CSC 416/565: DESIGN AND CONSTRUCTION OF COMPILERS

West Chester University
Dr. Richard Burns
Fall 2023

Register Allocation

Motivation: intermediate code uses unlimited number of registers to hold temporary values

- Simplifies optimization, code generation
- Complicates final translation to assembly

The task: rewrite intermediate code to assign the *many* temporaries to the *small* number of machine registers

- Assign *multiple* temporaries to each register
  - Cannot change program behavior!

Example

Assume a and e are dead after use.

```
a := c+d
e := a+b
f := e-1
```

Prompt: how many registers are needed?

Golden Rule

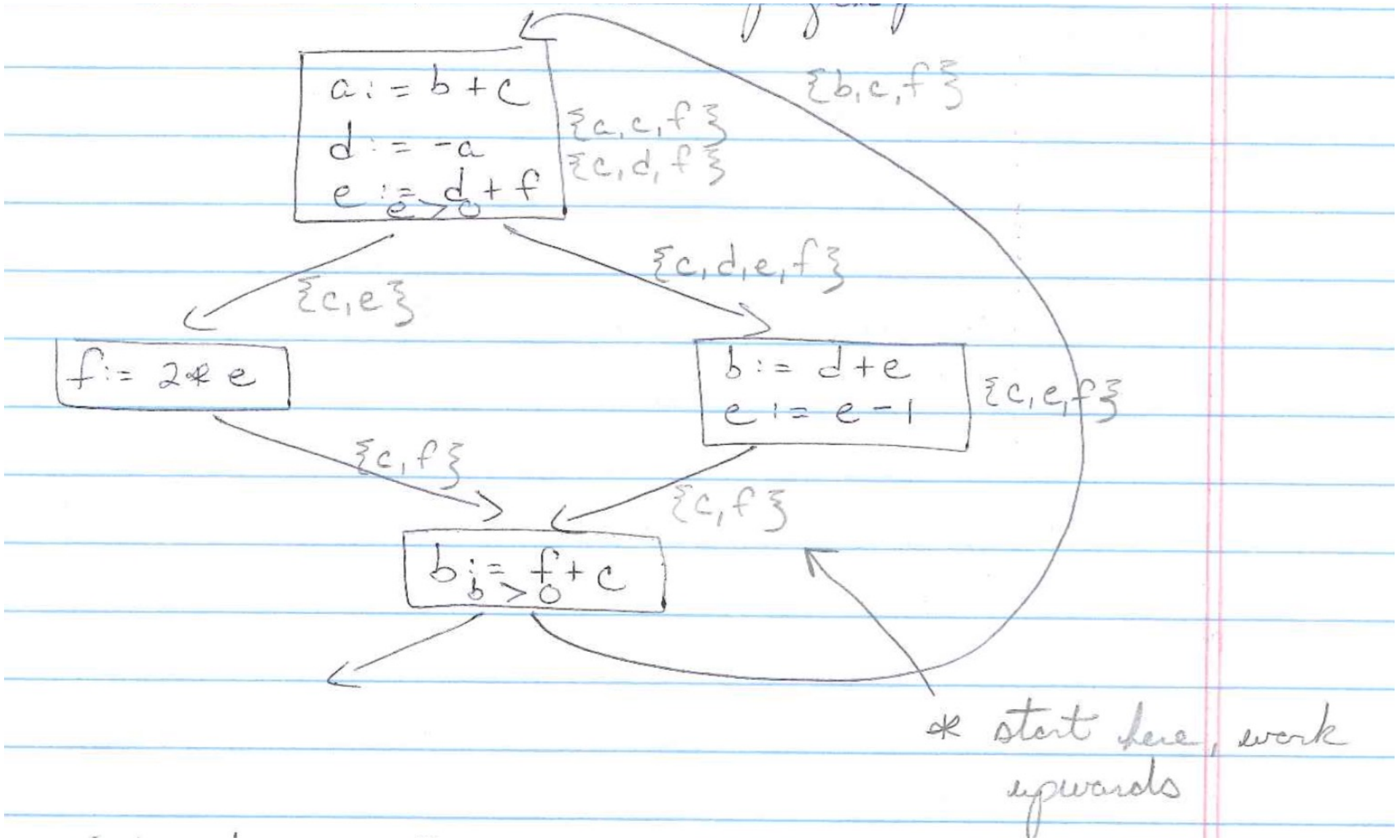
Temporaries t_1 and t_2 can share the same register if at most one of t_1 or t_2 is alive at any point in the program.

- If t_1 and t_2 are live at the same time, they cannot share a register
- Need to determine the live variables at every point in the program.
- How can we do this?

Use liveness analysis.

Example: Control Flow Graph

1. What are the live variables at each program point?
2. What registers to assign?


a := b + c
d := -a
e := d + f
{a, c, f}
{c, d, f}
{b, c, f}
{c, e}
{c, d, e, f}
f := 2 * e
b := d + e
e := e - 1
{c, e, f}
{c, f}
{c, f}
b := f + c
* start here, work upwards

Graph Coloring

Can solve register assignment using graph coloring algorithm.

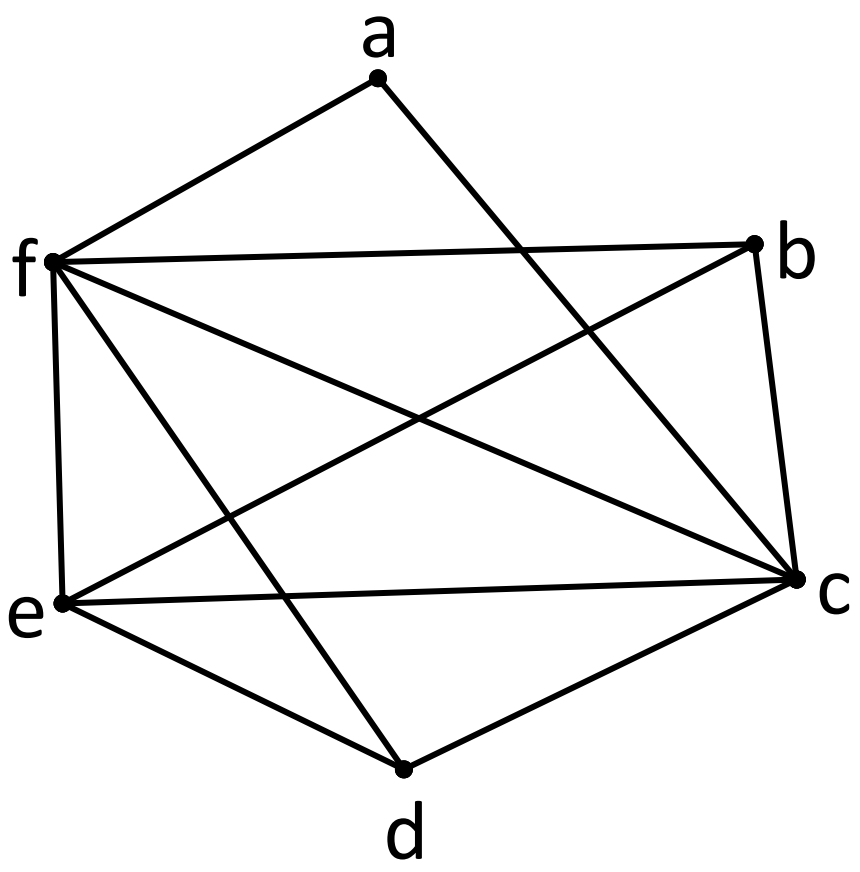
(There are *linear time approximation algorithms* despite graph coloring being an NP-complete problem.)

Graph Coloring

Big Idea: **Construct a "register" interference graph**

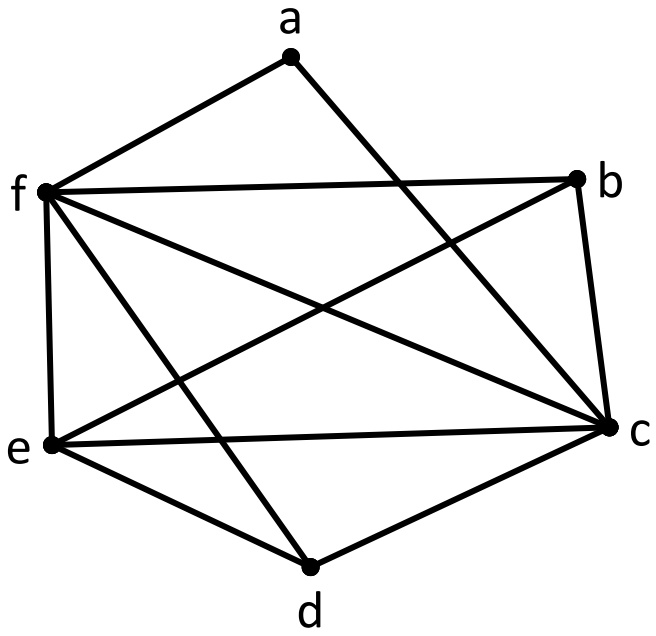
- Undirected graph
- Node for each temporary
- Edge between t_1 and t_2 if they are live simultaneously at same point in the program (and cannot be assigned to the same register)

Example: Interference Graph



- b and c cannot be in same register
- b and d can be in same register

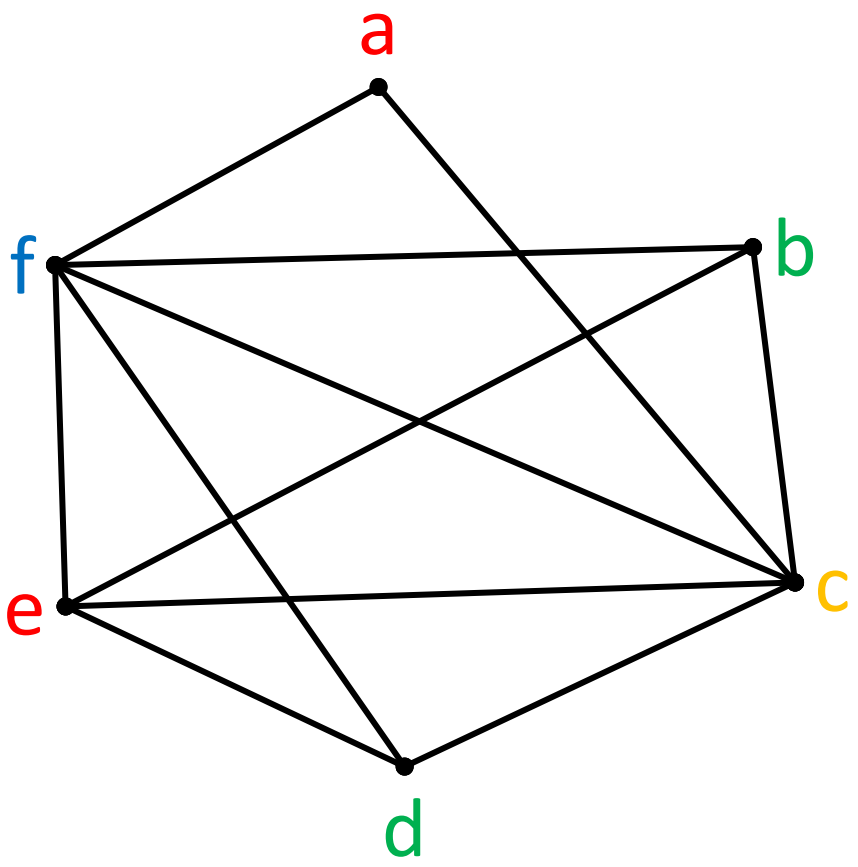
- Find a coloring of the graph, such that no pair of nodes connected by an edge is assigned the same color.
- Use as few colors as possible?
- *Terminology:* a graph is *k-colorable* if it has a coloring with k colors



In register assignment problem: colors = registers

- Assign colors (registers) to graph nodes (temporaries)
- Let k = # of machine registers
- If interference graph if k-colorable, then there is a register assignment that uses no more than k registers.

The interference graph is 4-colorable.



BLUE = r1
RED = r2
GREEN = r3
YELLOW = r4

Coloring By Simplification

Phases: (1) build, (2) simplify, (3) spill, (4) select

1. Build the interference graph
2. Simplify (simple heuristic):
   - pick a node t with fewer than k neighbors, where k is the # of machine registers
   - eliminate node t and its edges
   - *observation:* if the resulting graph is k-colorable, then so is the original graph
     - (a free color can always be found for t since it has at most k-1 neighbors)
   - put t on a stack
   - repeat until the graph is empty (each such simplification will decrease the degrees of the other nodes, leading to more opportunity for simplification)

Coloring By Simplification

Phases: (1) build, (2) simplify, (3) spill, (4) select

3. Spill - we get stuck, the heuristic fails
   - more on this later
4. Select - we get to an empty graph
   - assign colors to nodes on the stack
   - start w/ last node added (top of the stack)
   - pop stack, add node to graph, assign color to node that is different than neighbors' coloring
     - (there must be a color for it because of the observation made during (2) simplify)

Simplification and Selection Example

Spilling

Spilling: occurs when the graph coloring heuristic fails to find a color (e.g. at some point *all* nodes in the graph have: $degree \geq k$)

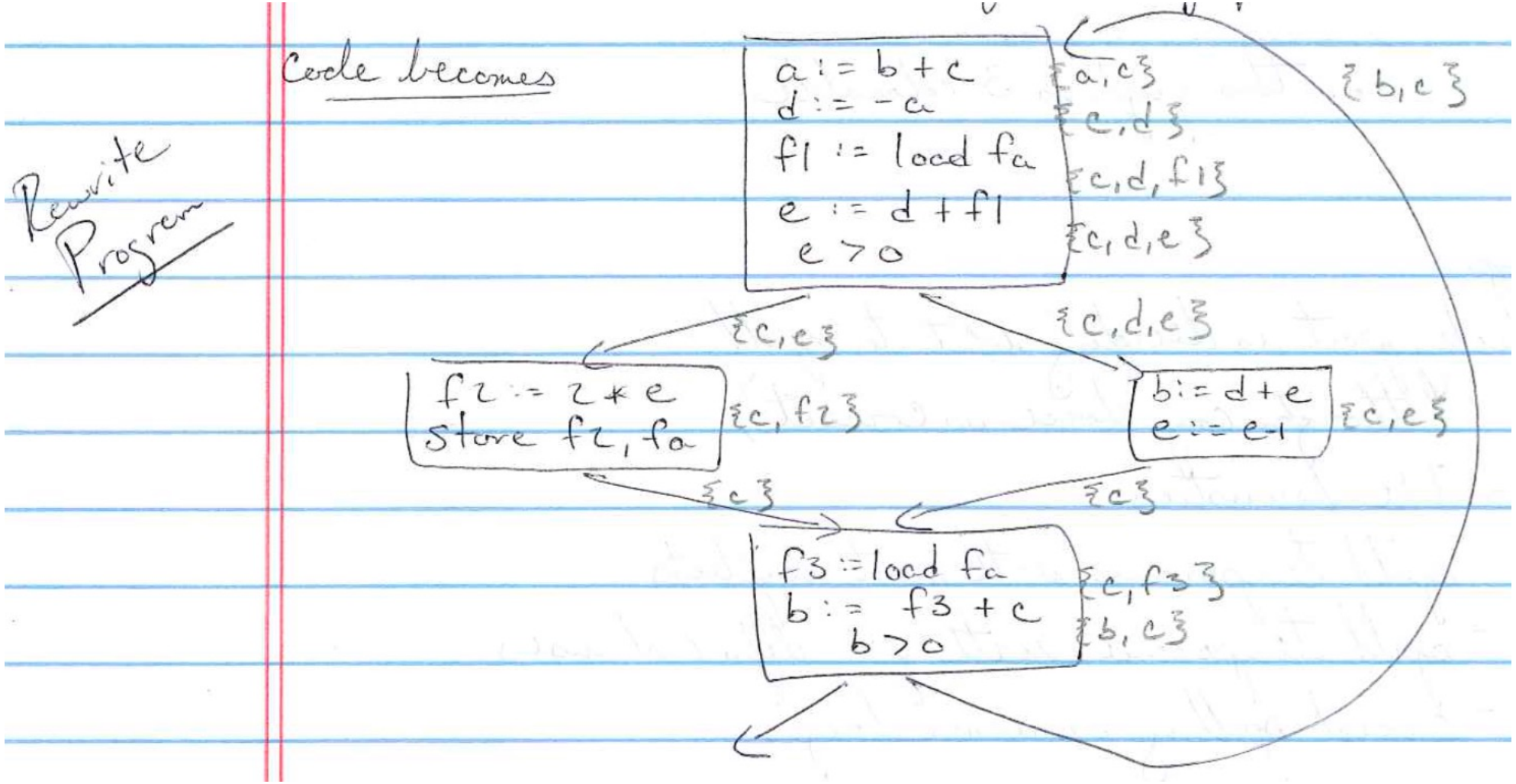
- Implication: cannot hold *all* values in registers.
- Some values are spilled to memory.

Example

- Try to find 3-coloring of the previous interference graph.
- Mark/pick a node as a candidate for spilling *(potential spill node)*
  - Will represent it in memory, not registers, in program execution
- Try again to find a 3-coloring.

Spilling

- For actual spills, for some temporary f:
- allocate a memory location for f -- typically in current stack frame
- *Step 1:* rewrite program to fetch f from memory just *before* each use; and store it back *after* each definition
- e.g. store temporary f at memory location f_a
  - f := load fa // before each operation that reads f
  - store f, fa // after every operation that writes f



Thus, a spilled temporary will turn into *several* new temporaries with *tiny* live ranges.

Spilling

- *Step 2:* Recompute liveness
- f_i is live only between `fi = load fa` and the *next* instruction
- f_i is live only between `store fi, fa` and the *previous* instruction

Observe: spilling reduces the live range of f

- And thus reduces its interferences
- Which results in fewer interference graph neighbors
- But new temporaries may interfere with other temporaries in the graph

Spilling

- *Step 3:* re-run “Simplify” step on rewritten program until graph coloring heuristic succeeds with no spills.

Example – New Interference Graph

How many registers are needed now?

Wrapping Up

- The trick part is deciding *which* temporary to spill.
- Any choice is “correct”, but some spills are more desirable than others.
- Possible heuristics:
  - Spill temporaries with most conflicts
  - Spill temporaries with few defs and uses
  - Avoid spilling in inner loops