Basic Blocks, Next-Use, Liveness & Register Allocation

Prof. James L. Frankel
Harvard University

Version of 6:09 PM 30-Mar-2022

Identifying Basic Blocks

- A basic block is a maximum length sequence of instructions that must always be executed without interruption
  - This implies that the only entry point into the block is at the beginning
  - This implies that the only exit point from the block is at the end

Basic Blocks Algorithm (Finding Leaders)

- Apply the following algorithm to a list of IR instructions generated for a module
- First, identify **leaders**
  - A leader is an IR instruction that begins a basic block

① The first IR instruction is a leader

② Any target of a conditional or unconditional branch/jump is a leader

③ Any IR instruction immediately after a conditional or unconditional branch/jump is a leader

Basic Blocks Algorithm (Finding Basic Blocks)

- After finding all leaders, identify **basic blocks**

For each leader, its basic block consists of itself & all IR instructions up to but not including the next leader or the end of the IR program

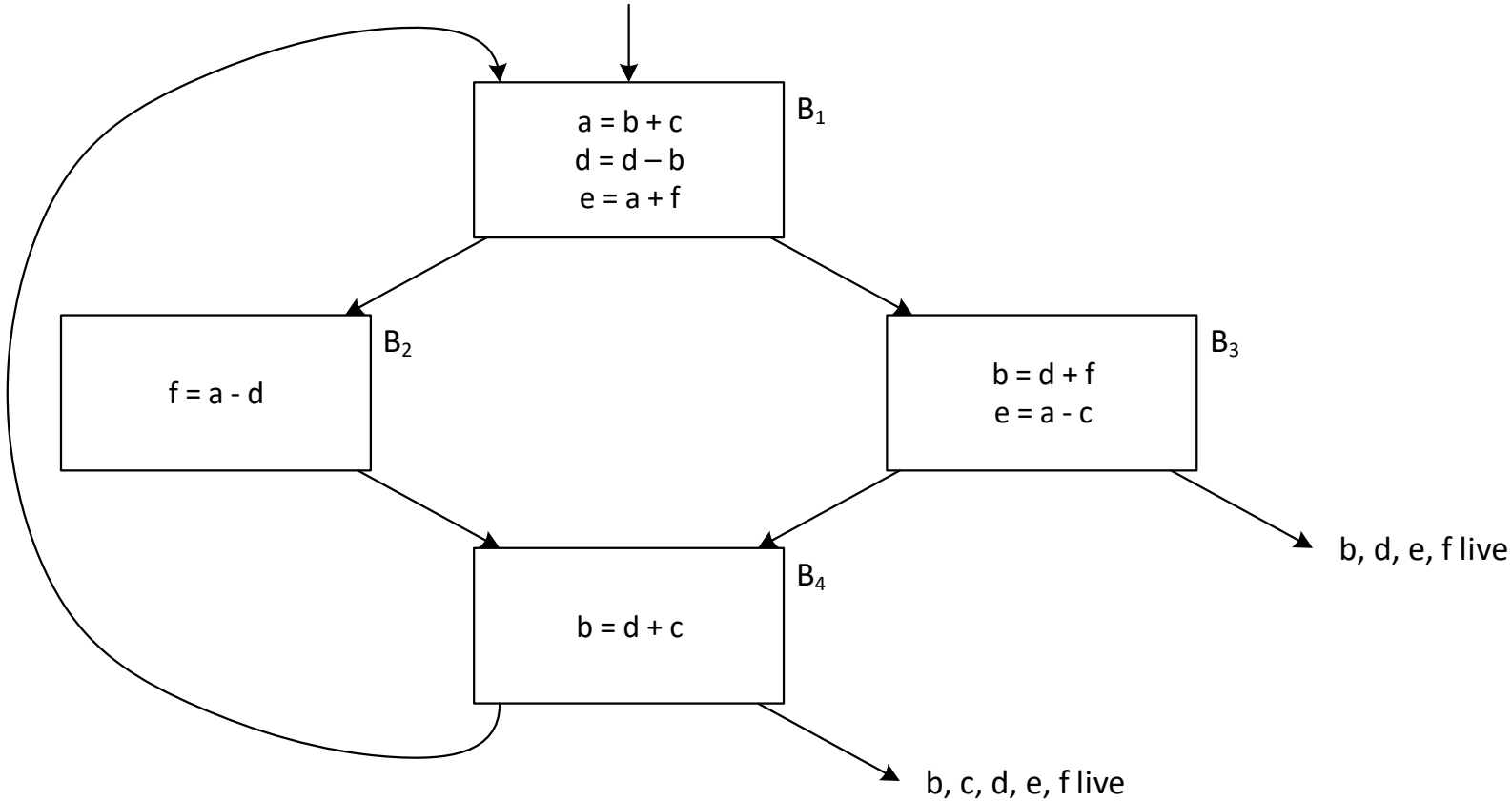
Next-Use & Liveness Information

- We examine each basic block separately
- Initially, all non-temp variables in basic block *B* are labeled as **live** on exit with their **next-use** after the block
- Progress from the last IR statement to the first IR statement in the block
- At each statement *i*: **x = y ⊗ z** in *B*, where ⊗ represents any operation and where *i* identifies this statement
  - First, attach to statement *i* the information about next use & liveness for *x*, *y* & *z*
  - Then, set *x* to “not live” & “no next use”
  - Finally, set *y* & *z* to “live” & set the next uses of *y* & *z* to *i*
  - The order of these three steps is important
    - Associate the information about *x*, *y* & *z* with *i* before changing their liveness & next-use information
    - In case a variable appears on both the left- and right-hand sides, always set the information for the lhs variable before setting the information for the rhs variable

Liveness Across Basic Blocks

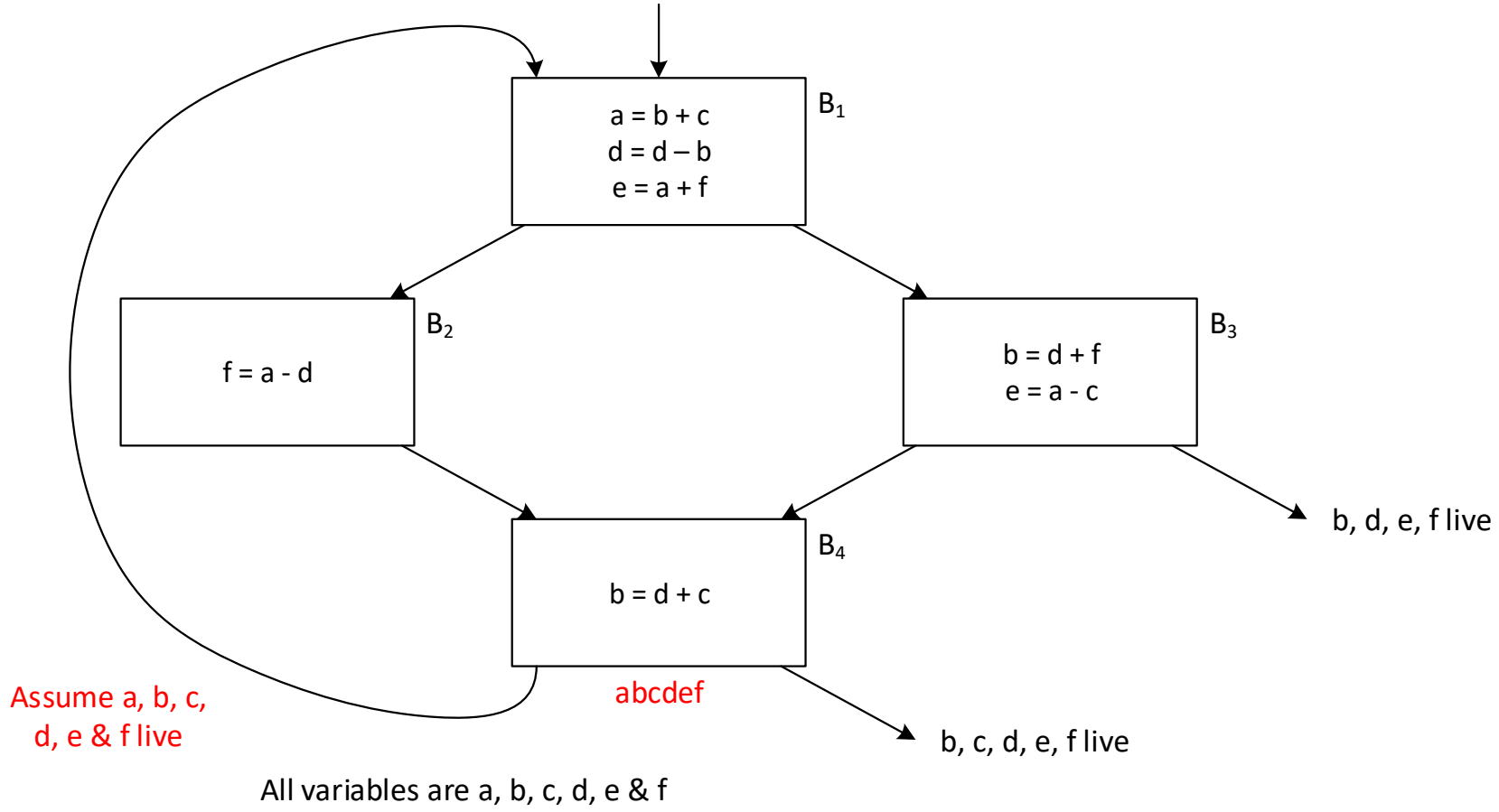
- The set of variables that are live at the end of a basic block is the union of those live at the beginning of each of its successor blocks

Flow Graph with Initial Constraints

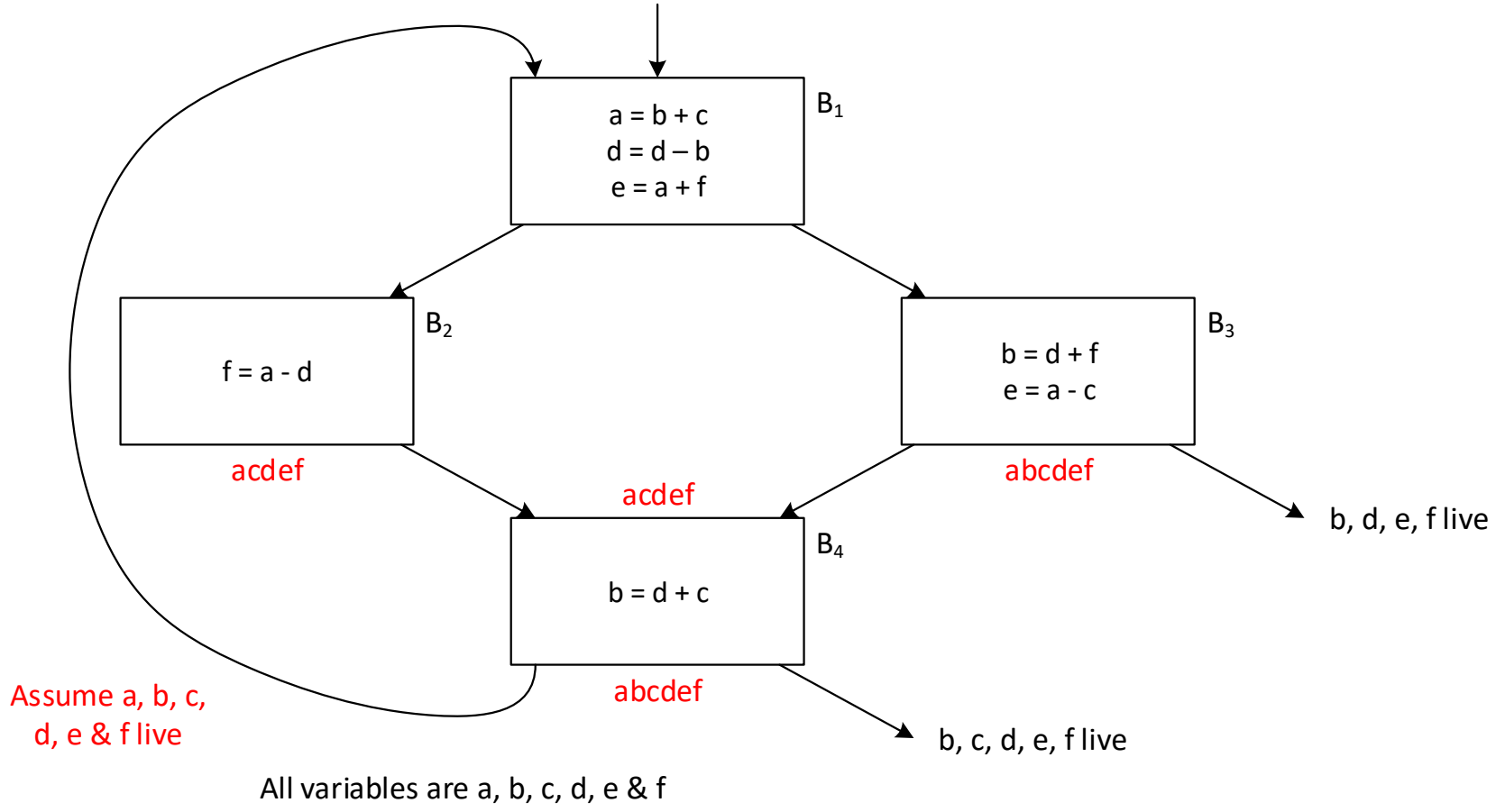


All variables are a, b, c, d, e & f

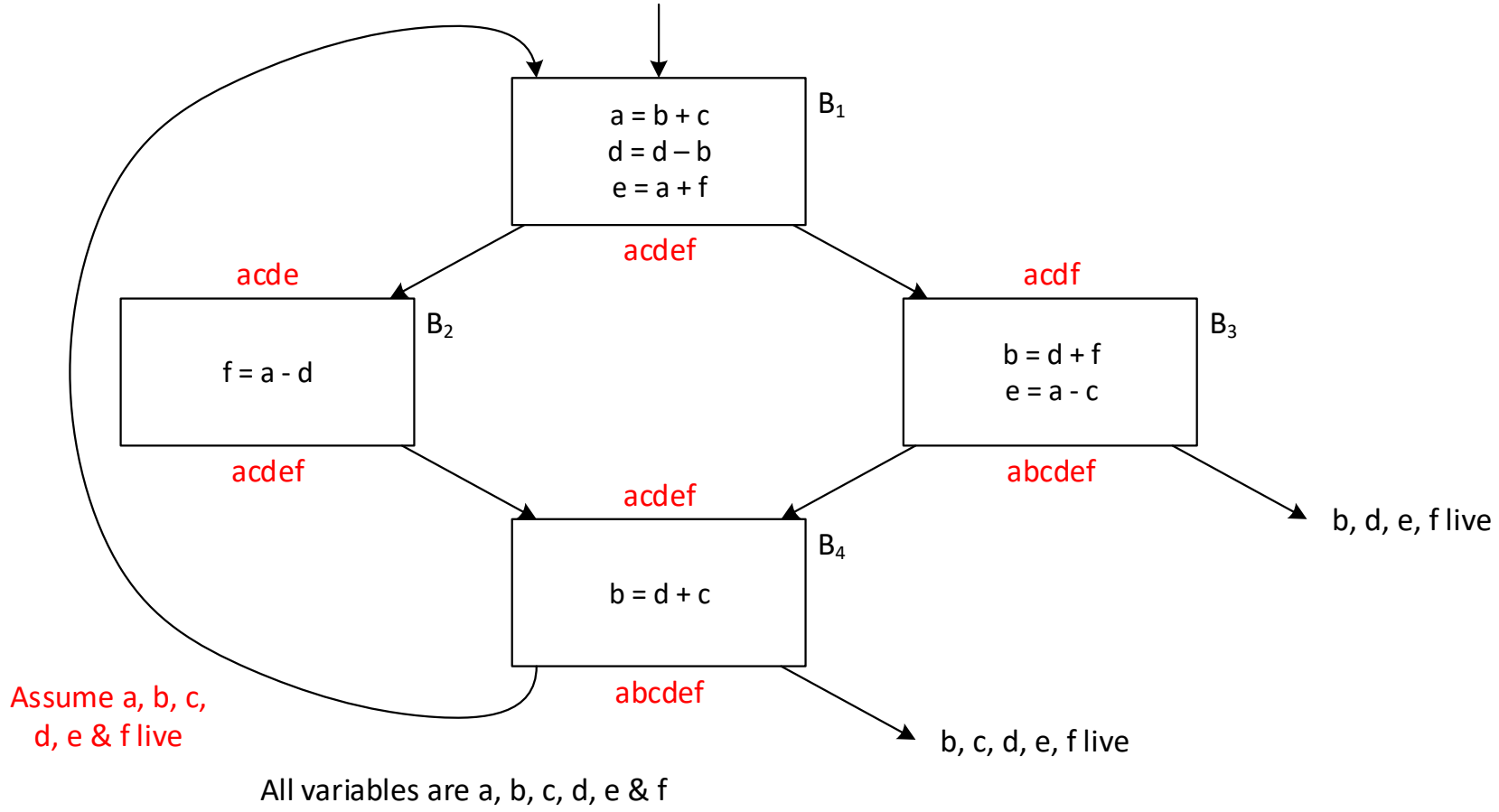
Establish Liveness on Exit from Basic Block B_4



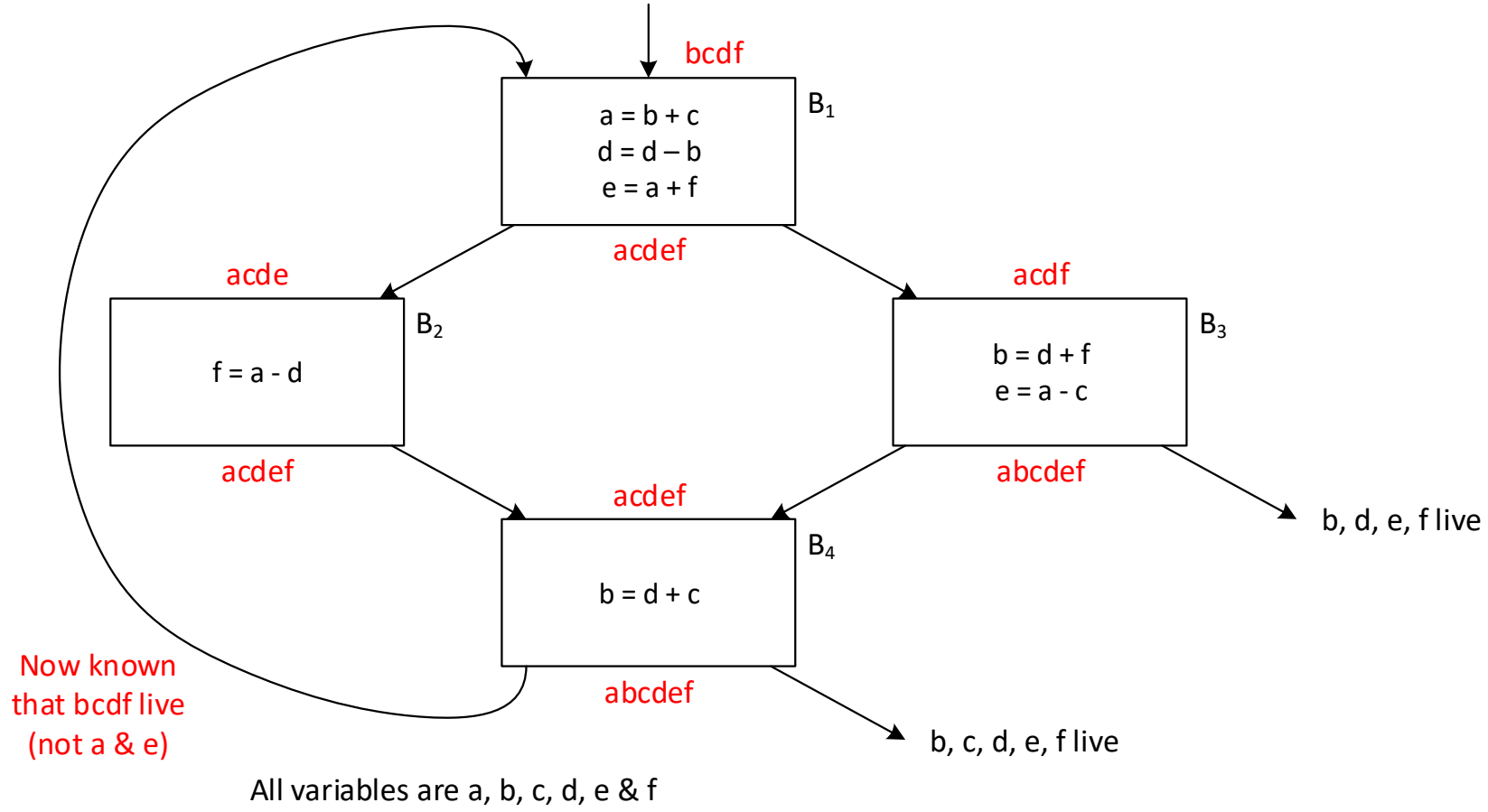
Establish Liveness on Entry to Basic Block B_4 and on Exit from Basic Blocks B_2 and B_3



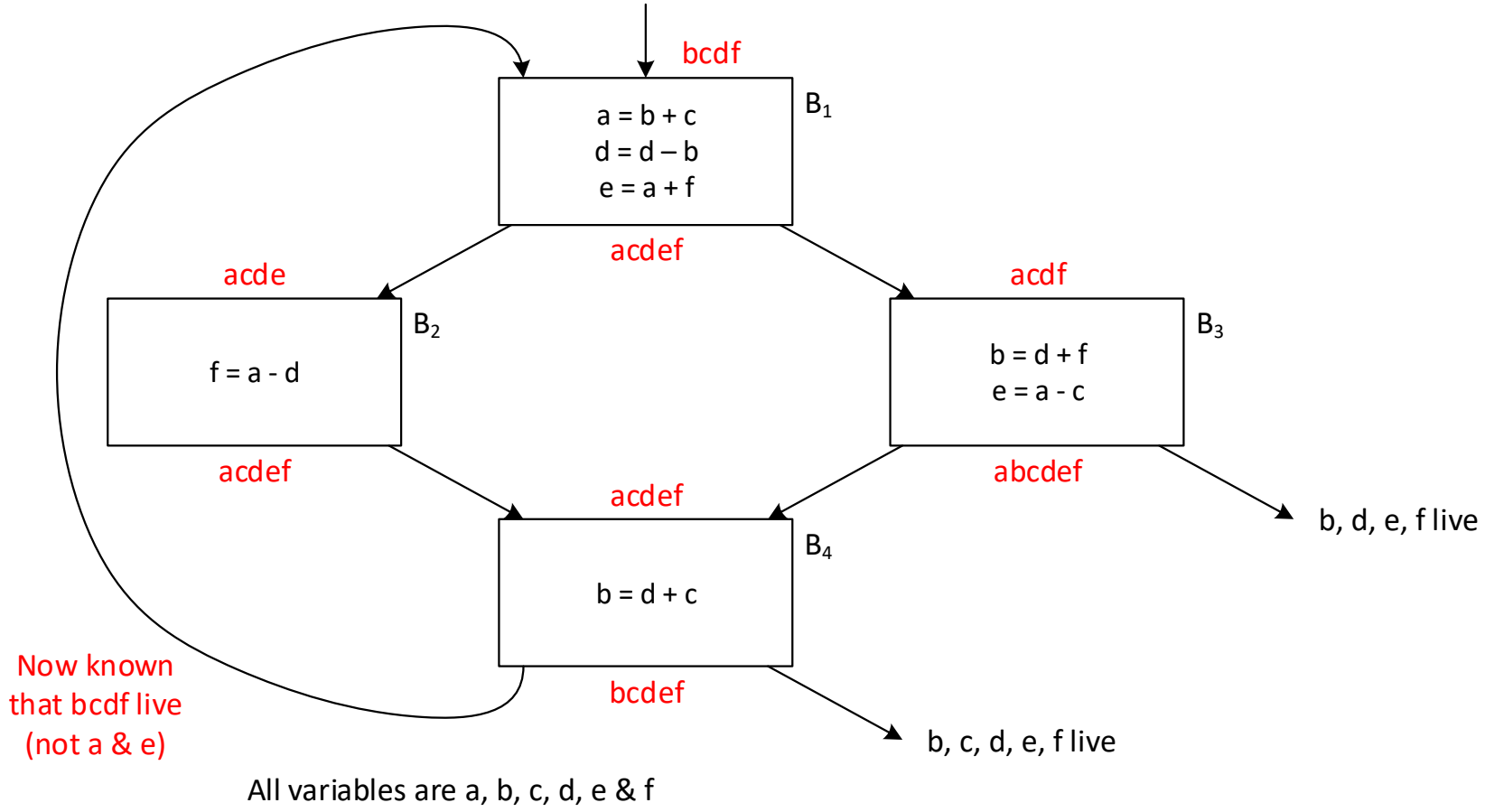
Establish Liveness on Entry to Basic Blocks B_2 and B_3 and on Exit from Basic Block B_1



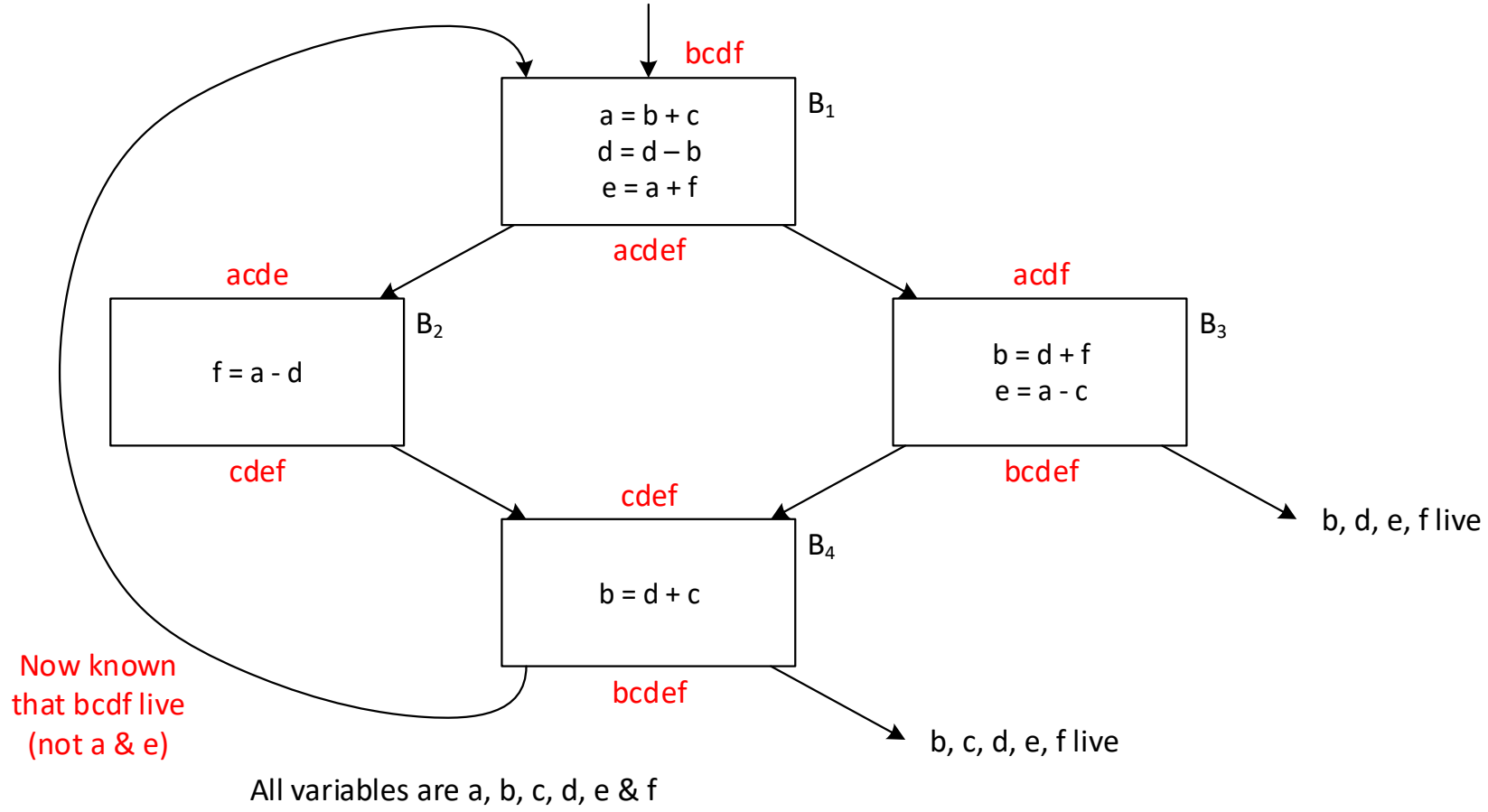
Establish Liveness on Entry to Basic Block B_1 and feed that information back to Block B_4



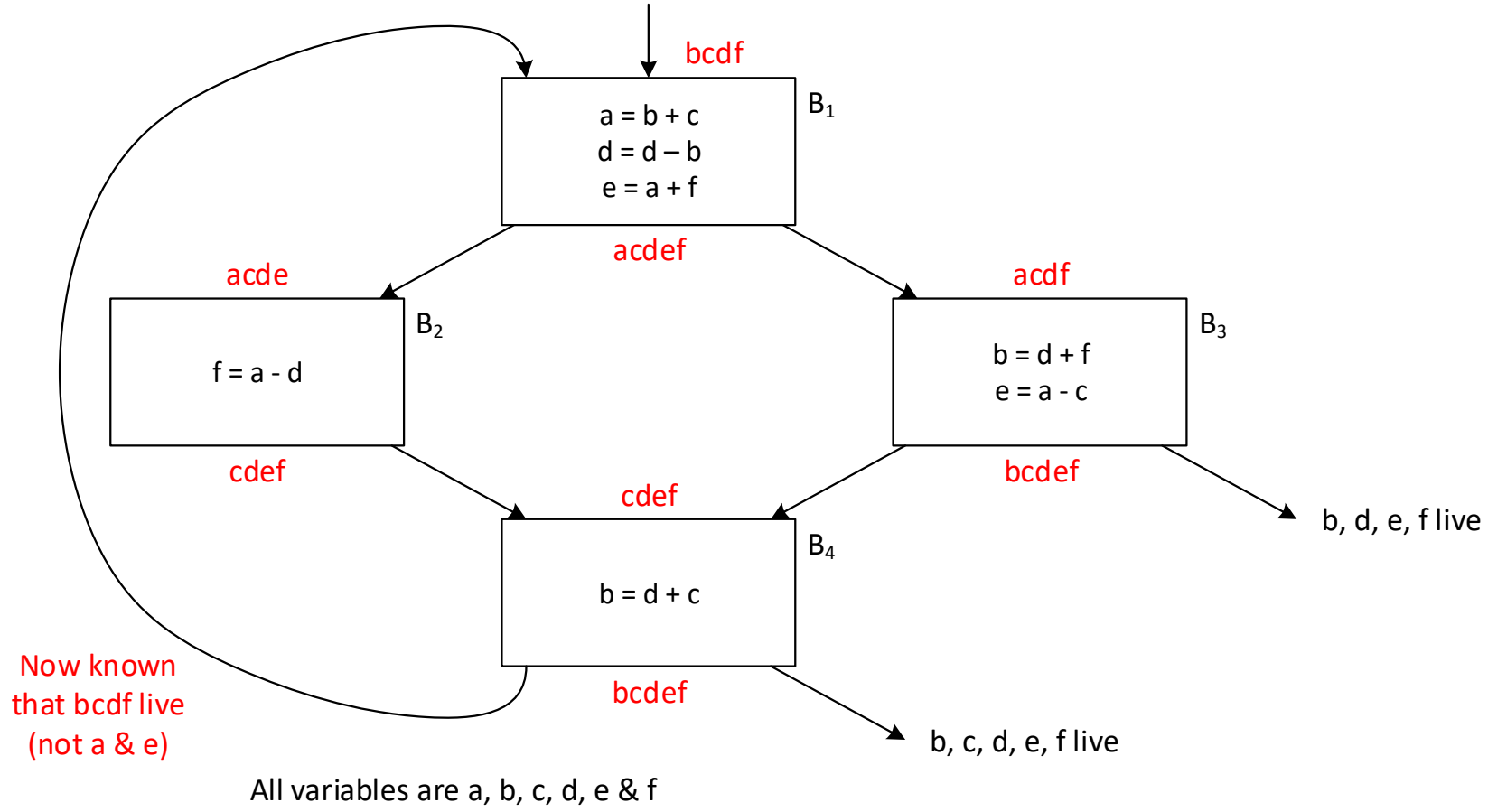
Update Liveness on Exit from Basic Block B_4



Update Liveness on Entry to Basic Block B_4 and on Exit from Basic Blocks B_2 and B_3



Update Liveness on Entry to Basic Blocks B_2 and B_3 – No Changes to Liveness Discovered



Liveness Across Basic Blocks in our Unoptimized Code

- *In our compilers and before any optimizations are performed*, our temporaries/registers are never live across *basic blocks*
  - We never use a temporary/register from one statement to another
  - We are using this behavior so that we are able to reset the use of temporaries/registers at the start of each new statement
  - Therefore, *in our compilers and before any optimizations are performed*, temporaries/registers are never live across *statements*
- Certain optimizations may cause temporaries/registers to be live across statements within a basic block
  - Unless you perform optimizations across basic blocks, temporaries/registers will not be live across basic blocks

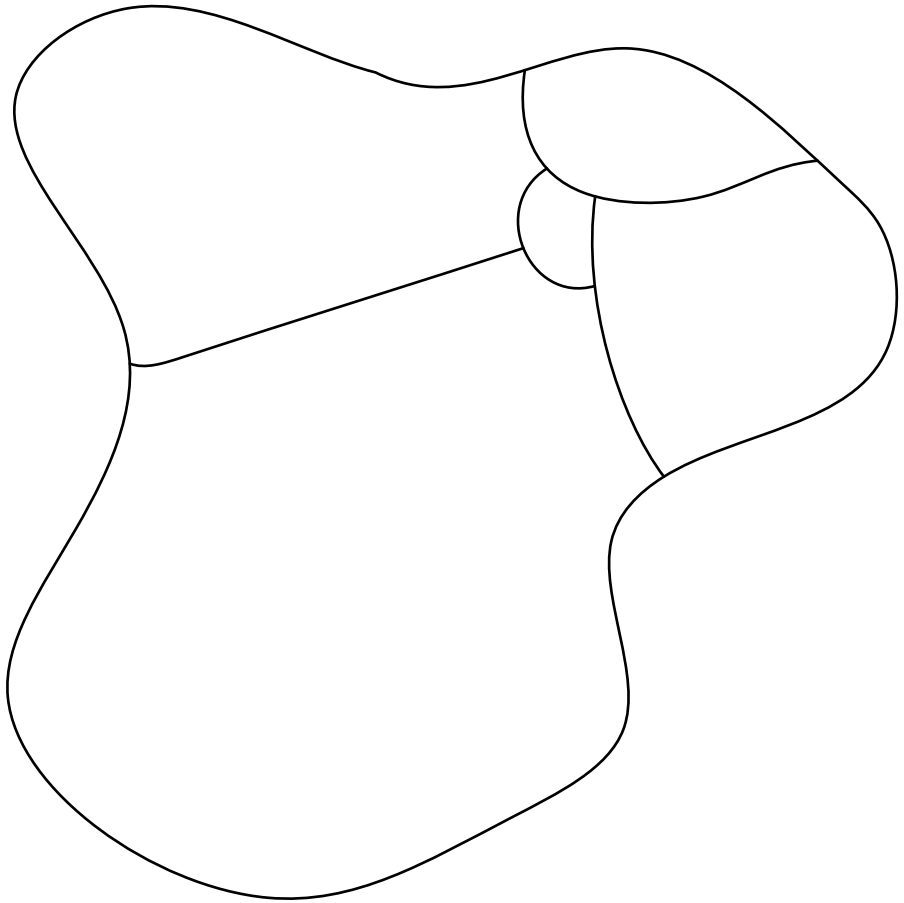
Graph Coloring

- For an arbitrary graph, how many colors are needed to color each node such that no two connected nodes are the same color?
  - We say that a graph is *k*-Colorable, where *k* is the minimum number of colors to colorize such a graph
- It can be proven that this problem is NP-Complete
  - That is, in general it *cannot* be solved in polynomial time
  - This is a member of a set of problems that are very difficult to solve including the knapsack problem – packing 3D objects into a defined volume
- However, for many real-world problems, there are heuristics that can solve these problems in quite reasonable time

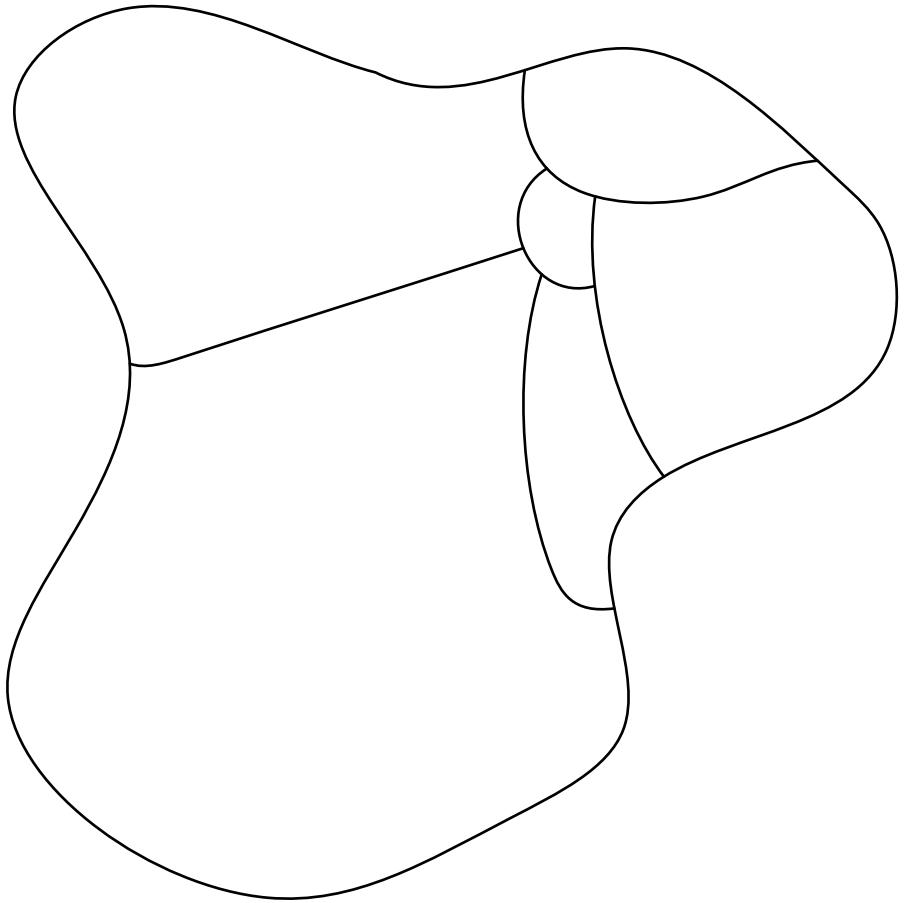
Graph Coloring for Planar Graphs

- If a graph is planar – that is, it can be mapped onto a planar (2D) surface with no crossing lines – how many colors are needed for such a graph?
- Problem was mentioned by Möbius as early as 1840
- Of course, planar graphs include our usual 2D maps in, say, a Mercator projection
- Imagine a Mercator projection representing a continent on the Earth
  - Each nation would be a node
    - Each nation must be contiguous
  - Borders would be connecting arcs
  - No two nations sharing a border would have the same color

Planar Graphs – 3-Colorable



Planar Graphs – 4-Colorable



Four Color Theorem for Planar Graphs

- The so-called Four Color Theorem showed that any planar graph is 4-Colorable
  - It was proven in 1976 by Kenneth Appel and Wolfgang Haken using a computer program that systematically identified all possible ways that nodes could be connected on a planar graph
  - 1,936 different maps were identified

Application of Graph Coloring to Register Allocation

- Initially assume an infinite number of registers/temporaries during code generation
- Create a graph in which nodes (representing registers/temporaries in a program fragment) are connected by an arc when the registers/temporaries are needed at the same time
- Then, the minimum number of colors to color such a graph would represent the minimum number of registers required for that program fragment
- Moreover, such a graph would identify the constraints on register assignment
- We call such a graph a **Register Interference Graph**

Register Interference Graph

- Each register is a node
- An edge connects two registers whenever one is live where the other is defined
  - For this example, we'll assume that all named identifiers are registers

- We'll examine basic block B_1 from above

```
a = b + c
d = d – b
e = a + f
```

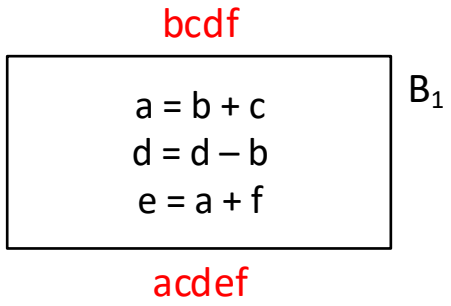
Register Interference Graph Coloring (1 of 2)

- Color graph with *k* colors, where *k* is the number of available registers
- No two adjacent nodes have the same color
- Color represents a register
- This problem is NP-Complete
- Heuristic: If a node has fewer than *k* edges, remove the node and its edges to simplify the graph and then try coloring the graph again
  - This works because, if a node has fewer than *k* edges we can always color that node with a different color from its neighbors and still use no more than k colors

Register Interference Graph Coloring (2 of 2)

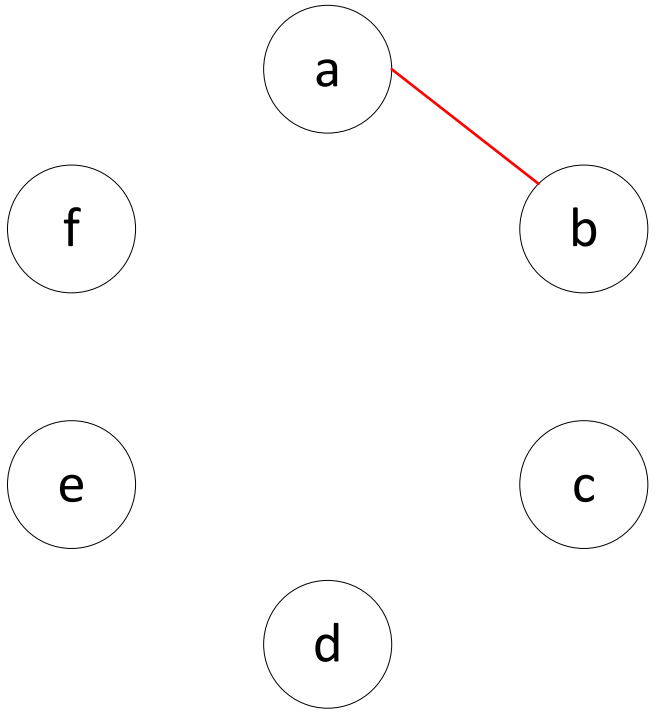
- Simplify the graph by removing all nodes with fewer than *k* edges and those nodes' edges
  - If the resultant graph is empty, then we're done!
  - Assign colors (*i.e.,* map temporaries into registers) in the opposite order
- If nodes still remain with *k* or more edges, introduce spill code to move the contents of registers into temporaries in memory and then to restore them when needed
  - Spilling a register into memory frees the register and breaks edges in the graph
  - Using memory is slow, so…
    - Avoid spills in inner loops

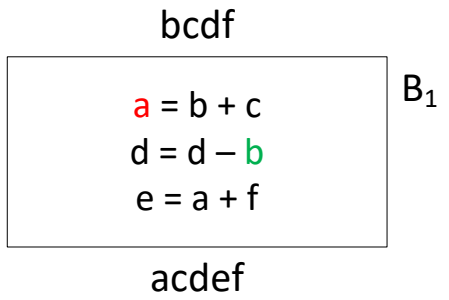
Basic Block B_1 Alone Including Liveness



Building Register Interference Graph (1 of 10)

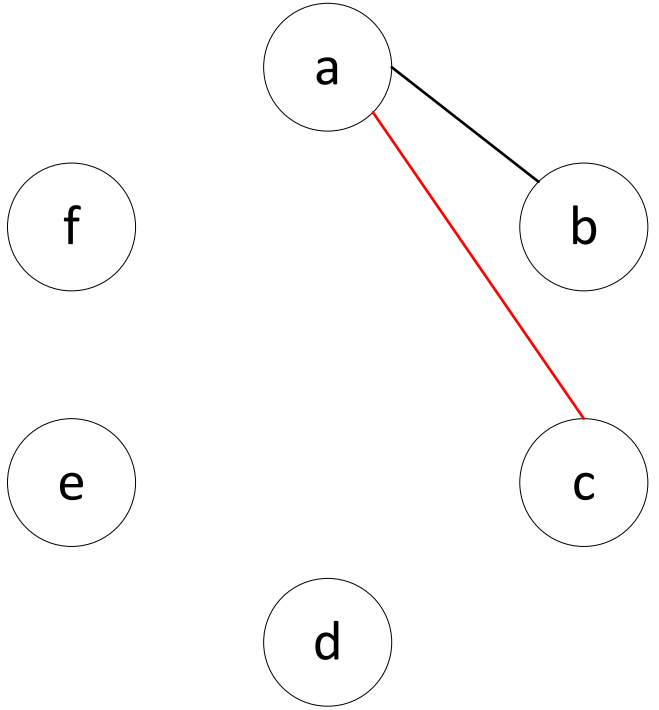
- b is live when a is defined

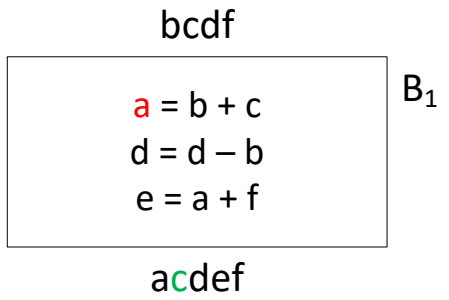




Building Register Interference Graph (2 of 10)

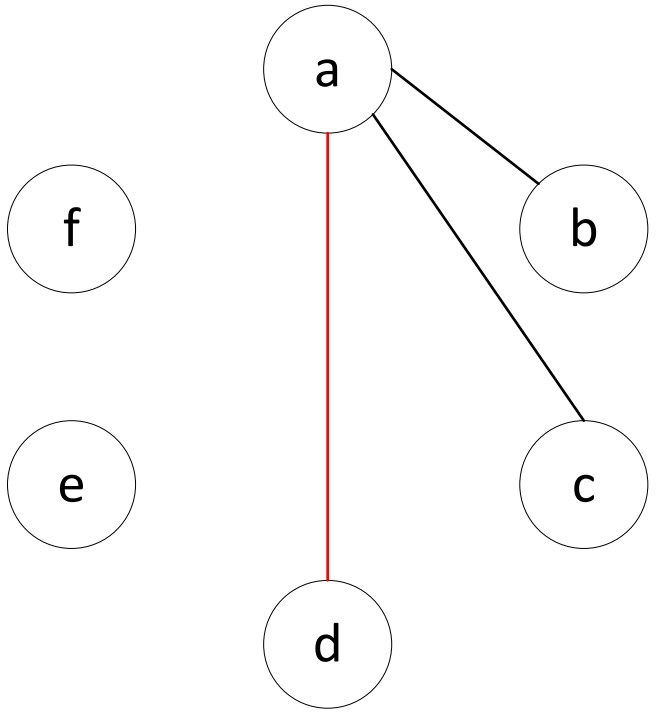
- c is live when a is defined

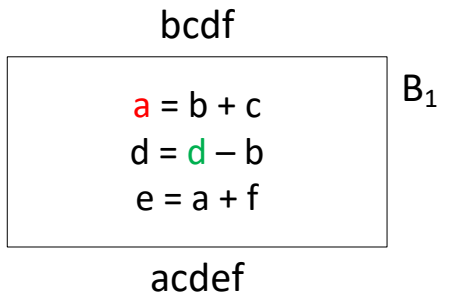




Building Register Interference Graph (3 of 10)

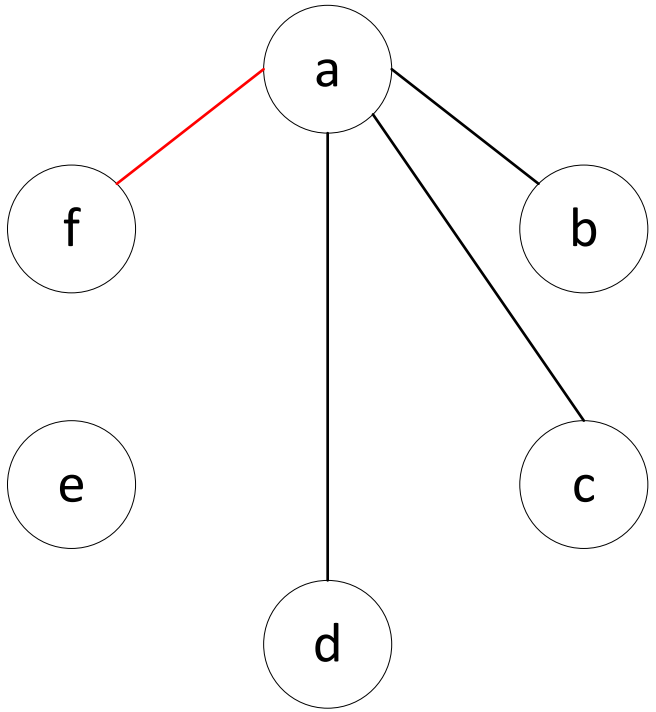
- d is live when a is defined





Building Register Interference Graph (4 of 10)

- f is live when a is defined



bcdf

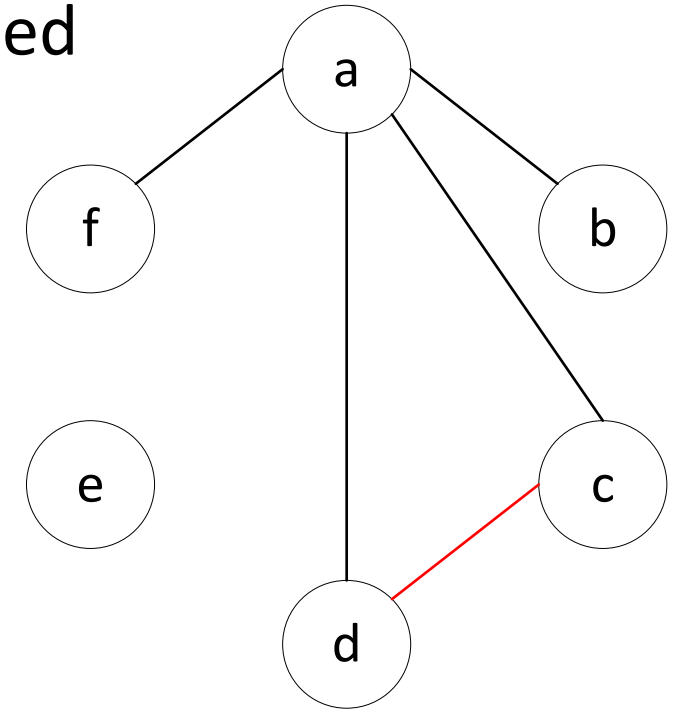
a = b + c
d = d – b
e = a + f

B_1

acdef

Building Register Interference Graph (5 of 10)

- a is live when d is defined
- c is live when d is defined



bcdf

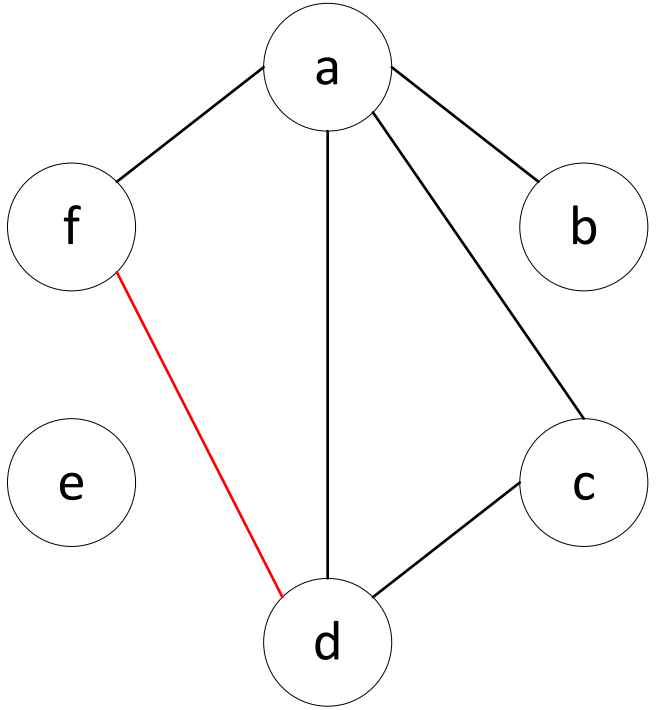
a = b + c
d = d – b
e = a + f

B_1

acdef

Building Register Interference Graph (6 of 10)

- f is live when d is defined



bcdf

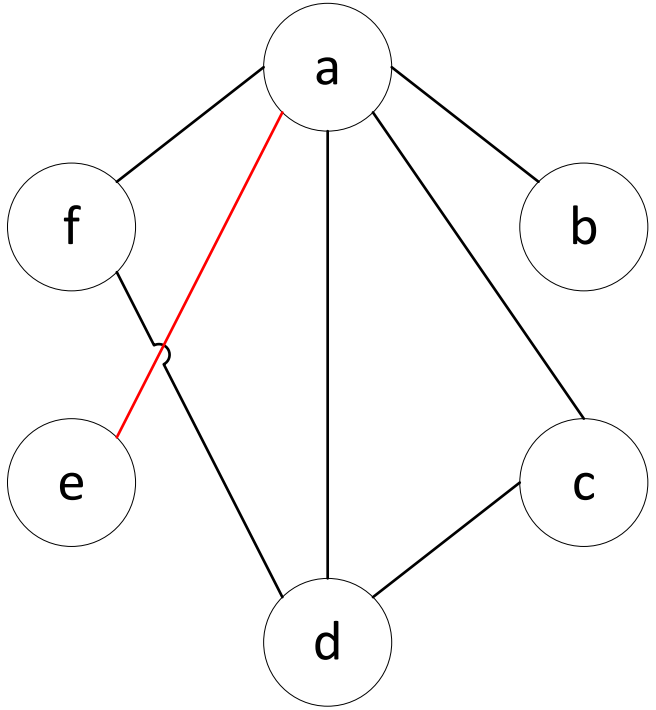
a = b + c
d = d – b
e = a + f

B_1

acdef

Building Register Interference Graph (7 of 10)

- a is live when e is defined



bcdf

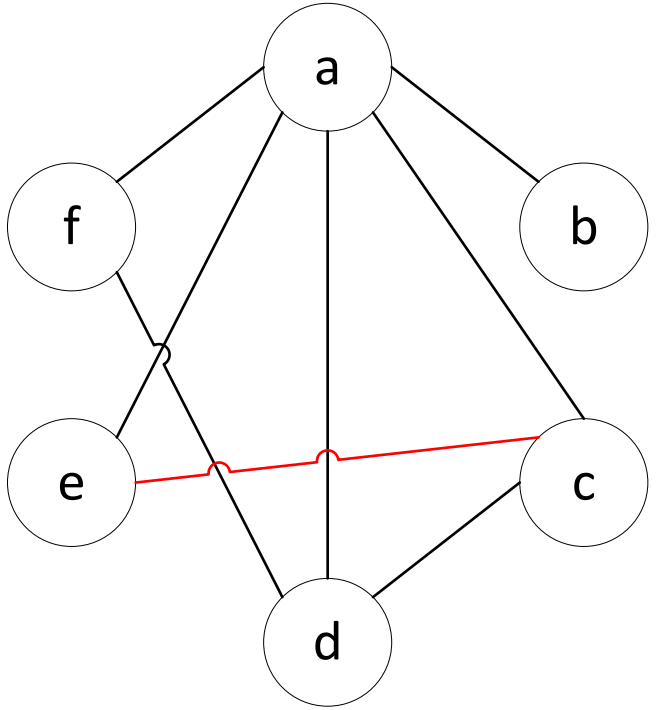
a = b + c
d = d – b
e = a + f

B_1

acdef

Building Register Interference Graph (8 of 10)

- c is live when e is defined



bcdf

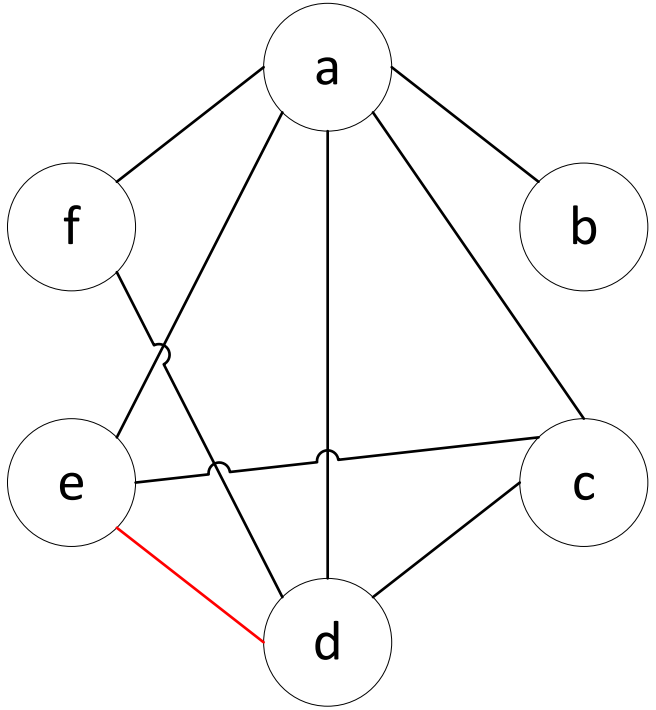
a = b + c
d = d – b
e = a + f

B_1

acdef

Building Register Interference Graph (9 of 10)

- d is live when e is defined



bcdf

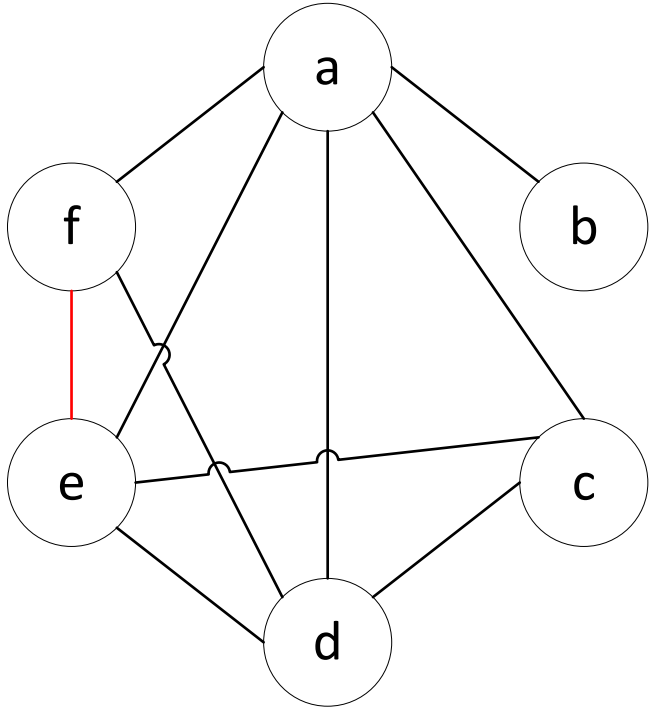
a = b + c
d = d – b
e = a + f

B_1

acdef

Building Register Interference Graph (10 of 10)

- f is live when e is defined



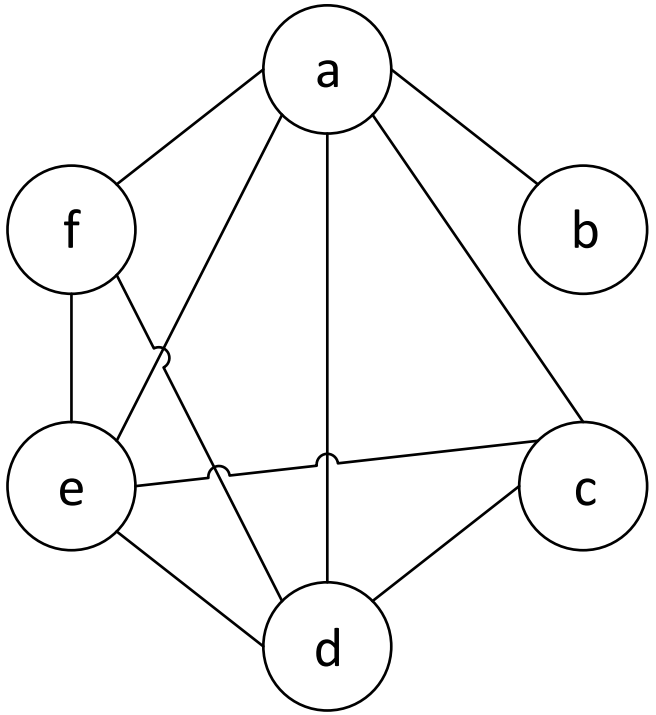
bcdf

a = b + c
d = d – b
e = a + f

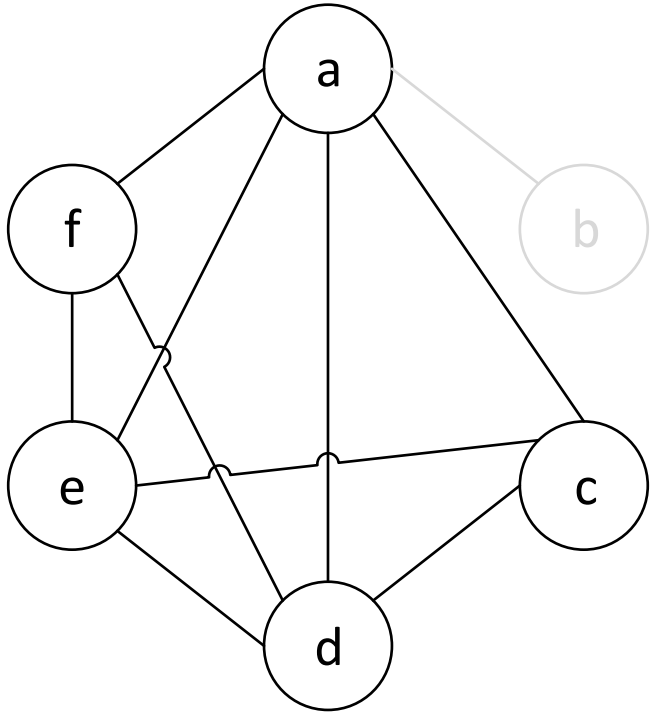
B_1

acdef

Register Interference Graph for Basic Block B_1

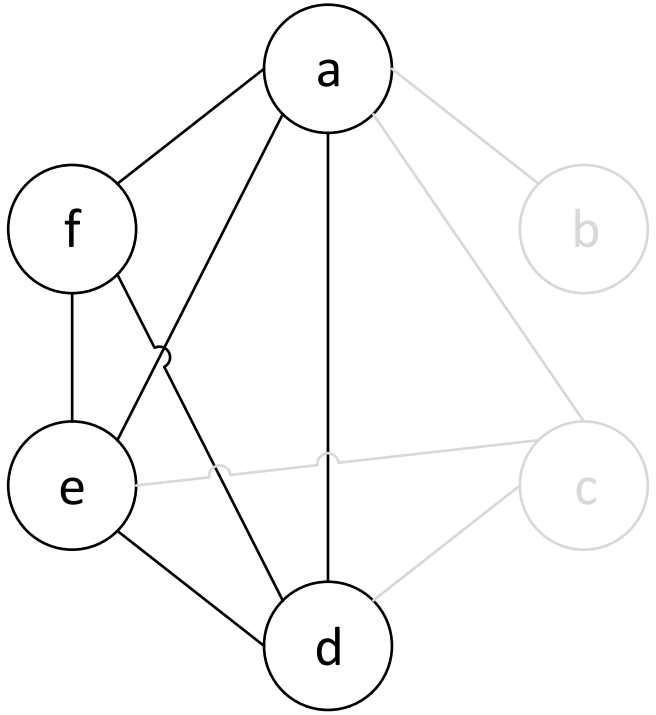


Node Removal (4-Colorable) (1 of 5)



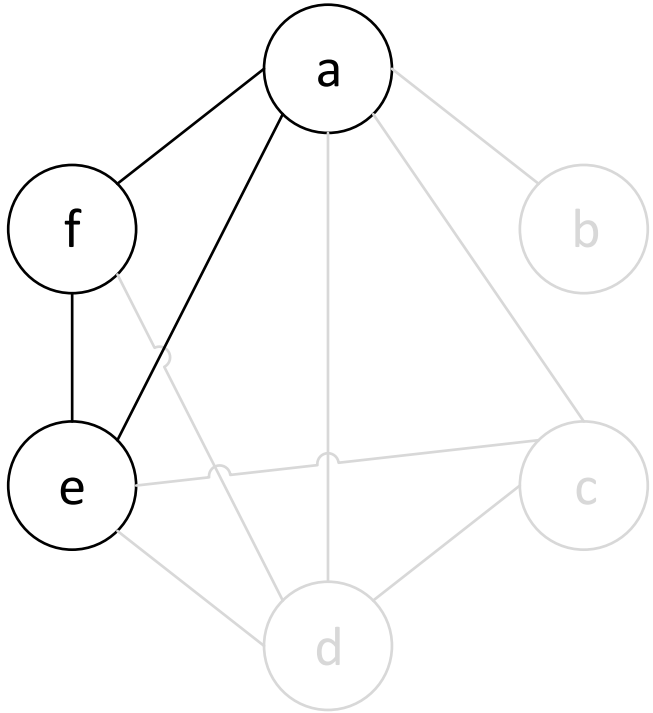
Remove Node b
(1 Edge < 4 Edges)

Node Removal (4-Colorable) (2 of 5)



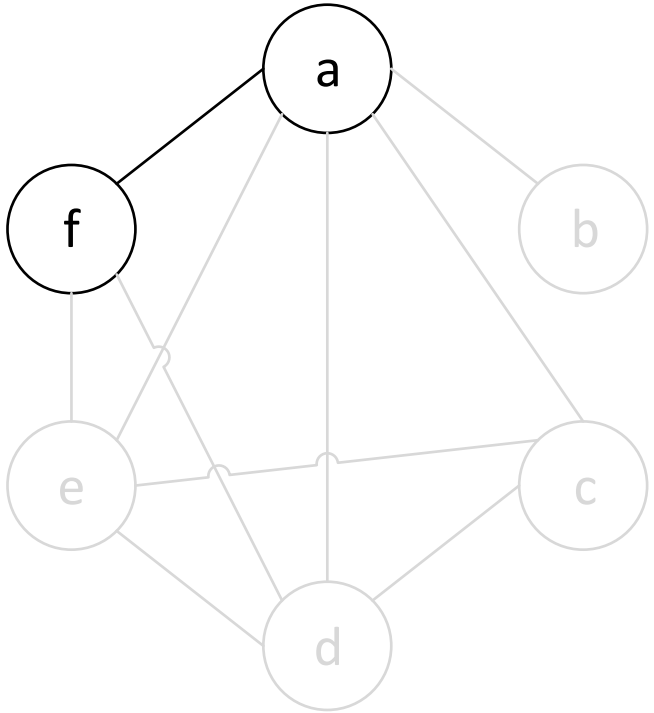
Remove Node c
(3 Edges < 4 Edges)

Node Removal (4-Colorable) (3 of 5)



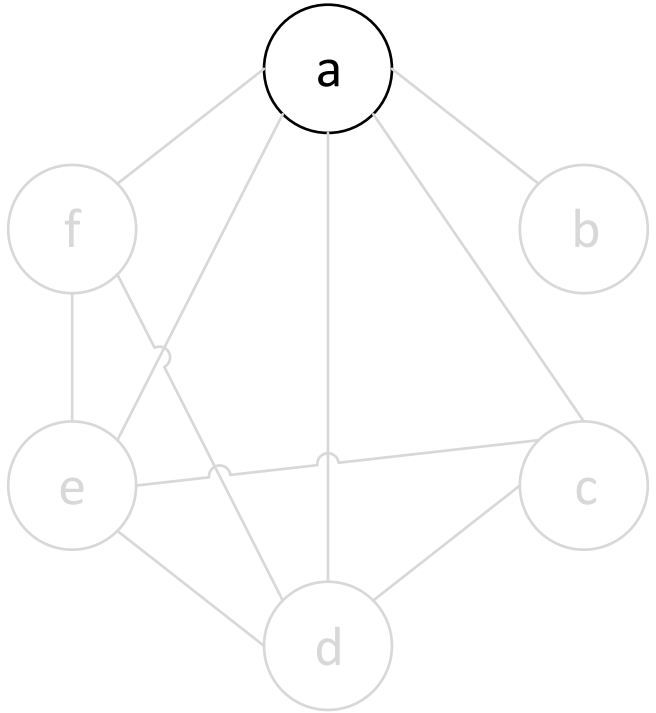
Remove Node d
(3 Edges < 4 Edges)

Node Removal (4-Colorable) (4 of 5)



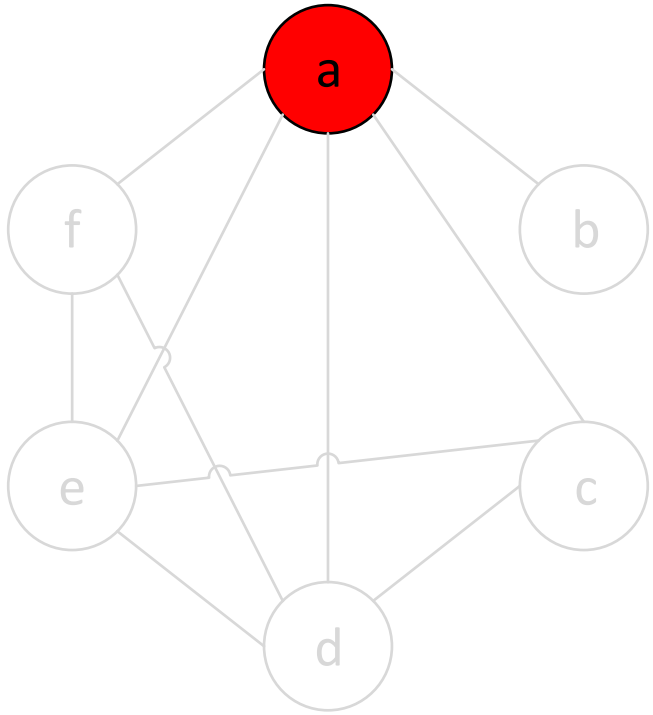
Remove Node e
(2 Edges < 4 Edges)

Node Removal (4-Colorable) (5 of 5)



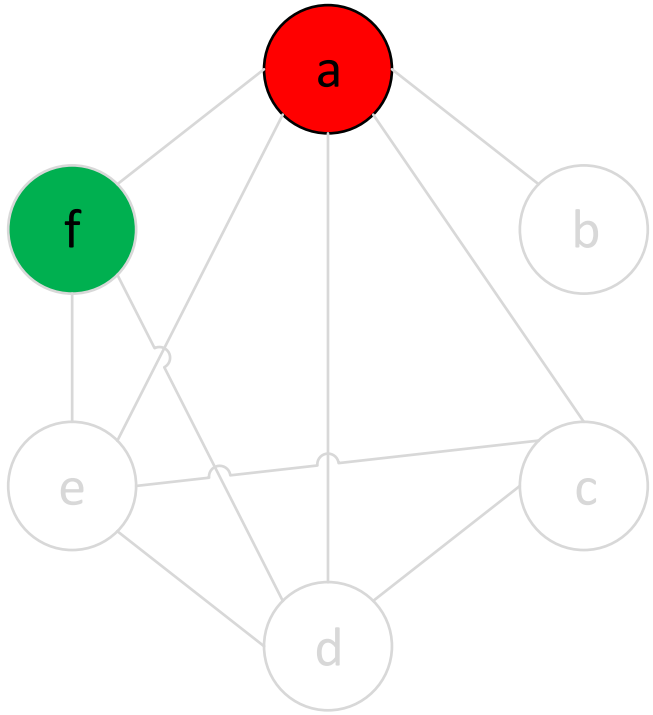
Remove Node f
(1 Edge < 4 Edges)

Node Coloring (4-Colorable) (1 of 6)



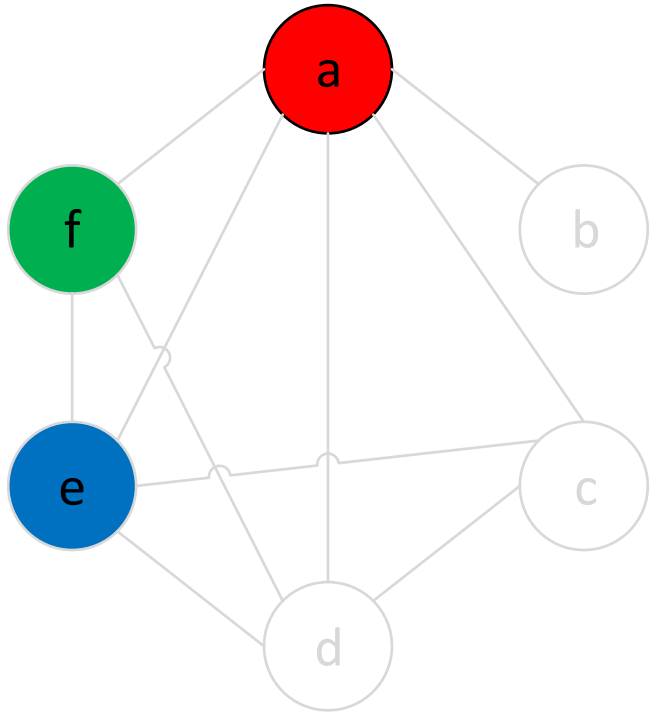
Color Node “a” Red (r0)

Node Coloring (4-Colorable) (2 of 6)



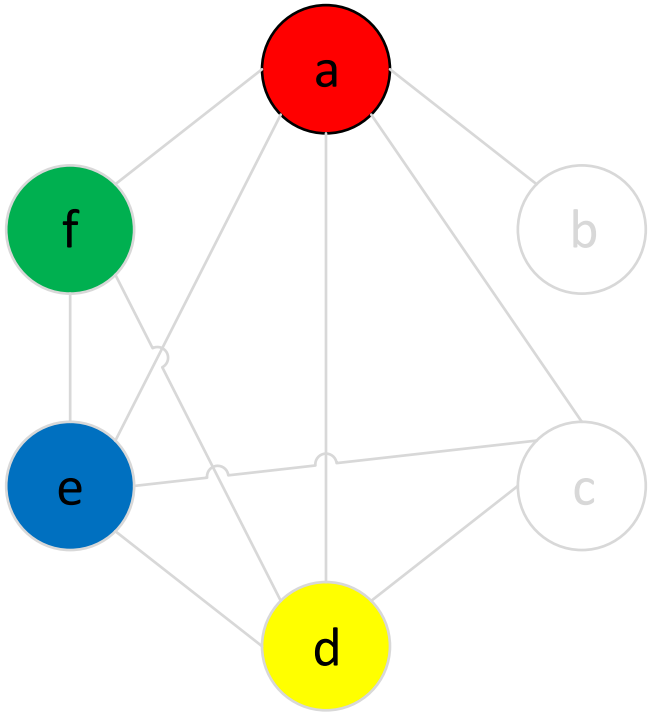
Color Node “f” Green (r1)

Node Coloring (4-Colorable) (3 of 6)



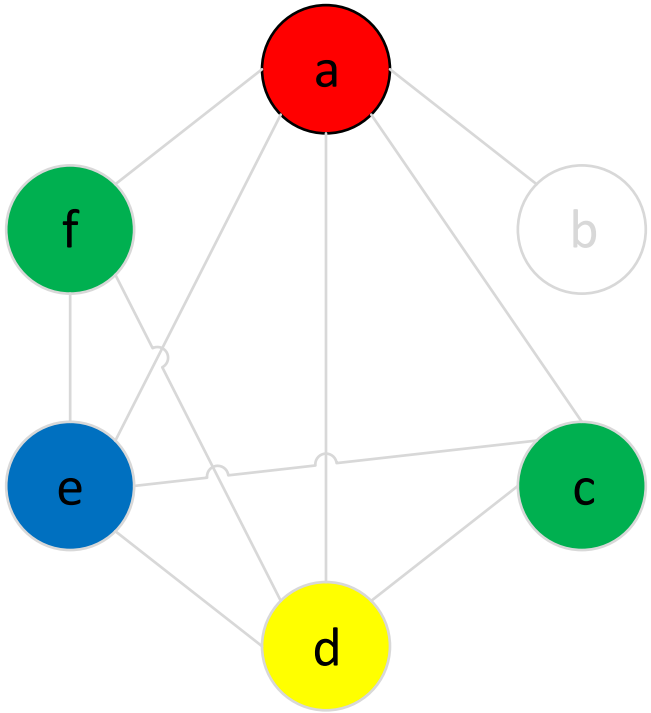
Color Node “e” Blue (r2)

Node Coloring (4-Colorable) (4 of 6)



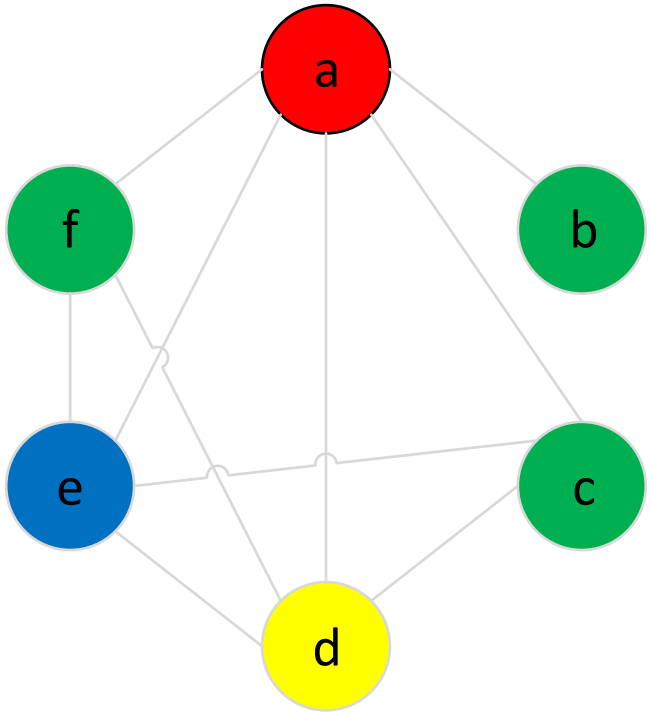
Color Node “d” Yellow (r3)

Node Coloring (4-Colorable) (5 of 6)



Color Node “c” Green (r1)

Node Coloring (4-Colorable) (6 of 6)



Color Node “b” Green (r1)

Final Register Assignment for Basic Block B_1

- Final Register Assignment for B_1
  - a is in r0 (red)
  - b is in r1 (green)
  - c is in r1 (green)
  - d is in r3 (yellow)
  - e is in r2 (blue)
  - f is in r1 (green)

- We could have chosen to use more than four registers

How Many Registers to Use?

- We'll leave this as an optimization question for the students
- Perhaps try to use the minimum number of registers possible
  - But, if we're saving all of the $s registers anyway this is of no benefit
- Perhaps make all of the $s registers available and minimize the number of $t registers used if there can be no spill code