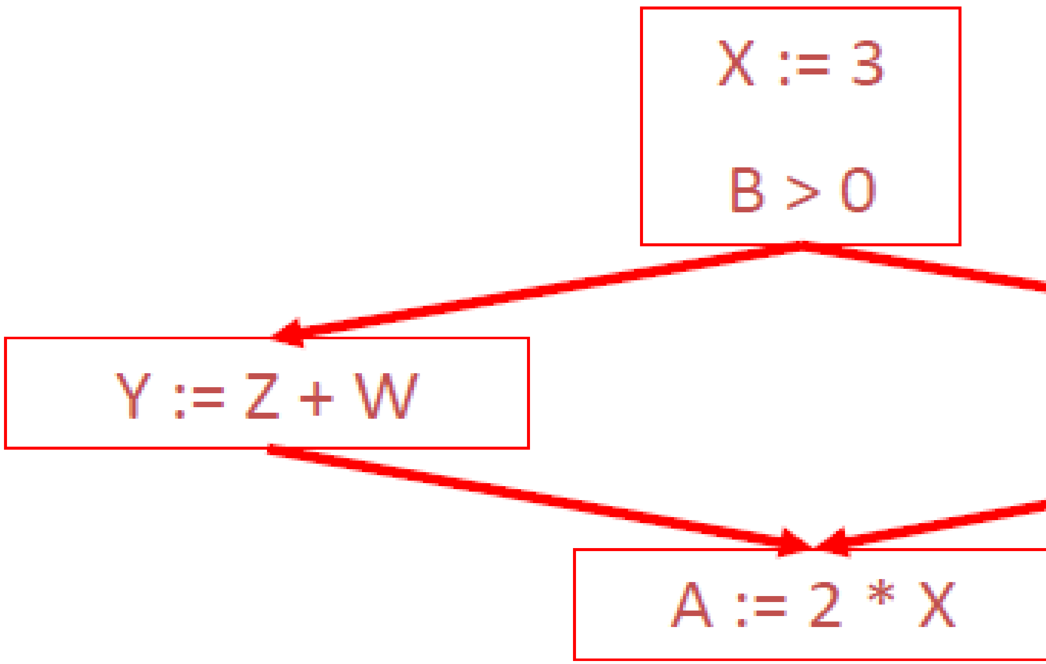
Global Optimization
Register Allocation

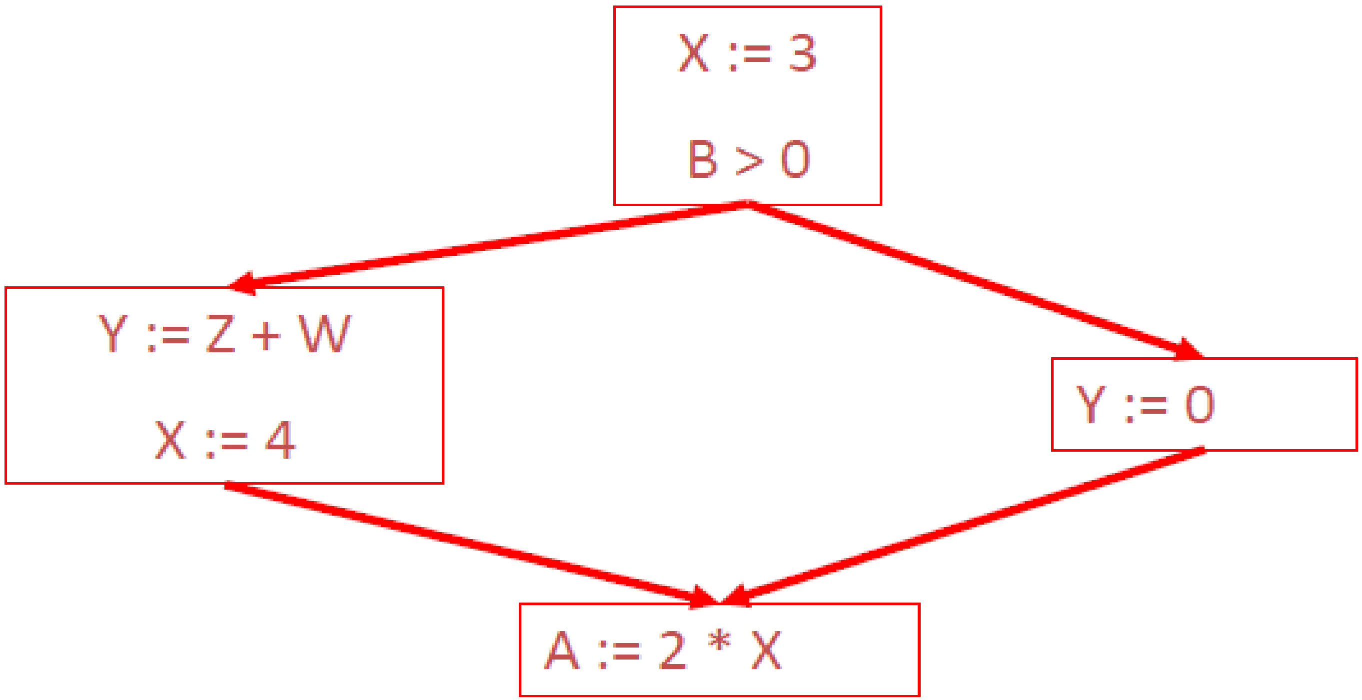
Dataflow Analysis

- Simple optimizations over a basic block may be extended over the entire CFG
- Example: Global Constant Propagation



Dataflow Analysis

- To replace a use of **x** by a constant **k** we must know:
  - On every path to the use of **x**, the last assignment to **x** is **x = k**



Global Optimization

- The correctness condition is not trivial to check
  - *Paths* include paths around loops and through branches of conditionals
- Generally global optimization depends on knowing a property **Γ** at a particular point in program execution
  - Proving **Γ** at any point requires knowledge of the entire program

- It is OK to be conservative. If the optimization requires **Γ** to be true, then want to know either
  - **Γ** is definitely true
  - Don’t know if **Γ** is true
    - It is always safe to say “don’t know”

Global Constant Propagation

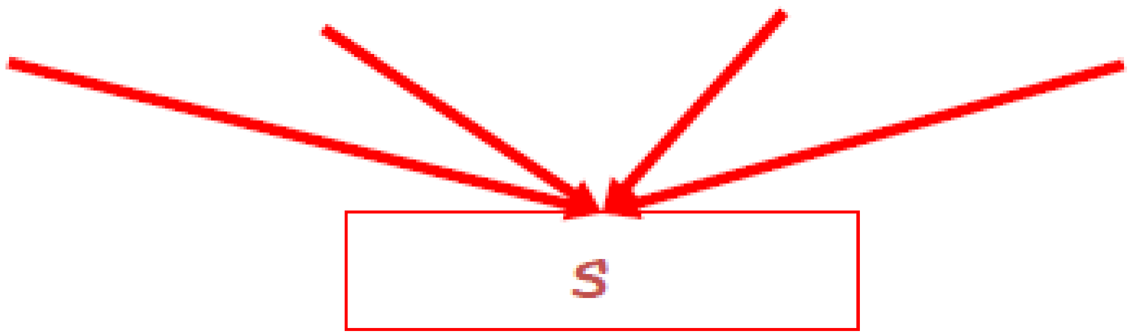
- We must know whether:
  - On every path to the use of **x**, the last assignment to **x** is **x = k**

- We associate one of the following values with **x** at every program point:
  - $\perp$ (Bottom) : This statement never executes
  - C : **x** equals to constant **C**
  - $\top$ (Top) : **x** is not a constant

- For each statement **s**, the value of **x** immediately before and after **s** is calculated
  - C(s, x, in): Value of **x** before **s**
  - C(s, x, out): Value of **x** after **s**

Global Constant Propagation

- Rule #1

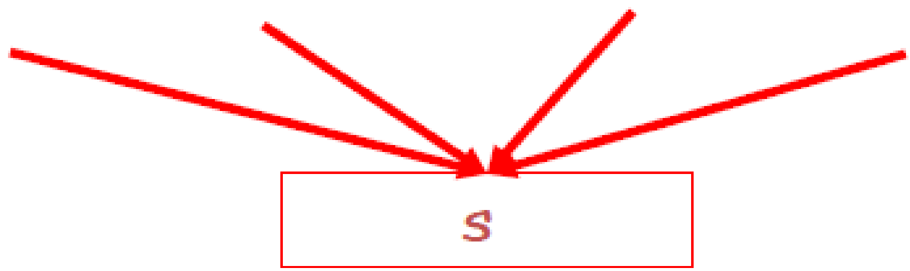


if $C(p_i, x, out) = \top$

for any i, then $C(s, x, in) = \top$

Global Constant Propagation

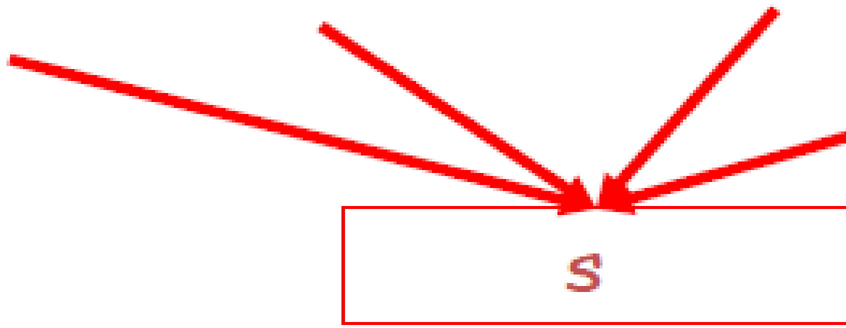
- Rule #2



$$\text{if } C(p_i, x, out) = c \;\&\; C(p_j, x, out) = d \;\&\; d <> c$$
$$\text{then } C(s, x, in) = \top$$

Global Constant Propagation

- Rule #3

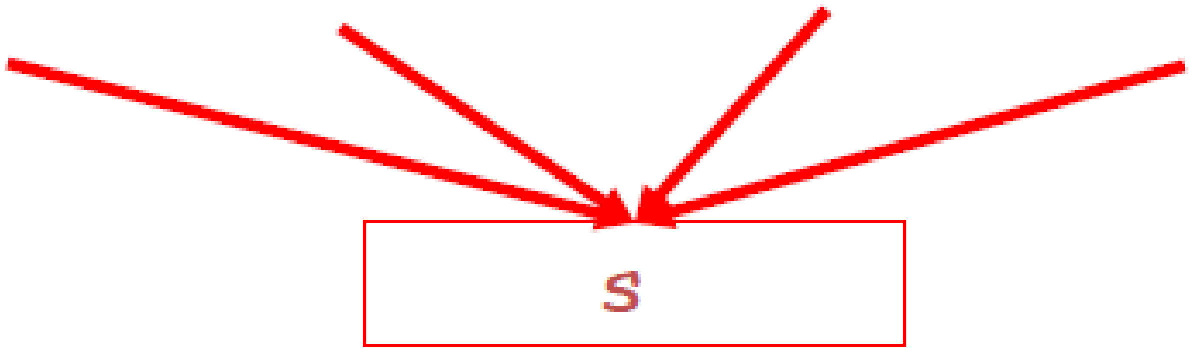


$$\text{if } C(p_i, x, out) = c \text{ or } \perp \text{ for all } i,$$

$$\text{then } C(s, x, in) = c$$

Global Constant Propagation

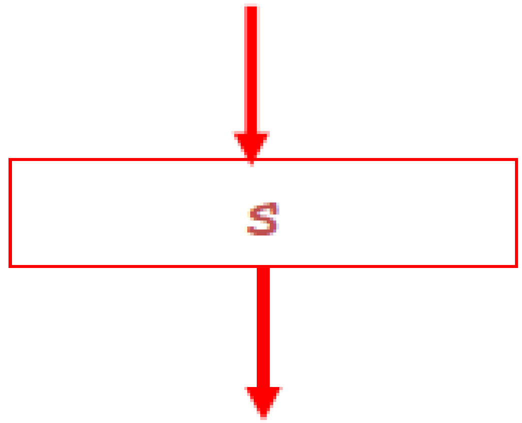
- Rule #4



$$\text{if } C(p_i, x, out) = \bot \text{ for all } i,$$
$$\text{then } C(s, x, in) = \bot$$

Global Constant Propagation

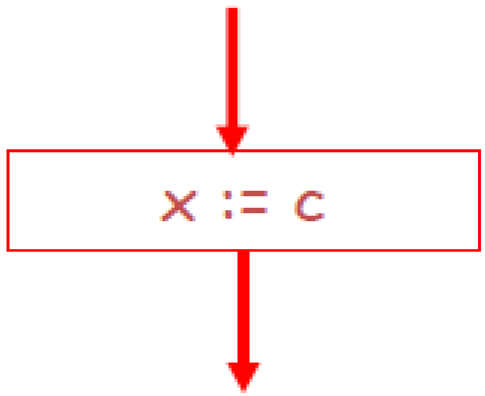
- Rule #5



$$C(s, x, out) = \perp \text{ if } C(s, x, in) = \perp$$

Global Constant Propagation

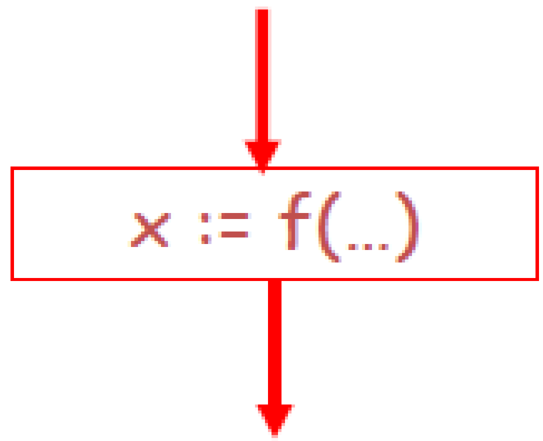
- Rule #6



C(x := c, x, out) = c if c is a constant

Global Constant Propagation

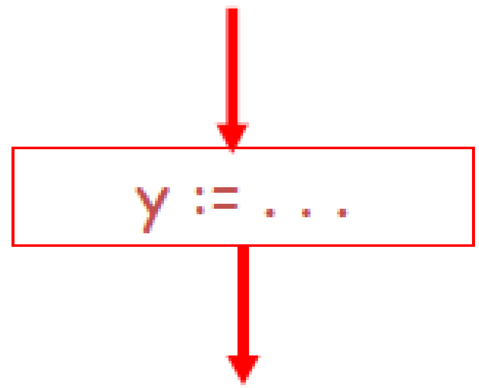
- Rule #7



$$C(x := f(\ldots), x, out) = \top$$

Global Constant Propagation

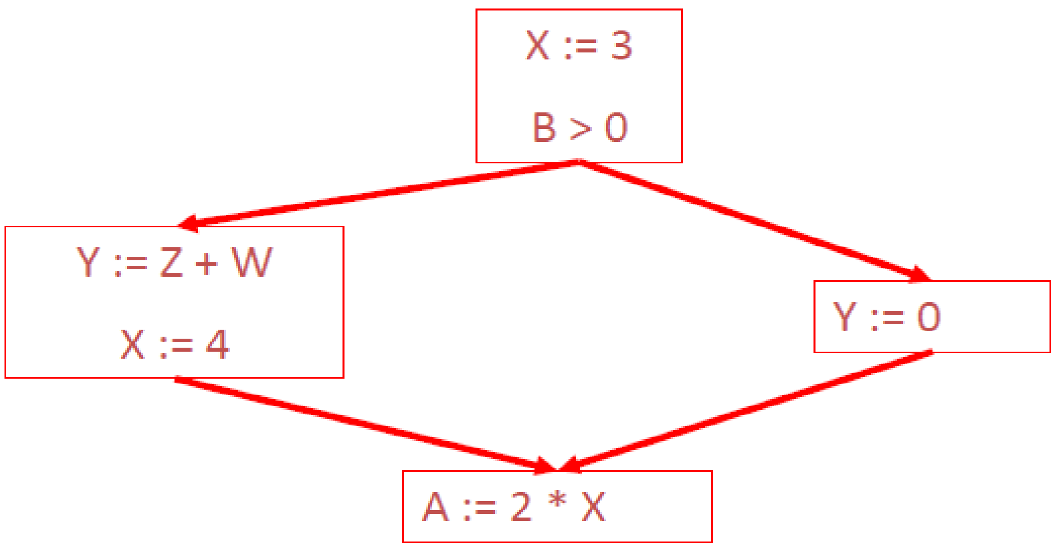
- Rule #8



$$C(y := \ldots, x, out) = C(y := \ldots, x, in) \text{ if } x <> y$$

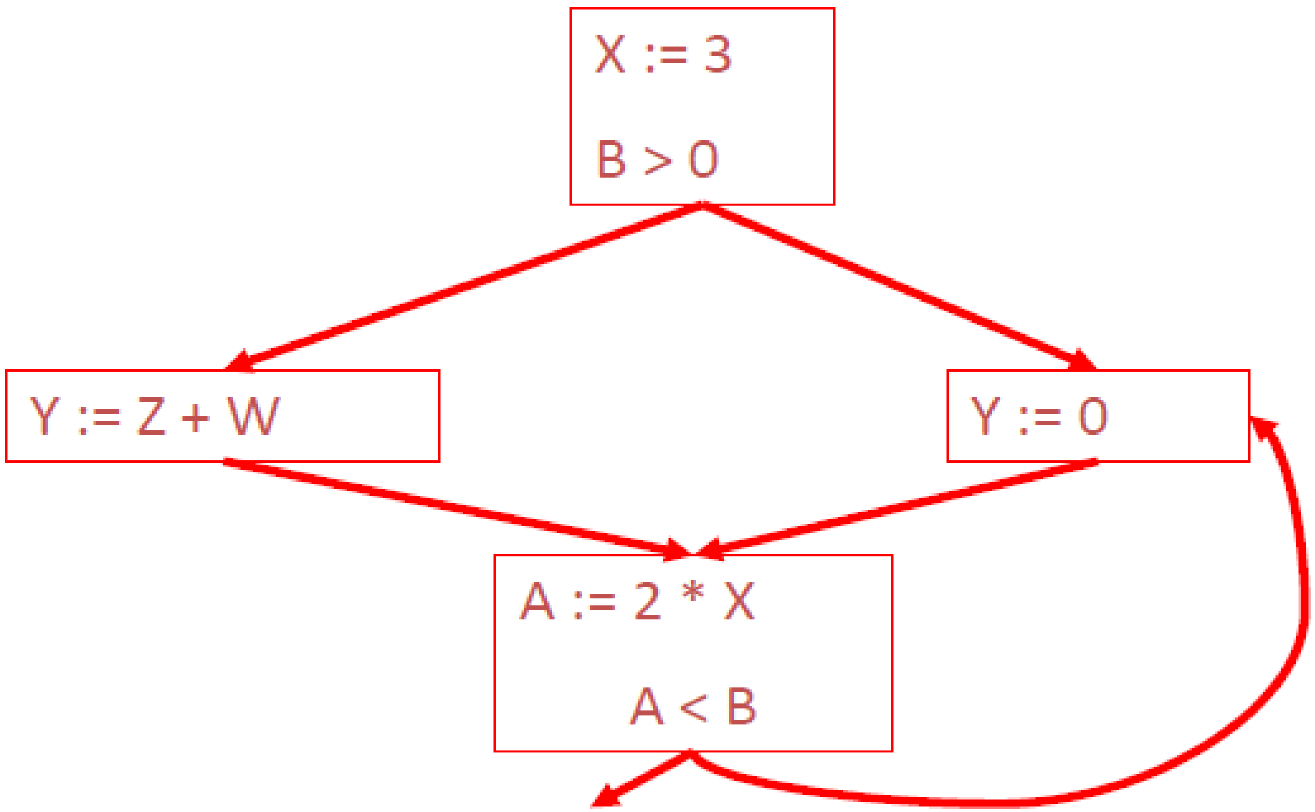
Global Constant Propagation

- For every entry s to the program, set C(s, x, in) = $\top$
- Set C(s, x, in) = C(s, x, out) = $\perp$ everywhere else
- Repeat until all points satisfy 1-8:
  - Pick s not satisfying 1-8 and update using the appropriate rule

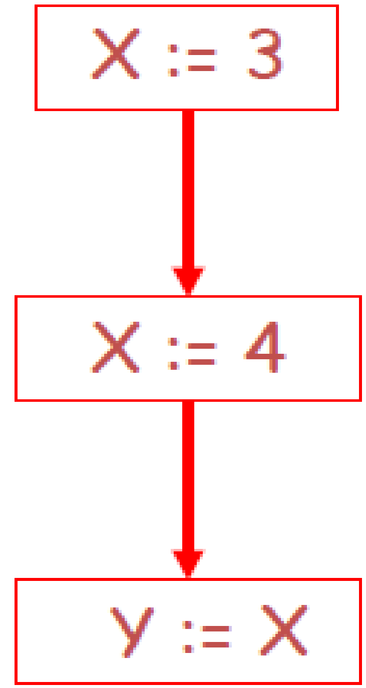


Analysis of Loops

- How can global constant propagation for the following CFG be performed?



Liveness Analysis



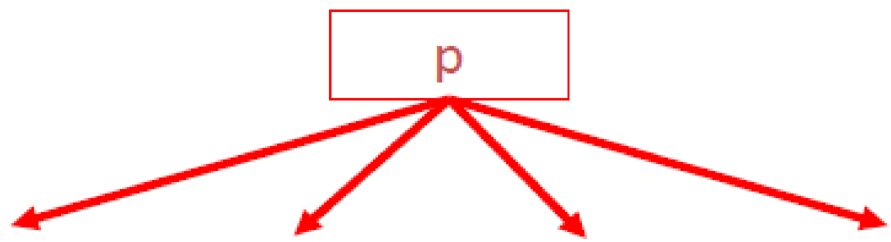
- The first value of x is *dead (never used)*
- The second value of x is *live (may be used)*

- A variable **x** is live at statement **s** if
    - There exists a statement **s'** that uses **x**
    - There is a path from s to **s'**
    - That path has no intervening assignment to **x**

- A statement **x = …** is dead code if **x** is dead after the assignment
    - Dead statements can be deleted from the program

Liveness Analysis

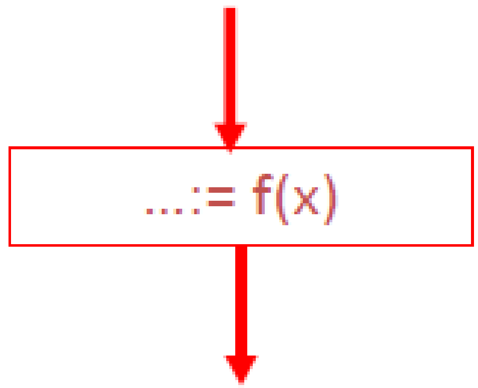
- Liveness can be expressed in terms of information transferred between adjacent statements, just as in copy propagation
- Rule #1



$$L(p, x, out) = \vee \{ L(s, x, in) \mid s \text{ a successor of } p \}$$

Liveness Analysis

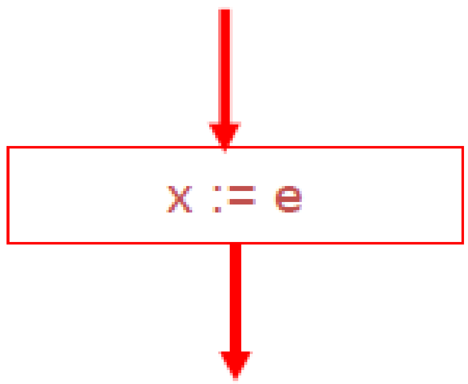
- Rule #2



L(s, x, in) = true if s refers to x on the rhs

Liveness Analysis

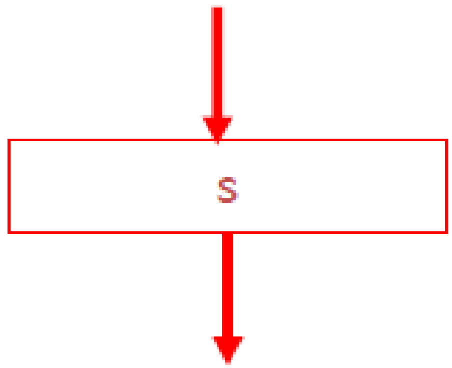
- Rule #3



L(x := e, x, in) = false if e does not refer to x

Liveness Analysis

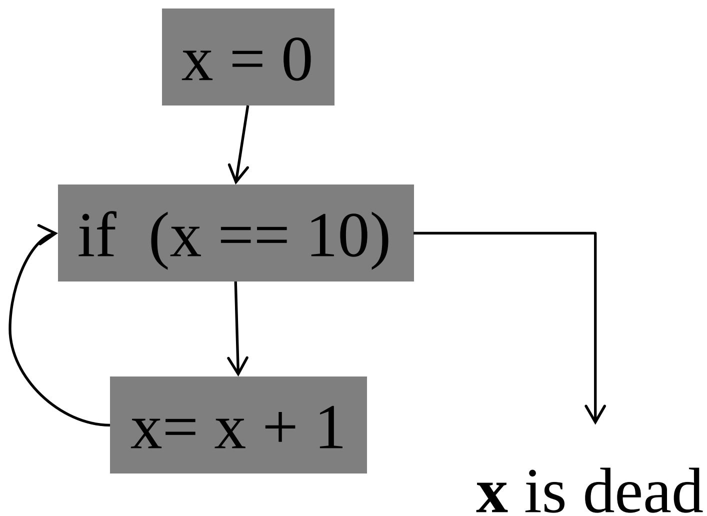
- Rule #4



$L(s, x, in) = L(s, x, out)$ if s does not refer to x

Liveness Analysis

- Let all **L(…) = false** initially
- Repeat until all statements **s** satisfy rules 1-4
  - Pick **s** where one of 1-4 does not hold and update using the appropriate rule



Register Allocation

- The process of assigning a large number of target program variables onto a small number of CPU registers

- **Method:** Assign multiple temporaries to each register without changing the program behavior
  - Example:

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

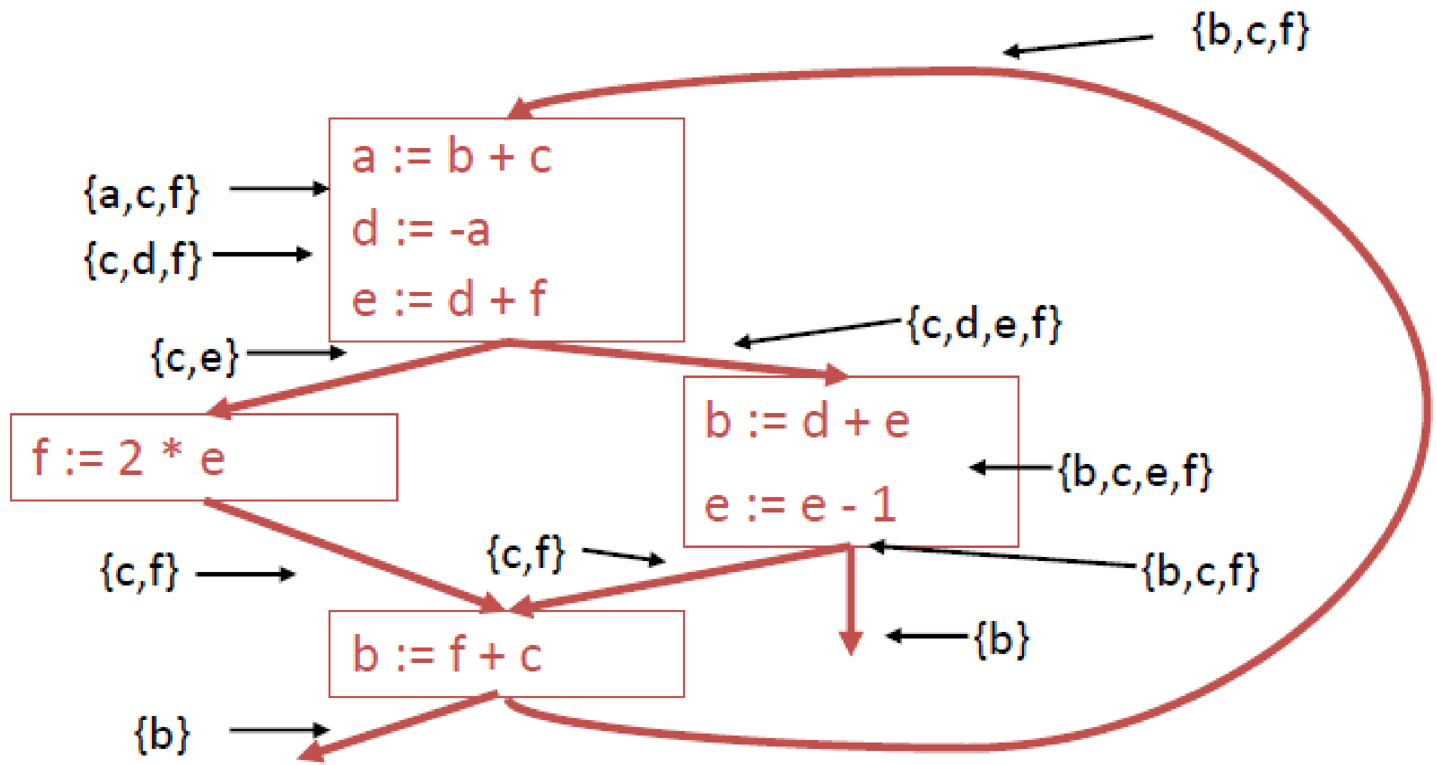
r_1 := r_2 + r_3
r_1 := r_1 + r_4
r_1 := r_1 - 1

a, **e** and **f** can all be allocated to the same register assuming **a** and **e** are dead after use

Register Allocation via Graph Colouring

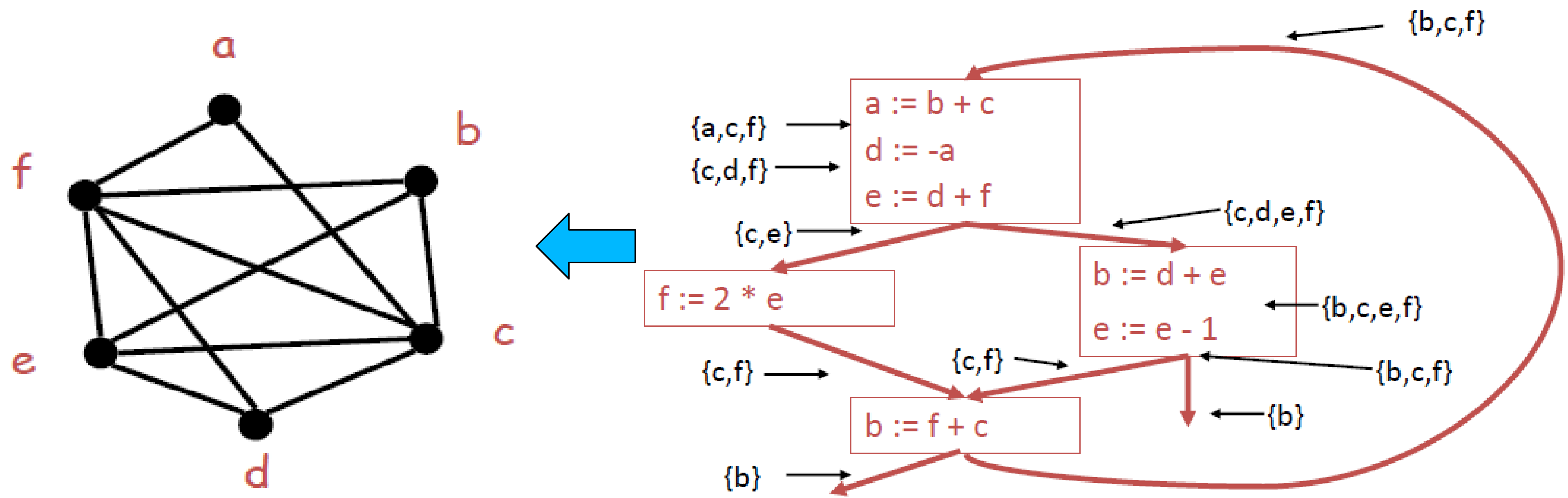
- **Basic Idea:** *Two temporary variables t_1 and t_2 can share the same register if at any point in the program at most one of t_1 or t_2 is live.*

The variables alive simultaneously at each program point



Register Allocation via Graph Colouring

- **Register Interference Graph (RIG)**
  - An **undirected** graph
  - Each temporary variable is a **node**
  - Two temporary variables t_1 and t_2 share an **edge** if they are simultaneously alive at some program point



Register Allocation via Graph Colouring

- **Graph Colouring:** An assignment of colours to nodes, such that nodes connected by an edge have different colours
- ***k*-colouring :** A coloring using at most *k* colours.
- **Chromatic number:** The smallest number of colours needed to colour a graph
- **Independent set:** A subset of vertices assigned to the same colour

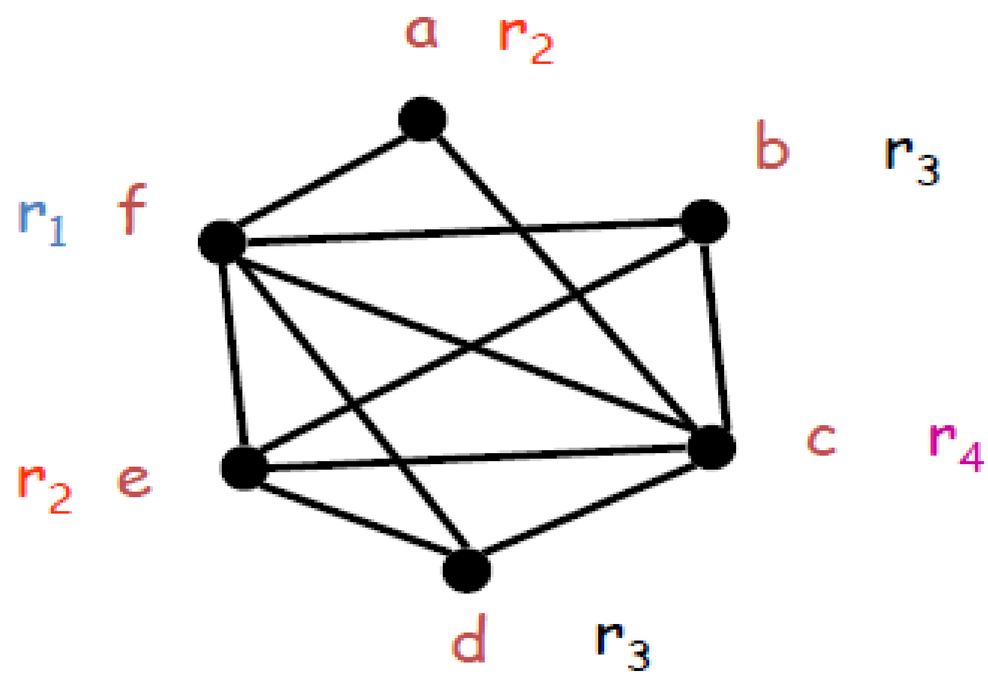
- *k*-coloring is the same as a partition of the vertex set into *k* independent sets
  - The terms *k-partite* and *k-colourable* are equivalent

- *The graph colouring problem is* ***NP-Hard***
  - Heuristics needed to solve it

Register Allocation via Graph Colouring

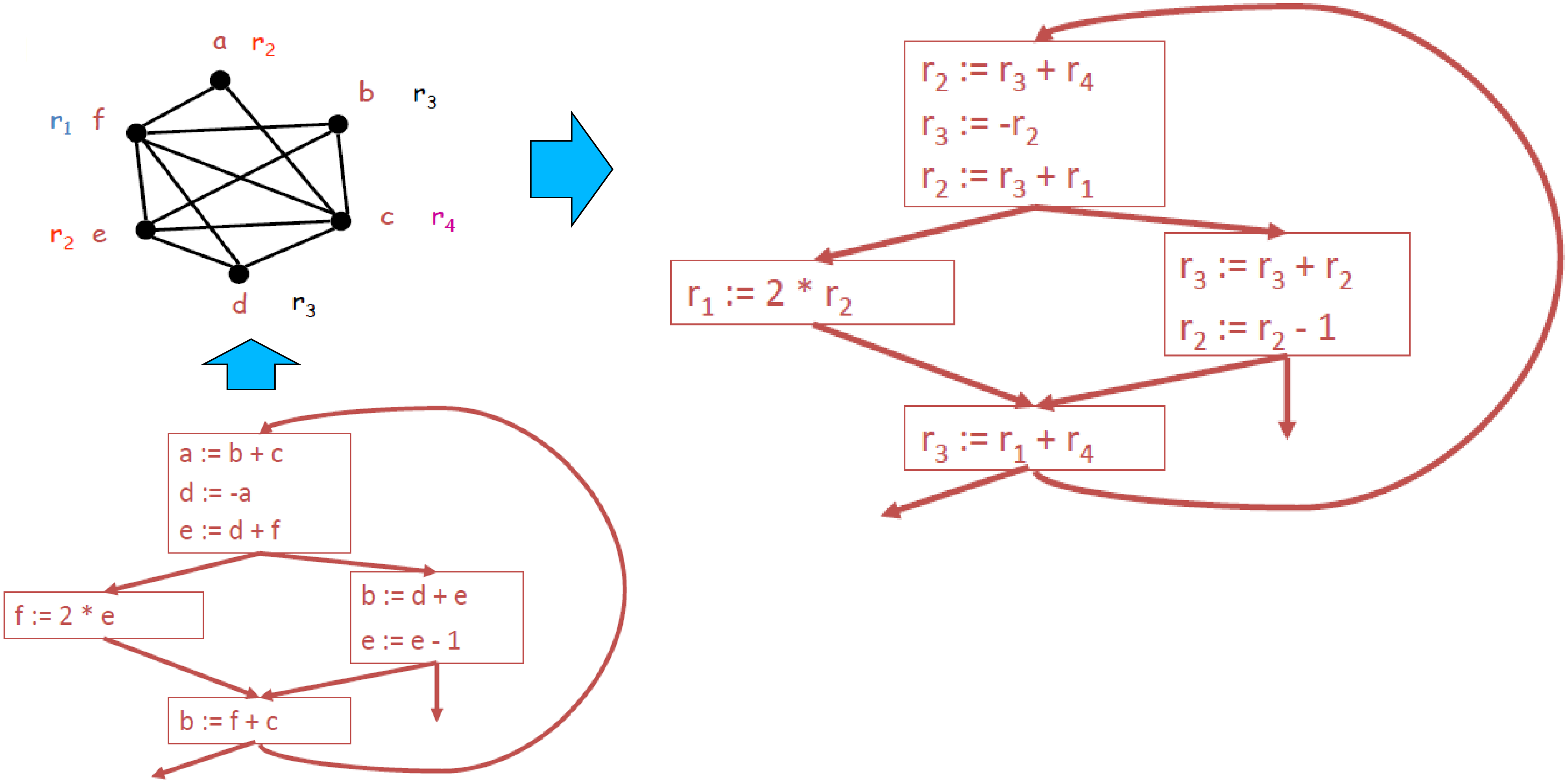
- In the register allocation problem, colours = registers
- We need to assign colours (registers) to graph nodes (temporaries)
- Let *k* = number of machine registers
- If the RIG is *k-colourable* then there is a register assignment that uses no more than *k* registers

In our example RIG there is no coloring with less than 4 colours



Register Allocation via Graph Colouring

- Under the colouring, the code becomes:



Register Allocation via Graph Colouring

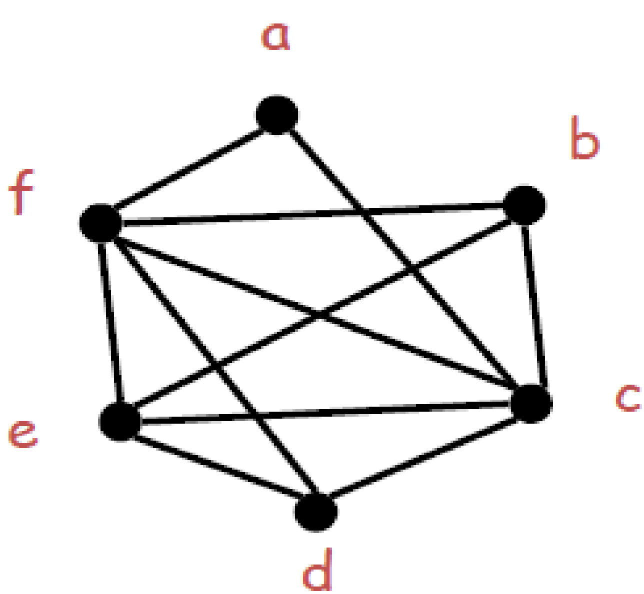
- A heuristic algorithm:
  - Pick a node *t* with fewer than *k* neighbors
  - Put *t* on a stack and remove it from the RIG
  - Repeat until the graph is empty

- Assign colors to nodes on the stack:
  - Start with the last node added
  - At each step pick a color different from those assigned to already colored neighbors

Register Allocation via Graph Colouring

- Example: Let $k = 4$
- Initial RIG:

  Stack: {}

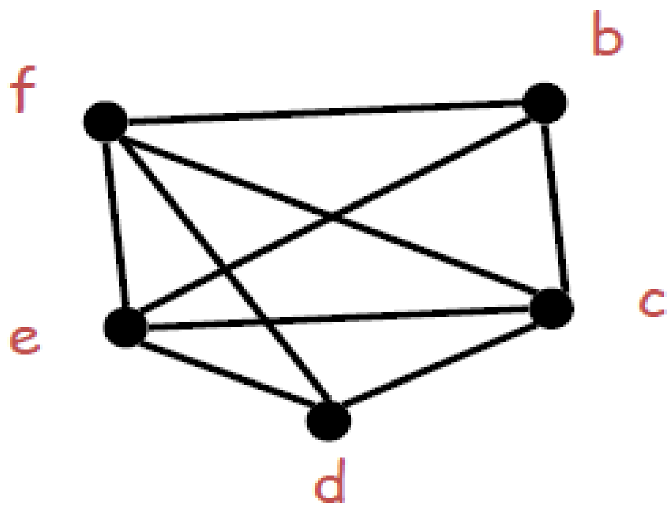


Remove *a*

Register Allocation via Graph Colouring

- Step 2:

  Stack: {a}

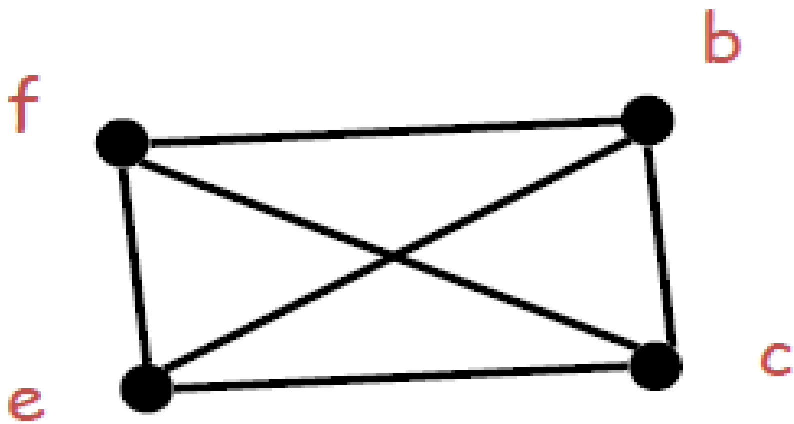


Remove *d*

Register Allocation via Graph Colouring

- All nodes now have fewer than 4 nodes
- Step 3:

  Stack: {*d*, *a*}

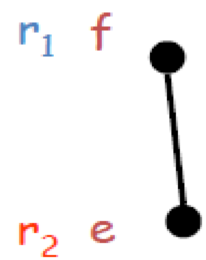


- Remove *any node*
- Continue removing nodes until the graph is *empty*
- Let the stack be: {*f*, *e*, *b*, *c*, *d*, *a*} after removal of all nodes

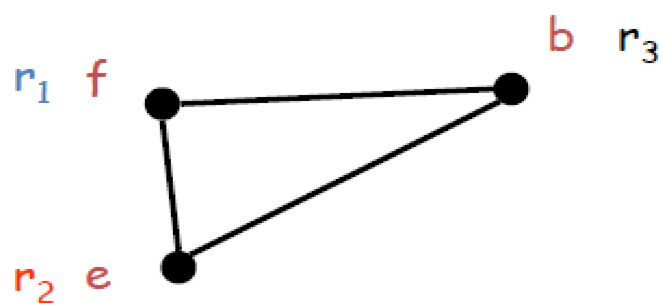
Register Allocation via Graph Colouring

- Now start assigning colours to the nodes, starting from the top of the stack
- Stack: {*f*, *e*, *b*, *c*, *d*, *a*}
- Stack: {*e*, *b*, *c*, *d*, *a*}

- Stack: {*b*, *c*, *d*, *a*}

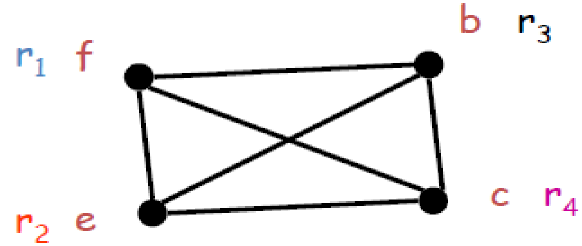


- Stack: {*c*, *d*, *a*}

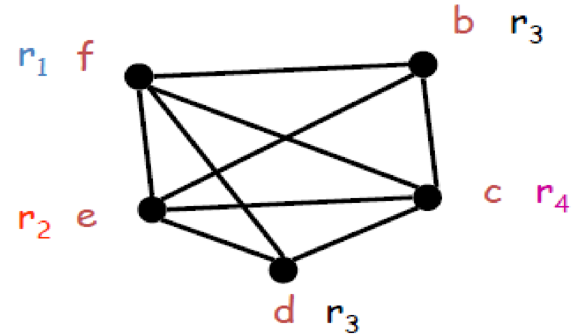


Register Allocation via Graph Colouring

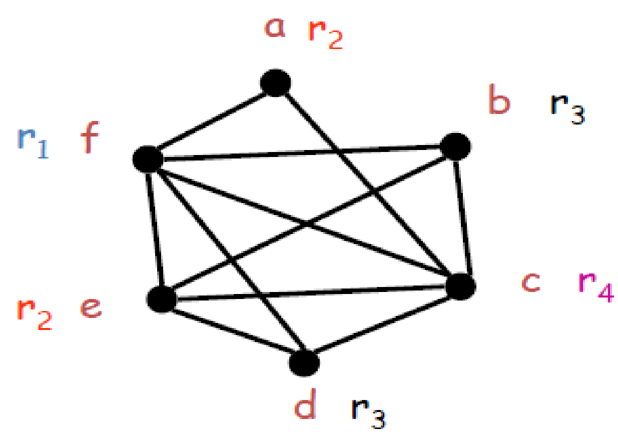
- Stack: {*d*, *a*}



- Stack: {*a*}(*d* and *b* can have the same register)

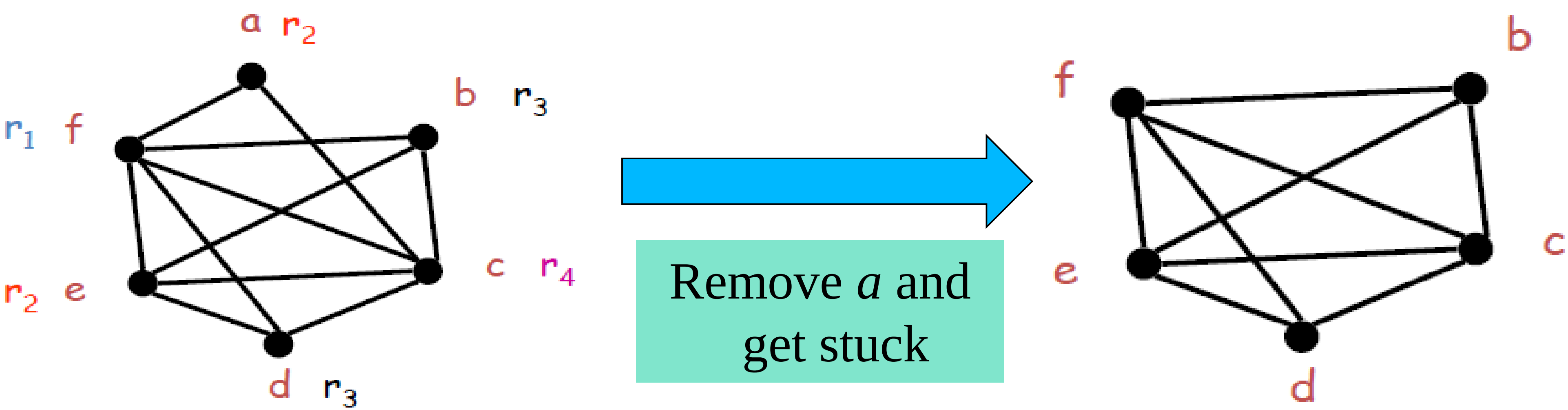


- Stack: {} }(*a* and *e* can have the same register)



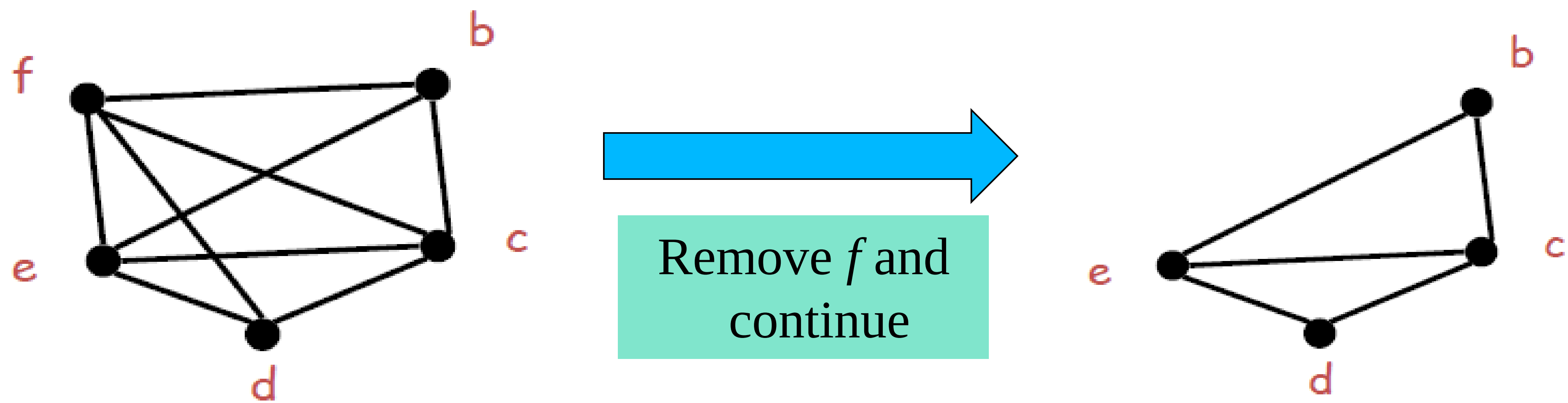
What if the heuristic fails?

- Example: Try to do a 3-colouring of the graph:



What if the heuristic fails?

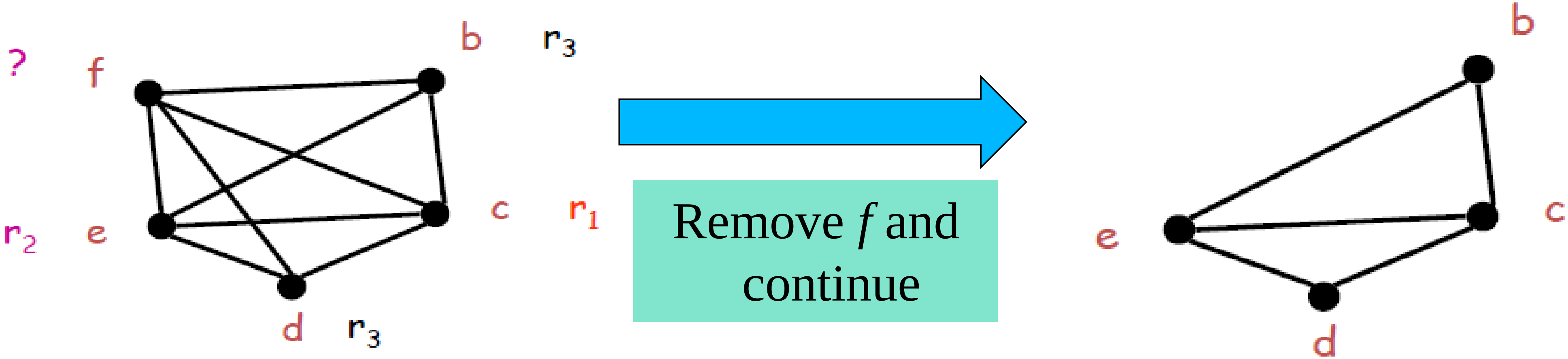
- Pick a node as a candidate for *spilling*
    - *A spilled temporary lives in memory*
- Assume we choose *f* as a candidate for spilling



- The algorithm now succeeds: *b*, *d*, *e*, *c*

What if the heuristic fails?

- On the assignment phase we get to the point when we have to assign a color to *f*
- We hope that among the 4 neighbors of *f* we use less than 3 colors ⇒ <u>optimistic coloring</u>



- The algorithm now succeeds: *b*, *d*, *e*, *c*

Spilling

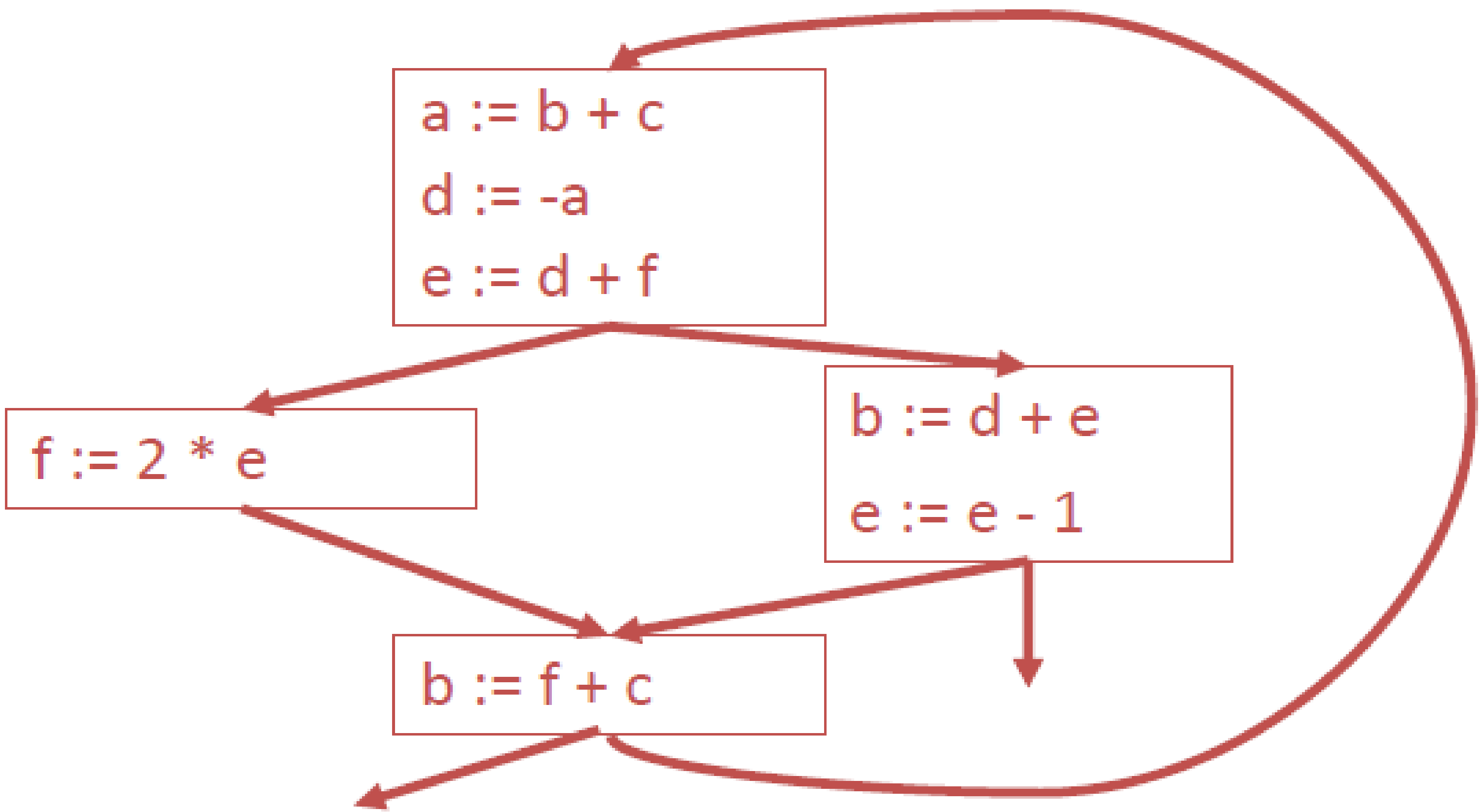
- Since optimistic coloring failed we must spill temporary *f*
- We must allocate a memory location as the home of *f*
  - Typically this is in the current stack frame
  - Call this address *fa*

- Before each operation that uses *f*, insert
  - *f* := load *fa*

- After each operation that defines *f*, insert
  - store *f*, *fa*

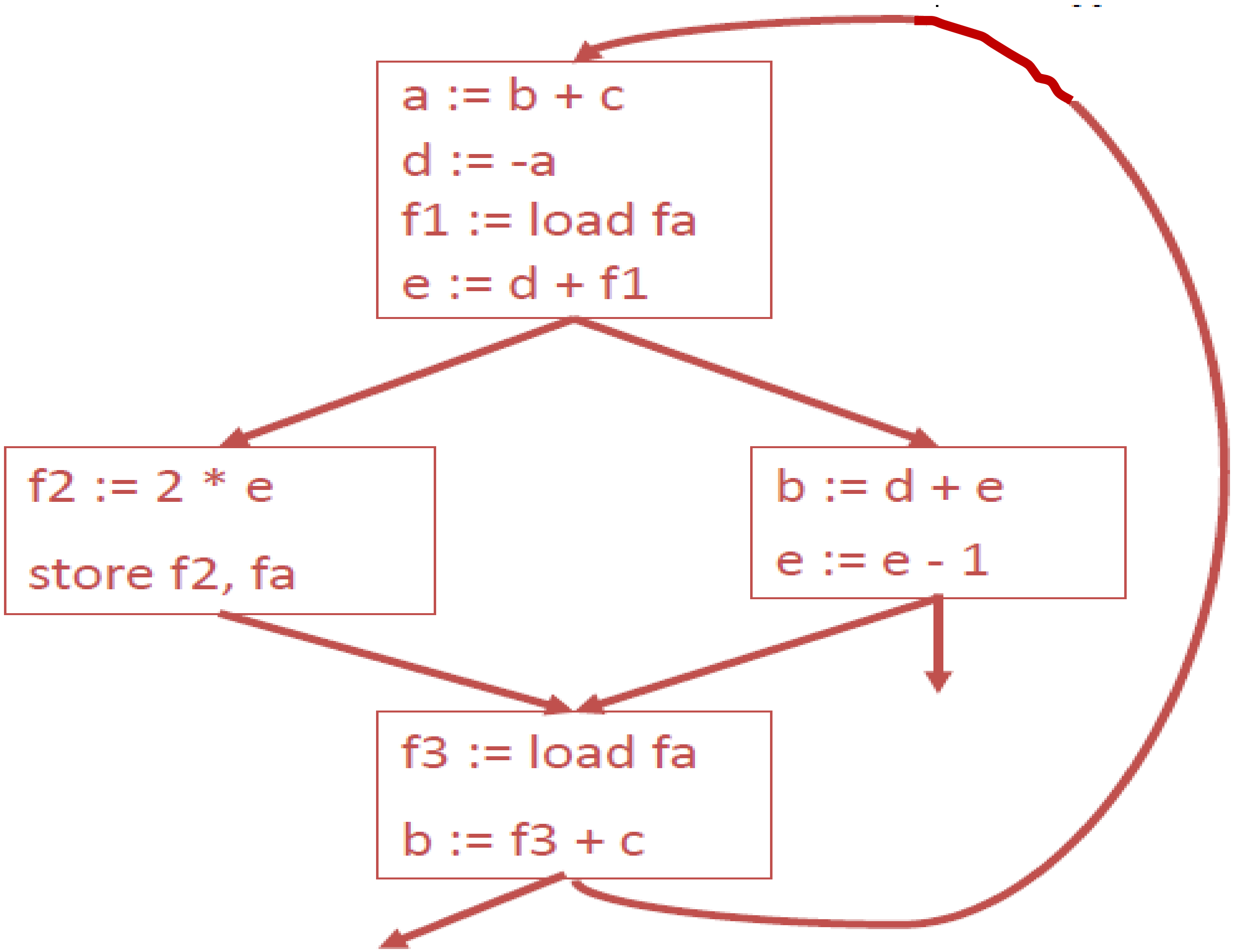
Spilling

- Original code:



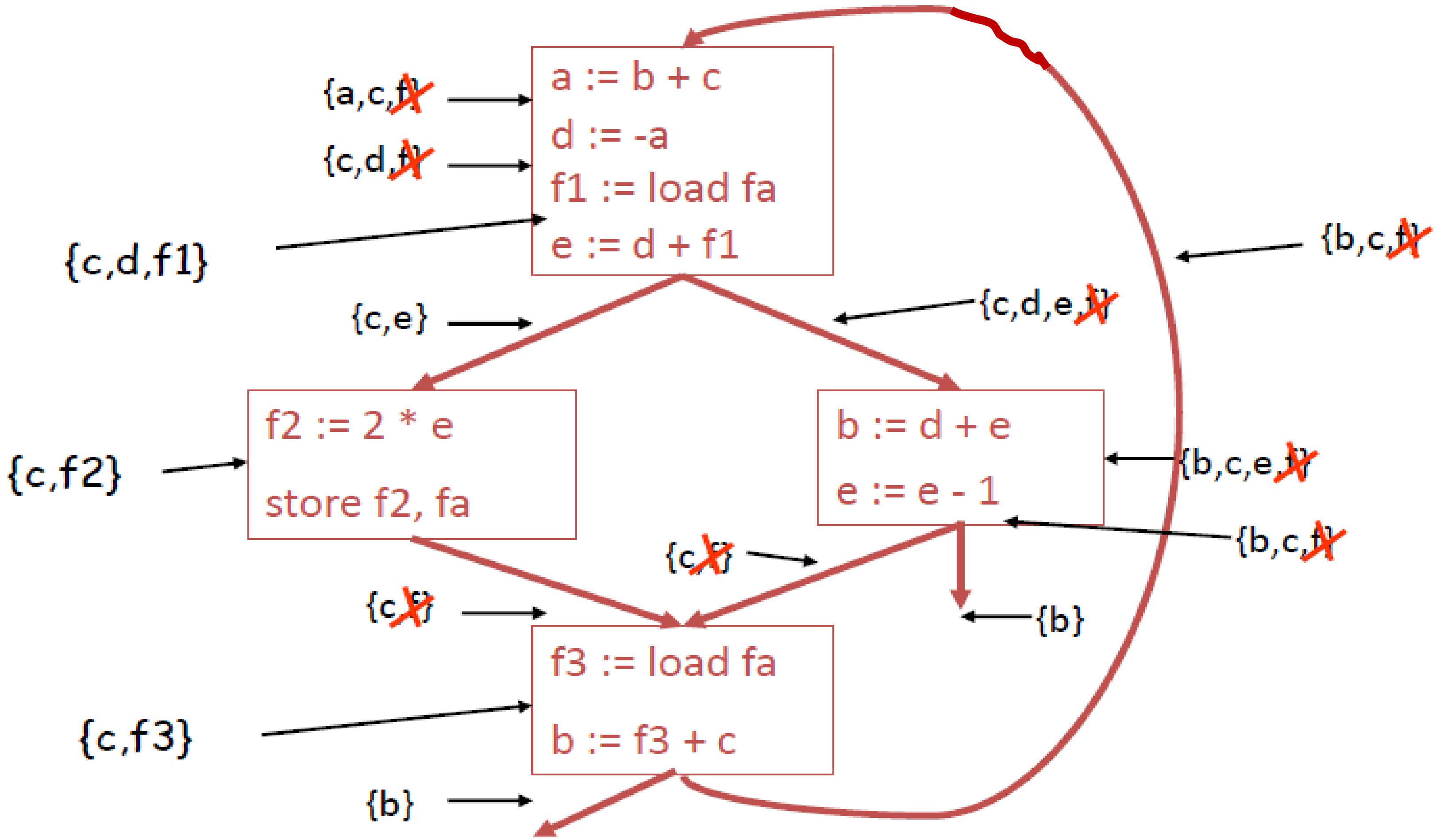
Spilling

- Code after spilling:



Spilling

- Re-compute Liveness:



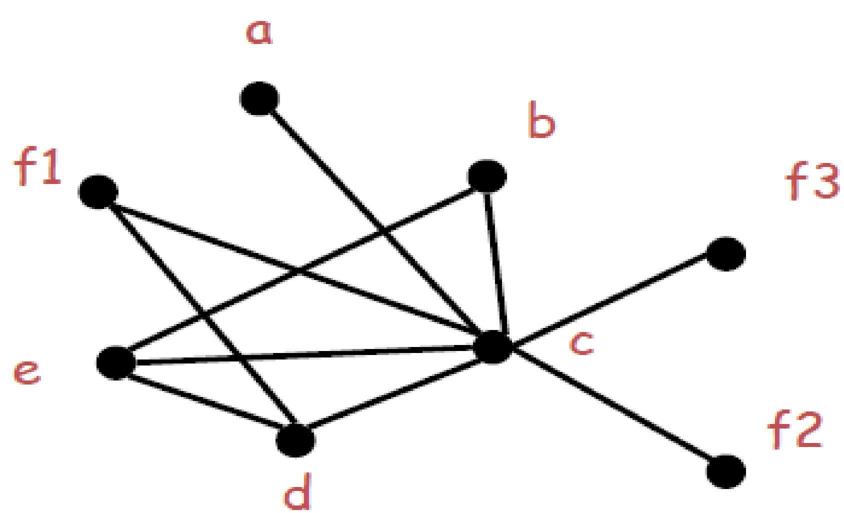
Spilling

- The new liveness information is almost as before
- f_i is live only
  - Between a f_i:= load *fa* and the next instruction
  - Between a store f_i, *fa* and the preceding instruction

- Spilling reduces the live range of *f* and thus reduces its interferences
  - Which result in fewer neighbors in RIG for *f*

Spilling

- With the new liveness information, we need to rebuild the RIG
    - And try to colour the resulting graph again



- Now *f* only interfaces with *c* and *d*
- The new RIG is 3-colourable

Spilling

- 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