Topic 6
Basic Back-End Optimization

Instruction Selection
Instruction scheduling
Register allocation
ABET Outcome

• Ability to apply knowledge of basic code generation techniques, e.g. Instruction selection, instruction scheduling, register allocation, to solve code generation problems.

• Ability to analyze the basic algorithms on the above techniques and conduct experiments to show their effectiveness.

• Ability to use a modern compiler development platform and tools for the practice of above.

• A Knowledge on contemporary issues on this topic.
Three Basic Back-End Optimization

**Instruction selection**
- Mapping IR into assembly code
- Assumes a fixed storage mapping & code shape
- Combining operations, using address modes

**Instruction scheduling**
- Reordering operations to hide latencies
- Assumes a fixed program (set of operations)
- Changes demand for registers

**Register allocation**
- Deciding which values will reside in registers
- Changes the storage mapping, may add false sharing
- Concerns about placement of data & memory operations
Instruction Selection

Some slides are from CS 640 lecture in George Mason University
Reading List

(1) K. D. Cooper & L. Torczon, Engineering a Compiler, Chapter 11
(2) Dragon Book, Chapter 8.7, 8.9
Objectives

• Introduce the complexity and importance of instruction selection

• Study practical issues and solutions

• Case study: Instruction Selection in Open64
Instruction Selection: Retargetable

Machine description → Back-end Generator → Tables → Pattern Matching Engine

Description-based retargeting
Complexity of Instruction

Selection

Modern computers have many ways to do anything.

Consider a register-to-register copy

• Obvious operation is: move rj, ri

• Many others exist

    add rj, ri, 0  sub rj, ri, 0  rshiftI rj, ri, 0
    mul rj, ri, 1  or rj, ri, 0  divI rj, r, 1
    xor rj, ri, 0  others …
Complexity of Instruction

Selection (Cont.)

- Multiple addressing modes
- Each alternate sequence has its cost
  - Complex ops (mult, div): several cycles
  - Memory ops: latency vary
- Sometimes, cost is context related
- Use under-utilized FUs
- Dependent on objectives: speed, power, code size
Additional constraints on specific operations
- Load/store multiple words: contiguous registers
- Multiply: need special register Accumulator

Interaction between instruction selection, instruction scheduling, and register allocation
- For scheduling, instruction selection predetermines latencies and function units
- For register allocation, instruction selection pre-colors some variables. e.g. non-uniform registers (such as registers for multiplication)
Instruction Selection Techniques

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

Peephole-based Matching
- Linear IR suggests using some sort of string matching
- Inspired by peephole optimization
- Strings as input, matcher as output
- Each string maps to a target-machine instruction sequence

In practice, both work well; matchers are quite different.
A Simple Tree-Walk Code Generation Method

• Assume starting with a Tree-like IR
• Starting from the root, recursively walking through the tree
• At each node use a simple (unique) rule to generate a low-level instruction
Tree Pattern-Matching

- Assumptions
  - tree-like IR - an AST
  - Assume each subtree of IR – there is a corresponding set of tree patterns (or “operation trees” - low-level abstract syntax tree)

- Problem formulation: Find a best mapping of the AST to operations by “tiling” the AST with operation trees (where tiling is a collection of (AST-node, operation-tree) pairs).
Tile AST

An AST tree

Tile 1
val

num

Tile 2
val

num

Tile 3
lab

num

Tile 4
ref

num

ref

Tile 5
ref

num

ref

Tile 6
gets

+
Tile AST with Operation Trees

Goal is to “tile” AST with operation trees.

- A tiling is collection of `<ast-node, op-tree>` pairs
  - `ast-node` is a node in the AST
  - `op-tree` is an operation tree
  - `<ast-node, op-tree>` means that `op-tree` could implement the subtree at `ast-node`
- A tiling ‘implements” an AST if it covers every node in the AST and the overlap between any two trees is limited to a single node
  - `<ast-node, op-tree>` tiling means `ast-node` is also covered by a leaf in another operation tree in the tiling, unless it is the root
  - Where two operation trees meet, they must be compatible (expect the value in the same location)
Tree Walk by Tiling: An Example

\[ a = a + 22; \]
Example

\[a = a + 22;\]
Example: An Alternative

\[ a = a + 22; \]

\[ \text{ld } t1, [sp+a] \]
\[ \text{add } t2, t1, 22 \]
\[ \text{st } [sp+a], t2 \]
Finding Matches to Tile the Tree

- Compiler writer connects operation trees to AST subtrees
  - Provides a set of rewrite rules
  - Encode tree syntax, in linear form
  - Associated with each is a code template
Generating Code in Tilings

Given a tiled tree

- Postorder treewalk, with node-dependent order for children
  - Do right child before its left child

- Emit code sequence for tiles, in order

- Tie boundaries together with register names
  - Can incorporate a “real” register allocator or can simply use “NextRegister++” approach
Optimal Tilings

• Best tiling corresponds to least cost instruction sequence

• Optimal tiling
  - no two adjacent tiles can be combined to a tile of lower cost
Dynamic Programming for Optimal Tiling

- For a node $x$, let $f(x)$ be the cost of the optimal tiling for the whole expression tree rooted at $x$. Then

$$f(x) = \min_{\forall \text{tile } T \text{ covering } x} \left( \text{cost}(T) + \sum_{\forall \text{child } y \text{ of tile } T} f(y) \right)$$
Dynamic Programming for Optimal Tiling (Con’t)

- Maintain a table: node $x$ $\rightarrow$ the optimal tiling covering node $x$ and its cost
- Start from root recursively:
  - check in table for optimal tiling for this node
  - If not computed, try all possible tiling and find the optimal, store lowest-cost tile in table and return
- Finally, use entries in table to emit code
Peephole-based Matching

- Basic idea inspired by peephole optimization
- Compiler can discover local improvements locally
  - Look at a small set of adjacent operations
  - Move a “peephole” over code & search for improvement

A Classic example is store followed by load

**Original code**

```
st $r1,($r0)
ld $r2,($r0)
```

**Improved code**

```
st $r1,($r0)
move $r2,$r1
```
Implementing Peephole Matching

• Early systems used limited set of hand-coded patterns
• Window size ensured quick processing
• Modern peephole instruction selectors break problem into three tasks

IR → Expander (IR → LLIR) → LLIR → Simplifier (LLIR → LLIR) → LLIR → Matcher (LLIR → ASM) → ASM

LLIR: Low Level IR
ASM: Assembly Code
Implementing Peephole Matching (Con’t)

**Expander**
- Turns IR code into a low-level IR (LLIR)
- Operation-by-operation, template-driven rewriting
- LLIR form includes all direct effects
- Significant, albeit constant, expansion of size

**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 and benefit of peephole optimization shows up in this step

**Matcher**
- Compares simplified LLIR against a library of patterns
- Picks low-cost pattern that captures effects
- Must preserve LLIR effects, may add new ones
- Generates the assembly code output
Some Design Issues of Peephole Optimization

- **Dead values**
  
  - Recognizing dead values is critical to remove useless effects, e.g., condition code
  
  - **Expander**
    
    - Construct a list of dead values for each low-level operation by backward pass over the code
    
    - Example: consider the code sequence:
      
      \[
      r1 = r_i * r_j \\
      cc = fx(r_i, r_j) \quad \text{// is this dead ?} \\
      r2 = r1 + r_k \\
      cc = fx(r1, r_k)
      \]
Some Design Issues of Peephole Optimization (Cont.)

- Control flow and predicated operations
  - A simple way: Clear the simplifier’s window when it reaches a branch, a jump, or a labeled or predicated instruction
  - A more aggressive way: to be discussed next
Some Design Issues of Peephole Optimization (Cont.)

- Physical vs. Logical Window
  - Simplifier uses a window containing adjacent low level operations
  - However, adjacent operations may not operate on the same values
    - In practice, they may tend to be independent for parallelism or resource usage reasons
Some Design Issues of Peephole Optimization (Cont.)

- **Use Logical Window**
  - Simplifier can link each definition with the next use of its value in the same basic block
    - Simplifier largely based on forward substitution
    - No need for operations to be physically adjacent
  - More aggressively, extend to larger scopes beyond a basic block.
An Example

<table>
<thead>
<tr>
<th>OP</th>
<th>Arg₁</th>
<th>Arg₂</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult</td>
<td>2</td>
<td>y</td>
<td>t₁</td>
</tr>
<tr>
<td>sub</td>
<td>x</td>
<td>t₁</td>
<td>w</td>
</tr>
</tbody>
</table>

\[ \text{r13: } y \]
\[ \text{r14: } t₁ \]
\[ \text{r17: } x \]
\[ \text{r20: } w \]

where (@x,@y,@w are offsets of x, y and w from a global location stored in r0)
An Example (Con’t)

**LLIR Code**

- \( r_{10} \leftarrow 2 \)
- \( r_{11} \leftarrow @y \)
- \( r_{12} \leftarrow r_0 + r_{11} \)
- \( r_{13} \leftarrow \text{MEM}(r_{12}) \)
- \( r_{14} \leftarrow r_{10} * r_{13} \)
- \( r_{15} \leftarrow @x \)
- \( r_{16} \leftarrow r_0 + r_{15} \)
- \( r_{17} \leftarrow \text{MEM}(r_{16}) \)
- \( r_{18} \leftarrow r_{17} - r_{14} \)
- \( r_{19} \leftarrow @w \)
- \( r_{20} \leftarrow r_0 + r_{19} \)
- \( \text{MEM}(r_{20}) \leftarrow r_{18} \)

**Simplify**

\( r_{13} \leftarrow \text{MEM}(r_0 + @y) \)
\( r_{14} \leftarrow 2 * r_{13} \)
\( r_{17} \leftarrow \text{MEM}(r_0 + @x) \)
\( r_{18} \leftarrow r_{17} - r_{14} \)
\( \text{MEM}(r_0 + @w) \leftarrow r_{18} \)

**Original IR Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>Arg₁</th>
<th>Arg₂</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult</td>
<td>2</td>
<td>y</td>
<td>( t_1 )</td>
</tr>
<tr>
<td>sub</td>
<td>x</td>
<td>( t_1 )</td>
<td>( w )</td>
</tr>
</tbody>
</table>
**An Example (Con’t)**

**LLIR Code**
- \( r_{13} \leftarrow \text{MEM}(r_0 + @y) \)
- \( r_{14} \leftarrow 2 * r_{13} \)
- \( r_{17} \leftarrow \text{MEM}(r_0 + @x) \)
- \( r_{18} \leftarrow r_{17} - r_{14} \)
- \( \text{MEM}(r_0 + @w) \leftarrow r_{18} \)

**ILOC Assembly Code**
- `loadAI r_0, @y \rightarrow r_{13}`
- `multI 2 * r_{13} \rightarrow r_{14}`
- `loadAI r_0, @x \rightarrow r_{17}`
- `sub r_{17} - r_{14} \rightarrow r_{18}`
- `storeAI r_{18} \rightarrow r_0, @w`

- Introduced all memory operations & temporary names
- Turned out pretty good code

**loadAI:** load from memory to register
**Multi:** multiplication with a constant operand
**storeAI:** store to memory

**Original IR Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>Arg₁</th>
<th>Arg₂</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult</td>
<td>2</td>
<td>y</td>
<td>( t_1 )</td>
</tr>
<tr>
<td>sub</td>
<td>x</td>
<td>( t_1 )</td>
<td>w</td>
</tr>
</tbody>
</table>
Simplifier (3-operation window)

LLIR Code

\[
\begin{align*}
    r_{10} &\leftarrow 2 \\
    r_{11} &\leftarrow @y \\
    r_{12} &\leftarrow r_{0} + r_{11} \\
    r_{13} &\leftarrow \text{MEM}(r_{12}) \\
    r_{14} &\leftarrow r_{10} \times r_{13} \\
    r_{15} &\leftarrow @x \\
    r_{16} &\leftarrow r_{0} + r_{15} \\
    r_{17} &\leftarrow \text{MEM}(r_{16}) \\
    r_{18} &\leftarrow r_{17} - r_{14} \\
    r_{19} &\leftarrow @w \\
    r_{20} &\leftarrow r_{0} + r_{19} \\
    \text{MEM}(r_{20}) &\leftarrow r_{18}
\end{align*}
\]
**Simplifier (3-operation window)**

**LLIR Code**

\[
\begin{align*}
    r_{10} & \leftarrow 2 \\
    r_{11} & \leftarrow @y \\
    r_{12} & \leftarrow r_0 + r_{11} \\
    r_{13} & \leftarrow \text{MEM}(r_{12}) \\
    r_{14} & \leftarrow r_{10} \ast r_{13} \\
    r_{15} & \leftarrow @x \\
    r_{16} & \leftarrow r_0 + r_{15} \\
    r_{17} & \leftarrow \text{MEM}(r_{16}) \\
    r_{18} & \leftarrow r_{17} - r_{14} \\
    r_{19} & \leftarrow @w \\
    r_{20} & \leftarrow r_0 + r_{19} \\
    \text{MEM}(r_{20}) & \leftarrow r_{18}
\end{align*}
\]
Simplifier (3-operation window)

LLIR Code

\[
\begin{align*}
    r_{10} &\leftarrow 2 \\
r_{11} &\leftarrow @y \\
r_{12} &\leftarrow r_0 + r_{11} \\
r_{13} &\leftarrow \text{MEM}(r_{12}) \\
r_{14} &\leftarrow r_{10} \times r_{13} \\
r_{15} &\leftarrow @x \\
r_{16} &\leftarrow r_0 + r_{15} \\
r_{17} &\leftarrow \text{MEM}(r_{16}) \\
r_{18} &\leftarrow r_{17} - r_{14} \\
r_{19} &\leftarrow @w \\
r_{20} &\leftarrow r_0 + r_{19} \\
\text{MEM}(r_{20}) &\leftarrow r_{18}
\end{align*}
\]
Simplifier (3-operation window)

LLIR Code

\[
\begin{align*}
r_{10} & \leftarrow 2 \\
r_{11} & \leftarrow @y \\
r_{12} & \leftarrow r_0 + r_{11} \\
r_{13} & \leftarrow \text{MEM}(r_{12}) \\
r_{14} & \leftarrow r_{10} * r_{13} \\
r_{15} & \leftarrow @x \\
r_{16} & \leftarrow r_0 + r_{15} \\
r_{17} & \leftarrow \text{MEM}(r_{16}) \\
r_{18} & \leftarrow r_{17} - r_{14} \\
r_{19} & \leftarrow @w \\
r_{20} & \leftarrow r_0 + r_{19} \\
\text{MEM}(r_{20}) & \leftarrow r_{18}
\end{align*}
\]
**Simplifier (3-operation window)**

**LLIR Code**

\[
\begin{align*}
  r_{10} &\leftarrow 2 \\
  r_{11} &\leftarrow @y \\
  r_{12} &\leftarrow r_0 + r_{11} \\
  r_{13} &\leftarrow \text{MEM}(r_{12}) \\
  r_{14} &\leftarrow r_{10} \times r_{13} \\
  r_{15} &\leftarrow @x \\
  r_{16} &\leftarrow r_0 + r_{15} \\
  r_{17} &\leftarrow \text{MEM}(r_{16}) \\
  r_{18} &\leftarrow r_{17} - r_{14} \\
  r_{19} &\leftarrow @w \\
  r_{20} &\leftarrow r_0 + r_{19} \\
  \text{MEM}(r_{20}) &\leftarrow r_{18}
\end{align*}
\]

**1st op it has rolled out of window**

\[
\begin{align*}
  r_{13} &\leftarrow \text{MEM}(r_0 + @y) \\
  r_{14} &\leftarrow 2 \times r_{13} \\
  r_{15} &\leftarrow @x \\
  r_{16} &\leftarrow r_0 + r_{15}
\end{align*}
\]
**Simplifier (3-operation window)**

**LLIR Code**

\[
\begin{align*}
r_{10} & \leftarrow 2 \\
r_{11} & \leftarrow @y \\
r_{12} & \leftarrow r_0 + r_{11} \\
r_{13} & \leftarrow \text{MEM}(r_{12}) \\
r_{14} & \leftarrow r_{10} \times r_{13} \\
r_{15} & \leftarrow @x \\
r_{16} & \leftarrow r_0 + r_{15} \\
r_{17} & \leftarrow \text{MEM}(r_{16}) \\
r_{18} & \leftarrow r_{17} - r_{14} \\
r_{19} & \leftarrow @w \\
r_{20} & \leftarrow r_0 + r_{19} \\
\text{MEM}(r_{20}) & \leftarrow r_{18}
\end{align*}
\]

\[
\begin{align*}
r_{14} & \leftarrow 2 \times r_{13} \\
r_{15} & \leftarrow @x \\
r_{16} & \leftarrow r_0 + @x \\
r_{17} & \leftarrow \text{MEM}(r_{16})
\end{align*}
\]
Simplifier (3-operation window)

LLIR Code

\[
\begin{align*}
    r_{10} &\leftarrow 2 \\
    r_{11} &\leftarrow @y \\
    r_{12} &\leftarrow r_0 + r_{11} \\
    r_{13} &\leftarrow \text{MEM}(r_{12}) \\
    r_{14} &\leftarrow r_{10} * r_{13} \\
    r_{15} &\leftarrow @x \\
    r_{16} &\leftarrow r_0 + r_{15} \\
    r_{17} &\leftarrow \text{MEM}(r_{16}) \\
    r_{18} &\leftarrow r_{17} - r_{14} \\
    r_{19} &\leftarrow @w \\
    r_{20} &\leftarrow r_0 + r_{19} \\
    \text{MEM}(r_{20}) &\leftarrow r_{18}
\end{align*}
\]

\[
\begin{align*}
    r_{14} &\leftarrow 2 * r_{13} \\
    r_{16} &\leftarrow r_0 + @x \\
    r_{17} &\leftarrow \text{MEM}(r_0 + @x) \\
    r_{18} &\leftarrow r_{17} - r_{14}
\end{align*}
\]
**Simplifier (3-operation window)**

**LLIR Code**

\[
\begin{align*}
    r_{10} & \leftarrow 2 \\
    r_{11} & \leftarrow \@y \\
    r_{12} & \leftarrow r_0 + r_{11} \\
    r_{13} & \leftarrow \text{MEM}(r_{12}) \\
    r_{14} & \leftarrow r_{10} \times r_{13} \\
    r_{15} & \leftarrow \@x \\
    r_{16} & \leftarrow r_0 + r_{15} \\
    r_{17} & \leftarrow \text{MEM}(r_{16}) \\
    r_{18} & \leftarrow r_{17} - r_{14} \\
    r_{19} & \leftarrow \@w \\
    r_{20} & \leftarrow r_0 + r_{19} \\
    \text{MEM}(r_{20}) & \leftarrow r_{18}
\end{align*}
\]

\[
\begin{align*}
    r_{13} & \leftarrow \text{MEM}(r_0 + \@y) \\
    r_{14} & \leftarrow 2 \times r_{13} \\
    r_{17} & \leftarrow \text{MEM}(r_0 + \@x) \\
    r_{18} & \leftarrow r_{17} - r_{14} \\
    r_{19} & \leftarrow \@w
\end{align*}
\]
Simplifier (3-operation window)

LLIR Code

\[
\begin{align*}
    r_{10} & \leftarrow 2 \\
    r_{11} & \leftarrow @y \\
    r_{12} & \leftarrow r_0 + r_{11} \\
    r_{13} & \leftarrow \text{MEM}(r_{12}) \\
    r_{14} & \leftarrow r_{10} \times r_{13} \\
    r_{15} & \leftarrow @x \\
    r_{16} & \leftarrow r_0 + r_{15} \\
    r_{17} & \leftarrow \text{MEM}(r_{16}) \\
    r_{18} & \leftarrow r_{17} - r_{14} \\
    r_{19} & \leftarrow @w \\
    r_{20} & \leftarrow r_0 + r_{19} \\
    \text{MEM}(r_{20}) & \leftarrow r_{18}
\end{align*}
\]

\[
\begin{align*}
    r_{13} & \leftarrow \text{MEM}(r_0 + @y) \\
    r_{14} & \leftarrow 2 \times r_{13} \\
    r_{17} & \leftarrow \text{MEM}(r_0 + @x) \\
    r_{18} & \leftarrow r_{17} - r_{14} \\
    r_{19} & \leftarrow @w \\
    r_{20} & \leftarrow r_0 + r_{19}
\end{align*}
\]
Simplifier (3-operation window)

LLIR Code

\[
\begin{align*}
    & r_{10} \leftarrow 2 \\
    & r_{11} \leftarrow \@y \\
    & r_{12} \leftarrow r_0 + r_{11} \\
    & r_{13} \leftarrow \text{MEM}(r_{12}) \\
    & r_{14} \leftarrow r_{10} \times r_{13} \\
    & r_{15} \leftarrow \@x \\
    & r_{16} \leftarrow r_0 + r_{15} \\
    & r_{17} \leftarrow \text{MEM}(r_{16}) \\
    & r_{18} \leftarrow r_{17} - r_{14} \\
    & r_{19} \leftarrow \@w \\
    & r_{20} \leftarrow r_0 + r_{19} \\
    & \text{MEM}(r_{20}) \leftarrow r_{18}
\end{align*}
\]

\[
\begin{align*}
    & r_{13} \leftarrow \text{MEM}(r_{0+\@y}) \\
    & r_{14} \leftarrow 2 \times r_{13} \\
    & r_{17} \leftarrow \text{MEM}(r_{0+\@x}) \\
    & r_{18} \leftarrow r_{17} - r_{14} \\
    & r_{19} \leftarrow \@w \\
    & r_{20} \leftarrow r_0 + \@w \\
    & \text{MEM}(r_{20}) \leftarrow r_{18}
\end{align*}
\]
Simplifier \((3\text{-}operation\ window)\)

**LLIR Code**

\[
\begin{aligned}
  r_{10} &\leftarrow 2 \\
r_{11} &\leftarrow @y \\
r_{12} &\leftarrow r_0 + r_{11} \\
r_{13} &\leftarrow \text{MEM}(r_{12}) \\
r_{14} &\leftarrow r_{10} \times r_{13} \\
r_{15} &\leftarrow @x \\
r_{16} &\leftarrow r_0 + r_{15} \\
r_{17} &\leftarrow \text{MEM}(r_{16}) \\
r_{18} &\leftarrow r_{17} - r_{14} \\
r_{19} &\leftarrow @w \\
r_{20} &\leftarrow r_0 + r_{19} \\
\text{MEM}(r_{20}) &\leftarrow r_{18}
\end{aligned}
\]

\[
\begin{aligned}
  r_{13} &\leftarrow \text{MEM}(r_0 + @y) \\
r_{14} &\leftarrow 2 \times r_{13} \\
r_{17} &\leftarrow \text{MEM}(r_0 + @x) \\
r_{18} &\leftarrow r_{17} - r_{14} \\
r_{20} &\leftarrow r_0 + @w \\
\text{MEM}(r_{20}) &\leftarrow r_{18}
\end{aligned}
\]
Simplifier (3-operation window)

LLIR Code

\[
\begin{align*}
& r_{10} \leftarrow 2 \\
& r_{11} \leftarrow y \\
& r_{12} \leftarrow r_0 + r_{11} \\
& r_{13} \leftarrow \text{MEM}(r_{12}) \\
& r_{14} \leftarrow r_{10} \times r_{13} \\
& r_{15} \leftarrow x \\
& r_{16} \leftarrow r_0 + r_{15} \\
& r_{17} \leftarrow \text{MEM}(r_{16}) \\
& r_{18} \leftarrow r_{17} - r_{14} \\
& r_{19} \leftarrow w \\
& r_{20} \leftarrow r_0 + r_{19} \\
& \text{MEM}(r_{20}) \leftarrow r_{18}
\end{align*}
\]

\[
\begin{align*}
& r_{13} \leftarrow \text{MEM}(r_0 + y) \\
& r_{14} \leftarrow 2 \times r_{13} \\
& r_{17} \leftarrow \text{MEM}(r_0 + x) \\
& r_{18} \leftarrow r_{17} - r_{14} \\
& \text{MEM}(r_0 + w) \leftarrow r_{18}
\end{align*}
\]
An Example (Con’t)

**LLIR Code**

```
\[\begin{align*}
\text{r}_{10} & \leftarrow 2 \\
\text{r}_{11} & \leftarrow \text{@y} \\
\text{r}_{12} & \leftarrow \text{r}_0 + \text{r}_{11} \\
\text{r}_{13} & \leftarrow \text{MEM(}\text{r}_{12}) \\
\text{r}_{14} & \leftarrow \text{r}_{10} \times \text{r}_{13} \\
\text{r}_{15} & \leftarrow \text{@x} \\
\text{r}_{16} & \leftarrow \text{r}_0 + \text{r}_{15} \\
\text{r}_{17} & \leftarrow \text{MEM(}\text{r}_{16}) \\
\text{r}_{18} & \leftarrow \text{r}_{17} - \text{r}_{14} \\
\text{r}_{19} & \leftarrow \text{@w} \\
\text{r}_{20} & \leftarrow \text{r}_0 + \text{r}_{19} \\
\text{MEM(}\text{r}_{20}) & \leftarrow \text{r}_{18}
\end{align*}\]
```

**Simplify**

```
\[\begin{align*}
\text{r}_{13} & \leftarrow \text{MEM(}\text{r}_0 + \text{@y}) \\
\text{r}_{14} & \leftarrow 2 \times \text{r}_{13} \\
\text{r}_{17} & \leftarrow \text{MEM(}\text{r}_0 + \text{@x}) \\
\text{r}_{18} & \leftarrow \text{r}_{17} - \text{r}_{14} \\
\text{MEM(}\text{r}_0 + \text{@w}) & \leftarrow \text{r}_{18}
\end{align*}\]
```
Making It All Work

- LLIR is largely machine independent
- Target machine described as LLIR → ASM pattern
- Actual pattern matching
  - Use a hand-coded pattern matcher
  - Turn patterns into grammar & use LR parser
- Several important compilers use this technology
- It seems to produce good portable instruction selectors
- Key strength appears to be late low-level optimization
Case Study: Code Selection in Open64
KCC/Open64: Where Instruction Selection Happens?

Front End

Machine Description

Fortran C++ C

Source to IR
Scanner → Parser → RTL → WHIRL

GCC Compile

Very High WHIRL

lowering

High WHIRL

DDG

IPA
• IPL(Pre_IPA)
• IPA_LINK(main_IPA)
• Analysis
• Optimization
W2C/W2F

LNO
• Loop unrolling/
• Loop reversal/Loop fission/Loop fusion
• Loop tiling/Loop peeling…

PREOPT

Middle WHIRL

SSA

lowering

Middle WHIRL

SSA

lowering

Low WHIRL

SSA

lowering

Very Low WHIRL

lowering

CGIR

CFG/DDG

Assembly Code

Back End

WHIRL-to-TOP lowering

2008/4/8

VHO(Very High WHIRL Optimizer)
Standalone Inliner
W2C/W2F

WOPT
• SSAPRE(Partial Redundency Elimination)
• VNFRE(Value Numbering based Full Redundancy Elimination)
RVI-1(Register Variable Identification)

• RVI-2
• IVR(Induction Variable Recognition)

Some peephole optimization

• Cflow(control flow opt), HBS (hyperblock schedule)
• EBO (Extended Block Opt.)
• GCM (Global Code Motion)
• PQS (Predicate Query System)
• SWP, Loop unrolling
• IGLS(Loop unrolling)
• IGLS(Global and Local Instruction Scheduling)
• GRA(Global Register Allocation)
• LRA(Local Register Allocation)
• IGLS(post-pass)

2008/4/8
Code Selection in Open64

- It is done in code generator module
- The input to code selector is tree-structured IR – the lowest WHIRL.
- Input: statements are linked together with list; kids of statement are expressions, organized in tree; compound statement is... -- see next slide
- Code selection order: statement by statement, for each statement’s kids – expr, it is done bottom up.
- CFG is built simultaneously
- Generated code is optimized by EBO
- Retain higher level info
The input of code section

```
PR1
Load c
```

```
store
store
```

```
a = d/c;
If (i < j) {
    a = e/c;  c = 0;
}
```

**The input WHIRL tree to code selection**

- **Source:**
  
  ```
a = d/c;
If (i < j) {
    a = e/c;  c = 0;
}
```

- **A pseudo register PR1**
  - Statements are lined with list
  - Sign-ext higher-order 32-bit (suppose 64 bit machine)
Code selection in dynamic programming flavor

- Given a expression E with kids $E_1, E_2, .. E_n$, the code selection for E is done this way:
  - Conduct code selection for $E_1, E_2, ... E_n$ first, and the result of $E_i$ is saved to temporary value $R_i$.
  - The best possible code selection for E is then done with $R_i$.
- So, generally, it is a traversal the tree top-down, but the code is generated from bottom-up.
The code selection for simple statement: \( a = 0 \)

- The RHS is “\( \text{ldc 0} \)”, (load constant 0). Code selection is applied to this expr first. Some arch has a dedicated register, say r0, holding value 0, if so, return r0 directly. Otherwise, generate instruction “\( \text{mov TN100, 0} \)” and return TN100 as the result for the expr.
- The LHS is variable ‘c’ (LHS need not code selection in this case)
- Then generate instruction “\( \text{store @a, v} \)” for the statement, where v is the result of “\( \text{ldc 0} \)” (the first step).
Optimize with context

- See example \((i < j)\)
- Why “cvtl 32” (basically sign-ext) is necessary
  - Underlying arch is 64 bit, and
  - \(i\) and \(j\) are 32 bit quantum, and
  - “load” is zero-extended, and
  - There is no 4-byte comparison instruction
- So long as one of the above condition is not satisfied, the “cvtl” can be ignored. The selector need some context, basically by looking ahead a little bit.

\[
\begin{align*}
&\text{Load } i \\
&\text{cvtl 32} \\
&\text{Cmp\_lt} \\
&\text{cvtl 32} \\
&\text{Load } j
\end{align*}
\]