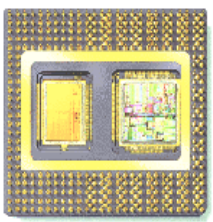
Register Allocation via Graph Coloring

John Cavazos

University of Delaware

The Memory Hierarchy

- Higher = smaller, faster, closer to CPU



registers	8 integer, 8 floating-point; 1-cycle latency
L1 cache	8K data & instructions; 2-cycle latency
L2 cache	512K; 7-cycle latency
RAM	1GB; 100 cycle latency
Disk	40 GB; 38,000,000 cycle latency (!)

Managing the Memory Hierarchy

- Programmer view: only two levels of memory
  - Main memory (stores & loads)
  - Disk (file I/O)

- Two things maintain this abstraction:
  - Hardware
    - Moves data between memory and caches
  - Compiler
    - **Moves data between memory and registers**

Overview

- Introduction
- Register Allocation
  - Definition
  - History
  - Interference graphs
  - Graph coloring
  - Register spilling

Register Allocation: Definition

- **Register allocation** assigns registers to values
  - Candidate values:
    - Variables
    - Temporaries
    - Large constants
  - When needed, **spill** registers to memory

- Important low-level optimization
  - Registers are 2x – 7x faster than cache
    - Can lead to big performance improvements

Register Allocation Example

- Consider this program with six variables:

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

with the assumption that a and e die after use

  - Variable a can be "reused" after $e := a + b$
  - Same with variable e

- Can allocate a, e, and f all to one register (r_1):

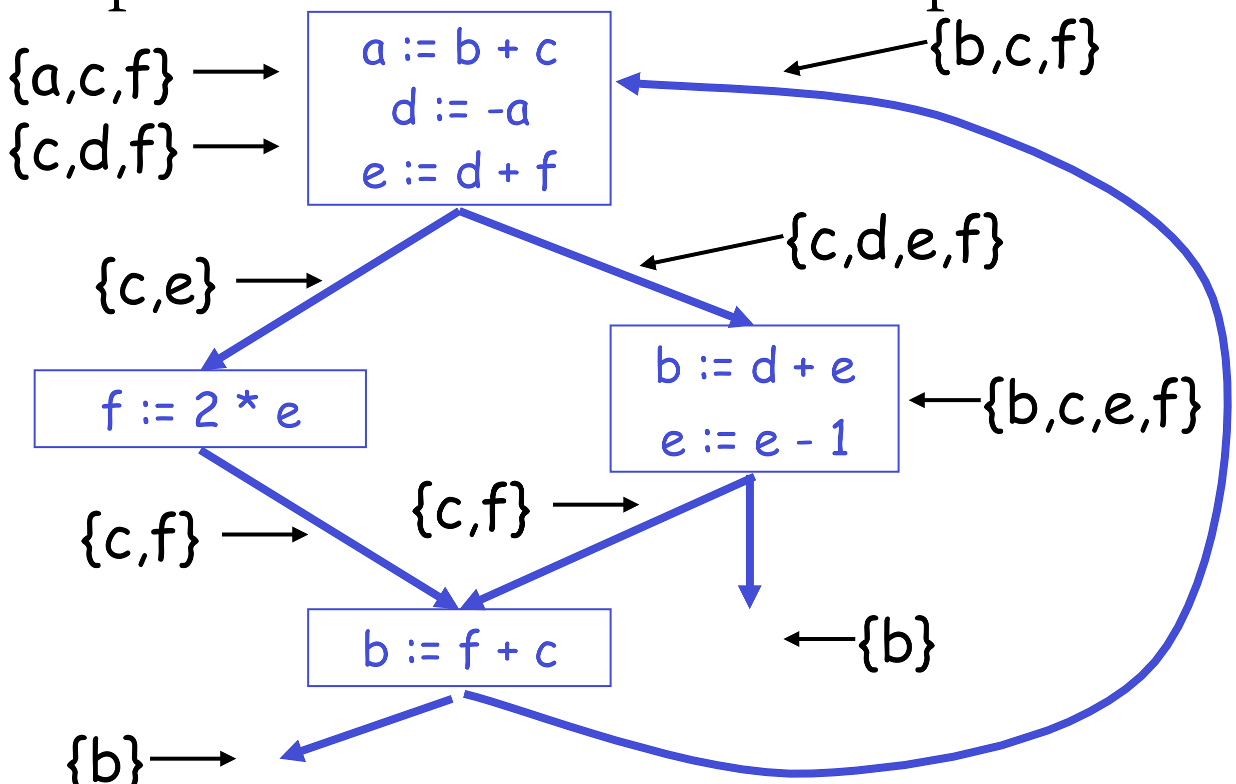
$r_1 := r_2 + r_3$
$r_1 := r_1 + r_4$
$r_1 := r_1 - 1$

Basic Register Allocation Idea

- *Variables t_1 and t_2 can share 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:



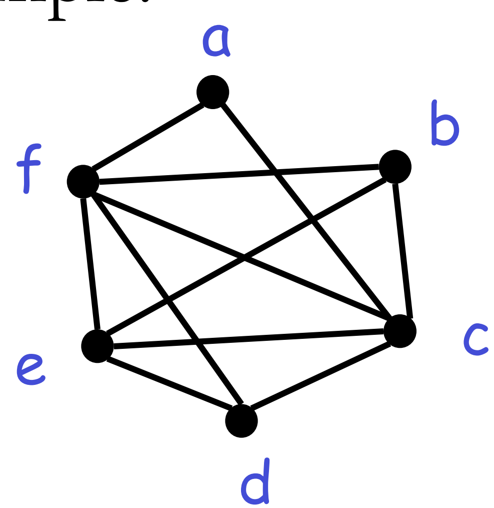
Interference Graph

- Two variables live simultaneously
  - Cannot be allocated in the same register
- Construct an **interference graph (IG)**
  - Node for each variable
  - Undirected edge between t_1 and t_2
    - If live simultaneously at some point in the program
- Two variables can be allocated to same register if no edge connects them

Interference Graph: Example

- For our example:

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



b and c **cannot** be in the same register

b and d **can** be in the same register

Graph Coloring

- **Graph coloring**:
  assignment of colors to nodes
  - Nodes connected by edge have different colors

- Graph **k-colorable** =
  can be colored with k colors

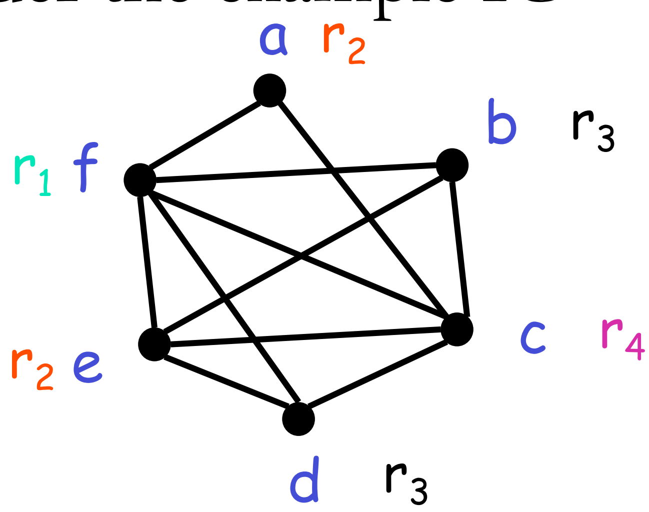
Register Allocation Through Graph Coloring

- In our problem, colors = registers
  - We need to assign colors (registers) to graph nodes (variables)
  - Let k = number of machine registers

- If the IG is k-colorable, there's a register assignment that uses no more than k registers

Graph Coloring Example

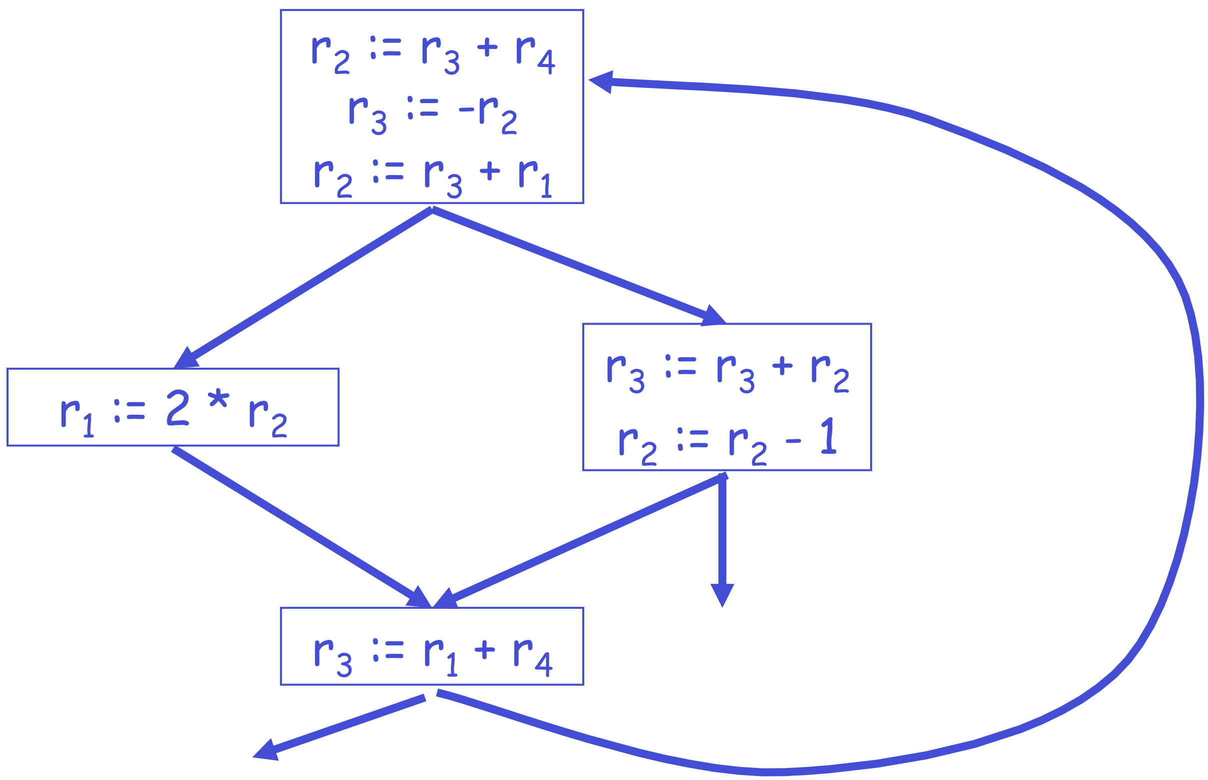
- Consider the example IG



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

Graph Coloring Example, Continued

- Under this coloring the code becomes:



Computing Graph Colorings

- How do we compute coloring for IG?
  - NP-hard!
  - For given # of registers, coloring may not exist

- Solution
  - Use heuristics

Graph Coloring Algorithm (Chaitin)

```
while G cannot be k-colored
    while graph G has node N with degree less than k
        remove N and its edges from G and push N on a stack S
    end while
    if all nodes removed then graph is k-colorable
       while stack S contains node N
           add N to graph G and assign it a color from k colors
       end while
    else graph G cannot be colored with k colors
       simplify graph G choosing node N to spill and remove node
      (spill nodes chosen based number of definitions and uses)
end while
```

Graph Coloring Heuristic

- Observation: “degree < k” rule
  - Reduce graph:
    - Pick node N with < k neighbors in IG
    - Eliminate N and its edges from IG
  - If the resulting graph has k-coloring, so does the original graph

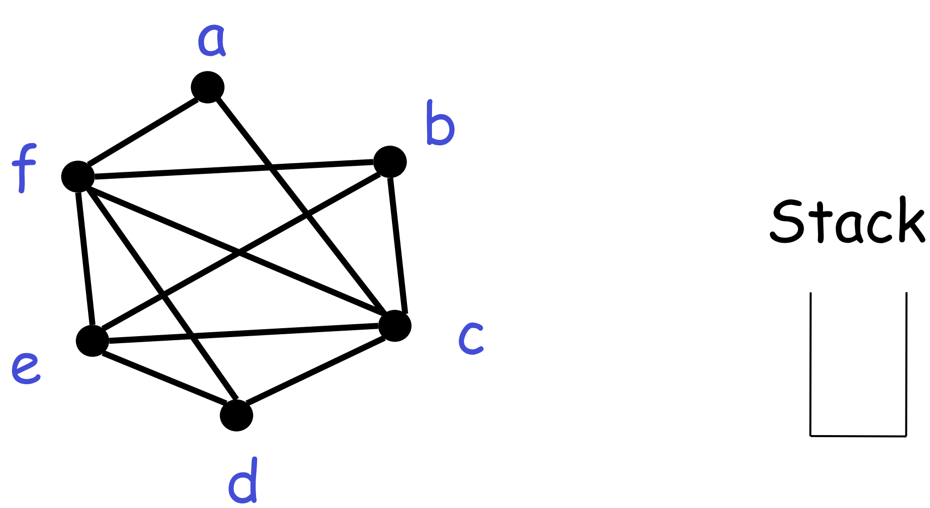
- Why?
  - Let $c_1,\ldots,c_n$ be colors assigned to neighbors of t in reduced graph
  - Since n < k, we can pick some color for t different from those of its neighbors

Graph Coloring Heuristic, Continued

- Heuristic:
  - Pick node t with fewer than k neighbors
  - Put t on a stack and remove it from the IG
  - Repeat until all nodes have been removed
- Start assigning colors to nodes on the stack (starting with the last node added)
  - At each step, pick color different from those assigned to already-colored neighbors

Graph Coloring Example (1)

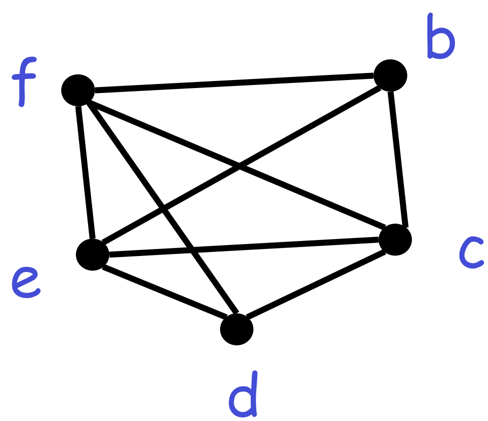
- Start with the IG and with k = 4:

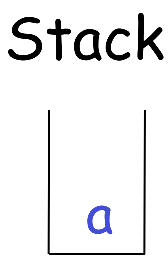


- Remove a

Graph Coloring Example (1)

- Start with the IG and with k = 4:

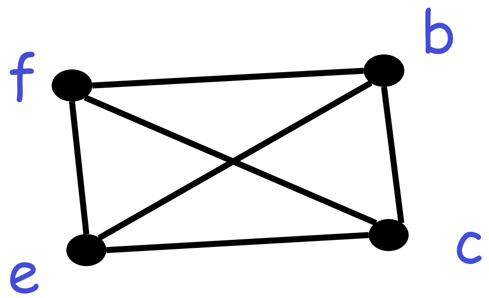


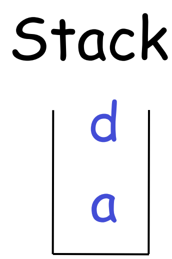


- Remove d

Graph Coloring Example (2)

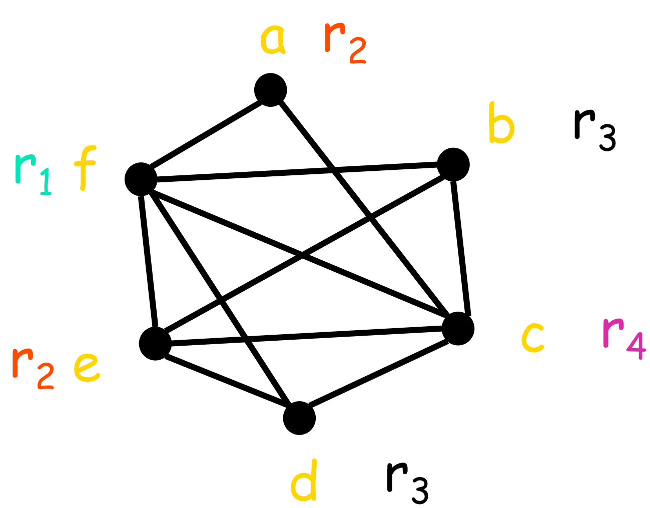
- Now all nodes have fewer than 4 neighbors and can be removed: c, b, e, f





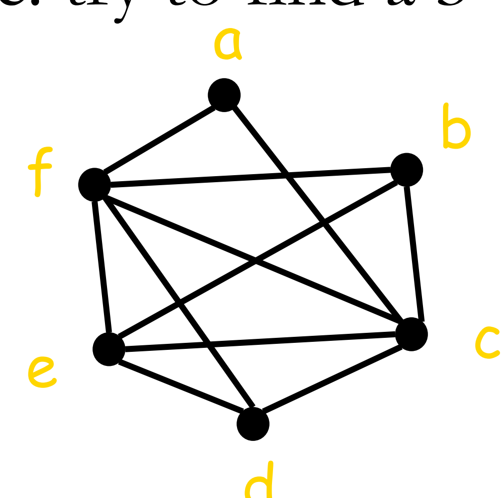
Graph Coloring Example (2)

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



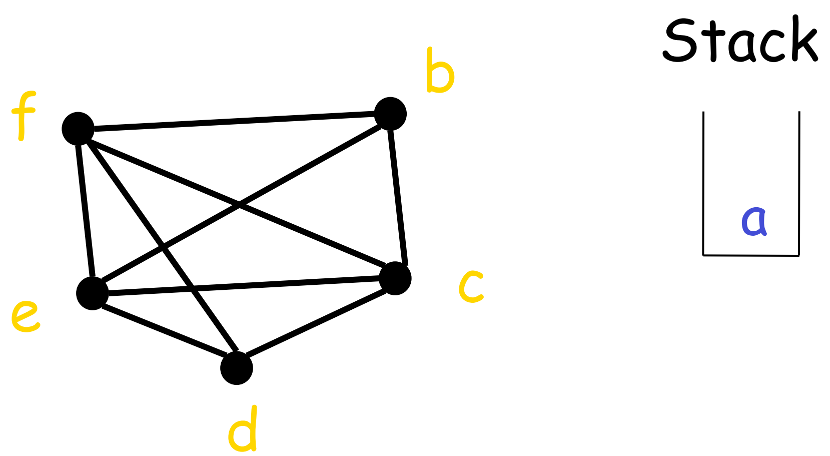
What if the Heuristic Fails?

- 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 IG:



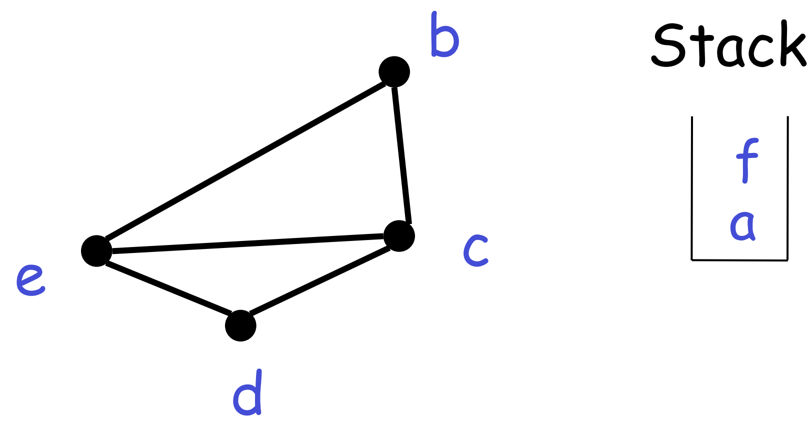
What if the Heuristic Fails?

- Remove a and get stuck (as shown below)
  - Pick a node as a candidate for spilling
  - Assume that f is picked



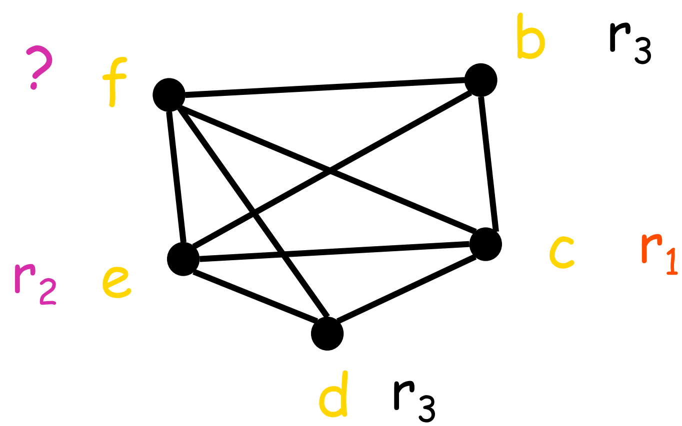
What if the Heuristic Fails?

- Remove f and continue the simplification
  - Optimistically push on stack
  - Simplification now succeeds: b, d, e, c



What if the Heuristic Fails?

- During assignment phase, we get to the point when we have to assign a color to f
- Hope: among the 4 neighbors of f, we use less than 3 colors ⇒ **optimistic coloring**



Spilling

- Optimistic coloring failed = must spill variable f
- Allocate memory location as home of f
  - Typically in 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, Continued

- Additional spills might be required before coloring is found
- Tricky part: deciding what to spill
  - Possible heuristics:
    - Spill variables with most conflicts
    - Spill variables with few definitions and uses
    - Avoid spilling in inner loops
  - All are "correct"

Conclusion

- Register allocation: “must have” optimization in most compilers:
  - Intermediate code uses too many temporaries
  - Makes a big difference in performance
- Graph coloring:
  - Powerful register allocation scheme