CS 4240: Compilers

Lecture 10: Instruction Selection

Instructor: Vivek Sarkar (vsarkar@gatech.edu)

February 13, 2019

ANNOUNCEMENTS & REMINDERS

- Project 1 is due by 11:59pm TODAY on Canvas
  - Must be submitted as zip file including instructions on how to build and run your project
  - 100 points total, with an extra credit option for 15 points
    - Extra credit relates to use of copy propagation
  - 5% of course grade

- MIDTERM EXAM: Wednesday, March 13, 4:30pm - 5:45pm

- FINAL EXAM: Wednesday, May 1, 2:40 pm - 5:30 pm

- Acknowledgements: today's lecture includes slides from Fall 2018 offering of CS 4240 taught by Prof. Santosh Pande, as well as slides from textbook authors as usual

Don't forget the design document in your Project 1 submission!

1.1 Design of your Optimizer (30 points)

With your submission, please include a design document called 'design.pdf' in the zip file mentioned in Section 1.2. This document should briefly describe the following:

- » High-level architecture of your optimizer, including the analysis and optimization algorithm(s) implemented, and why you chose that approach.
- » Low-level design decisions you made in selection of implementation language, and their rationale.
- » Software engineering challenges and issues that arose and how you resolved them.
- » **Any known outstanding bugs or deficiencies that you were unable to resolve before the project submission.**
- » Build and usage instructions for your optimizer.
- » A summary of the test results for the public test cases (see Section 1.2).

Worksheet 9 solution: Summarize the effect of the following MIPS instructions (from lecture given on 2/11/2019)

```
la $a0, array        // Load address of "array" data segment into $a0
addi $t0, $a0, 4     // $t0 := $a0 + 4
lw $t1, 0($t0)       // Load contents of address $t0 into $t1
lw $t2, 4($t0)       // Load contents of address ($t0 + 4) into $t2
sw $t2, 0($t0)       // Store contents of $t2 into address
sw $t1, 4($t0)       // Store contents of $t1 into address ($t0 + 4)

array:
.word 5, 4, 3, 2, 1
```

```
la $a0, array
addi $t0, $a0, 4
lw $t1, 0($t0)
lw $t2, 4($t0)
sw $t2, 0($t0)
sw $t1, 4($t0)

array:
.word 5, 4, 3, 2, 1
```

Assuming

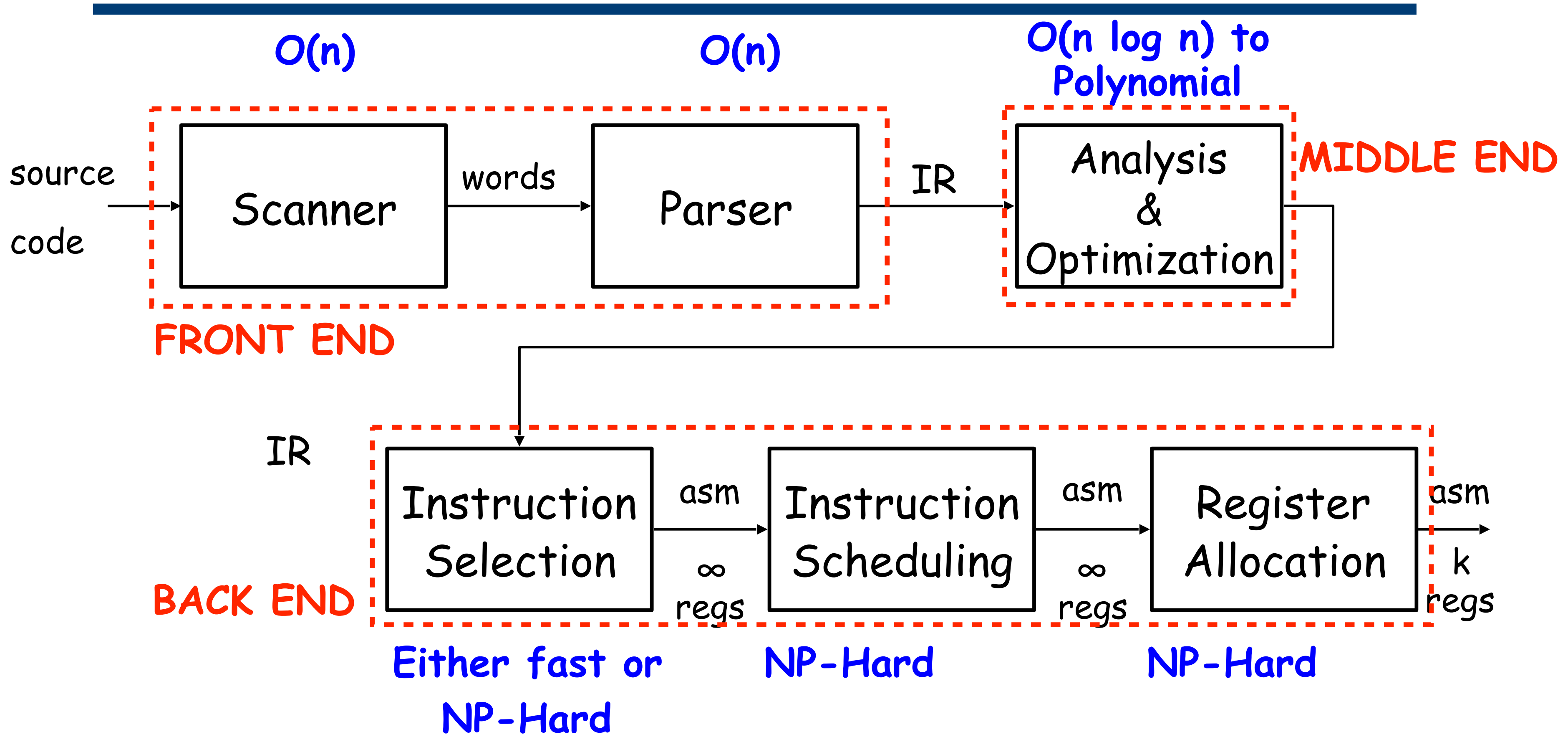
Little Endian

WORD					
HALF		HALF			
BYTE	BYTE	BYTE	BYTE	Starting address	Word Value
3	2	1	0		
0x00	0x00	0x01	0x01	array	5
0x00	0x00	0x01	0x00	array+4	4
0x00	0x00	0x00	0x11	array+8	3
0x00	0x00	0x00	0x10	array+12	2
0x00	0x00	0x00	0x01	array+16	1

EFFECT of the given MIPS code :

Contents of second and third elements of the word array are switched.

Structure of a Compiler



A compiler is a lot of fast stuff followed by some hard problems

- The hard stuff is mostly in the middle end and back end
- For most CPU processors, the register allocation & instruction scheduling phases have the biggest impact in the back end
  - Phase ordering and combining of register allocation and instruction scheduling can be even more challenging!

Definitions

Instruction selection

- Mapping IR into assembly code for a target processor, e.g., MIPS
- 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)
- Impacts the demand for registers

These 3 problems are tightly coupled.

Register allocation

- Deciding which values will reside in registers
- Reduces load, store, and copy statements in the IR
- Can create "false" dependences when two variables are mapped to the same register

The Big Picture

How hard are these problems?

Instruction selection

- » Can make locally optimal choices, with automated tool
- » Global optimality is NP-Hard

Instruction scheduling

- » Single basic block $\Rightarrow$ heuristics work quickly, and optimal solutions are possible in limited cases (0/1 latencies)
- » General problem, with control flow $\Rightarrow$ NP-Hard

Register allocation

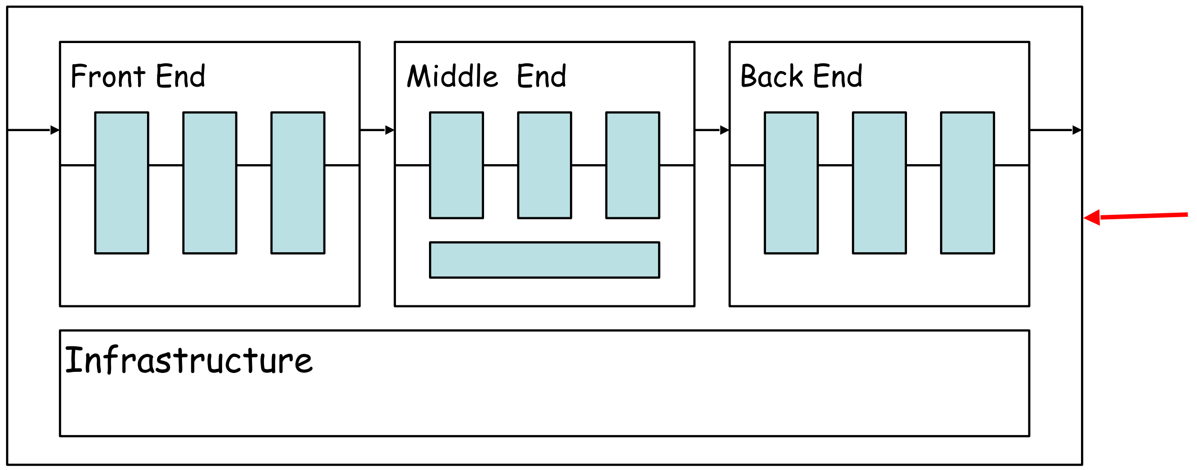
- » Special cases can be solved optimally in linear time
- » Whole procedure is NP-Hard

The Problem

Writing a compiler is a lot of work

» Would like to reuse components whenever possible

» Would like to automate construction of components



Today's lecture:

Automating Instruction Selection

» Front end construction is largely automated with scanner generators and parser generators

» Middle end is largely hand crafted, though dataflow analysis frameworks offer opportunities for code reuse

» (Parts of) back end can be automated

The Problem

Modern computers (still) have many ways to do anything

Consider register-to-register copy in ILOC IR from textbook

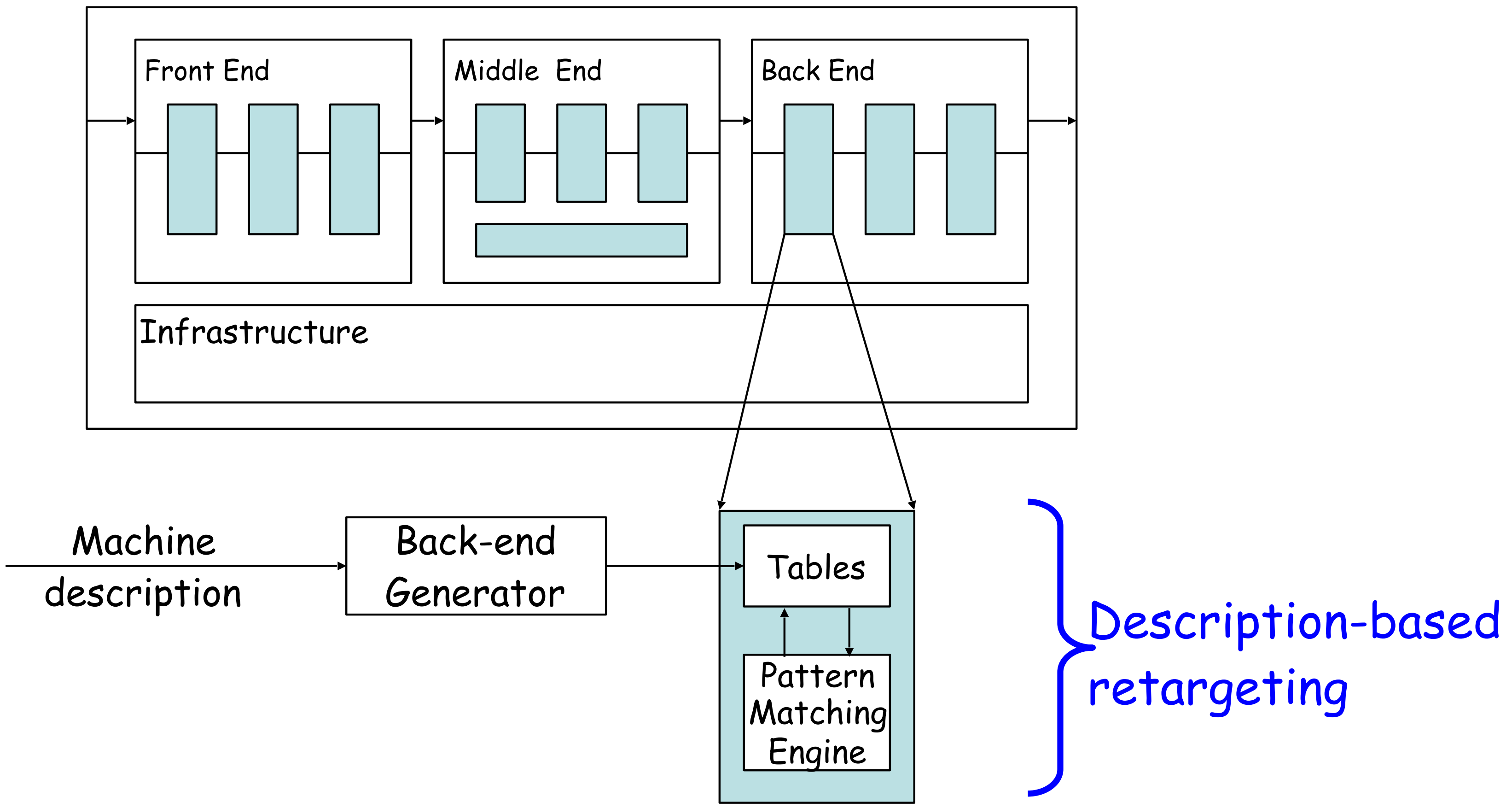
- Obvious operation is `i2i` $r_i \Rightarrow r_j$
- Many others exist

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 (e.g., busy functional units)

The Goal

Want to automate generation of instruction selectors



Machine description should also help with scheduling & allocation

The Back End

- Essential tasks:
- Instruction selection
  - Map low-level IR to actual machine instructions
  - Not necessarily 1-1 mapping
  - CISC architectures, addressing modes
- Register allocation
  - Low-level IR assumes unlimited registers
  - Map to actual resources of machines
  - Goal: maximize use of registers

Instruction Selection

- Low-level IR different from machine Instruction Set Architecture (ISA)
    - Why?
    - Allow different back ends
    - Abstraction – to make optimization easier

- Differences between IR and ISA
    - IR: simple, uniform set of operations
    - ISA: many specialized instructions

- Often a single instruction does the work of multiple IR operations

Example

Consider an example IR generated from the source code statement, a[i+1] = b[j]:

	IR
Address of b[j]:	`t1 = j*4`
	`t2 = b+t1`
Load value b[j]:	`t3 = *t2`
Address of a[i+1]:	`t4 = i+1`
	`t5 = t4*4`
	`t6 = a+t5`
Store into a[i+1]:	`*t6 = t3`

Instruction Selection

One approach

- Map each IR operation to an instruction (or set of)
- May need to include memory operations

```
x = y + z;
```

```
mov y, r1
mov z, r2
add r2, r1
mov r1, x
```

Problem: inefficient use of ISA

Possible Translation (Unoptimized)

	IR	***Assembly***

Address of b[j]:	`t1 = j*4`
	`t2 = b+t1`
		`muli 4, rj`
		`add rj, rb`

Load value b[j]:	`t3 = *t2`	`load rb, r1`

Address of a[i+1]:	`t4 = i+1`
	`t5 = t4*4`
	`t6 = a+t5`
		`addi 1, ri`
		`muli 4, ri`
		`add ri, ra`

Store into a[i+1]:	`*t6 = t3`	`store r1, ra`

Another Translation for a CISC processor (unlike MIPS)

	IR	*Assembly*
Address of b[j]:	`t1 = j*4`	`muli 4, rj`
	`t2 = b+t1`	`add rj, rb`
(no load)	`t3 = *t2`	
Address of a[i+1]:	`t4 = i+1`	`addi 1, ri`
	`t5 = t4*4`	`muli 4, ri`
	`t6 = a+t5`	`add ri, ra`
Store into a[i+1]:	`*t6 = t3`	`movem rb, ra`

Direct memory-to-memory operation

Yet Another Translation

	IR	*Assembly*
Index of b[j]:	`t1 = j*4`	`muli 4, rj`
	`t2 = b+t1`	
(no load)	`t3 = *t2`	
Address of a[i+1]:	`t4 = i+1`	`addi 1, ri`
	`t5 = t4*4`	`muli 4, ri`
	`t6 = a+t5`	`add ri, ra`
Store into a[i+1]:	`*t6 = t3`	`movex rj,rb,ra`

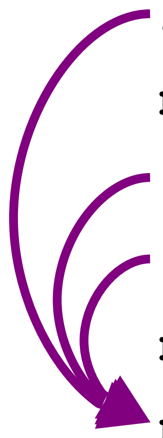
Compute the address of b[j] in the memory move operation

`movex rj, rb, ra` ***ra ← *(rj + rb)**

Different Translations

- Why is the last translation preferable?
  - Fewer instructions
  - Instructions have different costs
    - Space cost: size of each instruction
    - Time cost: number of cycles to complete

- Example



`add r2, r1`	**cost = 1 cycle**
`muli c, r1`	**cost = 10 cycles**
`load r2, r1`	**cost = 3 cycles**
`store r2, r1`	**cost = 3 cycles**
`movem r2, r1`	**cost = 4 cycles**
`movex r3, r2, r1`	**cost = 5 cycles**

The Big Picture

Need pattern matching techniques

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

Linear IR with three-address code 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 or **peephole matching (today's lecture)**

Tree-oriented IR suggests **pattern matching on trees (next lecture)**

- 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
- It is possible to extract trees from Linear IR as well

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

  » Expressed using textbook's IR

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

» 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 $r_2,r_2 \Rightarrow r_7$

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

```
       jumpI  → L10
L10:   jumpI  → L11
```

Must be within the window

Improved code

```
L10:   jumpI  → L11
```

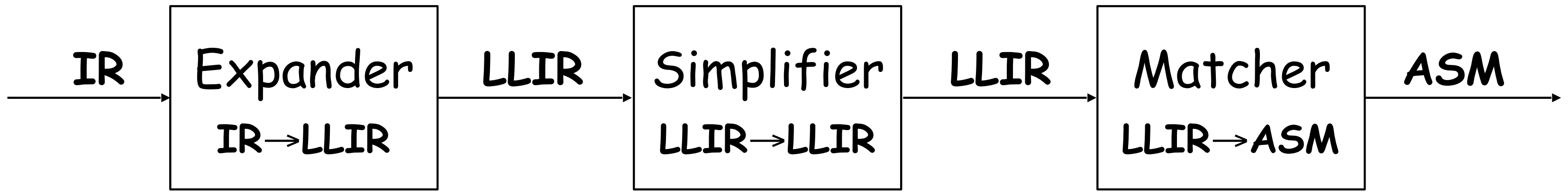
Peephole Matching

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

- » Break problem into three tasks

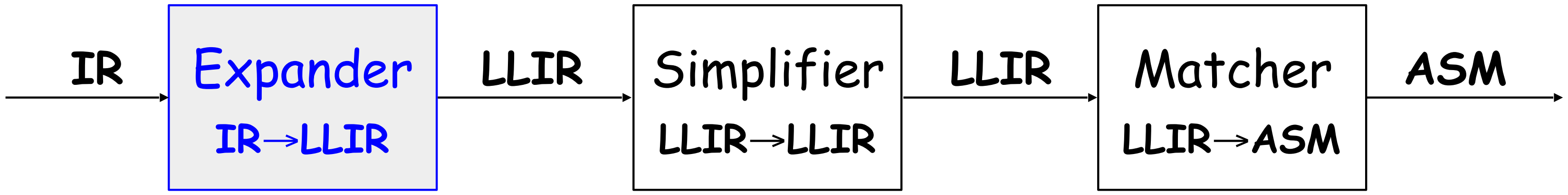


- » Apply symbolic interpretation & simplification systematically

Peephole Matching

Expander

- Turns IR code into a low-level IR (LLIR) such as RTL
  - Not to be confused by LLVM IR …
- Operation-by-operation, template-driven rewriting
- LLIR form includes all direct effects
- Significant, albeit constant, expansion of size



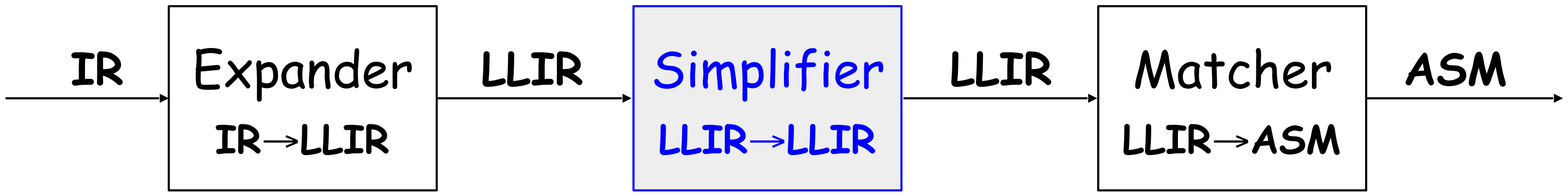
Peephole Matching

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



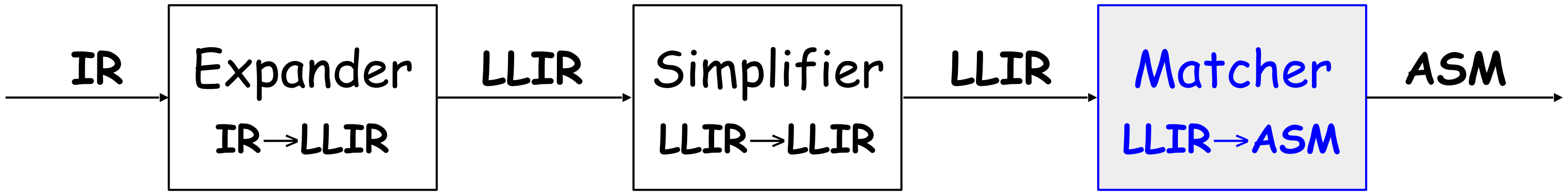
» This is the heart of the peephole system

– 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
- Generates the assembly code output



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

Example (after Expander phase)

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

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

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

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

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

$r_{15} \leftarrow @x$

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

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

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

$r_{19} \leftarrow @w$

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

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

Example (after Simplifier phase)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

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

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

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

$r_{15} \leftarrow @x$

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

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

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

$r_{19} \leftarrow @w$

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

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

$\xrightarrow{\text{Simplify}}$

LLIR Code

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

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

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

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

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

Example (after Matcher phase for Instruction Selection)

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

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

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

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

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

$r_{15} \leftarrow @x$

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

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

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

$r_{19} \leftarrow @w$

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

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

Steps of the Simplifier (3-operation window)

LLIR Code

$$\left.\begin{array}{l} r_{10} \leftarrow 2 \\ r_{11} \leftarrow @y \\ r_{12} \leftarrow r_0 + r_{11} \end{array}\right\} \quad \boxed{\begin{array}{l} r_{10} \leftarrow 2 \\ r_{11} \leftarrow @y \\ r_{12} \leftarrow r_0 + r_{11} \end{array}}$$

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

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

$r_{15} \leftarrow @x$

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

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

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

$r_{19} \leftarrow @w$

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

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

Steps of the Simplifier (3-operation window)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

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

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

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

$r_{15} \leftarrow @x$

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

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

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

$r_{19} \leftarrow @w$

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

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

$$\boxed{\begin{array}{l} r_{10} \leftarrow 2 \\ r_{11} \leftarrow @y \\ r_{12} \leftarrow r_0 + r_{11} \end{array}} \Rightarrow \boxed{\begin{array}{l} r_{10} \leftarrow 2 \\ r_{12} \leftarrow r_0 + @y \\ r_{13} \leftarrow \text{MEM}(r_{12}) \end{array}}$$

Steps of the Simplifier (3-operation window)

LLIR Code

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

$$
\boxed{\begin{array}{l} r_{10} \leftarrow 2 \\ r_{12} \leftarrow r_0 + @y \\ r_{13} \leftarrow \mathrm{MEM}(r_{12}) \end{array}}
\Longrightarrow
\boxed{\begin{array}{l} r_{10} \leftarrow 2 \\ r_{13} \leftarrow \mathrm{MEM}(r_0 + @y) \\ r_{14} \leftarrow r_{10} \times r_{13} \end{array}}
$$

Steps of the Simplifier (3-operation window)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

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

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

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

$r_{15} \leftarrow @x$

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

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

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

$r_{19} \leftarrow @w$

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

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

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

Steps of the Simplifier (3-operation window)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

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

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

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

$r_{15} \leftarrow @x$

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

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

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

$r_{19} \leftarrow @w$

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

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

1st op it has rolled out of window

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

$\Rightarrow$

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

Steps of the Simplifier (3-operation window)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

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

$$\left.\begin{array}{l} r_{13} \leftarrow \text{MEM}(r_{12}) \\ r_{14} \leftarrow r_{10} \times r_{13} \\ r_{15} \leftarrow @x \end{array}\right\} \boxed{\begin{array}{l} r_{14} \leftarrow 2 \times r_{13} \\ r_{15} \leftarrow @x \\ r_{16} \leftarrow r_0 + r_{15} \end{array}} \Rightarrow \boxed{\begin{array}{l} r_{14} \leftarrow 2 \times r_{13} \\ r_{16} \leftarrow r_0 + @x \\ r_{17} \leftarrow \text{MEM}(r_{16}) \end{array}}$$

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

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

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

$r_{19} \leftarrow @w$

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

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

Steps of the Simplifier (3-operation window)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

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

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

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

$r_{15} \leftarrow @x$

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

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

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

$r_{19} \leftarrow @w$

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

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

$$\boxed{\begin{array}{l} r_{14} \leftarrow 2 \times r_{13} \\ r_{16} \leftarrow r_0 + @x \\ r_{17} \leftarrow \text{MEM}(r_{16}) \end{array}} \Rightarrow \boxed{\begin{array}{l} r_{14} \leftarrow 2 \times r_{13} \\ r_{17} \leftarrow \text{MEM}(r_0+@x) \\ r_{18} \leftarrow r_{17} - r_{14} \end{array}}$$

Steps of the Simplifier (3-operation window)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

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

$$\left.\begin{array}{l} r_{13} \leftarrow \text{MEM}(r_{12}) \\ r_{14} \leftarrow r_{10} \times r_{13} \\ r_{15} \leftarrow @x \\ r_{16} \leftarrow r_0 + r_{15} \end{array}\right\} \boxed{\begin{array}{l} \color{red}{r_{14} \leftarrow 2 \times r_{13}} \\ r_{17} \leftarrow \text{MEM}(r_0+@x) \\ r_{18} \leftarrow r_{17} - r_{14} \end{array}} \Rightarrow \boxed{\begin{array}{l} r_{17} \leftarrow \text{MEM}(r_0+@x) \\ r_{18} \leftarrow r_{17} - r_{14} \\ r_{19} \leftarrow @w \end{array}}$$

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

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

$r_{19} \leftarrow @w$

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

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

Steps of the Simplifier (3-operation window)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

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

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

$$\left.\begin{array}{l} r_{14} \leftarrow r_{10} \times r_{13} \\ r_{15} \leftarrow @x \\ r_{16} \leftarrow r_0 + r_{15} \\ r_{17} \leftarrow MEM(r_{16}) \end{array}\right\} \boxed{\begin{array}{l} r_{17} \leftarrow MEM(r_0+@x) \\ r_{18} \leftarrow r_{17} - r_{14} \\ r_{19} \leftarrow @w \end{array}} \Rightarrow \boxed{\begin{array}{l} r_{18} \leftarrow r_{17} - r_{14} \\ r_{19} \leftarrow @w \\ r_{20} \leftarrow r_0 + r_{19} \end{array}}$$

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

$r_{19} \leftarrow @w$

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

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

Steps of the Simplifier (3-operation window)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

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

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

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

$r_{15} \leftarrow @x$

$$\left.\begin{array}{l} r_{16} \leftarrow r_0 + r_{15} \\ r_{17} \leftarrow \mathrm{MEM}(r_{16}) \\ r_{18} \leftarrow r_{17} - r_{14} \end{array}\right\} \boxed{\begin{array}{l} r_{18} \leftarrow r_{17} - r_{14} \\ r_{19} \leftarrow @w \\ r_{20} \leftarrow r_0 + r_{19} \end{array}} \Rightarrow \boxed{\begin{array}{l} r_{18} \leftarrow r_{17} - r_{14} \\ r_{20} \leftarrow r_0 + @w \\ \mathrm{MEM}(r_{20}) \leftarrow r_{18} \end{array}}$$

$r_{19} \leftarrow @w$

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

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

Steps of the Simplifier (3-operation window)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

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

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

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

$r_{15} \leftarrow @x$

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

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

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

$r_{19} \leftarrow @w$

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

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

$$\left.\begin{array}{l} r_{16} \leftarrow r_0 + r_{15} \\ r_{17} \leftarrow \mathrm{MEM}(r_{16}) \\ r_{18} \leftarrow r_{17} - r_{14} \end{array}\right\} \boxed{\begin{array}{l} r_{18} \leftarrow r_{17} - r_{14} \\ r_{20} \leftarrow r_0 + @w \\ \mathrm{MEM}(r_{20}) \leftarrow r_{18} \end{array}} \Rightarrow \boxed{\begin{array}{l} r_{18} \leftarrow r_{17} - r_{14} \\ \mathrm{MEM}(r_0 + @w) \leftarrow r_{18} \end{array}}$$

Steps of the Simplifier (3-operation window)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

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

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

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

$r_{15} \leftarrow @x$

$$\left.\begin{array}{l} r_{16} \leftarrow r_0 + r_{15} \\ r_{17} \leftarrow \text{MEM}(r_{16}) \\ r_{18} \leftarrow r_{17} - r_{14} \end{array}\right\} \boxed{\begin{array}{l} r_{18} \leftarrow r_{17} - r_{14} \\ r_{20} \leftarrow r_0 + @w \\ \text{MEM}(r_{20}) \leftarrow r_{18} \end{array}} \Rightarrow \boxed{\begin{array}{l} r_{18} \leftarrow r_{17} - r_{14} \\ \text{MEM}(r_0 + @w) \leftarrow r_{18} \end{array}}$$

$r_{19} \leftarrow @w$

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

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

Example

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

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

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

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

$r_{15} \leftarrow @x$

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

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

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

$r_{19} \leftarrow @w$

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

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

$$\xrightarrow{\text{Simplify}}$$

LLIR Code

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

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

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

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

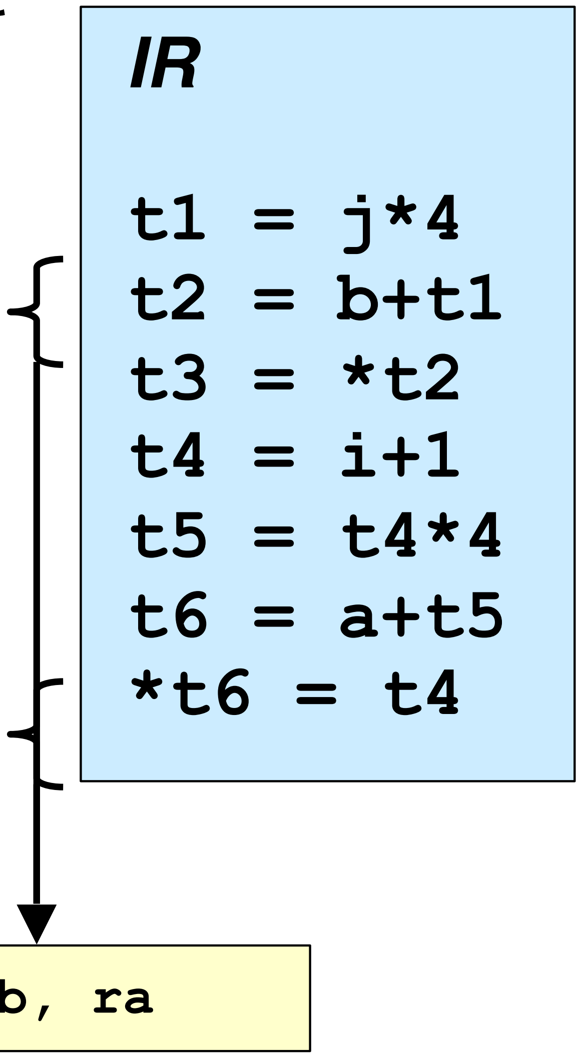
$\text{MEM}(r_0 + @w) \leftarrow r_{18}$

Minimizing Cost

- Goal:
  - Find instructions with low overall cost

- Difficulty
  - How to find these patterns?
  - Machine idioms may subsume IR operations that are not adjacent

- <u>Idea</u>: use tree matching to go beyond peephole matching
  - Convert computation into a tree
  - Match parts of the tree



IR

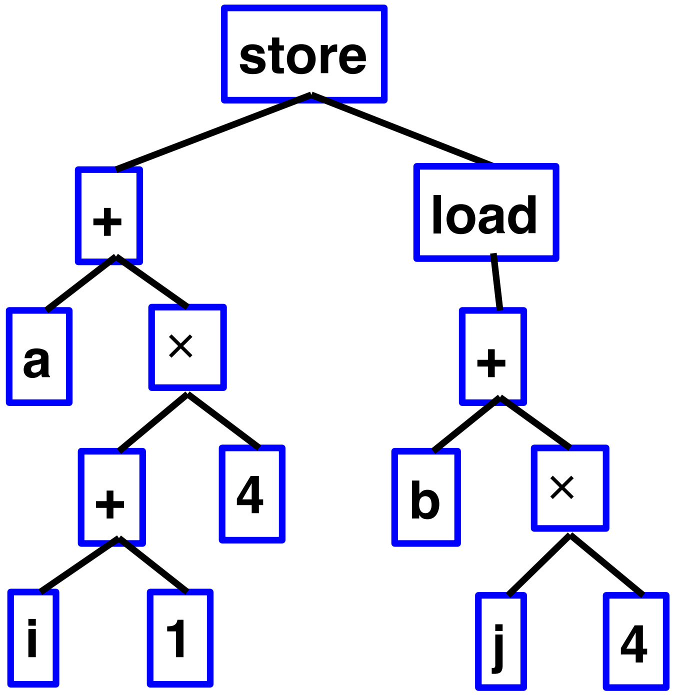
```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t4
```

```
movem rb, ra
```

Tree Representation

- Build a tree:

```
a[i+1] = b[j]
```



IR

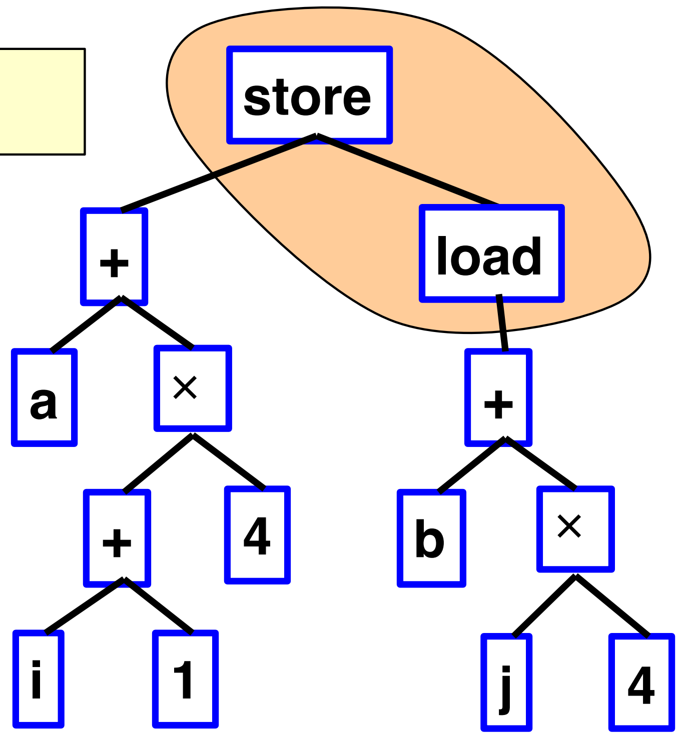
```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```

- Goal: find parts of the tree that correspond to machine instructions

Tiles

- <u>Idea</u>: a *tile* is contiguous piece of the tree that correponds to a machine instruction

```
movem rb, ra
```

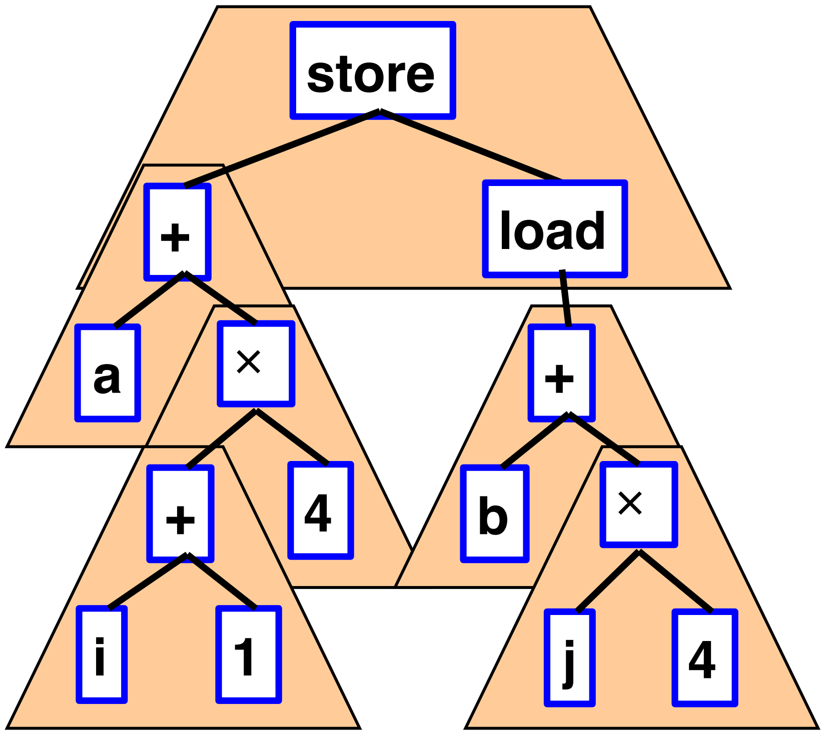


IR

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```

Tiling

- *Tiling*: cover the tree with tiles



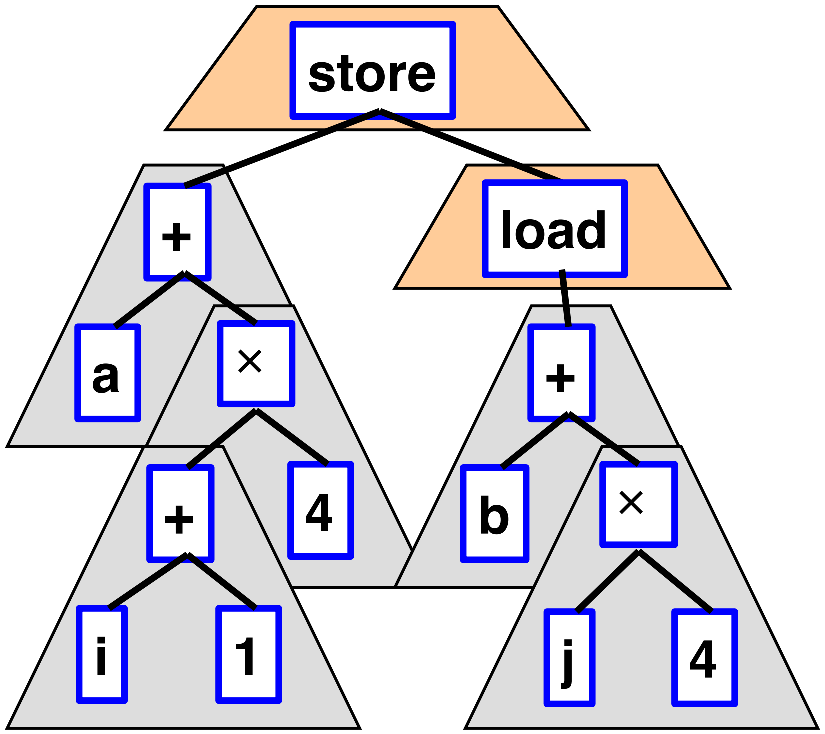
Assembly

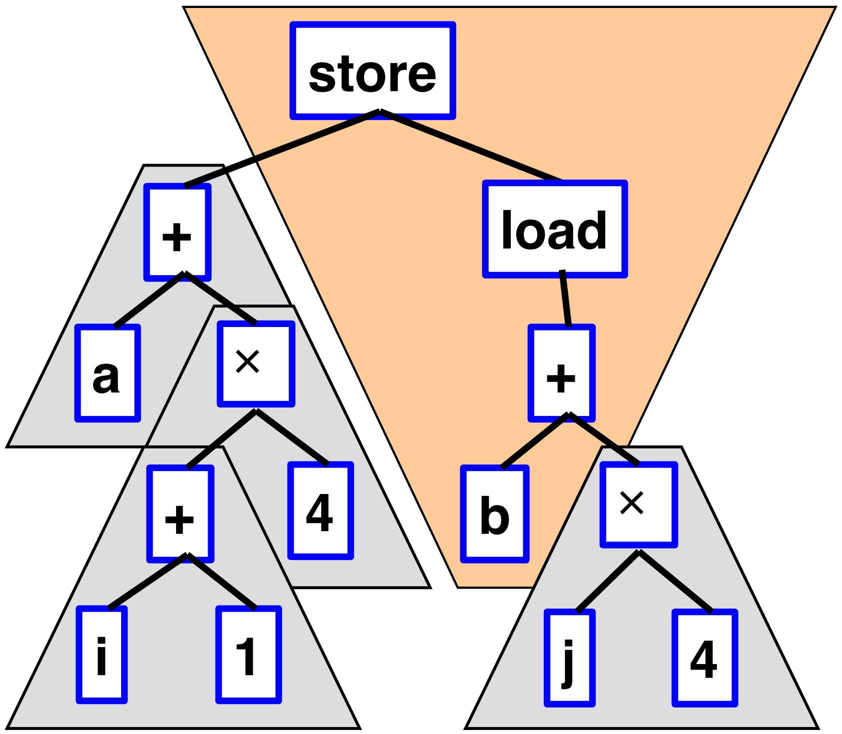
```
muli 4, rj
add rj, rb
addi 1, ri
muli 4, ri
add ri, ra
movem rb, ra
```

Tiling

```
load rb, r1
store r1, ra
```

```
movex rj, rb, ra
```

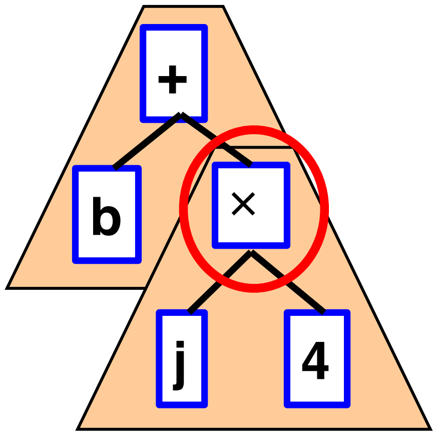




Generating Code

- Given a tiling of a tree
  - A tiling ***implements*** a tree if:
    - It covers all nodes in the tree

- Post-order tree walk
  - Emit machine instructions for each tile
  - Tie boundaries together with registers
  - <u>Note</u>: order of children matters

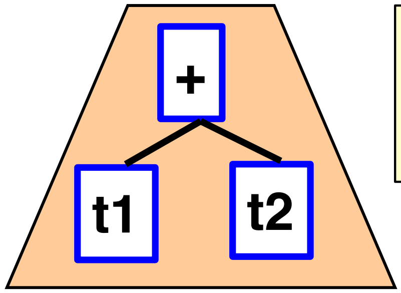


Tiling

- What's hard about this?
  - Define system of tiles in the compiler
  - Finding a tiling that implements the tree
    (Covers all nodes in the tree)
  - Finding a "good" tiling

To guarantee every tree can be tiled, provide a tile for each individual kind of node

- Different approaches
  - Ad-hoc pattern matching
  - Automated tools



```
mov    t1, t3
add    t2, t3
```