Total points: 100, 25% of course grade

Good luck!
75 minutes (maximum)
Closed Book

• You may use one side of one sheet (8.5x11) of paper with any notes you like.
• This exam has 9 pages, including this cover page. Do all your work on these exam sheets.
• Be specific and clear in your answers. If there is any question about what is being asked, then indicate the assumptions you need to make to answer the question.
• Show all your work if you wish to be considered for partial credit.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

Name: 

Email: 

DO NOT TURN TO NEXT PAGE TILL YOU GET PERMISSION
1. [15 points] Garbage Collection. For the following code,

```java
class Main {
    public static void main(String[] a) { System.out.println(new List().testing()); }
}
class List {
    Node head;

    public Node insert(Node n) { n.setNext(head); head = n; return n; }

    public int testing() {
        int i; i = 0;
        while (i < X) {
            this.insert(new Node());
            i = i + 1;
        }
        return 7;
    }
}

class Node {
    Node next;
    int val;

    Node setNext(Node n) { next = n; return n; }
}
```

a) indicate the number of objects created by the whole program as a function of X,
b) and indicate the number of times a mark and sweep collector will be called as a function of X assuming 12 bytes of GC overhead (as in the MiniJava GC assignment), 4 bytes to store a pointer, and a 40 byte heap. Assume that if the GC is called and it can not find garbage that the program is then terminated with an exception.

c) Assume that heap locations have no guarantees in terms of their initial value. What problem might a mark and sweep garbage collector have when attempting to garbage collect for the above program? How could this problem be fixed if all newly allocated memory was zeroed out?
2. [25 points] Register Allocation. The goal is to perform register allocation on the procedure body provided below using two registers: $s0 is callee-saved and $t0 is caller-saved.

Liveness

\[
\begin{align*}
t1 & = \ldots \\
t2 & = 4 \times t1 \\
t3 & = 42 + t2 \\
\text{if } ( t1 < 5 ) & \\
& \quad t4 = t3 \times 5 \\
\text{else } & \\
& \quad t4 = t2 + 20 \\
\text{foo( } t4 \text{ )}
\end{align*}
\]

(a) Modify the above procedure body to model the live ranges of the callee-saved register $s0 and the caller-saved register $t0. Copy $s0 into a new temporary called $t5.

(b) In the column entitled “Liveness” indicate the set of variables and/or registers that are live between each statement in the above procedure body.

(c) Construct the interference graph based on the calculated liveness information and include the callee-save and caller-saved registers as precolored nodes.
(d) Use optimistic simplification ala Briggs with no coalescing to color the interference graph. Label each node in the interference graph with a color to indicate the coloring. If it is necessary to select a temporary that might be spilled, select the temporary with the highest temporary number first. For example, decide to spill t5 over t3.

(e) Based on the coloring, indicate which variables/temporaries map to which registers below:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Register or spill</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td></td>
</tr>
</tbody>
</table>
3. [25 points] Partial Redundancy Elimination. Fill in the given table with anticipated, PRE availability, earliest, postponable, latest, and used expression data-flow analysis results. Indicate on the control-flow graph where any code changes will occur based on the analysis result. In other words, perform PRE.
<table>
<thead>
<tr>
<th></th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
</tr>
</thead>
<tbody>
<tr>
<td>anticipated out</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>anticipated in</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>available in</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>available out</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>earliest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>postponable in</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>postponable out</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>latest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>used in</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>used out</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4. [15 points] After PRE. What other data-flow optimizations should follow PRE? Why? Perform the two most important on your 3-address code from problem 3.
5. [20 points] Instruction scheduling.
   a) Rewrite the following code with the loop unrolled twice and using 3-address code or some approximation of assembly code. Use a different set of temporaries/registes for the second loop body. You can assume that N is divisible by 2. Recall that there should only be one set of loop overhead statements.

   ```c
   for (i=0; i<N; i++) {
       x[i] = 6.0 * x[i];
   }
   ```

   b) Draw the DAG for the new loop body. Do NOT include the branch or the iterator update. Assume a latency of 2 between any instructions with a flow dependence, a latency of one between output dependences, and a latency of zero between anti dependences. A latency between two instructions means that the first instruction is started, then latency cycles must occur, and then the next instruction can start.

   c) Use list scheduling with the following priorities to find a schedule. How many cycles does the schedule require assuming only one instruction can be issued each cycle?

   **Priorities**
   1) Avoid stalls with previously scheduled instructions.
   2) Pick instructions that interlock with the most immediate successors.
   3) All else being equal pick the initial instruction ordering.