Lecture 1: Introduction
Staff

• Lecturer
  – Prof. Michael Carbin  mcarbin@mit.edu  253-5881  32-G782
  – Prof. Martin Rinard  rinard@mit.edu  258-6922  32-G828

• Rooms
  – MWF – 3-370
  – TH – 4-149

• Course Secretary
  – Cree Bruins  cbruins@csail.mit.edu  32-G784a

• Teaching Assistants
  – David Vargas  dvargas@mit.edu

• Website
  – http://6.035.scripts.mit.edu
Reference Textbooks

- **Modern Compiler Implementation in Java (Tiger book)**
  A.W. Appel
  Cambridge University Press, 1998
  ISBN 0-52158-388-8
  A textbook tutorial on compiler implementation, including techniques for many language features.

- **Advanced Compiler Design and Implementation (Whale book)**
  Steven Muchnick
  Morgan Kaufman Publishers, 1997
  ISBN 1-55860-320-4
  Essentially a recipe book of optimizations; very complete and suited for industrial practitioners and researchers.

- **Compilers: Principles, Techniques and Tools (Dragon book)**
  Aho, Lam, Sethi and Ullman
  Addison-Wesley, 2006
  ISBN 0321486811
  The classic compilers textbook, although its front-end emphasis reflects its age. New edition has more optimization material.

- **Engineering a Compiler (Ark book)**
  Keith D. Cooper, Linda Torczon
  Morgan Kaufman Publishers, 2003
  ISBN 1-55860-698-X
  A modern classroom textbook, with increased emphasis on the back-end and implementation techniques.

- **Optimizing Compilers for Modern Architectures**
  Randy Allen and Ken Kennedy
  Morgan Kaufman Publishers, 2001
  ISBN 1-55860-286-0
  A modern textbook that focuses on optimizations including parallelization and memory hierarchy optimization.
The Project: The Five Segments

1. Lexical and Syntax Analysis
2. Semantic Analysis
3. Code Generation
4. Dataflow Analysis
5. Optimizations
Each Segment...

- Segment Start
  - Project Description
- Lectures
  - 2 to 5 lectures
- Project Time
  - (Design Document)
  - (Project Checkpoint)
- Project Due
Project Groups

- 1st project is an individual project
- Projects 2 to 5 are group projects
- Each group consists of 3 to 4 students
- Projects are designed to produce a compiler by the end of class

Grading
- All group members (mostly) get the same grade
- Scanner/parser ungraded
  (you can use this to evaluate potential group members)
- Semantic Checker/Code Generator graded together
- Dataflow Analyzer/Optimizer graded together
- 5 turnins total, 2 turnins are graded
Quizzes

• Two Quizzes

• In-Class Quiz
  – 50 Minutes (be on time!)
  – Open book, open notes
Mini Quizzes

- You already got one
- Given at the beginning of the class
- Collected at the end
- Collaboration is OK

- This is in lieu of time consuming problem sets
Grading Breakdown

- Project = 70% of grade
  - 25% Semantic Checker/Code Generator
  - 45% Dataflow Analyzer/Optimizer
- Quizzes – 24%, 12% each
- Miniquizzes/class participation – 6%
- Relectures
More Course Stuff

• Blank page project – all the rope you what!
• Challenging project
• You are on your own!
• Collaboration policy
  – Talk all you want
  – Write all of your code yourself
• Accepted Languages
  – Java
  – Scala (people have done well with this language)
  – Haskell
Why Study Compilers?

• Compilers enable programming at a high level language instead of machine instructions.
  – Malleability, Portability, Modularity, Simplicity, Programmer Productivity
  – Also Efficiency and Performance
• Indispensable programmer productivity tool
• One of most complex software systems to build
Compilers Construction touches many topics in Computer Science

- **Theory**
  - Finite State Automata, Grammars and Parsing, data-flow
- **Algorithms**
  - Graph manipulation, dynamic programming
- **Data structures**
  - Symbol tables, abstract syntax trees
- **Systems**
  - Allocation and naming, multi-pass systems, compiler construction
- **Computer Architecture**
  - Memory hierarchy, instruction selection, interlocks and latencies, parallelism
- **Security**
  - Detection of and Protection against vulnerabilities
- **Software Engineering**
  - Software development environments, debugging
- **Artificial Intelligence**
  - Heuristic based search for best optimizations
What a Compiler Does

- Input: High-level programming language
- Output: Low-level assembly instructions

- Compiler does the translation:
  - Read and understand the program
  - Precisely determine what actions it requires
  - Figure-out how to faithfully carry out those actions
  - Instruct the computer to carry out those actions
Input to the Compiler

- Standard imperative language (Java, C, C++)
  - State
    - Variables,
    - Structures,
    - Arrays
  - Computation
    - Expressions (arithmetic, logical, etc.)
    - Assignment statements
    - Control flow (conditionals, loops)
    - Procedures
Output of the Compiler

• State
  – Registers
  – Memory with Flat Address Space

• Machine code – load/store architecture
  – Load, store instructions
  – Arithmetic, logical operations on registers
  – Branch instructions
Example (input program)

```c
int sumcalc(int a, int b, int N)
{
    int i, x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
        x = x + (4*a/b)*i + (i+1)*(i+1);
        x = x + b*y;
    }
    return x;
}
```
Example (Output assembly code)

sumcalc:
    pushq  %rbp
    movq  %rsp, %rbp
    movl  %edi, -4(%rbp)
    movl  %esi, -8(%rbp)
    movl  %edx, -12(%rbp)
    mov $0, -20(%rbp)
    mov $0, -24(%rbp)
    mov $0, -16(%rbp)
    .L2:
        movl  -16(%rbp), %eax
        cmpl  -12(%rbp), %eax
        jg .L3
        movl  -4(%rbp), %eax
        leal  0(%rax,4), %edx
        leaq  -8(%rbp), %rax
        movq  %rax, -40(%rbp)
        movl  %edx, %eax
        movq  -40(%rbp), %rcx
        cltd
        idivl  (%rcx)
        movl  %eax, -28(%rbp)
        movl  -28(%rbp), %edx
        imull  -16(%rbp), %edx
        movl  -16(%rbp), %eax
        incl  %eax
        imull  %eax, %eax
        addl  %eax, %edx
        leaq  -20(%rbp), %rax
        addl  %edx, (%rax)
        movl  -8(%rbp), %eax
        movl  %eax, %edx
        imull  -24(%rbp), %edx
        leaq  -20(%rbp), %rax
        addl  %edx, (%rax)
        leaq  -16(%rbp), %rax
        incl  (%rax)
        jmp .L2
    .L3:
        movl  -20(%rbp), %eax
    leave
    ret
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
        x = x + (4*a/b)*i + (i+1)*(i+1);
        x = x + b*y;
    }
    return x;
}
pushq %rbp
movq %rsp, %rbp
movl %edi, -4(%rbp)
movl %esi, -8(%rbp)
movl %edx, -12(%rbp)
movl $0, -20(%rbp)
movl $0, -24(%rbp)
movl $0, -16(%rbp)
.L2:
movl -16(%rbp), %eax
cmpl -12(%rbp), %eax
jg .L3
movl -4(%rbp), %eax
leal 0(,%rax,4), %edx
leaq -8(%rbp), %rax
movq %rax, -40(%rbp)
movl %edx, %eax
movq -40(%rbp), %rcx
cld
idivl (%rcx)
movl %eax, -28(%rbp)
movl -28(%rbp), %edx
imull -16(%rbp), %edx
movl -16(%rbp), %eax
incl %eax
imull %eax, %eax
addl %eax, %edx
leaq -20(%rbp), %rax
addl %edx, (%rax)
movl -8(%rbp), %eax
movl %eax, %edx
imull -24(%rbp), %edx
leaq -20(%rbp), %rax
addl %edx, (%rax)
leaq -16(%rbp), %rax
incl (%rax)
jmp .L2
.L3:
movl -20(%rbp), %eax
leave
ret
int sumcalc(int a, int b, int N) {
    int i, x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
        x = x + (4*a/b)*i + (i+1)*(i+1);
        x = x + b*y;
    }
    return x;
}
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
    x = x + b*y;
}
return x;
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
    x = x + b*y;
}
return x;
Constant Propagation

```c
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
    x = x + b*0;
}
return x;
```
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
    x = x + b*0;
}
return x;
int i, x, y;

x = 0;
y = 0;

for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
    x = x + b*0;
}

return x;
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
    x = x;
}
return x;
Copy Propagation

```c
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
    x = x;
}
return x;
```
Copy Propagation

```c
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
    x = x;
}
return x;
```
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
}
return x;
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
}
return x;
Common Subexpression Elimination

```c
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
}
return x;
```
int i, x, y, t;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + (4*a/b)*i + t*t;
}
return x;
int i, x, y, t;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + (4*a/b)*i + t*t;
}
return x;
int i, x, y, t;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + (4*a/b)*i + t*t;
}
return x;
int i, x, t;
x = 0;

for (i = 0; i <= N; i++) {
    t = i+1;
    x = x + (4*a/b)*i + t*t;
}
return x;
Loop Invariant Removal

```c
int i, x, t;
x = 0;

for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + (4*a/b)*i + t*t;
}
return x;
```
Loop Invariant Removal

```c
int i, x, t;
x = 0;

for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + (4*a/b)*i + t*t;
}
return x;
```
int i, x, t, u;
x = 0;
u = (4*a/b);
for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + u*i + t*t;
}
return x;
Strength Reduction

```c
int i, x, t, u;
x = 0;
u = (4*a/b);

for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + u*i + t*t;
}

return x;
```
int i, x, t, u;
x = 0;
u = (4*a/b);

for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + u*i + t*t;
}

return x;
Strength Reduction

```c
int i, x, t, u, v;
x = 0;
u = ((a<<2)/b);
v = 0;
for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + v + t*t;
    v = v + u;
}
return x;
```
Register Allocation

Local variable X
Local variable Y
Local variable Z

fp
Register Allocation

\[\begin{align*}
&\text{Local variable } \mathbf{X} \\
&\text{Local variable } \mathbf{Y} \\
&\text{Local variable } \mathbf{I}
\end{align*}\]

\[
\begin{align*}
\$r8d &= X \\
\$r9d &= t \\
\$r10d &= u \\
\$ebx &= v \\
\$ecx &= i
\end{align*}
\]
int sumcalc(int a, int b, int N)
{
    int i, x, t, u, v;
    x = 0;
    u = ((a<<2)/b);
    v = 0;
    for(i = 0; i <= N; i++) {
        t = i+1;
        x = x + v + t*t;
        v = v + u;
    }
    return x;
}

Unoptimized Code

```asm
pushq   %rbp
movq   %rsp, %rbp
movl   %edi, -4(%rbp)
movl   %esi, -8(%rbp)
movl   %edx, -12(%rbp)
movl   $0, -20(%rbp)
lea    -24(%rbp)
lea    -16(%rbp)
lea    -16(%rbp), %eax
lea    -12(%rbp), %eax
lea    -12(%rbp), %eax
lea    0(%rax,4), %edx
lea    -8(%rbp), %rax
movq   %rax, -40(%rbp)
movl   %edx, %eax
movq   -40(%rbp), %rcx
cltd
idivl  (%rcx)
movl   %eax, -28(%rbp)
movl   -28(%rbp), %edx
imull  -16(%rbp), %edx
movl   -16(%rbp), %eax
incl   %eax
imull  %eax, %edx
addl   %eax, %edx
lea    -20(%rbp), %rax
addl   %edx, (%rax)
movl   -8(%rbp), %eax
movl   %eax, %edx
imull  -24(%rbp), %edx
lea    -20(%rbp), %rax
addl   %edx, (%rax)
lea    -16(%rbp), %rax
incl   (%rax)
jmp    .L2
.L3:   movl   -20(%rbp), %eax
leave
ret
```

Optimized Code

```asm
xorl   %r8d, %r8d
xorl   %ecx, %ecx
movl   %edx, %r9d
cmpl   %edx, %r8d
jg     .L7
sall   $2, %edi
.L5:   movl   %edi, %eax
cld
idivl  %esi
lea    1(%rcx), %edx
movl   %eax, %r10d
imull  %ecx, %r10d
movl   %edx, %ecx
imull  %edx, %ecx
lea    (%r10,%rcx), %eax
movl   %edx, %ecx
addl   %eax, %r8d
cmpl   %r9d, %edx
jle    .L5
.L7:   movl   %r8d, %eax
ret
```

Inner Loop:

- 10*mov + 5*lea + 5*add/inc + 4*div/mul + 5*cmp/br/jmp = 29 instructions
- Execution time = 43 sec

4*mov + 2*lea + 1*add/inc + 3*div/mul + 2*cmp/br/jmp = 12 instructions
- Execution time = 17 sec
Compilers Optimize Programs for...

- Performance/Speed
- Code Size
- Power Consumption
- Fast/Efficient Compilation
- Security/Reliability
- Debugging