CS 243 Homework 3

Winter 2023

Due: February 8, 2023 at 11:59pm

Directions:

• Submit written answers via Gradescope.

• Complete the corresponding Gradiance quizzes by the due date.

• You have two late days on assignments for the entire quarter. See course website for details. There are no late days for Gradiance.

• This is an individual assignment. You are allowed to discuss the homework with others, but you must write the solution individually. If you look up any material in the textbook or online, you should cite it appropriately.
**Problem 1. Buggy Processor Mitigation.**

Consider a simplified language where variables can only have values in the set \{0, 1, \ldots, 15\} (i.e., all variables are unsigned 4-bit integers). The only constructs that can define a variable are:

- \(x = c\) where \(c\) is a constant in \{0, 1, \ldots, 15\},
- \(x = y\) where \(x\) and \(y\) are variables, and
- \(x = y + z\) where \(x, y, z\) are variables and \(+\) refers to addition modulo 16 (e.g., \(15 + 1 = 0\)).

Assume all variables are initialized to value 0 at program entry. You may also assume that each instruction is in its own basic block.

Now, suppose your client has a buggy processor that gives faulty results for any addition operations involving the number 7. Your task is to define a dataflow framework with any post-processing steps necessary to discover instructions that can potentially trigger the bug. Design a safe data flow analysis that is as precise as possible.

Additionally, for a program with \(m\) basic blocks (not counting ENTRY or EXIT) and \(n\) variables, what is the maximum number of iterations required until the iterative algorithm converges? Provide your answer as an upper bound. Assume you know nothing about the node iteration order the algorithm takes, and nothing else about the program.

<table>
<thead>
<tr>
<th>Direction of your analysis (forward/backward)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lattice elements and meaning</td>
<td></td>
</tr>
<tr>
<td>Is there a top element?</td>
<td></td>
</tr>
<tr>
<td>If yes, what is it?</td>
<td></td>
</tr>
<tr>
<td>Is there a bottom element?</td>
<td></td>
</tr>
<tr>
<td>If yes, what is it?</td>
<td></td>
</tr>
<tr>
<td>Meet operator</td>
<td></td>
</tr>
<tr>
<td>Transfer function</td>
<td></td>
</tr>
<tr>
<td>Boundary condition</td>
<td></td>
</tr>
<tr>
<td>Interior points</td>
<td></td>
</tr>
</tbody>
</table>

Post-processing steps:
Bound on iteration count:

---

1This question is inspired by the Intel Pentium FDiv bug.
Problem 2. Partial Redundancy Elimination.

Apply partial redundancy elimination to the following program. For simplicity, we omitted instructions that do not redefine variables; you must assume that all variables are used in all basic blocks shown. As a special case, the expression “read()” can return different values when called at different times, and thus does not participate in the lazy code motion algorithm.

You do not need to show the intermediate steps – just show the optimized code. You may add basic blocks to the flow graph, but only show those that are not empty in your solution. (Existing “…” basic blocks are not empty. They merely do not redefine any variable.)
Problem 3. Your task is to optimize the code below. You are only allowed to run the following four optimization techniques (in any order and multiple times if necessary):

- Dead code elimination using liveness analysis (as discussed in class and Homework 2),
- Constant propagation (as discussed in Lecture 4),
- Partial redundancy elimination (as discussed in Lecture 5),
- Copy propagation (as discussed in § 9.1.5 (pp. 590–591) of the textbook).

You cannot modify the control flow graph or eliminate empty basic blocks, except to preprocess it for PRE. Assume that expressions can take both registers and constants. You should not make any assumptions about the return value of `read()`, and assume `print(Z)` is some function that uses `Z`.

```
entry
X = 9;
B = Q;
R = B + C;
X = X - 7;
V = X + 1;

Q = read();

Q = Q - 1;
M = V + Q;
B = 2 × M;
V = 3;

B = B + M;
C = V + Q;
Z = B × C;
print(Z);
```

exit

1. What are the optimizations and their order to produce the best optimized code for this specific program? Remember that you may run the same optimization more than once. You are allowed to abuse the optimizations, but try your best to minimize the number of passes of optimization to do the job.

2. What is the final optimized program?
Problem 4. Register Allocation.

Perform register allocation for the above control flow graph. Specifically, show the results of each of the following steps:

1. Show the merged live ranges for the following program by updating the graph with unique variable notations, e.g., replace definitions and usages of $B$ with $B_1$, $B_2$, etc. Note: if different definitions form a merged live range, use the same variable notation (you may convince yourself that this avoids ambiguity when a used variable may come from different definitions). We provide you with an example. For the following simple program:
The resulting updated graph is:

![Graph Diagram]

Taking the definitions of $A$ in B2 and B3 as example, $A$ is live at the beginning of B4, and both definitions reach that point, therefore they can be merged. **Note:** you do not need to consider any optimization such as PRE or constant propagation.

2. Draw the register interference graph with edges between nodes that represent merged live ranges. Using the same example program, the register inference graph should look like:

![Interference Graph]

3. Apply the coloring algorithm for a machine with 3 registers to the interference graph from the previous part. Show a possible resulting “stack” of registers and show which ones, if any, are marked as spilled.

4. Assign the merged live ranges (i.e., $A_1, B_1, ...$) to registers. You may assume the three registers are labeled as $R_1, R_2, R_3$. 
Problem 5. Register Allocation and Live Ranges.

Part a. Let $n$ be the largest number of overlapping live ranges seen in a program.

1. Is it possible to assign all variables to registers on a machine with $n - 1$ registers, without spilling? Explain your answer.

2. Is it always possible to find a register allocation on a machine with $n$ registers, without spilling? Explain your answer.

Part b. Observe the control-flow graph below and answer the following questions.

1. What is the largest number of overlapping live ranges seen at any program point?

2. What is the minimum number of registers you need in order to successfully assign all variables without spilling?

3. Now, imagine that, as part of register allocation, you can insert MOVE $x$ $y$ operations that copy a value from register $x$ to another register $y$. Can you allocate all of the variables with fewer registers than before? If so, how many would it require?

```
ENTRY

B = ...
C = ...

print(C)
A = ...
print(B)
C = ...
print(A)
B = ...

EXIT
```
Problem 6. Dominance Relation.

Read through textbook § 9.6.1 (pp. 656–659) to learn about dominators. Then, answer the following questions.

1. Draw the dominator tree for the above graph.

2. A node $d$ strictly dominates a node $n$ if $d$ dominates $n$ and $d \neq n$. The dominance frontier of a basic block $b$, $\text{DF}(b)$, is the set of all blocks $n$ such that (1) $b$ dominates an immediate predecessor of $n$ and (2) $b$ does not strictly dominate $n$. This is the boundary of the flow graph wherein the dominance of $b$ terminates.

   (a) Let $\text{DOM\_BY}(b)$ be the set of all basic blocks dominated by a basic block $b$ and $\text{SUCC}(b)$ be the set of blocks $s$ such that there exists an edge $b \rightarrow s$. Express $\text{DF}(b)$ in terms of $\text{DOM\_BY}$ and $\text{SUCC}$.

   (b) Can a block be in its own dominance frontier? If not, provide a brief explanation. Otherwise, provide an example of such a block in the graph.