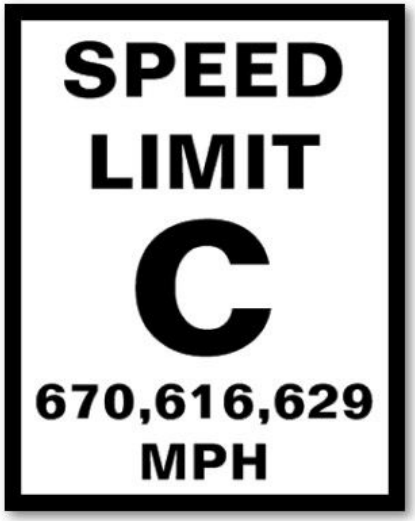
Great Ideas in Computer Architecture

DLP, Amdahl's Law, Intro to Multi-thread/processor systems

Instructor: Nick Riasanovsky



Review of Last Lecture

- Performance measured in *latency* or *bandwidth*
- Latency measurement for a program:
  - $\text{CPU Time} = \text{Instructions} \times \text{CPI} \times \text{Clock Cycle Time}$
- Flynn Taxonomy of Parallel Architectures
  - SIMD: Single Instruction Multiple Data
  - MIMD: Multiple Instruction Multiple Data
  - SISD: Single Instruction Single Data
  - MISD: Multiple Instruction Single Data (unused)
- Intel SSE SIMD Instructions
  - One instruction fetch that operates on multiple operands simultaneously
  - 128 bit XMM registers

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	

{SS} Scalar Single precision FP: **1** 32-bit operand in a 128-bit register

{PS} Packed Single precision FP: **4** 32-bit operands in a 128-bit register

{SD} Scalar Double precision FP: **1** 64-bit operand in a 128-bit register

{PD} Packed Double precision FP, or **2** 64-bit operands in a 128-bit register

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

{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 Single Precision FP 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:

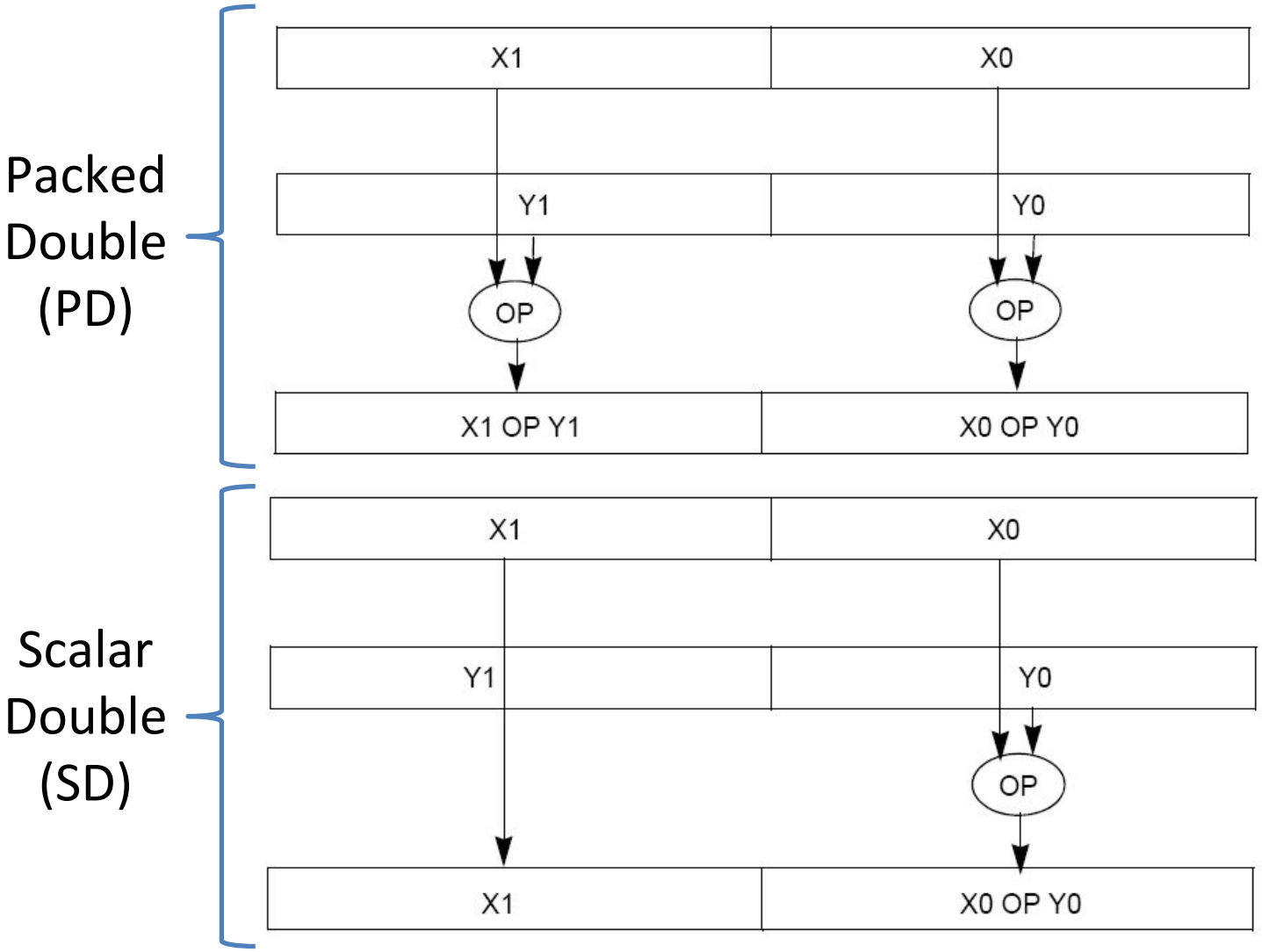
```
movaps  address-of-v1, %xmm0
   // v1.w | v1.z | v1.y | v1.x -> xmm0
addps   address-of-v2, %xmm0
   // v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x
  -> xmm0
movaps  %xmm0, address-of-vec_res
```

move from mem to XMM register, memory **a**ligned, **p**acked **s**ingle precision

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

move from XMM register to mem, memory **a**ligned, **p**acked **s**ingle precision

Packed and Scalar Double-Precision Floating-Point Operations



Example: Image Converter (1/5)

- Converts BMP (bitmap) image to a YUV (color space) 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
- Bitmap image consists of 8-bit monochrome pixels
  - By packing these pixel values in a 128-bit register, we can operate on 128/8 = 16 values at a time
  - Significant performance boost

Example: Image Converter (2/5)

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

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

Example: Image Converter (3/5)

- FP 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];
```

1) Regular x86 instructions for the inner loop:

```
fmul   […]
faddp  […]
```

   – Instructions executed: 4 * 2 = 8 (x86)

Example: Image Converter (4/5)

- FP 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];
```

2) SSE2 instructions for the inner loop:

```
//xmm0=p, xmm1=src1[i], xmm2=src2[i]
mulps %xmm1,%xmm2  // xmm2 * xmm1 -> xmm2
addps %xmm2,%xmm0  // xmm0 + xmm2 -> xmm0
```

– Instructions executed: 2 (SSE2)

Example: Image Converter (5/5)

- FP 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];
```

3) SSE5 accomplishes the same in **one** instruction:

```
fmaddps %xmm0, %xmm1, %xmm2, %xmm0
// xmm2 * xmm1 + xmm0 -> xmm0
// multiply xmm1 x xmm2 paired single,
// then add product paired single to sum
in xmm0
```

Agenda

- Intel SSE Intrinsics
- Administrivia
- Loop Unrolling
- Amdahl's Law
- Meet the Staff
- Multiprocessor Systems

Intel SSE Intrinsics

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

Sample of 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 × 2 Matrix Multiply

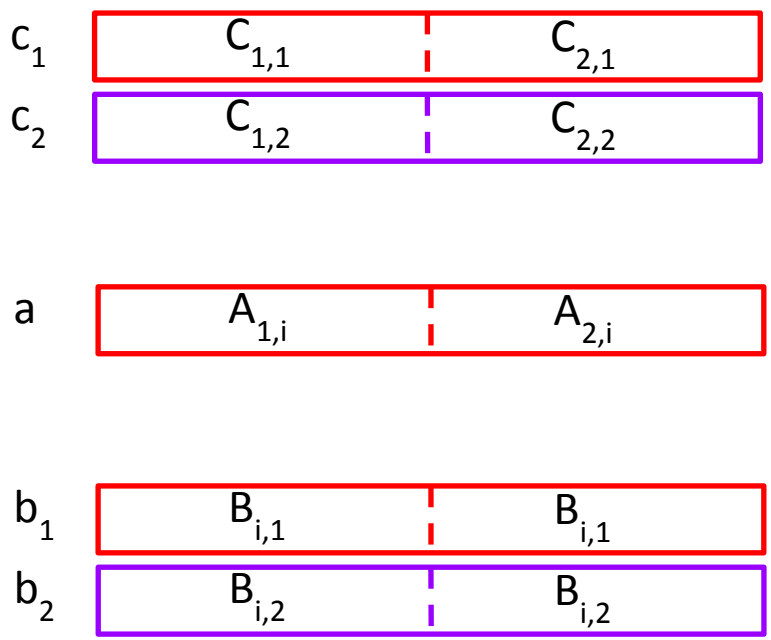
Definition of Matrix Multiply:

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

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

Example: 2 × 2 Matrix Multiply

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



Memory is column major

Example: 2 × 2 Matrix Multiply

- Initialization

c_1	0	0
c_2	0	0

- 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

Example: 2 × 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));
```

- 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

Example: 2 × 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));
```

- i = 2

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

Example: 2 × 2 Matrix Multiply

- Second iteration intermediate result

	$C_{1,1}$	$C_{2,1}$
c_1	$A_{1,1}B_{1,1}+A_{1,2}B_{2,1}$	$A_{2,1}B_{1,1}+A_{2,2}B_{2,1}$
c_2	$A_{1,1}B_{1,2}+A_{1,2}B_{2,2}$	$A_{2,1}B_{1,2}+A_{2,2}B_{2,2}$
	$C_{1,2}$	$C_{2,2}$

```
c1 = _mm_add_pd(c1,_mm_mul_pd(a,b1));
c2 = _mm_add_pd(c2,_mm_mul_pd(a,b2));
```

- i = 2

a	$\mathbf{A_{1,2}}$	$\mathbf{A_{2,2}}$

_mm_load_pd: Stored in memory in Column order

b_1	$\mathbf{B_{2,1}}$	$\mathbf{B_{2,1}}$
b_2	$\mathbf{B_{2,2}}$	$\mathbf{B_{2,2}}$

_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register

2 x 2 Matrix Multiply Code (1/2)

```
#include <stdio.h>
// header file for SSE4.2 compiler intrinsics
#include <nmmintrin.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;
/* continued on next slide */
```

2 x 2 Matrix Multiply Code (2/2)

```
// 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: [0 + a_11*b_11 | 0 + 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: [0 + a_11*b_12 | 0 + 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: two instructions in one (e.g. multiply by 5)
  - Multiply-add: two instructions in one (x := c + a * b)
  - Multiply-accumulate: reduce round-off error (s := s + a * b)
  - Conditional copy: avoid some branches (e.g. if-then-else)
  - Conditional greater than...

Agenda

- Intel SSE Intrinsics
- Administrivia
- Loop Unrolling
- Amdahl's Law
- Meet the Staff
- Multiprocessor Systems

Administrivia

- HW5 due 7/23, Proj3 due 7/20
- Proj 3 party on Fri (7/20), 4-6PM @Woz
- “Lost” Discussion Sat. Cory 540AB, 12-2PM
- Midterm 2 is coming up! Next Wed. in lecture
  - Covering up to Performance
  - Review Session Sunday 2-4pm @GPB 100
  - There will be discussion after MT2 :(

Agenda

- Intel SSE Intrinsics
- Administrivia
- Loop Unrolling
- Amdahl's Law
- Meet the Staff
- Multiprocessor Systems

Data Level Parallelism and SIMD

- SIMD wants adjacent values in memory that can be operated in parallel
- Usually specified in programs as loops

```
for(i=0; i<1000; i++)
  x[i] = x[i] + s;
```

- How can we reveal more data level parallelism than is available in a single iteration of a loop?
  - *Unroll the loop* and adjust iteration rate

Looping in RISC-V

Assumptions:

s0 → initial address (top of array)

s1 → scalar value s

s2 → termination address (end of array)

```
Loop:
      lw     t0, 0(s0)
      addu   t0,t0,s1      # add s to array element
      sw     t0,0(s0)      # store result
      addi   s0,s0,4       # move to next element
      bne    s0,s2,Loop    # repeat Loop if not done
```

Loop Unrolled

```
Loop:   lw   t0,0(s0)
        add  t0,t0,s1
        sw   t0,0(s0)
        lw   t1,4(s0)
        add  t1,t1,s1
        sw   t1,4(s0)
        lw   t2,8(s0)
        add  t2,t2,s1
        sw   t2,8(s0)
        lw   t3,12(s0)
        add  t3,t3,s1
        sw   t3,12(s0)
        addi s0,s0,16
        bne s0,s2,Loop
```

NOTE:

1. Loop overhead (addiu, bne) encountered only once every 4 data iterations

2. This unrolling works if `loop_limit mod 4 = 0`

3. Using different registers allows us to eliminate stalls by reordering...

Loop Unrolled Scheduled

Note: We just switched from integer instructions to single-precision FP instructions!

```
Loop:   flw     t0,0(s0)
        flw     t1,4(s0)
        flw     t2,8(s0)
        flw     t3,12(s0)
        fadd.s  t0,t0,s1
        fadd.s  t1,t1,s1
        fadd.s  t2,t2,s1
        fadd.s  t3,t3,s1
        fsw     t0,0(s0)
        fsw     t1,4(s0)
        fsw     t2,8(s0)
        fsw     t3,12(s0)
        addi    s0,s0,16
        bne     s0,s2,Loop
```

4 Loads side-by-side:
Could replace with 4 wide SIMD Load

4 Adds side-by-side:
Could replace with 4 wide SIMD Add

4 Stores side-by-side:
Could replace with 4 wide SIMD Store

Can SIMD-ize AND unroll if desired

Loop Unrolling in C

- Instead of compiler doing loop unrolling, could do it yourself in C:

```
for(i=0; i<1000; i++)
  x[i] = x[i] + s;
```

Loop Unroll

```
for(i=0; i<1000; i=i+4) {
  x[i]   = x[i]   + s;
  x[i+1] = x[i+1] + s;
  x[i+2] = x[i+2] + s;
  x[i+3] = x[i+3] + s;
}
```

What is downside of doing this in C?

Generalizing Loop Unrolling

- Take a loop of **n iterations** and perform a **k-fold** unrolling of the body of the loop:
  - First run the loop with k copies of the body **floor(n/k)** times
  - To finish leftovers, then run the loop with 1 copy of the body **n mod k** times

Drawbacks to Loop Unrolling

- Loop unrolling can greatly speedup your code but isn't perfect for a couple of reasons
  - If you are doing it by hand its a really inefficient/tedious task
    - In reality you would want your compiler to do this but we want you to understand it
  - Loop unrolling increases your static code size
    - Static code size is important for accesses to your instruction cache
    - You might not want k to be too large
    - Try find a balance between less executed instructions and small static code size

Code Optimization

- Loop unrolling isn't really a form of parallelism but is instead an example of code optimization
  - Code is converted from a form easy to understand to one with better performance
- This is often the work of your compiler but it may not always be able to make the best optimizations
- Let's consider another example of how you can optimize your code

Loop Invariants

```
for (int i = 0; i < n; i++) {
    arr[i] = (f(x) - g(y)) * arr[i];
}
```

- This is an example of what we call a loop invariant
  - Invariant meaning does not change in the loop
- What happens if f and g are expensive?
  - Then f and g are computed each iteration, n times in total
  - But the loop recomputes the result

Loop Invariants

```
z = (f(x) - g(y))
for (int i = 0; i < n; i++) {
    arr[i] = z * arr[i];
}
```

- Solution: Move the code outside of the loop and only compute it once since it never changes
  - Now n expensive calls has become 1 expensive call
- But can we do better?

Loop Invariants

- What happens is f and/or g is really really expensive
  - We want compute it as little as possible
- Now we always compute it once
- But what happens if n <= 0
  - Then we compute the invariant once
  - But we never enter the loop so we never use it
- Solution: Add a check to avoid computing it if we don't enter the loop

Loop Invariants

```
if (n > 0) {
    z = f(x) - g(y);
    for (int i = 0; i < n; i++) {
      arr[i] += z * arr[i];
    }
}
```

Now we compute the invariant once if we enter the loop and otherwise not at all

Agenda

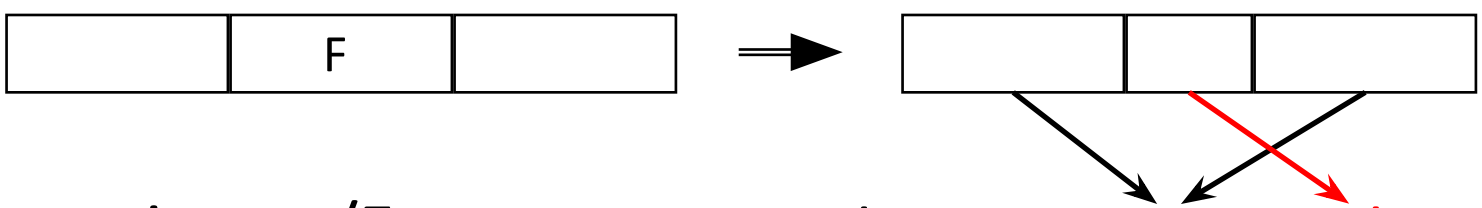
- Intel SSE Intrinsics
- Administrivia
- Loop Unrolling
- Amdahl's Law
- Meet the Staff
- Multiprocessor Systems

Amdahl's Law

- Speedup due to enhancement E:

$$\text{Speedup w/E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/E}}$$

- **Example:** Suppose that enhancement E accelerates a fraction F (F<1) of the task by a factor S (S>1) and the remainder of the task is unaffected



- Exec time w/E = Exec Time w/o E × [(1-F) + F/S]

  Speedup w/E = 1 / [(1-F) + F/S]

Amdahl's (Amdahl's Leaky) Law

- Speedup = $\dfrac{1}{(1 - F) + \dfrac{F}{S}}$

Non-speed-up part → $(1 - F)$

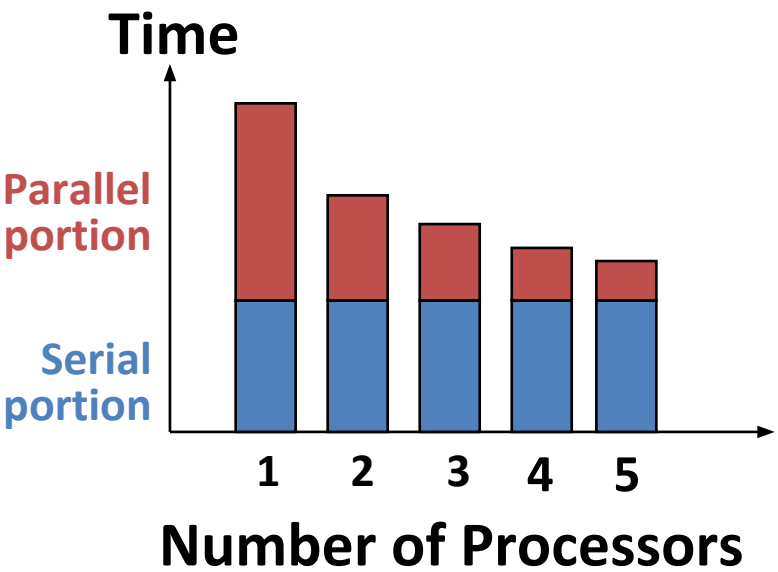
Speed-up part → $\dfrac{F}{S}$

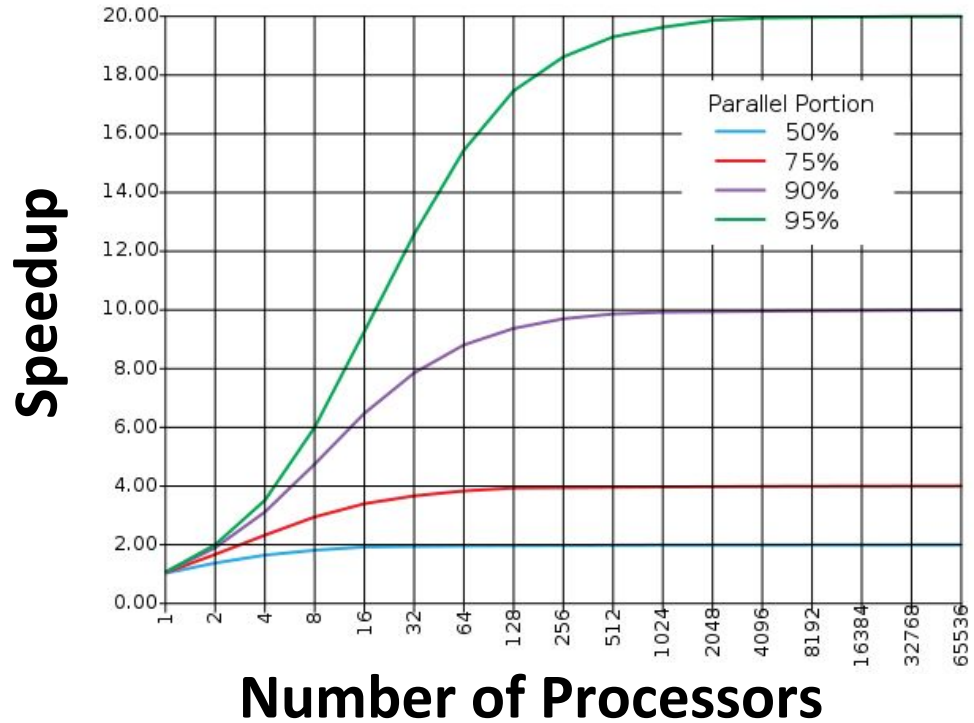
- **Example:** the execution time of half of the program can be accelerated by a factor of 2. What is the program speed-up overall?

$$\frac{1}{0.5 + \frac{0.5}{2}} = \frac{1}{0.5 + 0.25} = 1.33$$

Amdahl's (Heartbreaking) Law

- The amount of speedup that can be achieved through parallelism is limited by the non-parallel portion of your program!





Parallel Speed-up Examples (1/3)

Speedup w/ E = 1 / [(1-F) + F/S]

- Consider an enhancement which runs 20 times faster but which is only usable 15% of the time

  Speedup = 1/(.85 + .15/20) = 1.166

- What if it's usable 25% of the time?

  Speedup = 1/(.75 + .25/20) = 1.311

Nowhere near 20x speedup!

- Amdahl's Law tells us that to achieve linear speedup with more processors, none of the original computation can be scalar (non-parallelizable)
- To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less

  Speedup = 1/(.001 + .999/100) = 90.99

Parallel Speed-up Examples (2/3)

$Z_1 + Z_2 + \ldots + Z_{10}$

$$\begin{bmatrix} X_{1,1} & X_{1,10} \\ \vdots & \\ X_{10,1} & X_{10,10} \end{bmatrix} + \begin{bmatrix} Y_{1,1} & Y_{1,10} \\ \vdots & \\ Y_{10,1} & Y_{10,10} \end{bmatrix}$$

Partition 10 ways and perform on 10 parallel processing units

Non-parallel part

Parallel part

- 10 "scalar" operations (non-parallelizable)
- 100 parallelizable operations
  - Say, element-wise addition of two 10x10 matrices.
- 110 operations
  - 100/110 = .909 Parallelizable, 10/110 = 0.091 Scalar

Parallel Speed-up Examples (3/3)

Speedup w/ E = 1 / [(1-F) + F/S]

- Consider summing 10 scalar variables and two 10 by 10 matrices (matrix sum) on 10 processors

  Speedup = 1/(.091 + .909/10) = 1/0.1819 = 5.5

- What if there are 100 processors ?

  Speedup = 1/(.091 + .909/100) = 1/0.10009 = 10.0

- What if the matrices are 100 by 100 (or 10,010 adds in total) on 10 processors?

  Speedup = 1/(.001 + .999/10) = 1/0.1009 = 9.9

- What if there are 100 processors ?

  Speedup = 1/(.001 + .999/100) = 1/0.01099 = 91

Question: Suppose a program spends 80% of its time in a square root routine. How much must you speed up square root to make the program run 5 times faster?

$$\text{Speedup w/ E} = 1 / [\ (1 - F) + F/S\ ]$$

(A) 10

(B) 20

(C) 100

(D) None of the above

Question: Suppose a program spends 80% of its time in a square root routine. How much must you speed up square root to make the program run 5 times faster?

$$\text{Speedup w/ E} = 1 / [\,(1 - F) + F/S\,]$$

$$5 = 1 / [\,(1 - 0.8) + 0.8/S]$$

$$S = 0.8 / ((1/5) - 0.2) = \infty$$

(A) 10

(B) 20

(C) 100

(D) None of the above

Agenda

- Intel SSE Intrinsics
- Administrivia
- Loop Unrolling
- Amdahl's Law
- Meet the Staff
- Multiprocessor Systems

Meet the Staff

	 Steven	Nick 
Trash TV Show	The Bachelorette	Mike Tyson Mysteries
Favorite Plot Twist	Get Out	Hot Fuzz
Best Bathroom in Cal	MLK Secret Bathroom	Campbell Hall
Best Study Spot	O'Brien Hall	Cory 540AB

Agenda

- Intel SSE Intrinsics
- Administrivia
- Loop Unrolling
- Amdahl's Law
- Meet the Staff
- Multiprocessor Systems

Great Idea #4: Parallelism

Software

- Parallel Requests
  Assigned to computer
  e.g. search "Garcia"
- 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

Leverage Parallelism & Achieve High Performance

Hardware

Warehouse Scale Computer

Smart Phone

Computer

Core … Core

Memory

Input/Output

Core

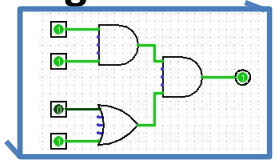
Instruction Unit(s)

Functional Unit(s)

A_0+B_0 | A_1+B_1 | A_2+B_2 | A_3+B_3

Cache Memory

Logic Gates



Threads

- *Thread of execution:* Smallest unit of processing scheduled by operating system
- On uniprocessor, multithreading occurs by *time-division multiplexing*
  - Processor switches between different threads
  - *Context switching* happens frequently enough user perceives threads as running at the same time
- On a multiprocessor, threads run at the same time, with each processor running a thread

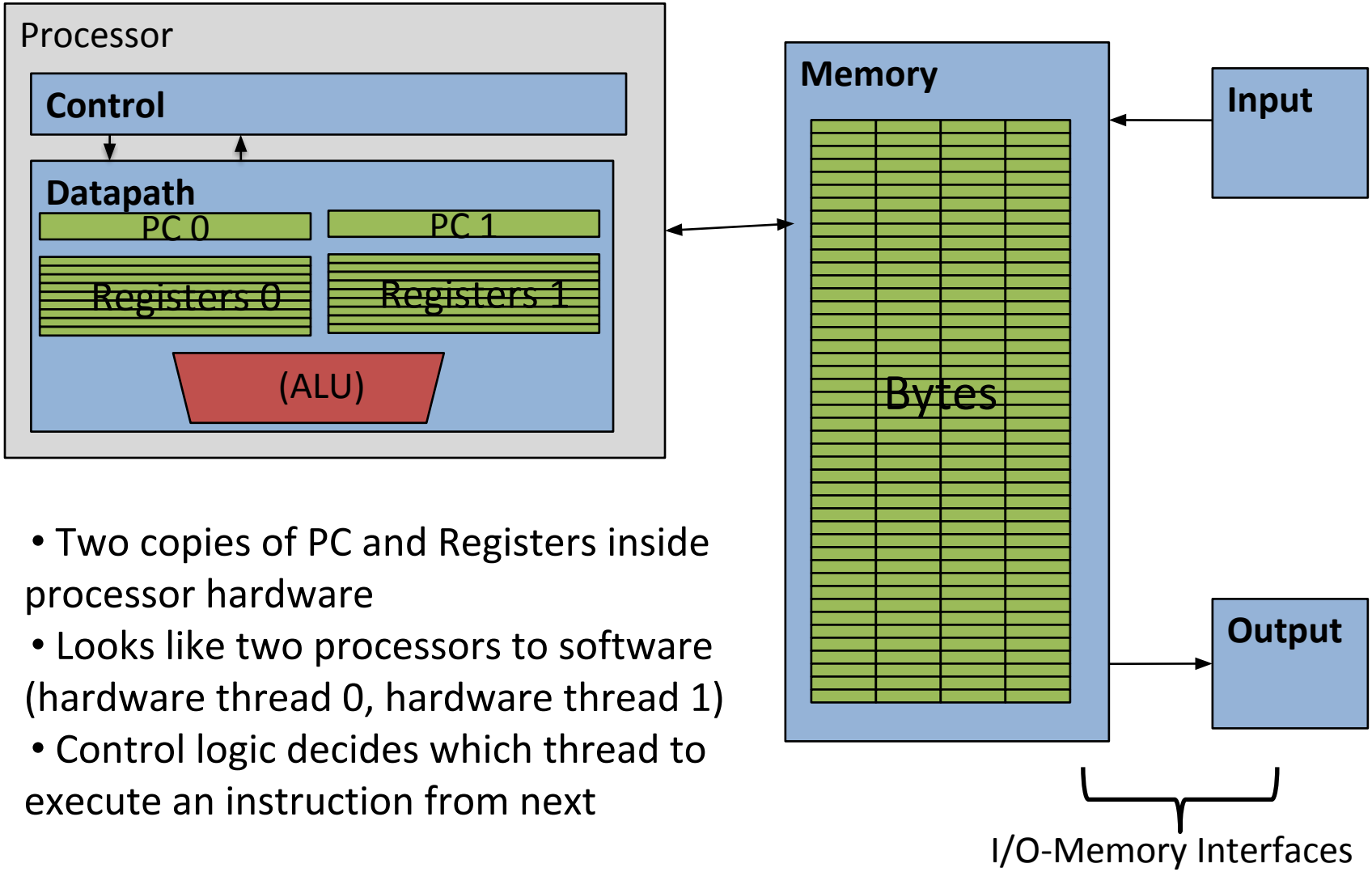
Terminology

- Program: An executable
  - Example your server from proj1
- Process: A running program or portion of program that completes some task
  - Example a running instance of your server
- Software Thread: A unit of processing that is described by the process running and the values of the registers
- Hardware Thread: Hardware to allow the execution of a single software thread

Multithreading

- **Basic idea:** Processor resources are expensive and should not be left idle
- Long memory latency to memory on cache miss?
  - Hardware switches threads to bring in other useful work while waiting for cache miss
  - Cost of thread context switch must be much less than cache miss latency
- Put in redundant hardware so don't have to save context on every thread switch:
  - PC, Registers, L1 caches?

Hardware Support for Multithreading



- Two copies of PC and Registers inside processor hardware
- Looks like two processors to software (hardware thread 0, hardware thread 1)
- Control logic decides which thread to execute an instruction from next

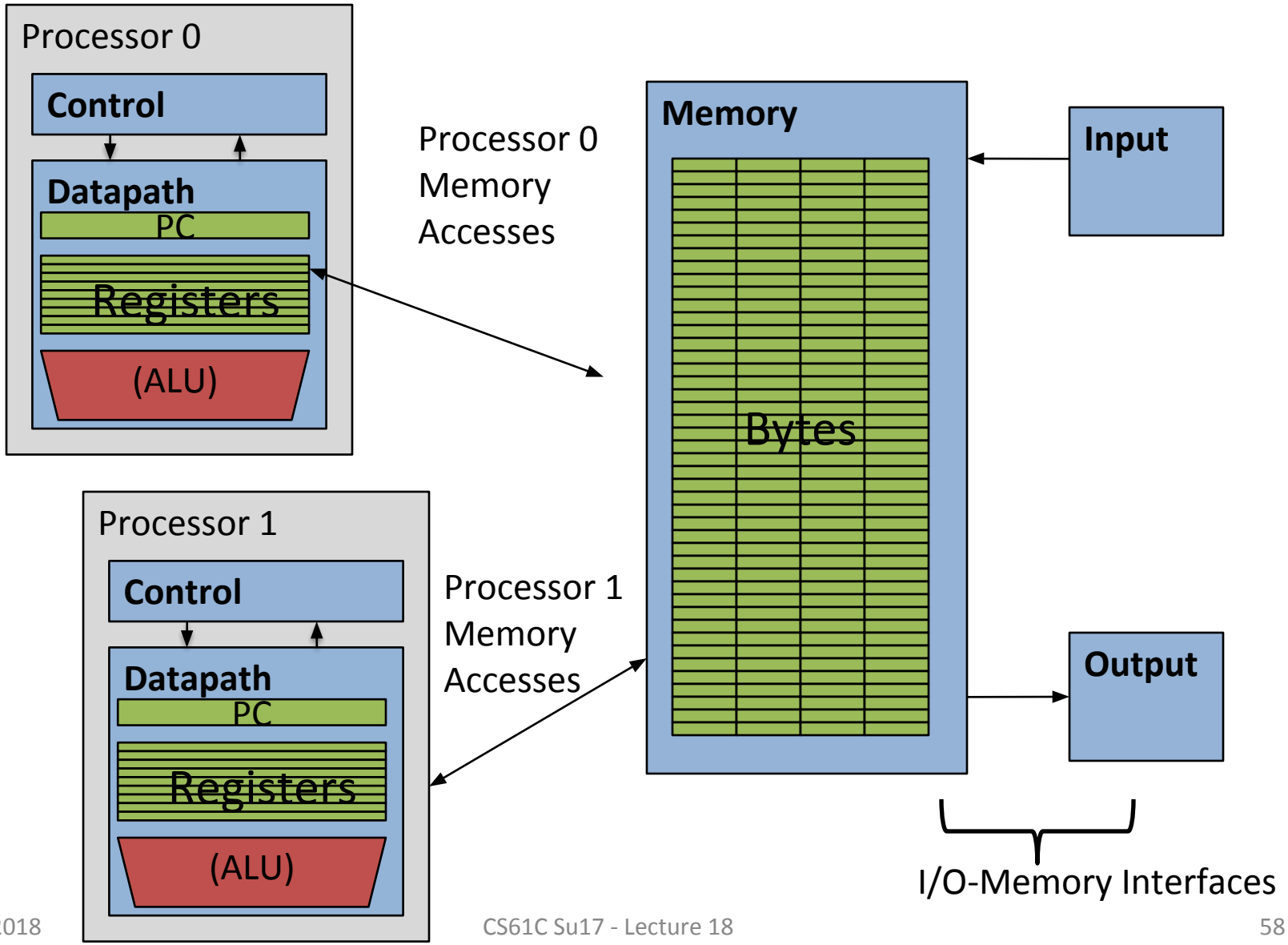
Multiprocessor Systems (MIMD)

- Multiprocessor (MIMD): a computer system with at least 2 processors or *cores* ("multicore")
  - Each core has its own PC and executes an independent instruction stream
  - Processors share the same memory space and can communicate via loads and stores to common locations

1. Deliver high throughput for independent jobs via request-level or task-level parallelism

2. *Improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel processing program*

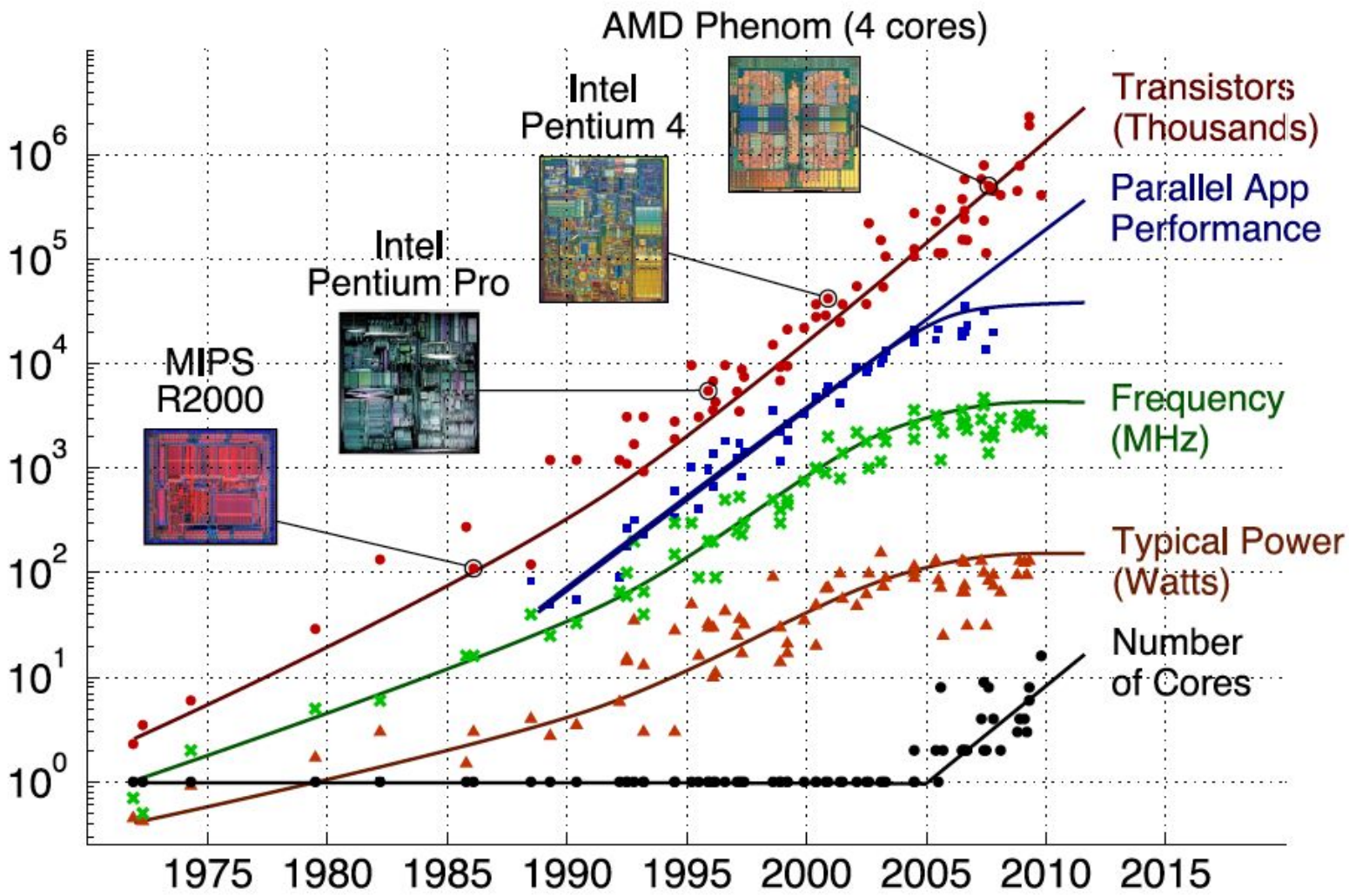
Multiprocessor Systems



Multithreading vs. Multicore

- Multithreading => Better Utilization
  - ≈1% more hardware, 1.10X better performance?
  - Share integer adders, floating point adders, caches (L1 I, L1 D, L2 cache, L3 cache), Memory Controller
- Multicore => Duplicate Processors
  - ≈50% more hardware, ≈2X better performance?
  - Share some caches (L2 cache, L3 cache), Memory Controller
- Modern machines do both
  - Multiple cores with multiples threads per core

Transition to Multicore



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

Summary

- Intel SSE SIMD Instructions
  - One instruction fetch that operates on multiple operands simultaneously
  - 128/64 bit XMM registers
  - Embed the SSE machine instructions directly into C programs through use of **intrinsics**
- Loop Unrolling: Access more of array in each iteration of a loop
- Amdahl's Law limits benefits of parallelization
- Multiprocessor systems use shared memory (single address space)