RICE

COMP 412
FALL 2010

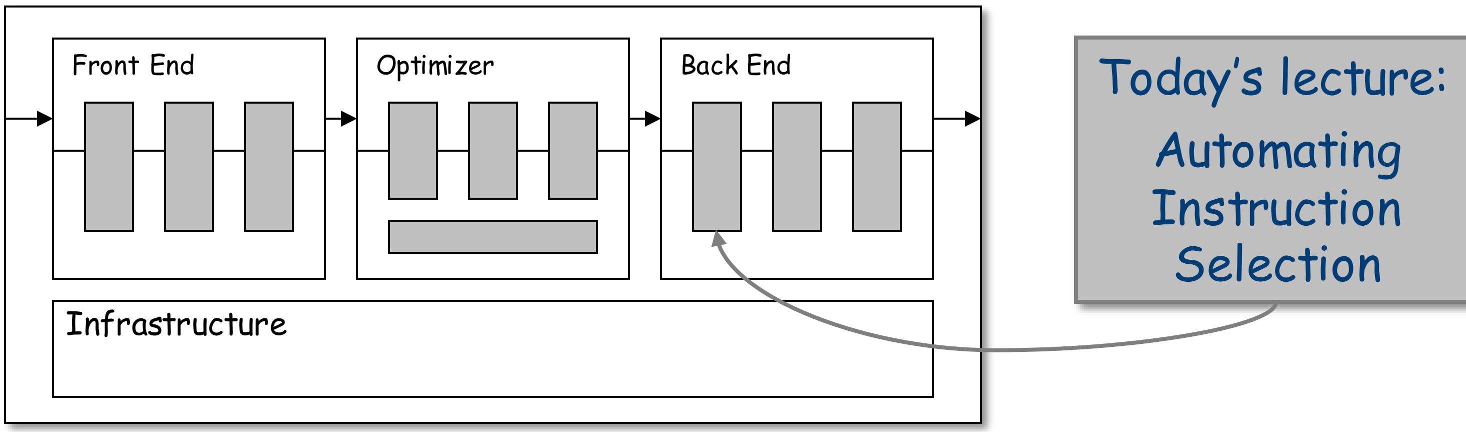
Instruction Selection, Part I

Selection via Peephole Optimization

The Problem

Writing a 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
- Middle is largely hand crafted
- (Parts of) back end can be automated

Definitions

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

The Problem

Modern computers (still) have many ways to do anything

Consider register-to-register copy in ILOC

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

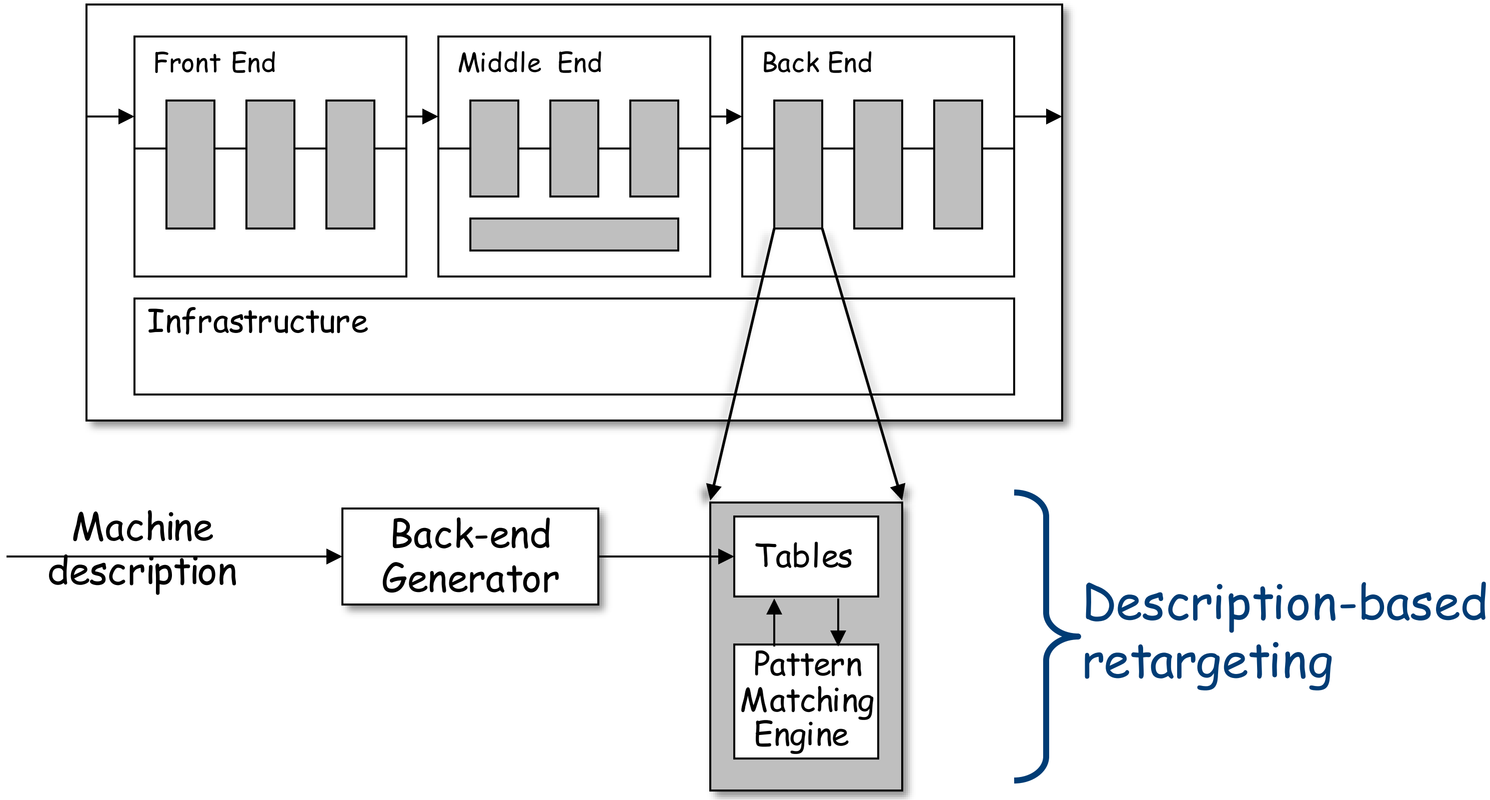
addI r_i, 0 $\Rightarrow$ r_j	subI r_i, 0 $\Rightarrow$ r_j	lshiftI r_i, 0 $\Rightarrow$ r_j
multI r_i, 1 $\Rightarrow$ r_j	divI r_i, 1 $\Rightarrow$ r_j	rshiftI r_i, 0 $\Rightarrow$ r_j
orI r_i, 0 $\Rightarrow$ r_j	xorI r_i, 0 $\Rightarrow$ r_j	*… and others …*

- 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

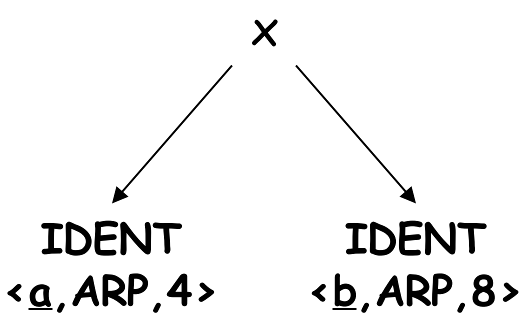
Need pattern matching techniques

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

Our treewalk code generator (Lec. 22) ran quickly

How good was the code?

Tree



Treewalk 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	$\Rightarrow r_7$

The Big Picture

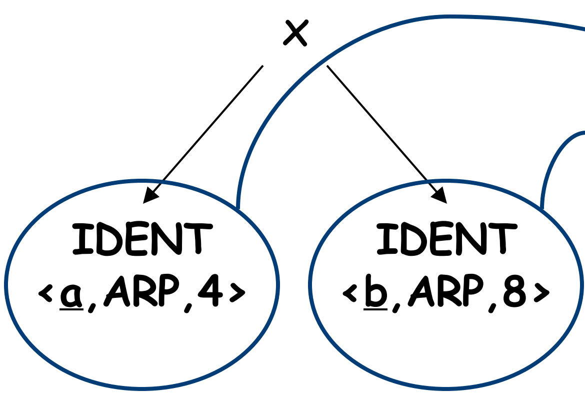
Need pattern matching techniques

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

Our treewalk code generator (Lec. 22) ran quickly

How good was the code?

Tree



Treewalk 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

This inefficiency is easy to fix.
See digression on page 317 of EaC1e.

The Big Picture

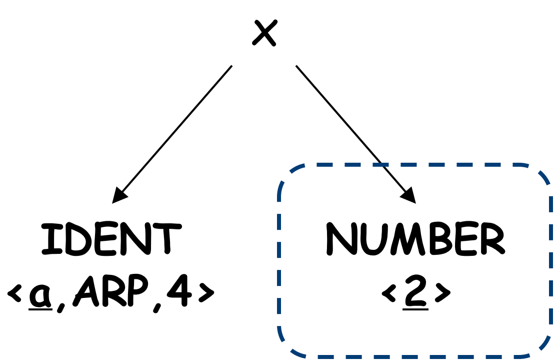
Need pattern matching techniques

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

Our treewalk code generator (Lec. 22) ran quickly

How good was the code?

Tree



Treewalk Code

loadI	4	$\Rightarrow r_5$
loadAO	r_0, r_5	$\Rightarrow r_6$
loadI	2	$\Rightarrow r_7$
mult	r_6, r_7	$\Rightarrow r_8$

Desired Code

loadAI	$r_0, 4$	$\Rightarrow r_5$
multI	$r_5, 2$	$\Rightarrow r_7$

The Big Picture

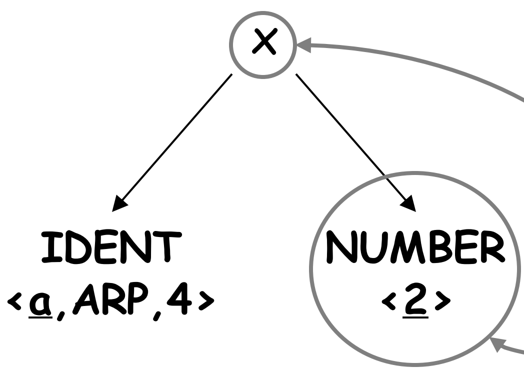
Need pattern matching techniques

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

Our treewalk code generator (Lec. 22) ran quickly

How good was the code?

Tree



Treewalk Code

loadI	4	$\Rightarrow r_5$
loadAO	r_0, r_5	$\Rightarrow r_6$
loadI	2	$\Rightarrow r_7$
mult	r_6, r_7	$\Rightarrow r_8$

Desired Code

loadAI	$r_0, 4$	$\Rightarrow r_5$
multI	$r_5, 2$	$\Rightarrow r_7$

Must use info from both these nodes. This is not a local problem.

The Big Picture

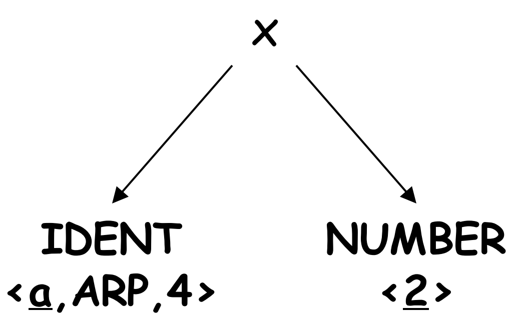
Need pattern matching techniques

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

Our treewalk code generator (Lec. 22) ran quickly

How good was the code?

Tree



Treewalk Code

loadI	4	$\Rightarrow r_5$
loadAO	r_0, r_5	$\Rightarrow r_6$
loadI	2	$\Rightarrow r_7$
mult	r_6, r_7	$\Rightarrow r_8$

Desired Code

loadAI	$r_0, 4$	$\Rightarrow r_5$
add	r_5, r_5	$\Rightarrow r_7$

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

The Big Picture

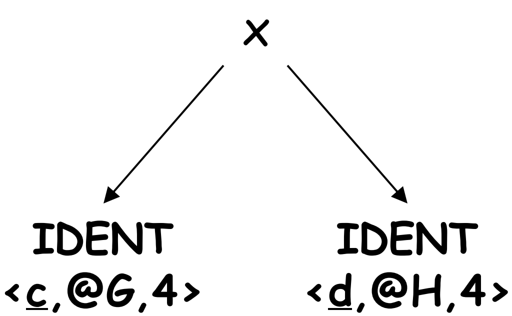
Need pattern matching techniques

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

Our treewalk code generator (Lec. 22) ran quickly

How good was the code?

Tree	Treewalk Code	Desired Code



Treewalk Code

loadI	@G	$\Rightarrow r_5$
loadI	4	$\Rightarrow r_6$
loadAO	r_5,r_6	$\Rightarrow r_7$
loadI	@H	$\Rightarrow r_7$
loadI	4	$\Rightarrow r_8$
loadAO	r_8,r_9	$\Rightarrow r_{10}$
mult	r_7,r_{10}	$\Rightarrow r_{11}$

Desired Code

loadI	4	$\Rightarrow r_5$
loadAI	r_5,@G	$\Rightarrow r_6$
loadAI	r_5,@H	$\Rightarrow r_7$
mult	r_6,r_7	$\Rightarrow r_8$

The Big Picture

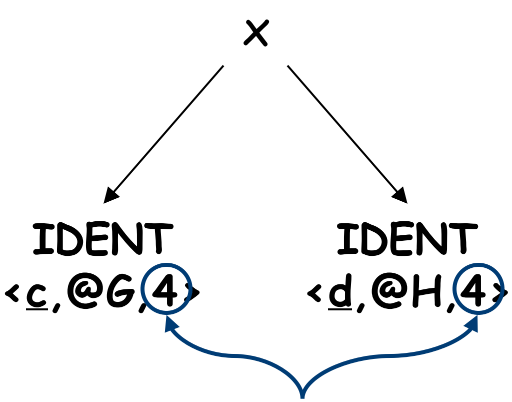
Need pattern matching techniques

- 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 ?

Tree



Common offset

Treewalk Code

loadI	@G	$\Rightarrow r_5$
loadI	4	$\Rightarrow r_6$
loadAO	r_5, r_6	$\Rightarrow r_7$
loadI	@H	$\Rightarrow r_7$
loadI	4	$\Rightarrow r_8$
loadAO	r_8, r_9	$\Rightarrow r_{10}$
mult	r_7, r_{10}	$\Rightarrow r_{11}$

Desired Code

loadI	4	$\Rightarrow r_5$
loadAI	r_5,@G	$\Rightarrow r_6$
loadAI	r_5,@H	$\Rightarrow r_7$
mult	r_6, r_7	$\Rightarrow r_8$

Another nonlocal problem

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 $r_1 \Rightarrow r_0,8$	storeAI $r_1 \Rightarrow r_0,8$
loadAI $r_0,8 \Rightarrow r_{15}$	i2i $r_1 \Rightarrow r_{15}$

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

Original code	Improved code
addI $r_2, 0 \Rightarrow r_7$ mult $r_4, r_7 \Rightarrow r_{10}$	mult $r_4, r_2 \Rightarrow r_{10}$
multI $r_5, 2 \Rightarrow r_7$	add $r_2, r_2 \Rightarrow r_7$

See Table on p 401 of EaC (§ 8.3)

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	Improved code
jumpI $\rightarrow L_{10}$ L_{10}: jumpI $\rightarrow L_{11}$	L_{10}: jumpI $\rightarrow L_{11}$

Must be within the window

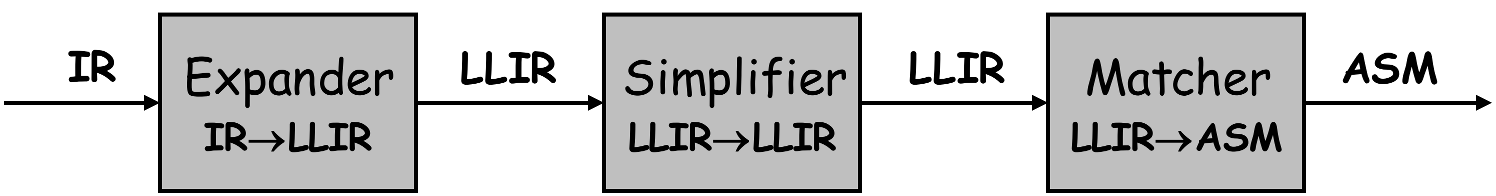
Peephole Matching

Implementing it

- Early systems used limited set of hand-coded patterns
- Window size ensured quick processing $O(n^2) \Rightarrow O(n)$

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

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 what is 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 operation with a list of useless values
- What about non-local effects?
  - Most useless effects are local — defined & used in same block
  - It can be conservative & assume LIVE until proven dead

As in Lab 1

ASM		LLIR		LLIR		ASM
mult $r_5,r_9 \Rightarrow r_{12}$ add $r_{12},r_{17} \Rightarrow r_{13}$	expand $\Rightarrow$	$r_{12} \Leftarrow r_5 * r_9$ $cc \Leftarrow f(r_5*r_9)$ $r_{13} \Leftarrow r_{12} + r_{17}$ $cc \Leftarrow f(r_{12}+r_{17})$	simplify $\Rightarrow$	$r_{12} \Leftarrow r_5 * r_9$ $\cancel{cc \Leftarrow f(r_5*r_9)}$ $r_{13} \Leftarrow r_{12} + r_{17}$ $cc \Leftarrow f(r_{12}+r_{17})$	match $\Rightarrow$	madd $r_5,r_9,r_{17} \Rightarrow r_{13}$

This effect would prevent multiply-add from matching

Example

$x - 2 \times y$ *becomes*

Original IR Code

OP	Arg_1	Arg_2	Result
mult	2	y	t_1
sub	x	t_1	w

Symbolic names for memory-bound variables

Example

x - 2 * y *becomes*

Original IR Code

OP	Arg_1	Arg_2	Result
mult	2	y	t_1
sub	x	t_1	w

Symbolic names for memory-bound variables

Expand

LLIR Code

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

This version of the example assumes that x, y, and w are all stored in the AR.

The example in § 11 of EaC assumes that x is a call-by-reference formal and y is a global. The results are different.

Example

LLIR Code

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

Simplify $\Rightarrow$

LLIR Code

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

Example

LLIR Code	Match	ILOC Code
$r_{13} \leftarrow$ MEM(r_0+ @y)		loadAI r_0,@y $\rightarrow r_{13}$
$r_{14} \leftarrow 2 \times r_{13}$		multI $2 \times r_{13} \rightarrow r_{14}$
$r_{17} \leftarrow$ MEM(r_0 + @x)		loadAI r_0,@x $\rightarrow r_{17}$
$r_{18} \leftarrow r_{17} - r_{14}$		sub $r_{17} - r_{14} \rightarrow r_{18}$
MEM(r_0 + @w) $\leftarrow r_{18}$		storeAI $r_{18} \rightarrow r_0$,@w

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

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

$r_{12} \leftarrow r_0 + r_{11}$

$r_{13} \leftarrow MEM(r_{12})$

$r_{14} \leftarrow r_{10} \times r_{13}$

$r_{15} \leftarrow @x$

$r_{16} \leftarrow r_0 + r_{15}$

$r_{17} \leftarrow MEM(r_{16})$

$r_{18} \leftarrow r_{17} - r_{14}$

$r_{19} \leftarrow @w$

$r_{20} \leftarrow r_0 + r_{19}$

$MEM(r_{20}) \leftarrow r_{18}$

Steps of the Simplifier (*3-operation window*)

LLIR Code

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

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

$r_{12} \leftarrow r_0 + r_{11}$

$r_{13} \leftarrow MEM(r_{12})$

$r_{14} \leftarrow r_{10} \times r_{13}$

$r_{15} \leftarrow @x$

$r_{16} \leftarrow r_0 + r_{15}$

$r_{17} \leftarrow MEM(r_{16})$

$r_{18} \leftarrow r_{17} - r_{14}$

$r_{19} \leftarrow @w$

$r_{20} \leftarrow r_0 + r_{19}$

$MEM(r_{20}) \leftarrow r_{18}$

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$

→

$r_{10} \leftarrow 2$
$r_{12} \leftarrow r_0 + @y$
$r_{13} \leftarrow MEM(r_{12})$

Steps of the Simplifier (*3-operation window*)

LLIR Code

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

$r_{10} \leftarrow 2$
$r_{12} \leftarrow r_0 + @y$
$r_{13} \leftarrow MEM(r_{12})$

$\Rightarrow$

$r_{10} \leftarrow 2$
$r_{13} \leftarrow MEM(r_0 + @y)$
$r_{14} \leftarrow r_{10} \times r_{13}$

Steps of the Simplifier (*3-operation window*)

LLIR Code

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

$$\boxed{\begin{array}{l} r_{10} \leftarrow 2 \\ r_{13} \leftarrow MEM(r_0 + @y) \\ r_{14} \leftarrow r_{10} \times r_{13} \end{array}} \Rightarrow \boxed{\begin{array}{l} r_{13} \leftarrow MEM(r_0 + @y) \\ r_{14} \leftarrow 2 \times r_{13} \\ r_{15} \leftarrow @x \end{array}}$$

Folding 2 into computation of r_{14} made the 1st op *dead*.

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

$r_{12} \leftarrow r_0 + r_{11}$

$r_{13} \leftarrow MEM(r_{12})$

$r_{14} \leftarrow r_{10} \times r_{13}$

$r_{15} \leftarrow @x$

$r_{16} \leftarrow r_0 + r_{15}$

$r_{17} \leftarrow MEM(r_{16})$

$r_{18} \leftarrow r_{17} - r_{14}$

$r_{19} \leftarrow @w$

$r_{20} \leftarrow r_0 + r_{19}$

$MEM(r_{20}) \leftarrow r_{18}$

$r_{13} \leftarrow MEM(r_0 + @y)$
$r_{14} \leftarrow 2 \times r_{13}$
$r_{15} \leftarrow @x$

$\Rightarrow$

$r_{14} \leftarrow 2 \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$

Simplifier emits ops that are *live* when they roll out of the window.

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

$r_{12} \leftarrow r_0 + r_{11}$

$r_{13} \leftarrow MEM(r_{12})$

$r_{14} \leftarrow r_{10} \times r_{13}$

$r_{15} \leftarrow @x$

$r_{16} \leftarrow r_0 + r_{15}$

$r_{17} \leftarrow MEM(r_{16})$

$r_{18} \leftarrow r_{17} - r_{14}$

$r_{19} \leftarrow @w$

$r_{20} \leftarrow r_0 + r_{19}$

$MEM(r_{20}) \leftarrow r_{18}$

$r_{14} \leftarrow 2 \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$

$\Rightarrow$

$r_{14} \leftarrow 2 \times r_{13}$
$r_{16} \leftarrow r_0 + @x$
$r_{17} \leftarrow MEM(r_{16})$

Steps of the Simplifier (*3-operation window*)

LLIR Code

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

$r_{14} \leftarrow 2 \times r_{13}$
$r_{16} \leftarrow r_0 + @x$
$r_{17} \leftarrow MEM(r_{16})$

$\Rightarrow$

$r_{14} \leftarrow 2 \times r_{13}$
$r_{17} \leftarrow MEM(r_0+@x)$
$r_{18} \leftarrow r_{17} - r_{14}$

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

$r_{12} \leftarrow r_0 + r_{11}$

$r_{13} \leftarrow \text{MEM}(r_{12})$

$r_{14} \leftarrow r_{10} \times r_{13}$

$r_{15} \leftarrow @x$

$r_{16} \leftarrow r_0 + r_{15}$

$r_{17} \leftarrow \text{MEM}(r_{16})$

$r_{18} \leftarrow r_{17} - r_{14}$

$r_{19} \leftarrow @w$

$r_{20} \leftarrow r_0 + r_{19}$

$\text{MEM}(r_{20}) \leftarrow r_{18}$

$r_{14} \leftarrow 2 \times r_{13}$
$r_{17} \leftarrow \text{MEM}(r_0+@x)$
$r_{18} \leftarrow r_{17} - r_{14}$

→

$r_{17} \leftarrow \text{MEM}(r_0+@x)$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

$r_{12} \leftarrow r_0 + r_{11}$

$r_{13} \leftarrow MEM(r_{12})$

$r_{14} \leftarrow r_{10} \times r_{13}$

$r_{15} \leftarrow @x$

$r_{16} \leftarrow r_0 + r_{15}$

$r_{17} \leftarrow MEM(r_{16})$

$r_{18} \leftarrow r_{17} - r_{14}$

$r_{19} \leftarrow @w$

$r_{20} \leftarrow r_0 + r_{19}$

$MEM(r_{20}) \leftarrow r_{18}$

$r_{17} \leftarrow MEM(r_0+@x)$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$

$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

$r_{12} \leftarrow r_0 + r_{11}$

$r_{13} \leftarrow MEM(r_{12})$

$r_{14} \leftarrow r_{10} \times r_{13}$

$r_{15} \leftarrow @x$

$r_{16} \leftarrow r_0 + r_{15}$

$r_{17} \leftarrow MEM(r_{16})$

$r_{18} \leftarrow r_{17} - r_{14}$

$r_{19} \leftarrow @w$

$r_{20} \leftarrow r_0 + r_{19}$

$MEM(r_{20}) \leftarrow r_{18}$

$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$

$\Rightarrow$

$r_{18} \leftarrow r_{17} - r_{14}$
$r_{20} \leftarrow r_0 + @w$
$MEM(r_{20}) \leftarrow r_{18}$

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

$r_{12} \leftarrow r_0 + r_{11}$

$r_{13} \leftarrow MEM(r_{12})$

$r_{14} \leftarrow r_{10} \times r_{13}$

$r_{15} \leftarrow @x$

$r_{16} \leftarrow r_0 + r_{15}$

$r_{17} \leftarrow MEM(r_{16})$

$r_{18} \leftarrow r_{17} - r_{14}$

$r_{19} \leftarrow @w$

$r_{20} \leftarrow r_0 + r_{19}$

$MEM(r_{20}) \leftarrow r_{18}$

$r_{18} \leftarrow r_{17} - r_{14}$
$r_{20} \leftarrow r_0 + @w$
$MEM(r_{20}) \leftarrow r_{18}$

$\Rightarrow$

$r_{18} \leftarrow r_{17} - r_{14}$
$MEM(r_0 + @w) \leftarrow r_{18}$

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

$r_{12} \leftarrow r_0 + r_{11}$

$r_{13} \leftarrow MEM(r_{12})$

$r_{14} \leftarrow r_{10} \times r_{13}$

$r_{15} \leftarrow @x$

$r_{16} \leftarrow r_0 + r_{15}$

$r_{17} \leftarrow MEM(r_{16})$

$r_{18} \leftarrow r_{17} - r_{14}$

$r_{19} \leftarrow @w$

$r_{20} \leftarrow r_0 + r_{19}$

$MEM(r_{20}) \leftarrow r_{18}$

$r_{18} \leftarrow r_{17} - r_{14}$
$r_{20} \leftarrow r_0 + @w$
$MEM(r_{20}) \leftarrow r_{18}$

$\Rightarrow$

$r_{18} \leftarrow r_{17} - r_{14}$
$MEM(r_0 + @w) \leftarrow r_{18}$

Example

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

$r_{12} \leftarrow r_0 + r_{11}$

$r_{13} \leftarrow MEM(r_{12})$

$r_{14} \leftarrow r_{10} \times r_{13}$

$r_{15} \leftarrow @x$

$r_{16} \leftarrow r_0 + r_{15}$

$r_{17} \leftarrow MEM(r_{16})$

$r_{18} \leftarrow r_{17} - r_{14}$

$r_{19} \leftarrow @w$

$r_{20} \leftarrow r_0 + r_{19}$

$MEM(r_{20}) \leftarrow r_{18}$

Simplify

LLIR Code

$r_{13} \leftarrow MEM(r_0 + @y)$

$r_{14} \leftarrow 2 \times r_{13}$

$r_{17} \leftarrow MEM(r_0 + @x)$

$r_{18} \leftarrow r_{17} - r_{14}$

$MEM(r_0 + @w) \leftarrow r_{18}$

Making It All Work

Details

- LLIR is largely machine independent (RTL)
  - Some compilers use LLIR as one of their IRs
  - Eliminates the Expander
- Target machine described as LLIR $\rightarrow$ ASM pattern
- 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.