The exam is open book/notes/laptop. We do not guarantee power or Internet access, however.

Answer all 5 questions on the exam paper itself.

Write your name here: ____________________________________________

I acknowledge and accept the honor code.

(signed) __________________________________________

<table>
<thead>
<tr>
<th>Question</th>
<th>Max</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
QUESTION 1 (Flow graphs)

The figure above is a flow graph with entry 1. Answer the following questions about it.

a) Fill in the table of dominators. The trivial dominators (i.e. the entry dominates every node, and every node dominates itself) have already been filled in.

<table>
<thead>
<tr>
<th>Dominator</th>
<th>Dominates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 2, 3, 4, 5, 6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4, 5, 6</td>
</tr>
<tr>
<td>5</td>
<td>5, 6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

b) If we build a depth-first spanning tree for the flow graph, there are several different trees that can result, depending on which successor we visit first, when there is a choice. However, in all these trees, node 1 is the root, four of the other nodes have the same parent, and one node can have two different parents. Which node can have different parents?

3


c) List each back edge and its corresponding natural loop.  
(Note that the number of rows in the table isn’t necessarily representative of the number of back edges in the graph.)

<table>
<thead>
<tr>
<th>Back edge</th>
<th>Natural loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>3→1</td>
<td>1, 2, 3, 4, 5, 6</td>
</tr>
<tr>
<td>5→4</td>
<td>4, 5, 6</td>
</tr>
<tr>
<td>6→5</td>
<td>5, 6</td>
</tr>
<tr>
<td>6→4</td>
<td>4, 5, 6</td>
</tr>
</tbody>
</table>

d) What is the depth of the flow graph?

3 (Consider path 6-5-4-3-1, with three retreating edges.)
QUESTION 2 (DFA Frameworks)

A useful analysis in some applications is a framework that determines when two program variables must have the same value. In this framework,

- \( V \) is set of all possible partitions.
- A partition is a set of equivalence classes.
- Variables in an equivalence class have the same value.

For example, if the program has only two variables, \( x \) and \( y \), then,
\[
V = \{
\{\{x,y\}\}, \quad \text{// meaning } x \text{ and } y \text{ have the same value}
\{\{x\},\{y\}\} \quad \text{// meaning } x \text{ and } y \text{ are not known to have the same value}
\}
\]

The meet operation \( \wedge \) for this framework is the partition that places variables \( x \) and \( y \) in the same equivalence class if and only if both argument partitions place them in the same equivalence class. For example, \( \{\{a,b,c\},\{d,e\},\{f\}\} \wedge \{\{a,b\},\{c,d\},\{e,f\}\} = \{\{a,b\},\{c\},\{d\},\{e\},\{f\}\} \). Intuitively, as a confluence operator, this meet says that we know two variables have the same value coming into a block if and only if they had the same value coming out of all the predecessors of that block. Answer the following questions:

a) If there are three variables in the program, list the elements of \( V \) (the set of possible partitions):

\[
\{\{x,y,z\}\}
\{\{x,y\}\{z\}\}
\{\{x,z\}\{y\}\}
\{\{y,z\}\{x\}\}
\{\{x\}\{y\}\{z\}\}
\]

b) What are the top and bottom elements for this semilattice of three variable program?

Top element = \{\{x,y,z\}\}

Bottom element = \{\{x\}\{y\}\{z\}\}

c) Let \( p \) and \( q \) be partitions in \( V \). When is \( p \leq q \)? Note, we are looking for an answer in terms of the equivalence classes in the two partitions, not just “if \( p \wedge q = p \).”

Every block of the partition \( p \) must be a subset of some block of \( q \). Note that the reverse, "every block of \( q \) is a superset of some block of \( p \) doesn't work. Consider \( p = \{\{a\}\{d\}\{b,e\}\{c,f\}\} \) and \( q = \{\{a,b,c\}\{d,e,f\}\} \).

d) Describe the transfer function for a block that consists of only the assignment statement \( x = y \).

Remove \( x \) from its current block and add it to the block of \( y \). If \( x \) was in a singleton block, delete the resulting empty block. Note: It is wrong to say that the blocks of \( x \) and \( y \) should be merged. Consider the result of the following two steps: \( z = x; \ x = y \); The proper partition at the end is
e) Describe the transfer function for a block that consists of only the assignment statement \( x = y + 1 \).

If \( x \) is in a singleton block, do nothing. Else, remove \( x \) from its current block and place it in a new singleton block.
QUESTION 3 (Dataflow solver)

Ben Bitdiddle is implementing a forward dataflow solver. The pseudocode for his solver is below.

procedure MakeIterationOrder(G):
1: let I be a list
2: call DFS(G,G.entry()),I
3: return I

procedure DFS(G,v,I):
4: label v as discovered
5: for all edges from v to w in G.successors(v) do
6: if w is not discovered
7: recursively call DFS(G,w,I)
8: append v to list I

procedure ForwardSolver(G,V,∧,F):
9: for all blocks B in G do
10: IN[B] = OUT[B] = V
11: OUT[G.entry()] = V\_ENTRY
12: I = MakeIterationOrder(G)
13: while changes to any OUT do
14: for each block B in I do
15: IN[B] = V
16: for each block P in G.predecessors(B)
17: IN[B] = ∧(IN[B],OUT[P])
18: OUT[B] = F\_B(IN[B])

The parameters to ForwardSolver are:
G: control flow graph
V: domain (possible values for IN/OUT)
∧: meet operator
F: transfer functions

Ben’s good friend Alyssa P. Hacker notices a bug in Ben’s code that makes his forward dataflow solver take a long time to converge.

a) (4 points) Assume the input graph G is reducible and has N nodes, E edges, and maximum depth D, and the dataflow problem is reaching definitions. What are the maximum number of iterations around the while loop at line 13 before Ben’s dataflow solver will converge and detect there are no changes?

N. The iteration order is post-order which is the exact reverse of the optimal order. Information from the entry node will only propagate one step forward on each iteration. The worst case is a
straight line graph, in which case it will take N-1 iterations for the information at the entry node to propagate through the entire graph, then 1 more iteration to detect convergence. A more accurate answer is "the length of the longest acyclic path from the entry" (which is in the worst case N).

A number of students made a fencepost error and answered N+1. If there are N nodes in a graph, the maximum acyclic path length that touches every node is N-1. Other students answered E or some variant thereof. A graph with N nodes can have up to N^2 edges, e.g. if it is a complete graph.

b) (10 points) Alyssa points out an easy, one-line fix to Ben’s code that will make it converge much more quickly. What is the fix?

Change line 8 to “prepend v to list I”. Or, add code to reverse the list I. This will change the iteration order from postorder to reverse-postorder.

A number of students said to move line 8 to above line 5. This will give a pre-order list, which is quite different from a reverse post-order list. Reverse post-order gives nodes in a topological ordering, so in general you visit predecessors before successors. Pre-order simply follows the DFST visit order, which will give the wrong order when you have an if-then-else construct, for example.

c) (4 points) Again assume the input graph G is reducible and has N nodes, E edges, and maximum depth D, and the dataflow problem is reaching definitions. After Alyssa’s fix, what are the maximum number of iterations around the while loop at line 13 before the fixed dataflow solver will converge and detect there are no changes?

D+2. Reaching definitions is a “rapid” problem where all events of significance at a node are propagated to that node along some acyclic path, so we can get a tight bound on iterations. See Section 9.6.7 in the textbook for more information.

d) (2 points) Does your answer to the previous question change if we drop the assumption that the input graph is reducible? If so, how?

No, the answer is still D+2 because D is the maximum depth across all DFST, so no matter which DFST we choose, we will encounter at most depth D and therefore at most D+2 iterations. See Section 9.6.5 in the textbook for more information.

e) (5 points) Does your answer to the previous question change if we also drop the assumption that the dataflow problem is reaching definitions, and instead assume the dataflow problem is monotone with lattice height H? If so, how?

Yes, it changes to N*H. This is because the dataflow problem may not be "rapid" (i.e. all events of significance at a node may not be propagated to that node along some acyclic path, e.g.
QUESTION 4 (Register Allocation)

a) Allocate registers using the register coloring algorithm we discussed in class on a machine with 3 registers. **Ignore the impact of “...”**. Show the interference graph and each step of the algorithm.

A = r1
D = r2
B = r3
C = r3
b) Can you allocate with two registers? If not, can you rearrange the code within one or more blocks so that you can? If you need to rearrange the code, show the rearrangement and the new register assignment.

No, you can not allocate with two registers. The two loads of “A” can be moved ahead of the stores of “C” and “B”. Some people moved “D” but that wasn’t allowed as the move would be across blocks (for some people I accidentally marked them for doing a cross-iteration move when I meant to say a cross block move, sorry). Some people constant-propagated C, but the problem said rearrange. Once the code is rearranged then D can be assigned one register and A, B and C the other.
QUESTION 5 (Instruction Scheduling)

Consider a processor with two functional units Load/Store and Everything_else. The result of a load is not available for an ALU operation until two cycles later; i.e. if the load is in cycle i, a use of the loaded data can not issue until cycle i+2. Loaded data can be stored in cycle i+1. All other results are available in the next cycle.

Consider the following program

1: \texttt{load a8 \leftarrow 2(a4)} \quad \# \text{ load a8 from } *(2+a4)
2: \texttt{load a9 \leftarrow 0(a4)} \quad \# \text{ load a9 from } *(a4)
3: \texttt{load a10 \leftarrow 4(a4)} \quad \# \text{ load a10 from } *(4+a4)
4: \texttt{add a8, a8, a9} \quad \# \texttt{a8 = a8 + a9}
5: \texttt{addi a10, a10, 2} \quad \# \texttt{a10 = a10 + 2}
6: \texttt{store a8 \rightarrow 0(a4)} \quad \# \texttt{store a8 into } *(a4)
7: \texttt{store a10 \rightarrow 2(a4)} \quad \# \texttt{store a10 into } *(2+a4)

a. Draw a data dependency graph of the code above. Denote true dependencies (RaW) as solid lines and label them by their latencies. Denote anti- and output dependencies (WaR and WaW) as dashed lines.

b) Schedule the code using the list scheduling algorithm from class. One possible answer below.
c) Give the basic global scheduling algorithm from class and the following code. Assume that the load is a load of a global variable that is always safe to load. Will the algorithm speculate the load? Is it always the right thing to speculate the load?

Give should have been “given”. No points were deducted for failing to describe the algorithm. If the algorithm was described correctly, extra credit was given against any other mistakes in question c. The algorithm will speculate the load because 1) it is only one level of speculation, 2) there is an empty slot and 3) it is safe to load. It might or might not be a good thing to do. It might, for example, cause a cache miss.