AVX 512

;CREEL?

Introduction

In this video and accompanying text, we will explore 3 different ways to program AVX 512. Then we will look at the low level mechanisms that AVX512 offers. The three methods are:

1. Agner Fog's Vector Class Library
2. C++ Compiler Intrinsics
3. Native Assembly Language

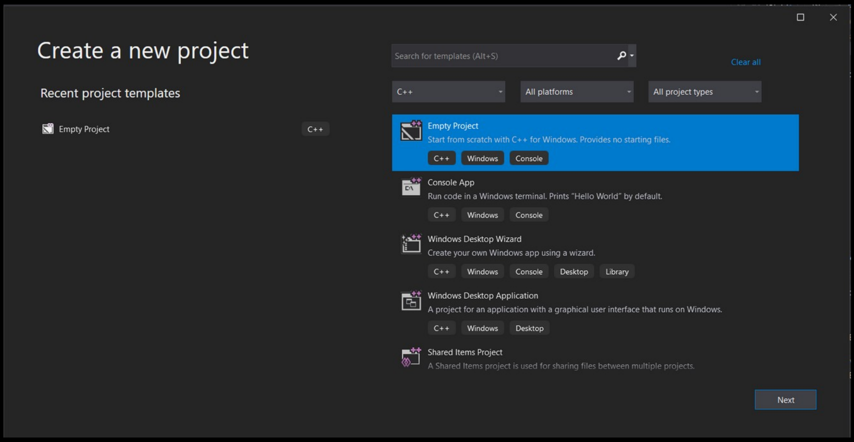
I will use Visual Studio 2019 Community. This software is available from the Microsoft Website:
https://visualstudio.microsoft.com/vs/community/

Note: It is possible to use other software, for example GNU C++ compiler and the GNU Assembler and NASM with Clang C++. The steps for adding files, linking and compiling are slightly different for other tool chains, as well as the Assembly syntax.

AVX512 is not available on many modern CPU's. The correct method for testing if AVX512 is available on your hardware is to call the CPUID instruction. For this text, I will assume we know that the hardware is AVX512 capable.

0. Beginning a new C++ Application

We will begin by creating a new C++ project.



Create a new project

Recent project templates

Empty Project — C++

Search for templates (Alt+S) — Clear all

C++ | All platforms | All project types

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

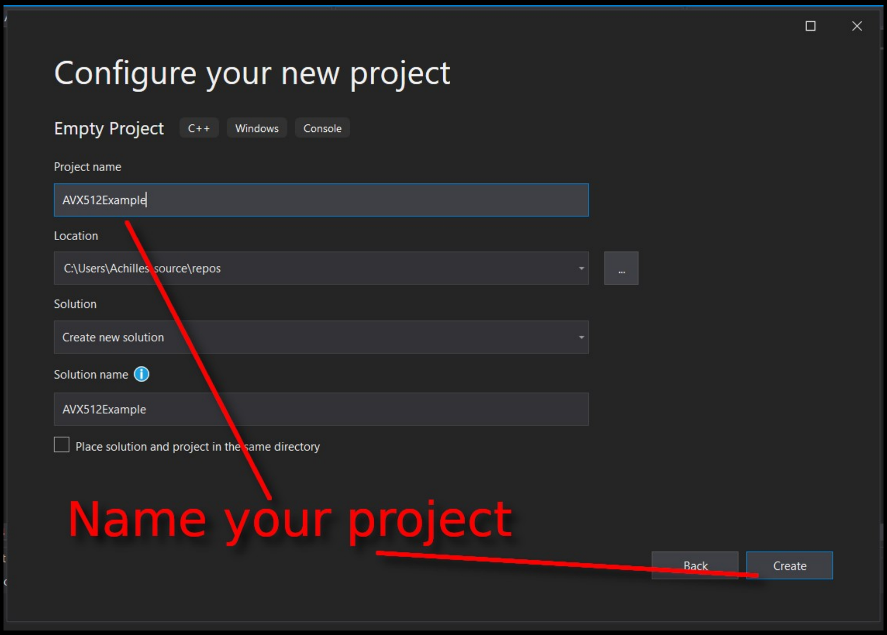
Windows Desktop Wizard
Create your own Windows app using a wizard.
C++ · Windows · Console · Desktop · Library

Windows Desktop Application
A project for an application with a graphical user interface that runs on Windows.
C++ · Windows · Desktop

Shared Items Project
A Shared Items project is used for sharing files between multiple projects.

I will use an "empty project" for this demonstration, but you can also add AVX512 capabilities to existing projects using similar steps to those described.

Give your project an appropriate name, and click the "Create" button.



Configure your new project

Empty Project — C++ · Windows · Console

Project name: AVX512Example

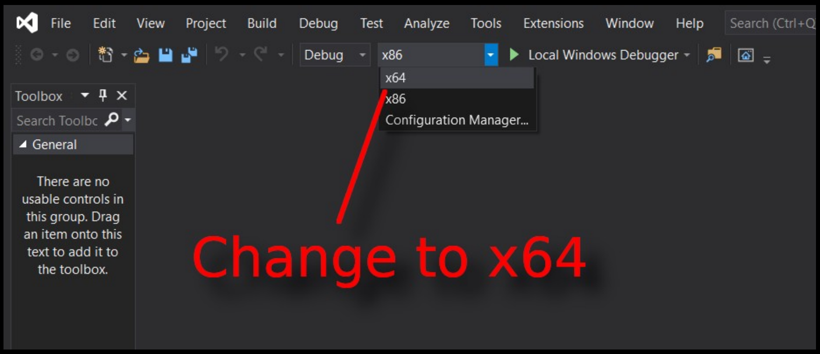
Location: C:\Users\Achilles\source\repos

Solution: Create new solution

Solution name: AVX512Example

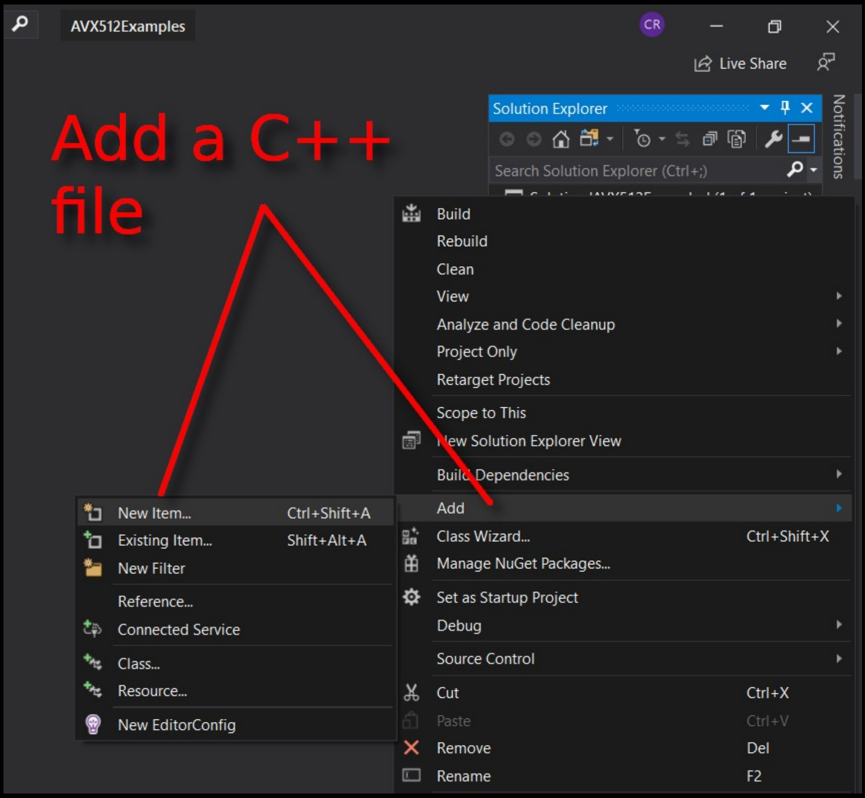
☐ Place solution and project in the same directory

We should ensure our project is set to build a 64 bit executable.



There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.

Click the "Configurations Manager", and select x64.



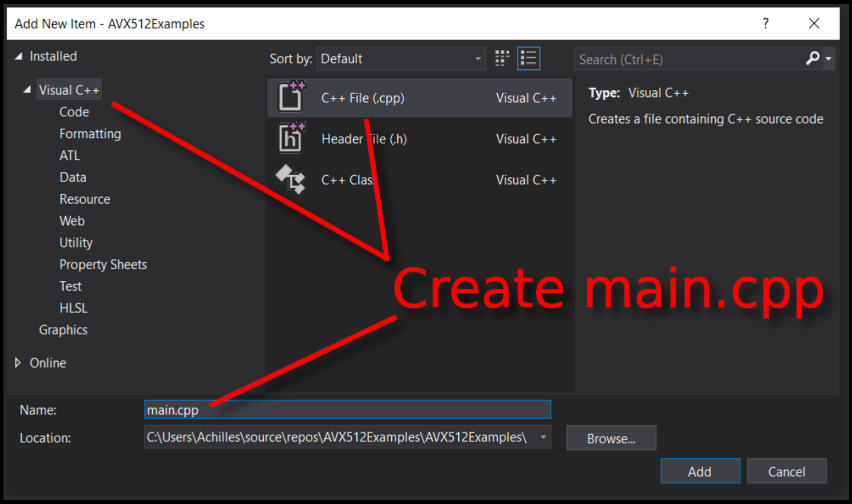
We are about to change the project properties, but, we have to add a C++ source file to our project or the C++ compiler options will not be visible in the project properties!

Right click on the name of your project in the “Solution Explorer”, select “Add” and “New Item”.

Alternatively, you can add a new item from the file menu “Project → Add New Item” or use short cut Ctrl+Shift+A.

On the “Add New Item” form, select “Visual C++” from the left panel, and “C++ File (.cpp)” from the centre panel. Then type a name for your file, “main.cpp”.

Click “Add” to add the file and close the page.



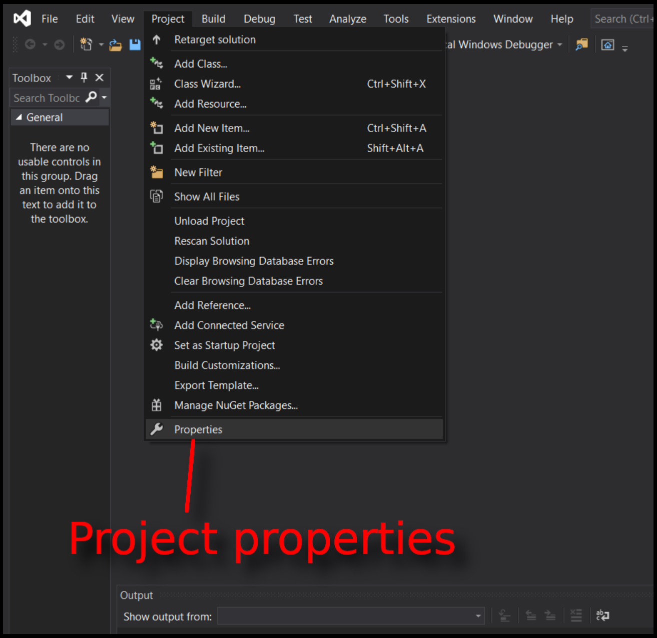
At this point, the next steps depend on whether we are coding VCL, Compiler Intrinsics, or if we are programming native Assembly. You can skip to the appropriate section or follow along all three methods for coding AVX512 and create a project where you can mix and match methods!

1. Agner Fog's VCL

Download the latest version of Agner Fog's VCL from the github:
https://github.com/vectorclass/version2

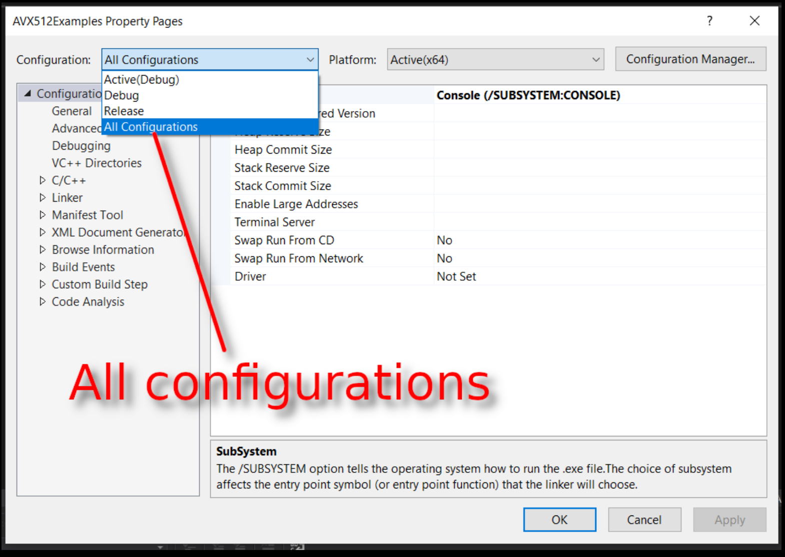
Extract the library somewhere on your computer. The library comes as a zipped archive of headers and source files. I have extracted the library to a folder on my Desktop called "../Prog/AgnerFogVCL".

Note: As an alternative to copying the VCL to a folder and reusing it, you can also simply copy and paste the extracted VCL files into your project's source folder. For more details on how to use the VCL, see the manual, available at:
https://www.agner.org/optimize/vcl_manual.pdf



Once you have exacted the library somewhere on your computer, return to your C++ project. In order to use the VCL, we need to change some of the project properties. Click the "Project Properties" button in the file menu:

The changes we will make are probably useful in both Debug as well as Release mode, so we can select "All Configurations" in the Configuration drop down box.



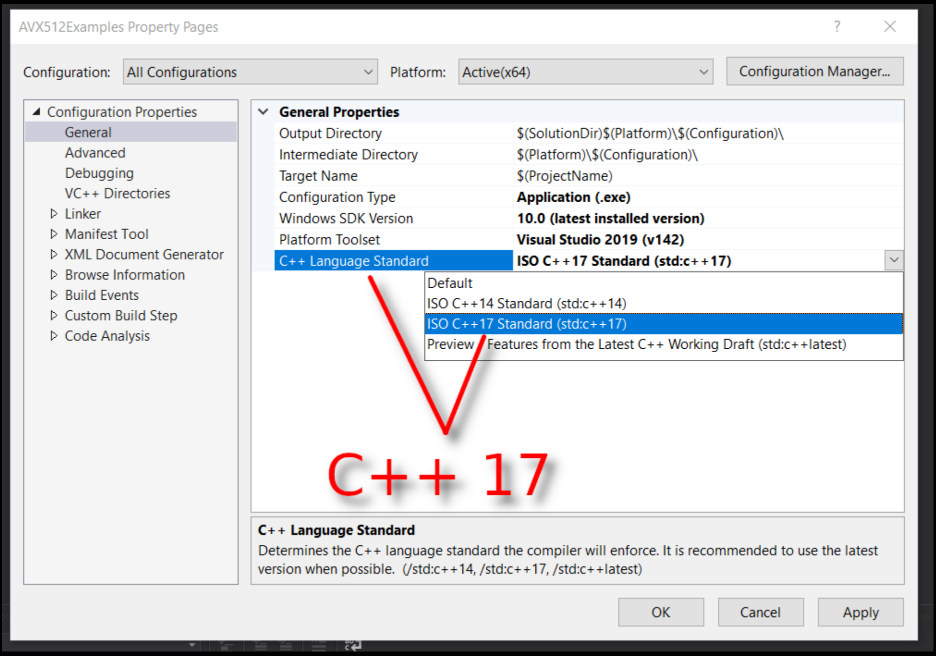
Configuration: All Configurations
Platform: Active(x64)

	Console (/SUBSYSTEM:CONSOLE)
red Version	
ze	
Heap Commit Size	
Stack Reserve Size	
Stack Commit Size	
Enable Large Addresses	
Terminal Server	
Swap Run From CD	No
Swap Run From Network	No
Driver	Not Set

SubSystem
The /SUBSYSTEM option tells the operating system how to run the .exe file.The choice of subsystem affects the entry point symbol (or entry point function) that the linker will choose.

Next, select the latest version of C++ in the language standard.

The VCL employs extensive use of modern C++.

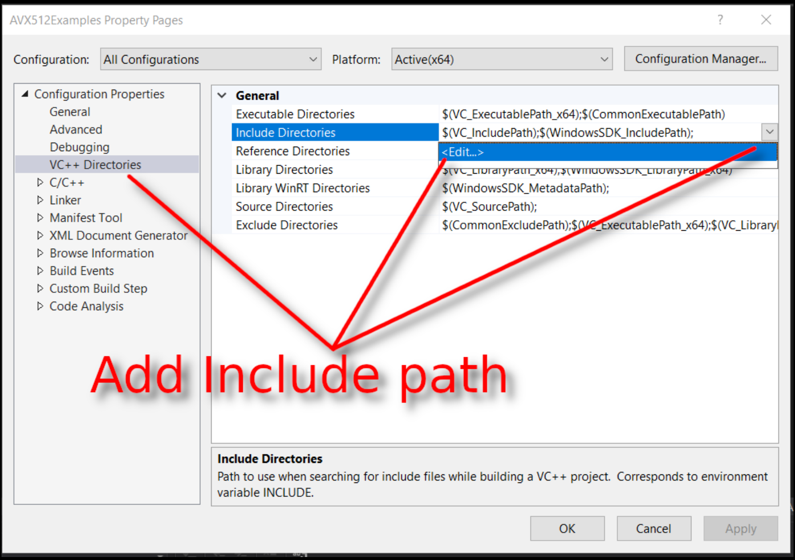


General Properties	
Output Directory	$(SolutionDir)$(Platform)\$(Configuration)\
Intermediate Directory	$(Platform)\$(Configuration)\
Target Name	$(ProjectName)
Configuration Type	**Application (.exe)**
Windows SDK Version	**10.0 (latest installed version)**
Platform Toolset	**Visual Studio 2019 (v142)**
C++ Language Standard	**ISO C++17 Standard (std:c++17)**

```
Default
ISO C++14 Standard (std:c++14)
ISO C++17 Standard (std:c++17)
Preview - Features from the Latest C++ Working Draft (std:c++latest)
```

C++ Language Standard
Determines the C++ language standard the compiler will enforce. It is recommended to use the latest version when possible. (/std:c++14, /std:c++17, /std:c++latest)

Next, we need to specify the VCL folder as an Include Directory, so Visual Studio knows where to find the headers.

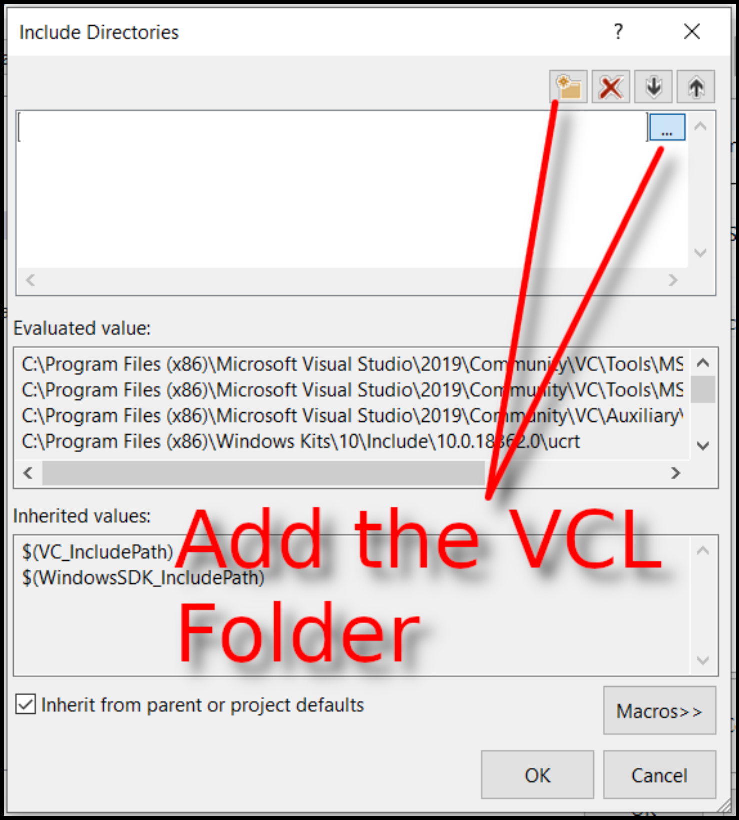


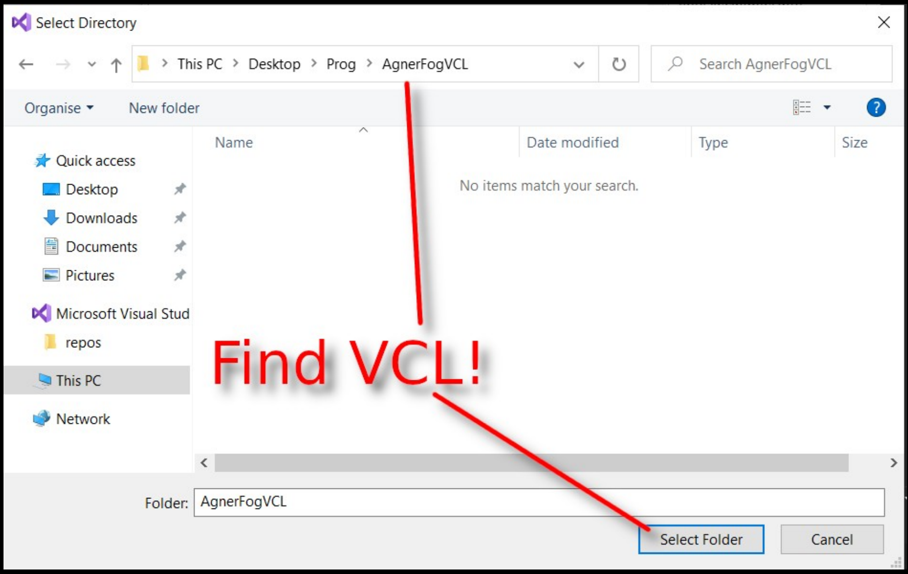
General	
Executable Directories	$(VC_ExecutablePath_x64);$(CommonExecutablePath)
Include Directories	$(VC_IncludePath);$(WindowsSDK_IncludePath);
Reference Directories	<Edit...>
Library Directories	$(VC_LibraryPath_x64);$(WindowsSDK_LibraryPath_x64)
Library WinRT Directories	$(WindowsSDK_MetadataPath);
Source Directories	$(VC_SourcePath);
Exclude Directories	$(CommonExcludePath);$(VC_ExecutablePath_x64);$(VC_Library

Include Directories
Path to use when searching for include files while building a VC++ project. Corresponds to environment variable INCLUDE.

In the "VC++ Directories" section, click the small drop down arrow for "Include Directories", and then "Edit".

In the “Include Directories” form, click the small new folder button, and then the “…”, to add a new folder to the search paths.


Include Directories
?
...
Evaluated value:
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MS
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MS
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Auxiliary\
C:\Program Files (x86)\Windows Kits\10\Include\10.0.18362.0\ucrt
Inherited values:
$(VC_IncludePath)
$(WindowsSDK_IncludePath)
Add the VCL Folder
Inherit from parent or project defaults
Macros>>
OK
Cancel



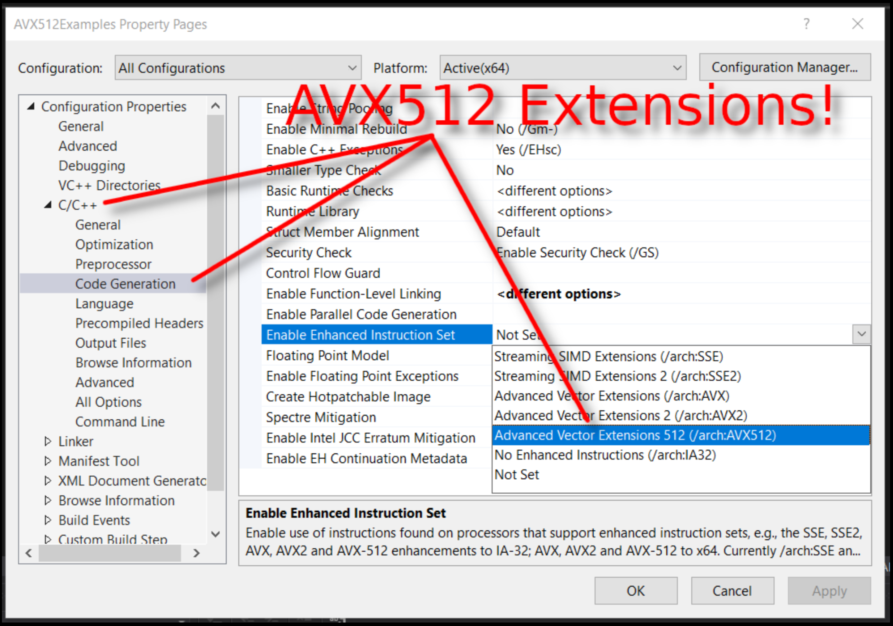
Find the folder where you extracted the VCL zip file to, and then click “Select Folder”.

Click “Ok” on the “Include Directories” form to add the directory and close the form.

Now that Visual Studio knows where to find the VCL headers, we have to specify the Enhanced Instruction Set for the compiler.

Set the Instruction set to AVX512 in the C/C++ compiler options.

VCL will automatically generate low level code based on the specified instruction set.



Configuration: All Configurations Platform: Active(x64)

- Configuration Properties
  - General
  - Advanced
  - Debugging
  - VC++ Directories
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled Headers
    - Output Files
    - Browse Information
    - Advanced
    - All Options
    - Command Line
  - Linker
  - Manifest Tool
  - XML Document Generatc
  - Browse Information
  - Build Events
  - Custom Build Step

Enable String Pooling	
Enable Minimal Rebuild	No (/Gm-)
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	<different options>
Runtime Library	<different options>
Struct Member Alignment	Default
Security Check	Enable Security Check (/GS)
Control Flow Guard	
Enable Function-Level Linking	**<different options>**
Enable Parallel Code Generation	
Enable Enhanced Instruction Set	Not Set
Floating Point Model	
Enable Floating Point Exceptions	
Create Hotpatchable Image	
Spectre Mitigation	
Enable Intel JCC Erratum Mitigation	
Enable EH Continuation Metadata	

Streaming SIMD Extensions (/arch:SSE)
Streaming SIMD Extensions 2 (/arch:SSE2)
Advanced Vector Extensions (/arch:AVX)
Advanced Vector Extensions 2 (/arch:AVX2)
Advanced Vector Extensions 512 (/arch:AVX512)
No Enhanced Instructions (/arch:IA32)
Not Set

Enable Enhanced Instruction Set
Enable use of instructions found on processors that support enhanced instruction sets, e.g., the SSE, SSE2, AVX, AVX2 and AVX-512 enhancements to IA-32; AVX, AVX2 and AVX-512 to x64. Currently /arch:SSE an...

We have finished setting up our project to use the VCL. You can add the following code to your main.cpp file to test the library:

```
#include <iostream>
#include <vectorclass.h>

int main()
{
        Vec8d a = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 };
        Vec8d b = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 };
        Vec8d c = a + b;

        for (int i = 0; i < 8; i++)
        {
                std::cout << i << "." << c[i] << std::endl;
        }

        return 0;
}
```

Vec8d is a VCL vector, storing 8 double precision floating point values. In this test code, we store the numbers 1.0 to 8.0 in two vectors called `a` and `b`, and then we use the +, to add the corresponding values and store the results in the c vector.

One of the most important features of VCL is the convenience of the code. We can use standard operators for arithmetic, such as addition, subtraction, multiplication and division (although integer division is more complicated, since there are no SIMD integer division instructions). VCL code is translated into compiler intrinsics, so very little speed is lost. In addition to basic arithmetic, there are a large number of other interesting and powerful functions in the VCL. For more detailed information on the library, please see the manual.

The addition here will be carried out using an AVX512 instruction, “VADDPD zmm, zmm, zmm/mem256”.

2. Compiler Intrinsics

To use the compiler intrinsics, we need to include the <intrin.h> header.

```
#include <iostream>
#include <intrin.h>

int main()
{
	__m512d a = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 };
	__m512d b = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 };

	__m512d c = _mm512_add_pd(a, b);

	for (int i = 0; i < 8; i++)
		std::cout << i << "." << ((double*)&c)[i] << std::endl;

	return 0;
}
```

The compiler intrinsics are specific to particular instructions. The intrinsic *_mm512_add_pd* corresponds to the AVX512 instruction “VADDPD zmm, zmm, zmm/mem512”. There are many instruction intrinsics available. To see a list, you can type “_mm512_” and Visual Studio’s intellisense will provide a context menu showing available intrinsics.

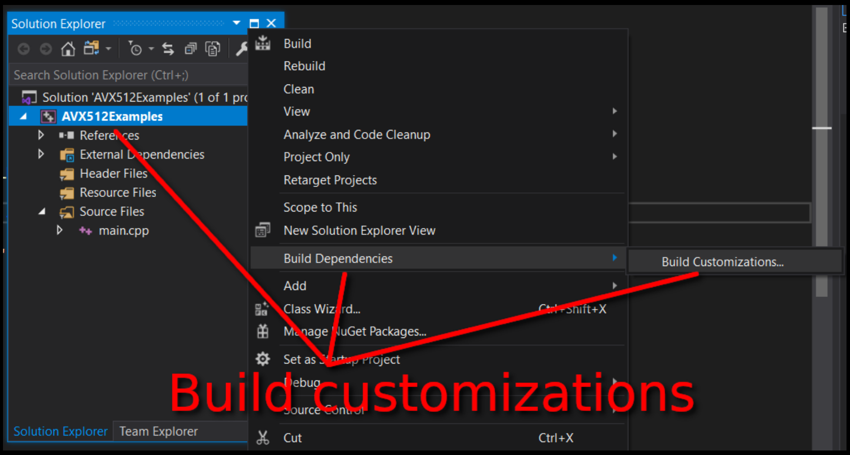
There are several different data types for vectors available, the __m512d corresponds to an ANV512 vector containing 8 double precision floating point values.

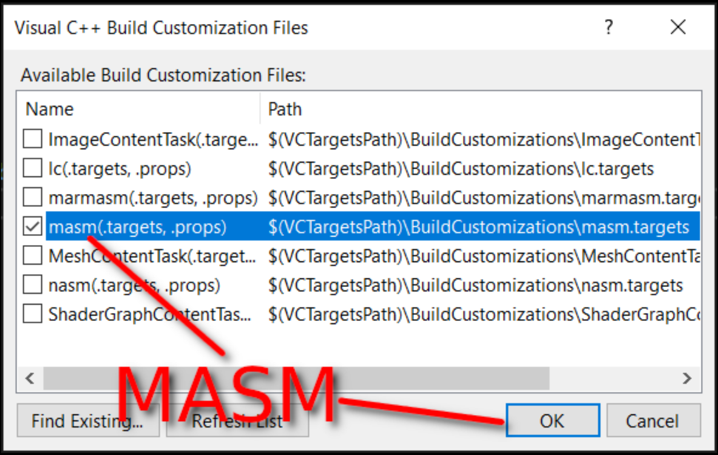
3. Native Assembly Language

The final way we will explore for implementing AVX 512 is native x86/64 Assembly language.

To include an Assembly language source file in your project, we have to specify the MASM build customisation.

Right click on your project name in the solution explorer. Select the “Build Dependencies” option from the context menu, and click “Build Customizations…”





Visual C++ Build Customization Files

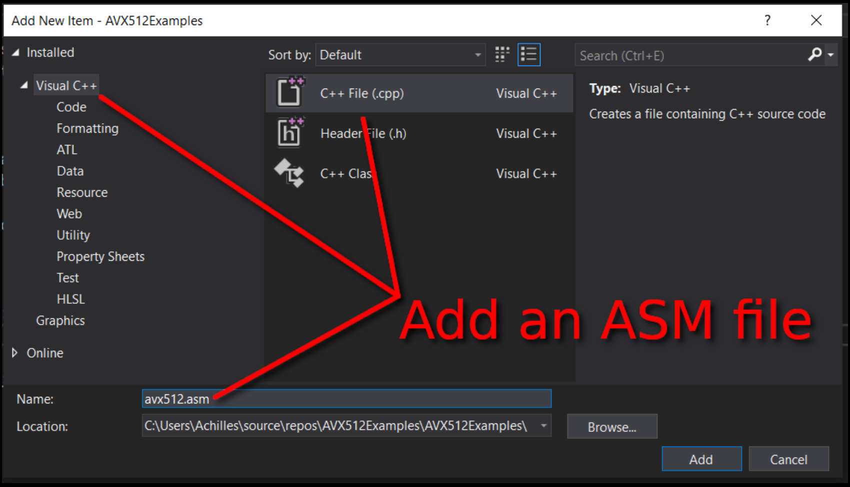
Available Build Customization Files:

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

In the “Build Customization” form, you will see a list of all the build customisations you have installed. We want to use MASM to assemble “.asm” files, so check the box beside “masm(.targets, .props)”. Then click “ok”.

Next, we have to add an assembly source file to our project. Press Ctrl+Shift+A, or right your project in the solution explorer and select “Add new item” to add a new item to your project.

In the Add New Item box, select “Visual C++” from the left panel, then “C++ File (.cpp)”. Type a name for your new file. Remember that your file name should end with the “.asm” extension, such that Visual Studio will know to use the MASM build customisation. I have called my file “avx512.asm”. Click “Add” to create and add the file to your project.



Add New Item - AVX512Examples

Name: avx512.asm

Location: C:\Users\Achilles\source\repos\AVX512Examples\AVX512Examples\

Type: Visual C++

Creates a file containing C++ source code

Type or copy the following code into the “.asm” file:

```
.code
; void ASM_ADDPD(*rcx=C, *rdx=A, *r8=B)
; Adds vectors A and B, each containing 8 double
; precision floats, and stores the 8 results in
; the C vector.
ASM_ADDPD proc
       ; Move A into the ZMM0 register
       vmovupd zmm0, zmmword ptr [rdx]

       ; Add B to ZMM0, store the 8 sums in ZMM0
       vaddpd zmm0, zmm0, zmmword ptr [r8]

       ; Write the sums to C
       vmovupd zmmword ptr [rcx], zmm0
       ret
ASM_ADDPD endp
end
```

This code defines a function which adds two vectors of 8 doubles each, and stores the result. There are two different AVX512 instruction in this code:

VMOVUPD: Move unaligned packed doubles. This instruction is used to move data from RAM into the registers. It is also used to move data from the registers back to RAM. There is also an instruction to move aligned doubles, MOVAPD, if you know that your data is aligned and/or you are moving data from register to register.

VADDPD: Add packed doubles. This instruction adds all 8 pairs of doubles from the second and third operands, and it stores the 8 results in the first operand.

Note: The 256 bit AVX versions of these instructions have the same mnemonics: VMOVUPD and VADDPD. In the code above, we use the ZMM registers, rather than the YMM registers. This will ensure we are calling the 512 bit versions of the instruction.

To call this function from a C++ code file, we must declare it first. The following code is an example of how we might call this function from the main.cpp file:

```
#include <iostream>

// Declare the ASM function
extern "C" void ASM_ADDPD(double* C, double* A, double* B);

int main()
{
       double A[8] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 };
       double B[8] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 };
       double C[8];

       ASM_ADDPD(C, A, B);  // Call the ASM function

       for (int i = 0; i < 8; i++)
       {
              std::cout << i << "." << C[i] << std::endl;
       }

       return 0;
}
```

Notice the line “extern “C””. This line is the declaration for the external Assembly function. We have to include a declaration like this for every function we want to be available to the C++. The output of this program is as follows:

```
0.2
1.4
2.6
3.8
4.10
5.12
6.14
7.16
```

We can see from the 2, 4, 6, 8, etc. That all 8 pairs of double from the A and B arrays were added together.

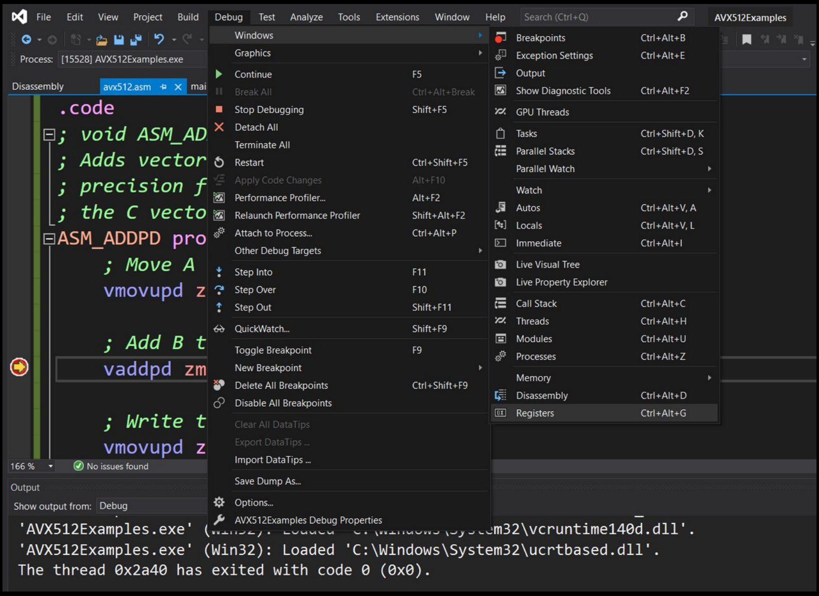
AVX 512 Mechanisms

We have briefly looked at how we can execute AVX 512 code in three different ways. Let us explore some of the other important mechanisms which AVX 512 offers.

32 512 Bit Registers

AVX512 contains 512 bit registers. We can view the contents of these registers by examining the Registers Window while the program stops at a break point. You can find the Registers window in the file menu: Debug → Windows → Registers.

The registers window is only available when the program is stopped at a breakpoint.

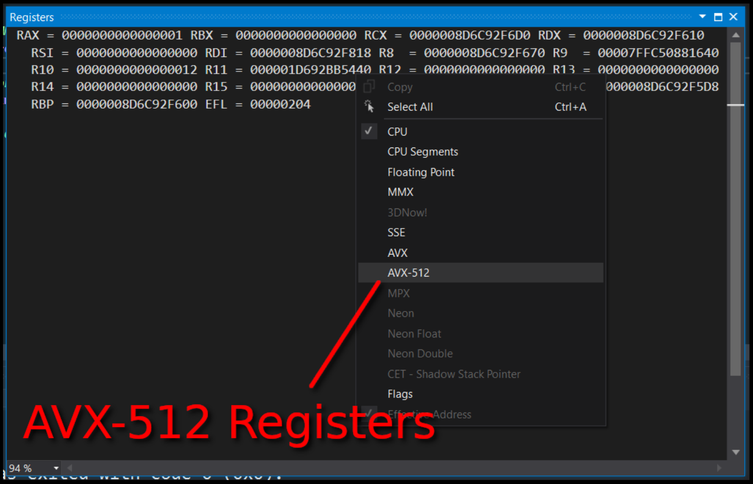


```
.code
; void ASM_AD
; Adds vector
; precision f
; the C vecto
ASM_ADDPD pro
    ; Move A
    vmovupd z

    ; Add B t
    vaddpd zm

    ; Write t
    vmovupd z
```

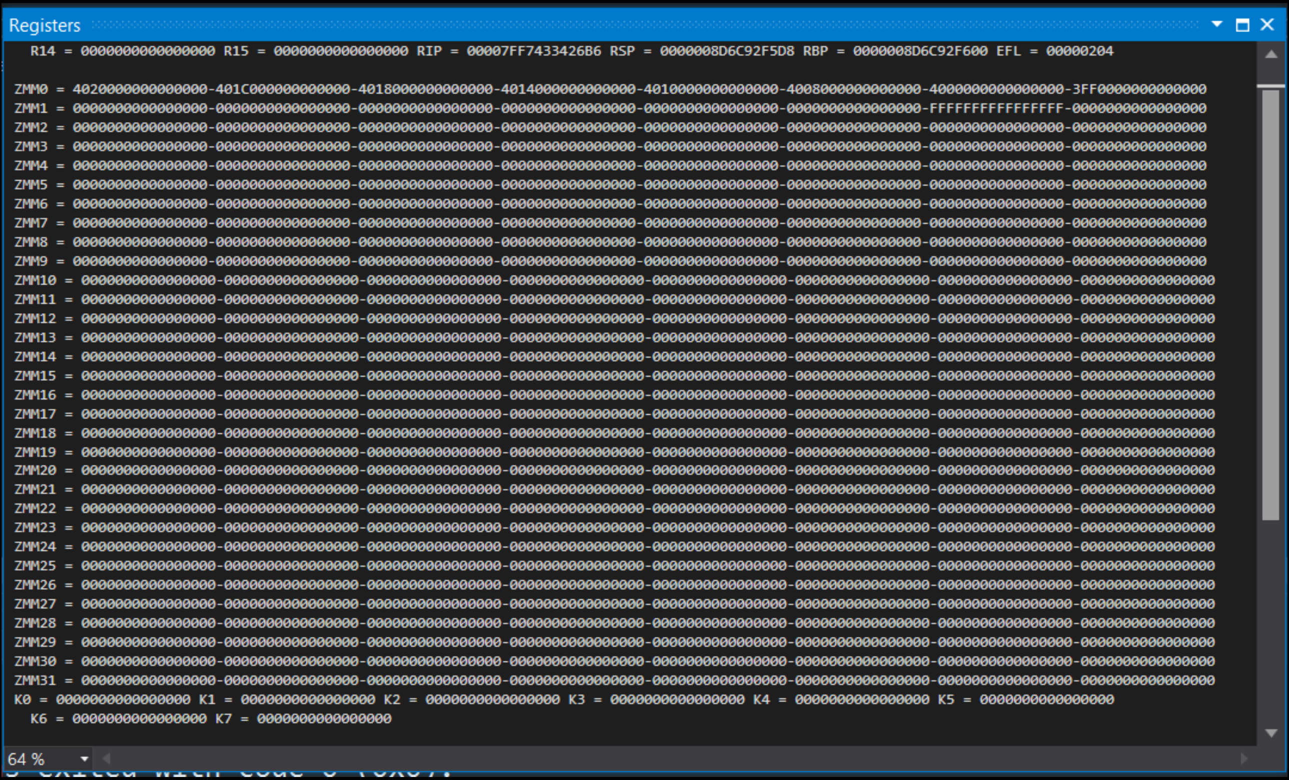
```
'AVX512Examples.exe' (Win32): Loaded 'C:\Windows\System32\vcruntime140d.dll'.
'AVX512Examples.exe' (Win32): Loaded 'C:\Windows\System32\ucrtbased.dll'.
The thread 0x2a40 has exited with code 0 (0x0).
```



```
RAX = 0000000000000001 RBX = 0000000000000000 RCX = 0000008D6C92F6D0 RDX = 0000008D6C92F610
  RSI = 0000000000000000 RDI = 0000008D6C92F818 R8  = 0000008D6C92F670 R9  = 00007FFC50881640
  R10 = 0000000000000012 R11 = 000001D692BB5440 R12 = 0000000000000000 R13 = 0000000000000000
  R14 = 0000000000000000 R15 = 00000000000000                              000008D6C92F5D8
  RBP = 0000008D6C92F600 EFL = 00000204
```

Once open, the registers window will not show the AVX512 registers by default. To show them, right-click somewhere in the widow to open the context menu, and select AVX512.

You will most likely find that the AVX 512 registers do not display in a particularly convenient way, being as they extremely wide. But if you resize the window, you should see something like the image pictured below.



Registers

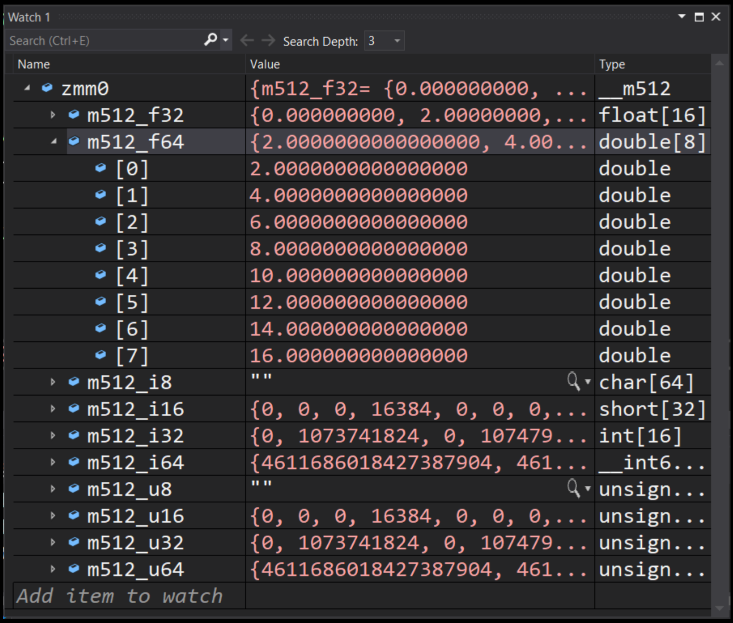
```
  R14 = 0000000000000000 R15 = 0000000000000000 RIP = 00007FF7433426B6 RSP = 0000008D6C92F5D8 RBP = 0000008D6C92F600 EFL = 00000204

ZMM0 = 4020000000000000-401C000000000000-4018000000000000-4014000000000000-4010000000000000-4008000000000000-4000000000000000-3FF0000000000000
ZMM1 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-FFFFFFFFFFFFFFFF-0000000000000000
ZMM2 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM3 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM4 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM5 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM6 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM7 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM8 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM9 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM10 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM11 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM12 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM13 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM14 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM15 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM16 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM17 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM18 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM19 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM20 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM21 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM22 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM23 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM24 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM25 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM26 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM27 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM28 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM29 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM30 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
ZMM31 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000-0000000000000000
K0 = 0000000000000000 K1 = 0000000000000000 K2 = 0000000000000000 K3 = 0000000000000000 K4 = 0000000000000000 K5 = 0000000000000000
  K6 = 0000000000000000 K7 = 0000000000000000
```

64 %

Notice that there are 32 of the registers named ZMM0 to ZMM31. AVX512 increases the available register count from 16 to 32. Also, you can see the K mask registers here, listed along the bottom of the window. We will see K masks in action just a moment!

We can also set watches on the registers and the vectors from the Assembly code, VCL or the compiler intrinsics. The watch window on a register displays all the possible data interpretations, so be careful and make sure you are looking at the correct data types when you debug your AVX 512 code. Here is an image of the Watch Window on ZMM0 after the VADDPD instruction was executed from the Assembly code in the previous section:



Watch 1

Search (Ctrl+E) Search Depth: 3

Name	Value	Type
zmm0	{m512_f32= {0.000000000, ...	__m512
m512_f32	{0.000000000, 2.00000000,...	float[16]
m512_f64	{2.0000000000000000, 4.00...	double[8]
[0]	2.0000000000000000	double
[1]	4.0000000000000000	double
[2]	6.0000000000000000	double
[3]	8.0000000000000000	double
[4]	10.000000000000000	double
[5]	12.000000000000000	double
[6]	14.000000000000000	double
[7]	16.000000000000000	double
m512_i8	""	char[64]
m512_i16	{0, 0, 0, 16384, 0, 0, 0,...	short[32]
m512_i32	{0, 1073741824, 0, 107479...	int[16]
m512_i64	{4611686018427387904, 461...	__int6...
m512_u8	""	unsign...
m512_u16	{0, 0, 0, 16384, 0, 0, 0,...	unsign...
m512_u32	{0, 1073741824, 0, 107479...	unsign...
m512_u64	{4611686018427387904, 461...	unsign...
Add item to watch		

In the code, we were working with 64 bit doubles. So I have expanded the m512_f64 branch. The data in a register has no particular type – Assembly is not a type safe language. But you can see the results from the additions here: 2.0, 4.0, 6.0, etc.

K Masks

There are 8 masking registers, called K0 to K7. Each is 64 bits wide. When a K mask is used with an instruction, the results are written to the destination register only when there is a 1 in the corresponding position in the K mask. All elements which correspond to 0's in the K mask will remain unchanged!

For example, the code from the Assembly example is listed below, but I have included the use of a K mask. Here, we use a mask of 55h. Which, in binary is 01010101_2. This means every second result will be stored, because there 1's in every second position of he mask. But where there are 0's, the results will not change:

```
.code
; void ASM_ADDPD(*rcx=C, *rdx=A, *r8=B)
; Adds vectors A and B, each containing 8 double
; precision floats, and stores the 8 results in
; the C vector.
ASM_ADDPD proc
       ; Move A into the ZMM0 register
       vmovupd zmm0, zmmword ptr [rdx]

       ; Move some data into EAX
       mov eax, 55h

       ; Copy that data into K1
       kmovd k1, eax

       ; Add using K mask!
       vaddpd zmm0{k1}, zmm0, zmmword ptr [r8]

       ; Write the sums to C
       vmovupd zmmword ptr [rcx], zmm0
       ret
ASM_ADDPD endp
end
```

Note: K0 is reserved. It is used implicitly when a mask is not needed (Intel Reference Manuals). It is set to 0xffffffffffffffff, and this should not be changed. If you do try to change the value of K0, the Visual Studio watch window reports the updated K0 value, but the mask is still applied as 0xffffffffffffffff regardless.

Notice the instructions to move data into a K mask. The MOV instruction cannot directly move data into a K mask. So, I have used two instructions:

```
MOV EAX, 55h      ; Move data into EAX
KMOVD K1, EAX     ; Move data into K1
```

Once we have the bits in our K1 mask register, we can selectively write the outputs by supplying the {k1} decoration:

VADDPD zmm0{k1}, zmm0, zmmword ptr [r8]

If we execute the instruction with the same data as before, the result will be:

2.0, 2.0, 6.0, 4.0, 10.0, 6.0, 14.0, 8.0

The vectors each contained 1, 2, 3, 4, 5, 6, 7, 8. Because of the mask, elements with an even index have been doubled, but elements with odd indices remain unchanged.

Zero Mask

In addition to a K mask, we can include the Zero decoration, {z}. If used, we place this decoration after the K mask in the instruction mnemonic:

```
VADDPD zmm0{k1}{z}, zmm0, zmmword ptr [r8]
```

When we use a Zero mask, it means that the results which correspond to Zero in the K mask should be Zeroed. That is, instead of writing the results to positions where there is 1 in the K mask, and leaving the others unchanged; using a Zero mask will write the results where there is a 1 in the K mask but it will Zero everything else.

Using a Zero mask on the vectors from before will result in the following:

2.0, 0.0, 6.0, 0.0, 10.0, 0.0, 14.0, 0.0

We still get the sums, the same as before, only the remaining elements have been Zeroed.

We are using K masks and doubles in these examples, but the K masks are 64 bits wide – which corresponds to the number of bytes in an AVX512 register. We can perform extremely fast branchless conditions using K masks with up to 64 byte operations at a time. The K mask instructions do not appear to take any extra time to execute.

Note: Because of the masking abilities of AVX512, the blending instructions have changed considerably since the AVX 1 and 2. And, many instructions which did not previously require us to specify a data type, must now specify one. For instance, the AVX instruction VPAND performs a Boolean AND between 256 bit registers – and the data type in the registers is irrelevant. But in AVX512, we have to add the data size, just in case a K mask is used: VPANDD.

Automatic Broadcasting

AVX512 features a new broadcasting decoration. To broadcast is to copy the same value to all elements of a SIMD vector. If we wish to add 5.0 to every value of a vector of doubles, for example, we might fill a vector with 5.0 by using a broadcast instruction, and then perform the addition.

In AVX512, we can skip the explicit broadcast instruction, and perform the broadcast and the add in a single instruction!

Note: There is currently no Intrinsic support for the broadcasting feature, so if you want to use this feature, Assembly is the only option.

```
.data
scalar_double real8 5.0

.code
ASM_ADD5 proc
       ; Move A into the ZMM0 register
       vmovupd zmm0, zmmword ptr [rdx]

       ; Store address of scalar_double in rax
       lea rax, scalar_double

       ; Broadcast the 5.0 from the data segment and add
       vaddpd zmm0, zmm0, real8 bcst [rax]

       ; Write the sums to C
       vmovupd zmmword ptr [rcx], zmm0
       ret
ASM_ADD5 endp
end
```

In the code above, we first define a double, or real8 in the data segment, called scalar_double. This will allocate 5.0 as a 64 bit double, it is not a vector. This is the single value 5.0 as an 8 byte double.

Then in the code we have a function called ASM_ADD5. The function begins by reading 8 double precision floating point values passed as the first parameter (*RDX). The values are stored in ZMM0.

Then, we read the address of the *scalar_double* variable into the RAX register, using the LEA instruction. LEA is used to create pointers to data. RAX points to *scalar_double*.

Next, we perform the addition using the VADDPD instruction. But notice the "bcst":

VADDPD zmm0, zmm0, real8 bcst [rax]

This means that *scalar_double* (which RAX points to), will be broadcast to a temporary vector:

temporary = [5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0]

And then this temporary vector will be used as the third parameter to the sum instruction.

Note: The MASM syntax for the broadcast is not standard! The Intel syntax (also used by NASM) for this operation would be "VADDPD zmm0, zmm0, [rdx] {1to8}". In the Intel syntax, there is also decorations {1to4} and {1to16}, depending on the number of elements being broadcast.

These broadcasts are only available for the data elements of size 32 and 64 bits. They are not available for every instruction.

Rounding Control

In many floating point instructions, implicit rounding is performed. This is because floating point is not able to accurately represent most values.

In older instruction sets, SSE and AVX, when a result is rounded, the rounding mode is specified in the MXCSR register. In AVX512, we can specify the rounding mode in the instructions themselves. This means we can change rounding modes on an instruction by instruction basis!

The rounding mode decorations in MASM are:

{rn-sae}	Round to nearest. Numbers are rounded up or down toward the nearest whole number.

{rd-sae}	Round down. Numbers are rounded down, towards -Infinity, regardless of how near they are to any whole number.

{ru-sae}	Round up. Numbers are rounded up, towards +Infinity, regardless of how near they are to any whole number.

{rz-sae}	Round to ward Zero. Positive number are rounded down, and negative numbers are rounded up.

Note: If we do not specify a rounding mode, the rounding mode specified in the MXCSR register will be used.

To use the rounding modes, specify the appropriate decoration at the end of the instruction. The following code shows the VCVTPS2DQ instruction:

```
.data
some_singles real4 1.5, 2.7, -2.7, 1.9, 0.5, 7.3, -7.3, 12.3, -5.9, 1.3, 1.6, -6.1, 5.4,
1.5, 4.1, -4.2

.code
ASM_ADDPD proc
       ; Move some_singles into
       vmovupd zmm0, zmmword ptr [some_singles]

       vcvtps2dq zmm1, zmm0{rn-sae}
       ret
ASM_ADDPD endp
end
```

In the code above, we use the VCVTPS2DQ instruction to cast the single precision floating point array in the data segment to 32 bit integers. We use the {rn-sae} decoration, to cause the rounding mode to be “to nearest”. This will cause the elements to be cast as follows:

Name	Value	Type
◢ m512_i32	{2, 3, -3, 2, 0, 7, -7, 12, -6, 1, 2, -6, 5, 2, 4, -4}	int[16]
[0]	2	int
[1]	3	int
[2]	-3	int
[3]	2	int
[4]	0	int
[5]	7	int
[6]	-7	int
[7]	12	int
[8]	-6	int
[9]	1	int
[10]	2	int
[11]	-6	int
[12]	5	int
[13]	2	int
[14]	4	int
[15]	-4	int

As you can see from the output, the rounding has been applied. The 2.7 was rounded up to 3, and the -2.7 was rounded down to -3. This is different from the normal C++ casting, which would truncate, or round towards Zero, and return 2 and -2 respectively.

Note: The exact details of how floating point results are rounded is specified in the IEEE754 floating point standard. Please see Wikipedia for more information: https://en.wikipedia.org/wiki/IEEE_754

Note: The NASM sytax requires a comma between the final parameter and the rounding mode: VCVTPS2DQ zmm0, zmm1, {rz-sae}

Compressed Displacement

The new compressed displacement addressing mode is mostly handled by compilers and assemblers automatically. It is a space saving mechanism for use within unrolled vector loops. It allows any displacements which are evenly divisible by the operand size to be stored in the machine code in a compressed format. To best illustrate the mechanism, we can look at the disassembly of two instructions, one that does not use AVX512 compressed displacement, and another that does:

```
; This instructions will not use Compressed Displacement!
vaddps zmm0, zmm0, zmmword ptr [rcx+123]

; This instruction will use Compressed Displacement!
vaddps zmm0, zmm0, zmmword ptr [rcx+64*2]
```

The first instruction uses the displacement 123, which is not a multiple of the operand size. The operands are ZMM registers, their size is 64 bytes.

In the second instruction, the offset is 64*2, or 128, which is a multiple of the operand size. The Assembler will encode this second instruction to use the compressed displacement. We can read the machine code of these two instructions and compare to see what compressed displacement is:

```
62 F1 7C 48 58 81 7B 00 00 00      vaddps      zmm0,zmm0,zmmword ptr [rcx+7Bh]
62 F1 7C 48 58 41 02               vaddps      zmm0,zmm0,zmmword ptr [rcx+80h]
```

The bytes on the left are the machine code for the two instructions. Notice that the first instruction translates to longer machine code. The displacement, 123, is encoded as the 32 bit value 0x0000007B, in hexadecimal.

But, in the second instruction, compressed displacement is used. And the displacement 0x80, or 2x64 is encoded with a "2". In other words, compressed displacement expresses the displacement as a byte, representing a multiple of the operand size.

Compressed displacement is designed to save space in unrolled SIMD code. A common pattern in such code is as follows:

```
       ; Unrolled vector addition example
       vaddps zmm0, zmm0, zmmword ptr [rcx]
       vaddps zmm1, zmm1, zmmword ptr [rcx+64*1]
       vaddps zmm2, zmm2, zmmword ptr [rcx+64*2]
       vaddps zmm3, zmm3, zmmword ptr [rcx+64*3]
       vaddps zmm4, zmm4, zmmword ptr [rcx+64*4]
       vaddps zmm5, zmm5, zmmword ptr [rcx+64*5]
       vaddps zmm6, zmm6, zmmword ptr [rcx+64*6]
       vaddps zmm7, zmm7, zmmword ptr [rcx+64*7]
```

Here we see a heavily unrolled vector loop. There are 8 accumulators being used, ZMM0 through to ZMM7. Notice the access pattern in the final parameter. The accumulators each sum different groups of 8 packed singles. Each displacement is a consecutive multiple of 64.

The pattern is [rcx+op_size*n], where op_size is the operand size, and where n is the multiple. Compressed displacement works with 32 and 16 byte operands too – i.e. AVX and SSE vectors.

If we look at the machine code for the above instructions, we can see that compressed displacement has saved quite a lot of space in the code:

```
62 F1 7C 48 58 01    vaddps      zmm0,zmm0,zmmword ptr [rcx]
62 F1 74 48 58 49 01 vaddps      zmm1,zmm1,zmmword ptr [rcx+40h]
62 F1 6C 48 58 51 02 vaddps      zmm2,zmm2,zmmword ptr [rcx+80h]
62 F1 64 48 58 59 03 vaddps      zmm3,zmm3,zmmword ptr [rcx+0C0h]
62 F1 5C 48 58 61 04 vaddps      zmm4,zmm4,zmmword ptr [rcx+100h]
62 F1 54 48 58 69 05 vaddps      zmm5,zmm5,zmmword ptr [rcx+140h]
62 F1 4C 48 58 71 06 vaddps      zmm6,zmm6,zmmword ptr [rcx+180h]
62 F1 44 48 58 79 07 vaddps      zmm7,zmm7,zmmword ptr [rcx+1C0h]
```

The first instruction has no displacement. But for each of the other instructions, we can see the AVX512 compressed displacement bytes in the machine code on the left. The final bytes of the instructions are the compressed displacement bytes, 01, 02, 03, etc.

If compressed displacement was not used, each of the instructions (excepting the first) would have consumed 3 extra bytes in RAM – this is a total saving of 3x7, or 21 bytes.

Conclusion

There are many ways to employ AVX 512 in your own projects. We have looked at 3 different ways in this text: Agner fog's VCL, Compiler Intrinsics, and Assembly Language.

AVX 512 is a gigantic instruction set. It is complicated, flexible and extremely powerful. Not only does it contain larger vector registers, and more of them, but it also includes K masks, Zero masking, compressed displacement, rounding control and automatic broadcasting.

In this text and accompanying videos, we have explored the surface of the instruction set. We have said almost nothing of the instructions themselves! AVX 512 adds a vast number of new instructions. Many are just upgraded versions of AVX and SSE instructions, but there are also a lot of brand new instructions. It would be impossible to cover the entire instruction set in a single document or video series. But hopefully we can explore some more in upcoming adventures.

Thank you for watching and reading :)

; CREEL?

References and further Reading:

Agner Fog's VCL github: https://github.com/vectorclass/version2

VCL Manual: https://www.agner.org/optimize/vcl_manual.pdf

AVX512 Wikipedia Article: https://en.wikipedia.org/wiki/AVX-512

Intel Specifications and Programmer's References:
https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

IEEE 754 Wikipedia Article (featuring rounding modes):
https://en.wikipedia.org/wiki/IEEE_754