This is an open-book, open-notes, open-laptop, closed-network exam. Please do not post anything on Piazza until the solutions are put up on the class website.

You will have 2 hours to complete the exam, but the exam itself is designed to take no longer than 1 hour and 20 minutes. The examination has 5 problems worth 70 points. Please budget your time accordingly. Write your answers in the space provided on the exam. If you use additional scratch paper, please turn that in as well.

Your Name: ___________________  SUNet ID: ___________________

The following is a statement of the Stanford University Honor Code:

1. The Honor Code is an undertaking of the students, individually and collectively:

   (a) that they will not give or receive aid in examinations; that they will not give or receive unpermitted aid in class work, in the preparation of reports, or in any other work that is to be used by the instructor as the basis of grading;

   (b) that they will do their share and take an active part in seeing to it that others as well as themselves uphold the spirit and letter of the Honor Code.

2. The faculty on its part manifests its confidence in the honor of its students by refraining from proctoring examinations and from taking unusual and unreasonable precautions to prevent the forms of dishonesty mentioned above. The faculty will also avoid, as far as practicable, academic procedures that create temptations to violate the Honor Code.

3. While the faculty alone has the right and obligation to set academic requirements, the students and faculty will work together to establish optimal conditions for honorable academic work.

Signature: ___________________
Problem 1. Short questions. For true or false questions, you must justify your answers in 1 to 2 sentences [10 Points].

1. (3 points) True or false. Suppose we have applied the Lazy Code Motion algorithm as discussed in class to a program. Applying the algorithm again on the program can still change the program.

   False. PRE will not introduce any redundancy to the program, so after detecting any partial redundancy that it can remove, a second round of PRE will not find any new redundancies it can eliminate.

2. (3 points) True or false. If a semi-lattice has an infinite domain with a monotonic framework, then the iterative algorithm will never converge to a fixed-point solution. Ignore the degenerate case when the MFP to the dataflow problem is $\top$ in every program point.

   False. Consider the constant propagation dataflow analysis. The size of its domain is infinite (e.g. all natural numbers with special top and bottom elements) and the transfer function is monotonic. However, the iterative algorithm will still converge. Therefore, the size of the domain does not matter, but rather the length of the chains of elements ($x_1 \leq x_2 \leq ...$). Since the length of the chains are finite, the dataflow will converge even if there are an infinite number of elements in the domain.

3. (4 points) Consider a forward dataflow analysis that is monotone with finite descending chains. Suppose we initialize the output of all non-entry basic blocks to a non top element in the semilattice. Ignoring the trivial case of a straight-line program, could the iterative algorithm (that arbitrarily processes basic blocks) produce the MFP solution for some class of control flow graphs? If so, please define such control flow graphs.

   For any acyclic CFG, there is a topologically sorted order that the iterative algorithm could follow when arbitrarily processing basic blocks. Even if the algorithm traverses basic blocks in an arbitrary order with random initializations, after at most $n$ iterations (where $n$ is the number of basic blocks), the correct dataflow information will eventually propagate from entry to exit.

   We can reason about this inductively, as since the graph is acyclic, there exists a topological sort $v_1, v_2, ..., v_n$. On our first pass, we will process $v_1$ at some point, and since its only predecessor is the entry block, at the end of the first iteration, at least $v_1$ will have the correct value. Then in iteration $i$, we will have processed all the predecessors of block $v_i$, so it will also have the correct value. At the end of iteration $v_n$, regardless of initialization, all basic blocks will have the correct value.
Problem 2. Register Allocation [10 points]

Consider the following interference graph:

```
Suppose we have a machine with 3 registers.

Is it possible to allocate this interference graph? If yes, demonstrate a valid coloring.

(4, R0), (3, R1), (2, R0), (1, R1), (5, R2), (7, R2), (8, R1), (6, R0), (9, R0).

Will the full coloring algorithm presented in class always, sometimes, or never spill to memory while trying to color this graph? Explain.

Sometimes. Consider the following removal order: 9 4 3 2 6 1 8 5 7.

Working backwards: (7, R0), (5, R1), (8, R2), (1, R2). At this point, the algorithm is forced to spill 6.
```
Problem 3. Instruction Scheduling [10 points]

Assume you have a statically scheduled machine that can only issue one operation every clock. All operations have a latency of one clock with the exception of a memory load operation, which has a latency of three clocks. Assume that loads can be speculatively executed and that loads can be pipelined (a new load can be issued a single clock after the previous one).

Consider the following locally scheduled program:

Assume that only k is live at the end of the program and that every variable is initialized with some value at the beginning of the program. Each branch is labelled with the probability that it is taken dynamically. To answer the following, you may apply any of the upward code motions discussed in class (with the additional ability to speculatively hoist code across multiple branches), but no other optimizations.

Given that $p = 0.9$, apply global scheduling to this code; provide the improved code with at least a 20% performance improvement, along with its expected execution time and percentage improvement compared to the original program.
Original execution time: $16p + 16p(1 - p) + 17(1 - p)^2 = 16.01$
New execution time: $12p + 14p(1 - p) + 17(1 - p)^2 = 12.23$
$\frac{16.01-12.23}{16.01} \times 100\% \sim 23.6\%$
**Problem 4.** Partial Redundancy Elimination [15 points]

Show the result of running partial redundancy elimination. What’s the final optimized flow graph? You don’t need to show the intermediate steps.

**Solution:**
Problem 5. Data-flow Analysis [25 points]

Consider a language that supports disk reads and writes, with write buffering. Each file has its own write buffer, which is a temporary area that sequentially records the content of all write operations before committing them to disk. All read operations read from disk.

A flush operation forces the contents stored in a file's write buffer to be written to disk. To ensure all read operations see the latest writes, the programmer should issue a flush operation after the last write operation before a read operation. Your task is to create warnings to help the programmer write the code correctly.

All file operations are performed via file variables. You may assume that all the files used in a program have been created and are given a name, represented as a literal string $s \in S$. Different file names always refer to different files. All relevant operations for this task are listed below:

- $f = \text{open}(s)$: open a file with name $s$, and assign it to $f$, where $s \in S$, the set of all file names used in the program.
- $f\.\text{write}(\ldots)$: write content to the file's write buffer.
- $\text{print}(f\.\text{read}(\ldots))$: read content from the file on disk and print the content.
- $f\.\text{flush}()$: flush the file buffer to disk
- $f\.\text{close}()$: flush the file buffer to disk and close the file

The same file can be opened multiple times and assigned to different file variables.

You may assume no file is open at the beginning of the program. For simplicity, you may assume that the program is correct in that a file is already open whenever it is used in a write, read, flush, or close operation, and that a file variable is always closed before it is used in an open operation.

You can assume that each basic block contains only one instruction.

Design one or more data flow analyses to issue warnings on all read operations that may possibly read from a file whose latest write has not been flushed. For full credit, your analysis should work on general control flow graphs and is precise enough to issue warnings on the following 2 programs as shown.

Example 1:

```python
z = \text{open}("file1")
x = \text{open}("file1")
z\.\text{write}(\ldots)
\text{print}(z\.\text{read}(\ldots)) \quad // \text{Issue a warning here.}
x\.\text{flush}()
\text{print}(z\.\text{read}(\ldots)) \quad // \text{Do NOT issue a warning here.}
```
Example 2:

If you define multiple data flow analyses, answer the following questions for each analysis.

Describe your data flow analysis by specifying the following:

- Direction of your data-flow analysis (forward/backward)
- Elements in the semi-lattice and their meaning
- Meet operator or lattice diagram
- Top and bottom elements if they exist
- Transfer function
- Boundary condition initialization
- Interior points initialization

Solution:

In the first pass, compute \[\text{FILES}[p][x] = \{ s \mid s \in S \text{ a file } \text{, } x \text{ may point to } s \text{ at program point } p \}\].
| **Direction of your analysis** (forward/backward) | Forwards. |
| **Lattice elements and meaning** | For each variable, the set of files it can point to. The final lattice will be a product of these lattices. |
| **Meet operator or lattice diagram** | Union. |
| **Is there a top element?**<br>**If yes, what is it?** | {} |
| **Is there a bottom element?**<br>**If yes, what is it?** | Set of all file names in the program. |
| **Transfer function of a basic block** | \[
\begin{align*}
  f = \text{open}(s) & \quad \text{FILES}[\text{B.out}][f] = \{ s \} \\
  f.\text{close}() & \quad \text{FILES}[\text{B.out}][f] = \{ \} \\
  \text{otherwise} & \quad \text{FILES}[\text{B.out}][f] = \text{FILES}[\text{B.in}][f]
\end{align*}
\] |
| **Boundary condition initialization** | {} |
| **Interior points initialization** | {} |
Direction of your analysis (forward/backward) | Forwards.
---|---
Lattice elements and meaning | For each file name $s$ at program point $p$, the lattice element is True if $s$ must have been flushed since its latest writes, and False otherwise. The final lattice will be a product of these lattices.
Meet operator or lattice diagram | And.
Is there a top element? | True.
If yes, what is it? | True.
Is there a bottom element? | False.
If yes, what is it? | False.
Transfer function of a basic block | $f = \text{open}(s); \text{f.read(...)}$
| $\text{f.write(...)}$ For each $s$ in $\text{FILES[B.in][f]}$, $\text{FLUSHED[B.out][s]} = \text{FLUSHED[B.in][s]}$ for all $s \in S$
| $\text{f.flush(); f.close();}$ If $\text{FILES[B.in][f]}$ is a singleton $\{s\}$, then $\text{FLUSHED[B.out][s]} = True$, and for all other $s' \in S$, $\text{FLUSHED[B.out][s']} = \text{FLUSHED[B.in][s']}$. Otherwise, for all $s \in S$, $\text{FLUSHED[B.out][s]} = \text{FLUSHED[B.in][s]}$.
Boundary condition initialization | True.
Interior points initialization | True.

Finally, for each instruction `print(f.read())` in basic block B, issue a warning if there exists a file $s$ in $\text{FILES[B.in][f]}$ for which $\text{FLUSHED[B.in][s]}$ is False.