

CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Instructors:
Randy H. Katz
David A. Patterson
<http://inst.eecs.Berkeley.edu/~cs61c/fa10>

10/10/10 Fall 2010 -- Lecture #18 1

Agenda

- Intel SSE SIMD Instructions
- Administrivia
- Technology Break
- SSE in C

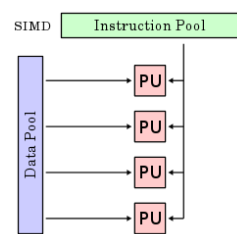
10/10/10 Fall 2010 -- Lecture #18 2

Agenda

- Intel SSE SIMD Instructions
- Administrivia
- Technology Break
- SSE in C

10/10/10 Fall 2010 -- Lecture #18 3

Single Instruction/Multiple Data Stream

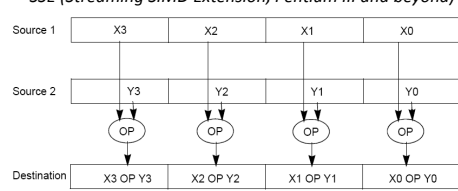


- Single Instruction, Multiple Data streams (SIMD)
 - Computer that exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., an array processor or Graphics Processing Unit (GPU)

10/10/10 Fall 2010 -- Lecture #18 4

“Advanced Digital Media Boost”

- To improve performance, Intel’s SIMD instructions
 - Fetch one instruction, do the work of multiple instructions
 - MMX (MultiMedia eXtension, Pentium II processor family)
 - SSE (Streaming SIMD Extension, Pentium III and beyond)



10/10/10 Fall 2010 -- Lecture #18 5

Example: SIMD Array Processing

```

for each f in array
    f = sqrt(f)

for each f in array
{
    load f to the floating-point register
    calculate the square root
    write the result from the register to memory
}

for each 4 members in array
{
    load 4 members to the SSE register
    calculate 4 square roots in one operation
    write the result from the register to memory
}
    
```

10/10/10 Fall 2010 -- Lecture #18 6

SSE Instruction Categories for Multimedia Support

Instruction category	Operands
Unsigned add/subtract	Eight 8-bit or Four 16-bit
Saturating add/subtract	Eight 8-bit or Four 16-bit
Max/min/minimum	Eight 8-bit or Four 16-bit
Average	Eight 8-bit or Four 16-bit
Shift right/left	Eight 8-bit or Four 16-bit

- SSE-2+ supports wider data types to allow 16 x 8-bit and 8 x 16-bit operands

10/10/10 Fall 2010 – Lecture #18 7

Intel Architecture 128-Bit SIMD Data Types

Fundamental 128-Bit Packed SIMD Data Types

- Note: in Intel Architecture (unlike MIPS) a word is 16 bits
 - Single precision FP: Double words (32 bits)
 - Double precision FP: Quad words (64 bits)

10/10/10 Fall 2010 – Lecture #18 8

XMM Registers

- Architecture extended with eight 128-bit data registers: XMM registers
 - 64-bit modes: available as 16 64-bit registers (XMM8 – XMM15)
 - E.g., 128-bit packed single-precision floating-point data type (doublewords), allows four single-precision operations to be performed simultaneously

10/10/10 Fall 2010 – Lecture #18 9

SSE/SSE2 Floating Point Instructions

Data transfer	Arithmetic	Compare
MOV {A/U} {SS/PS/SD/PD} xmm, mem/xmm	ADD {SS/PS/SD/PD} xmm, mem/xmm	CMP {SS/PS/SD/PD}
	SUB {SS/PS/SD/PD} xmm, mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	MUL {SS/PS/SD/PD} xmm, mem/xmm	
	DIV {SS/PS/SD/PD} xmm, mem/xmm	
	SQRT {SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN {SS/PS/SD/PD} mem/xmm	

xmm: one operand is a 128-bit SSE2 register
 mem/xmm: other operand is in memory or an SSE2 register
 {SS} Scalar Single precision FP: one 32-bit operand in a 128-bit register
 {PS} Packed Single precision FP: four 32-bit operands in a 128-bit register
 {SD} Scalar Double precision FP: one 64-bit operand in a 128-bit register
 {PD} Packed Double precision FP: two 64-bit operands in a 128-bit register
 {A} 128-bit operand is aligned in memory
 {U} means the 128-bit operand is unaligned in memory
 {H} means move the high half of the 128-bit operand
 {L} means move the low half of the 128-bit operand

10/10/10 Fall 2010 – Lecture #18 10

Example: Add Two Quad Word Vectors

Computation to be performed:

```
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;
```

SSE Instruction Sequence:

```
movaps xmm0, address-of-v1
; xmm0=v1.w | v1.z | v1.y | v1.x
addps xmm0, address-of-v2
; xmm0=v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x
movaps address-of-vec_res, xmm0
```

(Note: Red arrows in the original image point from the assembly instructions to the corresponding variables in the computation above.)

10/10/10 Fall 2010 – Lecture #18 11

Displays and Pixels

- Each coordinate in frame buffer on left determines shade of corresponding coordinate for the raster scan CRT display on right. Pixel (X0, Y0) contains bit pattern 0011, a lighter shade on the screen than the bit pattern 1101 in pixel (X1, Y1)

10/10/10 Fall 2010 – Lecture #18 12

Example: Image Converter

- Converts BMP image to a YUV image format:
 - Read individual pixels from the BMP image, convert pixels into YUV format
 - Can pack the pixels and operate on a set of pixels with a single instruction
- E.g., bitmap image consists of 8 bit monochrome pixels
 - Pack these pixel values in a 128 bit register (8 bit * 16 pixels), can operate on 16 values at a time
 - Significant performance boost

10/10/10

Fall 2010 – Lecture #18

13

Example: Image Converter

- FMADDPS – Multiply and add packed single precision floating point instruction
- One of the typical operations computed in transformations (e.g., DFT or FFT)

$$P = \sum_{n=1}^N f(n) \times x(n)$$

10/10/10

Fall 2010 – Lecture #18

14

Example: Image Converter

Floating point numbers $f(n)$ and $x(n)$ in `src1` and `src2`; p in `dest`;
C implementation for $N = 4$ (128 bits):

```
for (int i = 0; i < 4; i++)
{
    p = p + src1[i] * src2[i];
}
```

SSE2 instructions for the inner loop:

```
//xmm0 = p, xmm1 = src1, xmm2 = src2
mulps xmm1, xmm2
addps xmm0, xmm1
```

SSE5 instruction accomplishes same in one instruction:

```
//xmm0 = p, xmm1 = src1, xmm2 = src2
fmaddps xmm0, xmm1, xmm2, xmm0
```

10/10/10

Fall 2010 – Lecture #18

15

Agenda

- Intel SSE SIMD Instructions
- Administrivia
- Technology Break
- SSE in C

10/10/10

Fall 2010 – Lecture #18

16

Agenda

- Intel SSE SIMD Instructions
- Administrivia
- Technology Break
- SSE in C

10/10/10

Fall 2010 – Lecture #18

17

Agenda

- Intel SSE SIMD Instructions
- Administrivia
- Technology Break
- SSE in C

10/10/10

Fall 2010 – Lecture #18

18

Intel SSE Intrinsics

- Intrinsics are C functions and procedures for SSE instructions
 - With intrinsics, can program using these instructions indirectly
 - One-to-one correspondence between SSE instructions and intrinsics

Example SSE Intrinsics

- Vector data type: `_m128d`
- Load and store operations:
 - `_mm_load_pd` MOVAPD/aligned, packed double
 - `_mm_store_pd` MOVAPD/aligned, packed double
 - `_mm_loadu_pd` MOVUPD/unaligned, packed double
 - `_mm_storeu_pd` MOVUPD/unaligned, packed double
- Load and broadcast across vector
 - `_mm_load1_pd` MOVSD + shuffling
- Arithmetic:
 - `_mm_add_pd` ADDPD/add, packed double
 - `_mm_mul_pd` MULPD/multiple, packed double

Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

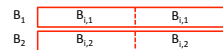
$$C_{i,j} = (A \times B)_{i,j} = \sum_{k=1}^2 A_{i,k} \times B_{k,j}$$

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$ $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
 $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$ $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

Example: 2 x 2 Matrix Multiply

- Using the XMM registers
 - 64-bit/double precision/two doubles per XMM reg



Example: 2 x 2 Matrix Multiply

- Initialization



Example: 2 x 2 Matrix Multiply

- Initialization



- $i = 1$



Example: 2 x 2 Matrix Multiply

- First iteration intermediate result

C_1	$0+A_{1,1}B_{1,1}$ $0+A_{2,1}B_{1,1}$	$c1 = _mm_add_pd(c1, _mm_mul_pd(a,b1));$
C_2	$0+A_{1,1}B_{1,2}$ $0+A_{2,1}B_{2,2}$	$c2 = _mm_add_pd(c2, _mm_mul_pd(a,b2));$

- $| = 1$

A	$A_{1,1}$ $A_{2,1}$	$_mm_load_pd$: Stored in memory in Column order
-----	-----------------------	---

B_1	$B_{1,1}$ $B_{1,1}$	$_mm_load1_pd$: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register
B_2	$B_{1,2}$ $B_{1,2}$	

10/10/10 Fall 2010 – Lecture #18 25

Example: 2 x 2 Matrix Multiply

- First iteration intermediate result

C_1	$0+A_{1,1}B_{1,1}$ $0+A_{2,1}B_{1,1}$	$c1 = _mm_add_pd(c1, _mm_mul_pd(a,b1));$
C_2	$0+A_{1,1}B_{1,2}$ $0+A_{2,1}B_{2,2}$	$c2 = _mm_add_pd(c2, _mm_mul_pd(a,b2));$

- $| = 2$

A	$A_{1,2}$ $A_{2,2}$	$_mm_load_pd$: Stored in memory in Column order
-----	-----------------------	---

B_1	$B_{2,1}$ $B_{2,1}$	$_mm_load1_pd$: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register
B_2	$B_{2,2}$ $B_{2,2}$	

10/10/10 Fall 2010 – Lecture #18 26

Example: 2 x 2 Matrix Multiply

- Second iteration intermediate result

	$C_{1,1}$ $C_{2,1}$	
C_1	$A_{1,1}B_{1,1}+A_{1,2}B_{2,1}$ $A_{2,1}B_{1,1}+A_{2,2}B_{2,1}$	$c1 = _mm_add_pd(c1, _mm_mul_pd(a,b1));$
C_2	$A_{1,1}B_{1,2}+A_{1,2}B_{2,2}$ $A_{2,1}B_{1,2}+A_{2,2}B_{2,2}$	$c2 = _mm_add_pd(c2, _mm_mul_pd(a,b2));$
	$C_{1,2}$ $C_{2,2}$	

- $| = 2$

A	$A_{1,2}$ $A_{2,2}$	$_mm_load_pd$: Stored in memory in Column order
-----	-----------------------	---

B_1	$B_{2,1}$ $B_{2,1}$	$_mm_load1_pd$: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register
B_2	$B_{2,2}$ $B_{2,2}$	

10/10/10 Fall 2010 – Lecture #18 27

Example: 2 x 2 Matrix Multiply (Part 1 of 2)

```

#include <stdio.h>
// header file for SSE compiler intrinsics
#include <emmintrin.h>

// NOTE: vector registers will be represented in comments as v1 = [ a | b ]
// where v1 is a variable of type __m128d and a,b are doubles

int main(void) {
    // allocate A,B,C aligned on 16-byte boundaries
    double A[4] __attribute__((aligned(16)));
    double B[4] __attribute__((aligned(16)));
    double C[4] __attribute__((aligned(16)));
    int lda = 2;
    int i = 0;
    // declare a couple 128-bit vector variables
    __m128d c1,c2,a,b1,b2;

    /* A = (note column order!)
    1 0
    0 1
    */
    A[0] = 1.0; A[1] = 0.0; A[2] = 0.0; A[3] = 1.0;

    /* B = (note column order!)
    1 3
    2 4
    */
    B[0] = 1.0; B[1] = 2.0; B[2] = 3.0; B[3] = 4.0;

    /* C = (note column order!)
    0 0
    0 0
    */
    C[0] = 0.0; C[1] = 0.0; C[2] = 0.0; C[3] = 0.0;
    
```

10/10/10 Fall 2010 – Lecture #18 28

Example: 2 x 2 Matrix Multiply (Part 2 of 2)

```

// used aligned loads to set
// c1 = [c_11 | c_21]
c1 = _mm_load_pd(C+0*lda);
// c2 = [c_12 | c_22]
c2 = _mm_load_pd(C+1*lda);

for (i = 0; i < 2; i++) {
    /* a =
    i=0: [a_11 | a_21]
    i=1: [a_12 | a_22]
    */
    a = _mm_load_pd(A+i*lda);
    /* b1 =
    i=0: [b_11 | b_11]
    i=1: [b_21 | b_21]
    */
    b1 = _mm_load1_pd(B+i*0*lda);
    /* b2 =
    i=0: [b_12 | b_12]
    i=1: [b_22 | b_22]
    */
    b2 = _mm_load1_pd(B+i+1*lda);

    /* c1 =
    i=0: [c_11 + a_11*b_11 | c_21 + a_21*b_11]
    i=1: [c_11 + a_21*b_21 | c_21 + a_22*b_21]
    */
    c1 = _mm_add_pd(c1, _mm_mul_pd(a,b1));
    /* c2 =
    i=0: [c_12 + a_11*b_12 | c_22 + a_21*b_12]
    i=1: [c_12 + a_21*b_22 | c_22 + a_22*b_22]
    */
    c2 = _mm_add_pd(c2, _mm_mul_pd(a,b2));
}

// store c1,c2 back into C for completion
_mm_store_pd(C+0*lda,c1);
_mm_store_pd(C+1*lda,c2);

// print C
printf("%g,%g\n%g,%g\n",C[0],C[2],C[1],C[3]);
return 0;
    
```

10/10/10 Fall 2010 – Lecture #18 29

Summary

- Intel SSE SIMD Instructions
 - One instruction fetch that operates on multiple operands simultaneously
- SSE Instructions in C
 - Can embed the SSE machine instructions directly into C programs through the use of intrinsics

10/10/10 Fall 2010 – Lecture #18 30