Assignment 4: Optimizations

15-411: Compiler Design
Nathan Snyder (npsnyder@andrew) and Anand Subramanian(asubrama@andrew)

Due: Tuesday, October 19, 2010 (1:30 pm)

Reminder: Assignments are individual assignments, not done in pairs. The work must be all your own.

You may hand in a handwritten solution or a printout of a typeset solution at the beginning of lecture on Tuesday, October 19. Please read the late policy for written assignments on the course web page. If you decide not to typeset your answers, make sure the text and pictures are legible and clear.

Problem 1: Optimization with SSA (30 points)

In this problem you will perform several optimizations on a program in SSA form. Because this involves producing several very similar control flow graphs, you may be interested in auto-generating graphs. On the website you will find the file that specifies this graph, which you can modify and produce pdfs from using the command sequence
dot -Tsvg ssagraph.dot -o ssagraph.svg
inkscape –file=ssagraph.svg –export-eps=ssagraph.eps
epstopdf ssagraph.eps –outfile ssagraph.pdf

This might be a bit unnecessarily complicated, but I encountered strange issues when doing it in simpler ways (like inkscape systematically removing all copies of the letters x and T from the image o.0)

(a) Write down the control flow graph after applying copy propagation to the graph on the next page

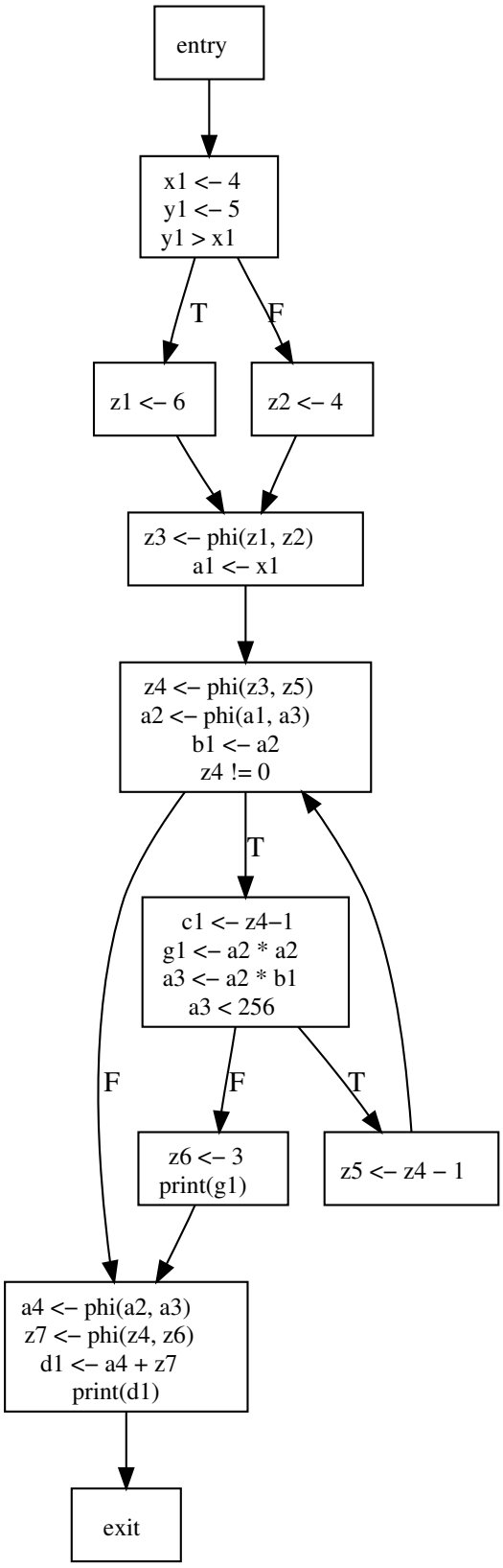
(b) Write down the control flow graph after applying constant propagation and constant folding to the graph from part (a)

(c) Write down the control flow graph after applying common subexpression elimination to the graph from part (b)

(d) Write down the control flow graph after applying dead code elimination to the graph from part (c)

(e) How would the effectiveness of this optimization sequence be affected by applying the optimizations in a different order? You may refer to this case as an example, but your answer should be applicable to programs in general.

(f) Write down the control flow graph after performing deSSA on the **original** control flow graph, and include a brief description of what technique you used to do so.


entry
x1 <– 4
y1 <– 5
y1 > x1
T
F
z1 <– 6
z2 <– 4
z3 <– phi(z1, z2)
a1 <– x1
z4 <– phi(z3, z5)
a2 <– phi(a1, a3)
b1 <– a2
z4 != 0
T
c1 <– z4–1
g1 <– a2 * a2
a3 <– a2 * b1
a3 < 256
F
F
T
z6 <– 3
print(g1)
z5 <– z4 – 1
a4 <– phi(a2, a3)
z7 <– phi(z4, z6)
d1 <– a4 + z7
print(d1)
exit

Problem 2: Jump Tables (20 points)

The switch statement is a language construct found in C but not C0. One way of compiling a switch statement is to treat it as a cascade of if-else statements (with some extra considerations made for letting control flow fall through to the next case if the "break" statement is missing from a case). However, there is a more efficient implementation that can often be used: a jump table. In a jump table, the value being switched on is used as an index into a table of jump statements that each lead to the instructions for a particular case.

(a) Write down an assembly representation of the switch statement below that uses a jump table. We are mostly interested in seeing the concept of a jump table, so feel free to use "abstract" assembly (with variable names, for example) so long as all of your instructions have a clear counterpart on a real processor.

```
unsigned int x, y, z;
// Other code
switch (x) {
    case 0: y = 1; break;
    case 1: y = 8; z = 9; break;
    case 2: y = 3; break;
    default: y = 0;
}
// More code
```

(b) What condition does a switch statement need to meet for a jump table to be used?

(c) Describe how a compiler could transform a switch statement which doesn't meet the necessary condition into one that does.

(d) When would applying your transformation from part (c) not be a good idea?

Problem 3: Bitfields (10 points)

To get the most value out of their memory when using structs, programmers may want to use data sizes that aren't the standard word sizes. Most processors don't support moving arbitrary numbers of bits around, but the same effect can be achieved by storing multiple small fields into one memory word. This introduces some extra complications in using this data, but a compiler can automatically handle this.

(a) Suppose you want to store a 12 bit field named x and a 20 bit field named y in a 32 bit memory word with offset z in the struct s. Write down abstract assembly for both reading and writing x, using a variable r as the source/destination and whatever temps you need.

(b) What problem would this technique face on a multi-core architecture? Give a detailed example.