SSE and SSE2

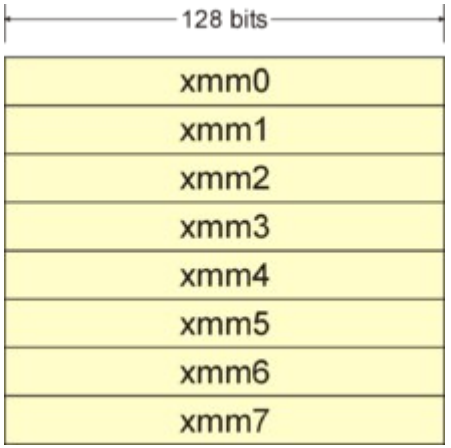
Timothy A. Chagnon

18 September 2007

All images from Intel® 64 and IA-32 Architectures Software Developer's Manuals

Overview

- SSE: Streaming SIMD (Single Instruction Multiple Data) Extensions
- Successor to 64-bit MMX integer and AMD's 3DNow! extensions



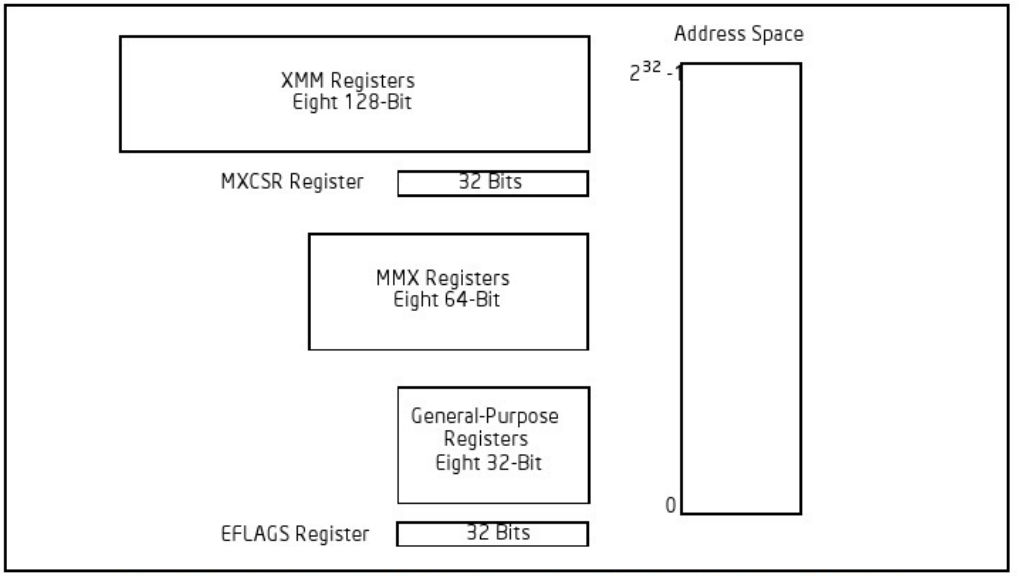


Figure 11-1. Steaming SIMD Extensions 2 Execution Environment

SSE

- Introduced in 1999 with the Pentium III
- 128-bit packed single precision floating point
- 8 128-bit registers xmm0-xmm7, hold 4 floats each
- Additional 8 on x86-64 (xmm8-xmm15)

SSE2

- Introduced with Pentium 4
- Added double precision FP and several integer types

SSE 3 & 4

- Not yet widespread
- More ops: DSP, permutations, fixed point, dot product...

Data Types

- Instructions end in D, S, I, Q etc. to designate type
- Conversion instructions CVTxx2xx

SSE

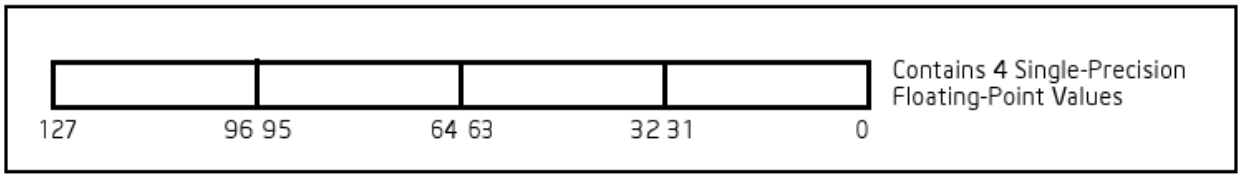


Figure 10-4. 128-Bit Packed Single-Precision Floating-Point Data Type

SSE2

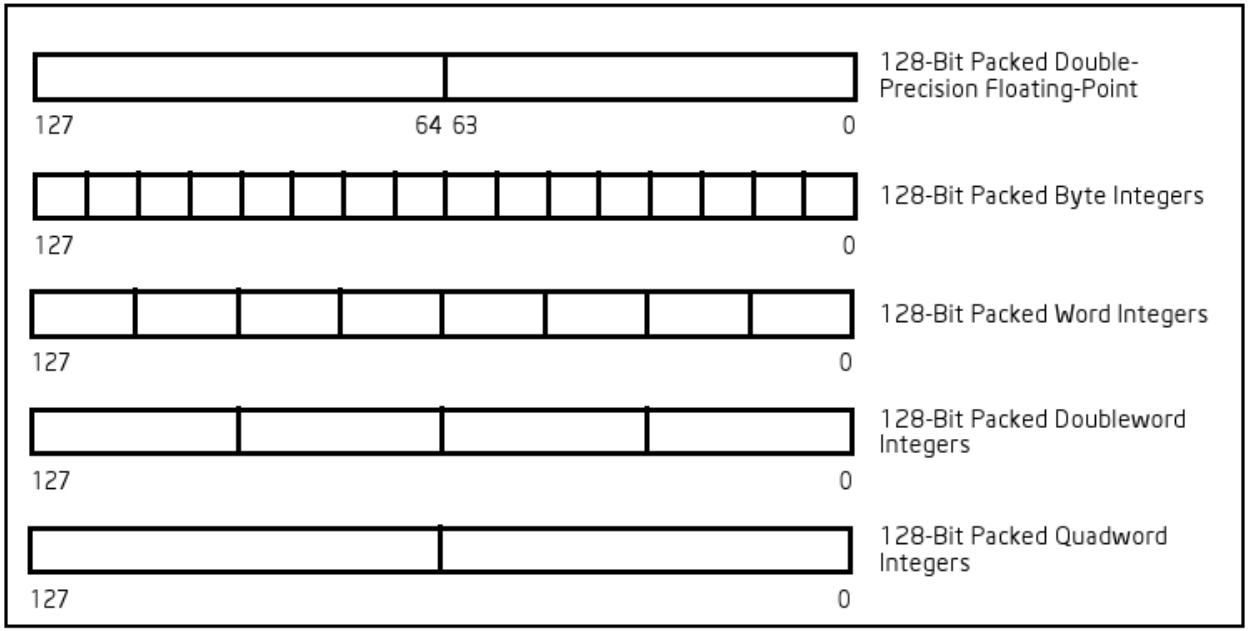


Figure 11-2. Data Types Introduced with the SSE2 Extensions

Instruction Basics

- Packed vs. Scalar Ops
- Most operate on 2 args
  - OP *xmm-dest, xmm/m128-src*
  - In-place operation on dest.
- Instruction Varieties
  - MOV loads, stores
  - Arithmetic & Logical
  - Shuffle & Unpack
  - CVT conversion
  - Cache & Ordering Control

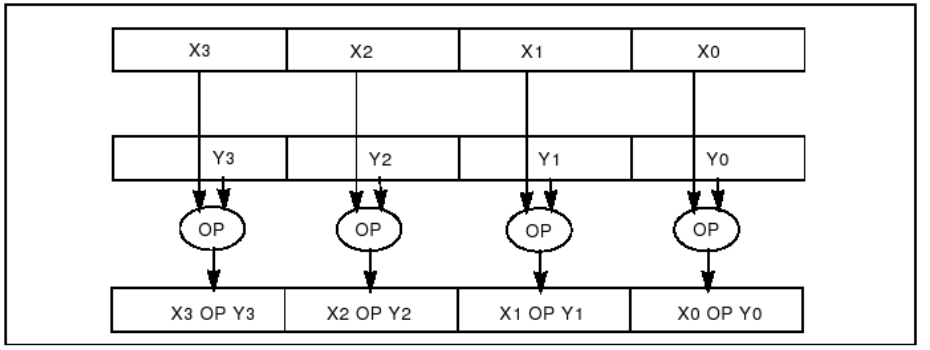


Figure 10-5. Packed Single-Precision Floating-Point Operation

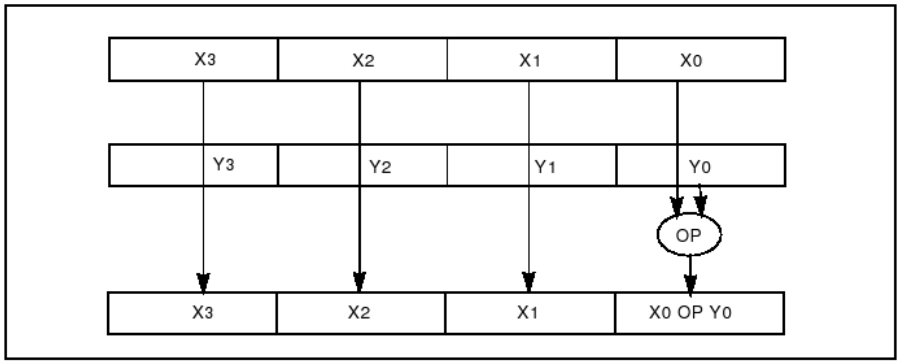


Figure 10-6. Scalar Single-Precision Floating-Point Operation

Arithmetic & Logical Instructions

ADD
SUB
MUL
DIV
SQRT
MAX
MIN
CMP

- Can take a memory reference as 2nd argument
- All put results into 1st argument
  - CMP can be made to use EFLAGS
- Full instruction names are OP+TYPE...
  - ADDPD add packed double
  - ANDSS and scalar single
  - etc.

```
MOVAPD    xmm0, [eax]
MOVAPD    xmm1, [eax+16]
MOVAPD    xmm2, xmm0
ADDPD     xmm2, xmm1
SUBPD     xmm0, xmm1
MOVAPD    [eax], xmm2
MOVAPD    [eax+16], xmm0
```

Moving Data to/from Memory

- MOVAPD move aligned
- MOVUPD move unaligned
  - reg-reg, reg-mem, mem-reg
- Memory should be 16-byte aligned if possible
  - Starts at a 128-bit boundary in Virtual?/Physical? addressing
  - Special types and attributes are used in C/C++
- MOVUPD takes *much* longer than MOVAPD
  - Intel's optimization manual says it's actually faster to do

```
MOVSD xmm0, [mem]
MOVSD xmm1, [mem+8]
UNPCKLPD xmm0, xmm1
```

Shuffle & Unpack

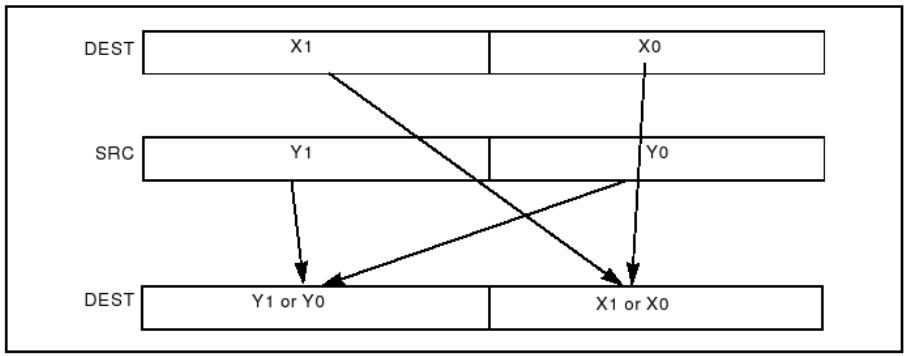


Figure 11-5. SHUFPD Instruction, Packed Shuffle Operation

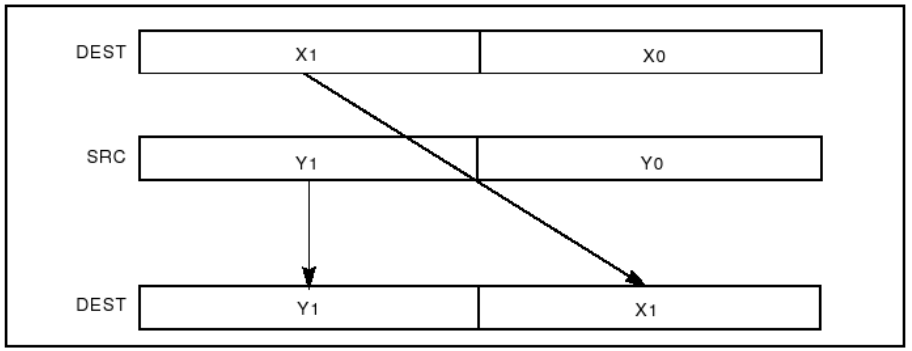


Figure 11-6. UNPCKHPD Instruction, High Unpack and Interleave Operation

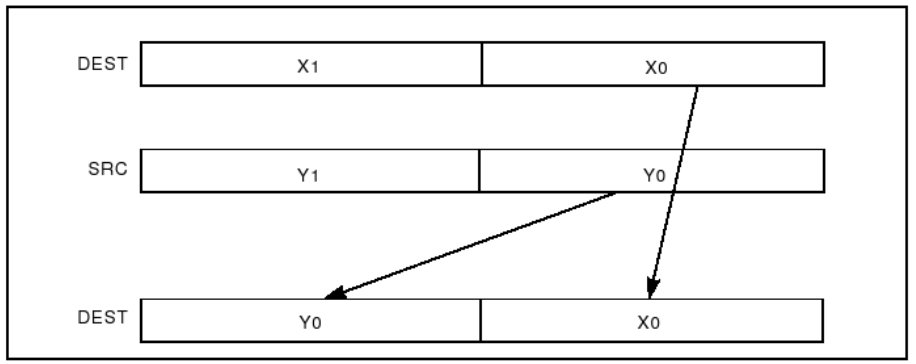


Figure 11-7. UNPCKLPD Instruction, Low Unpack and Interleave Operation

- SHUFPD xmm0, xmm1, pattern
- UNPCKHPD xmm0, xmm1
- UNPCKLPD xmm0, xmm1
- Combine parts of 2 vectors
- In-Place
- Value(s) from 1st argument alway go into low ½ of dest.
- Note that UNPCKs are just a shortcut of SHUF
  - But is it faster or slower?

C/C++ Intrinsics

- Both Intel's ICC and GCC have “intrinsic” functions to access the SSE instructions
  - More or less a 1:1 mapping of instructions
- Instead of in-place, they look like 2-in, 1-out functions
  - Does this indicate that penalties are incurred for unnecessary register copies/loads/stores?
  - Or are these optimized away? ... It appears that way.
- MOV is split up into `load`, `store`, and `set`

```
__m128d _mm_add_pd(__m128d a, __m128d b)
```

Adds the two DP FP values of a and b.
r0 := a0 + b0
r1 := a1 + b1

```
__m128d _mm_load_pd(double const*dp)
```

(uses MOVAPD) Loads two DP FP values.
The address p must be 16-byte aligned.
r0 := p[0]
r1 := p[1]

Example

```
#include <xmmintrin.h>

int main(void){
  int i;
  __m128d a, b, c;
  double x0[2] __attribute__((aligned(16))) = {1.2, 3.5};
  double x1[2] __attribute__((aligned(16))) = {-0.7, 2.6};

  a = _mm_load_pd(x0);
  b = _mm_load_pd(x1);
  c = _mm_add_pd(a, b);
  _mm_store_pd(x0, c);

  for(i = 0; i < 2; i++)
    printf("%.2f\n", x0[i]);

  return 0;
}
```

Efficient Use

- Use 16-byte aligned memory
- Compiler driven automated vectorization is limited
- Use struct-of-arrays (SoA) instead of AoS (3d vectors)
  - i.e. block across multiple entities and do normal operations

To Investigate

- Denormals and underflow can cause penalties (1500? cycles)
- Are intrinsics efficient? Do they use direct memory refs?
- SHUF vs UNPCK ?
- Cache utilization, ordering instructions ...
- x87 FPU and SSE are disjoint and can be mixed ...
- MMX registers can be used for shuffle or copy
- Intel Core architecture has more efficient SIMD than NetBurst
- Direct memory references
- Trace cache and reorder buffer effects on loop unrolling
- Non-temporal stores
- Prefetch

SHUF vs. UNPCK

- Intel Optimization Manual shows, sometimes UNPCK is faster
  - Probably because there's no decoding of immediate field
- gcc-4.x will convert a call to the shuffle intrinsic to an unpck instruction if it can
- Table data represents usage of register arguments; delays from direct memory references depend heavily on cache state.

NetBurst Arch. (Pentium 4, Xeon)

Table C-4. Streaming SIMD Extension 2 Double-precision Floating-point Instructions

Instruction	Latency[1]		Throughput		Execution Unit[2]
DisplayFamily_DisplayModel	0F_03H	0F_02H	0F_03H	0F_02H	0F_02H
ADDPD xmm, xmm	5	4	2	2	FP_ADD
SHUFPD[3] xmm, xmm, imm8	6	6	2	2	MMX_SHFT
UNPCKHPD xmm, xmm	6	6	2	2	MMX_SHFT
UNPCKLPD[3] xmm, xmm	4	4	2	2	MMX_SHFT

Core Arch. (Core Duo, Pentium M)

Table C-4a. Streaming SIMD Extension 2 Double-precision Floating-point Instructions

Instruction	Latency[1]				Throughput			
DisplayFamily_DisplayModel	06_0FH	06_0EH	06_0DH	06_09H	06_0FH	06_0EH	06_0DH	06_09H
ADDPD xmm, xmm	3	4	4	4	1	2	2	2
SHUFPD xmm, xmm, imm8	1	2	2	2	1	2	2	2
UNPCKHPD xmm, xmm	1			1	1	1	1	1
UNPCKLPD xmm, xmm	1			1	1	1	1	1

DisplayFamily_DisplayModel represent variation across different processors. Family 0F is NetBurst Architecture, Family 06 is Intel Core Architecture Model for NetBurst in range 00H to 06H, data for 03H applies to 04H and 06H Model for Pentium M are 09H and 0DH, Core Solo/Duo is 0EH, Core is 0FH

References

- Wikipedia: Streaming SIMD Extensions
  - http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions
- Intel® 64 and IA-32 Architectures Software Developer's Manuals
  - http://www.intel.com/products/processor/manuals/index.htm
- Intel® C++ Compiler Documentation
  - http://www.intel.com/software/products/compilers/docs/clin/main_cls/index.htm
- Apple Developer Connection – SSE Performance Programming
  - http://developer.apple.com/hardwaredrivers/ve/sse.html