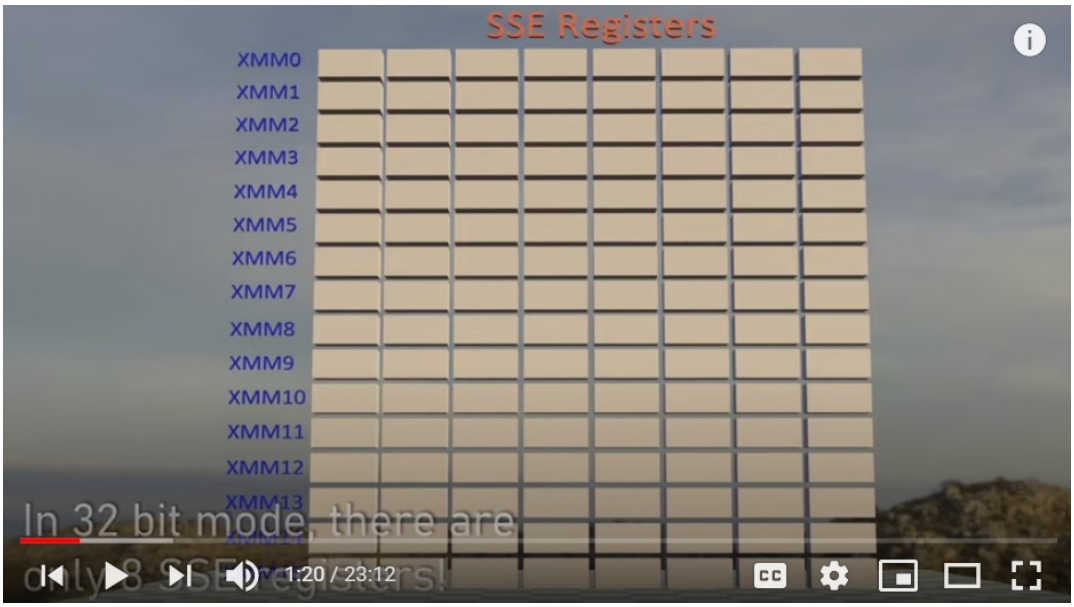

SIMD Floating-
Point Arithmetic
0:14 / 23:12


Floats and
Doubles
0:27 / 23:12


SSE and
AVX
0:32 / 23:12


SSE Registers
XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7
XMM8
XMM9
XMM10
XMM11
XMM12
XMM13
In 32 bit mode, there are
only 8 SSE registers!
1:20 / 23:12

Intrinsics Guide

The Intel Intrinsics Guide is an interactive reference tool for Intel intrinsic instructions, which are C style functions that provide access to many Intel instructions - including Intel® SSE, AVX, AVX-512, and more - without the need to write assembly code.

Technologies

- MMX
- SSE
- SSE2
- SSE3
- SSSE3
- SSE4.1
- SSE4.2
- AVX
- AVX2
- FMA
- AVX-512
- AVX-512F
- AVX-512BW
- AVX-512CD
- AVX-512DQ
- AVX-512ER

_mm_search

```
void _mm_2intersect_epi32 (__m128i a, __m128i b, __mmask8* k1, __mmask8* k2)
void _mm256_2intersect_epi32 (__m256i a, __m256i b, __mmask8* k1, __mmask8* k2)
void _mm512_2intersect_epi32 (__m512i a, __m512i b, __mmask16* k1, __mmask16* k2)
void _mm_2intersect_epi64 (__m128i a, __m128i b, __mmask8* k1, __mmask8* k2)
void _mm256_2intersect_epi64 (__m256i a, __m256i b, __mmask8* k1, __mmask8* k2)
void _mm512_2intersect_epi64 (__m512i a, __m512i b, __mmask8* k1, __mmask8* k2)
__m512i _mm512_4dpwssd_epi32 (__m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)
__m512i _mm512_mask_4dpwssd_epi32 (__m512i src, __mmask16 k, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)
__m512i _mm512_maskz_4dpwssd_epi32 (__mmask16 k, __m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)
__m512i _mm512_4dpwssds_epi32 (__m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)
__m512i _mm512_mask_4dpwssds_epi32 (__m512i src, __mmask16 k, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)
__m512i _mm512_maskz_4dpwssds_epi32 (__mmask16 k, __m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)
__m512 _mm512_4fmadd_ps (__m512 src, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)
__m512 _mm512_mask_4fmadd_ps (__m512 src, __mmask16 k, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)
```

- SSE2
- SSE3
- SSSE3
- SSE4.1
- SSE4.2
- AVX
- AVX2
- FMA
- AVX-512
- AVX-512F
- AVX-512BW
- AVX-512CD
- AVX-512DQ
- AVX-512ER
- AVX-512IFMA52
- AVX-512PF
- AVX-512VL
- AVX-512VPOPCNTDQ
- AVX-512_4FMAPS
- AVX-512_4VNNIW
- AVX-512_BF16
- AVX-512_BITALG
- AVX-512_VBMI
- AVX-512_VBMI2

```
void _mm_2intersect_epi32 (__m128
void _mm256_2intersect_epi32 (__m
void _mm512_2intersect_epi32 (__m
void _mm_2intersect_epi64 (__m128
void _mm256_2intersect_epi64 (__m
void _mm512_2intersect_epi64 (__m
__m512i _mm512_4dpwssd_epi32 (__m
__m512i _mm512_mask_4dpwssd_epi32
__m512i _mm512_maskz_4dpwssd_epi3
__m512i _mm512_4dpwssds_epi32 (__
__m512i _mm512_mask_4dpwssds_epi3
__m512i _mm512_maskz_4dpwssds_epi
__m512 _mm512_4fmadd_ps (__m512 s
__m512 _mm512_mask_4fmadd_ps (__m
__m512 _mm512_maskz_4fmadd_ps (__
__m128 _mm_4fmadd_ss (__m128 src,
__m128 _mm_mask_4fmadd_ss (__m128
__m128 _mm_maskz_4fmadd_ss (__mma
__m512 _mm512_4fnmadd_ps (__m512
__m512 _mm512_mask_4fnmadd_ps (__
__m512 _mm512_maskz_4fnmadd_ps (_
__m128 _mm_4fnmadd_ss (__m128 src
__m128 _mm_mask_4fnmadd_ss (__m12
__m128 _mm_maskz_4fnmadd_ss (__mm
__m128i _mm_abs_epi16 (__m128i a)
```

Modern x64 Assembly

Assembly language (ASM) is a generic term for a hardware language. Many devices have their own assembly languages; CPU's, micro-controllers, graphics cards, etc.

This video series is about x64 Assembly Language, the language of Intel and AMD CPU's as used in most desktops and laptops.

Visual Studio

I'll be using the free version of Visual Studio (Visual Studio Community 2017 is currently the best option). It's a free IDE from Microsoft. Includes C++, C#, and MASM.

Assembler

An Assembler is a program that translates ASM code into machine code so the CPU can execute it.

Instead of “compiling” it's supposed to be “assembling”.

The Assembler we will be using is called MASM. It is by Microsoft and included with Visual Studio. The name is short for Macro Assembler, it offers us some extra useful tools in addition to the normal ASM language.

There are many other Assemblers, GNU-Assembler, NASM etc.

Parameter Passing

Windows “C” Calling Convention

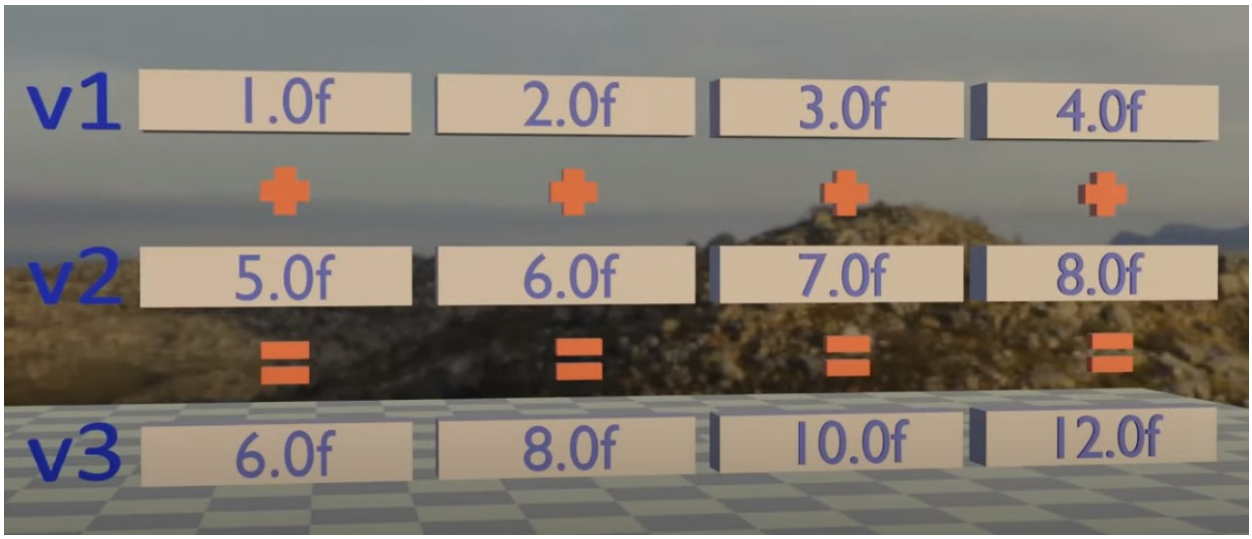
	Int	Float/ Double	Pointer/Obj/ Array
1st	RCX	XMM0	RCX
2nd	RDX	XMM1	RDX
3rd	R8	XMM2	R8
4th	R9	XMM3	R9
More	Stack	Stack	Stack

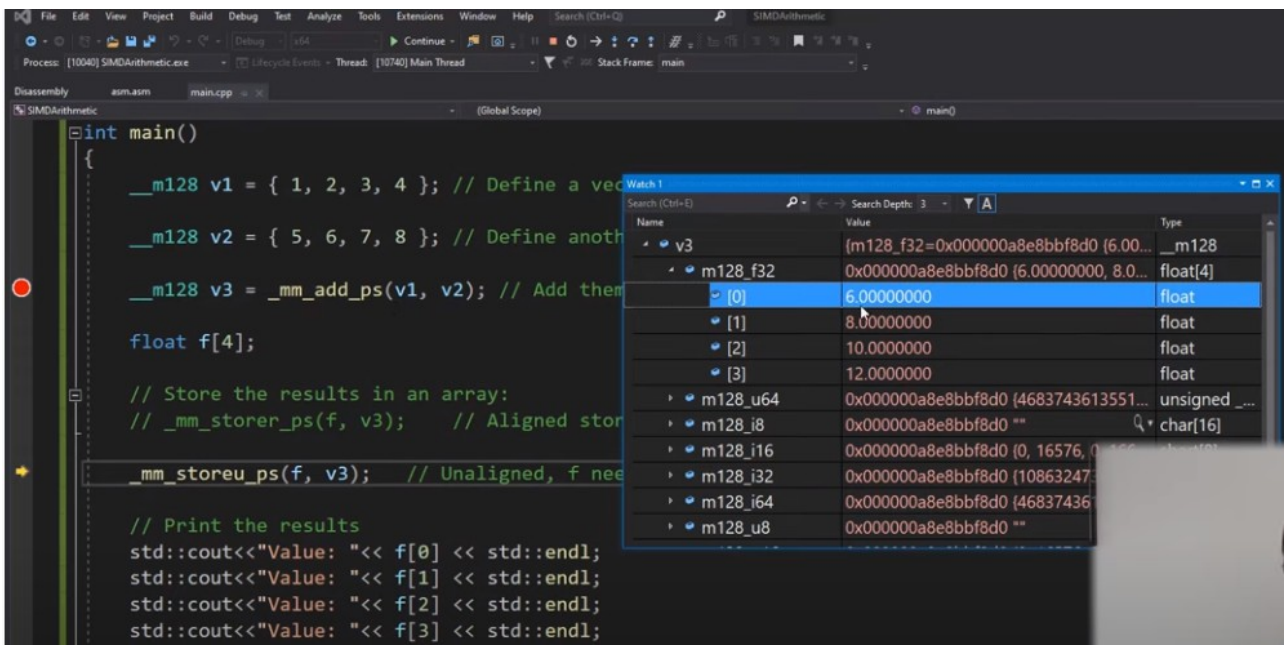
Integer and pointer returns are in RAX.
Floating point returns are in XMM0

Flynn's Taxonomy: Classification of computer architectures by Michael Flynn.

SISD: Single instruction, single data
SIMD: Single instruction, multiple data
MISD: Multiple instruction, single data
MIMD: Multiple instruction, multiple data

Note: Regular scalar programming is SISD, Multicore is MIMD, AVX512 is SIMD, and MISD is rare, sometimes used for error checking.


v1
1.0f
2.0f
3.0f
4.0f
v2
5.0f
6.0f
7.0f
8.0f
v3
6.0f
8.0f
10.0f
12.0f


int main()
{
__m128 v1 = { 1, 2, 3, 4 }; // Define a vec
__m128 v2 = { 5, 6, 7, 8 }; // Define anoth
__m128 v3 = _mm_add_ps(v1, v2); // Add them
float f[4];
// Store the results in an array:
// _mm_storer_ps(f, v3); // Aligned stor
_mm_storeu_ps(f, v3); // Unaligned, f nee
// Print the results
std::cout<<"Value: "<< f[0] << std::endl;
std::cout<<"Value: "<< f[1] << std::endl;
std::cout<<"Value: "<< f[2] << std::endl;
std::cout<<"Value: "<< f[3] << std::endl;
Watch 1
Name
Value
Type
v3
{m128_f32=0x000000a8e8bbf8d0 {6.00...
__m128
m128_f32
0x000000a8e8bbf8d0 {6.00000000, 8.0...
float[4]
[0]
6.00000000
float
[1]
8.00000000
float
[2]
10.0000000
float
[3]
12.0000000
float
m128_u64
0x000000a8e8bbf8d0 {4683743613551...
unsigned _...
m128_i8
0x000000a8e8bbf8d0 ""
char[16]
m128_i16
m128_i32
m128_i64
m128_u8
0x000000a8e8bbf8d0 ""

```
int main()
{
    __m128 v1 = { 1, 2, 3, 4 }; // Define a vector

    __m128 v2 = { 5, 6, 7, 8 }; // Define another

    __m128 v3 = _mm_add_ps(v1, v2); // Add them together using ADDPS

    float f[4];

    // Store the results in an array:
    // _mm_storer_ps(f, v3);    // Aligned store! f must be aligned to 16 bytes!

    _mm_storeu_ps(f, v3);   // Unaligned, f need not be aligned

    // Print the results
    std::cout<<"Value: "<< f[0] << std::endl;
    std::cout<<"Value: "<< f[1] << std::endl;
    std::cout<<"Value: "<< f[2] << std::endl;
    std::cout<<"Value: "<< f[3] << std::endl;
```

Disassembly mode

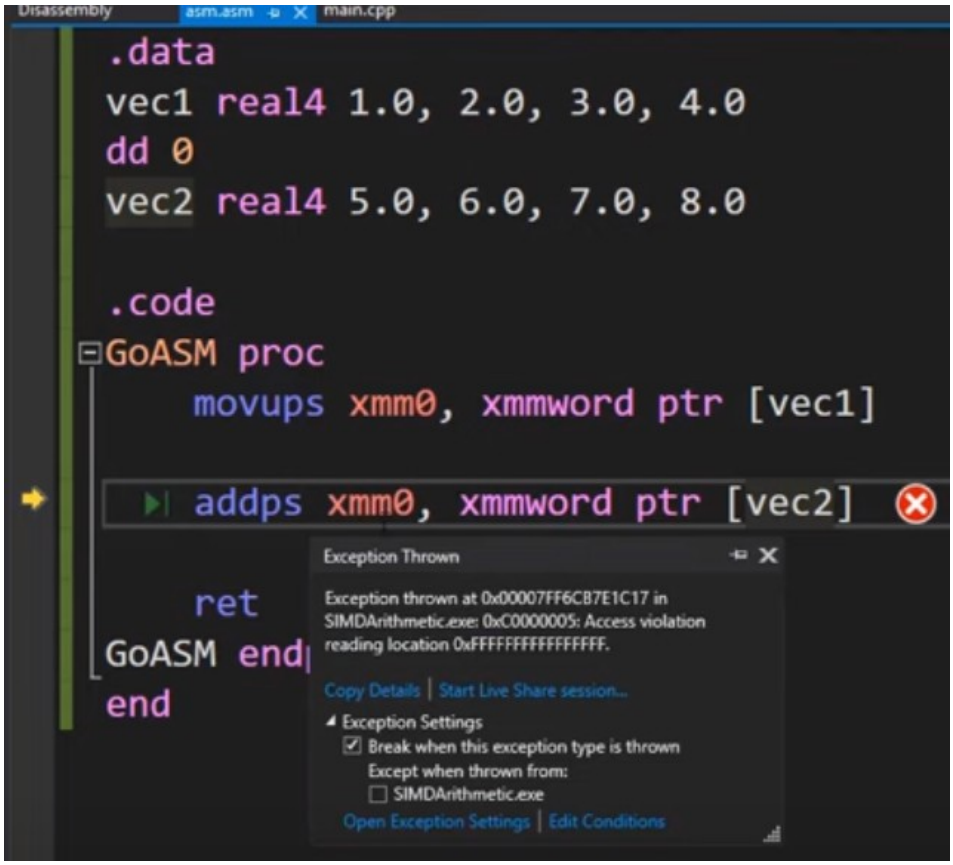
```
00007FF7FBDF23AA F3 0F 11 45 1C       movss     dword ptr [rbp+1Ch],xmm0
00007FF7FBDF23AF F3 0F 10 05 D1 88 00 00 movss     xmm0,dword ptr [__real@400
00007FF7FBDF23B7 F3 0F 11 45 40       movss     dword ptr [v2],xmm0
00007FF7FBDF23BC F3 0F 10 05 C8 88 00 00 movss     xmm0,dword ptr [__real@40c
00007FF7FBDF23C4 F3 0F 11 45 44       movss     dword ptr [rbp+44h],xmm0
00007FF7FBDF23C9 F3 0F 10 05 BF 88 00 00 movss     xmm0,dword ptr [__real@401
00007FF7FBDF23D1 F3 0F 11 45 48       movss     dword ptr [rbp+48h],xmm0
00007FF7FBDF23D6 F3 0F 10 05 B6 88 00 00 movss     xmm0,dword ptr [__real@410
00007FF7FBDF23DE F3 0F 11 45 4C       movss     dword ptr [rbp+4Ch],xmm0
00007FF7FBDF23E3 0F 28 45 10          movaps    xmm0,xmmword ptr [v1]
00007FF7FBDF23E7 0F 5E 45 40          divps     xmm0,xmmword ptr [v2]
00007FF7FBDF23EB 0F 29 85 90 01 00 00 movaps    xmmword ptr [rbp+190h],xmm0
00007FF7FBDF23F2 0F 28 85 90 01 00 00 movaps    xmm0,xmmword ptr [rbp+190h]
00007FF7FBDF23F9 0F 29 45 70          movaps    xmmword ptr [v3],xmm0
00007FF7FBDF23FD 0F 28 45 70          movaps    xmm0,xmmword ptr [v3]
00007FF7FBDF2401 0F 11 85 98 00 00 00 movups    xmmword ptr [f],xmm0
00007FF7FBDF2408 B8 04 00 00 00       mov       eax,4
00007FF7FBDF240D 48 6B C0 00          imul      rax,rax,0
00007FF7FBDF2411 48 89 85 A8 01 00 00 mov       qword ptr [rbp+1A8h],rax
```

```
.data
vec1 real4 1.0, 2.0, 3.0, 4.0
vec2 real4 5.0, 6.0, 7.0, 8.0

.code
GoASM proc
    movups xmm0, xmmword ptr [vec1]
    movups xmm1, xmmword ptr [vec2]

    addps xmm0, xmm1

    ret
GoASM endp
end
```

```
.data
vec1 real4 1.0, 2.0, 3.0, 4.0
vec2 real4 5.0, 6.0, 7.0, 8.0

.code
GoASM proc
    movups xmm0, xmmword ptr [vec1]
    movups xmm1, xmmword ptr [vec2]

    addps xmm0, xmm1

    ret
GoASM endp
end
```

Name	Value	Type
xmm0	{m128_f32= {6.00000000, 8.00000000, ...	__m128
m128_f32	{6.00000000, 8.00000000, 10.0000000, ...	float[4]
[0]	6.00000000	float
[1]	8.00000000	float
[2]	10.0000000	float
[3]	12.0000000	float
m128_f64	{131072.03161621094, 2097152.508789...	double[2]
m128_i8	""	char[16]
m128_i16	{0, 16576, 0, 16640, 0, 16672, 0, 16704}	short[8]
m128_i32	{1086324736, 1090519040, 1092616192...	int[4]
m128_i64	{4683743613551640576, 470175801206...	__int6
m128_u8	""	unsig

DESALINEADO (NO MULTIPLO DE 16): CUIDADO!!

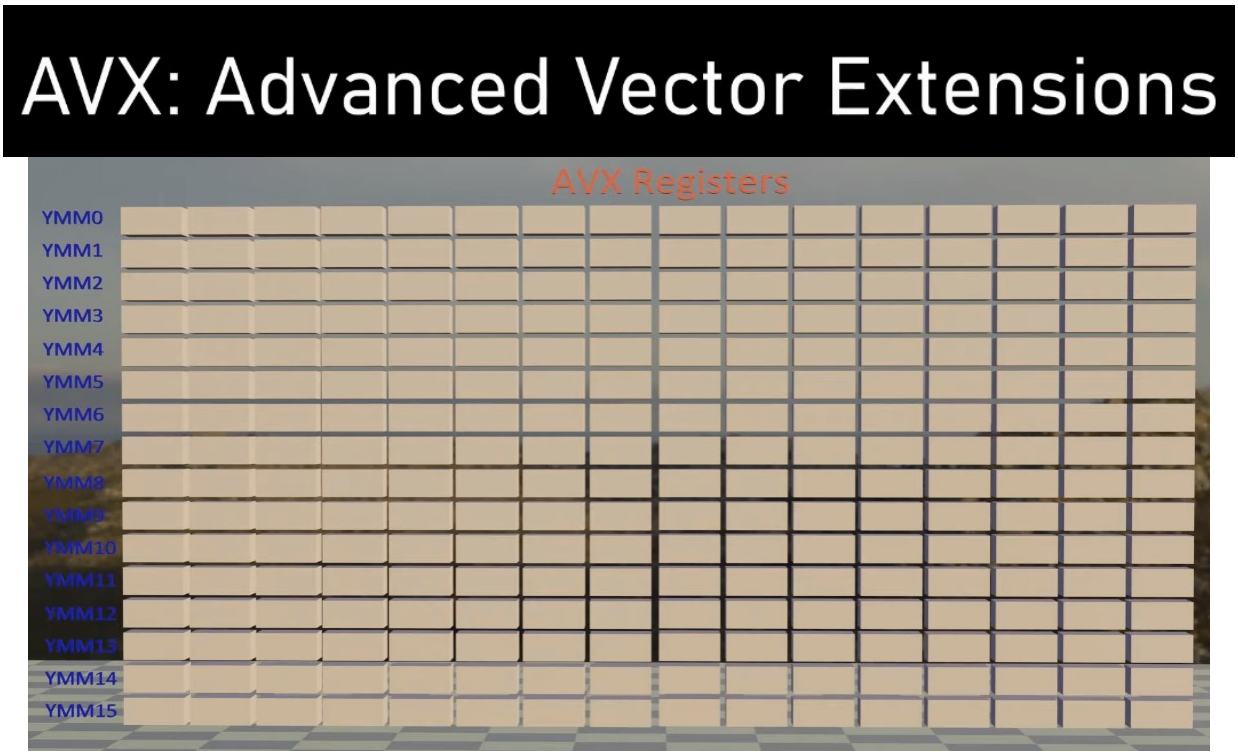



```
.data
vec1 real8 1.0, 2.0
vec2 real8 3.0, 4.0

.code
GoASM proc
    movapd xmm0, xmmword ptr [vec1]

    divpd xmm0, xmmword ptr [vec2]

    ret
GoASM endp
end
```

Intrinsics DOUBLEs

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

int main()
{
    __m128d v1 = { 1.0, 2.0 };  // Define a vector

    __m128d v2 = { 3.0, 4.0 };  // Define another

    __m128d v3 = _mm_add_pd(v1, v2);    // Add them together

    double d[2];

    _mm_storeu_pd(d, v3);   // Store results a C++ array

    // Print the results
    std::cout<<"Value: "<< d[0] << std::endl;
    std::cout<<"Value: "<< d[1] << std::endl;
```

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

int main()
{
    __m128d v1 = { 1.0, 2.0 };  // Define a v

    __m128d v2 = { 3.0, 4.0 };  // Define ano

    __m128d v3 = _mm_div_pd(v1, v2);    // Ad

    double d[2];

    _mm_storeu_pd(d, v3);   // Store results

    // Print the results
    std::cout<<"Value: "<< d[0] << std::endl;
    std::cout<<"Value: "<< d[1] << std::endl;
```



Los registros AVX YMM son los mismo SSE anteriores (XMM) extendidos

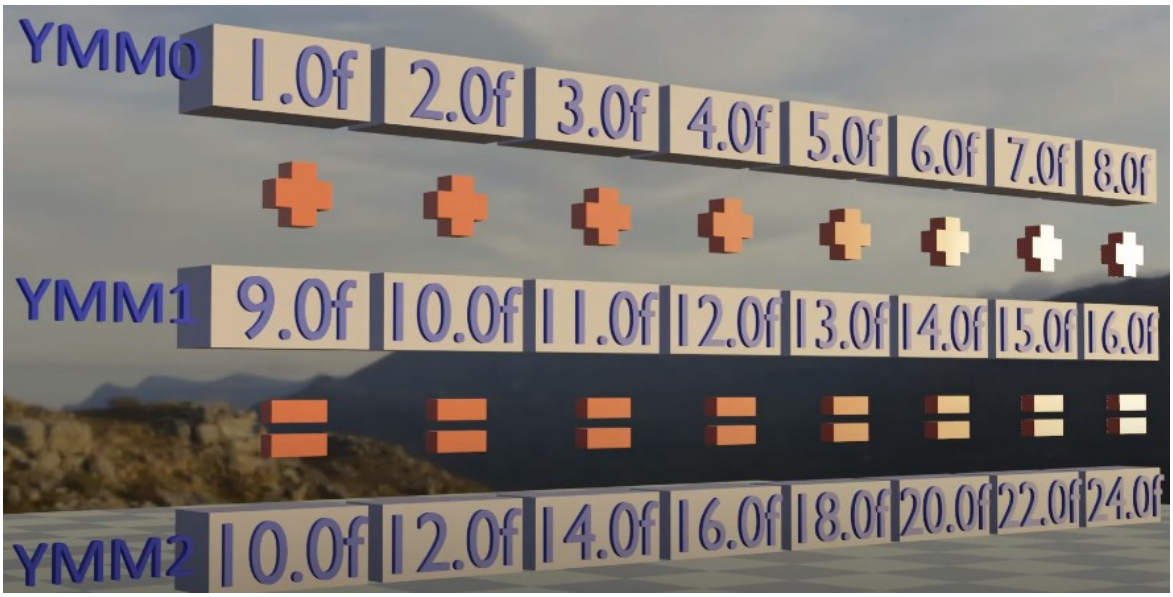
```
int main()
{
    __m256 v1 = { 1, 2, 3, 4, 5, 6, 7, 8 }; // Define a

    __m256 v2 = { 9, 10, 11, 12, 13, 14, 15, 16 };  //

    __m256 v3 = _mm256_add_ps(v1, v2);  // Add them tog

    float f[8];

    // Store the results in an array:
    // _mm_storer_ps(f, v3);    // Aligned store! f mus

    _mm256_storeu_ps(f, v3);    // Unaligned, f need no

    // Print the results
    std::cout<<"Value: "<< f[0] << std::endl;
    std::cout<<"Value: "<< f[1] << std::endl;
    std::cout<<"Value: "<< f[2] << std::endl;
    std::cout<<"Value: "<< f[3] << std::endl;
```

Name	Value
v3	{m256_f32=0x
m256_f32	0x000000a971
[0]	10.0000000
[1]	12.0000000
[2]	14.0000000
[3]	16.0000000
[4]	18.0000000
[5]	20.0000000
[6]	22.0000000
[7]	24.0000000
Add item to watch	

Ahora C++ llama a ASM:

```cpp
extern "C" void GoASM();

int main()
{
    GoASM();

    return 0;
}
```

```asm
.data
vec1 real4 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0
vec2 real4 9.0,10.0,11.0,12.0,13.0,14.0,15.0,16.0

.code
GoASM proc
    vmovups ymm0, ymmword ptr [vec1]
    vmovups ymm1, ymmword ptr [vec2]

    vaddps ymm2, ymm0, ymm1

    ret
GoASM endp
end
```



```
.data
vec1 real4 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0
vec2 real4 9.0,10.0,11.0,12.0,13.0,14.0,15.0,16.0

.code
GoASM proc
    vmovups ymm0, ymmword pt
    vmovups ymm1, ymmword pt

    vaddps ymm2, ymm0, ymm1

    ret  ≤1ms elapsed
GoASM endp
end
```

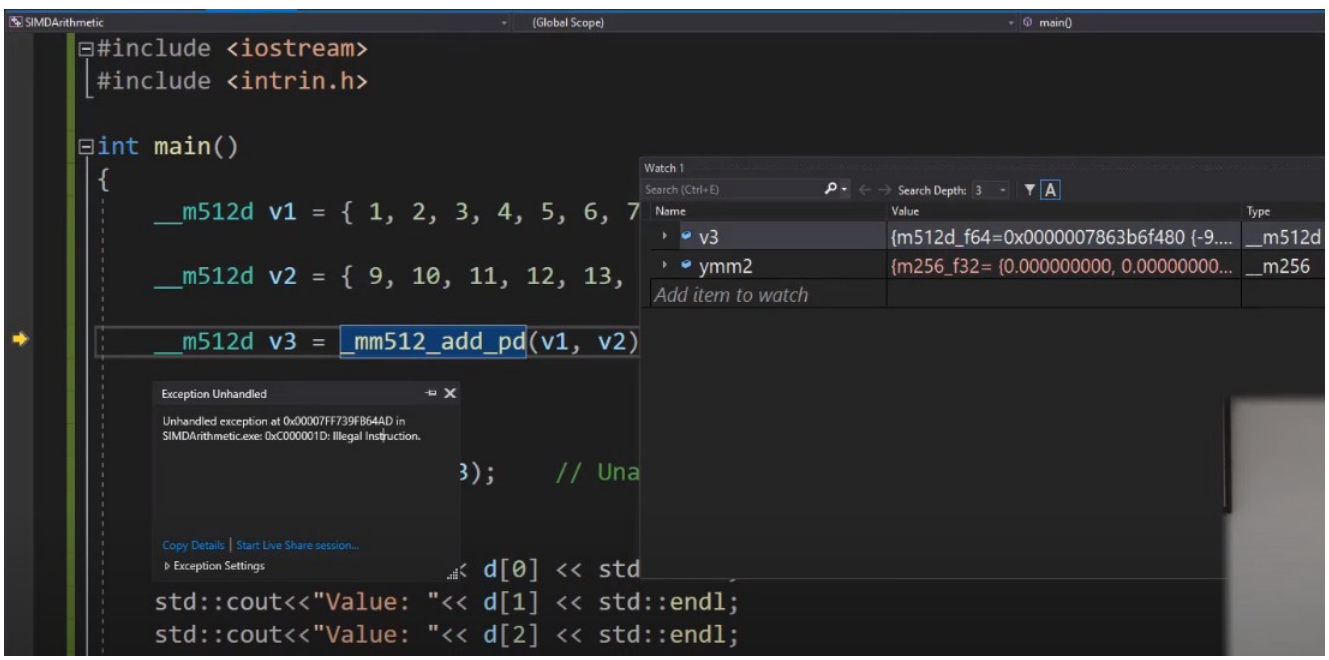
Watch 1

Search (Ctrl+E) Search Depth: 3

Name	Value	Type
v3	identifier "v3" is undefined	
ymm2	{m256_f32= {10.0000000, 12.0000000, ...	__m256
m256_f32	{10.0000000, 12.0000000, 14.0000000, ...	float[8]
[0]	10.0000000	float
[1]	12.0000000	float
[2]	14.0000000	float
[3]	16.0000000	float
[4]	18.0000000	float
[5]	20.0000000	float
[6]	22.0000000	float
[7]	24.0000000	float
m256_f64	{2097152.5087890625, 33554440.17187...	double

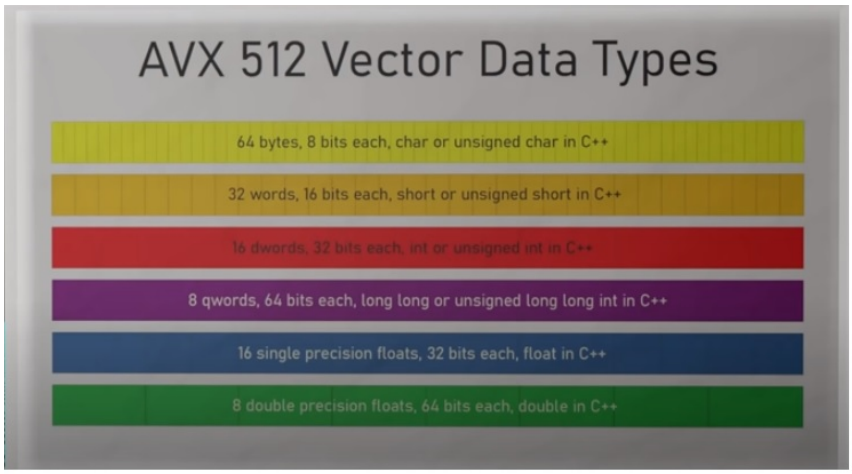
CPUID is an instruction which determines the hardware capabilities at runtime.

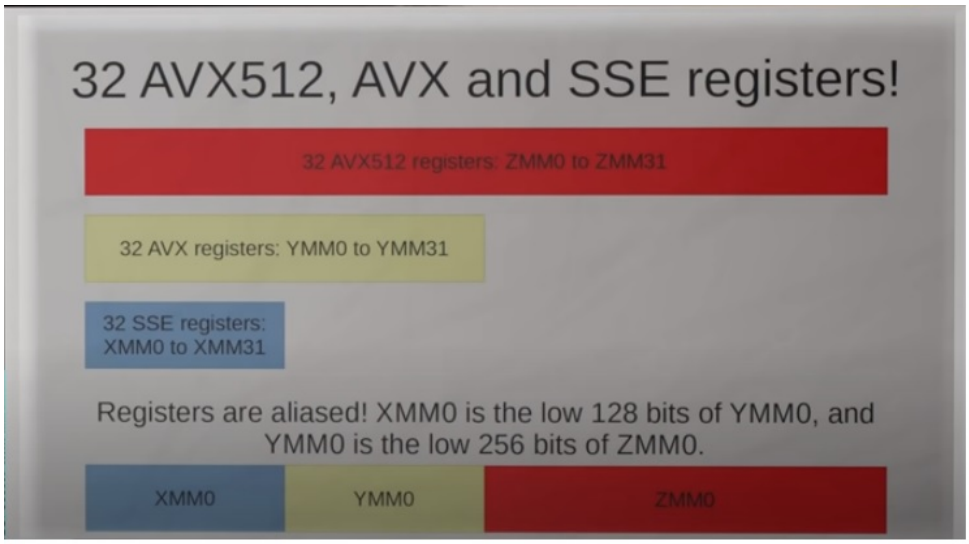
We assumed SSE, SSE2 and AVX today, but it's usually safer to check the hardware with a CPUID call!



```cpp
#include <iostream>

extern "C" void AVX512Test(double* c, double* a, double* b);

extern "C" bool AVXFoundationDetection();

int main()
{
    if (AVXFoundationDetection())
        std::cout << "This CPU is capabale of ZVX512 Fondation instruction set!" << std::endl;
    else
        std::cout << "Nope" << std::endl;
```

```asm
.code
AVXFoundationDetection proc
    push rbx

    mov eax, 7
    mov ecx, 0

    cpuid

    shr ebx, 16
    and ebx, 1

    mov eax, ebx

    pop rbx
    ret
AVXFoundationDetection endp
```


AVX 512 Vector Data Types
64 bytes, 8 bits each, char or unsigned char in C++
32 words, 16 bits each, short or unsigned short in C++
16 dwords, 32 bits each, int or unsigned int in C++
8 qwords, 64 bits each, long long or unsigned long long int in C++
16 single precision floats, 32 bits each, float in C++
8 double precision floats, 64 bits each, double in C++


32 AVX512, AVX and SSE registers!
32 AVX512 registers: ZMM0 to ZMM31
32 AVX registers: YMM0 to YMM31
32 SSE registers:
XMM0 to XMM31
Registers are aliased! XMM0 is the low 128 bits of YMM0, and
YMM0 is the low 256 bits of ZMM0.
XMM0
YMM0
ZMM0

AVX512 Way

In AVX512, we can read and broadcast a scalar value in the Addition instruction!!

We do not need to use a separate broadcast instruction and we save a register!

3.14 is a scalar, it's just a float. When we use AX512 broadcasting, it is copied to all elements of a temporary register.

3.14	3.14	3.14	3.14	3.14	3.14	3.14	3.14	3.14	3.14	3.14	3.14	3.14	3.14	3.14	3.14
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
2.75	-5.4	92.1	3.50	0.12	-2.13	78.4	7.42	98.2	28.4	0.42	-6.7	84.3	28.4	-1.65	8.2