15-745
Introduction

Seth Copen Goldstein
Seth@cs.cmu.Edu
CMU

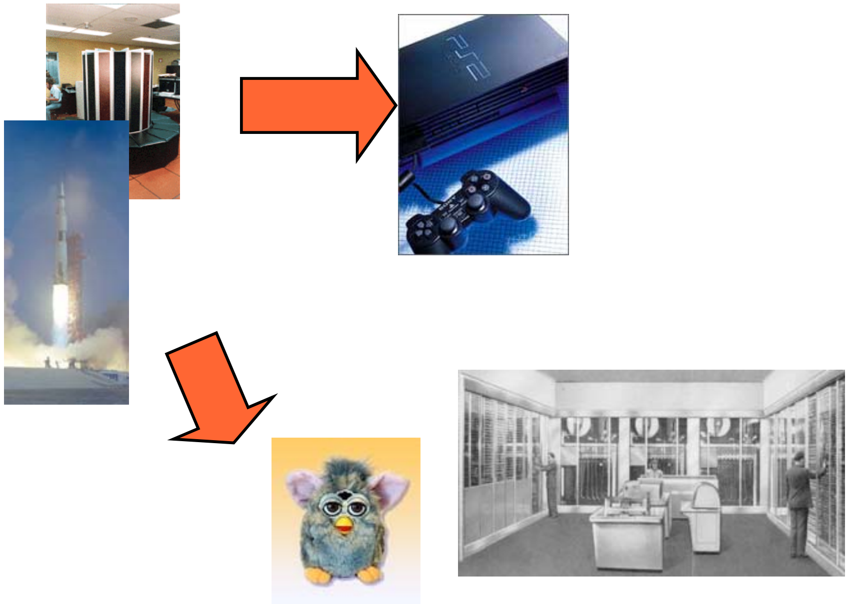
Based in part on slides by
Todd Mowry and Michael Voss

Introduction

- Why study compilers?
- Administriva
- Structure of a Compiler
- Optimization Example

Reference: Muchnick 1.3-1.5

Moore's Law



Moore's Law

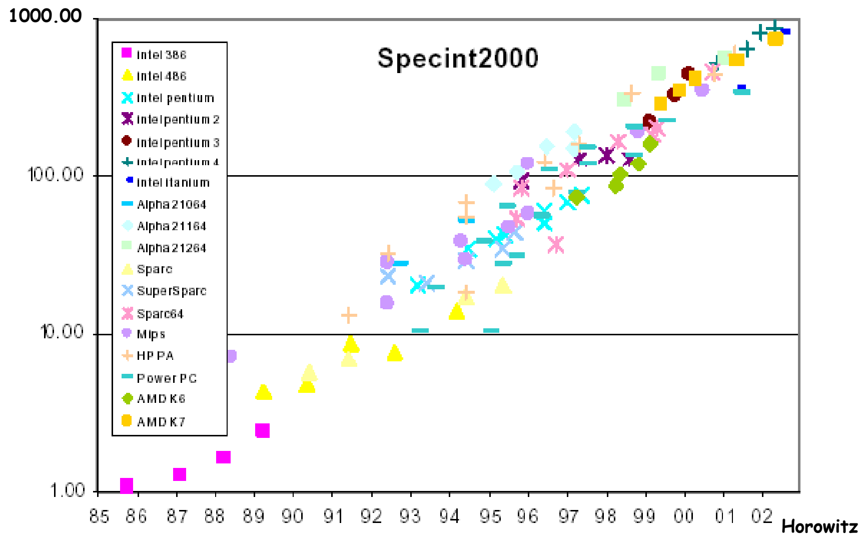
Imagine: Computers that

Imagining it is hard enough, achieving it requires a rethink of the entire tool chain.

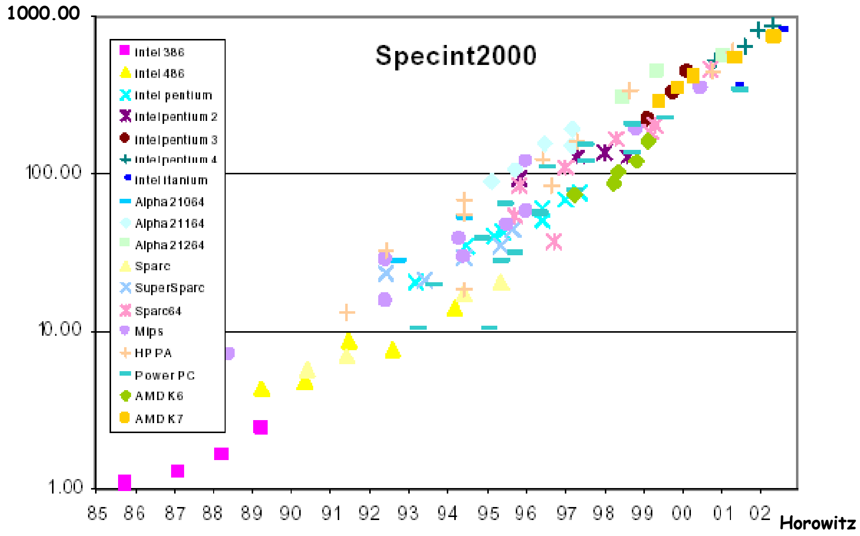
What is Behind Moore's Law?

- A lot of hard work!
- Two most important tools:
  - Parallelism
    - Bit-level
    - Pipeline
    - Function unit
    - Multi-core
  - Locality

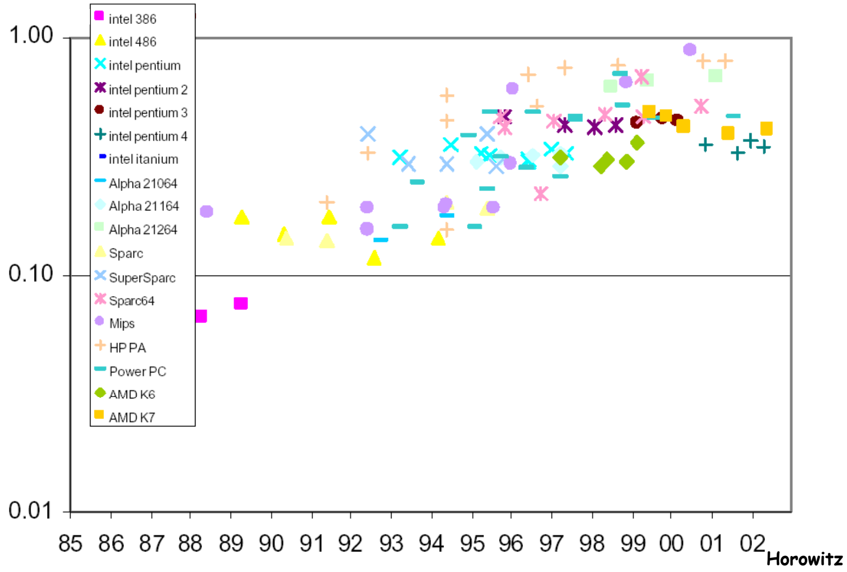
Performance: Ops/Sec



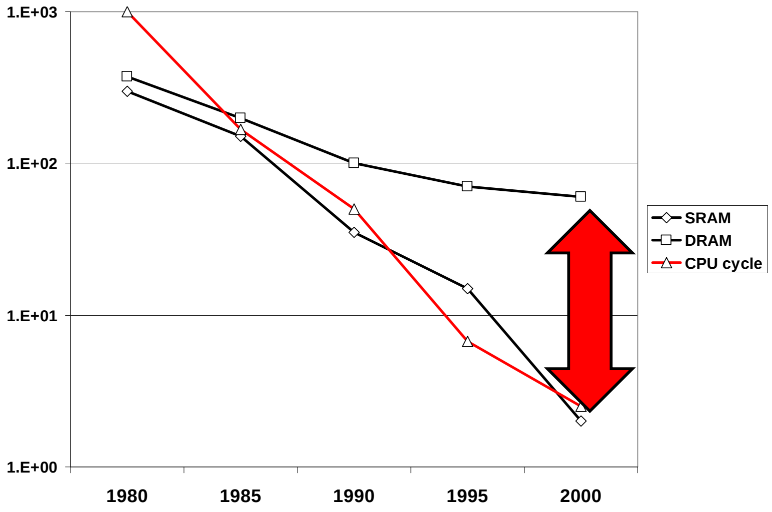
Performance: Ops/Clk * Clks/Sec



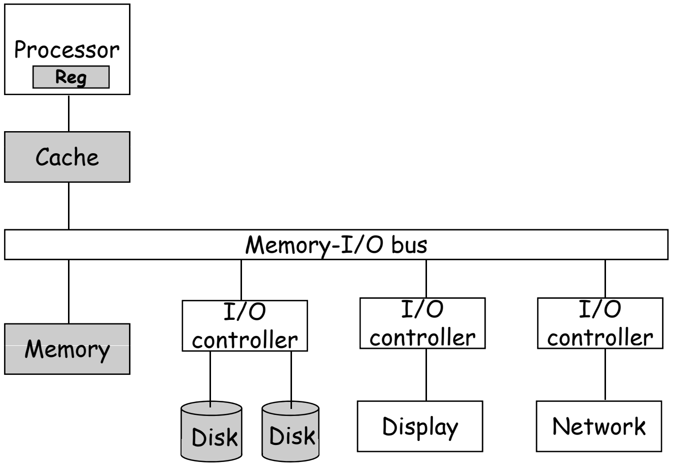
SpecInt/Mhz



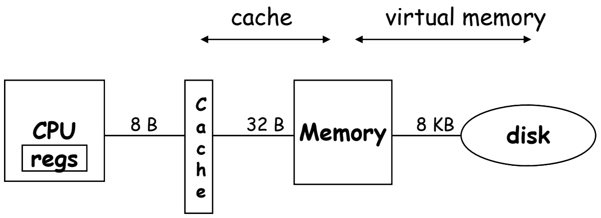
Another View of Moore's Law



The Computer System



The Memory Hierarchy



	Register	Cache	Memory	Disk Memory
size:	200 B	32 KB/4MB	128 MB	20 GB
speed:	3 ns	6 ns	60 ns	8 ms
$/Mbyte:		$100/MB	$.30/MB	$0.005/MB
block size:	8 B	32 B	8 KB	

larger, slower, cheaper

Compiler Writer's Job

- Improve locality
- Increase parallelism
- Tolerate latency
- Reduce power

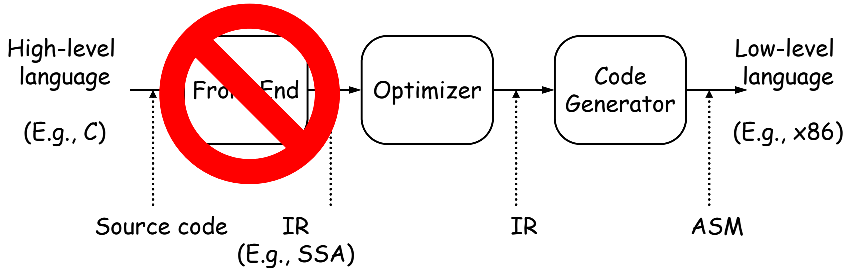
Why study compilers

- They are really amazing
- Combines theory & practice
  - CS is about abstraction
    - Primary abstraction: programming language
    - Compiler lowers PL to ISA (or further!)
  - Compiler is a big system
- Crucial for performance
  - especially for modern processors
  - practically part of the architecture
- I bet: Everyone will write a compiler

Why study compilers

- They are really amazing
- Combines theory & practice
  - CS is about abstraction
    - Primary abstraction: programming language
    - Compiler lowers PL to ISA (or further!)
  - Compiler is a big system
- Crucial for performance
  - especially for modern processors
  - practically part of the architecture
- I bet: Everyone will write a compiler

What this course is about



- Theory and practice of modern optimizing compilers
  - No lexing or parsing
  - Focus on IR, back-end, optimizations
- Internals of today's (and tomorrow's) compilers
- Working with a real compiler

Prerequisites

- 211 & 213 or the equivalent
- Parts of 411 or the equivalent
  - Basic compiler data structures
  - Frames, calling conventions, def-use chains, etc.
  - Don't really care about front-end
- Proficient in C/C++ programming
- Basic understanding of architecture

My Expectations

- You have the prerequisites
  - If not come see me asap
- 3 assignments + a project
- Class participation
  - THIS IS A MUST!
  - Read text/papers before class
  - Attendance is essentially mandatory

Grading

• Class participation	~20%
– Throughout the semester	
– During paper presentations	
– Project presentations	
• assignments	~20%
• Project	~40%
• Midterm	~20%

Assignments

- Intro to LLVM/Liveness
- Dependence analysis
- Locality/Parallel transformations

- All labs and the final project will be done in a state-of-the-art research compiler: LLVM

The Text

- No assigned text. Its really up to you.
- Muchnick, Advanced Compiler Design & Impl., 1997
- Allen, et.al., Optimizing Compilers for Modern Archs, 2001
- Copper, et.al., Engineering a compiler, 2003
- Aho, et.al., Compilers: …, 2006

- Papers will be assigned

Before we get too bored

- More admin at the end, but first …
- What exactly is an optimizing compiler?
  - An optimizing compiler transforms a program into an equivalent, but "better" form.
  - What is equivalent?
  - What is better?

Full Employment Theorem

- No such thing as "The optimizing compiler"
  - Why not?
- There is always a better optimizing compiler, but …
  - Compiler must preserve correctness
  - On average improve X, where X is:
    - Performance
    - Power
    - …
  - Finish in your lifetime

How might performance be improved?

$$\text{execution time} = \sum_{\text{instructions}} \text{cycles per instruction}$$

- Reduce the number of instructions
- Replace "expensive" instructs with "cheap" ones
- Reduce memory cost
  - Improve locality
  - Reduce # of memory operations
- Increase parallelism

Ingredients to a compiler opt

- Identify opportunity
  - Avail in many programs
  - Occurs in key areas (what are these?)
  - Amenable to "efficient" algorithm
- Formulate Problem
- Pick a Representation
- Develop an Analysis
  - Detect when legal
  - And desirable
- Implement Code Transformation
- Evaluate (and repeat!)

Examples of Optimizations

- Machine Independent
  - Algebraic simplification
  - Constant propagation
  - Constant folding
  - Common Sub-expression elimination
  - Dead Code elimination
  - Loop Invariant code motion
  - Induction variable elimination
- Machine Dependent
  - Jump optimization
  - Reg allocation
  - Scheduling
  - Strength reduction
  - Loop permutations

Really Powerful Opts we won't do

- How to optimize:

```
Sumfrom1toN(int max) {
  sum = 0;
  for (i=1; i<=max; i++) sum+=i;
  return sum;
}
```

Really Powerful Opts we won't do

- How to optimize:

```
Sumfrom1toN(int max) {
  sum = 0;
  for (i=1; i<=max; i++) sum+=i;
  return sum;
}
```

- What we should, but won't do:

```
inline sumfrom1toN(int max) {
  return max > 0 ?
   ((max+max*max)>>1) : 0;
}
```

Algebraic Simplifications

a*1; ⇒ a

a/1; ⇒ a

a*0; ⇒ 0

a+0; ⇒ a

a-0; ⇒ a

a = b + 1
c = a - 1 ⇒ c = b

Use algebraic identities to simplify computations

Jump Optimizations

```
     cmp d0,d1              cmp d0,d1
     beq  L1
     bra  L2                bne  L2
L1:         :     ⇒    L1:         :
            :                      :
L2:         :          L2:         :
       :                      :
```

Simplify jump and branch instructions.

Constant Propagation

```
a = 5;                              a = 5;
b = 3;                              b = 3;
   :                                   :
   :                                   :
n = a + b;                    ⇒     n = 5 + 3;
for (i = 0; i<n; ++i)               for (i = 0; i<n; ++i)
{                                   {
      :                                   :
}                                   }
```

If the compiler can determine that the values of **a** and **b** are constants, then it can replace the variable uses with constant values.

Constant Folding

```
   :
   :
   :
   :
n =   8  ;
for (i = 0 ; i < 8 ; ++i) {
      :
}
```

- The compiler evaluates an expression (at compile time) and inserts the result in the code.
- Can lead to further optimization opportunities; esp. constant propagation.

Common Subexpression Elimination (CSE)

```
a = c*d;                    a = c*d;
      :                           :
      :          ⇒                :
d = (c*d + t) * u           d = (a + t) * u
```

If the compiler can determine that:

- an expression was previously computed
- and that the values of its variables have not changed since the previous computation,

Then, the compiler can use the previously computed value.

Strength Reduction

- On some processors, the cost of an addition is less than the cost of multiplication.
- The compiler can replace expensive multiplication instructions by less expensive ones.

```
c = b * 2;

move $2000, d0
muls  #2, d0
move d0, $3000
```

```
c = b + b;

move $2000, d0
add   d0, d0
move d0, $3000
```

```
c = lsh(b);

move $2000, d0
lsl      #1, d0
move d0, $3000
```

```
c = -1*b;

move $2000, d0
muls  #-1,d0
move d0, $3000
```

```
c = negative(b);

move $2000, d0
neg   d0
move d0, $3000
```

Dead Code Elimination

```
debug = False;
      :
      :
if (debug) {
     :
     :
}
a = f(b);
```

If the compiler can determine that code will never be executed or that the result of a computation will never be used, then it can eliminate the code or the computation.

Loop Invariant Code Motion

```
for (i=0; i<100 ;  ++i) {
  for (j=0; j<100 ; ++j) {
    for (k=0 ; k<100 ; ++k)
    {
       a[i][j][k] = i*j*k;
    }
  }
}
```

```
for (i=0; i<100 ;  ++i) {
  for (j=0; j<100 ; ++j) {
    t1 = a[i][j];
    t2 = i*j;
    for (k=0 ; k<100 ; ++k)
    {
       t1[k] = t2*k;
    }
  }
}
```

- Loop invariant: expression evaluates to the same value each iteration of the loop.
- Code motion: move loop invariant outside loop.
- Very important because inner-most loop executes most frequently.

Loop Invariant Code Motion

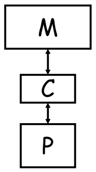
```
int *a;
int n;
    :
    :
scanf("%d", &n);
for (i=0; i<n ;  ++i) {
 for (j=0; j<n ; ++j) {
  for (k=0 ; k<n ; ++k)
  {
    f = q/p;
    a[i][j][k] = f*i*j*k;
  }
 }
}
```

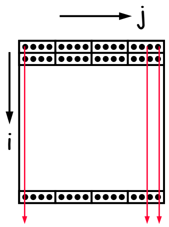
```
int *a;
int n;
    :
    :
scanf("%d", &n);
f = q/p;
for (i=0; i<n ;  ++i) {
 for (j=0; j<n ; ++j) {
  t1 = a[i][j];
  t2 = i*j;
  for (k=0 ; k<n ; ++k)
  {
     t1[k] = f*t2*k;
  }
 }
}
```

Oooops!!!!!

Cache Optimizations

```
for (j=0; j<n ;  ++j) {
   for (i=0; i<n ; ++i) {
         x += a[i][j];
   }
}
```



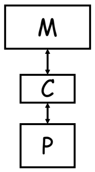


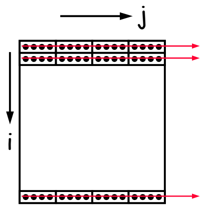
Loop permutation changes the order of the loops to improve the spatial locality of a program.

Cache Optimizations

```
for (j=0; j<n ;  ++j) {
   for (i=0; i<n ; ++i) {
         x += a[i][j];
   }
}
```

```
for (i=0; i<n ;  ++i) {
   for (j=0; j<n ; ++j) {
         x += a[i][j];
   }
}
```



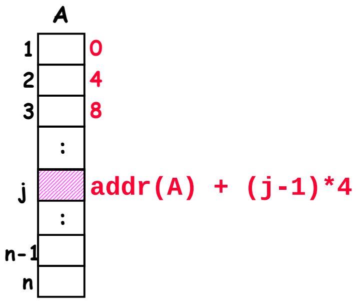


Loop permutation changes the order of the loops to improve the spatial locality of a program.

Example

A program that sorts 4-byte elements in an n-element array of integers A[1..n] using bubblesort.

```
for (i=n-1; i >= 1 ; --i) {
    for (j = 1; j <= i ; ++j) {
         if (A[j] > A[j+1]) {
             temp = A[j];
             A[j] = A[j+1];
             A[j+1] = temp;
         }
    }
}
// i and j are not used later
```



A Generated IR

```
for i {        i = n-1
          S5:  if i < 1 goto Exit
for j {        j = 1
          S4:  if j > i goto S2
A[j]  {        t1 = j-1
               t2 = 4*t1
               t3 = [A+t2]
A[j+1]{        t4 = j+1
               t5 = t4-1
               t6 = 4*t5
               t7 = [A+t6]
if    {        if t3 <= t7 goto S3
A[j]  {        t8 = j-1
               t9 = 4*t8
temp= {        temp = [A+t9]
A[j]
```

```
     t10 = j+1          }
     t11 = t10-1        }A[j+1]
     t12 = 4*t11        }
     t13 = [A+t12]      }
     t14 = j-1          }A[j]
     t15 = 4*t14
     [A+t15] = t13      }A[j]=A[j+1]
     t16 = j+1          }
     t17 = t16-1        }A[j+1]
     t18 = 4*t17
     [A+t18] = temp     }A[j+1]=temp

S3:  j = j+1            }for j
     goto S4
S2:  i = i-1            }for i
     goto S5
Exit:
```

Optimizations I - Algebraic Simplifications

```
      i = n-1
S5:   if i < 1 goto Exit
      j = 1
S4:   if j > i goto S2
      t1 = j-1
      t2 = 4*t1
      t3 = [A+t2]
      t4 = j+1
      t5 = t4-1
      t6 = 4*t5
      t7 = [A+t6]
      if t3 <= t7 goto S3
      t8 = j-1
      t9 = 4*t8
      temp = [A+t9]
```

```
      t10 = j+1
      t11 = t10-1
      t12 = 4*t11
      t13 = [A+t12]

      t14 = j-1
      t15 = 4*t14
      [A+t15] = t13
      t16 = j+1
      t17 = t16-1
      t18 = 4*t17
      [A+t18] =
temp
S3:   j = j+1
      goto S4
S2:   i = i-1
      goto S5
Exit:
```

```
t12 = 4*j
```

```
t18 = 4*j
```

```
t6 = 4*j
```

Optimizations II - CSE

```
      i = n-1
S5:   if i < 1 goto Exit
      j = 1
S4:   if j > i goto S2
      t1 = j-1
      t2 = 4*t1
      t3 = [A+t2]
      t6 = 4*j
      t7 = [A+t6]
      if t3 <= t7 goto S3
      t8 = j-1
      t9 = 4*t8
      temp = [A+t9]
```

```
      t12 = 4*j
      t13 = [A+t12]
      t14 = j-1
      t15 = 4*t14
      [A+t15] = t13
      t18 = 4*j
      [A+t18] = temp
S3:   j = j+1
      goto S4
S2:   i = i-1
      goto S5
Exit:
```

```
t14 = t1
```

```
t8 = t1
```

Optimizations II - CSE

```
      i = n-1
S5:   if i < 1 goto Exit
      j = 1
S4:   if j > i goto S2
      t1 = j-1
      t2 = 4*t1
      t3 = [A+t2]
      t6 = 4*j
      t7 = [A+t6]
      if t3 <= t7 goto S3
      t8 = t1
      t9 = 4*t8
      temp = [A+t9]
```

```
      t12 = 4*j
      t13 = [A+t12]
      t14 = t1
      t15 = 4*t14
      [A+t15] = t13
      t18 = 4*j
      [A+t18] = temp
S3:   j = j+1
      goto S4
S2:   i = i-1
      goto S5
Exit:
```

```
t12 = t6
```

```
t18 = t6
```

Optimizations III - Copy Propagation

```
t13 = [A+t6]
```

```
      i = n-1
S5:   if i < 1 goto Exit
      j = 1
S4:   if j > i goto S2
      t1 = j-1
      t2 = 4*t1
      t3 = [A+t2]
      t6 = 4*j
      t7 = [A+t6]
      if t3 <= t7 goto S3
      t8 = t1
      t9 = 4*t8
      temp = [A+t9]
```

```
      t12 = t6
      t13 = [A+t12]
      t14 = t1
      t15 = 4*t14
      [A+t15] = t13
      t18 = t6
      [A+t18] = temp
S3:   j = j+1
      goto S4
S2:   i = i-1
      goto S5
Exit
```

```
t15 = 4*t1
```

```
[A+t6] = temp
```

```
t9 = 4*t1
```

Optimizations IV - CSE (2)

```
    i = n-1
S5: if i < 1 goto Exit
    j = 1
S4: if j > i goto S2
    t1 = j-1
    t2 = 4*t1
    t3 = [A+t2]
    t6 = 4*j
    t7 = [A+t6]
    if t3 <= t7 goto S3
    t9 = 4*t1
    temp = [A+t9]
    t13 = [A+t6]
    t15 = 4*t1
    [A+t15] = t13
    [A+t6] = temp
S3: j = j+1
    goto S4
S2: i = i-1
    goto S5
Exit:
```

```
t13 = t7
t15 = t2
t9 = t2
```

Optimizations V - Copy Propagation (2)

```
    i = n-1
S5: if i < 1 goto Exit
    j = 1
S4: if j > i goto S2
    t1 = j-1
    t2 = 4*t1
    t3 = [A+t2]
    t6 = 4*j
    t7 = [A+t6]
    if t3 <= t7 goto S3
    t9 = t2
    temp = [A+t9]
    t13 = t7
    t15 = t2
    [A+t15] = t13
    [A+t6] = temp
S3: j = j+1
    goto S4
S2: i = i-1
    goto S5
Exit:
```

```
[A+t2] = t7
temp = [A+t2]
```

Optimization VI - CSE (3)

```
    i = n-1
S5: if i < 1 goto Exit
    j = 1
S4: if j > i goto S2
    t1 = j-1
    t2 = 4*t1
    t3 = [A+t2]
    t6 = 4*j
    t7 = [A+t6]
    if t3 <= t7 goto S3
    temp = [A+t2]
    [A+t2] = t7
    [A+t6] = temp
S3: j = j+1
    goto S4
S2: i = i-1
    goto S5
Exit:
```

```
temp = t3
```

Optimization VII - Copy Propagation (3)

```
    i = n-1
S5: if i < 1 goto Exit
    j = 1
S4: if j > i goto S2
    t1 = j-1
    t2 = 4*t1
    t3 = [A+t2]
    t6 = 4*j
    t7 = [A+t6]
    if t3 <= t7 goto S3
    temp = [t3]
    [A+t2] = t7
    [A+t6] = temp
S3: j = j+1
    goto S4
S2: i = i-1
    goto S5
Exit:
```

```
[A+t6] = [t3]
```

Optimizations VIII - IVE & Strength Reduction

```
     i = n-1
S5:  if i < 1 goto Exit
     j = 1
S4:  if j > i goto S2
     t1 = j-1
     t2 = 4*t1
     t3 = [A+t2]
     t6 = 4*j
     t7 = [A+t6]
     if t3 <= t7 goto S3
     [A+t2] = t7
     [A+t6] = t3
S3:  j = j+1
     goto S4
S2:  i = i-1
     goto S5
Exit:
```

Optimizations VIII - IVE & Strength Reduction

```
     i = n-1
S5:  if i < 1 goto Exit
     j = 1
S4:  if j > i goto S2
     t1 = j-1
     t2 = 4*t1
     t3 = [A+t2]
     t6 = 4*j
     t7 = [A+t6]
     if t3 <= t7 goto S3
     [A+t2] = t7
     [A+t6] = t3
S3:  j = j+1
     goto S4
S2:  i = i-1
     goto S5
Exit:
```

Loop Invariant Code Motion…

```
     i = n-1
S5:  if i < 1 goto Exit
     t2 = 0
     t6 = 4
S4:  t19 = 4*i
     if t6 > t19 goto S2
     t3 = [A+t2]

     t7 = [A+t6]
     if t3 <= t7 goto S3
     [A+t2] = t7
     [A+t6] = t3
S3:  t2 = t2+4
     t6 = t6+4
     goto S4
S2:  i = i-1
     goto S5
Exit:
```

Done?

     i = n-1   t19 = i*4
S5: if ~~i < 1~~ goto Exit   t19 < 4
     ~~t2 = 0~~
     t6 = 4
S4: ~~t19 = 4*i~~
     if t6 > t19 goto S2
[A-4+t6]
     t3 = [A+t2]
     t7 = [A+t6]
     if t3 <= t7 goto S3
     [A+t2] = t7
     [A+t6] = t3
S3: ~~t2 = t2+4~~
     t6 = t6+4
     goto S4
S2: ~~i = i-1~~   t19 = t19-4
     goto S5
Exit:

Done?

```
     i = n-1
     t19 = i*4
S5:  if t19 < 4 goto Exit
     t6 = 4
S4:  if t6 > t19 goto S2
     t3 = [A+t6-4]
     t7 = [A+t6]
     if t3 <= t7 goto S3
     [A+t6-4] = t7
     [A+t6] = t3
S3:  t6 = t6+4
     goto S4
S2:  t19 = t19 - 4
     goto S5
Exit:
```

Eliminate mult,
Use double load (if aligned?)
Unroll?
Eliminate jmp
…

Done For Now.

```
    i = n-1
    t19 = i<<2
    if t19 < 4 goto Exit
S5: t6 = 4
    if t6 > t19 goto S2
S4: t3 = [A+t6-4]
    t7 = [A+t6]
    if t3 <= t7 goto S3
    [A+t6-4] = t7
    [A+t6] = t3
S3: t6 = t6+4
    if t6 <= t19 goto s4
S2: t19 = t19 - 4
    if t19 >= 4 goto s5
Exit:
```

Inner loop: 7 instructions
4 mem ops
2 branches
1 addition

Original inner loop: 25 instructi
6 mem ops
3 branches
10 addition
6 multiplication

Course Schedule

- www.cs.cmu.edu/afs/cs/academic/class/15745-s09/www/
- The Web site is a vital resource
- (And, of course us too.)

Course Staff

- Seth Goldstein www…./~seth
- Jim Cipar jcipar@cs.cmu.edu
- Marilyn Walgora
  mwalgora@cs.cmu.edu

First Assignment

- Install llvm on your favorite machine
- Get familiar with llvm tools, IR, structure
- Lots of docs at www.llvm.org
- First part of assignment 1 will be posted later today.