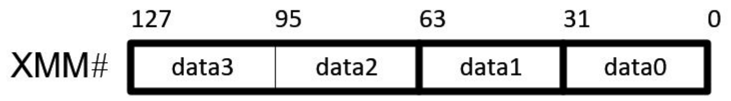
SSE (Streaming SIMD Extentions)

Download: sse_msvc.zip, cpuid_msvc.zip

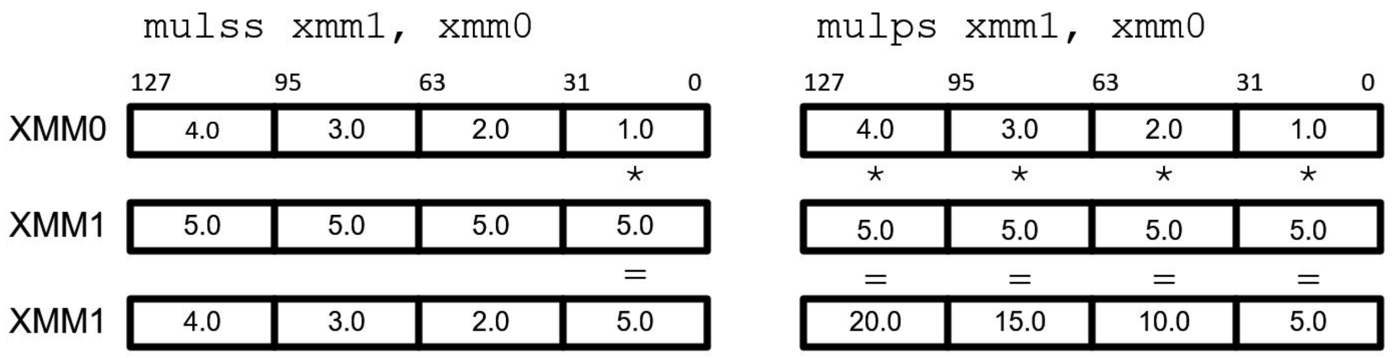
SIMD (Single Instruction, Multiple Data, pronounced "seem-dee") computation processes multiple data in parallel with a single instruction, resulting in significant performance improvement; 4 computations at once.

SSE defines 8 new 128-bit registers (xmm0 ~ xmm7) for single-precision floating-point computations. These registers are used for data computations only. Since each register has 128-bit long, we can store total 4 of 32-bit floating-point numbers (1-bit sign, 8-bit exponent, 23-bit mantissa).



Scalar and Packed Intructions

SSE defines two types of operations; scalar and packed. Scalar operation only operates on the least-significant data element (bit 0~31), and packed operation computes all four elements in parallel. SSE instructions have a suffix -ss for scalar operations (*Single Scalar*) and -ps for packed operations (*Parallel Scalar*).



Note that upper 3 elements in xmm0 for scalar operation remain unchanged.

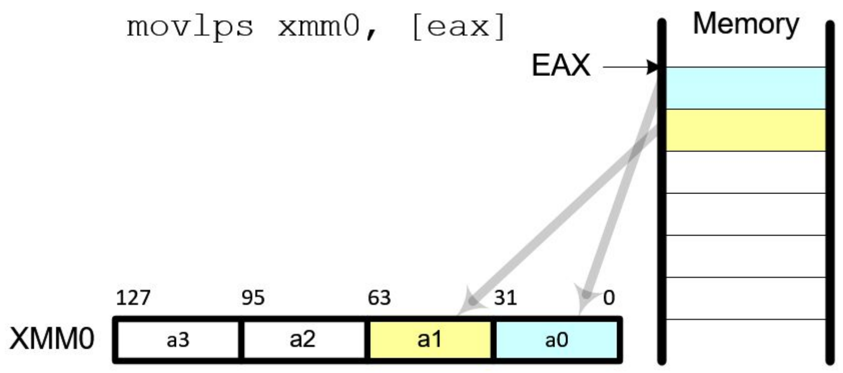
Data Movements

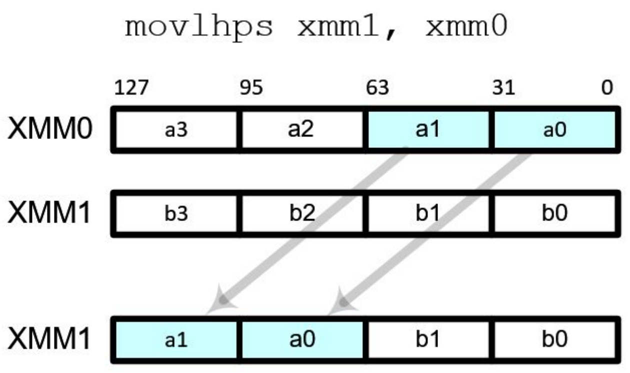
The first things that you should know are how to copy data from memory to xmm registers and how to get the results back to your application from xmm registers after SIMD operation. The data movement instructions move scalar and packed data between memory and xmm registers.

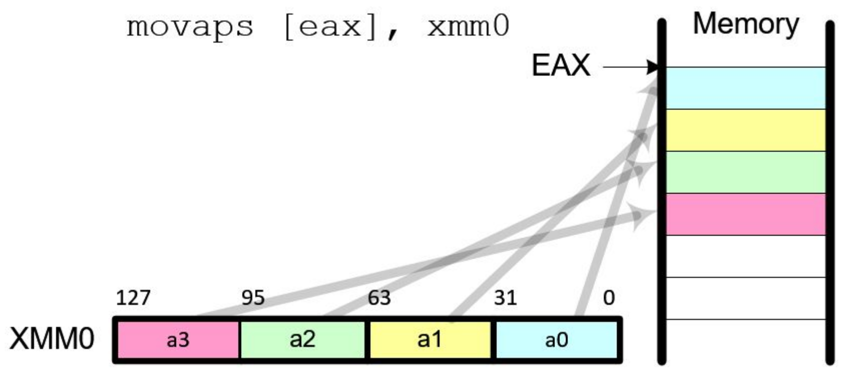
- **movss**: copy a single floating-point data

- **movlps**: copy 2 floating-point data (low packed)
- **movhps**: copy 2 floating-point data (high packed)
- **movaps**: copy aligned 4 floating-point data (fast)
- **movups**: copy unaligned 4 floating-point data (slow)
- **movhlps**: copy 2 high elements to low position
- **movlhps**: copy 2 low elements to high position

movaps requires that the data in memory must be aligned 16 byte boundary for better performance. Read more about how to align data in Data Alignment. The source and destination operands for **movhlps** and **movlhps** must be xmm registers.

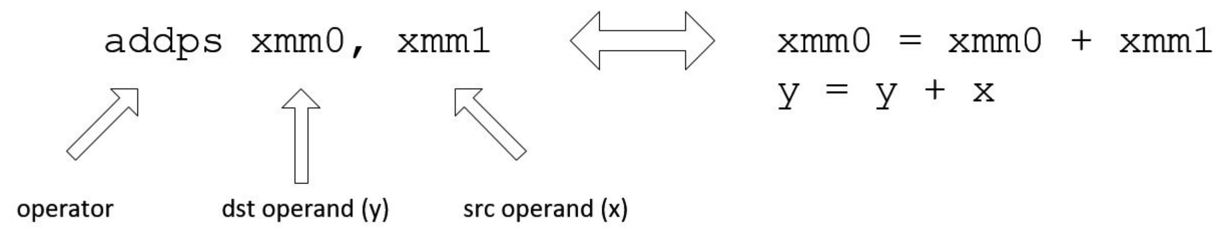






Arithmetic Instructions

Arithmetic Instruction requires 2 operands (registers or memory) to perform arithmetic computation and write the result in the first register. The source operand can be xmm register or memory, but the destination operand must be xmm register.



```
addps xmm0, xmm1        xmm0 = xmm0 + xmm1
                        y = y + x
```

Arithmetic	Scalar Operator	Packed Operator
$y = y + x$	addss	addps
$y = y - x$	subss	subps
$y = y \times x$	mulss	mulps
$y = y \div x$	divss	divps
$y = \frac{1}{x}$	rcpss	rcpps
$y = \sqrt{x}$	sqrtss	sqrtps
$y = \frac{1}{\sqrt{x}}$	rsqrtss	rsqrtps
$y = \max(y, x)$	maxss	maxps
$y = \min(y, x)$	minss	minps

Shuffle Instruction

shufps requires 2 operands and 1 mask. **shufps** selects 2 elements from each operand (register) based on the mask. 2 elements from the first operand are copied to the lower 2 elements in destination register and 2 elements from the second operand are copied to the higher 2 elements in the destination register.

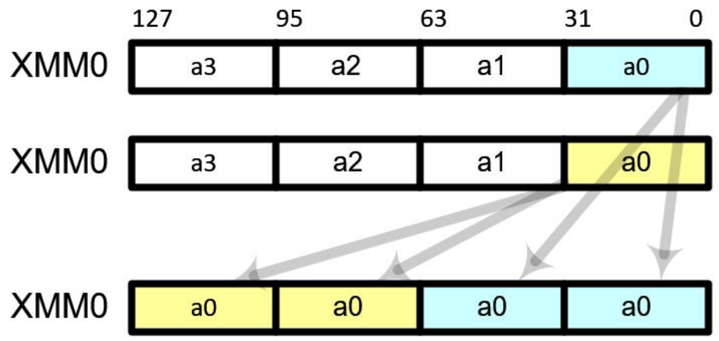
Using **shufps** instruction, you can shuffle any 4 data elements with any order. The frequent usages of **shufps** are broadcast, swap and rotate.

Broadcast

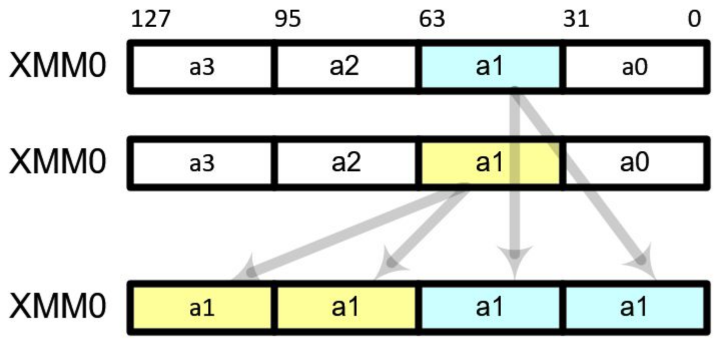
It copies all 4 fields with a single data element. The possible masks are

00h: Broadcast the least significant data element
55h: Broadcast the second data element
AAh: Broadcast the third data element
FFh: Broadcast the most significant data element

```
shufps xmm0, xmm0, 0h
```



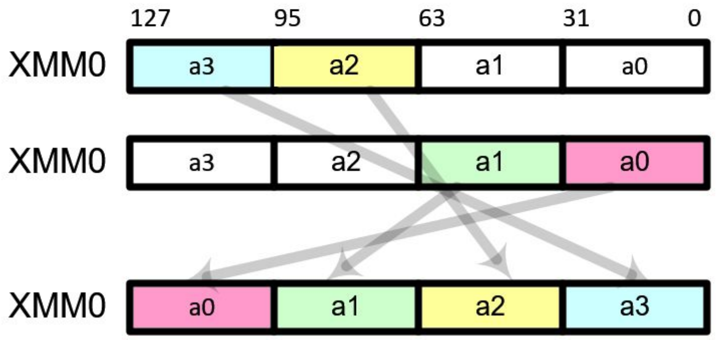
```
shufps xmm0, xmm0, 55h
```



Swap

This instruction switches the order of data elements reverse with **1Bh** (00011011b) mask.

```
shufps xmm0, xmm0, 1Bh
```

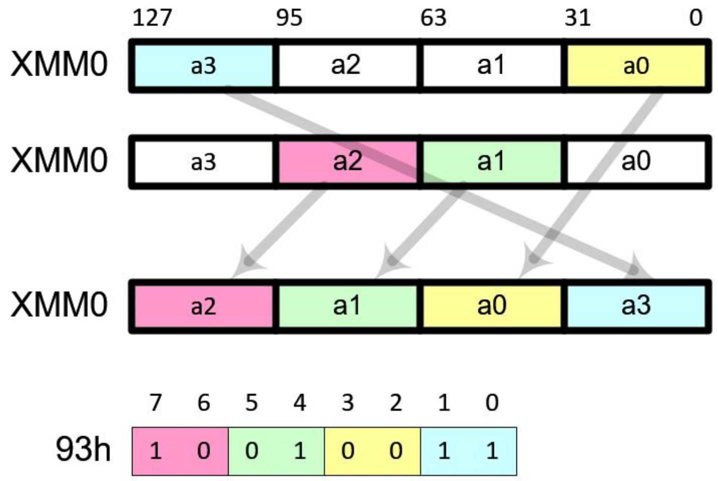


	7	6	5	4	3	2	1	0
1Bh	0	0	0	1	1	0	1	1

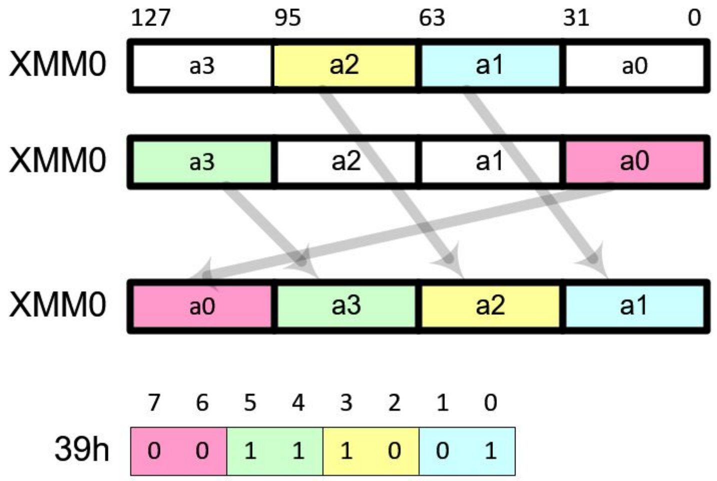
Rotate

It performs left or right rotation of data elements. Use **93h** (10010011b) to shift data to left direction and the most significant data element is moved to the least significant position. Use **39h** (00111001b) to shift data to right and the least significant data element is moved to the most significant position.

```
shufps xmm0, xmm0, 93h
```



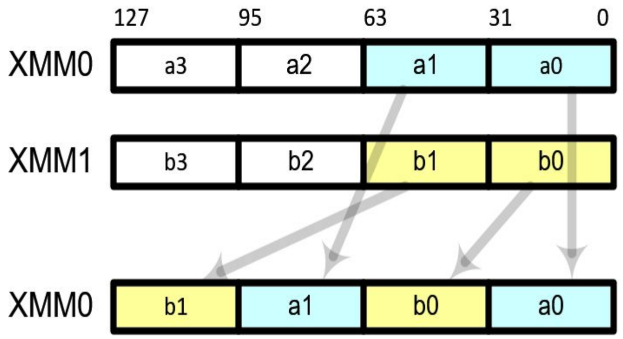
```
shufps xmm0, xmm0, 39h
```



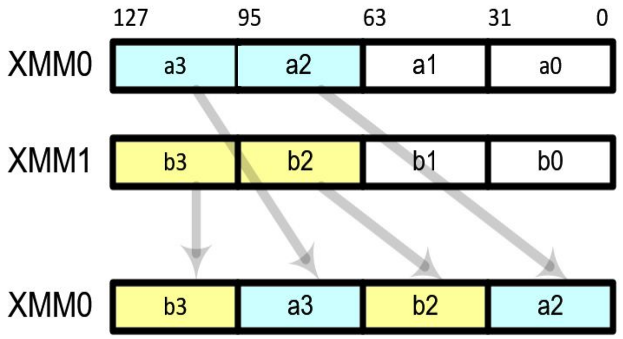
Unpack

unpcklps copies and interleaves the 2 lower elements from each of the 2 operands. **unpckhps** copies and interleaves the 2 higher elements from each of the 2 operands into the destination register.

```
unpcklps xmm0, xmm1
```



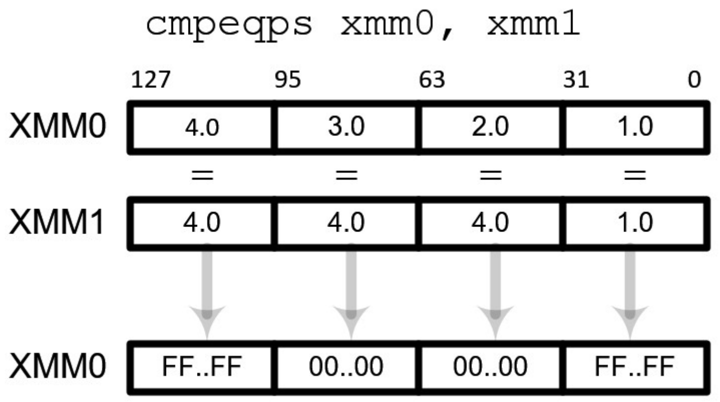
```
unpckhps xmm0, xmm1
```

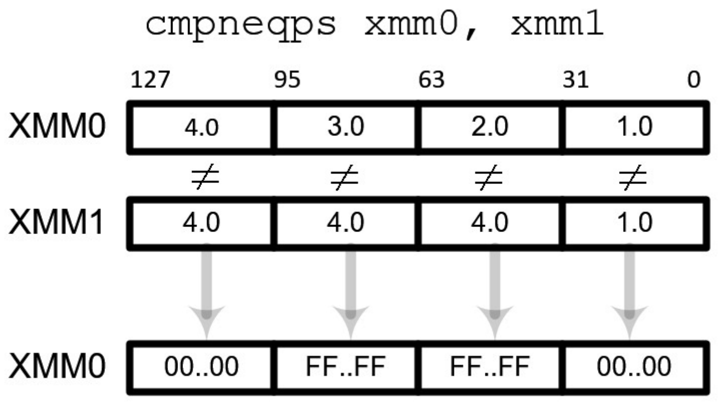


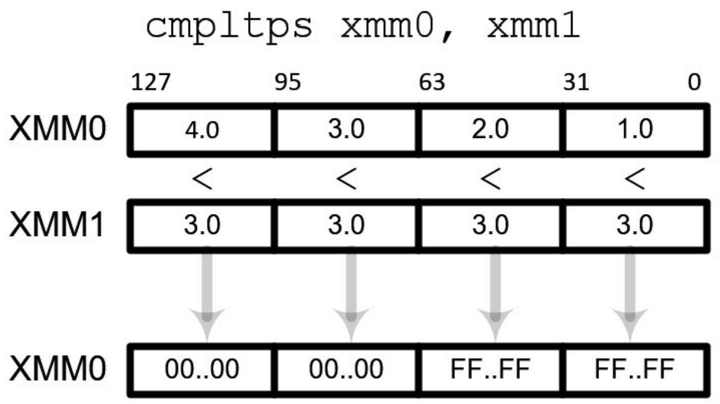
Comparison Instructions

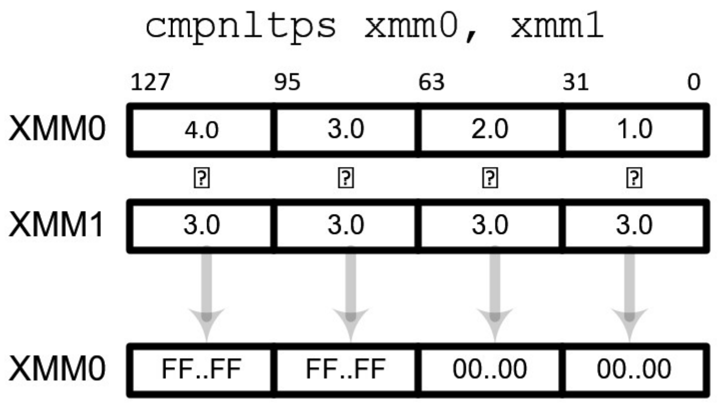
The comparison instructions compare 2 operands and set true (all 1s) or false (all 0s) into destination register. Source operand can be an xmm register or memory, but the destination must be an xmm register.

Condition	Scalar Operation	Packed Operation
$x = y$, $x \neq y$	cmpeqss, cmpneqss	cmpeqps, cmpneqps
$x < y$, $x \not< y$	cmpltss, cmpnltss	cmpltps, cmpnltps
$x \leq y$, $x \not\leq y$	cmpless, cmpnless	cmpleps, cmpnleps









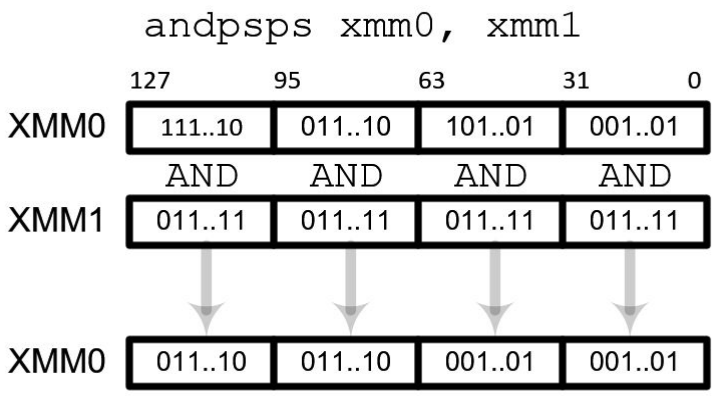
Bitwise Logical Instructions

Logical instructions perform bitwise logical operation on packed floating-point elements. The typical usages are negating numbers and converting to absolute values.

Operation	Instruction
AND	andps
OR	orps
XOR	xorps
AND NOT	andnps

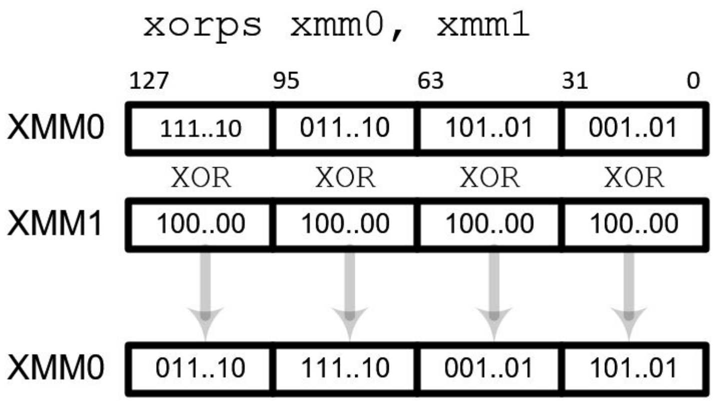
Absolute Value

To perform absolute value operation, store 0 at the most significant bit (sign bit) and 1s at the rest bits in source register. Then perform AND operation: *number* & **7FFFFFFFh**.



Negate

To perform negating, store 1 at the most significant bit and 0s at the rest bits. Then perform XOR operation: *number* ^ **8000000h**.

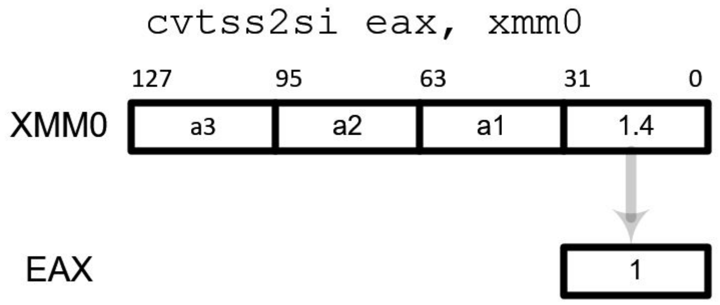


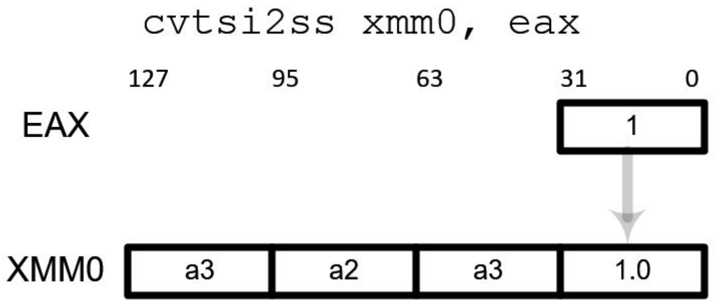
Conversion

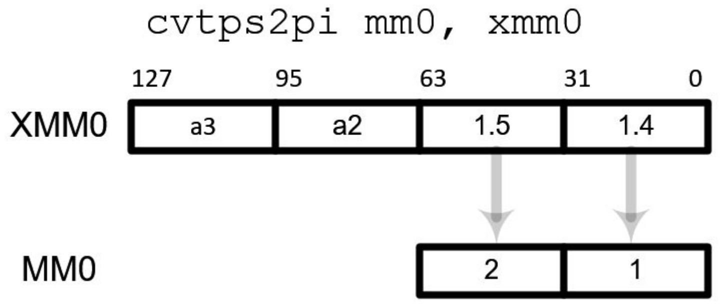
Conversion instructions convert from floating-point number to integer or vise versa.

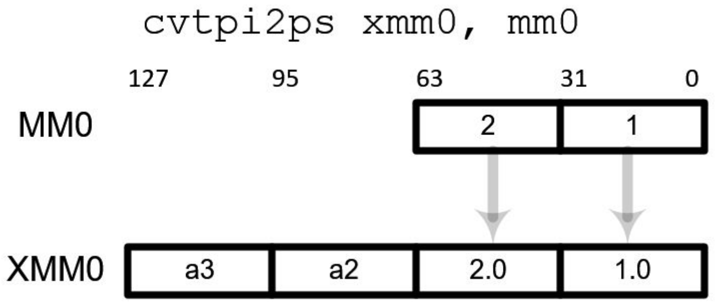
Conversion	Scalar Operation	Packed Operation
Float to integer with rounding	cvtss2si	cvtps2pi
Float to integer with truncation	cvttss2si	cvttps2pi
Integer to float	cvtsi2ss	cvtpi2ps

The packed operations, **cvtps2pi**, **cvttps2pi** and **cvtpi2ps** convert 2 numbers in parallel, not 4 because MMX registers (mm0~mm7) are 64-bit long (2x32-bit). Therefore, two upper elements in XMM registers are not used in conversion.









Streaming memory Instructions

SSE lets read-miss latency overlap execution via the use of prefetching, and it allowes write-miss latency to be reduced by overlapping execution via streaming stores.

Prefetch Instructions

The prefetch instructions provide cache hints to fetch data to the L1 and/or L2 cache before the program actually needs the data. This minimizes the data access latency. These instructions are executed asynchronously, therefore, program executions are not stalled while prefetching.

prefetcht0: move the data from memory to L1 and L2 caches using t0 hint.
prefetcht1: move the data from memory to L2 cache using t1 hint.
prefetchnta: move non-temporal aligned data from memory to L1 cache directly (bypass L2).

Note that AMD athlonXP, Intel Pentium4 or higher CPUs include automatic cache prefetching, therefore, it is not necessary to call these instructions manually in your code.

Streaming Store Instructions

Streaming store move instructions store non-temporal data directly to memory without updating the cache. This minimizes cache pollution and unnecessary bus bandwidth between cache and XMM registers because it does not write-allocate on a write miss.

Non-temporal means the data are accessed irregularly at long intervals (referenced once and not reused in immediate future) , for example, vertex data in 3D graphics are re-generated every frame. Write-allocate means that data write into the cache when cache miss occurs.

movntps: move 4 of non-temporal floating-point elements from XMM register to memory directly and bypasses the cache. The memory address must be aligned 16-byte boundaries.

movntq: move non-temporal quadword (2 integers, 4 shorts or 8 chars) from XMM register to memory and bypasses the cache.

```
movntps [edi], xmm0

movntq [edi], mm0
```

Store Fence

sfence guarantees that the data of any store instructions earlier than **sfence** instruction will be written to memory before any subsequent store instruction.

The following inline assembly example shows copying 4 float data (16-byte block) at once from source to destination array.

```
// move 4 floats (16-bytes) at once
__asm {
    mov     ecx,    count       // # of float data
    chr     ecx,    2           // # of 16-byte blocks (4 floats)
    mov     edi,    dst         // dst pointer
    mov     esi,    src         // src pointer

loop1:
    movaps  xmm0,   [esi]       // get from src
    movaps  [edi],  xmm0        // put to dst

    add     esi,    16
    add     edi,    16
    dec     ecx                 // next
    jnz     loop1
}
```

Detecting SSE support

cpuid instruction can be used whether the processor supports SSE or not. Most x86 processors support **cpuid** instruction nowadays, which returns CPU information and supported features. In order to determine your CPU supports **cpuid** instruction, try to toggle(modify) bit 21 in EFLAGS. If bit 21 can be toggled, cpuid can be called.

Calling **cpuid** with **eax=01h** returns standard feature flags to the **edx** register. SSE is supported if bit 25 (26th bit from the least significant bit) of **edx** register is 1. In addition, bit-26 is for SSE2 support and bit-23 is for MMX support.

This is a very simple program to detect SSE support and other features: cpuid_msvc.zip.
(*Note that this program uses MSVC specific inline assembly codes in it.*)

Example of Inline Assembly

The following program is an example of SSE usages in MSVC inline assembly. It includes example codes of all above SSE instructions.

Download the source and binary: sse_msvc.zip

←Back