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.\footnote{This question is inspired by the Intel Pentium FDIV bug.}

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>Forward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lattice elements and meaning</td>
<td>Mappings of each variable to a subset of {0, 1, \ldots, 15} as potential values</td>
</tr>
<tr>
<td>Meet operator</td>
<td>Pointwise set union (i.e., ((A \land B)[v] = A[v] \cup B[v]))</td>
</tr>
<tr>
<td>Is there a top element? If yes, what is it?</td>
<td>Yes, where ( T[v] = \emptyset ) for all ( v )</td>
</tr>
<tr>
<td>Is there a bottom element? If yes, what is it?</td>
<td>Yes, where ( \bot[v] = {0, 1, \ldots, 15} ) for all ( v )</td>
</tr>
<tr>
<td>Transfer function</td>
<td>For each variable ( x ), ( \text{OUT}[b][x] = )</td>
</tr>
<tr>
<td></td>
<td>{c} if ( b ) is of form ( x = c ),</td>
</tr>
<tr>
<td></td>
<td>( \text{IN}[b][y] ) if ( b ) is of form ( x = y ),</td>
</tr>
<tr>
<td></td>
<td>{( \tilde{y} + \tilde{z} \mid \tilde{y} \in \text{IN}[b][y], \tilde{z} \in \text{IN}[b][z]}} if ( b ) is of form ( x = y + z ),</td>
</tr>
<tr>
<td></td>
<td>( \text{IN}[b][x] ) if LHS variable ( \neq x ).</td>
</tr>
<tr>
<td>Boundary condition</td>
<td>The mapping of all variables to the set {0}</td>
</tr>
<tr>
<td>Interior points</td>
<td>The top element</td>
</tr>
</tbody>
</table>

Post-processing steps: Issue warning for instruction \( b: x = y + z \) if \( 7 \in \text{IN}[b][y] \cup \text{IN}[b][z] \).

Bound on iteration count: \( 16mn \). The longest descending chain in the semilattice (from \( T \) to \( \bot \)) has length \( 16n \). This reflects mutating all \( n \) variables from the empty set to the universal set, adding one value at a time. Additionally, we can pessimistically assume that only one
block changes every iteration – just enough to avoid convergence – and all \( m \) blocks undergo the \( 16n \)-iteration descent. This gives a rough bound of \( 16mn \) iterations. \( 15mn \) (starting from the boundary condition, longest descending chain has length \( 15n \)), \( 15mn + 1 \) (need one more iteration to determine convergence), \( 16mn + 1 \) are also accepted answers.
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 \( \text{read()} \), and assume \( \text{print}(Z) \) is some function that uses \( Z \).

\[
\begin{align*}
X &= 9; \\
B &= Q; \\
R &= B + C; \\
X &= X - 7; \\
V &= X + 1; \\
Q &= \text{read}(); \\
M &= V + Q; \\
Q &= Q - 1; \\
B &= 2 \times M; \\
V &= 3; \\
& \vdots \quad \text{more statements} \\
& \text{print}(Z); \\
& \text{exit}
\end{align*}
\]

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?
1. Note: there is not one unique answer. Your answer should be reasonable enough – you can run an optimization more times than necessary, but running all four optimizations ten times, for example, is not an acceptable answer.

Reference solution: First run constant propagation and DCE to get rid of all statements associated with $R$ and $X$ and replace occurrences of $V$ with 3. After that, apply PRE to the expression $3 + Q$. Next, apply copy propagation to make $C$ dead. Finally, run a pass of DCE to eliminate $C$.

2. 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:

```plaintext
ENTRY
A = 9;
D = 0;
E = 5;

B1

B2
C = A - 2;

B7
C = B + 1;
D = C + 1;
B = D + 6;

B8
D = B - 2;

B4
D = E - 2;
E = 12;

B3
B = A + 5;

B5
C = B + 2;

B9
E = D + E;

B10

EXIT
```

entry
A = 0;
C = 0;

B1

B2
A = 1;
B = 1;

B3
A = 2;

B4
B = A + C;

EXIT
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:

![Inference 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 $R_1, R_2, R_3$. 
1. The graph with updated variable notations:

More specifically, merged live ranges at each program point:

- B1 beginning: empty (could include $C_1$)
- B1 end: $A_1, D_1, E_1$ (could include $C_1$)
- B2 beginning: $A_1, D_1, E_1$ (could include $C_1$)
- B2 end: $C_1, D_1, E_1$
- B3 beginning: $A_1, D_1, E_1$ (could include $C_1$)
- B3 end: $B_2, D_1, E_1$ (could include $C_1$)
- B4 beginning: $C_1, E_1$
- B4 end: $C_1, D_1, E_1$
- B5 beginning: $B_2, D_1, E_1$ (could include $C_1$)
- B5 end: $C_1, D_1, E_1$
- B6 beginning: $C_1, D_1, E_1$
- B6 end: $D_1, E_1$
- B7 beginning: $D_1, E_1$
- B7 end: $B_1, E_1$
- B8 beginning: $B_1, E_1$
- B8 end: $D_1, E_1$
- B9 beginning: $B_1, E_1$
- B9 end: $B_1, E_1$
- B10 beginning: $D_1, E_1$
- B10 end: empty
- within B4: $C_1, D_1$
- within B9, between the first and second instruction: $C_2, E_1$
- within B9, between the second and third instruction: $D_2, E_1$
- within B1, between the first and second instruction: $A_1$ (could include $C_1$)
- within B1, between the second and third instruction: $A_1, D_1$ (could include $C_1$)

For occurrences of “could include $C_1$”, you can either ignore all or apply all. The former choice means you ignore the buggy path B1–B2–B4–B6 and assume $C$ is not defined before entry. The latter choice means you assume $C$ is defined before entry so that there is no buggy path such that $C$ is used before definition. Answers derived from both assumptions are acceptable.

2. The register interference graph:

![Register Interference Graph]

As mentioned above, if you assume $C$ is defined before entry, there will be an edge between $A_1$ and $C_1$ and between $B_2$ and $C_1$. Also, because we did not specify the granularity of merged live ranges for this problem, if you don’t have the edges between $C_2, E_1$ and $D_2, E_1$, the answer is also acceptable (which means you are only considering points between basic blocks to calculate merged live range overlap). It will also not affect the answers to the next two questions. However, in general, you should account for merged live range overlap at a finer granularity (i.e. between instructions).

3. Possible order assuming $C$ undefined before entry: $A_1, B_1, B_2, C_1, D_1, E_1, C_2, D_2, E_2$; possible order assuming the opposite, $B_1, C_2, D_2, E_2, E_1$(spilled), $A_1, D_1, C_1, B_2$

4. Possible assignment assuming $C$ undefined before entry: $R_1 \rightarrow \{A_1, B_1, B_2, C_1, C_2, D_2, E_2\}, R_2 \rightarrow \{D_1\}, R_3 \rightarrow \{E_1\}$. Possible assignment assuming the opposite: $R_1 \rightarrow \{A_1, B_1, B_2, C_2, D_2, E_2\}, R_2 \rightarrow \{C_1\}, R_3 \rightarrow \{D_1\}, E_1$ spilled.
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?

Part a.

1. No. The $n$ overlapping live ranges at some point must be allocated to $n$ distinct registers in order not to spill.

2. Not always. Part b is a perfect example.

Part b.

1. Two live ranges. See the above diagram.
2. Three registers, due to the odd-length cycle $A_1 - B_1 - C_1 - A_1$ in the interference graph.

3. Yes, with only two registers. Example below. The MOVE instruction essentially splits the live range $A_1$ into two.
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 $b$, $DF(b)$, 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 $DF(b)$ 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.
1.

2. (a) \[ \text{DF}(b) = \bigcup_{d \in \text{DOM}_{BY}(b)} (\text{SUCC}(d) - (\text{DOM}_{BY}(b) - \{b\})) \]

(b) Yes, \( b_1 \) (which dominates \( b_{10} \)), \( b_2 \) (which dominates \( b_8 \)), \( b_5 \) (which dominates \( b_7 \)) are examples of a block in its own dominance frontier.