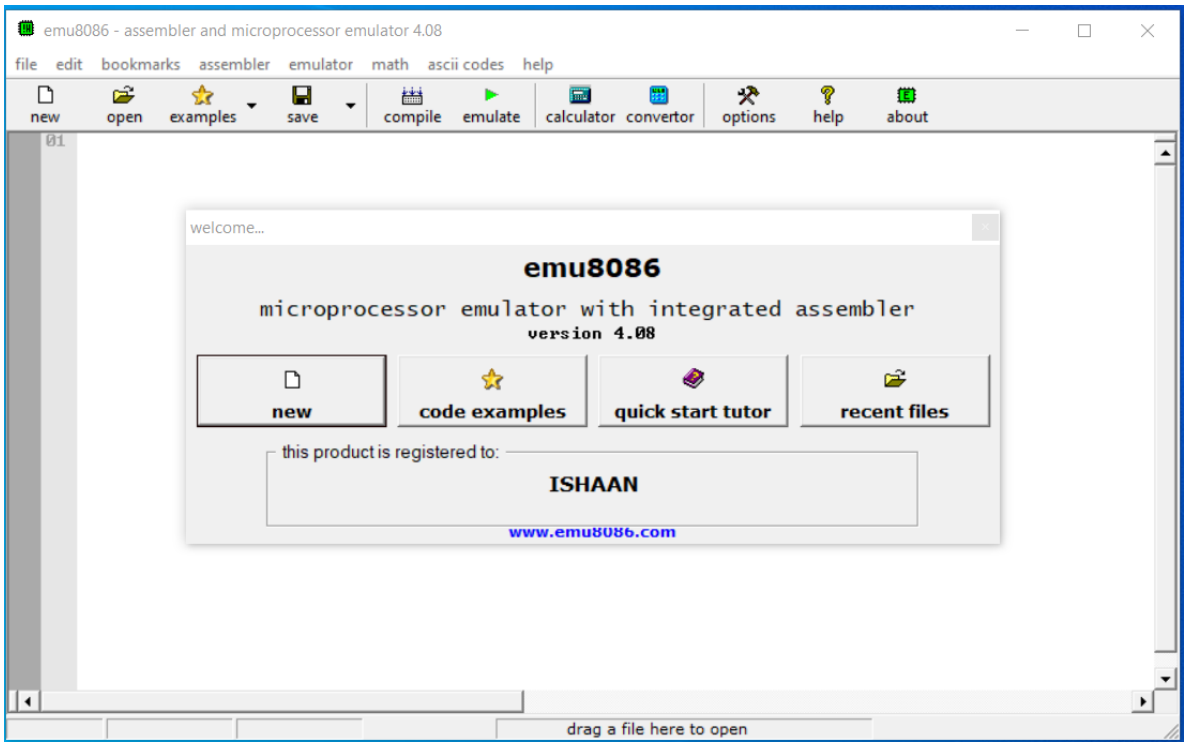
```
.model small

.stack 100h

.data

n db ?

.code

Mov ax,@data
Mov ds,ax

mov ah,0x1    ; read character from keyboard
int 0x21

sub al,0x30   ; subtract 0x30 from the character to make it a number
mov n,al      ; store the number to variable

mov ah,0x4c
int 0x21
```

Emu8086 Tutorial Step by Step

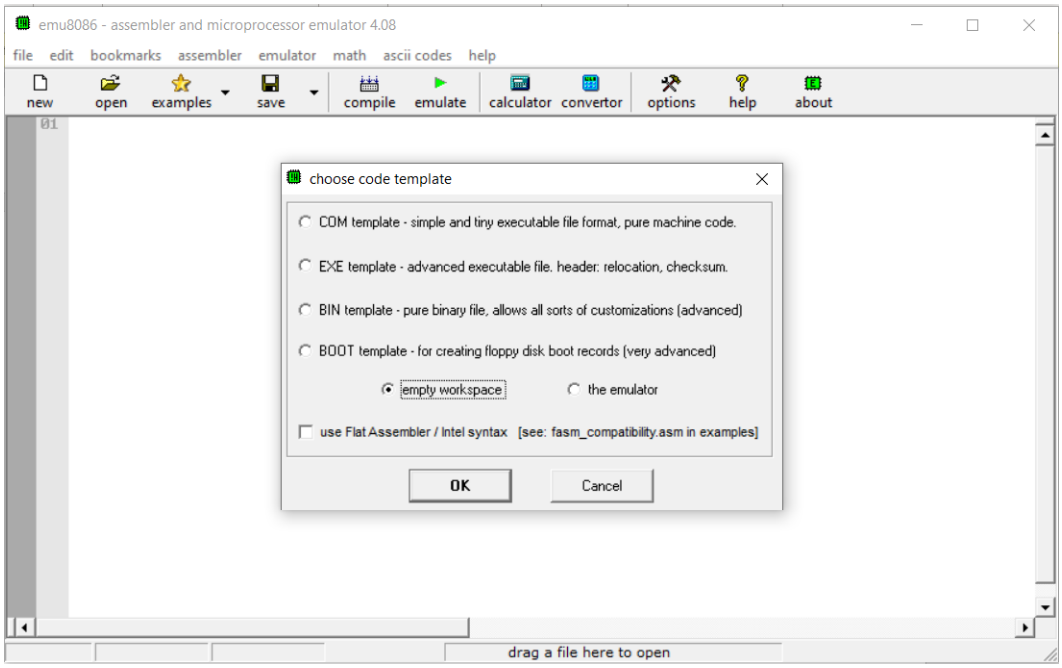
Step-1

Double click on the icon on the desktop 

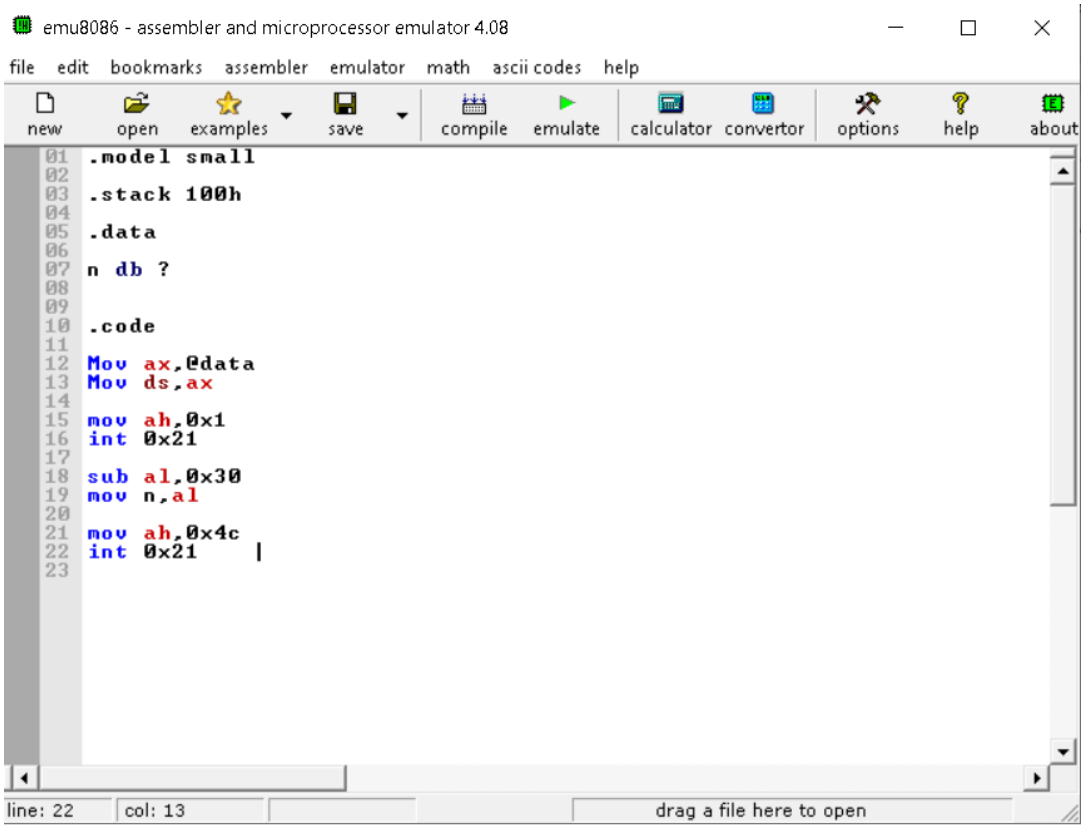
Step-2
The following window will appear. Click on “new”.



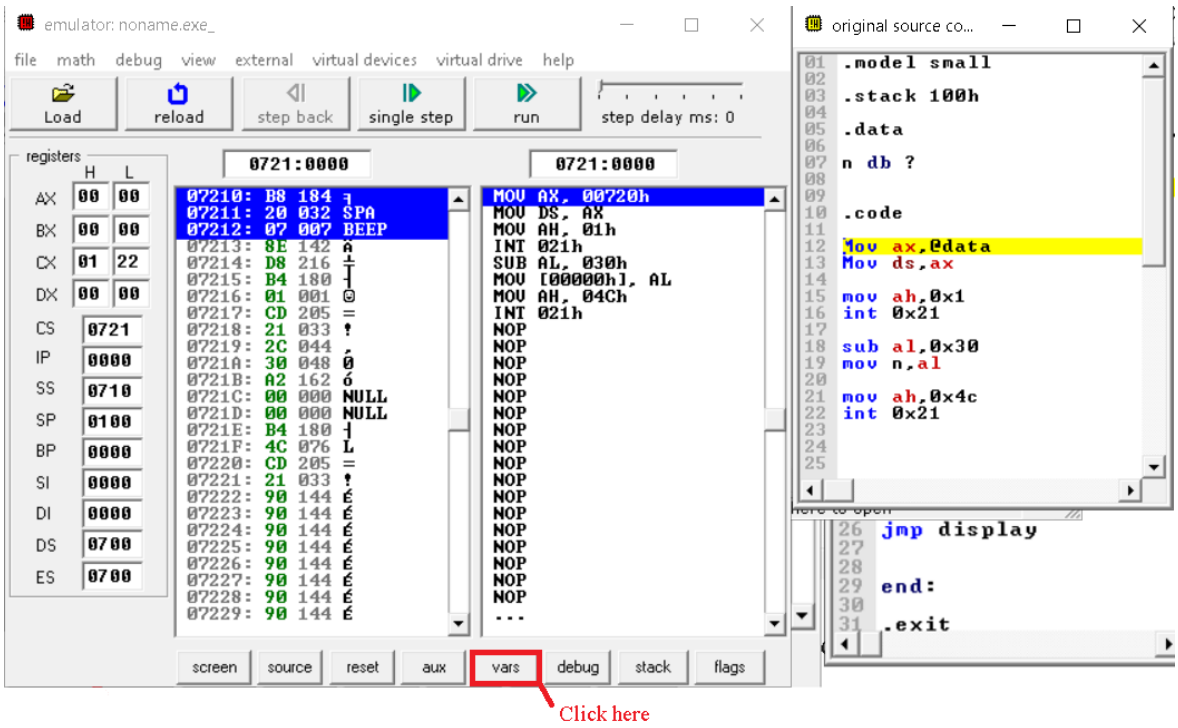
Step-3:
Click on the "empty workspace" and press "OK".



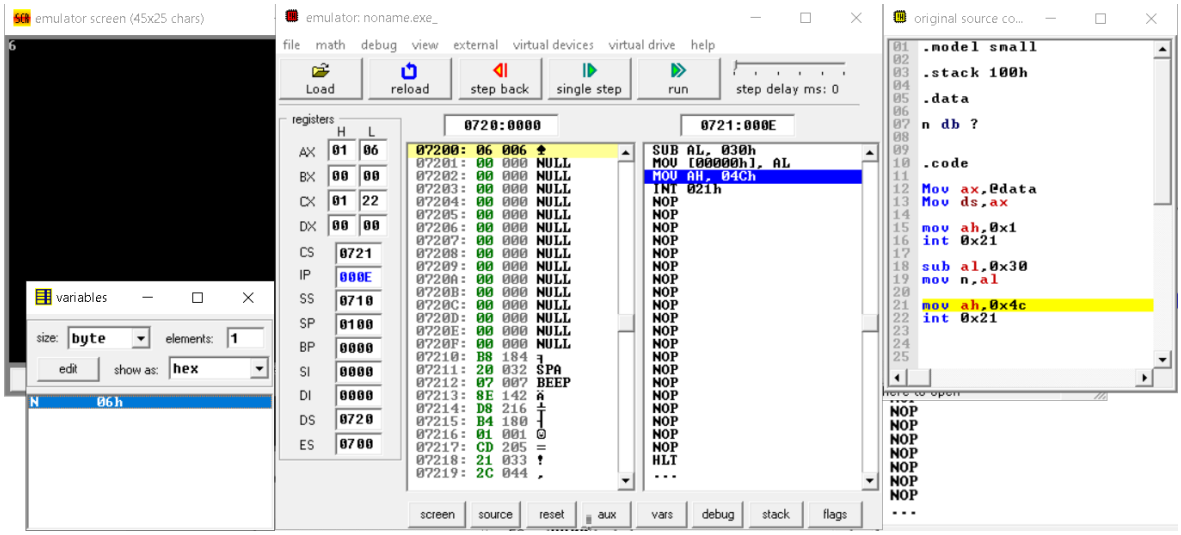
Step-4:
Type the code given in example#2 and click on "emulate".



Step-5:
Click on the vars button.



Step-6:
Keep clicking on "Single step" till the last instruction and provide input by pressing a numeric key when prompted.



Hooking Software Interrupts & Exceptions

INT#0
This interrupt is generated when a number is divided by zero or the result of division overflows. The following code handles this exception.

Example#1:
Program to write ISR for "**DIV BY Zero**" exception i.e., **INT#0** and hook it.

```
.model small

.data


.code

mov ax,@data
mov ds,ax

mov ax,0
mov es,ax

mov word ptr es:[0 * 4 + 0],isr0     ; offset address of ISR0
mov word ptr es:[ 0 * 4 + 2],cs       ; Segment address of ISR0


mov ax,100
mov bl,0

div bl                          ;Dividing a number by zero to check if the hooking


.exit


ISR0 proc

jmp l@1
msg: db "Divide by Zero Exception!$"
l@1:
mov dx,offset msg
mov ah,9
int 0x21

IRET

ISR0 endp
```

INT#4
This interrupt is generated in response to an INTO instruction that checks the overflow flag to generate the interrupt. If the overflow flag is not set, **INT#4** will not be generated.

Example#2:
Program to hook "**OVERFLOW**" exception, i.e., **INT#4**.

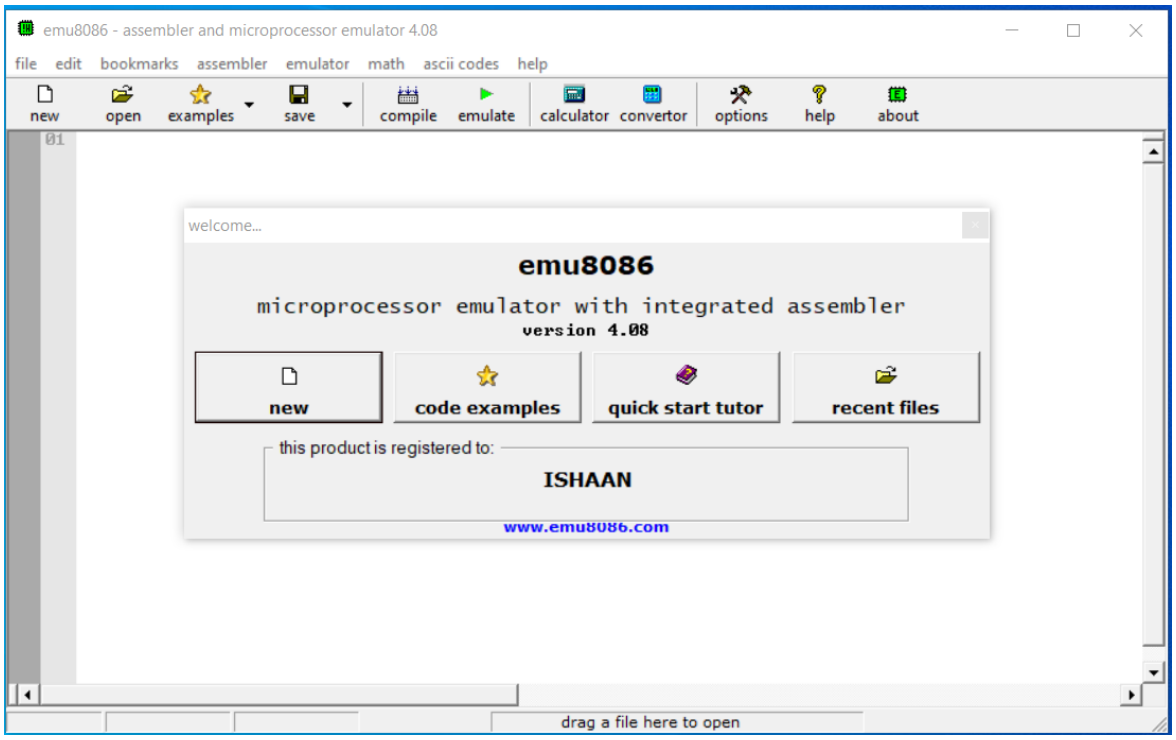
```
.model small

.data


.code

mov ax,@data
mov ds,ax

mov ax,0
mov es,ax

mov word ptr es:[ 4 * 4 + 0],isr4
mov word ptr es:[ 4 * 4 + 2],cs

mov al,127
mov bl,1
add al,bl

into                    ;if OF is set, generate int#4

.exit

isr4 proc

jmp l@1
msg: db "Overflow flag is set$"
l@1:
mov dx,offset msg
mov ah,9
int 0x21

iret
isr4 endp
```

Emu8086 Tutorial Step by Step

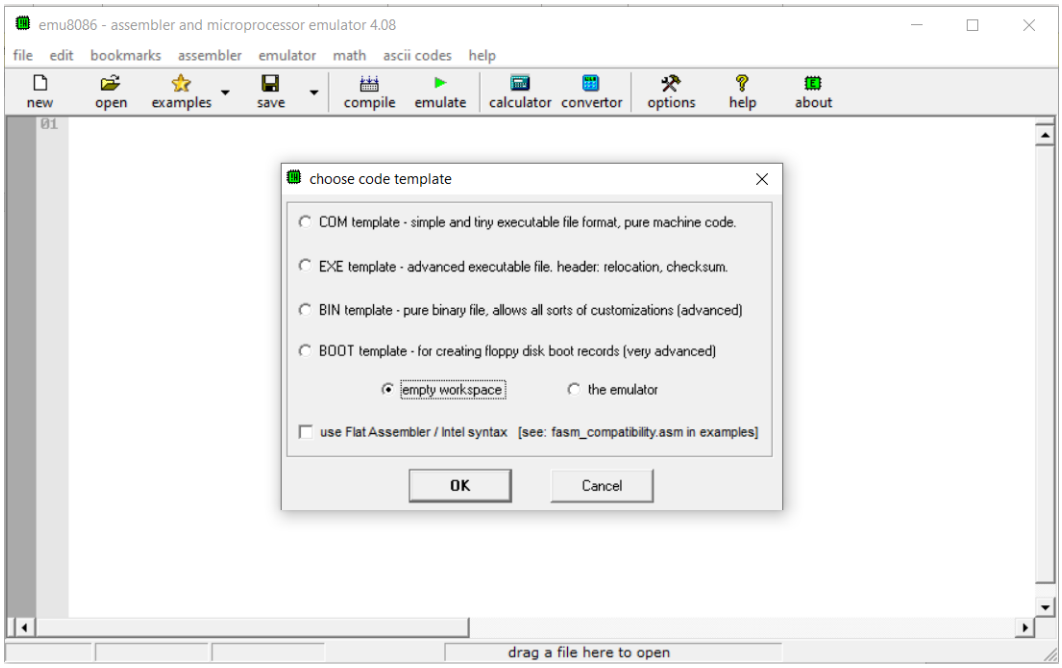
Step-1

Double click on the icon on the desktop 

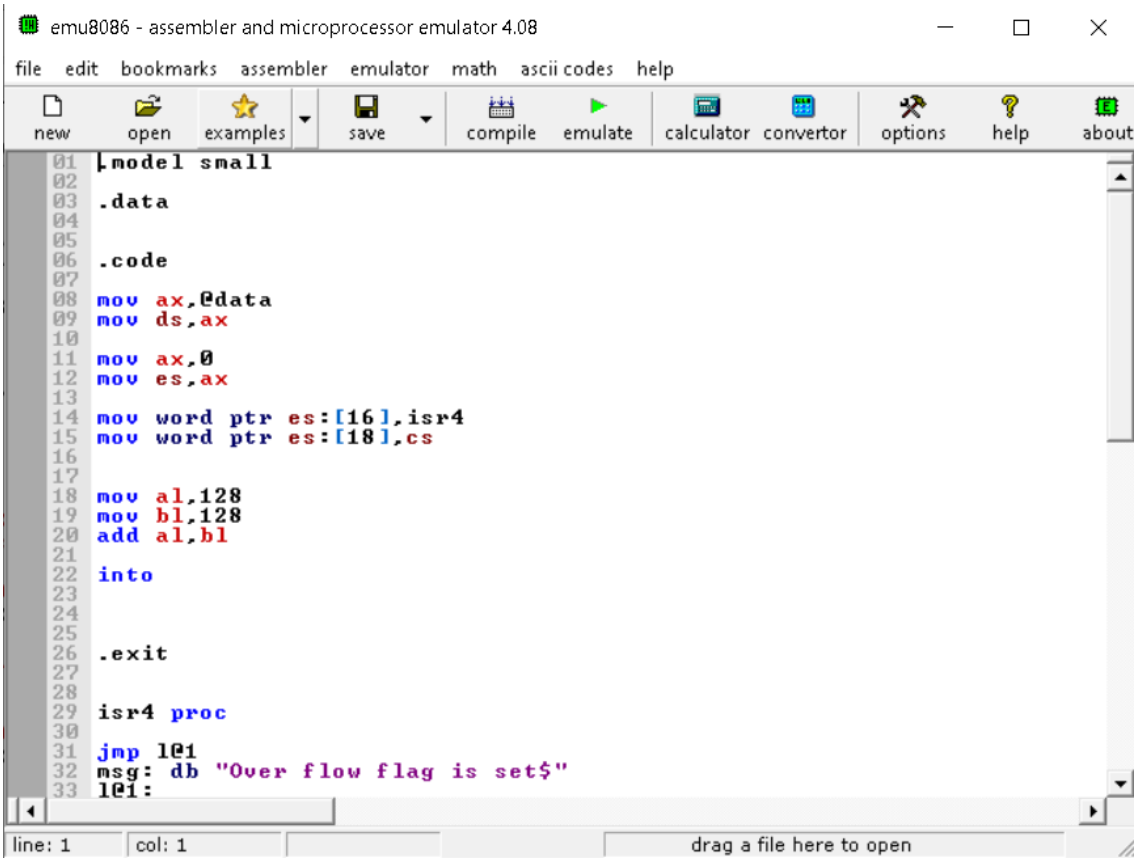
Step-2
The following window will appear. Click on "new".



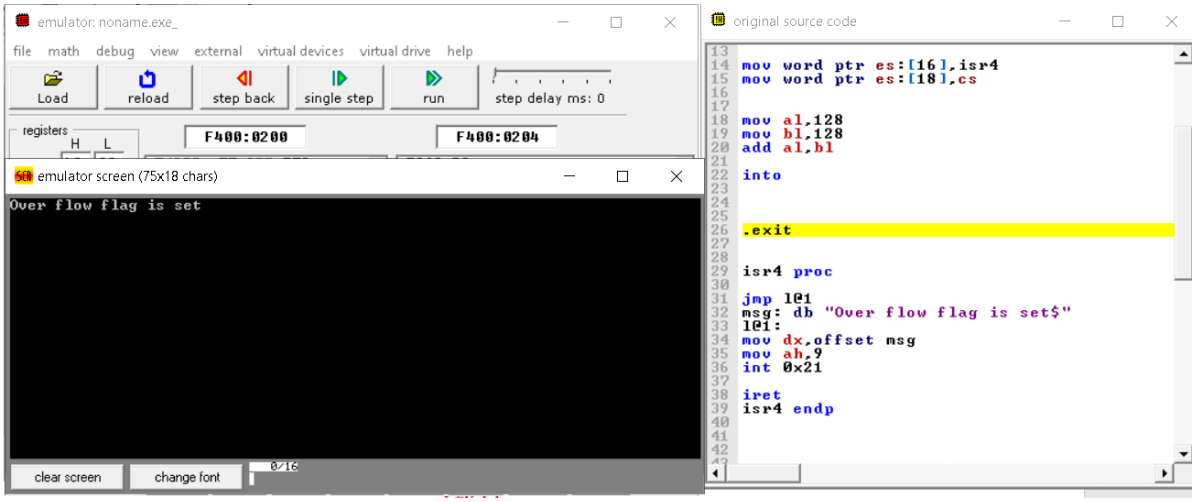
Step-3:
Click on the "empty workspace" and press "OK".



Step-4:
Type the code given in example#2 and click on "emulate".



Step-5:
Keep clicking on "Single step" till the last instruction and observe the behavior of the program.



Lab-9 Exercise

Task-1

Write a code that inputs a 16-bit integer number in hexadecimal radix from the keyboard and store it in a variable.

Task-2

Write a code that prints a number stored in a 16-bit variable on screen in a hexadecimal radix.

Task-3

Write a program that inputs two 16-bit hexadecimal numbers from the user and displays their sum on the screen.

Task-4

Write a program that hooks 0x65 interrupt with the following services. The ISR should preserve all the registers except AX.

Service#	Description
0x1	Add two words pointed by SI and DI and store the result in the memory pointed by BX. **[BX] ← [SI] + [DI]**
0x2	Multiply two words pointed by SI and DI and store result in the memory pointed by BX. **[BX] ← [SI] x [DI]**
0x3	Divide the word pointed by SI by the byte pointed by DI and store quotient and remainder in the AL and AH registers, respectively. **AL ← [SI]/[DI]** **AH ← [SI] % [DI]**

CSCS3543 Computer Organization and Assembly Language

Lab 10: 32-bit Inline Assembly Language Programming in Visual Studio

Learning Outcomes

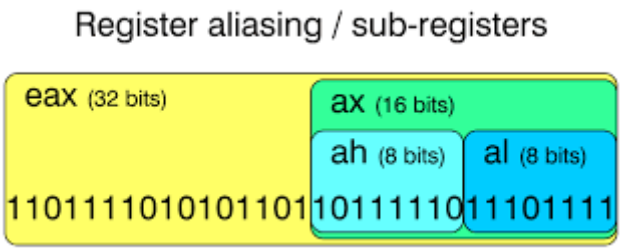
- Students will be able to write 32-bit assembly language programs.
- Students will be able to run their assembly language code on the underlying processor natively, without any virtual environment.
- Students will learn about addressing modes in 32-bit programming.
- Students will realize how assembly language optimizes their code in terms of number of instructions and speed.
- Students will write relevant parts of the code in assembly language while keeping the rest in C/C++.

Introduction to 32-bit programming

Intel 80386 was the first 32-bit processor of the Intel family of microprocessors. It has 32-bit addresses and data buses. The protected mode was also introduced in this processor that limits the operation of user programs. It doesn't allow user programs to go beyond their memory limits assign to them and directly access IO devices. The size of existing registers, except the segment registers, were extended to 32-bit as shown below.

32-bit	31 16	15 8	7 0	16-bit
EAX		AH	AL	AX
EBX		BH	BL	BX
ECX		CH	CL	CX
EDX		DH	DL	DX
EBP		BP		
ESI		SI		
EDI		DI		
ESP		SP		

32-bit General-Purpose Registers



Addressing Modes

The addressing modes are the same.

In the 80386 and later processors, registers **EAX, EBX, ECX, EDX, EBP, EDI, ESI, and ESP** can be used in register indirect addressing mode. By default, **EBP and ESP** reference the stack segment (SS), whereas the other registers typically reference the data segment (DS), unless overridden by a segment override prefix.

Example:
The MOV [**EDX**], CL instruction copies the byte sized contents of register CL into the data segment memory location addressed by EDX.

In 80386 and above, any two registers from **EAX, EBX, ECX, EDX, EBP, EDI, or ESI** may be combined to generate the memory address.

> **Example:**
>
> The MOV [**EAX+EBX**], CL instruction copies the byte sized contents of register CL into the data segment memory location addressed by EAX plus EBX.

The 80386 and above **use any 32-bit register to address memory**. (Example: MOV AX, [ECX] or MOV AX, ARRAY[EBX]. The first instruction loads AX from the data segment address formed by ECX. The second instruction loads AX from the data segment memory location ARRAY plus the contents of EBX.

Inline Assembler (Microsoft Specific)

Assembly language serves many purposes, such as improving program speed, reducing memory needs, and controlling hardware when writing kernel modules. You can use the inline assembler to embed assembly-language instructions directly in your C and C++ source programs without extra assembly and link steps. The inline assembler is built into the compiler, so you don't need a separate assembler such as the Microsoft Macro Assembler (MASM). Because the inline assembler doesn't require separate assembly and link steps, it is more convenient than a separate assembler. Inline assembly code can use any C or C++ variable or function name that is in scope, so it is easy to integrate it with your program's C and C++ code. And because the assembly code can be mixed with C and C++ statements, it can do tasks that are cumbersome in C or C++ alone.

The__asm keyword invokes the inline assembler and can appear wherever a C or C++ statement is legal. It cannot appear by itself. It must be followed by an assembly instruction, a group of instructions enclosed in braces, or, at the very least, an empty pair of braces.

Example

```
__asm mov eax,ebx
```

or

```
__asm
{
Mov eax,ebx
}
```

Using and Preserving Registers in Inline Assembly (Microsoft Specific)

In general, you should not assume that a register will retain its value when an __asm block begins. Register values are not guaranteed to be preserved across separate __asm blocks. If you end one block of inline assembly and begin another, you cannot rely on registers in the second block to retain their values from the first block. An __asm block inherits the register values resulting from the normal flow of execution.

When using __asm to write assembly language in C/C++ functions, you are responsible for preserving callee-saved registers (EBX, ESI, EDI, and EBP) if your code modifies them. However, caller-saved registers (EAX, ECX, and EDX) do not need to be preserved, as they are expected to be modified by function calls. Additionally, you should preserve segment registers (DS, SS) and the stack-related registers (ESP and EBP), as well as the flags register (EFLAGS), if your code relies on specific flag settings across the inline assembly block.

Example#1:

Program to add two integer numbers using the inline assembly.

```
#include<iostream>
using namespace std;


int main(void)
{
	int a = 10, b = 20, c = 0;
	__asm
	{
		mov ebx,a
		add ebx,b
		mov c,ebx
	}
	cout << "Sum of " << a << " and " << b << " is " << c << endl;

}
```

Example#2:
Program to access array using inline 32-bit assembly. Both codes are incrementing every element of the array.

```
#include<iostream>
using namespace std;

int array1[] = {1,2,3,4,5};
int main(void)
{
        __asm
        {
                push ebp
                mov ebp, offset array1
                mov esi, 0
                mov ecx, 5
        l1:
                inc dword ptr[ebp + esi]
                add esi, 4
                loop l1
                pop ebp
        }
        for (int i = 0; i < 5; i++)
                cout << array1[i];
}
```

```
#include<iostream>
using namespace std;

int array1[] = {1,2,3,4,5};
int main(void)
{

        for (int i = 0; i < 20; i=i+4)
        {

                __asm
                {
                mov esi,i
                inc dword ptr [array1+esi]

                }
        }
        for (int i = 0; i < 5; i++)
                cout << array1[i];
}
```

Example#3:
Program to rotate left each element of the array using C++ and inline assembly. Observe that assembly language makes our task simpler.

Rotate left in C++

```
#include<iostream>
using namespace std;
int main()
{
int array1[] = { 1,2,3,4,5 };
for (int i = 0; i < 5; i++)
{
array1[i] = (array1[i] << 1) | (array1[i] >>31);
}
        for (int i = 0; i < 5; i++)
        {
                cout << array1[i] << endl;
        }
        return 0;
}
```

Rotate left using inline assembly

```
#include<iostream>
using namespace std;
int main(void)
{
        int array1[] = { 1,2,3,4,5 };
        for (int i = 0; i < 20; i=i+4)
        {
        __asm
        {
        mov eax,i
        rol dword ptr [array1+eax],1
        }
        }
        for (int i = 0; i < 5; i++)
        {
                cout << array1[i] <<
endl;
        }
}
```

Example#4:
Program to add 1 to each element of a 2D array

```
#include<iostream>
using namespace std;
int main(void)
{
	int array1[3][4] = { {1,2,3,1},{4,5,6,1},{7,8,9,1} };
	int index = 0;
	for (int i = 0; i < 3; i++)
		for (int j = 0; j < 4; j++)
		{
			index = (i * 4 + j) * sizeof(int);
			__asm
			{
				mov eax, index
				add dword ptr [array1+eax],1
			}
			cout << "(" << i << "," << j << ")" << "->" << array1[i][j] << endl;
		}
}
```

Example#5:
Write a C/C++ program to call function written in assembly language.

Create another file with the .asm extension within the project. In Microsoft Visual Studio, a project can include multiple source files with different extensions. Here, we name the assembly file library.asm since it will contain reusable functions, similar to a library. Write assembly code in library.asm file and C++ code in the source.cpp file as shown below.

Source.cpp file	library.asm file

#include<iostream>
using namespace std;
 	
extern "C" int sum1(int a, int b);
 	
int main()
{
 	
cout << "sum is " << sum1(4, 5);
 	
return 0;
}
	.model flat
	
	.CODE
	
	_sum1 PROC
	
	push ebp
	mov ebp,esp
	mov eax,[ebp+8]
	add eax,[ebp+12]
	pop ebp
	ret
	
	_sum1 endp
	END

Convention:

- **Flat model** means a single linear address space (used in modern 32-bit systems, unlike segmented memory in older architectures like 16-bit mode).
- **.CODE** tells MASM that the following code belongs to the **code segment.**
- **_sum** declares the function as a procedure (PROC). The leading underscore (_) suggests it follows stdcall conventions, typically for calling from C/C++. The caller will call this function as sum( ) without (_) underscore.
- Assembly functions (written in TASM, MASM, NASM) follow C-style calling conventions. **extern "C"** ensures the function name remains unaltered, so it can be called from C++.

Example 6:
Program to call function written in C/C++ in assembly language program.

Source.asm	**Library.cpp**

.386
.model flat, c
extern _add:PROC
 	
.code
main PROC
  push 5
  push 10
  call _add ; Call add(10, 5)
  add esp, 8 ; Clean up stack
  ret
main ENDP
END main
	int add(int a, int b) {
	  return a + b;
	}

Convention:

- **.386** directive tells the assembler that the code is for Intel 80386 and later processors. It enables 32-bit instructions and registers.
- **.model flat** specifies a flat memory model, meaning there's a single continuous memory space (used in 32-bit applications).
- **C** specifies that the C calling convention is used.
- **extern** declares that **_add** is an external function, meaning it is defined in another file (likely a C file).
- **_add** In MSVC, C function names are prefixed with **_** in stdcall calling convention.
- **:PROC** indicates that **_add** is a procedure.

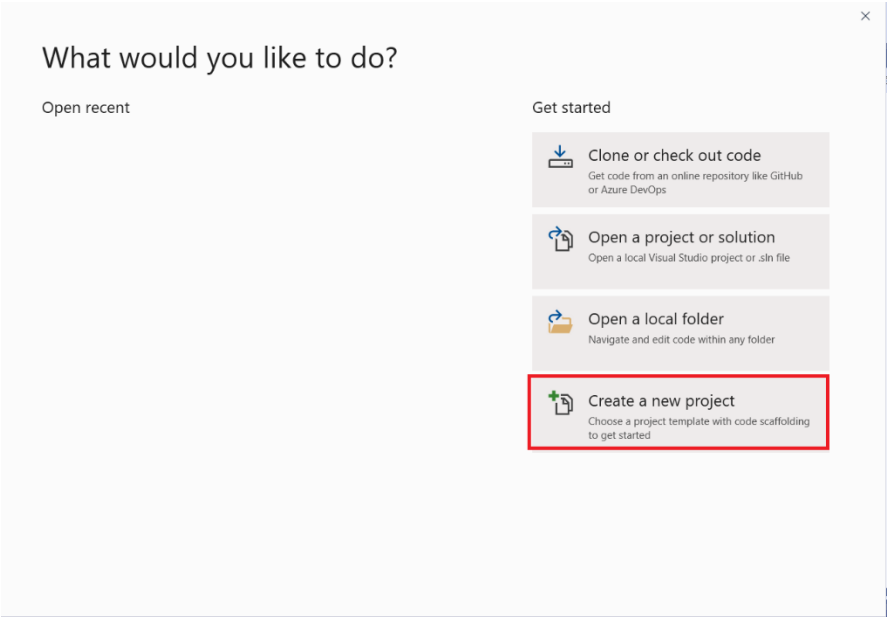
How to RUN 32-bit inline assembly language programs

Step#1:

Open Visual Studio 2019

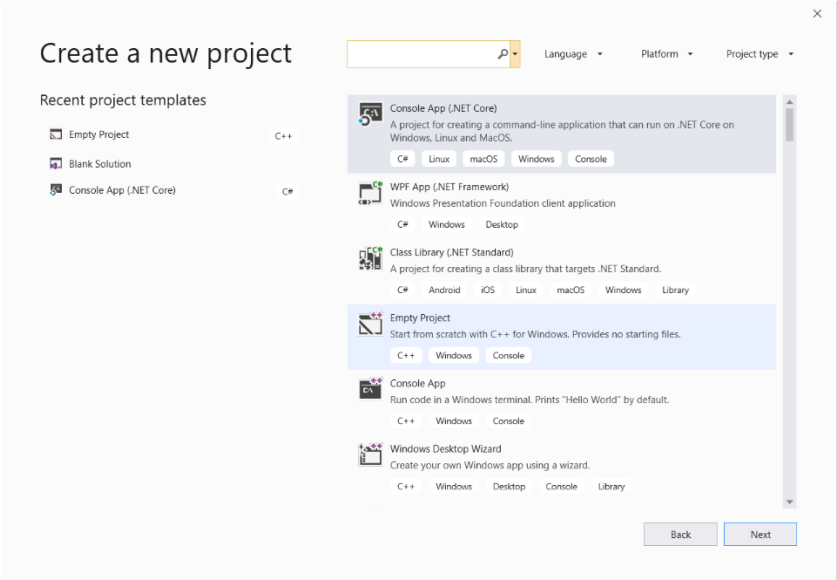
Step#2:

Click on “Create a new project”



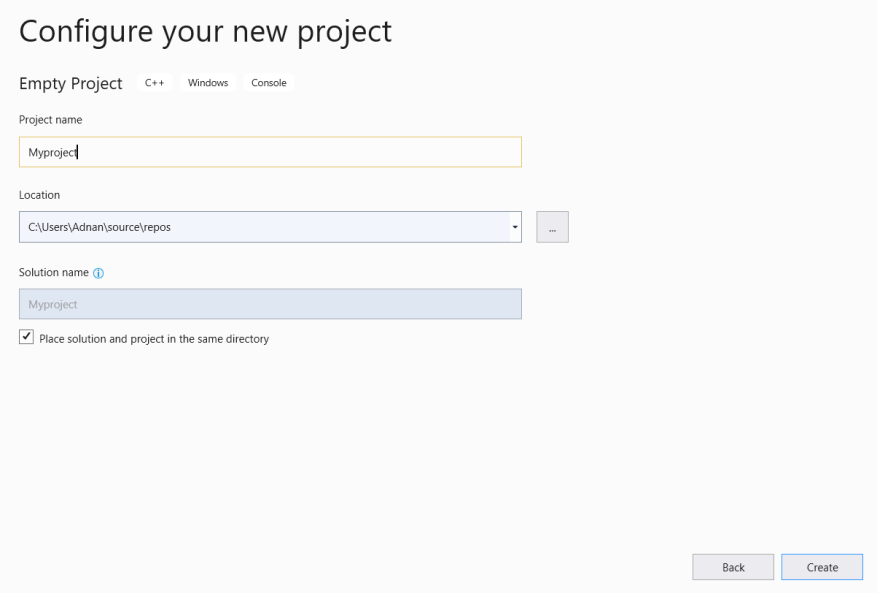
Step#3:

Click on C++ under “Empty Project” and press “Next”



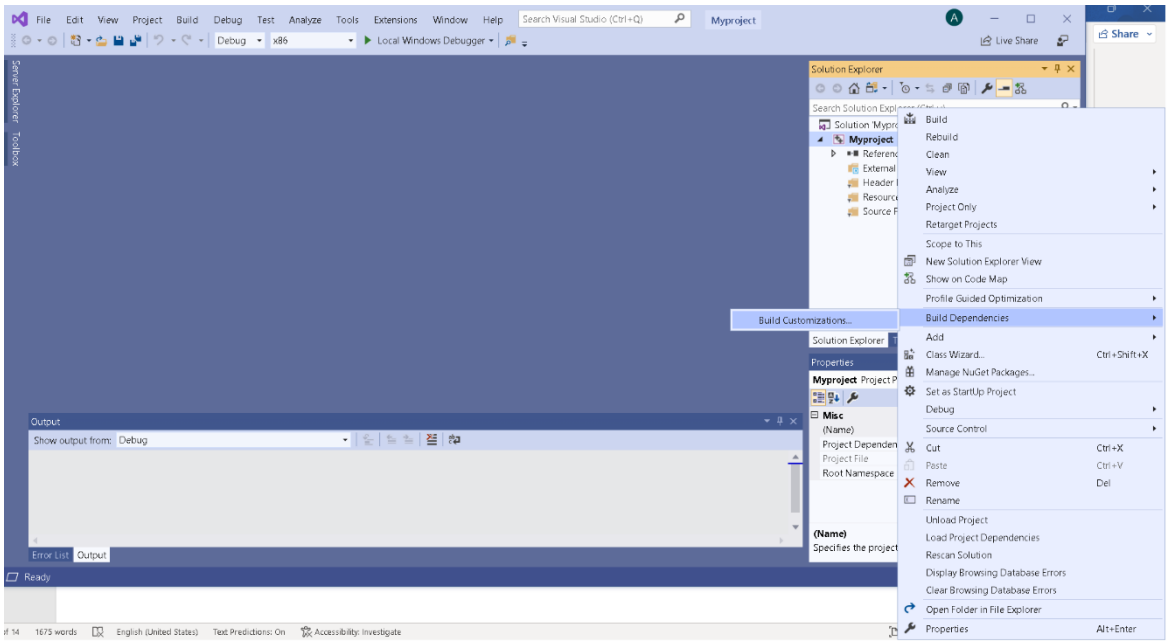
Step#4:

Write your "project name" (e.g., Myproject) and click on "Create".



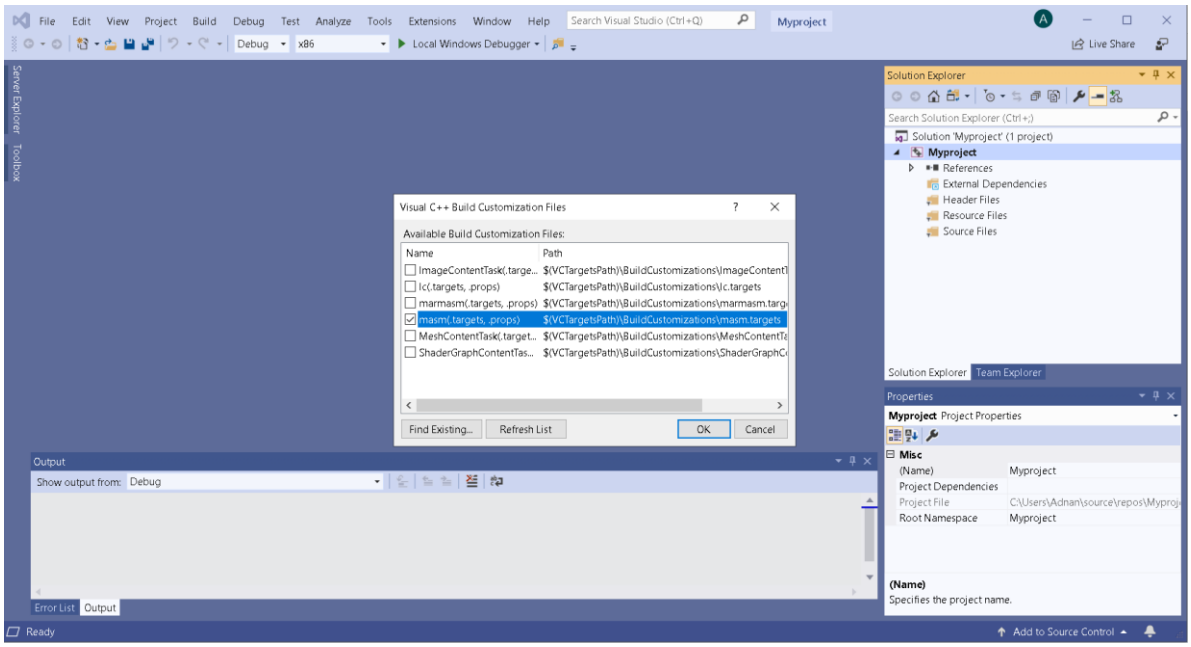
Step#5:

Right click on "project name" (i.e., Myproject) → Build Dependencies →Build Customizations



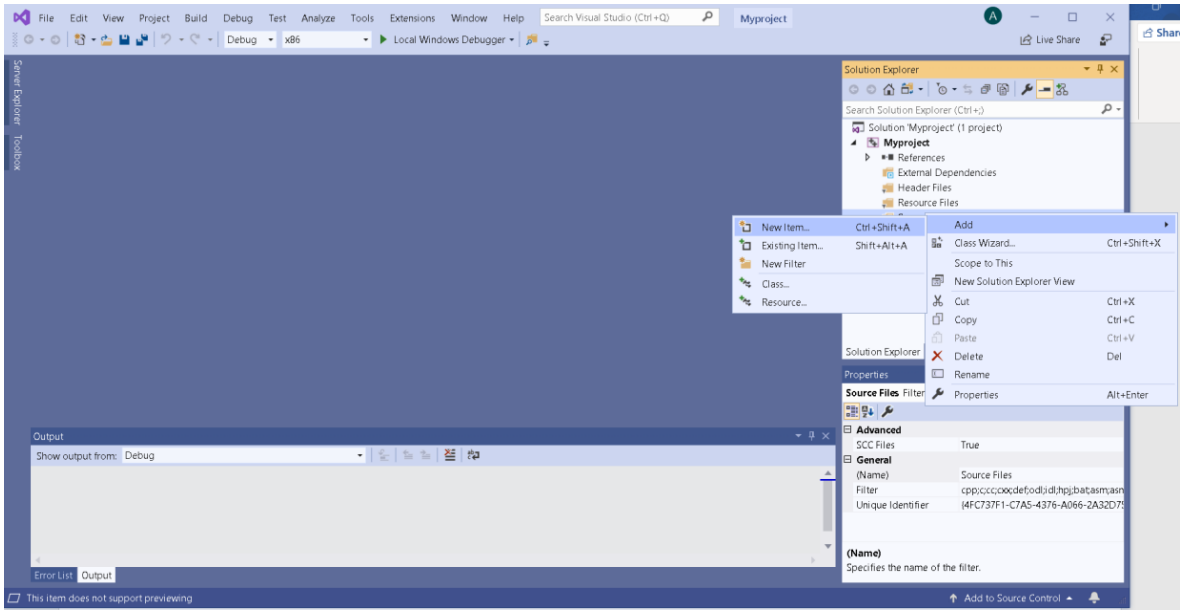
Step#6:

Select “masm” and press “OK”



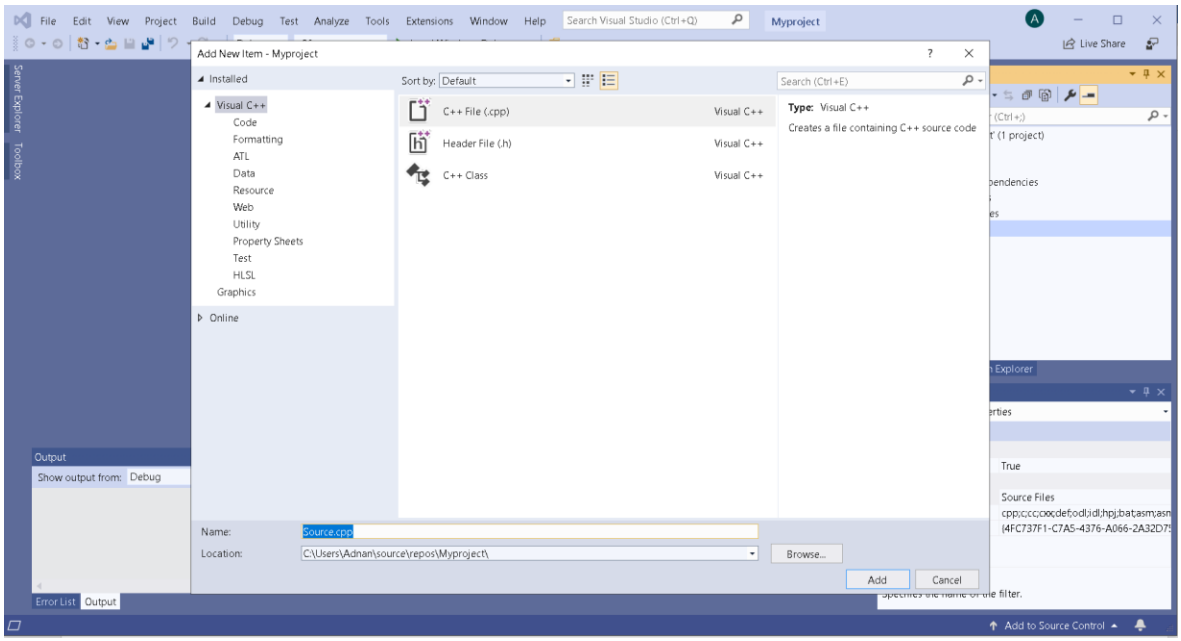
Step#7:

Right click on “Source Files” → Add → New Item



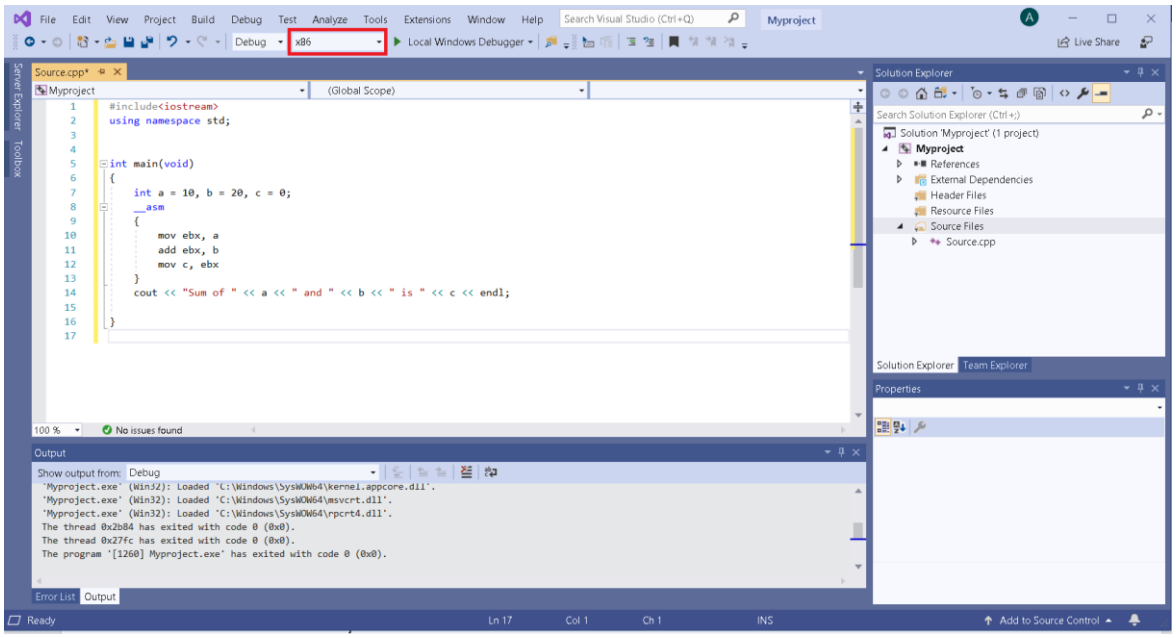
Step#8:

Click in “C++ File (.cpp), write a file name and click on “Add”.



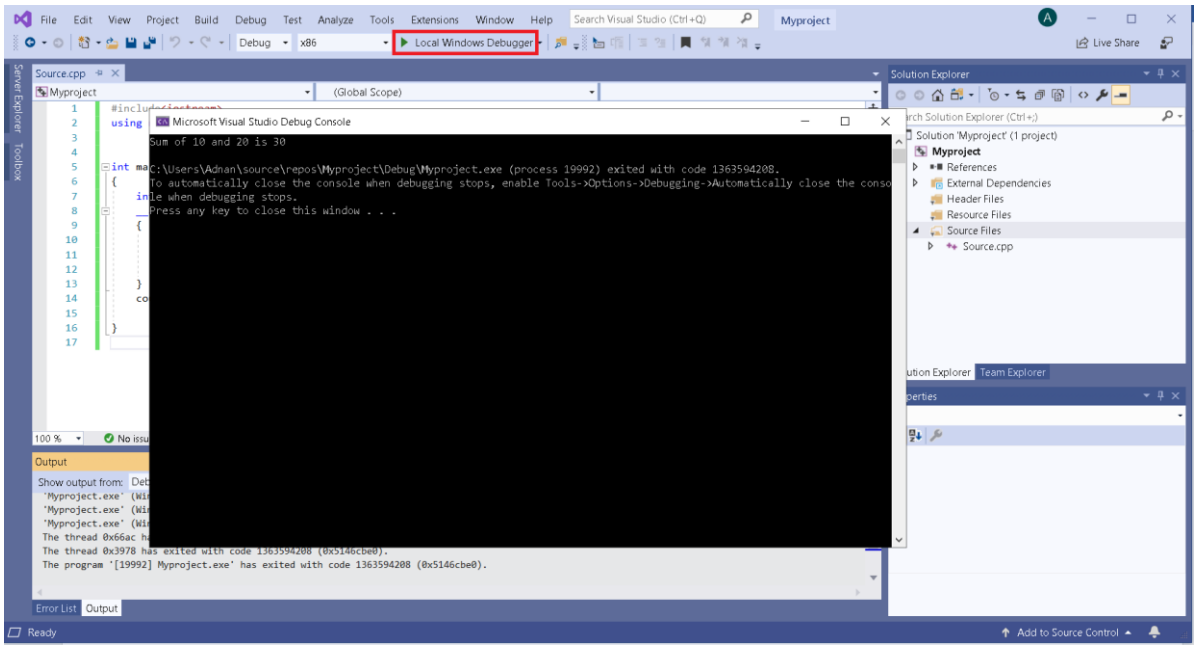
Step#9:

Paste code in Example#1 in the text window. Make sure that the environment is set to x86.



Step#10:

Click on “Local Windows Debugger” to run the code.

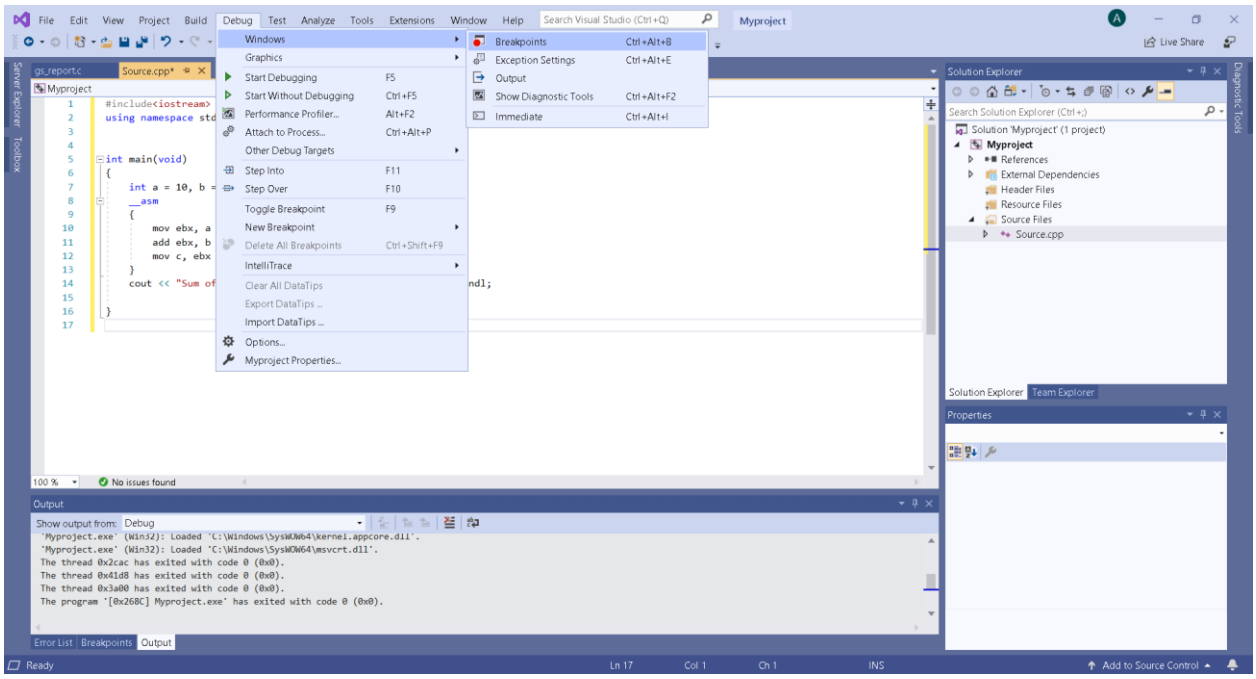


Computer Organization and Assembly Language

How to Debug a program

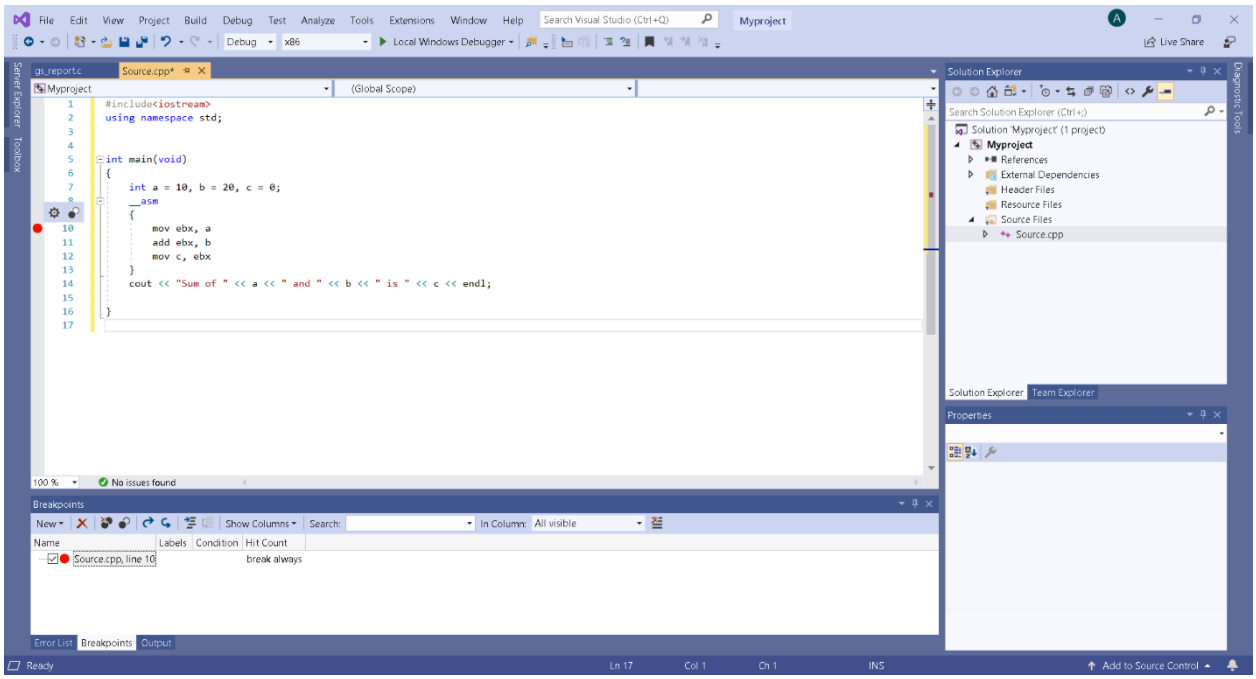
Step#1:

Click on Debug→Windows→Breakpoints



Step#2:

Click at the left side of any instruction to set breakpoints.

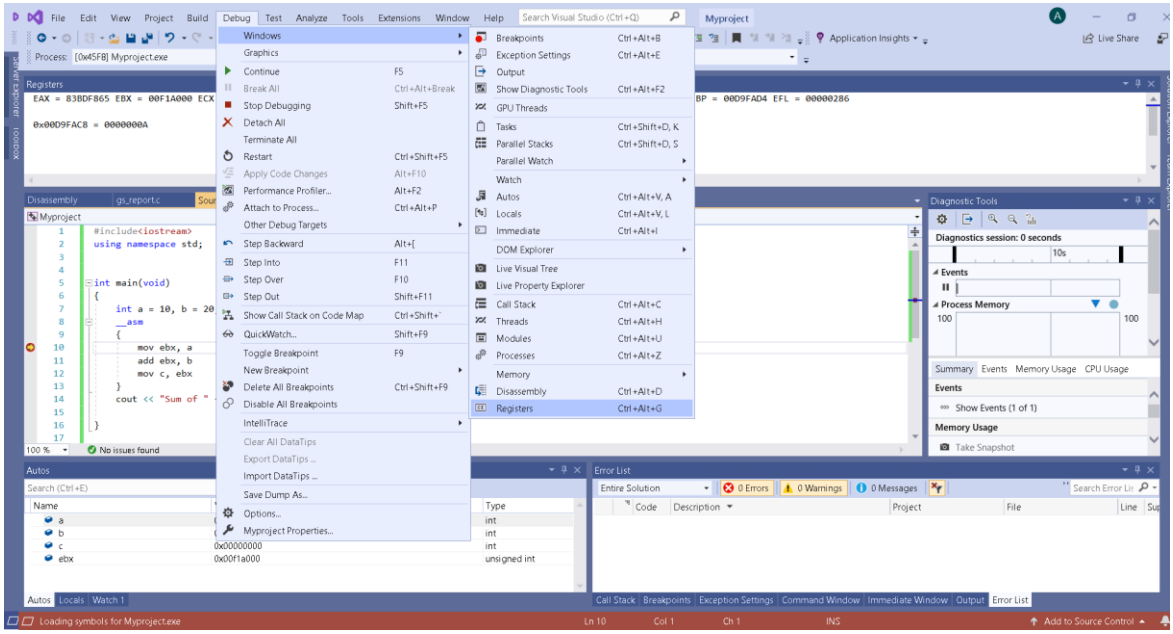


Step#3:

Click on “Local Windows Debugger”

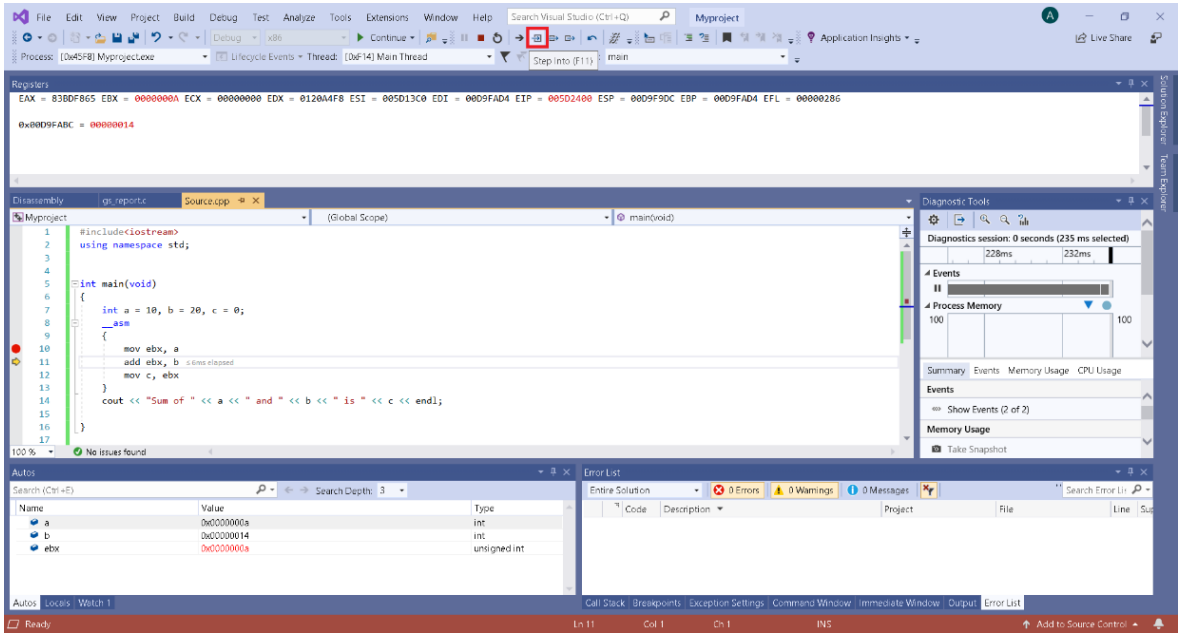
Step#4:

a) Click on Debug → Windows → Registers (to view status of registers)
b) Click on Debug → Windows → Disassembly (to view assembled code)
c) Click on Debug → Windows → Memory (to view memory)



Step#5:

Click on “Step Into” (or F11) to debug program.



Computer Organization and Assembly Language

Lab-10 Exercise

Task-1

Write a program in C++ that declares and initializes an integer array of 5 elements. The program then swaps the least and most significant bytes and the nibbles of the second byte of each element of the array using inline assembly language programming.

Task-2

Write a program in C++ that declares and initializes an integer array of 5 elements. The program then rotates the least significant byte (i.e., byte #0) and most significant byte (i.e., byte #3) to the left and byte #1 and byte #2 to the right of each element of the array using inline assembly language programming.

Task-3

Write an assembly language procedure that performs bitwise rotation on a given number. Then, integrate this procedure with a C++ program to verify its functionality.

void* rotate (void* number, int size, int direction);

- This function rotates the bits of the given number either to the left or right, based on the direction parameter.
- The number parameter is a pointer to the value that needs to be rotated.
- The size parameter specifies the size of the data type (e.g., 1 byte, 2 bytes, 4 bytes, or 8 bytes).
- The direction parameter determines the rotation direction:
    - 0 for left rotation.
    - 1 for right rotation.

The function should support various data types such as char, short, int, and long long by considering their sizes.

Computer Organization and Assembly Language

Lab 11: MMX Programming

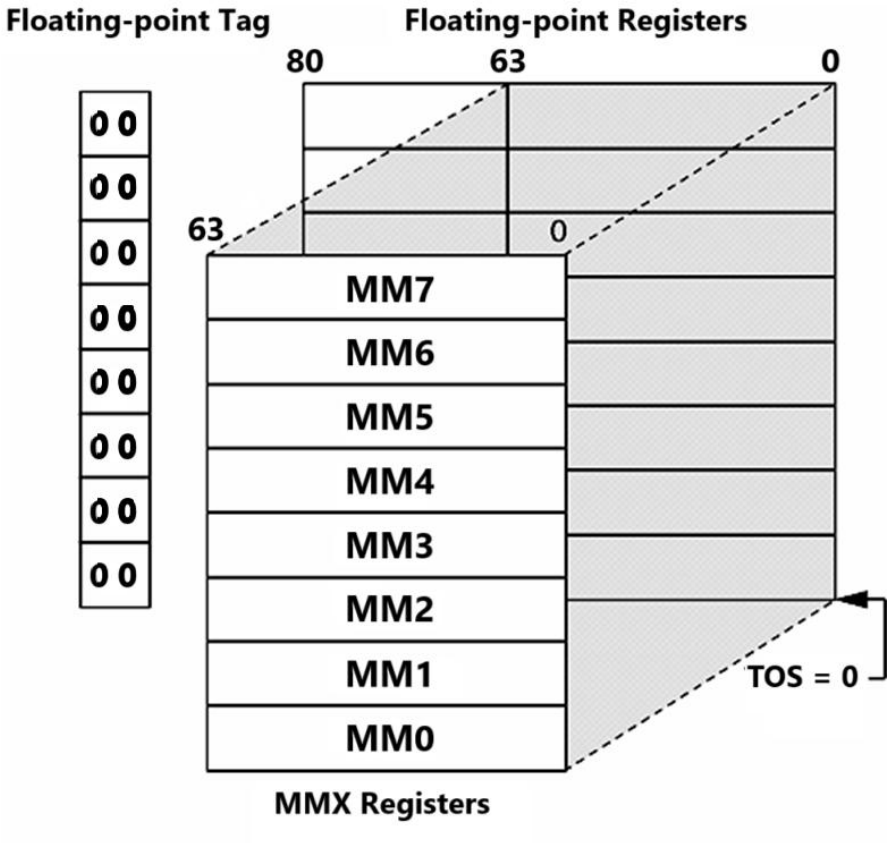
Learning outcome

- Students will gain an understanding of how parallel processing works and how it can be used to improve performance.
- Students will learn the different MMX instructions and how to use them to perform various operations.
- how to optimize performance of their code by using the MMX instruction set effectively.
- Students will learn how to use MMX in assembly language, and how to integrate it into their assembly language programs.

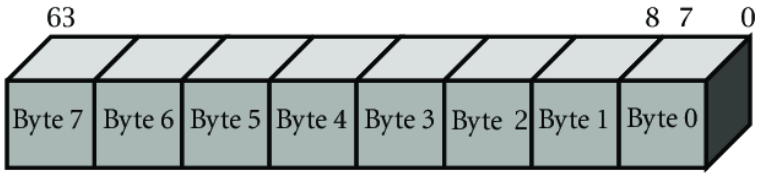
Multimedia Extension (MMX)

MMX refers to a feature that exists in the Intel processors, which can boost multimedia software, by executing integer instructions on several data parallelly. It allows Single Instruction to work on Multiple Data (SIMD). Though the processor supports the acceleration, it does not mean your application can make use of it automatically. Hence, your application must be manually written in order to take advantage of it.

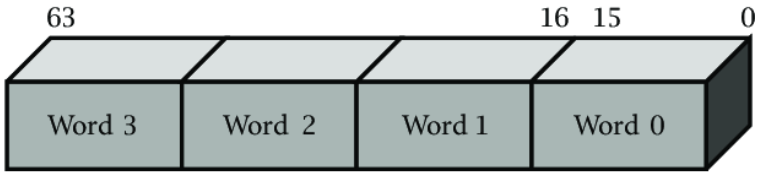
So far, we have seen general purpose registers. Besides that, we also have another 8 registers of 80-bit located in the FPU (Floating Point Unit) named as mm0, mm1, mm2, mm3, mm4, mm5, mm6 and mm7 as shown below. We can make use of these 8 registers in FPU if we have MMX feature in our processor. Only the least 64 bits of these registers are accessible for mmx operations.



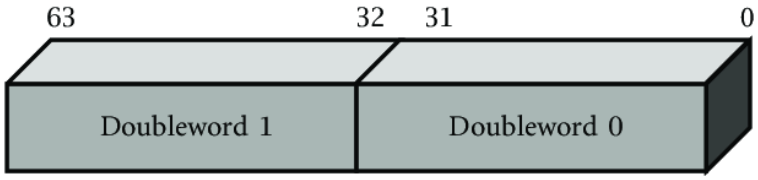
The MMX registers can store 8 bytes, 4 words, 2 double words and 1 quad word as shown in the image below.



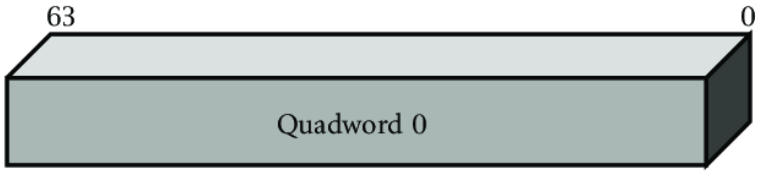
Packed bytes: 8 elements per operand



Packed words: 4 elements per operand

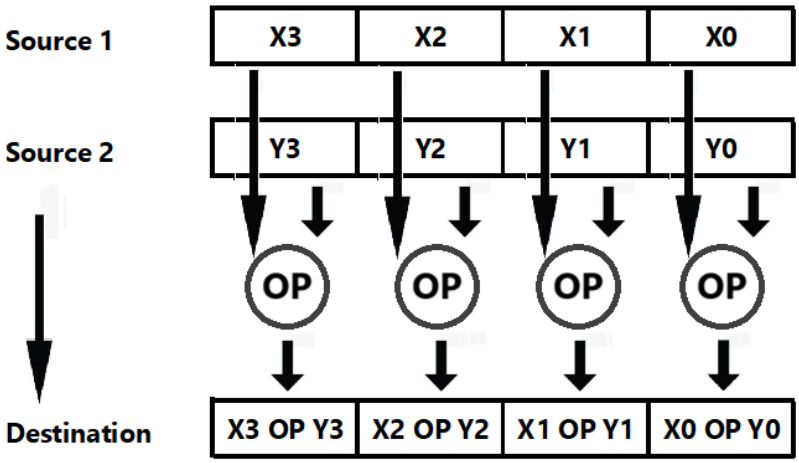


Packed doublewords (Dword): 2 elements per operand



Quadword (Qword): 1 element per operand

MMX instructions perform operations on the corresponding bytes, words or doublewords stored in registers as shown below.



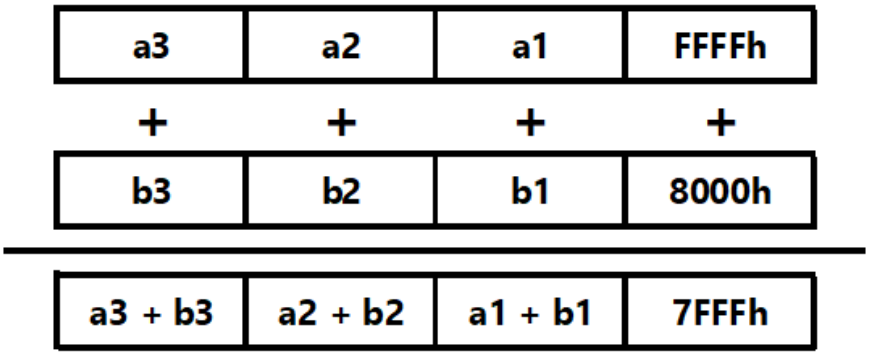
MMX™ Instruction Set Summary

The instructions and corresponding mnemonics in the table below are grouped by function categories. If an instruction supports multiple data types—byte (B), word (W), doubleword (DW), or quadword (QW), the datatypes are listed in brackets. Only one data type may be chosen for the given instruction. For example, the base mnemonic PADD (packed add) has the following variations: PADDB, PADDW, and PADDD. The number of opcodes associated with each base mnemonic is listed.

Category	Mnemonic	Number of Different OpCodes	Description
Arithmetic	PADD[B,W,D]	3	Add with wrap-around on [byte, word, doubleword]
	PADDS[B,W]	2	Add signed with saturation on [byte, word]
	PADDUS[B,W]	2	Add unsigned with saturation on [byte, word]
	PSUB[B,W,D]	3	Subtraction with wrap-around on [byte, word, doubleword]
	PSUBS[B,W]	2	Subtract signed with saturation on [byte, word]
	PSUBUS[B,W]	2	Subtract unsigned with saturation on [byte, word]
	PMULHW	1	Packed multiply high on words
	PMULLW	1	Packed multiply low on words
	PMADDWD	1	Packed multiply on words and add resulting pairs
Comparison	PCMPEQ[B,W,D]	3	Packed compare for equality [byte, word, doubleword]
	PCMPGT[B,W,D]	3	Packed compare greater than [byte, word, doubleword]
Conversion	PACKUSWB	1	Pack words into bytes (unsigned with saturation)
	PACKSS[WB,DW]	2	Pack [words into bytes, doublewords into words] (signed with saturation)
	PUNPCKH[BW,WD.DQ]	3	Unpack (interleave) high order [bytes, words, doublewords] from MMX register
	PUNPCKL[BW,WD.DQ]	3	Unpack (interleave) low order [bytes, words, doublewords) from MMX register
Logical	PAND	1	Bitwise AND
	PANDN	1	Bitwise AND NOT
	POR	1	Bitwise OR
	PXOR	1	Bitwise XOR
Shift	PSLL[W,D,Q]	6	Packed shift left logical [word, doubleword, quadword] by amount specified in MMX register or by immediate value
	PSRL[W,D,Q]	6	Packed shift right logical [word, doubleword, quadword] by amount specified in MMX register or by immediate value
	PSRA[W,D,Q]	4	Packed shift right arithmetic [word, doubleword] by amount specified in MMX register or by immediate value
Data Transfer	MOV[D,Q]	4	Move [doubleword, quadword] to MMX register or from MMX register
FP & MMX State Mgmt	EMMS	1	Empty MMX state

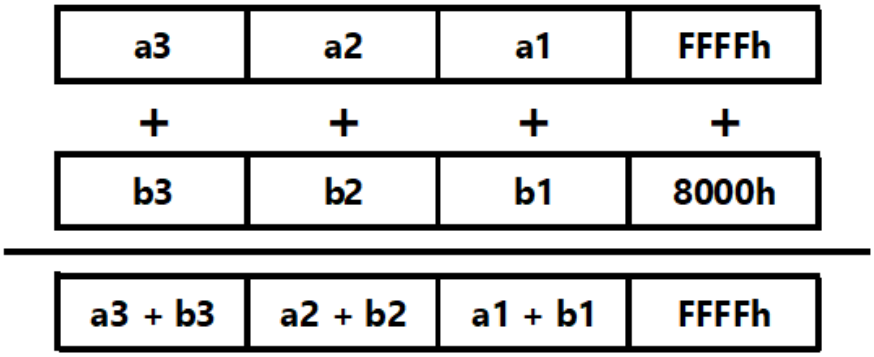
Instruction Examples

The following section will briefly describe five examples of MMX instructions. For illustration, the data type shown in this section will be the 16-bit word data type; most of these operations also exist for 8-bit or 32-bit packed data types. The following example shows a packed add word with wrap around. It performs four additions of the eight, 16-bit elements, with each addition independent of the others and in parallel. In this case, the right-most result exceeds the maximum value representable in 16-bits—thus it wraparound. This is the way regular IA arithmetic behaves. FFFFh + 8000h would be a 17-bit result. The 17th bit is lost because of wrap around, so the result is 7FFFh.



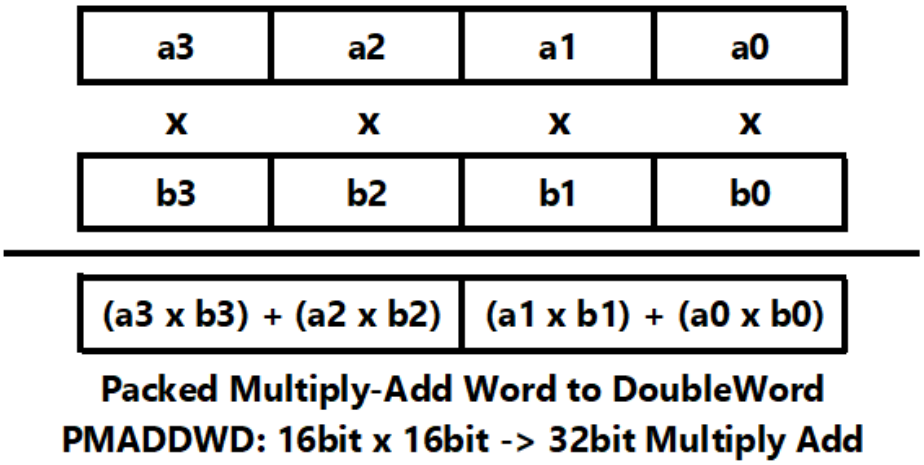
PADD[W]: Wrap-around Add

The following example is for a packed add word with unsigned saturation. This example uses the same data values from before. The right-most addition generates a result that does not fit into 16 bits; consequently, in this case saturation occurs. Saturation means that if addition results in overflow or subtraction results in underflow, the result is clamped to the largest or the smallest value representable. For an unsigned, 16-bit word, the largest and the smallest representable values are FFFFh and 0x0000; for a signed word the largest and the smallest representable values are 7FFFh and 0x8000.



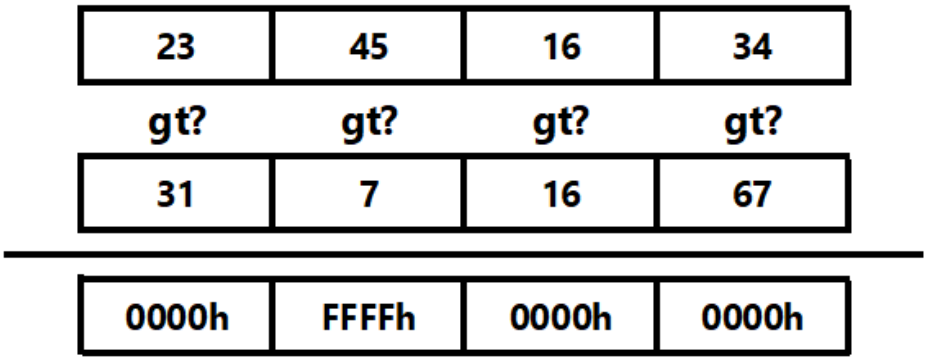
PADDUS[W]: Saturating Arithmetic

The PMADD instruction starts with a 16-bit, packed data type and generates a 32-bit packed, data type result. It multiplies all the corresponding elements, generating four 32-bit results and adds the two products on the left together for one result and the two products on the right together for the other result. To complete a multiply-accumulate operation, the results would then be added to another register which is used as the accumulator.



Packed Multiply-Add Word to DoubleWord
PMADDWD: 16bit x 16bit -> 32bit Multiply Add

The following example is a packed parallel signed compare. This example compares four pairs of 16-bit words. It creates a result of true (FFFFh), or false (0000h). This result is a packed mask of ones for each true condition, or zeros for each false condition. The following example shows an example of a compare "greater than" on packed word data.



PCMPGT[W]: Parallel Compares

EMMS instruction

The EMMS (Empty Multimedia State) instruction is used to clear the multimedia state after performing MMX (Multimedia Extension) operations. It is used to ensure that the MMX state is not accidentally used in subsequent floating-point operations.

In short, the EMMS instruction is used to reset the multimedia state of the processor, clearing any MMX registers and freeing them up for use by other instructions. This is important because MMX registers are different from the regular registers used by the processor and using them incorrectly can cause unexpected results or errors.

The EMMS instruction is typically used at the end of a block of code that uses MMX instructions to ensure that the processor's multimedia state is in a known state before continuing to execute other instructions.

Example#1:
Program to add 2 arrays using MMX instructions.

```
#include<iostream>
using namespace std;
int main()
{
   const int size = 8;
   char array1[size] = { 127,2,3,4,5,6,7,8 };
   char array2[size] = { 1,7,8,9,10,11,12,13 };
   char array3[size];

   __asm
   {
      movq mm0, [array1]
      // move 8 bytes from address pointed by array1 to mm0 register
      movq mm1, [array2]
      // move 8 bytes from address pointed by array2 to mm1 register

      paddb mm0, mm1
      // add corresponding bytes of mm0 and mm1 and store result to mm0
      movq[array3], mm0
      // mov 8 bytes from mm0 to address pointed by array3

      emms  // clearing multimedia state
   }
   for (int i = 0; i < size; i++)
   {
      cout << "Sum of " << (int)array1[i] << " and " << (int)array2[i]
   << " is " << (int)array3[i] << endl;
   }
   return 0;
}
```

Example#2:
Program to shift left all elements of int array using MMX instructions.

```
#include<iostream>
using namespace std;
int main()
{
    const int size = 6;
    int array1[size] = { 1,2,3,4,5,6 };
    int array2[size];
    for (int i = 0; i < size * 4; i = i + 8)
    __asm
    {
        mov esi,i       ; mov i to esi register
        movq mm0,[array1+esi]
        ; moving 8 bytes from address pointed by array1+esi
        pslld mm0,1     ; Shifting left every double word in mm0
    register
        movq  [array2+esi],mm0
        ; moving contents of mm0 to memory pointed by array2+esi
        emms            ; clearing mmx state
    }
    for (int i = 0; i < size; i++)
    {
        cout << "Sum of " << (int)array2[i] << endl;
    }
    return 0;
}
```

Example#3:
Program that sums all numbers that are greater than 100 using mmx instructions.

```
#include<iostream>
using namespace std;
int main()
{
    const int size = 8;
    char array1[size] = {25,50,80,110,125,127,30,90};
    char array2[size] = {100,100,100,100,100,100,100,100};
    char array3[size];
    int sum = 0;
    __asm
    {
      movq mm0, [array1]     // moving 8 bytes from array1 to mm0
      movq mm1, [array2]     // moving 8 bytes from array2 to mm1
      pcmpgtb mm0, mm1
      /*/ 0xFF will replace bytes in mm0 that are greater than the
      corresponding bytes in mm1 and 0x00 will replace bytes in mm0 that
      are less than the corresponding bytes in mm1 /*/

      pand mm0,[array1]
      /*/  After bitwise and operation, mm0 will contain bytes that will
      be greater than 100 /*/

      movq [array3], mm0      // moving mm0 to array3
      emms
     }
     for (int i = 0; i < size; i++)
      sum += array3[i];
     cout << "Sum of numbers greater than 100 is: "<< sum << endl;

    return 0;
}
```

Example#4:
Program to multiply 4 words using MMX instructions. This example keeps the lower 16 bits of the product, discarding the higher 16 bits.

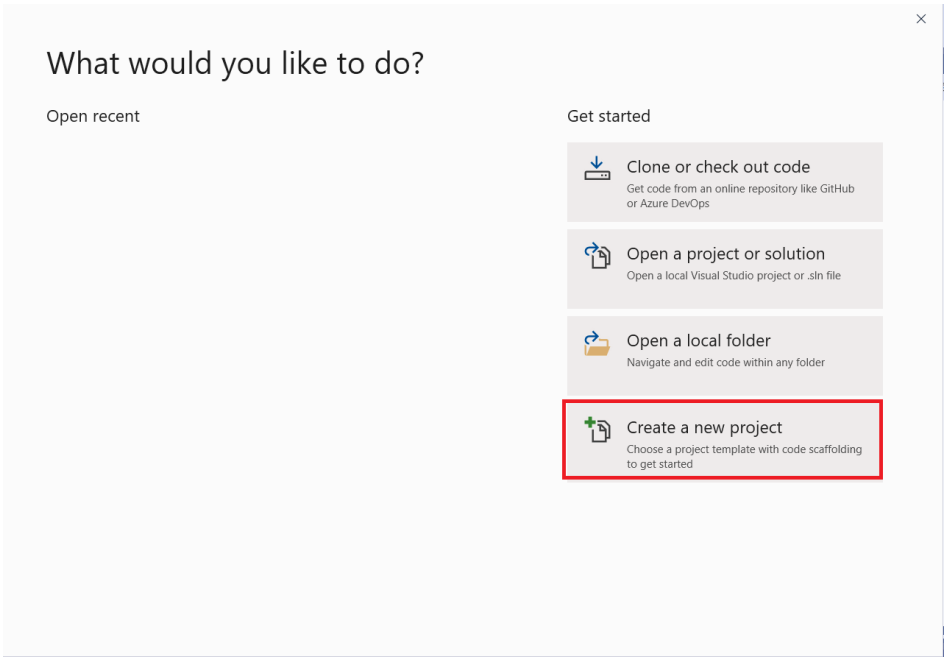
```
#include<iostream>
using namespace std;
int main()
{
    const int size = 4;
    short int array1[] = {2,3,4,5};
    short int array2[] = {1,2,3,4};
    short int array3[size];

    __asm
    {
        movq mm0,[array1] // moving 8 bytes from array1 to mm0
        movq mm1,[array2] // moving 8 bytes from array2 to mm1
        pmullw mm0,mm1
        /*/ multiply corresponding word of mm0 and mm1 and store
        the lower word to mm0/*/

        movq [array3],mm0 //moving product to array3
        emms              // clearing multimedia state
    }
    for (int i = 0; i < size; i++)
    {
        cout << "Sum of " << (int)array1[i] << " and " <<
        (int)array2[i] << " is " << (int)array3[i] << endl;
    }
    return 0;
}
```

Step#1:

Open Visual Studio 2019

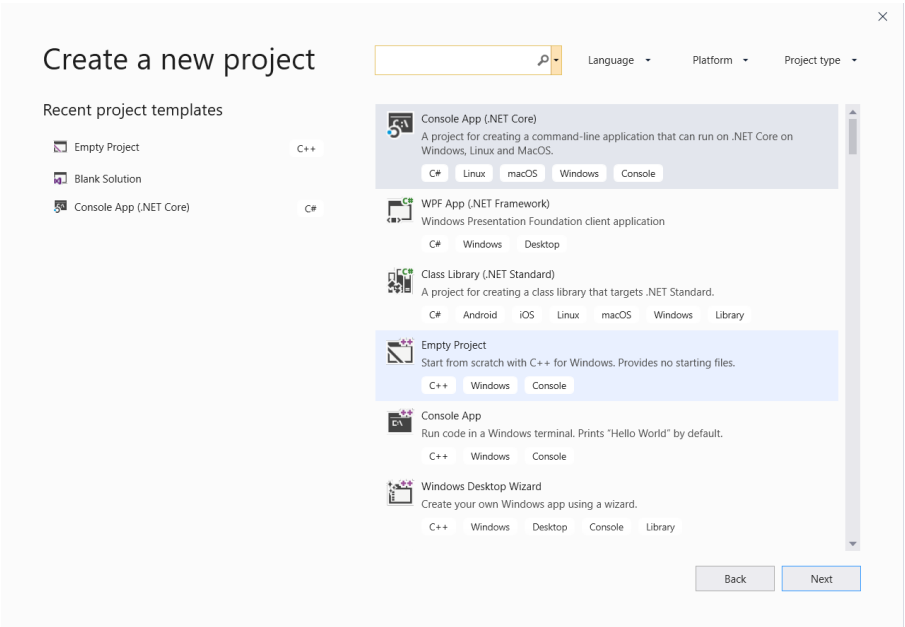
Step#2:

Click on “Create a new project”



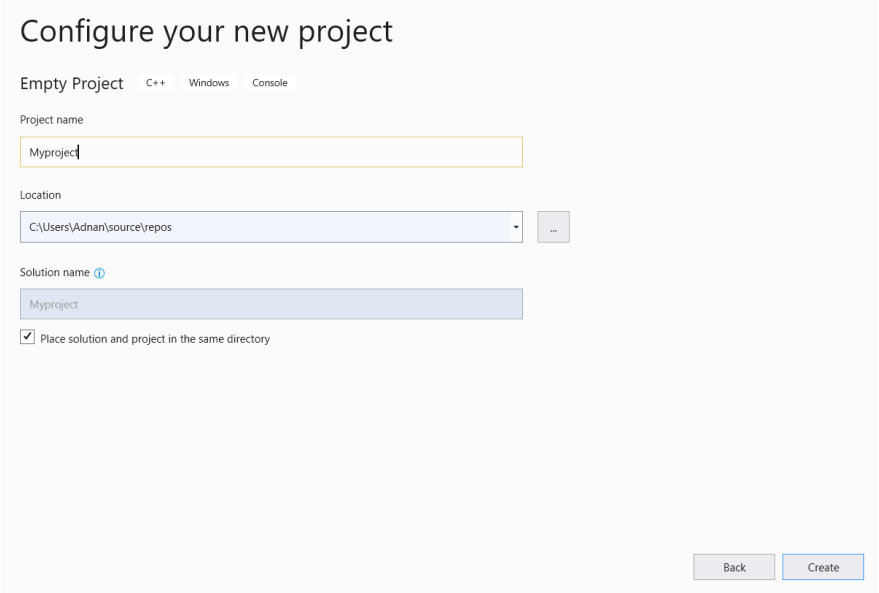
Step#3:

Click on C++ under “Empty Project” and press “Next”



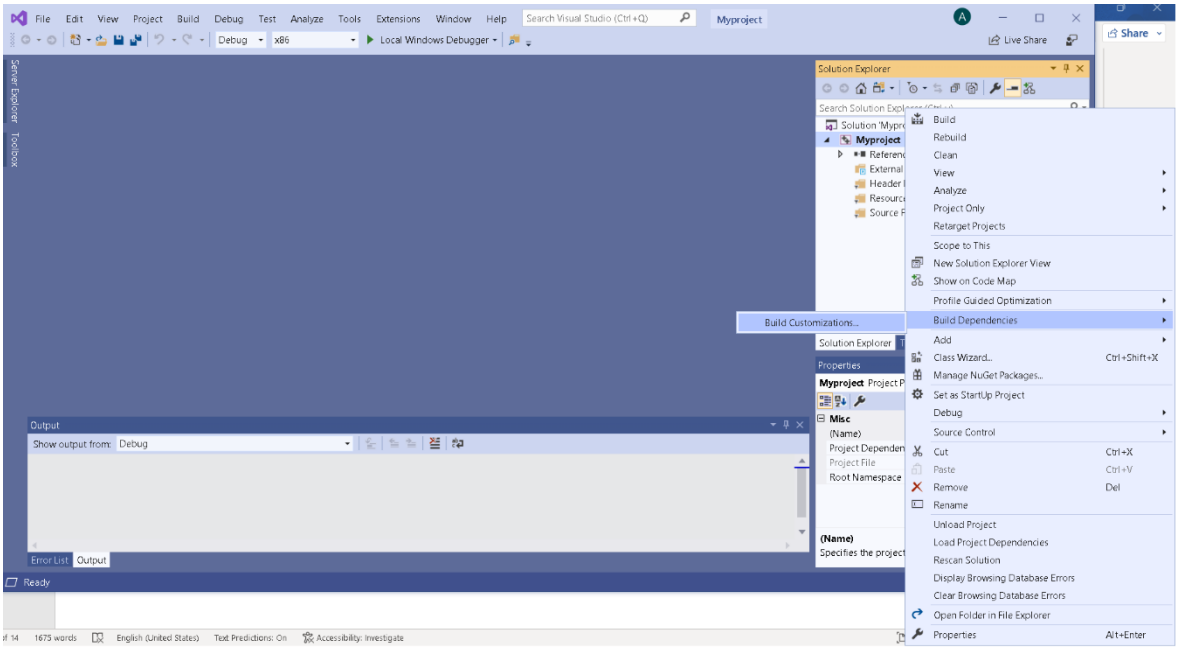
Step#4:

Write your project name (e.g., Myproject) and click on “Create”



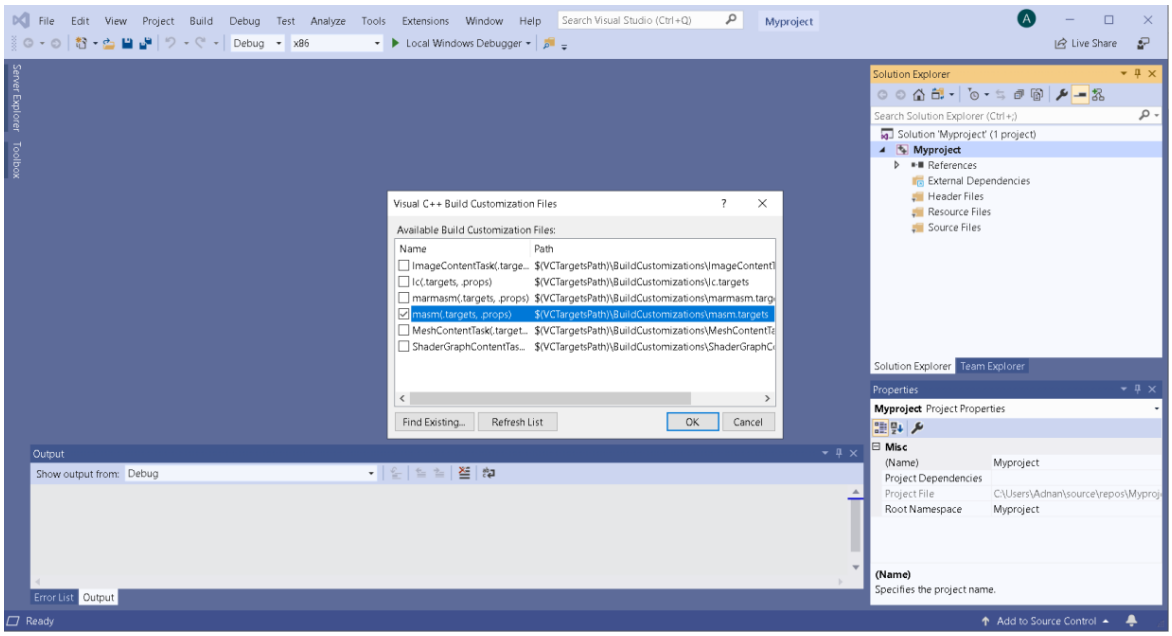
Step#5:

Right click on project name (i.e., Myproject) → Build Dependencies →Build Customizations



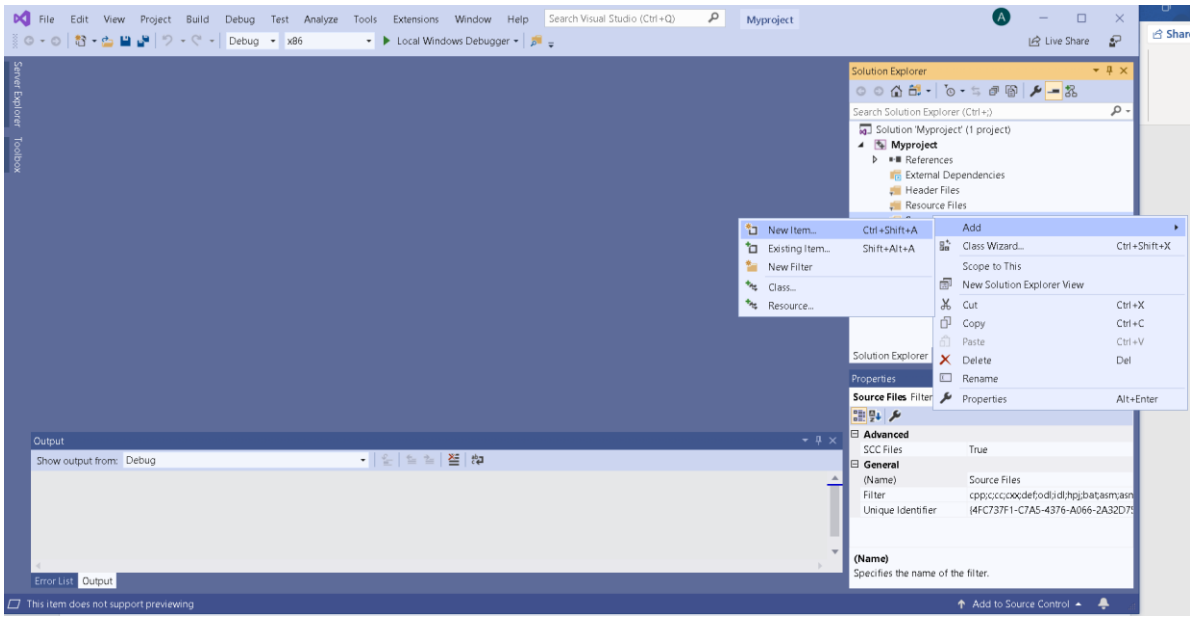
Step#6:

Select “masm” and press “OK”



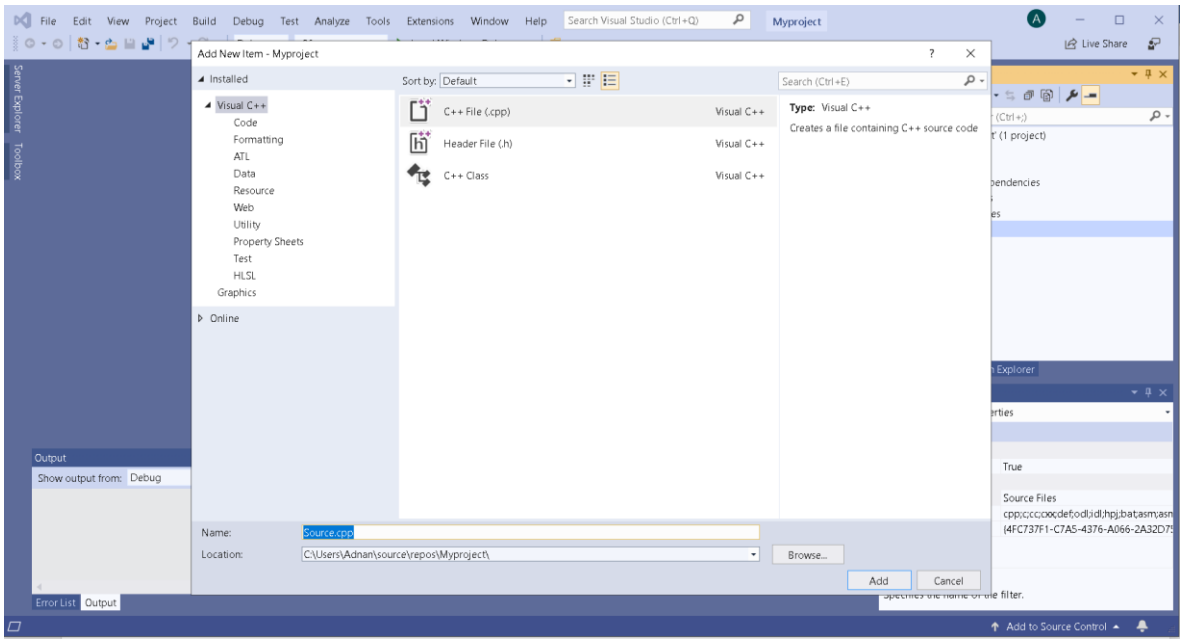
Step#7:

Right click on “Source Files” → Add → New Item



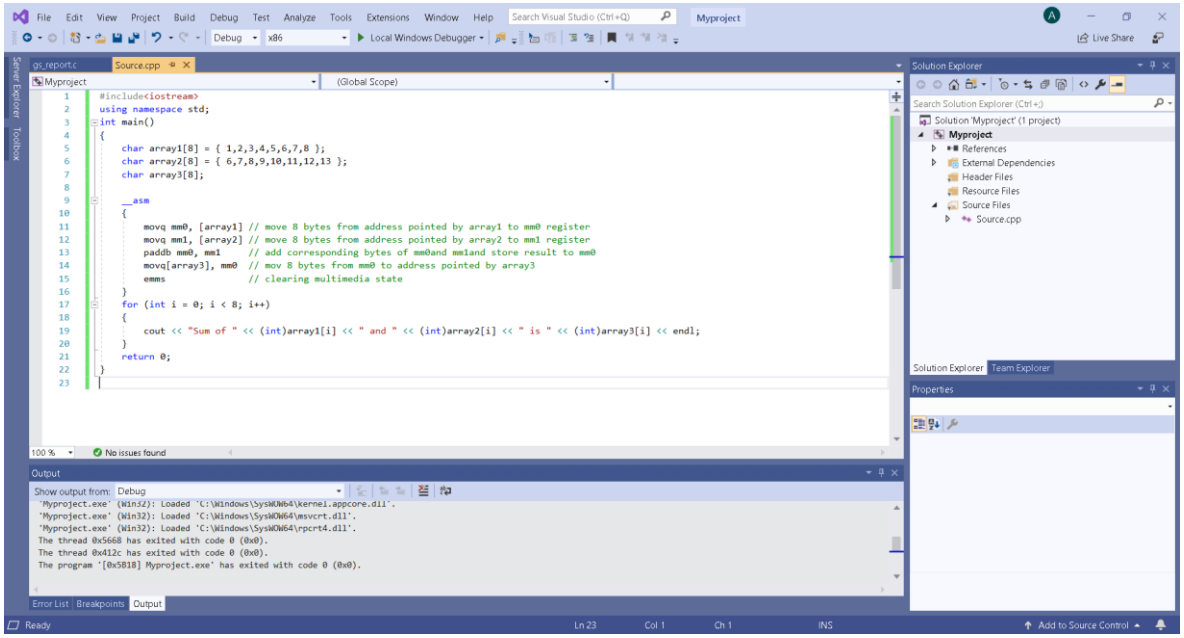
Step#8:

Click in "**C++ File (.cpp)** , write a file name and click on "Add".



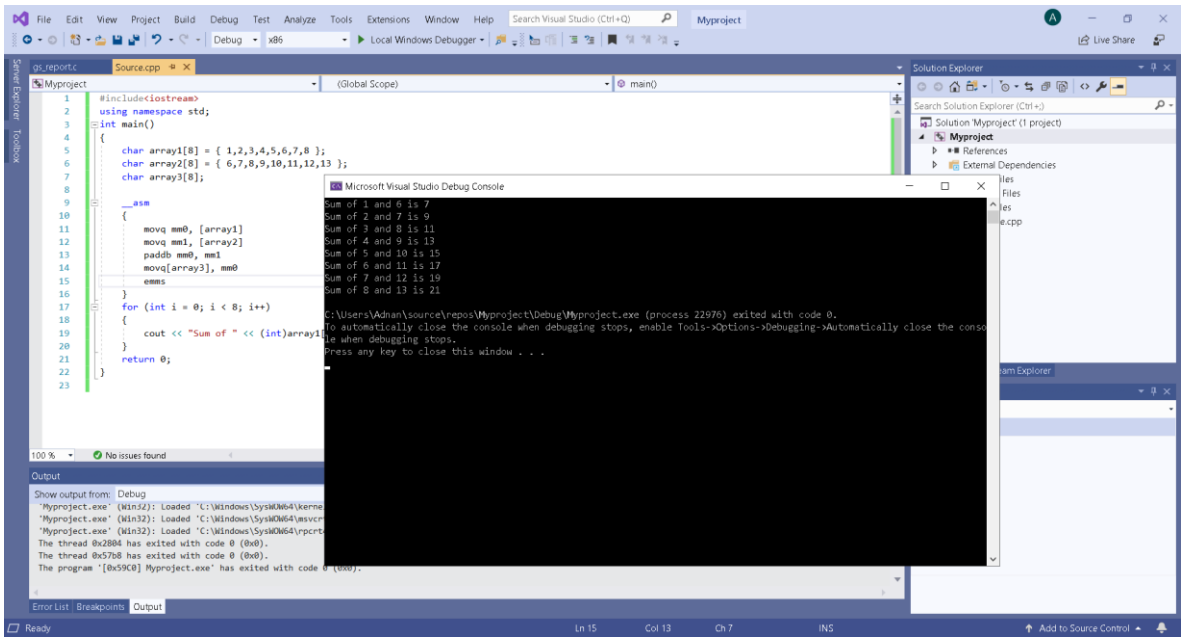
Step#9:

Paste code in **Example#1** in the text window. Make sure that the environment is set to x86.



Step#10:

Click on “Local Windows Debugger” to run the code.



Computer Organization and Assembly Language

Lab-11 Exercise

Task-1

Write a program in C++ that declares two byte-type arrays of 80 elements each and initialize them with some data. The program then adds the corresponding elements of these two arrays and stores them in a third array using MMX instructions.

Task-2

Write a program in C++ that declares a byte array of 80 elements and initialize it with some data. The program then calculates the sum of even and odd elements using MMX instructions and prints them.

Task-3

Write an assembly language procedure that computes the sum of all elements in an array using **MMX (MultiMedia eXtensions) instructions**. The procedure should receive the array, its size, and the size of each element as arguments.

void sum_array(void* array, int array_size, int element_size);*

The function takes an array of numeric values and returns the sum of all elements.

It utilizes **MMX instructions** for SIMD (Single Instruction, Multiple Data) parallel processing to improve performance.

The function parameters are:

- array: A pointer to the array.
- array_size: The total number of elements in the array.
- element_size: The size of each element (e.g., 1 byte, 2 bytes, 4 bytes, or 8 bytes).
- The function should support different data types such as:
  - **8-bit (char)**
  - **16-bit (short)**
  - **32-bit (int)**
  - **64-bit (long long)**
- The sum should be accumulated efficiently using MMX registers and stored in a variable to be returned.