Norwegian University of Science and Technology

Compiler Construction

Lecture 22: Code generation

Michael Engel

Overview

- Instruction selection
- Register allocation (next lecture)

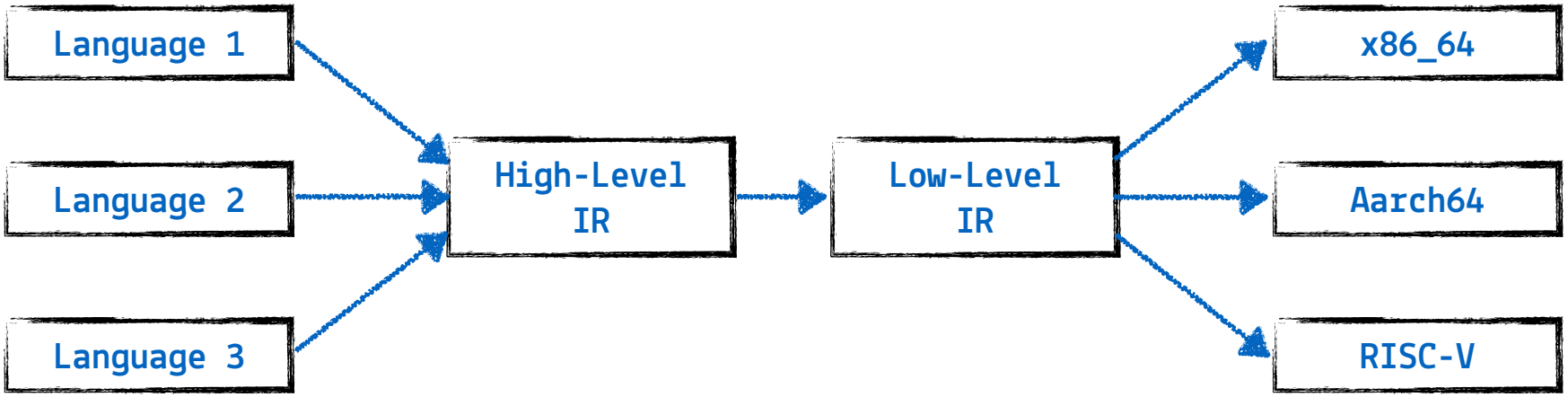
Where are we now?

- We have a fairly low-level view of the program, but
  - It features a memory model of infinite temporary variables
  - It isn't specific in terms of operations provided by the architecture

- These will be our last two topics
  - Selecting machine-specific operations
  - Mapping variables to memory locations

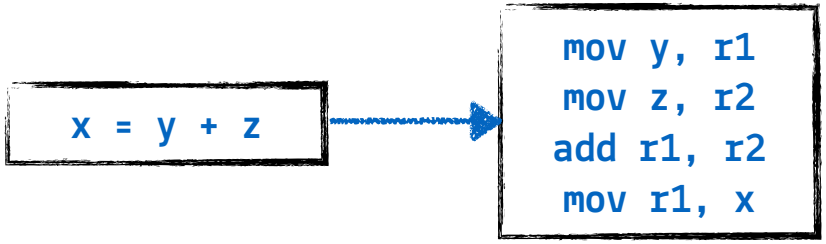
Low-level IR vs. machine level

- The instructions of a low-level IR are not the same as the ones of the target machine



Straight forward solution

- Map every low-level IR to a fixed sequence of assembly instructions



- Disadvantages:
  - Lots of redundant operations
  - More memory traffic than necessary

Multiple possible alternatives

- Translate `a[i+1] = b[j]` using these operations

```
add r2,r1          ← r1 = r1+r2
mul c, r1          ← r1 = r1*c
load r2, r1        ← r1 = *r2
store r2, r1       ← *r1 = r2
movem r2, r1       ← *r1 = *r2
movex r3, r2, r1 ← *r1 = *(r2+r3)
```

General code generation steps

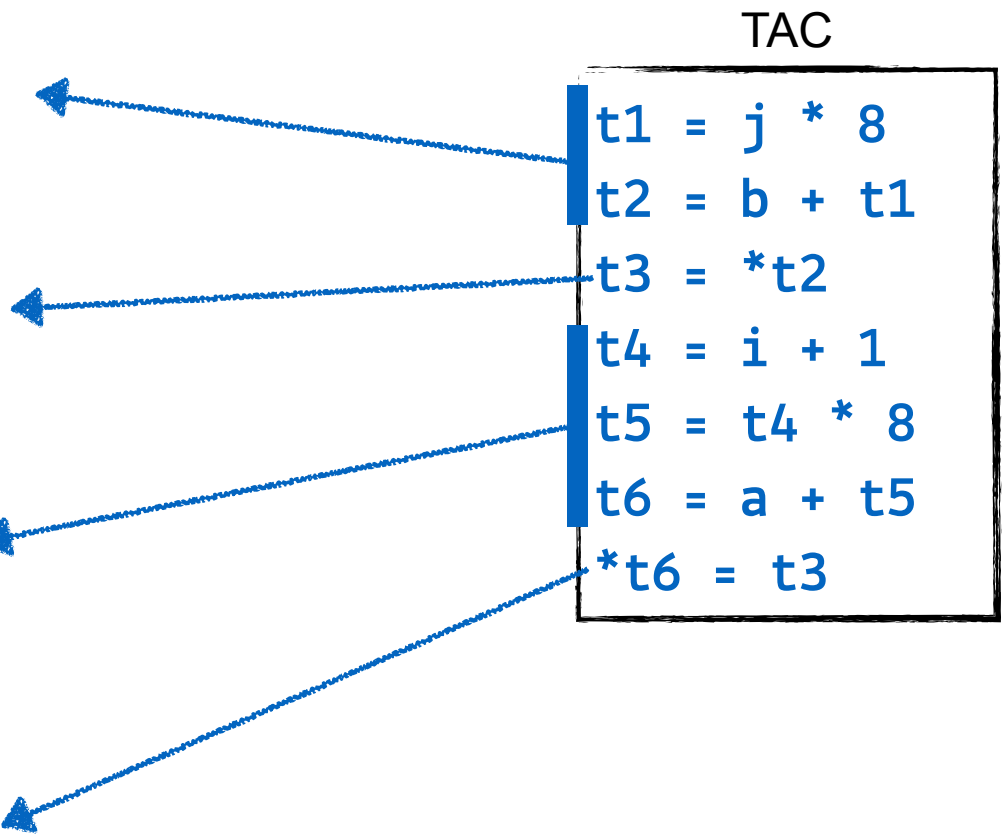
- Let us assume that everything is represented by 8-byte elements, and
  - Register r_a holds `&a`
  - Register r_b holds `&b`
  - Register r_i holds `i`
  - Register r_j holds `j`

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

  - Find address of `b[j]`
  - Load `b[j]`
  - Find address of `a[i+1]`
  - Store into `a[i+1]`

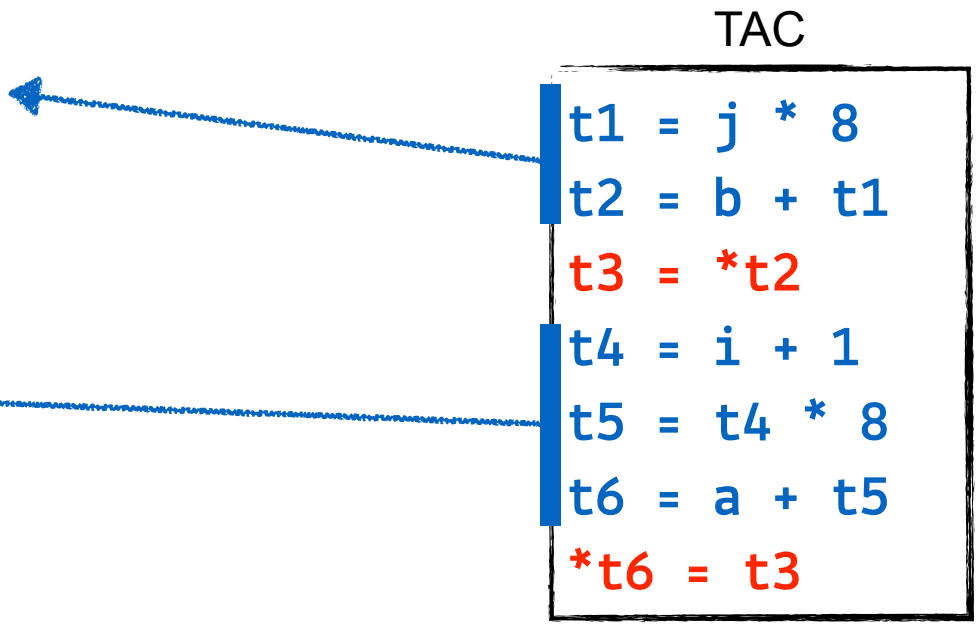
One translation

- Address of `b[j]`
  - `mulc 8, r_j`
  - `add r_j,r_b`
- Load `b[j]`
  - `load r_b,r1`
- Address of `a[i+1]`
  - `add 1,r_i`
  - `mulc 8, r_i`
  - `add r_i, r_a`
- Store into `a[i+1]`
  - `store r1, r_a`



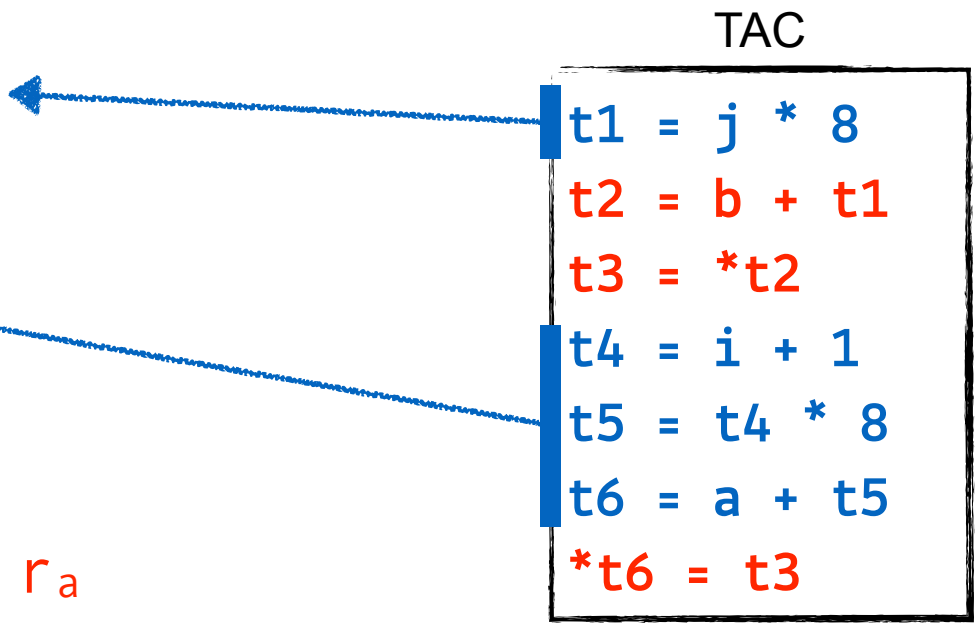
Another possible translation

- Address of `b[j]`
  - `mulc 8, rj`
  - `add rj,rb`
- Address of `a[i+1]`
  - `add 1,ri`
  - `mulc 8, ri`
  - `add ri, ra`
- Store into `a[i+1]`
  - `movem rb, ra`



One more translation

- Address of `b[j]`
  - `mulc 8, r_j`
- Address of `a[i+1]`
  - `add 1,r_i`
  - `mulc 8, r_1`
  - `add r_i, r_a`
- Store into `a[i+1]`
  - `movex r_j, r_b, r_a`



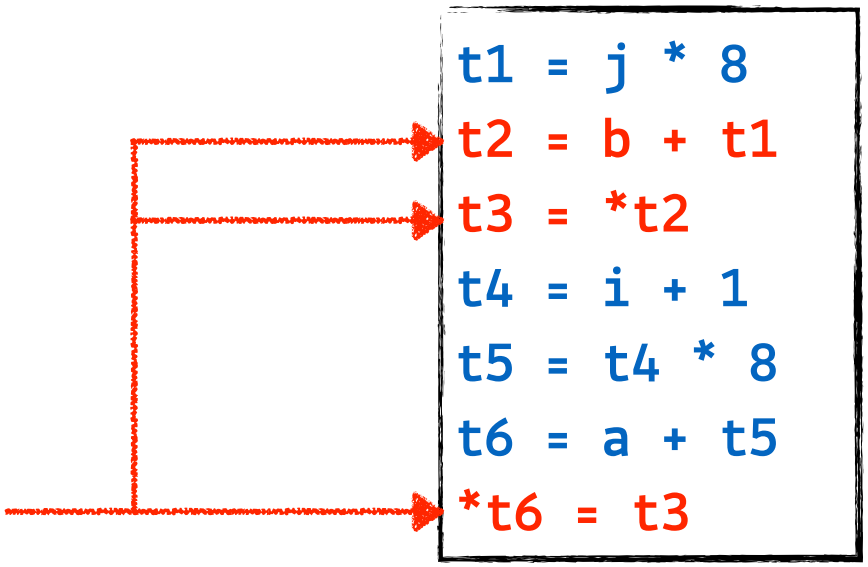
Why should we care?

- Not all instructions are created equal
- Some complete in a clock cycle
- Others decompose into a sequence of steps, and take many cycles

- If we have a choice of translations, we'd like the one with the smallest sum of costs

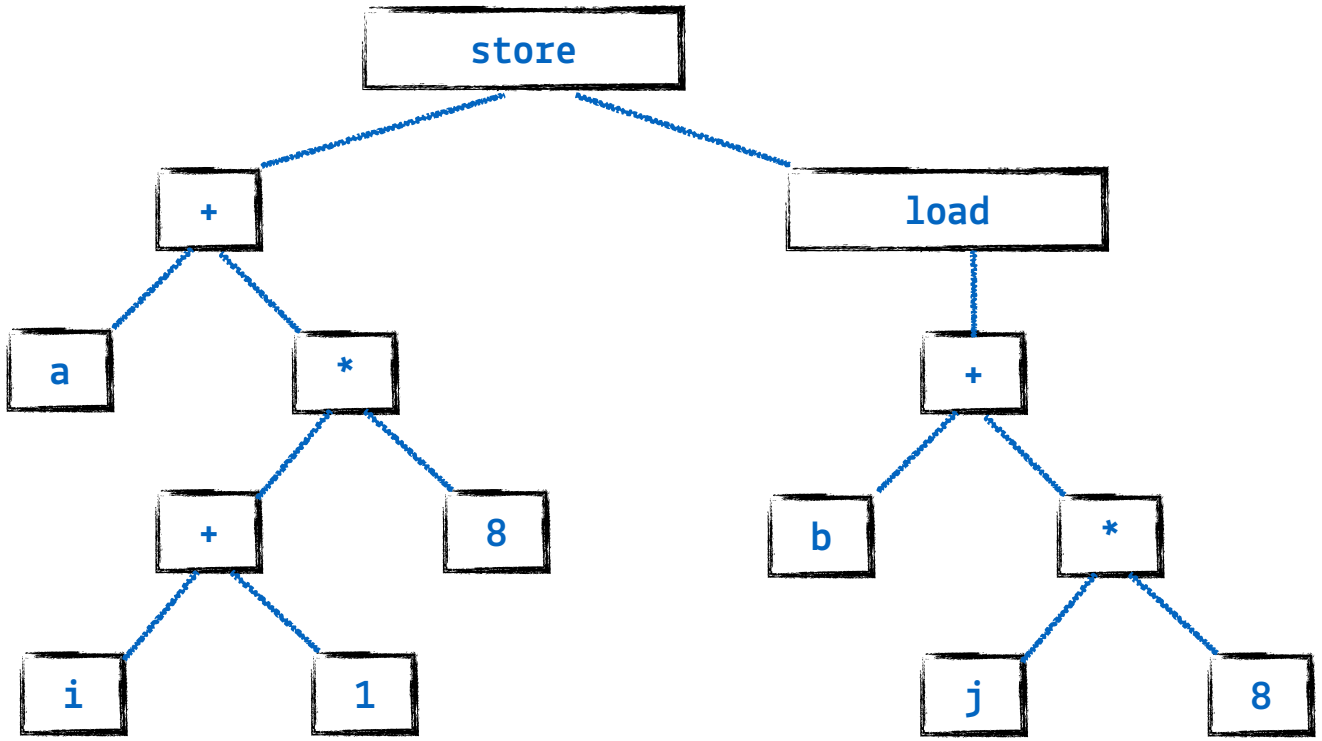
Partial instructions aren't necessarily adjacent

- Address of `b[j]`
  - `mulc 8, r_j`
- Address of `a[i+1]`
  - `add 1,r_i`
  - `mulc 8, r_1`
  - `add r_i, r_a`
- Store into `a[i+1]`
  - `movex r_j, r_b, r_a`



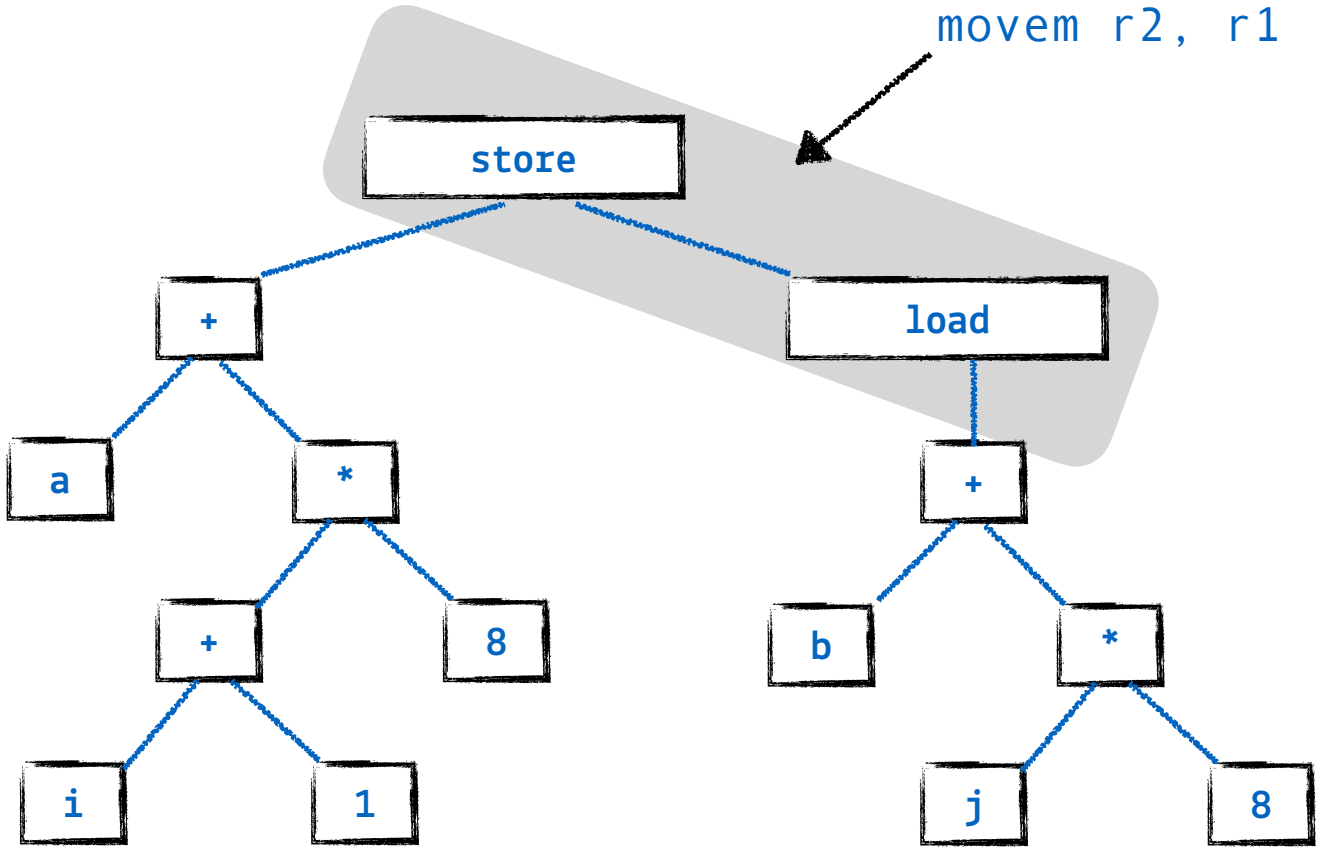
Tree representation

- The 4 overall steps can be written as a tree



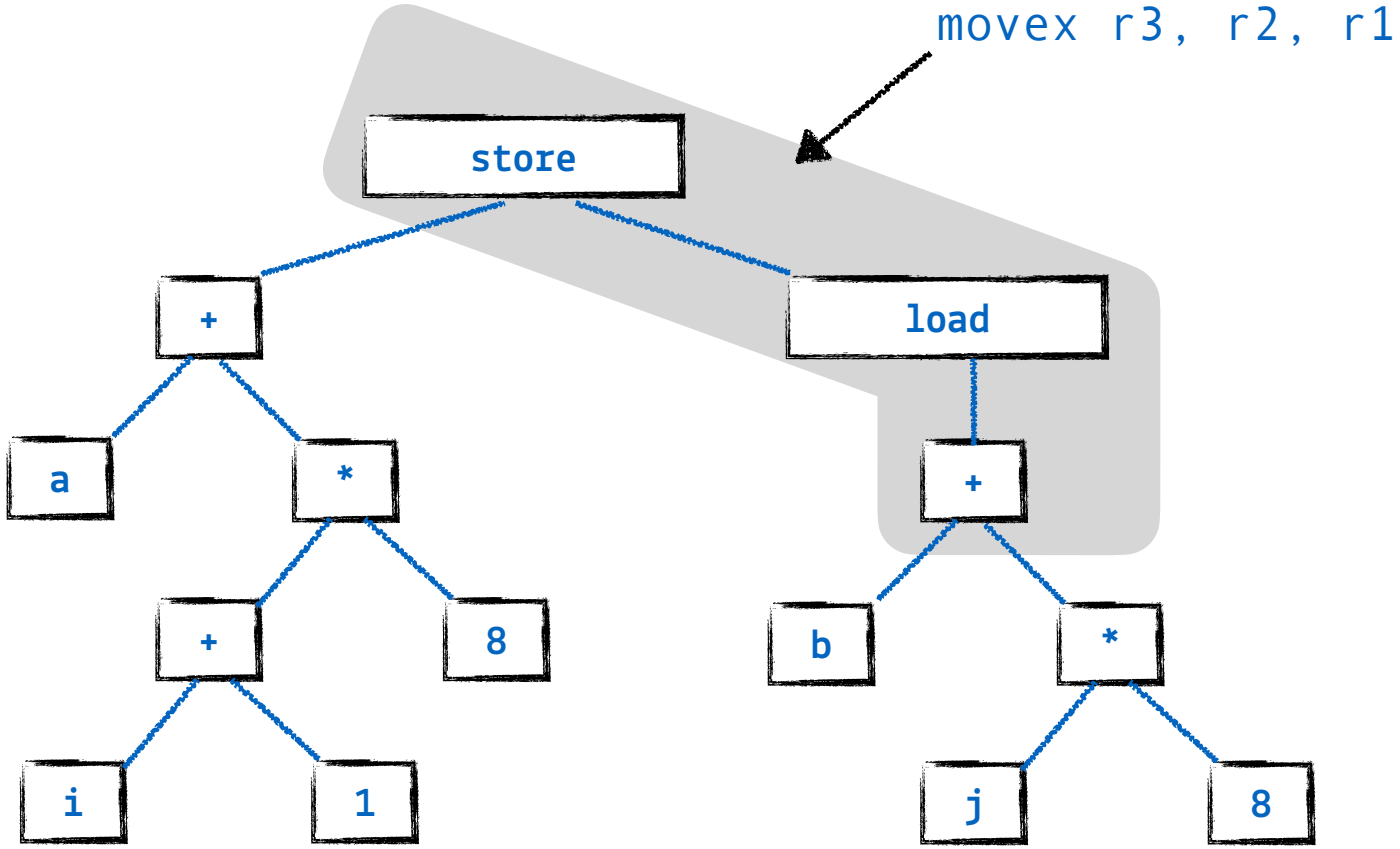
Instructions can be *tiles*

- tile = subtree of a particular pattern



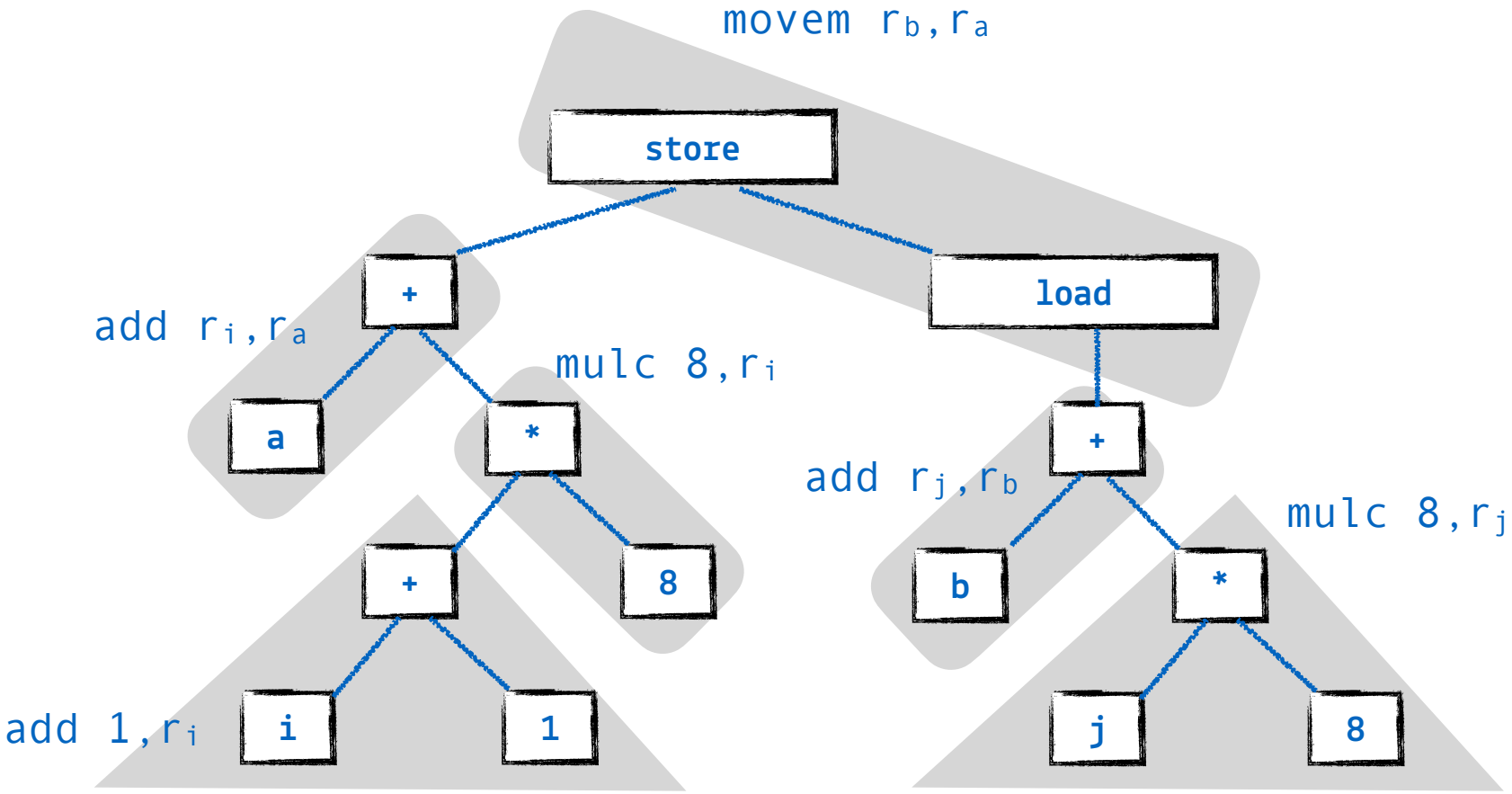
Instructions can be *tiles*

- tile = subtree of a particular pattern



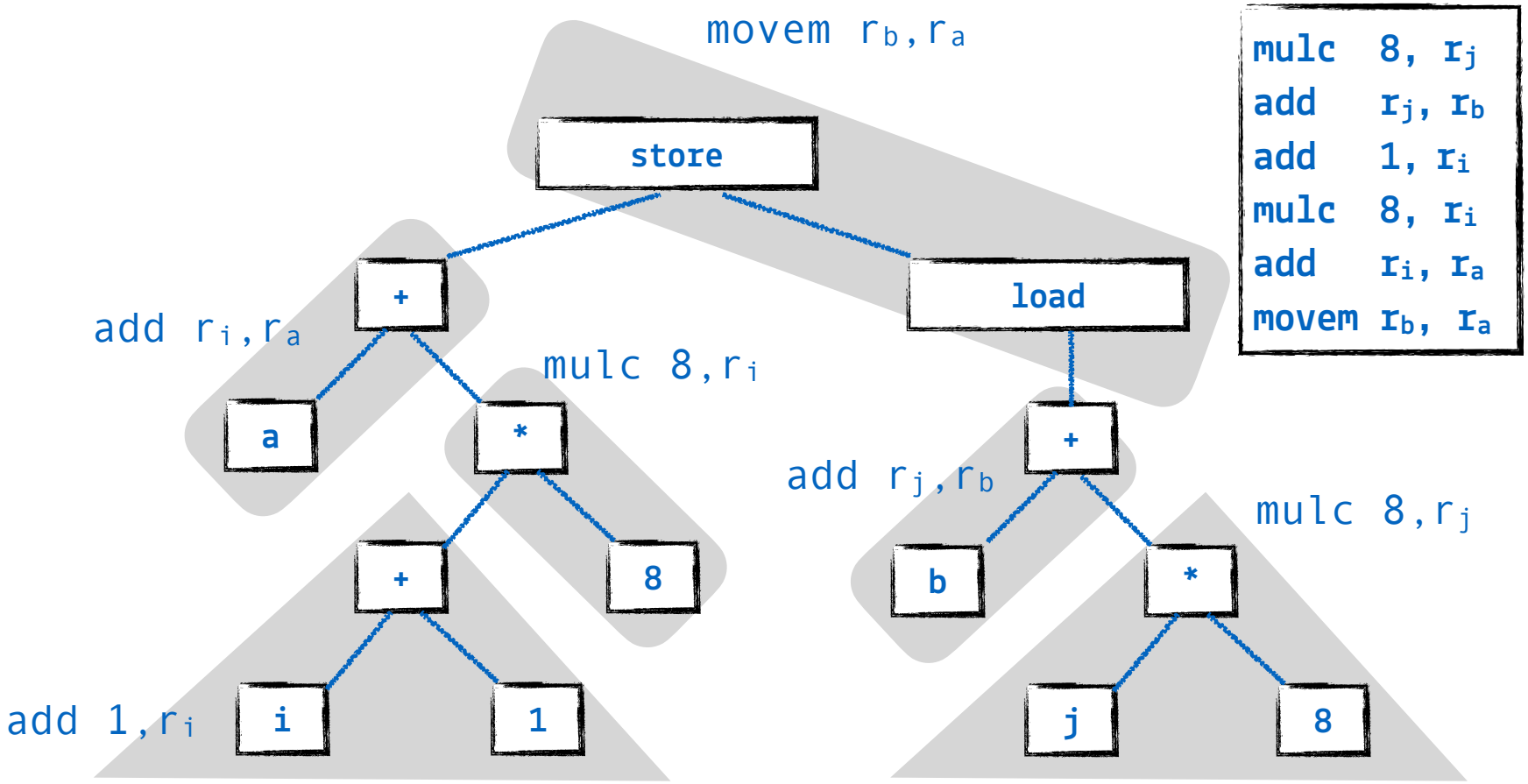
Tiling

- An instruction selection covers the tree with disjoint tiles



Tiling

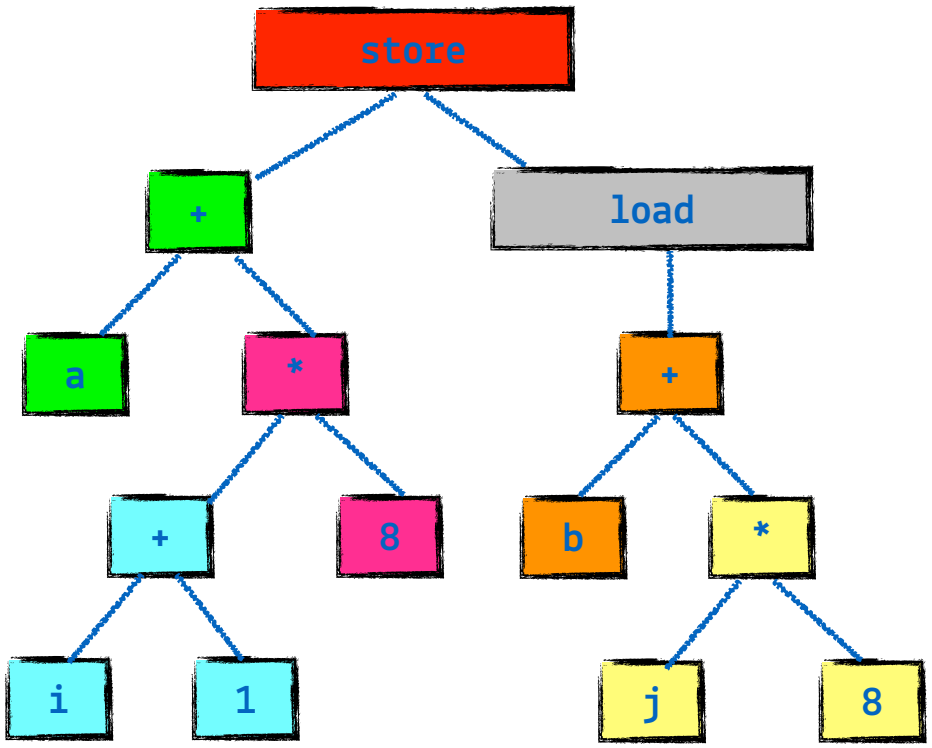
- An instruction selection covers the tree with disjoint tiles



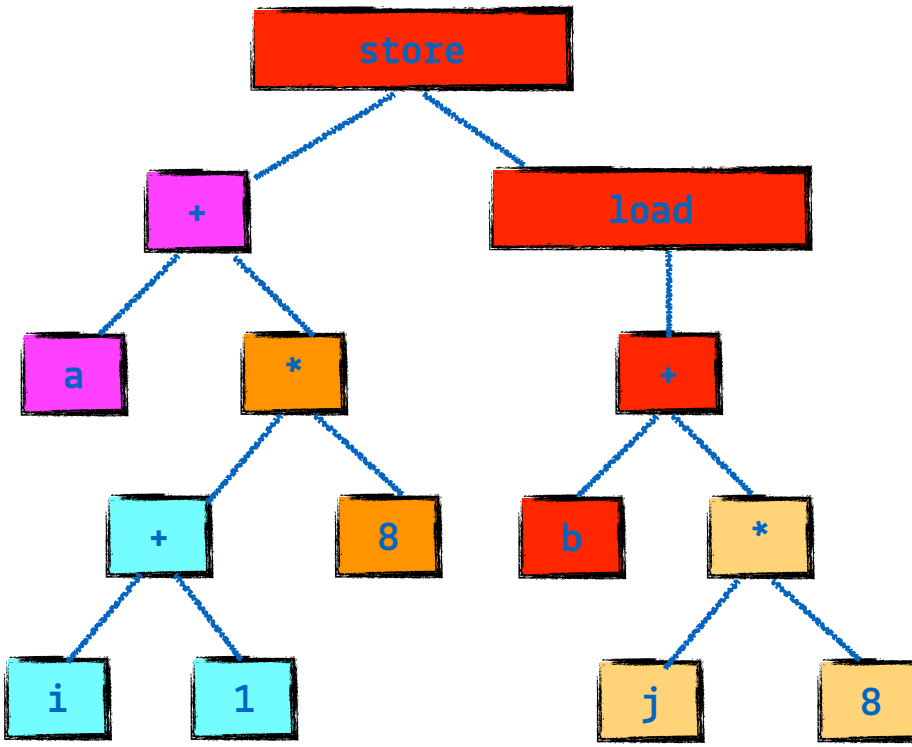
Comparing the tilings

- Alternate tilings give different costs

Using `store` r_b, r_a



Using `movex` r_j, r_b, r_a



Better than trees

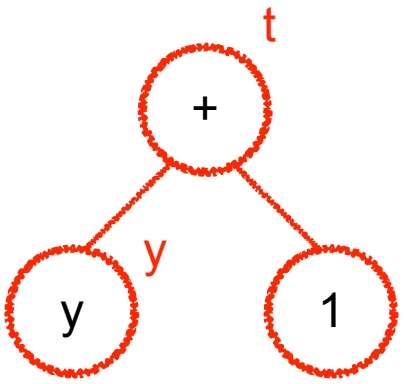
- If we let common sub-expressions be represented by the same node, the trees become *directed acyclic graphs* (DAGs)
- Separate labels and annotations
  - Label nodes with variables, constants or operators
  - Annotate nodes with variables that hold their value
  - Construct DAG from low-level IR

Basic approach

- For each instruction in a basic block
  - if it's "x = y op z"
    - find or create a node annotated y
    - find or create a node annotated z
    - find or create a node labeled op with operands y and z
    - remove annotation x from everywhere
    - add annotation x to the op node
  - if it's "x = y"
    - find or create a node annotated y
    - add annotation x to it

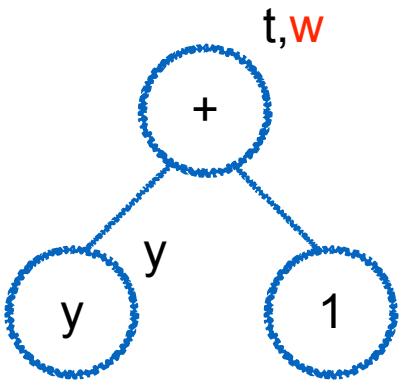
Step 1

```
t = y + 1
w = y + 1
y = z * t
t = t + 1
z = t * y
w = z
```



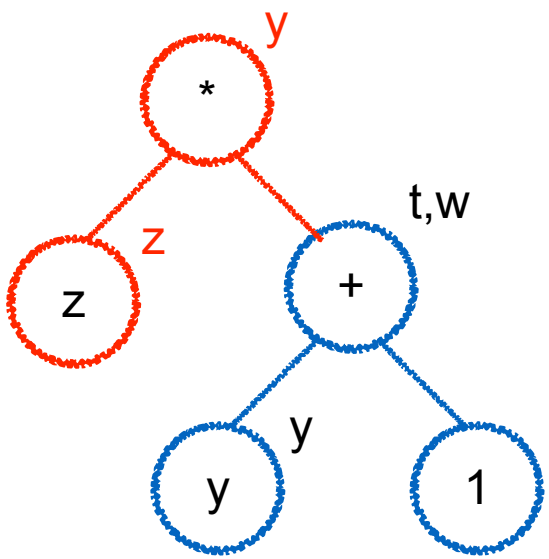
Step 2

```
t = y + 1
w = y + 1
y = z * t
t = t + 1
z = t * y
w = z
```



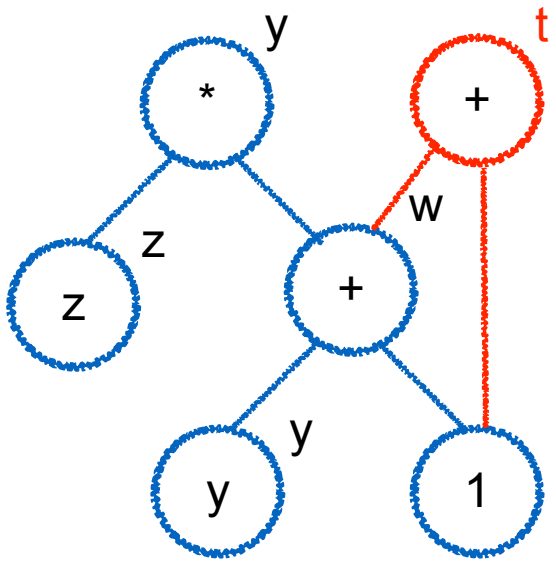
Step 3

```
t = y + 1
w = y + 1
y = z * t
t = t + 1
z = t * y
w = z
```



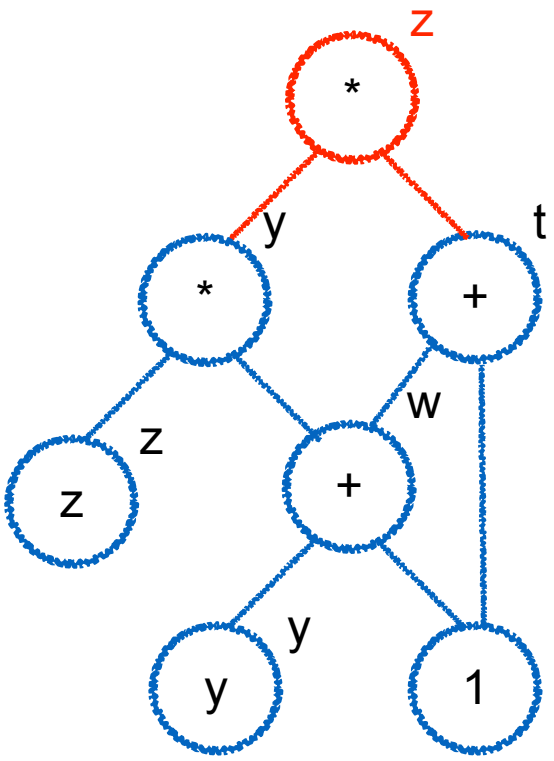
Step 4

```
t = y + 1
w = y + 1
y = z * t
t = t + 1
z = t * y
w = z
```



Step 5

```
t = y + 1
w = y + 1
y = z * t
t = t + 1
z = t * y
w = z
```



Step 6

```
t = y + 1
w = y + 1
y = z * t
t = t + 1
z = t * y
w = z
```