SIMD Extensions
Contents

Articles

SIMD 1
MMX (instruction set) 6
3DNow! 8
Streaming SIMD Extensions 12
SSE2 16
SSE3 18
SSSE3 20
SSE4 22
SSE5 26
Advanced Vector Extensions 28
CVT16 instruction set 31
XOP instruction set 31

References

Article Sources and Contributors 33
Image Sources, Licenses and Contributors 34

Article Licenses

License 35
**SIMD**

<table>
<thead>
<tr>
<th>Single data</th>
<th>Multiple data</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD</td>
<td>SIMD</td>
</tr>
<tr>
<td>MISD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

**Single instruction, multiple data** (SIMD), is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data simultaneously. Thus, such machines exploit data level parallelism.

**History**

The first use of SIMD instructions was in vector supercomputers of the early 1970s such as the CDC Star-100 and the Texas Instruments ASC, which could operate on a vector of data with a single instruction. Vector processing was especially popularized by Cray in the 1970s and 1980s. Vector-processing architectures are now considered separate from SIMD machines, based on the fact that vector machines processed the vectors one word at a time through pipelined processors (though still based on a single instruction), whereas modern SIMD machines process all elements of the vector simultaneously.[1]

The first era of modern SIMD machines was characterized by massively parallel processing-style supercomputers such as the Thinking Machines CM-1 and CM-2. These machines had many limited-functionality processors that would work in parallel. For example, each of 64,000 processors in a Thinking Machines CM-2 would execute the same instruction at the same time, to do multiplications on 64,000 pairs of numbers at a time.

Supercomputing moved away from the SIMD approach when inexpensive scalar MIMD approaches based on commodity processors such as the Intel i860 XP [2] became more powerful, and interest in SIMD waned.

The current era of SIMD processors grew out of the desktop-computer market rather than the supercomputer market. As desktop processors became powerful enough to support real-time gaming and video processing, demand grew for this particular type of computing power, and microprocessor vendors turned to SIMD to meet the demand. Sun Microsystems introduced SIMD integer instructions in its "VIS" instruction set extensions in 1995, in its UltraSPARC I microprocessor. The first widely-deployed desktop SIMD was with Intel's MMX extensions to the x86 architecture in 1996, followed in 1999 by SSE after IBM and Motorola added AltiVec to the POWER
architecture. Since then, there have been several extensions to the SIMD instruction sets for both architectures. All of these developments have been oriented toward support for real-time graphics, and are therefore oriented toward processing in two, three, or four dimensions, usually with vector lengths of between two and sixteen words, depending on data type and architecture. When new SIMD architectures need to be distinguished from older ones, the newer architectures are then considered "short-vector" architectures, as earlier SIMD and Vector supercomputers had vector lengths from 64 to 64,000. A modern supercomputer is almost always a cluster of MIMD machines, each of which implements (short-vector) SIMD instructions. A modern desktop computer is often a multiprocessor MIMD machine where each processor can execute short-vector SIMD instructions.

**Advantages**

An application that may take advantage of SIMD is one where the same value is being added to (or subtracted from) a large number of data points, a common operation in many multimedia applications. One example would be changing the brightness of an image. Each pixel of an image consists of three values for the brightness of the red (R), green (G) and blue (B) portions of the color. To change the brightness, the R, G and B values are read from memory, a value is added to (or subtracted from) them, and the resulting values are written back out to memory.

With a SIMD processor there are two improvements to this process. For one the data is understood to be in blocks, and a number of values can be loaded all at once. Instead of a series of instructions saying "get this pixel, now get the next pixel", a SIMD processor will have a single instruction that effectively says "get lots of pixels" ("lots" is a number that varies from design to design). For a variety of reasons, this can take much less time than "getting" each pixel individually, as with traditional CPU design.

Another advantage is that SIMD systems typically include only those instructions that can be applied to all of the data in one operation. In other words, if the SIMD system works by loading up eight data points at once, the add operation being applied to the data will happen to all eight values at the same time. Although the same is true for any super-scalar processor design, the level of parallelism in a SIMD system is typically much higher.

**Disadvantages**

- Not all algorithms can be vectorized. For example, a flow-control-heavy task like code parsing wouldn't benefit from SIMD.
- It also has large register files which increases power consumption and chip area.
- Currently, implementing an algorithm with SIMD instructions usually requires human labor; most compilers don't generate SIMD instructions from a typical C program, for instance. Vectorization in compilers is an active area of computer science research. (Compare vector processing.)
- Programming with particular SIMD instruction sets can involve numerous low-level challenges.
  - SSE (Streaming SIMD Extension) has restrictions on data alignment; programmers familiar with the x86 architecture may not expect this.
  - Gathering data into SIMD registers and scattering it to the correct destination locations is tricky and can be inefficient.
  - Specific instructions like rotations or three-operand addition aren't in some SIMD instruction sets.
  - Instruction sets are architecture-specific: old processors and non-x86 processors lack SSE entirely, for instance, so programmers must provide non-vectorized implementations (or different vectorized implementations) for them.
  - The early MMX instruction set shared a register file with the floating-point stack, which caused inefficiencies when mixing floating-point and MMX code. However, SSE2 corrects this.
Chronology

Examples of SIMD supercomputers (not including vector processors):

- ILLIAC IV, circa 1974
- ICL Distributed Array Processor (DAP), circa 1974
- Burroughs Scientific Processor, circa 1976
- Geometric-Arithmetic Parallel Processor, from Martin Marietta, starting in 1981, continued at Lockheed Martin, then at Teranex [3] and Silicon Optix
- Connection Machine, models 1 and 2 (CM-1 and CM-2), from Thinking Machines Corporation, circa 1985
- MasPar MP-1 and MP-2, circa 1987-1996
- Zephyr DC computer from Wavetramer, circa 1991

There were many others from that era too.

Hardware

Small-scale (64 or 128 bits) SIMD has become popular on general-purpose CPUs in the early 1990s and continuing through 1997 and later with Motion Video Instructions (MVI) for Alpha. SIMD instructions can be found, to one degree or another, on most CPUs, including the IBM's AltiVec and SPE for PowerPC, HP's PA-RISC Multimedia Acceleration eXtensions (MAX), Intel's MMX and iwMMXt, SSE, SSE2, SSE3 and SSSE3, AMD's 3DNow!, ARC's ARC Video subsystem, SPARC's VIS and VIS2, Sun's MAJC, ARM's NEON technology, MIPS' MDMX (MaDMaX) and MIPS-3D. The IBM, Sony, Toshiba co-developed Cell Processor's SPU's instruction set is heavily SIMD based. NXP founded by Philips developed several SIMD processors named Xetal. The Xetal has 320 16bit processor elements especially designed for vision tasks.

Modern graphics processing units (GPUs) are often wide SIMD implementations, capable of branches, loads, and stores on 128 or 256 bits at a time.

Intel's AVX SIMD instructions now process 256 bits of data at once. Intel's Larrabee prototype microarchitecture includes more than two 512-bit SIMD registers on each of its cores (VPU: Wide Vector Processing Units), and this 512-bit SIMD capability is being continued in Intel's future Many Integrated Core Architecture (Intel MIC).
Software

SIMD instructions are widely used to process 3D graphics, although modern graphics cards with embedded SIMD have largely taken over this task from the CPU. Some systems also include permute functions that re-pack elements inside vectors, making them particularly useful for data processing and compression. They are also used in cryptography.\[^{4}][^{5}][^{6}\]

The trend of general-purpose computing on GPUs (GPGPU) may lead to wider use of SIMD in the future.

Adoption of SIMD systems in personal computer software was at first slow, due to a number of problems. One was that many of the early SIMD instruction sets tended to slow overall performance of the system due to the re-use of existing floating point registers. Other systems, like MMX and 3DNow!, offered support for data types that were not interesting to a wide audience and had expensive context switching instructions to switch between using the FPU and MMX registers. Compilers also often lacked support, requiring programmers to resort to assembly language coding.

SIMD on x86 had a slow start. The introduction of 3DNow! by AMD and SSE by Intel confused matters somewhat, but today the system seems to have settled down (after AMD adopted SSE) and newer compilers should result in more SIMD-enabled software. Intel and AMD now both provide optimized math libraries that use SIMD instructions, and open source alternatives like libSIMD, SIMDx86 and SLEEF have started to appear.

Apple Computer had somewhat more success, even though they entered the SIMD market later than the rest. AltiVec offered a rich system and can be programmed using increasingly sophisticated compilers from Motorola, IBM and GNU, therefore assembly language programming is rarely needed. Additionally, many of the systems that would benefit from SIMD were supplied by Apple itself, for
example iTunes and QuickTime. However, in 2006, Apple computers moved to Intel x86 processors. Apple's APIs and development tools (XCode) were rewritten to use SSE2 and SSE3 instead of AltiVec. Apple was the dominant purchaser of PowerPC chips from IBM and Freescale Semiconductor and even though they abandoned the platform, further development of AltiVec is continued in several Power Architecture designs from Freescale and IBM.

**SIMD within a register**, or SWAR, is a range of techniques and tricks used for performing SIMD in general-purpose registers on hardware that doesn't provide any direct support for SIMD instructions. This can be used to exploit parallelism in certain algorithms even on hardware that does not support SIMD directly.

### Commercial applications

Though it has generally proven difficult to find sustainable commercial applications for SIMD-only processors, one that has had some measure of success is the GAPP, which was developed by Lockheed Martin and taken to the commercial sector by their spin-off Teranex. The GAPP's recent incarnations have become a powerful tool in real-time video processing applications like conversion between various video standards and frame rates (NTSC to/from PAL, NTSC to/from HDTV formats, etc.), deinterlacing, image noise reduction, adaptive video compression, and image enhancement.

A more ubiquitous application for SIMD is found in video games: nearly every modern video game console since 1998 has incorporated a SIMD processor somewhere in its architecture. The PlayStation 2 was unusual in that one of its vector-float units could function as an autonomous DSP executing its own instruction stream, or as a coprocessor driven by ordinary CPU instructions. 3D graphics applications tend to lend themselves well to SIMD processing as they rely heavily on operations with 4-dimensional vectors. Microsoft's Direct3D 9.0 now chooses at runtime processor-specific implementations of its own math operations, including the use of SIMD-capable instructions.

One of the recent processors to use vector processing is the Cell Processor developed by IBM in cooperation with Toshiba and Sony. It uses a number of SIMD processors (each with independent RAM and controlled by a general purpose CPU) and is geared towards the huge datasets required by 3D and video processing applications.

A recent advancement by Ziilabs was the production of an SIMD type processor which can be used on mobile devices, such as media players and mobile phones.[7]

Larger scale commercial SIMD processors are available from ClearSpeed Technology, Ltd. and Stream Processors, Inc. ClearSpeed's CSX600 (2004) has 96 cores each with 2 double-precision floating point units while the CSX700 (2008) has 192. Stream Processors is headed by computer architect Bill Dally. Their Storm-1 processor (2007) contains 80 SIMD cores controlled by a MIPS CPU.

### See Also

- OpenCL

### References


[4] RE: SSE2 speed (http://marc.info/?l=openssl-dev&m=108530261323715&w=2), showing how SSE2 is used to implement SHA hash algorithms

[5] Salsa20 speed; Salsa20 software (http://cr.yp.to/snuffle.html#speed), showing a stream cipher implemented using SSE2

[6] Subject: up to 1.4x RSA throughput using SSE2 (http://markmail.org/message/tygs74tyjagwwmp4), showing RSA implemented using a non-SIMD SSE2 integer multiply instruction.

External links

- Short Vector Extensions in Commercial Microprocessor (http://www.eecg.toronto.edu/~corinna/vector/svx/)

MMX (instruction set)

MMX is a single instruction, multiple data (SIMD) instruction set designed by Intel, introduced in 1996 with their P5-based Pentium line of microprocessors, designated as "Pentium with MMX Technology". It developed out of a similar unit introduced on the Intel i860, and earlier the Intel i750 video pixel processor. MMX is a processor supplementary capability that is supported on recent IA-32 processors by Intel and other vendors.

Naming

MMX is officially a meaningless initialism trademarked by Intel; unofficially, the initials have been variously explained as standing for MultiMedia eXtension, Multiple Math eXtension, or Matrix Math eXtension.

AMD, during one of its numerous court battles with Intel, produced marketing material from Intel indicating that MMX stood for "Matrix Math Extensions". Since an initialism cannot be trademarked, this was an attempt to invalidate Intel's trademark. In 1997, Intel filed suit against AMD and Cyrix Corp. for misuse of its trademark MMX. AMD and Intel settled, with AMD acknowledging MMX as a trademark owned by Intel, and with Intel granting AMD rights to use the MMX trademark as a technology name, but not a processor name.

Technical details

MMX defined eight registers, known as MM0 through MM7 (henceforth referred to as M Mn). To avoid compatibility problems with the context switch mechanisms in existing operating systems, these registers were aliases for the existing x87 FPU stack registers (so no new registers needed to be saved or restored). Hence, anything that was done to the floating point stack would also affect the MMX registers and vice versa. However, unlike the FP stack, the MMn registers are directly addressable (random access).

Each of the MMn registers holds 64 bits (the mantissa-part of a full 80-bit FPU register). The main usage of the MMX instruction set is based on the concept of packed data types, which means that instead of using the whole register for a single 64-bit integer, two 32-bit integers, four 16-bit integers, or eight 8-bit integers may be processed concurrently.

The mapping of the MMX registers onto the existing FPU registers made it somewhat difficult to work with floating point and SIMD data in the same application. To maximize performance, programmers often used the processor exclusively in one mode or the other, deferring the relatively slow switch between them as long as possible.

Because the FPU stack registers are 80 bits wide, the upper 16 bits of the stack registers go unused in MMX, and these bits are all set to ones, making them NaNs or infinities in the floating point representation. This can be used to
MMX (instruction set)

decide whether a particular register's content is intended as floating point or SIMD data.

MMX provides only integer operations. When originally developed, for the Intel i860, the use of integer math made sense (both 2D and 3D calculations required it), but as graphics cards that did much of this became common, integer SIMD in the CPU became somewhat redundant for graphical applications. On the other hand, the saturation arithmetic operations in MMX could significantly speed up some digital signal processing applications.

**Successor**

AMD, a competing x86 microprocessor vendor, enhanced Intel's MMX with their own 3DNow! instruction set. 3DNow is best known for adding single-precision (32-bit) floating-point support to the SIMD instruction-set, among other integer and more general enhancements.

Following MMX, Intel's next major x86 extension was the SSE, introduced with the Pentium-III family (roughly a year after AMD's 3DNow! was introduced.)

SSE addressed the core shortcomings of MMX (inability to mix integer-SIMD ops with any floating-point ops) by creating a new 128-bit wide register file (XMM0 - XMM7) and new SIMD instructions for it. Like 3DNow, SSE focused exclusively on single-precision floating-point operations (32-bit); integer SIMD operations were still performed using the MMX register and instruction set. However, the new XMM register-file allowed SSE SIMD-operations to be freely mixed with either MMX or x87 FPU ops.

SSE2, introduced with the Pentium 4, further extended the x86 SIMD instruction set with integer (8/16/32 bit) and double-precision floating-point data support for the XMM register file. SSE2 also allowed the MMX opcodes to use XMM register operands, but ended this support with SSE4 (and recently with SSE4.2, introduced in the Core microarchitecture.) However, since processor support for any SSE revision also implies support for MMX, the removal does not limit the types of data types usable by x86 SIMD.

**MMX in embedded applications**

Intel's and Marvell's XScale microprocessor core starting with PXA270 include an SIMD instruction set extension to the ARM core called iwMMXt whose functions are similar to those of the IA-32 MMX extension. iwMMXt stands for "Intel Wireless MMX Technology". It provides arithmetic and logic operations on 64-bit integer numbers (the software may choose to instead perform two 32-bit, four 16-bit or eight 8-bit operations in a single instruction). The extension contains 16 data registers of 64-bits and eight control registers of 32-bits. All registers are accessed through standard ARM architecture coprocessor mapping mechanism. iwMMXt occupies coprocessors 0 and 1 space, and some of its opcodes clash with the opcodes of the earlier floating-point extension, FPA.

Later versions of Marvell's ARM processors supports both WMMX (Wireless MMX)and WMMX2 (Wireless MMX2) support.

**References**


[3] Controversy brews over use of MMX moniker - Intel's multimedia extension - Industry Trend or Event (http://findarticles.com/p/articles/mi_m0EKF/is_2149_v43/ai_19010184)

External links

- Intel Pentium Processor with MMX Technology Documentation (http://www.intel.com/design/archives/Processors/mmx/)
- The MMX Instruction Set (http://webster.cs.ucr.edu/AoA/Windows/HTML/TheMMXInstructionSet.html) from The Art of Assembly Language (http://webster.cs.ucr.edu/AoA/Windows/HTML/AoATOC.html)

3DNow!

3DNow! is an extension to the x86 instruction set developed by Advanced Micro Devices (AMD). It adds single instruction multiple data (SIMD) instructions to the base x86 instruction set, enabling it to perform simple vector processing, which improves the performance of many graphic-intensive applications. The first microprocessor to implement 3DNow! was the AMD K6-2, which was introduced in 1998. When the application was appropriate this raised the speed by about 2-4 times.\(^1\) However the instruction set never gained much popularity, and AMD announced on August 2010 that support for 3DNow! will be dropped in future AMD processors, except for two instructions.\(^2\)

History

3DNow! was developed at a time when 3D graphics were becoming mainstream in PC multimedia and gaming software. Realtime display of 3D graphics depended heavily on the host CPU's floating-point unit (FPU) to perform floating-point calculations, a task in which AMD's K6 processor was easily outperformed by its competitor, the Intel Pentium-II.

As an enhancement to the MMX instruction set, the 3DNow! instruction-set augmented the MMX SIMD registers to support common arithmetic operations (add/subtract/multiply) on single-precision (32-bit) floating-point data. Software written to use AMD's 3DNow! instead of the slower x87 FPU could execute up to 4x faster, depending on the instruction-mix.

Versions

3DNow!

The first implementation of 3DNow! technology contains 21 new instructions that support SIMD floating-point operations. The 3DNow! data format is packed, single-precision, floating-point. The 3DNow! instruction set also includes operations for SIMD integer operations, data prefetch, and faster MMX-to-floating-point switching. Later, Intel would add similar (but incompatible) instructions to the Pentium III, known as SSE for Streaming SIMD Extensions.

3DNow! floating-point instructions

- PI2FD - Packed 32-bit integer to floating-point conversion
- PF2ID - Packed floating-point to 32-bit integer conversion
- PFCMPGE - Packed floating-point comparison, greater or equal
- PFCMPGT - Packed floating-point comparison, greater
- PFCMPEQ - Packed floating-point comparison, equal
- PFACC - Packed floating-point accumulate
- PFADD - Packed floating-point addition
3DNow!

- PFSUB - Packed floating-point subtraction
- PFSUBR - Packed floating-point reverse subtraction
- PFMN - Packed floating-point minimum
- PFMX - Packed floating-point maximum
- PFML - Packed floating-point multiplication
- PFRCP - Packed floating-point reciprocal approximation
- PFRSQRT - Packed floating-point reciprocal square root approximation
- PFRCPIT1 - Packed floating-point reciprocal, first iteration step
- PFRSQIT1 - Packed floating-point reciprocal square root, first iteration step
- PFRCPIT2 - Packed floating-point reciprocal/reciprocal square root, second iteration step

3DNow! integer instructions
- PAVGUSB - Packed 8-bit unsigned integer averaging
- PMULHRW - Packed 16-bit integer multiply with rounding

3DNow! performance-enhancement instructions
- FEMMS - Faster entry/exit of the MMX or floating-point state
- PREFETCH/PREFETCHW - Prefetch at least a 32-byte line into L1 data cache (this is the non-deprecated instruction)

3DNow! extensions

There is little or no evidence that the second version of 3DNow! was ever officially given its own trade name. This has led to some confusion in documentation that refers to this new instruction set. The most common terms are Extended 3DNow!, Enhanced 3DNow! and 3DNow!+. The phrase "Enhanced 3DNow!" can be found in a few locations on the AMD website but the capitalization of "Enhanced" appears to be either purely grammatical or used for emphasis on processors that may or may not have these extensions (the most notable of which references a benchmark page for the K6-III-P that does not have these extensions).[3][4]

This extension to the 3DNow! instruction set was introduced with the first-generation Athlon processors. The Athlon added 5 new 3DNow! instructions and 19 new MMX instructions. Later, the K6-2+ and K6-III+ (both targeted at the mobile market) included the 5 new 3DNow! instructions, leaving out the 19 new MMX instructions. The new 3DNow! instructions were added to boost DSP. The new MMX instructions were added to boost streaming media.

3DNow! or MMX extensions?

The 19 new MMX instructions are a subset of Intel's SSE1 instruction set. In AMD technical manuals, AMD segregates these instructions apart from the 3DNow! extensions. In AMD customer product literature, however, this segregation is less clear where the benefits of all 24 new instructions are credited to enhanced 3DNow! technology.[5] This has led programmers to come up with their own name for the 19 new MMX instructions. The most common appears to be Integer SSE (ISSE).[6] SSEMMX and MMX2 are also found in video filter documentation from the public domain sector. [It should also be noted that ISSE could also refer to Internet SSE, an early name for SSE.]

3DNow! extension DSP instructions
- PF2IW - Packed floating-point to integer word conversion with sign extend
- PI2FW - Packed integer word to floating-point conversion
- PFNACC - Packed floating-point negative accumulate
- PFPNACC - Packed floating-point mixed positive-negative accumulate
- PSWAPD - Packed swap doubleword

MMX extension instructions (Integer SSE)
- MASKMOVQ - Streaming (cache bypass) store using byte mask
3DNow! Professional

3DNow! Professional is a trade name used to indicate processors that combine 3DNow! technology with a complete SSE instructions set (such as SSE1, SSE2 or SSE3). The Athlon XP was the first processor to carry the 3DNow! Professional trade name, and was the first product in the Athlon family to support the complete SSE1 instruction set (for the total of: 21 original 3DNow! instructions; 5 3DNow! extension DSP instructions; 19 MMX extension instructions; and 52 additional SSE instructions for complete SSE1 compatibility).

3DNow! and the Geode GX/LX

The Geode GX and Geode LX added two new 3DNow! instructions which are currently absent in all the other processors.

3DNow! Professional instructions unique to the Geode GX/LX

- PFRSQRTV - Reciprocal square root approximation for a pair of 32-bit floats
- PFRCPVF - Reciprocal approximation for a pair of 32-bit floats

Advantages and disadvantages

One advantage of 3DNow! is that it is possible to add or multiply the two numbers that are stored in the same register. With SSE, each number can only be combined with a number in the same position in another register. This capability, known as horizontal in Intel terminology, was the major addition to the SSE3 instruction set.

A disadvantage with 3DNow! is that 3DNow instructions and MMX instructions share the same register-file, whereas SSE adds 8 new independent registers (XMM0 - XMM7.) Because MMX/3DNow! registers are shared by the standard x87 FPU, 3DNow! instructions and x87 instructions cannot be executed simultaneously. However, because it is aliased to the x87 FPU, the 3DNow! & MMX register states can be saved and restored by the traditional x87 F(N)SAVE and F(N)RSTOR instructions. This arrangement allowed operating systems to support 3DNow! with no explicit modifications, whereas SSE registers required explicit operating system support to properly save and restore the new XMM registers (via the added FXSAVE and FXRSTOR instructions.)
The FX* instructions are an upgrade to the older x87 save and restore instructions because these could save not only SSE register states but also those x87 register states (hence which meant that it could save MMX and 3DNow! registers too).

On AMD Athlon XP and K8-based cores (i.e. Athlon 64), assembly programmers have noted that it is possible to combine 3DNow! and SSE instructions to reduce register pressure, but in practice it is difficult to improve performance due to the instructions executing on shared functional units.[9]

**Processors supporting 3DNow!**

- All AMD processors after K6-2 (inclusive) up to August 2010. Discontinued for future AMD processors.
- National Semiconductor Geode, later AMD Geode.
- VIA C3 (also known as Cyrix III) "Samuel", "Samuel 2" "Ezra", and "Eden ESP" cores.
- IDT Winchip 2

**References**


Further reading


External links

- AMD Extensions to the 3DNow! and MMX Instruction Sets Manual (http://support.amd.com/us/Processor_TechDocs/22466.pdf)
- AMD Geode LX Processors Data Book (http://support.amd.com/it/Embedded_TechDocs/33234d_lx_ds.pdf)
Streaming SIMD Extensions

In computing, Streaming SIMD Extensions (SSE) is an SIMD instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in their Pentium III series processors as a reply to AMD's 3DNow! (which had debuted a year earlier). SSE contains 70 new instructions, most of which work on single precision floating point data. SIMD instructions can greatly increase performance when exactly the same operations are to be performed on multiple data objects. Typical applications are digital signal processing and graphics processing.

Intel’s first IA-32 SIMD effort was the MMX instruction set. MMX had two main problems: it re-used existing floating point registers making the CPU unable to work on both floating point and SIMD data at the same time, and it only worked on integers. SSE floating point instructions operate on a new independent register set (the XMM registers), and it adds a few integer instructions that work on MMX registers.

SSE was subsequently expanded by Intel to SSE2, SSE3, SSSE3, and SSE4. Because it supports floating point math, it had a wider application than MMX and became more popular. The addition of integer support in SSE2 made MMX largely redundant, though further performance increases can be attained in some situations by using MMX in parallel with SSE operations.

SSE was originally known as KNI for Katmai New Instructions (Katmai being the code name for the first Pentium III core revision). During the Katmai project Intel sought to distinguish it from their earlier product line, particularly their flagship Pentium II. It was later renamed ISSE, for Internet Streaming SIMD Extensions, then SSE. AMD eventually added support for SSE instructions, starting with its Athlon XP and Duron (Morgan core) processors.

Registers

SSE originally added eight new 128-bit registers known as XMM0 through XMM7. The AMD64 extensions from AMD (originally called x86-64) added a further eight registers XMM8 through XMM15, and this extension is duplicated in the Intel 64 architecture. There is also a new 32-bit control/status register, MXCSR. The registers XMM8 through XMM15 are accessible only in 64-bit operating mode.

SSE used only a single data type for XMM registers:

• four 32-bit single-precision floating point numbers

SSE2 would later expand the usage of the XMM registers to include:

• two 64-bit double-precision floating point numbers or
• two 64-bit integers or
• four 32-bit integers or
• eight 16-bit short integers or
• sixteen 8-bit bytes or characters.

Because these 128-bit registers are additional program states that the operating system must preserve across task switches, they are disabled by default until the operating system
explicitly enables them. This means that the OS must know how to use the FXSAVE and FXRSTOR instructions, which is the extended pair of instructions which can save all x86 and SSE register states all at once. This support was quickly added to all major IA-32 operating systems.

The first CPU to support SSE, the Pentium III, shared execution resources between SSE and the FPU. While a compiled application can interleave FPU and SSE instructions side-by-side, the Pentium III will not issue an FPU and an SSE instruction in the same clock-cycle. This limitation reduces the effectiveness of pipelining, but the separate XMM registers do allow SIMD and scalar floating point operations to be mixed without the performance hit from explicit MMX/floating point mode switching.

**SSE Instructions**

SSE introduced both scalar and packed floating point instructions.

**Floating point instructions**

- Memory-to-register/register-to-memory/register-to-register data movement
  - Scalar – MOVSS
  - Packed – MOVAPS, MOVUPS, MOVLP, MOVHPS, MOVHLP, MOVHLPS
- Arithmetic
  - Scalar – ADDSS, SUBSS, MULSS, DIVSS, RCPSS, SQRTSS, MAXSS, MINSS, RSQRTSS
  - Packed – ADDPS, SUBPS, MULPS, DIVPS, RCPPS, SQRTPS, MAXPS, MINPS, RSQRTPS
- Compare
  - Scalar – CMPSS, COMISS, UCOMISS
  - Packed – CMPPS
- Data shuffle and unpacking
  - Packed – SHUFPS, UNPCKHPS, UNPCKLPS
- Data type conversion
  - Scalar – CVTSI2SS, CVTSS2SI, CVTTSS2SI
  - Packed – CVTPS2PS, CVTPS2PS, CVTTPS2PS
- Bitwise logical operations
  - Packed – ANDPS, ORPS, XORPS, ANDNPS

**Integer instructions**

- Arithmetic
  - PMULHUW, PSADBW, PAVGB, PAVGW, PMAXUB, PMINUB, PMAXSW, PMINSW
- Data movement
  - PEXTRW, PINSRW
- Other
  - PMOVMSKB, PSHUFW
Streaming SIMD Extensions

Other instructions
- MXCSR management
  - LDMXCSR, STMXCSR
- Cache and Memory management
  - MOVNTQ, MOVNTPS, MASKMOVQ, PREFETCH0, PREFETCH1, PREFETCH2, PREFETCHNTA, SFENCE

Example
The following simple example demonstrates the advantage of using SSE. Consider an operation like vector addition, which is used very often in computer graphics applications. To add two single precision, four-component vectors together using x86 requires four floating-point addition instructions:

```plaintext
vec_res.x = v1.x + v2.x;
vec_res.y = v1.y + v2.y;
vec_res.z = v1.z + v2.z;
vec_res.w = v1.w + v2.w;
```

This would correspond to four x86 FADD instructions in the object code. On the other hand, as the following pseudo-code shows, a single 128-bit 'packed-add' instruction can replace the four scalar addition instructions:

```plaintext
movaps xmm0, [v1]; xmm0 = v1.w | v1.z | v1.y | v1.x
addps xmm0, [v2]; xmm0 = v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x
movaps [vec_res], xmm0
```

Later versions
- SSE2, introduced with the Pentium 4, is a major enhancement to SSE. SSE2 adds new math instructions for double-precision (64-bit) floating point and also extends MMX integer instructions to operate on 128-bit XMM registers. Until SSE2, SSE integer instructions introduced with later SSE extensions could still operate on 64-bit MMX registers because the new XMM registers require operating system support. SSE2 enables the programmer to perform SIMD math on any data type (from 8-bit integer to 64-bit float) entirely with the XMM vector-register file, without the need to use the legacy MMX or FPU registers. Many programmers consider SSE2 to be "everything SSE should have been", as SSE2 offers an orthogonal set of instructions for dealing with common data types.
- SSE3, also called Prescott New Instructions (PNI), is an incremental upgrade to SSE2, adding a handful of DSP-oriented mathematics instructions and some process (thread) management instructions.
- SSSE3 is an incremental upgrade to SSE3, adding 16 new instructions which include permuting the bytes in a word, multiplying 16-bit fixed-point numbers with correct rounding, and within-word accumulate instructions. SSSE3 is often mistaken for SSE4 as this term was used during the development of the Core microarchitecture.
- SSE4 is another major enhancement, adding a dot product instruction, additional integer instructions, a popcnt instruction, and more.
- XOP, FMA4 and CVT16 are new iterations announced by AMD in August 2007[1][2] and revised in May 2009.[3]
- AVX (Advanced Vector Extensions) is an advanced version of SSE announced by Intel featuring a widened data path from 128 bits to 256 bits and 3-operand instructions (up from 2). Intel released processors in early 2011 with
Software and hardware issues
With all x86 instruction set extensions, it is up to the BIOS, operating system and application programmer to test and detect their existence and proper operation.

- Intel and AMD offer applications to detect what extensions your CPU supports.
- The CPUID opcode is a processor supplementary instruction (its name derived from CPU IDentification) for the x86 architecture. It was introduced by Intel in 1993 when it introduced the Pentium and SL-Enhanced 486 processors.

User application uptake of the x86 extensions has been slow with even bare minimum baseline MMX and SSE support (in some cases) not being supported by applications some 10 years after these extensions became commonly available. Distributed computing has accelerated the use of these extensions in the scientific community—and many scientific applications refuse to run unless the CPU supports SSE2 or SSE3.

The use of multiple revisions of an application to cope with the many different sets of extensions available is the simplest way around the x86 extension optimization problem. Software libraries and some applications have begun to support multiple extension types hinting that full use of available x86 instructions may finally become common some 5 to 15 years after the instructions were initially introduced.

Related links
Processor ID applications
- Intel Processor Identification Utility [5]
- CPU-Z CPU, motherboard and memory identifier utility
- BOINC that identifies AMD and Intel x86 features.

References
**SSE2**

**SSE, Streaming SIMD Extensions 2**, is one of the Intel SIMD (Single Instruction, Multiple Data) processor supplementary instruction sets first introduced by Intel with the initial version of the Pentium 4 in 2001. It extends the earlier SSE instruction set, and is intended to fully supplant MMX. Intel extended SSE2 to create SSE3 in 2004. SSE2 added 144 new instructions to SSE, which has 70 instructions. Rival chip-maker AMD added support for SSE2 with the introduction of their Opteron and Athlon 64 ranges of AMD64 64-bit CPUs in 2003.

**Changes**

SSE2 extends MMX instructions to operate on XMM registers, allowing the programmer to completely avoid the eight 64-bit MMX registers "aliased" on the original IA-32 floating point register stack. This permits mixing integer SIMD and scalar floating point operations without the mode switching required between MMX and x87 floating point operations. However, this is over-shadowed by the value of being able to perform MMX operations on the wider SSE registers.

Other SSE2 extensions include a set of cache-control instructions intended primarily to minimize cache pollution when processing indefinite streams of information, and a sophisticated complement of numeric format conversion instructions.

AMD's implementation of SSE2 on the AMD64 (x86-64) platform includes an additional eight registers, doubling the total number to 16 (XMM0 through XMM15). These additional registers are only visible when running in 64-bit mode. Intel adopted these additional registers as part of their support for x86-64 architecture (or in Intel's parlance, "Intel 64") in 2004.

**Differences between x87 FPU and SSE2**

FPU (x87) instructions provide higher precision by calculating intermediate results with 80 bits of precision, by default, to minimise roundoff error in numerically unstable algorithms (see IEEE 754 design rationale and references therein). However, the x87 FPU is a scalar unit only whereas SSE2 which can process a small vector of operands in parallel.

If codes designed for x87 are ported to the lower precision double precision SSE2 floating point, certain combinations of math operations or input datasets can result in measurable numerical deviation, which can be an issue in reproducible scientific computations, e.g. if the calculation results must be compared against results generated from a different machine architecture. A related issue is that, historically, language standards and compilers had been inconsistent in their handling of the x87 80-bit registers implementing double extended precision variables, compared with the double and single precision formats implemented in SSE2: the rounding of extended precision intermediate values to double precision variables was not fully defined and was dependent on implementation details such as when registers were spilled to memory. However, modern language standards such as C99 and Fortran 2003 have incorporated IEEE 754 floating point support and now exactly specify the semantics of double extended ("long double") precision expressions to avoid such reproducibility problems.
**Differences between MMX and SSE2**

SSE2 extends MMX instructions to operate on XMM registers. Therefore, it is possible to convert all existing MMX code to an SSE2 equivalent. Since an XMM register is twice as long as an MMX register, loop counters and memory access may need to be changed to accommodate this. However, 8 byte loads and stores to XMM are available, so this is not strictly required.

Although one SSE2 instruction can operate on twice as much data as an MMX instruction, performance might not increase significantly. Two major reasons are: accessing SSE2 data in memory not aligned to a 16-byte boundary can incur significant penalty, and the throughput of SSE2 instructions in older x86 implementations was half that for MMX instructions. Intel addressed the first problem by adding an instruction in SSE3 to reduce the overhead of accessing unaligned data and improving the overall performance of misaligned loads, and the last problem by widening the execution engine in their Core microarchitecture in Core 2 Duo and later products.

Since MMX and x87 register files alias one another, using MMX will prevent x87 instructions from working as desired. Once MMX has been used, the programmer must use the emms instruction (C: _mm_empty()) to restore operation to the x87 register file. On some operating systems, x87 is not used very much, but may still be used in some critical areas like pow() where the extra precision is needed. In such cases, the corrupt floating-point state caused by failure to emit emms may go undetected for millions of instructions before ultimately causing the floating-point routine to fail, returning NaN. Since the problem is not locally apparent in the MMX code, the bug can be very time consuming to find and correct. As SSE2 does not have this problem, usually provides much better throughput and provides more registers in 64-bit code, it should be preferred for nearly all vectorization work.

**Compiler usage**

When first introduced in 2000, SSE2 was not supported by software development tools. For example, to use SSE2 in a Microsoft Developer Studio project, the programmer had to either manually write inline-assembly or import object-code from an external source. Later the Visual C++ Processor Pack added SSE2 support to Visual C++ and MASM.

The Intel C++ Compiler can automatically generate SSE4/SSSE3/SSE3/SSE2 and/or SSE-code without the use of hand-coded assembly.

Since GCC 3, GCC can automatically generate SSE/SSE2 scalar code when the target supports those instructions. Automatic vectorization for SSE/SSE2 has been added since GCC 4.

The Sun Studio Compiler Suite can also generate SSE2 instructions when the compiler flag -xvector=simd is used.

**CPUs supporting SSE2**

- AMD K8-based CPUs (Athlon 64, Sempron 64, Turion 64, etc.)
- AMD Phenom CPUs
- Intel NetBurst-based CPUs (Pentium 4, Xeon, Celeron, Celeron D, etc.)
- Intel Pentium M and Celeron M
- Intel Core family (including Intel Core 2, Intel Core i5, Intel Core i7)
- Intel Atom
- Transmeta Efficeon
- VIA C7
- VIA Nano
Notable IA-32 CPUs not supporting SSE2

SSE2 is an extension of the IA-32 architecture. Therefore any architecture that does not support IA-32 does not support SSE2. x86-64 CPUs all implement IA-32. All known x86-64 CPUs also implement SSE2. Since IA-32 predates SSE2, early IA-32 CPUs did not implement it. SSE2 and the other SIMD instruction sets were intended primarily to improve CPU support for realtime graphics, notably gaming. A CPU that is not marketed for this purpose or that has an alternative SIMD instruction set has no need for SSE2.

The following CPUs implemented IA-32 after SSE2 was developed, but did not implement SSE2:

- AMD CPUs prior to Athlon 64, including all Socket A-based CPUs
- Intel CPUs prior to Pentium 4
- VIA C3
- Transmeta Crusoe

SSE3

SSE3, Streaming SIMD Extensions 3, also known by its Intel code name Prescott New Instructions (PNI), is the third iteration of the SSE instruction set for the IA-32 (x86) architecture. Intel introduced SSE3 in early 2004 with the Prescott revision of their Pentium 4 CPU. In April 2005, AMD introduced a subset of SSE3 in revision E (Venice and San Diego) of their Athlon 64 CPUs. The earlier SIMD instruction sets on the x86 platform, from oldest to newest, are MMX, 3DNow! (developed by AMD), SSE and SSE2.

SSE3 contains 13 new instructions over SSE2.

Changes

The most notable change is the capability to work horizontally in a register, as opposed to the more or less strictly vertical operation of all previous SSE instructions. More specifically, instructions to add and subtract the multiple values stored within a single register have been added. These instructions simplify the implementation of a number of DSP and 3D operations. There is also a new instruction to convert floating point values to integers without having to change the global rounding mode, thus avoiding costly pipeline stalls. Finally, the extension adds LDDQU, an alternative misaligned integer vector load that has better performance on NetBurst based platforms for loads that cross cacheline boundaries.

CPUs with SSE3

- AMD:
  - Athlon 64 (since Venice Stepping E3 and San Diego Stepping E4)
  - Athlon 64 X2
  - Athlon 64 FX (since San Diego Stepping E4)
  - Opteron (since Stepping E4)
  - Sempron (since Palermo. Stepping E3)
  - Phenom
  - Phenom II
  - Athlon II
  - Turion 64
  - Turion 64 X2
- Intel:
  - Celeron D
- Celeron (starting with Core microarchitecture)
- Pentium 4 (since Prescott)
- Pentium D
- Pentium Extreme Edition (but NOT Pentium 4 Extreme Edition)
- Pentium Dual-Core
- Pentium (starting with Core microarchitecture)
- Core
- Xeon (since Nocona)
- Atom
- VIA/Centaur:
  - C7
  - Nano
- Transmeta
  - Efficeon TM88xx (NOT Model Numbers TM86xx)

### New instructions

#### Common instructions

##### Arithmetic

- **ADDSUBPD** — *(Add-Subtract-Packed-Double)*
  - Input: { A0, A1 }, [ B0, B1 ]
  - Output: { A0 − B0, A1 + B1 }

- **ADDSUBPS** — *(Add-Subtract-Packed-Single)*
  - Input: { A0, A1, A2, A3 }, [ B0, B1, B2, B3 ]
  - Output: { A0 − B0, A1 + B1, A2 − B2, A3 + B3 }

##### AOS (Array Of Structures)

- **HADDPPD** — *(Horizontal-Add-Packed-Double)*
  - Input: { A0, A1 }, [ B0, B1 ]
  - Output: { A0 + A1, B0 + B1 }

- **HADDPS** *(Horizontal-Add-Packed-Single)*
  - Input: { A0, A1, A2, A3 }, [ B0, B1, B2, B3 ]
  - Output: { A0 + A1, A2 + A3, B0 + B1, B2 + B3 }

- **HSUBPD** — *(Horizontal-Subtract-Packed-Double)*
  - Input: { A0, A1 }, [ B0, B1 ]
  - Output: { A0 − A1, B0 − B1 }

- **HSUBPS** — *(Horizontal-Subtract-Packed-Single)*
  - Input: { A0, A1, A2, A3 }, [ B0, B1, B2, B3 ]
  - Output: { A0 − A1, A2 − A3, B0 − B1, B2 − B3 }

- **LDDQU** — As stated above, this is an alternative misaligned integer vector load. It can be helpful for video compression tasks.

- **MOVDDUP, MOVSHDUP, MOVSLDUP** — These are also used for complex numbers, and can be helpful for wave calculation like sound.

- **FISTTP** — Like the older x87 FISTP instruction, but ignores the floating point control register's rounding mode settings and uses the "chop" (truncate) mode instead. Allows omission of the expensive loading and re-loading of the control register in languages such as C where float-to-int conversion requires truncate behaviour by standard.
Intel instructions

- MONITOR, MWAIT - These optimize multi-threaded applications, giving processors with Hyper-Threading better performance.

External links

- X-bit Labs [1]

References


SSSE3

Supplemental Streaming SIMD Extensions 3 (SSSE3) is a SIMD instruction set created by Intel and is the fourth iteration of the SSE technology.

History

SSSE3 was first introduced with Intel processors based on the Core microarchitecture on 26 June 2006 with the "Woodcrest" Xeons.

SSSE3 has been referred to by the codenames Tejas New Instructions (TNI) or Merom New Instructions (MNI) for the first processor designs intended to support it.

Functionality

SSSE3 contains 16 new discrete instructions.

Each instruction can act on 64-bit MMX or 128-bit XMM registers. Therefore, Intel's materials refer to 32 new instructions.

CPUs with SSSE3

- AMD:
  - Bobcat
  - Bulldozer
- Intel:
  - Xeon 5100 Series
  - Xeon 5300 Series
  - Xeon 3000 Series
  - Core 2 Duo
  - Core 2 Extreme
  - Core 2 Quad
  - Core i7
  - Core i5
  - Core i3
  - Pentium Dual Core (NOT "Pentium D")
  - Celeron 4xx Sequence Conroe-L
  - Celeron Dual Core E1200
• Celeron M 500 series
• Atom
• VIA:
• Nano

New Instructions

In the table below, satsw(X) (read as 'saturate to signed word') takes a signed integer X, and converts it to $-32768$ if it's less than $-32768$, to $+32767$ if it's greater than $32767$, and leaves it unchanged otherwise. As normal for the Intel architecture, bytes are 8 bits, words 16 bits, and dwords 32 bits; 'register' refers to an MMX or XMM vector register.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSIGNB, PSIGNW, PSIGND</td>
<td>Packed Sign</td>
<td>Negate the elements of a register of bytes, words or dwords if the sign of the corresponding elements of another register is negative.</td>
</tr>
<tr>
<td>PABS, PABSW, PABSD</td>
<td>Packed Absolute Value</td>
<td>Fill the elements of a register of bytes, words or dwords with the absolute values of the elements of another register.</td>
</tr>
<tr>
<td>PALIGNR</td>
<td>Packed Align Right</td>
<td>take two registers, concatenate their values, and pull out a register-length section from an offset given by an immediate value encoded in the instruction.</td>
</tr>
<tr>
<td>PSHUFB</td>
<td>Packed Shuffle Bytes</td>
<td>takes registers of bytes $A = [a_0 a_1 a_2 ...]$ and $B = [b_0 b_1 b_2 ...]$ and replaces $A$ with $[a_0 a_1 a_2 ...]$; except that it replaces the ith entry with 0 if the top bit of $b_i$ is set.</td>
</tr>
<tr>
<td>PMULHRSW</td>
<td>Packed Multiply High with Round and Scale</td>
<td>treat the sixteen-bit words in registers $A$ and $B$ as signed 15-bit fixed-point numbers between $-1$ and $1$ (e.g. $0x4000$ is treated as $0.5$ and $0xa000$ as $-0.75$), and multiply them together with correct rounding.</td>
</tr>
<tr>
<td>PMADDBSW</td>
<td>Multiply and Add Packed Signed and Unsigned Bytes</td>
<td>Take the bytes in registers $A$ and $B$, multiply them together, add pairs, signed-saturate and store. I.e. $[a_0 a_1 a_2 ...]$ pmaddbsw $[b_0 b_1 b_2 ...] = [\text{satsw}(a_0 b_0 + a_1 b_1) \text{satsw}(a_2 b_2 + a_3 b_3) ...]$</td>
</tr>
<tr>
<td>PHSUBW, PHSUBD</td>
<td>Packed Horizontal Subtract (Words or Doublewords)</td>
<td>takes registers $A = [a_0 a_1 a_2 ...]$ and $B = [b_0 b_1 b_2 ...]$ and outputs $[a_0-a_1 a_2-a_3 ... b_0-b_1 b_2-b_3 ...]$</td>
</tr>
<tr>
<td>PHSUBSW</td>
<td>Packed Horizontal Subtract and Saturate Words</td>
<td>like PHSUBW, but outputs $[\text{satsw}(a_0-a_1) \text{satsw}(a_2-a_3) ... \text{satsw}(b_0-b_1) \text{satsw}(b_2-b_3) ...]$</td>
</tr>
<tr>
<td>PHADDW, PHADDD</td>
<td>Packed Horizontal Add (Words or Doublewords)</td>
<td>takes registers $A = [a_0 a_1 a_2 ...]$ and $B = [b_0 b_1 b_2 ...]$ and outputs $[a_0+a_1 a_2+a_3 ... b_0+b_1 b_2+b_3 ...]$</td>
</tr>
<tr>
<td>PHADDSW</td>
<td>Packed Horizontal Add and Saturate Words</td>
<td>like PHADDD, but outputs $[\text{satsw}(a_0+a_1) \text{satsw}(a_2+a_3) ... \text{satsw}(b_0+b_1) \text{satsw}(b_2+b_3) ...]$</td>
</tr>
</tbody>
</table>

References

External links

• Core 2 Mobile specifications (http://download.intel.com/design/mobile/datashts/31407801.pdf)
• Intel white-paper admitting the existence of SSSE3 and describing SSE4 (ftp://download.intel.com/technology/architecture/new-instructions-paper.pdf)
• Instruction set documentation listing the functions of the SSSE3 instructions (http://www.intel.com/design/processor/manuals/253667.pdf)
SSE4

SSE4 (Streaming SIMD Extensions 4) is a CPU instruction set used in the Intel Core microarchitecture and AMD K10 (K8L). It was announced on 27 September 2006 at the Fall 2006 Intel Developer Forum, with vague details in a white paper; more precise details of 47 instructions became available at the Spring 2007 Intel Developer Forum in Beijing, in the presentation. The SSE4 Programming Reference is available from Intel.

SSE4 subsets

Intel SSE4 consists of 54 instructions. A subset consisting of 47 instructions, referred to as SSE4.1 in some Intel documentation, is available in Penryn. Additionally, SSE4.2, a second subset consisting of the 7 remaining instructions, is first available in Nehalem-based Core i7. Intel credits feedback from developers as playing an important role in the development of the instruction set.

AMD supports 4 instructions from the SSE4 instruction set, but have also added four new SSE instructions, naming the group SSE4a. These instructions are not found in Intel's processors supporting SSE4.1 and AMD processors only started supporting Intel's SSE4.1 and SSE4.2 in the Bulldozer-based FX processors. Support was added for SSE4a for unaligned SSE load-operation instructions (which formerly required 16-byte alignment).

Name confusion

What is now known as SSSE3 (Supplemental Streaming SIMD Extensions 3), introduced in the Intel Core 2 processor line, was referred to as SSE4 by some media until Intel came up with the SSSE3 moniker. Internally dubbed Merom New Instructions, Intel originally did not plan to assign a special name to them, which was criticized by some journalists. Intel eventually cleared up the confusion and reserved the SSE4 name for their (at the time) future instruction set extension.

Intel is using the marketing term HD Boost to refer to SSE4.

New instructions

Unlike all previous iterations of SSE, SSE4 contains instructions that execute operations which are not specific to multimedia applications. It features a number of instructions whose action is determined by a constant field and a set of instructions that take XMM0 as an implicit third operand.

Several of these instructions are enabled by the single-cycle shuffle engine in Penryn. (Shuffle operations reorder bytes within a register.)

SSE4.1

These instructions were introduced with Penryn microarchitecture, the 45 nm shrink of Intel's Core microarchitecture. Support is indicated via the CPUID.01H:ECX.SSE41[Bit 19] flag.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPSADBW</td>
<td>Compute eight offset sums of absolute differences, four at a time (i.e., $l_0-x_0+y_0</td>
</tr>
<tr>
<td>PHMINPOSUW</td>
<td>Sets the bottom unsigned 16-bit word of the destination to the smallest unsigned 16-bit word in the source, and the next-from-bottom to the index of that word in the source.</td>
</tr>
<tr>
<td>PMULDQ</td>
<td>Packed signed multiplication on two sets of two out of four packed integers, the 1st and 3rd per packed 4, giving two packed 64-bit results.</td>
</tr>
<tr>
<td>PMULLD</td>
<td>Packed signed multiplication, four packed sets of 32-bit integers multiplied to give 4 packed 32-bit results.</td>
</tr>
<tr>
<td>DPPS, DPPD</td>
<td>Dot product for AOS (Array of Structs) data. This takes an immediate operand consisting of four (or two for DPPD) bits to select which of the entries in the input to multiply and accumulate, and another four (or two for DPPD) to select whether to put 0 or the dot-product in the appropriate field of the output.</td>
</tr>
<tr>
<td>BLENDPS, BLENDPD, BLENDVPS, BLENDVDP, PBLENDVB, PBLENDW</td>
<td>Conditional copying of elements in one location with another, based (for non-V form) on the bits in an immediate operand, and (for V form) on the bits in register XMM0.</td>
</tr>
<tr>
<td>PMINSB, PMAXSB, PMINUW, PMAXUW, PMINUD, PMAXUD, PMINSD, PMAXSD</td>
<td>Packed minimum/maximum for different integer operand types</td>
</tr>
<tr>
<td>ROUNDPS, ROUNDSS, ROUNDPD, ROUNDSD</td>
<td>Round values in a floating-point register to integers, using one of four rounding modes specified by an immediate operand</td>
</tr>
<tr>
<td>INSERTPS, PINSRB, PINSRD/PINSRQ, EXTRACTPS, PEXTRB, PEXTRD/PEXTRQ</td>
<td>The INSERTPS and PINSR instructions read 8, 16 or 32 bits from an x86 register memory location and insert it into a field in the destination register given by an immediate operand, EXTRACTPS and PEXTR read a field from the source register and insert it into an x86 register or memory location. For example, PEXTRD eax, [xmm0], 1; EXTRACTPS [addr+4*eax], xmm1, 1 stores the first field of xmm1 in the address given by the first field of xmm0.</td>
</tr>
<tr>
<td>PMOVSBW, PMOVZXBW, PMOVSBXD, PMOVZXBD, PMOVSBQ, PMOVZXQB, PMOVSSWD, PMOVZXWD, PMOVSSXQ, PMOVZXWXQ, PMOVSSXD, PMOVZXWD</td>
<td>Packed sign/zero extension to wider types</td>
</tr>
<tr>
<td>PTEST</td>
<td>This is similar to the TEST instruction, in that it sets the Z flag to the result of an AND between its operators: ZF is set, if DEST AND SRC is equal to 0. Additionally it sets the C flag if (NOT DEST) AND SRC equals zero. This is equivalent to setting the Z flag if none of the bits masked by SRC are set, and the C flag if all of the bits masked by SRC are set.</td>
</tr>
<tr>
<td>PCMPEQQ</td>
<td>Quadword (64 bits) compare for equality</td>
</tr>
<tr>
<td>PACKUSDW</td>
<td>Convert signed DWORDs into unsigned WORDs with saturation.</td>
</tr>
<tr>
<td>MOVNTDQA</td>
<td>Efficient read from write-combining memory area into SSE register; this is useful for retrieving results from peripherals attached to the memory bus.</td>
</tr>
</tbody>
</table>
SSE4.2

SSE4.2 added STTNI (STring and Text New Instructions),[9] several new instructions that perform character searches and comparison on two operands of 16 bytes at a time. These were designed (among other things) to speed the parsing of XML documents.[10] It also added a CRC32 instruction to compute cyclic redundancy checks as used in certain data transfer protocols. These instructions were first implemented in the Nehalem-based Intel Core i7 product line and complete the SSE4 instruction set. Support is indicated via the CPUID.01H:ECX.SSE42[Bit 20] flag.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC32</td>
<td>Accumulate CRC32C value using the polynomial 0x11EDC6F41 (or, without the high order bit, 0x1EDC6F41).[11][12]</td>
</tr>
<tr>
<td>PCMPESTRI</td>
<td>Packed Compare Explicit Length Strings, Return Index</td>
</tr>
<tr>
<td>PCMPESTRM</td>
<td>Packed Compare Explicit Length Strings, Return Mask</td>
</tr>
<tr>
<td>PCMPISTRI</td>
<td>Packed Compare Implicit Length Strings, Return Index</td>
</tr>
<tr>
<td>PCMPISTRM</td>
<td>Packed Compare Implicit Length Strings, Return Mask</td>
</tr>
<tr>
<td>PCMPGTQ</td>
<td>Compare Packed Signed 64-bit data For Greater Than</td>
</tr>
</tbody>
</table>

POPCNT and LZCNT

These instructions operate on integer rather than SSE registers, and although introduced at the same time, are not considered part of the SSE4.2 instruction set; instead, they have their own dedicated CPUID bits to indicate support. Intel implements POPCNT beginning with the Nehalem microarchitecture, and AMD implements both beginning with the Barcelona microarchitecture.

AMD calls this pair of instructions Advanced Bit Manipulation (ABM).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POPCNT</td>
<td>Population count (count number of bits set to 1). Support is indicated via the CPUID.01H:ECX.POPCNT[Bit 23] flag.[13]</td>
</tr>
<tr>
<td>LZCNT</td>
<td>Leading zero count. Support is indicated via the CPUID.80000001H:ECX.ABM[Bit 5] flag.[14] This instruction is not available on Intel processors, but the same value can be computed easily using bsr (BitScanReverse).</td>
</tr>
</tbody>
</table>

Trailing zeros can be counted using the existing bsf instruction.

SSE4a

The SSE4a instruction group was introduced in AMD’s Barcelona microarchitecture. These instructions are not available in Intel processors. Support is indicated via the CPUID.80000001H:ECX.SSE4A[Bit 6] flag.[14]
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTRQ/INSERTQ</td>
<td>Combined mask-shift instructions.</td>
</tr>
<tr>
<td>MOVNTSD/MOVNTSS</td>
<td>Scalar streaming store instructions.</td>
</tr>
</tbody>
</table>

**References**


SSE5

The SSE5 (short for Streaming SIMD Extensions version 5) was an instruction set extension proposed by AMD on 30 August 2007 as a supplement to the 128-bit SSE core instructions in the AMD64 architecture.

AMD chose not to implement SSE5 as originally proposed. In May 2009, AMD replaced SSE5 with three smaller instruction set extensions named as XOP, FMA4, and CVT16, which retain the proposed functionality of SSE5, but encode the instructions differently for better compatibility with Intel's proposed AVX instruction set.

The three SSE5-derived instruction sets were introduced in the Bulldozer processor core, released in October 2011 on a 32nm process.[1]

Compatibility

AMD's SSE5 extension bundle does not include the full set of Intel's SSE4 instructions, making it a competitor to SSE4 rather than a successor.

This complicates software development. It is recommended practice for a program to test for the presence of instruction set extensions by means of the CPUID instruction before entering a code path which depends upon those instructions to function correctly. For maximum portability, an optimized application will require three code paths: a base code path for compatibility with older processors (from either vendor), a separately optimized Intel code path exploiting SSE4 or AVX, and a separately optimized AMD code path exploiting SSE5.

Due to this proliferation, benchmarks between Intel and AMD processors increasingly reflect the cleverness or implementation quality of the divergent code paths rather than the strength of the underlying platform.

SSE5 enhancements

The proposed SSE5 instruction set consisted of 170 instructions (including 46 base instructions), many of which are designed to improve single-threaded performance. Some SSE5 instructions are 3-operand instructions, the use of which will increase the average number of instructions per cycle achievable by x86 code.[2] Selected new instructions include:[3]

- Fused multiply–accumulate (FMACxx) instructions
- Integer multiply–accumulate (IMAC, IMADC) instructions
- Permutation (PPERM, PERMPx) and conditional move (PCMOV) instructions
- Precision control, rounding, and conversion instructions

AMD claims SSE5 will provide dramatic performance improvements, particularly in high-performance computing (HPC), multimedia, and computer security applications, including a 5x performance gain for Advanced Encryption Standard (AES) encryption and a 30% performance gain for discrete cosine transform (DCT) used to process video streams.[2]

For more detailed information, consult the instruction sets as subsequently divided.

- XOP: A revision of most of the SSE5 instruction set
- FMA4: Floating-point vector multiply–accumulate.
- CVT16: Half-precision floating-point conversion.
2009 revision

The SSE5 specification included a proposed extension to the general coding scheme of X86 instructions in order to allow instructions to have more than two operands. In 2008, Intel announced their planned AVX instruction set which proposed a different way of coding instructions with more than two operands. The two proposed coding schemes, SSE5 and AVX, are mutually incompatible, although the AVX scheme has certain advantages over the SSE5 scheme: most importantly, AVX has plenty of space for future extensions, including larger vector sizes.

In May 2009, AMD published a revised specification for the planned future instructions. This revision changes the coding scheme to make it compatible with the AVX scheme, but with a differing prefix byte in order to avoid overlap between instructions introduced by AMD and instructions introduced by Intel.

The revised instruction set no longer carries the name SSE5, which has been criticized for being misleading, but most of the instructions in the new revision are functionally identical to the original SSE5 specification - only the way the instructions are coded differs. The planned additions to the AMD instruction set consists of three subsets:

1. XOP: Integer vector multiply–accumulate instructions, integer vector horizontal addition, integer vector compare, shift and rotate instructions, byte permutation and conditional move instructions, floating point fraction extraction.
2. FMA4: Floating-point vector multiply–accumulate.

These new instruction sets include support for future extensions for the vector size from 128 bits to 256 bits. It is unclear from these preliminary specifications whether the Bulldozer processor will support 256-bit vector registers (YMM registers) [4].

References


External links

• AMD SSE5 page (http://developer.amd.com/archive/cpu/SSE5/Pages/default.aspx)
• AMD Announces SSE5 Instruction Set (http://www.dailytech.com/article.aspx?newsid=8666), DailyTech
Advanced Vector Extensions

**Advanced Vector Extensions** (AVX) is an extension to the x86 instruction set architecture for microprocessors from Intel and AMD proposed by Intel in March 2008 and first supported by Intel with the Sandy Bridge processor shipping in Q1 2011 and now by AMD with the Bulldozer processor shipping in Q3 2011.

AVX provides new features, new instructions and a new coding scheme.

**New features**

The width of the SIMD register file is increased from 128 bits to 256 bits, and renamed from XMM0–XMM15 to YMM0–YMM15. In processors with AVX support, the legacy SSE instructions (which previously operated on 128-bit XMM registers) now operate on the lower 128 bits of the YMM registers.

AVX introduces a three-operand SIMD instruction format, where the destination register is distinct from the two source operands. For example, an SSE instruction using the conventional two-operand form \( a = a + b \) can now use a non-destructive three-operand form \( c = a + b \), preserving both source operands. AVX's three-operand format is limited to the instructions with SIMD operands (YMM), and does not include instructions with general purpose registers (e.g. EAX). Such support will first appear in AVX2.\(^1\)

The alignment requirement of SIMD memory operands is relaxed.

**New coding scheme**

The new VEX coding scheme introduces a new set of code prefixes that extends the opcode space, allows instructions to have more than two operands, and allows SIMD vector registers to be longer than 128 bits.

**Applications**

- Suitable for floating point-intensive calculations in multimedia, scientific and financial applications (integer operations are expected in later extensions).
- Increases parallelism and throughput in floating point SIMD calculations.
- Reduces register load due to the non-destructive instructions.

Prime95/MPrime, the software used for GIMPS, started using the AVX instructions since version 27.x.

**Compiler and assembler support**

Recent releases of GCC starting with version 4.6 (although there was a 4.3 branch with certain support) and the Intel Compiler Suite starting with version 11.1 support AVX. The Visual Studio 2010 compiler supports AVX via intrinsic and /arch:AVX switch. The Open64 compiler version 4.5.1 supports AVX with -mavx flag. PathScale supports via the -mavx flag. The GNU Assembler (GAS) inline assembly functions support these instructions (accessible via GCC), as do Intel primitives and the Intel inline assembler (closely compatible to GAS, although more general in its handling of local references within inline code). Other assemblers such as MASM VS2010 version, YASM 1.1.0, FASM, NASM and JWASM also apparently support AVX instructions.
Operating system support

AVX adds new register-state through the 256-bit wide YMM register file, so explicit operating system support is required to properly save & restore AVX's new registers between context switches. The following operating system versions will support AVX:

- Windows: supported in Windows 7 SP1 and Windows Server 2008 R2 SP1[^5]; hotfix 2517374 available for non-SP1 version of Windows Server 2008 R2[^6]; Windows 8
- Windows Server 2008 R2 SP1 with Hyper-V requires a hotfix to support AMD AVX (Opteron 6200 and 4200 series) processors, kb 2568088[^7].
- FreeBSD in a patch submitted on 21 January 2012[^8] which will probably be included in some future release.

CPUs with AVX

- Intel
  - Sandy Bridge processor, Q1 2011[^9]
  - Sandy Bridge E processor, Q4 2011[^10]
  - Ivy Bridge processor, Q1 2012.
- AMD:
  - Bulldozer processor, Q3 2011[^11]

Issues regarding compatibility between future Intel and AMD processors are discussed under XOP instruction set.

Future instruction sets

The VEX coding scheme allows future extensions of the SIMD register size.

Descriptions of other future x86 instruction sets:

- Intel FMA3
- AMD FMA4
- AMD XOP
- AMD CVT16

Advanced Vector Extensions 2

Advanced Vector Extensions 2 (AVX2), also known as Haswell New Instructions[^1] is an expansion of the AVX instruction set to be first introduced in Intel's Haswell microarchitecture. AVX2 makes the following additions:

- Expansion of most integer AVX instructions to 256 bits
- 3-operand general-purpose bit manipulation and multiply
- Gather support, enabling vector elements to be loaded from non-contiguous memory locations
- DWORD- and QWORD-granularity any-to-any permutes
- Vector shifts
- 3-operand fused multiply-accumulate support
CPUs with AVX2

- Intel
  - Haswell processor, 2013.
  - Broadwell processor, 2014.

New instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBCASTSS, VBCASTSD, VBCASTF128</td>
<td>Copy a 32-bit, 64-bit or 128-bit memory operand to all elements of a XMM or YMM vector register.</td>
</tr>
<tr>
<td>VINSTRF128</td>
<td>Replaces either the lower half or the upper half of a 256-bit YMM register with the value of a 128-bit source operand. The other half of the destination is unchanged.</td>
</tr>
<tr>
<td>VEXTRACTF128</td>
<td>Extracts either the lower half or the upper half of a 256-bit YMM register and copies the value to a 128-bit destination operand.</td>
</tr>
<tr>
<td>VMASKMOVPD</td>
<td>Conditionally reads any number of elements from a SIMD vector memory operand into a destination register, leaving the remaining vector elements unread and setting the corresponding elements in the destination register to zero. Alternatively, conditionally writes any number of elements from a SIMD vector register operand to a vector memory operand, leaving the remaining elements of the memory operand unchanged.</td>
</tr>
<tr>
<td>VPERMILPS, VPERMILPD</td>
<td>Shuffle 32-bit or 64-bit vector elements, with a register or memory operand as selector.</td>
</tr>
<tr>
<td>VPERM2F128</td>
<td>Shuffle the four 128-bit vector elements of two 256-bit source operands into a 256-bit destination operand, with an immediate constant as selector.</td>
</tr>
<tr>
<td>VZEROALL</td>
<td>Set all YMM registers to zero and tag them as unused. Used when switching between 128-bit use and 256-bit use.</td>
</tr>
<tr>
<td>VZEROUPPER</td>
<td>Set the upper half of all YMM registers to zero. Used when switching between 128-bit use and 256-bit use.</td>
</tr>
</tbody>
</table>

References

[8] Add support for the extended FPU states on amd64, both for native 64bit and 32bit ABIs (http://svnweb.freebsd.org/base?view=revision&revision=230426), svnweb.freebsd.org, retrieved 2012-01-21
CVT16 instruction set

The CVT16 instruction set, announced by AMD on May 1, 2009, is an extension to the 128-bit SSE core instructions in the x86 and AMD64 instruction set.

CVT16 is a revision of part of the SSE5 instruction set proposal announced on August 30, 2007. This revision makes the binary coding of the proposed new instructions more compatible with Intel's AVX instruction extensions, while the functionality of the instructions is unchanged.

The CVT16 instructions allow conversion of floating point vectors between single precision and half precision.

The CVT16 instruction set is supplemented by the XOP and FMA4 instruction sets, which were also included in SSE5.

References

XOP instruction set

The XOP (eXtended Operations[^1]) instruction set, announced by AMD on May 1, 2009, is an extension to the 128-bit SSE core instructions in the x86 and AMD64 instruction set for the Bulldozer processor core, which was released on October 12th, 2011.[^2]

XOP is a revision of the SSE5 instruction set proposal announced on August 30, 2007. This revision makes the binary coding of the proposed new instructions more compatible with Intel's AVX instruction extensions, while the functionality of the instructions is unchanged.[^1]

The XOP instructions include:

- Integer vector multiply–accumulate instructions
- Integer vector horizontal addition
- Integer vector compare
- Integer vector shift and rotate instructions
- Vector byte permutation
- Vector conditional move instructions
- Floating-point fraction extraction

The XOP instruction set is supplemented by the FMA4 (floating-point vector multiply–accumulate) and CVT16 (Half-precision floating-point conversion) instruction sets, which were also included in SSE5.

Compatibility issues

AMD has changed the encoding from the original SSE5 specification in order to improve compatibility with Intel's AVX instruction set and the new VEX coding scheme.

All SSE5 instructions that were equivalent or similar to instructions in the AVX and FMA4 instruction sets announced by Intel have been changed to use the coding proposed by Intel. Integer instructions without equivalents in AVX were classified as the XOP extension.[^1] The XOP instructions have an opcode byte 8F (hexadecimal), but otherwise almost identical coding scheme as AVX with the 3-byte VEX prefix.

Commentators[^3] have seen this as evidence that Intel has not allowed AMD to use any part of the large VEX coding space. AMD has been forced to use different codes in order to avoid using any code combination that Intel might possibly be using in their development pipeline for something else. The XOP coding scheme is as close to the VEX scheme as technically possible without risking that the AMD codes overlap with any future Intel codes. It must be
noted that this inference is speculative, since no public information is available about negotiations between the two companies on this issue.

The use of the 8F byte requires that the m-bits (see VEX coding scheme) have a value bigger than or equal to 8 in order to avoid overlap with existing instructions. The C4 byte used in the VEX scheme has no such restriction. This may prevent the use of the m-bits for other purposes in the future in the XOP scheme, but not in the VEX scheme. Another possible problem is that the pp bits have the value 00 in the XOP scheme, while they have the value 01 in the VEX scheme for instructions that have no legacy equivalent. This may complicate the use of the pp bits for other purposes in the future.

A similar compatibility issue is the difference between the FMA3 and FMA4 instruction sets. Intel initially proposed FMA4 in AVX/FMA specification version 3 to supersede the 3-operand FMA proposed by AMD in SSE5. After AMD adopted FMA4, however, Intel canceled FMA4 support and reverted back to FMA3 in the AVX/FMA specification version .

References

Image Sources, Licenses and Contributors


