Register Allocation

CMPT 379: Compilers
Instructor: Anoop Sarkar
anoopsarkar.github.io/compilers-class

Register Allocation

- Intermediate code uses unlimited temporaries
  - Simplifying code generation and optimization
  - Complicates final translation to assembly

Register Allocation

- The problem:

  Rewrite the intermediate code to use no more temporary locations than there are machine registers

- Method:
  - Assign multiple temporaries to each register
  - But without changing the program behavior

Example

- Consider the program

  $a = c + d$

  $e = a + b$

  $f = e - 1$

- Assume a & e dead after use
  - "dead" means it is never used
  - A dead temporary location can be "reused"

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

  $r_1 = r_2 + r_3$

  $r_1 = r_1 + r_4$

  $r_1 = r_1 - 1$

History

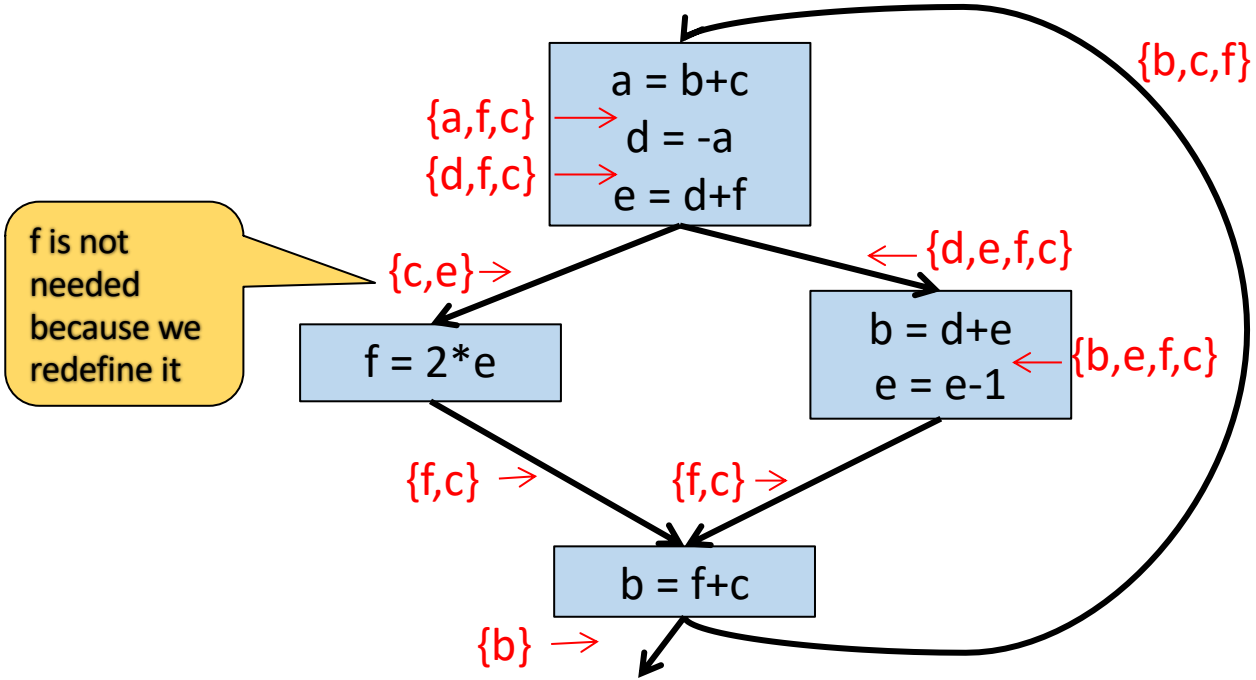
- Register allocation is as old as compilers
  - Register allocation was used in the original FORTRAN compiler in 1950's
  - Very crude algorithm
- A breakthrough came in 1980
  - Register allocation scheme based on graph coloring
  - Relatively simple, global and works well in practice

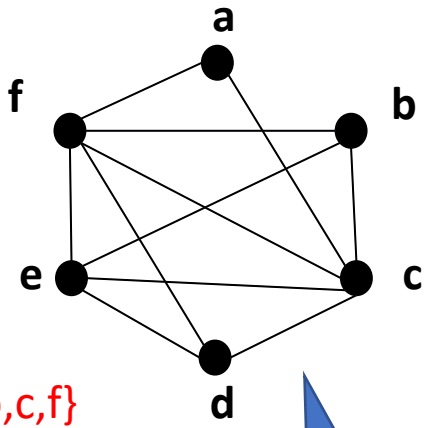
Principles of Register Allocation

- *Temporaries t_1 can 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*
  - *If t_1 and t_2 are live at the same time, they cannot share a register*
- We need liveness analysis: which locations are live at the same time?

Live Variables

- Compute live variables for each point





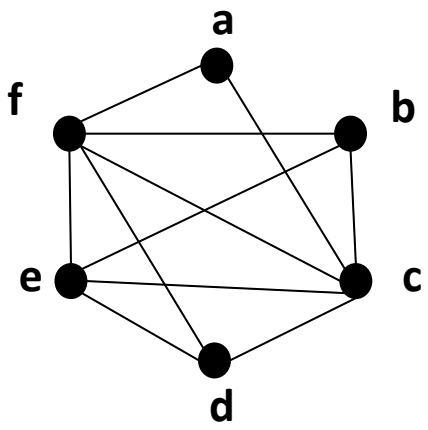
Graph coloring = register allocation

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

Register Interference Graph

- For our example



- a and c cannot be in the same register
- a and d could be in the same register

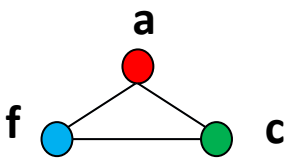
Register Interference Graph

- Extracts exactly the information we need to characterize legal register allocation
- Gives the global view (i.e., over the entire control flow graph) of the register requirements
- After RIG construction the register allocation algorithm is architecture independent

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

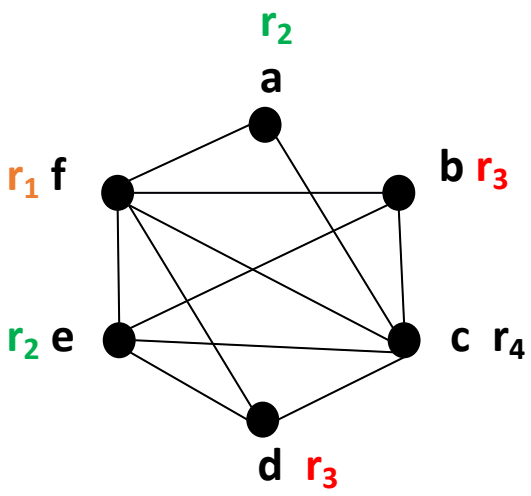


Register Allocation as Graph Coloring

- 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

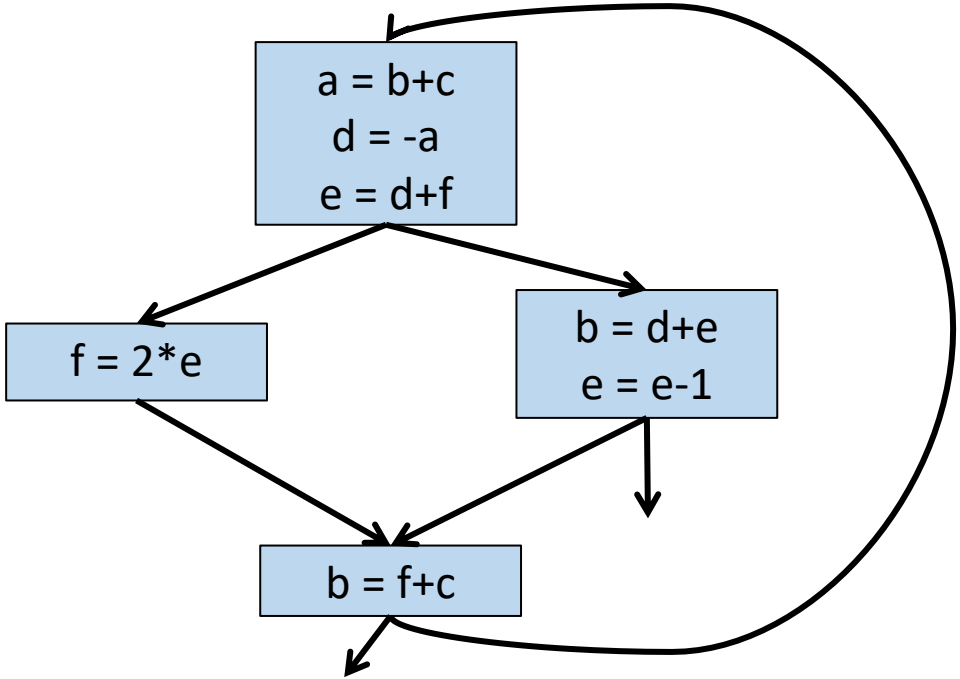
Example

- For our example

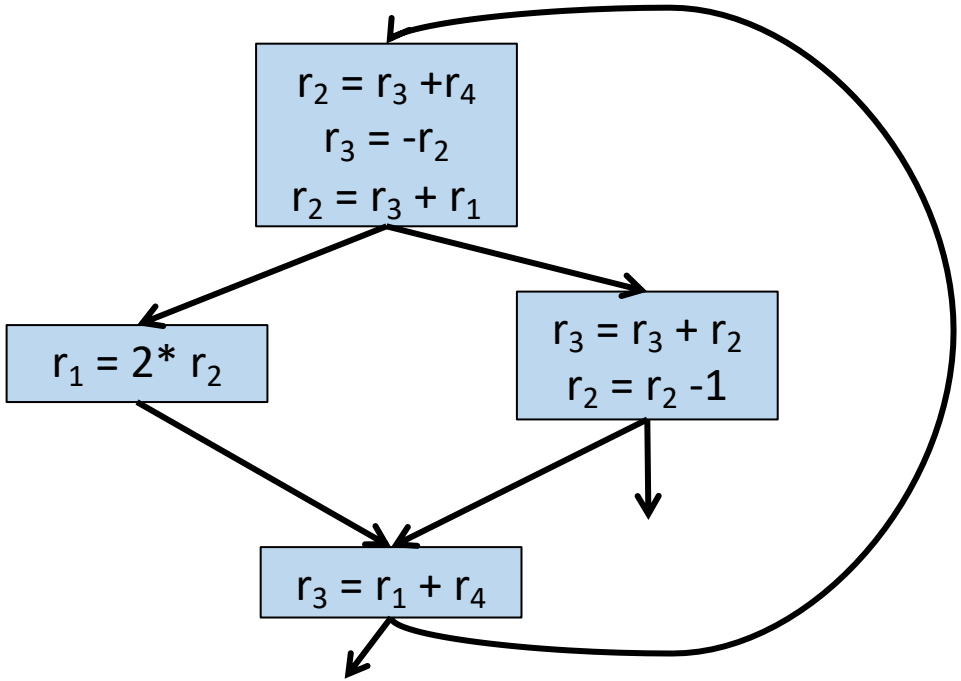


- There is no coloring with less than 4 colors
- There is a 4-coloring of this graph

Control Flow Graph



Register Allocation



Graph Coloring

- How do we compute graph coloring?
- It is not easy :
  - The problem is NP-hard. No efficient algorithms are known
    - Solution: use heuristics
  - A coloring might not exist for a given number of registers
    - Solution: register spilling to memory

Register Allocation as Graph Coloring

- Main idea for solving whether a graph G is *k*-colorable:
- Pick any node *t* with fewer than *k* neighbors
- Remove *n* adjacent edges of node *t* to create a new graph G'
- If G' is *k*-colorable, then so is G (the original graph)
- Let $c_1,\ldots,c_n$ be the colors assigned to the neighbors of t in G'
- Since $n<k$ we can pick some color for *t* that is different from its neighbors

Register Allocation as Graph Coloring

- Heuristic for graph coloring:
  - Ordering nodes (in a stack)

  1. Pick a node t with fewer than k neighbors
  2. Put *t* on a stack and remove it from the register interference graph (RIG)
  3. Repeat until the graph is empty

  - Assigning color to nodes on the stack:

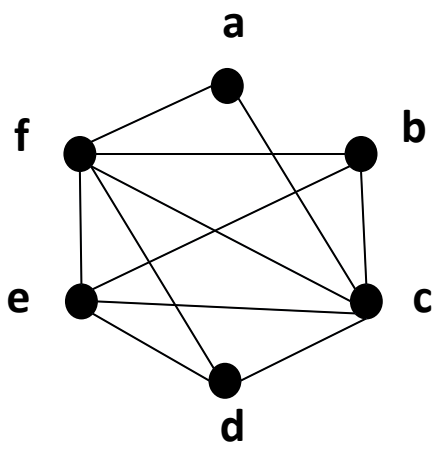
  1. Start with the last node added
  2. At each step pick a color different from those assigned to already colored neighbors

Example

- Assume k=4

Remove a

stack={}

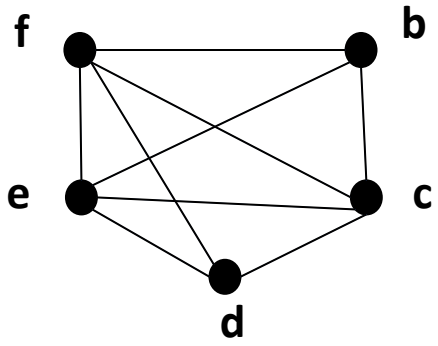


Example

- Assume k=4

Remove d

stack={a}

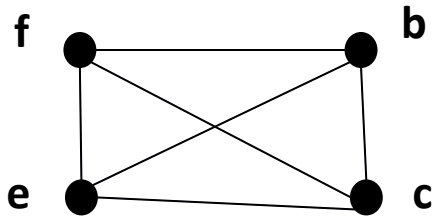


Example

- Assume k=4

  All nodes now have fewer than 4 neighbors

  The graph coloring is guaranteed to succeed



  Remove c

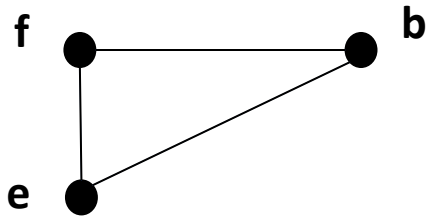
  stack={d,a}

Example

- Assume k=4

Remove b

stack={c,d,a}

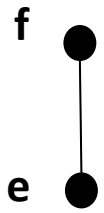


Example

- Assume k=4

Remove e

stack={b,c,d,a}



Example

- Assume k=4

Remove f

stack={e,b,c,d,a}

Example

- Assume k=4

  Empty graph – done with the first part

  Now we have the order for assigning colors to nodes, start coloring the nodes (from the top of the stack)

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

Example

- Assume k=4

stack={e,b,c,d,a}

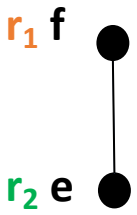
r_1 **f**

Example

- Assume k=4

  e must be in a different register from f

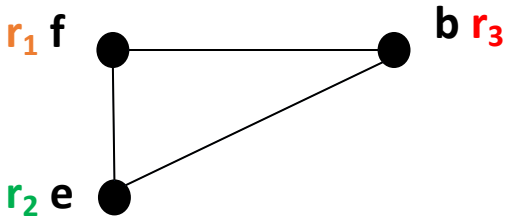
  stack={b,c,d,a}



Example

- Assume k=4

stack={c,d,a}

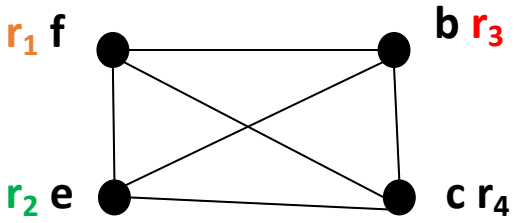


Example

- Assume k=4

  The ordering insures we can find a color for all nodes

  stack={d,a}

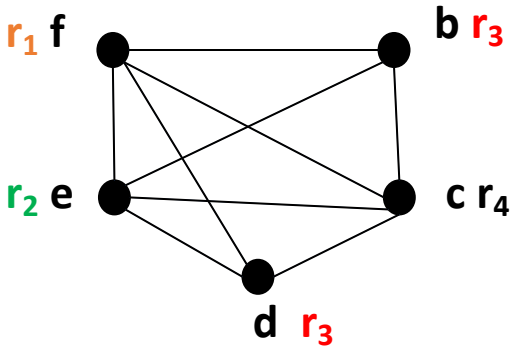


Example

- Assume k=4

  d can be in the same register as b

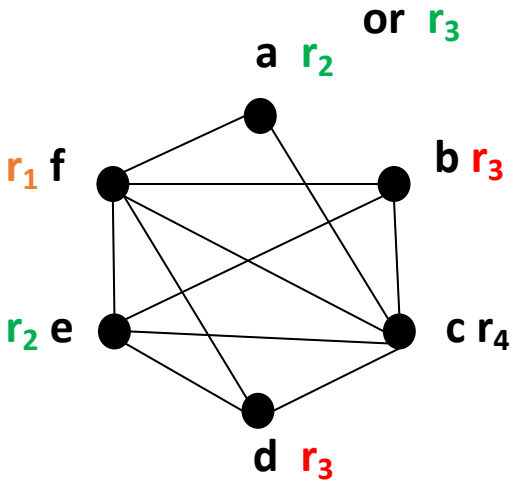
  stack={a}



Example

- Assume k=4

stack={}



Summary

- Register allocation is a “must have” in compilers, because:
  - Intermediate code uses too many temporaries
  - It makes a big difference in performance
- Register allocation can be reduced to a graph colouring problem where the number or registers equals the number of colours.