

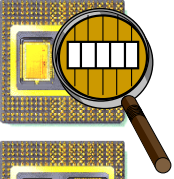
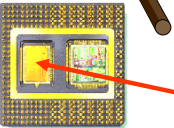


# Global Register Allocation

## Lecture Outline

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

2

## The Memory Hierarchy

	Registers	1 cycle	256-8000 bytes
	Cache	3 cycles	256k-16M
	Main memory	20-100 cycles	512M-64G
	Disk	0.5-5M cycles	10G-1T

3

## 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

4

## Current Trends

---

- Power usage limits
  - Size and speed of registers/caches
  - Speed of processors
    - Improves faster than memory speed (and disk speed)
    - The cost of a cache miss is growing
    - The widening gap between processors and memory is bridged with more levels of caches
- It is very important to:
  - Manage registers properly
  - Manage caches properly
- Compilers are good at managing registers

5

## The Register Allocation Problem

---

- Recall that 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 at most as many temporaries as there are machine registers
  - Method: Assign multiple temporaries to a register
    - But without changing the program behavior

6

## 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
  - Register allocation scheme based on graph coloring
  - Relatively simple, global, and works well in practice

7

## 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<sub>1</sub>**):

```
r1 := r2 + r3
r1 := r1 + r4
r1 := r1 - 1
```

8

## 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

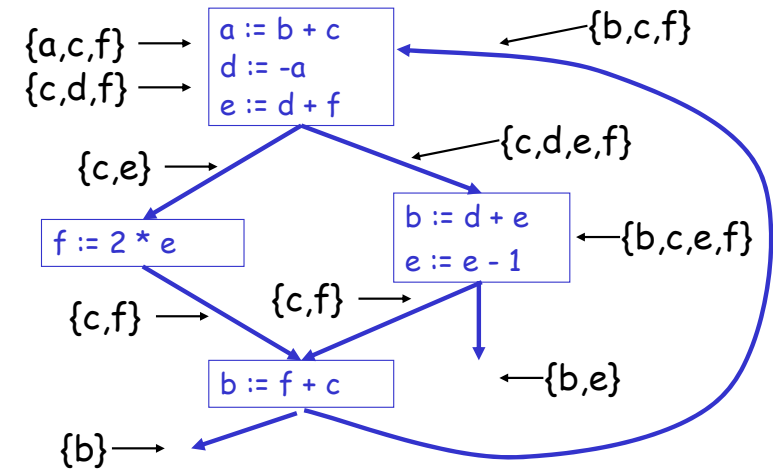
- Basic rule:

*Temporaries  $t_1$  and  $t_2$  can share the same register if at all points in the program at most one of  $t_1$  or  $t_2$  is live!*

9

## Algorithm: Part I

Compute live variables for each program point:



10

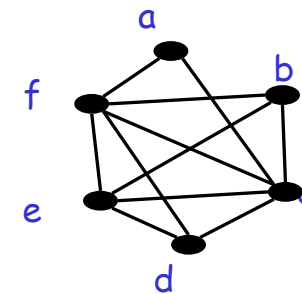
## The Register Interference Graph

- Two temporaries that are live simultaneously cannot be allocated in the same register
- We construct an undirected graph with
  - 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

11

## 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

12

## 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

13

## 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

14

## Register Allocation Through Graph Coloring

---

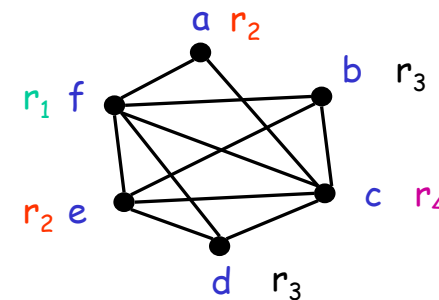
- In our problem, colors = registers
  - We need to assign colors (registers) to graph nodes (temporaries)
- Let  $k$  = number of machine registers
- If the RIG is  $k$ -colorable then there is a register assignment that uses no more than  $k$  registers

15

## Graph Coloring: Example

---

- Consider the example RIG

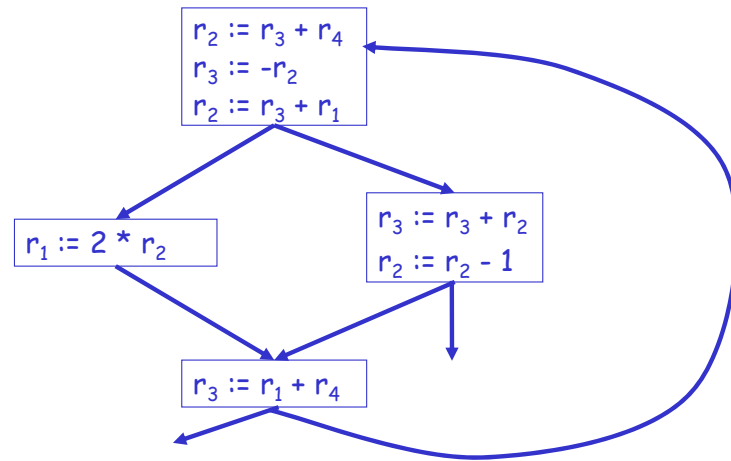


- There is no coloring with less than 4 colors
- There are various 4-colorings of this graph

16

## Graph Coloring: Example

- Under this coloring the code becomes:



17

## Computing Graph Colorings

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

18

## Graph Coloring Heuristic

- Observation:
  - Pick a node  $\dagger$  with fewer than  $k$  neighbors in RIG
  - Eliminate  $\dagger$  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  $\dagger$  in the reduced graph
  - Since  $n < k$  we can pick some color for  $\dagger$  that is different from those of its neighbors

19

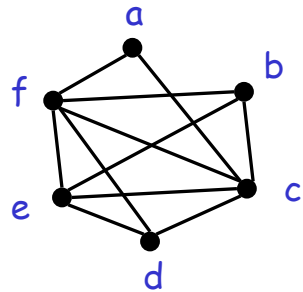
## Graph Coloring Simplification Heuristic

- The following works well in practice:
  - Pick a node  $\dagger$  with fewer than  $k$  neighbors
  - Put  $\dagger$  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

20

## Graph Coloring Example (1)

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

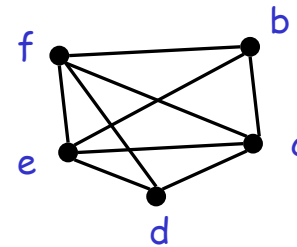


Stack: {}

- Remove  $a$

## Graph Coloring Example (2)

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

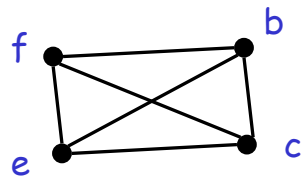


Stack: {a}

- Remove  $d$

## Graph Coloring Example (3)

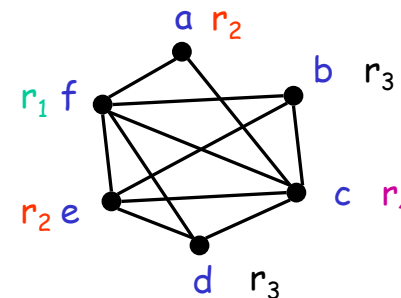
- Now all nodes have fewer than 4 neighbors and can be removed:  $c, b, e, f$



Stack: {d, a}

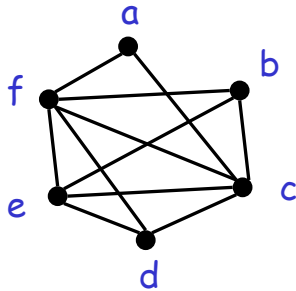
## Graph Coloring Example (4)

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



## What if the Heuristic Fails?

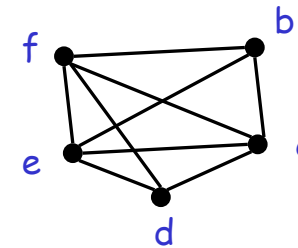
- What if during simplification we get to a state where all nodes have k or more neighbors?
- Example: try to find a 3-coloring of the RIG:



25

## What if the Heuristic Fails?

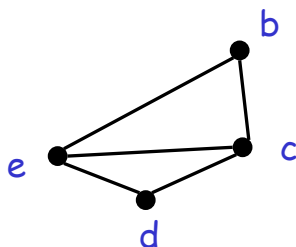
- Remove **a** and get stuck (as shown below)
- Pick a node as a possible candidate for **spilling**
  - A spilled temporary "lives" is memory
  - Assume that **f** is picked as a candidate



26

## What if the Heuristic Fails?

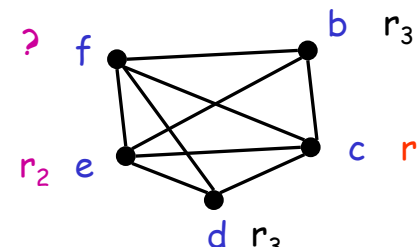
- Remove **f** and continue the simplification
  - Simplification now succeeds: **b, d, e, c**



27

## 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 used less than 3 colors  $\Rightarrow$  **optimistic coloring**



28

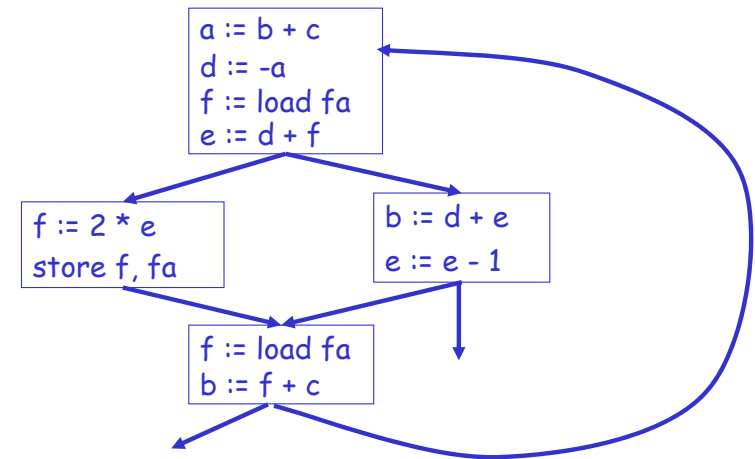
## Spilling

- Since optimistic coloring failed, we must spill temporary **f** (actual spill)
- 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  $\text{store } f, fa$

29

## Spilling: Example

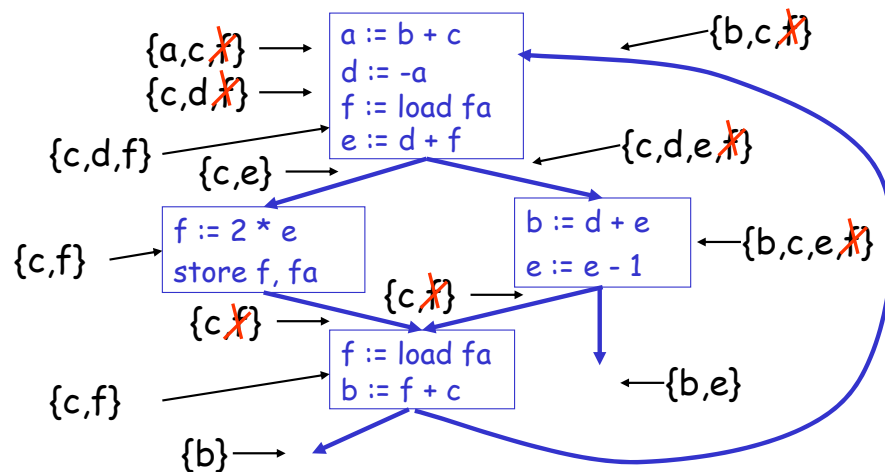
- This is the new code after spilling **f**



30

## Recomputing Liveness Information

- The new liveness information after spilling:



31

## Recomputing Liveness Information

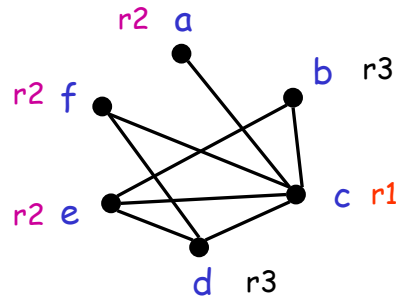
- New liveness information is almost as before
- **f** is live only
  - Between a  $f := \text{load } fa$  and the next instruction
  - Between a  $\text{store } f, fa$  and the preceding instruction
- Spilling reduces the live range of **f**
  - And thus reduces its interferences
  - Which results in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

32



## Recompute RIG After Spilling

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



33

## Spilling Notes

- 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

34

## Precolored Nodes

- Precolored nodes are nodes which are *a priori* bound to actual machine registers
- These nodes are usually used for some specific (time-critical) purpose, e.g.:
  - for the frame pointer
  - for the first  $N$  arguments ( $N=2,3,4,5$ )

35

## Precolored Nodes (Cont.)

- For each color, there should be only one precolored node with that color; all precolored nodes usually interfere with each other
- We can give an ordinary temporary the same color as a precolored node as long as it does not interfere with it
- However, we cannot simplify or spill precolored nodes; we thus treat them as having "infinite" degree

36

## Effects of Global Register Allocation

### Reduction in % for MIPS C Compiler

Program	cycles	total loads/stores	scalar loads/stores
boyer	37.6	76.9	96.2
diff	40.6	69.4	92.5
yacc	31.2	67.9	84.4
nroff	16.3	49.0	54.7
ccom	25.0	53.1	67.2
upas	25.3	48.2	70.9
as1	30.5	54.6	70.8
<b>Geo Mean</b>	28.4	59.0	75.4

37

## Managing 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

38

## Cache Optimization

- Consider the loop

```
for (j = 1; j < 10; j++)
  for (i = 1; i < 1000; i++)
    a[i] *= b[i]
```
- This program has terrible cache performance
  - Why?

39

## Cache Optimization (Cont.)

- Consider now the program:

```
for (i = 1; i < 1000; 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*

40

## 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 (with many variations on the heuristics)
- Register allocation is more complicated for CISC machines