**Instruction Selection**

**The Problem**

Modern computers (still) have many ways to do anything

Consider register-to-register copy in ILOC

- Obvious operation is \( \text{addI} \ r_i,0 \Rightarrow r_j \)
- Many others exist

\[
\begin{array}{c|c|c}
\text{addI} & r_i,0 & \Rightarrow r_j \\
\text{subI} & r_i,0 & \Rightarrow r_j \\
\text{multI} & r_i,1 & \Rightarrow r_j \\
\text{divI} & r_i,1 & \Rightarrow r_j \\
\text{orI} & r_i,0 & \Rightarrow r_j \\
\text{xorI} & r_i,0 & \Rightarrow r_j \\
\end{array}
\]

- Human would ignore all of these
- Algorithm must look at all of them & find low-cost encoding
  
  - Take context into account (busy functional unit?)

And ILOC is an overly-simplified case
The Goal
Want to automate generation of instruction selectors

Machine description should also help with scheduling & allocation

The Big Picture
Need pattern matching techniques
- Must produce good code
- Must run quickly

Our treewalk code generator (Lec. 22) ran quickly
How good was the code?

Tree | Treewalk Code | Desired Code
--- | --- | ---
IDENT \(<a, \text{ARP}, 4>\) | loadI \(4 \rightarrow r_5\) | loadAI \(r_{0,4} \rightarrow r_5\)
IDENT \(<b, \text{ARP}, 8>\) | loadAO \(r_0, r_5 \rightarrow r_6\) | loadAO \(r_0, r_7 \rightarrow r_8\)
        | loadI \(8 \rightarrow r_7\) | loadAI \(r_{0,8} \rightarrow r_6\)
        | loadAO \(r_0, r_7 \rightarrow r_8\) | loadAI \(r_{0,8} \rightarrow r_8\)
        | mult \(r_6, r_8 \rightarrow r_9\) | mult \(r_5, r_6 \rightarrow r_7\)
The Big Picture

Need pattern matching techniques
  • Must produce good code \((\text{some metric for good})\)
  • Must run quickly

Our treewalk code generator (Lec. 22) ran quickly
How good was the code?

Tree Treewalk Code Desired Code

\[
\begin{array}{c}
\text{IDENT} \\
\langle a, \text{ARP}, 4 \rangle
\end{array}
\begin{array}{c}
\text{NUMBER} \\
\langle 2 \rangle
\end{array}
\]

\[
\begin{array}{l}
\text{loadI} 4 \Rightarrow r_5 \\
\text{loadAO} r_0, r_5 \Rightarrow r_6 \\
\text{loadI} 2 \Rightarrow r_7 \\
\text{mult} r_6, r_7 \Rightarrow r_8 \\
\text{loadAI} r_0, 4 \Rightarrow r_5 \\
\text{multI} r_5, 2 \Rightarrow r_7
\end{array}
\]
The Big Picture

Need pattern matching techniques
• Must produce good code
  (some metric for good)
• Must run quickly

Our treewalk code generator (Lec. 22) ran quickly
How good was the code?

<table>
<thead>
<tr>
<th>Tree</th>
<th>Treewalk Code</th>
<th>Desired Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDENT &lt;g,ARP,4&gt;</td>
<td>loadI 4 ⇒ r₅</td>
<td>add r₅,r₅ ⇒ r₇</td>
</tr>
<tr>
<td>NUMBER &lt;2&gt;</td>
<td>loadAO r₅,r₅ ⇒ r₆</td>
<td></td>
</tr>
<tr>
<td></td>
<td>loadI 2 ⇒ r₇</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mult r₆,r₇ ⇒ r₈</td>
<td></td>
</tr>
</tbody>
</table>

Another possibility that might take less time & energy — an algebraic identity

Comp 412, Fall 2010
The Big Picture

Need pattern matching techniques
• Must produce good code (some metric for good)
• Must run quickly

Our treewalk code generator met the second criteria (lec. 22)

How did it do on the first?

How do we perform this kind of matching?

Tree-oriented IR suggests pattern matching on trees
• Process takes tree-patterns as input, matcher as output
• Each pattern maps to a target-machine instruction sequence
• Use dynamic programming or bottom-up rewrite systems

Linear IR suggests using some sort of string matching
• Process takes strings as input, matcher as output
• Each string maps to a target-machine instruction sequence
• Use text matching (Aho-Corasick) or peephole matching

In practice, both work well; matchers are quite different
Peephole Matching

Basic idea
- Compiler can discover local improvements locally
  - Look at a small set of adjacent operations
  - Move a "peephole" over code & search for improvement
- Classic example was store followed by load

Original code                  Improved code
storeAI r_1           ⇒ r_0.8  storeAI r_1           ⇒ r_0.8
loadAI    r_0.8          ⇒ r_{15}  i2i       r_1           ⇒ r_{15}  

Peephole Matching

Basic idea
- Compiler can discover local improvements locally
  - Look at a small set of adjacent operations
  - Move a "peephole" over code & search for improvement
- Classic example was store followed by load
- Simple algebraic identities

<table>
<thead>
<tr>
<th>Operation</th>
<th>Original Code</th>
<th>Improved Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>addI</td>
<td>r_2,0         ⇒ r_7</td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>r_4,r_7       ⇒ r_{10}</td>
<td></td>
</tr>
<tr>
<td>multI</td>
<td>r_5,2          ⇒ r_7</td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>r_4,r_2       ⇒ r_10</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>r_2,r_2       ⇒ r_7</td>
<td></td>
</tr>
</tbody>
</table>
Peephole Matching

Basic idea
• Compiler can discover local improvements locally
  — Look at a small set of adjacent operations
  — Move a “peephole” over code & search for improvement
• Classic example was store followed by load
• Simple algebraic identities
• Jump to a jump

Original code

<table>
<thead>
<tr>
<th>Jump to 10</th>
<th>Jump to 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_{10}: jumpI \rightarrow L_{10}</td>
<td>L_{10}: jumpI \rightarrow L_{11}</td>
</tr>
</tbody>
</table>

Improved code

<table>
<thead>
<tr>
<th>Jump to 10</th>
<th>Jump to 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_{10}: jumpI \rightarrow L_{11}</td>
<td>L_{10}: jumpI \rightarrow L_{11}</td>
</tr>
</tbody>
</table>

Must be within the window