

# Register Allocation

CMPT 379: Compilers

Instructor: Anoop Sarkar

[anoopsarkar.github.io/compilers-class](https://anoopsarkar.github.io/compilers-class)

# Register Allocation

- Intermediate code uses unlimited temporaries
  - Simplifying code generation and optimization
  - Complicates final translation to assembly

# Register Allocation

- The problem:

Rewrite the intermediate code to use no more temporaries than there are machine registers

- Method:

- Assign multiple temporaries to each register
- But without changing the program behavior

# Example

- Consider the program

$a = c + d$

$e = a + b$

$f = e - 1$

- Assume  $a$  &  $e$  dead after use
  - A dead temporary can be “reused”

- Can allocate  $a$ ,  $e$  and  $f$  all to one register ( $r1$ )

$r1 = r_2 + r_3$

$r_1 = r_1 + r_4$

$r_1 = r_1 - 1$

# History

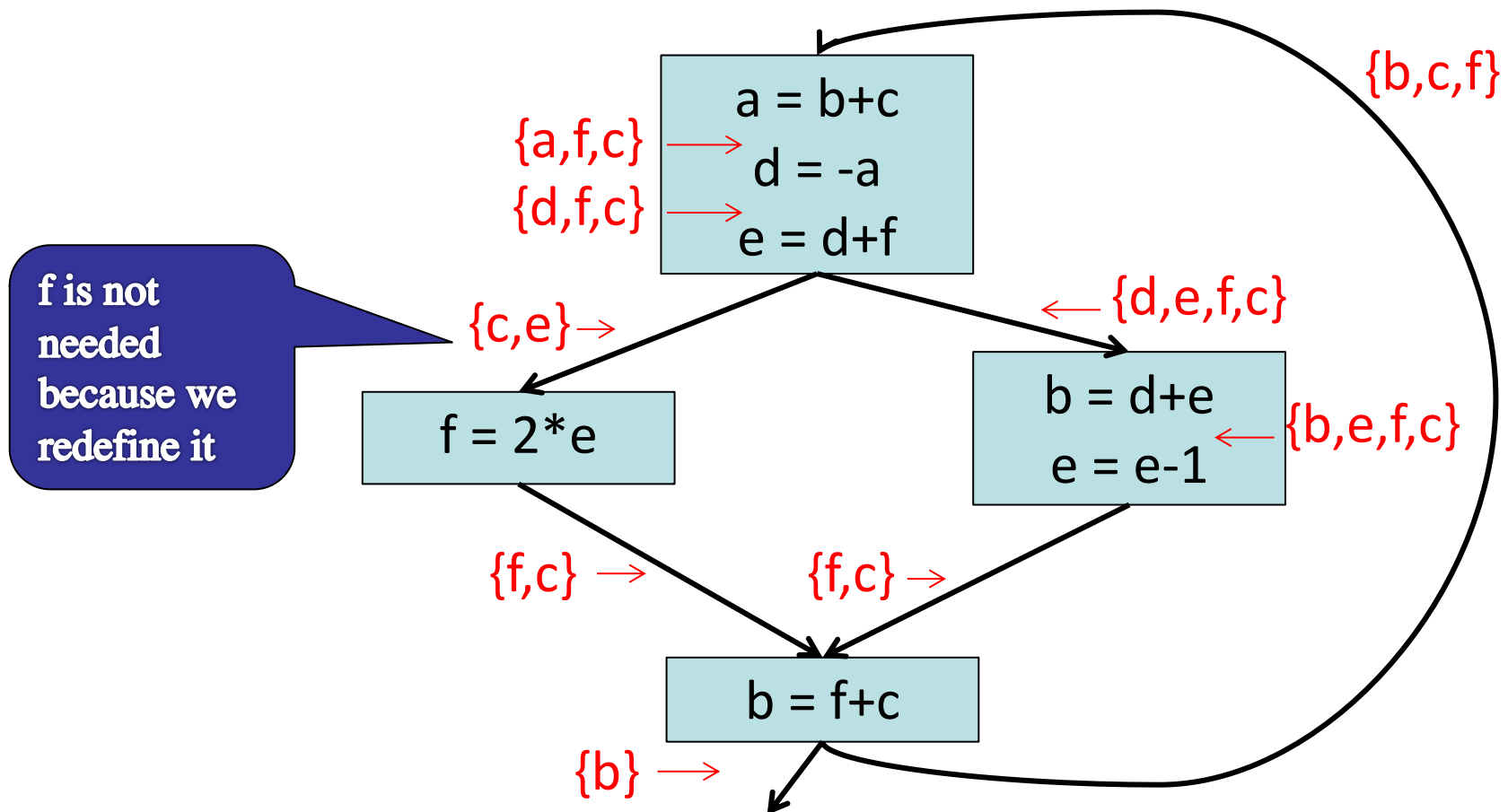
- Register allocation is as old as compilers
  - Register allocation was used in the original FORTRAN compiler in 1950's
  - Very crude algorithm
- A breakthrough came in 1980
  - Register allocation scheme based on graph coloring
  - Relatively simple, global and works well in practice

# Principles of Register Allocation

- *Temporaries  $t_1$  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***
  - *If  $t_1$  and  $t_2$  are live at the same time, they cannot share a register*
- We need liveness analysis

# Live Variables

- Compute live variables for each point



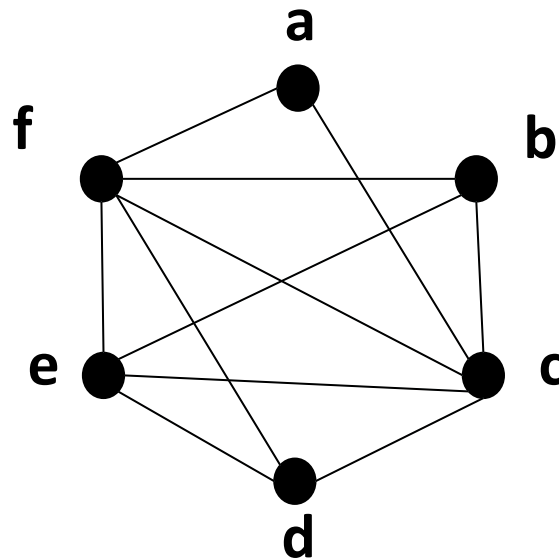
# Register Interference Graph

- Construct an undirected graph
  - A node for each temporary
  - An edge between  $t_1$  and  $t_2$  if they are live simultaneously at some point in the program
- This is the *register interference graph* (RIG)
  - Two temporaries can be allocated to the same register if there is no edge connecting them



# Register Interference Graph

- For our example



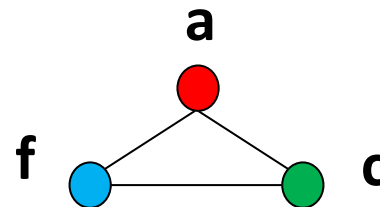
- **a** and **c** cannot be in the same register
- **a** and **d** could be in the same register

# Register Interference Graph

- Extracts exactly the information we need to characterize legal register allocation
- Gives the global view (i.e., over the entire control flow graph) picture of the register requirements
- After RIG construction the register allocation algorithm is architecture independent

# Graph Coloring

- A coloring of a graph is an assignment of colors to nodes, such that nodes connected by an edge have different colors
- A graph is  $k$ -colorable if it has a coloring with  $k$  colors

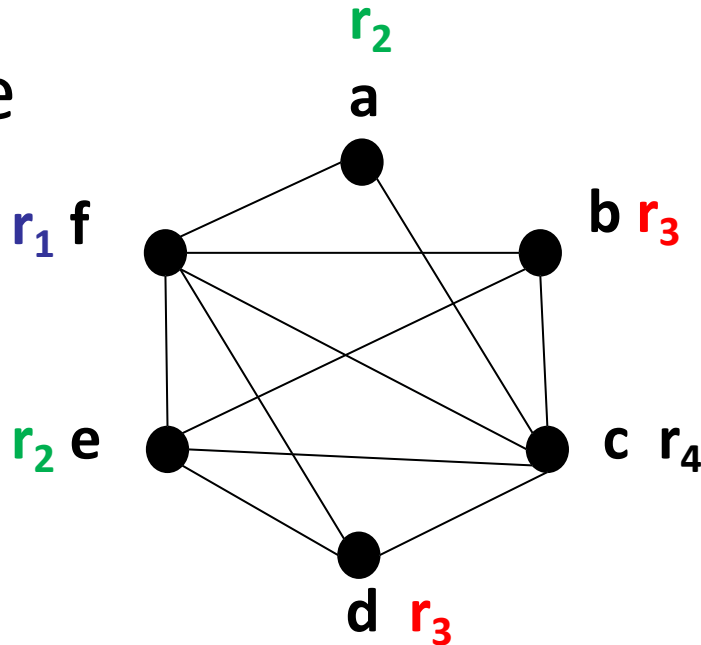


# Register Allocation as Graph Coloring

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

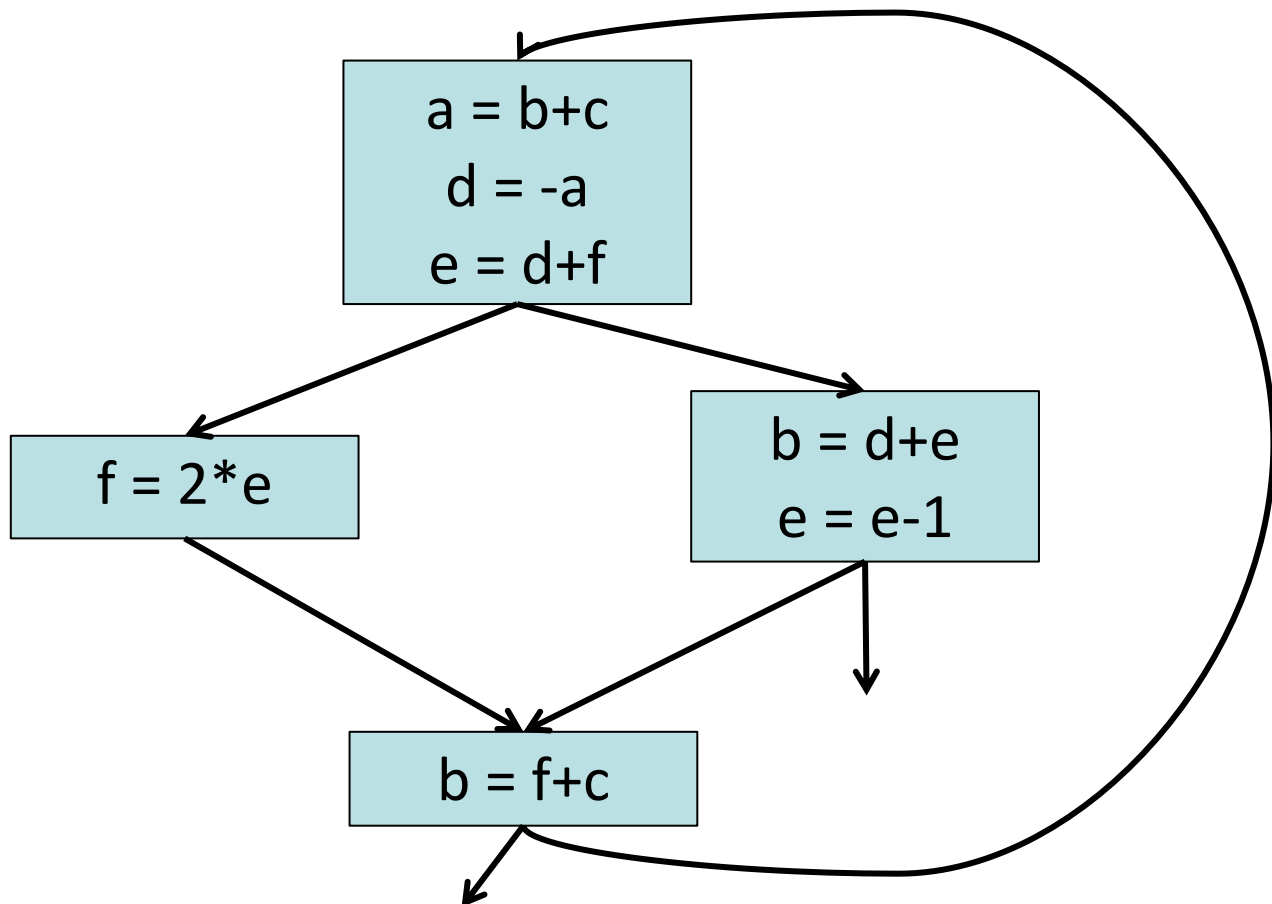
# Example

- For our example

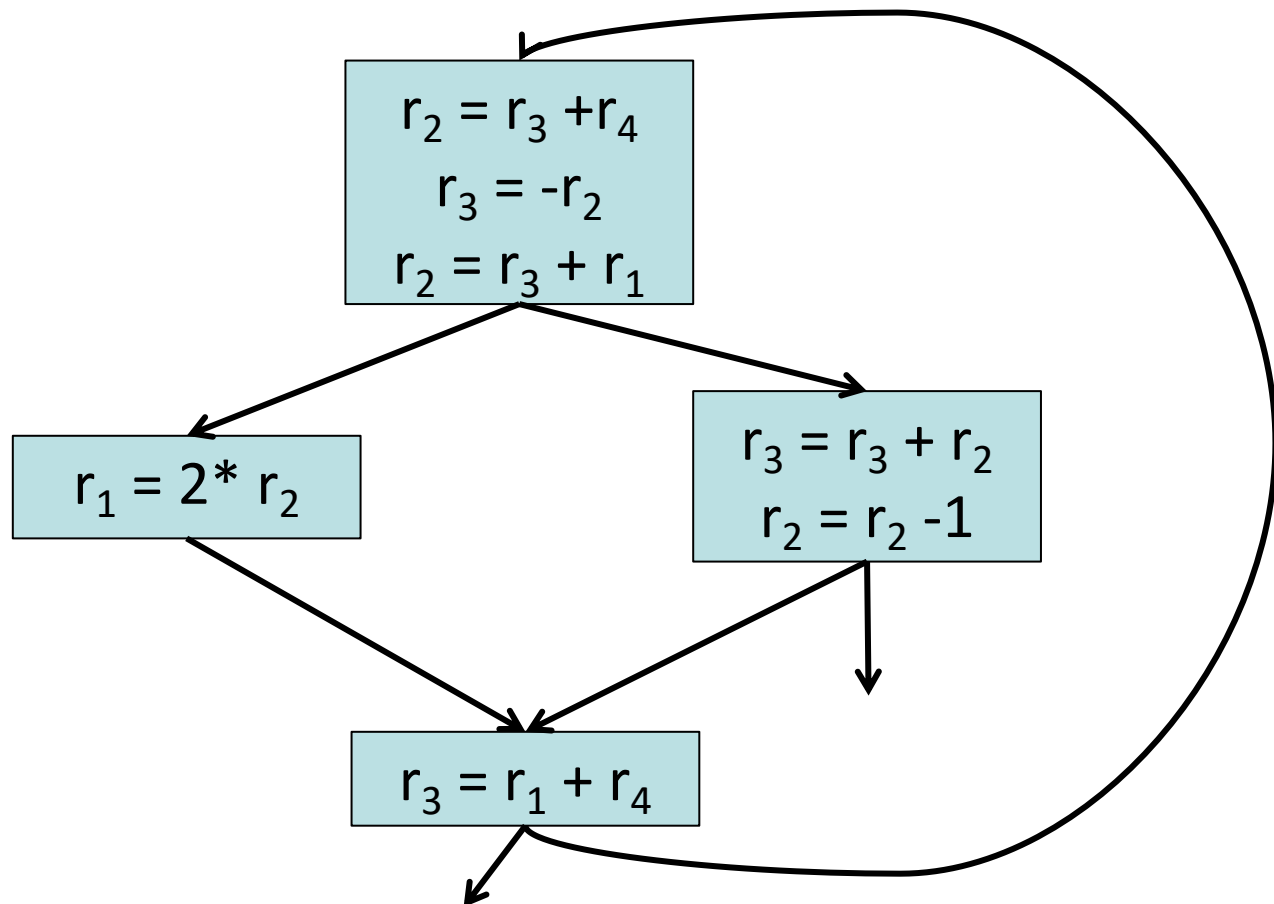


- There is no coloring with less than 4 colors
- There is a 4-coloring of this graph

# Control Flow Graph



# Register Allocation



# Graph Coloring

- How do we compute graph coloring?
- It is not easy :
  - The problem is NP-hard. No efficient algorithms are known
    - Solution: use heuristics
  - A coloring might not exist for a given number of registers
    - Solution: register spilling



# Register Allocation as Graph Coloring

- Main idea for solving whether a graph  $G$  is  $k$ -colorable:
- Pick any node  $t$  with fewer than  $k$  neighbors
- Remove  $n$  adjacent edges to create a new graph  $G'$
- If  $G'$  is  $k$ -colorable, then so is  $G$  (the original graph)
- Let  $c_1, \dots, c_n$  be the colors assigned to the neighbors of  $t$  in  $G'$
- Since  $n < k$  we can pick some color for  $t$  that is different from its neighbors

# Register Allocation as Graph Coloring

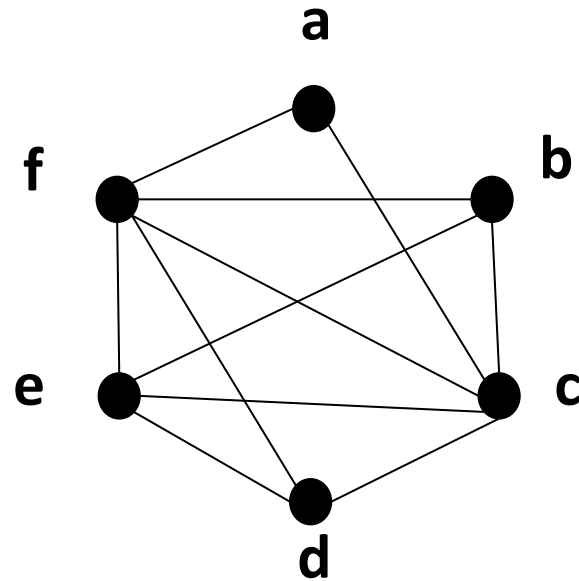
- Heuristic for graph coloring:
  - Ordering nodes (in an stack)
    1. Pick a node  $t$  with fewer than  $k$  neighbors
    2. Put  $t$  on a stack and remove it from the register interference graph (RIG)
    3. Repeat until the graph is empty
  - Assigning color to nodes on the stack:
    1. Start with the last node added
    2. At each step pick a color different from those assigned to already colored neighbors

# Example

- Assume  $k=4$

Remove **a**

stack={}

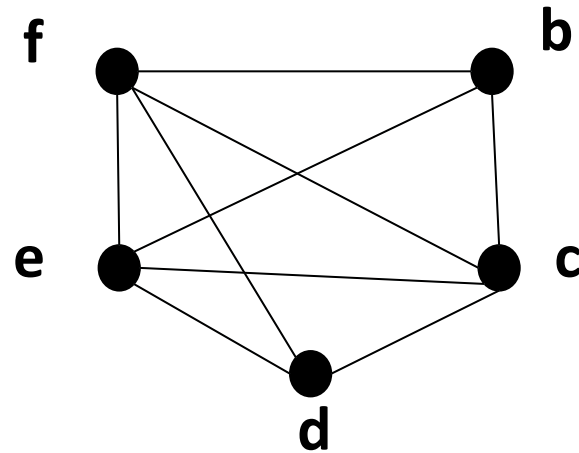


# Example

- Assume  $k=4$

Remove **d**

stack={a}

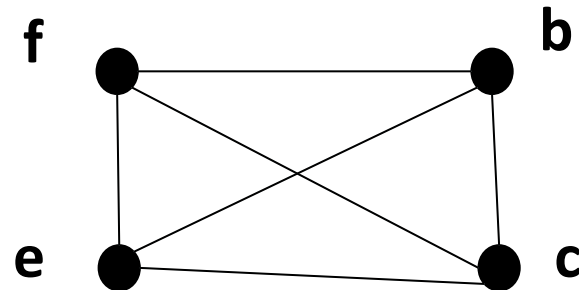


# Example

- Assume  $k=4$

Note: All nodes now have fewer than 4 neighbors

The graph coloring is  
guaranteed to succeed



Remove **c**

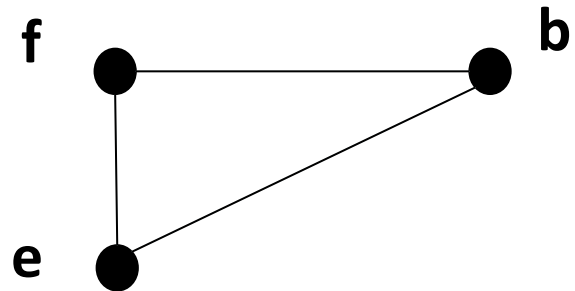
stack={d,a}

# Example

- Assume  $k=4$

Remove **b**

stack={c,d,a}

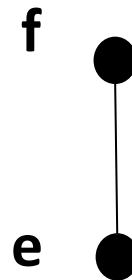


# Example

- Assume  $k=4$

Remove **e**

stack={b,c,d,a}



# Example

- Assume  $k=4$

f ●

Remove f

stack={e,b,c,d,a}



# Example

- Assume  $k=4$

Empty graph – done with the first part

Now we have the order for assigning colors to nodes, start coloring the nodes (from the top of the stack)

$\text{stack}=\{f,e,b,c,d,a\}$

# Example

- Assume  $k=4$

$r_1 f$  ●

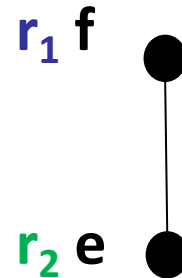
stack={e,b,c,d,a}

# Example

- Assume  $k=4$

$e$  must be in a different register from  $f$

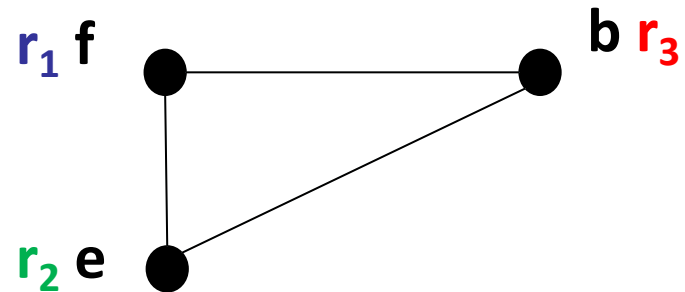
stack={b,c,d,a}



# Example

- Assume  $k=4$

stack={c,d,a}

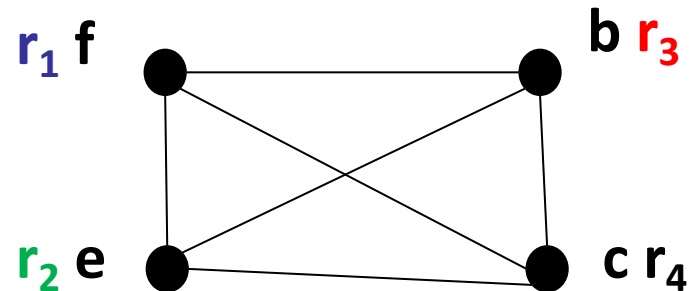


# Example

- Assume  $k=4$

The ordering insures we can find a color for all nodes

stack={d,a}

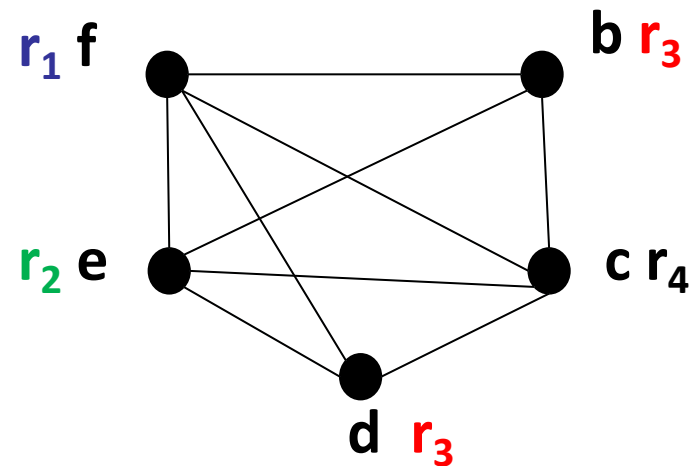


# Example

- Assume  $k=4$

$d$  can be in the same register as  $b$

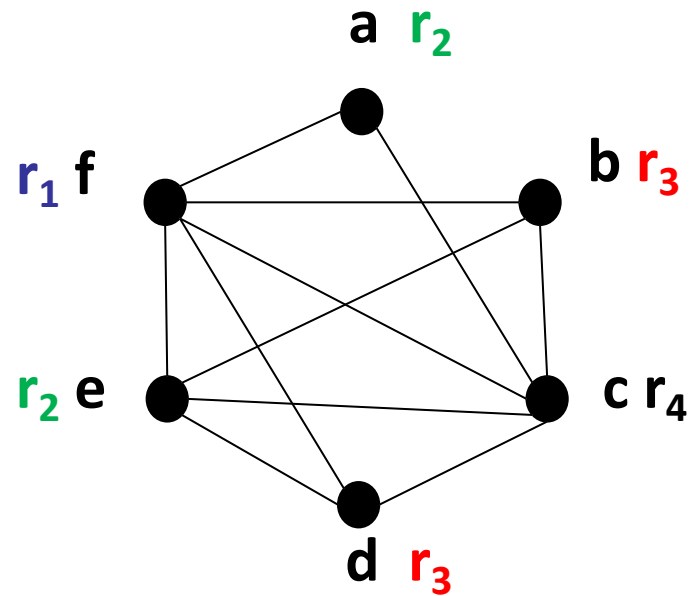
stack={a}



# Example

- Assume  $k=4$

stack={}



# Register Allocation as Graph Coloring

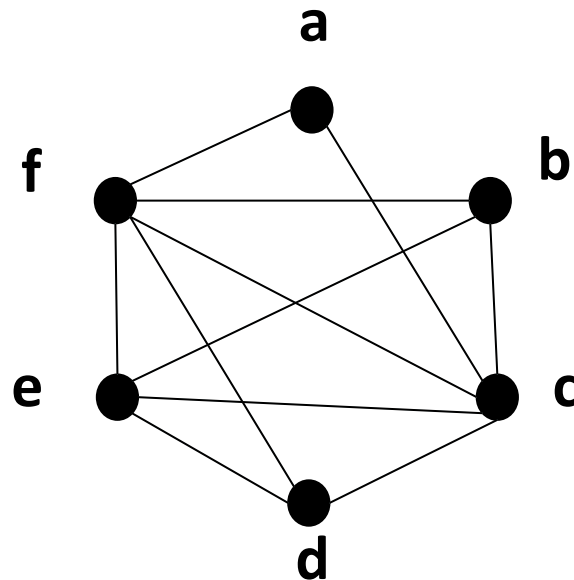
- What happens if the graph coloring heuristic fails to find a coloring?
- In this case we cannot hold all values in the registers
  - Some values should be *spilled* to memory



# K-coloring fails

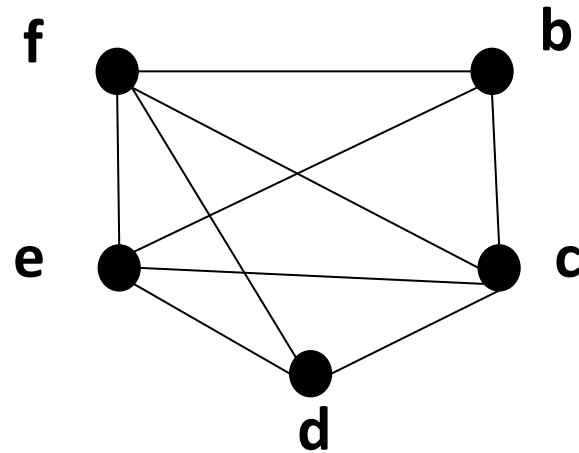
- What if all nodes have  $k$  or more neighbors?
- Try to find a 3 coloring of this graph

Remove **a**



# Example of 3-coloring

- There is no node such that if we remove it then 3-coloring for the graph is available

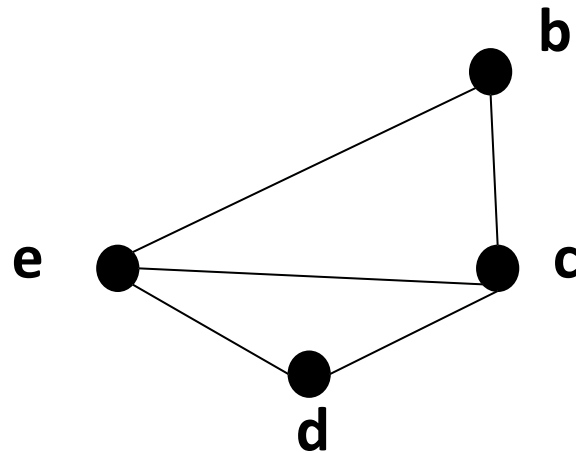


# Optimistic Coloring

- If every node in  $G$  has more than  $k$  neighbors,  $k$ -coloring of  $G$  might not be possible
- Pick a node as candidate for spilling, remove it from the graph and continue  $k$ -coloring

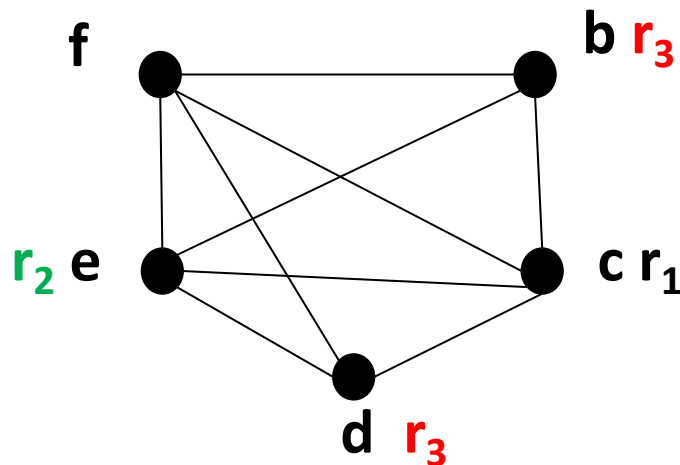
# Optimistic Coloring

- Remove **f** and continue:
  - The ordering:  $\{c, e, d, b, \mathbf{f}, a\}$



# Optimistic Coloring

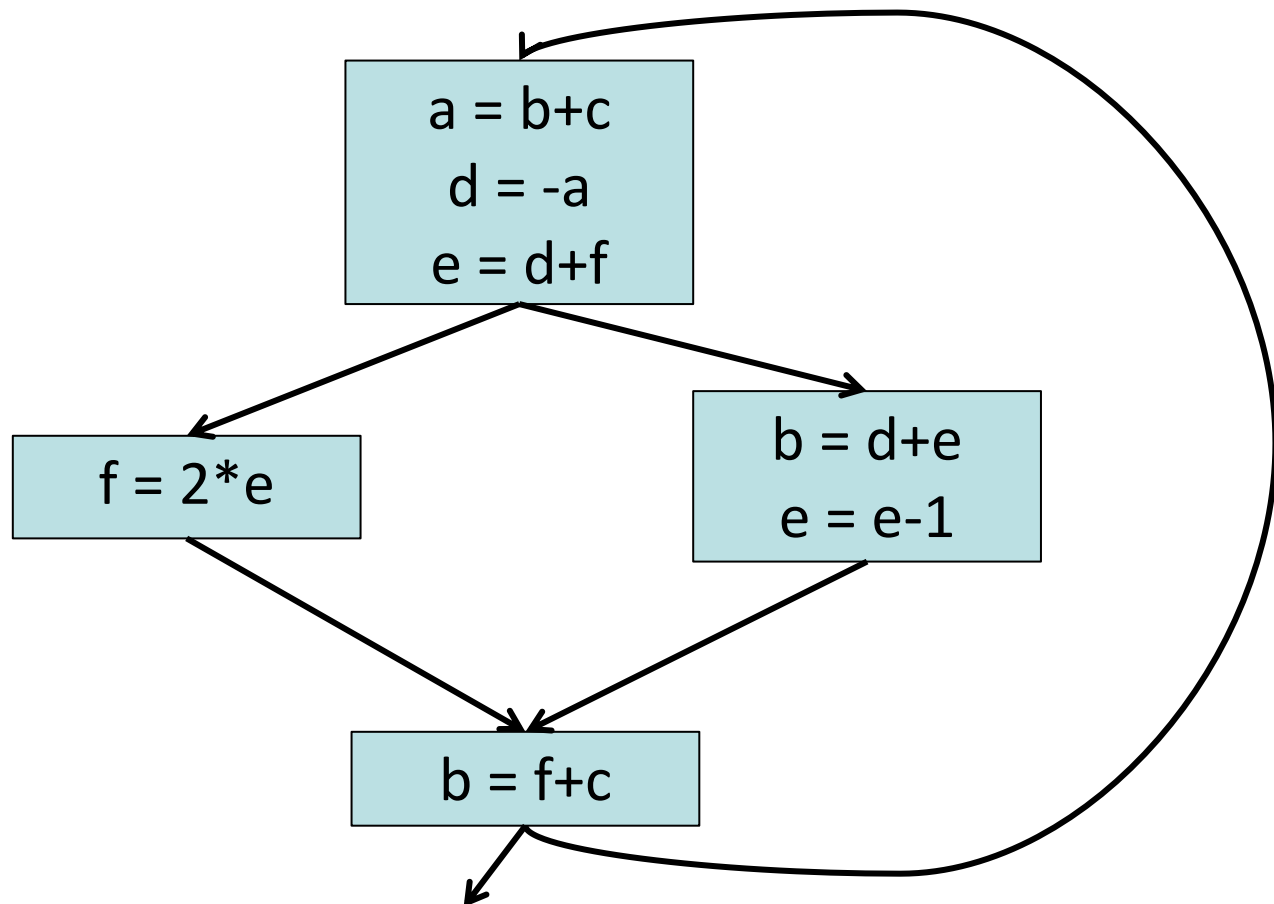
- Color the nodes  $\{c, e, d, b, f, a\}$
- Try to assign a color to  $f$
- We hope that among 4 neighbors of  $f$  we use less than 3 colors (*optimistic coloring*)



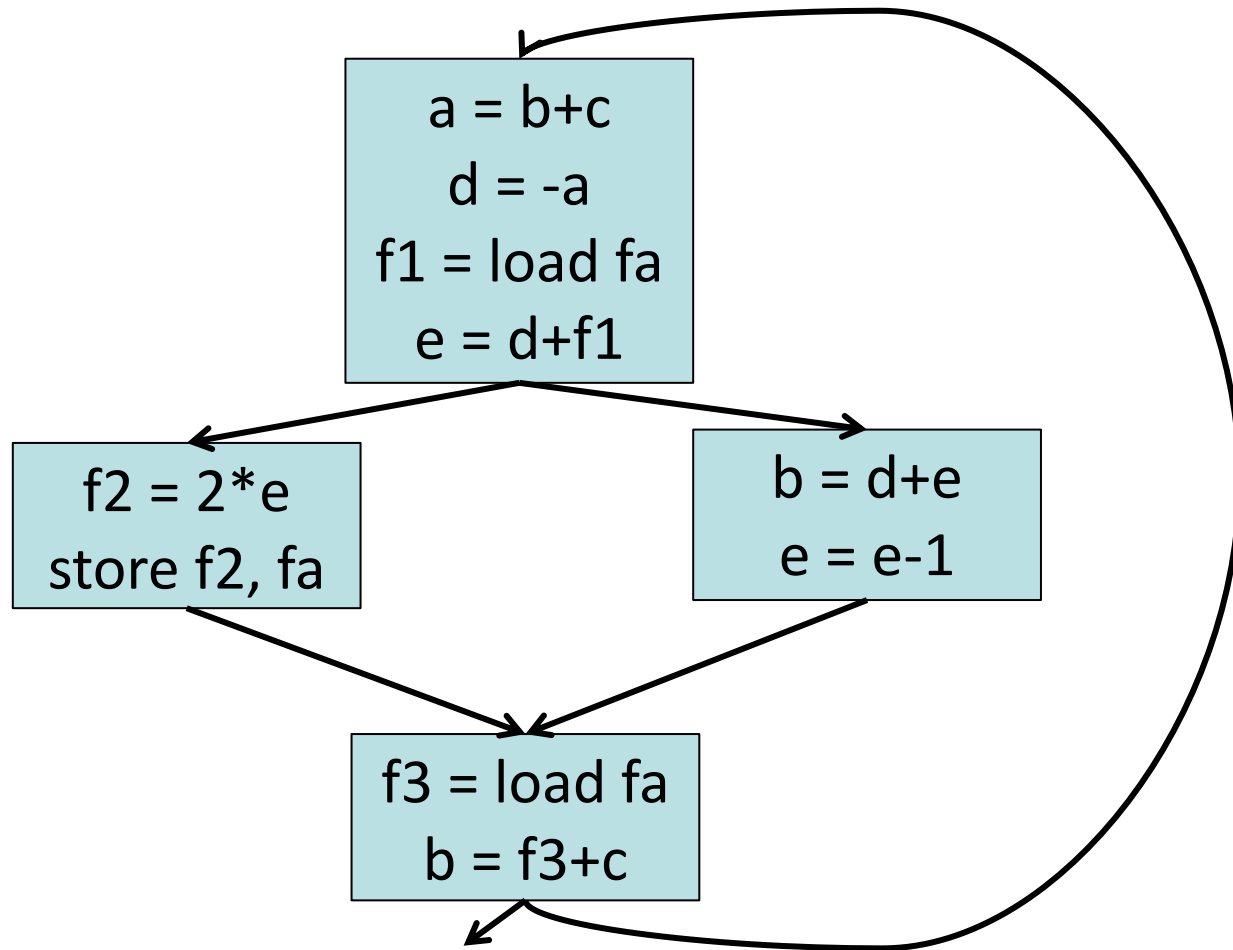
# Spilling

- 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 is slow but sometimes necessary.

# Original Code

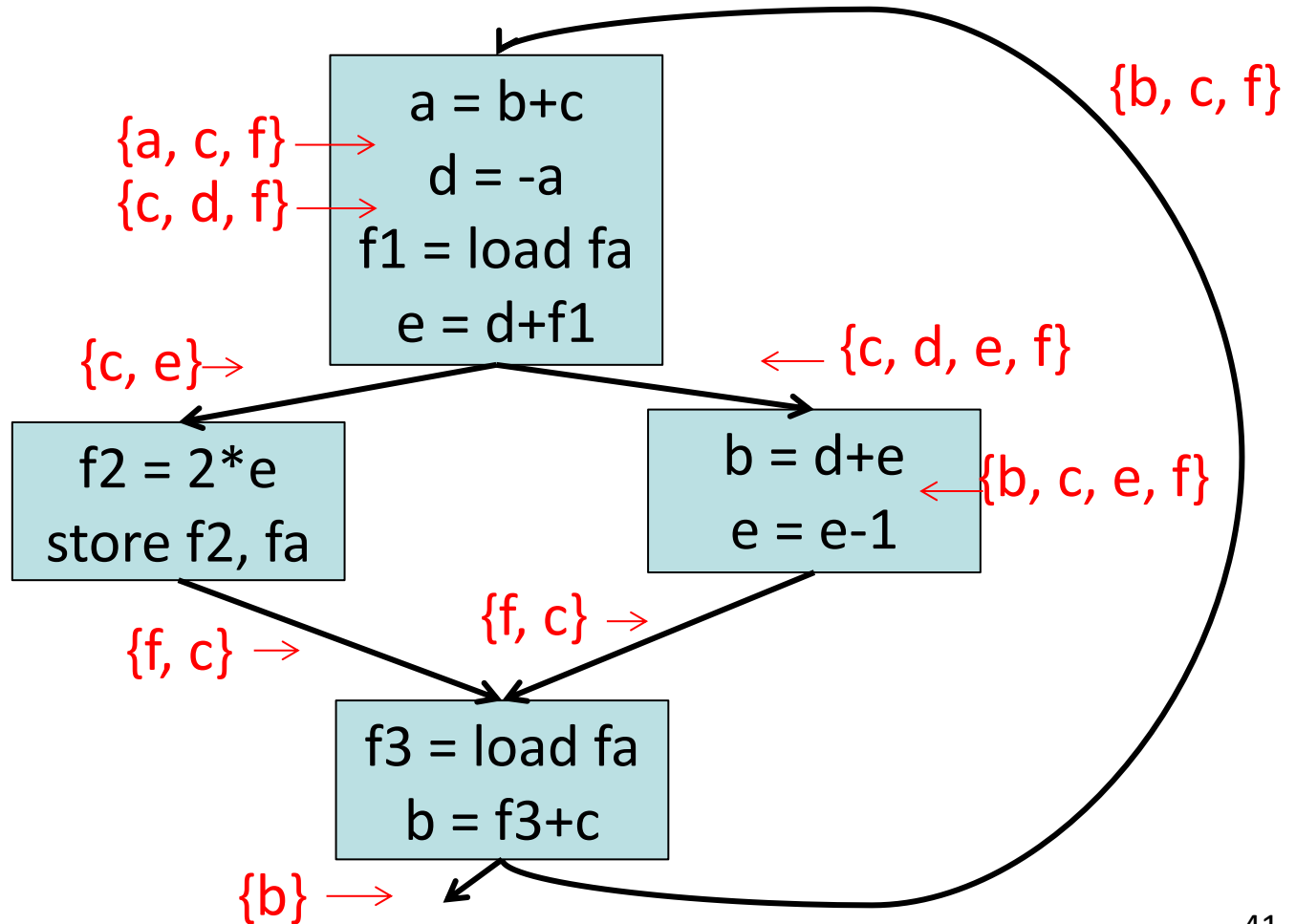


# Code after Spilling f

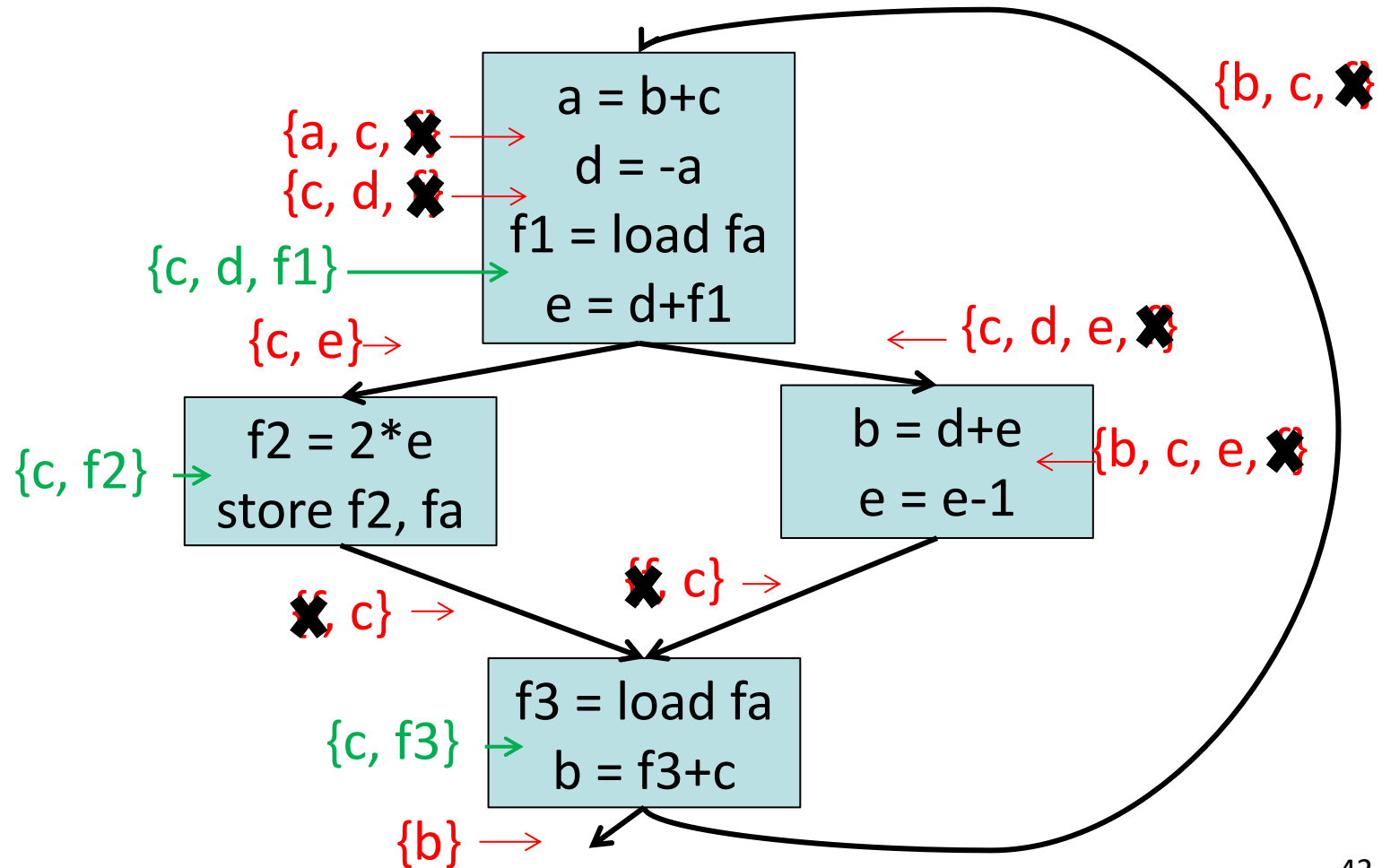




# Recompute the Liveness



# Recompute the Liveness

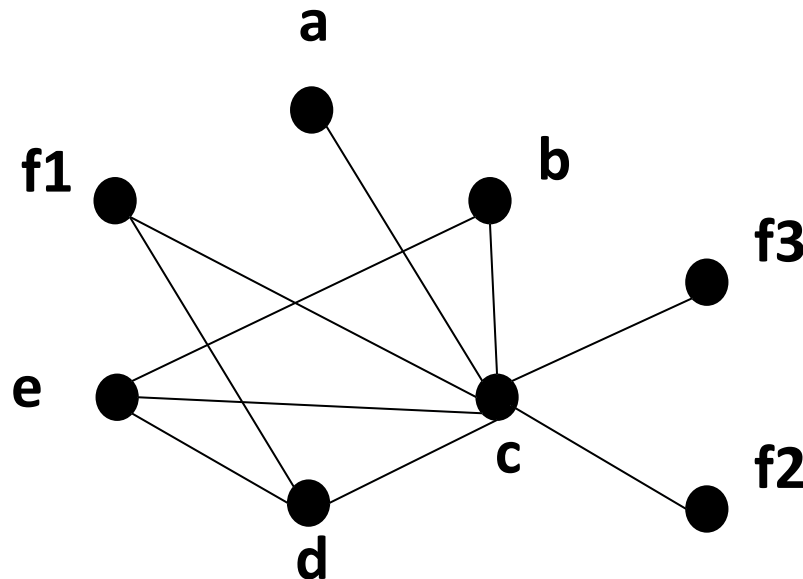


# Rebuild the Interference Graph

- 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

# Rebuild the Interference Graph

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



# Spilling

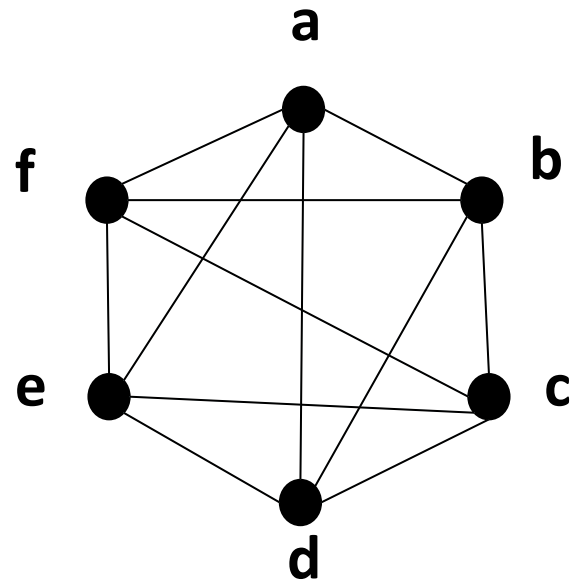
- Additional spilling might be required before a coloring is found

# Example

$K=3$

remove a

Stack: {}

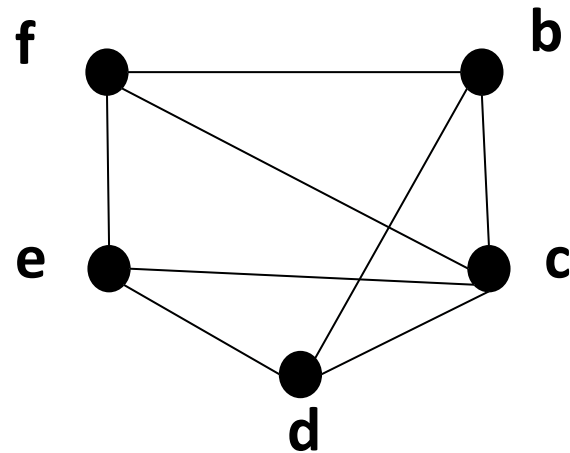


# Example

$K=3$

remove c

Stack: {**a**}

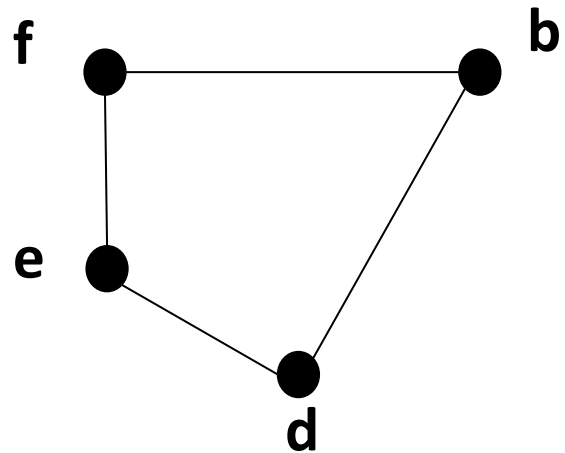


# Example

$K=3$

remove b

Stack: {c,a}



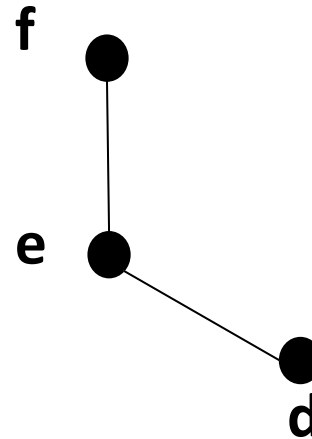


# Example

K=3

remove e

Stack: {b,c,a}



# Example

K=3

remove f

Stack: {e,b,c,a}

f ●

●  
d

# Example

K=3

remove d

Stack: {f,e,b,c,a}

●  
d

# Example

K=3

Stack: {d,f,e,b,c,a}

# Example

K=3

Stack: {f,e,b,c,a}

●  
d r1

# Example

K=3

Stack: {e,b,c,a}

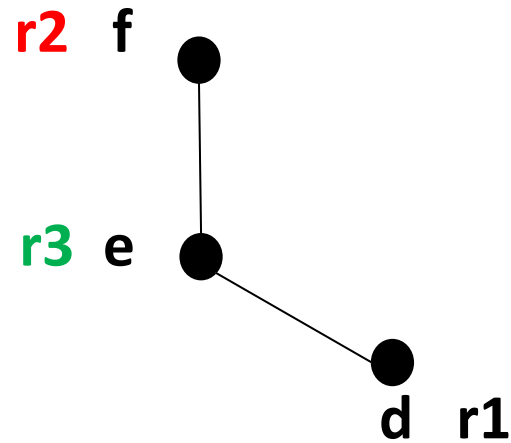
r2 f ●

●  
d r1

# Example

K=3

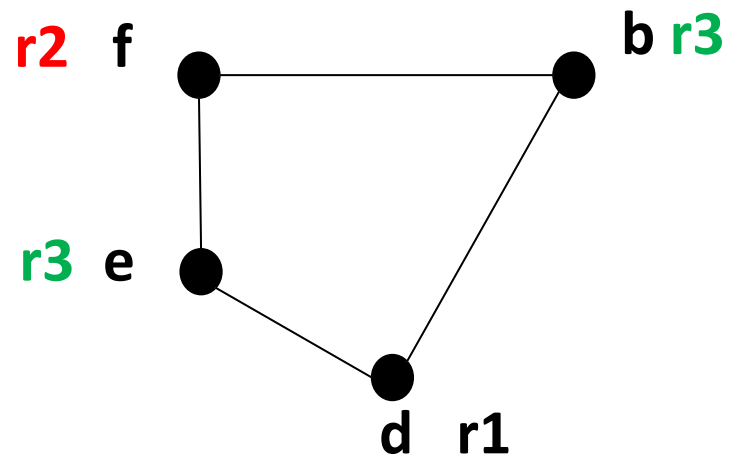
Stack: {b,c,a}



# Example

K=3

Stack: {c,a}

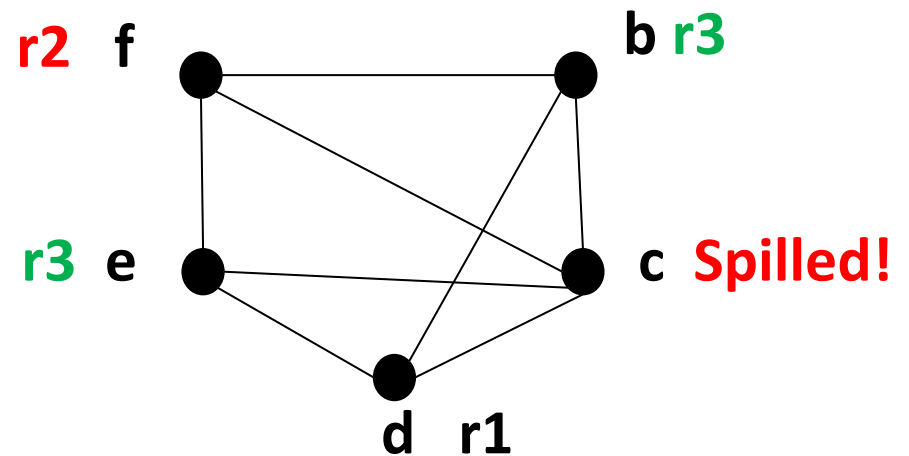




# Example

K=3

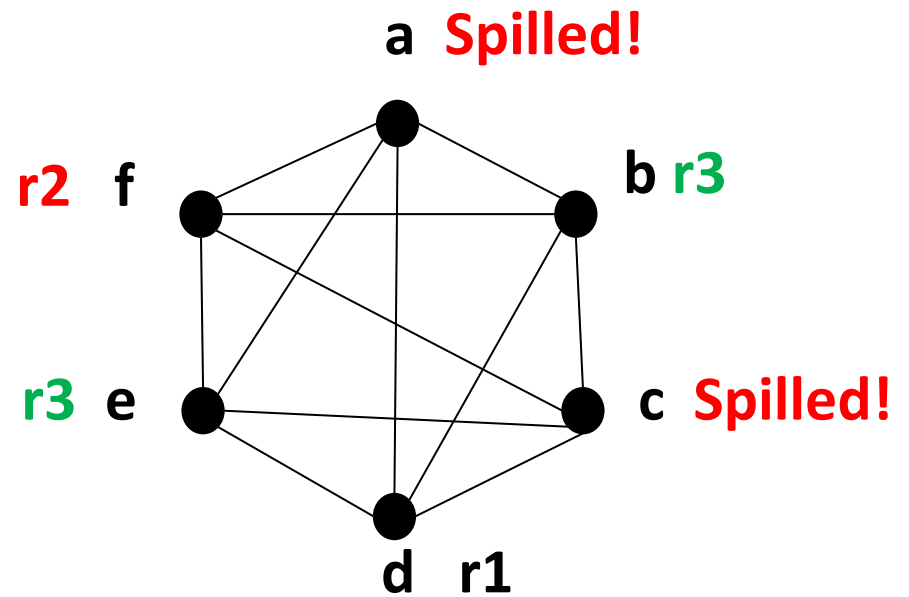
Stack: {**a**}



# Example

K=3

Stack: {}



# Example

K=3

Stack: {d,f,e,b,c,a}

# Example

K=3

Stack: {f,e,b,c,a}

●  
d r1

# Example

K=3

Stack: {e,b,c,a}

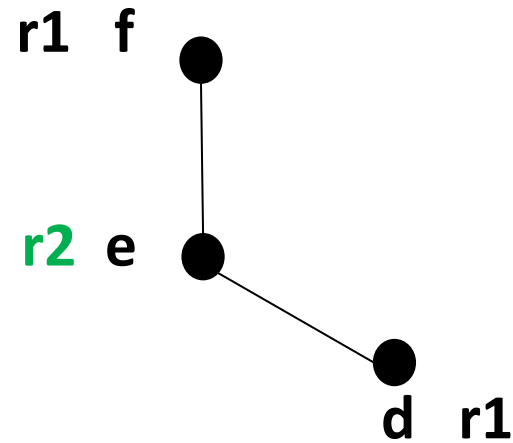
r1 f ●

●  
d r1

# Example

K=3

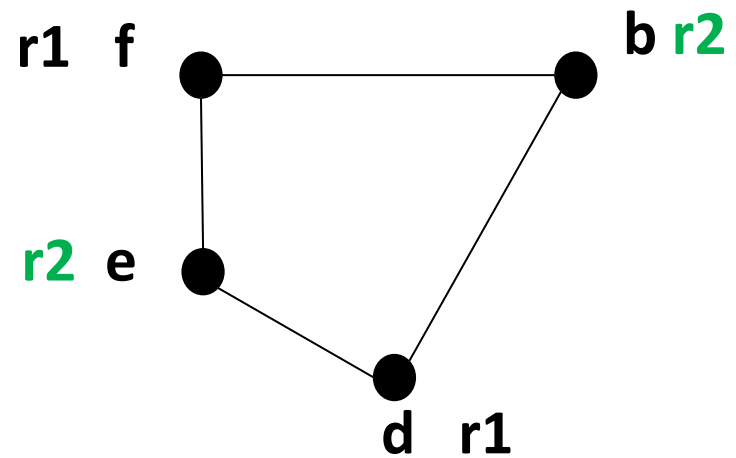
Stack: {b,**c**,**a**}



# Example

K=3

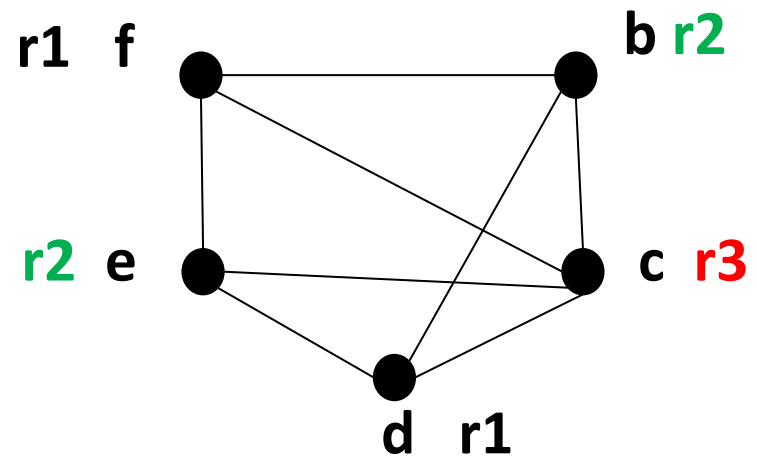
Stack: {c,a}



# Example

K=3

Stack: {**a**}

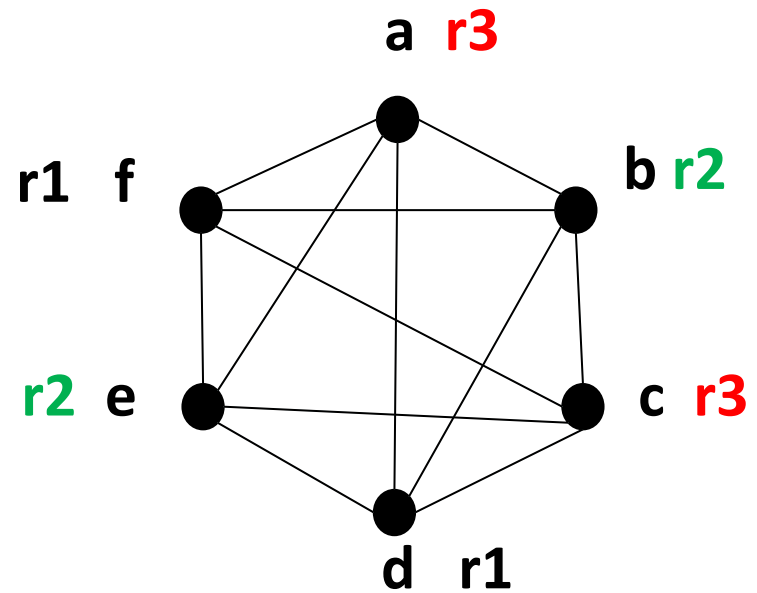




# Example

K=3

Stack: {}



# Spilling

- Many different heuristics for picking a node to spill
  - Spill temporaries with most conflicts
  - Spill temporaries with few definitions and uses
  - Avoid spilling in inner loops (heavily visited regions of the code)
- C allows a *register* keyword to direct the compiler whether a variable contains a value that is heavily used.

# Live Ranges and Live Intervals

- The live range for a variable is the set of program points at which that variable is live.
- The live interval for a variable is the smallest subrange of the IR code containing all a variable's live ranges.
  - A property of the IR code, not CFG.
  - Less precise than live ranges, but simpler to work with

# Live Intervals

$e = d + a$

$f = b + c$

$f = f + b$

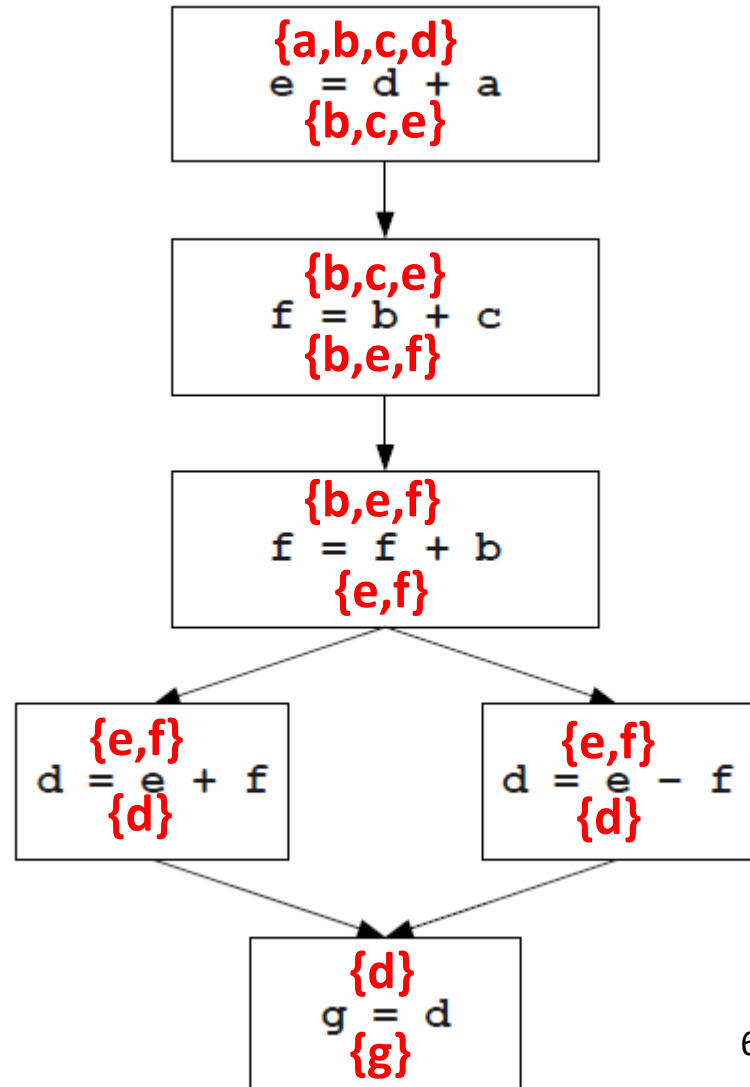
if  $e == 0$  goto \_L0

$d = e + f$

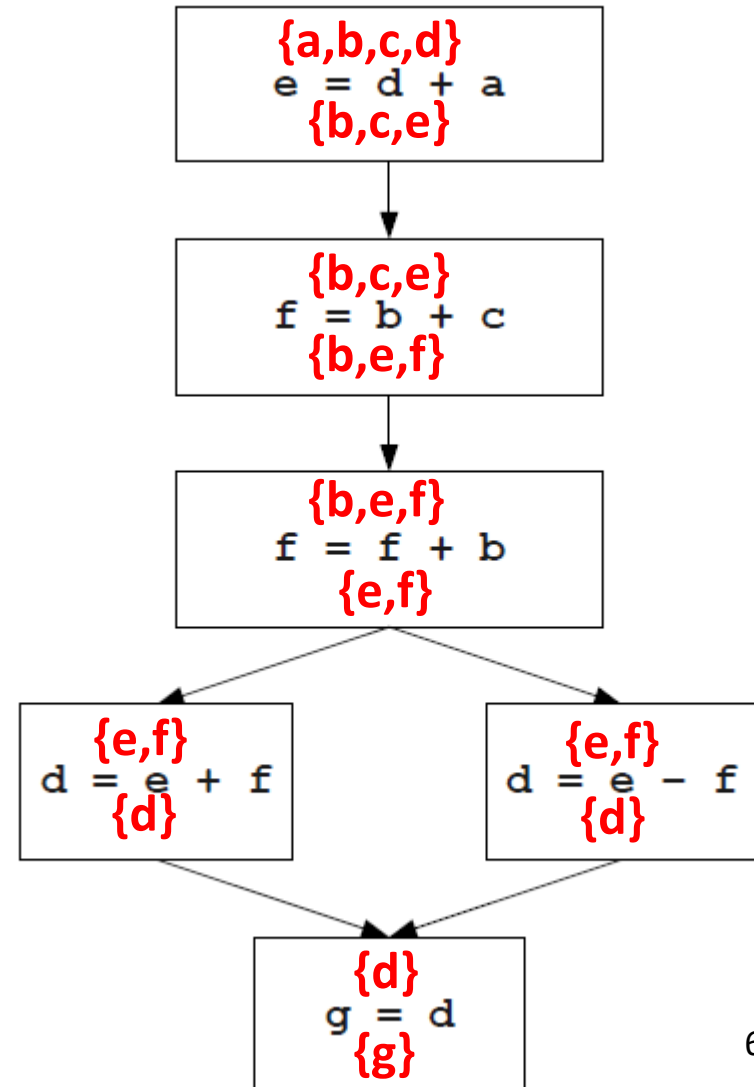
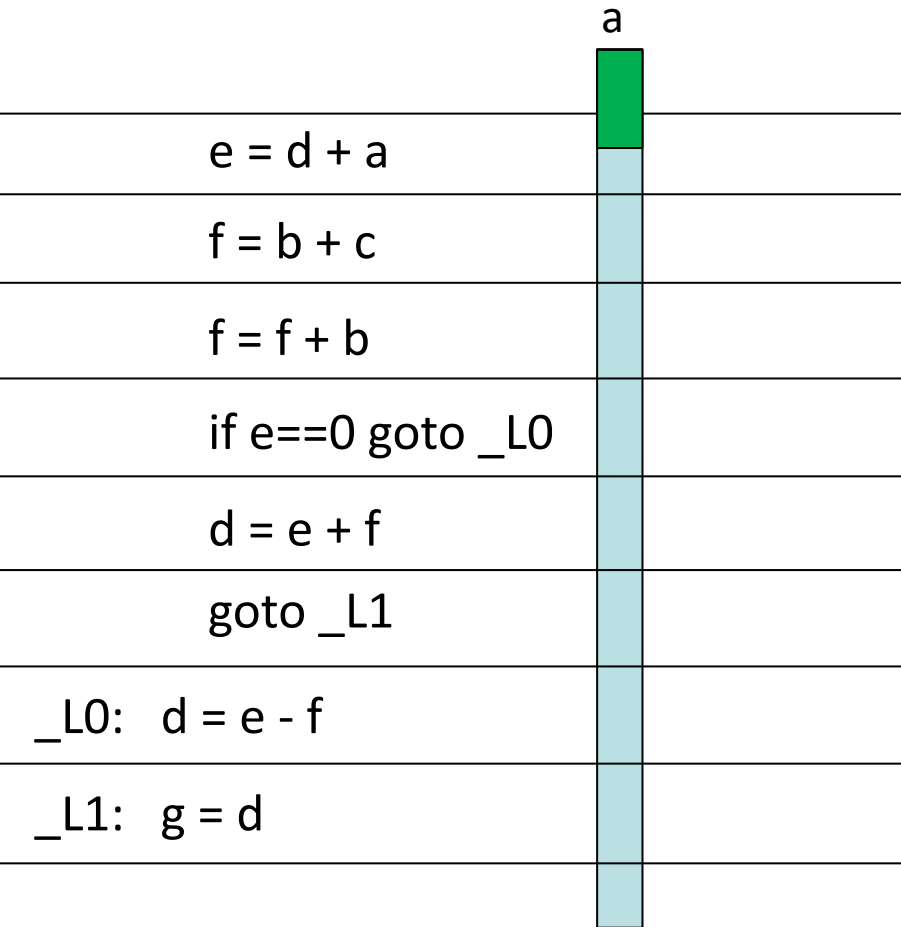
goto \_L1

\_L0:  $d = e - f$

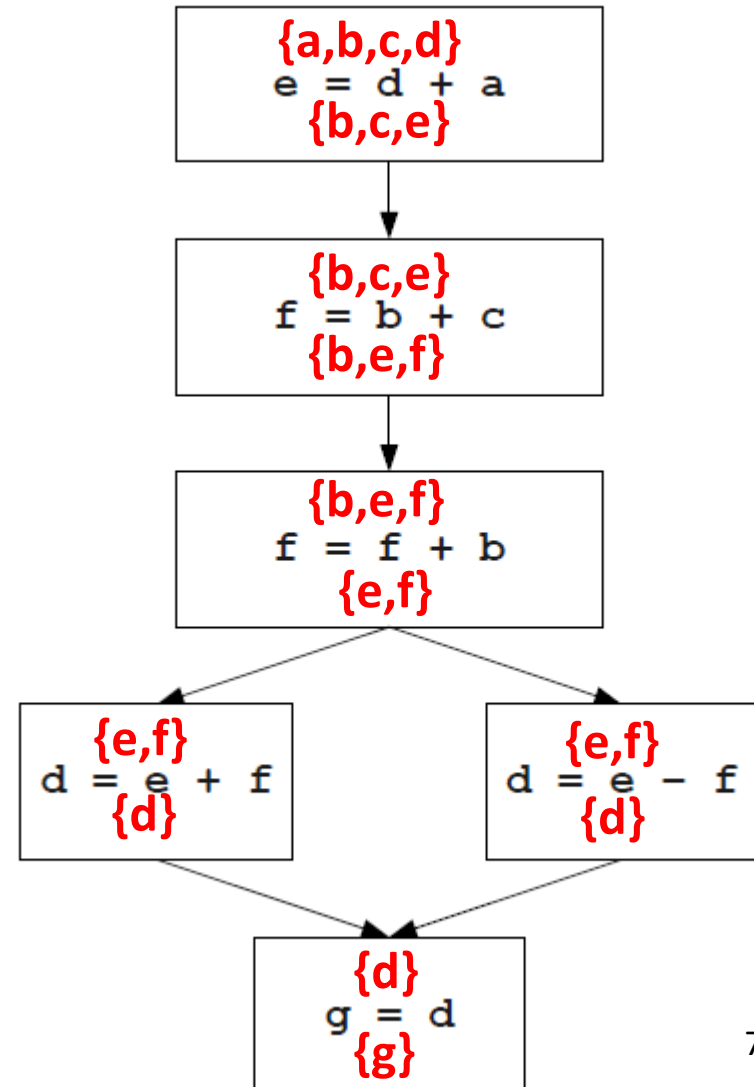
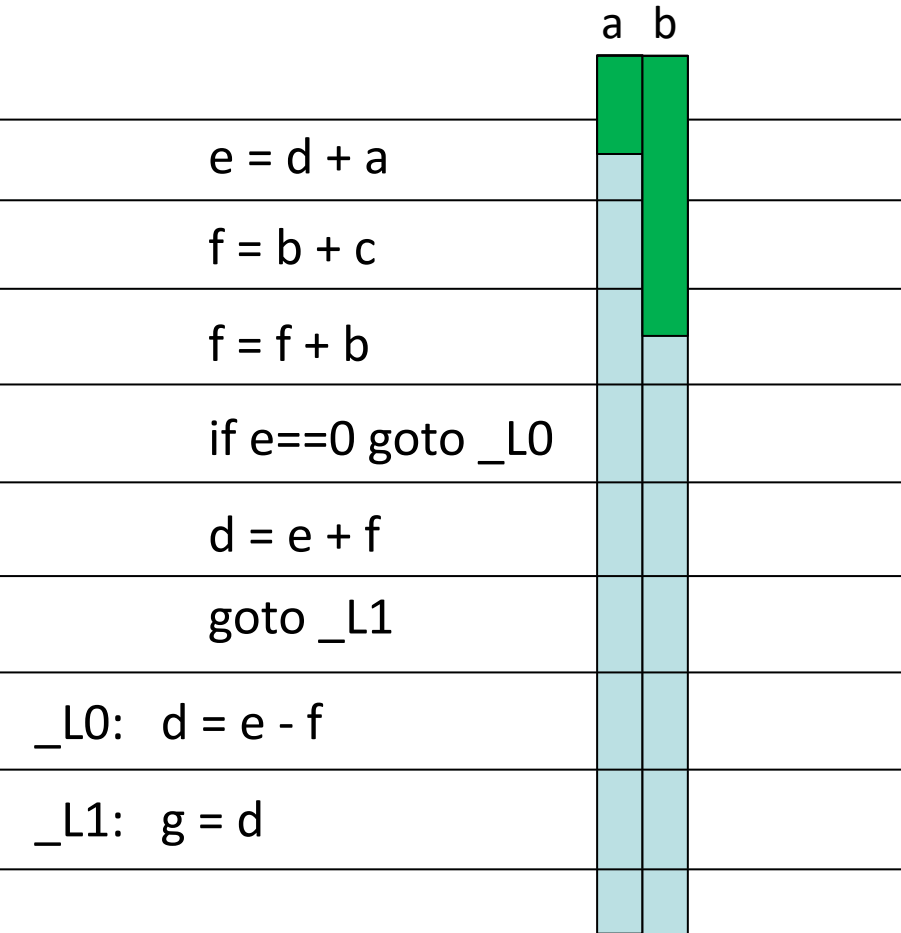
\_L1:  $g = d$



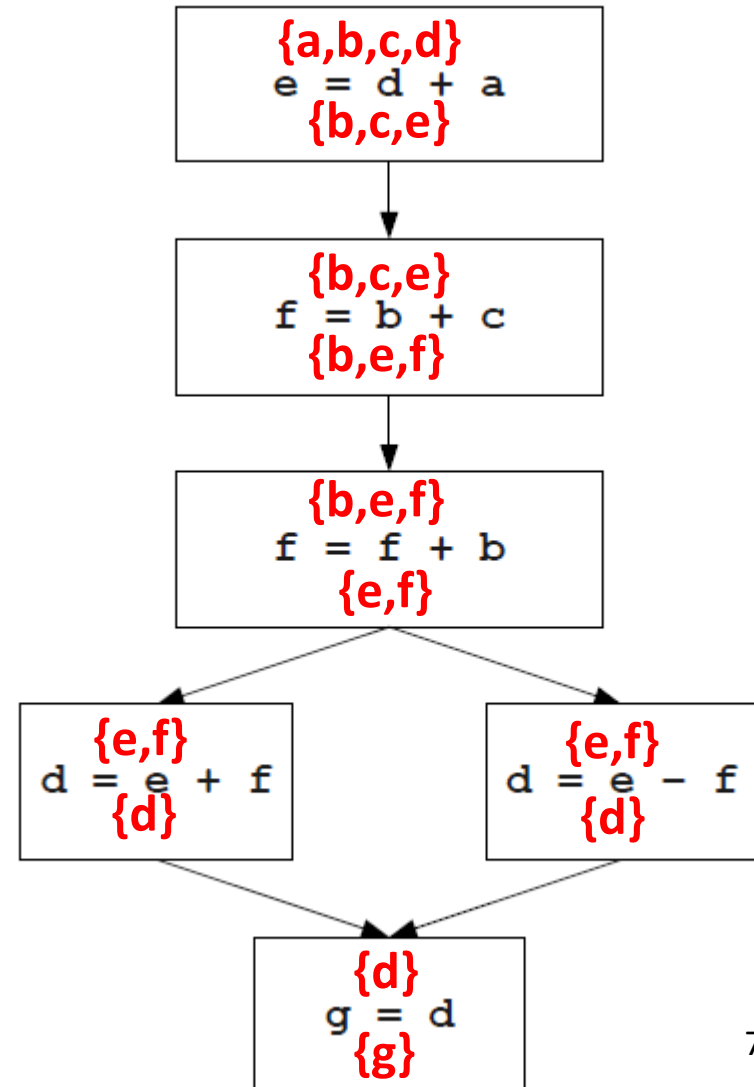
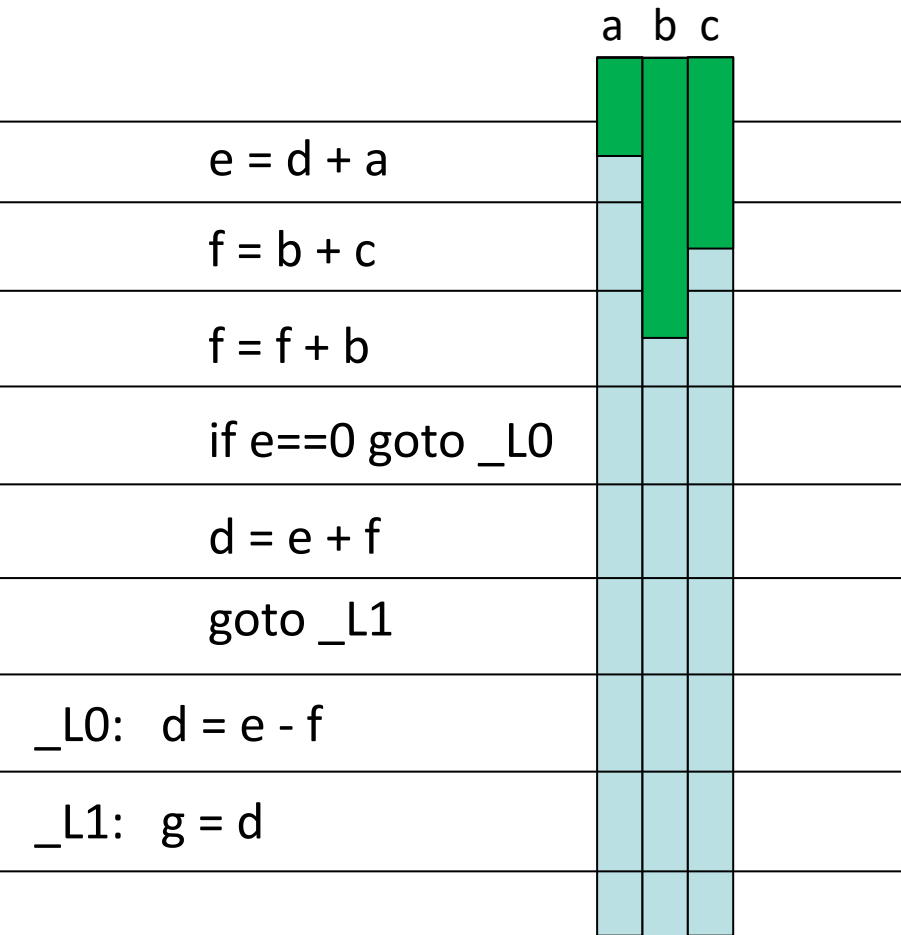
# Live Intervals



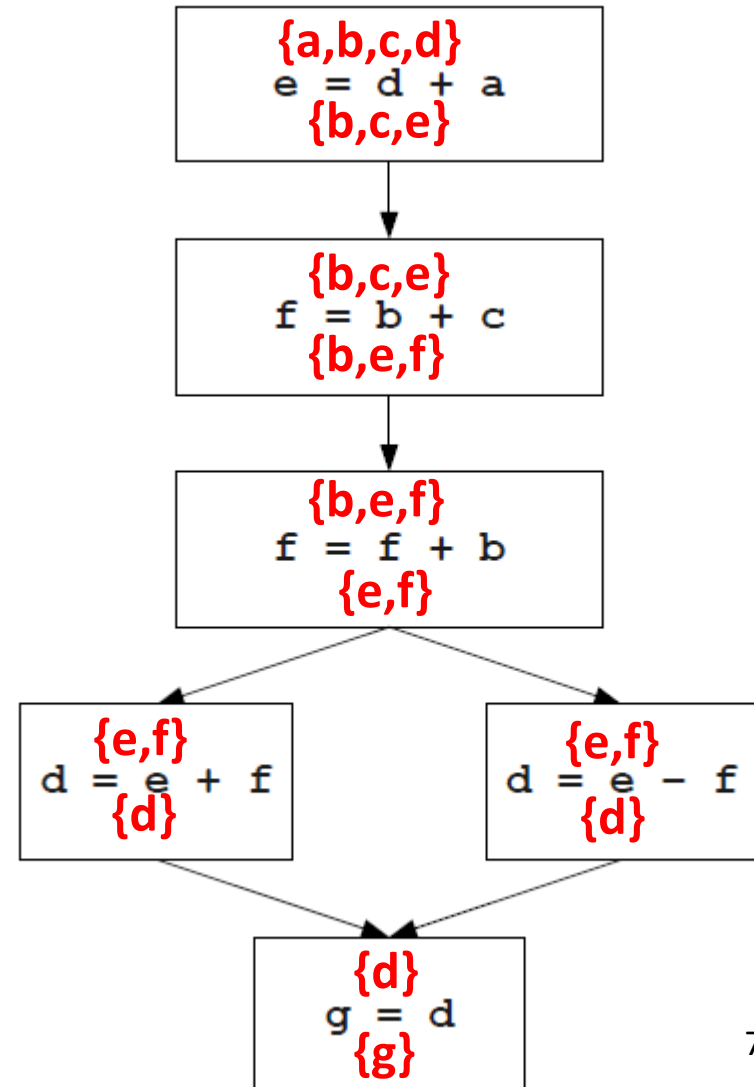
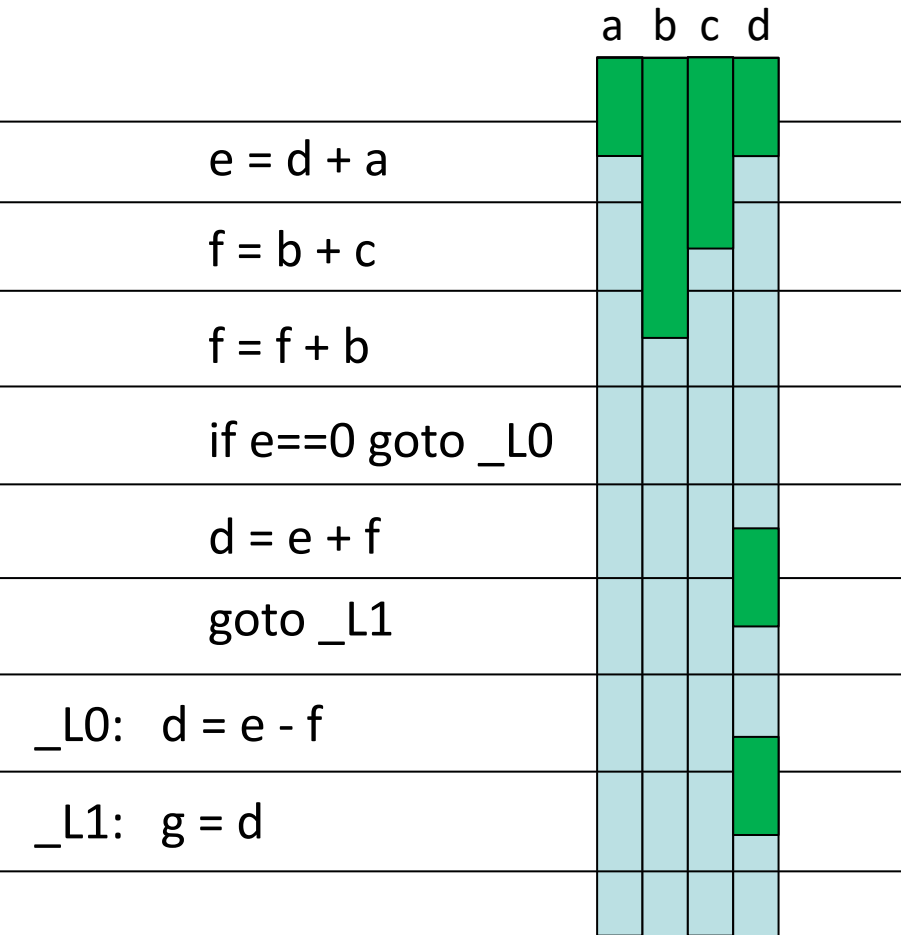
# Live Intervals



# Live Intervals

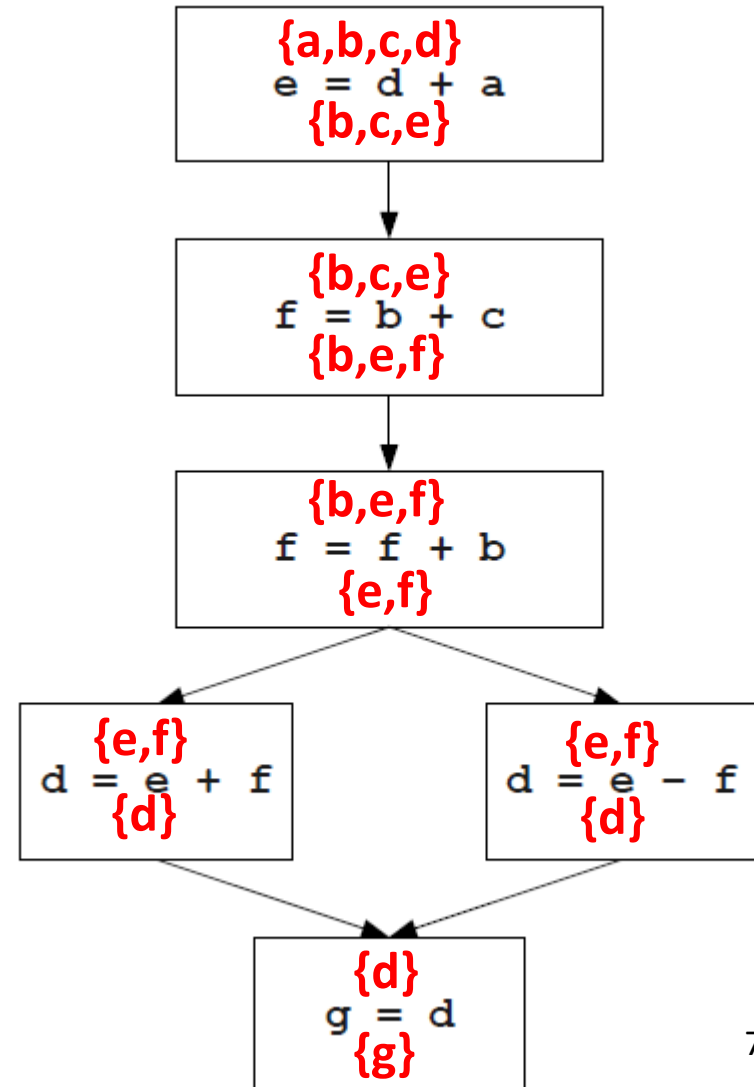
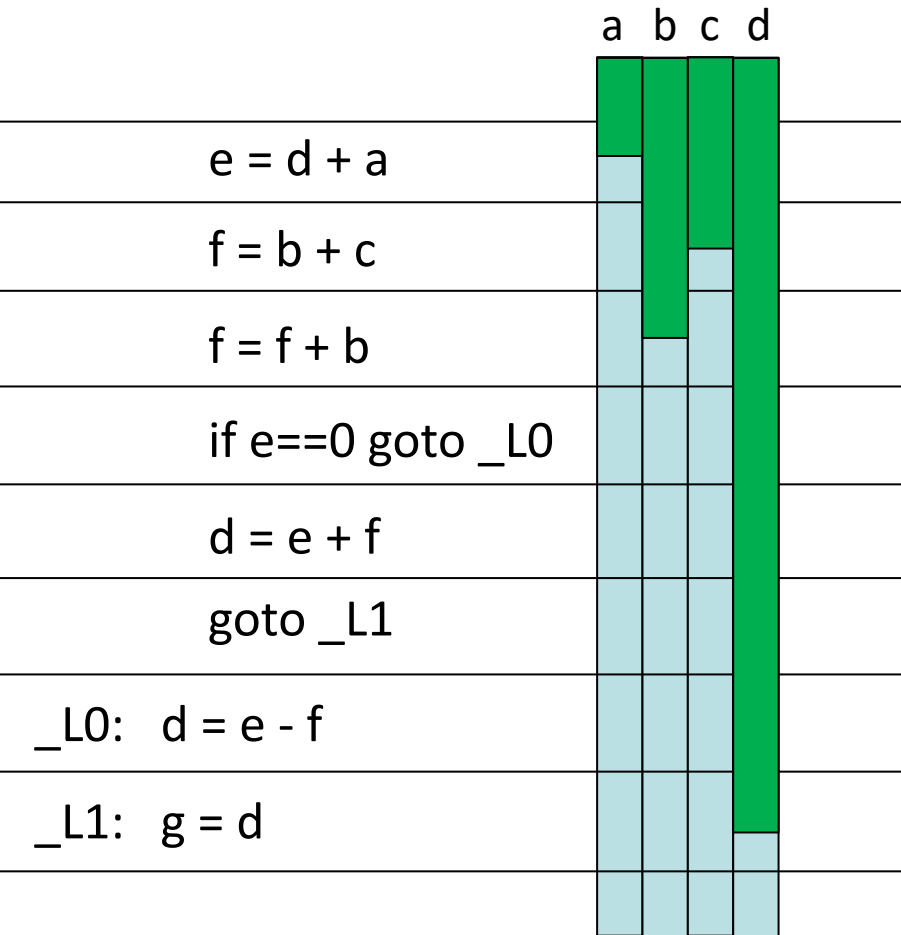


# Live Intervals

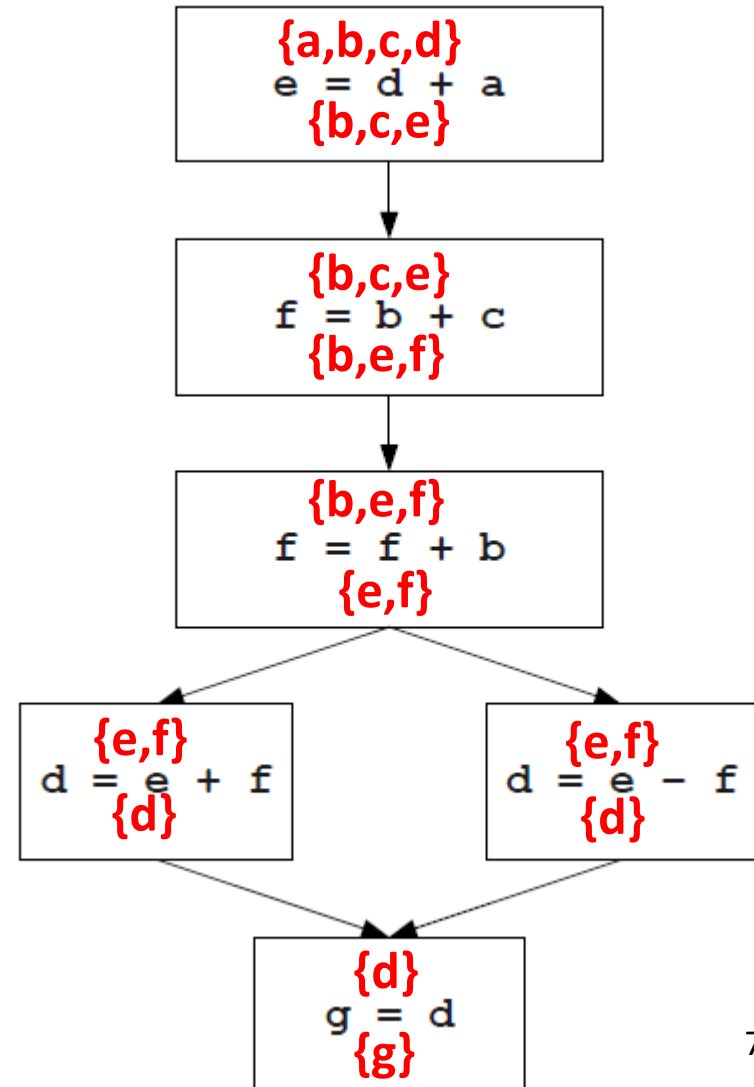
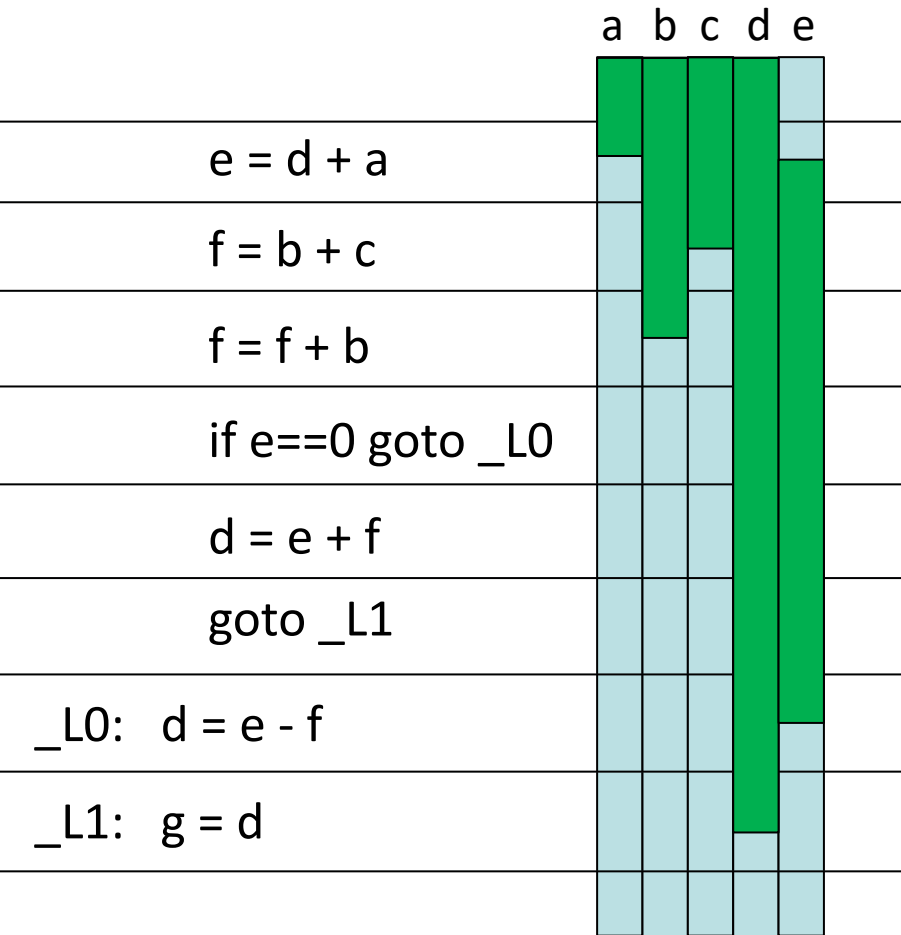




# Live Intervals

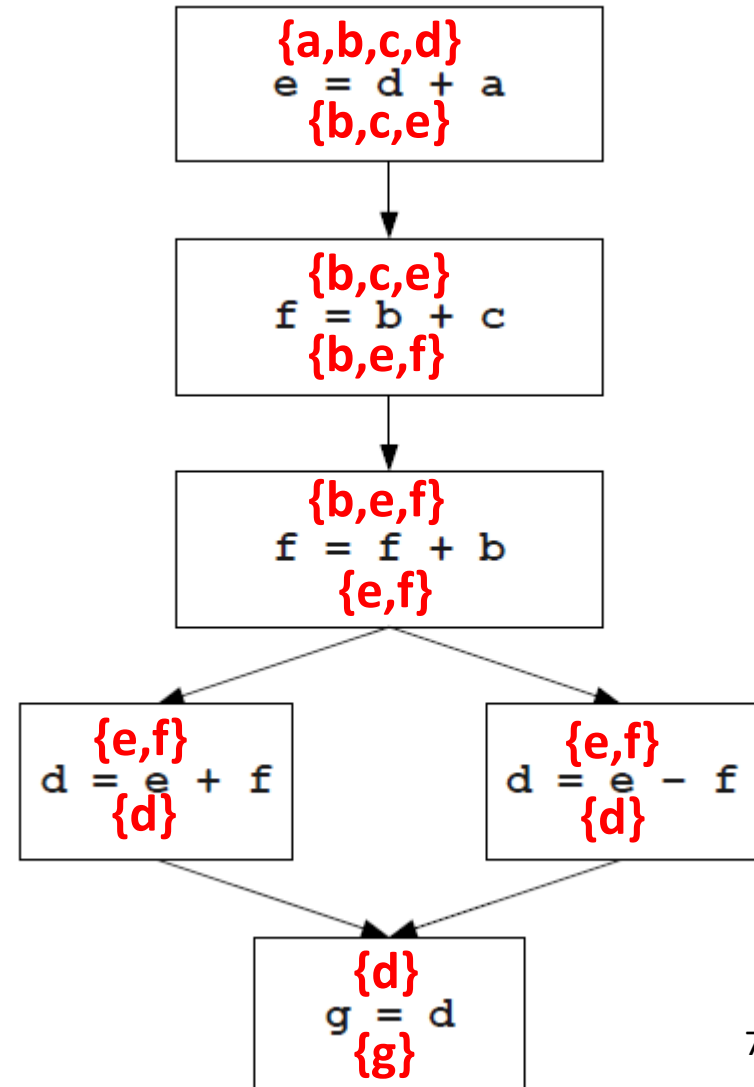


# Live Intervals



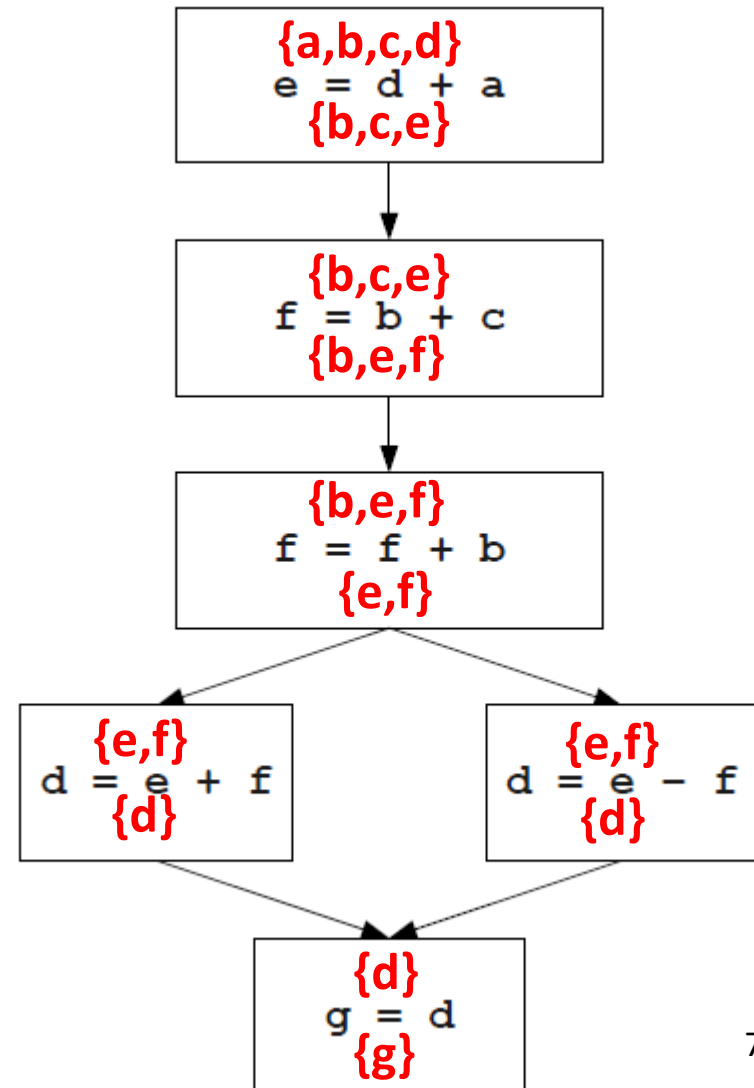
# Live Intervals

	a	b	c	d	e	f
	█	█	█	█	█	█
$e = d + a$	█	█	█	█	█	█
$f = b + c$	█	█	█	█	█	█
$f = f + b$	█	█	█	█	█	█
if $e == 0$ goto _L0	█	█	█	█	█	█
$d = e + f$	█	█	█	█	█	█
goto _L1	█	█	█	█	█	█
_L0: $d = e - f$	█	█	█	█	█	█
_L1: $g = d$	█	█	█	█	█	█



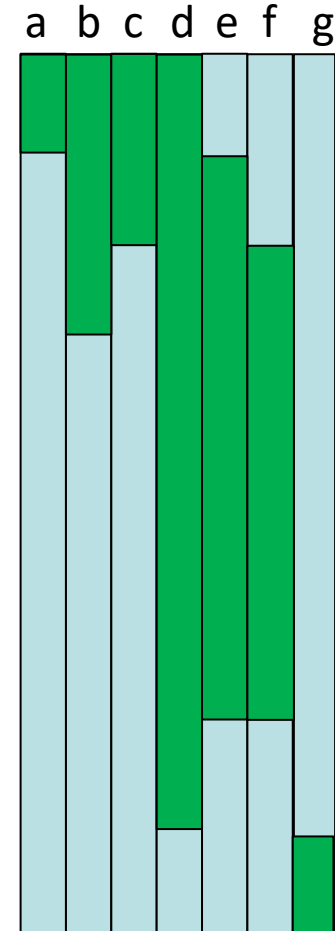
# Live Intervals

	a	b	c	d	e	f	g
	█	█	█	█	█	█	█
$e = d + a$	█	█	█	█	█	█	█
$f = b + c$	█	█	█	█	█	█	█
$f = f + b$	█	█	█	█	█	█	█
if $e == 0$ goto _L0	█	█	█	█	█	█	█
$d = e + f$	█	█	█	█	█	█	█
goto _L1	█	█	█	█	█	█	█
_L0: $d = e - f$	█	█	█	█	█	█	█
_L1: $g = d$	█	█	█	█	█	█	█

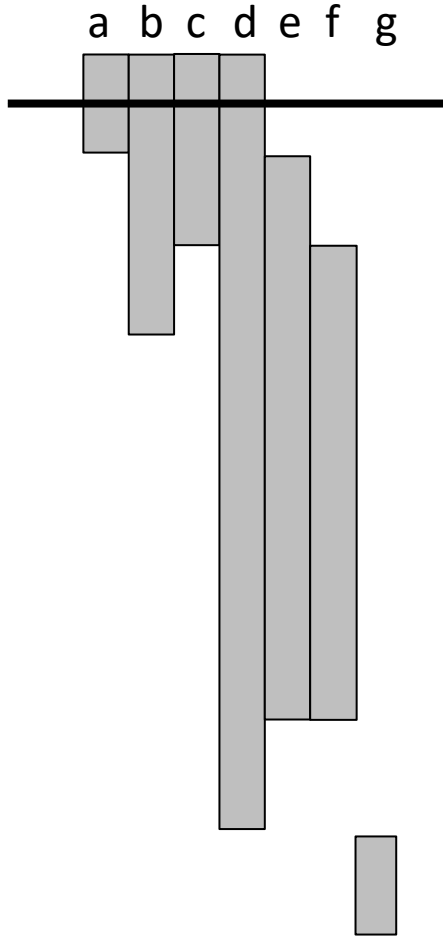


# Register Allocation with Live Intervals

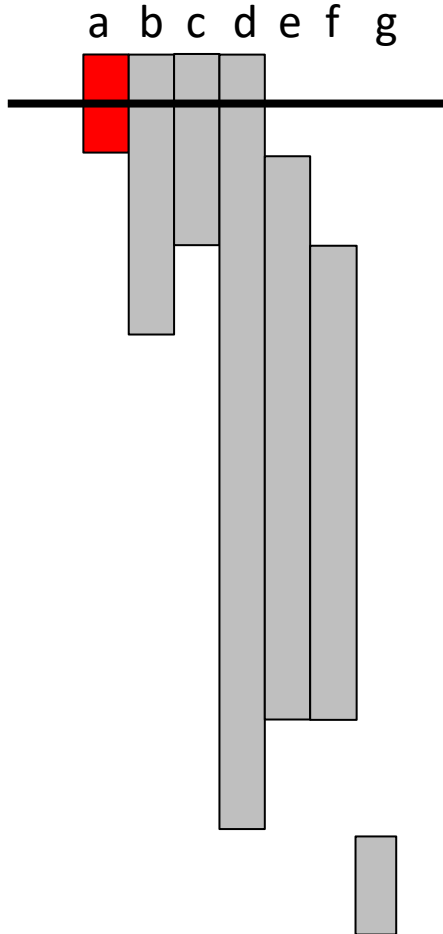
- Given the live intervals for all the variables in the program, we can allocate registers using a simple greedy algorithm.
- Idea: Track which registers are free at each point.
- When a live interval begins, give that variable a free register.
- When a live interval ends, the register is once again free.



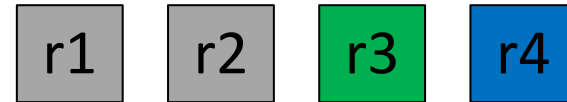
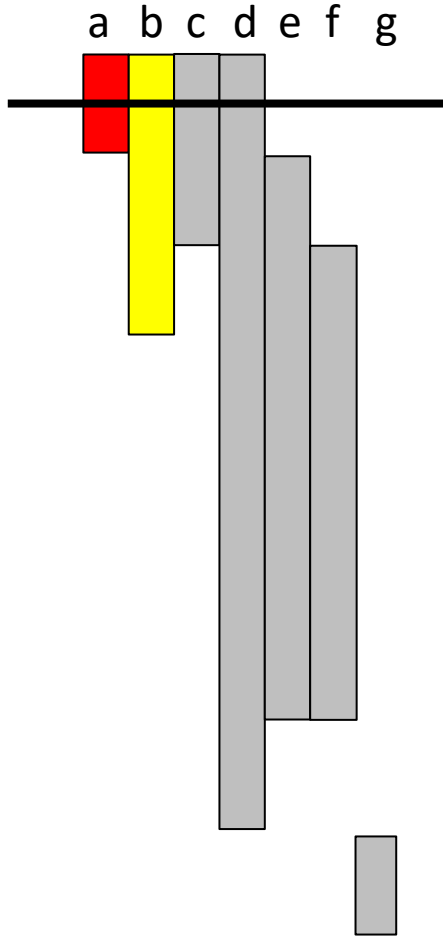
# Register Allocation with Live Intervals



# Register Allocation with Live Intervals

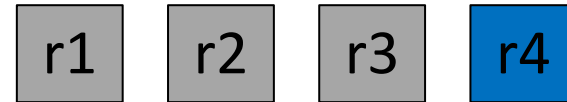
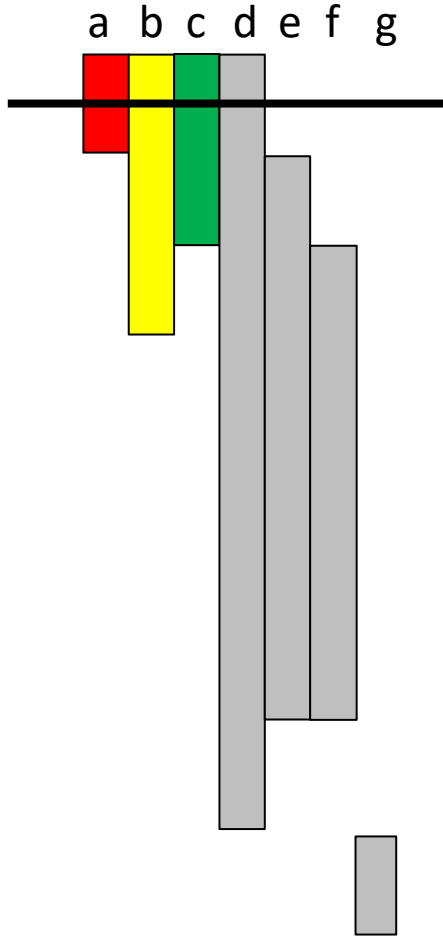


# Register Allocation with Live Intervals

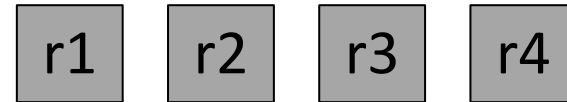
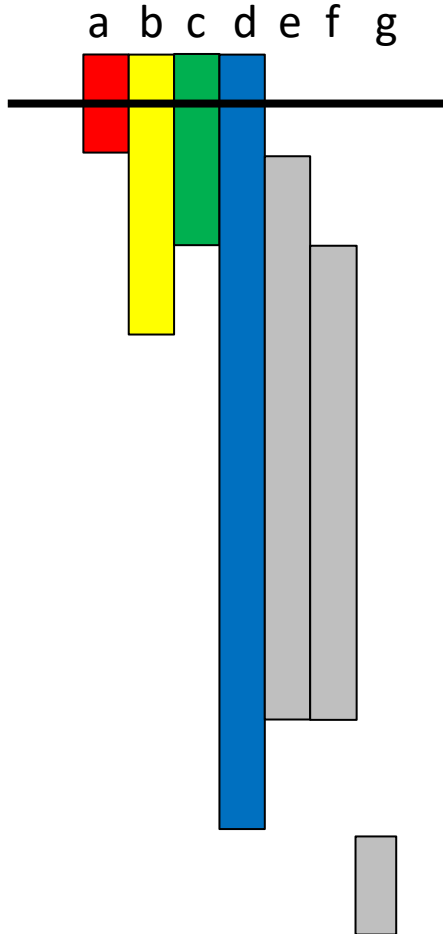




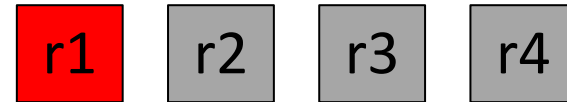
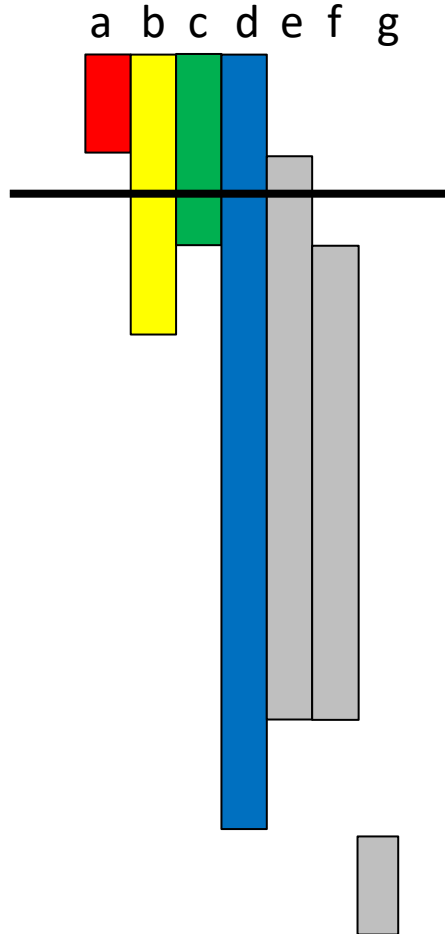
# Register Allocation with Live Intervals



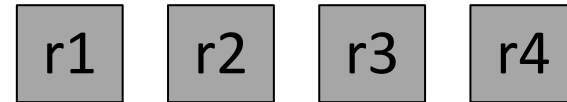
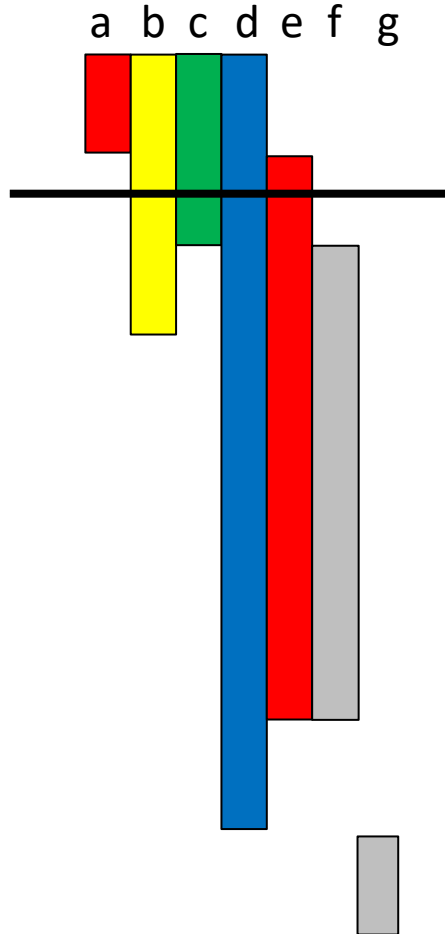
# Register Allocation with Live Intervals



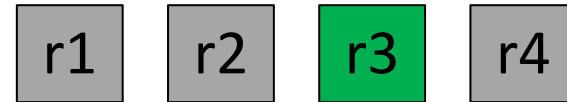
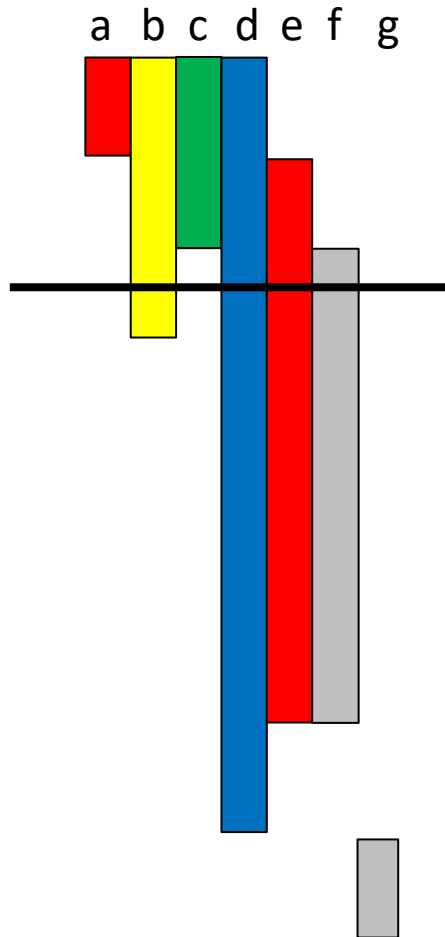
# Register Allocation with Live Intervals



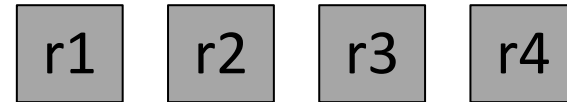
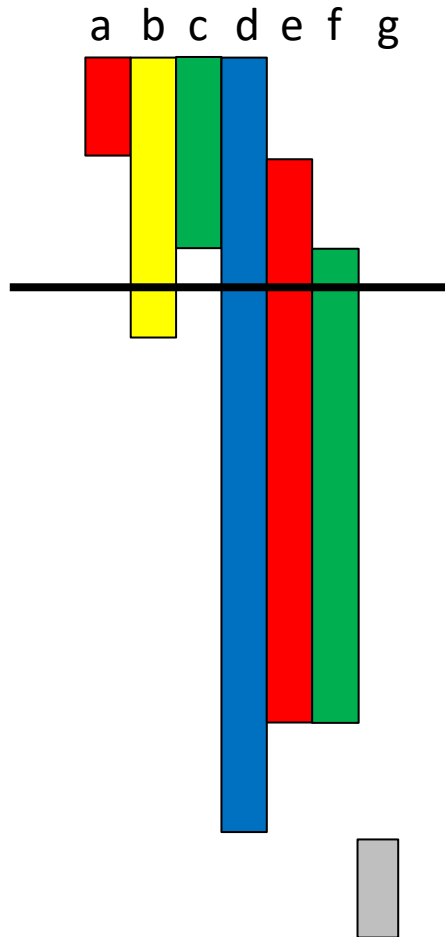
# Register Allocation with Live Intervals



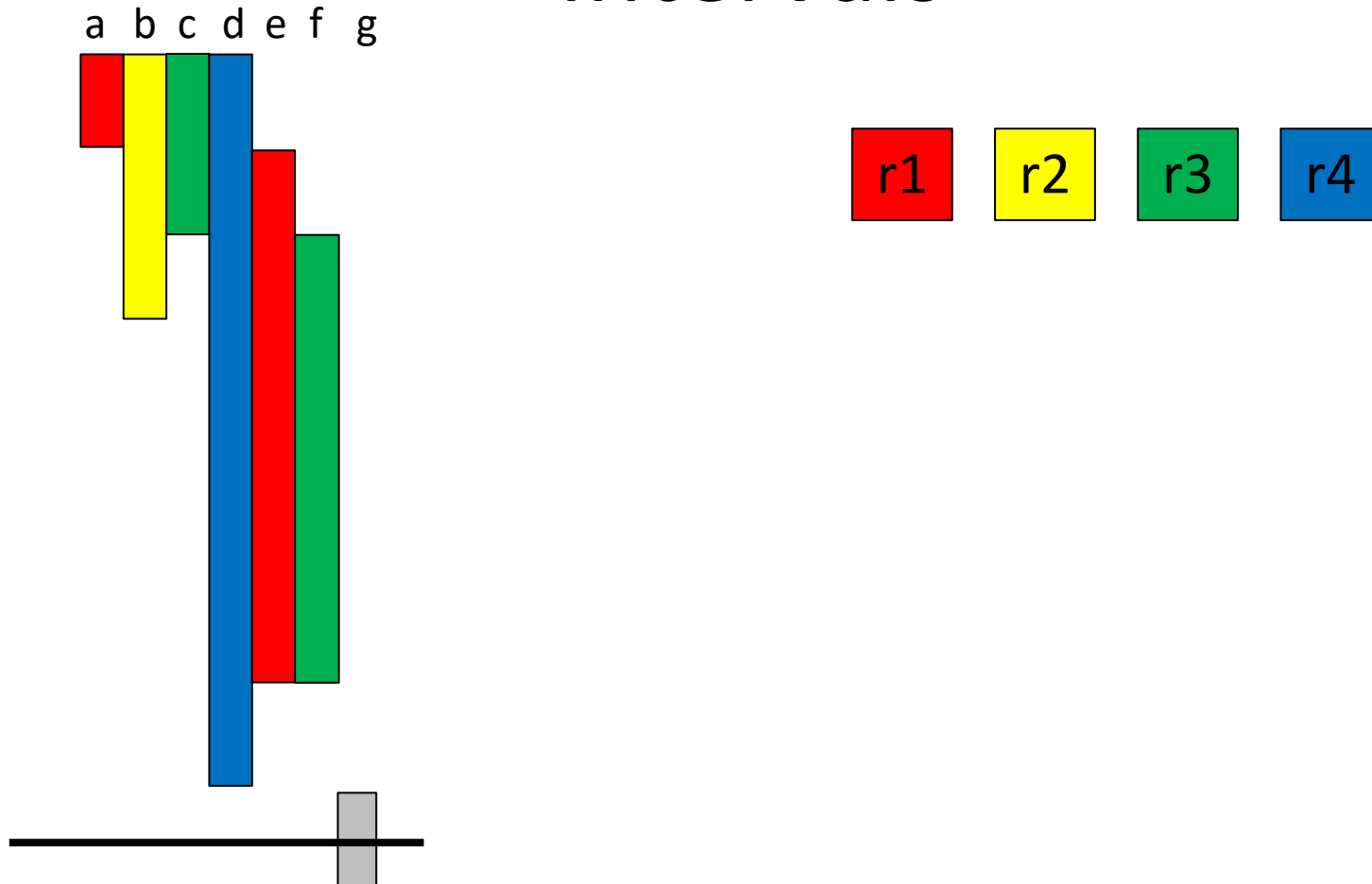
# Register Allocation with Live Intervals



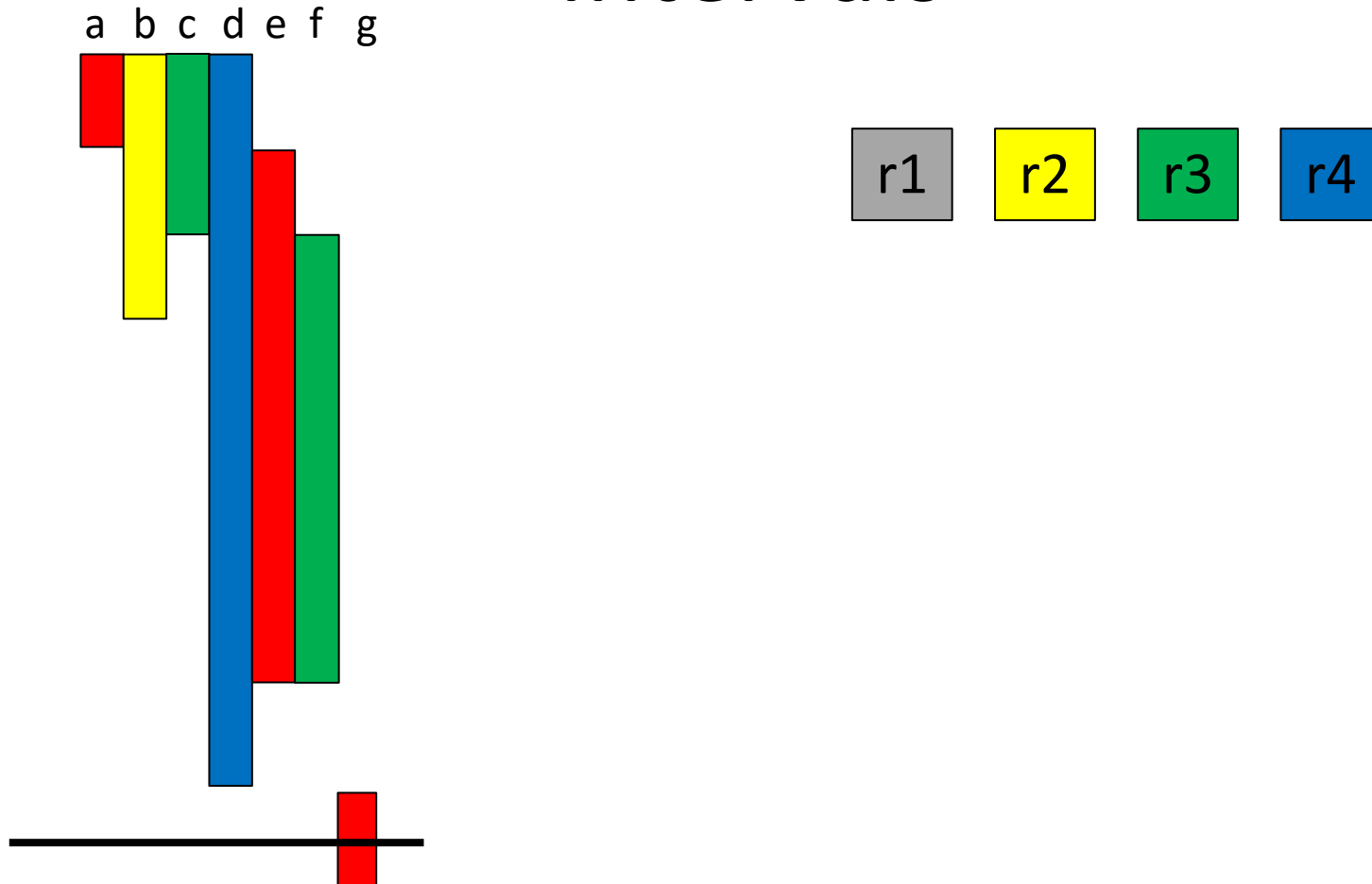
# Register Allocation with Live Intervals



# Register Allocation with Live Intervals



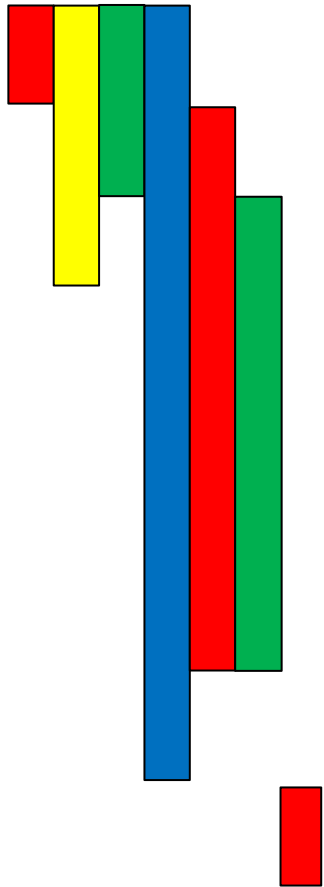
# Register Allocation with Live Intervals





# Register Allocation with Live Intervals

a b c d e f g



# Linear Scan Register Allocation

- If a register cannot be found for a variable  $v$ , we may need to spill a variable.
- This algorithm is called linear scan register allocation and is a comparatively new algorithm.
- Pros:
  - Very efficient
  - Works well in many cases
  - Allocation needs one pass, the code can be generated simultaneously
  - Used in JIT compilers like Java HotSpot
- Cons:
  - Not as good as graph coloring approach

# Summary

- Register allocation is a “must have” in compilers, because:
  - Intermediate code uses too many temporaries
  - It makes a big difference in performance
- The liveness at each location can be used for register allocation
- Register allocation as heuristic graph coloring uses live ranges
  - The basis for the technique used in GCC
- Linear scan register allocation uses live intervals
  - Often used in JIT compilers due to efficiency