

HOMEWORK ON SIMPLE VECTOR OPERATIONS AND CPUID

CSC 343 Fall 2017

NOVEMBER, 13 2017

Jeter Gutierrez

TABLE OF CONTENTS**Objective PG. 2****CPUID TEST PG. 2-10****Simple Four-Iteration Loop PG. 10-14****Streaming SIMD Extensions Using Inlined Assembly Encoding. PG. 14-15****Simple Four-Iteration Loop coded with intrinsics. PG. 15-19****Accessing Arrays Using the Vector Class PG. 19-22****Conclusion PG. 22**

Objective: The purpose of this homework assignment is to use C++ code on Visual Studio in order to identify what kinds of advanced SIMD operations among other operations my computer is capable of running and then performing some tests using those operations. I will be using code that was found online that will test CPUID X86 and identify what features my intel computer supports. Operations like SSE AVX and MMX just to name a few. After I run this program and identify what operations my processor is capable of performing I will run 4 examples in order to test the efficiency of interacting with arrays using different methods. We will be performing operations on 3 different arrays that each have 12 elements. We will test the following:

1. Simple Four-iteration Loop.
2. Streaming SIMD Extensions Using Inlined Assembly Encoding.
3. Simple Four-Iteration Loop coded with intrinsics.
4. Accessing Arrays using the Vector Class.

CPUID TEST

```

#include <iostream>
#include <vector>
#include <bitset>
#include <array>
#include <string>
#include <intrin.h>

class InstructionSet
{
    // forward declarations
    class InstructionSet_Internal;

public:
    // getters
    static std::string Vendor(void) { return CPU_Rep.vendor_; }
    static std::string Brand(void) { return CPU_Rep.brand_; }

    static bool SSE3(void) { return CPU_Rep.f_1_ECX_[0]; }
    static bool PCLMULQDQ(void) { return CPU_Rep.f_1_ECX_[1]; }
    static bool MONITOR(void) { return CPU_Rep.f_1_ECX_[3]; }
    static bool SSSE3(void) { return CPU_Rep.f_1_ECX_[9]; }
    static bool FMA(void) { return CPU_Rep.f_1_ECX_[12]; }
    static bool CMPXCHG16B(void) { return CPU_Rep.f_1_ECX_[13]; }
    static bool SSE41(void) { return CPU_Rep.f_1_ECX_[19]; }
    static bool SSE42(void) { return CPU_Rep.f_1_ECX_[20]; }

```

Figure 1: C++ CODE FOR CALLING CPUID AND DETECTING SUPPORTED OPERATIONS ON MY COMPUTER PART 1.

```

    static bool MOVBE(void) { return CPU_Rep.f_1_ECX_[22]; }
    static bool POPCNT(void) { return CPU_Rep.f_1_ECX_[23]; }
    static bool AES(void) { return CPU_Rep.f_1_ECX_[25]; }
    static bool XSAVE(void) { return CPU_Rep.f_1_ECX_[26]; }
    static bool OSXSAVE(void) { return CPU_Rep.f_1_ECX_[27]; }
    static bool AVX(void) { return CPU_Rep.f_1_ECX_[28]; }
    static bool F16C(void) { return CPU_Rep.f_1_ECX_[29]; }
    static bool RDRAND(void) { return CPU_Rep.f_1_ECX_[30]; }

    static bool MSR(void) { return CPU_Rep.f_1_EDX_[5]; }
    static bool CX8(void) { return CPU_Rep.f_1_EDX_[8]; }
    static bool SEP(void) { return CPU_Rep.f_1_EDX_[11]; }
    static bool CMOV(void) { return CPU_Rep.f_1_EDX_[15]; }
    static bool CLFSH(void) { return CPU_Rep.f_1_EDX_[19]; }
    static bool MMX(void) { return CPU_Rep.f_1_EDX_[23]; }
    static bool FXSR(void) { return CPU_Rep.f_1_EDX_[24]; }
    static bool SSE(void) { return CPU_Rep.f_1_EDX_[25]; }
    static bool SSE2(void) { return CPU_Rep.f_1_EDX_[26]; }

    static bool FSGSBASE(void) { return CPU_Rep.f_7_EBX_[0]; }
    static bool BMI1(void) { return CPU_Rep.f_7_EBX_[3]; }
    static bool HLE(void) { return CPU_Rep.isIntel_ && CPU_Rep.f_7_EBX_[4]; }
    static bool AVX2(void) { return CPU_Rep.f_7_EBX_[5]; }
    static bool BMI2(void) { return CPU_Rep.f_7_EBX_[8]; }
    static bool ERMS(void) { return CPU_Rep.f_7_EBX_[9]; }
    static bool INVPCID(void) { return CPU_Rep.f_7_EBX_[10]; }
    static bool RTM(void) { return CPU_Rep.isIntel_ && CPU_Rep.f_7_EBX_[11]; }
    static bool AVX512F(void) { return CPU_Rep.f_7_EBX_[16]; }
    static bool RDSEED(void) { return CPU_Rep.f_7_EBX_[18]; }
    static bool ADX(void) { return CPU_Rep.f_7_EBX_[19]; }
    static bool AVX512PF(void) { return CPU_Rep.f_7_EBX_[26]; }
    static bool AVX512ER(void) { return CPU_Rep.f_7_EBX_[27]; }
    static bool AVX512CD(void) { return CPU_Rep.f_7_EBX_[28]; }
    static bool SHA(void) { return CPU_Rep.f_7_EBX_[29]; }

```

Figure 2: C++ CODE FOR CALLING CPUID AND DETECTING SUPPORTED OPERATIONS ON MY COMPUTER PART 2.

```

static bool PREFETCHWT1(void) { return CPU_Rep.f_7_ECX[0]; }

static bool LAHF(void) { return CPU_Rep.f_81_ECX[0]; }
static bool LZCNT(void) { return CPU_Rep.isIntel_ && CPU_Rep.f_81_ECX[5]; }
static bool ABM(void) { return CPU_Rep.isAMD_ && CPU_Rep.f_81_ECX[5]; }
static bool SSE4a(void) { return CPU_Rep.isAMD_ && CPU_Rep.f_81_ECX[6]; }
static bool XOP(void) { return CPU_Rep.isAMD_ && CPU_Rep.f_81_ECX[11]; }
static bool TBM(void) { return CPU_Rep.isAMD_ && CPU_Rep.f_81_ECX[21]; }

static bool SYSCALL(void) { return CPU_Rep.isIntel_ && CPU_Rep.f_81_EDX[11]; }
static bool MMXEXT(void) { return CPU_Rep.isAMD_ && CPU_Rep.f_81_EDX[22]; }
static bool RDTSCP(void) { return CPU_Rep.isIntel_ && CPU_Rep.f_81_EDX[27]; }
static bool _3DNOWEXT(void) { return CPU_Rep.isAMD_ && CPU_Rep.f_81_EDX[30]; }
static bool _3DNOW(void) { return CPU_Rep.isAMD_ && CPU_Rep.f_81_EDX[31]; }

private:
    static const InstructionSet_Internal CPU_Rep;

class InstructionSet_Internal
{
public:
    InstructionSet_Internal()
        : nIds_{ 0 },
          nExIds_{ 0 },
          isIntel_{ false },
          isAMD_{ false },
          f_1_ECX{ 0 },
          f_1_EDX{ 0 },
          f_7_EBX{ 0 },
          f_7_ECX{ 0 },
          f_81_ECX{ 0 },
          f_81_EDX{ 0 },
          data_(),
          extdata_()
    {}
};

```

Figure 3: C++ CODE FOR CALLING CPUID AND DETECTING SUPPORTED OPERATIONS ON MY COMPUTER PART 3.

```

{
    //int cpuInfo[4] = {-1};
    std::array<int, 4> cpui;

    // Calling __cpuid with 0x0 as the function_id argument
    // gets the number of the highest valid function ID.
    __cpuid(cpui.data(), 0);
    nIds_ = cpui[0];

    for (int i = 0; i <= nIds_; ++i)
    {
        __cpuidex(cpui.data(), i, 0);
        data_.push_back(cpui);
    }

    // Capture vendor string
    char vendor[0x20];
    memset(vendor, 0, sizeof(vendor));
    *reinterpret_cast<int*>(vendor) = data_[0][1];
    *reinterpret_cast<int*>(vendor + 4) = data_[0][3];
    *reinterpret_cast<int*>(vendor + 8) = data_[0][2];
    vendor_ = vendor;
    if (vendor_ == "GenuineIntel")
    {
        isIntel_ = true;
    }
    else if (vendor_ == "AuthenticAMD")
    {
        isAMD_ = true;
    }
}

```

Figure 4: C++ CODE FOR CALLING CPUID AND DETECTING SUPPORTED OPERATIONS ON MY COMPUTER PART 4.

```
// load bitset with flags for function 0x00000001
if (nIds_ >= 1)
{
    f_1_ECX_ = data_[1][2];
    f_1_EDX_ = data_[1][3];
}

// load bitset with flags for function 0x00000007
if (nIds_ >= 7)
{
    f_7_EBX_ = data_[7][1];
    f_7_ECX_ = data_[7][2];
}

// Calling __cpuid with 0x80000000 as the function_id argument
// gets the number of the highest valid extended ID.
__cpuid(cpui.data(), 0x80000000);
nExIds_ = cpui[0];

char brand[0x40];
memset(brand, 0, sizeof(brand));

for (int i = 0x80000000; i <= nExIds_; ++i)
{
    __cpuidex(cpui.data(), i, 0);
    extdata_.push_back(cpui);
}

// load bitset with flags for function 0x80000001
if (nExIds_ >= 0x80000001)
{
    f_81_ECX_ = extdata_[1][2];
    f_81_EDX_ = extdata_[1][3];
}
```

Figure 5: C++ CODE FOR CALLING CPUID AND DETECTING SUPPORTED OPERATIONS ON MY COMPUTER PART 5.

```
// Interpret CPU brand string if reported
if (nExIds_ >= 0x80000004)
{
    memcpy(brand, extdata_[2].data(), sizeof(cpu));
    memcpy(brand + 16, extdata_[3].data(), sizeof(cpu));
    memcpy(brand + 32, extdata_[4].data(), sizeof(cpu));
    brand_ = brand;
}

};

int nIds_;
int nExIds_;
std::string vendor_;
std::string brand_;
bool isIntel_;
bool isAMD_;
std::bitset<32> f_1_ECX_;
std::bitset<32> f_1_EDX_;
std::bitset<32> f_7_EBX_;
std::bitset<32> f_7_ECX_;
std::bitset<32> f_81_ECX_;
std::bitset<32> f_81_EDX_;
std::vector<std::array<int, 4>> data_;
std::vector<std::array<int, 4>> extdata_;

};

// Initialize static member data
const InstructionSet::InstructionSet_Internal InstructionSet::CPU_Rep;

// Print out supported instruction set extensions
```

Figure 6: C++ CODE FOR CALLING CPUID AND DETECTING SUPPORTED OPERATIONS ON MY COMPUTER PART 6.

```

int main()
{
    auto& outstream = std::cout;

    auto support_message = [&outstream](std::string isa_feature, bool is_supported) {
        outstream << isa_feature << (is_supported ? " supported" : " not supported") << std::endl;
    };

    std::cout << InstructionSet::Vendor() << std::endl;
    std::cout << InstructionSet::Brand() << std::endl;

    support_message("3DNOW", InstructionSet::_3DNOW());
    support_message("3DNOWEXT", InstructionSet::_3DNOWEXT());
    support_message("ABM", InstructionSet::ABM());
    support_message("ADX", InstructionSet::ADX());
    support_message("AES", InstructionSet::AES());
    support_message("AVX", InstructionSet::AVX());
    support_message("AVX2", InstructionSet::AVX2());
    support_message("AVX512CD", InstructionSet::AVX512CD());
    support_message("AVX512ER", InstructionSet::AVX512ER());
    support_message("AVX512F", InstructionSet::AVX512F());
    support_message("AVX512PF", InstructionSet::AVX512PF());
    support_message("BMI1", InstructionSet::BMI1());
    support_message("BMI2", InstructionSet::BMI2());
    support_message("CLFSH", InstructionSet::CLFSH());
    support_message("CMPXCHG16B", InstructionSet::CMPXCHG16B());
    support_message("CX8", InstructionSet::CX8());
    support_message("ERMS", InstructionSet::ERMS());
    support_message("F16C", InstructionSet::F16C());
    support_message("FMA", InstructionSet::FMA());
    support_message("FSGSBASE", InstructionSet::FSGSBASE());
    support_message("FXSR", InstructionSet::FXSR());
    support_message("HLE", InstructionSet::HLE());
    support_message("INVPCID", InstructionSet::INVPCID());
    support_message("LAHF", InstructionSet::LAHF());
}

```

Figure 7: C++ CODE FOR CALLING CPUID AND DETECTING SUPPORTED OPERATIONS ON MY COMPUTER PART 7.


```

support_message("LZCNT", InstructionSet::LZCNT());
support_message("MMX", InstructionSet::MMX());
support_message("MMXEXT", InstructionSet::MMXEXT());
support_message("MONITOR", InstructionSet::MONITOR());
support_message("MOVBE", InstructionSet::MOVBE());
support_message("MSR", InstructionSet::MSR());
support_message("OSXSAVE", InstructionSet::OSXSAVE());
support_message("PCLMULQDQ", InstructionSet::PCLMULQDQ());
support_message("POPCNT", InstructionSet::POPCNT());
support_message("PREFETCHWT1", InstructionSet::PREFETCHWT1());
support_message("RDRAND", InstructionSet::RDRAND());
support_message("RDSEED", InstructionSet::RDSEED());
support_message("RDTSCP", InstructionSet::RDTSCP());
support_message("RTM", InstructionSet::RTM());
support_message("SEP", InstructionSet::SEP());
support_message("SHA", InstructionSet::SHA());
support_message("SSE", InstructionSet::SSE());
support_message("SSE2", InstructionSet::SSE2());
support_message("SSE3", InstructionSet::SSE3());
support_message("SSE4.1", InstructionSet::SSE41());
support_message("SSE4.2", InstructionSet::SSE42());
support_message("SSE4a", InstructionSet::SSE4a());
support_message("SSSE3", InstructionSet::SSSE3());
support_message("SYSCALL", InstructionSet::SYSCALL());
support_message("TBM", InstructionSet::TBM());
support_message("XOP", InstructionSet::XOP());
support_message("XSAVE", InstructionSet::XSAVE());
}

```

Figure 8: C++ CODE FOR CALLING CPUID AND DETECTING SUPPORTED OPERATIONS ON MY COMPUTER PART 8.

The previous code has been used to use the libraries that are stored in Intel to detect the advanced operations that the specific processor is capable of performing. The output of running this file will return whether or not the processor is capable of performing a given instruction from the instruction set class which is how we are checking which operations the processor I am using can perform so that we can use them for further testing and for implementation for future programs.


```
GenuineIntel
Intel(R) Core(TM) i7-5600U CPU @ 2.60GHz
3DNOW not supported
3DNOWEXT not supported
ABM not supported
ADX supported
AES supported
AUX supported
AUX2 supported
AUX512CD not supported
AUX512ER not supported
AUX512F not supported
AUX512PF not supported
BMI1 supported
BMI2 supported
CLFSH supported
CMPXCHG16B supported
CX8 supported
ERMS supported
F16C supported
FMA supported
FSGSBASE supported
FXSR supported
HLE supported
INUPCID supported
LAHF supported
LZCNT supported
MMX supported
MMXEXT not supported
MONITOR supported
MOVB supported
MSR supported
OSXSAVE supported
PCLMULQDQ supported
POPCNT supported
PREFETCHWT1 not supported
RDRAND supported
RDSEED supported
RDTSCP supported
RTM supported
SEP supported
SHA not supported
SSE supported
SSE2 supported
SSE3 supported
SSE4.1 supported
SSE4.2 supported
SSE4a not supported
SSSE3 supported
```

Figure 9: OUTPUT OF SUPPORTED OPERATIONS ON MY INTEL I7 PROCESSOR PART

1.

```

SSE4.1 supported
SSE4.2 supported
SSE4a not supported
SSSE3 supported
SYSCALL not supported
TBM not supported
XOP not supported
XSAVE supported

```

Figure 10: OUTPUT OF SUPPORTED OPERATIONS ON MY INTEL I7 PROCESSOR PART

2.

Based on the output of running the file to detect what CPUID my computer is capable of performing enough of the advanced operations that we need in order to test our four examples and so we will now test the 4 examples and observe the registers that are being used. We will continue with the four examples now.

Simple Four-iteration Loop.

```

#include "stdafx.h"
void add(float *a, float *b, float *c)
{
    int i;
    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}

int main()
{
    float a[12];
    for (int i = 0; i < 12; i++) {
        a[i] = i;
    }
    float b[12];
    for (int i = 0; i < 12; i++) {
        b[i] = i;
    }
    float c[12];
    for (int i = 0; i < 12; i++) {
        c[i] = i;
    }
    add(a, b, c);
    return 0;
}

```

Figure 11: C++ code for simple four-iteration loop.

We are using 3 float type arrays, each has 12 elements then we are adding the elements from a and b and storing them in c. We are using array index to access the elements in the array, we will consider runtime in comparison to the other 3 examples and we will observe which of the registers are used to perform the accessing, and the adding of the elements in the arrays.

```

int main()
{
00151750  push      ebp
00151751  mov       ebp,esp
00151753  sub       esp,190h
00151759  push      ebx
0015175A  push      esi
0015175B  push      edi
0015175C  lea       edi,[ebp-190h]
00151762  mov       ecx,64h
00151767  mov       eax,0CCCCCCCCh
0015176C  rep stos  dword ptr es:[edi]
0015176E  mov       eax,dword ptr [__security_cookie (015A000h)]
00151773  xor       eax,ebp
00151775  mov       dword ptr [ebp-4],eax

    float a[12];
    for (int i = 0; i < 12; i++) {
00151778  mov       dword ptr [ebp-44h],0
0015177F  jmp       main+3Ah (015178Ah)
00151781  mov       eax,dword ptr [ebp-44h]
00151784  add       eax,1
00151787  mov       dword ptr [ebp-44h],eax
0015178A  cmp       dword ptr [ebp-44h],0Ch
0015178E  jge       main+50h (01517A0h)

        a[i] = i;
00151790  cvtsi2ss  xmm0,dword ptr [ebp-44h]
00151795  mov       eax,dword ptr [ebp-44h]
00151798  movss     dword ptr a[eax*4],xmm0
    }
0015179E  jmp       main+31h (0151781h)

    float b[12];

```

Figure 12: DISASSEMBLY CODE FOR FOUR-ITERATION LOOP PART 1.

```

    for (int i = 0; i < 12; i++) {
001517A0  mov     dword ptr [ebp-88h],0
001517AA  jmp     main+6Bh (01517BBh)
001517AC  mov     eax,dword ptr [ebp-88h]
001517B2  add     eax,1
001517B5  mov     dword ptr [ebp-88h],eax
001517BB  cmp     dword ptr [ebp-88h],0Ch
001517C2  jge     main+8Ah (01517DAh)
        b[i] = i;
001517C4  cvtsi2ss  xmm0,dword ptr [ebp-88h]
001517CC  mov     eax,dword ptr [ebp-88h]
001517D2  movss    dword ptr b[eax*4],xmm0
    }
001517D8  jmp     main+5Ch (01517ACh)
    float c[12];
    for (int i = 0; i < 12; i++) {
001517DA  mov     dword ptr [ebp-0CCh],0
001517E4  jmp     main+0A5h (01517F5h)
001517E6  mov     eax,dword ptr [ebp-0CCh]
001517EC  add     eax,1
001517EF  mov     dword ptr [ebp-0CCh],eax
001517F5  cmp     dword ptr [ebp-0CCh],0Ch
001517FC  jge     main+0C7h (0151817h)
        c[i] = i;
001517FE  cvtsi2ss  xmm0,dword ptr [ebp-0CCh]
00151806  mov     eax,dword ptr [ebp-0CCh]
0015180C  movss    dword ptr c[eax*4],xmm0
    }
00151815  jmp     main+96h (01517E6h)
    add(a, b, c);

```

Figure 13: DISASSEMBLY CODE FOR FOUR-ITERATION LOOP PART 2.

```

00151817  lea     eax,[c]
0015181D  push    eax
    add(a, b, c);
0015181E  lea     ecx,[b]
00151821  push    ecx
00151822  lea     edx,[a]
00151825  push    edx
00151826  call    add (015127Bh)
0015182B  add     esp,0Ch
    return 0;
0015182E  xor     eax,eax
}

```

Figure 14: DISASSEMBLY CODE FOR FOUR-ITERATION LOOP PART 3.

In the disassembly code we can see that when the add function is called first it adds the parameters into the stack from right to left and then it stores the address of the function. When it calls the function it jumps to the function and it uses the addresses of the arrays directly to

modify and to add. Then it reads from the updated addresses when the arrays are updated in the add function.

```
EAX = 0000000C EBX = 7E72F000 ECX = 00000000 EDX = 0015A590 ESI = 00151055 EDI = 00CAF8F4
EIP = 00151817 ESP = 00CAF758 EBP = 00CAF8F4 EFL = 00000246

CS = 0023 DS = 002B ES = 002B SS = 002B FS = 0053 GS = 002B

ST0 = +0.0000000000000000e+0000 ST1 = +0.0000000000000000e+0000 ST2 = +0.0000000000000000e
+0000 ST3 = +0.0000000000000000e+0000 ST4 = +0.0000000000000000e+0000
ST5 = +0.0000000000000000e+0000 ST6 = +0.0000000000000000e+0000 ST7 = +0.0000000000000000e
+0000

MM0 = 0000000000000000 MM1 = 0000000000000000 MM2 = 0000000000000000 MM3 = 0000000000000000
MM4 = 0000000000000000 MM5 = 0000000000000000 MM6 = 0000000000000000 MM7 = 0000000000000000

XMM0 = 0000000000000000-0000000041300000
XMM1 = 0000000000000000-0000000000000000
XMM2 = 0000000000000000-0000000000000000
XMM3 = 0000000000000000-0000000000000000
XMM4 = 0000000000000000-0000000000000000
XMM5 = 0000000000000000-0000000000000000
XMM6 = 0000000000000000-0000000000000000
XMM7 = 0000000000000000-0000000000000000
MXCSR = 00001F80

YMM0 = 0000000000000000-0000000000000000-0000000000000000-0000000041300000 |
YMM1 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM2 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM3 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM4 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM5 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM6 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM7 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
```

Figure 15: Registers for Four-iteration loop.

The registers that are newly being used that we have not encountered before are MXCSR and XMM0 and YMM0 which are being used because they can store much higher values and because we are using arrays. Using array index and taking an offset from an address in order to find the value that we are looking for that we are going to be adding that is located in an array. Unlike when we are adding integers together that are not stored in array, performing array arithmetic can not be done simply with eax, ecx and edx in this case, at least not with this implementation which is why the registers that can stores such larger values are being used in this case.


```

Registers
EAX = 00000002 EBX = 7E9B9000 ECX = 0075FC8C EDX = 00000002 ESI = 0075FCD0 EDI = 0075FB9C
EIP = 00151727 ESP = 0075FAC4 EBP = 0075FB9C EFL = 00000293

CS = 0023 DS = 002B ES = 002B SS = 002B FS = 0053 GS = 002B

ST0 = +0.0000000000000000e+0000 ST1 = +0.0000000000000000e+0000 ST2 = +0.0000000000000000e
+0000 ST3 = +0.0000000000000000e+0000 ST4 = +0.0000000000000000e+0000
ST5 = +0.0000000000000000e+0000 ST6 = +0.0000000000000000e+0000 ST7 = +0.0000000000000000e
+0000

MM0 = 0000000000000000 MM1 = 0000000000000000 MM2 = 0000000000000000 MM3 = 0000000000000000
MM4 = 0000000000000000 MM5 = 0000000000000000 MM6 = 0000000000000000 MM7 = 0000000000000000

XMM0 = 0000000000000000-0000000040800000
XMM1 = 0000000000000000-0000000000000000
XMM2 = 0000000000000000-0000000000000000
XMM3 = 0000000000000000-0000000000000000
XMM4 = 0000000000000000-0000000000000000
XMM5 = 0000000000000000-0000000000000000
XMM6 = 0000000000000000-0000000000000000
XMM7 = 0000000000000000-0000000000000000
MXCSR = 00001F80

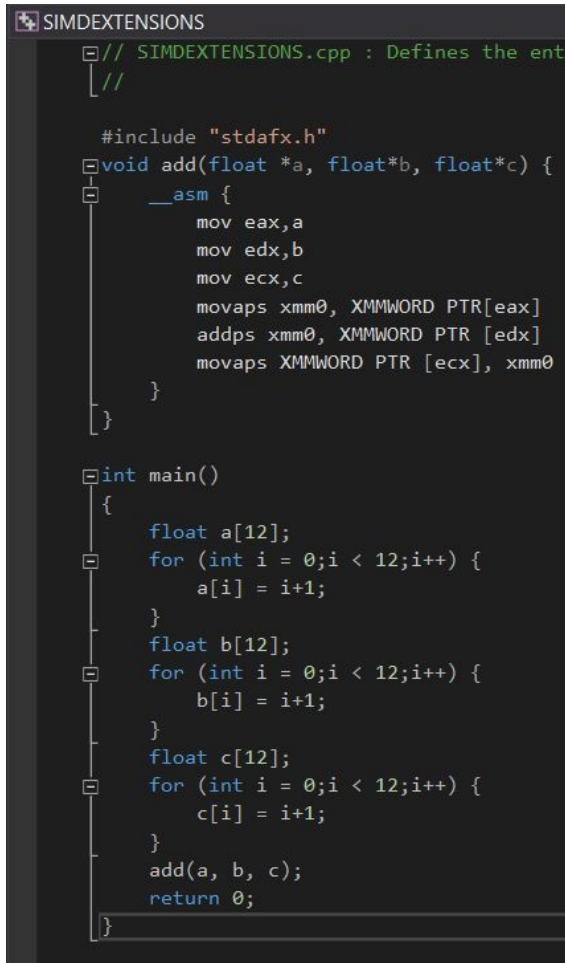
YMM0 = 0000000000000000-0000000000000000-0000000000000000-0000000040800000
YMM1 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM2 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM3 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM4 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM5 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM6 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM7 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000

```

Figure 16: Registres for four-iteration loop after running a couple of time.

YMM0 is a register that can store 4 numbers at a time, and the reason that it is used when we are performing array arithmetic is because we want to be as efficient as possible. Using a register that can perform an operation in parallel is much better than adding values or multiplying them one bit at a time, in this example it is not necessary to use such a large register since the values are small and we are not performing highly advanced operations but in this case the compiler does not know the value that we are using so it assume that because we are using an array it needs to be more efficient so it stores the values in these capable registers.

Streaming SIMD Extensions Using Inlined Assembly Encoding.



```

SIMDEXTENSIONS
// SIMDEXTENSIONS.cpp : Defines the ent
//

#include "stdafx.h"
void add(float *a, float*b, float*c) {
    __asm {
        mov eax,a
        mov edx,b
        mov ecx,c
        movaps xmm0, XMMWORD PTR[eax]
        addps xmm0, XMMWORD PTR [edx]
        movaps XMMWORD PTR [ecx], xmm0
    }
}

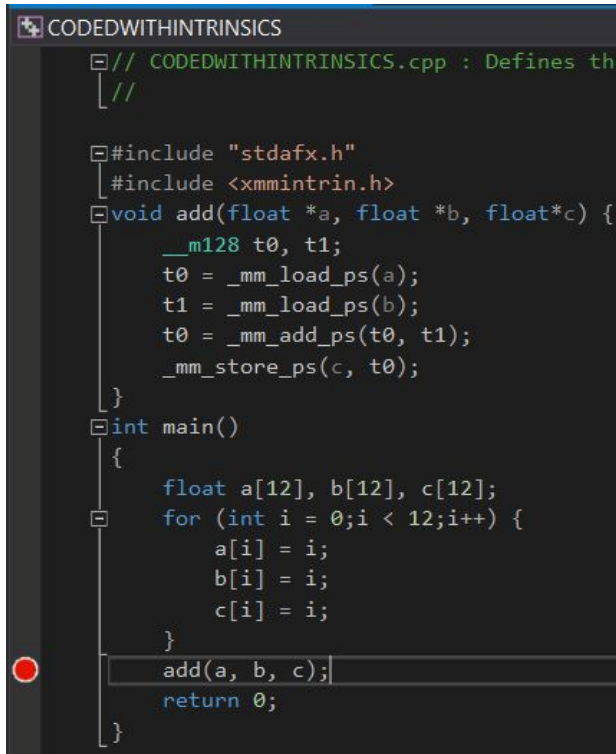
int main()
{
    float a[12];
    for (int i = 0;i < 12;i++) {
        a[i] = i+1;
    }
    float b[12];
    for (int i = 0;i < 12;i++) {
        b[i] = i+1;
    }
    float c[12];
    for (int i = 0;i < 12;i++) {
        c[i] = i+1;
    }
    add(a, b, c);
    return 0;
}

```

Figure 17: C++ CODE for Streaming SIMD Extensions Using Inlined Assembly Encoding.

In this section we will not display the assembly code or the registers. The reason we are not going to display any of that information is because in the add function we already know which registers are going to be used to perform array arithmetic because we have specified them to be, xmm0, eax, edx and ecx we will simply perform the operation in visual studio and it is more efficient than the previous example for larger sets of information because it is directly storing and the data onto the registers instead of having to first take an offset to a given address and then wait to perform the operation.

Simple Four-Iteration Loop coded with intrinsics.

A screenshot of a code editor window titled "CODEDWITHINTRINSICS". The code is in C++ and defines a function "add" that takes three float pointers and uses Intel MMX intrinsics to add two floats and store the result. The "main" function initializes three arrays of 12 floats, sets each element to its index, and then calls the "add" function. The code is as follows:

```
// CODEDWITHINTRINSICS.cpp : Defines the entry point for the application.
//

#include "stdafx.h"
#include <xmmintrin.h>

void add(float *a, float *b, float*c) {
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}

int main()
{
    float a[12], b[12], c[12];
    for (int i = 0; i < 12; i++) {
        a[i] = i;
        b[i] = i;
        c[i] = i;
    }
    add(a, b, c);
    return 0;
}
```

Figure 18: C++ code for simple Four-Iteration Loop coded with intrinsics.

The difference in this example is that we will be using intrinsics from the intel library to perform the operations that we want to, we will be storing a and b into specific locations, then adding into where we have stored a and then finally storing that in c.

```

int main()
{
00F01780  push      ebp
00F01781  mov       ebp,esp
00F01783  sub       esp,178h
00F01789  push      ebx
00F0178A  push      esi
00F0178B  push      edi
00F0178C  lea       edi,[ebp-178h]
00F01792  mov       ecx,5Eh
00F01797  mov       eax,0CCCCCCCCh
00F0179C  rep stos  dword ptr es:[edi]
00F0179E  mov       eax,dword ptr [__security_cookie (0F0A000h)]
00F017A3  xor       eax,ebp
00F017A5  mov       dword ptr [ebp-4],eax

    float a[12], b[12], c[12];
    for (int i = 0; i < 12; i++) {
00F017A8  mov       dword ptr [ebp-0B4h],0
00F017B2  jmp       main+43h (0F017C3h)
00F017B4  mov       eax,dword ptr [ebp-0B4h]
00F017BA  add       eax,1
00F017BD  mov       dword ptr [ebp-0B4h],eax
00F017C3  cmp       dword ptr [ebp-0B4h],0Ch
00F017CA  jge       main+8Dh (0F0180Dh)
        a[i] = i;
00F017CC  cvtsi2ss  xmm0,dword ptr [ebp-0B4h]
00F017D4  mov       eax,dword ptr [ebp-0B4h]
00F017DA  movss     dword ptr a[eax*4],xmm0
        b[i] = i;
00F017E0  cvtsi2ss  xmm0,dword ptr [ebp-0B4h]
00F017E8  mov       eax,dword ptr [ebp-0B4h]
00F017EE  movss     dword ptr b[eax*4],xmm0
        c[i] = i;
00F017F4  cvtsi2ss  xmm0,dword ptr [ebp-0B4h]

```

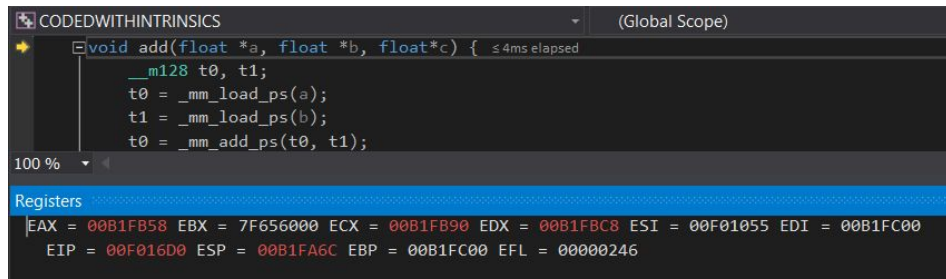
Figure 19: Disassembly for Simple Four-Iteration Loop coded with intrinsics part 1.

```

00F017FC  mov       eax,dword ptr [ebp-0B4h]
00F01802  movss     dword ptr c[eax*4],xmm0
    }
00F0180B  jmp       main+34h (0F017B4h)
    add(a, b, c);
00F0180D  lea       eax,[c]
00F01813  push      eax
00F01814  lea       ecx,[b]
00F01817  push      ecx
00F01818  lea       edx,[a]
00F0181B  push      edx
00F0181C  call      add (0F0127Bh)
00F01821  add       esp,0Ch
    return 0;
00F01824  xor       eax,eax
}

```

Figure 20: Simple Four-Iteration Loop coded with intrinsics part 2.



```

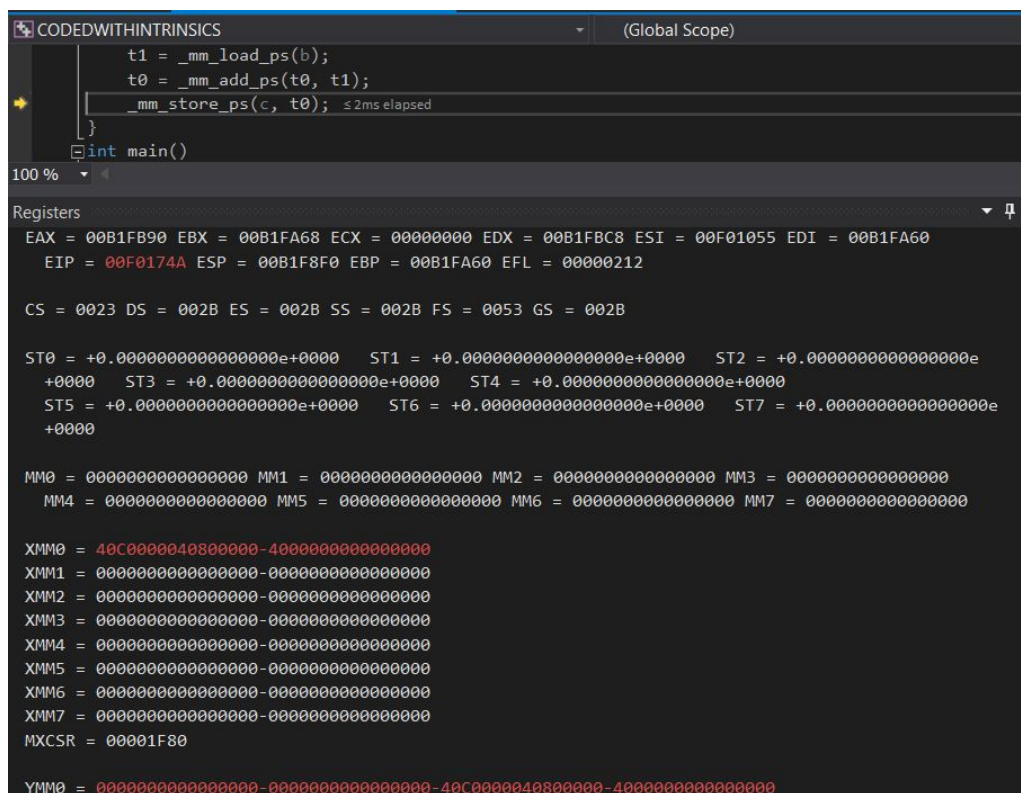
CODEDWITHINTRINSICS (Global Scope)
void add(float *a, float *b, float *c) { ≤ 4ms elapsed
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
}

Registers
EAX = 00B1FB58 EBX = 7F656000 ECX = 00B1FB90 EDX = 00B1FBC8 ESI = 00F01055 EDI = 00B1FC00
EIP = 00F016D0 ESP = 00B1FA6C EBP = 00B1FC00 EFL = 00000246

```

Figure 21: Registers for Simple Four-Iteration Loop coded with intrinsics part 1.

As we can see the registers that are being used are EAX, ECX, EDX to store the values that we will be using.



```

CODEDWITHINTRINSICS (Global Scope)
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0); ≤ 2ms elapsed
}
int main()

Registers
EAX = 00B1FB90 EBX = 00B1FA68 ECX = 00000000 EDX = 00B1FBC8 ESI = 00F01055 EDI = 00B1FA60
EIP = 00F0174A ESP = 00B1F8F0 EBP = 00B1FA60 EFL = 00000212

CS = 0023 DS = 002B ES = 002B SS = 002B FS = 0053 GS = 002B

ST0 = +0.0000000000000000e+0000 ST1 = +0.0000000000000000e+0000 ST2 = +0.0000000000000000e+0000
ST3 = +0.0000000000000000e+0000 ST4 = +0.0000000000000000e+0000 ST5 = +0.0000000000000000e+0000
ST6 = +0.0000000000000000e+0000 ST7 = +0.0000000000000000e+0000

MM0 = 0000000000000000 MM1 = 0000000000000000 MM2 = 0000000000000000 MM3 = 0000000000000000
MM4 = 0000000000000000 MM5 = 0000000000000000 MM6 = 0000000000000000 MM7 = 0000000000000000

XMM0 = 40C0000040800000-4000000000000000
XMM1 = 0000000000000000-0000000000000000
XMM2 = 0000000000000000-0000000000000000
XMM3 = 0000000000000000-0000000000000000
XMM4 = 0000000000000000-0000000000000000
XMM5 = 0000000000000000-0000000000000000
XMM6 = 0000000000000000-0000000000000000
XMM7 = 0000000000000000-0000000000000000
MXCSR = 00001F80

YMM0 = 0000000000000000-0000000000000000-40C0000040800000-4000000000000000

```

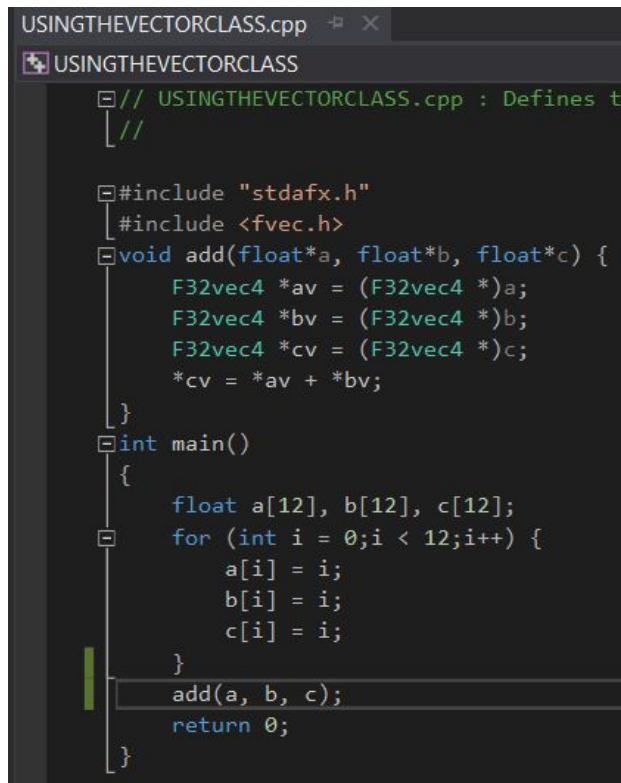
Figure 22: Registers for Simple Four-Iteration Loop coded with intrinsics part 2.

In this example we are still using register XMM0 and YMM0 but the major difference here is that we are using more of the data. We are being more efficient with performing array arithmetic. As you can see in YMM0 the first two numbers both have more than one value stored in them because the compiler is being as efficient as possible it is using a larger part of one register to

store two values instead of one, it keeps track of how many bytes one value is in order for it to perform advanced operations properly without incorrectly changing the value that is being used.

It is performing operations in parts like using threads in order to save time.

Accessing Arrays using the Vector Class.



```
USINGTHEVECTORCLASS.cpp
USINGTHEVECTORCLASS

// USINGTHEVECTORCLASS.cpp : Defines the entry point for the application.
//

#include "stdafx.h"
#include <fvec.h>

void add(float*a, float*b, float*c) {
    F32vec4 *av = (F32vec4 *)a;
    F32vec4 *bv = (F32vec4 *)b;
    F32vec4 *cv = (F32vec4 *)c;
    *cv = *av + *bv;
}

int main()
{
    float a[12], b[12], c[12];
    for (int i = 0; i < 12; i++) {
        a[i] = i;
        b[i] = i;
        c[i] = i;
    }
    add(a, b, c);
    return 0;
}
```

Figure 23: C++ code for Accessing Arrays using the Vector Class.

```

Disassembly  X USINGTHEVECTORCLASS.cpp
Address: main(void)
v Viewing Options
int main()
{
01391940  push     ebp
01391941  mov     ebp,esp
01391943  sub     esp,178h
01391949  push     ebx
0139194A  push     esi
0139194B  push     edi
0139194C  lea     edi,[ebp-178h]
01391952  mov     ecx,5Eh
01391957  mov     eax,0CCCCCCCCh
0139195C  rep stos dword ptr es:[edi]
0139195E  mov     eax,dword ptr [__security_cookie (0139A024h)]
01391963  xor     eax,ebp
01391965  mov     dword ptr [ebp-4],eax

    float a[12], b[12], c[12];
    for (int i = 0; i < 12; i++) {
01391968  mov     dword ptr [ebp-0B4h],0
01391972  jmp     main+43h (01391983h)
01391974  mov     eax,dword ptr [ebp-0B4h]
0139197A  add     eax,1
0139197D  mov     dword ptr [ebp-0B4h],eax
01391983  cmp     dword ptr [ebp-0B4h],0Ch
0139198A  jge     main+8Dh (013919CDh)

        a[i] = i;
0139198C  cvtsi2ss xmm0,dword ptr [ebp-0B4h]
01391994  mov     eax,dword ptr [ebp-0B4h]
0139199A  movss   dword ptr a[eax*4],xmm0

        b[i] = i;
013919A0  cvtsi2ss xmm0,dword ptr [ebp-0B4h]
013919A8  mov     eax,dword ptr [ebp-0B4h]
013919AE  movss   dword ptr b[eax*4],xmm0

        c[i] = i;
013919B4  cvtsi2ss xmm0,dword ptr [ebp-0B4h]

```

Figure 24: Disassembly code for Accessing Arrays using the Vector Class part 1.

```

013919BC  mov     eax,dword ptr [ebp-0B4h]
013919C2  movss   dword ptr c[eax*4],xmm0
    }
013919CB  jmp     main+34h (01391974h)
    add(a, b, c);
013919CD  lea     eax,[c]
013919D3  push    eax
013919D4  lea     ecx,[b]
013919D7  push    ecx
013919D8  lea     edx,[a]
013919DB  push    edx
013919DC  call    add (01391285h)
013919E1  add     esp,0Ch
    return 0;
013919E4  xor     eax,eax
}

```


Figure 25: Disassembly code for Accessing Arrays using the Vector Class part 2.

```

Registers
EAX = CCCCCCCC EBX = 005BF89C ECX = 00000000 EDX = 005BF9FC ESI = 01391055 EDI = 005BF890
EIP = 013918E0 ESP = 005BF780 EBP = 005BF890 EFL = 00000216

CS = 0023 DS = 002B ES = 002B SS = 002B FS = 0053 GS = 002B

ST0 = +0.000000000000000e+0000 ST1 = +0.000000000000000e+0000 ST2 = +0.000000000000000e+0000
ST3 = +0.000000000000000e+0000 ST4 = +0.000000000000000e+0000
ST5 = +0.000000000000000e+0000 ST6 = +0.000000000000000e+0000 ST7 = +0.000000000000000e+0000

MM0 = 0000000000000000 MM1 = 0000000000000000 MM2 = 0000000000000000 MM3 = 0000000000000000
MM4 = 0000000000000000 MM5 = 0000000000000000 MM6 = 0000000000000000 MM7 = 0000000000000000

XMM0 = 0000000000000000-0000000041300000
XMM1 = 0000000000000000-0000000000000000
XMM2 = 0000000000000000-0000000000000000
XMM3 = 0000000000000000-0000000000000000
XMM4 = 0000000000000000-0000000000000000
XMM5 = 0000000000000000-0000000000000000
XMM6 = 0000000000000000-0000000000000000
XMM7 = 0000000000000000-0000000000000000
MXCSR = 00001F80

YMM0 = 0000000000000000-0000000000000000-0000000000000000-0000000041300000
YMM1 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM2 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM3 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM4 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM5 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM6 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM7 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000

0x005BF8A4 = 005BF9FC

```

Figure 26: Registers for Accessing Arrays using the Vector Class, as we can see we are storing values in YMM0 and XMM0 the values are there to be more efficient.

```

Registers
EAX = 005BF660 EBX = 005BF76C ECX = 005BF660 EDX = 005BF790 ESI = 01391055 EDI = 005BF760
EIP = 0139183F ESP = 005BF610 EBP = 005BF760 EFL = 00000212

CS = 0023 DS = 002B ES = 002B SS = 002B FS = 0053 GS = 002B

ST0 = +0.000000000000000e+0000 ST1 = +0.000000000000000e+0000 ST2 = +0.000000000000000e+0000
ST3 = +0.000000000000000e+0000 ST4 = +0.000000000000000e+0000
ST5 = +0.000000000000000e+0000 ST6 = +0.000000000000000e+0000 ST7 = +0.000000000000000e+0000

MM0 = 0000000000000000 MM1 = 0000000000000000 MM2 = 0000000000000000 MM3 = 0000000000000000
MM4 = 0000000000000000 MM5 = 0000000000000000 MM6 = 0000000000000000 MM7 = 0000000000000000

XMM0 = 4040000040000000-3F80000000000000
XMM1 = 0000000000000000-0000000000000000
XMM2 = 0000000000000000-0000000000000000
XMM3 = 0000000000000000-0000000000000000
XMM4 = 0000000000000000-0000000000000000
XMM5 = 0000000000000000-0000000000000000
XMM6 = 0000000000000000-0000000000000000
XMM7 = 0000000000000000-0000000000000000
MXCSR = 00001F80

YMM0 = 0000000000000000-0000000000000000-4040000040000000-3F80000000000000
YMM1 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM2 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM3 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM4 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM5 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM6 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000
YMM7 = 0000000000000000-0000000000000000-0000000000000000-0000000000000000

```

Figure 27: Registers for Accessing Arrays using the Vector Class.

Thus far we can see that this is the most efficient way to perform the array arithmetic. It is the most efficient because it makes a much fuller use of XMM0 and YMM0. Instead of storing only a single value into either of those two large registers it is storing up to 4 values at a time in each of the registers in order to perform the addition at a much faster rate in less time. Using Vector classes from C++ in this implementation is the most efficient way to perform Array arithmetic operations in out of all four examples demonstrated here.

Conclusion: Performing array arithmetic can happen in multiple ways, the results will be the same but the implementation will be different. Different implementations lead to different run times. We were able to identify what sort of advanced CPU operations my processor is capable of performing and then we were able to run 4 different tests to gain experience on how the different implementations compare in efficiency and in runtime.