Register Allocation

(via Graph Coloring)

Presented By
Rakesh Kaparthi

Register Allocation

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

- Typical intermediate code uses too many temporaries

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

Simple 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 (r_1):

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

Steps to Perform Register Allocation

Step 1: Draw the Control Flow Graph (CFG)

Step 2: Perform Liveness Analysis

Step 3: Draw the Register Interference Graph (RIG)

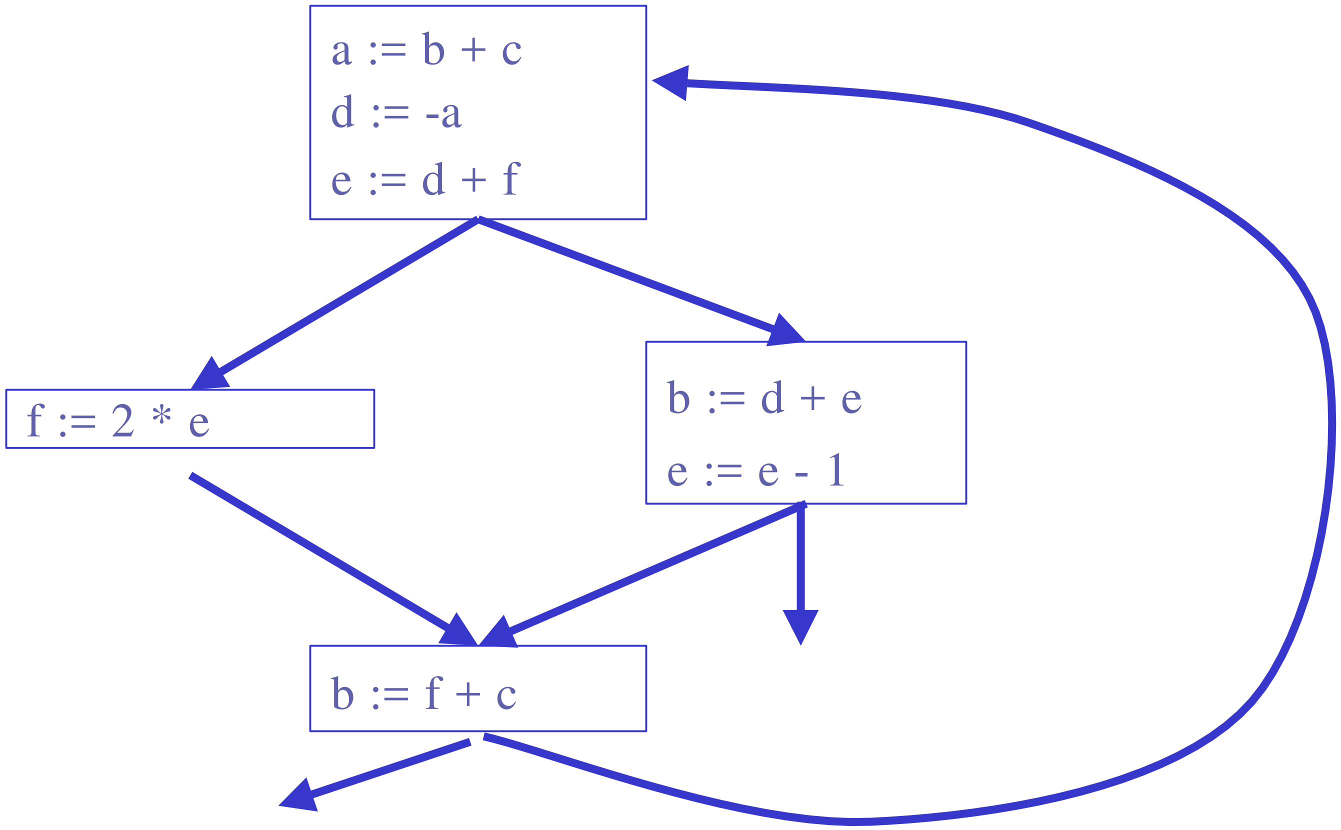
Step 4: Perform Graph Coloring

Step 5: Allocate Registers based on Colored Graph

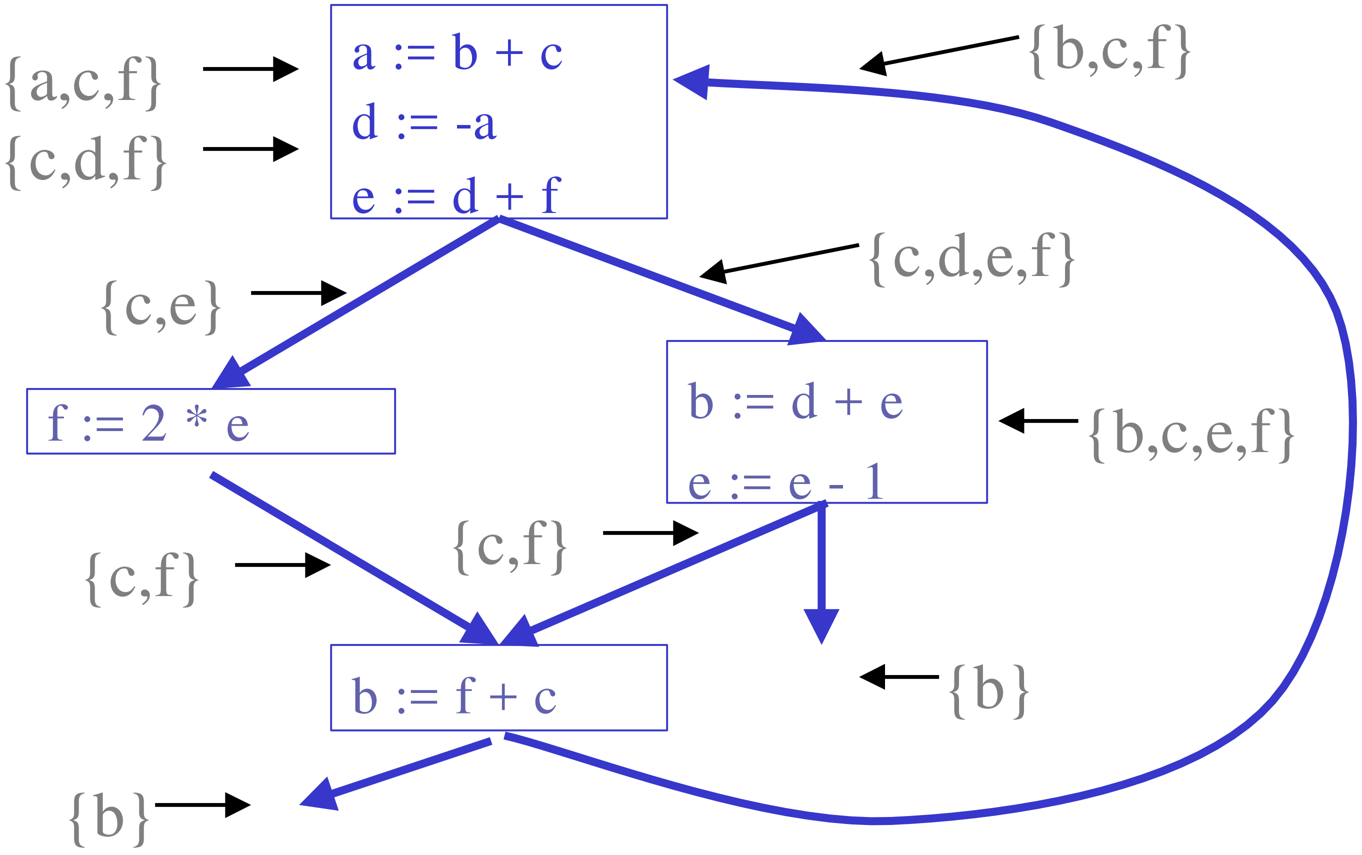
Example

```
L1: a=b + c
d:= -a
e:= d + f
 if(expression) then
      f:= 2 * e
else
      b:= d + e
      e:= e - 1
      …
end if
b := f + c
goto to L1
….
….
```

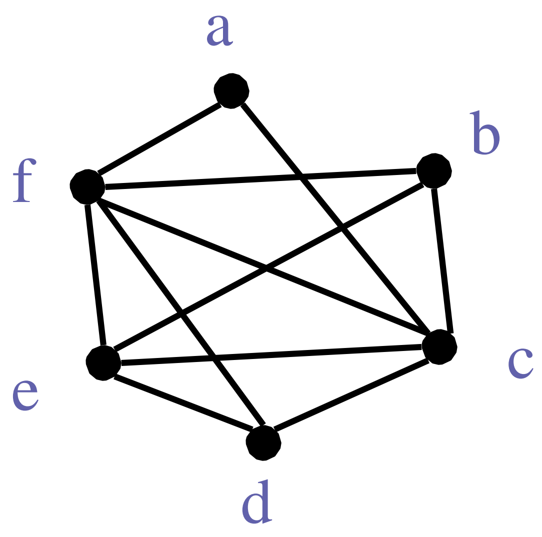
Step 1: Control Flow Graph



Step 2: Perform Liveness Analysis



Step 3: Register Interference Graph



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

Step 4: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 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 $c_1,\ldots,c_n$ 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

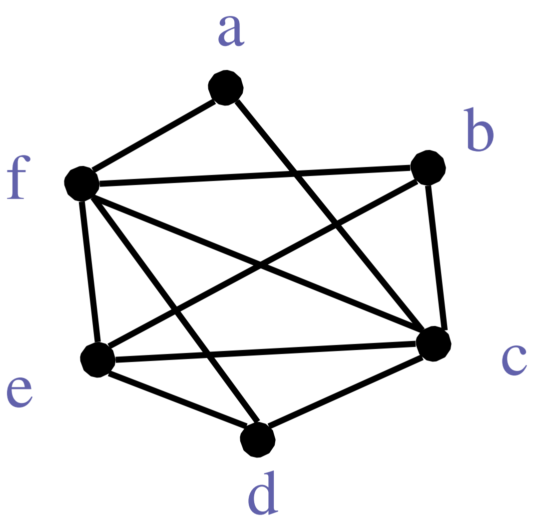
Graph Coloring Heuristic

- The following works well in practice:
  - Pick a node t with fewer than k neighbors
  - Put t on a stack and remove it from the RIG
  - Repeat until the graph has one node

- Then start assigning colors to nodes on 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:

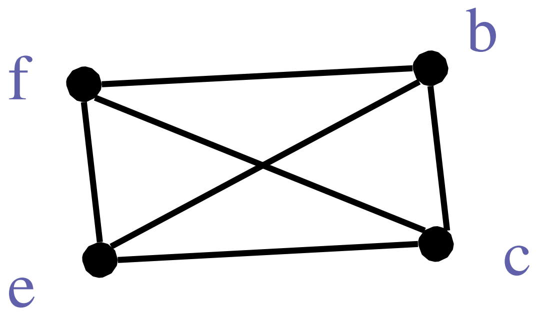


Stack: { }

- Remove a and then d

Graph Coloring Example(2)

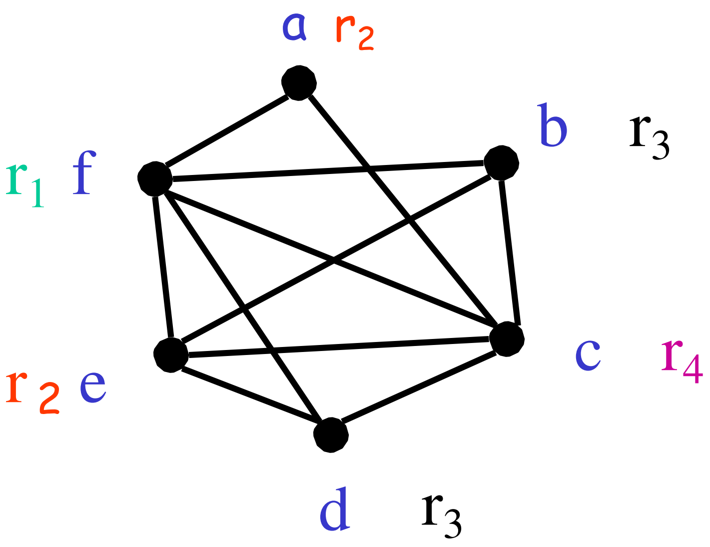
- Now all nodes have fewer than 4 neighbors and can be removed: c, b, e, f



Stack: {d, a}

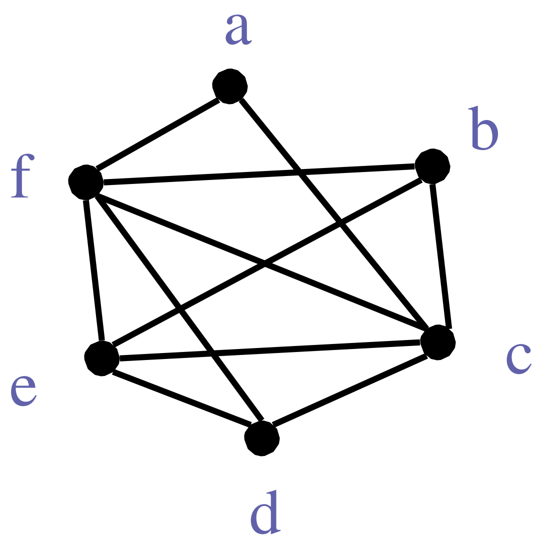
Graph Coloring Example(3)

- Start assigning colors to: f, e, b, c, d, a



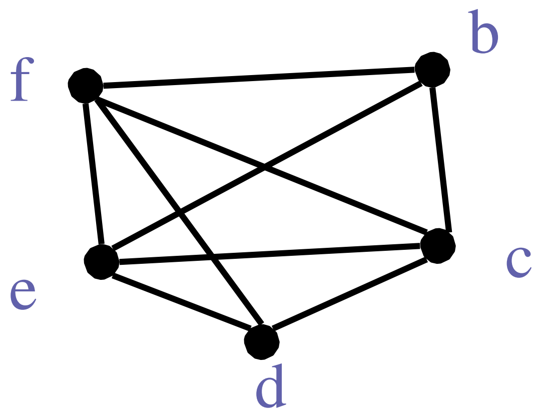
What if the Heuristic Fails?

- 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:



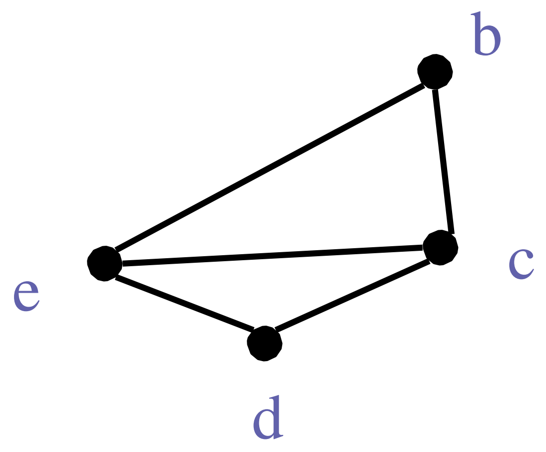
What if the Heuristic Fails?

- 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



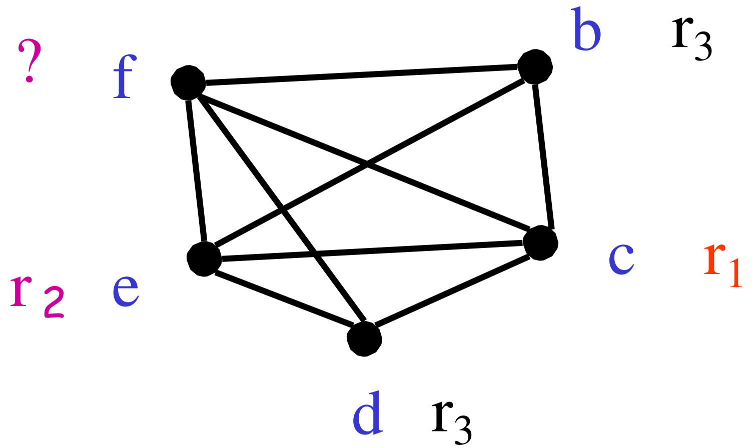
What if the Heuristic Fails?

- 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 ⇒optimistic coloring



Spilling

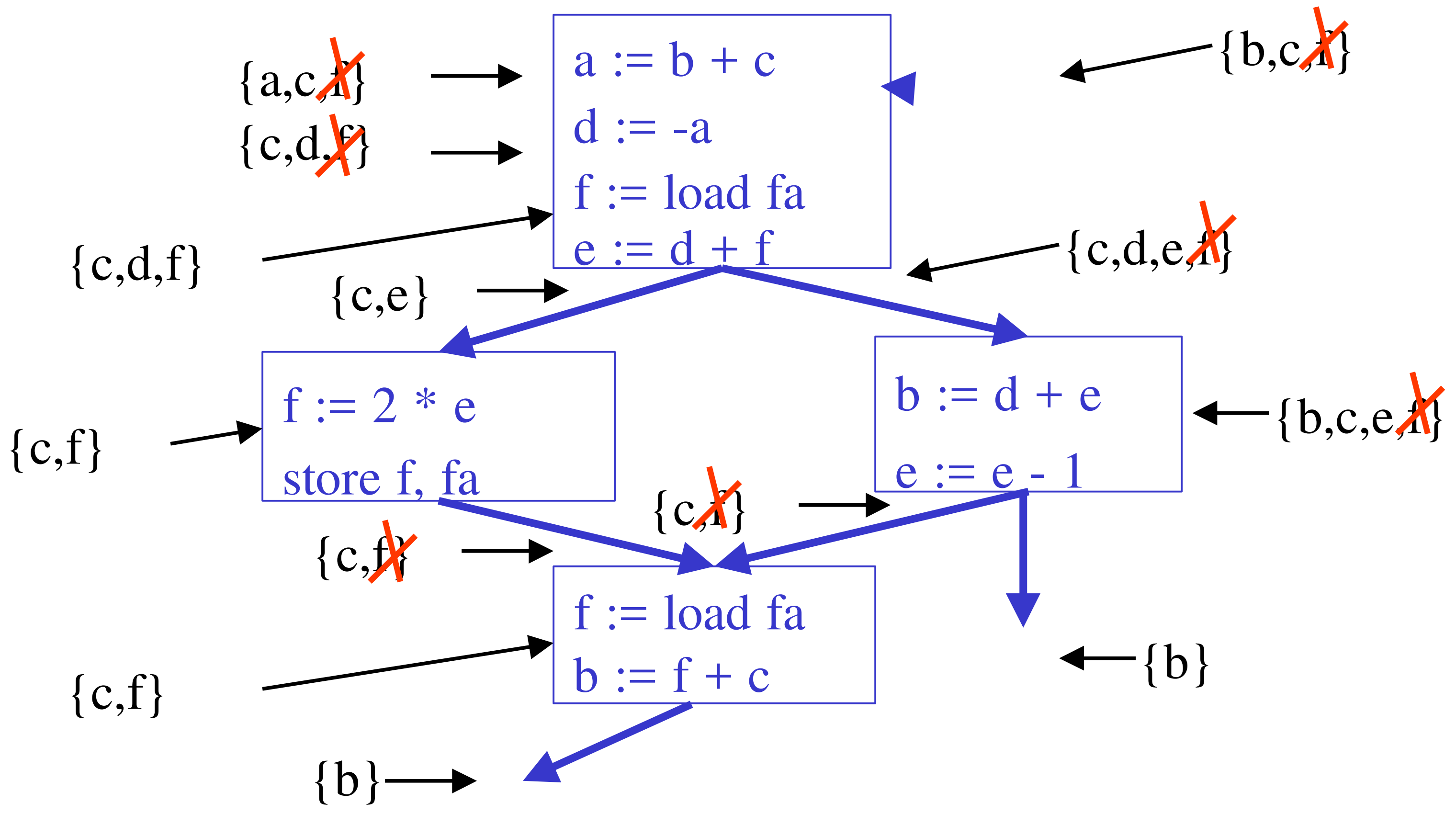
- Since optimistic coloring failed we must spill temporary f
- We must allocate a memory location as the home of f
  - Typically this is in the current stack frame
  - Call this address fa
- Before each operation that uses f, insert

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

  store f, fa

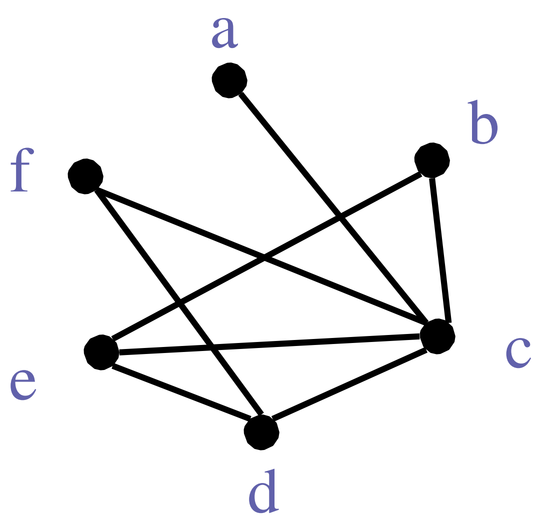
Recomputing Liveness Information

- The new liveness information after spilling:



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

THANK YOU