Lecture 6

Register Allocation

I Introduction

II Abstraction and the Problem

III Algorithm

Reading: Chapter 8.8.4
Before next class: Chapter 10.1 - 10.2

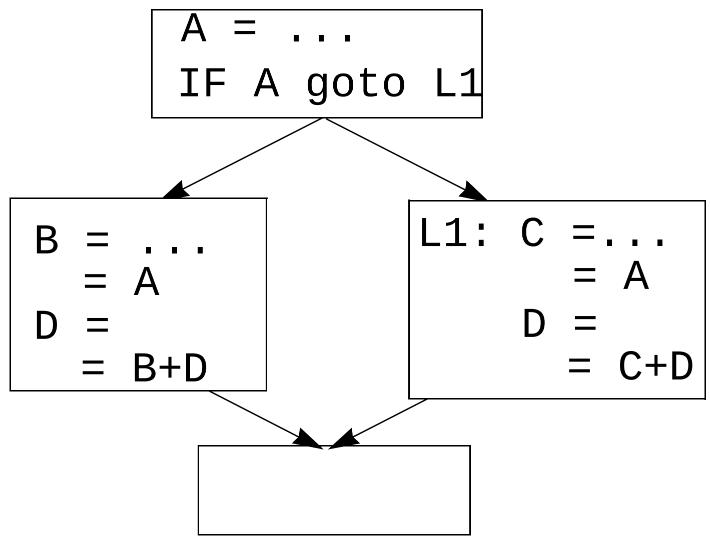
I. Motivation

- **Problem**
  - Allocation of variables (pseudo-registers) to hardware registers in a procedure
- **Perhaps the most important optimization**
  - Directly reduces running time (memory access $\Rightarrow$ register access)
  - Useful for other optimizations e.g. cse assumes old values are kept in registers.
- **More important as processor speeds grow faster than memory speeds**

Goal

- **Find an assignment for all pseudo-registers, if possible.**
- **If there are not enough registers in the machine, choose registers to spill to memory**

Example



II. An Abstraction for Allocation & Assignment

- **Intuitively**
  - Two pseudo-registers **interfere** if at some point in the program they cannot both occupy the same register.

- **Interference graph**: an undirected graph, where
  - nodes = pseudo-registers
  - there is an edge between two nodes if their corresponding pseudo-registers interfere

- **What is not represented**
  - Extent of the interference between uses of different variables
  - Where in the program is the interference

Register Allocation and Coloring

- A graph is **n-colorable**
  if every node in the graph can be colored with one of the n colors
  such that two adjacent nodes do not have the same color.

- **Assigning n registers (without spilling) = Coloring with n colors**
  - assign a node to a register (color) such that
    no two adjacent nodes are assigned same registers(colors)

- **Is spilling necessary? = Is the graph n-colorable?**

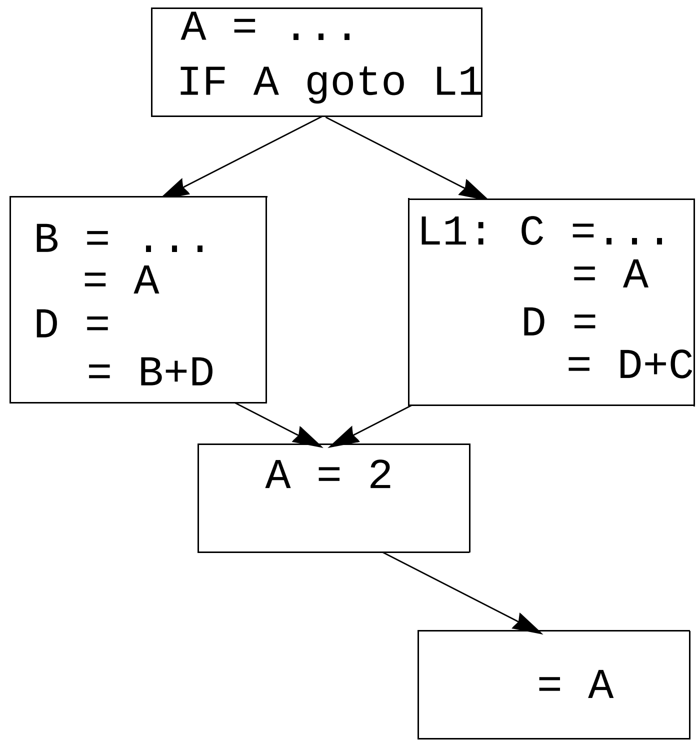
- **To determine if a graph is n-colorable is NP-complete, for n>2**
  - Too expensive
  - Heuristics

III. Algorithm

- **Step 1. Build an interference graph**
  - a. refining notion of a node
  - b. finding the edges

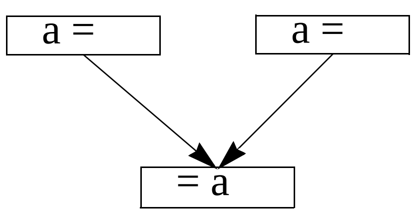
- **Step 2. Coloring**
  - use heuristics to try to find an n-coloring
    - Successful $\Rightarrow$ colorable and we have an assignment
    - Failure $\Rightarrow$ graph not colorable, or
      graph is colorable, but it is too expensive to color

Step 1a. Nodes in an Interference Graph

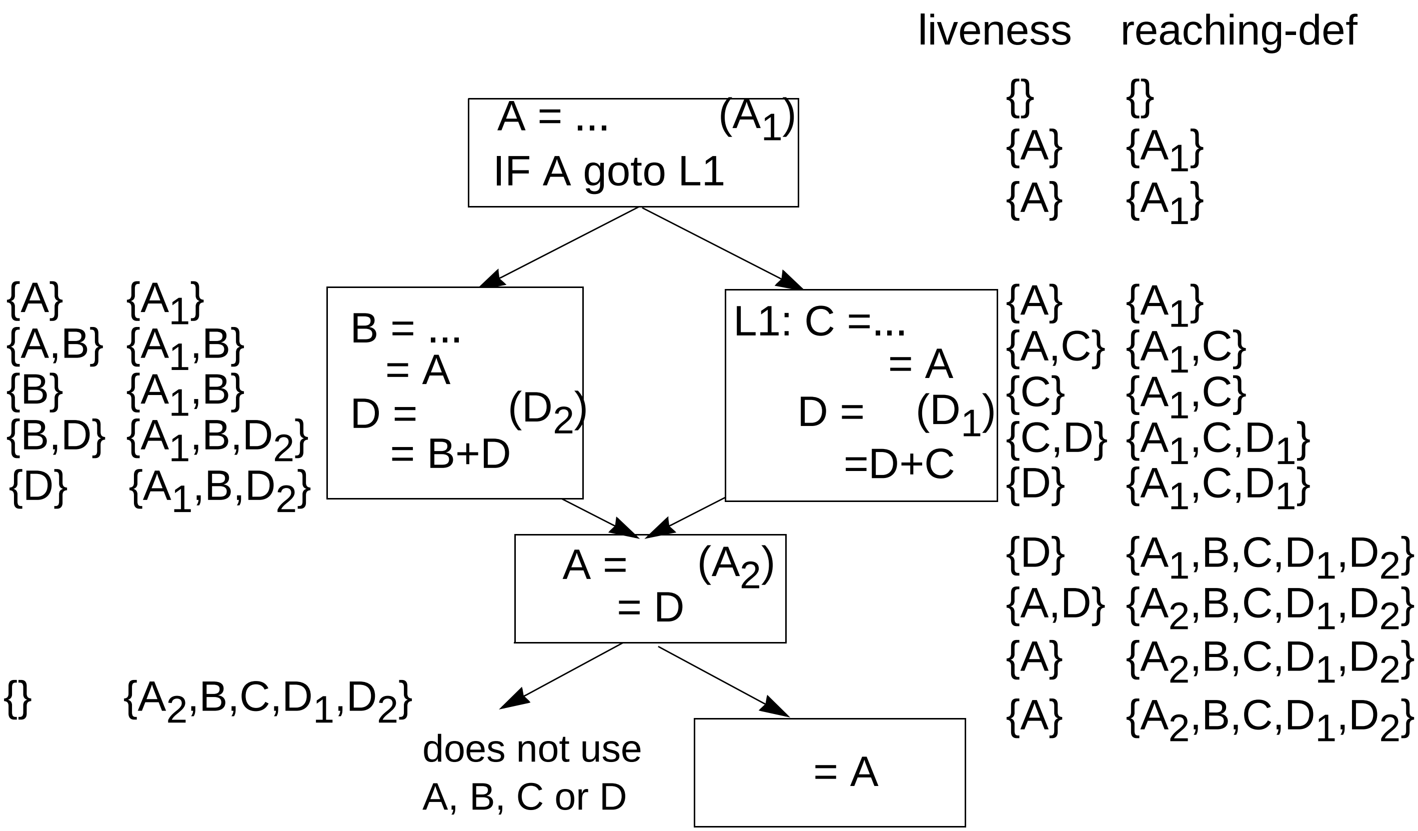


Live Ranges and Merged Live Ranges

- **Motivation: to create an interference graph that is easier to color**
  - Eliminate interference in a variable's "dead" zones.
  - Increase flexibility in allocation:
    can allocate same variable to different registers
- A **live range** consists of a definition and all the points in a program (e.g. end of an instruction) in which that definition is live.
  - How to compute a live range?
- Two overlapping live ranges for same variable must be merged



Example (revisited)



Merging Live Ranges

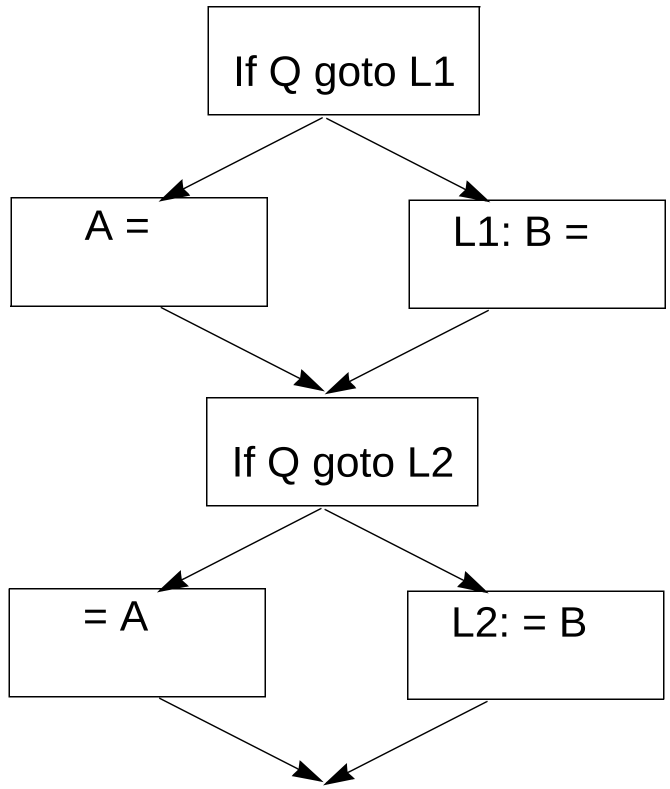
- **Merging definitions into equivalence classes**
  - Start by putting each definition in a different equivalence class
  - For each point in a program
    - if variable is live,
      and there are multiple reaching definitions for the variable
    - merge the equivalence classes of all such definitions into one equivalence class
- **From now on, refer to merged live ranges simply as live ranges**

Step 1b. Edges of Interference Graph

- **Intuitively**
  - Two live ranges (necessarily of different variables) may interfere if they overlap at some point in the program.
  - Algorithm
    - At each point in program
      enter an edge for every pair of live ranges at that point.

- **An optimized definition&algorithm for edges:**
  - Algorithm:
    check for interference only at the start of each live range
  - Faster
  - Better quality

Example 2

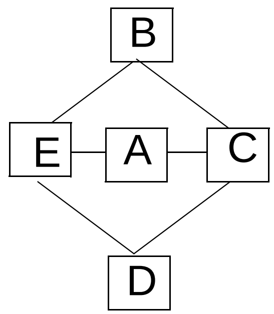


Step 2. Coloring

- **Reminder: coloring for n > 2 is NP-complete**
- **Observations**
  - a node with degree $< n \Rightarrow$
    - can always color it successfully, given its neighbors' colors
  - a node with degree $= n \Rightarrow$
  - a node with degree $> n \Rightarrow$

Coloring Algorithm

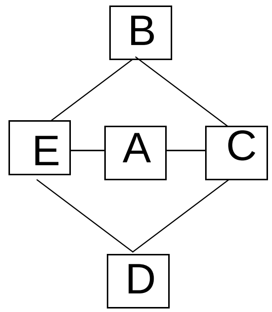
- **Algorithm**
  - Iterate until stuck or done
    - Pick any node with degree < n
    - Remove the node and its edges from the graph
  - If done (no nodes left)
    - reverse the process and add colors
- **Example (n = 3)**



- **Note: degree of a node may drop in iteration**
- **Avoids making arbitrary decisions that make coloring fail**

What Does Coloring Accomplish?

- **Done:** colorable, also obtained an assignment
- **Stuck:**
  - colorable or not?



What to Do if Coloring Fails?

- **Use heuristics to improve its chance of success and to spill code**

```
Build interference graph

Iterative until there are no nodes left
        If there exists a node v with less than n neighbor
                place v on stack to register allocate
        else
                v = node chosen by heuristics
                    (least frequently executed, has many neighbors)
                place v on stack to register allocate (mark as spilled)
        remove v and its edges from graph

While stack is not empty
        Remove v from stack
        Reinsert v and its edges into the graph
        Assign v a color that differs from all its neighbors
        (guaranteed to be possible for nodes not marked as spilled)
```

Summary

- **Problems:**
  - Given n registers in a machine, is spilling avoided?
  - Find an assignment for all pseudo-registers, whenever possible.
- **Solution**
  - Abstraction: an interference graph
    - nodes: merged live ranges
    - edges: presence of live range at time of definition
  - Register allocation and assignment
    = n-colorability of interference graph
    $\Rightarrow$ NP-complete
  - Heuristics to find an assignment for n colors
    - successful: colorable, and finds assignment
    - not successful: colorability unknown & no assignment