# CS 701

## Charles N. Fischer

## Fall 2005

**http://www.cs.wisc.edu/~fischer/cs701.html**

# Class Meets

Tuesdays & Thursdays, 11:00 — 12:15

3418 Engineering Hall

# Instructor

Charles N. Fischer

6367 Computer Sciences

Telephone: 262-6635

E-mail: fischer@cs.wisc.edu

Office Hours:

10:30 - Noon, Mondays & Wednesdays, or by appointment

# Teaching Assistant

Anne Mulhern

3361 Computer Sciences

Telephone: 446-3841

E-mail: mulhern@cs.wisc.edu

Office Hours:

1:00 - 3:00
Thursdays or by appointment

# Key Dates

- September 27: Project 1 due
- October 25: Project 2 due (tentative)
- November 1: Midterm (tentative)
- November 29: Project 3 due (tentative)
- December 15: Project 4 due
- December ??: Final Exam, date to be determined

# Class Text

There is no required text.

Handouts and Web-based reading will be used.

Suggested reference:

*Advanced Compiler Design & Implementation*,
by Steven S. Muchnick,
published by Morgan Kaufman.

# Instructional Computers

Departmental SPARC Processors (n01.cs.wisc—n16.cs.wisc) are assigned to this class.

Your own workstation probably isn't SPARC-based, so you will need to log onto a machine that uses a SPARC processor to do SPARC-specific assignments.

# CS701 Projects

1. SPARC Code Optimization
2. Global Register Allocation
   (using Graph Coloring)
3. Global Code Optimizations
4. Individual Research Topics

# Academic Misconduct Policy

- You must do your assignments—no copying or sharing of solutions.
- You may discuss general concepts and Ideas.
- All cases of Misconduct *must* be reported.
- Penalties may be severe.

# Reading Assignment

- Read Chapters 0-6 and Appendices G&H of the SPARC Architecture Manual. Also skim Appendix A.
- Read section 15.2 of Chapter 15.
- Read Assignment #1.

# Overview of Course Topics

1. Register Allocation

## Local Allocation

Avoid unnecessary loads and stores within a *basic block*. Remember and reuse register contents.
Consider effects of *aliasing*.

## Global Allocation

Allocate registers within a single subprogram. Choose "most profitable" values. Map several values to the *same* register.

## Interprocedural Allocation

Avoid saves and restores across calls.
Share globals in registers.

# 2. Code Scheduling

We can reorder code to reduce latencies and to maximize ILP (*Instruction Level Parallelism*). We must respect *data dependencies* and *control dependencies*.

```
ld  [a],%r1          ld [a],%r1
add %r1,1,%r2        mov 3,%r3
mov 3,%r3            add %r1,1,%r2
```

(before) (after)

# 3. Automatic Instruction Selection

How do we map an IR (*Intermediate Representation*) into Machine Instructions?

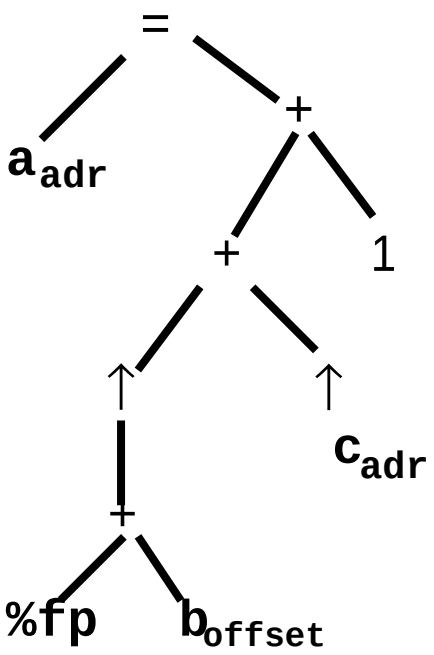
Can we guarantee the *best* instruction sequence?

Idea—Match instruction patterns (represented as trees) against an IR that is a low-level tree. Each match is a generated instruction; the best overall match is the best instruction sequence.

Example:

**a=b+c+1;**

In IR tree form:




Generated code:

```
ld  [%fp+b_offset],%r1
ld  [c_adr],%r2
add %r1,%r2,%r3
add %r3,1,%r4
st  %r4,[a_adr]
```

Why use four *different* registers?

# 4. Peephole Optimization

Inspect generated code sequences and replace pairs/triples/tuples with better alternatives.

```
ld  [a],%r1          ld  [a],%r1
mov const,%r2        add %r1,const,%r3
add %r1,%r2,%r3
```

(before) (after)

```
mov 0,%r1            OP  %g0,%r2,%r3
OP  %r1,%r2,%r3
```

(before) (after)

But why not just generate the better code sequence to begin with?

## 5. Cache Improvements

We want to access data & instructions from the L1 cache whenever possible; misses into the L2 cache (or memory) are *expensive*!

We will layout data and program code with consideration of cache sizes and access properties.

## 6. Local & Global Optimizations

Identify unneeded or redundant code.

Decide where to place code.

Worry about debugging issues (how reliable are current values and source line numbers after optimization?)

# 7. Program representations

- Control Flow Graphs
- Program Dependency Graphs
- Static Single Assignment Form (SSA)

  Each program variable is assigned to in only *one* place.

  After an assignment $x_i = y_j$, the relation $x_i = y_j$ *always* holds.

  Example:

```
if (a)                 if (a)
     x = 1                  x1 =1
else x = 2;            else x2 =2;
print(x)               x3 = φ(x1,x2)
                       print(x3)
```

# 8. Data Flow Analysis

Determine invariant properties of subprograms; analysis can be extended to entire programs.

Model abstract execution.

Prove correctness and efficiency properties of analysis algorithms.

# Review of Compiler Optimizations

1. Redundant Expression Elimination (Common Subexpression Elimination)

   Use an address or value that has been previously computed. Consider control and data dependencies.

2. Partially Redundant Expression (PRE) Elimination

   A variant of Redundant Expression Elimination. If a value or address is redundant along *some* execution paths, add computations to other paths to create a fully redundant expression (which is then removed).

   Example:

```
if (i > j)
   a[i] = a[j];
a[i] = a[i] * 2;
```

## 3. Constant Propagation

If a variable is known to contain a particular constant value at a particular point in the program, replace references to the variable at that point with that constant value.

## 4. Copy Propagation

After the assignment of one variable to another, a reference to one variable may be replaced with the value of the other variable (until one or the other of the variables is reassigned).

(This may also “set up” dead code elimination. Why?)

## 5. Constant Folding

An expression involving constant (literal) values may be evaluated and simplified to a constant result value. Particularly useful when constant propagation is performed.

## 6. Dead Code Elimination

Expressions or statements whose values or effects are unused may be eliminated.

## 7. Loop Invariant Code Motion

An expression that is *invariant* in a loop may be moved to the loop's header, evaluated once, and reused within the loop.

*Safety* and *profitability* issues may be involved.

## 8. Scalarization (Scalar Replacement)

A field of a structure or an element of an array that is repeatedly read or written may be copied to a local variable, accessed using the local, and later (if necessary) copied back.

This optimization allows the local variable (and in effect the field or array component) to be allocated to a register.