The second exam will be a take-home exam, available Monday after class, return it by Wednesday noon.

Instruction Selection: Preliminaries

Comp 412

Copyright 2019, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.
The Compiler’s Back End

Front End
• Scans & parses the input program
• Performs semantic checking & builds an IR version of the code

Optimizer
• Analyzes & transforms the code to improve its performance

Back End
• Transforms the IR program into target-machine assembly code
• Maps resources in the IR program into target-machine resources
Writing a good compiler is a lot of work

• Would like to reuse components whenever possible
• Would like to automate construction of components

• Front end construction is largely automated
• Optimizer is largely hand crafted
• (Parts of) the back end can be automated

Chap. 11 Goal: Automating Instruction Selection
In discussing the compiler’s back end, we will assume the model shown to the left:

- Instruction selection is largely solved
  - A major problem of the 1970’s & 1980s
- Register allocation & instruction scheduling are more complex
  - Both are NP-complete in any real case
- We assume a unified register set
  - Partitioned register sets add operation placement to the mix — another hard problem

Parallelism either comes from the application, or is derived with techniques that are beyond the scope of this course (ask Luay to offer COMP 515).
Structure of the Compiler’s Back End

What about the IR?

• Common case today is an ILOC-like IR
  – Branches, memory tags, hierarchy of memory operations, multiple ops/cycle
  – `clang` is ILOC-like but in SSA form, `gcc` is a lower level, linear IR

• Assume that several things are explicit:
  – “Enough” register names
  – Order of operations & flow of values

• All of these techniques also work on a tree-like IR
Definitions

**Instruction selection** (§ 11)
- Maps the IR into assembly code
- Assumes a fixed *storage mapping & code shape*
- Combines operations, uses address modes

**Instruction scheduling** (§ 12)
- Reorders the operations to hide latencies
- Assumes a *fixed choice of operations*
- Changes demand for registers

**Register allocation** (§ 13)
- Decides which values will reside in registers
- Assumes a *fixed order of existing operations*
- Changes name space; may add false sharing

These three problems are tightly coupled.
Conventional wisdom says that we lose little by solving these problems independently.

**Instruction selection**
- Use some form of pattern matching
- Assume enough registers or target “important” values

**Instruction scheduling**
- Within a block, local list scheduling is “close” to optimal
- Across blocks, build framework to apply list scheduling

**Register allocation**
- Start from virtual registers & map “enough” into $k$
- Global allocation requires global model or paradigm

Local allocation & local scheduling have a similar algorithmic flavor. We don’t have a good paradigm for global scheduling.
The fundamental problem is that modern computers have many ways to perform any single logical operation.

Consider register-to-register copy in ILOC:

- Obvious operation is $i2i \ r_i \Rightarrow r_j$
- Many others exist

\[
\begin{array}{ccc}
\text{addI} & r_i,0 \Rightarrow r_j & \text{subI} & r_i,0 \Rightarrow r_j & \text{multI} & r_i,1 \Rightarrow r_j \\
\text{lshiftI} & r_i,0 \Rightarrow r_j & \text{orI} & r_i,0 \Rightarrow r_j & \text{divI} & r_i,1 \Rightarrow r_j \\
\text{rshiftI} & r_i,0 \Rightarrow r_j & \text{xorI} & r_i,0 \Rightarrow r_j & \text{... and others ...}
\end{array}
\]

Some of these are cheap

Some of these are expensive

- Human would ignore most, if not all, of these
- An algorithm will look at all of them & find the low-cost choice
  - Take context into account

And ILOC is an much simpler than a real ISA
We want to automate generation of instruction selectors (as with parsers & scanners)

The machine description should also help with scheduling & allocation

Simplification: we will restrict our discussion to signed integers. Expanding to handle other types is straightforward. It adds more rules to the examples, not more complexity.
The Big Picture

**Need pattern matching techniques**

- Must produce good code  
  *(under some metric for “good”)*
- Must run quickly

The simple schemes we talked about in SDT ran quickly

How good was the code?

**Syntax-Driven Translation**

**Tree**

- `x`
- `IDENT <a,ARP,4>`
- `IDENT <b,ARP,8>`

**AHSDT Code**

- `loadI 4 \Rightarrow r_5`
- `loadAO r_0,r_5 \Rightarrow r_6`
- `loadI 8 \Rightarrow r_7`
- `loadAO r_0,r_7 \Rightarrow r_8`
- `mult r_6,r_8 \Rightarrow r_9`

**Desired Code**

- `loadAI r_0,4 \Rightarrow r_5`
- `loadAI r_0,8 \Rightarrow r_6`
- `mult r_5,r_6 \Rightarrow r_7`

Assume **ARP** is in $r_0$

**Notation:** Variable named `b`, stored at offset 8 from the **ARP**, which is in $r_0$
The Big Picture

Need pattern matching techniques

• Must produce good code
• Must run quickly

The simple schemes we talked about in SDT ran quickly

How good was the code?

Tree

IDENT <a,ARP,4>
IDENT <b,ARP,8>

x

AhSDT Code

loadI 4 \Rightarrow r_5
loadAO r_0,r_5 \Rightarrow r_6
loadI 8 \Rightarrow r_7
loadAO r_0,r_7 \Rightarrow r_8
mult r_6,r_8 \Rightarrow r_9

Desired Code

loadAl r_0,4 \Rightarrow r_5
loadAl r_0,8 \Rightarrow r_6
mult r_5,r_6 \Rightarrow r_7

Assume ARP is in r_0

This inefficiency is easy to fix. See digression on page 346 of EaC2e.
The Big Picture

Need pattern matching techniques
• Must produce good code
• Must run quickly

The simple schemes we saw earlier ran quickly
How good was the code?

Tree

x

IDENT
<a,ARP,4>

NUMBER
<2>

AHSĐT Code

loadI
loadAO
loadI
mult

4 ➞ r₅
r₀,r₅ ➞ r₆
2 ➞ r₇
r₆,r₇ ➞ r₈

Desired Code

loadAI
multI

r₀,4 ➞ r₅
r₅,2 ➞ r₇
The Big Picture

Need pattern matching techniques
• Must produce good code
• Must run quickly

The simple schemes we saw earlier ran quickly
How good was the code?

Tree

<table>
<thead>
<tr>
<th>IDENT &lt;a,ARP,4&gt;</th>
<th>NUMBER &lt;2&gt;</th>
</tr>
</thead>
</table>

AHSĐT Code
- loadI 4 ⇒ r₅
- loadAO r₀,r₅ ⇒ r₆
- loadI 2 ⇒ r₇
- mult r₆,r₇ ⇒ r₈

Desired Code
- loadAl r₀,4 ⇒ r₅
- mult r₅,2 ⇒ r₇

Must use info from both these nodes. The loadAI example was purely local.
The Big Picture

Need pattern matching techniques
• Must produce good code
• Must run quickly

The simple schemes we saw earlier ran quickly
How good was the code?

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

Pattern matching is not the answer to all problems with code quality.

LVN finds this algebraic identity easily.

Tree

<table>
<thead>
<tr>
<th>IDENT</th>
<th>NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;a,ARP,4&gt;</td>
<td>&lt;2&gt;</td>
</tr>
</tbody>
</table>

AHSĐT Code

- loadI 4 → r₅
- loadAO r₅, r₅ → r₆
- loadI 2 → r₇
- mult r₆, r₇ → r₈

Desired Code

- loadAI r₀,4 → r₅
- add r₅, r₅ → r₇

Pattern matching techniques must produce good code and run quickly.
How do we perform this kind of matching?

**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

**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

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

\[
x \leftarrow w + z;
\]
\[
a \leftarrow x + y;
\]

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</td>
<td>add</td>
</tr>
<tr>
<td>r_w, r_z ⇒ r_1</td>
<td>r_w, r_z ⇒ r_1</td>
</tr>
<tr>
<td>storeAl r_1 ⇒ r_0,8</td>
<td>storeAl r_1 ⇒ r_0,8</td>
</tr>
<tr>
<td>loadAl r_0,8 ⇒ r_2</td>
<td>i2i r_1 ⇒ r_2</td>
</tr>
<tr>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td>r_2, r_y ⇒ r_3</td>
<td>r_2, r_y ⇒ r_3</td>
</tr>
</tbody>
</table>

1. Assume x is at offset 8 from the ARP, which is stored in r_0
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 $r_4, r_2 \Rightarrow r_{10}$</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>

Algebraic simplification in LVN

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**

```
jump l
L_{10}: jump l \to L_{10}
```

**Improved code**

```
L_{10}: jump l \to L_{11}
```

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 $O(n^2) \Rightarrow O(n)$*

Modern peephole instruction selectors

- Break problem into three tasks

  - Apply symbolic interpretation & simplification systematically

COMP 412, Fall 2019
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
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., dead)

- Expander works in a context-independent fashion
- It can process the operations in any order
  - Use a backward walk and compute local 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 LIVE until proven dead

```asm
mult r_5, r_9 => r_12
add r_12, r_17 => r_13
```

```llir
r_12 <- r_5 * r_9
cc  <- f(r_5 * r_9)
r_13 <- r_12 + r_17
cc  <- f(r_12 + r_17)
```

```asm
madd r_5, r_9, r_17 => r_13
```

This effect would prevent multiply-add from matching

As in Lab 1

Assume that values LIVE on exit are LIVE

COMP 412, Fall 2019
Example

\[ a \leftarrow b - 2 * c \] 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>local variable</td>
<td>ARP</td>
<td>4</td>
</tr>
<tr>
<td>cbr parameter</td>
<td>ARP</td>
<td>-16</td>
</tr>
<tr>
<td>global variable</td>
<td>@G</td>
<td></td>
</tr>
</tbody>
</table>
Example

a ↵ b - 2 * c becomes

<table>
<thead>
<tr>
<th>Original IR</th>
<th>Expand</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>t₁ ↵ 2 * c</td>
</tr>
<tr>
<td>2</td>
<td>a ↵ b - t₁</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</td>
</tr>
<tr>
<td>b</td>
<td>ARP</td>
</tr>
<tr>
<td>c</td>
<td>@G</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

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

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

a ← b − 2 * c  becomes

<table>
<thead>
<tr>
<th>Original IR</th>
<th>Expand</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Original IR" /></td>
<td><img src="image" alt="Expand" /></td>
</tr>
</tbody>
</table>

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

COMP 412, Fall 2019
You will sometimes hear compiler writers talk about “establishing addressability” for a global variable.

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>

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.

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

LLIR Code

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

Improved LLIR Code

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

\textbf{Simplify}

15 LLIR operations to 7
Example

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

**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**

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

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