EC3204: Programming Languages and Compilers

Lecture 18 — Register Allocation

Sunbeom So

Fall 2024

Back-End

Generate the target machine code from IR:

- A key component of the back-end is register allocation.
- Thus, we focus on covering register allocation.

Register Allocation

- Rewrite the intermediate code to use temporaries no more than the number of machine registers available.
- Why? IR translation introduces many temporaries, but we typically have a smaller, limited number of registers.

(Example) Assuming a and e dead after use, they can be allocated to the same register without changing the original semantics.

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

$\Longrightarrow$

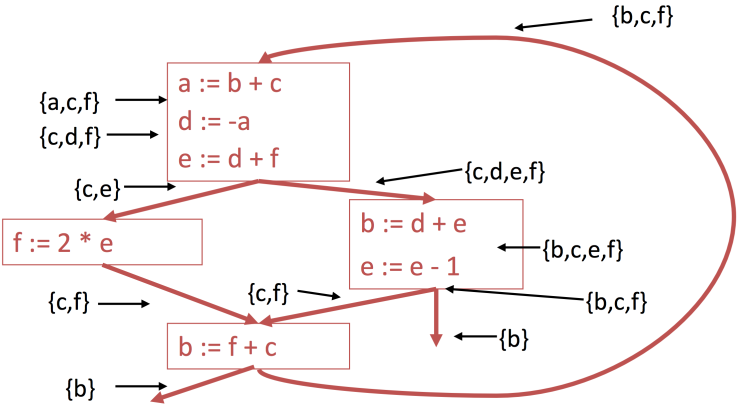
```
r1 := r2 + r3
r1 := r1 + r4
r1 := r1 - 1
```

Basic Idea & Workflow

- Basic Idea:
  - If at most one of t_1 or t_2 is live at any point in the program, they can share the same register.
  - If t_1 and t_2 are live at the same time, they cannot share the same register.
- Workflow:
  - Step 1. Liveness analysis
  - Step 2. RIG (register interference graph) construction
  - Step 3. Register allocation via graph coloring

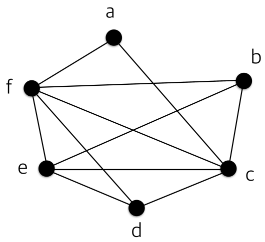
Example: Liveness Analysis

Compute live variables for each point:



Example: RIG construction

- Construct a **register interference graph** (RIG), an undirected graph such that
  - A node for each temporary
  - An edge between t_1 and t_2 if they are live simultaneously at some program point
- For our example:

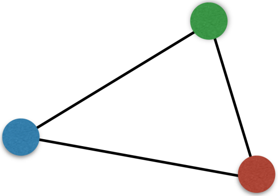


  - Interpretation: If there is no edge connecting some two temporaries, they can be allocated to the same register.
  - E.g., b and c cannot be in the same register
  - E.g., b and d could be in the same register

Graph Coloring

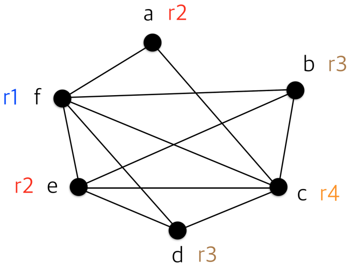
- We reduce the problem of register allocation to the graph coloring problem.
- Graph coloring: find an assignment of colors to nodes, such that nodes connected by an edge have different colors.
- A graph is k-colorable if it has a coloring with k colors.

For example, the below graph is 3-colorable, but not 2-colorable.



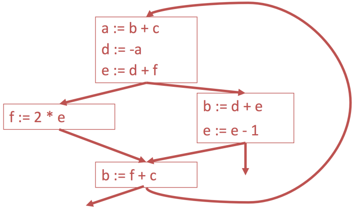
Register Allocation with Graph Coloring

- 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, there is a register assignment that uses no more than k registers
- (Example) The following RIG is 4-colorable.

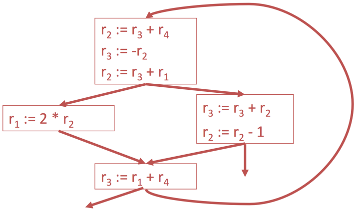


Example: Register Allocation with Graph Coloring

Using the colored graph, we can transform the IR



into



Intuition behind Graph Coloring

- How do we compute graph coloring?
- The problem is NP-hard, but there is a heuristic algorithm that works well in practice.
- The heuristic is made based on the observation:
  - Pick a node t with fewer than k neighbors in RIG.
  - Eliminate t and its edges from RIG.
  - If resulting graph is k-colorable, then so is the original graph.
- Why does the last statement hold?
  - If the neighbors of t are colored with fewer than k colors, we can choose a t's color different from the neighbors' colors.

Steps in Graph Coloring

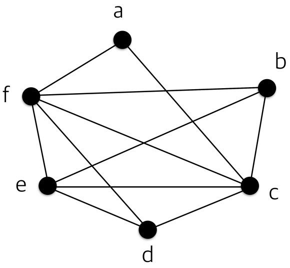
Graph coloring proceeds in two steps.

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.

Step 1 is responsible for checking whether RIG is k-colorable or not, and Step 2 is responsible for assigning colors.

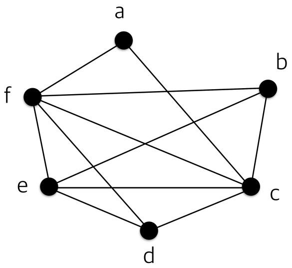
Example

Assume $k = 4$:



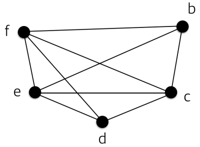
What Happens if the Coloring Heuristic Fails?

- In such cases, we cannot hold all variables in available registers. Some variables may need to be **spilled** to memory.
  - Spilling: storing a variable to memory (slower than registers).
- (Example) Find a 3-coloring of the RIG.

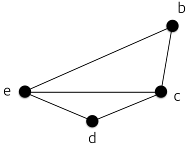


Spilling

- Remove a and get stuck.
  - Why? All nodes have 3 or more neighbors.

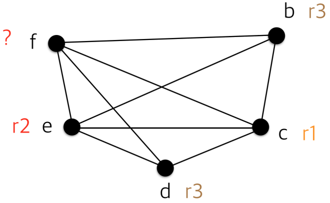


- At this point, we pick a node as a candidate for spilling.
  - Assume f is chosen.
  - A spilled value "lives" in memory.
- Remove f and continue the process. The coloring now succeeds for b, d, e, c.



Spilling

- Since our ultimate goal is to assign colors to every variable, we just try assigning a register to f ("optimistic coloring").



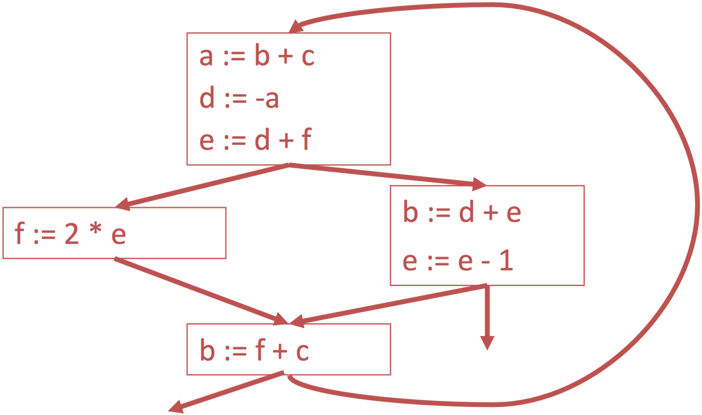
- If optimistic coloring fails, we end up spilling f.
  - Allocate a memory location a for f.
- Before each operation that reads (uses) f, insert

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

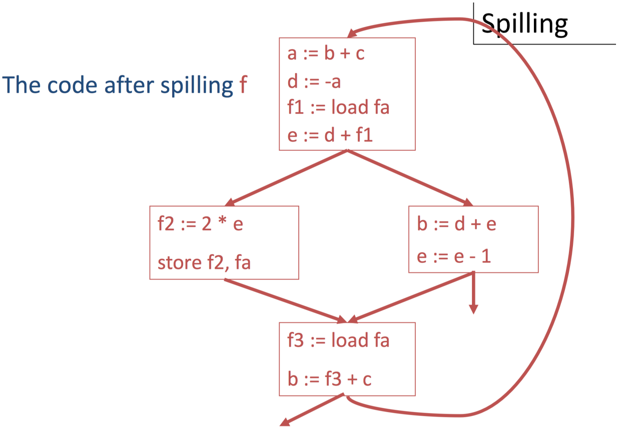
- Before each operation that writes (defines) f, insert

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

Example: IR Before Spilling

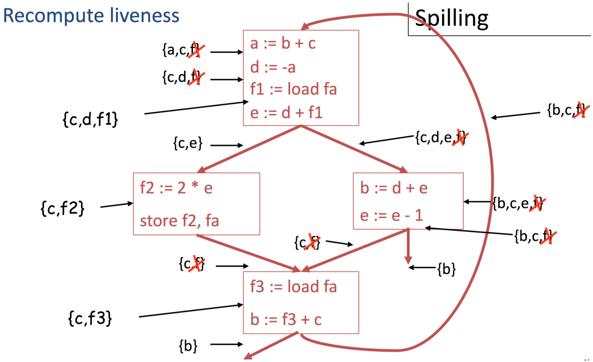


Example: IR After Spilling



In spilling, f has been split into three temporaries (f_1, f_2, f_3) to reduce potential interference.

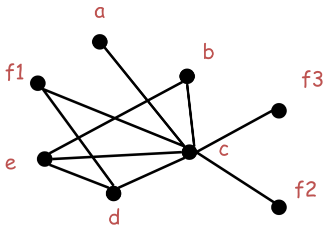
Example: Recomputing Liveness After Spilling



- 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
  - Reduces f's interferences (i.e., RIG neighbors).
  - Thus more likely to succeed in coloring.

Example: RIG After Spilling

- The resulting RIG is 3-colorable.



More on Spilling

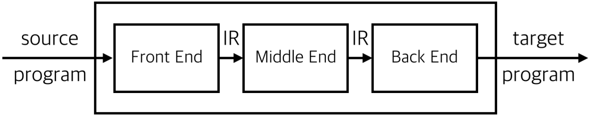
- Additional spills might be required before a coloring is found.
- The tricky part is deciding what to spill, but any choice is correct (only affects performance).
- Possible spilling heuristics:
  - Spill temporaries with most conflicts.
  - Spill temporaries with few definitions and uses.
  - Avoid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

Summary

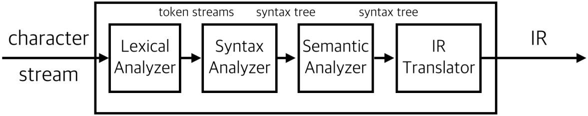
- Register allocation is a must-have in compilers since IR translation introduces too many temporaries.
- Workflow
  - Step 1. Liveness analysis
  - Step 2. RIG construction
  - Step 3. Register allocation via graph coloring
    - If spilling is done, go to Step 1.

Wrap-Up

Compiler architecture:



Frontend:



Topics not covered (sorry!):

- Proving soundness of semantic analyzer (SAA Ch. 7.3)
- Proving correctness of IR translation (SAA Ch. 4)