Lecture Q:

Instruction Selection *(Backend 1)*

CSE401/501m:

Introduction to Compiler Construction

Instructor: Gilbert Bernstein

Administrivia

- Compiler codegen (due last night)
  - ✦ If you have late days and want to fix bugs…
- Project Reports (due Tuesday, no late days allowed)
- HW4 is out now, (due Thursday, standard late days apply)

Outline

Intro — Major Backend Passes

What Real CPUs Do

A Low-Level IR to Work With

The Instruction Selection Problem

Algorithms for Instruction Selection

Outline

Intro — Major Backend Passes

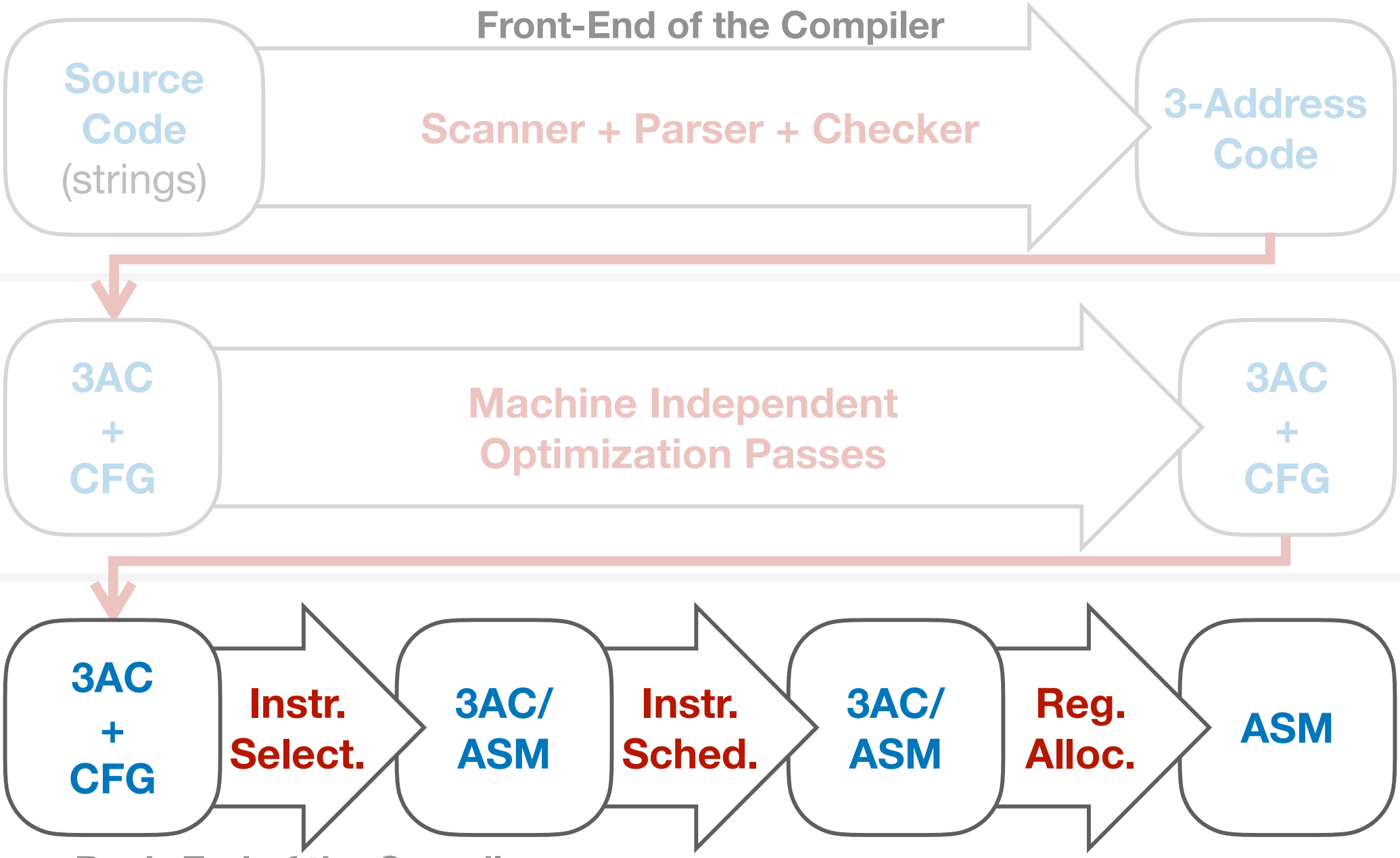
What Real CPUs Do

A Low-Level IR to Work With

The Instruction Selection Problem

Algorithms for Instruction Selection

Our Compiler Journey



Save the Hard Problems for Last

- The front-end should work very quickly
  - Scanner is $O(n)$, Parser is $O(n)$
  - Checking is $O(n)$ *(or maybe* $O(n \log n)$*)*
- Dataflow, SSA, optimization *can* be $O(n^2)$ or worse, but is usually much closer to linear time
- Backend
  - Instruction Selection — fast (naive) or NP-hard
  - Instruction Scheduling — NP-hard
  - Register Allocation — NP-hard

IRs for the Backend

- Generally some kind of three-address code (3AC)
  - $r_1 \leftarrow r_2 \; op \; r_3$
  - Low-level IR — either assembly or just above assembly; objects and other high-level features should be gone by this point
- (Often) Focus on basic blocks
  - If 3AC is SSA, then this implies a dataflow DAG
- Assume "enough" (= ∞) registers, as in SSA
  - Change to a finite number of registers during Reg. Allocation

1. Instruction Selection

- **Goal** — Map from a machine/target-independent IR into machine/target-specific assembly code
- Assume that we have already decided how objects on the heap/stack will be stored/laid-out, and code shape has been determined
  - ✦ i.e. middle optimizations are complete
- **The problem** — in general, an assembly instruction (in the output) corresponds to one ***or more*** low-level machine-independent operations (from the input)
  - ✦ *How do we optimally group input operations into specific assembly instructions?*

2. Instruction Scheduling

- **Goal** — In what order should the instructions be executed? (usually, within a basic block)
- **Fundamental Tradeoffs!**
  - Latency — loads and stores take many cycles; we need to *schedule* other instructions to do while we're waiting
  - Parallelism — the processor can compute some instructions *at the same time* if we stick them nearby
  - Locality — want to minimize the number of *live* registers that we need at any point in the sequence
- *Given a sequence of instructions, how should they be reordered to navigate the above tradeoffs?*

3. Register Allocation

- **Goal** — Go from using ∞ registers down to K registers
- The Good — As long as we use less than K registers, we're fine (all such solutions are equally good)
- **The Bad** — If we need more than K registers, we have to insert additional loads & stores, called **register spilling**
- **The Ugly** — Different solutions to Instruction Selection and Scheduling require different numbers of registers
  - ✦ That is, there's a *phase ordering* problem for all of the backend passes
  - ✦ If registers spill, run a second instruction scheduling pass afterwards to account for new loads/stores

Conventional Wisdom

- Even though these three problems are interdependent, we can still do pretty well by solving them independently
- Instruction Selection
  - **tree pattern matching** + cost problem
  - Still ∞ registers / SSA
- Instruction Scheduling
  - Within a block, use a **list scheduling** algorithm
  - If not good enough, we use tricks to expose bigger blocks
- Register Allocation
  - Use a **graph coloring** algorithm

Outline

Intro — Major Backend Passes

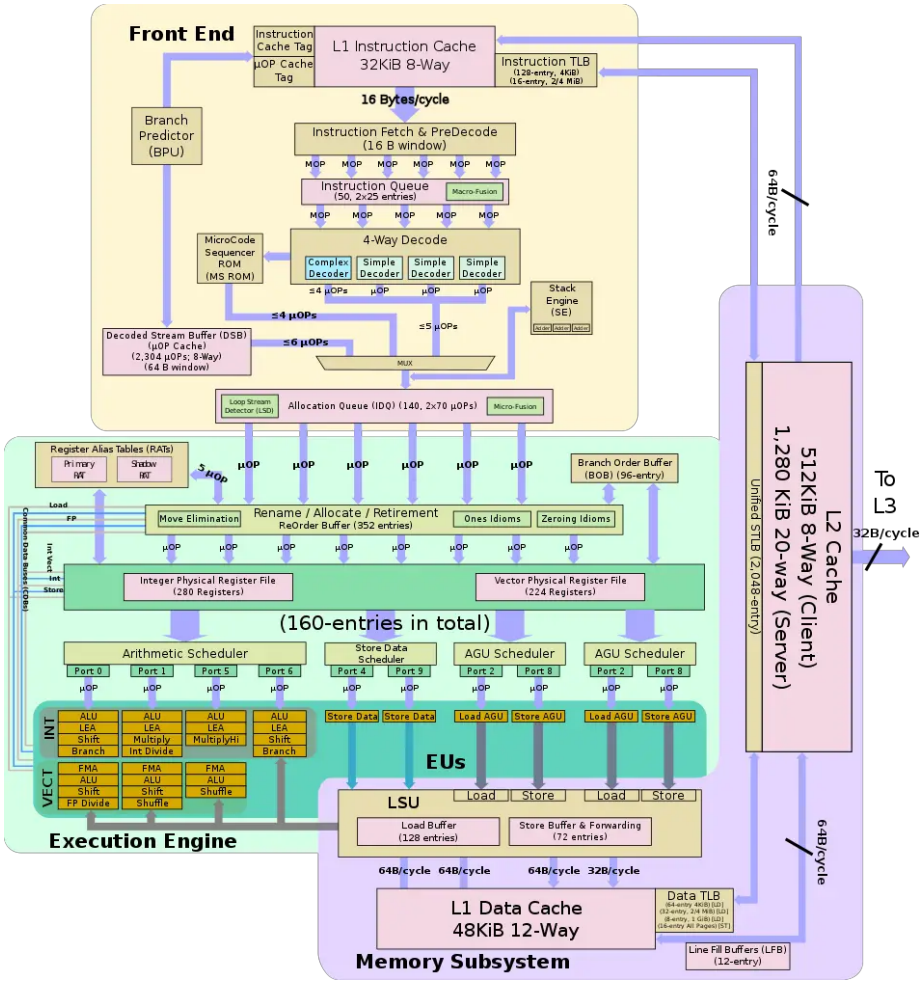
What Real CPUs Do

A Low-Level IR to Work With

The Instruction Selection Problem

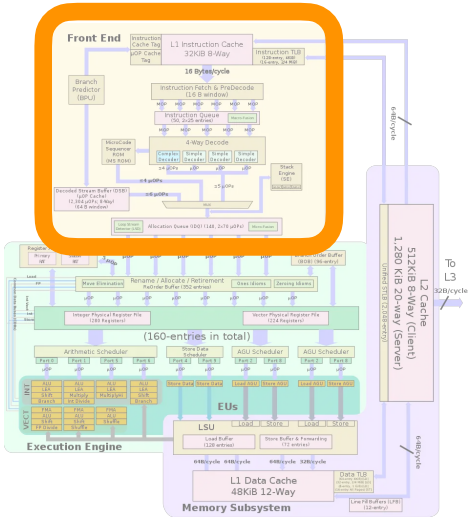
Algorithms for Instruction Selection

Intel Sunny Cove Microarchitecture



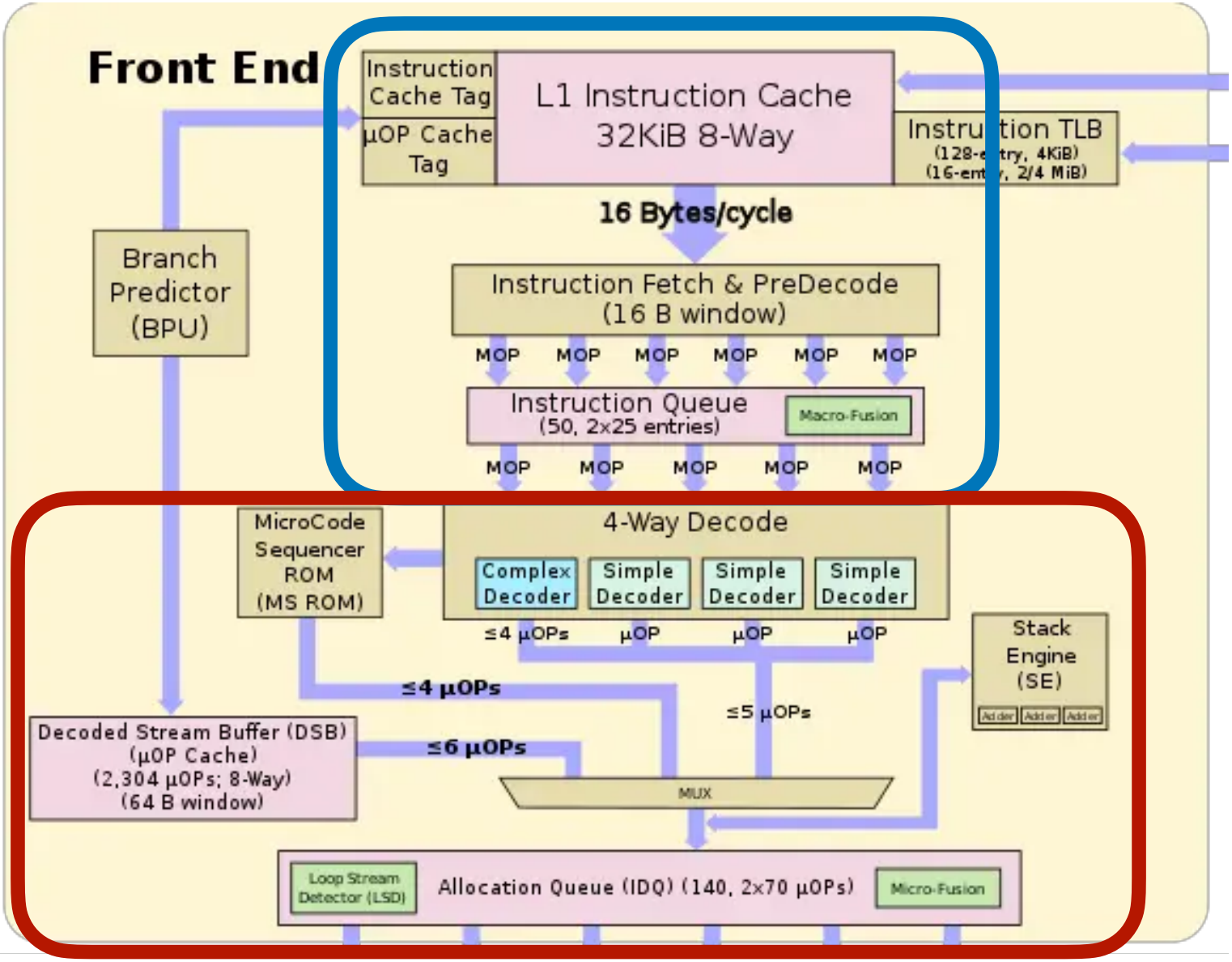
- Intel core microarchitecture circa 2019
- 3 big Components
  - ✦ Front-End
  - ✦ Execution Engine
  - ✦ Memory Subsystem
- Let's see (some of) what a real processor does…
- **(not on any test!)**

Sunny Cove Front End

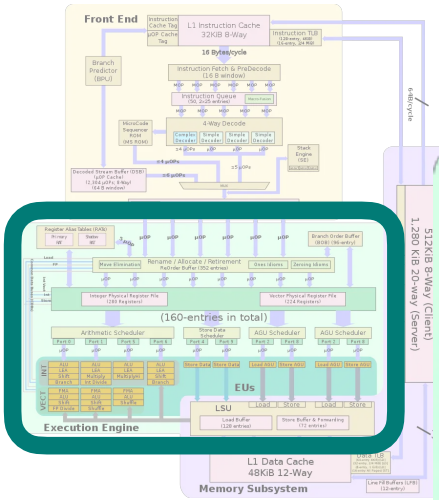


MOP = Macro Op (CISC, i.e. x86)

μOP = micro Op (RISC)



Sunny Cove Execution Engine



Register Renaming!

Schedulers!

Register Alias Tables (RATs)
Primary RAT
Shadow RAT
5 µOP
µOP µOP µOP µOP µOP µOP
Branch Order Buffer (BOB) (96 entry)
Load
FP
Move Elimination
Rename / Allocate / Retirement
ReOrder Buffer (352 entries)
Ones Idioms
Zeroing Idioms
Vect
Int
Store
Integer Physical Register File (280 Registers)
Vector Physical Register File (224 Registers)
(160-entries in total)
Arithmetic Scheduler
Port 0 Port 1 Port 5 Port 6
Store Data Scheduler
Port 4 Port 9
AGU Scheduler
Port 2 Port 8
AGU Scheduler
Port 2 Port 8
INT
ALU LEA Shift Branch
ALU LEA Multiply Int Divide
ALU LEA MultiplyHi
ALU LEA Shift Branch
VECT
FMA ALU Shift FP Divide
FMA ALU Shift Shuffle
FMA ALU Shuffle
Store Data Store Data
Load AGU Store AGU
Load AGU Store AGU
EUs
LSU
Load Store Load Store
Load Buffer (128 entries)
Store Buffer & Forwarding (72 entries)
Execution Engine

Modern Out-of-Order Processors

- CPUs (especially x86 CPUs) will undo and redo all of the work the backend does! What?!?! Why even bother?
  - ✦ Instruction Selection — x86 processors have to support legacy instructions, hence translation; choice of instruction still matters for performance
  - ✦ Register Renaming — If you can't fit everything into the 16 x86-64 registers, your program still has to spill, even if there are many more microarchitectural registers
  - ✦ Instruction Scheduling — the CPU can only reorder instructions within a limited window, so order still roughly matters (and affects register usage, etc.)

Why Do CPUs Do All This?

- Performance! (well duh, but…)
- Out-of-Order Cores
  - ✦ Memory latencies are unpredictable — dynamically rescheduling the instruction stream is fundamentally more effective than any static schedule
- In-Order Cores
  - ✦ If you start stripping this stuff away, (energy efficiency!) then the quality of static scheduling, instruction selection and register allocation becomes much more important!
  - ✦ (very important for low-power & accelerators)

Takeaways for Compiler Writing

- The cost model is (always) very wrong
- Some false claims
  - ✦ a program that executes fewer instructions will run faster
  - ✦ Minimizing register usage is always better than using more registers
  - ✦ Maximizing the amount of work that can be done in parallel is always better than not doing that
- Biggest Costs — loads, stores, cache misses, no instruction-level parallelism to exploit

Outline

Intro — Major Backend Passes

What Real CPUs Do

A Low-Level IR to Work With

The Instruction Selection Problem

Algorithms for Instruction Selection

A Low-Level IR*

Expressions

$e ::= CONST(int)$

| $TEMP(tname)$
(value of register w/this name)

| $BINOP(op, e_1, e_2)$

| $READ(e)$
(read from mem. addr. e)

| $CALL(f, e_1, \ldots, e_n)$

Statements

$s ::= MOVE(tname, e)$

| $WRITE(e_1, e_2)$
(write e_2 to addr. e_1)

| $SEQ(s_1, s_2)$

| $LABEL(lname)$

| $JUMP(lname)$

| $CJUMP(op, e_1, e_2, lname_1, lname_2)$
(e.g. if $e_1 < e_2$, jump to $lname_1$, otherwise jump to $lname_2$)

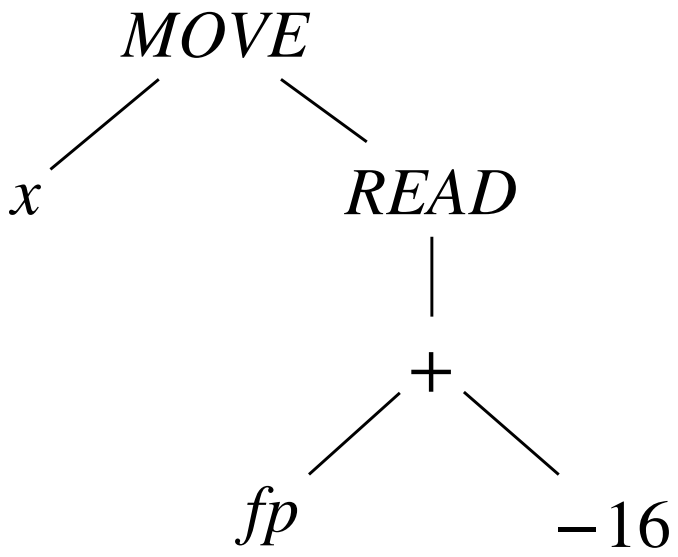
*This Example IR is adapted from Appel

Low-Level IR Example (1)

- Read a local variable at a known offset -16 from the frame pointer *fp*, and assign it to temporary x
  - Linear Code

$$MOVE\left(x, READ\left(BINOP(+, TEMP(fp), CONST(-16))\right)\right)$$

  - Tree Diagram

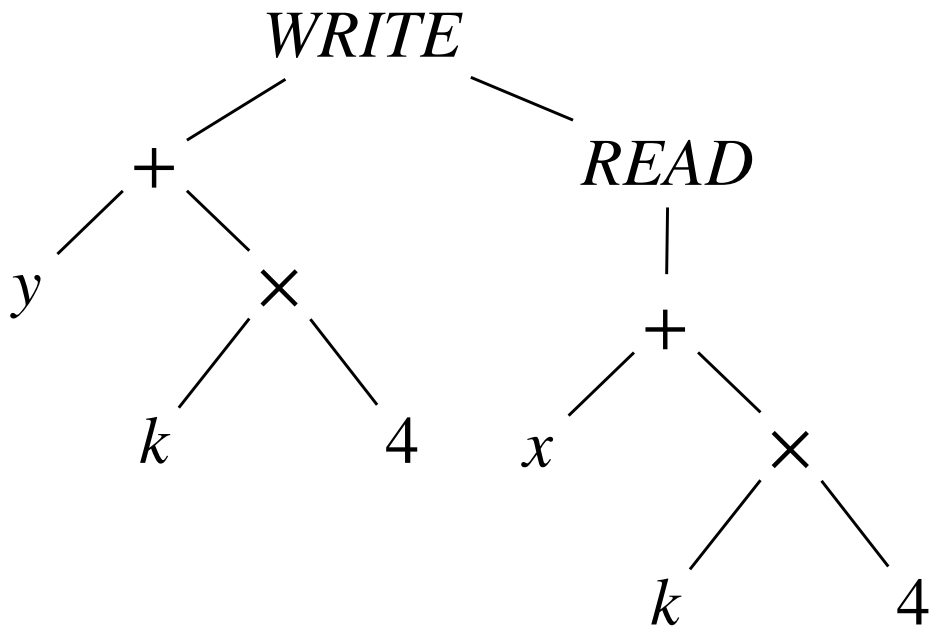


Low-Level IR Example (2)

- Copy a 32-bit integer from x[k] to y[k]
  - ✦ Linear Code

$$WRITE\Big(BINOP\left(+, TEMP(y), BINOP(\times, TEMP(k), CONST(4))\right),$$
$$READ\Big(BINOP\left(+, TEMP(x), BINOP(\times, TEMP(k), CONST(4))\right)\Big)$$

  - ✦ Tree Diagram



Linear IR as Trees or DAGs

- If we have SSA, trees will be much shorter
- However, SSA in a basic block can be interpreted as a DAG (directed acyclic graph)
- Two possible ways to proceed
  - ✦ Use an instruction selection algorithm designed for DAGs
  - ✦ Use an instruction selection algorithm designed for trees and adapt it to DAG structure

A Target Language

Instructions

```
ADD   r1 <- r2 + r3
SUB   r1 <- r2 - r3
MUL   r1 <- r2 * r3
ADDI  r1 <- r2 + c
SUBI  r1 <- r2 - c
LOAD  r1 <- Mem[r2 + c]
STORE Mem[r1 + c] <- r2
// jumps, calls, etc.
// omitted for simplicity
```

Outline

Intro — Major Backend Passes

What Real CPUs Do

A Low-Level IR to Work With

The Instruction Selection Problem

Algorithms for Instruction Selection

The Instruction Selection Problem

- In most low-level IRs, there are many possible code sequences that implement the input “low-level IR” code correctly
  - ✦ e.g. set %rax to 0 on x86-64

```
movq  $0,%rax          salq  64,%rax
subq  %rax,%rax        shrq  64,%rax
xorq  %rax,%rax        imulq $0,%rax
```

  - ✦ Other (esp. CISC) machine instructions do many things at the same time, e.g. x86 addressing modes

```
movq  offset(%rbase, %rindex, scale), %rdst
```

- So which instructions should we use & in what way?

Instruction Selection Criteria

- Recall — Why pick one selection over another?
  - ✦ Fastest Execution, Smallest Code, Minimal Power Consumption, Reduce Memory Traffic, …
- As we discussed earlier, almost impossible to model this accurately. (*so don't try too hard to be accurate!*)
  - ✦ Simple answer — Each instruction has cost 1
  - ✦ Slightly less bad — maintain a table of costs for each instruction (e.g. loads and stores have higher cost; divide higher than addition, etc.)

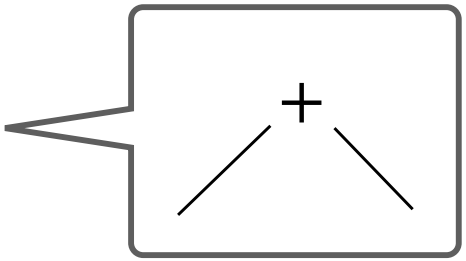
Defining Patterns

- We can specify the "meaning" of target instructions by describing potential patterns to match
- Each pattern has a target instruction on the left with ***holes*** and a source (low-level IR) tree on the right using those same holes
  - ✦ exception — the target code on the left designates a special position * as the "result" (assigned a new name)
- e.g.

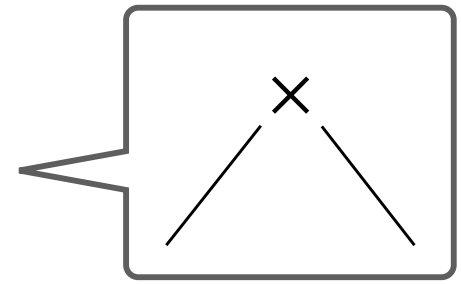
`ADDI r* <- e + c` $\Longleftarrow$ $BINOP(+, CONST(c), e)$

Patterns for Example Target (1)

- Arithmetic
  - `r` $\Longleftarrow$ $TEMP(r)$
  - `ADD  r* <- e1 + e2` $\Longleftarrow$ $BINOP(+, e_1, e_2)$



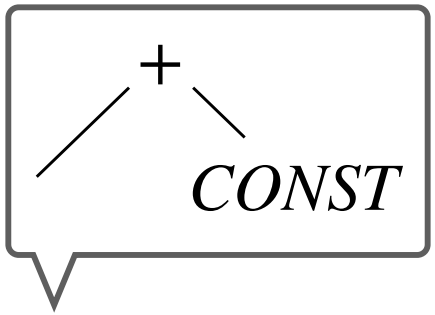
  - `MUL  r* <- e1 × e2` $\Longleftarrow$ $BINOP(\times, e_1, e_2)$



  - `SUB` and `DIV` similarly

Patterns for Example Target (1)

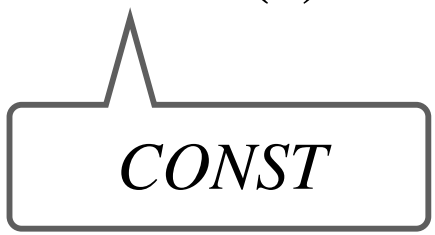
- Immediate Instructions
  - *assume we have a register r0 that always holds 0*

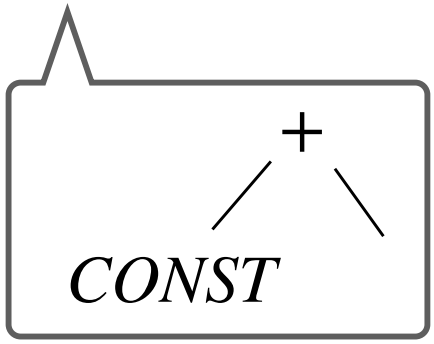


- `ADDI r* <- e + c` $\Longleftarrow$ $BINOP(+, e, CONST(c))$

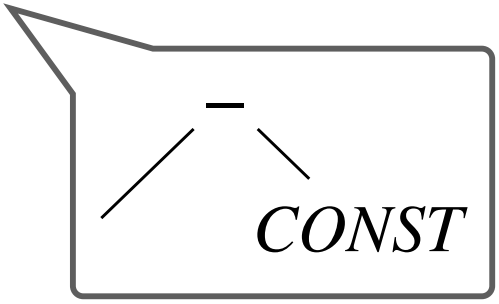
  `ADDI r* <- e + c` $\Longleftarrow$ $BINOP(+, CONST(c), e)$

  `ADDI r* <- r0 + c` $\Longleftarrow$ $CONST(c)$

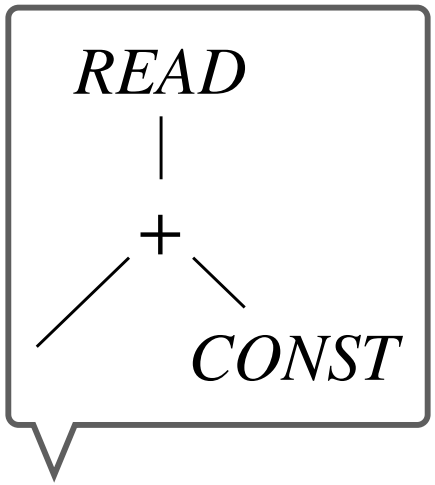




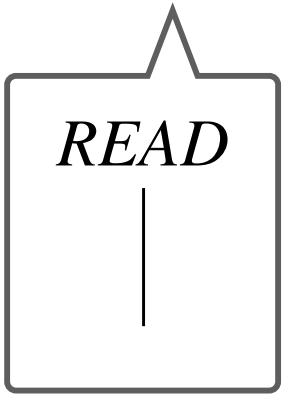
- `SUBI r* <- r - c` $\Longleftarrow$ $BINOP(-, e, c)$

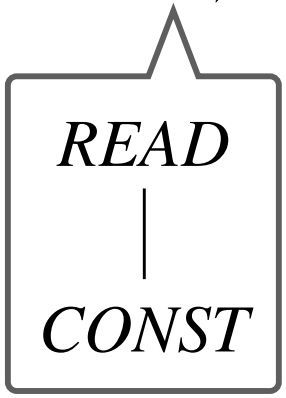


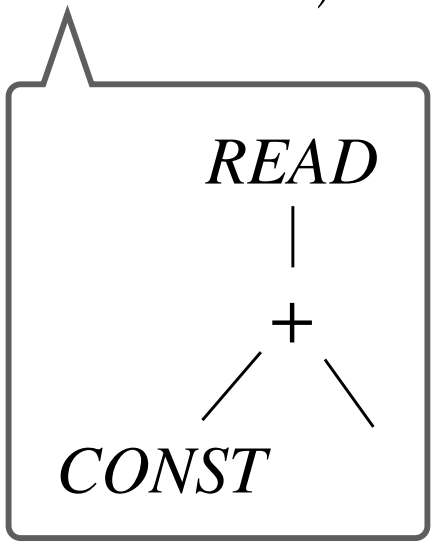
Target Patterns (3)



- Load
  - *assume we have a register r0 that always holds 0*
    - `LOAD r* <- M[e + c]` $\Longleftarrow$ $READ\left(BINOP(+, e, CONST(c))\right)$
      `LOAD r* <- M[e + c]` $\Longleftarrow$ $READ\left(BINOP(+, CONST(c), e)\right)$
      `LOAD r* <- M[r0 + c]` $\Longleftarrow$ $READ\left(CONST(c)\right)$
      `LOAD r* <- M[e + 0]` $\Longleftarrow$ $READ(e)$

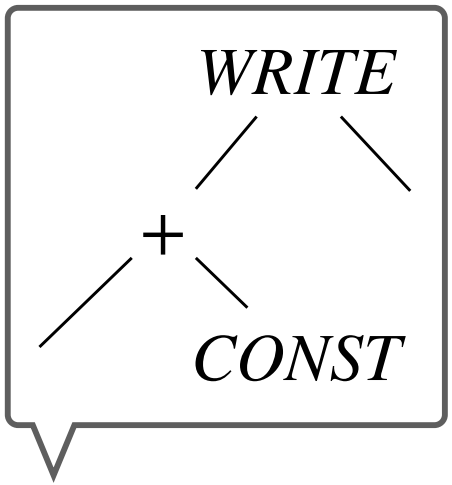






Target Patterns (4)

- Store
  - *assume we have a register r0 that always holds 0*

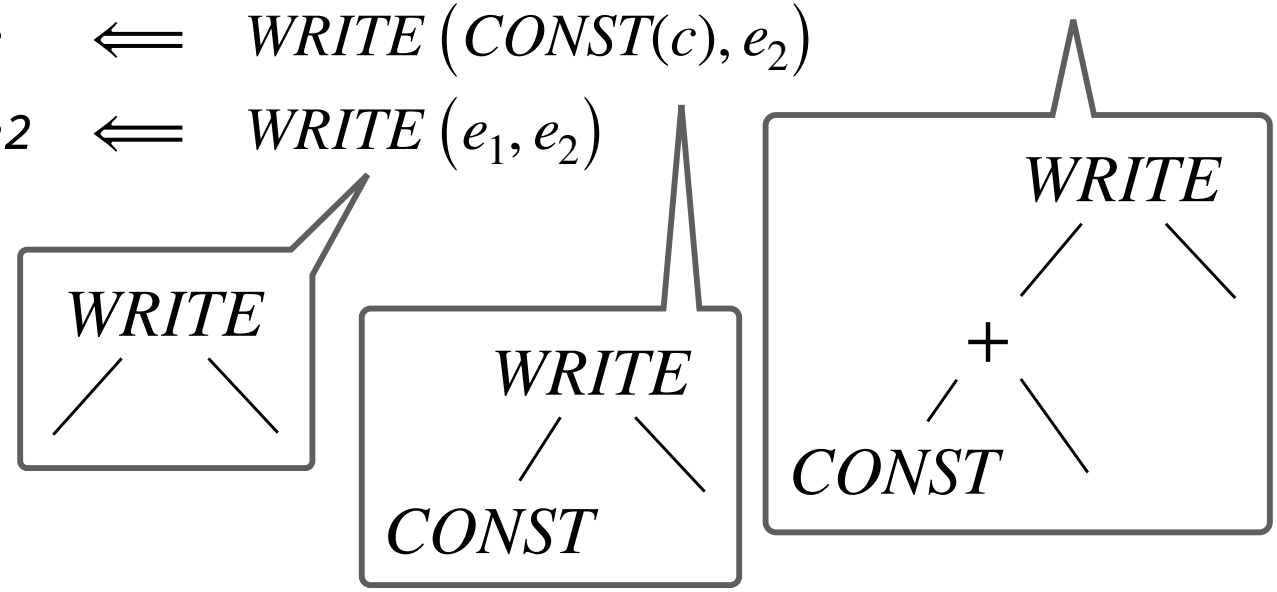


- `STORE M[e1 + c] <- e2` $\Longleftarrow$ $WRITE\left(BINOP(+, e_1, CONST(c)), e_2\right)$

  `STORE M[e1 + c] <- e2` $\Longleftarrow$ $WRITE\left(BINOP(+, CONST(c), e_1), e_2\right)$

  `STORE M[r0 + c] <- e` $\Longleftarrow$ $WRITE\left(CONST(c), e_2\right)$

  `STORE M[e1 + 0] <- e2` $\Longleftarrow$ $WRITE\left(e_1, e_2\right)$



Outline

Intro — Major Backend Passes

What Real CPUs Do

A Low-Level IR to Work With

The Instruction Selection Problem

Algorithms for Instruction Selection

Tree Pattern Matching — Tiles

- **Goal** — Recursively match / "tile" the input tree using the provided patterns
- The cost of each tile is given by the cost model
- A tiling is a set of <*node*, *pattern*> pairs
  - (*pattern gives the shape of the tile*) the input tree at *node* ***n*** must match the right-hand-side of the *pattern* ***p***
- A valid tiling must cover the tree
  - For each <***n***, ***p***> every node ***m*** (except ***n***) matched by ***p*** is covered by the tile <***n***, ***p***>
  - A valid tiling covers all nodes except the root, and no two tiles overlap

Tree Pattern Matching — Parsing

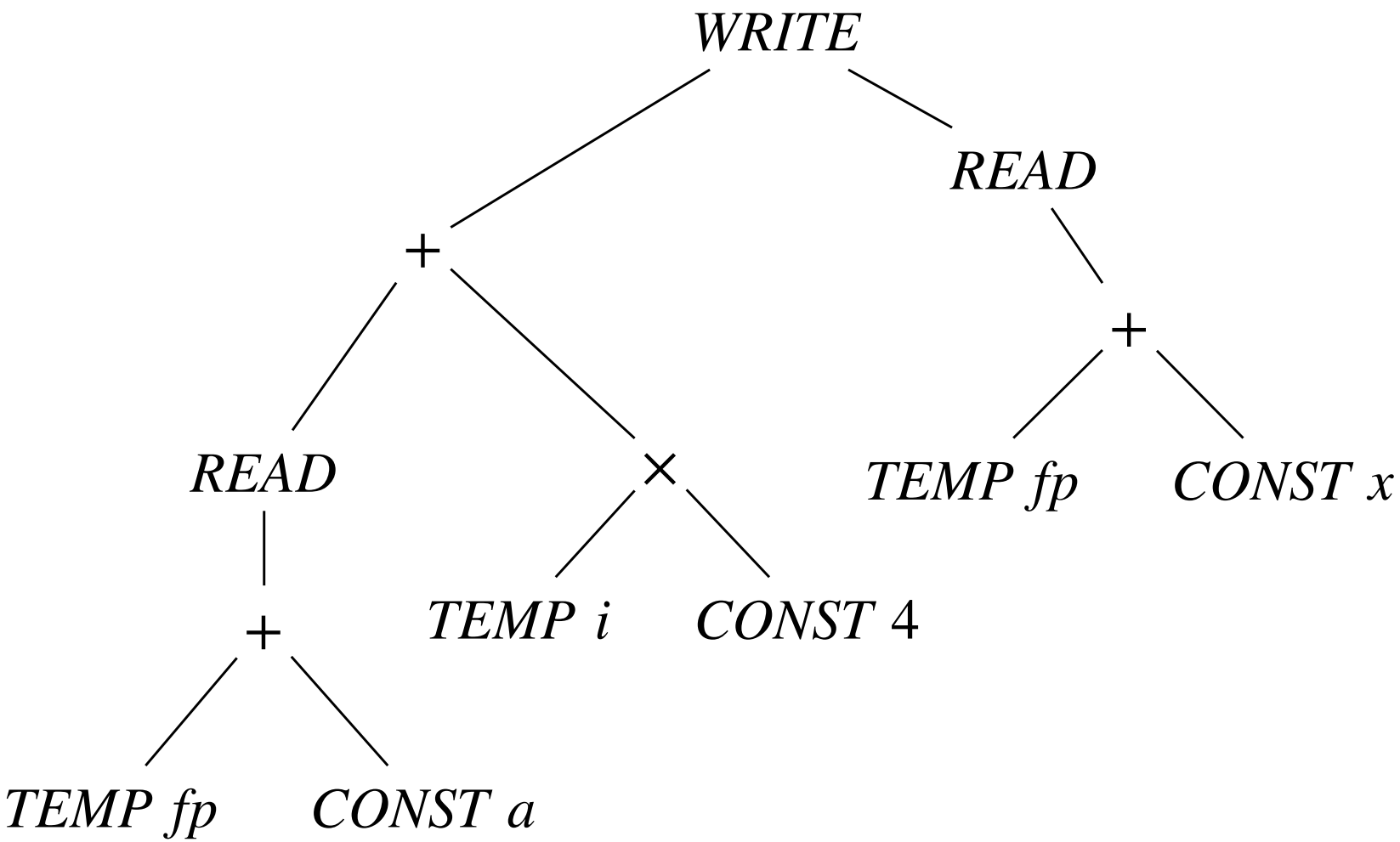
- **Goal** — Recursively match / "parse" the input tree using the provided *tree grammar rules*
- Works very similar to parsing theory from earlier on!
- e.g. we can use LR parsing to do instruction selection
- One key point — different grammatical classes (i.e. non-terminals)
  - In our example low-level IR, there are only expressions
  - In general, we want to support ISAs/IRs with different integer/floating-point register types
  - Patterns/productions can only "plug together" where they agree on these "types"

Two Major Algorithm Styles

- Maximal Munch (Greedy)
  - ✦ Walk the input IR tree from the top-down
  - ✦ Match the largest possible tile at each step (corresponds to all ISA instructions having equal cost)
- Dynamic Programming
  - ✦ Process the tree bottom-up
  - ✦ Assign costs to each sub-tree by trying each possible matching pattern and looking up the cheapest solution to sub-problems/sub-trees (i.e. dynamic programming)

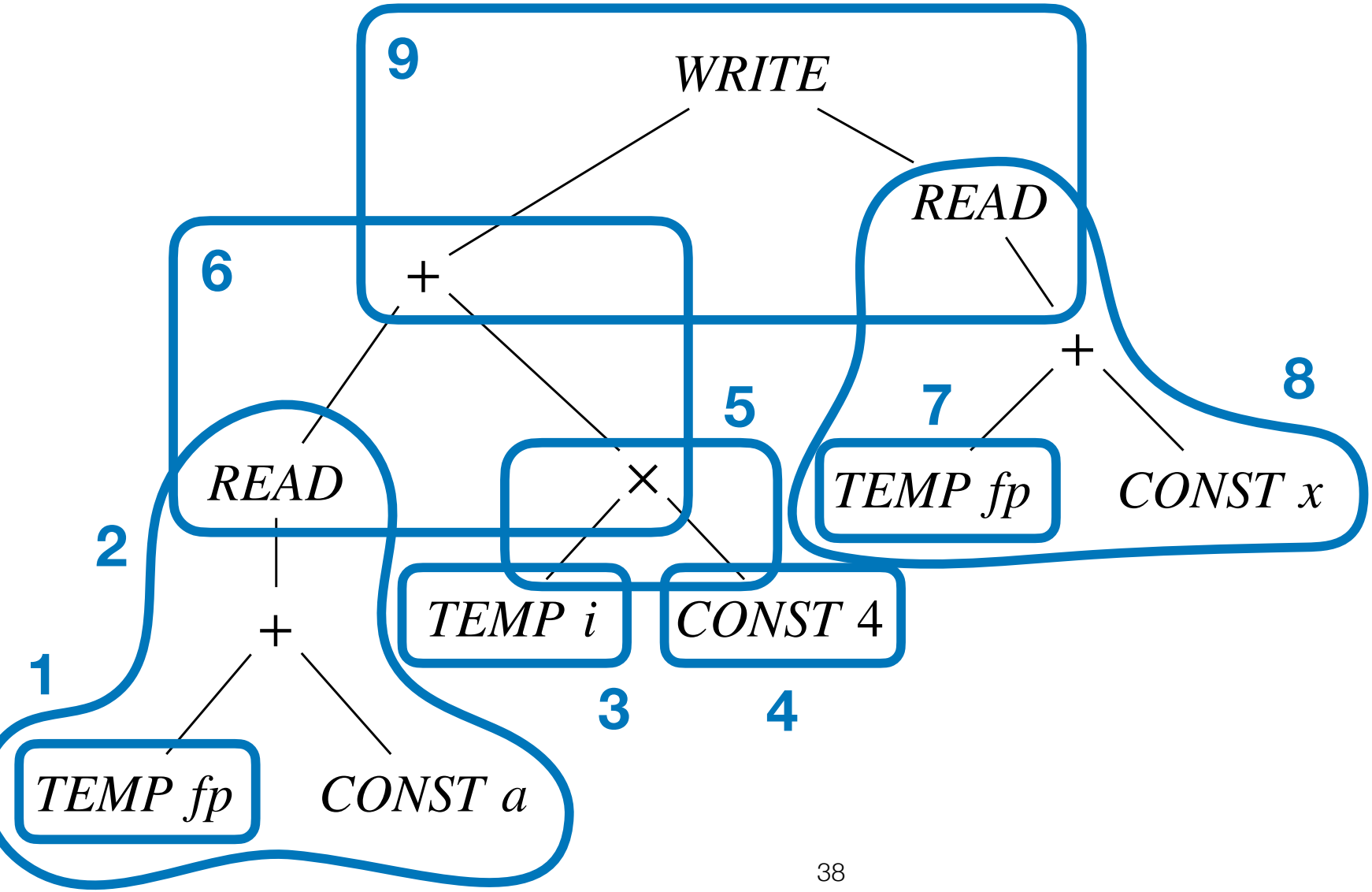
Example Tree Match

- a[i] := x (assume i in register/temp and a,x on stack)



Example Tree Match

- a[i] := x (assume i in register/temp and a,x on stack)



Generating Code

- Given a tiled tree, to generate code
  - ✦ Do a post-order walk of the tiles
  - ✦ Each tile generates a code sequence after you're done visiting it
  - ✦ Intermediate values (where the tiles are connected) correspond to newly named temporaries/registers

Example Tree Match

```
2 LOAD  r1 <- M[fp + a]
4 ADDI  r2 <- r0 + 4
5 MUL   r3 <- i * r2
6 ADD   r4 <- r1 + r3
8 LOAD  r5 <- M[fp + x]
9 STORE M[r4 + 0] <- r5
```