CS 61C: Great Ideas in Computer Architecture *SIMD II*

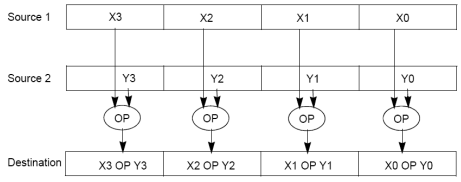
Instructors:
Krste Asanovic, Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa12

Review

- Three C's of cache misses
  - Compulsory
  - Capacity
  - Conflict
- Amdahl's Law: Speedup = 1/((1-F)+F/S)
- Flynn's Taxonomy: SISD/SIMD/MISD/MIMD
- Exploiting Data-Level Parallelism with SIMD instructions

Intel SIMD Instructions

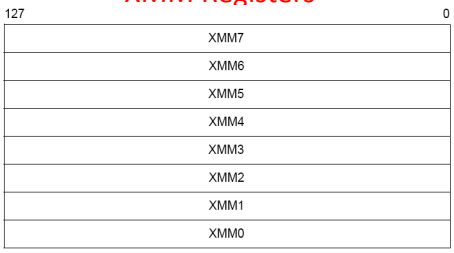
- Fetch one instruction, do the work of multiple instructions



Intel SIMD Extensions

- MMX 64-bit registers, reusing floating-point registers [1992]
- SSE2/3/4, new 128-bit registers [1999]
- AVX, new 256-bit registers [2011]
  - Space for expansion to 1024-bit registers

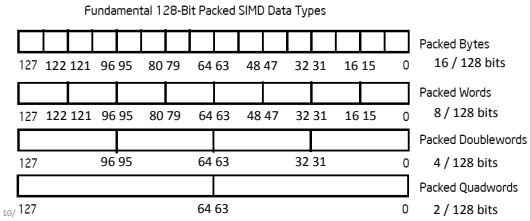
XMM Registers



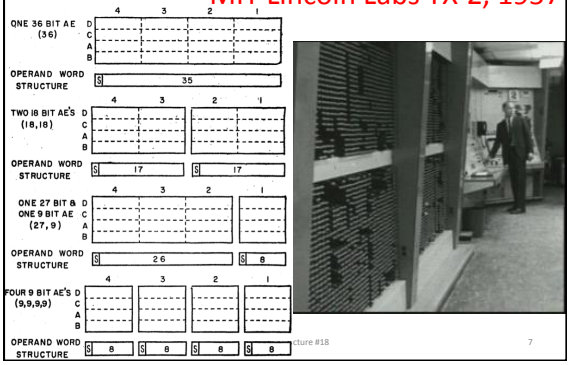
- Architecture extended with eight 128-bit data registers: XMM registers
  - x86 64-bit address architecture adds 8 additional registers (XMM8 – XMM15)

Intel Architecture SSE2+ 128-Bit SIMD Data Types

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



First SIMD Extensions: MIT Lincoln Labs TX-2, 1957



SSE/SSE2 Floating Point Instructions

Move does both load and store

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, or 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

Example: Add Two Single-Precision Floating-Point 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:
(Note: Destination on the right in x86 assembly)

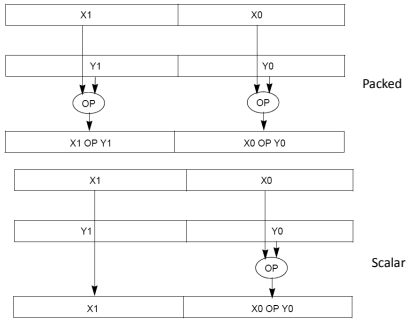
```
movaps address-of-v1, %xmm0
          // v1.w | v1.z | v1.y | v1.x -> xmm0
addps address-of-v2, %xmm0
          // v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x -> xmm0
movaps %xmm0, address-of-vec_res
```

mov a ps : **mov**e from mem to XMM register, memory **a**ligned, **p**acked **s**ingle precision

add ps : **add** from mem to XMM register, **p**acked **s**ingle precision

mov a ps : **mov**e from XMM register to mem, memory **a**ligned, **p**acked **s**ingle precision

Packed and Scalar Double-Precision Floating-Point Operations



Intel SSE Intrinsics

- Intrinsics are C functions and procedures for inserting assembly language into C code, including SSE instructions
  - With intrinsics, can program using these instructions indirectly
  - One-to-one correspondence between SSE instructions and intrinsics

Example SSE Intrinsics

Instrinsics:	Corresponding SSE instructions:
• 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/duplicating
• 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:

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

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

Example: 2 x 2 Matrix Multiply

- Using the XMM registers
  - Two 64-bit doubles per XMM reg

C_1	$C_{1,1}$	$C_{2,1}$
C_2	$C_{1,2}$	$C_{2,2}$

Stored in memory in Column-major order

A	$A_{1,i}$	$A_{2,i}$

B_1	$B_{i,1}$	$B_{i,1}$
B_2	$B_{i,2}$	$B_{i,2}$

Example: 2 x 2 Matrix Multiply

- Initialization

C_1	0	0
C_2	0	0

Example: 2 x 2 Matrix Multiply

- Initialization

C_1	0	0
C_2	0	0

- I = 1

A	$A_{1,1}$	$A_{2,1}$

_mm_load_pd: Load 2 doubles into XMM reg, Stored in memory in Column-major order

B_1	$B_{1,1}$	$B_{1,1}$
B_2	$B_{1,2}$	$B_{1,2}$

_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)

Example: 2 x 2 Matrix Multiply

- First iteration intermediate result

C_1	$0+A_{1,1}B_{1,1}$	$0+A_{2,1}B_{1,1}$
C_2	$0+A_{1,1}B_{1,2}$	$0+A_{2,1}B_{1,2}$

c1 = _mm_add_pd(c1,_mm_mul_pd(a,b1));
c2 = _mm_add_pd(c2,_mm_mul_pd(a,b2));
SSE instructions first do parallel multiplies and then parallel adds in XMM registers

- I = 1

A	$A_{1,1}$	$A_{2,1}$

_mm_load_pd: Stored in memory in Column order

B_1	$B_{1,1}$	$B_{1,1}$
B_2	$B_{1,2}$	$B_{1,2}$

_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)

Example: 2 x 2 Matrix Multiply

- First iteration intermediate result

C_1	$0+A_{1,1}B_{1,1}$	$0+A_{2,1}B_{1,1}$
C_2	$0+A_{1,1}B_{1,2}$	$0+A_{2,1}B_{1,2}$

c1 = _mm_add_pd(c1,_mm_mul_pd(a,b1));
c2 = _mm_add_pd(c2,_mm_mul_pd(a,b2));
SSE instructions first do parallel multiplies and then parallel adds in XMM registers

- I = 2

A	$A_{1,2}$	$A_{2,2}$

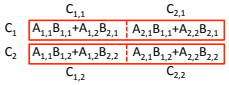
_mm_load_pd: Stored in memory in Column order

B_1	$B_{2,1}$	$B_{2,1}$
B_2	$B_{2,2}$	$B_{2,2}$

_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)

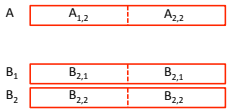
Example: 2 x 2 Matrix Multiply

- Second iteration intermediate result



c1 = _mm_add_pd(c1,_mm_mul_pd(a,b1));
c2 = _mm_add_pd(c2,_mm_mul_pd(a,b2));
SSE instructions first do parallel multiplies and then parallel adds in XMM registers

- I = 2



_mm_load_pd: Stored in memory in Column order

_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)

Live Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

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

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

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} = \begin{bmatrix} C_{1,1}= 1*1 + 0*2 = 1 & C_{1,2}= 1*3 + 0*4 = 3 \\ C_{2,1}= 0*1 + 1*2 = 2 & C_{2,2}= 0*3 + 1*4 = 4 \end{bmatrix}$$

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 several 128-bit vector variables
  __m128d c1,c2,a,b1,b2;

  // Initialize A, B, C for example
  /* 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;
```


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;
}
```


Inner loop from gcc –O -S

```
L2: movapd   (%rax,%rsi), %xmm1  //Load  aligned A[i,i+1]->m1
    movddup  (%rdx), %xmm0       //Load B[j], duplicate->m0
    mulpd    %xmm1, %xmm0        //Multiply m0*m1->m0
    addpd    %xmm0, %xmm3        //Add m0+m3->m3
    movddup  16(%rdx), %xmm0     //Load B[j+1], duplicate->m0
    mulpd    %xmm0, %xmm1        //Multiply m0*m1->m1
    addpd    %xmm1, %xmm2        //Add m1+m2->m2
    addq     $16, %rax           // rax+16 -> rax (i+=2)
    addq     $8, %rdx            // rdx+8 -> rdx (j+=1)
    cmpq     $32, %rax           // rax == 32?
    jne      L2                  // jump to L2 if not equal
    movapd   %xmm3, (%rcx)       //store aligned m3 into C[k,k+1]
    movapd   %xmm2, (%rdi)       //store aligned m2 into C[l,l+1]
```


Performance-Driven ISA Extensions

- Subword parallelism, used primarily for multimedia applications
  - Intel MMX: multimedia extension
    - 64-bit registers can hold multiple integer operands
  - Intel SSE: Streaming SIMD extension
    - 128-bit registers can hold several floating-point operands
- Adding instructions that do more work per cycle
  - Shift-add: replace two instructions with one (e.g., multiply by 5)
  - Multiply-add: replace two instructions with one (x := c + a × b)
  - Multiply-accumulate: reduce round-off error (s := s + a × b)
  - Conditional copy: to avoid some branches (e.g., in if-then-else)

Administrivia

- Lab #6, Project#2b posted
- Midterm Tuesday Oct 9, 8PM:
  - Two rooms: 1 Pimentel and 2050 LSB
  - Check your room assignment!
  - Covers everything through lecture Wednesday 10/3
  - Closed book, can bring one sheet notes, both sides
  - Copy of Green card will be supplied
  - No phones, calculators, …; just bring pencils & eraser
  - TA Review: Sun. Oct. 7, 3-5pm, 2050 VLSB

Midterm Room Assignment by Login

1 Pimentel = logins ab – mk

2050 VLSB = logins mm - xm

Midterm Review

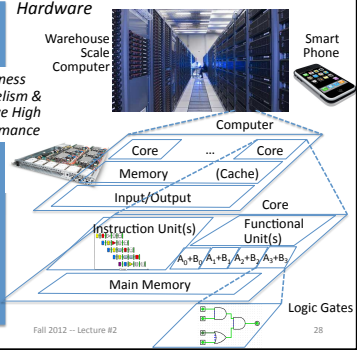
Topics we've covered

New-School Machine Structures (It's a bit more complicated!)

Software *Hardware*

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates functioning in parallel at same time

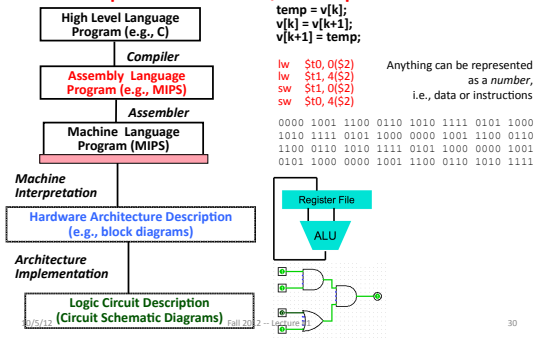
Harness Parallelism & Achieve High Performance



6 Great Ideas in Computer Architecture

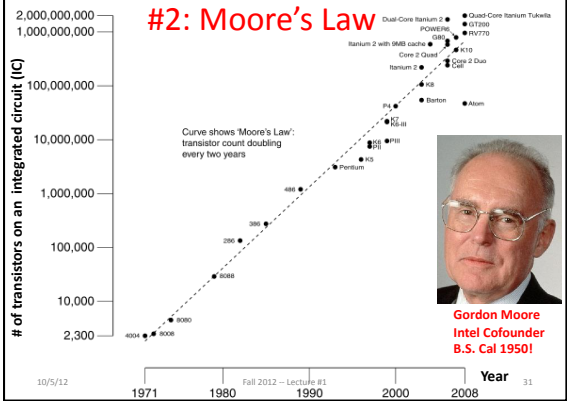
1. Layers of Representation/Interpretation
2. Moore's Law
3. Principle of Locality/Memory Hierarchy
4. Parallelism
5. Performance Measurement & Improvement
6. Dependability via Redundancy

Great Idea #1: Levels of Representation/Interpretation



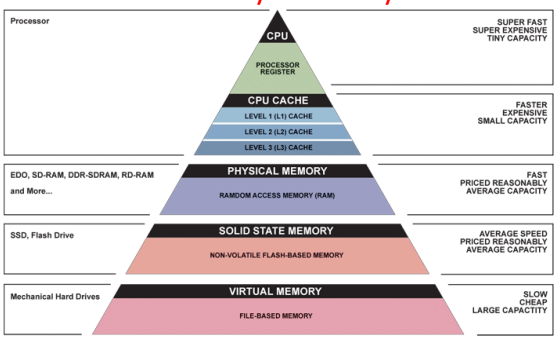
Predicts: 2X Transistors / chip every 2 years

#2: Moore's Law

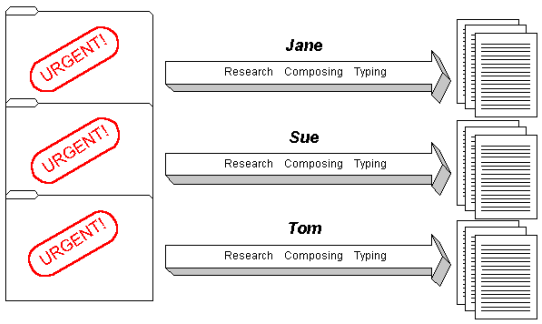


Gordon Moore
Intel Cofounder
B.S. Cal 1950!

Great Idea #3: Principle of Locality/ Memory Hierarchy



Great Idea #4: Parallelism

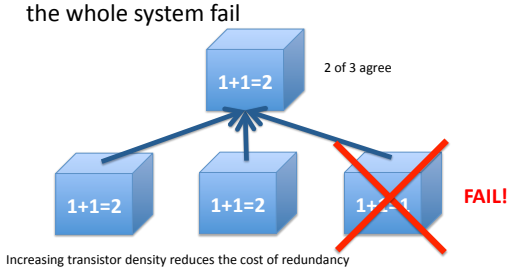


Great Idea #5: Performance Measurement and Improvement

- Match application to underlying hardware to exploit:
  - Locality
  - Parallelism
  - Special hardware features, like specialized instructions (e.g., matrix manipulation)
- Latency
  - How long to set the problem up
  - How much faster does it execute once it gets going
  - It is all about *time to finish*
- *Make common case fast!*

Great Idea #6: Dependability via Redundancy

- Redundancy so that a failing piece doesn't make the whole system fail



Increasing transistor density reduces the cost of redundancy

Warehouse-Scale Computers

- Power Usage Effectiveness
- Request-Level Parallelism
- MapReduce
- Handling failures
- Costs of WSC

C Language and Compilation

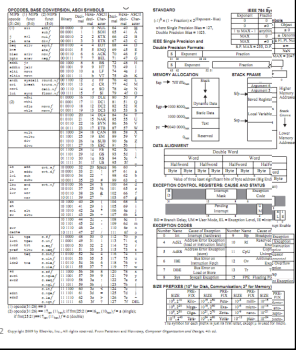
- C Types, including Structs, Consts, Enums
- Arrays and strings
- C Pointers
- C functions and parameter passing

MIPS Instruction Set

- ALU operations
- Loads/Stores
- Branches/Jumps

- Registers
- Memory

- Function calling conventions
- Stack

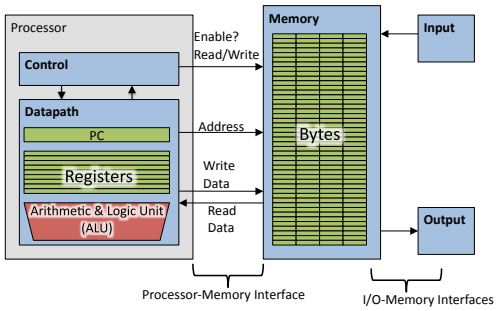


Everything is a Number

- Binary
- Signed versus Unsigned
- One's Complement/Two's Complement
- Floating-Point numbers
- Character Strings

- Instruction Encoding

Components of a Computer



Caches

- Spatial/Temporal Locality
- Instruction versus Data
- Block size, capacity
- Direct-Mapped cache
- 3 C's
- Cache-aware performance programming

Parallelism

- SIMD/MIMD/MISD/SISD
- Amdahl's Law
- Strong vs weak scaling
- Data-Parallel execution

…in Conclusion

- Intel SSE SIMD Instructions
  - One instruction fetch that operates on multiple operands simultaneously
  - 128-bit XMM registers
- SSE Instructions in C
  - Embed the SSE machine instructions directly into C programs through use of intrinsics
  - Achieve efficiency beyond that of optimizing compiler