

Register Allocation

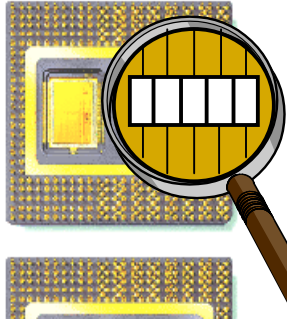
Lecture 28

(from notes by G. Necula and R. Bodik)

Lecture Outline

- Memory Hierarchy Management
- Register Allocation
 - Register interference graph
 - Graph coloring heuristics
 - Spilling
- Cache Management

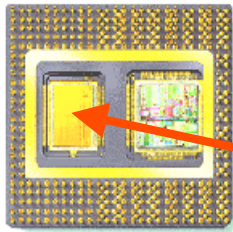
The Memory Hierarchy



Registers

1 cycle

256-2000 bytes



Cache

1-100 cycles

256k-16M



Main memory

150-1000 cycles

32M-16G



Disk

0.5-10M cycles

10G-1T

Managing the Memory Hierarchy

- Programs are written as if there are only two kinds of memory: main memory and disk
- Programmer is responsible for moving data from disk to memory (e.g., file I/O)
- Hardware is responsible for moving data between memory and caches
- Compiler is responsible for moving data between memory and registers

Current Trends

- Cache and register sizes are growing slowly
- Processor speed improves faster than memory speed and disk speed
 - The cost of a cache miss is growing
 - The widening gap is bridged with more caches
- It is very important to:
 - Manage registers properly
 - Manage caches properly
- Compilers are good at managing registers

The Register Allocation Problem

- Intermediate code uses as many temporaries as necessary
 - This complicates final translation to assembly
 - But simplifies code generation and optimization
 - Typical intermediate code uses too many temporaries
- The register allocation problem:
 - Rewrite the intermediate code to use fewer temporaries than there are machine registers
 - Method: assign more temporaries to a register
 - But without changing the program behavior

History

- Register allocation is as old as intermediate code
- Register allocation was used in the original FORTRAN compiler in the '50s
 - Very crude algorithms
- A breakthrough was not achieved until 1980 when Chaitin invented a register allocation scheme based on graph coloring
 - Relatively simple, global and works well in practice

An Example

- Consider the program

$a := c + d$

$e := a + b$

$f := e - 1$

- with the assumption that a and e die after use
- Temporary a can be "reused" after " $a + b$ "
- Same with temporary e after " $e - 1$ "
- Can allocate a , e , and f all to one register (r_1):

$r_1 := c + d$

$r_1 := r_1 + b$

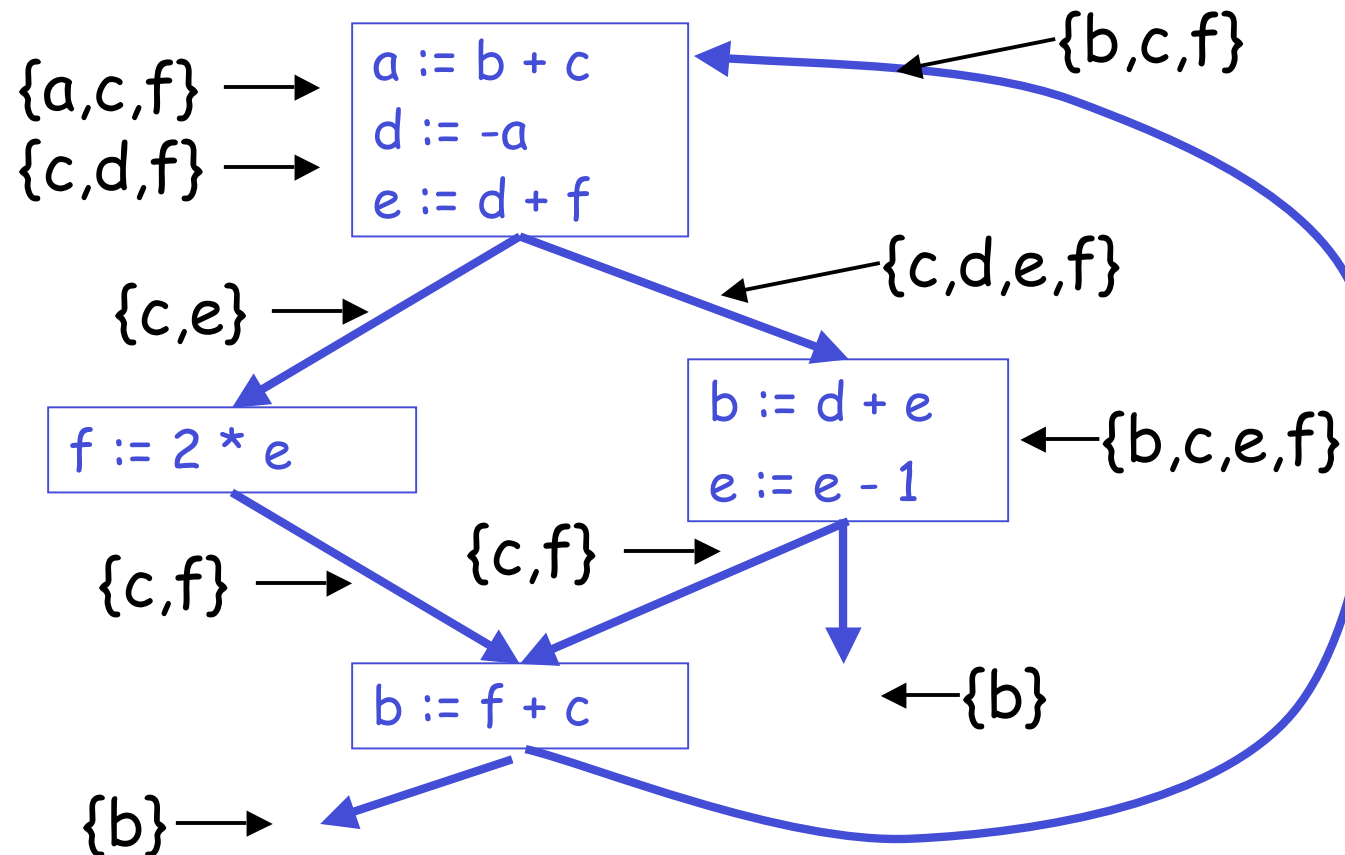
$r_1 := r_1 - 1$

Basic Register Allocation Idea

- The value in a dead temporary is not needed for the rest of the computation
 - A dead temporary can be reused
- Basic rule:
 - 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!*

Algorithm: Part I

- Compute live variables for each point:

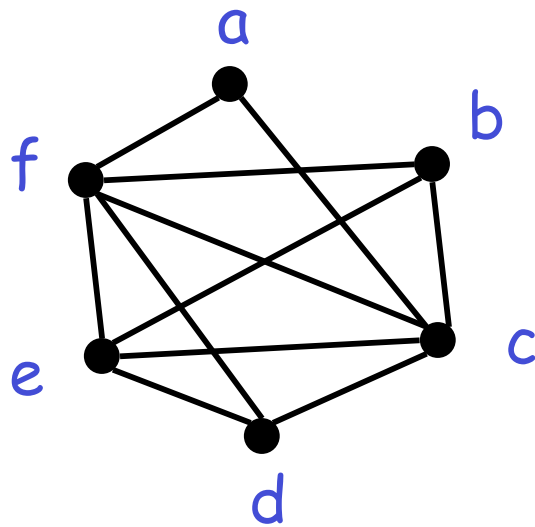


The Register Interference Graph

- Two temporaries that are live simultaneously cannot be allocated in the same register
- We 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. Example.

- For our example:



- E.g., **b** and **c** cannot be in the same register
- E.g., **b** and **d** can be in the same register

Register Interference Graph. Properties.

- It extracts exactly the information needed to characterize legal register assignments
- It gives a global (i.e., over the entire flow graph) picture of the register requirements
- After RIG construction the register allocation algorithm is architecture independent

Graph Coloring. Definitions.

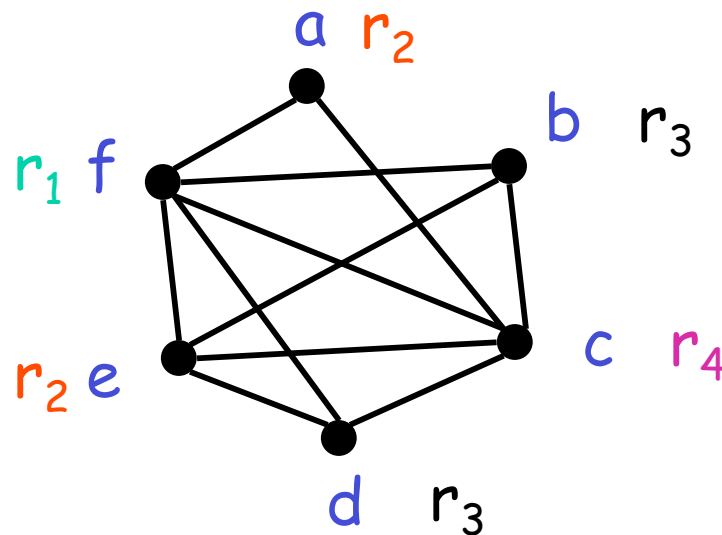
- A *coloring of a graph* is an assignment of colors to nodes, such that nodes connected by an edge have different colors
- A graph is *k-colorable* if it has a coloring with k colors

Register Allocation Through 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

Graph Coloring. Example.

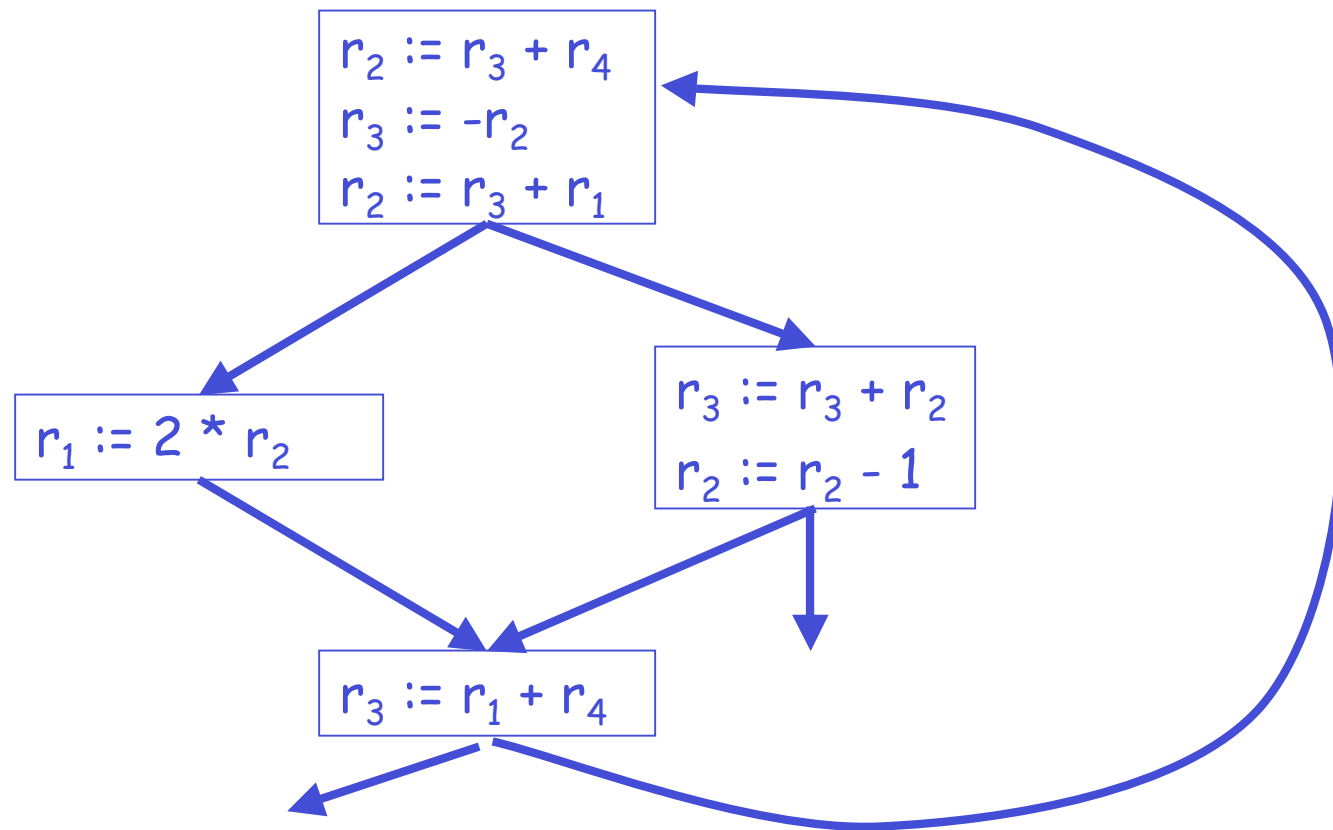
- Consider the sample RIG



- There is no coloring with fewer than 4 colors
- There are 4-colorings of this graph

Graph Coloring. Example.

- Under this coloring the code becomes:



Computing Graph Colorings

- The remaining problem is to compute a coloring for the interference graph
- But:
 1. This problem is very hard (NP-hard). No efficient algorithms are known.
 2. A coloring might not exist for a given number of registers
- The solution to (1) is to use heuristics
- We'll consider later the other problem

Graph Coloring Heuristic

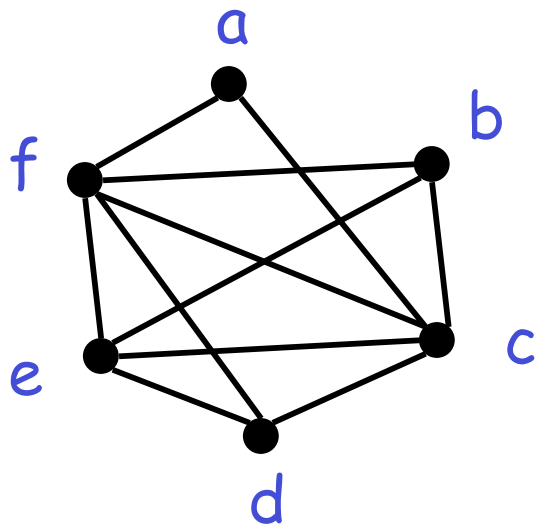
- Observation:
 - Pick a node t with fewer than k neighbors in RIG
 - Eliminate t and its edges from RIG
 - If the resulting graph has a k -coloring then so does the original graph
- Why:
 - Let c_1, \dots, c_n be the colors assigned to the neighbors of t in the reduced graph
 - Since $n < k$ we can pick some color for t that is different from those of its neighbors

Graph Coloring Heuristic

- The following works well in practice:
 - Pick a node t with fewer than k neighbors
 - Push t on a stack and remove it from the RIG
 - Repeat until the graph has one node
- Then start assigning colors to nodes in the stack (starting with the last node added)
 - At each step pick a color different from those assigned to already colored neighbors

Graph Coloring Example (1)

- Start with the RIG and with $k = 4$:

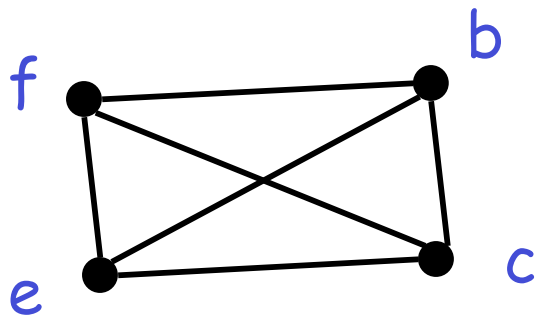


Stack: {}

- Remove **a** and then **d**

Graph Coloring Example (2)

Resulting graph:

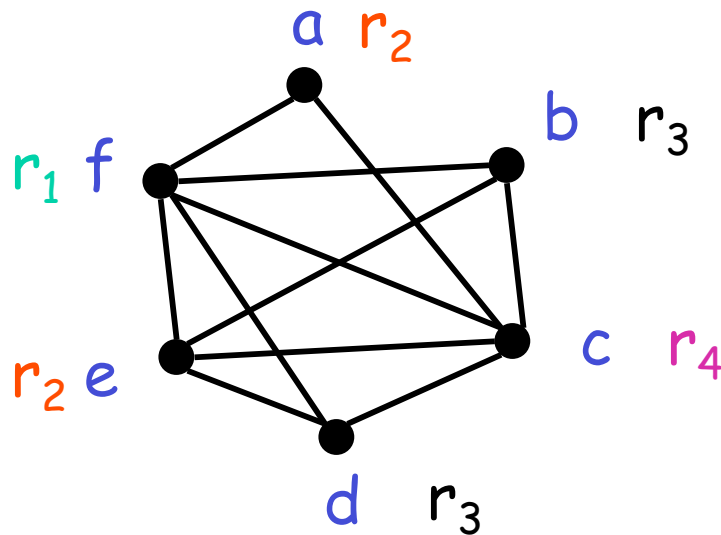


Stack: {d, a}

- Now all nodes have fewer than 4 neighbors and can be removed: c, b, e, f
- Leaving an empty graph and stack containing {f, e, b, c, d, a}

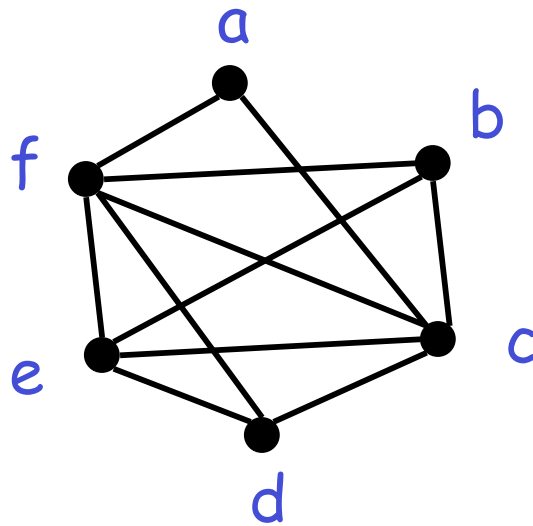
Graph Coloring Example (2)

- Start assigning colors to: f, e, b, c, d, a
- At each step, guaranteed there's a free color



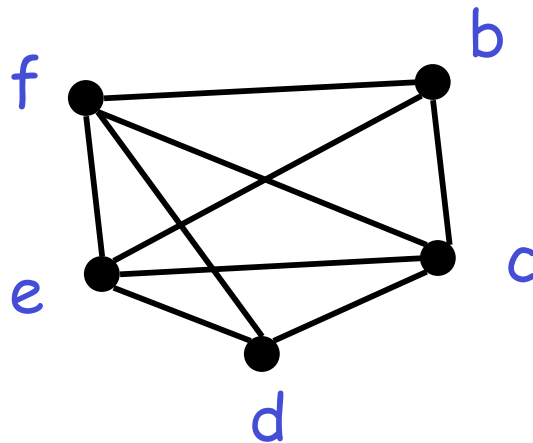
What if the Heuristic Fails?

- What if during simplification we get to a state where all nodes have k or more neighbors ?
- Example: try to find a 3-coloring of the RIG:



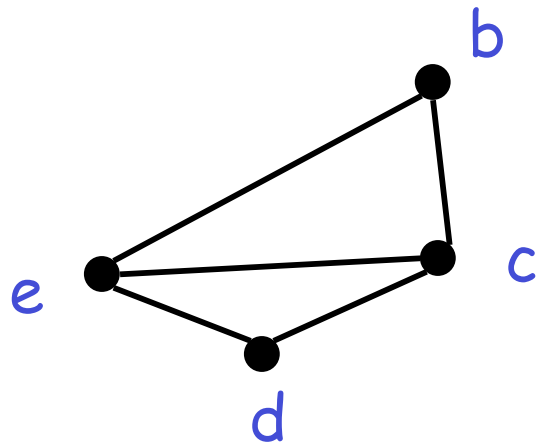
What if the Heuristic Fails?

- 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



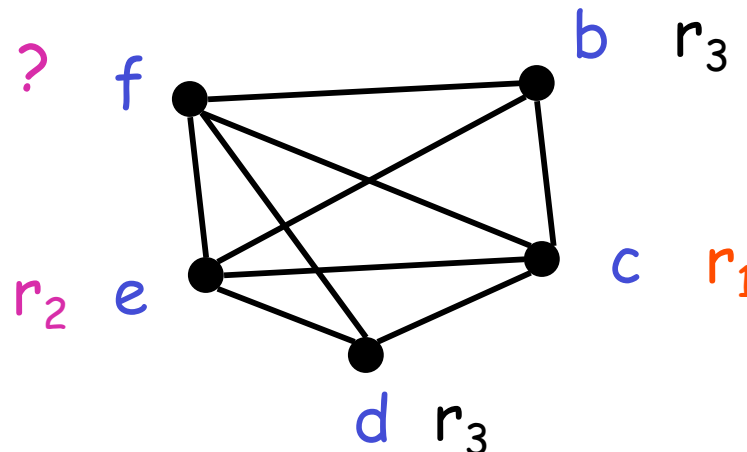
What if the Heuristic Fails?

- Remove **f** and continue the simplification
 - Simplification 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 \Rightarrow *optimistic coloring*

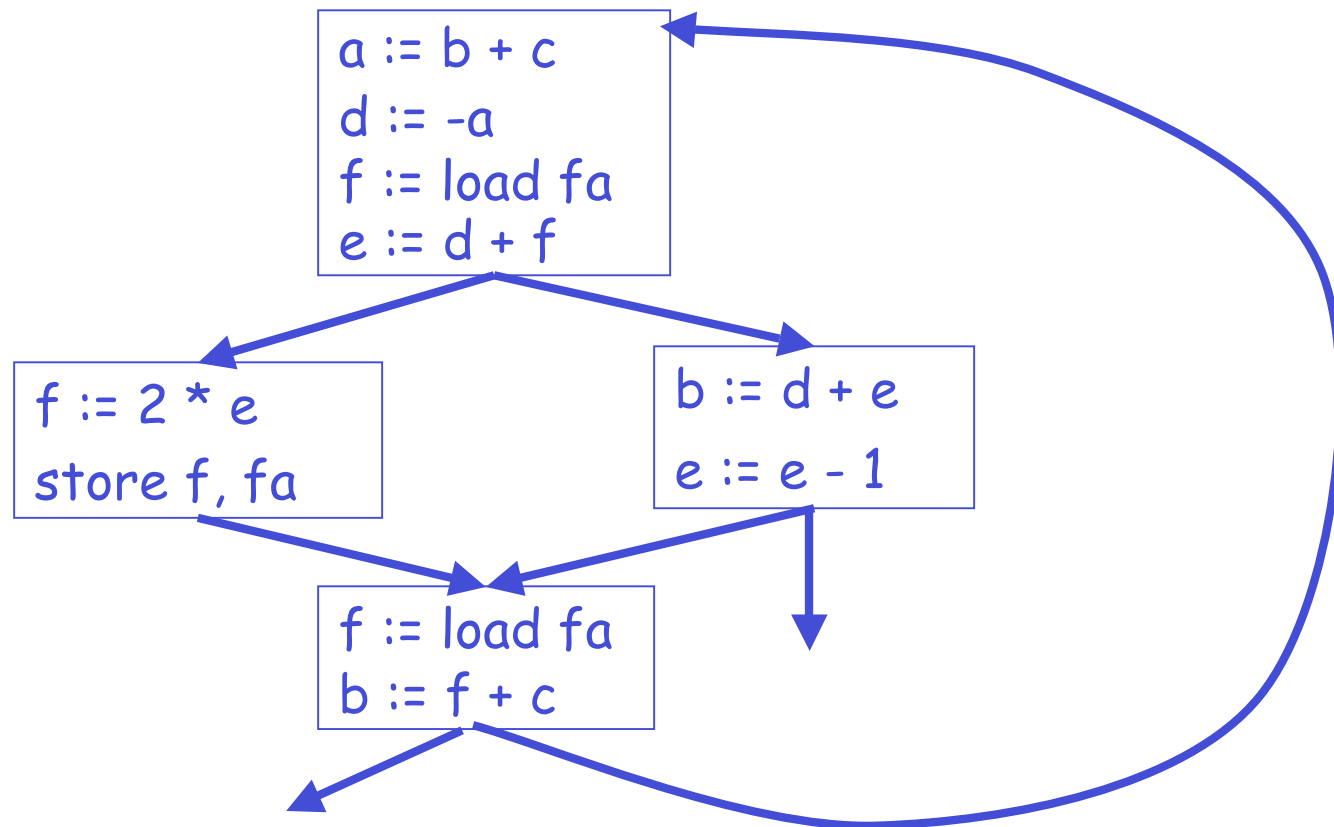


Spilling

- 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
 $f := \text{load } fa$
- After each operation that defines f , insert
 $\text{store } f, fa$

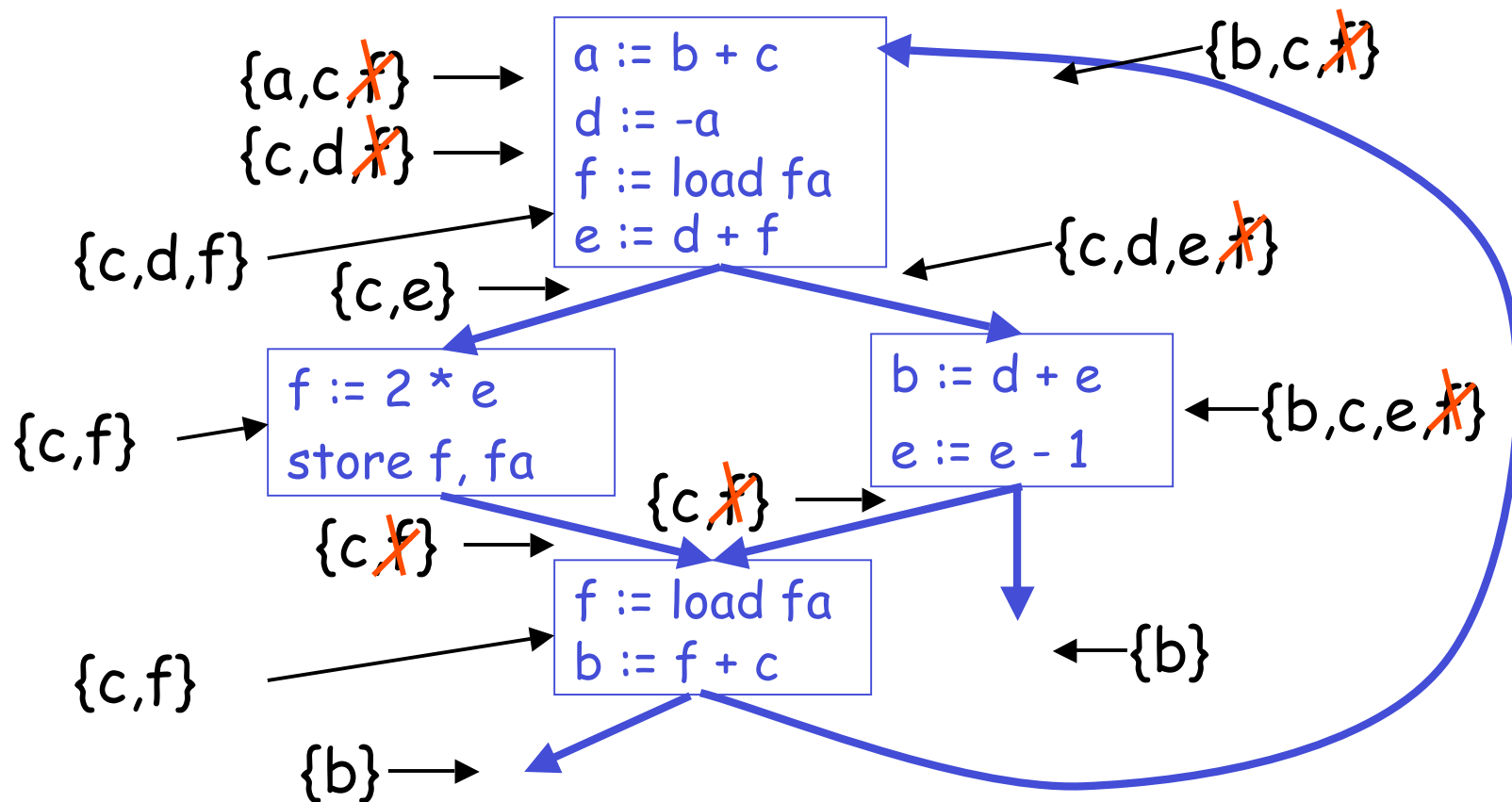
Spilling. Example.

- This is the new code after spilling f



Recomputing Liveness Information

- The new liveness information after spilling:

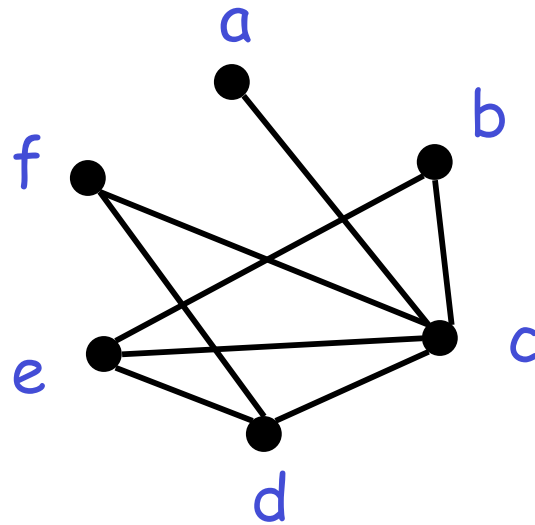


Recomputing Liveness Information

- The new liveness information is almost as before
- f is live only
 - Between a $f := \text{load } fa$ and the next instruction
 - Between a $\text{store } f, fa$ and the preceding instr.
- Spilling reduces the live range of f
- And thus reduces its interferences
- Which result in fewer neighbors in RIG for f

Recompute RIG After Spilling

- The only changes are in removing some of the edges of the spilled node
- In our case **f** still interferes only with **c** and **d**
- And the resulting RIG is 3-colorable



Spilling (Cont.)

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

Caches

- Compilers are very good at managing registers
 - Much better than a programmer could be
- Compilers are not good at managing caches
 - This problem is still left to programmers
 - It is still an open question whether a compiler can do anything general to improve performance
- Compilers can, and a few do, perform some simple cache optimization

Cache Optimization

- Consider the loop

```
for(j = 1; j < 10; j++)  
    for(i=1; i<1000000; i++)  
        a[i] *= b[i]
```
- This program has a terrible cache performance
 - Why?

Cache Optimization (II)

- Consider the program:

```
for(i=1; i<1000000; i++)  
    for(j = 1; j < 10; j++)  
        a[i] *= b[i]
```

 - Computes the same thing
 - But with much better cache behavior
 - Might actually be more than 10x faster
- A compiler can perform this optimization
 - called loop interchange

Cache Optimization (III)

- Other kinds of memory layout decisions possible, such as *padding* rows of matrix for better fit in a cache.
- *Prefetching* informs cache of anticipated future memory fetches so that they can proceed in parallel.
- Hardware often does this, in which case compiler may have to tell it when *not* to prefetch.

Conclusions

- Register allocation is a “must have” optimization in most compilers:
 - Because intermediate code uses too many temporaries
 - Because it makes a big difference in performance
- Graph coloring is a powerful register allocation scheme
- Good cache management could give much larger payoffs, but it is difficult.