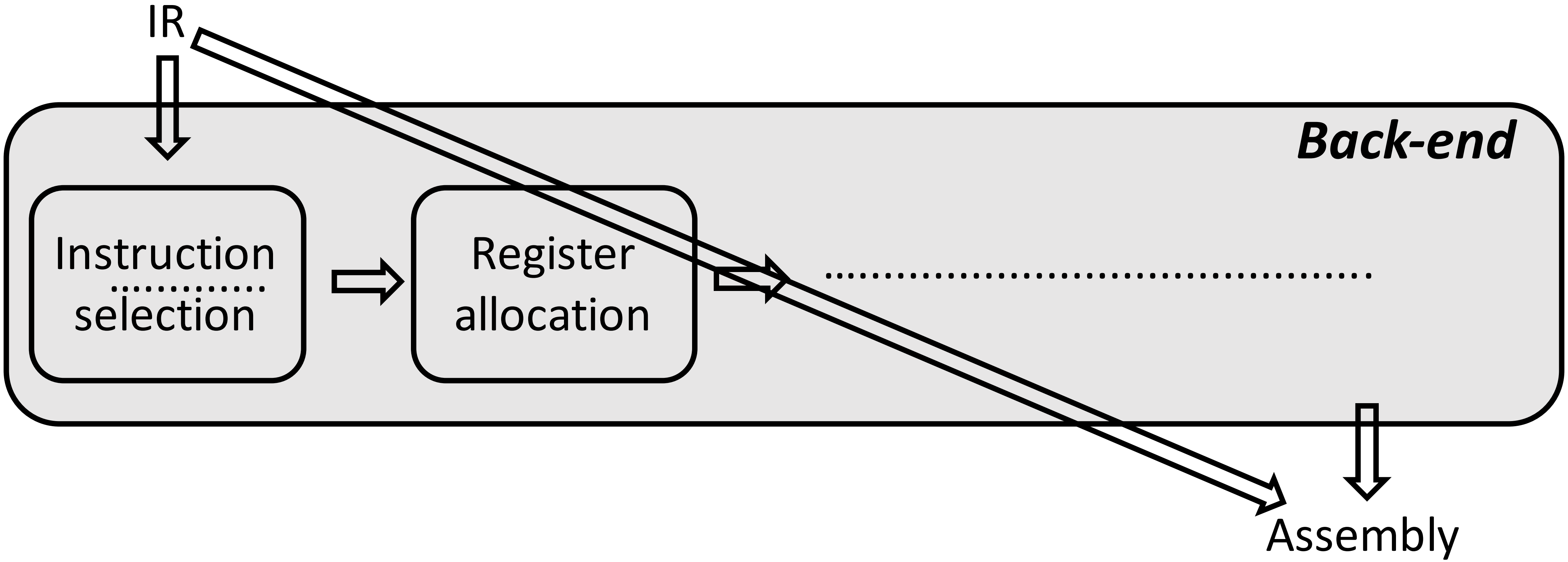
322 and
the missing pieces
of the back-end

EECS 322: Compiler Construction

Simone Campanoni
Robby Findler

5/11/2016

Instruction selection is part of the backend



Example of instruction selection

```
(v1 <- 1)               movq $1, %v1

(v1 <- v2)              movq %v2, %v1

(v2 <- (mem v1 0))      movq 0(%v1), %v2
```

From L1 to x64 assembly

Problem of our current instruction selection

```
(v1 *= 4)
(v2 <- (mem v1 0))
```

```
imulq %v1, $4
movq 0(%v1), %v2
```

… but x64 has

movq 0($4,%v1), %v2

Instruction selection may depend on the context!

The problem of having multiple choices

```
(v1 *= 4)
(v2 <- (mem v1 0))
(v3 <- v1)
```

```
movq 0($4,%v1), %v2
imulq %v1, 4
movq %v1, %v3
```

?

```
imulq %v1, 4
movq 0(%v1), %v2
movq %v1, %v3
```

Instruction selection: it isn't that easy

(v1 *= 5)	imulq %v1, $5
(v2 <- (mem v1 0))	movq 0(%v1), %v2

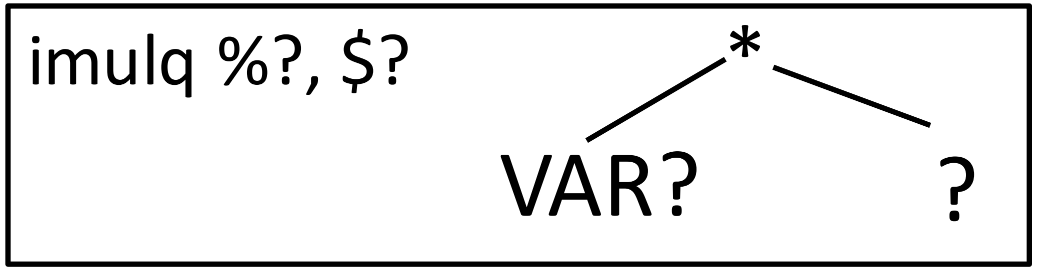
~~**movq 0($5,%v1), %v2**~~

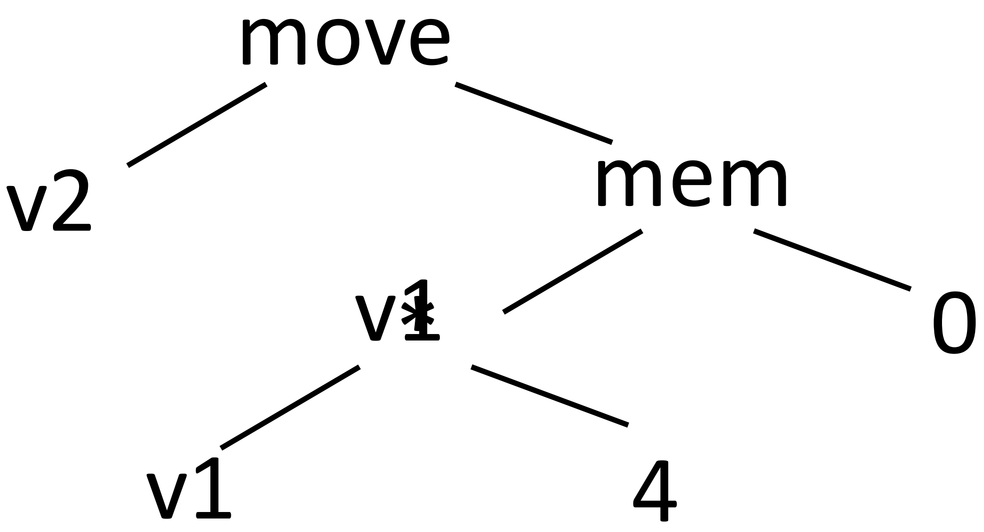
Instruction selection as tree matching

- In order to take context into account, instruction selectors often use pattern-matching on IR trees
  - Use a tree-based IR
  - Each assembly instruction defines a tile (pattern) that can be used to cover the tree
  - Used tiles (patterns) = selected assembly instructions to generate

(v1 *= 4)

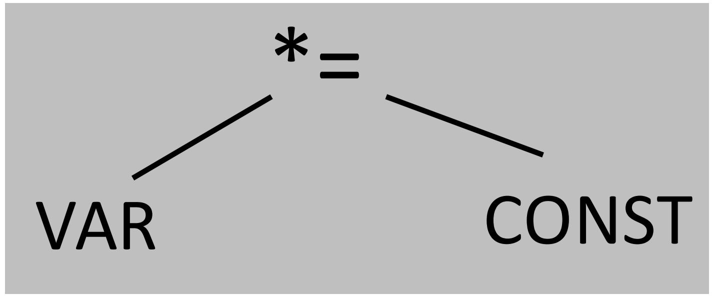
(v2 <- (mem v1 0))



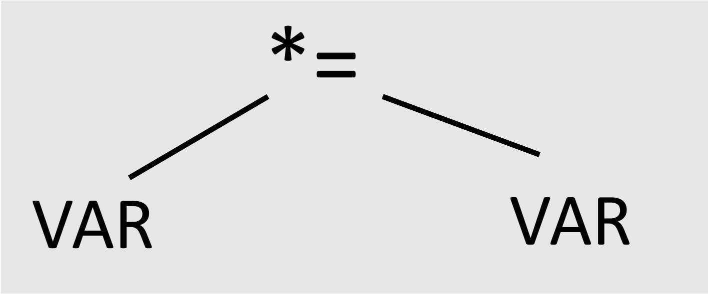


Example: tiles and tiling

- imulq

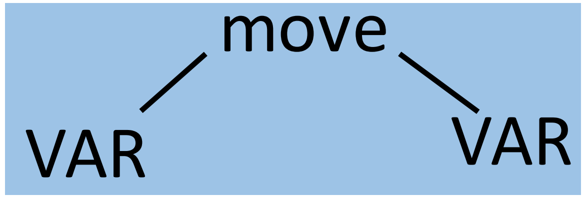


imulq %,$



imulq %,%

- movq

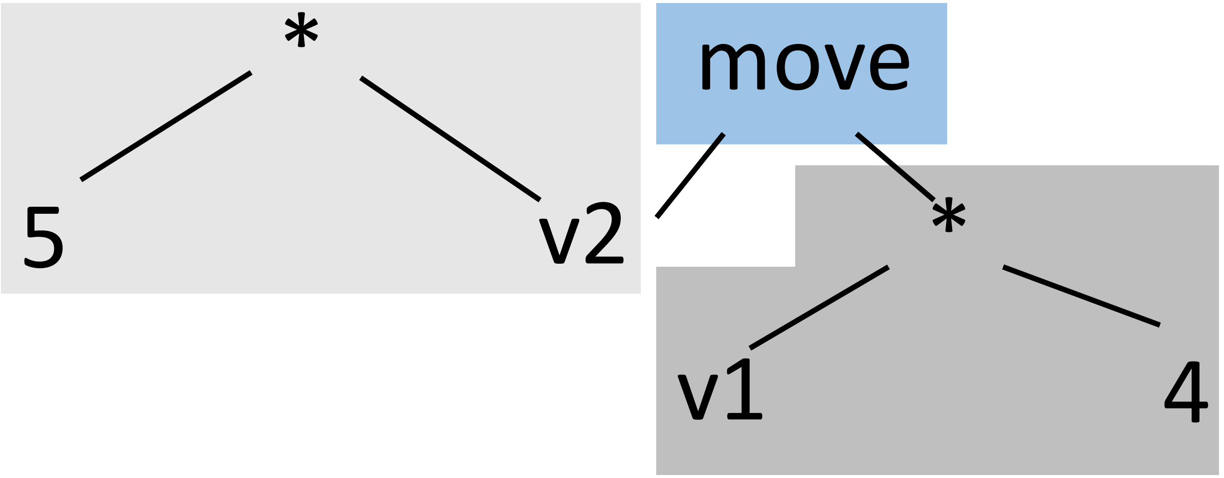


movq %,%

(v1 *= 4)

(v2 <- v1)

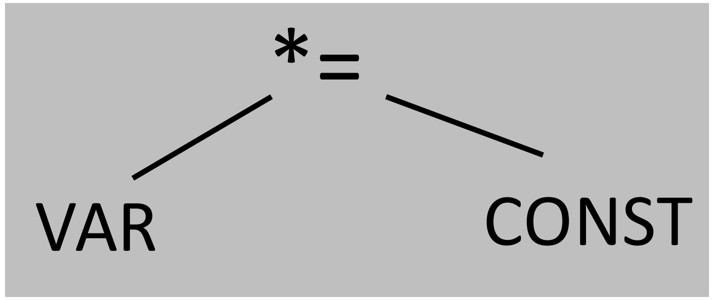
(v2 *= 5)



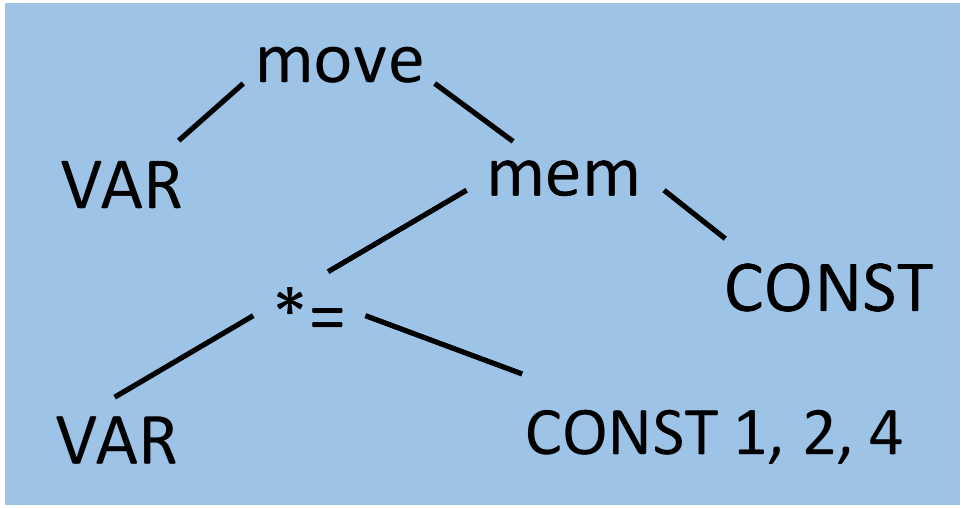
imulq %v1,%4
movq %v1, %v2
imulq %v2, $5

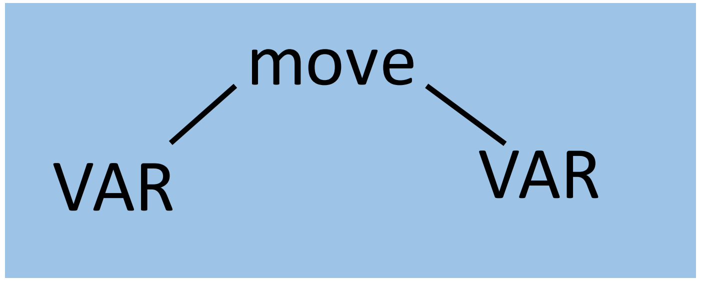
Multiple tiles for an assembly instruction

- imulq



- movq





Multiple tiles for an instruction

- Multiple types of inputs
  - movq %v1, %v2
  - movq 0(%v1), %v2

Tiles and tiling

- Tiles capture compiler's understanding of instruction set
- In general, for any given tree, many tilings are possible
    - Each resulting in a different instruction sequence
- We can ensure pattern coverage by covering, at a minimum, all atomic IR trees

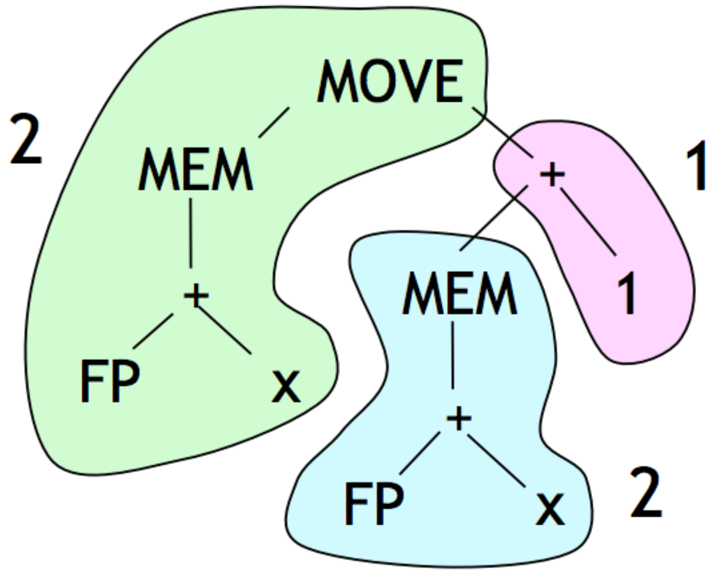
Problem

- How to pick tiles that cover IR statement tree with minimum execution time?
- Need a good selection of tiles
  - Small tiles to make sure we can tile every tree
  - Large tiles for efficiency
- Usually want to pick large tiles: fewer instructions
- Instructions ≠ cycles:
  RISC core instructions take 1 cycle,
  other instructions may take more

Timing model

- Idea: associate cost with each tile
  (proportional to # cycles to execute)
  - Caveat: cost is fictional on modern architectures
- Estimate of total execution time is sum of costs of all tiles

Total cost: 5



Global vs. local optimal solution

- We want the “lowest cost” tiling
  - Take into account cost/delay of each instruction (i.e., timing model)

- **Optimum** tiling: lowest-cost tiling
- **Locally Optimal** tiling: no two adjacent tiles can be combined into one tile of lower cost

Locally optimal tilings

- A simple greedy algorithm works extremely well in practice:
  Maximal munch
- Choose the largest pattern with lowest cost,
  i.e., the “maximal munch”
- Algorithm:
  - Start at root
  - Use “biggest” match (in # of nodes)
    - This is the munch
    - Use cost to break ties
  - Recursively apply maximal much
    at each subtree of this munch

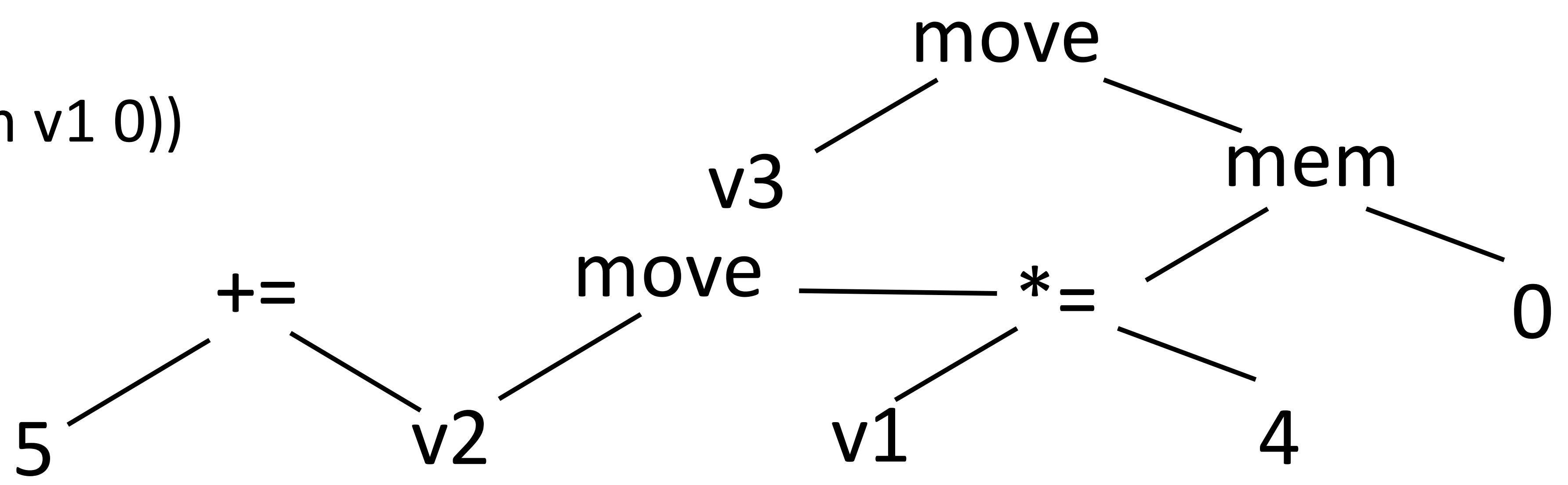
Maximal munch example

(v1 *= 4)

(v2 <- v1)

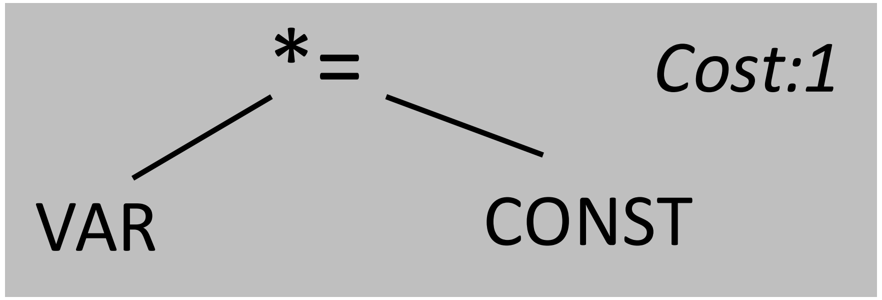
(v2 += 5)

(v3 <- (mem v1 0))



Example: tiles

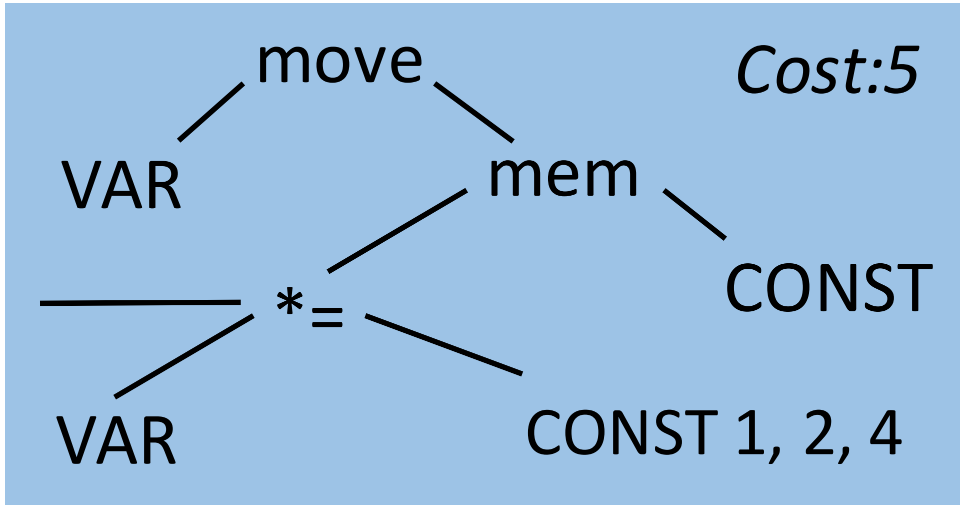
- imulq

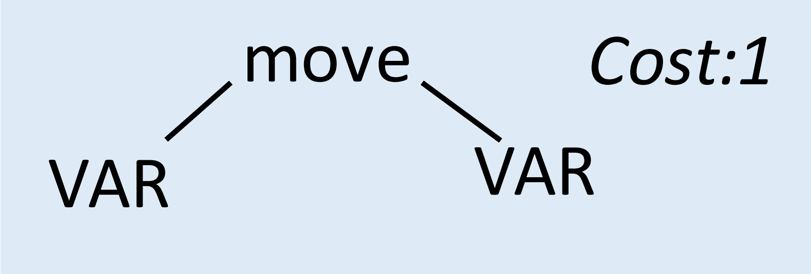


imulq %,$

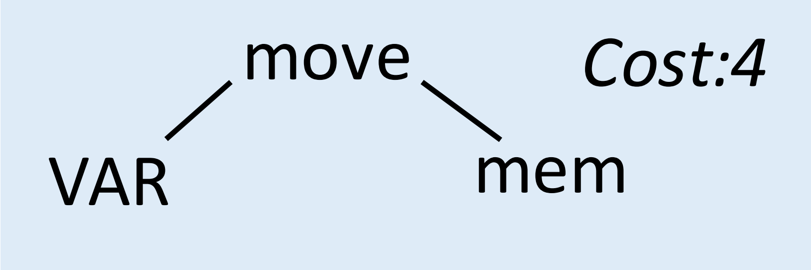
- movq

movq 0($,%), %
imulq %, $





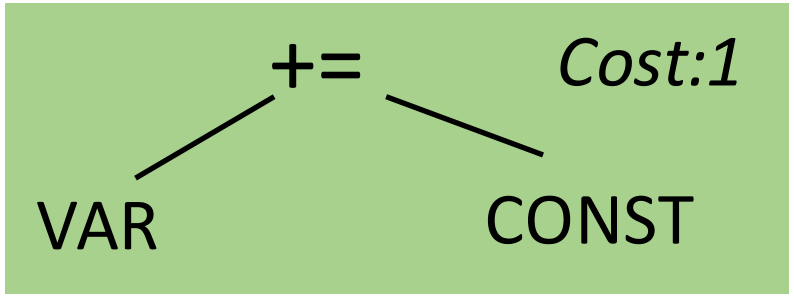
movq %,%



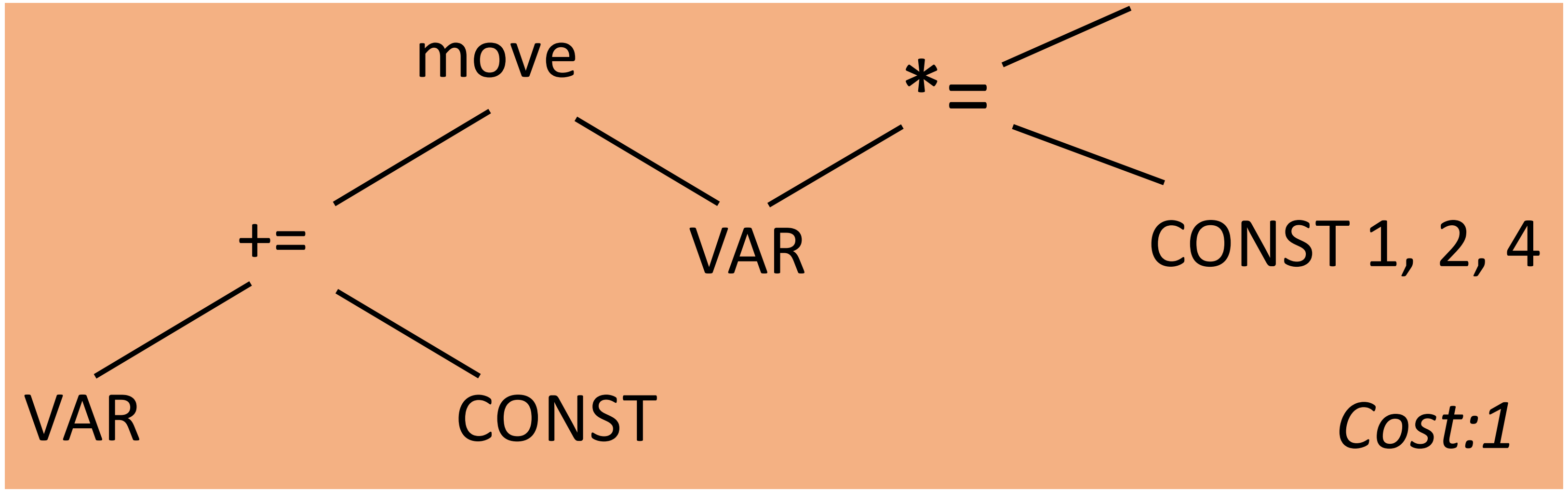
movq (%),%

Example: tiles (2)

- addq

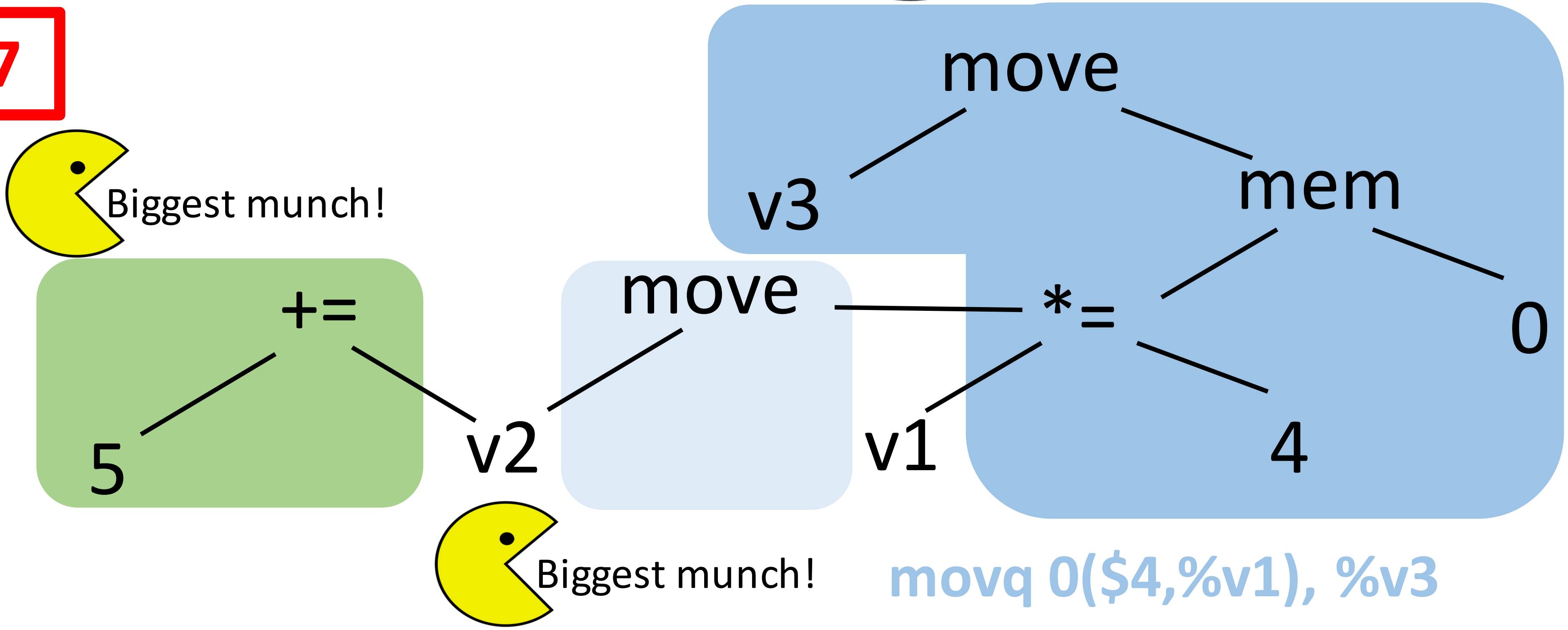


- lea



Maximal munch example

Total cost: 7



(v1 *= 4)

(v2 <- v1)

(v2 += 5)

(v3 <- (mem v1 0))

movq 0($4,%v1), %v3

imulq %v1, $4

movq %v1, %v2

addq %v2, 5

Maximal munch

- Maximal munch does not necessarily produce the optimum selection of instructions

- But:
  - it is easy to implement
  - it tends to work “well” for current instruction-set architectures

… but if we want the optimum?

Finding optimum tiling

- **Goal**: find minimum total cost tiling of tree
- **Algorithm**:
  - For every node, find minimum total cost tiling of that node and sub-tree
- **Lemma**:
  - Once minimum cost tiling of all children of a node is known,
  - We can find minimum cost tiling of the node by trying out all possible tiles matching the node
- **Therefore**: start from leaves, work upward to top node

Optimum selection

- To achieve optimum instruction selection:
  Dynamic programming
- In contrast to maximal munch,
  the trees are matched bottom-up
- But
  - Significantly more complex to implement
  - More time and memory consuming than maximal munch

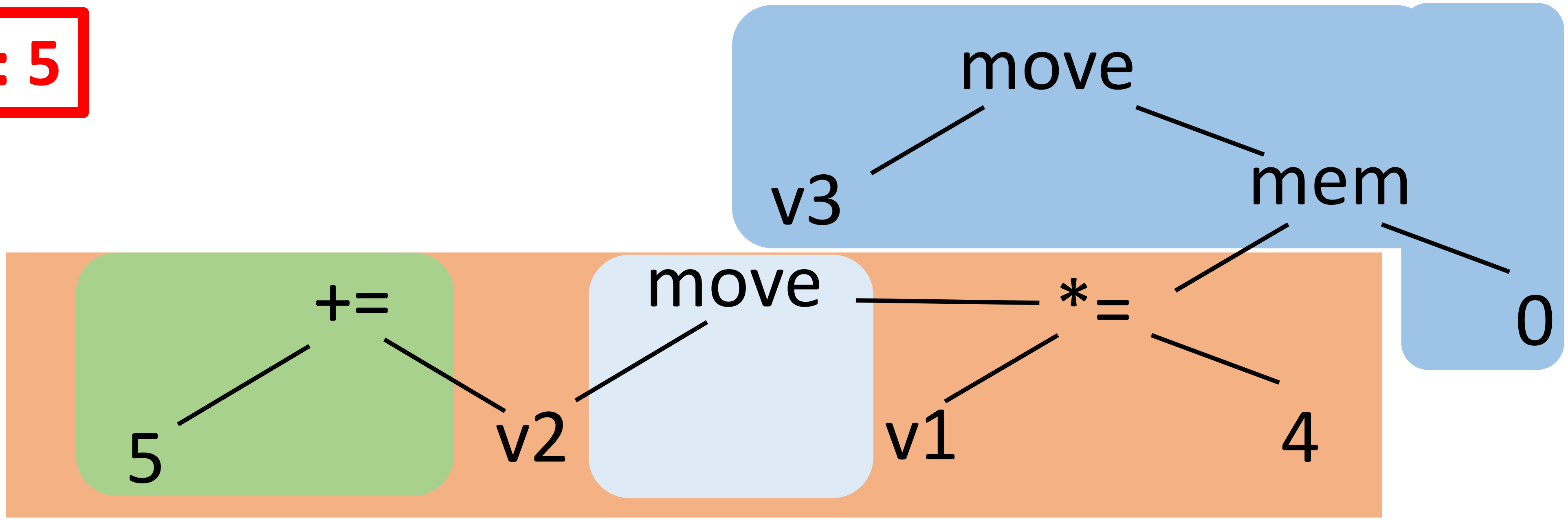
Dynamic programming

- First pass: tiling
  - Working bottom up
  - Given the optimum tilings of all subtrees, generate optimum tiling of the current tree
    - Consider all tiles for the root of the current tree
    - Sum cost of best subtree tiles and each tile
    - Choose tile with minimum total cost

- Second pass: code generation
  - Generates the code using the obtained tiles

Dynamic programming example

Total cost: 5



(v1 *= 4)

(v2 <- v1)

(v2 += 5)

(v3 <- (mem v1 0))

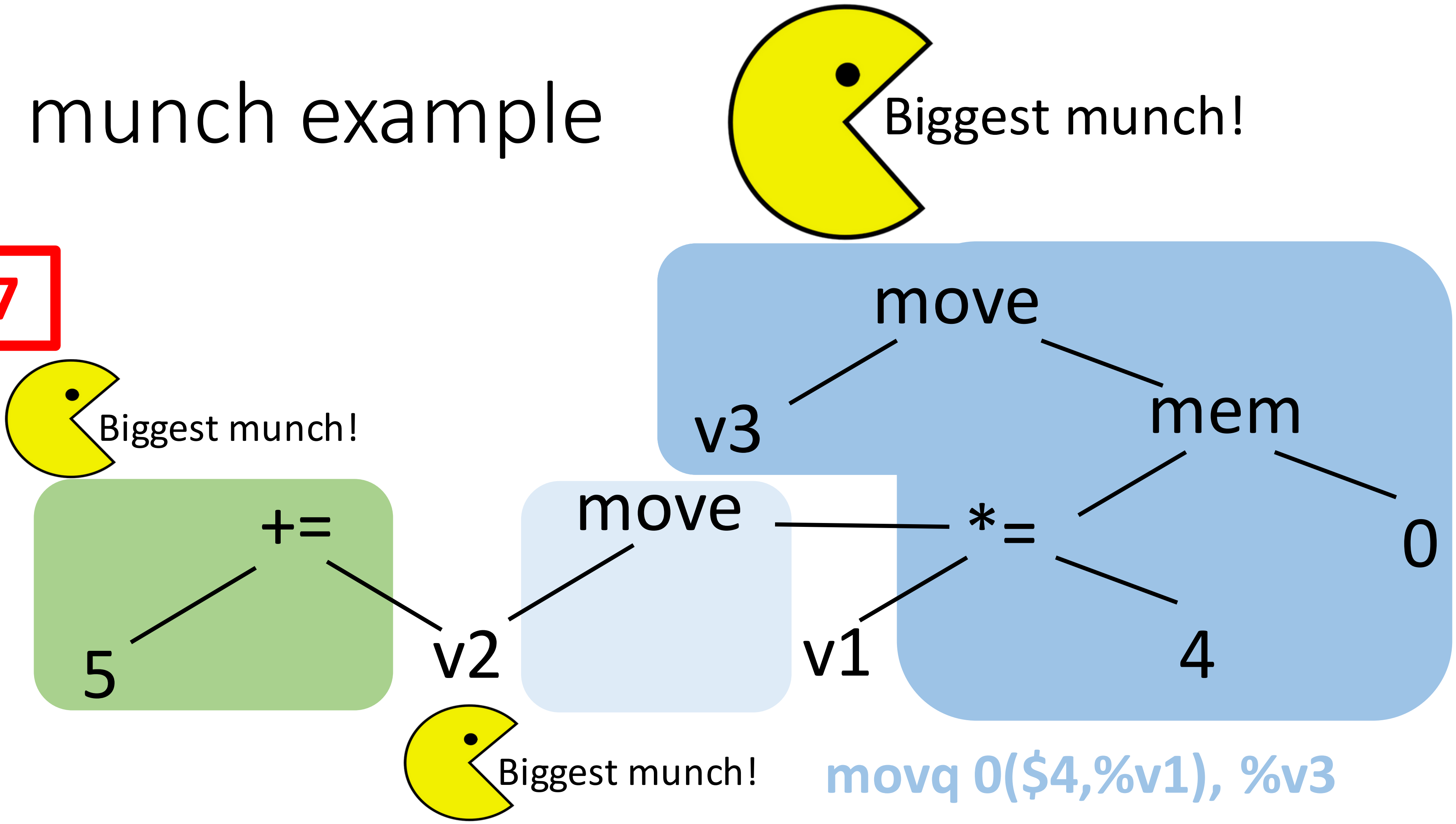
lea (5+%v1*4), %v2

subq %v2, %v1

movq 0(%v1), %v3

Maximal munch example

Total cost: 7



(v1 *= 4)

(v2 <- v1)

(v2 += 5)

(v3 <- (mem v1 0))

movq 0($4,%v1), %v3

imulq %v1, $4

movq %v1, %v2

addq %v2, 5

Value of instruction selection

- The simpler the target ISA is,
  the less important obtaining the optimum is
  - Reduced Instruction Set Computing (RISC)

- The more complex the target ISA is,
  the bigger is the gap between the solution found by
  a simple (e.g., maximal munch) instruction selection and
  the optimum one (e.g., dynamic programming)
  - Complex Instruction Set Computing (CISC)

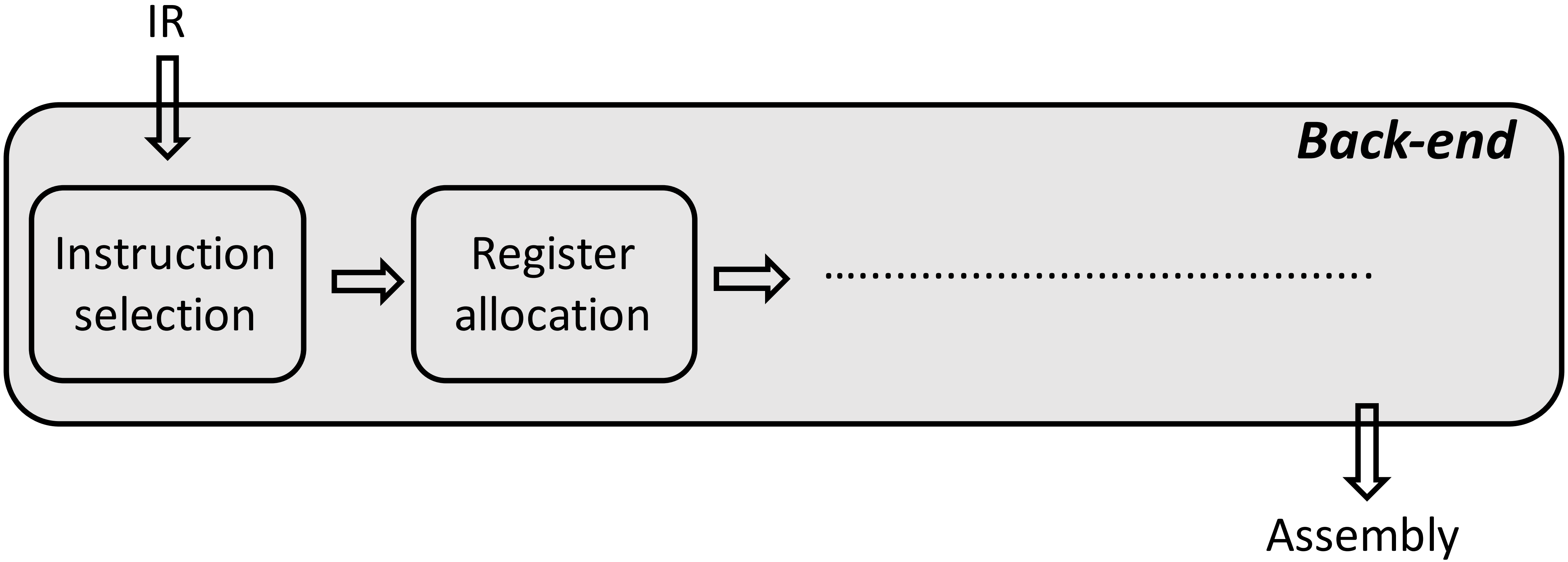
Instruction selection complexity

- Finding the optimum for tree: P

- Finding the optimum for DAG: NP
  - Countless number of heuristics proposed

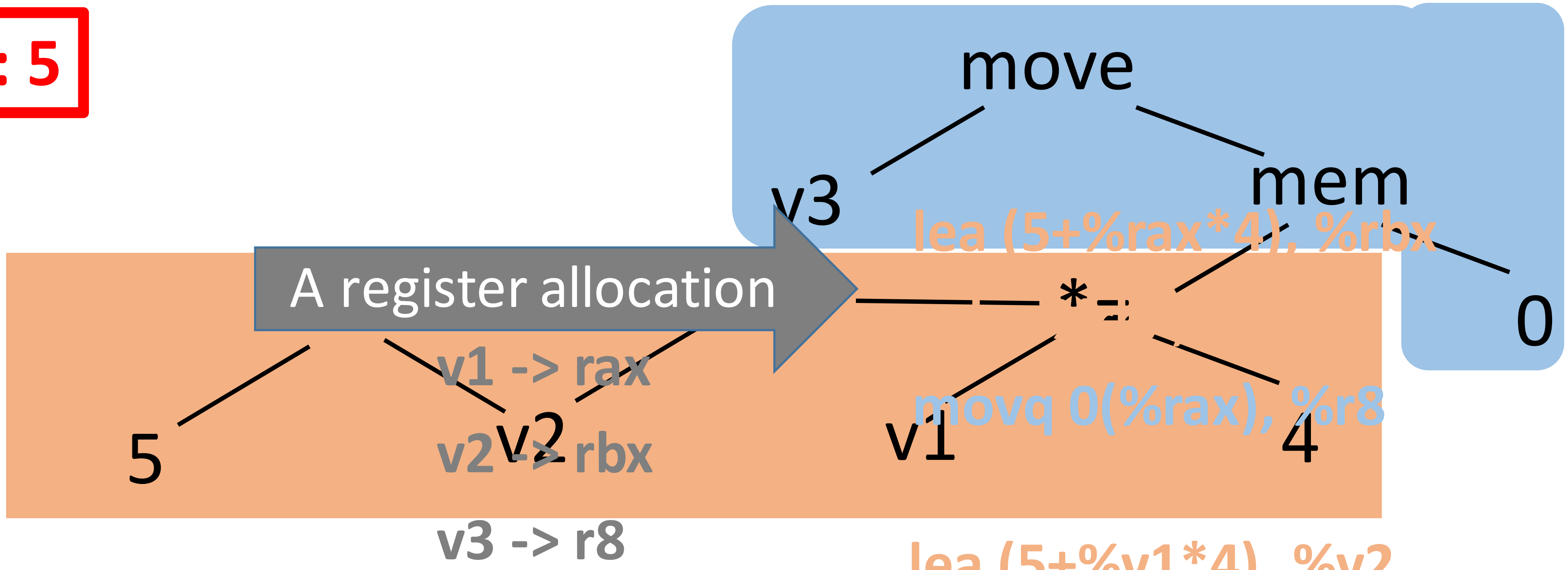
- Most (all) of programs we run are DAGs

Instruction selection is part of the backend



Register allocation after instruction selection

Total cost: 5



(v1 *= 4)

(v2 <- v1)

(v2 += 5)

(v3 <- (mem v1 0))

lea (5+%v1*4), %v2

subq %v2, %v1

movq 0(%v1), %v3

Register allocation after instruction selection

```
lea (5+%v1*4), %v2
subq %v2, %v1
movq 0(%v1), %v3
```

A register allocation

v1 -> rax

v2 -> rbx

v3 -> stack 0

Temporary register

```
lea (5+%rax*4), %rbx
subq %rbx, %rax
movq 0(%rax), %r10
movq %r10, 0(%rsp)
```

v3

Register allocation after instruction selection (2)

```
lea (5+%v1*4), %v2
subq %v2, %v1
movq 0(%v1), %v3
movq %v3, %v4
```

A register allocation

v1 -> rax

v2 -> rbx

v3 -> stack O

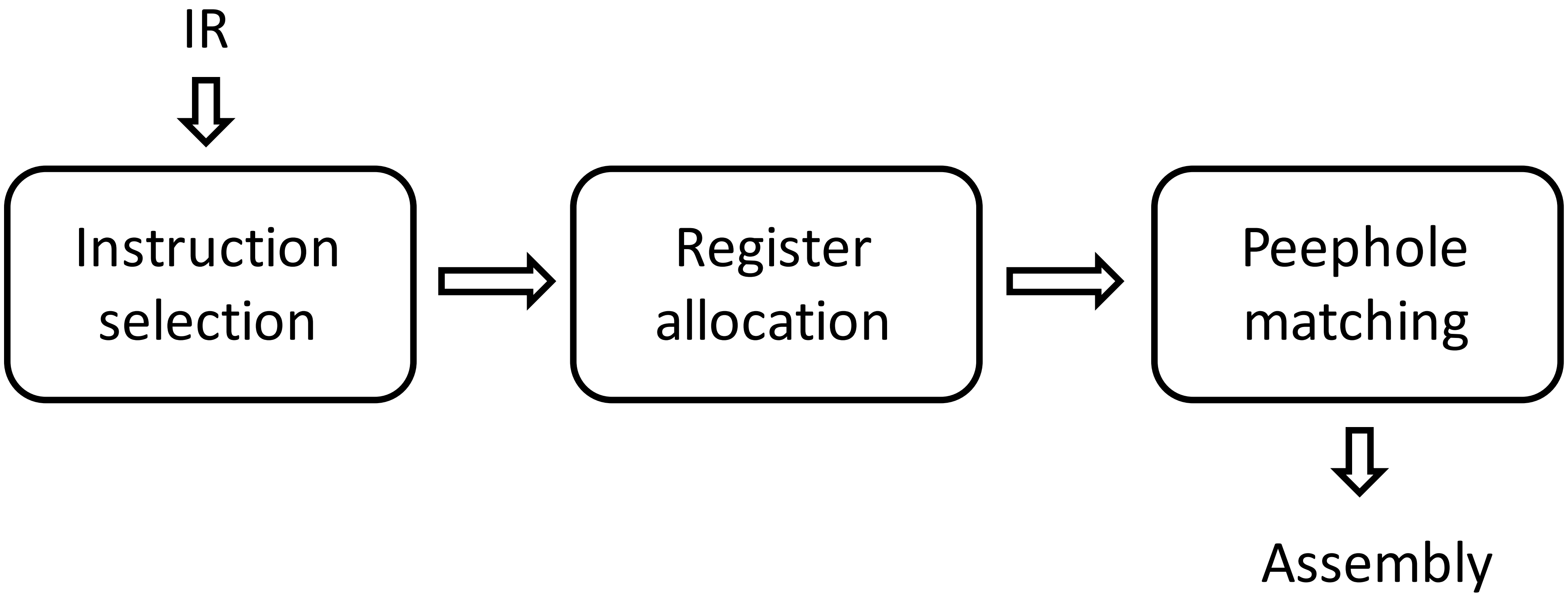
v4 -> r8

```
lea (5+%rax*4), %rbx
subq %rbx, %rax
movq 0(%rax), %r10
movq %r10, O(%rsp)
movq O(%rsp), %r8
```

Peephole matching

Wait, I thought we found the optimum …

Peephole matching



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

- Example: store followed by load

movq %r10, O(%rsp)
movq O(%rsp), %r8

Peephole matching →

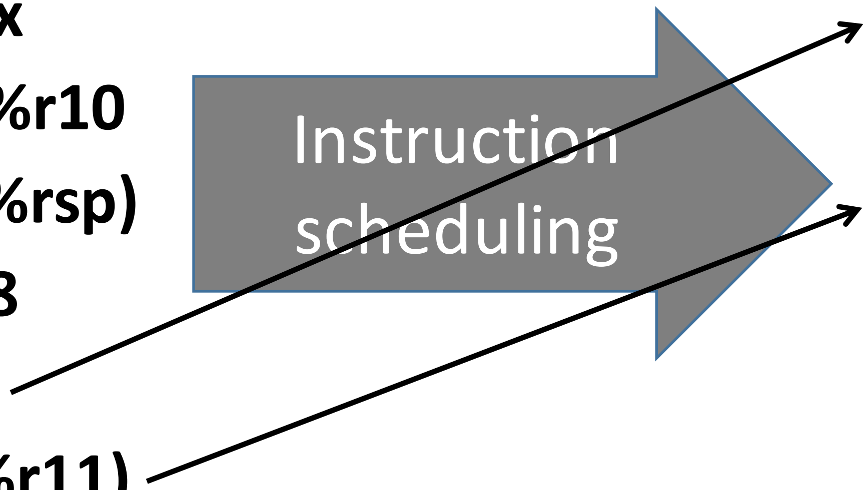
movq %r10, O(%rsp)
Movq %r10, %r8

Are we happy now
with the generated assembly?

Of course NOT!

The problem left

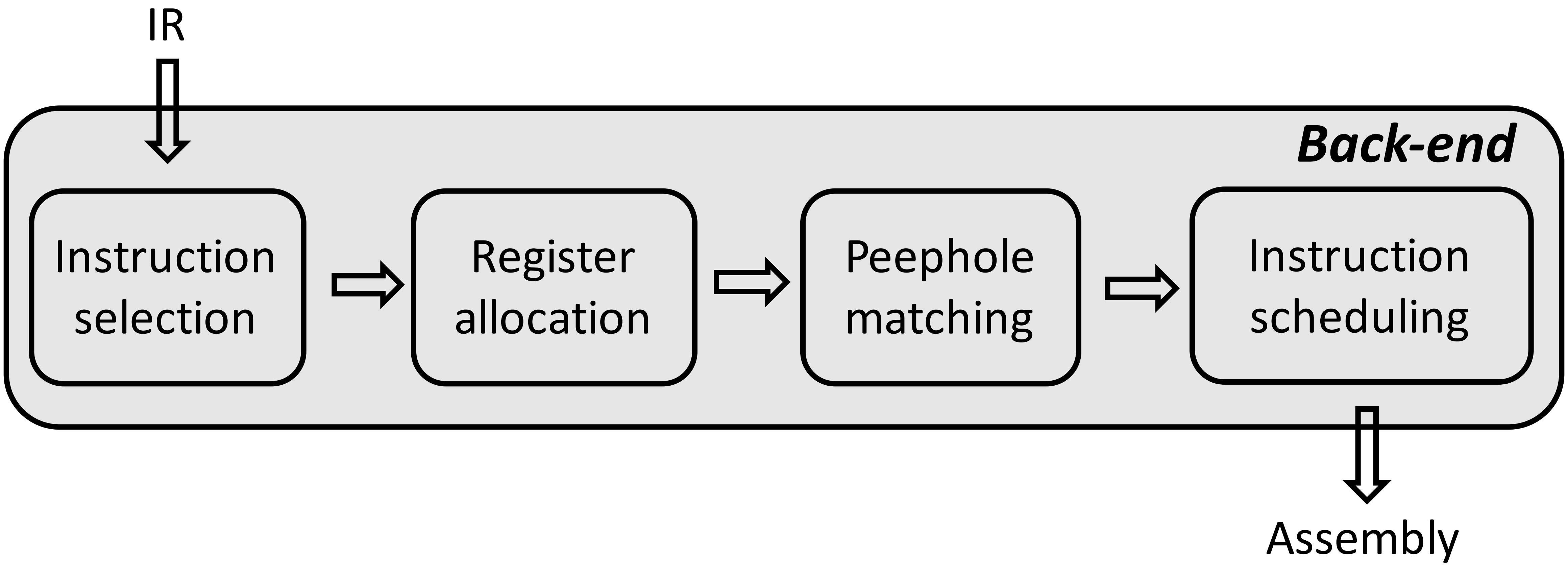
```
lea (5+%rax*4), %rbx
subq %rbx, %rax
movq 0(%rax), %r10
movq %r10, O(%rsp)
movq %r10, %r8
subq %r9, %r10
movq %r10, 0(%r11)
```




```
lea (5+%rax*4), %rbx
subq %r9, %r10
subq %rbx, %rax
movq %r10, 0(%r11)
movq 0(%rax), %r10
movq %r10, O(%rsp)
movq %r10, %r8
```

Is this a better code?

Putting them all together



Thank you!