

## Register Allocation

### Lecture 16

## Lecture Outline

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

## The Memory Hierarchy

Registers	1 cycle	256-8000 bytes
Cache	3 cycles	256k-40MB
Main memory	20-100 cycles	4GB-32+G
Disk	0.5-5M cycles	1-10TB's

## Managing the Memory Hierarchy

- Most 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

- Power usage limits
  - Size and speed of registers/caches
  - Speed of processors
- But
  - The cost of a cache miss is very high
  - Typically requires 2-3 caches to bridge fast processor with large main memory
- It is very important to:
  - Manage registers properly
  - Manage caches properly
- Compilers are good at managing registers

## The Register Allocation Problem

- Intermediate code uses unlimited temporaries
  - Simplifies code generation and optimization
  - Complicates final translation to assembly
- Typical intermediate code uses too many temporaries

## The Register Allocation Problem (Cont.)

- 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

## History

- Register allocation is as old as compilers
  - Register allocation was used in the original FORTRAN compiler in the '50s
  - Very crude algorithms
- A breakthrough came in 1980
  - Register allocation scheme based on graph coloring
  - Relatively simple, global and works well in practice

## An Example

- Consider the program
  - $a := c + d$   
 $e := a + b$   
 $f := e - 1$
- Can allocate  $a$ ,  $e$ , and  $f$  all to one register ( $r_1$ ):
  - $r_1 := r_2 + r_3$   
 $r_1 := r_1 + r_4$   
 $r_1 := r_1 - 1$
- Assume  $a$  and  $e$  dead after use
  - Temporary  $a$  can be "reused" after  $e := a + b$
  - So can temporary  $e$
- A dead temporary is not needed
  - A dead temporary can be reused

## The Idea

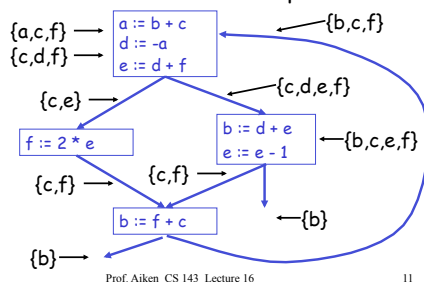
Temporaries  $t_1$  and  $t_2$  can share the same register if at any point in the program at most one of  $t_1$  or  $t_2$  is live.

Or

If  $t_1$  and  $t_2$  are live at the same time, they cannot share a register

## Algorithm: Part I

- Compute live variables for each point:

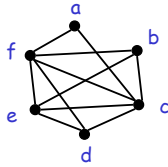


## The Register Interference Graph

- 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

## Example

- For our example:



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

Prof. Aiken CS 143 Lecture 16

13

## Notes on Register Interference Graphs

- Extracts exactly the information needed to characterize legal register assignments
- 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

Prof. Aiken CS 143 Lecture 16

14

## 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

Prof. Aiken CS 143 Lecture 16

15

## Register Allocation Through Graph Coloring

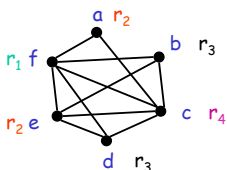
- In our problem, colors = registers
  - We need to assign colors (registers) to graph nodes (temporaries)
- Let  $k$  = number of machine registers
- If the RIG is  $k$ -colorable then there is a register assignment that uses no more than  $k$  registers

Prof. Aiken CS 143 Lecture 16

16

## Graph Coloring Example

- Consider the example RIG

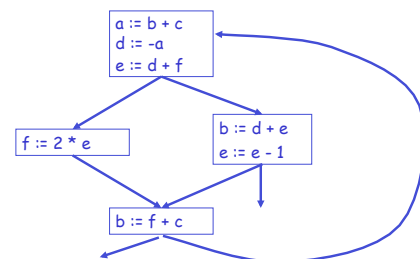


- There is no coloring with less than 4 colors
- There are 4-colorings of this graph

Prof. Aiken CS 143 Lecture 16

17

## Example Review

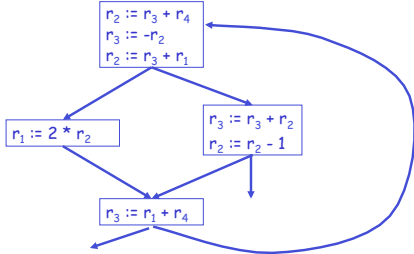


Prof. Aiken CS 143 Lecture 16

18

### Example After Register Allocation

- Under this coloring the code becomes:



Prof. Aiken CS 143 Lecture 16

19

### Computing Graph Colorings

- How do we compute graph colorings?
- It isn't easy:
  - This problem is very hard (NP-hard). No efficient algorithms are known.
    - Solution: use heuristics
  - A coloring might not exist for a given number of registers
    - Solution: later

Prof. Aiken CS 143 Lecture 16

20

### Graph Coloring Heuristic

- Observation:
  - Pick a node  $\dagger$  with fewer than  $k$  neighbors in RIG
  - Eliminate  $\dagger$  and its edges from RIG
  - If resulting graph is  $k$ -colorable, then so is 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

Prof. Aiken CS 143 Lecture 16

21

### Graph Coloring Heuristic

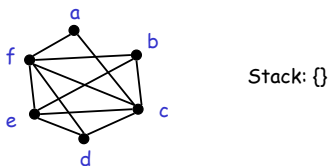
- The following works well in practice:
  - Pick a node  $\dagger$  with fewer than  $k$  neighbors
  - Put  $\dagger$  on a stack and remove it from the RIG
  - Repeat until the graph has one node
- Assign colors to nodes on the stack
  - Start with the last node added
  - At each step pick a color different from those assigned to already colored neighbors

Prof. Aiken CS 143 Lecture 16

22

### Graph Coloring Example (1)

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

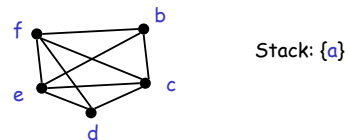


- Remove  $a$

Prof. Aiken CS 143 Lecture 16

23

### Graph Coloring Example (2)



- Remove  $d$

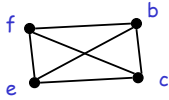
Prof. Aiken CS 143 Lecture 16

24

### Graph Coloring Example (3)

---

- Note: all nodes now have fewer than 4 neighbors



Stack: {d, a}

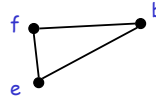
- Remove c

Prof. Aiken CS 143 Lecture 16

25

### Graph Coloring Example (4)

---



Stack: {c, d, a}

- Remove b

Prof. Aiken CS 143 Lecture 16

26

### Graph Coloring Example (5)

---



Stack: {b, c, d, a}

- Remove e

Prof. Aiken CS 143 Lecture 16

27

### Graph Coloring Example (6)

---



Stack: {e, b, c, d, a}

- Remove f

Prof. Aiken CS 143 Lecture 16

28

### Graph Coloring Example (7)

---

- Now start assigning colors to nodes, starting with the top of the stack

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

Prof. Aiken CS 143 Lecture 16

29

### Graph Coloring Example (8)

---



Stack: {e, b, c, d, a}

Prof. Aiken CS 143 Lecture 16

30

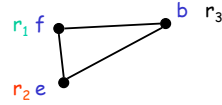
### Graph Coloring Example (9)



Stack: {b, c, d, a}

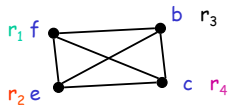
- e must be in a different register from f

### Graph Coloring Example (10)



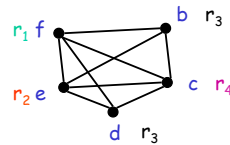
Stack: {c, d, a}

### Graph Coloring Example (11)



Stack: {d, a}

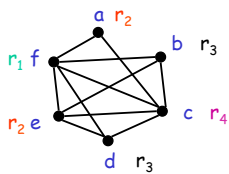
### Graph Coloring Example (12)



Stack: {a}

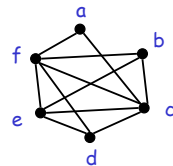
- d can be in the same register as b

### Graph Coloring Example (13)



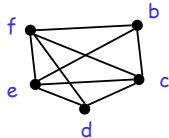
### What if the Heuristic Fails?

- What if 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

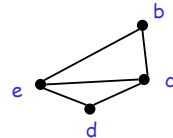


Prof. Aiken CS 143 Lecture 16

37

### What if the Heuristic Fails?

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

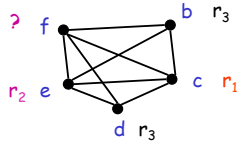


Prof. Aiken CS 143 Lecture 16

38

### What if the Heuristic Fails?

- Eventually we must assign a color to **f**
- We hope that among the 4 neighbors of **f** we use less than 3 colors  $\Rightarrow$  optimistic coloring



Prof. Aiken CS 143 Lecture 16

39

### Spilling

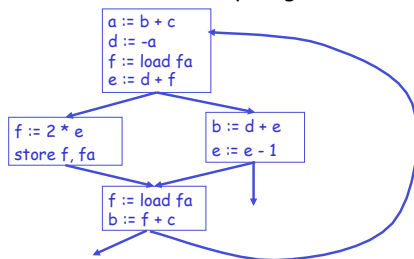
- If optimistic coloring fails, we spill **f**
  - Allocate a memory location for **f**
    - Typically in the current stack frame
    - Call this address **fa**
- Before each operation that reads **f**, insert **f := load fa**
- After each operation that writes **f**, insert **store f, fa**

Prof. Aiken CS 143 Lecture 16

40

### Spilling Example

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



Prof. Aiken CS 143 Lecture 16

41

### A Problem

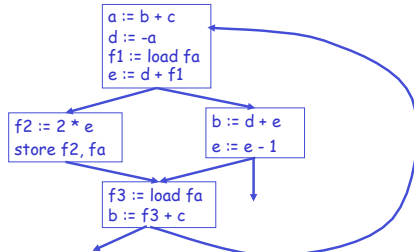
- This code reuses the register name **f**
- Correct, but suboptimal
  - Should use distinct register names whenever possible
  - Allows different uses to have different colors

Prof. Aiken CS 143 Lecture 16

42

## Spilling Example

- This is the new code after spilling  $f$

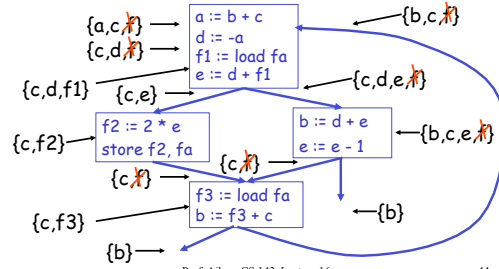


Prof. Aiken CS 143 Lecture 16

43

## Recomputing Liveness Information

- The new liveness information after spilling:



Prof. Aiken CS 143 Lecture 16

44

## Recomputing Liveness Information

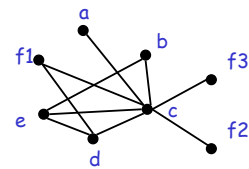
- New liveness information is almost as before
  - Note  $f$  has been split into three temporaries
- $f_i$  is live only
  - Between a  $f_i := \text{load } fa$  and the next instruction
  - Between a  $\text{store } f_i, fa$  and the preceding instr.
- Spilling reduces the live range of  $f$ 
  - And thus reduces its interferences
  - Which results in fewer RIG neighbors

Prof. Aiken CS 143 Lecture 16

45

## Recompute RIG After Spilling

- Some edges of the spilled node are removed
- In our case  $f$  still interferes only with  $c$  and  $d$
- And the resulting RIG is 3-colorable



Prof. Aiken CS 143 Lecture 16

46

## Spilling Notes

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

Prof. Aiken CS 143 Lecture 16

47

## 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 how much a compiler can do to improve cache performance
- Compilers can, and a few do, perform some cache optimizations

Prof. Aiken CS 143 Lecture 16

48



## 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?

Prof. Aiken CS 143 Lecture 16

49

## Cache Optimization (Cont.)

---

- Consider the program:

```
for(i=1; i<1000; i++)
  for(j := 1; j < 10; j++)
    a[i] *= b[j]
```

  - 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

Prof. Aiken CS 143 Lecture 16

50

## Conclusions

---

- Register allocation is a “must have” in compilers:
  - Because intermediate code uses too many temporaries
  - Because it makes a big difference in performance
- Register allocation is more complicated for CISC machines

Prof. Aiken CS 143 Lecture 16

51