Intermediate Code & Local Optimizations
Lecture Outline

• What is “Intermediate code”?

• Why do we need it?

• How to generate it?

• How to use it?

• Optimizations
  - Local optimizations
Code Generation Summary

• We have so far discussed
  - Runtime organization
  - Simple stack machine code generation
  - Improvements to stack machine code generation

• Our compiler goes directly from the abstract syntax tree (AST) to assembly language...
  - ... and does not perform optimizations

Most real compilers use intermediate languages
Why Intermediate Languages?

**ISSUE**: Reduce code complexity

- **Multiple front-ends**
  - gcc can handle C, C++, Java, Fortran, Ada, ...
  - each front-end translates source to the same generic language (called GENERIC)

- **Multiple back-ends**
  - gcc can generate machine code for various target architectures: x86, x86_64, SPARC, ARM, ...

- **One Icode to bridge them!**
  - Do most optimization on intermediate representation before emitting machine code
ISSUE: When to perform optimizations

- On abstract syntax trees
  - Pro: Machine independent
  - Con: Too high level

- On assembly language
  - Pro: Exposes most optimization opportunities
  - Con: Machine dependent
  - Con: Must re-implement optimizations when re-targeting

- On an intermediate language
  - Pro: Exposes optimization opportunities
  - Pro: Machine independent
Kinds of Intermediate Languages

High-level intermediate representations:
- closer to the source language (structs, arrays)
- easy to generate from the input program
- code optimizations may not be straightforward

Low-level intermediate representations:
- closer to target machine: GCC’s RTL, 3-address code
- easy to generate code from
- generation from input program may require effort

“Mid”-level intermediate representations:
- programming language and target independent
- Java bytecode, Microsoft CIL, LLVM IR, ...
Intermediate Code Languages: Design Issues

• Designing a good ICode language is not trivial
• The set of operators in ICode must be rich enough to allow the implementation of source language operations
• ICode operations that are closely tied to a particular machine or architecture, make retargeting harder
• A small set of operations
  - may lead to long instruction sequences for some source language constructs,
  - but on the other hand makes retargeting easier
Intermediate Languages

• Each compiler uses its own intermediate language

• Nowadays, usually an intermediate language is a high-level assembly language
  - Uses register names, but has an unlimited number
  - Uses control structures like assembly language
  - Uses opcodes but some are higher level
    • E.g., push translates to several assembly instructions
    • Most opcodes correspond directly to assembly opcodes
Architecture of gcc
Three-Address Intermediate Code

• Each instruction is of the form
  \( x := y \text{ op } z \)
  - \( y \) and \( z \) can only be registers or constants
  - Just like assembly

• Common form of intermediate code

• The expression \( x + y \times z \) gets translated as
  \( t_1 := y \times z \)
  \( t_2 := x + t_1 \)
  - temporary names are made up for internal nodes
  - each sub-expression has a “home”
Generating Intermediate Code

- Similar to assembly code generation
- Major difference
  - Use any number of IL registers to hold intermediate results

**Example:** if \((x + 2 > 3 \ast (y - 1) + 42)\) then \(z := 0;\)

\[
\begin{align*}
t_1 & := x + 2 \\
t_2 & := y - 1 \\
t_3 & := 3 \ast t_2 \\
t_4 & := t_3 + 42 \\
if & \ t_1 <= t_4 \ goto \ L \\
z & := 0 \\
L: &
\end{align*}
\]
**Generating Intermediate Code (Cont.)**

\( \text{igen}(e, t) \): a function that generates code to compute the value of \( e \) in register \( t \)

- Example:

  \[
  \text{igen}(e_1 + e_2, t) = \quad \text{(t is a fresh register)}
  \]

  \[
  \begin{align*}
  \text{igen}(e_1, t_1) && (t_1 \text{ is a fresh register}) \\
  \text{igen}(e_2, t_2) && (t_2 \text{ is a fresh register}) \\
  t &:= t_1 + t_2
  \end{align*}
  \]

- Unlimited number of registers

  \( \Rightarrow \) simple code generation
From ICode to Machine Code

This is almost a macro expansion process

<table>
<thead>
<tr>
<th>ICode</th>
<th>MIPS assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := A[i]</td>
<td>load i into r1</td>
</tr>
<tr>
<td></td>
<td>la r2, A</td>
</tr>
<tr>
<td></td>
<td>add r2, r2, r1</td>
</tr>
<tr>
<td></td>
<td>lw r2, (r2)</td>
</tr>
<tr>
<td></td>
<td>sw r2, x</td>
</tr>
<tr>
<td>x := y + z</td>
<td>load y into r1</td>
</tr>
<tr>
<td></td>
<td>load z into r2</td>
</tr>
<tr>
<td></td>
<td>add r3, r1, r2</td>
</tr>
<tr>
<td></td>
<td>sw r3, x</td>
</tr>
<tr>
<td>if x &gt;= y goto L</td>
<td>load x into r1</td>
</tr>
<tr>
<td></td>
<td>load y into r2</td>
</tr>
<tr>
<td></td>
<td>bge r1, r2, L</td>
</tr>
</tbody>
</table>
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)
  - Each instruction in a basic block is executed after all the preceding instructions have been executed
Basic Block Example

Consider the basic block

\[ L:\]
\[ t := 2 \times x \quad (2) \]
\[ w := t + x \quad (3) \]
\[ \text{if } w > 0 \text{ goto } L' \quad (4) \]

- No way for (3) to be executed without (2) having been executed right before
  - We can change (3) to \( w := 3 \times x \)
  - Can we eliminate (2) as well?
Identifying Basic Blocks

• Determine the set of leaders, i.e., the first instruction of each basic block:
  - The first instruction of a function is a leader
  - Any instruction that is a target of a branch is a leader
  - Any instruction immediately following a (conditional or unconditional) branch is a leader

• For each leader, its basic block consists of itself and all instructions up to, but not including, the next leader (or end of function)
A control-flow graph is a directed graph with
- Basic blocks as nodes
- An edge from block A to block B if the execution can flow from the last instruction in A to the first instruction in B
  E.g., the last instruction in A is goto $L_B$
  E.g., the execution can fall-through from block A to block B

Frequently abbreviated as CFGs
Control-Flow Graphs: Example

- The body of a function (or method or procedure) can be represented as a control-flow graph.
- There is one initial node.
- All “return” nodes are terminal.

Example:

```
x := 1
i := 1
```

Loop:
```
x := x * x
i := i + 1
if i < 42 goto L
```
Constructing the Control Flow Graph

• First identify the basic blocks of the function
• There is a directed edge between block $B_1$ to block $B_2$ if
  - there is a (conditional or unconditional) jump from the last instruction of $B_1$ to the first instruction of $B_2$ or
  - $B_2$ immediately follows $B_1$ in the textual order of the program, and $B_1$ does not end in an unconditional jump.
**Optimization Overview**

- Compiler “optimizations” seek to improve a program’s utilization of some resource
  - Execution time (most often)
  - Code size
  - Network messages sent
  - (Battery) power used, etc.

- Optimization should not alter what the program computes
  - The return value must be the same.
  - Any observable behavior must be the same.
    (This typically also includes termination behavior.)
A Classification of Optimizations

For languages like C, 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 (function body) in isolation

(3) **Inter-procedural optimizations**
   - Apply across function/procedure boundaries

Most compilers do (1), many do (2), and very few do (3).

**Note:** there are also **link-time optimizations**
Cost of Optimizations

• In practice, a conscious decision is made not to implement the fanciest optimizations

• Why?
  - Some optimizations are hard to implement
  - Some optimizations are costly in terms of compilation time
  - Some optimizations are hard to get completely right
  - The fancy optimizations are often hard, costly, and difficult to get completely correct

• Goal: maximum improvement with minimum cost
Local Optimizations

- The simplest form of optimizations
- No need to analyze the whole procedure body
  - Just the basic block in question

- Example: algebraic simplification
Algebraic Simplification

• Some statements can be deleted
   \[ x := x + 0 \]
   \[ x := x * 1 \]

• Some statements can be simplified
   \[ x := x * 0 \quad \Rightarrow \quad x := 0 \]
   \[ y := y ** 2 \quad \Rightarrow \quad y := y * y \]
   \[ x := x * 8 \quad \Rightarrow \quad x := x << 3 \]
   \[ x := x * 15 \quad \Rightarrow \quad t := x << 4; x := t - x \]
   (on some machines \(<\) is faster than \(*\); but not on all!)
Constant Folding

- Operations on constants can be computed at compile time
- In general, if there is a statement
  \[ x := y \text{ op } z \]
  - where \( y \) and \( z \) are constants
  - then \( y \text{ op } z \) can be computed at compile time

- Example: \( x := 20 + 22 \Rightarrow x := 42 \)
- Example: \( \text{if } 42 < 17 \text{ goto L} \) can be deleted
Flow of Control Optimizations

• Eliminating unreachable code:
  - Code that is unreachable in the control-flow graph
  - Basic blocks that are not the target of any jump or “fall through” from a conditional
  - Such basic blocks can be eliminated

• Why/how would such basic blocks occur?

• Removing unreachable code makes the program smaller
  - And sometimes also faster
    • Due to memory cache effects (increased spatial locality)
Single Assignment Form

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

• Basic blocks of intermediate code can be rewritten to be in **single assignment** form

  \[
  x := z + y \\
  a := x \\
  \Rightarrow \\
  b := z + y \\
  a := b \\
  x := 2 \times x \\
  x := 2 \times b
  \]

  \(b\) is a fresh temporary

• More complicated in general, due to control flow (e.g., loops)
  - *Static single assignment (SSA)* form
Common Subexpression Elimination

- **Assume**
  - A basic block is in single assignment form
  - A definition $x :=$ is the first use of $x$ in a block
- **All assignments with same RHS compute the same value**

- **Example:**
  
  $x := y * z$
  
  $w := y * z$

  $x := y * z$
  
  $w := x$

  (due to the block being in single assignment form, the values of $x$, $y$ and $z$ do not change in the ... code)
Copy Propagation

• If $w := x$ appears in a block, all subsequent uses of $w$ can be replaced with uses of $x$

• Example:

  $b := z + y$
  $a := b$  \(\Rightarrow\)  $a := b$
  $x := 2 \times a$
  $x := 2 \times b$

• This does not make the program smaller or faster but might enable other optimizations
  - Constant folding
  - Dead code elimination
Constant Propagation and Constant Folding

- Example:
  \[
  \begin{align*}
  a & := 5 \\
  x & := 2 \times a \\
  y & := x + 6 \\
  t & := x \times y \\
  \Rightarrow \\
  a & := 5 \\
  x & := 10 \\
  y & := 16 \\
  t & := 160
  \end{align*}
  \]
Dead Code Elimination

If

\[ w := \text{RHS} \] appears in a basic block
\[ w \] does not appear anywhere else in the program

Then

the statement \[ w := \text{RHS} \] is dead and can be eliminated
- *Dead* = does not contribute to the program’s result

Example: *(a is not used anywhere else)*

\[
\begin{align*}
  x &:= z + y & x &:= z + y & x &:= z + y \\
  a &:= x & \Rightarrow & a &:= x & \Rightarrow & b &:= 2 \times x \\
  b &:= 2 \times a & b &:= 2 \times x
\end{align*}
\]
Applying Local Optimizations

• Each local optimization does very little by itself

• Typically optimizations interact
  – Performing one optimization enables another

• Optimizing compilers repeatedly perform optimizations until no improvement is possible
  – The optimizer can also be stopped at any time to limit the compilation time
An Example

Initial code:

\[
\begin{align*}
  a &:= x^{**2} \\
  b &:= 3 \\
  c &:= x \\
  d &:= c \cdot c \\
  e &:= b \cdot 2 \\
  f &:= a + d \\
  g &:= e \cdot f
\end{align*}
\]

assume that only \( f \) and \( g \) are used in the rest of program
An Example

Algebraic simplification:

\[
\begin{align*}
a & := x \times 2 \\
b & := 3 \\
c & := x \\
d & := c \times c \\
e & := b \times 2 \\
f & := a + d \\
g & := e \times f
\end{align*}
\]
An Example

Algebraic simplification:

\[
\begin{align*}
a & := x \times x \\
b & := 3 \\
c & := x \\
d & := c \times c \\
e & := b \ll 1 \\
f & := a + d \\
g & := e \times f
\end{align*}
\]
An Example

Copy and constant propagation:

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

Copy and constant propagation:

\[
\begin{align*}
a & := x \times x \\
b & := 3 \\
c & := x \\
d & := x \times x \\
e & := 3 \ll 1 \\
f & := a + d \\
g & := e \times f
\end{align*}
\]
An Example

Constant folding:

\begin{align*}
a & := x \times x \\
b & := 3 \\
c & := x \\
d & := x \times x \\
e & := 3 \ll 1 \\
f & := a + d \\
g & := e \times f
\end{align*}
An Example

**Constant folding:**

\[
\begin{align*}
a &:= x \times x \\
b &:= 3 \\
c &:= x \\
d &:= x \times x \\
e &:= 6 \\
f &:= a + d \\
g &:= e \times f
\end{align*}
\]
An Example

Common subexpression elimination:

\[
\begin{align*}
a &:= x \times x \\
b &:= 3 \\
c &:= x \\
d &:= x \times x \\
e &:= 6 \\
f &:= a + d \\
g &:= e \times f
\end{align*}
\]
An Example

Common subexpression elimination:

\[ a := x \times x \]
\[ b := 3 \]
\[ c := x \]
\[ d := a \]
\[ e := 6 \]
\[ f := a + d \]
\[ g := e \times f \]
Copy and constant propagation:

\[
\begin{align*}
a &:= x \times x \\
b &:= 3 \\
c &:= x \\
d &:= a \\
e &:= 6 \\
f &:= a + d \\
g &:= e \times f
\end{align*}
\]
An Example

Copy and constant propagation:

\[
\begin{align*}
a &:= x \times x \\
b &:= 3 \\
c &:= x \\
d &:= a \\
e &:= 6 \\
f &:= a + a \\
g &:= 6 \times f
\end{align*}
\]
An Example

Dead code elimination:

\[
\begin{align*}
a &:= x \times x \\
b &:= 3 \\
c &:= x \\
d &:= a \\
e &:= 6 \\
f &:= a + a \\
g &:= 6 \times f
\end{align*}
\]
An Example

Dead code elimination:

\[ a := x \times x \]

\[ f := a + a \]
\[ g := 6 \times f \]

This is the final form
Peephole Optimizations on Assembly Code

- The optimizations presented before work on intermediate code
  - They are target independent
  - But they can be applied on assembly language also

Peephole optimization is an effective technique for improving assembly code
- The “peephole” is a short sequence of (usually contiguous) instructions
- The optimizer replaces the sequence with another equivalent (but faster) one
Implementing Peephole Optimizations

• 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:

\[ \text{move } $a \ $b, \text{ move } $b \ $a \rightarrow \text{move } $a \ $b \]

- Works if \( \text{move } $b \ $a \) is not the target of a jump

• Another example:

\[ \text{addiu } $a \ $a \ i, \text{ addiu } $a \ $a \ j \rightarrow \text{addiu } $a \ $a \ i+j \]
Peephole Optimizations

- Redundant instruction elimination, e.g.:

  
  
  \[ \ldots \]
  \[ \text{goto L} \]
  \[ L: \]
  \[ \ldots \]
  
  \[ \Rightarrow \]
  
  
  \[ \ldots \]
  
  \[ L: \]
  \[ \ldots \]

- Flow of control optimizations, e.g.:

  
  
  \[ \ldots \]
  \[ \text{goto L1} \]
  \[ \ldots \]
  \[ \text{L1}: \text{goto L2} \]
  \[ \ldots \]
  
  \[ \Rightarrow \]
  
  
  \[ \ldots \]
  \[ \text{goto L2} \]
  \[ \ldots \]
  \[ \text{L1}: \text{goto L2} \]
  \[ \ldots \]
Peephole Optimizations (Cont.)

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

- Just like for local optimizations, peephole optimizations need to be applied repeatedly to get maximum effect
Concluding Remarks

• Multiple front-ends, multiple back-ends via intermediate codes

• Intermediate code is the right representation for many optimizations

• Many simple optimizations can still be applied on assembly language

• Next time: global optimizations