On the final exam, you are responsible for:

- Anything covered in class, except for stories about my good friend Ken Kennedy
- All lecture material after the midterm ("below the line" on the web page)
- Material in the book, by chapter:
  - 6: all of it
  - 7: all of it, but we won’t ask a question on string assignments ...
  - 8: 8.1, 8.2, 8.3, 8.4.1, 8.5.1
  - 9: intro, 9.3 (intro), 9.3.1, plus high-level understanding of reaching definitions from page 491
  - 10: 10.2
  - 11: the draft material from EaC3e (on web page) plus 11.5 from EaC2e
  - 12: 12.1, 12.2, 12.3, 12.4
Instruction Selection via Peephole Optimization

Comp 412

New and Improved, with extra complications

Chapter 11 in EaC3e
The Goal

We want to automate generation of instruction selectors (as with parsers & scanners)

The machine description should also help with scheduling & allocation
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 sequence of target-machine ops
• 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 sequence of target machine ops
• Use text matching (Aho-Corasick) or peephole matching

In practice, both work well & produce efficient matchers. Either can be implemented as a table-driven or hard-coded matcher.
Peephole Matching

Basic idea

• Compiler can discover local improvements locally
  — Look at a small window of adjacent operations — a peephole†
  — Slide the window over code & look for opportunities in the window

• Classic motivating example was a store followed by a load

Consider the AHSDT code generator’s actions for

\[
\begin{align*}
x & \leftarrow w + z; \\
a & \leftarrow x + y;
\end{align*}
\]

If \( x \) cannot be kept permanently in a register, the code generator will emit a store for \( x \), followed immediately by a load.

† A “peephole” was a pre-digital device to see who was on the other side of a door.
Peephole Matching

**Basic idea**

- Compiler can discover local improvements locally
  - Look at a small window of adjacent operations — a **peephole**
  - Slide the window over code & look for opportunities in the window

- Classic motivating example was a **store** followed by a **load**

<table>
<thead>
<tr>
<th>Original code</th>
<th>Improved code</th>
</tr>
</thead>
<tbody>
<tr>
<td>add ( r_w, r_z \Rightarrow r_1 )</td>
<td>add ( r_w, r_z \Rightarrow r_1 )</td>
</tr>
<tr>
<td>storeAl ( r_1 \Rightarrow r_{0,8} )</td>
<td>storeAl ( r_1 \Rightarrow r_{0,8} )</td>
</tr>
<tr>
<td>loadAl ( r_{0,8} \Rightarrow r_2 )</td>
<td>i2i ( r_1 \Rightarrow r_2 )</td>
</tr>
<tr>
<td>add ( r_2, r_y \Rightarrow r_3 )</td>
<td>add ( r_2, r_y \Rightarrow r_3 )</td>
</tr>
</tbody>
</table>

1. Assume \( x \) is at offset 8 from the **ARP**
2. The **store** and **load** are long-latency operations that address the same memory location. The only way the scheduler can avoid the full latency is if it proves enough to eliminate the **load**.
3. Copy coalescing is highly likely to eliminate the **i2i**.
Peephole Matching

Basic idea

• Compiler can discover local improvements locally
  – Look at a small window of adjacent operations — a peephole
  – Slide the window over code & look for opportunities in the window

• Classic motivating example was a store followed by a load

• Simple algebraic identities are another opportunity

<table>
<thead>
<tr>
<th>Original code</th>
<th>Improved code</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl ( r_2,0 \Rightarrow r_7 )</td>
<td>mult ( r_4, r_7 \Rightarrow r_{10} )</td>
</tr>
<tr>
<td>mult ( r_4, r_7 \Rightarrow r_{10} )</td>
<td></td>
</tr>
<tr>
<td>multl ( r_5, 2 \Rightarrow r_7 )</td>
<td>add ( r_5, r_5 \Rightarrow r_7 )</td>
</tr>
</tbody>
</table>

See Table on p 424 of EaC2e
Peephole Matching

Basic idea

• Compiler can discover local improvements locally
  – Look at a small window of adjacent operations — a peephole
  – Slide the window over code & look for opportunities in the window

• Classic motivating example was store followed by load
• Simple algebraic identities are another opportunity
• Jump to a jump

Original code

<table>
<thead>
<tr>
<th>Original code</th>
<th>Improved code</th>
</tr>
</thead>
<tbody>
<tr>
<td>jumpL</td>
<td>L_{10}: jumpL \rightarrow L_{11}</td>
</tr>
<tr>
<td>L_{10}: jumpL \rightarrow L_{11}</td>
<td></td>
</tr>
</tbody>
</table>

Merging an empty block in CLEAN

Second jump must be within the window
Peephole Matching

Implementing the idea

• Early peephole systems used limited set of hand-coded patterns
• Small window size ensured quick processing
  – *If we assume window size is 3 or 4, then O(n^2) ⇒ O(n)*

Modern peephole instruction selectors *(Davidson)*

• Break problem into three tasks

- Apply symbolic interpretation & simplification systematically
Peephole Matching

**Expander**

- Turns IR code into a low-level IR (LLIR) such as RTL
- Operation-by-operation, template-driven rewriting
- LLIR form includes *all* direct effects *(e.g., setting CC)*
- Significant, albeit constant, expansion of size

![Diagram of Peephole Matching process]

COMP 412, Fall 2017
Peephole Matching

**Simplifier**

- Looks at LLIR through window and rewrites it
- Uses forward substitution, algebraic simplification, local constant propagation, and dead-effect elimination
- Performs local optimization within window

- This is the heart of the peephole system
  - Benefit of peephole optimization shows up in this step
Peephole Matching

**Matcher**

- Compares simplified **LLIR** against a library of patterns
- Picks low-cost pattern that captures effects
- Must preserve **LLIR** effects, may add new ones (e.g., *set CC*)
- Generates the assembly code output
Finding Dead Effects

The Simplifier must know which ops are useless \( (i.e., \text{dead}) \)

- Expander works in a context-independent fashion
- It can process the operations in any order
  - Use a backward walk and compute local \text{LIVE} information
  - Tag each use that is a last use
- What about non-local effects?
  - Most useless effects are local — defined \& used in same block
  - It can be conservative \& assume \text{LIVE} until proven dead

\[ \text{ASM} \]
\[
\text{mult} \ r_5, r_9 \Rightarrow r_{12} \\
\text{add} \ r_{12}, r_{17} \Rightarrow r_{13}
\]

\[ \text{LLIR} \]
\[
\begin{align*}
\text{r}_{12} & \leftarrow r_5 \ast r_9 \\
\text{cc} & \leftarrow f(r_5 \ast r_9) \\
\text{r}_{13} & \leftarrow r_{12} + r_{17} \\
\text{cc} & \leftarrow f(r_{12}+r_{17})
\end{align*}
\]

\[ \text{ASM} \]
\[
\text{madd} \ r_5, r_9, r_{17} \Rightarrow r_{13}
\]

This effect would prevent multiply-add from matching

Assume that values \text{LIVE} on exit are \text{LIVE}
Example

\[ a \leftarrow b - 2 \times c \] becomes

<table>
<thead>
<tr>
<th>Original IR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Assume that:

<table>
<thead>
<tr>
<th>Kind</th>
<th>Base</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a  local variable</td>
<td>ARP</td>
<td>4</td>
</tr>
<tr>
<td>b  cbr parameter</td>
<td>ARP</td>
<td>-16</td>
</tr>
<tr>
<td>c  global variable</td>
<td>@G</td>
<td></td>
</tr>
</tbody>
</table>
Example

\[ a \leftarrow b - 2 \times c \] becomes

```
<table>
<thead>
<tr>
<th>Original IR</th>
<th>Expand</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( t_1 \leftarrow 2 \times c )</td>
<td></td>
</tr>
<tr>
<td>2 ( a \leftarrow b - t_1 )</td>
<td></td>
</tr>
</tbody>
</table>
```

We loaded \( c \) before 2 following Ravi Sethi's most demanding subtree first rule. Same thing with evaluating the expression before computing the address of \( a \).

<table>
<thead>
<tr>
<th>Base</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>ARP 4</td>
</tr>
<tr>
<td>b</td>
<td>ARP -16</td>
</tr>
<tr>
<td>c</td>
<td>@G 12</td>
</tr>
</tbody>
</table>

Assume that \( @G \) is the label's offset in the global constant pool, as in tree-pattern matching lecture.

LLIR Code

\[
\begin{align*}
  r_{10} & \leftarrow \text{@CP} \\
  r_{11} & \leftarrow \text{@G} \\
  r_{12} & \leftarrow r_{10} + r_{11} \\
  r_{13} & \leftarrow \text{MEM}(r_{12}) \\
  r_{14} & \leftarrow 12 \\
  r_{15} & \leftarrow r_{13} + r_{14} \\
  r_{16} & \leftarrow \text{MEM}(r_{15}) \\
  r_{17} & \leftarrow 2 \\
  r_{18} & \leftarrow r_{17} \times r_{16} \\
  r_{19} & \leftarrow 16 \\
  r_{20} & \leftarrow r_{\text{arp}} - r_{19} \\
  r_{21} & \leftarrow \text{MEM}(r_{20}) \\
  r_{22} & \leftarrow \text{MEM}(r_{21}) \\
  r_{23} & \leftarrow r_{22} - r_{18} \\
  r_{24} & \leftarrow 4 \\
  r_{25} & \leftarrow r_{\text{arp}} + r_{24} \\
  \text{MEM}(r_{25}) & \leftarrow r_{23}
\end{align*}
\]
Example

Remember, one of the primary goals of instruction selection is to make effective use of the ISA’s features, particularly address modes.

Among the additional details that the Expander adds are all of the nitty-gritty computations involved in addressing a, b, and c.

<table>
<thead>
<tr>
<th>Base</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>ARP</td>
</tr>
<tr>
<td>b</td>
<td>ARP</td>
</tr>
<tr>
<td>c</td>
<td>@G</td>
</tr>
</tbody>
</table>

COMP 412, Fall 2017
Example

You will sometimes hear compiler writers talk about “establishing addressability” for a global variable.

This code establishes addressability for \( c \) by retrieving the label from the constant pool.

If that label is used often (> 2 times), it may be worthwhile to load it on entry and spill it to the AR.

Among the additional details that the \textbf{Expander} adds are \textit{all of the nitty-gritty computations involved in addressing a, b, and c.}

<table>
<thead>
<tr>
<th>Base</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>ARP</td>
</tr>
<tr>
<td>b</td>
<td>ARP</td>
</tr>
<tr>
<td>c</td>
<td>@G</td>
</tr>
</tbody>
</table>

COMP 412, Fall 2017
**Example**

**LLIR Code**

| $r_{10}$ | ← | @CP |
| $r_{11}$ | ← | @G |
| $r_{12}$ | ← | $r_{10} + r_{11}$ |
| $r_{13}$ | ← | $\text{MEM}(r_{12})$ |
| $r_{14}$ | ← | 12 |
| $r_{15}$ | ← | $r_{13} + r_{14}$ |
| $r_{16}$ | ← | $\text{MEM}(r_{15})$ |
| $r_{17}$ | ← | 2 |
| $r_{18}$ | ← | $r_{17} \times r_{16}$ |
| $r_{19}$ | ← | 16 |
| $r_{20}$ | ← | $r_{\text{arp}} - r_{19}$ |
| $r_{21}$ | ← | $\text{MEM}(r_{20})$ |
| $r_{22}$ | ← | $\text{MEM}(r_{21})$ |
| $r_{23}$ | ← | $r_{22} - r_{18}$ |
| $r_{24}$ | ← | 4 |
| $r_{25}$ | ← | $r_{\text{arp}} + r_{24}$ |
| $\text{MEM}(r_{25})$ | ← | $r_{23}$ |

**Improved LLIR Code**

| $r_{13}$ | ← | $\text{MEM}(\text{CP + G})$ |
| $r_{16}$ | ← | $\text{MEM}(r_{13} + 12)$ |
| $r_{18}$ | ← | $r_{16} + r_{16}$ |
| $r_{21}$ | ← | $\text{MEM}(r_{\text{arp}} - 16)$ |
| $r_{22}$ | ← | $\text{MEM}(r_{21})$ |
| $r_{23}$ | ← | $r_{22} - r_{18}$ |
| $\text{MEM}(r_{\text{arp}} + 4)$ | ← | $r_{23}$ |

15 LLIR operations to 7
Remember, one of the primary goals of instruction selection is to make effective use of the ISA’s features, particularly address modes.

**Example**

**Improved LLIR Code**

<table>
<thead>
<tr>
<th>Register</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_{13}$</td>
<td>$\text{MEM}(@CP + @G)$</td>
</tr>
<tr>
<td>$r_{16}$</td>
<td>$\text{MEM}(r_{13} + 12)$</td>
</tr>
<tr>
<td>$r_{18}$</td>
<td>$r_{16} + r_{16}$</td>
</tr>
<tr>
<td>$r_{21}$</td>
<td>$\text{MEM}(r_{\text{arp}} - 16)$</td>
</tr>
<tr>
<td>$r_{22}$</td>
<td>$\text{MEM}(r_{21})$</td>
</tr>
<tr>
<td>$r_{23}$</td>
<td>$r_{22} - r_{18}$</td>
</tr>
<tr>
<td>MEM($r_{\text{arp}} + 4$)</td>
<td>$r_{23}$</td>
</tr>
</tbody>
</table>

**Final ILOC Code**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source(s)</th>
<th>Destination(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI</td>
<td>@CP</td>
<td>$r_{10}$</td>
</tr>
<tr>
<td>loadAl</td>
<td>$r_{10}$, @G</td>
<td>$r_{11}$</td>
</tr>
<tr>
<td>loadAl</td>
<td>$r_{11}$, 12</td>
<td>$r_{12}$</td>
</tr>
<tr>
<td>add</td>
<td>$r_{12}$, $r_{12}$</td>
<td>$r_{13}$</td>
</tr>
<tr>
<td>subl</td>
<td>$r_{\text{arp}}$, 16</td>
<td>$r_{14}$</td>
</tr>
<tr>
<td>load</td>
<td>$r_{14}$</td>
<td>$r_{15}$</td>
</tr>
<tr>
<td>load</td>
<td>$r_{15}$</td>
<td>$r_{16}$</td>
</tr>
<tr>
<td>sub</td>
<td>$r_{16}$, $r_{13}$</td>
<td>$r_{17}$</td>
</tr>
<tr>
<td>storeAl</td>
<td>$r_{17}$</td>
<td>$r_{\text{arp}}$, 4</td>
</tr>
</tbody>
</table>

- Introduces all memory operations & temporary names
- Turns out pretty good code; equivalent to tree pattern matcher’s code

7 LLIR operations to 9 final operations
LLIR Code

\[
\begin{align*}
r_{10} & \leftarrow @CP \\
r_{11} & \leftarrow @G \\
r_{12} & \leftarrow r_{10} + r_{11} \\
r_{13} & \leftarrow \text{MEM}(r_{12}) \\
r_{14} & \leftarrow 12 \\
r_{15} & \leftarrow r_{13} + r_{14} \\
r_{16} & \leftarrow \text{MEM}(r_{15}) \\
r_{17} & \leftarrow 2 \\
r_{18} & \leftarrow r_{17} \times r_{16} \\
r_{19} & \leftarrow 16 \\
r_{20} & \leftarrow r_{\text{arp}} - r_{19} \\
r_{21} & \leftarrow \text{MEM}(r_{20}) \\
r_{22} & \leftarrow \text{MEM}(r_{21}) \\
r_{23} & \leftarrow r_{22} - r_{18} \\
r_{24} & \leftarrow 4 \\
r_{25} & \leftarrow r_{\text{arp}} + r_{24} \\
\text{MEM}(r_{25}) & \leftarrow r_{23}
\end{align*}
\]

COMP 412, Fall 2017

Steps of the Simplifier

Fill the window

Three operation window

1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination
Steps of the Simplifier

### LLIR Code

<table>
<thead>
<tr>
<th>Step</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_{10}$</td>
<td>$\leftarrow \text{@CP}$</td>
</tr>
<tr>
<td>$r_{11}$</td>
<td>$\leftarrow \text{@G}$</td>
</tr>
<tr>
<td>$r_{12}$</td>
<td>$\leftarrow r_{10} + r_{11}$</td>
</tr>
<tr>
<td>$r_{13}$</td>
<td>$\leftarrow \text{MEM}(r_{12})$</td>
</tr>
<tr>
<td>$r_{14}$</td>
<td>$\leftarrow 12$</td>
</tr>
<tr>
<td>$r_{15}$</td>
<td>$\leftarrow r_{13} + r_{14}$</td>
</tr>
<tr>
<td>$r_{16}$</td>
<td>$\leftarrow \text{MEM}(r_{15})$</td>
</tr>
<tr>
<td>$r_{17}$</td>
<td>$\leftarrow 2$</td>
</tr>
<tr>
<td>$r_{18}$</td>
<td>$\leftarrow r_{17} \times r_{16}$</td>
</tr>
<tr>
<td>$r_{19}$</td>
<td>$\leftarrow 16$</td>
</tr>
<tr>
<td>$r_{20}$</td>
<td>$\leftarrow r_{\text{arp}} - r_{19}$</td>
</tr>
<tr>
<td>$r_{21}$</td>
<td>$\leftarrow \text{MEM}(r_{20})$</td>
</tr>
<tr>
<td>$r_{22}$</td>
<td>$\leftarrow \text{MEM}(r_{21})$</td>
</tr>
<tr>
<td>$r_{23}$</td>
<td>$\leftarrow r_{22} - r_{18}$</td>
</tr>
<tr>
<td>$r_{24}$</td>
<td>$\leftarrow 4$</td>
</tr>
<tr>
<td>$r_{25}$</td>
<td>$\leftarrow r_{\text{arp}} + r_{24}$</td>
</tr>
<tr>
<td>$\text{MEM}(r_{25})$</td>
<td>$\leftarrow r_{23}$</td>
</tr>
</tbody>
</table>

### Three operation window

<table>
<thead>
<tr>
<th>Step</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_{10}$</td>
<td>$\leftarrow \text{@CP}$</td>
</tr>
<tr>
<td>$r_{11}$</td>
<td>$\leftarrow \text{@G}$</td>
</tr>
<tr>
<td>$r_{12}$</td>
<td>$\leftarrow \text{@CP} + \text{@G}$</td>
</tr>
</tbody>
</table>

### Forward substitute

- Assignment to $r_{12}$ was the last use of $r_{10} \& r_{11}$
- The assignments to $r_{10} \& r_{11}$ are now dead

1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination
LLIR Code

1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination

Discard and fill
— delete ops with dead results
— will combine with Forward substitute from here on out
1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination
Steps of the Simplifier

| r_{13}   | \texttt{MEM}(\texttt{CP} + \texttt{G}) |
| r_{15}   | r_{13} + 12 |
| r_{16}   | \texttt{MEM}(r_{15}) |

Forward substitute, discard, and fill

1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination
### LLIR Code

<table>
<thead>
<tr>
<th>$r_{10}$</th>
<th>@CP</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_{11}$</td>
<td>@G</td>
</tr>
<tr>
<td>$r_{12}$</td>
<td>$r_{10} + r_{11}$</td>
</tr>
<tr>
<td>$r_{13}$</td>
<td>MEM($r_{12}$)</td>
</tr>
<tr>
<td>$r_{14}$</td>
<td>12</td>
</tr>
<tr>
<td>$r_{15}$</td>
<td>$r_{13} + r_{14}$</td>
</tr>
<tr>
<td>$r_{16}$</td>
<td>MEM($r_{15}$)</td>
</tr>
<tr>
<td>$r_{17}$</td>
<td>2</td>
</tr>
<tr>
<td>$r_{18}$</td>
<td>$r_{17} \times r_{16}$</td>
</tr>
<tr>
<td>$r_{19}$</td>
<td>16</td>
</tr>
<tr>
<td>$r_{20}$</td>
<td>$r_{\text{arp}} - r_{19}$</td>
</tr>
<tr>
<td>$r_{21}$</td>
<td>MEM($r_{20}$)</td>
</tr>
<tr>
<td>$r_{22}$</td>
<td>MEM($r_{21}$)</td>
</tr>
<tr>
<td>$r_{23}$</td>
<td>$r_{22} - r_{18}$</td>
</tr>
<tr>
<td>$r_{24}$</td>
<td>4</td>
</tr>
<tr>
<td>$r_{25}$</td>
<td>$r_{\text{arp}} + r_{24}$</td>
</tr>
<tr>
<td>MEM($r_{25}$)</td>
<td>$r_{23}$</td>
</tr>
</tbody>
</table>

### Steps of the Simplifier

#### Three operation window

- $r_{13} \leftarrow \text{MEM}(\text{CP} + \text{G})$
- $r_{16} \leftarrow \text{MEM}(r_{13} + 12)$
- $r_{17} \leftarrow 2$

#### Forward substitute, discard, and fill
- *nothing more to do in this window*

1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination

COMP 412, Fall 2017
**LLIR Code**

\[ r_{10} \leftarrow \@CP \]
\[ r_{11} \leftarrow \@G \]
\[ r_{12} \leftarrow r_{10} + r_{11} \]
\[ r_{13} \leftarrow \text{MEM}(r_{12}) \]
\[ r_{14} \leftarrow 12 \]
\[ r_{15} \leftarrow r_{13} + r_{14} \]
\[ r_{16} \leftarrow \text{MEM}(r_{15}) \]
\[ r_{17} \leftarrow 2 \]
\[ r_{18} \leftarrow r_{17} \times r_{16} \]
\[ r_{19} \leftarrow 16 \]
\[ r_{20} \leftarrow r_{\text{arp}} - r_{19} \]
\[ r_{21} \leftarrow \text{MEM}(r_{20}) \]
\[ r_{22} \leftarrow \text{MEM}(r_{21}) \]
\[ r_{23} \leftarrow r_{22} - r_{18} \]
\[ r_{24} \leftarrow 4 \]
\[ r_{25} \leftarrow r_{\text{arp}} + r_{24} \]
\[ \text{MEM}(r_{25}) \leftarrow r_{23} \]

**Steps of the Simplifier**

- Roll definition of \( r_{13} \) out of the window and fill — “roll” removes op from window and emits the op for the matcher

1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination

**Three operation window**

\[
\begin{align*}
  r_{16} &\leftarrow \text{MEM}(r_{13} + 12) \\
  r_{17} &\leftarrow 2 \\
  r_{18} &\leftarrow r_{17} \times r_{16}
\end{align*}
\]
LLIR Code

Forward substitute, discard, and fill

Steps of the Simplifier

1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination

Three operation window
Steps of the Simplifier

LLIR Code

1. $r_{10} \leftarrow \text{@CP}$
2. $r_{11} \leftarrow \text{@G}$
3. $r_{12} \leftarrow r_{10} + r_{11}$
4. $r_{13} \leftarrow \text{MEM}(r_{12})$
5. $r_{14} \leftarrow 12$
6. $r_{15} \leftarrow r_{13} + r_{14}$
7. $r_{16} \leftarrow \text{MEM}(r_{15})$
8. $r_{17} \leftarrow 2$
9. $r_{18} \leftarrow r_{17} \times r_{16}$
10. $r_{19} \leftarrow 16$
11. $r_{20} \leftarrow \text{arp} - r_{19}$
12. $r_{21} \leftarrow \text{MEM}(r_{20})$
13. $r_{22} \leftarrow \text{MEM}(r_{21})$
14. $r_{23} \leftarrow r_{22} - r_{18}$
15. $r_{24} \leftarrow 4$
16. $r_{25} \leftarrow \text{arp} + r_{24}$
17. \(\text{MEM}(r_{25}) \leftarrow r_{23}\)

COMP 412, Fall 2017

Three operation window

\[
\begin{array}{c}
r_{16} \leftarrow \text{MEM}(r_{13} + 12) \\
r_{18} \leftarrow r_{16} + r_{16} \\
r_{19} \leftarrow 16
\end{array}
\]

Apply algebraic identity

\(2 \times a = a + a\)

1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination
LLIR Code

\[
\begin{align*}
    r_{10} &\leftarrow \text{@CP} \\
    r_{11} &\leftarrow \text{@G} \\
    r_{12} &\leftarrow r_{10} + r_{11} \\
    r_{13} &\leftarrow \text{MEM}(r_{12}) \\
    r_{14} &\leftarrow 12 \\
    r_{15} &\leftarrow r_{13} + r_{14} \\
    r_{16} &\leftarrow \text{MEM}(r_{15}) \\
    r_{17} &\leftarrow 2 \\
    r_{18} &\leftarrow r_{17} \times r_{16} \\
    r_{19} &\leftarrow 16 \\
    r_{20} &\leftarrow r_{\text{arp}} - r_{19} \\
    r_{21} &\leftarrow \text{MEM}(r_{20}) \\
    r_{22} &\leftarrow \text{MEM}(r_{21}) \\
    r_{23} &\leftarrow r_{22} - r_{18} \\
    r_{24} &\leftarrow 4 \\
    r_{25} &\leftarrow r_{\text{arp}} + r_{24} \\
    \text{MEM}(r_{25}) &\leftarrow r_{23}
\end{align*}
\]

Steps of the Simplifier

1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination

Three operation window

- \[ r_{18} \leftarrow r_{16} + r_{16} \]
- \[ r_{19} \leftarrow 16 \]
- \[ r_{20} \leftarrow r_{\text{arp}} - r_{19} \]

Roll the definition of \( r_{16} \) out of the window and fill
LLIR Code

\[ r_{10} \leftarrow @CP \]
\[ r_{11} \leftarrow @G \]
\[ r_{12} \leftarrow r_{10} + r_{11} \]
\[ r_{13} \leftarrow \text{MEM}(r_{12}) \]
\[ r_{14} \leftarrow 12 \]
\[ r_{15} \leftarrow r_{13} + r_{14} \]
\[ r_{16} \leftarrow \text{MEM}(r_{15}) \]
\[ r_{17} \leftarrow 2 \]
\[ r_{18} \leftarrow r_{17} \times r_{16} \]
\[ r_{19} \leftarrow 16 \]
\[ r_{20} \leftarrow r_{\text{arp}} - r_{19} \]
\[ r_{21} \leftarrow \text{MEM}(r_{20}) \]
\[ r_{22} \leftarrow \text{MEM}(r_{21}) \]
\[ r_{23} \leftarrow r_{22} - r_{18} \]
\[ r_{24} \leftarrow 4 \]
\[ r_{25} \leftarrow r_{\text{arp}} + r_{24} \]
\[ \text{MEM}(r_{25}) \leftarrow r_{23} \]

Steps of the Simplifier

<table>
<thead>
<tr>
<th>Step</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Forward substitute, discard, and fill</td>
</tr>
<tr>
<td>2.</td>
<td>Algebraic simplification</td>
</tr>
<tr>
<td>3.</td>
<td>Local constant propagation</td>
</tr>
<tr>
<td>4.</td>
<td>Dead-effect elimination</td>
</tr>
</tbody>
</table>

Three operation window
LLIR Code

\[
\begin{align*}
    r_{10} & \leftarrow @CP \\
    r_{11} & \leftarrow @G \\
    r_{12} & \leftarrow r_{10} + r_{11} \\
    r_{13} & \leftarrow \text{MEM}(r_{12}) \\
    r_{14} & \leftarrow 12 \\
    r_{15} & \leftarrow r_{13} + r_{14} \\
    r_{16} & \leftarrow \text{MEM}(r_{15}) \\
    r_{17} & \leftarrow 2 \\
    r_{18} & \leftarrow r_{17} \times r_{16} \\
    r_{19} & \leftarrow 16 \\
    r_{20} & \leftarrow r_{\text{arp}} - r_{19} \\
    r_{21} & \leftarrow \text{MEM}(r_{20}) \\
    r_{22} & \leftarrow \text{MEM}(r_{21}) \\
    r_{23} & \leftarrow r_{22} - r_{18} \\
    r_{24} & \leftarrow 4 \\
    r_{25} & \leftarrow r_{\text{arp}} + r_{24} \\
    \text{MEM}(r_{25}) & \leftarrow r_{23}
\end{align*}
\]

Steps of the Simplifier

1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination

Three operation window

Forward substitute, discard, and fill
LLIR Code

\[
\begin{align*}
r_{10} & \leftarrow \text{@CP} \\
r_{11} & \leftarrow \text{@G} \\
r_{12} & \leftarrow r_{10} + r_{11} \\
r_{13} & \leftarrow \text{MEM}(r_{12}) \\
r_{14} & \leftarrow 12 \\
r_{15} & \leftarrow r_{13} + r_{14} \\
r_{16} & \leftarrow \text{MEM}(r_{15}) \\
r_{17} & \leftarrow 2 \\
r_{18} & \leftarrow r_{17} \times r_{16} \\
r_{19} & \leftarrow 16 \\
r_{20} & \leftarrow r_{\text{arp}} - r_{19} \\
r_{21} & \leftarrow \text{MEM}(r_{20}) \\
r_{22} & \leftarrow \text{MEM}(r_{21}) \\
r_{23} & \leftarrow r_{22} - r_{18} \\
r_{24} & \leftarrow 4 \\
r_{25} & \leftarrow r_{\text{arp}} + r_{24} \\
\text{MEM}(r_{25}) & \leftarrow r_{23}
\end{align*}
\]

Steps of the Simplifier

1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination

Three operation window

Roll definition of \( r_{18} \) out of the window and fill
### LLIR Code

\[ \begin{align*}
    r_{10} & \leftarrow @CP \\
    r_{11} & \leftarrow @G \\
    r_{12} & \leftarrow r_{10} + r_{11} \\
    r_{13} & \leftarrow \text{MEM}(r_{12}) \\
    r_{14} & \leftarrow 12 \\
    r_{15} & \leftarrow r_{13} + r_{14} \\
    r_{16} & \leftarrow \text{MEM}(r_{15}) \\
    r_{17} & \leftarrow 2 \\
    r_{18} & \leftarrow r_{17} \times r_{16} \\
    r_{19} & \leftarrow 16 \\
    r_{20} & \leftarrow r_{\text{arp}} - r_{19} \\
    r_{21} & \leftarrow \text{MEM}(r_{20}) \\
    r_{22} & \leftarrow \text{MEM}(r_{21}) \\
    r_{23} & \leftarrow r_{22} - r_{18} \\
    r_{24} & \leftarrow 4 \\
    r_{25} & \leftarrow r_{\text{arp}} + r_{24} \\
    \text{MEM}(r_{25}) & \leftarrow r_{23}
\end{align*} \]

### Steps of the Simplifier

- Roll definition of \( r_{21} \) out of the window and fill

### Three operation window

\[
\begin{align*}
    r_{22} & \leftarrow \text{MEM}(r_{21}) \\
    r_{23} & \leftarrow r_{22} - r_{18} \\
    r_{24} & \leftarrow 4
\end{align*}
\]

1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination

COMP 412, Fall 2017
LLIR Code

\[\begin{align*}
  r_{10} & \leftarrow @CP \\
  r_{11} & \leftarrow @G \\
  r_{12} & \leftarrow r_{10} + r_{11} \\
  r_{13} & \leftarrow \text{MEM}(r_{12}) \\
  r_{14} & \leftarrow 12 \\
  r_{15} & \leftarrow r_{13} + r_{14} \\
  r_{16} & \leftarrow \text{MEM}(r_{15}) \\
  r_{17} & \leftarrow 2 \\
  r_{18} & \leftarrow r_{17} \times r_{16} \\
  r_{19} & \leftarrow 16 \\
  r_{20} & \leftarrow \text{arp} - r_{19} \\
  r_{21} & \leftarrow \text{MEM}(r_{20}) \\
  r_{22} & \leftarrow \text{MEM}(r_{21}) \\
  r_{23} & \leftarrow r_{22} - r_{18} \\
  r_{24} & \leftarrow 4 \\
  r_{25} & \leftarrow \text{arp} + r_{24} \\
  \text{MEM}(r_{25}) & \leftarrow r_{23}
\end{align*}\]

Steps of the Simplifier

\[
\begin{array}{c|c}
  r_{23} & \leftarrow r_{22} - r_{18} \\
  r_{24} & \leftarrow 4 \\
  r_{25} & \leftarrow r_{\text{arp}} + r_{24}
\end{array}
\]

Three operation window

1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination

Roll definition of \( r_{22} \) out of the window and fill
LLIR Code

\[ \begin{align*} 
  r_{10} & \leftarrow \@CP \\
  r_{11} & \leftarrow \@G \\
  r_{12} & \leftarrow r_{10} + r_{11} \\
  r_{13} & \leftarrow \text{MEM}(r_{12}) \\
  r_{14} & \leftarrow 12 \\
  r_{15} & \leftarrow r_{13} + r_{14} \\
  r_{16} & \leftarrow \text{MEM}(r_{15}) \\
  r_{17} & \leftarrow 2 \\
  r_{18} & \leftarrow r_{17} \times r_{16} \\
  r_{19} & \leftarrow 16 \\
  r_{20} & \leftarrow r_{\text{arp}} - r_{19} \\
  r_{21} & \leftarrow \text{MEM}(r_{20}) \\
  r_{22} & \leftarrow \text{MEM}(r_{21}) \\
  r_{23} & \leftarrow r_{22} - r_{18} \\
  r_{24} & \leftarrow 4 \\
  r_{25} & \leftarrow r_{\text{arp}} + r_{24} \\
  \text{MEM}(r_{25}) & \leftarrow r_{23} 
\end{align*} \]

Steps of the Simplifier:

1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination

Three operation window:

\[ \begin{align*} 
  r_{23} & \leftarrow r_{22} - r_{18} \\
  r_{25} & \leftarrow r_{\text{arp}} + 4 \\
  \text{MEM}(r_{25}) & \leftarrow r_{23} 
\end{align*} \]
LLIR Code

\[\begin{align*}
    r_{10} & \leftarrow @CP \\
    r_{11} & \leftarrow @G \\
    r_{12} & \leftarrow r_{10} + r_{11} \\
    r_{13} & \leftarrow \text{MEM}(r_{12}) \\
    r_{14} & \leftarrow 12 \\
    r_{15} & \leftarrow r_{13} + r_{14} \\
    r_{16} & \leftarrow \text{MEM}(r_{15}) \\
    r_{17} & \leftarrow 2 \\
    r_{18} & \leftarrow r_{17} \times r_{16} \\
    r_{19} & \leftarrow 16 \\
    r_{20} & \leftarrow \text{arp} - r_{19} \\
    r_{21} & \leftarrow \text{MEM}(r_{20}) \\
    r_{22} & \leftarrow \text{MEM}(r_{21}) \\
    r_{23} & \leftarrow r_{22} - r_{18} \\
    r_{24} & \leftarrow 4 \\
    r_{25} & \leftarrow \text{arp} + r_{24} \\
\end{align*}\]

\[\text{MEM}(r_{25}) \leftarrow r_{23}\]

Steps of the Simplifier

<table>
<thead>
<tr>
<th>Step</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>(r_{23} \leftarrow r_{22} - r_{18})</td>
</tr>
<tr>
<td>24</td>
<td>(\text{MEM}(\text{arp} + 4) \leftarrow r_{23})</td>
</tr>
<tr>
<td>25</td>
<td>— empty —</td>
</tr>
</tbody>
</table>

Three operation window

Forward substitute, discard, and try to fill — out of ops

1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination
### LLIR Code

<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_{10}$</td>
<td>$\leftarrow @CP$</td>
</tr>
<tr>
<td>$r_{11}$</td>
<td>$\leftarrow @G$</td>
</tr>
<tr>
<td>$r_{12}$</td>
<td>$\leftarrow r_{10} + r_{11}$</td>
</tr>
<tr>
<td>$r_{13}$</td>
<td>$\leftarrow \text{MEM}(r_{12})$</td>
</tr>
<tr>
<td>$r_{14}$</td>
<td>$\leftarrow 12$</td>
</tr>
<tr>
<td>$r_{15}$</td>
<td>$\leftarrow r_{13} + r_{14}$</td>
</tr>
<tr>
<td>$r_{16}$</td>
<td>$\leftarrow \text{MEM}(r_{15})$</td>
</tr>
<tr>
<td>$r_{17}$</td>
<td>$\leftarrow 2$</td>
</tr>
<tr>
<td>$r_{18}$</td>
<td>$\leftarrow r_{17} \times r_{16}$</td>
</tr>
<tr>
<td>$r_{19}$</td>
<td>$\leftarrow 16$</td>
</tr>
<tr>
<td>$r_{20}$</td>
<td>$\leftarrow r_{\text{arp}} - r_{19}$</td>
</tr>
<tr>
<td>$r_{21}$</td>
<td>$\leftarrow \text{MEM}(r_{20})$</td>
</tr>
<tr>
<td>$r_{22}$</td>
<td>$\leftarrow \text{MEM}(r_{21})$</td>
</tr>
<tr>
<td>$r_{23}$</td>
<td>$\leftarrow r_{22} - r_{18}$</td>
</tr>
<tr>
<td>$r_{24}$</td>
<td>$\leftarrow 4$</td>
</tr>
<tr>
<td>$r_{25}$</td>
<td>$\leftarrow r_{\text{arp}} + r_{24}$</td>
</tr>
<tr>
<td>$\text{MEM}(r_{25})$</td>
<td>$\leftarrow r_{23}$</td>
</tr>
</tbody>
</table>

### Steps of the Simplifier

1. **Forward substitution**
2. **Algebraic simplification**
3. **Local constant propagation**
4. **Dead-effect elimination**
LLIR Code

1. Forward substitution
2. Algebraic simplification
3. Local constant propagation
4. Dead-effect elimination
Results of the Simplifier

**LLIR Code**

\[
\begin{align*}
    r_{10} & \leftarrow \ @CP \\
    r_{11} & \leftarrow \ @G \\
    r_{12} & \leftarrow r_{10} + r_{11} \\
    r_{13} & \leftarrow \text{MEM}(r_{12}) \\
    r_{14} & \leftarrow 12 \\
    r_{15} & \leftarrow r_{13} + r_{14} \\
    r_{16} & \leftarrow \text{MEM}(r_{15}) \\
    r_{17} & \leftarrow 2 \\
    r_{18} & \leftarrow r_{17} \times r_{16} \\
    r_{19} & \leftarrow 16 \\
    r_{20} & \leftarrow \text{arp} - r_{19} \\
    r_{21} & \leftarrow \text{MEM}(r_{20}) \\
    r_{22} & \leftarrow \text{MEM}(r_{21}) \\
    r_{23} & \leftarrow r_{22} - r_{18} \\
    r_{24} & \leftarrow 4 \\
    r_{25} & \leftarrow \text{arp} + r_{24} \\
\end{align*}
\]

**Improved LLIR Code**

\[
\begin{align*}
    r_{13} & \leftarrow \text{MEM}(\@CP + \@G) \\
    r_{16} & \leftarrow \text{MEM}(r_{13} + 12) \\
    r_{18} & \leftarrow r_{16} + r_{16} \\
    r_{21} & \leftarrow \text{MEM}(\text{arp} - 16) \\
    r_{22} & \leftarrow \text{MEM}(r_{21}) \\
    r_{23} & \leftarrow r_{22} - r_{18} \\
    \text{MEM}(\text{arp} + 4) & \leftarrow r_{23}
\end{align*}
\]

These are the ops that rolled out of the top of the window.
Matcher

Remember, one of the primary goals of instruction selection is to make effective use of the ISA’s features, particularly address modes.

### Improved LLIR Code

| r_{13}  | ← | Mem(@CP + @G) |
| r_{16}  | ← | Mem(r_{13} + 12) |
| r_{18}  | ← | r_{16} + r_{16} |
| r_{21}  | ← | Mem(r_{arp} - 16) |
| r_{22}  | ← | Mem(r_{21}) |
| r_{23}  | ← | r_{22} - r_{18} |
| Mem(r_{arp} + 4) | ← | r_{23} |

### Final ILOC Code

| loadI | @CP | ⇒ r_{10} |
| loadAI | r_{10}, @G | ⇒ r_{11} |
| loadAI | r_{11}, 12 | ⇒ r_{12} |
| add | r_{12}, r_{12} | ⇒ r_{13} |
| subI | r_{arp}, 16 | ⇒ r_{14} |
| load | r_{14} | ⇒ r_{15} |
| load | r_{15} | ⇒ r_{16} |
| sub | r_{16}, r_{13} | ⇒ r_{17} |
| storeAI | r_{17} | ⇒ r_{arp}, 4 |

### Introduced memory operations, address modes, & temporary names

- Similar code to tree pattern matcher, except for
  - `multI` versus `add`
  - different order of operations
- Used address modes and immediate operands where possible
Matcher

Remember, one of the primary goals of instruction selection is to make effective use of the ISA’s features, particularly address modes.

**Improved LLIR Code**

- \( r_{13} \leftarrow \text{MEM}(\@CP + \@G) \)
- \( r_{16} \leftarrow \text{MEM}(r_{13} + 12) \)
- \( r_{18} \leftarrow r_{16} + r_{16} \)
- \( r_{21} \leftarrow \text{MEM}(r_{\text{arp}} - 16) \)
- \( r_{22} \leftarrow \text{MEM}(r_{21}) \)
- \( r_{23} \leftarrow r_{22} - r_{18} \)
- \( \text{MEM}(r_{\text{arp}} + 4) \leftarrow r_{23} \)

**Final ILOC Code**

- \( \text{loadI} \quad \@CP \quad \Rightarrow r_{10} \)
- \( \text{loadAI} \quad r_{10}, \@G \quad \Rightarrow r_{11} \)
- \( \text{loadAI} \quad r_{11}, 12 \quad \Rightarrow r_{12} \)
- \( \text{add} \quad r_{12}, r_{12} \quad \Rightarrow r_{13} \)
- \( \text{subI} \quad r_{\text{arp}}, 16 \quad \Rightarrow r_{14} \)
- \( \text{load} \quad r_{14} \quad \Rightarrow r_{15} \)
- \( \text{load} \quad r_{15} \quad \Rightarrow r_{16} \)
- \( \text{sub} \quad r_{16}, r_{13} \quad \Rightarrow r_{17} \)
- \( \text{storeAI} \quad r_{17} \quad \Rightarrow r_{\text{arp}}, 4 \)

**Introduced memory operations, address modes, & temporary names**

1. Matcher must know about size of constants (e.g., \(@CP\) versus \(@G\))
2. Matcher had to reconstruct \(\text{subI}\) from \(\text{MEM}(r_{\text{arp}} - 16)\)
   - Other operations were a one-to-one match
Remember, one of the primary goals of instruction selection is to make effective use of the ISA’s features, particularly address modes.

Actual systems use different technologies for the matcher

• Hand-coded matcher for LLIR that emits target machine code (gcc)
• LR parser adapted to handle (wildly) ambiguous grammars (VPO)
• Some kind of formal matcher (e.g., Aho-Corasick)
Making It All Work

Details

• **LLIR** is largely machine independent (RTL in gcc)
  – Some compilers use **LLIR** as one of their **IRs**
  – Eliminates the need for the **Expander**

• Target machine described with **LLIR** → **ASM** patterns for **Matcher**

• Actual pattern matching
  – Use a hand-coded pattern matcher (gcc)
  – Turn patterns into grammar & use **LR** parser (VPO)

• Several important compilers use this technology

• It seems to produce good portable instruction selectors

Key strength appears to be late low-level optimization
Other Considerations

Control-flow operations

• Can clear simplifier’s window at branch or label
• Predication has similar effects
  – May want to special case predicated single operations so as not to disrupt the flow of the simplifier too often ...

Physical versus logical windows

• Can run optimizer over a logical window
  – $k$ operations connected definition to use
• Expander can link definitions & uses
• Logical windows (within block) improve effectiveness

Davidson & Fraser report 30% faster & 20% fewer ops with local logical window.
Instruction Selection

The Big Picture

• Selection by Tree-Pattern Matching
  – Assumed a tree structure for both IR and machine description
    → Note: trees not DAGs, and matches on syntax, not values
  – Recursive bottom-up procedure to find tiling(s)
  – Adding costs makes it behave like dynamic programming

• Selection by Peephole Optimization
  – Assumed a linear IR and performed late-stage optimization
  – Forward substitution, algebraic simplification, local constant propagation, & dead-effect elimination
  – Relies on a small window (or peephole) to control complexity

• Both techniques automate a complex part of the process
• Both techniques can produce “high-quality” code
How good is the code that these techniques produce?

• The code is not “optimal”
  – Tree-pattern matchers are limited by the available patterns
    → Initial 1-to-1 mapping might produce mediocre code
    → Adding more patterns can capitalize on complex ISA features
      – Multiply-add, address modes, decrement & branch, call/ret …
      – Leads to larger tables
    → The match is a syntactic match (cannot inspect values)
  – Peephole technique is limited by size of the window & strength of its transformations
    → Logical windows do better than physical windows
    → Can examine values (e.g., for algebraic simplification)
    → Adding more transformations would improve code & increase complexity

• The literature introduces the term “locally optimal” to describe quality
  – As well as can be done with only local knowledge
  – Both techniques can claim “locally optimal” code

• For non-local improvement, rely on the compiler’s optimizer