Lecture 12

Register Allocation & Spilling

I. Introduction
II. Abstraction and the Problem
III. Algorithm
IV. Spilling

ALSU 8.8

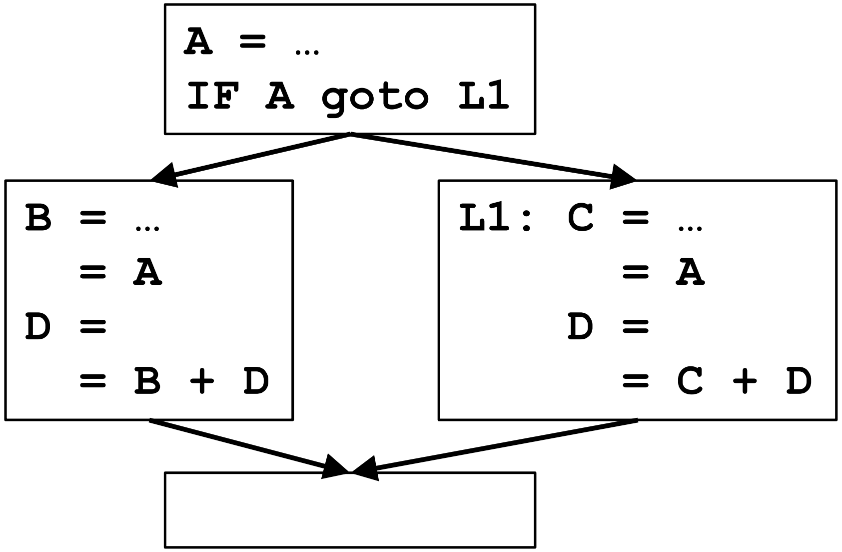
I. Introduction

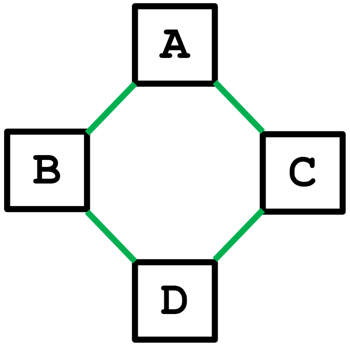
- **Problem**
  - Allocation of variables (pseudo-registers) to hardware registers in a procedure
- **Motivation: A very important optimization!**
  - Directly reduces running time
    - (memory access → register access)
  - Useful for other optimizations
    - e.g. CSE assumes old values are kept in registers

Goals

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

Register Assignment Example





- Find an assignment (without spilling) that uses only 2 registers:
  - A and D in one register, B and C in the other
- What does this assignment assume?
  - After code segment, no use of A & at most one of B or C is used

II. An Abstraction for Allocation & Assignment

- **Intuitively**
  - Two pseudo-registers (i.e., program variables) **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

Interfere many times vs. once

E.g., cold path vs. hot path

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 register (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
  - Use heuristics

III. Algorithm: Overview

Step 1. Build an interference graph

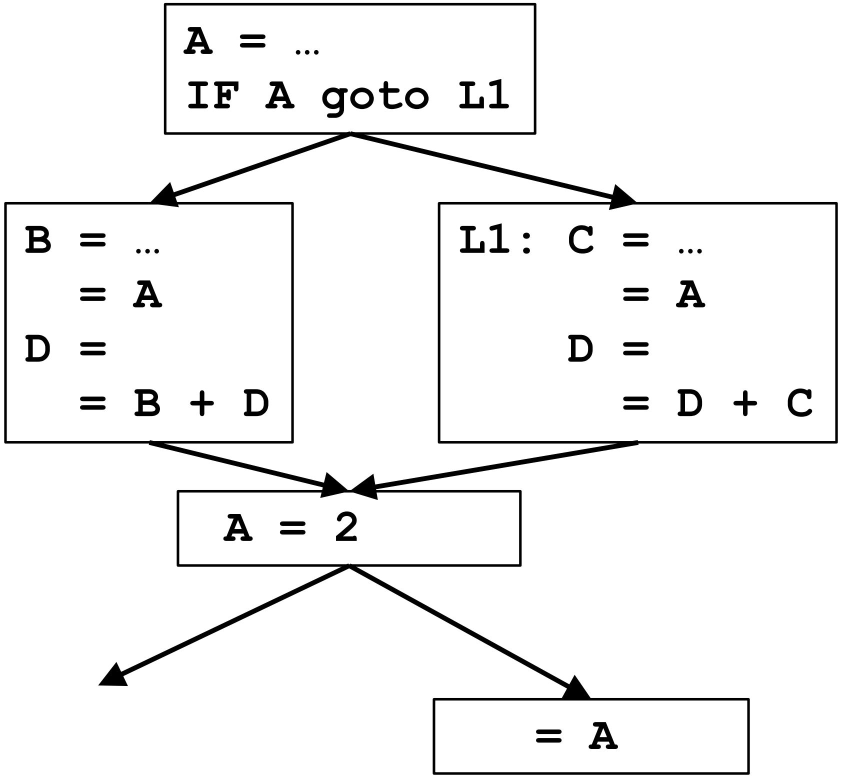
a. refining notion of a node

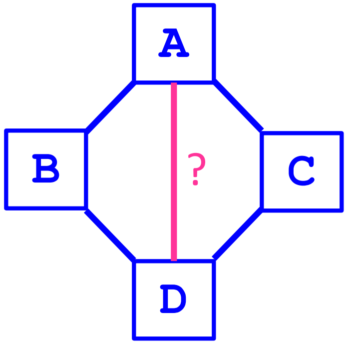
b. finding the edges

Step 2. Coloring

– use heuristics to try to find an n-coloring
  - Success:
    – colorable and we have an assignment
  - Failure:
    – graph not colorable, or
    – graph is colorable, but heuristics did not find a coloring

Step 1a. Nodes in an Interference Graph





Interference Graph

Should we add A-D edge?

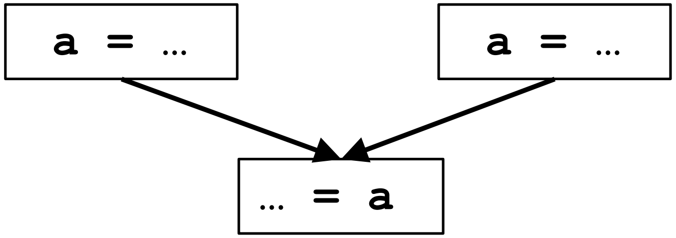
No, since new def of A

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?
    - **live variables** & **reaching definitions**

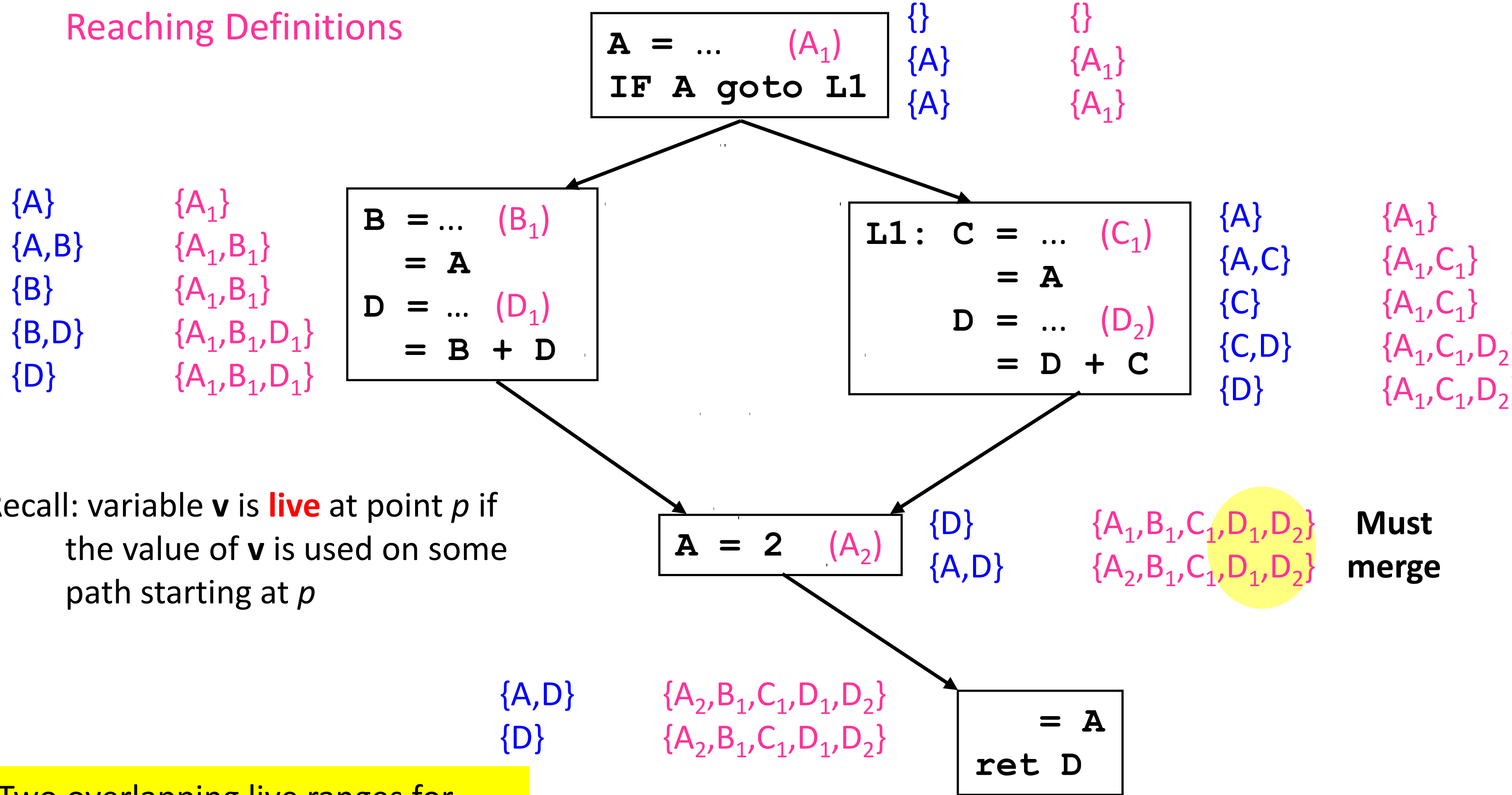
- Two overlapping live ranges for the **same** variable must be merged



Register Allocation Example (Revisited)

Live Variables

Reaching Definitions



Recall: variable **v** is **live** at point *p* if the value of **v** is used on some path starting at *p*

Two overlapping live ranges for the **same** variable must be merged

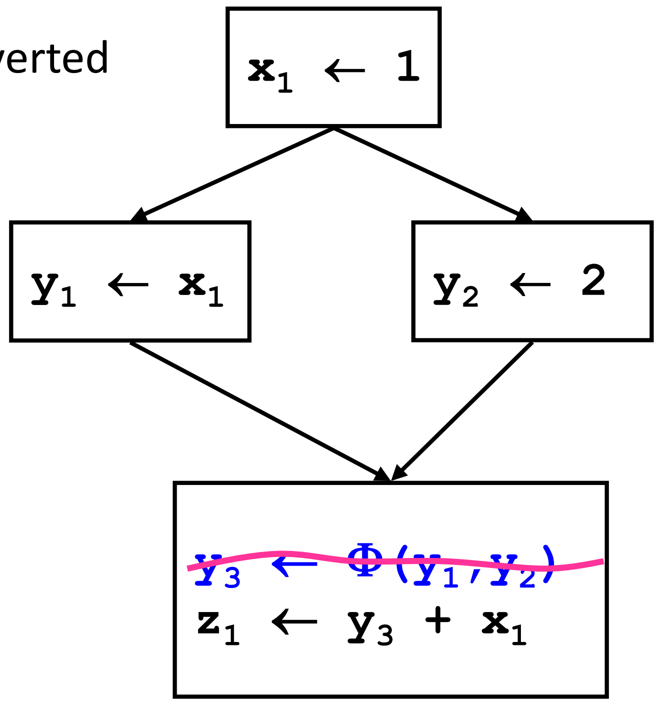
Merging Live Ranges

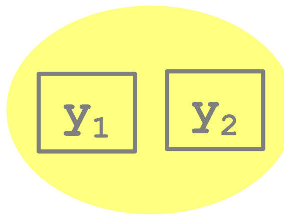
- **Merging definitions into equivalence classes**
  - Start by putting each definition in a different equivalence class
  - Then, for each point in a program:
    - if (i) variable is live, and (ii) there are multiple reaching definitions for the variable, then:
      - merge the equivalence classes of all such definitions into one equivalence class
    - *(Sound familiar?)*

- **From now on, refer to merged live ranges simply as live ranges**
  - merged live ranges are also known as "webs"

SSA Revisited: What Happens to Φ Functions

- Now we see why it is unnecessary to "implement" a Φ function
  - Φ functions and SSA variable renaming simply turn into merged live ranges
- When you encounter: $x_4 = \Phi(x_1, x_2, x_3)$
  - merge x_1, x_2, x_3, and x_4 into the same live range
  - delete the Φ function
- Now you have effectively converted back out of SSA form





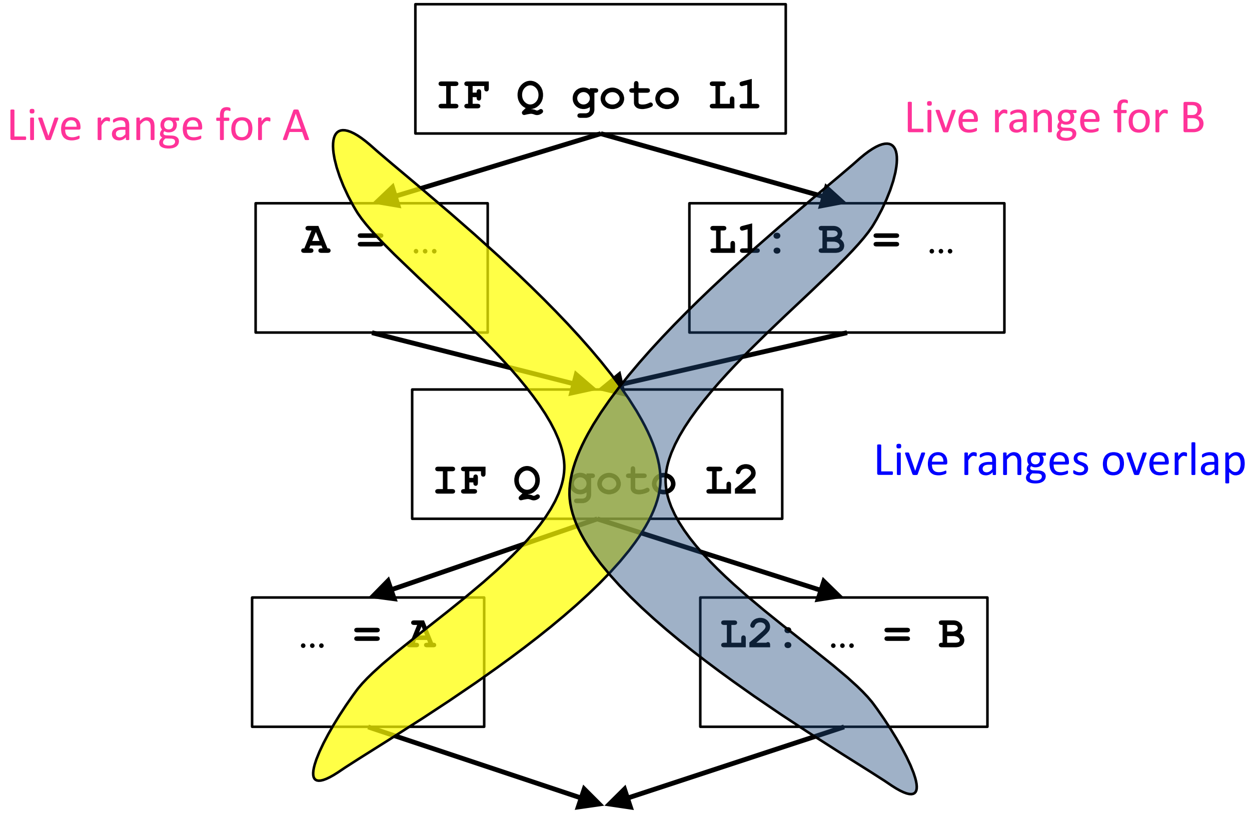
Merge

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 the 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

Live Range Example 2

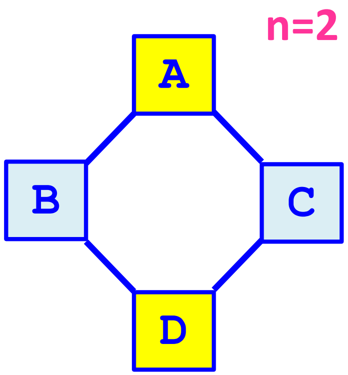


Because ranges overlap: Won't assign A and B to same register
(even though would have been ok: path sensitive vs. path insensitive analysis)

Step 2. Coloring

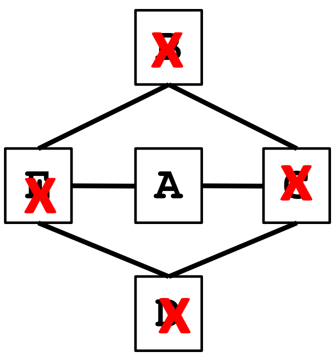
- **Reminder: coloring for n > 2 is NP-complete**

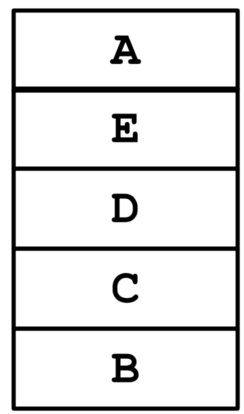
- **Observations:**
  - a node with degree < n ⇒
    - can always color it successfully, given its neighbors' colors
  - a node with degree = n ⇒
    - can color only if at least two neighbors share same color
  - a node with degree > n ⇒
    - maybe, not always

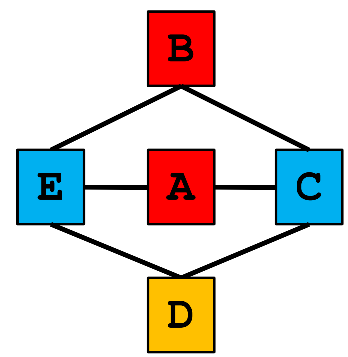


Coloring Heuristic

- 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 process and add colors
- Example (n = 3):







- Note: degree of a node may drop in iteration
- Avoids making arbitrary decisions that make coloring fail (e.g., B, A, D different colors)

Coloring + Register Assignment

- **Apply coloring heuristic**

  Build interference graph
  Iterate until there are no nodes left
  If there exists a node v with less than n neighbor
  push v on register allocation stack
  else
  return (coloring heuristics fail)
  remove v and its edges from graph

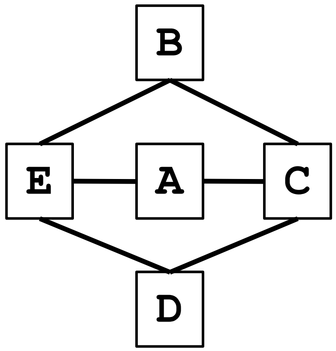
- **Assign registers**

  While stack is not empty
  Pop v from stack
  Reinsert v and its edges into the graph
  Assign v a color that differs from all its neighbors

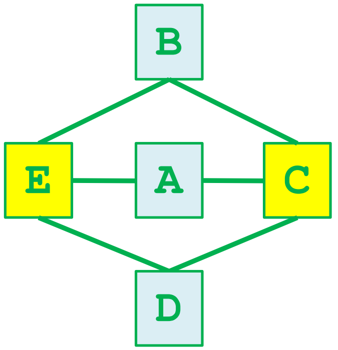
What Does Coloring Accomplish?

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

Is there a n=2 coloring? yes



n=2



Will heuristic find a coloring?

No: Stuck since no node with degree < n

IV. Extending Coloring: Design Principles

- **A pseudo-register is**
  - Colored successfully: allocated a hardware register
  - Not colored: left in memory
- **Objective function**
  - Cost of an uncolored node:
    - proportional to number of uses/definitions (dynamically)
    - estimate by its loop nesting
  - Objective: minimize sum of cost of uncolored nodes
- **Heuristics**
  - Benefit of spilling a pseudo-register:
    - increases colorability of pseudo-registers it interferes with
    - can approximate by its degree in interference graph
  - Greedy heuristic
    - spill the pseudo-register with lowest cost-to-benefit ratio, whenever spilling is necessary

Spilling to Memory

- CISC architectures
  - can operate on data in memory directly
  - memory operations are slower than register operations
- RISC architectures
  - machine instructions can only apply to registers
  - Use
    - must first load data from memory to a register before use
  - Definition
    - must first compute RHS in a register
    - store to memory afterwards
  - Even if spilled to memory, needs a register at time of use/definition

Chaitin: Coloring and Spilling

- **Apply coloring heuristic**

Build interference graph
Iterate until there are no nodes left
    If there exists a node v with less than n neighbor
        push v on register allocation stack
    else
        v = node with highest degree-to-cost ratio
        mark v as spilled
    remove v and its edges from graph

- **Spilling may require use of registers (must reload at each use, store at each def); change interference graph**

While there is spilling
    rebuild interference graph and perform step above

- **Assign registers**

While stack is not empty
    Pop v from stack
    Reinsert v and its edges into the graph
    Assign v a color that differs from all its neighbors

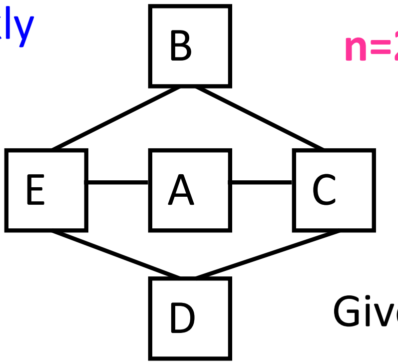
Spilling

- What should we spill?
  - Something that will eliminate a lot of interference edges
  - Something that is used infrequently
  - Maybe something that is live across a lot of calls?

- One Heuristic:
  - Cost-to-degree-ratio = [(# defs & uses)*$10^{\text{loop-nest-depth}}$]/degree
  - Spill node with highest degree-to-cost ratio

Quality of Chaitin's Algorithm

- Problem: Can give up on coloring too quickly



n=2

Gives up but colorable

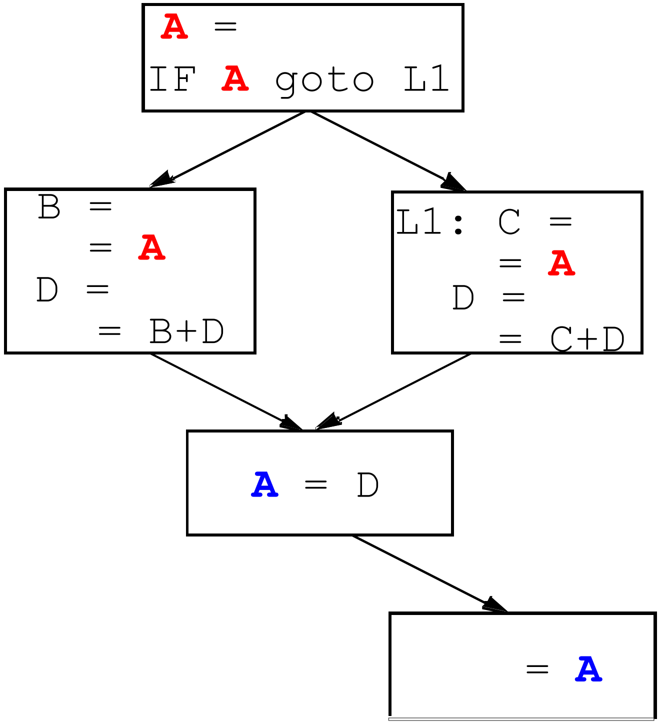
An optimization: "Prioritize the coloring"

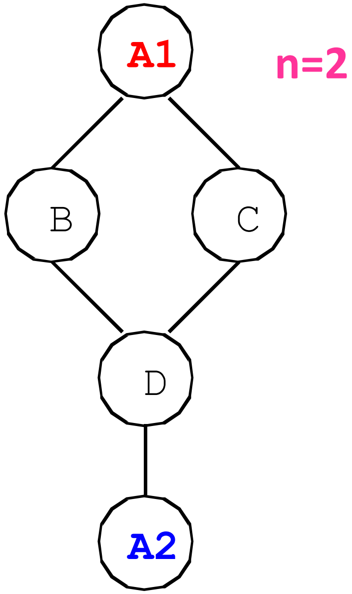
- Still eliminate a node and its edges from graph
- Do not commit to "spilling" just yet
- Try to color again in assignment phase

- Problem: All or nothing
  - Why not try to keep a pseudo-register in a hardware register part of the time?

Splitting Live Ranges

- Different perspective: Instead of choosing variables to spill, choose live ranges to split
- Split pseudo-registers into live ranges to make interference graph easier to color
  - Eliminate interference in a variable's "dead" zones
  - Increase flexibility in allocation:
    - can allocate same variable to different registers





Insight

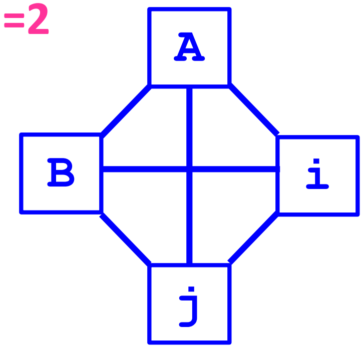
- Split a live range into smaller regions (by paying a small cost) to create an interference graph that is easier to color
  - Eliminate interference in a variable's "nearly dead" zones.
    - Cost: Memory loads and stores
      - Load and store at boundaries of regions with no activity
    - Initially: # active live ranges at a program point can be > # registers
  - Can allocate same variable to different registers
    - Cost: Register operations
      - a register copy between regions of different assignments
    - Goal: # active live ranges cannot be > # registers

Splitting Live Range Example

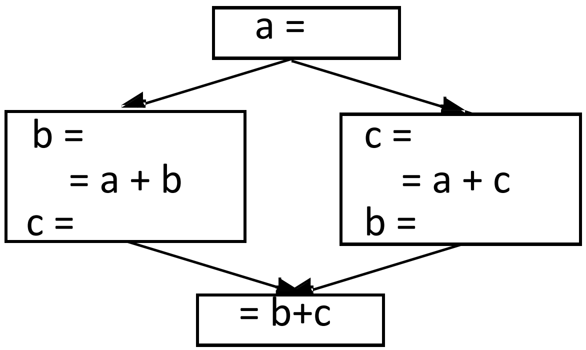
spill **i** (outer loop) — spill **B** (first inner loop) — spill **A** (second inner loop)

```
FOR i = 0 TO 10
    FOR j = 0 TO 10000
        A = A + ...
        (does not use B)
    FOR j = 0 TO 10000
        B = B + ...
        (does not use A)
```

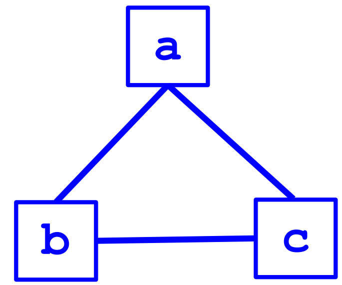
n=2



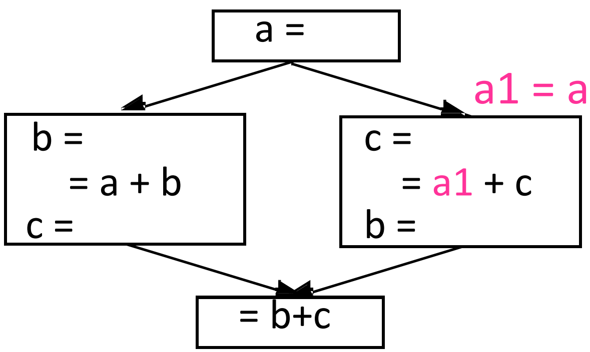
Example: Allocate Same Variable to Different Registers

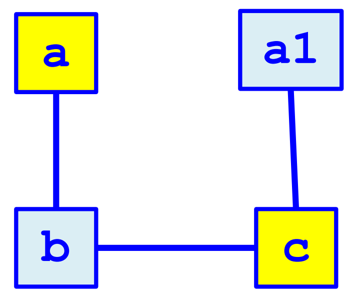


n=2



Can't 2-color





Can 2-color
("a" gets 2 regs)

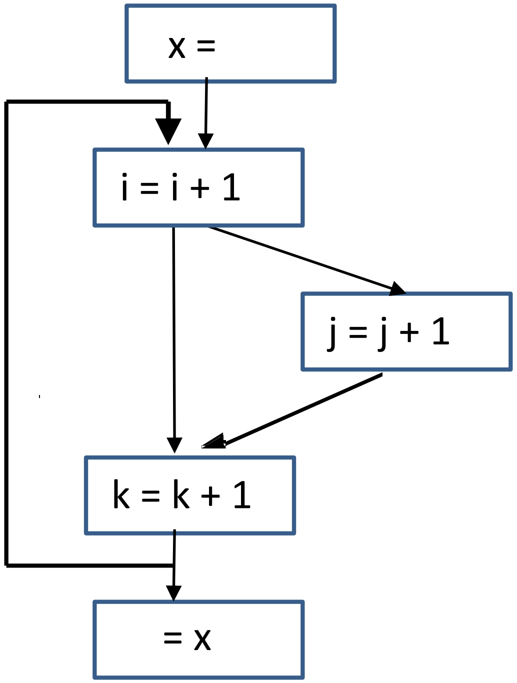
Live Range Splitting: Recap So Far

- When do we apply live range splitting? — when more live ranges than registers
- Which live range to split? — based on cost/benefit ratio
- Where should the live range be split? — split where large inactive region
- How to apply live-range splitting with coloring?
  - Advantage of coloring:
    - defers arbitrary assignment decisions until later
  - When coloring fails to proceed, may not need to split live range
    - degree of a node >= n does not mean that the graph definitely is not colorable
  - Interference graph does not capture positions of a live range

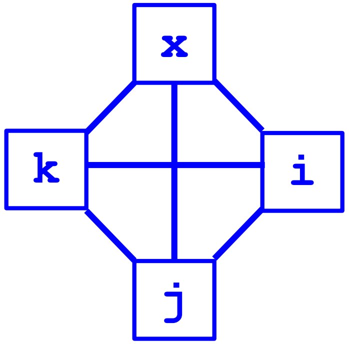
A Spilling Algorithm Focused on Live-Range Splitting

- Observation: spilling is absolutely necessary if
  - number of live ranges active at a program point > n
- Apply live-range splitting before coloring
  - Identify a point where number of live ranges > n
  - For each live range active around that point:
    - find the outermost "block construct" that does not access the variable
  - Choose a live range with the largest inactive region
  - Split the inactive region from the live range

n=3



split x,
then can color



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: live ranges
    - edges: presence of live range at time of definition
  - Register Allocation and Assignment problems
    - equivalent to **n-colorability** of interference graph

      ➔ NP-complete
  - Heuristics to find an assignment for n colors
    - successful: colorable, and finds assignment
    - not successful: colorability unknown & no assignment

Today's Class

I. Introduction
II. Abstraction and the Problem
III. Algorithm
IV. Spilling

Wednesday's Class

- Register Allocation: Coalescing