Introduction to Intel® Advanced Vector Extensions

By Chris Lomont

Intel® Advanced Vector Extensions (AVX) is a set of instructions for doing Single Instruction Multiple Data (SIMD) operations on Intel® architecture CPUs. These instructions extend previous SIMD offerings (under the acronyms *MMX* and *SSE*) by adding the following new features:

- The 128-bit SIMD registers have been expanded to 256 bits. Intel® AVX is designed to support 512 or 1024 bits in the future.
- Three-operand, nondestructive operations have been added. Previous two-operand instructions performed operations such as A = A + B, which overwrites a source operand; the new operands can perform operations like A = B + C, leaving the original source operands unchanged.
- A few instructions take four-register operands, allowing smaller and faster code by removing unnecessary instructions.
- Memory alignment requirements for operands are relaxed.
- A new extension coding scheme (VEX) has been designed to make future additions easier as well as making coding of instructions smaller and faster to execute.

Closely related to these advances are the new Fused–Multiply–Add (FMA) instructions, which allow faster and more accurate specialized operations such as single instruction A = A * B + C. The FMA instructions should be available in the second-generation Intel® Core™ CPU. Other features include new instructions for dealing with Advanced Encryption Standard (AES) encryption and decryption, a packed carry-less multiplication operation (PCLMULQDQ) useful for certain encryption primitives, and some reserved slots for future instructions, such as a hardware random number generator.

Instruction Set Overview

The new instructions are encoded using what Intel calls a *VEX prefix,* which is a two- or three-byte prefix designed to clean up the complexity of current and future x86/x64 instruction encoding. The two new VEX prefixes are formed from two obsolete 32-bit instructions—Load Pointer Using DS (LDS—0xC4, 3-byte form) and Load Pointer Using ES (LES—0xC5, two-byte form)—which load the DS and ES segment registers in 32-bit mode. In 64-bit mode, opcodes LDS and LES generate an invalid-opcode exception, but under Intel® AVX, these opcodes are repurposed for encoding new instruction prefixes. As a result, the VEX instructions can only be used when running in 64-bit mode. The prefixes allow encoding more registers than previous x86 instructions and are required for accessing the new 256-bit SIMD registers or using the three- and four-operand syntax. As a user, you do not need to worry about this (unless you're writing assemblers or disassemblers).

Note The rest of this article assumes operation in 64-bit mode.

SIMD instructions allow processing of multiple pieces of data in a single step, speeding up throughput for many tasks, from video encoding and decoding to image processing to data analysis to physics simulations. Intel® AVX instructions work on Institute of Electrical and Electronics Engineers (IEEE)-754 floating-point values in 32-bit length (called *single precision*) and in 64-bit length (called *double precision*). IEEE-754 is the standard defining reproducible, robust floating-point operation and is the standard for most mainstream numerical computations.

The older, related SSE instructions also support various signed and unsigned integer sizes, including signed and unsigned byte (B, 8-bit), word (W, 16-bit), doubleword (DW, 32-bit), quadword (QW, 64-bit), and doublequadword (DQ, 128-bit) lengths. Not all instructions are available in all size combinations; for details, see the links provided in "For More Information." See Figure 2 later in this article for a graphical representation of the data types.

The hardware supporting Intel® AVX (and FMA) consists of the 16 256-bit YMM registers YMM0-YMM15 and a 32-bit control/status register called *MXCSR.* The YMM registers are aliased over the older 128-bit XMM registers used for SSE, treating the XMM registers as the lower half of the corresponding YMM register, as shown in Figure 1.

Bits 0–5 of MXCSR indicate SIMD floating-point exceptions with "sticky" bits—after being set, they remain set until cleared using `LDMXCSR` or `FXRSTOR`. Bits 7–12 mask individual exceptions when set, initially set by a power-up or reset. Bits 0–5 represent invalid operation, denormal, divide by zero, overflow, underflow, and precision, respectively. For details, see the links "For More Information."

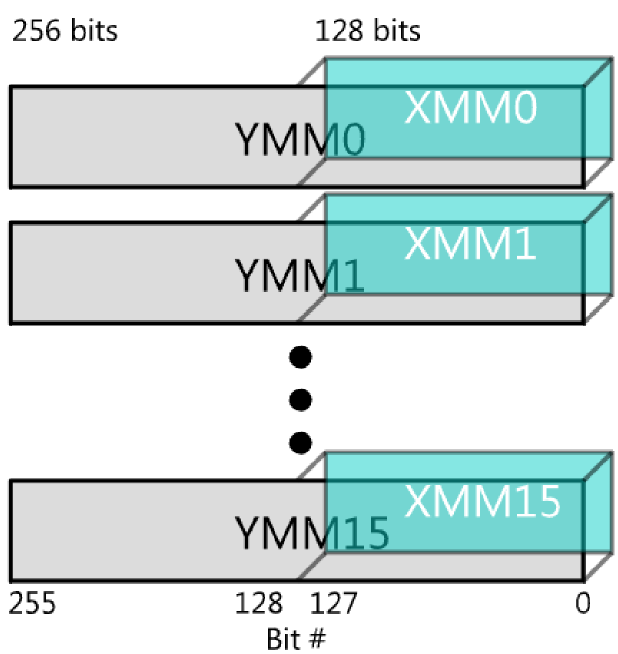

Figure 1. XMM registers overlay the YMM registers.

Figure 2 illustrates the data types used in the SSE and Intel® AVX instructions. Roughly, for Intel® AVX, any multiple of 32-bit or 64-bit floating-point type that adds to 128 or 256 bits is allowed as well as multiples of any integer type that adds to 128 bits.

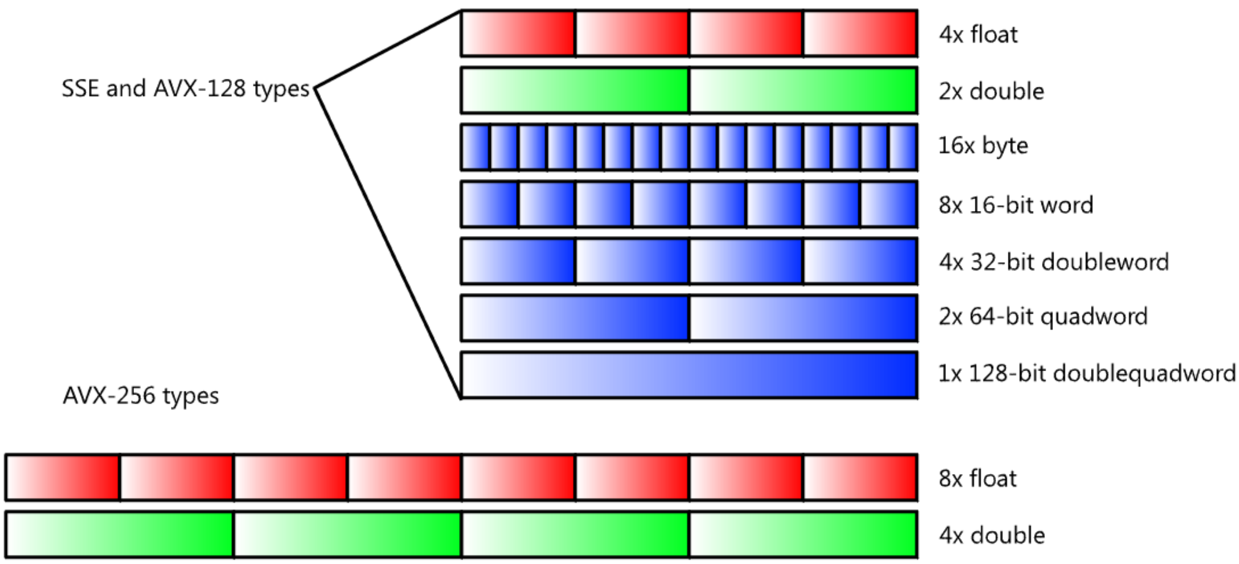

Figure 2. Intel® AVX and SSE data types

Instructions often come in scalar and vector versions, as illustrated in Figure 3. Vector versions operate by treating data in the registers in parallel "SIMD" mode; the scalar version only operates on one entry in each register. This distinction allows less data movement for some algorithms, providing better overall throughput.

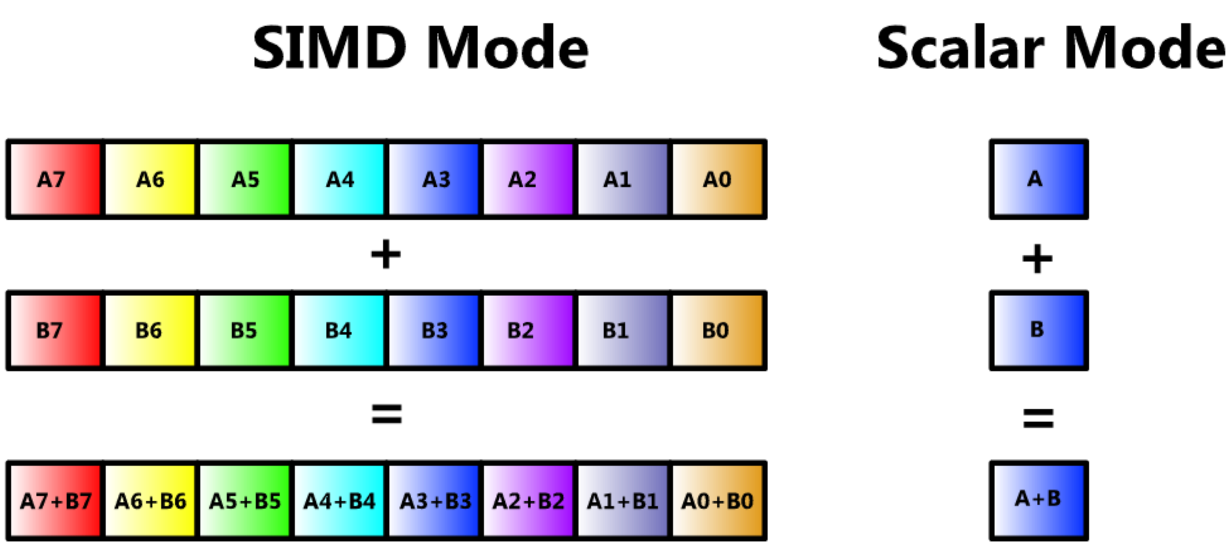

Figure 3. SIMD versus scalar operations

Data is *memory aligned* when the data to be operated upon as an *n*-byte chunk is stored on an *n*-byte memory boundary. For example, when loading 256-bit data into YMM registers, if the data source is 256-bit aligned, the data is called *aligned.*

For SSE operations, memory alignment was required unless explicitly stated. For example, under SSE, there were specific instructions for memory-aligned and memory-unaligned operations, such as the `MOVAPD` (move-aligned packed double) and `MOVUPD` (move-unaligned packed double) instructions. Instructions not split in two like this required aligned accesses.

Intel® AVX has relaxed some memory alignment requirements, so now Intel® AVX by default allows unaligned access; however, this access may come at a performance slowdown, so the old rule of designing your data to be memory aligned is still good practice (16-byte aligned for 128-bit access and 32-byte aligned for 256-bit access). The main exceptions are the VEX-extended versions of the SSE instructions that explicitly required memory-aligned data: These instructions still require aligned data. Other specific instructions requiring aligned access are listed in Table 2.4 of the *Intel® Advanced Vector Extensions Programming Reference* (see "For More Information" for a link).

Another performance concern besides unaligned data issues is that mixing legacy XMM-only instructions and newer Intel® AVX instructions causes delays, so minimize transitions between VEX-encoded instructions and legacy SSE code. Said another way, do not mix VEX-prefixed instructions and non–VEX-prefixed instructions for optimal throughput. If you must do so, minimize transitions between the two by grouping instructions of the same VEX/non-VEX class. Alternatively, there is no transition penalty if the upper YMM bits are set to zero via `VZEROUPPER` or `VZEROALL`, which compilers should automatically insert. This insertion requires an extra instruction, so profiling is recommended.

Intel® AVX Instruction Classes

As mentioned, Intel® AVX adds support for many new instructions and extends current SSE instructions to the new 256-bit registers, with most old SSE instructions having a *V*-prefixed Intel® AVX version for accessing new register sizes and three-operand forms. Depending on how instructions are counted, there are up to a few hundred new Intel® AVX instructions.

For example, the old two-operand SSE instruction `ADDPS xmm1, xmm2/m128` can now be expressed in three-operand syntax as `VADDPS xmm1, xmm2, xmm3/m128` or the 256-bit register using the form `VADDPS ymm1, ymm2, ymm3/m256`. A few instructions allow four operands, such as `VBLENDVPS ymm1, ymm2, ymm3/m256, ymm4`, which conditionally copies single-precision floating-point values from `ymm2` or `ymm3/m256` to `ymm1` based on masks in `ymm4`. This is an improvement on the previous form, where `xmm0` was implicitly needed, requiring compilers to free up `xmm0`. Now, with all registers explicit, there is more freedom for register allocation. Here, `m128` is a 128-bit memory location, `xmm1` is the 128-bit register, and so on.

Some new instructions are VEX only (not SSE extensions), including many ways to move data into and out of the YMM registers. Examples are the useful `VBROADCASTS[S/D]`, which loads a single value into all elements of an XMM or YMM register, and ways to shuffle data around in a register using `VPERMILP[S/D]`. (The bracket notation is explained in the Appendix A.)

Intel® AVX adds arithmetic instructions for variants of add, subtract, multiply, divide, square root, compare, min, max, and round on single- and double-precision packed and scalar floating-point data. Many new conditional predicates are also useful for 128-bit SSE, giving 32 comparison types. Intel® AVX also includes instructions promoted from previous SIMD covering logical, blend, convert, test, pack, unpack, shuffle, load, and store.

The toolset adds new instructions, as well, including non-strided fetching (broadcast of single or multiple data into a 256-bit destination, masked-move primitives for conditional load and store), insert and extract multiple-SIMD data to and from 256-bit SIMD registers, permute primitives to manipulate data within a register, branch handling, and packed testing instructions.

Future Additions

The Intel® AVX manual also lists some proposed future instructions, covered here for completeness. This is not a guarantee that these instructions will materialize as written.

Two instructions (`VCVTPH2PS` and `VCVTPS2PH`) are reserved for supporting 16-bit floating-point conversions to and from single– and double–floating-point types. The 16-bit format is called *half-precision* and has a 10-bit mantissa (with an implied leading 1 for non-denormalized numbers, resulting in 11-bit precision), 5-bit exponent (biased by 15), and 1-bit sign.

The proposed `RDRAND` instruction uses a cryptographically secure hardware digital random bit generator to generate random numbers for 16- 32- , and 64-bit registers. On success, the carry flag is set to 1 (`CF=1`). If not enough entropy is available, the carry flag is cleared (`CF=0`).

Finally, there are four instructions (`RDFDBASE`, `RDGSBASE`, `WRFSBASE`, and `WRGSBASE`) to read and write FS and GS registers at all privilege levels in 64-bit mode.

Another future addition is the FMA instructions, which perform operations similar to A = + A * B + C, where either of the plus signs (+) on the right can be changed to a minus sign (−) and the three operands on the right can be in any order. There are also forms for interleaved addition and subtraction. Packed FMA instructions can perform eight single-precision FMA operations or four double-precision FMA operations with 256-bit vectors.

FMA operations such as A = A * B + C are better than performing one step at a time, because intermediate results are treated as infinite precision, with rounding done on store, and thus are more accurate for computation. This single rounding is what gives the “fused” prefix. They are also faster than performing the computation in steps.

Each instruction comes in three forms for the ordering of the operands A, B, and C, with the ordering corresponding to a three-digit extension: form *132* does A = AC + B, form *213* does

A = BA + C, and form *231* does A = BC + A. The ordering number is just the order of the operands on the right side of the expression.

Availability and Support

Detecting availability of the Intel® AVX features in hardware requires using the `CPUID` instruction to query support in the CPU and in the operating system, as detailed later. Second-generation Intel® Core™ processors (code named Sandy Bridge), released in Q1, 2011, are the first from Intel® supporting Intel® AVX technology. These processors will not have the new FMA instructions. For development and testing without hardware support, the free Intel® Software Development Emulator (see "For More Information" for a link) includes support for all these features, including Intel® AVX, FMA, PCLMULQDQ, and AES instructions.

To use the Intel® AVX extensions reliably in most settings, the operating system must support saving and loading the new registers (with `XSAVE`/`XRSTOR`) on thread context switches to prevent data corruption. To help avoid such errors, operating systems supporting Intel® AVX–aware context switches explicitly set a CPU bit enabling the new instructions; otherwise, an undefined opcode (`#UD`) exception is generated when Intel® AVX instructions are used.

Windows* 7 with Service Pack 1 (SP1) and Windows Server* 2008 R2 with SP1—both 32- and 64-bit versions—and later versions Windows support Intel® AVX save and restore in thread and process switches. Linux* kernels from 2.6.30 (June 2009) and later support Intel® AVX, as well.

Detecting Availability and Support

Detection of support for the four areas—Intel® AVX, FMA, AES, and PCLMULQDQ—are similar and require similar steps consisting of checking for hardware and operating system support for the desired feature (see Table 1). These steps are (counting bits starting at bit 0):

1. Verify that the operating system supports `XGETBV` using `CPUID.1:ECX.OSXSAVE bit 27 = 1`.
2. At the same time, verify that `CPUID.1:ECX bit 28=1` (Intel® AVX supported) and/or `bit 25=1` (AES supported) and/or `bit 12=1` (FMA supported) and/or `bit 1=1` (PCLMULQDQ) are supported.
3. Issue `XGETBV`, and verify that the feature-enabled mask at bits 1 and 2 are 11b (XMM state and YMM state enabled by the operating system).

Table 1. Feature-detection Masks

Feature	Bits to check	Constant
AVX	28, 27	`018000000H`
VAES	28, 27, and 25	`01A000000H`
VPCLMULQDQ	28, 27, and 1	`018000002H`

FMA	28, 27, and 12	018001000H

Example code implementing this process is provided in Listing 1, where the CONSTANT is the value from Table 1. A Microsoft* Visual Studio* C++ intrinsic version is given later.

Listing 1. Feature Detection

```
INT Supports_Feature()
   {
   ; result returned in eax
   mov eax, 1
   cpuid
   and ecx, CONSTANT
   cmp ecx, CONSTANT; check desired feature flags
   jne not_supported
   ; processor supports features
   mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
   XGETBV; result in EDX:EAX
   and eax, 06H
   cmp eax, 06H; check OS has enabled both XMM and YMM state support
   jne not_supported
   mov eax, 1; mark as supported
   jmp done
   NOT_SUPPORTED:
   mov eax, 0 ; // mark as not supported
   done:
   }
```

Usage

At the lowest programming level, most common x86 assemblers now support Intel® AVX, FMA, AES, and the VPCLMULQDQ instructions, including MASM (VS2010 version), NASM, FASM, and YASM. See their respective documentation for details.

For language compilers, Intel® C++ version 11.1 and later and Intel® Fortran compilers support Intel® AVX through compiler switches, and both compilers support automatic vectorization of floating-point loops. The Intel® C++ compiler supports Intel® AVX intrinsics (use `#include <immintrin.h>` to access intrinsics) and inline assembly and even supports Intel® AVX intrinsics emulation using `#include "avxintrin_emu.h"`.

Visual Studio* C++ 2010 with SP1 and later has support for Intel® AVX (see "For More Information") when compiling 64-bit code (use the `/arch:AVX` compiler switch). It supports intrinsics using the `<immintrin.h>` header but not inline assembly. Intel® AVX support is also in MASM, the disassembly view of code, and the debugger views of registers (giving full YMM support).

In the GNU Compiler Connection (GCC), version 4.4 supports Intel® AVX intrinsics through the same header, `<immintrin.h>`. Other GNU toolchain support is found in Binutils 2.20.51.0.1 and later, gdb 6.8.50.20090915 and later, recent GNU Assembler (GAS) versions, and `objdump`. If your compiler does not support Intel® AVX, you can emit the required bytes under many circumstances, but first-class support makes your life easier.

Each of the three C++ compilers mentioned supports the same intrinsic operations to simplify using Intel® AVX from C or C++ code. *Intrinsics* are functions that the compiler replaces with the proper assembly instructions. Most Intel® AVX intrinsic names follow the following format:

```
_mm256_op_suffix(data_type param1, data_type param2, data_type param3)
```

where `_mm256` is the prefix for working on the new 256-bit registers; `_op` is the operation, like `add` for addition or `sub` for subtraction; and `_suffix` denotes the type of data to operate on, with the first letters denoting packed (**p**), extended packed (**ep**), or scalar (**s**). The remaining letters are the types in Table 2.

Table 2. Intel® AVX Suffix Markings

Marking	Meaning
`[s/d]`	Single- or double-precision floating point
`[i/u]nnn`	Signed or unsigned integer of bit size *nnn,* where *nnn* is 128, 64, 32, 16, or 8
`[ps/pd/sd]`	Packed single, packed double, or scalar double
`epi32`	Extended packed 32-bit signed integer
`si256`	Scalar 256-bit integer

Data types are in Table 3. The first two parameters are source registers, and the third parameter (when present) is an integer mask, selector, or offset value.

Table 3. Intel® AVX Intrinsics Data Types

Type	Meaning
`__m256`	256-bit as eight single-precision floating-point values, representing a YMM register or memory location
`__m256d`	256-bit as four double-precision floating-point values, representing a YMM register or memory location
`__m256i`	256-bit as integers, (bytes, words, etc.)
`__m128`	128-bit single precision floating-point (32 bits each)
`__m128d`	128-bit double precision floating-point (64 bits each)

Some intrinsics are in other headers, such as the AES and PCLMULQDQ being in `<wmmintrin.h>`. Consult your compiler documentation or the web to track down where various intrinsics live.

Visual Studio 2010*

For conciseness, the rest of this article uses Visual Studio* 2010 with SP1; similar code should work on the Intel® compiler or GCC. Visual Studio 2010 with SP1 can automatically generate Intel® AVX code if you click **Project Properties > Configuration > Code Generation**, select **Not Set** under **Enable Enhanced Instruction Set**, and then manually add `/arch:AVX` to the command line under the **Command Line** entry. As an example of using intrinsics, Listing 2 offers an intrinsic-based Intel® AVX feature-detection routine.

Listing 2. Intrinsic-based Feature Detection

```
// get AVX intrinsics
#include <immintrin.h>
// get CPUID capability
#include <intrin.h>

// written for clarity, not conciseness
#define OSXSAVEFlag (1UL<<27)
#define AVXFlag     ((1UL<<28)|OSXSAVEFlag)
#define VAESFlag    ((1UL<<25)|AVXFlag|OSXSAVEFlag)
#define FMAFlag     ((1UL<<12)|AVXFlag|OSXSAVEFlag)
#define CLMULFlag   ((1UL<< 1)|AVXFlag|OSXSAVEFlag)

bool DetectFeature(unsigned int feature)
	{
	int CPUInfo[4], InfoType=1, ECX = 1;
	__cpuidex(CPUInfo, 1, 1);       // read the desired CPUID format
	unsigned int ECX = CPUInfo[2];  // the output of CPUID in the ECX register.
	if ((ECX & feature) != feature) // Missing feature
		return false;
	__int64 val = _xgetbv(0);       // read XFEATURE_ENABLED_MASK register
	if ((val&6) != 6)               // check OS has enabled both XMM and YMM support.
		return false;
	return true;
	}
```

Mandelbrot Example

To demonstrate using the new instructions, compute Mandelbrot set images using straight C/C++ code (checking to ensure that the compiler did not convert the code to Intel® AVX instructions!) and the new Intel® AVX instructions as intrinsics, comparing their performance. A Mandelbrot set is a computationally intensive operation on complex numbers, defined in pseudocode as shown in Listing 3.

Listing 3. Mandelbrot Pseudocode

```
z,p are complex numbers
for each point p on the complex plane
	z = 0
	for count = 0 to max_iterations
		if abs(z) > 2.0
```

```
                    break
            z = z*z+p
    set color at p based on count reached
```

The usual image is over the portion of the complex plane in the rectangle (-2,-1) to (1,1). Coloring can be done in many ways (not covered here). Raise the maximum iteration count to zoom into portions and determine whether a value “escapes” over time.

To really stress the CPU, zoom in and draw the box (0.29768, 0.48364) to (0.29778, 0.48354), computing the grid of counts at multiple sizes and using a max iteration of 4096. The resulting grid of counts, when colored appropriately, is shown in Figure 4.

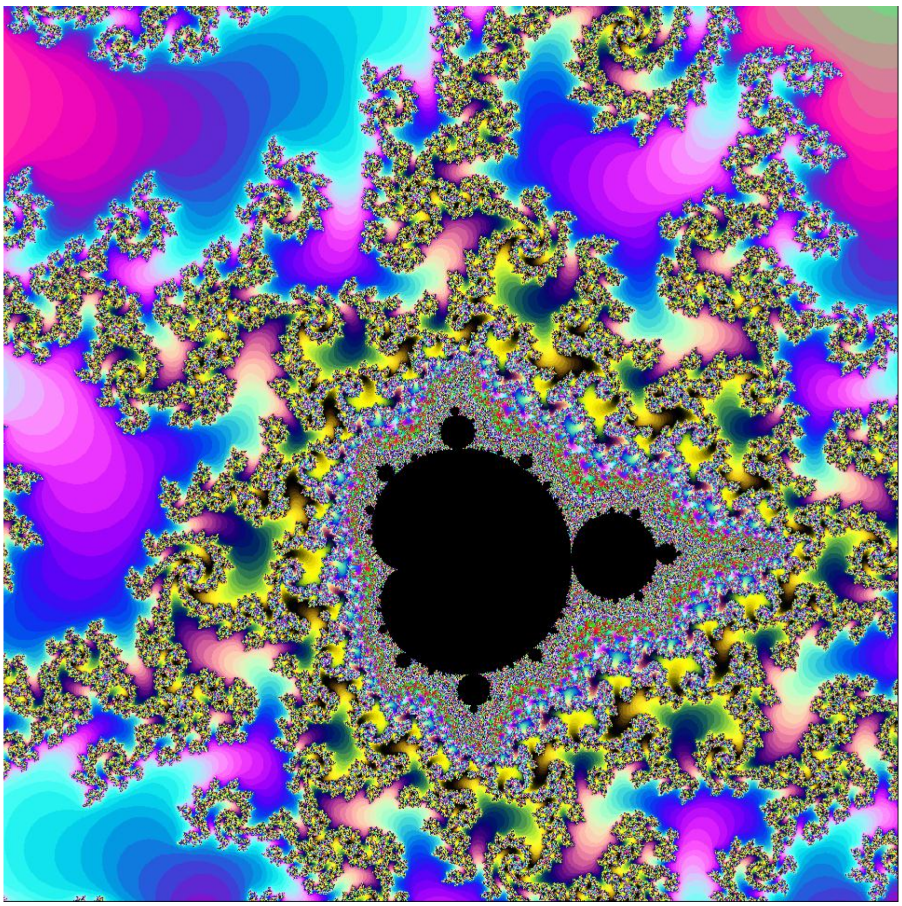

Figure 4. Mandelbrot set (0.29768, 0.48364) to (0.29778, 0.48354), with max iterations of 4096

A basic C++ implementation to compute the iteration counts is provided in Listing 4. The absolute value of the complex number compared to 2 is replaced with the norm compared to 4.0, almost doubling the speed by removing a square root. For all versions, use single-precision floats to pack as many elements into the YMM registers as possible, which is faster but loses precision compared to doubles when zooming in further.

Listing 4. Simple Mandelbrot C++ Code

```
// simple code to compute Mandelbrot in C++
#include <complex>
void MandelbrotCPU(float x1, float y1, float x2, float y2,
                   int width, int height, int maxIters, unsigned short * image)
{
       float dx = (x2-x1)/width, dy = (y2-y1)/height;
       for (int j = 0; j < height; ++j)
              for (int i = 0; i < width; ++i)
              {
                     complex<float> c (x1+dx*i, y1+dy*j), z(0,0);
                     int count = -1;
                     while ((++count < maxIters) && (norm(z) < 4.0))
                            z = z*z+c;
                     *image++ = count;
              }
}
```

Test multiple versions for performance: the basic one in Listing 4, a similar CPU version made by expanding the complex types with floats, an intrinsic-based SSE version, and an intrinsic-based Intel® AVX version shown in Listing 5. Each version is tested on image sizes of 128×128, 256×256, 512×512, 1024×1024, 2048×2048, and 4096×4096. The performance of each implementation could likely be improved while retaining its underlying instruction set constraints with more work, but they should be representative of what you can obtain.

The Intel® AVX version has been carefully crafted to fit as much as possible into the 16 YMM registers. To help track how you want them to be allocated, the variables are names `ymm0` through `ymm15`. Of course, the compiler allocates registers as it sees fit, but by being careful, you can try to make all computations stay in registers this way. (Actually, from looking at the disassembly, the compiler does not allocate them nicely, and recasting this in assembly code would be a good exercise to anyone learning Intel® AVX).

Listing 5. Intel® AVX–intrinsic Mandelbrot Implementation

```
float dx = (x2-x1)/width;
float dy = (y2-y1)/height;
// round up width to next multiple of 8
int roundedWidth = (width+7) & ~7UL;

float constants[] = {dx, dy, x1, y1, 1.0f, 4.0f};
__m256 ymm0 = _mm256_broadcast_ss(constants);   // all dx
__m256 ymm1 = _mm256_broadcast_ss(constants+1); // all dy
__m256 ymm2 = _mm256_broadcast_ss(constants+2); // all x1
__m256 ymm3 = _mm256_broadcast_ss(constants+3); // all y1
__m256 ymm4 = _mm256_broadcast_ss(constants+4); // all 1's (iter increments)
__m256 ymm5 = _mm256_broadcast_ss(constants+5); // all 4's (comparisons)

float incr[8]={0.0f,1.0f,2.0f,3.0f,4.0f,5.0f,6.0f,7.0f}; // used to reset the i position when
 j increases
__m256 ymm6 = _mm256_xor_ps(ymm0,ymm0); // zero out j counter (ymm0 is just a dummy)

for (int j = 0; j < height; j+=1)
{
       __m256 ymm7  = _mm256_load_ps(incr);  // i counter set to 0,1,2,..,7
       for (int i = 0; i < roundedWidth; i+=8)
       {
```

```
            __m256 ymm8 = _mm256_mul_ps(ymm7, ymm0);  // x0 = (i+k)*dx
            ymm8 = _mm256_add_ps(ymm8, ymm2);         // x0 = x1+(i+k)*dx
            __m256 ymm9 = _mm256_mul_ps(ymm6, ymm1);  // y0 = j*dy
            ymm9 = _mm256_add_ps(ymm9, ymm3);         // y0 = y1+j*dy
            __m256 ymm10 = _mm256_xor_ps(ymm0,ymm0);  // zero out iteration counter
            __m256 ymm11 = ymm10, ymm12 = ymm10;        // set initial xi=0, yi=0

            unsigned int test = 0;
            int iter = 0;
            do
            {
                    __m256 ymm13 = _mm256_mul_ps(ymm11,ymm11); // xi*xi
                    __m256 ymm14 = _mm256_mul_ps(ymm12,ymm12); // yi*yi
                    __m256 ymm15 = _mm256_add_ps(ymm13,ymm14); // xi*xi+yi*yi

                    // xi*xi+yi*yi < 4 in each slot
                    ymm15 = _mm256_cmp_ps(ymm15,ymm5, _CMP_LT_OQ);
                    // now ymm15 has all 1s in the non overflowed locations

        test = _mm256_movemask_ps(ymm15)&255;       // lower 8 bits are comparisons
                    ymm15 = _mm256_and_ps(ymm15,ymm4);
                    // get 1.0f or 0.0f in each field as counters
                    // counters for each pixel iteration
                    ymm10 = _mm256_add_ps(ymm10,ymm15);

                    ymm15 = _mm256_mul_ps(ymm11,ymm12);         // xi*yi
                    ymm11 = _mm256_sub_ps(ymm13,ymm14);         // xi*xi-yi*yi
                    ymm11 = _mm256_add_ps(ymm11,ymm8);          // xi <- xi*xi-yi*yi+x0 done!
                    ymm12 = _mm256_add_ps(ymm15,ymm15);         // 2*xi*yi
                    ymm12 = _mm256_add_ps(ymm12,ymm9);          // yi <- 2*xi*yi+y0

                    ++iter;
            } while ((test != 0) && (iter < maxIters));

            // convert iterations to output values
            __m256i ymm10i = _mm256_cvtps_epi32(ymm10);

            // write only where needed
            int top = (i+7) < width? 8: width&7;
            for (int k = 0; k < top; ++k)
                    image[i+k+j*width] = ymm10i.m256i_i16[2*k];

            // next i position - increment each slot by 8
            ymm7 = _mm256_add_ps(ymm7, ymm5);
            ymm7 = _mm256_add_ps(ymm7, ymm5);
        }
        ymm6 = _mm256_add_ps(ymm6,ymm4); // increment j counter
}
```

The full code for all versions and a Visual Studio* 2010 with SP1 project, including a testing harness, is available at from the links in the "For More Information" section.

The results are shown in Figures 5 and 6. To prevent tying numbers too much to a specific CPU speed, Figure 5 shows performance of each version relative the CPU version, which represents a straightforward non-SIMD C/C++ implementation of the algorithm. For those who much know, the tests were run on a system with Intel® Core™ i7-2600K CPU @ 3.40 GHz, RAM 16GB, Windows* 7 x64 Ultimate with Service Pack 1, and no other programs running during testing, but the relative performance should be similar on other machines. As expected, the SSE version performs almost 4 times as well, because it is doing 4 pixels per pass, and the Intel® AVX version performs almost 8 times as well as the CPU version. Because there is overhead from loops, memory access, less-than-perfect instruction ordering, and other factors, 4- and 8-fold improvements should be about the best possible, so this is pretty good for a first try.

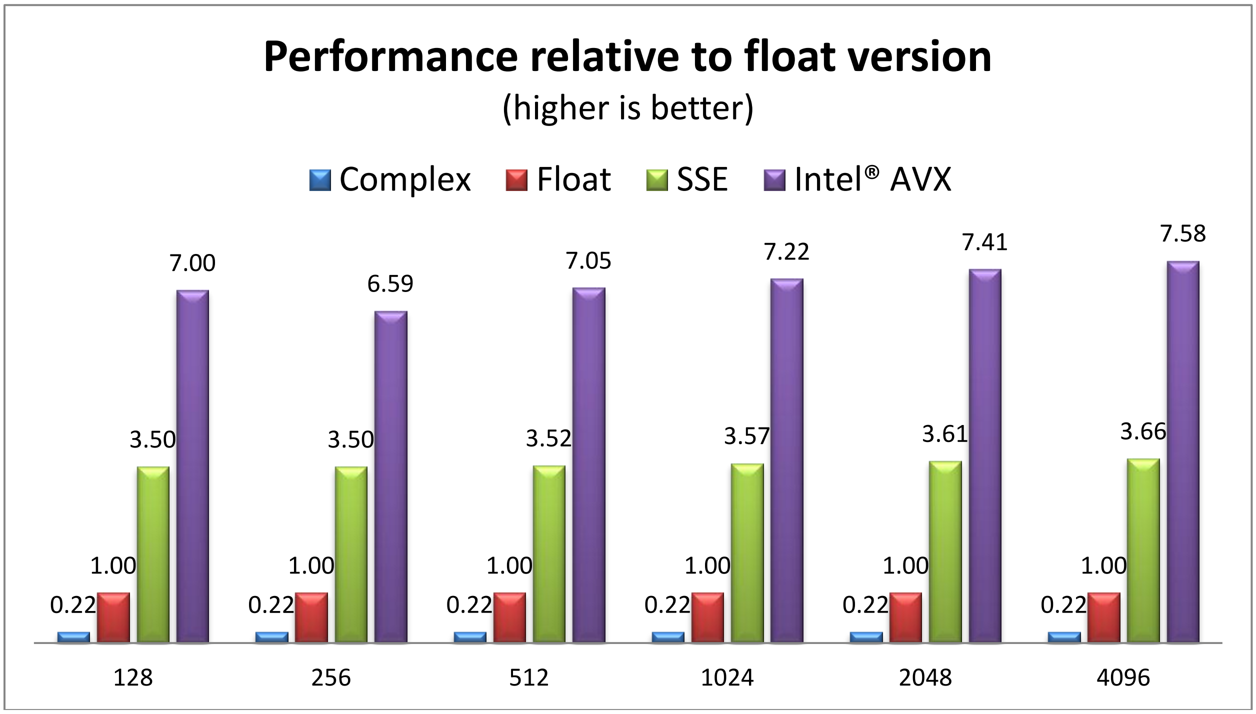

Figure 5. Relative performance across sizes

The second graph in Figure 6 shows that the pixels computed per millisecond are fairly constant over each size; again, the algorithms show almost quadrupling of performance from the CPU to SSE version and another doubling from the SSE to Intel® AVX version.

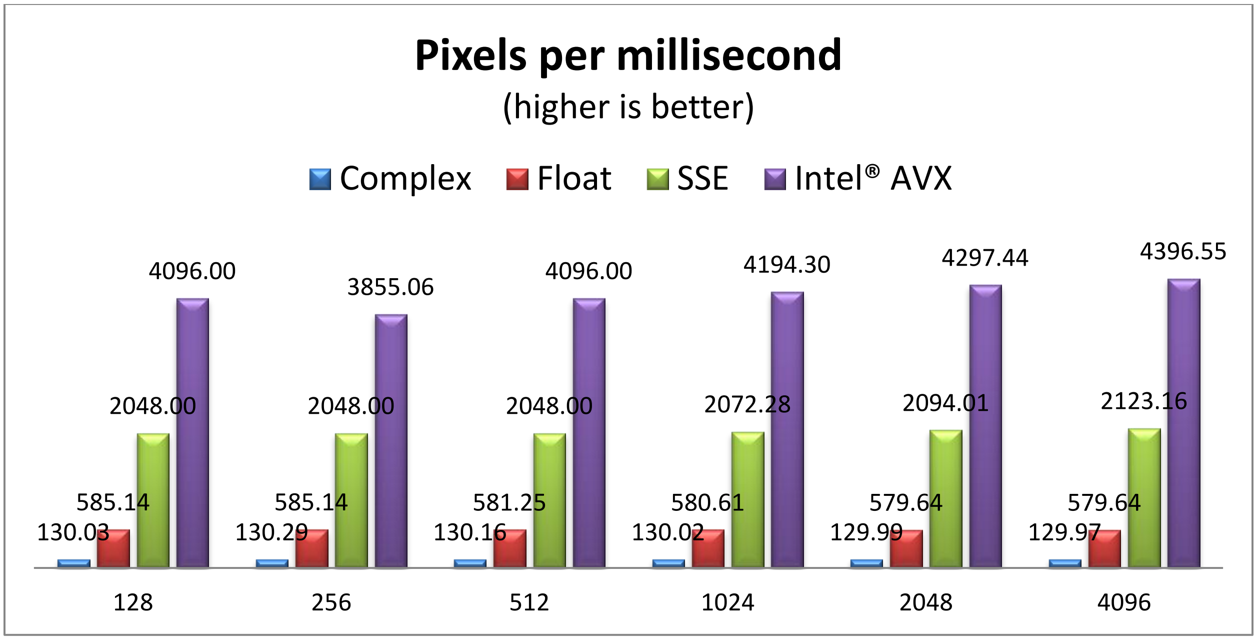

Figure 6. Absolute performance across sizes

Conclusion

This article provided a mid-level overview of the new Intel® Advanced Vector Extensions. These extensions are similar to previous SSE instructions but offer a much larger register space and add some new instructions. The Mandelbrot example shows performance gains over previous technology in the amount expected. For full details, be sure to check out the Intel® Advanced Vector Extensions Programming Reference (see "For More Information" for a link).

Happy hacking!

For More Information

Intel® Advanced Vector Extensions Programming Reference at http://software.intel.com/file/35247

Federal Information Processing Standards Publication 197, "Announcing the Advanced Encryption Standard," at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

The IEEE 754-2008 floating-point format standard at http://en.wikipedia.org/wiki/IEEE_754-2008

Floating-Point Support for 64-Bit Drivers at http://msdn.microsoft.com/en-us/library/ff545910.aspx

Wikipedia's entry on the Mandelbrot set at http://en.wikipedia.org/wiki/Mandelbrot_set

Intel® Software Development Emulator at http://software.intel.com/en-us/articles/intel-software-development-emulator

The complete Mandelbrot Intel® AVX implementation for download at http://www.lomont.org

Appendix A: Instruction Set Reference

Many instructions come in packed or scalar form, meaning that they work on multiple parallel elements or on a single element in the register—a distinction marked as *[P/S]*. Entry lengths come in double or single precision for floating-point (*doubles* and *singles,* for brevity); marked *[D/S]*; and the integer forms byte, word, doubleword, and quadword, marked *[B/W/D/Q].* Integer forms also sometimes come in signed or unsigned forms, marked *[S/U].* Some instructions work on high or low portions of registers, marked as *[H/L];* other optional components are in the tables. Instructions coming in SSE form and Intel® AVX form are prefixed with a *(V)* for the Intel® AVX form, allowing three operands and 256-bit register support. Entries in square brackets (`[]`) are required; entries in parentheses (`()`) are optional.

Examples:

- `(V)ADD[P/S][D/S]` is the addition of packed or scalar, double or single, with eight possible forms—`VADDPD`, `VADDPS`, `VADDSD`, `VADDSS`, and versions without the leading `V`.
- `(V)[MIN/MAX][P/S][D/S]` represents 16 different instructions for a `min` or `max` of packed or scalar of double or single precision.

The next table represents the multiple comparison types. VEX-prefixed instructions have 32 comparison types; non–VEX-prefixed comparisons only allow those eight types in parentheses. Each comparison type comes in multiple flavors, where `O` = ordered, `U` = unordered, `S` = signaling, and `Q` = non-signaling. Ordered/unordered tells whether the comparison is false or true if one operand is NaN (*Not-a-Number* in floating point, which happens when something failed during the computation, such as divide by 0 or the square root of a negative number). Signaling/non-signaling states whether an exception is fired when at least one operand is QNaN (*Quiet Not-a-Number*—useful for error trapping).

Type	Flavors	Meaning
`EQ`	`(OQ)`, `UQ`, `OS`, `US`	Equal
`LT`	`(OS)`, `OQ`	Less than
`LE`	`(OS)`, `OQ`	Less than or equal to
`UNORD`	`(Q)`, `S`	Tests for unordered (NaN)
`NEQ`	`(UQ)`, `US`, `OQ`, `OS`	Not equal
`NLT`	`(US)`, `UQ`	Not less than
`NLE`	`(US)`, `UQ`	Not less than or equal to
`ORD`	`(Q)`, `S`	Tests for ordered (not NaN)
`NGE`	`US`, `UQ`	Not greater than or equal to

Type	Flavors	Meaning
NGT	US, UQ	Not greater than
FALSE	OQ, OS	Comparison is always false
GE	OS, OQ	Greater than or equal to
GT	OS, OQ	Greater than
TRUE	UQ, US	Comparison is always true

Finally, here are all the Intel® AVX instructions:

Arithmetic	Description
(V)[ADD/SUB/MUL/DIV][P/S][D/S]	Add/subtract/multiply/divide packed/scalar double/single
(V)ADDSUBP[D/S]	Packed double/single add and subtract alternating indices
(V)DPP[D/S]	Dot product, based on immediate mask
(V)HADDP[D/S]	Horizontally add
(V)[MIN/MAX][P/S][D/S]	Min/max packed/scalar double/single
(V)MOVMSKP[D/S]	Extract double/single sign mask
(V)PMOVMSKB	Make a mask consisting of the most significant bits
(V)MPSADBW	Multiple sum of absolute differences
(V)PABS[B/W/D]	Packed absolute value on bytes/words/doublewords
(V)P[ADD/SUB][B/W/D/Q]	Add/subtract packed bytes/words/doublewords/quadwords
(V)PADD[S/U]S[B/W]	Add packed signed/unsigned with saturation bytes/words
(V)PAVG[B/W]	Average packed bytes/words
(V)PCLMULQDQ	Carry-less multiplication quadword
(V)PH[ADD/SUB][W/D]	Packed horizontal add/subtract word/doubleword
(V)PH[ADD/SUB]SW	Packed horizontal add/subtract with saturation
(V)PHMINPOSUW	Min horizontal unsigned word and position
(V)PMADDWD	Multiply and add packed integers
(V)PMADDUBSW	Multiply unsigned bytes and signed bytes into signed words
(V)P[MIN/MAX][S/U][B/W/D]	Min/max of packed signed/unsigned integers
(V)PMUL[H/L][S/U]W	Multiply packed signed/unsigned integers and store high/low result

Arithmetic	Description
(V)PMULHRSW	Multiply packed unsigned with round and shift
(V)PMULHW	Multiply packed integers and store high result
(V)PMULL[W/D]	Multiply packed integers and store low result
(V)PMUL(U)DQ	Multiply packed (un)signed doubleword integers and store quadwords
(V)PSADBW	Compute sum of absolute differences of unsigned bytes
(V)PSIGN[B/W/D]	Change the sign on each element in one operand based on the sign in the other operand
(V)PS[L/R]LDQ	Byte shift left/right amount in operand
(V)SL[L/AR/LR][W/D/Q]	Bit shift left/arithmetic right/logical right
(V)PSUB(U)S[B/W]	Packed (un)signed subtract with (un)signed saturation
(V)RCP[P/S]S	Compute approximate reciprocal of packed/scalar single precision
(V)RSQRT[P/S]S	Compute approximate reciprocal of square root of packed/scalar single precision
(V)ROUND[P/S][D/S]	Round packed/scalar double/single
(V)SQRT[P/S][D/S]	Square root of packed/scalar double/single
VZERO[ALL/UPPER]	Zero all/upper half of YMM registers

Comparison	Description
(V)CMP[P/S][D/S]	Compare packed/scalar double/single
(V)COMIS[S/D]	Compare scalar double/single, set EFLAGS
(V)PCMP[EQ/GT][B/W/D/Q]	Compare packed integers for equality/greater than
(V)PCMP[E/I]STR[I/M]	Compare explicit/implicit length strings, return index/mask

Control	Description
V[LD/ST]MXCSR	Load/store MXCSR control/status register
XSAVEOPT	Save processor extended states optimized

Conversion	Description
(V)CVTx2y	Convert type *x* to type *y*, where *x* and *y* are chosen from

	DQ and P[D/S], [P/S]S and [P/S]D, or S[D/S] and SI.

Load/store	**Description**
VBROADCAST[SS/SD/F128]	Load with broadcast (loads single value into multiple locations)
VEXTRACTF128	Extract 128-bit floating-point values
(V)EXTRACTPS	Extract packed single precision
VINSERTF128	Insert packed floating-point values
(V)INSERTPS	Insert packed single-precision values
(V)PINSR[B/W/D/Q]	Insert integer
(V)LDDQU	Move quad unaligned integer
(V)MASKMOVDQU	Store selected bytes of double quadword with NT Hint
VMASKMOVP[D/S]	Conditional SIMD packed load/store
(V)MOV[A/U]P[D/S]	Move aligned/unaligned packed double/single
(V)MOV[D/Q]	Move doubleword/quadword
(V)MOVDQ[A/U]	Move double to quad aligned/unaligned
(V)MOV[HL/LH]P[D/S]	Move high-to-low/low-to-high packed double/single
(V)MOV[H/L]P[D/S]	Move high/low packed double/single
(V)MOVNT[DQ/PD/PS]	Move packed integers/doubles/singles using a non-temporal hint
(V)MOVNTDQA	Move packed integers using a non-temporal hint, aligned
(V)MOVS[D/S]	Move or merge scalar double/single
(V)MOVS[H/L]DUP	Move single odd/even indexed singles
(V)PACK[U/S]SW[B/W]	Pack with unsigned/signed saturation on bytes/words
(V)PALIGNR	Byte align
(V)PEXTR[B/W/D/Q]	Extract integer
(V)PMOV[S/Z]X[B/W/D][W/D/Q]	Packed move with sign/zero extend (only up in length, DD, DW, etc. disallowed)

Logical	**Description**

Logical	Description
(V)[AND/ANDN/OR]P[D/S]	Bitwise logical AND/AND NOT/OR of packed double/single values
(V)PAND(N)	Logical AND (NOT)
(V)P[OR/XOR]	Bitwise logical OR/exclusive OR
(V)PTEST	Packed bit test, set zero flag if bitwise AND is all 0
(V)UCOMIS[D/S]	Unordered compare scalar doubles/singles and set EFLAGS
(V)XORP[D/S]	Bitwise logical XOR of packed double/single

Shuffle	Description
(V)BLENDP[D/S]	Blend packed double/single; selects elements based on mask
(V)BLENDVP[D/S]	Blend values
(V)MOVDDUP	Copies even values to all values
(V)PBLENDVB	Variable blend packed bytes
(V)PBLENDW	Blend packed words
VPERMILP[D/S]	Permute double/single values
VPERM2F128	Permute floating-point values
(V)PSHUF[B/D]	Shuffle packed bytes/doublewords based on immediate value
(V)PSHUF[H/L]W	Shuffle packed high/low words
(V)PUNPCK[H/L][BW/WD/DQ/QDQ]	Unpack high/low data
(V)SHUFP[D/S]	Shuffle packed double/single
(V)UNPCK[H/L]P[D/S]	Unpack and interleave packed/scalar doubles/singles

AES	Description
AESENC/AESENCLAST	Perform one round of AES encryption
AESDEC/AESDECLAST	Perform one round of AES decryption
AESIMC	Perform the AES InvMixColumn transformation
AESKEYGENASSIST	AES Round Key Generation Assist

Future Instructions	Description
[RD/WR][F/G]SBASE	Read/write FS/GS register

RDRAND	Read random number (into r16, r32, r64)
VCVTPH2PS	Convert 16-bit floats to single precision floating-point values
VCVTPS2PH	Convert single-precision values to 16-bit floating-point values

FMA	**Each `[z]` is the string 132 or 213 or 231, giving the order the operands A,B,C are used in:** **132 is A=AC+B** **213 is A=AB+C** **231 is A=BC+A**
VFMADD[z][P/S][D/S]	Fused multiply add A = r1 * r2 + r3 for packed/scalar of double/single
VFMADDSUB[z]P[D/S]	Fused multiply alternating add/subtract of packed double/single A = r1 * r2 + r3 for odd index, A = r1 * r2-r3 for even
VFMSUBADD[z]P[D/S]	Fused multiply alternating subtract/add of packed double/single A = r1 * r2-r3 for odd index, A = r1 * r2+r3 for even
VFMSUB[z][P/S][D/S]	Fused multiply subtract A = r1 * r2-r3 of packed/scalar double/single
VFNMADD[z][P/S][D/S]	Fused negative multiply add of packed/scalar double/single A = -r1 * r2+r3
VFNMSUB[z][P/S][D/S]	Fused negative multiply subtract of packed/scalar double/single A = -r1 * r2-r3