COSE312: Compilers

Lecture 17 — Register Allocation[1]

Hakjoo Oh
2025 Spring

[1]Slides adapted from the materials by Alex Aiken at Stanford University

Back-End of a Compiler

Generates the target machine code from IR:

- A key component of compiler back-end is register allocation.
- The remaining translation from IR to machine code is technically not difficult.

Register Allocation

- Intermediate representation (IR) 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
- Example:

```
a := c + d          r1 := r2 + r3
e := a + b          r1 := r1 + r4
f := e - 1          r1 := r1 - 1
```

  (assume `a` and `e` dead after use)
- A dead temporary can be reused.

Register Allocation

- Register allocation is as old as compilers
  - Register allocation was used in the original FORTRAN compiler in the 1950s
  - Very crude algorithms
- A breakthrough came in 1980
  - Register allocation scheme based on graph coloring
  - Relatively simple, global and works well in practice

Register Allocation

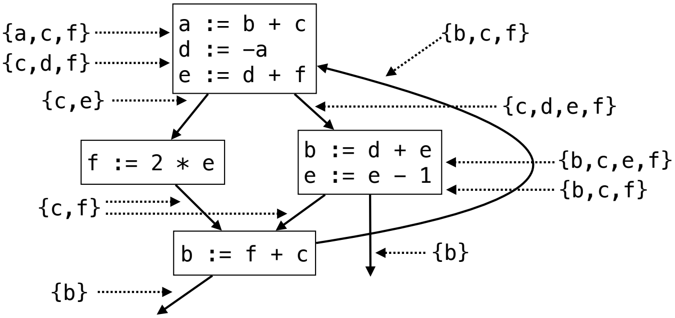
Temporaries t_1 and t_2 can share the same register if at any point in the program at most one of t_1 or t_2 is live.

Or

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

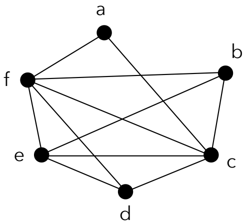
Example

Compute live variables for each point: e.g.,



Register Allocation

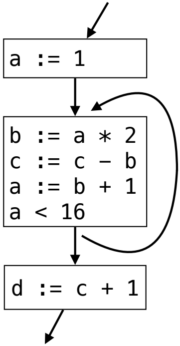
- Construct an undirected graph
  - A node for each temporary
  - An edge between t_1 and t_2 if they are live simultaneously at some point in the program
- This is the *register interference graph* (RIG).
  - Two temporaries can be allocated to the same register if there is no edge connecting them
- 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

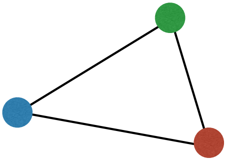
Exercise

Construct the register interference graph:



Graph Coloring

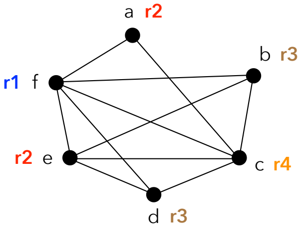
- A coloring of a graph is an assignment of colors to nodes, such that nodes connected by an edge have different colors



- A graph is k-colorable if it has a coloring with k colors.

Graph Coloring

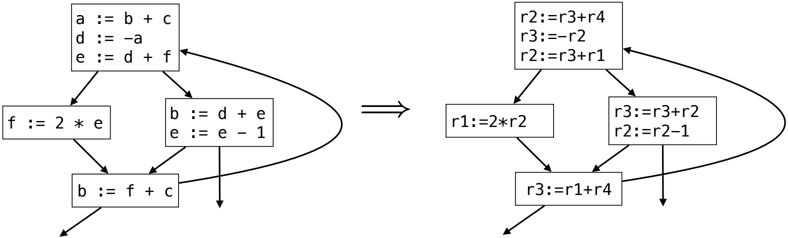
- In our problem, colors = registers
  - We need to assign colors (registers) to graph nodes (temporaries)
- Let k be the number of machine registers
- If the RIG is k-colorable then there is a register assignment that uses no more than k registers
- Consider the example RIG:



(There is no coloring with less than 4 colors)

Graph Coloring

Rename variables by the assigned registers:



Graph Coloring

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

Graph Coloring

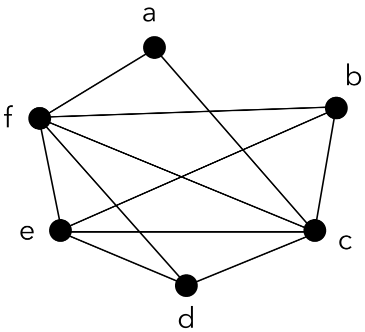
- Observation:
  - Pick a node t with fewer than k neighbors in RIG
  - Eliminate t and its edges from RIG
  - If resulting graph is k-coloring, then so is the original graph
- Why?
  - Let $c_1, \ldots, c_n$ be the colors assigned to the neighbors of t in the reduced graph
  - Since $n < k$, we can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

Graph Coloring

1. Push RIG nodes onto a stack:
   - Pick a node t with fewer than k neighbors
   - Put t on a stack and remove it from the RIG
   - Repeat until the graph is empty
2. 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

Example

Assume $k = 4$:

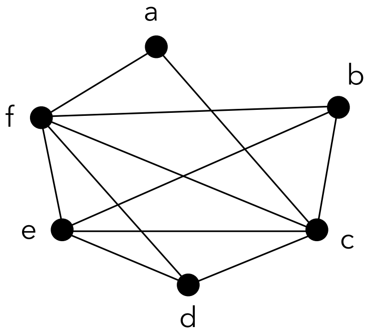


Spilling

- What happens if the graph coloring heuristic fails to find a coloring?
- In this case, we can't hold all values in registers.
  - Some values are *spilled* to memory

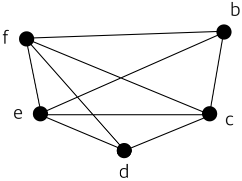
Spilling

- What if all nodes have k or more neighbors?
- Example: Try to find a 3-coloring of the RIG:

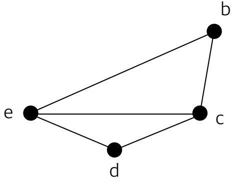


Spilling

- Remove a and get stuck

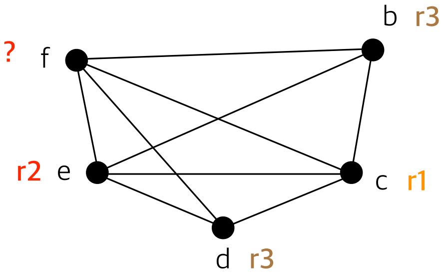


- Pick a node as a candidate for spilling
  - A spilled value "lives" in memory
  - Assume f is chosen
- Remove f and continue the simplification. Simplification now succeeds for b, d, e, c



Spilling

- Eventually, we must assign a color to f
- We hope that among the 4 neighbors of f we use less than 4 colors ("optimistic coloring")



Spilling

- If optimistic coloring fails, we spill f
  - Allocate a memory location for f
  - Call this address a
- Before each operation that reads f, insert

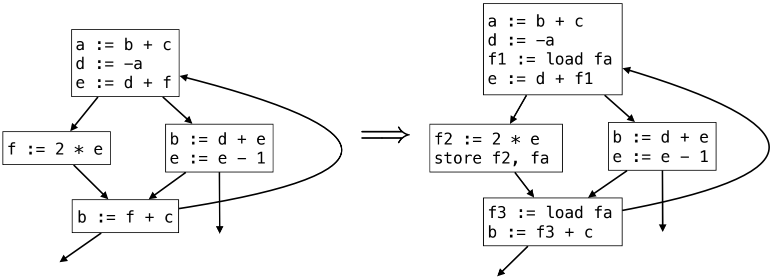
$$f := \text{load } a$$

- Before each operation that writes f, insert

$$\text{store } f, a$$

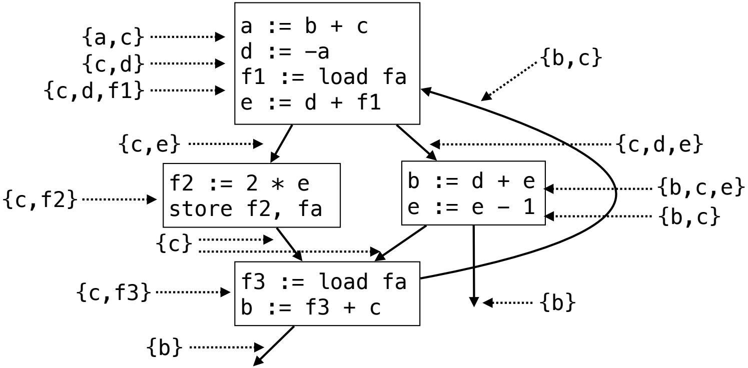
Example

The code after spilling f:



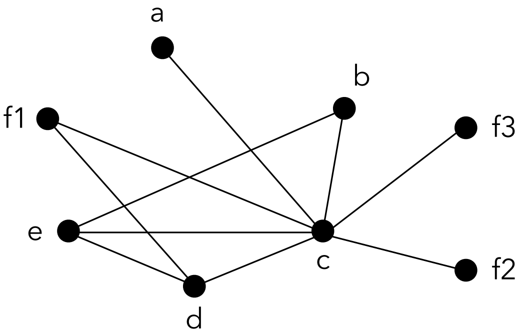
Example

Recompute liveness:



Spilling

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



Spilling

- 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

Summary

- Register allocation is a "must have" in compilers
  - Because intermediate code uses too many temporaries
  - Because it makes a big difference in performance