Today:

Liveness analysis, register allocation

Assignment 10

2 parts of same liveness range

```plaintext
i = 3
j = i/2
if (j == 0) goto exit

exit:
Either assignment may define i for j = i/2
```
Intuitions…

Goal of liveness analysis: to assign registers to temps

Observation: any use point of a temp can be statically assigned only one register

Observation: if multiple defs flow into single use, must share register

if (some condition) { x = 1; } // line 1
else { x = 2; } // line 2
y = x + 5; // line 3

Definitions of x at line 1 and 2 can both flow into use at line 3.

Each definition of x must be assigned same register

Moral: liveness ranges involving same use point must be unified
Register Allocation: overview

Want to maximize use of registers

Liveness ranges determine lifetimes of particular values

If ranges of lifetimes overlap, must store values in distinct registers

---

Technique:

Using liveness ranges, build interference graph

Interference graph:
- Nodes: locations
- Edges: undirected; connect locations with overlapping liveness ranges

---

Given interference graph, use graph coloring algorithm to assign registers
- Graph coloring: adjacent nodes must have different color
- Number of colors = number of registers

Solution to one problem obtained by recasting to another solved problem

True Computer Science!
Data flow eq’ns use 4 sets per node
use,def defined by instruction/stmt
in,out determined by data flow eq’ns
Solve (b-ward) over CFG

out based on immed succ’s in
\[ \text{out}[n] = \bigcup_{s \in \text{succ}(n)} \text{in}[s] \]
in based on own used, def and out
\[ \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \]

Interference graph constructed on the
foundation of data flow analysis…

consider following instructions
0  mov 1023,t100
loop:
1  and t100,256,t101
2  add t101,t102
3  sub t100,4,t100
4  cmp t102,100
5  bit loop
6  nop
assume no usage outside loop
liveness range for t100 is 0,1,2,3,4,5,6
liveness range for t101 is 1,2
liveness range for t102 is 2,3,4
can build interference graph from this

```assembly
0    mov 1023,t100
loop:
1    and t100,256,t101
2    add 1,t101,t102
3    sub t100,4,t100
4    cmp t102,100
5    blt loop
6    nop
```

liveness range for t100 is 0,1,2,3,4,5,6
liveness range for t101 is 1,2
liveness range for t102 is 2,3,4

why doesn't t101 interfere with t102?

```assembly
0    mov 1023,t100
loop:
1    and t100,256,t101
2    add 1,t101,t102
3    sub t100,4,t100
4    cmp t102,100
5    blt loop
6    nop
```

liveness range for t100 is 0,1,2,3,4,5,6
liveness range for t101 is 1,2
liveness range for t102 is 2,3,4

start/end in “middle” of stmts

```assembly
0    mov 1023
loop:
1
2
3
4
5
6
```

t100 is live almost all code
liveness range for t100 is 0,1,2,3,4,5,6
consider liveness for \( t_{101} \)

0  \hspace{1cm} \text{mov} \hspace{1cm} 1023, t_{100}

loop:
1  \hspace{1cm} \text{and} \hspace{1cm} t_{100}, 256, t_{101}
2  \hspace{1cm} \text{add} 1, t_{101}, t_{102}
3  \hspace{1cm} \text{sub} \hspace{1cm} t_{100}, 4, t_{100}
4  \hspace{1cm} \text{cmp} \hspace{1cm} t_{102}, 100
5  \hspace{1cm} \text{b} \hspace{1cm} \text{lt} \hspace{1cm} \text{loop}
6  \hspace{1cm} \text{n} \hspace{1cm} \text{op}

liveness range for \( t_{101} \) is 1,2

\[ t_{101} \text{ does not interfere with } t_{102} \]

0  \hspace{1cm} \text{mov} \hspace{1cm} 1023, t_{100}

loop:
1  \hspace{1cm} \text{and} \hspace{1cm} t_{100}, 256, t_{101}
2  \hspace{1cm} \text{add} 1, t_{101}, t_{102}
3  \hspace{1cm} \text{sub} \hspace{1cm} t_{100}, 4, t_{100}
4  \hspace{1cm} \text{cmp} \hspace{1cm} t_{102}, 100
5  \hspace{1cm} \text{b} \hspace{1cm} \text{lt} \hspace{1cm} \text{loop}
6  \hspace{1cm} \text{n} \hspace{1cm} \text{op}

liveness of \( t_{101} \) ends in beginning of 2
liveness of \( t_{102} \) starts with end of 2

Interference graph
used by register allocation

Two temps that interfere
must be in distinct registers

Two temps that don’t interfere
can share same register
To compute interference graph, use data flow sets
For simple version, only def and out
Refinements possible

First approximation: every definition interferes with every other output

```c
foreach instruction i
    foreach d in def
        foreach o in out
            if d != o
                add edge (d,o)
```

Move instructions
special case: same value on both sides of move
Variables can share register if always have same value

```c
j = i;
if(i);
g(j);
x = i + j;
```

i and j always same can be merged

Don’t add edges for source, dest of move

```c
foreach instruction i
    foreach d in def
        foreach o in out
            if d != o and o not in use
                add edge (d,o)
```

Handling of non-move instructions remains same

If not always same value other instruction will add edge
Also want to remember all move instrs
Register allocator preference same reg for source,dest of any move instr
Creates move from, to same reg
Move can then be eliminated
Called coalescing
Removes extra moves created during codegen

Want liveness graph to build interference graph
Why do we want interference graph?
Need to assign registers to each temp; register allocation

Allocate registers using approach called graph coloring
Color interference graph
  No adjacent nodes have same color
Each color maps to a register
Why does a graph coloring mean a valid register allocation?
Two temps adjacent iff
  Must hold values at same time, requiring two distinct locations
Otherwise
  Could be in same register
  Could have same color

Easy shown to be decidable (for finite graphs):

```c
bool chooseColor(G_nodeList * nodes)
if (nodes == NULL)
  return TRUE;
foreach color available
  map nodes->head to color
  if (any conflicts) continue
  if (!chooseColor(nodes->tail))
    continue
  return TRUE
end foreach
return FALSE
```

Unfortunately, \( O(\text{colors}^{\text{nodes}}) \)

Graph coloring problem is NP complete
  No tractable algorithm

But…

Efficient \textit{approximation} algorithms exist for graph coloring
  Any coloring returned by algorithm is valid
  Can’t find coloring in every case
  Good enough in practice
(Assume $k$ registers henceforth)

Simplest algorithm is simplify-select

Based on simple observation:

If node adjacent to $k$ other nodes
how many colors are needed
(in worst case)?

Any node with degree < $k$ can be colored
with $k$ colors
Simplify-select: recursive algorithm

simplify remove node degree < $k$
fail if no more easy nodes
recurse over smaller graph
select color on way back up
does not conflict with any others

1: mov l,a
2: add a,a,c
3: loop:
4: and a,c,b
5: xor c,b,d
6: sub b,d,e
7: or d,e,f
8: add e,a,g
9: xor f,g,c
10: add g,c,a
11: cmp c,a
12: bne loop
13: nop

Lifetimes

a: 1-8+10-13
b: 4-6
c: 2-5+9-13
d: 5-7
e: 6-8
f: 7-9
g: 8-10
1: mov 1,a
2: add a,a,c
3: loop:
4: and a,c,b
5: xor c,b,d
6: sub b,d,e
7: or d,e,f
8: add e,a,g
9: xor f,g,c
10: add g,c,a
11: cmp c,a
12: bne loop
13: nop

assume 3 colors
red, blue, green

degree(g) = 2
remove g
remove g
degree(f) = 2
degree(c) = 2
remove f

remove g
remove f
degree(e) = 2
degree(c) = 2
remove e

remove g
remove f
remove e
degree(d) = 2
degree(c) = 2
remove d
remove g
remove f
remove e
remove d
degree(a) = 2
degree(b) = 2
degree(c) = 2
remove c

degree(a) = 1
degree(b) = 1
remove b

degree(a) = 0
remove a
remove g
remove f
remove e
remove d
remove c
remove b

color a red

remove g
remove f
remove e
remove d
remove c

color b green

remove g
remove f
remove e
remove d

color c blue
remove g
remove f
remove e
color d blue

remove g
remove f
color e green

remove g
color f blue
Coloring not always possible
Nodes with degree $\geq k$ may exist

Indicates not enough registers for live values

Compiler spills temps to memory
removed from graph

Generates less efficient code

When temps spilled to memory, must rewrite code
Insert stores and loads

Iterate liveness liveness analysis over new code
Two iterations usually suffice

More on spilling later.
Assignment 10 a little different
Write 3 classes from scratch
Have interfaces (in javadoc)
Each fairly small
Inherit heavily

Build liveness graph
Build interference graph

New classes to build upon

Graph
  - supports nodes, edges
FlowGraph
  - defines computeForward, computeBackward
InstrGraph
  - build CFG from Instr's
StmtGraph
  - build CFG from Stmt's
InstrLivenessGraph
First class for you to implement
Choose computeForward vs computeBackward
Details of creating nodes

What is a node?

All nodes are instances of GraphNode
supports to/from (or adjacency)
FlowGraph nodes are FGNode
subclass of GraphNode
defines abstract initialize, update
changes display
LivenessNode
concrete subclass of FGNode
implements initialize, update

Last class is Interference
Subclass of Graph
Constructor takes liveness graph
Creates interference graph

Relevant command-line options:
-dump_liveness
-dump_interference
Extra credit:
   Clean up interference graph

Two issues:
   Don’t need reg-reg edges
      won’t be needed for reg-alloc

   Special handling of moves