Lab 12

MMX Programming

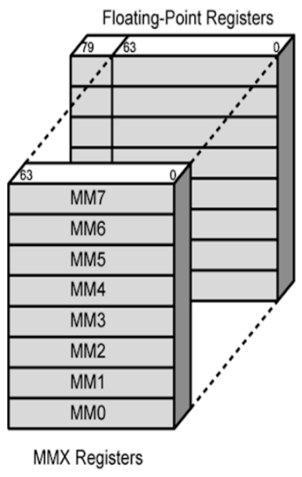
Learning outcome

- 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.

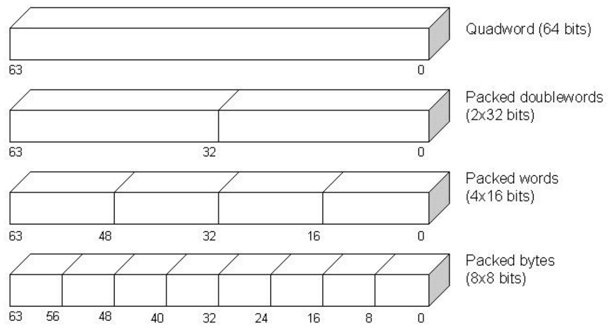
Multimedia Extension (MMX)

MMX refers to a feature that exist in the Intel processors, which can boost up 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.

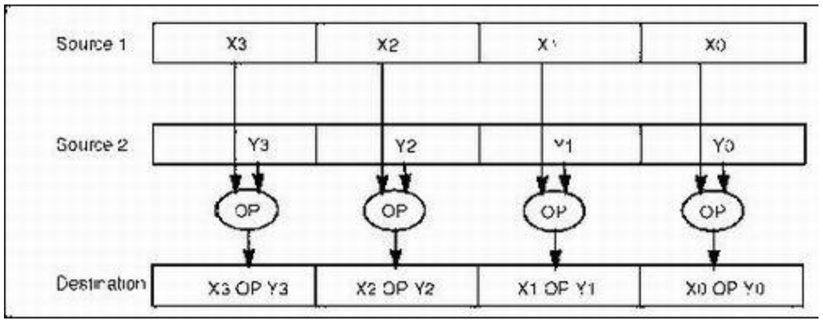
So far, we have seen general purpose registers. Besides that, we also have another 8 80-bit registers located in the FPU (Floating Point Unit) named as mm0, mm1, mm2, mm3, mm4, mm5, mm6 and mm7 as shown below. We can make use of these 8 registers in FPU if we have MMX feature in our processor. Only least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



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



MMX™ Instruction Set Summary

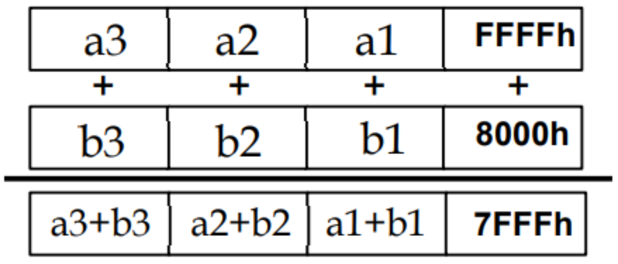
The instructions and corresponding mnemonics in the table below are grouped by function categories. If an instruction supports multiple data types—byte (B), word (W), doubleword (DW), or quadword (QW), the datatypes are listed in brackets. Only one data type may be chosen for a given instruction. For 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]	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

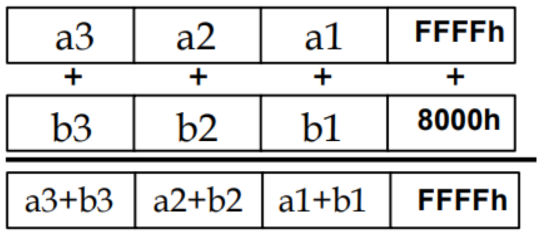
Instruction Examples

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 wrapsaround. 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.



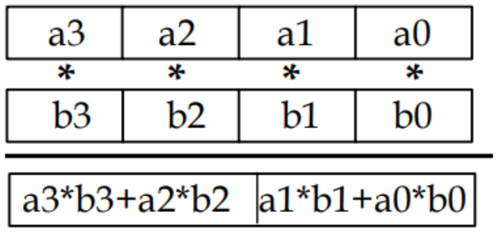
PADD[W]: Wrap-around Add

The following example is for a packed add word with unsigned saturation. This example uses the same data values from before. The right-most add 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.



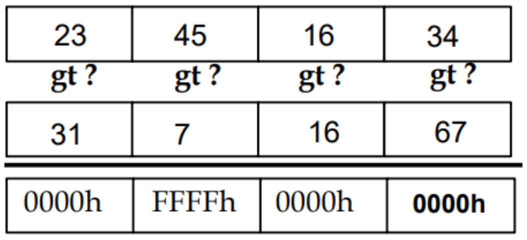
PADDUS[W]: Saturating Arithmetic

The PMADD instruction starts from 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.



PMADDWD: 16b x 16b -> 32b 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()
{
        char array1[8] = { 1,2,3,4,5,6,7,8 };
        char array2[8] = { 6,7,8,9,10,11,12,13 };
        char array3[8];

        __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 < 8; i++)
        {
                cout <<"Sum of "<<(uint8)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()
{
            int array1[6] = { 1,2,3,4,5,6 };
            int array2[6];
           for(int i=0;i<24;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 every double word in mm0 register left
                       movq  [array2+esi],mm0             ; moving contents of mm0 to memory pointed by array2+esi
                       emms                               ; clearing mmx state
           }
           for (int i = 0; i < 6; 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()
{
        char array1[8] = { 25,50,80,110,125,127,30,90 };
        char array2[8] = { 100,100,100,100,100,100,100,100 };
        char array3[8];
        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 correspoding bytes in mm1 and 0x0 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 < 8; i++)
              sum += array3[i];
       cout << "Sum of numbers greater than 100 is: "<< sum << endl;


       return 0;
}
```

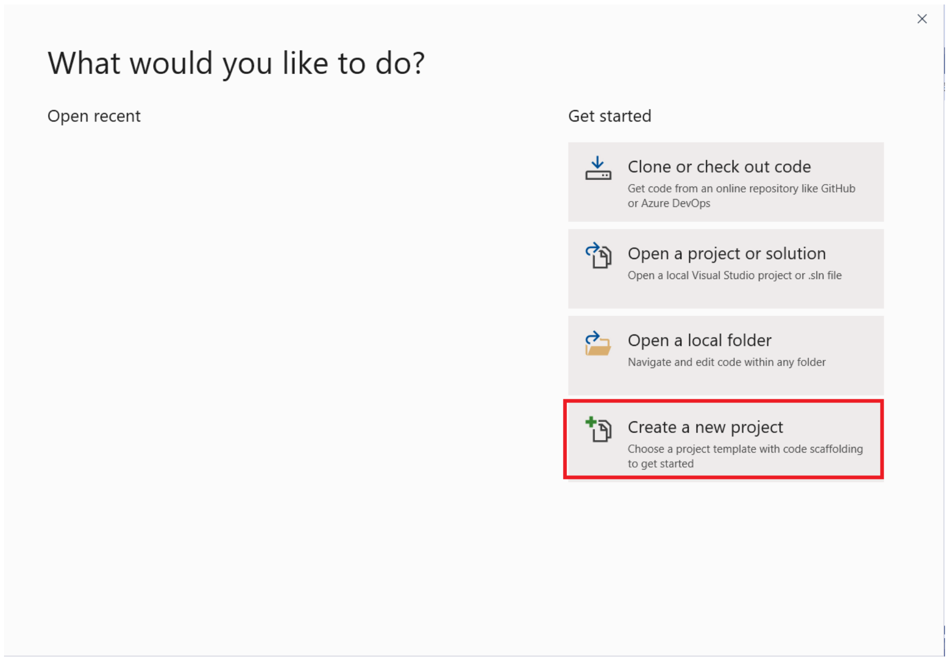
Example#4: Program to multiply 4 words using MMX instructions. This example keeps the lower 16 bits of the product, discarding the higher 16 bits.

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

       __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 < 4; 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”



What would you like to do?

Open recent

Get started

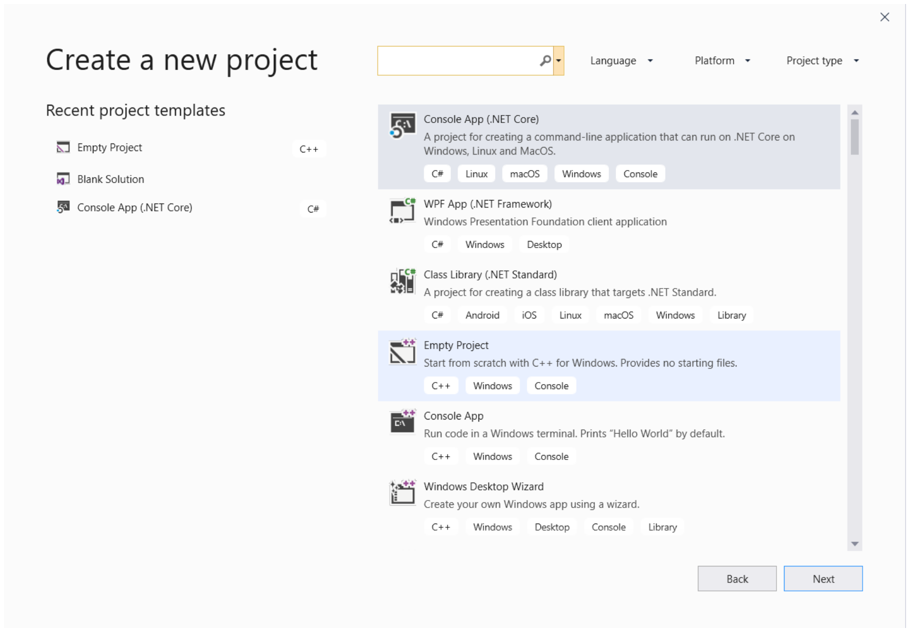
Clone or check out code
Get code from an online repository like GitHub or Azure DevOps

Open a project or solution
Open a local Visual Studio project or .sln file

Open a local folder
Navigate and edit code within any folder

Create a new project
Choose a project template with code scaffolding to get started

Step#3: Click on C++ under “Empty Project” and press Next



Create a new project

Recent project templates

Empty Project	C++
Blank Solution	
Console App (.NET Core)	C#

Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
C# Linux macOS Windows Console

WPF App (.NET Framework)
Windows Presentation Foundation client application
C# Windows Desktop

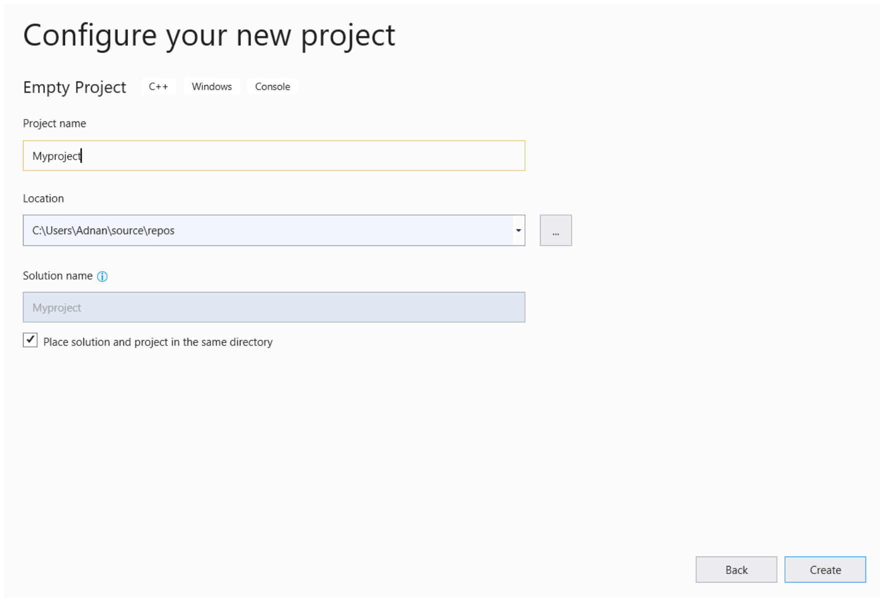
Class Library (.NET Standard)
A project for creating a class library that targets .NET Standard.
C# Android iOS Linux macOS Windows Library

Empty Project
Start from scratch with C++ for Windows. Provides no starting files.
C++ Windows Console

Console App
Run code in a Windows terminal. Prints "Hello World" by default.
C++ Windows Console

Windows Desktop Wizard
Create your own Windows app using a wizard.
C++ Windows Desktop Console Library

Step#4: Write your project name (e.g., Myproject) and click on Create



Configure your new project

Empty Project C++ Windows Console

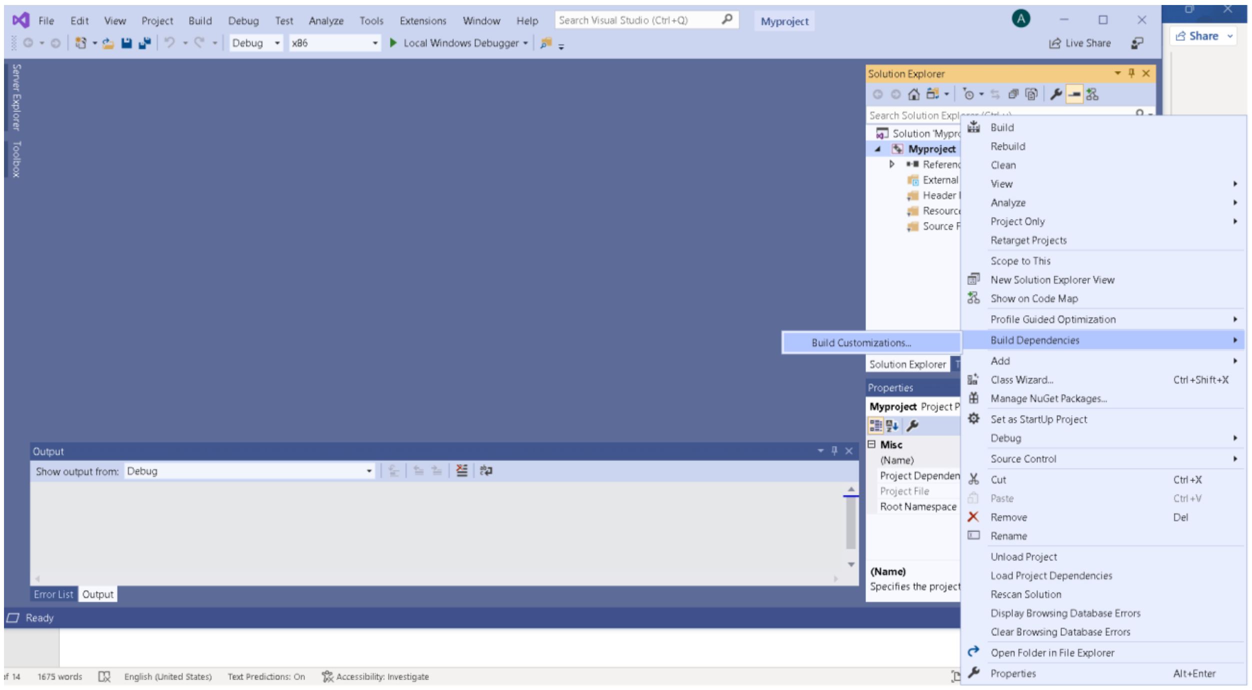
Project name: Myproject

Location: C:\Users\Adnan\source\repos

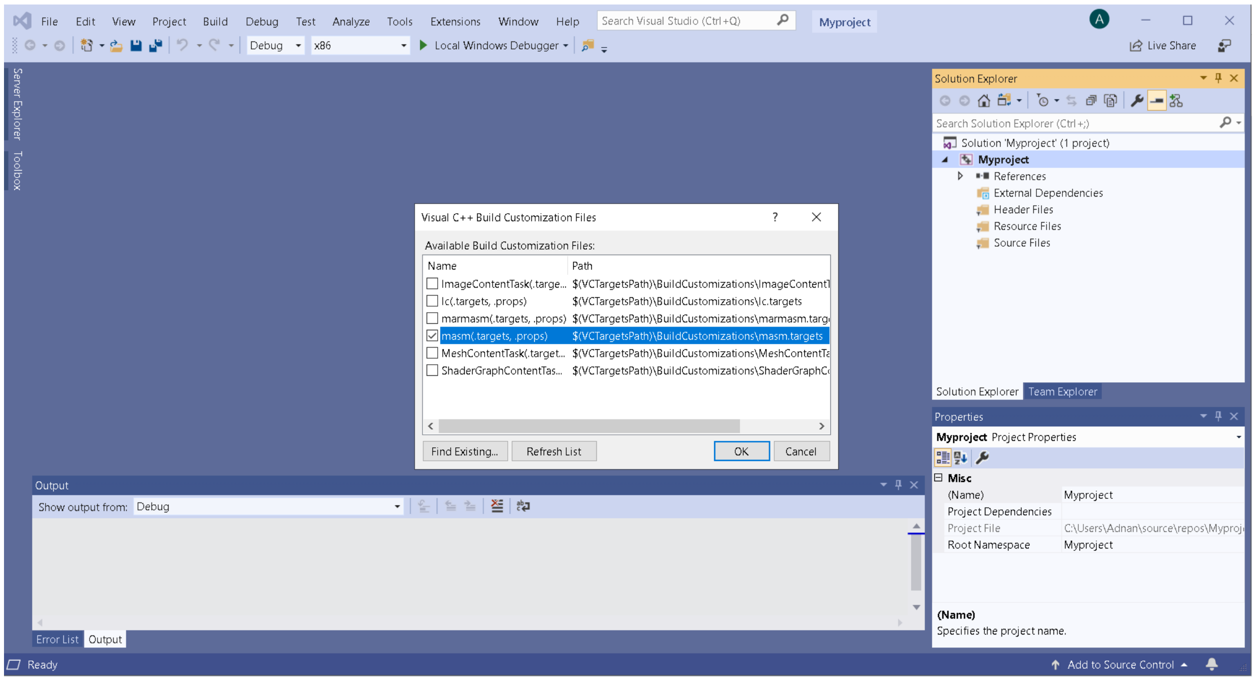
Solution name: Myproject

☑ Place solution and project in the same directory

Step#5: Right click on project name (i.e., Myproject) → Build Dependencies →Build Customizations



Step#5: Select “masm” and press OK

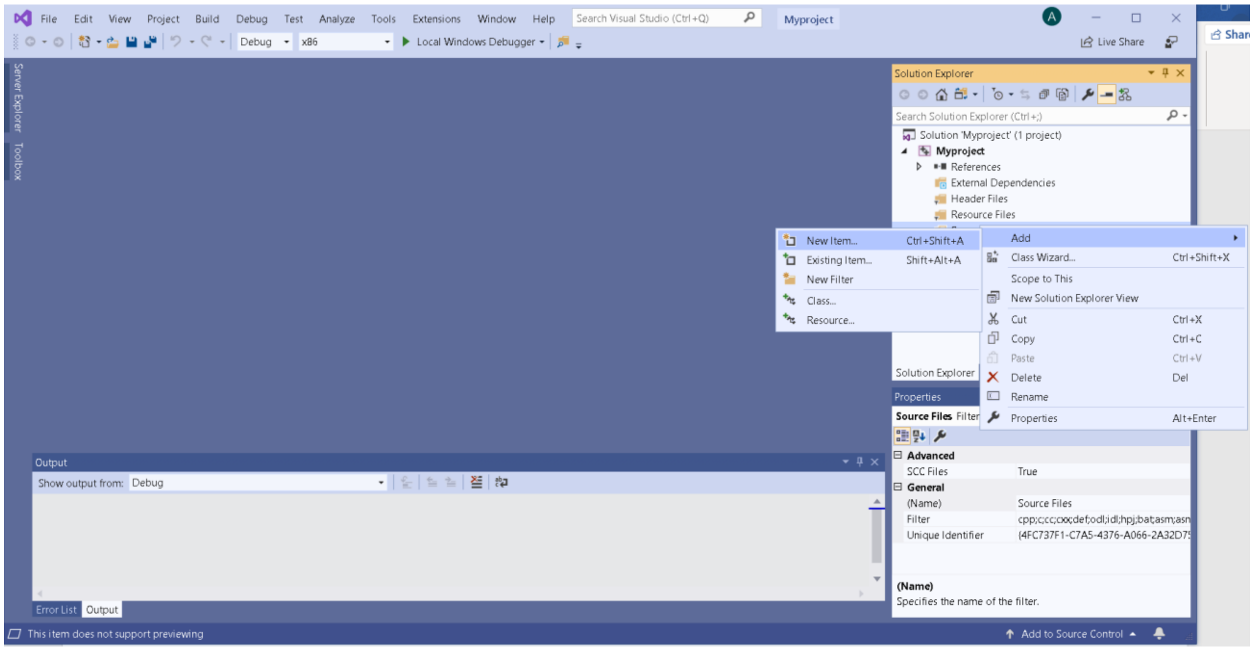


Visual C++ Build Customization Files

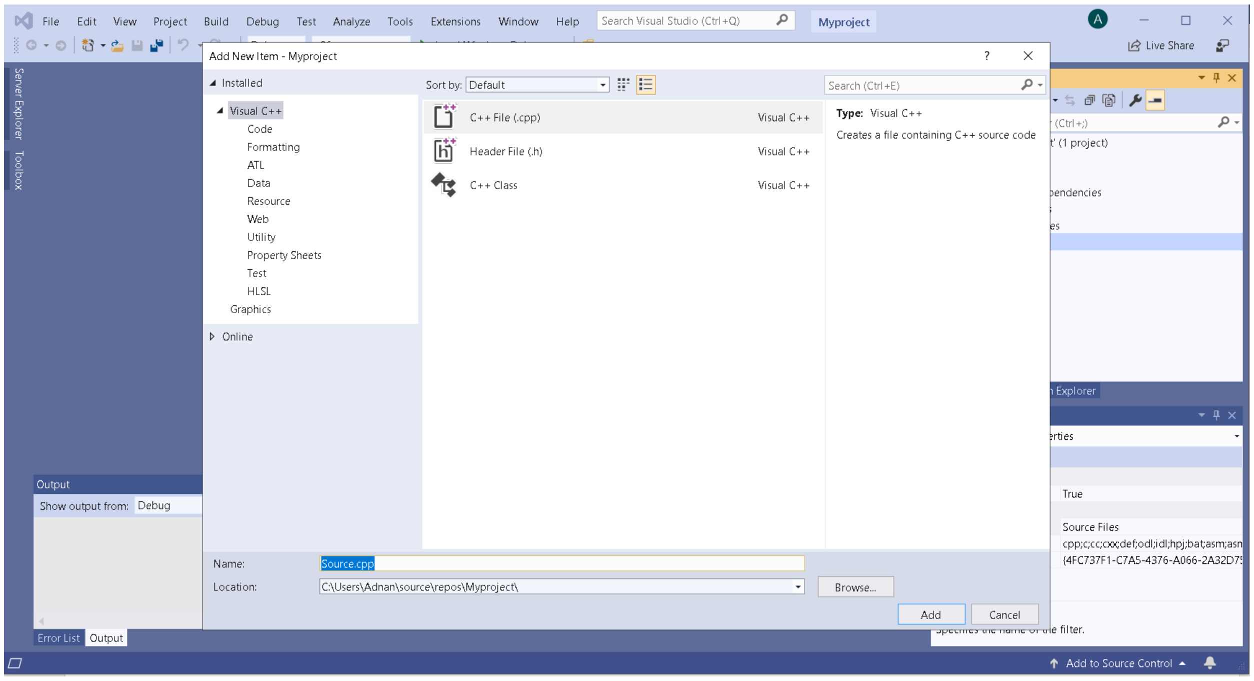
Available Build Customization Files:

	Name	Path
☐	ImageContentTask(.targe...	$(VCTargetsPath)\BuildCustomizations\ImageContentT
☐	lc(.targets, .props)	$(VCTargetsPath)\BuildCustomizations\lc.targets
☐	marmasm(.targets, .props)	$(VCTargetsPath)\BuildCustomizations\marmasm.targe
✓	masm(.targets, .props)	$(VCTargetsPath)\BuildCustomizations\masm.targets
☐	MeshContentTask(.target...	$(VCTargetsPath)\BuildCustomizations\MeshContentTa
☐	ShaderGraphContentTas...	$(VCTargetsPath)\BuildCustomizations\ShaderGraphC

Step#6: Right click on “Source Files” → Add → New Item



Step#7: Click in “C++ File (.cpp) , write a file name and click on Add.


Add New Item - Myproject

C++ File (.cpp)	Visual C++
Header File (.h)	Visual C++
C++ Class	Visual C++

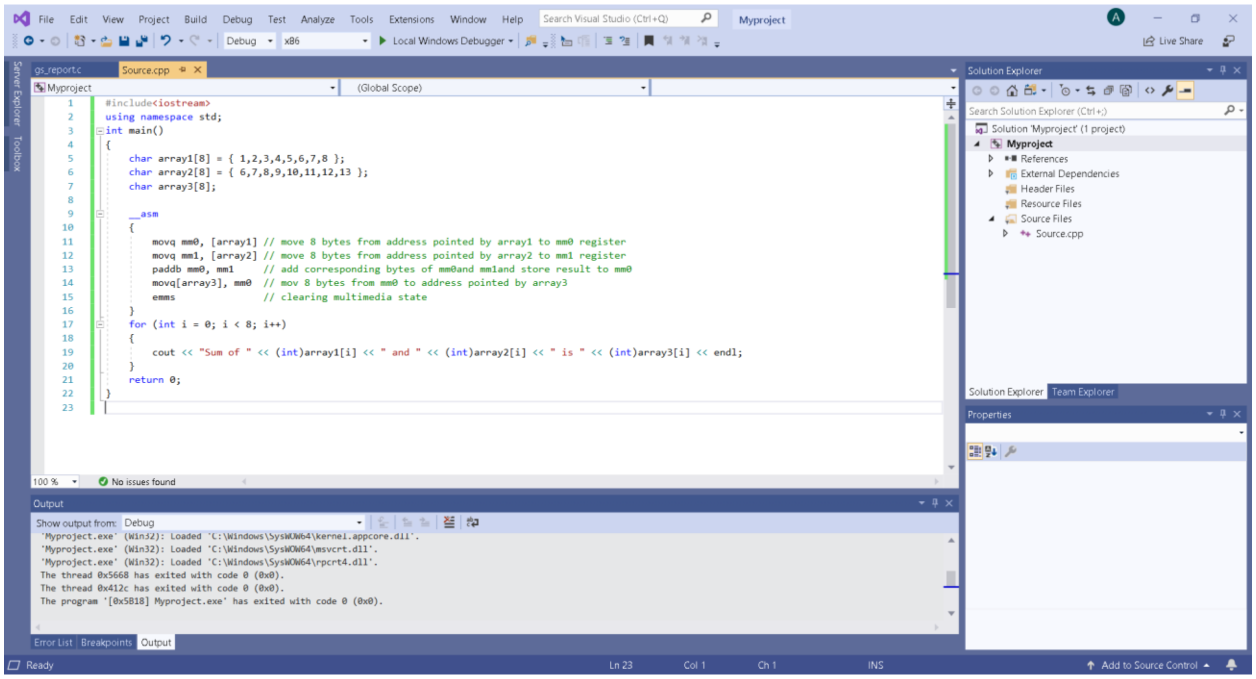
Type: Visual C++

Creates a file containing C++ source code

Name: Source.cpp

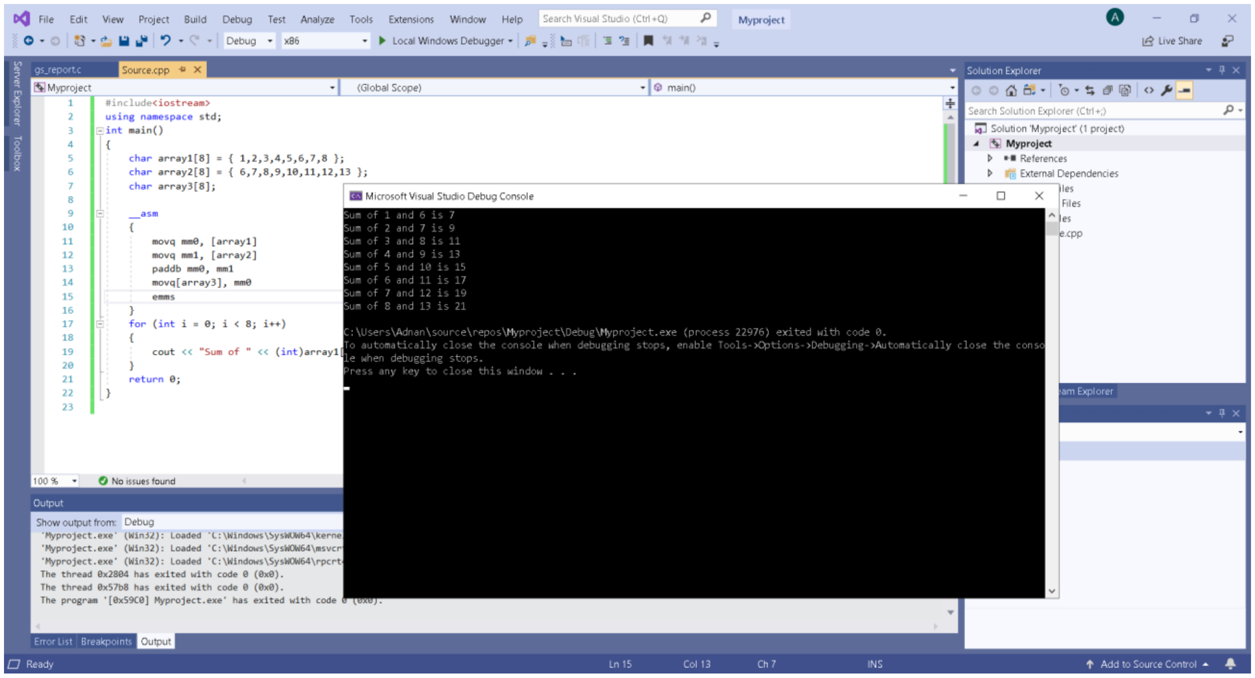
Location: C:\Users\Adnan\source\repos\Myproject\

Step#8: Paste code in Example#1 in the text window. Make sure that the environment is set to x86.


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

    __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 mm0and mm1and 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 < 8; i++)
    {
        cout << "Sum of " << (int)array1[i] << " and " << (int)array2[i] << " is " << (int)array3[i] << endl;
    }
    return 0;
}
```

```
'Myproject.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel.appcore.dll'.
'Myproject.exe' (Win32): Loaded 'C:\Windows\SysWOW64\msvcrt.dll'.
'Myproject.exe' (Win32): Loaded 'C:\Windows\SysWOW64\rpcrt4.dll'.
The thread 0x5668 has exited with code 0 (0x0).
The thread 0x412c has exited with code 0 (0x0).
The program '[0x5B18] Myproject.exe' has exited with code 0 (0x0).
```

Step#9: Click on “Local Windows Debugger” to run the code.



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

    __asm
    {
        movq mm0, [array1]
        movq mm1, [array2]
        paddb mm0, mm1
        movq[array3], mm0
        emms
    }
    for (int i = 0; i < 8; i++)
    {
        cout << "Sum of " << (int)array1[
    }
    return 0;
}
```

```
Sum of 1 and 6 is 7
Sum of 2 and 7 is 9
Sum of 3 and 8 is 11
Sum of 4 and 9 is 13
Sum of 5 and 10 is 15
Sum of 6 and 11 is 17
Sum of 7 and 12 is 19
Sum of 8 and 13 is 21

C:\Users\Adnan\source\repos\Myproject\Debug\Myproject.exe (process 22976) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

```
'Myproject.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kerne
'Myproject.exe' (Win32): Loaded 'C:\Windows\SysWOW64\msvcr
'Myproject.exe' (Win32): Loaded 'C:\Windows\SysWOW64\rpcrt
The thread 0x2804 has exited with code 0 (0x0).
The thread 0x57b8 has exited with code 0 (0x0).
The program '[0x59C0] Myproject.exe' has exited with code 0 (0x0).
```

Practice Tasks

Task-1

Write a program in C++ that declares two byte-type arrays of 80 elements each and initializes 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 initializes it with some data. The program then calculates the sum of even and odd elements using MMX instructions and prints them.