Compiler Construction

BITS Pilani
Pilani Campus

Vinti Agarwal

April 2021

innovate achieve lead

BITS Pilani

Pilani Campus

CS F363, Compiler Construction

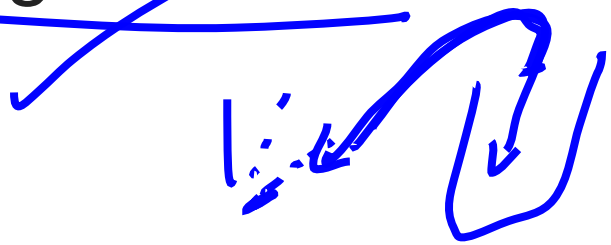
Lecture topic: Register Spilling

In previous lecture:

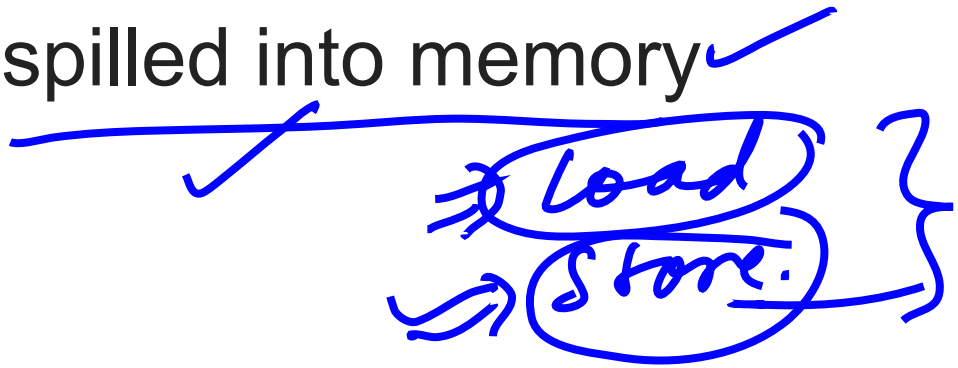
We have seen Register Allocation:

- Graph Coloring
- compute Liveness
  - *register interference graph*
- Graph coloring Heuristic

- What happens if graph coloring heuristics fails to find a coloring?

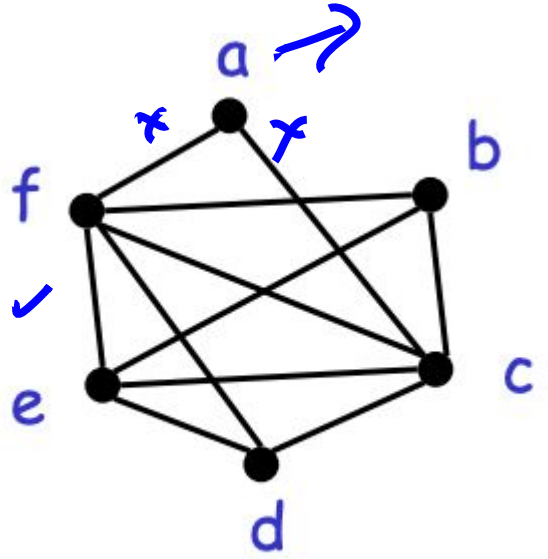


- In this case we can't hold all values in registers
  - Some values are spilled into memory



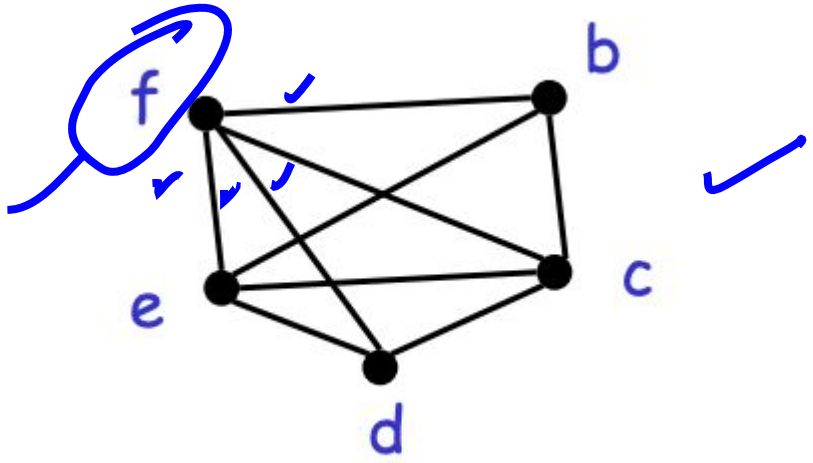
- When the heuristic Fails?
- If all nodes have k or more neighbors ?

Example: Try to find a 3-coloring of the RIG:



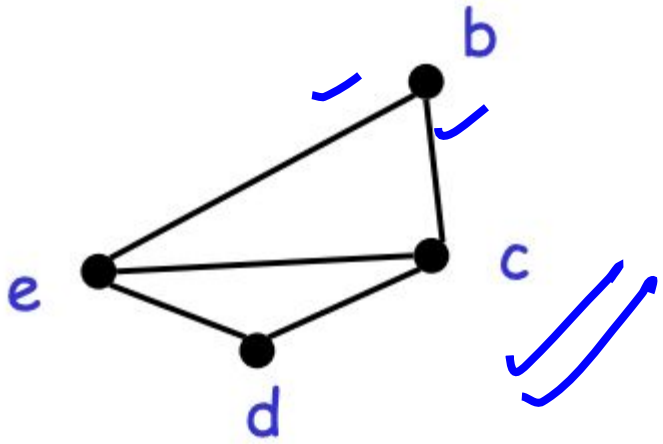
Example

- Remove a and get stuck (as shown below)
- Pick a node as a candidate for spilling
  - A spilled temporary "lives" in memory
  - Assume that f is picked as a candidate



Example

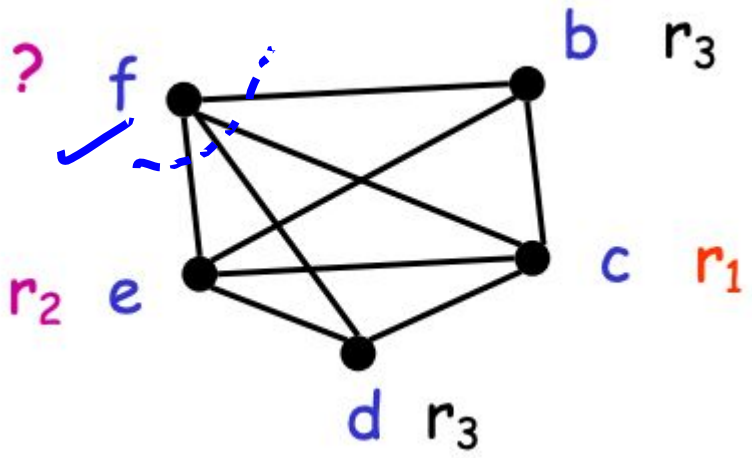
- Remove f and continue the simplification
  – Simplification now succeeds: b, d, e, c

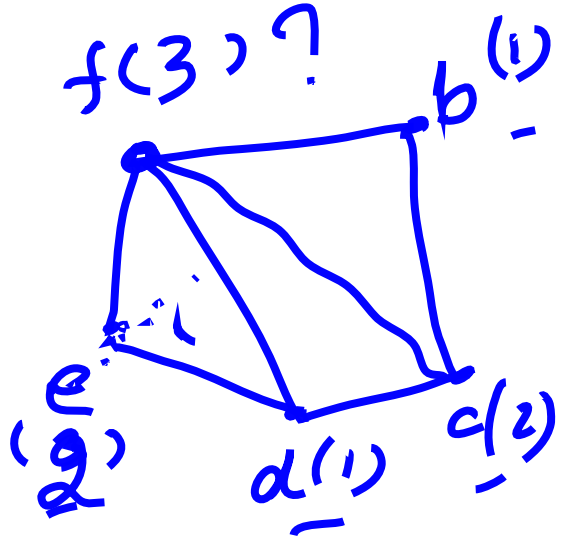


3-color.

Example

- Eventually we must assign a color to ***f***
- We hope that among the 4 neighbors of ***f*** we use less than 3 colors :: optimistic coloring





Spilling

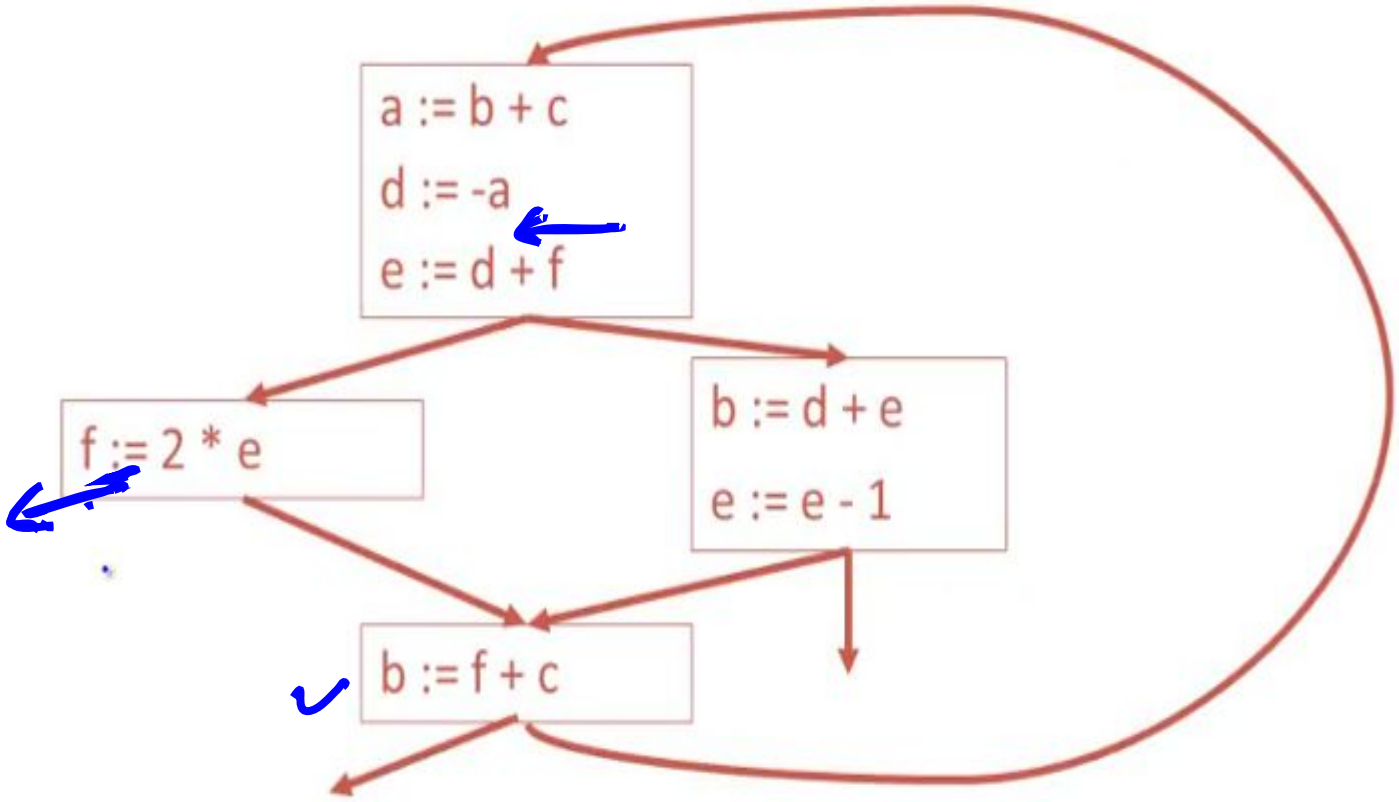
- If optimistic coloring fails, we spill f
  - Allocate a memory location for f
  - Typically in the current stack frame
  - Call this address fa
- Before each operation that reads f, insert

  f := load fa
- After each operation that writes f, insert

  store f, fa

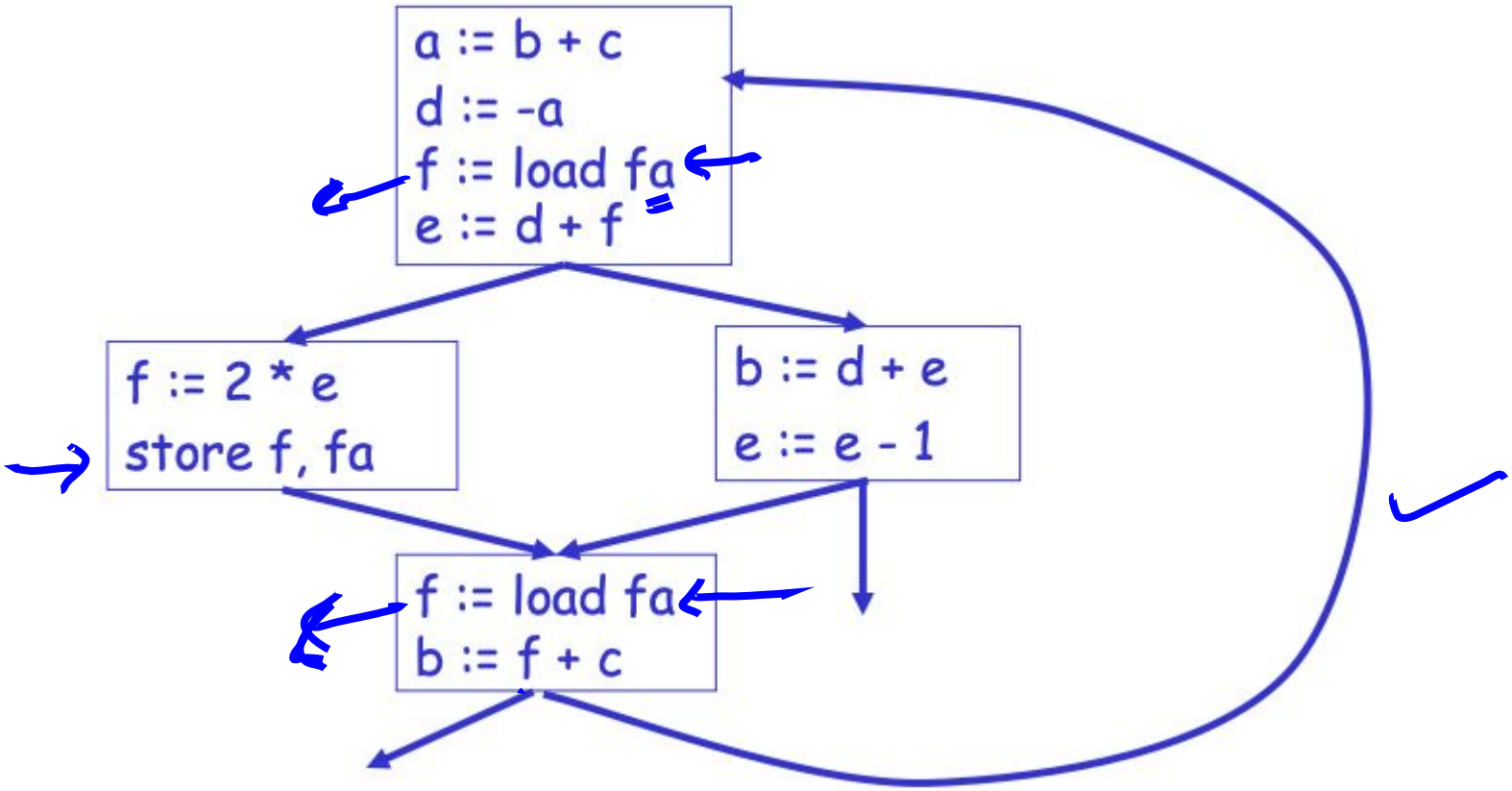
Spilling Example

- Original code



Spilling Example

- This is the new code after spilling f

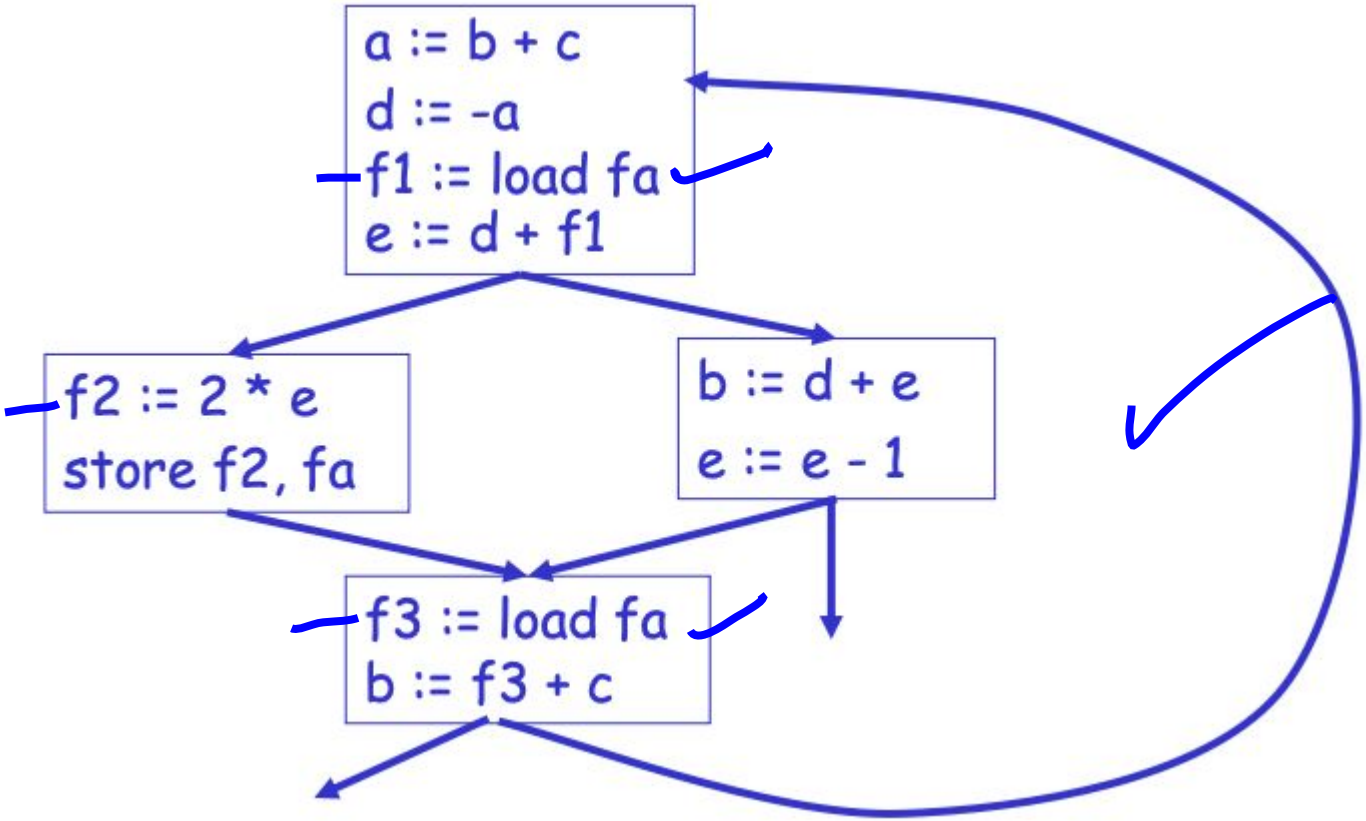


A Problem

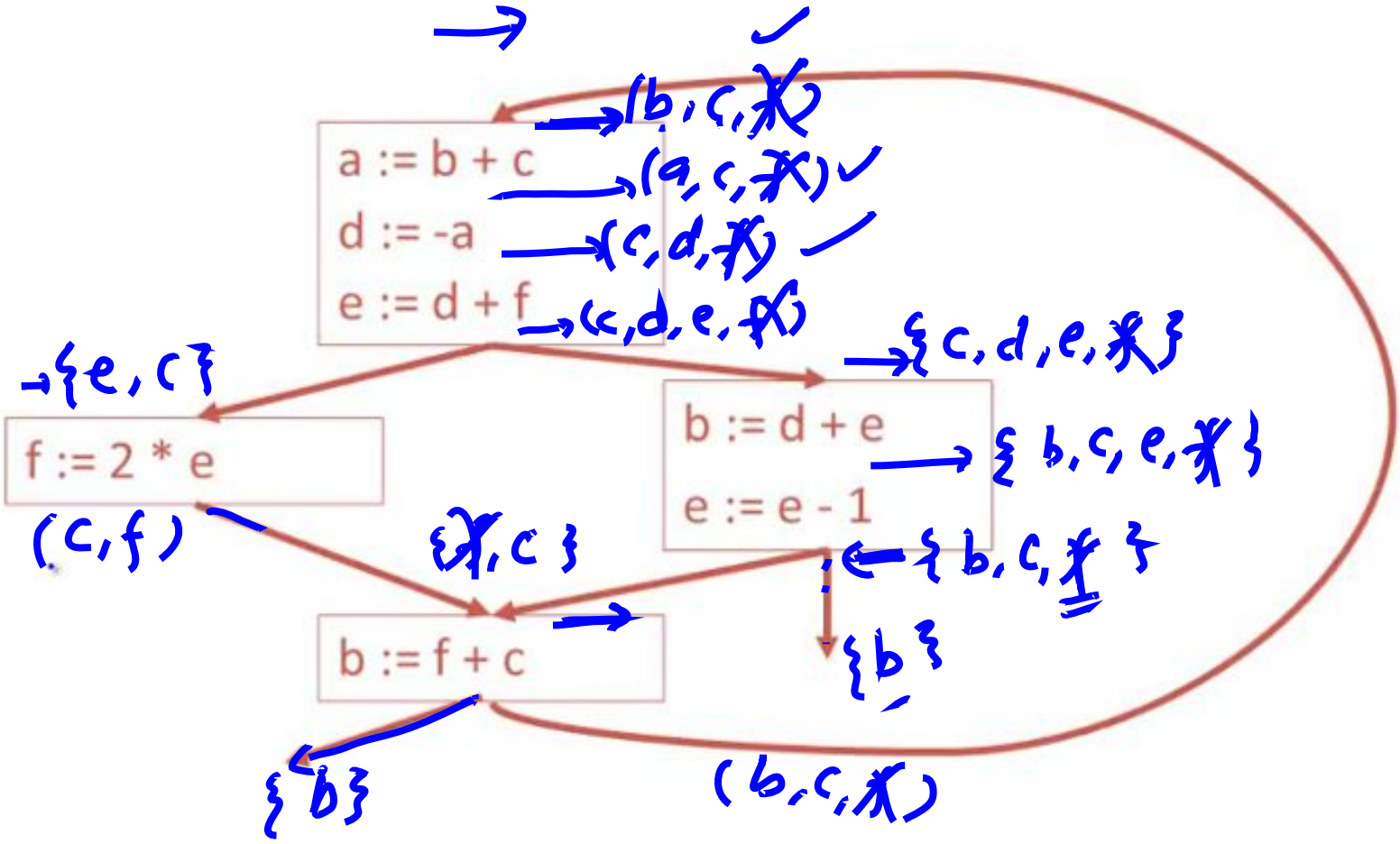
- This code reuses the register name f
- Correct, but suboptimal
  - Should use distinct register names whenever possible
  - Allows different uses to have different colors

Spilling Example

- This is the new code after spilling f

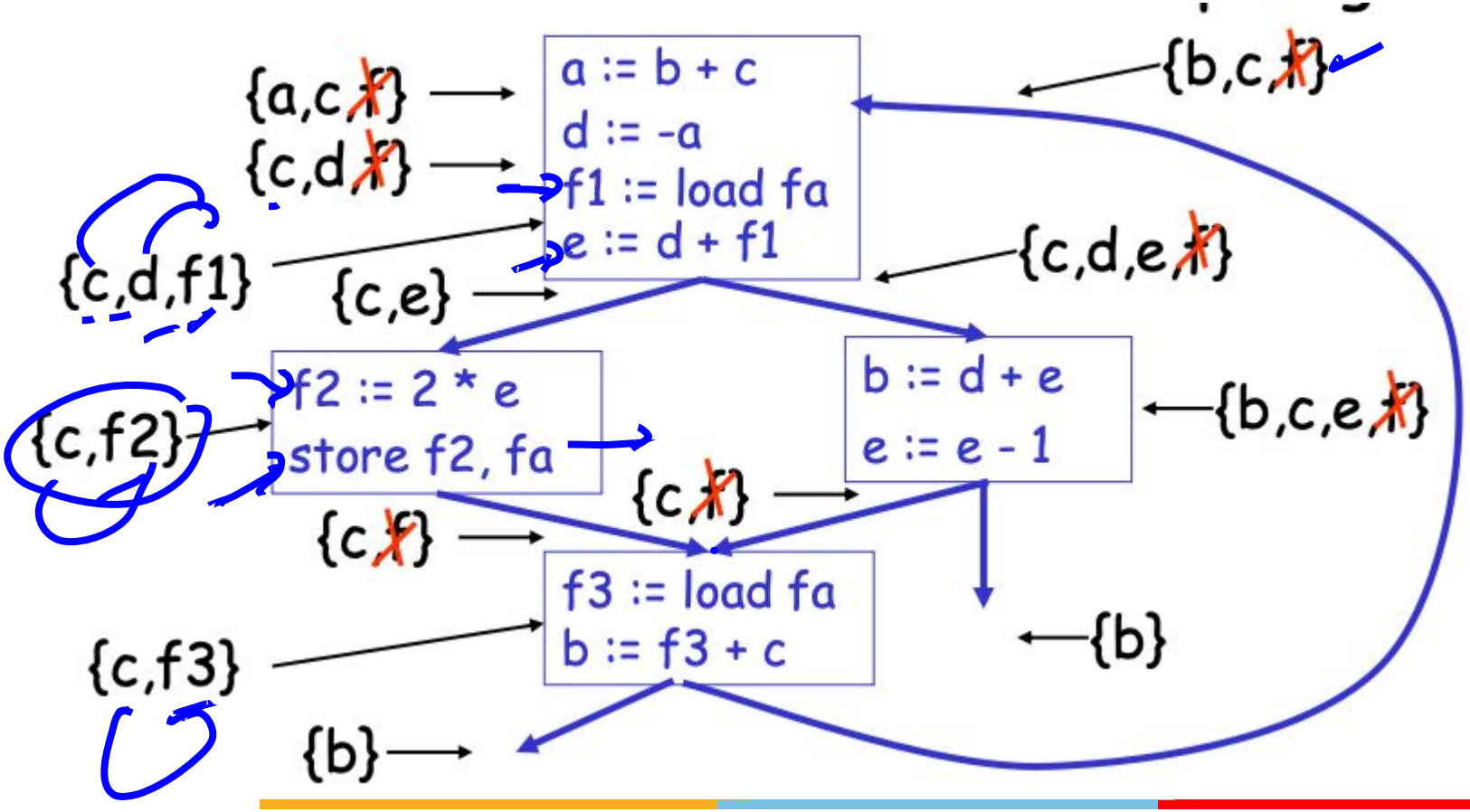


Liveness information in original code



Recomputing Liveness Information

- New liveness information after spilling f

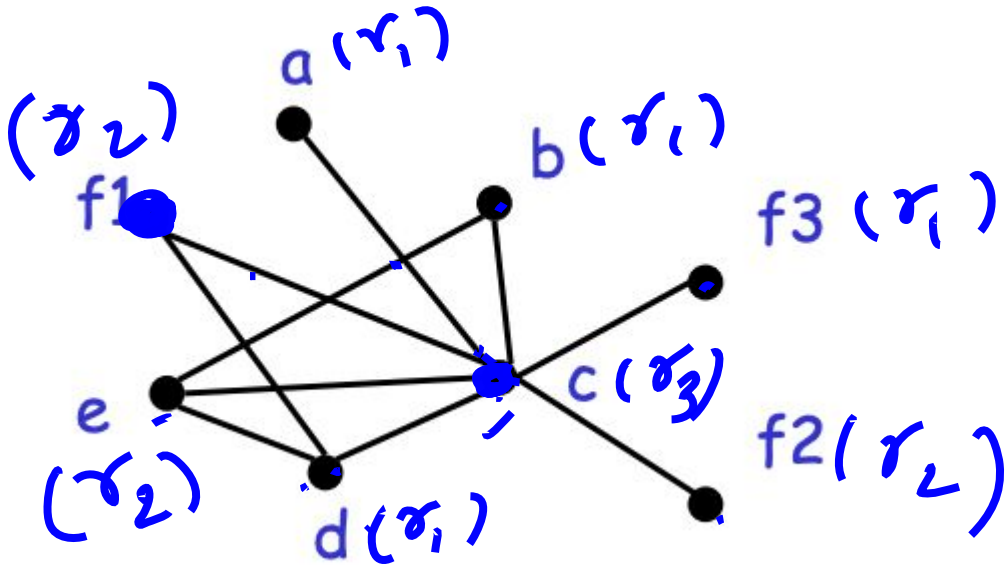


Recomputing Liveness Information

- New liveness information is almost as before
  - Note f has been split into three temporaries
- fi is live only
  - Between a fi := load fa and the next instruction
  - Between a store fi, fa and the preceding instr.
- Spilling reduces the live range of f
  - And thus reduces its interferences
  - Which results in fewer RIG neighbors

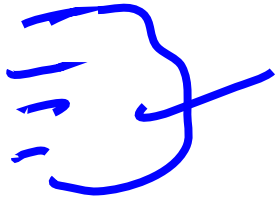
Recompute RIG After Spilling

- Some edges of the spilled node are removed
- In our case f still interferes only with c and d
- And the resulting RIG is 3-colorable



Spilling Notes

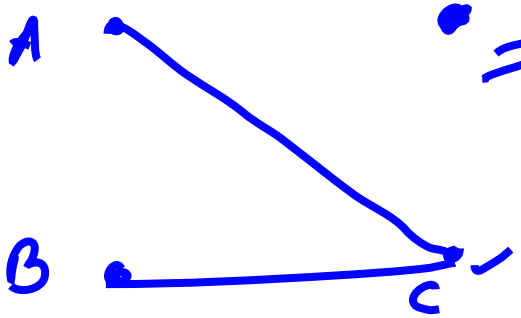
- Additional spills might be required before a coloring is found
- The tricky part is deciding what to spill
  - But any choice is correct
- Possible heuristics:
  - Spill temporaries with most conflicts
  - Spill temporaries with few definitions and uses
  - Avoid spilling in inner loops

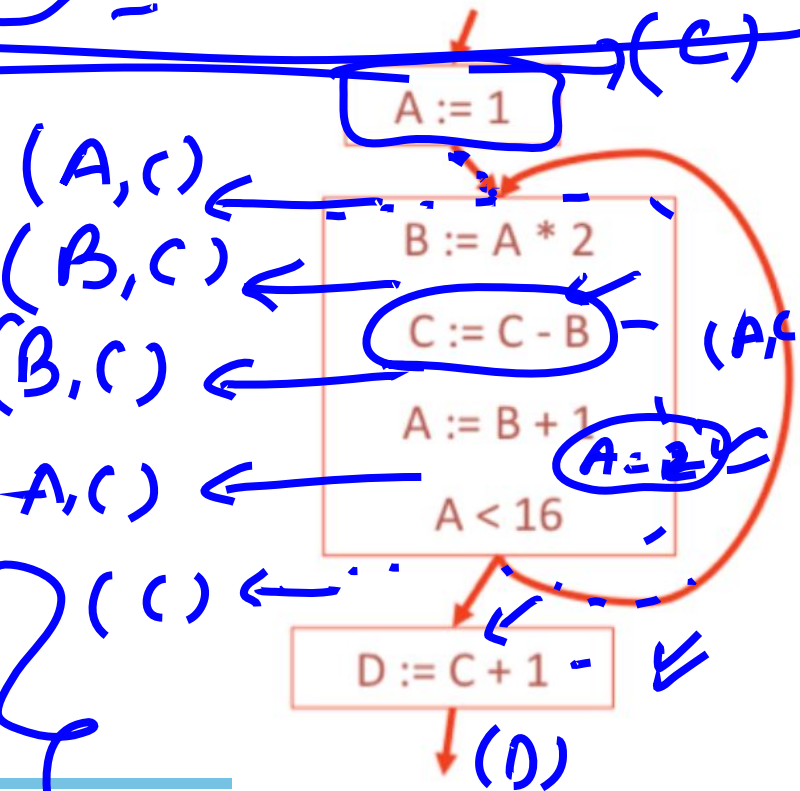


Calculate Spilling cost

For the given code fragment, draw RIG, and find the minimum cost spill. The cost of spilling a node is given by:

# of occurrences (use or definition) / # of conflicts + 5 if the node corresponds to a variable used in a loop

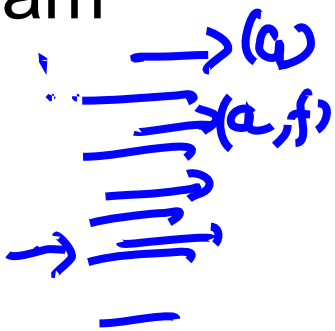




Spill cost (A) = 4 - 1 + 5 = 8
(B) = 3 - 1 + 5 = 7
(C) = 2 - 2 + 5 = 5

Live Ranges

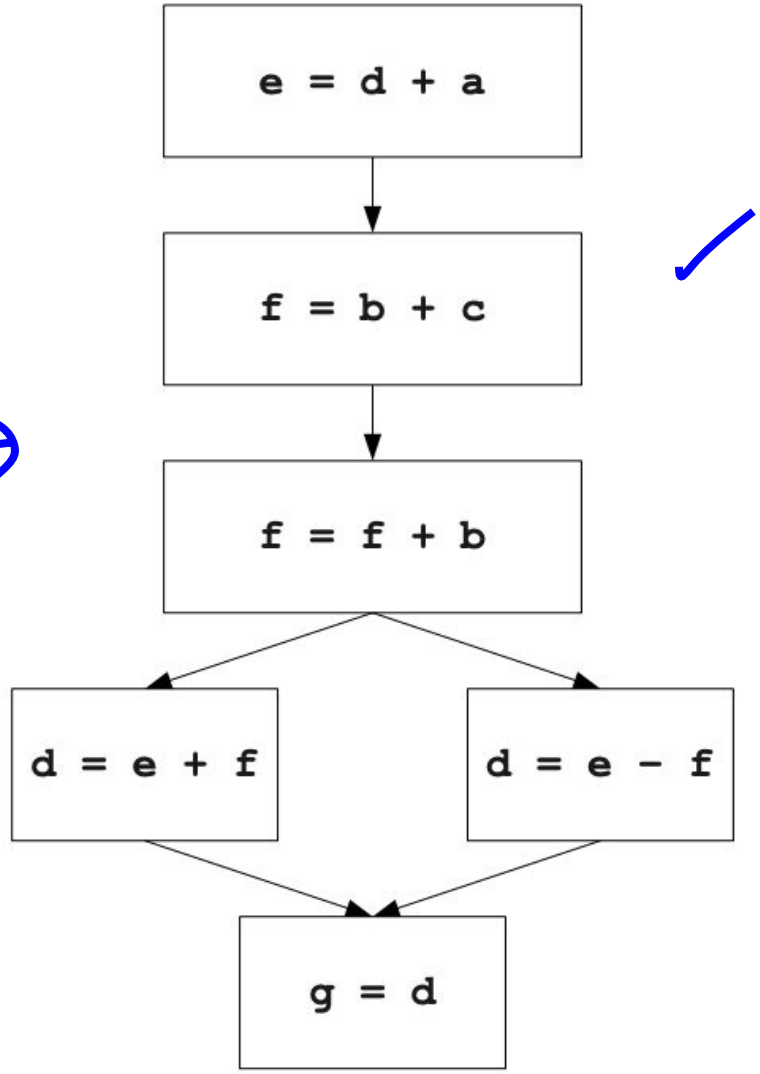
- Recall: A variable is live at a particular program point if its value may be read later before it is written.
  - Can find this using global liveness analysis.



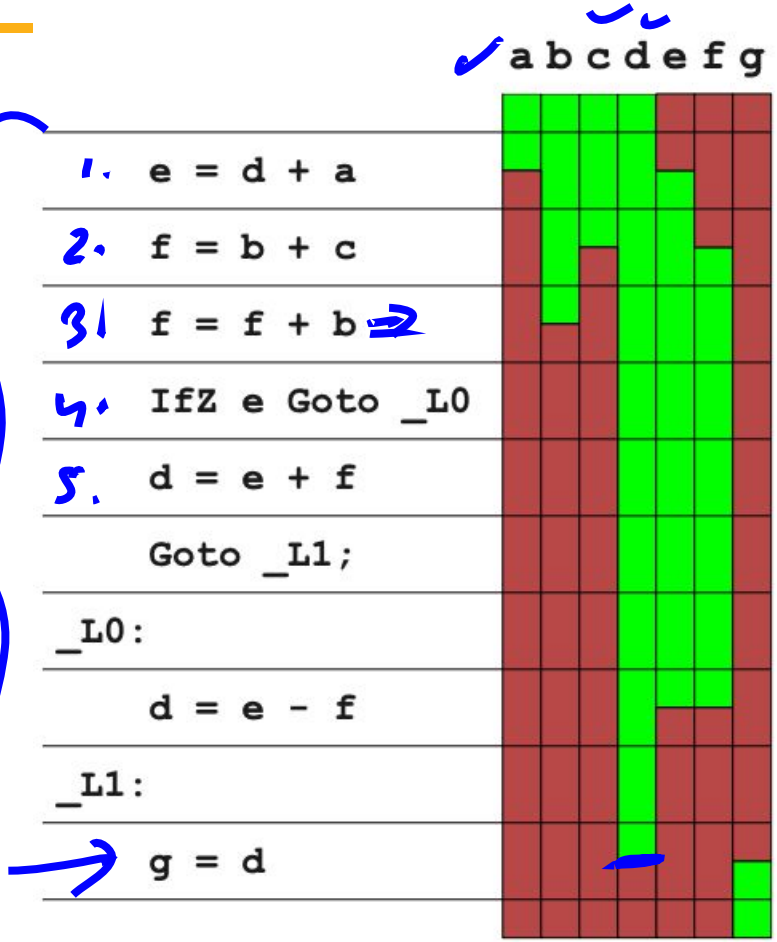
- The live range for a variable is the set of program points at which that variable is live.

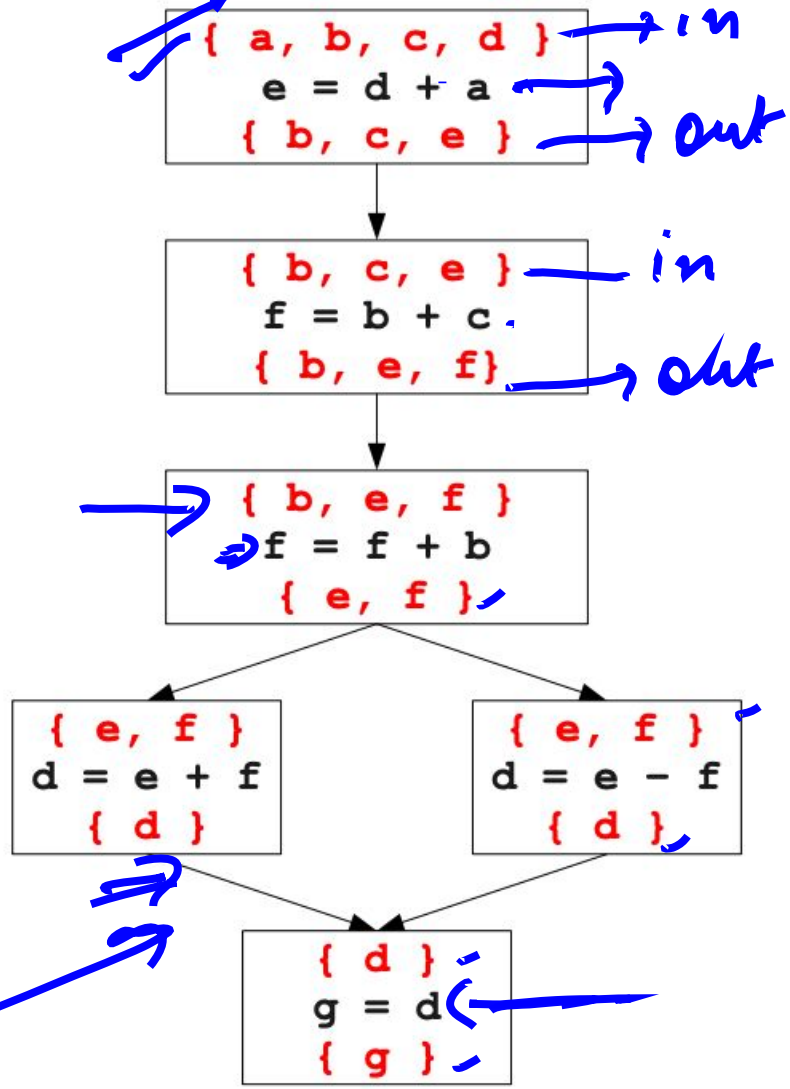
Live Ranges

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



Live Ranges


a b c d e f g
1. e = d + a
2. f = b + c
3. f = f + b
4. IfZ e Goto _L0
5. d = e + f
Goto _L1;
_L0:
d = e - f
_L1:
g = d


{ a, b, c, d }
e = d + a
{ b, c, e }
{ b, c, e }
f = b + c
{ b, e, f }
{ b, e, f }
f = f + b
{ e, f }
{ e, f }
d = e + f
{ d }
{ e, f }
d = e - f
{ d }
{ d }
g = d
{ g }

Thank You!