Dealing with Spills
Does this always work?

- Modified algorithm:
  - If no node can be safely removed, pick one anyway, mark it as a potential spill
  - Keep going

- If graph still can’t be colored, need to deal with spill
does this always work?

• Modified algorithm:
  • If no node can be safely removed, pick one anyway, mark it as a **potential spill**
  • Keep going

• If graph *still* can’t be colored, need to deal with spill
what do we do?

- If a variable cannot be assigned to a register, it needs to be placed on the stack.
- Need to generate extra instructions to load/store from stack --- those instructions need registers too!
- Naïve approach: reserve registers for managing spills
- Better approach: **rewrite code**
code rewriting

- Assign spilled temporary to memory location (e.g., T2)
- Introduce a new temporary for each instruction that uses T2

\[ T2 = T0 + T1 \]

becomes

\[ T19 = T0 + T1 \]

SW T19, [stack location of T2]
code rewriting

• Assign spilled temporary to memory location (e.g., T2)
• Introduce a new temporary for each instruction that uses T2

\[ T1 = T2 + T3 \]

becomes

\[ \text{LW } T37, \text{ [stack location of T2]} \]
\[ T1 = T37 + T3 \]
code rewriting

- Assign spilled temporary to memory location (e.g., T2)
- Introduce a new temporary for each instruction that uses T2
- Rerun liveness analysis, register allocation algorithm
code rewriting

- Why does this help?
- T2 is eliminated from the graph entirely
- Newly introduced temporaries have very short live range, so not too many edges!
- Less likely to have spills

- This is an example of **live range splitting**
  - Lots of refinement to reduce loads/stores
• Global register allocation allows for variables to be mapped to the same register across basic blocks

• Live range splitting allows for efficient generation of spill code

• Graph coloring-based allocation is effective but potentially slow
  • Iteration algorithm that keeps rewriting code, recomputing liveness, redoing allocation

• Many modern compilers, especially JITs, use simpler, but potentially less-efficient register allocators (e.g., linear-scan register allocation)
what do we have?

• We now have a full-featured language:
  • Arithmetic operations
  • Control flow
  • Functions

• And compiler:
  • Code generation
  • Register allocation

• Good base to keep adding features!
next: module 3!