# **6.110** Computer Language Engineering

## **Recitation 10:** Reg alloc & peephole

April 11, 2025

**Weekly updates ←**

Register allocator

Peephole optimization

# Schedule… **Week N+1**

| **Mon 4/7** | **Tue 4/8** | **Wed 4/9** | **Thu 4/10** | **Fri 4/11** |
|---|---|---|---|---|
| **No lecture** | | | | **Recitation** Register Allocation |
| | | | **Due:** Mini-quiz, weekly check-in | |

# Lecture forecast... **Week N+2**

| Mon 4/15 | Tue 4/16 | Wed 4/17 | Thu 4/18 | Fri 4/19 |
|---|---|---|---|---|
| **Lecture**<br>Dataflow Theory | **Lecture**<br>Dataflow Theory | **No lecture** | **No lecture** | **Recitation**<br>Phase 5 infosession |
| | | | **Due:**<br>Mini-quiz, weekly check-in | **Due: Project phase 4** |

Weekly updates

**Register allocator ←**

Peephole optimization

# Register allocator: overview

Probably the **most complicated optimization** so far

Many moving parts, with tricky bugs

**Think carefully before writing any code!**

Martin’s slides give a good overview of techniques for non-SSA IR

What if we used SSA?

# Register allocator: overview

Probably the **most complicated optimization** so far

Many moving parts, with tricky bugs

**Think carefully before writing any code!**

Martin’s slides give a good overview of techniques for non-SSA IR

What if we used SSA?

1. De-SSA first
2. Reg alloc directly on SSA (*Hack*)

# Why over SSA?

Register allocation over SSA has many merits:

- Interference graph is **chordal** — poly time optimal coloring
- # register needed **=** max variables live at program point
- Decoupling spill decision from coloring
- Intellectual superiority?

Challenges:

- Inserting spills on SSA without breaking it
- Writing the poly time coloring algorithm

**No time to cover entire algorithm — just pointers and tips**

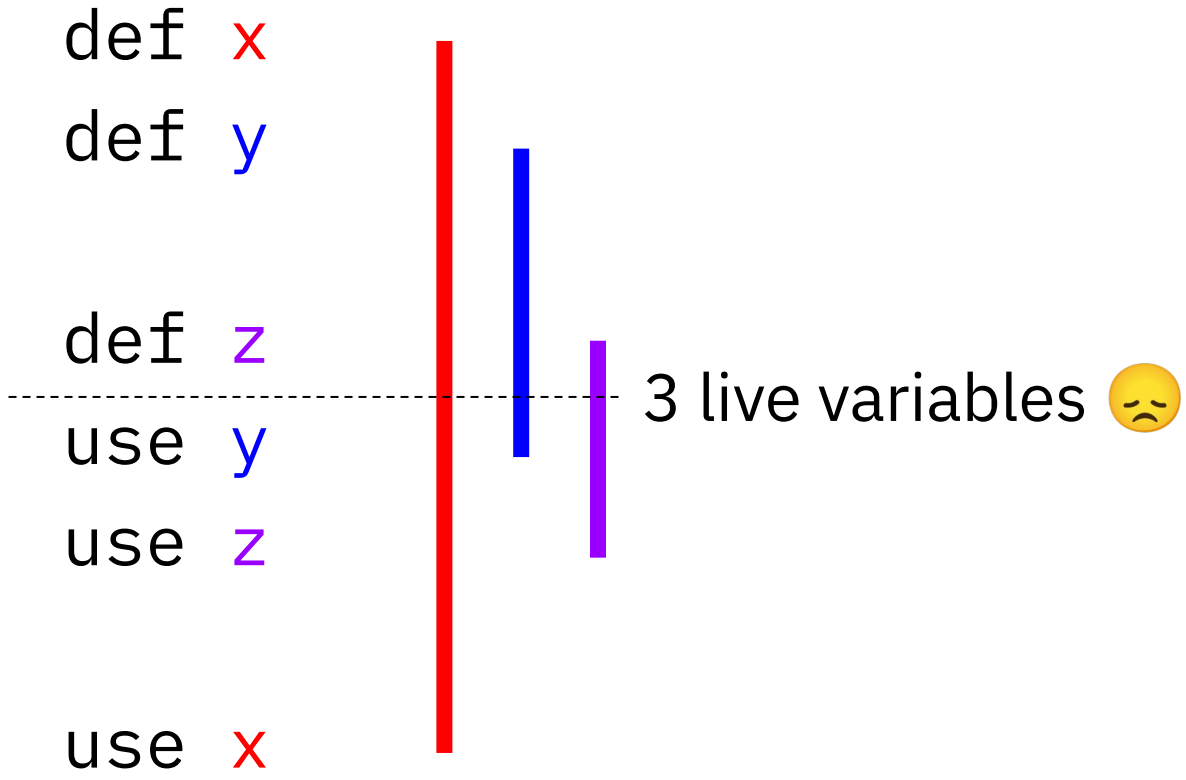
**Many also applies to non-SSA reg alloc**

# Roadmap

1. Compute spill cost
2. Insert spills and reloads
3. Reconstruct SSA
4. Do coloring
5. de-SSA

# Inserting spills and reloads

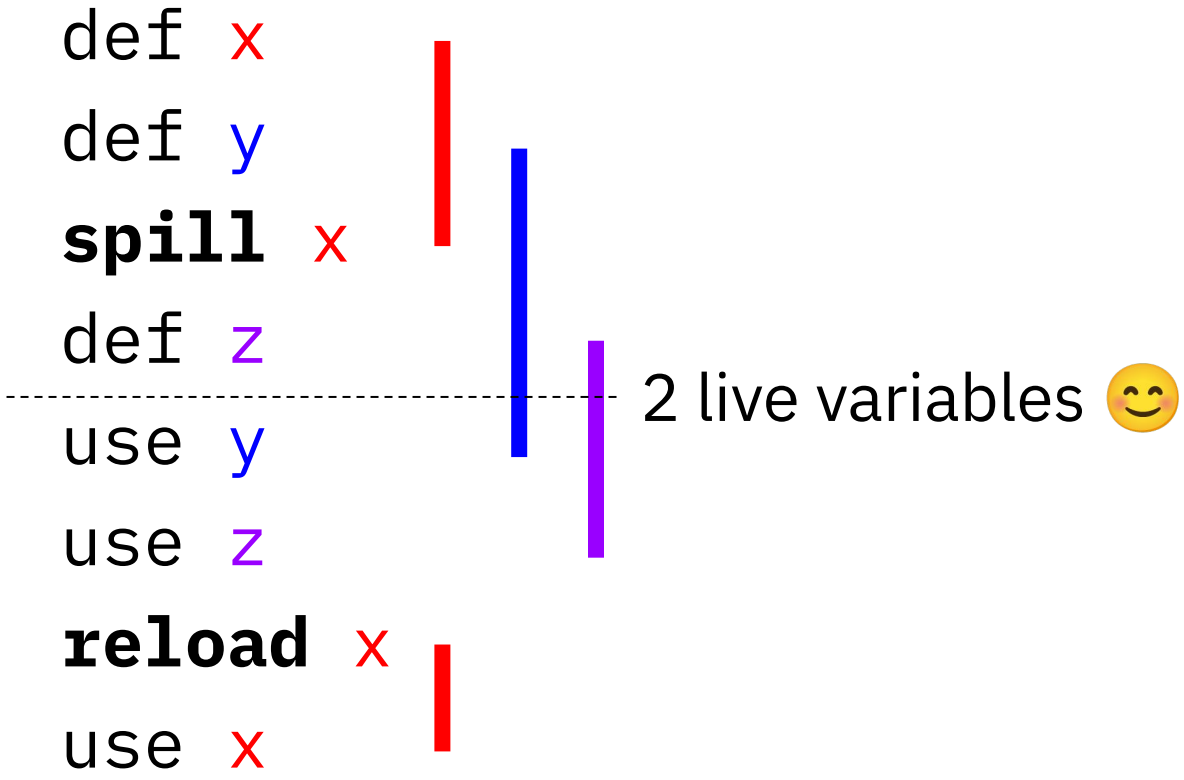
Assume a 2-register machine



# Inserting spills and reloads

Assume a 2-register machine

Spills **splits live ranges** and **lower register pressure**



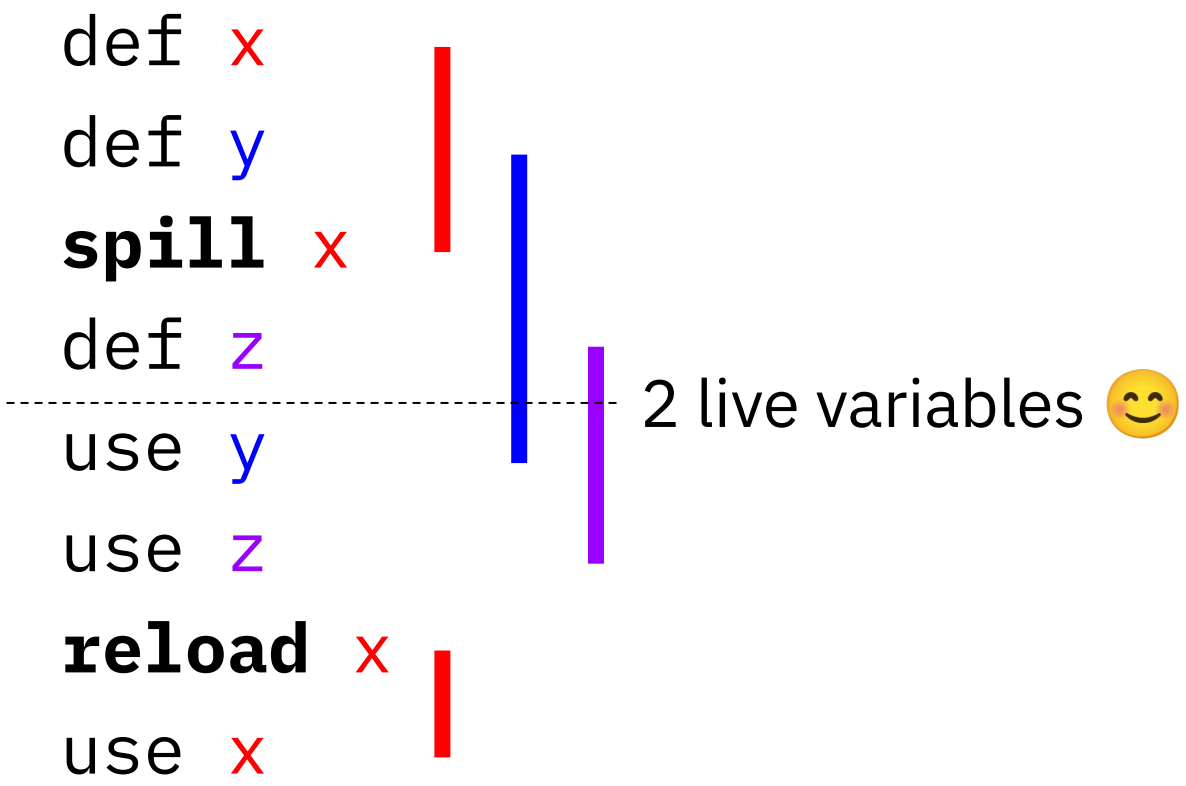
# Inserting spills and reloads

Assume a 2-register machine

In SSA:

**# register req'd = # live vars**

**Goal:** spill such that **# live vars ≤ # of physical registers**



# Spill cost: Belady’s heuristic

What variable should we spill if we run out of registers?

Idea: choose the one **whose next use is farthest down the line**

If we spill it, we don’t have to worry about it for *a long while*

# Spill cost: Belady's heuristic

In basic block:

```
d = {v: ∞ for v in vars} # distance to nearest use
for inst in reversed(block):
    for v in vars:
        d[v] = 0 if v in inst.operands else d[v] + 1
        if inst defines v:
            del d[v]
    cost_before[inst] = copy(d)
```

# Spill cost: Belady’s heuristic

Across basic blocks: **need to merge from successors**

- **MIN**: distance is **nearest** of all successors
  - A lattice, can use worklist algorithm
- Weighted Avg: distance is **avg** of all successors, weighted by **branch probability**
  - Can assign higher prob for loop backedges
  - Not a lattice, **convergence not guaranteed**

# Spill cost: data structures

Need a **spill cost map** of var → spill cost

- Frequent copies, sparse updates
- Persistent maps recommended

Need a representation of **program point**

- A block of $n$ instructions have $n + 1$ program points

Need a map of program point → spill cost map

Trick: keys of spill cost map can double as live variable set

# Fixing up spills across blocks

Spill and reload insertion are done on basic block level

E.g, variable may be

- spilled at the end of prev block
- assumed to have not been spilled in next block
- reload needs to be inserted

How to handle phi nodes, **phi spills?**

More in reading

# Reconstructing SSA

Reloads are **also definitions** of a variable!

- Essentially loading from memory

Need to re-run SSA construction w.r.t. reloads

Memory phi nodes???

See Alg. 4.1 in Hack's thesis

```
703         /// Insertion of spills and reloads breaks the SSA form. This function
704         /// restores it.
705 >       fn reconstruct_ssa(&mut self) -> Method { ...
1019        }
```

# Doing the actual coloring

Hack's paper gives an algorithm to do optimal coloring **without explicitly constructing the interference graph**

Our advice: **DON'T START WITH THAT ALGORITHM**

- More complicated
- Harder to hack in register constraints/preferences

# Using ILP solver 😈

Graph coloring is an NP hard

Your homemade brute-force likely does not perform well

Enter **Integer Linear Programming**

$$
\begin{aligned}
\text{minimize} \quad & \langle \text{objective} \rangle \\
\text{subject to} \quad & \textstyle\sum_{i \in C} x_{v,i} = 1, \quad && \forall v \in V, \\
& x_{u,i} + x_{v,i} \leq 1, && \forall (u,v) \in E,\ i \in C, \\
& x_{v,i} \in \{0,1\}, && \forall v \in V,\ i \in C.
\end{aligned}
$$

# Using ILP solver 😈

$$
\begin{aligned}
\text{minimize} \quad & \langle \text{objective} \rangle \\
\text{subject to} \quad & \textstyle\sum_{i \in C} x_{v,i} = 1, && \forall v \in V, \\
& x_{u,i} + x_{v,i} \leq 1, && \forall (u,v) \in E,\ i \in C, \\
& x_{v,i} \in \{0,1\}, && \forall v \in V,\ i \in C.
\end{aligned}
$$

Easy to modify objective to incorporate **affinity, e.g.**

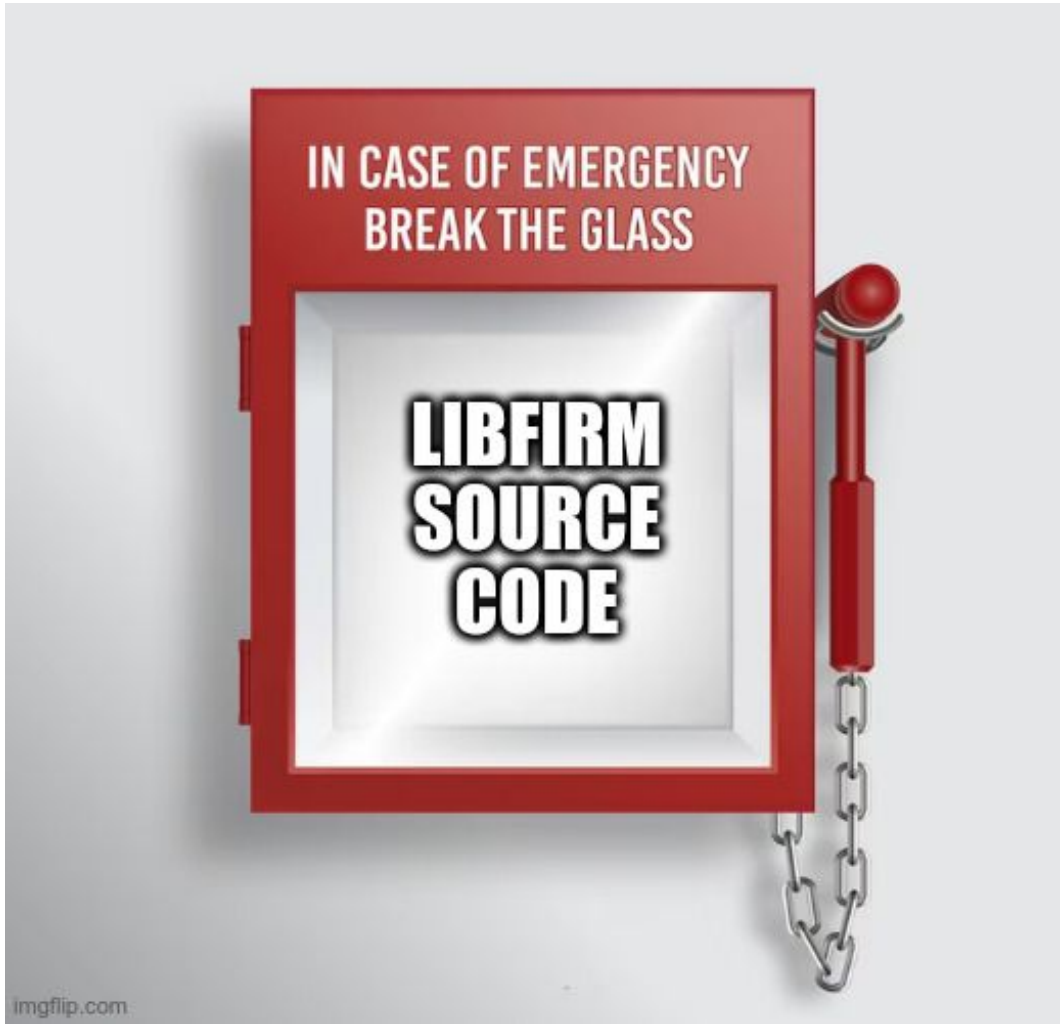
- Prefer same register for phi node arguments
- Prefer same register for src and dest var in three address code
- Prefer callee-saved register for live var across calls

Many good open source solver libraries out there:

- GLPK
- Cbc
- HiGHS

# References

- Hack's doctoral thesis
  - for everything about SSA reg alloc
- Braun and Hack '09
  - for fixing up spills & reloads
- libFirm
  - the code!
  - see course website for guidelines on approaching the codebase



# General tips

- Start stress-testing your register allocator **with very few registers (say 3)** — can catch many more bugs and edge cases
- Think it through before you type
- Comment extensively as you go
  - Document all **implicit invariants**
  - Add **assertions** if necessary
  - Sketch **proof of correctness**
- Use rubber duck or teammates

Weekly updates

Register allocator

**Peephole optimization ←**

# Peephole Optimization: overview

- assembly-level optimizations that take short snippets of assembly and transform them into better ones
- can often be performed simultaneously with codegen (try to emit good code in the first place)

# don’t do this

```
movq %r8, %r9
movq %r9, %r8

pushq %rax
popq %rax

movq <something> %rax
movq <otherthing> %rax
```

# Classic trick for zeroing a reggie

xor %reg, %reg

Not really an optimization so much as something to be aware of when reading x86.

Why do this over movq $0 %reg?

- Smaller code size -> better for icache
- register renamer recognizes the pattern points the register at hardwired zero !?!?

https://stackoverflow.com/a/29156824

# Fallthrough

Take advantage of it!

Loop bodies are executed more often than not, so try to fall through in to them, as the next instructions in memory should already be hot in cache.

# Stack Shenanigans

**-fomit-frame-pointer**

Keeping a chain of stack pointer is useful for debugging. Since compiled decaf executables are seldom distributed and don’t have variable-length arrays, you can just use %rbp as a general purpose register.

**The Red Zone**

Any function call is allowed to access up to 128 bytes below its initial stack pointer, without allocating stack space. For non-leaf calls, this is obviously dangerous. But for leaf calls, this means we can avoid stack manipulation if we need fewer than 128 bytes of space.

https://godbolt.org/z/K134nco4E

# Powers of 2

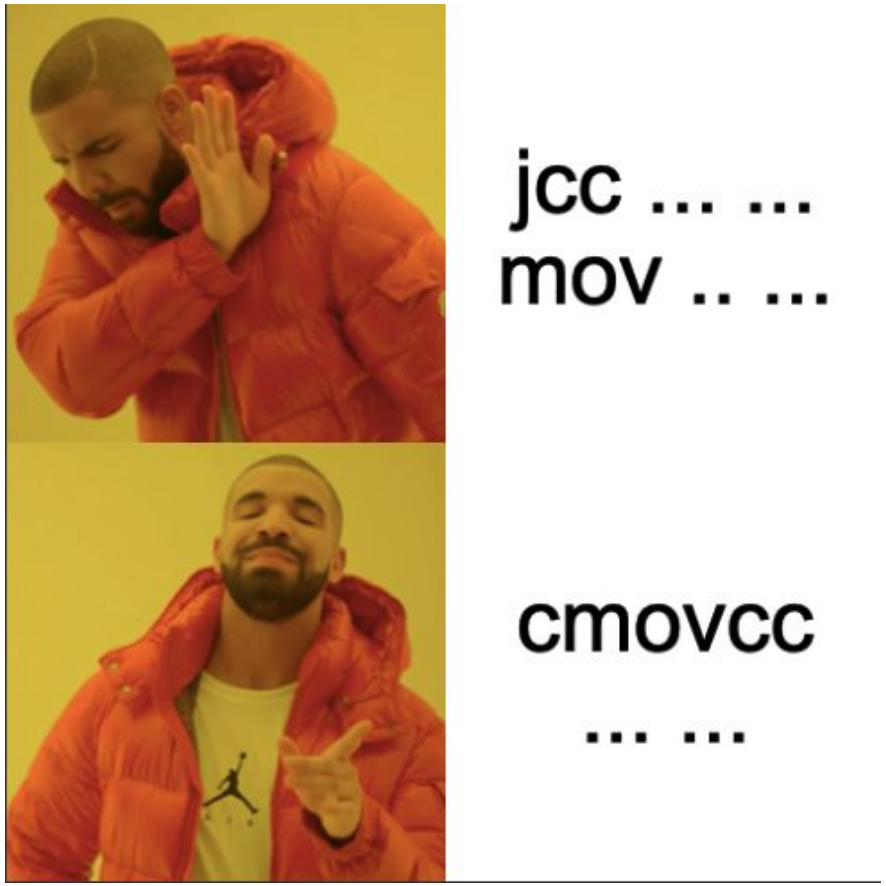
https://godbolt.org/z/GK4GTGWvE

# The many uses of lea

https://godbolt.org/z/hfTMcPTdh

# cmov

https://godbolt.org/z/oTnb564b3



# Hacker’s Delight

https://godbolt.org/z/c5s9qdecf
https://llvm.org/doxygen/DivisionByConstantInfo_8cpp_source.html
https://ridiculousfish.com/blog/posts/labor-of-division-episode-iii.html
https://ridiculousfish.com/blog/posts/labor-of-division-episode-i.html
https://github.com/llvm/llvm-project/blob/main/llvm/lib/Support/DivisionByConstantInfo.cpp#L74

A few notes:

- smaller magic numbers might generate faster muls, so you should try to use the smallest one

# Quiz 2

We’ll almost certainly ask you to calculate the magic number for some divisor.

# Just Kidding

Happy Hacking!