Instruction Selection
CPEG421/621

Back-End Overview

Two Methods to Perform Instruction Selection
- Tree Pattern-Matching
  - Overview
  - Rewriting Rules for Pattern-Matching
  - Finding a Tiling
  - Peephole Optimization
  - Control-Flow Operations
  - Physical versus Logical Windows

Recap

Back-End Overview

Instruction Selection Overview
From Intermediate Representation to ISA-Specific Instructions

[Diagram of Front-End, Middle-End, and Back-End processes]

Scanning → Parsing → Semantic Analysis → Optimization 1 → ... → Optimization N → Select → Schedule → Allocate
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
Instruction Selection

CPEG421/621

Back-End Overview

Instruction Selection

Instruction Selection Overview
From Intermediate Representation to ISA-Specific Instructions

Two Methods to Perform Instruction Selection

Tree Pattern-Matching Overview
Rewriting Rules for Pattern-Matching
Finding a Tiling
Peephole Optimization
Control-Flow Operations
Physical versus Logical Windows

Recap

Overview

Back-End

Select
Schedule
Allocate

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

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

Back-End

Selection

Recap
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
**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.

**Recap**

- 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

Table: Three-address code

\[
\begin{align*}
t_1 &= 2 \times c \\
t_2 &= b - t_1 \\
a &= t_2
\end{align*}
\]
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. ⇒ 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. ⇒ 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

\[
\begin{align*}
\text{add} & \quad r_i, 0 \quad \Rightarrow \quad r_j \\
\text{sub} & \quad r_i, 0 \quad \Rightarrow \quad r_j \\
\text{and} & \quad r_i, r_i \quad \Rightarrow \quad r_j
\end{align*}
\]
Main Problem of Instruction Selection

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

Examples

\[
\begin{align*}
\text{add} & \quad r_i, 0 \quad \Rightarrow \quad r_j \\
\text{sub} & \quad r_i, 0 \quad \Rightarrow \quad r_j \\
\text{and} & \quad r_i, r_i \quad \Rightarrow \quad r_j \\
\text{mov} & \quad 0 \quad \Rightarrow \quad r_i \\
\text{xor} & \quad r_i, r_i \quad \Rightarrow \quad r_i
\end{align*}
\]
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)
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):

$$B[\beta] = A[\alpha] + \text{SomeValue}$$

Some kind of linear code IR:

```
add $r_0$, @A $\Rightarrow$ $r_1$
add $r_0$, $\alpha$ $\Rightarrow$ $r_2$
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 $r_5$, $\beta$ $\Rightarrow$ $r_6$
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 \texttt{uint64_t} arrays):

\[ B[\beta] = A[\alpha] + \text{SomeValue} \]

Some kind of linear code IR:

- add \( r_0, @A \) \( \Rightarrow \) \( r_1 \)
- add \( r_0, \alpha \) \( \Rightarrow \) \( r_2 \)
- 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 \( r_5, \beta \) \( \Rightarrow \) \( r_6 \)
- mul \( r_6, 8 \) \( \Rightarrow \) \( r_7 \)
- store \( r_{11} \) \( \Rightarrow \) \( r_7 \)

A CISC-like aggregation:

\[ \text{add} \ [\@A + \alpha * 8], r_i \Rightarrow \ [@B + \beta * 8] \]
Two Methods to Perform Instruction Selection

- Tree Pattern-Matching
- Peephole Optimization
The Need for A Good Low-Level IR

Two Methods to Perform Instruction Selection

Tree Pattern-Matching

Rewriting Rules for Pattern-Matching
Finding a Tiling
Peephole Optimization
Control-Flow Operations
Physical versus Logical Windows

Recap
The Need for A Good Low-Level IR

Figure: $a = b - 2 \times c$
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 \(< AST \rightarrow node, op \rightarrow tree >\), where
AST \rightarrow node is a node in the AST, and op \rightarrow tree is an
operation tree (e.g. a tree representation of an operation such as
\( r_i \leftarrow r_j \oplus r_k \), \( \@ \text{mem\_address} \leftarrow r_i \), 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 \(<\text{AST} – \text{node}, \text{op} – \text{tree}>\) 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

```
=  
 |  
|  +  -  
|  |  |  
| Val  Num  Ref * 
| base_address  4  Ref 
|  |  |  
|  |  |  
|  |  |  
| Ref  Ref  Num 
|  |  |  
|  |  |  
|  |  |  
|  |  |  
| +  +  
|  |  |  
|  |  |  
|  |  |  
| Val  Num  Lab  Num 
| base_address  -16 @G 12 
```
Two Methods to Perform Instruction Selection

Tree Pattern-Matching

Overview
Rewriting Rules for Pattern-Matching
Finding a Tiling
Peephole Optimization
Control-Flow Operations
Physical versus Logical Windows

Example of Tiling
Example of Tiling
Instruction Selection

Two Methods to Perform Instruction Selection

Tree Pattern-Matching

Overview

Rewriting Rules for Pattern-Matching

Finding a Tiling

Peephole Optimization

Control-Flow Operations

Physical versus Logical Windows

Recap

Example of Tiling
Two Methods to Perform Instruction Selection

Tree Pattern-Matching

Overview

Rewriting Rules for Pattern-Matching

Finding a Tiling

Peephole Optimization

Control-Flow Operations

Physical versus Logical Windows

Example of Tiling
Example of Tiling

Two Methods to Perform Instruction Selection

- Tree Pattern-Matching
  - Overview
  - Rewriting Rules for Pattern-Matching
  - Finding a Tiling
- Peephole Optimization
  - Control-Flow Operations
  - Physical versus Logical Windows

Recap
Example of Tiling

Two Methods to Perform Instruction Selection
- Tree Pattern-Matching
- Rewriting Rules for Pattern-Matching
- Finding a Tiling
- Peephole Optimization
- Control-Flow Operations
- Physical versus Logical Windows

Recap
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>
<thead>
<tr>
<th>Production</th>
<th>Cost</th>
<th>Code Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Goal → Assign</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2 Assign → ← (Reg₁, Reg₂)</td>
<td>1</td>
<td>store r₂ ⇒ r₁</td>
</tr>
<tr>
<td>3 Assign → ← (+ (Reg₁, Reg₂), Reg₃)</td>
<td>1</td>
<td>storeAO r₃ ⇒ r₁, r₂</td>
</tr>
<tr>
<td>4 Assign → ← (+ (Reg₁, Num₂), Reg₃)</td>
<td>1</td>
<td>storeAl r₃ ⇒ r₁, n₂</td>
</tr>
<tr>
<td>5 Assign → ← (+ (Num₁, Reg₂), Reg₃)</td>
<td>1</td>
<td>storeAl r₃ ⇒ n₁, r₂</td>
</tr>
<tr>
<td>6 Reg → Lab₁</td>
<td>1</td>
<td>load l₁ ⇒ rₙew</td>
</tr>
<tr>
<td>7 Reg → Val₁</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8 Reg → Num₁</td>
<td>1</td>
<td>load l₁ ⇒ rₙew</td>
</tr>
<tr>
<td>9 Reg → ♦ (Reg₁)</td>
<td>1</td>
<td>load r₁ ⇒ rₙew</td>
</tr>
<tr>
<td>10 Reg → ♦ (+ (Reg₁, Reg₂))</td>
<td>1</td>
<td>loadAO r₁, r₂ ⇒ rₙew</td>
</tr>
<tr>
<td>11 Reg → ♦ (+ (Reg₁, Num₂))</td>
<td>1</td>
<td>loadAl r₁, n₂ ⇒ rₙew</td>
</tr>
<tr>
<td>12 Reg → ♦ (+ (Num₁, Reg₂))</td>
<td>1</td>
<td>loadAl r₂, n₁ ⇒ rₙew</td>
</tr>
<tr>
<td>13 Reg → ♦ (+ (Reg₁, Lab₂))</td>
<td>1</td>
<td>loadAl r₁, l₂ ⇒ rₙew</td>
</tr>
<tr>
<td>14 Reg → ♦ (+ (Lab₁, Reg₂))</td>
<td>1</td>
<td>loadAl r₂, l₁ ⇒ rₙew</td>
</tr>
<tr>
<td>15 Reg → + (Reg₁, Reg₂)</td>
<td>1</td>
<td>add r₁, r₂ ⇒ rₙew</td>
</tr>
<tr>
<td>16 Reg → + (Reg₁, Num₂)</td>
<td>1</td>
<td>add l₁, n₂ ⇒ rₙew</td>
</tr>
<tr>
<td>17 Reg → + (Num₁, Reg₂)</td>
<td>1</td>
<td>add r₂, n₁ ⇒ rₙew</td>
</tr>
<tr>
<td>18 Reg → + (Reg₁, Lab₂)</td>
<td>1</td>
<td>add r₁, l₂ ⇒ rₙew</td>
</tr>
<tr>
<td>19 Reg → + (Lab₁, Reg₂)</td>
<td>1</td>
<td>add r₂, l₁ ⇒ rₙew</td>
</tr>
<tr>
<td>20 Reg → − (Reg₁, Reg₂)</td>
<td>1</td>
<td>sub r₂, r₂ ⇒ rₙew</td>
</tr>
<tr>
<td>21 Reg → − (Reg₁, Num₂)</td>
<td>1</td>
<td>subl r₁, n₂ ⇒ rₙew</td>
</tr>
<tr>
<td>22 Reg → − (Num₁, Reg₂)</td>
<td>1</td>
<td>subl r₂, n₁ ⇒ rₙew</td>
</tr>
<tr>
<td>20 Reg → × (Reg₁, Reg₂)</td>
<td>1</td>
<td>mul r₂, r₂ ⇒ rₙew</td>
</tr>
<tr>
<td>21 Reg → × (Reg₁, Num₂)</td>
<td>1</td>
<td>mul r₁, n₂ ⇒ rₙew</td>
</tr>
<tr>
<td>22 Reg → × (Num₁, Reg₂)</td>
<td>1</td>
<td>mul r₂, n₁ ⇒ rₙew</td>
</tr>
</tbody>
</table>
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 → 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.

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

Within 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

Within 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

\[
\begin{align*}
r_{10} & : 2 \\
r_{11} & : @G \\
r_{12} & : 12
\end{align*}
\]
Sequences Produced by the Simplerifier (Taken from Cooper & Torczon)

Let’s assume a 3-instruction window.

**Sequence 1**

\[
\begin{align*}
  r_{10} & \quad 2 \\
  r_{11} & \quad @G \\
  r_{12} & \quad 12
\end{align*}
\]

**Sequence 2**

\[
\begin{align*}
  r_{11} & \quad @G \\
  r_{12} & \quad 12 \\
  r_{13} & \quad r_{11} + r_{12}
\end{align*}
\]
Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let’s assume a 3-instruction window.

Sequence 1

\[
\begin{align*}
  r_{10} & \quad 2 \\
  r_{11} & \quad @G \\
  r_{12} & \quad 12
\end{align*}
\]

Sequence 2

\[
\begin{align*}
  r_{11} & \quad @G \\
  r_{12} & \quad 12 \\
  r_{13} & \quad r_{11} + r_{12}
\end{align*}
\]

Sequence 3

\[
\begin{align*}
  r_{11} & \quad @G \\
  r_{13} & \quad r_{11} + 12 \\
  r_{14} & \quad M(r_{13})
\end{align*}
\]

Sequence 4

\[
\begin{align*}
  r_{11} & \quad @G \\
  r_{14} & \quad M(r_{11} + 12) \\
  r_{15} & \quad r_{10} \times r_{14}
\end{align*}
\]
Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let’s assume a 3-instruction window.

**Sequence 1**

\[
\begin{align*}
r_{10} & \quad 2 \\
r_{11} & \quad @G \\
r_{12} & \quad 12 
\end{align*}
\]

**Sequence 2**

\[
\begin{align*}
r_{11} & \quad @G \\
r_{12} & \quad 12 \\
r_{13} & \quad r_{11} + r_{12} 
\end{align*}
\]

**Sequence 3**

\[
\begin{align*}
r_{11} & \quad @G \\
r_{13} & \quad r_{11} + 12 \\
r_{14} & \quad M(r_{13}) 
\end{align*}
\]

**Sequence 4**

\[
\begin{align*}
r_{11} & \quad @G \\
r_{14} & \quad M(r_{11} + 12) \\
r_{15} & \quad r_{10} \times r_{14} 
\end{align*}
\]
Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let’s assume a 3-instruction window.

Sequence 5

\[ r_{14} \quad M(r_{11} + 12) \]
\[ r_{15} \quad r_{10} \times r_{14} \]
\[ r_{16} \quad -16 \]
Sequences Produced by the Simpler (Taken from Cooper & Torczon)

Let’s assume a 3-instruction window.

**Sequence 5**

\[
\begin{align*}
    r_{14} & \quad M(r_{11} + 12) \\
    r_{15} & \quad r_{10} \times r_{14} \\
    r_{16} & \quad -16
\end{align*}
\]

**Sequence 6**

\[
\begin{align*}
    r_{15} & \quad r_{10} \times r_{14} \\
    r_{16} & \quad -16 \\
    r_{17} & \quad r_{fp} + r_{16}
\end{align*}
\]
Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let’s assume a 3-instruction window.

Sequence 5

\[
\begin{align*}
 r_{14} & \quad M(r_{11} + 12) \\
 r_{15} & \quad r_{10} \times r_{14} \\
 r_{16} & \quad -16
\end{align*}
\]

Sequence 6

\[
\begin{align*}
 r_{15} & \quad r_{10} \times r_{14} \\
 r_{16} & \quad -16 \\
 r_{17} & \quad r_{fp} + r_{16}
\end{align*}
\]

Sequence 7

\[
\begin{align*}
 r_{15} & \quad r_{10} \times r_{14} \\
 r_{17} & \quad r_{fp} - 16 \\
 r_{18} & \quad M(r_{17})
\end{align*}
\]
Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let’s assume a 3-instruction window.

**Sequence 5**

\[ r_{14} \quad M(r_{11} + 12) \]
\[ r_{15} \quad r_{10} \times r_{14} \]
\[ r_{16} \quad -16 \]

**Sequence 7**

\[ r_{15} \quad r_{10} \times r_{14} \]
\[ r_{17} \quad r_{fp} - 16 \]
\[ r_{18} \quad M(r_{17}) \]

**Sequence 6**

\[ r_{15} \quad r_{10} \times r_{14} \]
\[ r_{16} \quad -16 \]
\[ r_{17} \quad r_{fp} + r_{16} \]

**Sequence 8**

\[ r_{15} \quad r_{10} \times r_{14} \]
\[ r_{18} \quad M(r_{fp} - 16) \]
\[ r_{19} \quad M(r_{18}) \]
Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let’s assume a 3-instruction window.

Sequence 9

\[
\begin{align*}
  r_{18} & \quad M(r_{fp} - 16) \\
  r_{19} & \quad M(r_{18}) \\
  r_{20} & \quad r_{19} - r_{15}
\end{align*}
\]
Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let’s assume a 3-instruction window.

**Sequence 9**

\[
\begin{align*}
    r_{18} & \quad M(r_{fp} - 16) \\
    r_{19} & \quad M(r_{18}) \\
    r_{20} & \quad r_{19} - r_{15}
\end{align*}
\]

**Sequence 10**

\[
\begin{align*}
    r_{19} & \quad M(r_{18}) \\
    r_{20} & \quad r_{19} - r_{15} \\
    r_{21} & \quad 4
\end{align*}
\]
Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let’s assume a 3-instruction window.

Sequence 9

\[
\begin{align*}
& r_{18} & M(r_{fp} - 16) \\
& r_{19} & M(r_{18}) \\
& r_{20} & r_{19} - r_{15}
\end{align*}
\]

Sequence 10

\[
\begin{align*}
& r_{19} & M(r_{18}) \\
& r_{20} & r_{19} - r_{15} \\
& r_{21} & 4
\end{align*}
\]

Sequence 11

\[
\begin{align*}
& r_{20} & r_{19} - r_{15} \\
& r_{21} & 4 \\
& r_{22} & r_{fp} + r_{21}
\end{align*}
\]
Sequences Produced by the Simplifier (Taken from Cooper & Torczon)

Let’s assume a 3-instruction window.

**Sequence 9**

\[
\begin{align*}
    r_{18} & \quad M(r_{fp} - 16) \\
    r_{19} & \quad M(r_{18}) \\
    r_{20} & \quad r_{19} - r_{15}
\end{align*}
\]

**Sequence 10**

\[
\begin{align*}
    r_{19} & \quad M(r_{18}) \\
    r_{20} & \quad r_{19} - r_{15} \\
    r_{21} & \quad 4
\end{align*}
\]

**Sequence 11**

\[
\begin{align*}
    r_{20} & \quad r_{19} - r_{15} \\
    r_{21} & \quad 4 \\
    r_{22} & \quad r_{fp} + r_{21}
\end{align*}
\]

**Sequence 12**

\[
\begin{align*}
    r_{20} & \quad r_{19} - r_{15} \\
    r_{22} & \quad r_{fp} + 4 \\
    M(r_{22}) & \quad r_{20}
\end{align*}
\]
Example (Taken from Cooper & Torczon)

<table>
<thead>
<tr>
<th>Op</th>
<th>Arg₁</th>
<th>Arg₂</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>2</td>
<td>c</td>
<td>t₁</td>
</tr>
<tr>
<td>−</td>
<td>b</td>
<td>t₁</td>
<td>a</td>
</tr>
</tbody>
</table>
Example (Taken from Cooper & Torczon)

<table>
<thead>
<tr>
<th>Op</th>
<th>Arg₁</th>
<th>Arg₂</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>2</td>
<td>c</td>
<td>t₁</td>
</tr>
<tr>
<td>−</td>
<td>b</td>
<td>t₁</td>
<td>a</td>
</tr>
</tbody>
</table>

1. Expansion

\[
\begin{align*}
    r_{10} & \leftarrow 2 \\
    r_{11} & \leftarrow @G \\
    r_{12} & \leftarrow 12 \\
    r_{13} & \leftarrow r_{11} + r_{12} \\
    r_{14} & \leftarrow M(R_{13}) \\
    r_{15} & \leftarrow r_{10} \times r_{14} \\
    r_{16} & \leftarrow -16 \\
    r_{17} & \leftarrow r_{fp} + r_{16} \\
    r_{18} & \leftarrow M(r_{17}) \\
    r_{19} & \leftarrow M(r_{18}) \\
    r_{20} & \leftarrow r_{19} - r_{15} \\
    r_{21} & \leftarrow 4 \\
    r_{22} & \leftarrow r_{fp} + r_{21} \\
    M(r_{22}) & \leftarrow r_{20}
\end{align*}
\]
Example (Taken from Cooper & Torczon)

1. Expansion

\[
\begin{align*}
   r_{10} & \leftarrow 2 \\
   r_{11} & \leftarrow @G \\
   r_{12} & \leftarrow 12 \\
   r_{13} & \leftarrow r_{11} + r_{12} \\
   r_{14} & \leftarrow M(R_{13}) \\
   r_{15} & \leftarrow r_{10} \times r_{14} \\
   r_{16} & \leftarrow -16 \\
   r_{17} & \leftarrow r_{fp} + r_{16} \\
   r_{18} & \leftarrow M(r_{17}) \\
   r_{19} & \leftarrow M(r_{18}) \\
   r_{20} & \leftarrow r_{19} - r_{15} \\
   r_{21} & \leftarrow 4 \\
   r_{22} & \leftarrow r_{fp} + r_{21} \\
   M(r_{22}) & \leftarrow r_{20}
\end{align*}
\]
Example (Taken from Cooper & Torczon)

1. Expansion

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

2. Simplify (Sequence 13)

\[
\begin{align*}
    r_{10} & \leftarrow 2 \\
    r_{11} & \leftarrow @G \\
    r_{14} & \leftarrow M(r_{13} + 12) \\
    r_{15} & \leftarrow r_{10} \times r_{14} \\
    r_{18} & \leftarrow r_{fp} - 16 \\
    r_{19} & \leftarrow M(r_{18}) \\
    r_{20} & \leftarrow r_{19} - r_{15} \\
    M(r_{fp} + 4) & \leftarrow r_{20}
\end{align*}
\]
Example (Taken from Cooper & Torczon)

2. Simplify

\[
\begin{align*}
    r_{10} & \leftarrow 2 \\
    r_{11} & \leftarrow @G \\
    r_{14} & \leftarrow M(r_{13} + 12) \\
    r_{15} & \leftarrow r_{10} \times r_{14} \\
    r_{18} & \leftarrow r_{fp} - 16 \\
    r_{19} & \leftarrow M(r_{18}) \\
    r_{20} & \leftarrow r_{19} - r_{15} \\
    M(r_{fp} + 4) & \leftarrow r_{20}
\end{align*}
\]
Example (Taken from Cooper & Torczon)

2. Simplify

| r_{10} | ← | 2 |
| r_{11} | ← | @G |
| r_{14} | ← | M(r_{13} + 12) |
| r_{15} | ← | r_{10} \times r_{14} |
| r_{18} | ← | r_{fp} – 16 |
| r_{19} | ← | M(r_{18}) |
| r_{20} | ← | r_{19} – r_{15} |
| M(r_{fp} + 4) | ← | r_{20} |

3. Match

| loadI | 2 | ⇒ | r_{10} |
| loadI | @G | ⇒ | r_{11} |
| AI | r_{11},12 | ⇒ | r_{14} |
| mult | r_{10},r_{14} | ⇒ | r_{15} |
| loadAI | r_{fp},–16 | ⇒ | r_{18} |
| load | r_{18} | ⇒ | r_{19} |
| sub | r_{19},r_{15} | ⇒ | r_{20} |
| storeAI | r_{20} | ⇒ | r_{fp},4 |
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 LIVO sets:
   1. Compute a LIVO set for 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 LIVO set, see next slide).
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 set for each basic block
   2. Do a post-order pass on each block

Several solutions:

1. Use LIVO sets:
   1. Compute a LIVO set for 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 \texttt{LIVEOUT} sets:
   1. Compute a \texttt{LIVEOUT} set for each basic block
   2. Do a post-order pass on each block

2. Take advantage of the SSA form
   \rightarrow Identify the names used in more than one basic block
   \textit{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 \texttt{LIVENOW} set, see next slide).
Building the LIVENow Set

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

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:

1. $r_{t1} \leftarrow r_i \times r_j$
2. $cc \leftarrow f_x(r_i, r_j)$
3. $r_{t2} \leftarrow r_{t1} \times r_k$
4. $cc \leftarrow f_+(r_{t1}, r_k)$

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):

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). ⇒ “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). ⇒ “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). ⇒ “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
  ⇒ Cannot combine a single definition with a single use (automatically)
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.