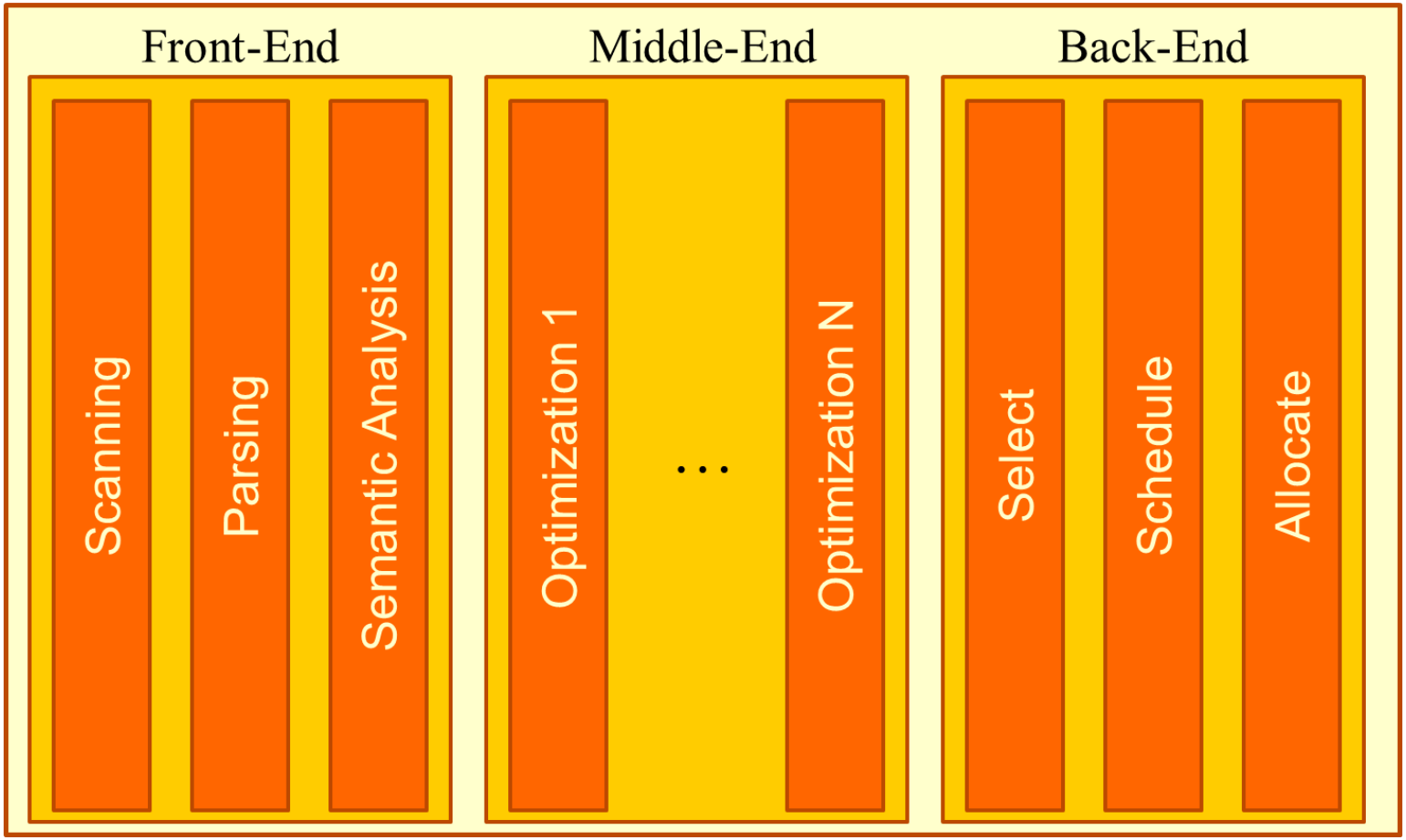
Instruction Selection

CPEG421/621: Compiler Construction

University of Delaware

March 12, 2012

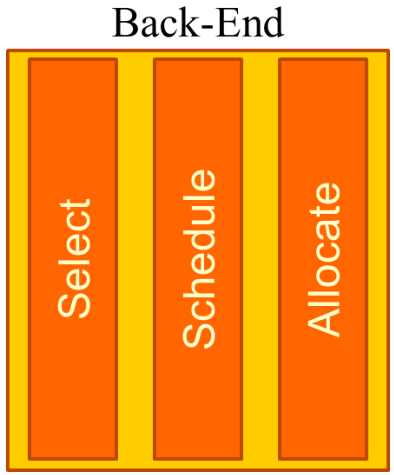
Back-End Overview



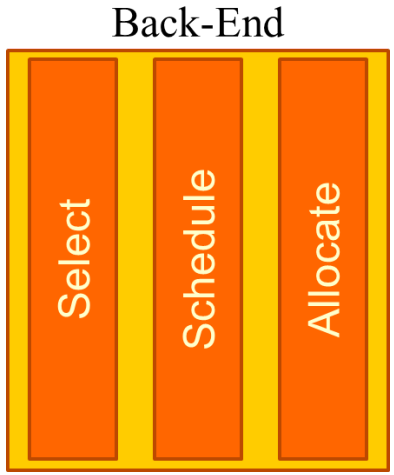
In the Reference Books

- Dragon Book: Chapter 8, Sections 8.7 (Peephole optimization) and 8.9 (Instruction selection by tree rewriting)
- Engineering a Compiler (Cooper and Torczon): chapter 11, sections 11.4 and 11.5

Overview



Overview



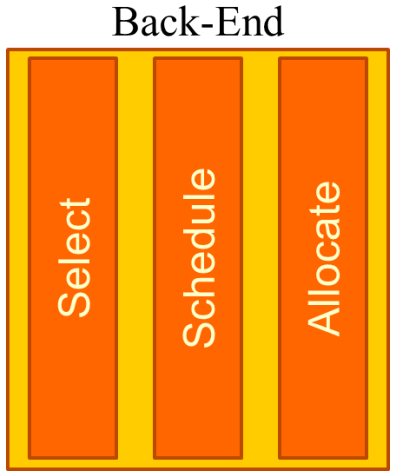
Instruction Selector

- Translates the IR given by the middle-end into ISA-specific instructions.
- Needs to choose between several operation sequences that have the same effect.

Overview

Instruction Selector

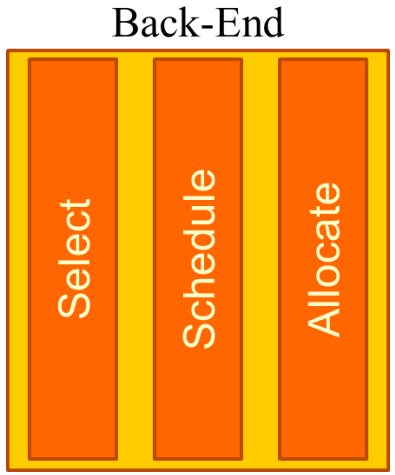
- Translates the IR given by the middle-end into ISA-specific instructions.
- Needs to choose between several operation sequences that have the same effect.



Instruction Scheduler

- Takes assembly code emitted by the instruction selector as input
- Decides in which *order* instructions should be inserted

Overview



Instruction Selector

- Translates the IR given by the middle-end into ISA-specific instructions.
- Needs to choose between several operation sequences that have the same effect.

Instruction Scheduler

- Takes assembly code emitted by the instruction selector as input
- Decides in which *order* instructions should be inserted

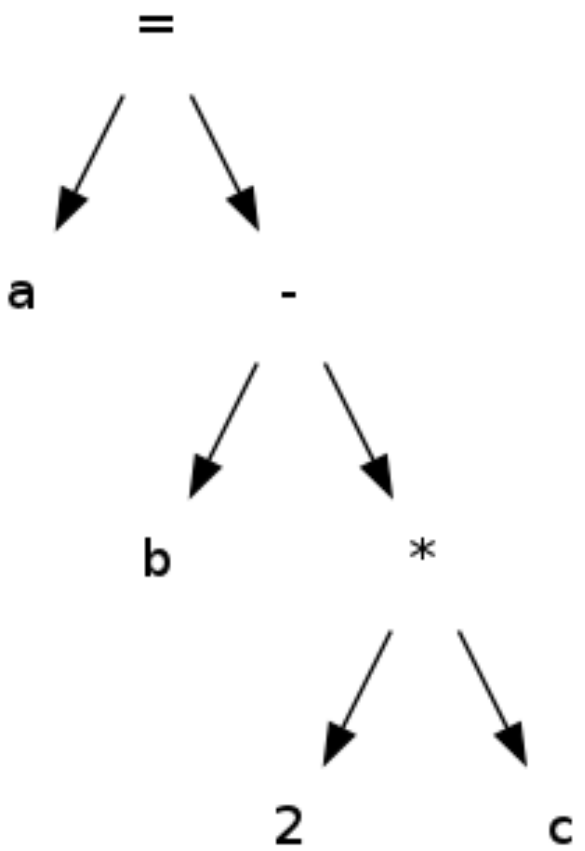
Register Allocator

- Takes a scheduled sequence of assembly instructions emitted by the instruction scheduler as input
- Decides which values should be spilled in memory, which should be kept in registers, etc., and when

Instruction Selection Overview

- Lowers an IR to a specific ISA
- Quality of instruction selection depends on how detailed is the IR
- Instruction selection has a deep impact on instruction scheduling

A Quick Reminder: Intermediate Representations



$$
\begin{aligned}
t_1 &= 2 \times c \\
t_2 &= b - t_1 \\
a &= t_2
\end{aligned}
$$

Table: Three-address code

Desired Qualities in an Instruction Selector

- Attain a certain level of abstraction
- Be fed with a machine description of the target architecture only
- Embed as little machine/target specific details in the selection algorithms themselves as possible

Desired Qualities in an Instruction Selector

- Attain a certain level of abstraction
- Be fed with a machine description of the target architecture only
- Embed as little machine/target specific details in the selection algorithms themselves as possible

If these features are reasonably fulfilled, then we have a retargetable compiler.

Instruction Selectors in a Nutshell

An instruction selector is composed of two main components

1. A pattern-matching engine
2. A set of tables to describe how to transition from the IR to the ISA.

The Devil is in the IR's Details

If the IR has a higher level of abstraction than the targeted ISA, then the instruction selector must embed some additional knowledge about the ISA, which makes it more complex and adds potential special cases only useful to this specific ISA.

On the contrary, if the IR has a lower level of abstraction than the target ISA, then all the needed information is provided at the IR level to the instruction selector. $\Rightarrow$ No additional knowledge required.

The Devil is in the IR's Details

If the IR has a higher level of abstraction than the targeted ISA, then the instruction selector must embed some additional knowledge about the ISA, which makes it more complex and adds potential special cases only useful to this specific ISA.

On the contrary, if the IR has a lower level of abstraction than the target ISA, then all the needed information is provided at the IR level to the instruction selector. $\Rightarrow$ No additional knowledge required.

Question: What kind of information makes an IR "low-level" enough?

Main Problem of Instruction Selection

Multiple sequences of instructions for a given ISA may yield the same results (w.r.t. correctness).

Main Problem of Instruction Selection

Multiple sequences of instructions for a given ISA may yield the same results (w.r.t. correctness).

Examples

$$\begin{array}{llll} \text{add} & r_i, 0 & \Rightarrow & r_j \\ \text{sub} & r_i, 0 & \Rightarrow & r_j \\ \text{and} & r_i, r_i & \Rightarrow & r_j \end{array}$$

Main Problem of Instruction Selection

Multiple sequences of instructions for a given ISA may yield the same results (w.r.t. correctness).

Examples

$$\begin{array}{llcl} \text{add} & r_i, 0 & \Rightarrow & r_j \\ \text{sub} & r_i, 0 & \Rightarrow & r_j \\ \text{and} & r_i, r_i & \Rightarrow & r_j \end{array}$$

$$\begin{array}{llcl} \text{mov} & 0 & \Rightarrow & r_i \\ \text{xor} & r_i, r_i & \Rightarrow & r_i \end{array}$$

Criteria Used for Instruction Selection

- Execution time / speed (emphasis on locality)
- Energy (emphasis of locality – even for register reuse!)
- Fault-tolerance (better to recompute some values sometimes)
- Code size (e.g. CISC vs RISC)

From Intermediate Representation to ISA-Specific Instructions

Lowering an IR to a given ISA depends on the target:

- Scalar RISC machines are almost a 1:1 match with the IR's operations
- CISC machines coalesce several RISC operations into one instruction

Example: Aggregating Operations Into One CISC Instruction

Initial expression (assuming that *A* and *B* are `uint64_t` arrays):

$$B[\beta] = A[\alpha] + SomeValue$$

Example: Aggregating Operations Into One CISC Instruction

Initial expression (assuming that *A* and *B* are `uint64_t` arrays):

$$B[\beta] = A[\alpha] + SomeValue$$

Some kind of linear code IR:

add	$r_0, @A$	$\Rightarrow$	r_1
add	r_0, α	$\Rightarrow$	r_2
mul	$r_2, 8$	$\Rightarrow$	r_3
load	r_3	$\Rightarrow$	r_4
add	r_4, r_{10}	$\Rightarrow$	r_{11}
add	$r_0, @B$	$\Rightarrow$	r_5
add	r_5, β	$\Rightarrow$	r_6
mul	$r_6, 8$	$\Rightarrow$	r_7
store	r_{11}	$\Rightarrow$	r_7

Example: Aggregating Operations Into One CISC Instruction

Initial expression (assuming that *A* and *B* are `uint64_t` arrays):

$$B[\beta] = A[\alpha] + SomeValue$$

Some kind of linear code IR:

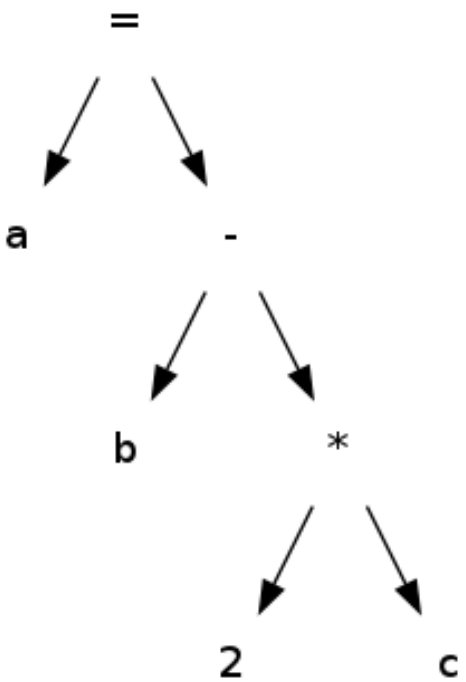
add	$r_0, @A$	$\Rightarrow$	r_1
add	r_0, α	$\Rightarrow$	r_2
mul	$r_2, 8$	$\Rightarrow$	r_3
load	r_3	$\Rightarrow$	r_4
add	r_4, r_{10}	$\Rightarrow$	r_{11}
add	$r_0, @B$	$\Rightarrow$	r_5
add	r_5, β	$\Rightarrow$	r_6
mul	$r_6, 8$	$\Rightarrow$	r_7
store	r_{11}	$\Rightarrow$	r_7

A CISC-like aggregation:
add $[@A + \alpha * 8], r_i \Rightarrow [@B + \beta * 8]$

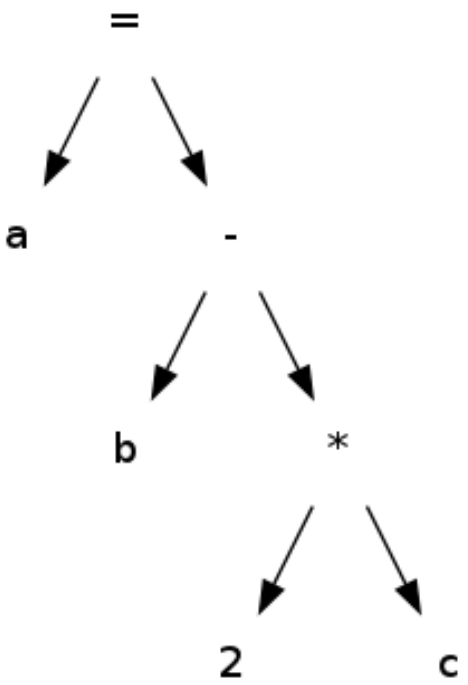
Two Methods to Perform Instruction Selection

- Tree Pattern-Matching
- Peephole Optimization

The Need for A Good Low-Level IR



The Need for A Good Low-Level IR



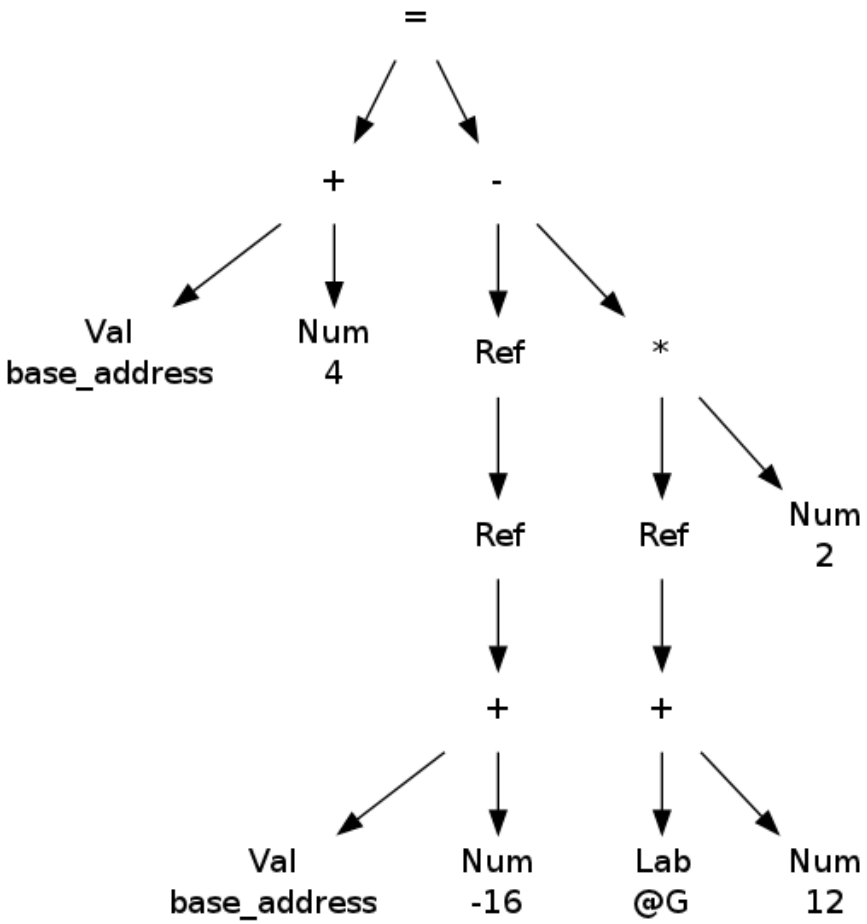


Figure: $a = b - 2 * c$

Tree Pattern-Matching (1/3)

- Uses an AST as an input
- Tries to map sub-trees in the AST to operations in the ISA

Tree Pattern-Matching (2/3)

Problem Formulation

Given an AST and a collection of *operation trees*, map the AST to operations.

This is done through the building of a *tiling*.

Tree Pattern-Matching (2/3)

Problem Formulation

Given an AST and a collection of *operation trees*, map the AST to operations.
This is done through the building of a *tiling*.

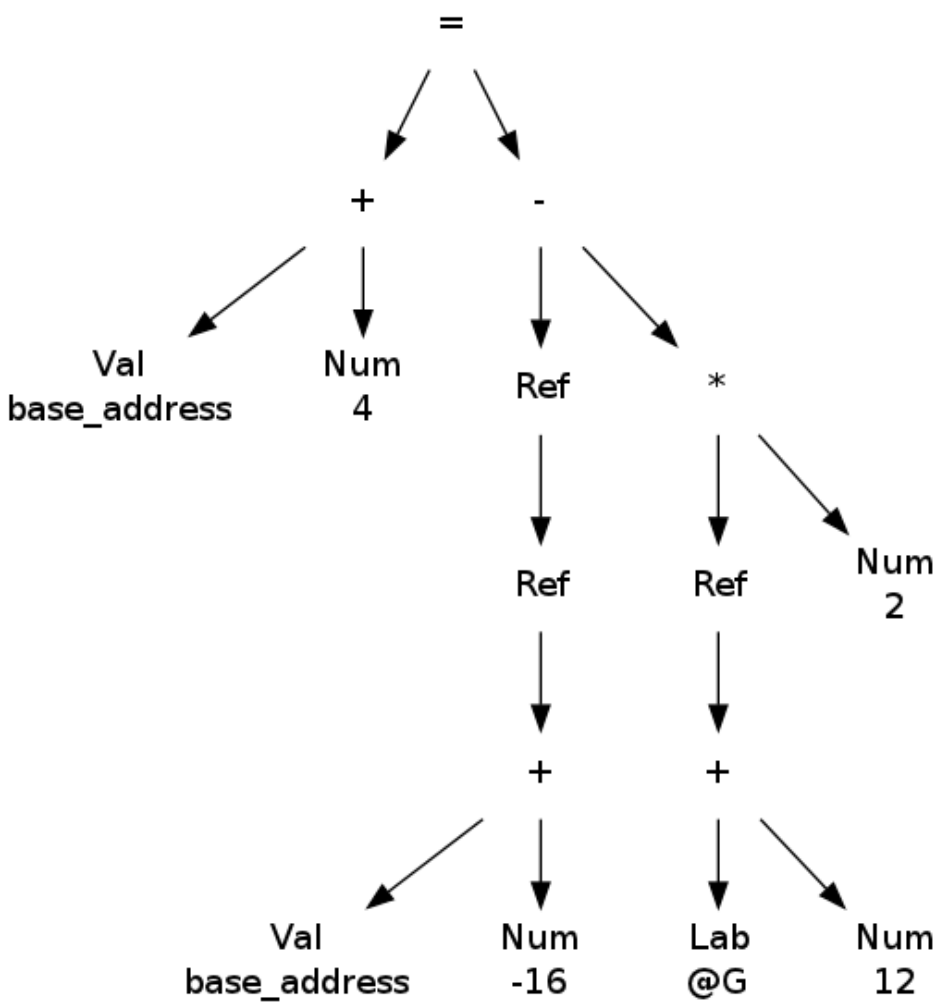
Definition: Tiling

A tiling is a pair $< AST - node, op - tree >$, where $AST - node$ is a node in the AST, and $op - tree$ is an operation tree (e.g. a tree representation of an operation such as $r_i \leftarrow r_j \oplus r_k$, $@mem\_address \leftarrow r_i$, etc.).

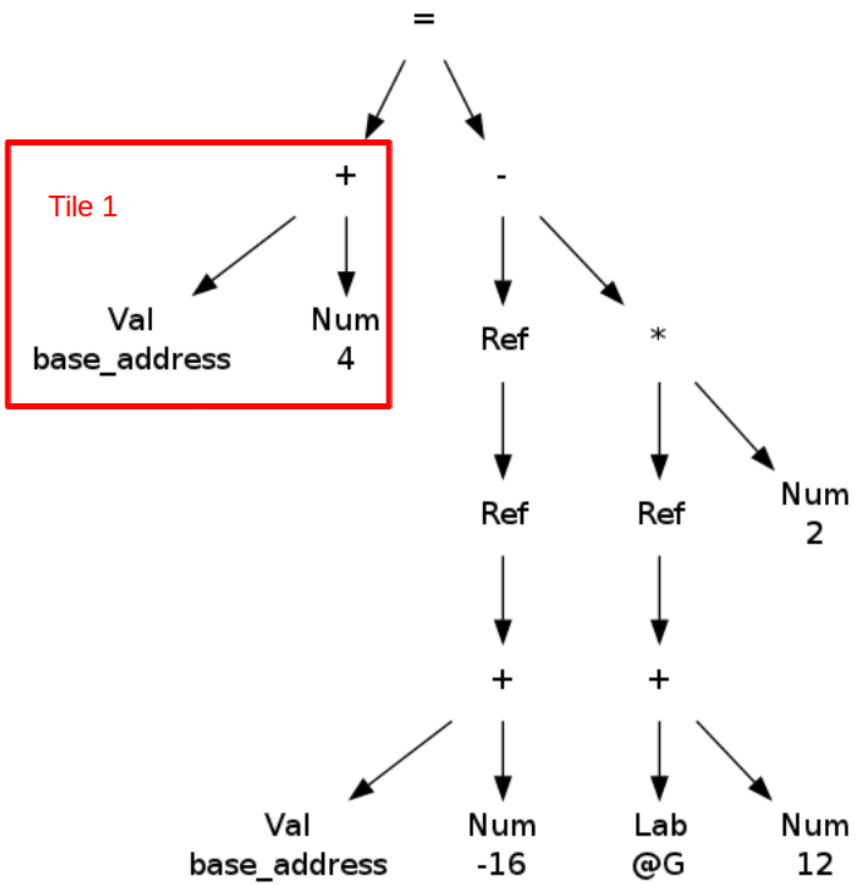
Tree Pattern-Matching (3/3)

- A tiling *implements* an $AST - node$ if every operation is implemented *and* each tile connects with its neighbors.
- A tile $< AST - node, op - tree >$ connects with its neighbors if the $AST - node$ is covered by a leaf in another $op - tree$ in the tiling (unless the $AST - node$ is the root of the AST).
- If two $AST - nodes$ overlap, they need to agree on the storage class of the common node.
  - To prevent "node incompatibilities," we use a bottom-up approach (BURS – Bottom-Up Rewriting System).

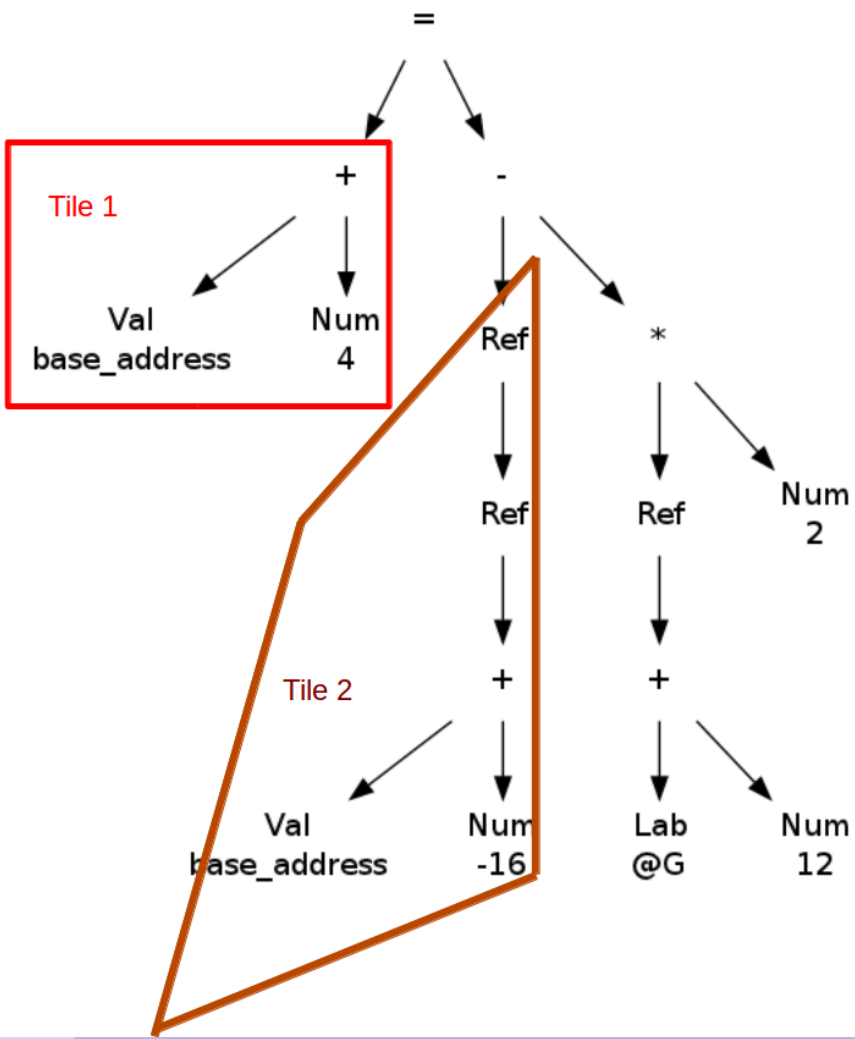
Example of Tiling



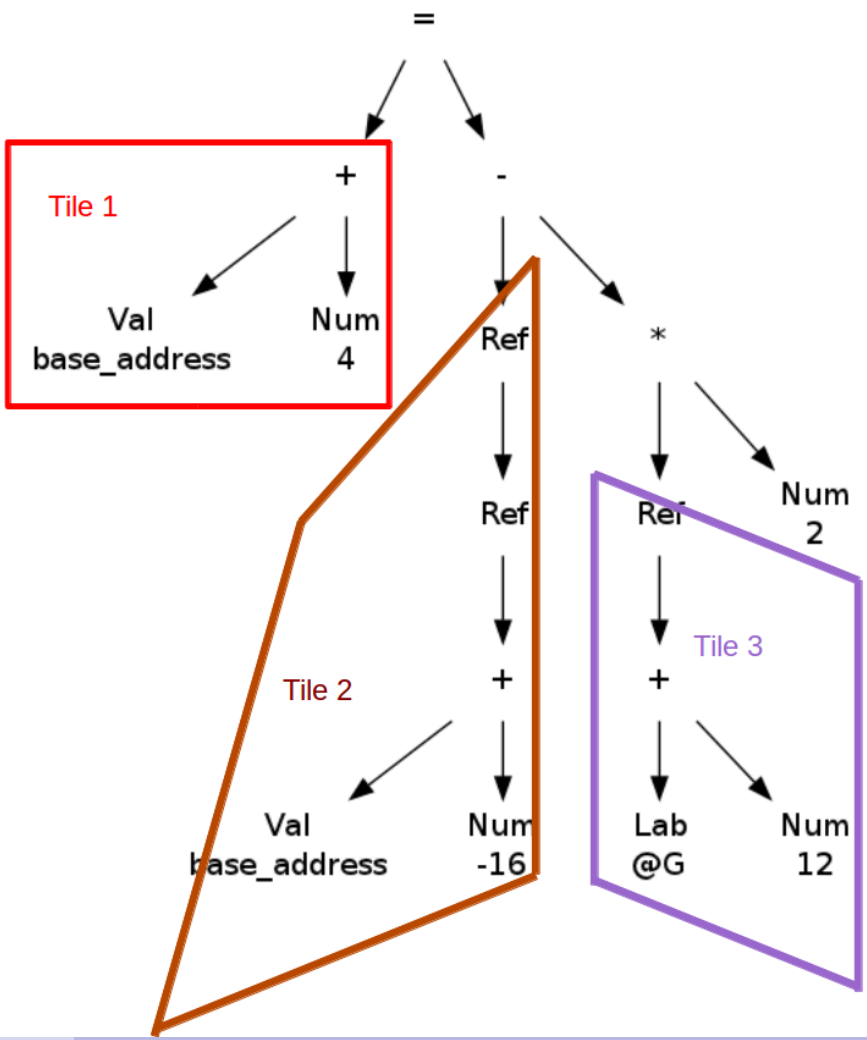
Example of Tiling



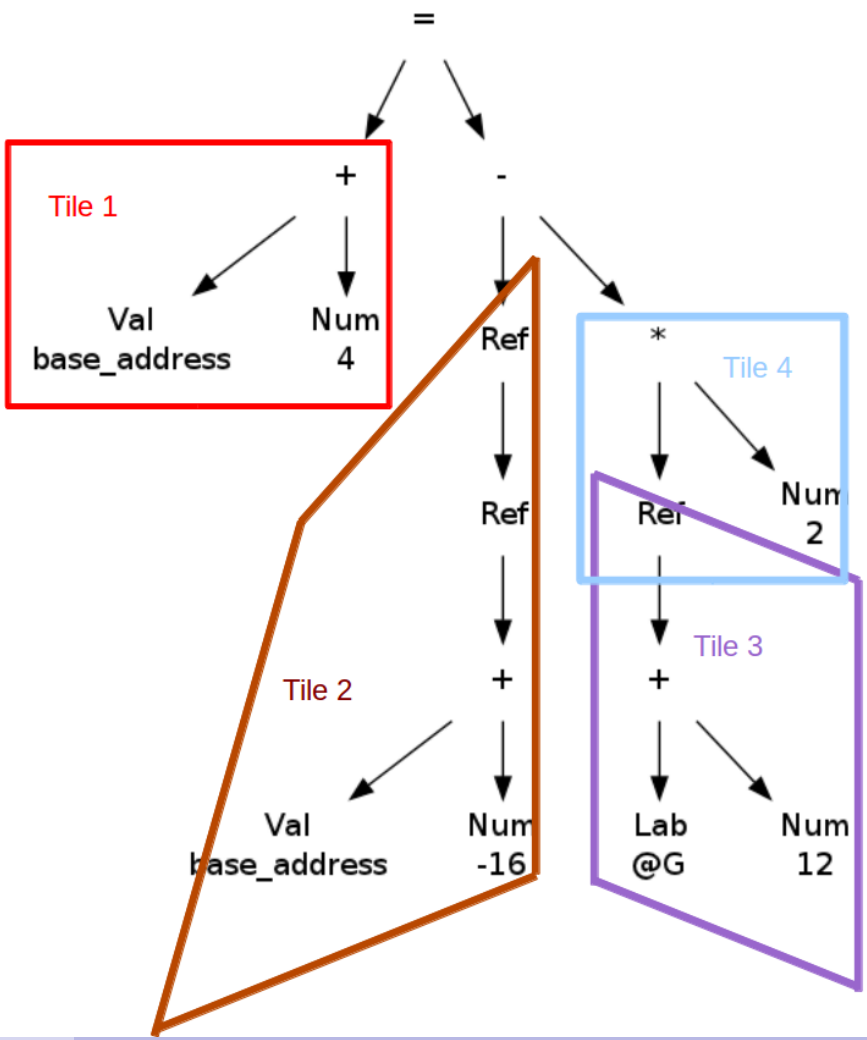
Example of Tiling



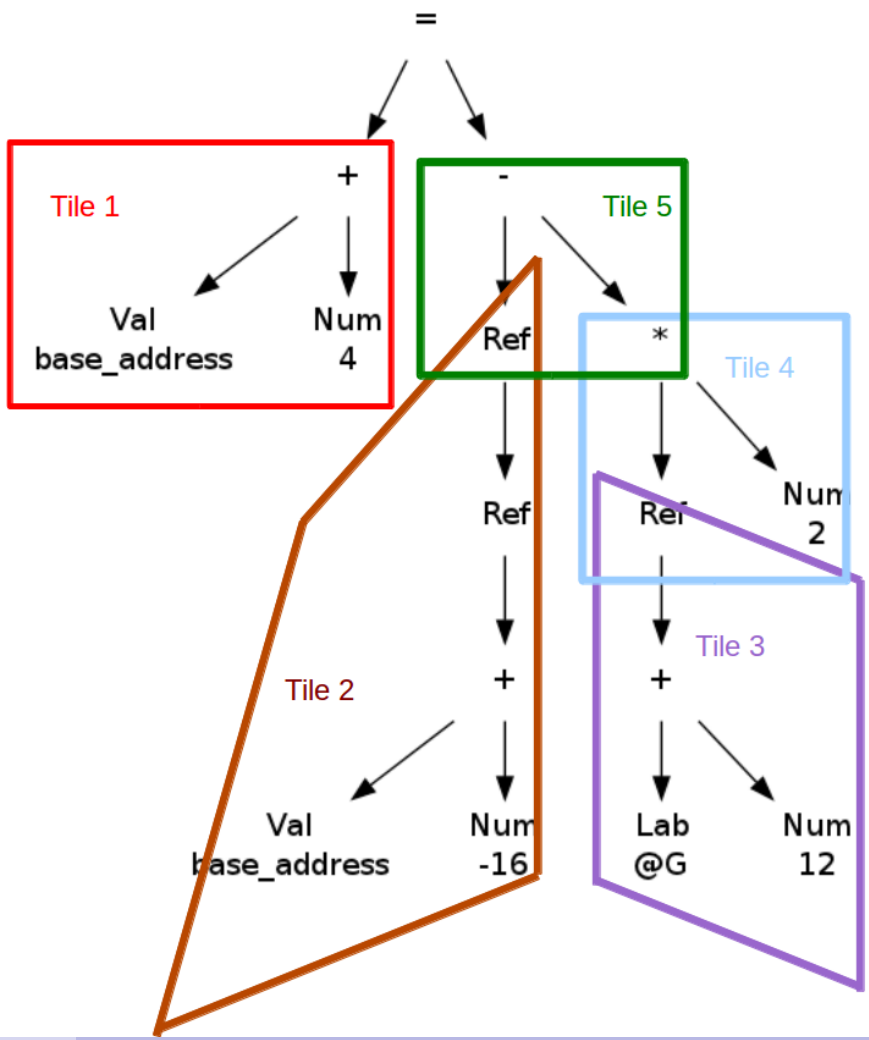
Example of Tiling



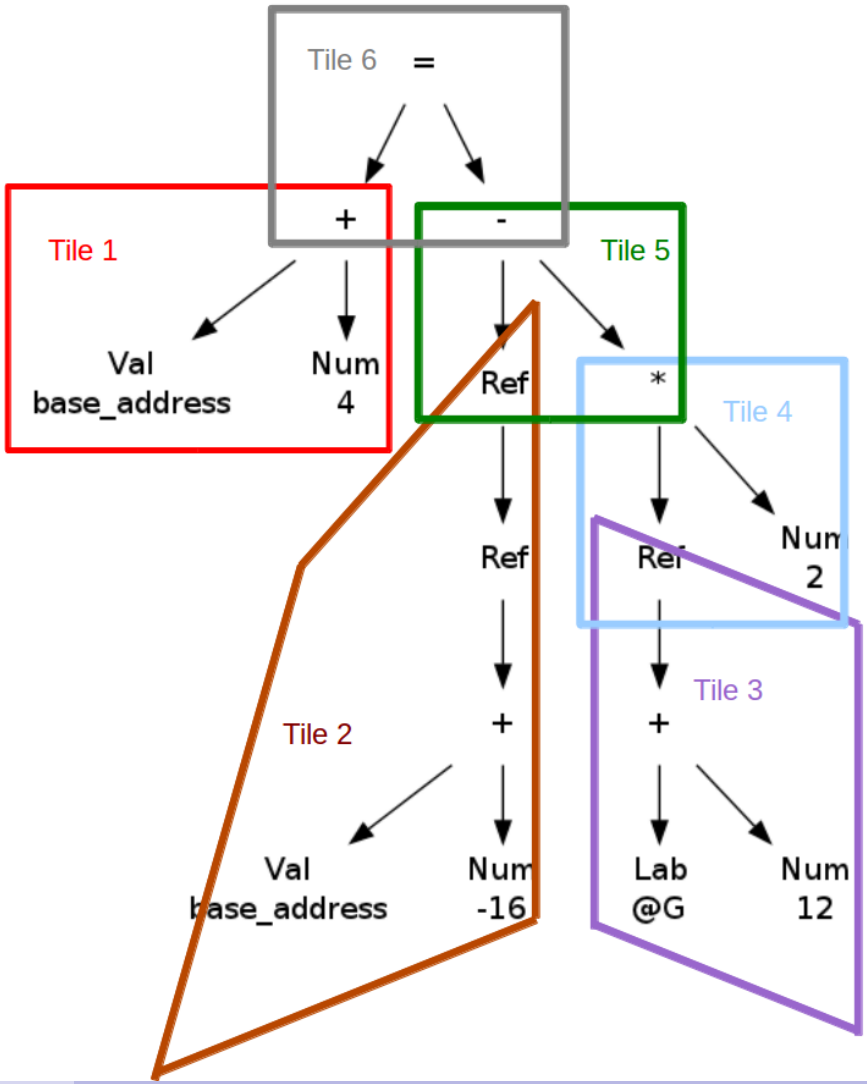
Example of Tiling



Example of Tiling



Example of Tiling



Rewriting Rules for Tree Pattern-Matching

The relationships for tiles (i.e. $< AST - node, op - tree >$ pairs) are encoded as a set of *rewriting rules*. Each rule comprises:

- A production in a tree grammar,
- A code template,
- An associated cost.

Example of Rewriting Rules Table

	Production	Cost	Code Template
1	Goal $\rightarrow$ Assign	0	
2	Assign $\rightarrow\leftarrow$ (Reg_1, Reg_2)	1	store $r_2 \Rightarrow r_1$
3	Assign $\rightarrow\leftarrow$ $(+(Reg_1, Reg_2), Reg_3)$	1	storeAO $r_3 \Rightarrow r_1, r_2$
4	Assign $\rightarrow\leftarrow$ $(+(Reg_1, Num_2), Reg_3)$	1	storeAI $r_3 \Rightarrow r_1, n_2$
5	Assign $\rightarrow\leftarrow$ $(+(Num_1, Reg_2), Reg_3)$	1	storeAI $r_3 \Rightarrow n_1, r_2$
6	Reg $\rightarrow$ Lab_1	1	loadI $l_1 \Rightarrow r_{new}$
7	Reg $\rightarrow$ Val_1	0	
8	Reg $\rightarrow$ Num_1	1	loadI $n_1 \Rightarrow r_{new}$
9	Reg $\rightarrow$ $\blacklozenge(Reg_1)$	1	load $r_1 \Rightarrow r_{new}$
10	Reg $\rightarrow$ $\blacklozenge(+(Reg_1, Reg_2))$	1	loadAO $r_1, r_2 \Rightarrow r_{new}$
11	Reg $\rightarrow$ $\blacklozenge(+(Reg_1, Num_2))$	1	loadAI $r_1, n_2 \Rightarrow r_{new}$
12	Reg $\rightarrow$ $\blacklozenge(+(Num_1, Reg_2))$	1	loadAI $r_2, n_1 \Rightarrow r_{new}$
13	Reg $\rightarrow$ $\blacklozenge(+(Reg_1, Lab_2))$	1	loadAI $r_1, l_2 \Rightarrow r_{new}$
14	Reg $\rightarrow$ $\blacklozenge(+(Lab_1, Reg_2))$	1	loadAI $r_2, l_1 \Rightarrow r_{new}$
15	Reg $\rightarrow$ $+(Reg_1, Reg_2)$	1	add $r_1, r_2 \Rightarrow r_{new}$
16	Reg $\rightarrow$ $+(Reg_1, Num_2)$	1	addI $r_1, n_2 \Rightarrow r_{new}$
17	Reg $\rightarrow$ $+(Num_1, Reg_2)$	1	addI $r_2, n_1 \Rightarrow r_{new}$
18	Reg $\rightarrow$ $+(Reg_1, Lab_2)$	1	addI $r_1, l_2 \Rightarrow r_{new}$
19	Reg $\rightarrow$ $+(Lab_1, Reg_2)$	1	addI $r_2, l_2 \Rightarrow r_{new}$
20	Reg $\rightarrow$ $-(Reg_1, Reg_2)$	1	sub $r_2, r_2 \Rightarrow r_{new}$
21	Reg $\rightarrow$ $-(Reg_1, Num_2)$	1	subI $r_1, n_2 \Rightarrow r_{new}$
22	Reg $\rightarrow$ $-(Num_1, Reg_2)$	1	subI $r_2, n_1 \Rightarrow r_{new}$
20	Reg $\rightarrow$ $\times(Reg_1, Reg_2)$	1	mul $r_2, r_2 \Rightarrow r_{new}$
21	Reg $\rightarrow$ $\times(Reg_1, Num_2)$	1	mulI $r_1, n_2 \Rightarrow r_{new}$
22	Reg $\rightarrow$ $\times(Num_1, Reg_2)$	1	mulI $r_2, n_1 \Rightarrow r_{new}$

Finding a Tiling

Initial assumptions: no more than two operands per operation; only one operation on the right-hand side (RHS). INSERT ALGORITHM FROM 11.8

Finding Low-Cost Matches

We take advantage of the post-order/bottom-up traversal:

- Locally choose the best cost match
- Resolve conflicts on overlapping tiles
- When going up the tree, “simply” add the resulting cost to the global cost.

Peephole Optimization

Initially, peephole optimization was an ASM $\rightarrow$ ASM transformation, using an exhaustive search on a set of limited hand-coded patterns. Today's ISAs are too complex to continue like this.

Problem Formulation

Given a sequence of instructions, determine an instruction window which will examine a sub-sequence of such instructions and try to determine their relationships.

Peephole Optimization

Initially, peephole optimization was an ASM $\rightarrow$ ASM transformation, using an exhaustive search on a set of limited hand-coded patterns. Today's ISAs are too complex to continue like this.

Problem Formulation

Given a sequence of instructions, determine an instruction window which will examine a sub-sequence of such instructions and try to determine their relationships.

The Three Steps of Peephole Optimization

1. Expand
2. Simplify
3. Match

Peephole Optimization

Initially, peephole optimization was an ASM $\rightarrow$ ASM transformation, using an exhaustive search on a set of limited hand-coded patterns. Today's ISAs are too complex to continue like this.

Problem Formulation

Given a sequence of instructions, determine an instruction window which will examine a sub-sequence of such instructions and try to determine their relationships.

The Three Steps of Peephole Optimization

1. Expand
2. Simplify
3. Match

Nowadays, peephole optimization tends to follow this pipeline: IR $\Rightarrow$ Expansion $\Rightarrow$ LLIR $\Rightarrow$ Simplification $\Rightarrow$ LLIR $\Rightarrow$ Matching $\Rightarrow$ ASM

Examples for Peephole Stages

Expansion

- Infinite number of registers
- Don't care about redundant operations
- Don't care about constants, etc.

Examples for Peephole Stages

Expansion

- Infinite number of registers
- Don't care about redundant operations
- Don't care about constants, etc.

Simplify

Withing an instruction window of *n* instructions, apply:

- Forward substitution
- Algebraic simplification
- Constant-value expression simplification
- Dead code elimination (unreachable code, "dead" labels, . . .)
- etc.

Examples for Peephole Stages

Expansion

- Infinite number of registers
- Don't care about redundant operations
- Don't care about constants, etc.

Matching

- Comparison between LLIR and pattern library

Simplify

Withing an instruction window of *n* instructions, apply:

- Forward substitution
- Algebraic simplification
- Constant-value expression simplification
- Dead code elimination (unreachable code, "dead" labels, . . .)
- etc.

Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let's assume a 3-instruction window.

Sequence 1

$$\begin{array}{ll} r_{10} & 2 \\ r_{11} & @G \\ r_{12} & 12 \end{array}$$

Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let's assume a 3-instruction window.

Sequence 1

$$\begin{array}{ll} r_{10} & 2 \\ r_{11} & @G \\ r_{12} & 12 \end{array}$$

Sequence 2

$$\begin{array}{ll} r_{11} & @G \\ r_{12} & 12 \\ r_{13} & r_{11} + r_{12} \end{array}$$

Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let's assume a 3-instruction window.

Sequence 1

$$\begin{array}{ll} r_{10} & 2 \\ r_{11} & @G \\ r_{12} & 12 \end{array}$$

Sequence 2

$$\begin{array}{ll} r_{11} & @G \\ r_{12} & 12 \\ r_{13} & r_{11} + r_{12} \end{array}$$

Sequence 3

$$\begin{array}{ll} r_{11} & @G \\ r_{13} & r_{11} + 12 \\ r_{14} & M(r_{13}) \end{array}$$

Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let's assume a 3-instruction window.

Sequence 1

$$\begin{array}{ll} r_{10} & 2 \\ r_{11} & @G \\ r_{12} & 12 \end{array}$$

Sequence 2

$$\begin{array}{ll} r_{11} & @G \\ r_{12} & 12 \\ r_{13} & r_{11} + r_{12} \end{array}$$

Sequence 3

$$\begin{array}{ll} r_{11} & @G \\ r_{13} & r_{11} + 12 \\ r_{14} & M(r_{13}) \end{array}$$

Sequence 4

$$\begin{array}{ll} r_{11} & @G \\ r_{14} & M(r_{11} + 12) \\ r_{15} & r_{10} \times r_{14} \end{array}$$

Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let's assume a 3-instruction window.

Sequence 5

$$\begin{array}{ll} r_{14} & M(r_{11} + 12) \\ r_{15} & r_{10} \times r_{14} \\ r_{16} & -16 \end{array}$$

Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let's assume a 3-instruction window.

Sequence 5

$$\begin{array}{ll} r_{14} & M(r_{11} + 12) \\ r_{15} & r_{10} \times r_{14} \\ r_{16} & -16 \end{array}$$

Sequence 6

$$\begin{array}{ll} r_{15} & r_{10} \times r_{14} \\ r_{16} & -16 \\ r_{17} & r_{fp} + r_{16} \end{array}$$

Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let's assume a 3-instruction window.

Sequence 5

$$\begin{array}{ll} r_{14} & M(r_{11}+12) \\ r_{15} & r_{10} \times r_{14} \\ r_{16} & -16 \end{array}$$

Sequence 7

$$\begin{array}{ll} r_{15} & r_{10} \times r_{14} \\ r_{17} & r_{fp}-16 \\ r_{18} & M(r_{17}) \end{array}$$

Sequence 6

$$\begin{array}{ll} r_{15} & r_{10} \times r_{14} \\ r_{16} & -16 \\ r_{17} & r_{fp}+r_{16} \end{array}$$

Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let's assume a 3-instruction window.

Sequence 5

$$\begin{array}{ll} r_{14} & M(r_{11}+12) \\ r_{15} & r_{10} \times r_{14} \\ r_{16} & -16 \end{array}$$

Sequence 6

$$\begin{array}{ll} r_{15} & r_{10} \times r_{14} \\ r_{16} & -16 \\ r_{17} & r_{fp}+r_{16} \end{array}$$

Sequence 7

$$\begin{array}{ll} r_{15} & r_{10} \times r_{14} \\ r_{17} & r_{fp}-16 \\ r_{18} & M(r_{17}) \end{array}$$

Sequence 8

$$\begin{array}{ll} r_{15} & r_{10} \times r_{14} \\ r_{18} & M(r_{fp}-16) \\ r_{19} & M(r_{18}) \end{array}$$

Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let's assume a 3-instruction window.

Sequence 9

$$\begin{array}{ll} r_{18} & M(r_{fp} - 16) \\ r_{19} & M(r_{18}) \\ r_{20} & r_{19} - r_{15} \end{array}$$

Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let's assume a 3-instruction window.

Sequence 9

$$\begin{array}{ll} r_{18} & M(r_{fp} - 16) \\ r_{19} & M(r_{18}) \\ r_{20} & r_{19} - r_{15} \end{array}$$

Sequence 10

$$\begin{array}{ll} r_{19} & M(r_{18}) \\ r_{20} & r_{19} - r_{15} \\ r_{21} & 4 \end{array}$$

Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let's assume a 3-instruction window.

Sequence 9

$$\begin{array}{ll} r_{18} & M(r_{fp} - 16) \\ r_{19} & M(r_{18}) \\ r_{20} & r_{19} - r_{15} \end{array}$$

Sequence 11

$$\begin{array}{ll} r_{20} & r_{19} - r_{15} \\ r_{21} & 4 \\ r_{22} & r_{fp} + r_{21} \end{array}$$

Sequence 10

$$\begin{array}{ll} r_{19} & M(r_{18}) \\ r_{20} & r_{19} - r_{15} \\ r_{21} & 4 \end{array}$$

Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let's assume a 3-instruction window.

Sequence 9

$$\begin{array}{ll} r_{18} & M(r_{fp} - 16) \\ r_{19} & M(r_{18}) \\ r_{20} & r_{19} - r_{15} \end{array}$$

Sequence 10

$$\begin{array}{ll} r_{19} & M(r_{18}) \\ r_{20} & r_{19} - r_{15} \\ r_{21} & 4 \end{array}$$

Sequence 11

$$\begin{array}{ll} r_{20} & r_{19} - r_{15} \\ r_{21} & 4 \\ r_{22} & r_{fp} + r_{21} \end{array}$$

Sequence 12

$$\begin{array}{ll} r_{20} & r_{19} - r_{15} \\ r_{22} & r_{fp} + 4 \\ M(r_{22}) & r_{20} \end{array}$$

Example (Taken from Cooper & Torczon)

Initial Expression

Op	Arg_1	Arg_2	*Result*
$\times$	2	c	t_1
$-$	b	t_1	a

Example (Taken from Cooper & Torczon)

Initial Expression

Op	Arg_1	Arg_2	$Result$
$\times$	2	c	t_1
$-$	b	t_1	a

1. Expansion

$$
\begin{aligned}
r_{10} &\leftarrow 2 \\
r_{11} &\leftarrow @G \\
r_{12} &\leftarrow 12 \\
r_{13} &\leftarrow r_{11} + r_{12} \\
r_{14} &\leftarrow M(R_{13}) \\
r_{15} &\leftarrow r_{10} \times r_{14} \\
r_{16} &\leftarrow -16 \\
r_{17} &\leftarrow r_{fp} + r_{16} \\
r_{18} &\leftarrow M(r_{17}) \\
r_{19} &\leftarrow M(r_{18}) \\
r_{20} &\leftarrow r_{19} - r_{15} \\
r_{21} &\leftarrow 4 \\
r_{22} &\leftarrow r_{fp} + r_{21} \\
M(r_{22}) &\leftarrow r_{20}
\end{aligned}
$$

Example (Taken from Cooper & Torczon)

1. Expansion

$$
\begin{array}{lcl}
r_{10} & \leftarrow & 2 \\
r_{11} & \leftarrow & @G \\
r_{12} & \leftarrow & 12 \\
r_{13} & \leftarrow & r_{11} + r_{12} \\
r_{14} & \leftarrow & M(R_{13}) \\
r_{15} & \leftarrow & r_{10} \times r_{14} \\
r_{16} & \leftarrow & -16 \\
r_{17} & \leftarrow & r_{fp} + r_{16} \\
r_{18} & \leftarrow & M(r_{17}) \\
r_{19} & \leftarrow & M(r_{18}) \\
r_{20} & \leftarrow & r_{19} - r_{15} \\
r_{21} & \leftarrow & 4 \\
r_{22} & \leftarrow & r_{fp} + r_{21} \\
M(r_{22}) & \leftarrow & r_{20}
\end{array}
$$

Example (Taken from Cooper & Torczon)

1. Expansion

$$
\begin{aligned}
r_{10} &\leftarrow 2 \\
r_{11} &\leftarrow @G \\
r_{12} &\leftarrow 12 \\
r_{13} &\leftarrow r_{11} + r_{12} \\
r_{14} &\leftarrow M(R_{13}) \\
r_{15} &\leftarrow r_{10} \times r_{14} \\
r_{16} &\leftarrow -16 \\
r_{17} &\leftarrow r_{fp} + r_{16} \\
r_{18} &\leftarrow M(r_{17}) \\
r_{19} &\leftarrow M(r_{18}) \\
r_{20} &\leftarrow r_{19} - r_{15} \\
r_{21} &\leftarrow 4 \\
r_{22} &\leftarrow r_{fp} + r_{21} \\
M(r_{22}) &\leftarrow r_{20}
\end{aligned}
$$

2. Simplify (Sequence 13)

$$
\begin{aligned}
r_{10} &\leftarrow 2 \\
r_{11} &\leftarrow @G \\
r_{14} &\leftarrow M(r_{13} + 12) \\
r_{15} &\leftarrow r_{10} \times r_{14} \\
r_{18} &\leftarrow r_{fp} - 16 \\
r_{19} &\leftarrow M(r_{18}) \\
r_{20} &\leftarrow r_{19} - r15 \\
M(r_{fp} + 4) &\leftarrow r_{20}
\end{aligned}
$$

Example (Taken from Cooper & Torczon)

2. Simplify

$$
\begin{array}{lcl}
r_{10} & \leftarrow & 2 \\
r_{11} & \leftarrow & @G \\
r_{14} & \leftarrow & M(r_{13} + 12) \\
r_{15} & \leftarrow & r_{10} \times r_{14} \\
r_{18} & \leftarrow & r_{fp} - 16 \\
r_{19} & \leftarrow & M(r_{18}) \\
r_{20} & \leftarrow & r_{19} - r15 \\
M(r_{fp} + 4) & \leftarrow & r_{20}
\end{array}
$$

Example (Taken from Cooper & Torczon)

2. Simplify

$$
\begin{array}{lcl}
r_{10} & \leftarrow & 2 \\
r_{11} & \leftarrow & @G \\
r_{14} & \leftarrow & M(r_{13} + 12) \\
r_{15} & \leftarrow & r_{10} \times r_{14} \\
r_{18} & \leftarrow & r_{fp} - 16 \\
r_{19} & \leftarrow & M(r_{18}) \\
r_{20} & \leftarrow & r_{19} - r15 \\
M(r_{fp} + 4) & \leftarrow & r_{20}
\end{array}
$$

3. Match

$$
\begin{array}{llcl}
\texttt{loadI} & 2 & \Rightarrow & r_{10} \\
\texttt{loadI} & @G & \Rightarrow & r_{11} \\
\texttt{AI} & r_{11}, 12 & \Rightarrow & r_{14} \\
\texttt{mult} & r_{10}, r_{14} & \Rightarrow & r_{15} \\
\texttt{loadAI} & r_{fp}, -16 & \Rightarrow & r_{18} \\
\texttt{load} & r_{18} & \Rightarrow & r_{19} \\
\texttt{sub} & r_{19}, r_{15} & \Rightarrow & r_{20} \\
\texttt{storeAI} & r_{20} & \Rightarrow & r_{fp}, 4
\end{array}
$$

Dead-Value Recognition

Question: How can we detect that a given value is dead in a given instruction window/instruction block?

Dead-Value Recognition

Question: How can we detect that a given value is dead in a given instruction window/instruction block? Several solutions:

1. Use LIVEOUT sets:
    1. Compute a LIVEOUT setfor each basic block
    2. Do a post-order pass on each block

Dead-Value Recognition

Question: How can we detect that a given value is dead in a given instruction window/instruction block? Several solutions:

1. Use LIVEOUT sets:
   1. Compute a LIVEOUT setfor each basic block
   2. Do a post-order pass on each block
2. Take advantage of the SSA form
   - → Identify the names used in more than one basic block *and* consider them live on exit of their respective blocks.

The expander stage can then mark last uses in the LLIR (through the construction of the LIVENOW set, see next slide).

Building the LIVENOW Set

1. For each basic block B, LIVENOW(B) = LIVEOUT(B)
   $\rightarrow$ If LIVEOUT(B) $= \emptyset$ then LIVENOW(B) = *GlobalNames*
2. For each operation of the type $r_i \leftarrow r_j \oplus r_k$, do
   LIVENOW(B) = (LIVENOW(B) $- r_i$) $\cup \{r_j, r_k\}$

If the target machine uses condition codes (*cc*) to control conditional branches, then the expander must insert the *cc* explicitly. Example:
If the IR expression is $r_i \times r_j + r_k$, then a possible LLIR expansion could be:

$$\begin{array}{llll} 1 & r_{t_1} & \leftarrow & r_i \times r_j \\ 2 & cc & \leftarrow & f_x(r_i, r_j) \\ 3 & r_{t_2} & \leftarrow & r_{t_1} \times r_k \\ 4 & cc & \leftarrow & f_+(r_{t_1}, r_k) \end{array}$$

If the target ISA proposes FMA instructions (fused multiply-add) then line 2 in the previous example is a dead expression. Otherwise, the expander must keep each individual *cc* assignment.

Handling Control-Flow Operations

There is an easy way to handle control-flow operations (i.e. branches):

Handling Control-Flow Operations

There is an easy way to handle control-flow operations (i.e. branches): Clear the instruction window when reaching a branch!
There are other (better) ways: Let the simplifier examine the surroundings of a branch. It produces potentially better code:

- It introduces some additional complexity to the algorithm (need to add special cases)
- BUT it leads to:
  - → Block merging
  - → Dead code elimination

Physical versus Logical Windows

So far, we have only considered *physical* windows. In a given (physical) instruction window, a sequence of instructions may feature unrelated instructions (e.g. middle-end optimizations made it happen to better exploit ILP).

Logical windows bring adjacent “def-use” chains of a (set of) variable(s). $\Rightarrow$ “low-level” dataflow analysis.
There are several problems to implement logical instruction windows:

- Def-Use chains may be found across basic blocks
- The simplifier must know that a single definition may produce multiple uses

Physical versus Logical Windows

So far, we have only considered *physical* windows. In a given (physical) instruction window, a sequence of instructions may feature unrelated instructions (e.g. middle-end optimizations made it happen to better exploit ILP).

Logical windows bring adjacent “def-use” chains of a (set of) variable(s). $\Rightarrow$ “low-level” dataflow analysis.
There are several problems to implement logical instruction windows:

- Def-Use chains may be found across basic blocks
  Reaching Definition Set!
- The simplifier must know that a single definition may produce multiple uses

Physical versus Logical Windows

So far, we have only considered *physical* windows. In a given (physical) instruction window, a sequence of instructions may feature unrelated instructions (e.g. middle-end optimizations made it happen to better exploit ILP).

Logical windows bring adjacent "def-use" chains of a (set of) variable(s). $\Rightarrow$ "low-level" dataflow analysis.
There are several problems to implement logical instruction windows:

- Def-Use chains may be found across basic blocks
  Reaching Definition Set!
- The simplifier must know that a single definition may produce multiple uses
  - $\Rightarrow$ Cannot combine a single definition with a single use (automatically)

Recap

- Instruction selection maps a set of low-level IR operations to possible sequences of ISA instructions.
- The main difficulty is in choosing the *right* sequence (cost-wise)
- We presented two methods to perform instruction selection:
  1. Tree Pattern-Matching
  2. Peephole optimization
- The quality of instruction selection will determine the quality of subsequent instruction scheduling and register allocation.