XMM

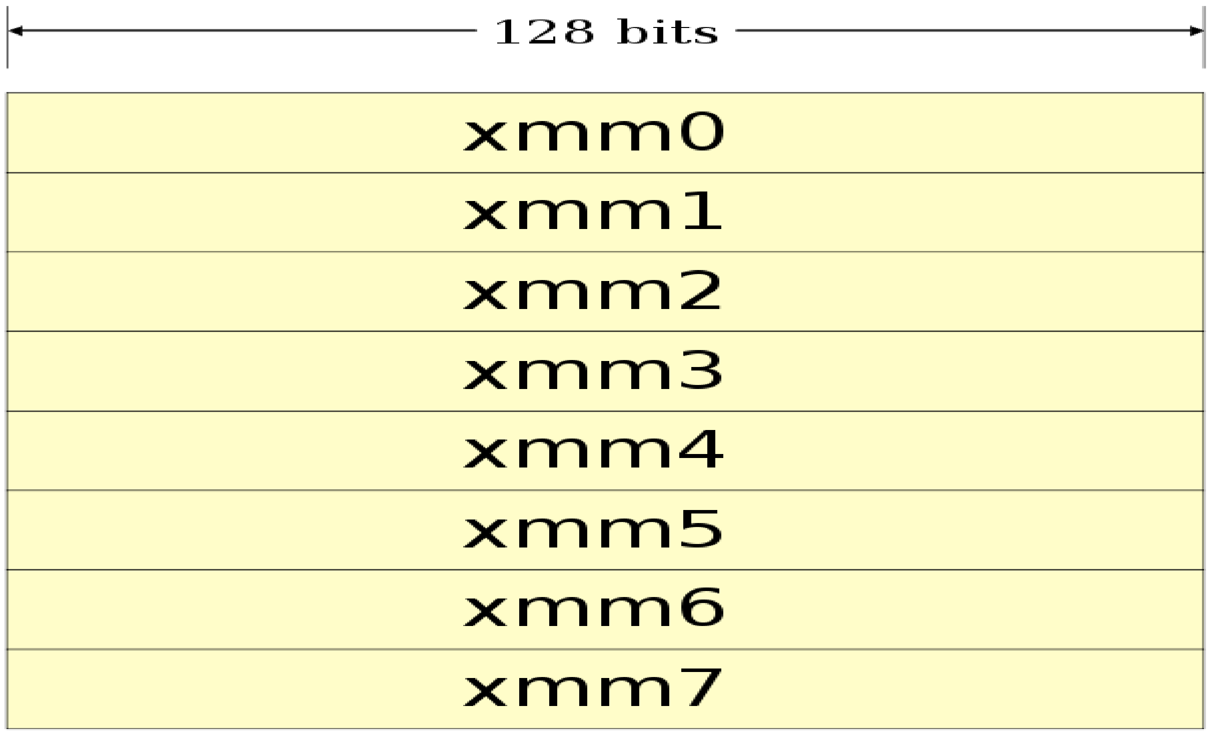
SIMD (Single Instruction, Multiple Data, pronounced "seem-dee") computation processes multiple data in parallel with a single instruction, resulting in significant performance improvement; 4 computations at once.

The SSE(Streaming SIMD Extensions) enhance the Intel x86 architecture in four ways:

1. 8 new 128-bit SIMD floating-point registers that can be directly addressed;
2. 50 new instructions that work on packed floating-point data;
3. 8 new instructions designed to control cacheability of all MMX and 32-bit x86 data types, including the ability to stream data to memory without polluting the caches, and to prefetch data before it is actually used;
4. 12 new instructions that extend the MMX instruction set.

This set enables the programmer to develop algorithms that can mix packed, single-precision, floating-point and integer using both SSE and MMX instructions respectively.

SSE adds 8 new 128-bit registers, divided into 4 32-bit (single precision) floating point values. These registers are called XMM0 - XMM7. 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). An additional control register, MXCSR, is also available to control and check the status of SSE instructions.



The MXCSR register is a 32-bit register containing flags for control and status information regarding SSE instructions. As of SSE3, only bits 0-15 have been defined.

Pnemonic	Bit Location	Description
FZ	bit 15	Flush To Zero
R+	bit 14	Round Positive
R-	bit 13	Round Negative
RZ	bits 13 and 14	Round To Zero
RN	bits 13 and 14 are 0	Round To Nearest
PM	bit 12	Precision Mask
UM	bit 11	Underflow Mask
OM	bit 10	Overflow Mask
ZM	bit 9	Divide By Zero Mask
DM	bit 8	Denormal Mask
IM	bit 7	Invalid Operation Mask
DAZ	bit 6	Denormals Are Zero
PE	bit 5	Precision Flag
UE	bit 4	Underflow Flag
OE	bit 3	Overflow Flag
ZE	bit 2	Divide By Zero Flag
DE	bit 1	Denormal Flag
IE	bit 0	Invalid Operation Flag

FZ mode causes all underflowing operations to simply go to zero. This saves some processing time, but loses precision.

The R+, R-, RN, and RZ rounding modes determine how the lowest bit is generated. Normally, RN is used.

PM, UM, MM, ZM, DM, and IM are masks that tell the processor to ignore the exceptions that happen, if they do. This keeps the program from having to deal with problems, but might cause invalid results.

DAZ tells the CPU to force all Denormals to zero. A Denormal is a number that is so small that FPU can't renormalize it due to limited exponent ranges. They're just like normal numbers, but they take considerably longer to process. Note that not all processors support DAZ.

PE, UE, ME, ZE, DE, and IE are the exception flags that are set if they happen, and aren't unmasked. Programs can check these to see if something interesting happened. These bits are "sticky", which means that once they're set, they stay set forever until the program clears them. This means that the indicated exception could has happened several operations ago, but nobody bothered to clear it.

DAZ wasn't available in the first version of SSE. Since setting a reserved bit in MXCSR causes a general protection fault, we need to be able to check the availability of this feature without causing problems. To do this, one needs to set up a 512-byte area of memory to save the SSE state to, using **`fxsave`**, and then one needs to inspect bytes 28 through 31 for the MXCSR_MASK value. If bit 6 is set, DAZ is supported, otherwise, it isn't.

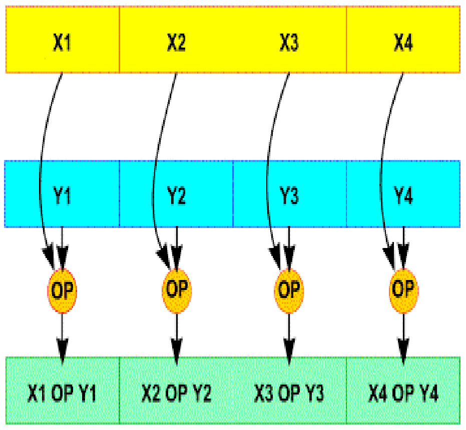
Detecting SSE support

`cupid` 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.

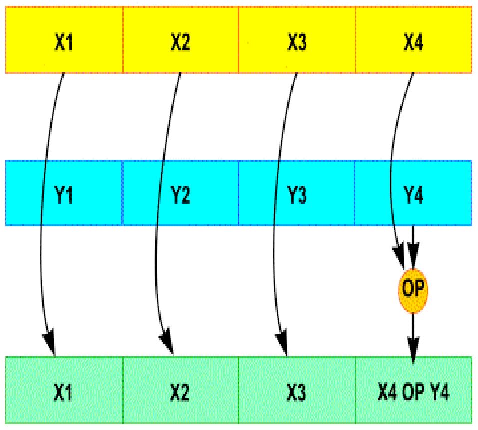
Calling **`cupid`** with **eax=01h** returns standard feature flags to the **edx** register. SSE is supported if bit 25 (26th bit from the least significant bit) of **edx** register is 1. In addition, bit-26 is for SSE2 support and bit-23 is for MMX support.

Scalar and Packed Instructions

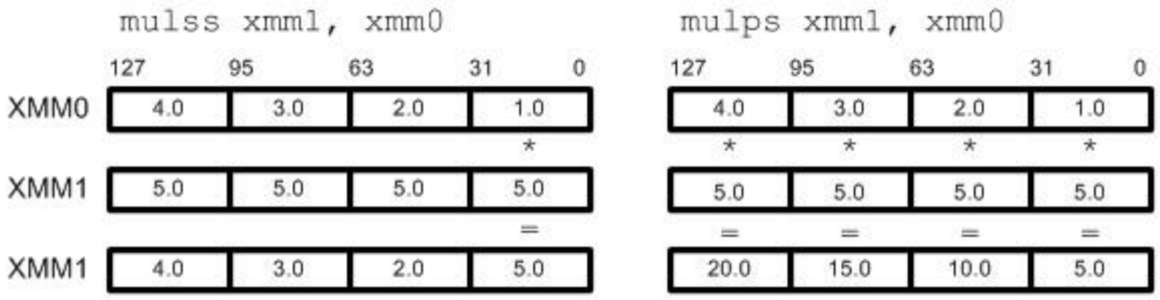
SSE defines two types of operations; scalar and packed. Scalar operation only operates on the least-significant data element (bit 0~31), and packed operation computes all four elements in parallel. SSE instructions have a suffix -ss for scalar operations (Single Scalar) and -ps for packed operations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



SSE Packed



SSE Scalar



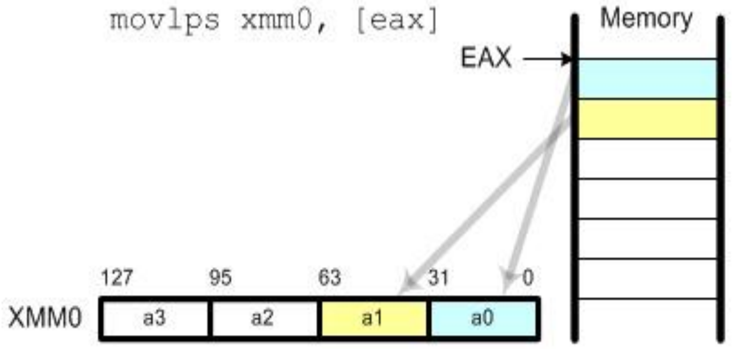
Note that upper 3 elements in xmm0 for scalar operation remain unchanged.

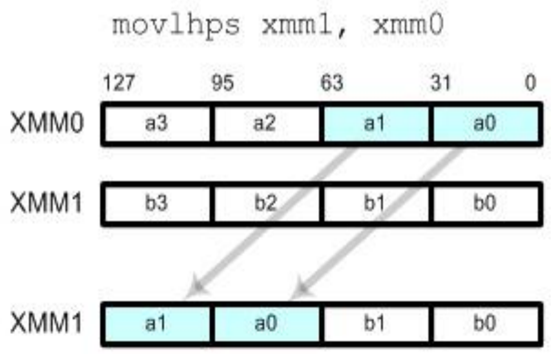
Data Movements

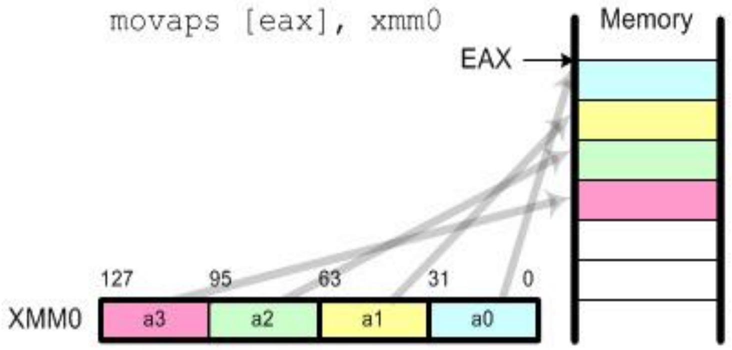
The first things that you should know are how to copy data from memory to xmm registers and how to get the results back to your application from xmm registers after SIMD operation. The data movement instructions move scalar and packed data between memory and xmm registers.

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. The source and destination operands for **movhlps** and **movlhps** must be xmm registers.

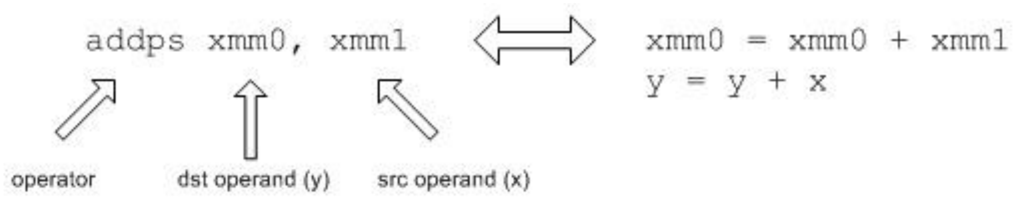






Arithmetic Instructions

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.



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

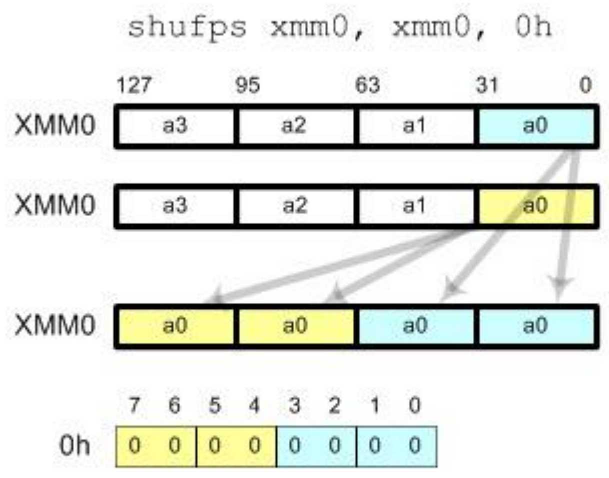
`shufps` requires 2 operands and 1 mask. `shufps` selects 2 elements from each operand (register) based on the mask. 2 elements from the first operand are copied to the lower 2 elements in destination register and 2 elements from the second operand are copied to the higher 2 elements in the destination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

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

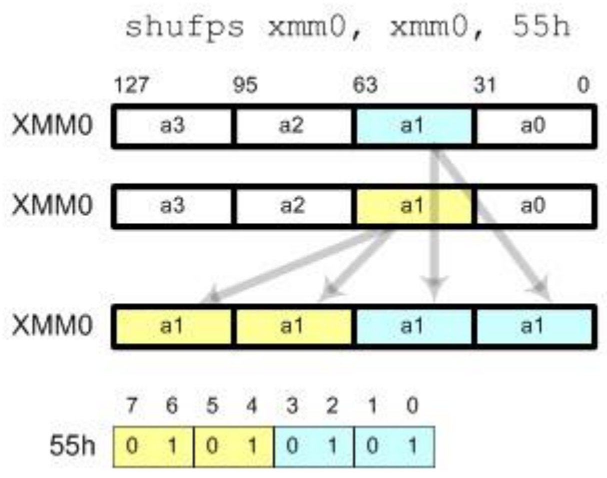
Broadcast

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



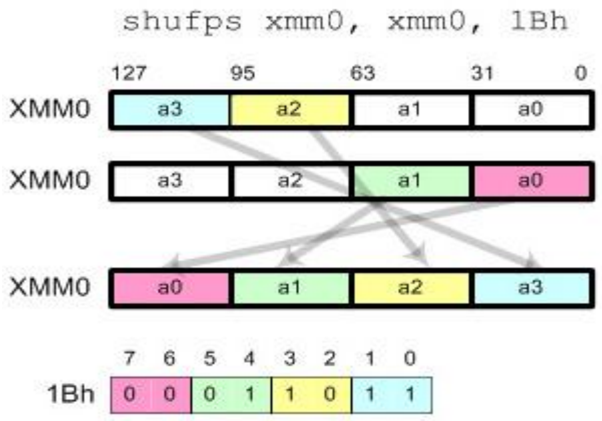
```
shufps xmm0, xmm0, 0h
```



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

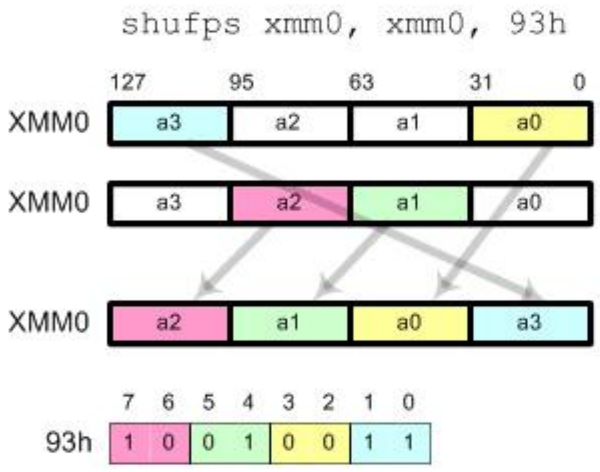
Swap

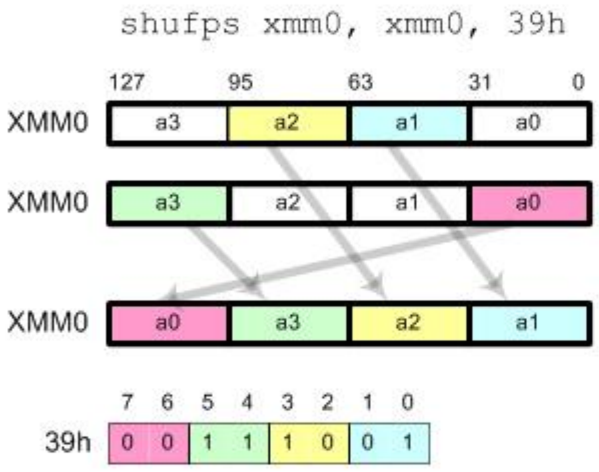
This instruction switches the order of data elements reverse with 1Bh mask.



Rotate

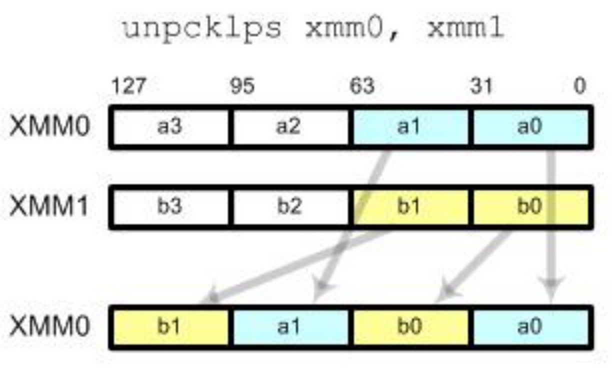
It performs left or right rotation of data elements. Use 93h to shift data to left direction and the most significant data element is moved to the least significant position. Use 39h to shift data to right and the least significant data element is moved to the most significant position.

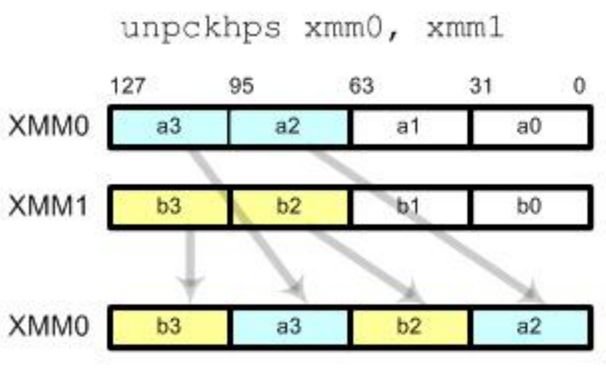




Unpack

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.

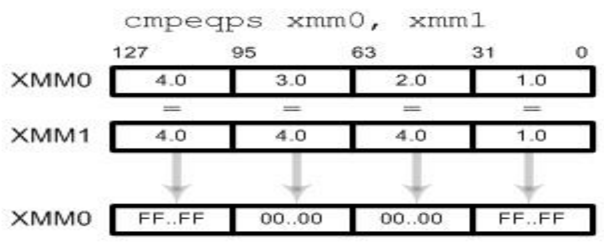


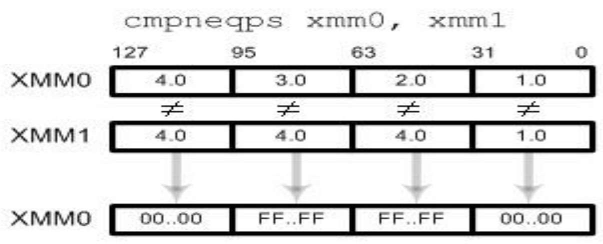


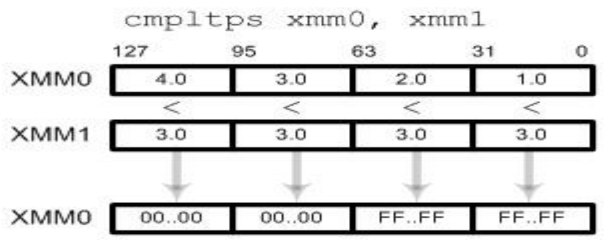
Comparison Instructions

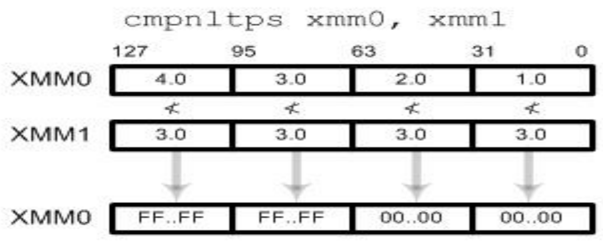
The comparison instructions compare 2 operands and set true (all 1s) or false (all 0s) into destination register. Source operand can be an xmm register or memory, but the destination must be an xmm register.

Condition	Scalar Operation	Packed Operation
$x = y$, $x \neq y$	cmpeqss, cmpneqss	cmpeqps, cmpneqps
$x < y$, $x \nless y$	cmpltss, cmpnltss	cmpltps, cmpnltps
$x \leq y$, $x \nleq y$	cmpless, cmpnless	cmpleps, cmpnleps









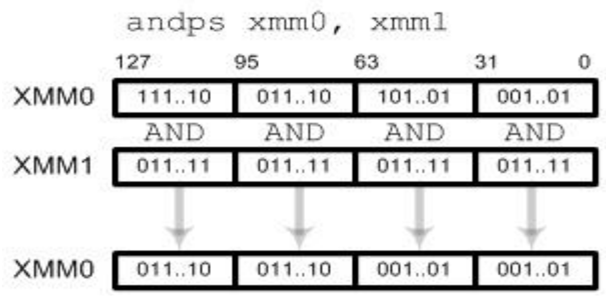
Bitwise Logical Instructions

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`

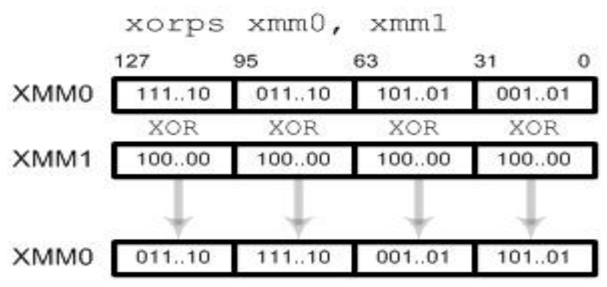
Absolute Value

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.

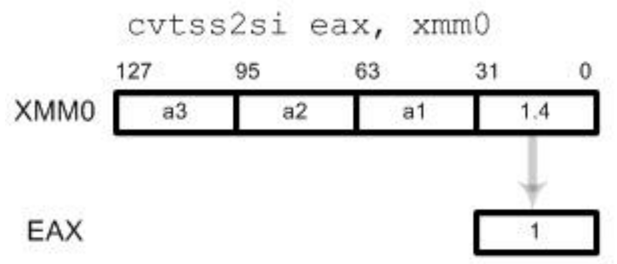


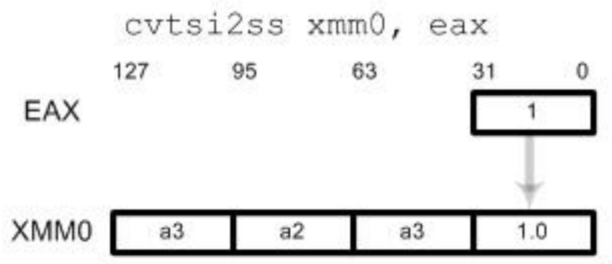
Conversion

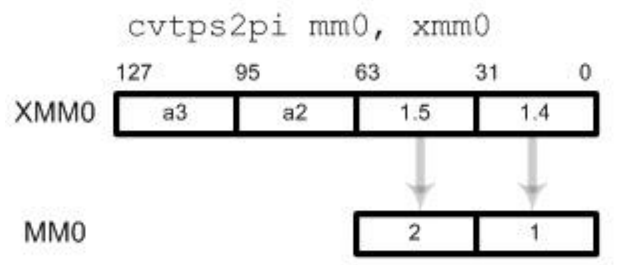
Conversion instructions convert from floating-point number to integer or vice versa.

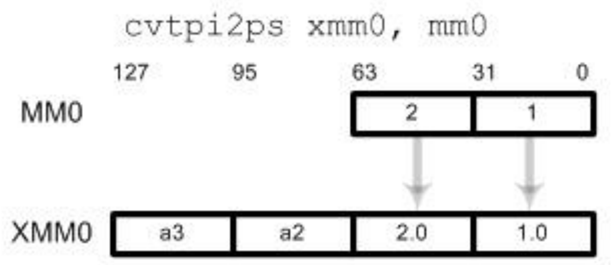
Conversion	Scalar Operation	Packed Operation
Float to integer with rounding	`cvtss2si`	`cvtps2pi`
Float to integer with truncation	`cvttss2si`	`cvttps2pi`
Integer to float	`cvtsi2ss`	`cvtpi2ps`

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.









SOURCES

1. http://www.songho.ca/misc/sse/sse.html

2. http://www.tommesani.com/SSE.html

3. http://softpixel.com/~cwright/programming/simd/sse.php

4. ORACLE x86 Assembly Language Reference Manual

http://docs.oracle.com/cd/E19963-01/html/821-1608/eojde.html

5. http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions