Intermediate Code & Local Optimizations

Lecture 14

Lecture Outline

- Intermediate code
- Local optimizations
- Next time: global optimizations

Code Generation Summary

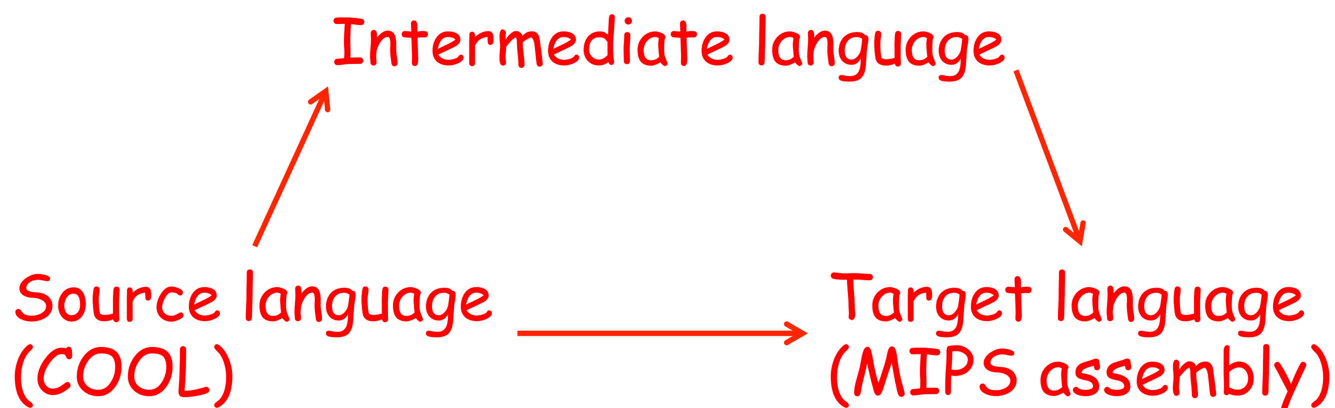
- We have discussed
  - Runtime organization
  - Simple stack machine code generation
  - Improvements to stack machine code generation
- Our compiler maps AST to assembly language
  - And does not perform optimizations

Optimization

- Optimization is our last compiler phase
- Most complexity in modern compilers is in the optimizer
  - Also by far the largest phase
- First, we need to discuss intermediate languages

Intermediate Language

- A language between the source language and the target language
- Provides an intermediate level of abstraction
  - More details than the source
  - Fewer details than the target



We've gone directly from COOL to MIPS assembly, but in many cases, compilers first go through an intermediate language. This leads to an obvious question…

Why Do This?

- That is, why make life more difficult by doing something in two steps that you could do in a single step?
- Because it turns out that in many cases, the intermediate level of abstraction is really quite useful.
  - Provides enough detail to allow for things like optimization
  - But not so low level that optimizations need to be reimplemented when retargeting compiler

E.g., What kind of detail do you need to express optimizations you might want to do with register? Ans: More than cool provides.

Why Do This?

- That is, why make life more difficult by doing something in two steps that you could do in a single step?
- Because it turns out that in many cases, the intermediate level of abstraction is really quite useful.
  - Provides enough detail to allow for things like optimization
  - But not so low level that optimizations need to be reimplemented when retargeting compiler

But, you don't want so much detail that you're getting down to the level of a particular instruction set on a particular machine.

Why Intermediate Languages?

- When should we perform optimizations?
  - On AST
    - Pro: Machine independent
    - Con: Too high level
  - On assembly language
    - Pro: Exposes optimization opportunities
    - Con: Machine dependent
    - Con: Must reimplement optimizations when retargetting
  - On an intermediate language
    - Pro: Machine independent
    - Pro: Exposes optimization opportunities

In Practice

- Experience has shown that having an intermediate language is a good idea
  - Almost all real compilers have an intermediate language
  - Some compilers translate through an entire series of intermediate languages between source and target
- We'll consider only one: a sort of high-level assembly language

Intermediate Languages

- Intermediate language = high-level assembly
  - Uses register names, but has an unlimited number of them
  - Uses control structures like assembly language
    - Explicit jumps and labels
  - Uses opcodes but some are higher level
    - E.g., push translates to several assembly instructions
    - Most opcodes correspond directly to MIPS assembly opcodes

Three-Address Intermediate Code

- Each instruction is of one of these forms

  x := y op z — binary op

  x := op y — unary op

  - y and z are registers or constants (immediate values)
  - Common form of intermediate code
- The expression x + y * z is translated

  t_1 := y * z

  t_2 := x + t_1

  note each instruction performs only one operation

  - Each subexpression has a "name"

Called "three-address code" since every instruction has at most three addresses in it

called three-address since each instruction has at most three addresses in it (2 ops and 1 destination)

Three-Address Intermediate Code

- Each instruction is of one of these forms

  $x := y \text{ op } z$ binary operation

  $x := \text{op } y$ unary operation

  - y and z are registers or constants (immediate values)
  - Common form of intermediate code
- The expression $x + y * z$ is translated

  $t_1 := y * z$

  $t_2 := x + t_1$

  - Each subexpression has a "name"
    - This is a result of only one operation per instruction

Generating Intermediate Code

- Generation of three-address code is similar to assembly code generation

- Main difference: use any number of intermediate language (IL) registers to hold intermediate results

Generating Intermediate Code (Cont.)

- igen(e, t) function generates code to compute the value of e and place in register t
- Example:

  igen(e_1 + e_2, t) =

  igen(e_1, t_1) (t_1 is a fresh register)

  igen(e_2, t_2) (t_2 is a fresh register)

  t := t_1 + t_2 ← note this is a three-address instruction

- Unlimited number of registers ⇒ simple code generation
  - Even simpler than generating for stack machine (no need to move SP on push, etc)

Intermediate Code Notes

- You should be able to use intermediate code
  - At the level discussed in lectures
  - Future lectures will use it quite a bit to express certain kinds of optimizations
  - Should also be able to write simple IC programs and write algorithms that work on IC
- You are not expected to know how to generate intermediate code
  - Because we won't discuss it
  - But really just a variation on code generation notions we've already discussed in detail.
    - No new ideas

Optimization

- Optimization is our last compiler phase
  - But recall from first day that it occurs before code generation
    - We want to improve the program before committing it to machine code
- Most complexity in modern compilers is in the optimizer
  - Also, by far, the most code
  - By far the most complex part of the compiler

Optimization

- When should we perform optimization?
  - On AST?
    - Pro: Machine independent
    - Con: Too high level – lacks details we need to express optimizations
  - On assembly language?
    - Pro: Exposes optimization opportunity
    - Con: Machine dependent
    - Con: Must reimplement optimization when retargeting
  - On an intermediate language?
    - Pro: Machine independent
    - Pro: Exposes optimization opportunities

An Intermediate Language

$P \rightarrow S\ P \mid \varepsilon$
$S \rightarrow$ id := id op id
| id := op id
| id := id
| push id
| id := pop
| if id relop id goto L
| L:
| jump L

- id's are register names
- Constants (immediates) can replace id's
- Typical operators: +, -, *

because optimizations typically work on groups of statements

Definition. Basic Blocks

- A basic block is a maximal sequence of instructions with:
  - no labels (except at the first instruction), and
  - no jumps (except possibly the last instruction)

- Idea:
  - Cannot jump into a basic block (except at beginning)
  - Cannot jump out of a basic block (except at end)
  - A basic block is a single-entry, single-exit, straight-line code segment
    - So flow of control within a basic block is completely predictable

because optimizations typically work on groups of statements

Definition. Basic Blocks

- A basic block is a maximal sequence of instructions with:
  - no labels (except at the first instruction), and
  - no jumps (except in the last instruction)

- Idea:
  - Cannot jump into a basic block (except at beginning)
  - Cannot jump out of a basic block (except at end)
  - A basic block is a single-entry, single-exit, straight-line code segment
    - Also, can't jump into middle of a basic block!

Basic Block Example

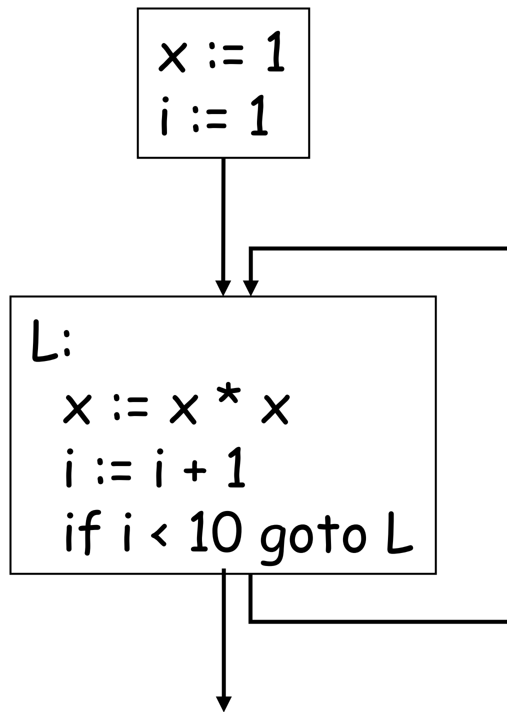
- Consider the basic block
  1. L:
  2.   t := 2 * x
  3.   w := t + x
  4.   if w > 0 goto L'

- (3) executes only after (2)
  - We can change (3) to w := 3 * x
  - Can we eliminate (2) as well?
    - Depends on whether t has any other uses elsewhere in the program (can't know just from this block)

Definition. Control-Flow Graphs

- A control-flow graph (CFG) is a directed graph with
  - Basic blocks as nodes
  - An edge from block A to block B if the execution can pass from the last instruction in A to the first instruction in B
    - E.g., the last instruction in A is jump L_B
    - E.g., execution can fall-through from block A to block B
- Since control flow within a basic block is simple, CFG summarizes interesting decision points within procedure or piece of code
  - Note use of acronym "CFG" is, somewhat ironically, context dependent

Example of Control-Flow Graphs



- The body of a method (or procedure) can always be represented as a control-flow graph
- There is one initial node (i.e., a start node)
- All "return" nodes are terminal
  - No edges out of these blocks

Optimization Overview

- Optimization seeks to improve a program's resource utilization
  - Execution time (most often)
    - What we focus on in this course
  - Code size
  - Network messages sent, etc.
  - Actually, for any resource you can imagine, there is likely a compiler out there that spends some effort optimizing for it
    - Memory usage
    - Disk accesses
    - Power

Optimization Overview

- Optimization seeks to improve a program's resource utilization
  - Execution time (most often)
    - What we focus on in this course
  - Code size
  - Network messages sent, etc.

- Optimization should not alter what the program computes
  - The answer must still be the same

A Classification of Optimizations

- For languages like C and Cool there are three granularities of optimizations
  1. Local optimizations
     - Apply to a basic block in isolation
  2. Global optimizations
     - Apply to a control-flow graph (method body) in isolation
     - Misnamed: global to the CFG (i.e. across the entire function)
  3. Inter-procedural optimizations
     - Apply across method boundaries

- Most compilers do (1), many do (2), few do (3)

A Classification of Optimizations

- For languages like C and Cool there are three granularities of optimizations
  1. Local optimizations
  2. Global optimizations
  3. Inter-procedural optimizations

- Most compilers do (1), many do (2), few do (3)
- Why?
  - (3) is more difficult to implement
  - Most of optimization payoff comes from (1) and (2)

Cost of Optimizations

- In practice, often a conscious decision is made not to implement the fanciest optimization known (e.g., in the research literature)
  - There are very many known
  - Reality can be frustrating for compiler researchers
    - Whose great ideas are not always implemented
- Why?
  - Boils down to software engineering

Cost of Optimizations

- In practice, a conscious decision is made not to implement the fanciest optimization known
  - There are very many known
  - Reality can be frustrating for compiler researchers
- Why?
  - Some optimizations are hard to implement
  - Some optimizations are costly in compilation time
    - Can take days!
  - Some optimizations have low benefit
  - Many fancy optimizations are all three!
    - Which explains why not implemented!
- Goal: Maximum benefit for minimum cost
  - Cost/benefit ratio

Local Optimizations

- The simplest form of optimizations
- Optimize one basic block
  - No need to analyze the whole procedure body
  - No need to worry about complicated control flow
- Example: algebraic simplification

Algebraic Simplification

- Some statements can be deleted

  x := x + 0

  x := x * 1

- Some statements can be simplified

  x := x * 0 ⇒ x := 0

  y := y ** 2 ⇒ y := y * y

  x := x * 8 ⇒ x := x << 3

  x := x * 15 ⇒ t := x << 4; x := t - x

  (on some machines << is faster than *; but not on all!)

Assume x has type int here

Algebraic Simplification

- Some statements can be deleted

```
x := x + 0
x := x * 1
```

- Some statements can be simplified

```
x := x * 0     ⇒  x := 0
y := y ** 2    ⇒  y := y * y
x := x * 8     ⇒  x := x << 3
x := x * 15    ⇒  t := x << 4; x := t - x
```

Surprisingly, it might be that these two statements take equal time to run. But having the constant assignment allows some other optimizations (note points to `x := 0`)

(on some machines << is faster than *; but not on all!)

Algebraic Simplification

- Some statements can be deleted

  x := x + 0

  x := x * 1

- Some statements can be simplified

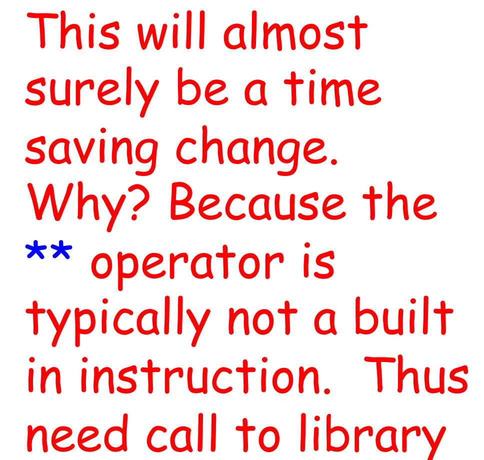
  x := x * 0 $\Rightarrow$ x := 0

  y := y ** 2 $\Rightarrow$ y := y * y

  x := x * 8 $\Rightarrow$ x := x << 3

  x := x * 15 $\Rightarrow$ t := x << 4; x := t - x

  (on some machines << is faster than *; but not on all!)



Algebraic Simplification

- Some statements can be deleted

  x := x + 0

  x := x * 1

- Some statements can be simplified

  x := x * 0 ⇒ x := 0

  y := y ** 2 ⇒ y := y * y

  x := x * 8 ⇒ x := x << 3

  x := x * 15 ⇒ t := x << 4; x := t - x

  (on some machines << is faster than *; but not on all!

  In fact, on most modern machines, no difference! On historical machines these were significant optimizations)

Algebraic Simplification

- Some statements can be deleted

  x := x + 0

  x := x * 1

- Some statements can be simplified

  x := x * 0 $\Rightarrow$ x := 0

  y := y ** 2 $\Rightarrow$ y := y * y

  x := x * 8 $\Rightarrow$ x := x << 3

  x := x * 15 $\Rightarrow$ t := x << 4; x := t - x

  All of these examples of algebraic simplification: exploiting properties of the mathematical operators to replace complex ops with simpler ones

Constant Folding

- Operations on constants can be computed at compile time
  - If there is a statement x := y op z
  - And y and z are constants
  - Then y op z can be computed at compile time

- Example: x := 2 + 2 $\Rightarrow$ x := 4
- Example: if 2 < 0 jump L can be deleted
  - In general, if the predicate consists only of immediate values, the target of the jump can be precomputed and the conditional eliminated

Constant Folding

- Operations on constants can be computed at compile time
  - If there is a statement x := y op z
  - And y and z are constants
  - Then y op z can be computed at compile time

- Example: x := 2 + 2 ⇒ x := 4
- Example: if 2 < 0 jump L can be deleted

constant folding is one of the most common and most important optimizations that compilers perform

Constant Folding

- Constant folding can be dangerous
  - Although this condition is not common, it is instructive
- Cross-compiler: compile on machine X, generate code to run on machine Y
  - Why? Perhaps Y is weak: low power, limited memory, etc.
  - Much code for embedded devices compiled this way

Constant Folding Danger (cont.)

- Assume X and Y are different architectures
- And you want to compute a:=1.5 +3.7, which you fold, on X, to a:=5.2
- But, if computed directly on Y, differences in floating point implementation might lead, in actual execution, to a:=5.19
- Becomes significant because some algorithms depend on floating point numbers being treated consistently
  - E.g., roundoff must be handled same way every time it's required
  - This can change results of the program

Constant Folding Danger (cont.)

- So how do cross-compilers handle this?
  - Represent floating point numbers as strings inside the compiler
  - Do long form addition, multiplication, etc, directly on the strings, keeping full precision inside compiler
  - Then in the running code, they produce the literal full precision floating point number and let Y decide how it wants to handle rounding
  - This is careful way to do cross-compilation if you're worried about this issue

Flow of Control Optimizations

- Eliminate unreachable basic blocks:
  - Code that is unreachable from the initial block
    - E.g., basic blocks that are not the target of any jump or "fall through" from a conditional

- Removing unreachable code makes the program smaller
  - And sometimes also faster
    - Due to memory cache effects (removal of unused code may increase spatial locality of remaining code)

Question:

- Why would unreachable basic blocks occur?
- Well, several ways it can occur
  - So this situation is actually quite common
- Ex: Code that is parameterized and only run in certain situations (e.g., my debug_print() function and the calls to it in Project 1)
- More generally:

```
#define DEBUG 0
if (DEBUG) then {…
```

Question:

- Why would unreachable basic blocks occur?
- Another Ex.: libraries
  - Libraries often contain several (sometimes hundreds of) methods. Your code might only use a few of them. So the remaining methods can be removed from the final executable binary
- Final Ex.: Other optimizations
  - Which can lead to further optimizations
  - Since, as we've discussed, the compiler is automated, this can result in the generation of blocks of code that might never be reachable

Single Assignment Form

- Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment

- Rewrite intermediate code in *single assignment* form (every register is assigned at most once)

```
x := z + y              b := z + y
a := x        ⇒         a := b
x := 2 * x              x := 2 * b
```

(b is a fresh register)

  - More complicated in general, due to loops

Common Subexpression Elimination

- If
  - Basic block is in single assignment form and
  - A definition x := is the first use of x in a block
- Then
  - When two assignments have the same rhs, they compute the same value
- Example:

```
x := y + z                    x := y + z
…              ⇒              …
w := y + z                    w := x
```

(the values of x, y, and z do not change in the … code)

Common Subexpression Elimination

- If
  - Basic block is in single assignment form and
  - A definition x := is the first use of x in a block
- Then
  - When two assignments have the same rhs, they compute the same value
- Example:

```
x := y + z                  x := y + z
…               ⇒           …
w := y + z                  w := x
```

(the values of x, y, and z do not change in the … code)
(because x, y, and z, must already have been defined)

Common Subexpression Elimination

- If
  - Basic block is in single assignment form and
  - A definition x := is the first use of x in a block
- Then
  - When two assignments have the same rhs, they compute the same value
- Example:

```
x := y + z                  x := y + z
…               ⇒           …
w := y + z                  w := x
```

(changing y + z to x saves us from having to recompute the sum)

Common Subexpression Elimination

- If
  - Basic block is in single assignment form and
  - A definition x := is the first use of x in a block
- Then
  - When two assignments have the same rhs, they compute the same value
- Example:

```
x := y + z                    ⇒      x := y + z
…                                    …
w := y + z                           w := x
```

(occurs quite often: an important compiler optimization)

Copy Propagation

- If w := x appears in a block, replace subsequent uses of w with uses of x
  - Assumes single assignment form

- Example:

b := z + y		b := z + y
a := b	⇒	a := b
x := 2 * a		x := 2 * b

- By itself makes no improvement to code. Only useful for enabling other optimizations
  - Constant folding
  - Dead code elimination
    - E.g., above, might be able to remove the line a:=b in right column

Copy Propagation and Constant Folding

- Example:

Before		After
a := 5		a := 5
x := 2 * a	⇒	x := 10
y := x + 6		y := 16
t := x * y		t := x << 4

Copy Propagation and Constant Folding

- Example:

a := 5		a := 5
x := 2 * 5	⇒	x := 10
y := x + 6		y := 16
t := x * y		t := x << 4

Note when constant is being propagated, it's called "constant propagation", rather than copy propagation

Copy Propagation and Constant Folding

- Example:

```
a := 5                   a := 5
x := 10          ⇒       x := 10
y := x + 6               y := 16
t := x * y               t := x << 4
```

Copy Propagation and Constant Folding

- Example:

```
a := 5                    a := 5
x := 10         ⇒         x := 10
y := 10 + 6               y := 16
t := 10 * y               t := x << 4
```

Copy Propagation and Constant Folding

- Example:

a := 5		a := 5
x := 10	⇒	x := 10
y := 16		y := 16
t := 10 * 16		t := x << 4

Copy Propagation and Constant Folding

- Example:

```
a := 5                    a := 5
x := 10          ⇒        x := 10
y := 16                   y := 16
t := 10 * 16              t := x << 4
```

Copy Propagation and Constant Folding

- Example:

a := 5		a := 5
x := 10	$\Rightarrow$	x := 10
y := 16		y := 16
t := 10 * 16		t := 160

Note final constant propagation better than leaving it as x << 4

Copy Propagation and Dead Code Elimination

If

w := rhs appears in a basic block

w does not appear anywhere else in the program

Then

the statement w := rhs is dead and can be eliminated

– Dead = does not contribute to the program's result

Example: (a is not used anywhere else)

x := z + y		b := z + y		b := z + y
a := x	⇒	a := b	⇒	x := 2 * b
x := 2 * a		x := 2 * b		

Applying Local Optimizations

- Each local optimization does little by itself
  - Some we've mentioned don't, by themselves, make the program run faster at all (though they don't make it run slower)
- Typically optimizations interact
  - Performing one optimization enables another
- Optimizing compilers repeat optimizations until no improvement is possible
  - The optimizer can also be stopped at any point to limit compilation time

Applying Local Optimizations

- Optimizing compilers repeat optimizations until no improvement is possible
  - The optimizer can also be stopped at any point to limit compilation time
- Basically, the optimizing compiler has a bag of tricks, which it looks through to see if any trick can be applied to some part of the current version of the code.
- If it finds one, it changes the code, then goes back and repeats the process on the new version of the code.

A Bigger Example

- Initial code:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

An Example

- Algebraic optimization:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

An Example

- Algebraic optimization:

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

An Example

- Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

An Example

- Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

An Example

- Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

An Example

- Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

An Example

- Dead code elimination:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

An Example

- Dead code elimination:

```
a := x * x



f := a + a
g := 6 * f
```

- This is the final form

An Example

- Dead code elimination:

```
a := x * x



f := a + a
g := 6 * f
```

- This is the final form (well, not really)

An Example

- Dead code elimination:

```
a := x * x




f := 2 * a
g := 6 * f
```

An Example

- Dead code elimination:

```
a := x * x




f := 2 * a
g := 6 * f
```

An Example

- Dead code elimination:

```
a := x * x




f := 2 * a
g := 12 * a
```

An Example

- Dead code elimination:

```
a := x * x



g := 12 * a
```

- This is the final form (really)

An Example

- Dead code elimination:

```
a := x * x




g := 12 * a
```

- This is the final form (really)
  - But to be fair, while some current compilers would find these last few steps, most would not

Peephole Optimizations on Assembly Code

- These optimizations work on intermediate code
  - Target independent
  - But they can be applied directly to assembly language also

- Peephole optimization is effective for improving assembly code
  - The "peephole" is a short sequence of (usually contiguous) instructions
    - It's some "window" onto the code
  - The optimizer replaces the sequence with another equivalent one (but faster)

Peephole Optimizations on Assembly Code

- These optimizations work on intermediate code
  - Target independent
  - But they can be applied directly to assembly language also

- Peephole optimization is effective for improving assembly code
  - The "peephole" is a short sequence of (usually contiguous) instructions
    - It's some "window" onto the code
  - The optimizer replaces the sequence with another equivalent one (but faster) (then rinse and repeat)

Peephole Optimizations (Cont.)

- Write peephole optimizations as replacement rules

  $i_1, \ldots, i_n \rightarrow j_1, \ldots, j_m$

  where the rhs is the improved version of the lhs

- Example:

  move \$a \$b, move \$b \$a → move \$a \$b

  - Works if move \$b \$a is not the target of a jump
    - Think about it: after first move, \$a and \$b have the same contents

- Another example

  addiu \$a \$a i, addiu \$a \$a j → addiu \$a \$a i+j

Peephole Optimizations (Cont.)

- Many (but not all) of the basic block optimizations can be cast also as peephole optimizations
  - Example: addiu $a $b 0 → move $a $b
  - Example: move $a $a →
  - These two optimizations together eliminate addiu $a $a 0

- As for local optimizations, peephole optimizations must be applied repeatedly for maximum effect

Local Optimizations: Notes

- Intermediate code is helpful for many optimizations

- Many simple optimizations can still be applied on assembly language

- "Program optimization" is grossly misnamed
  - Code produced by "optimizers" is not optimal in any reasonable sense
    - And if it happened to somehow produce an actual "optimal" version of the program, it would be a complete accident
  - "Program improvement" is a more appropriate term

- Next time: global optimizations