MO615 – Exercises 2s2012

Question 1 (Dataflow Analysis): In most production compilers, optimization is done using intermediate code that consists of simple 3-address instructions like the following:

```
r5 = r3 + r7
r6 = r5 * 8
```

Some of the instructions in the intermediate code may be dead code because the results of the instructions are never used later in the program. An important optimization in compilers is dead-code elimination, where we remove such instructions to save space and execution time.

(a) What data flow analysis should be used to discover which instructions are dead code? Describe the appropriate dataflow problem to use and explain how to use its results to identify dead instruction(s).

(b) After performing the data flow analysis in part (a) and removing the dead code that it identifies, will there be any dead code in the remaining instructions? If so, why, and what needs to be done to remove them? If not, why not?

Question 2 (Dataflow Analysis): To deal with security issues, several programming languages have a notion of "tainted" data. The idea is that any value read from the outside environment is marked as being tainted, i.e., potentially dangerous. The results of any operations that use tainted data are also marked tainted. There is also a way of marking a value as "not-tainted", presumably to be used only after verifying that it is "safe", whatever that may mean. For this problem, assume that the available operations in our intermediate language are:

x = y □ z	binary operation: x is tainted if either y or z or both are tainted
x = y	assignment: x is tainted if y is tainted
x = read()	input: result x is tainted
x = clean(y)	clean: assign y to x, but mark x as not tainted

Now we would like to use dataflow analysis on this low-level code to discover tainted variables. A variable that might contain a tainted value is marked as tainted. Only if we know that the value is guaranteed not to be tainted do we mark it so. To discover tainted variables we define the following sets for each basic block b:

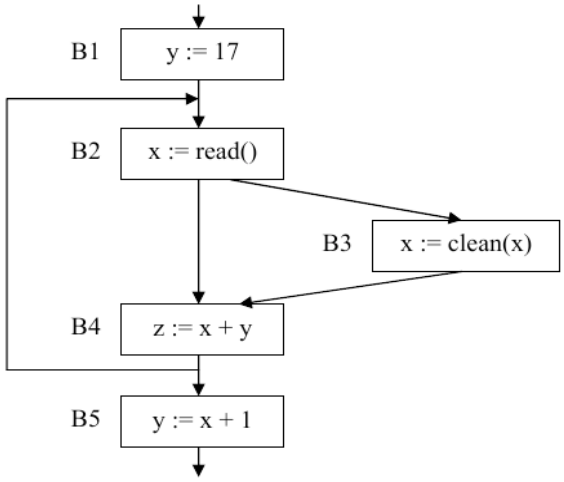
$IN_{[b]}$ =	set of all possibly tainted variables on entry to block b
$OUT_{[b]}$ =	set of all possibly tainted variable on exit from block b
$GEN_{[b]}$ =	set of all variables marked tainted in block b and not cleaned before exit
$CLEAN_{[b]}$ =	set of all variables cleaned in block b and not later tainted before exit

The sets $GEN_{[b]}$ and $CLEAN_{[b]}$ can be computed once based on the static contents of each block b. The $IN_{[b]}$ and $OUT_{[b]}$ sets need to be computed iteratively during the analysis.

(a) Give appropriate dataflow equations for the IN and OUT sets for a block b in terms of the IN, OUT, GEN, and CLEAN sets. As usual, these equations will involve some combination of local information

about the block itself as well as information about the block's predecessors and successors.

(b) Now consider the following flowgraph:



Complete the following table using iterative dataflow analysis to identify the tainted variables in the IN and OUT sets for each block in the above graph. You should first fill in the GEN and CLEAN entries for each block, then iteratively solve for IN and OUT. Choose whichever direction (forward or backward) you wish to solve the equations. You should assume that there are no tainted variables in the IN set for block B1.

Block	GEN	CLEAN	IN 1	OUT 1	IN 2	OUT 2	IN 3	OUT 3
B1								
B2								
B3								
B4								
B5								

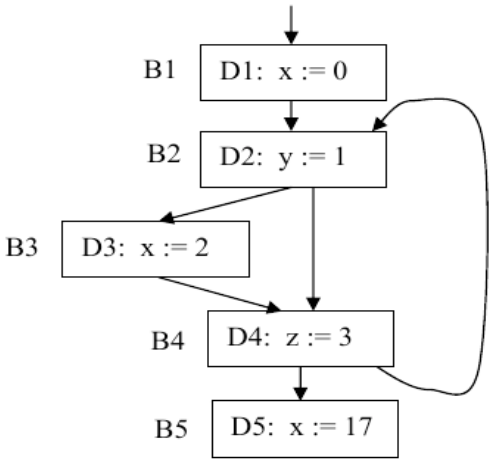
Question 3 (Dataflow Analysis) The dataflow problem for reaching definitions can be defined as follows:

$GEN_{[b]}$ = { set containing definition d from block b, if any }
$KILL_{[b]}$ = { all other definitions that assign to a variable that is defined in b }

$$IN_{[b]} = \bigsqcup_{p \in pred[b]} OUT_{[p]}$$

$$OUT_{[b]} = GEN_{[b]} \sqcup (IN_{[b]} - KILL_{[b]})$$

Compute the reaching definitions for the nodes in the following flowgraph. You should first fill in GEN and KILL in the table below for each block, then iteratively solve for the IN and OUT sets. Choose whichever direction to solve for IN and OUT that you wish (forward or backward).

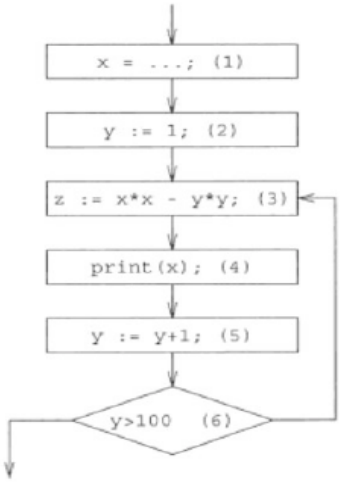


Block	GEN	CLEAN	IN 1	OUT 1	IN 2	OUT 2	IN 3	OUT 3
B1								
B2								
B3								
B4								
B5								

Question 4 (Dataflow Analysis) An optimization technique called "code hoisting" moves expressions to the earliest point beyond which they would always be evaluated. An expression that is always evaluated beyond a given point is called very busy at that point. Once it is known at which points an expression is very busy, the evaluation of that expression can be moved to the earliest of those points. Determining these point is a backwards dataflow problem.

(a) Give the general dataflow equations to determine the points at which an expression is very busy

(b) Consider the following CFG. Give the KILL and GEN for the expression x * x



(c) Solve the dataflow equations for the expression x * x. What optimization becomes possible?

Question 5 (Instruction Scheduling): Suppose we have a compiler that generate code to the program below, where A and B are two floating-point arrays:

```
innerprod := 0;
for (i = 1; i <= n; i++) {
        innerprod := A[i] * B[i] + innerprod;
}
```

The generated code, after local optimizations and "nops" inserted to solve latency times in the target machine, might be like this:

```
(a)          innerprod := 0
(b)          i := 1
(c)   loop:  t1 := 4 * i
             nop
             nop
(d)          t 2 := t1 (A)
(e)          t 4 := t1 (B)
             nop
(f)          t 5 := t 2 * t4
             nop
             nop
(g)          innerprod := t5 + innerprod
(h)          i := i + 1
(i)          if i <= n goto loop
             *nop
             *nop
             *nop
```

The loop body (7 instr.) takes 15 cycles to execute, i.e., less than 50% of the potential performance. There is some parallelism between instructions, e.g. innerprod := t 5 + innerprod and i := i + 1. But

there are many "bubbles" in the pipeline where the processor is stalled waiting for a previous instruction to complete.

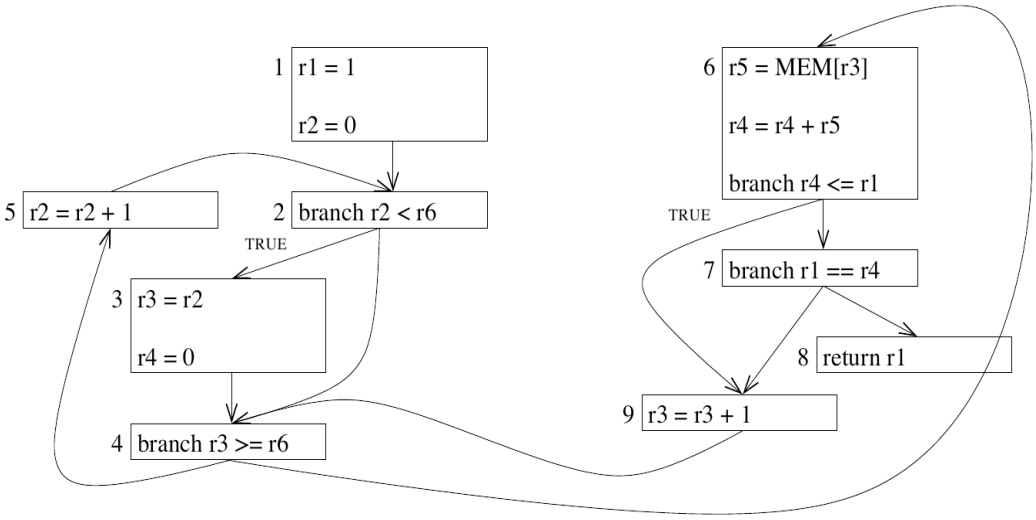
(a) Draw a precedence graph showing the dependencies between these instructions. Label each node (instruction) in the graph with both the instruction letter (a-i) and its latency – the number of cycles between the beginning of that instruction and the end of the graph

(b) Rewrite the instructions in the order they would be chosen by forward list scheduling (i.e., choosing at each step an instruction that is not dependent on any other instruction that has not yet been issued). If there is a tie at any step for which instruction would best be scheduled next, pick one of them arbitrarily.

(c) How many cycles are required by the new schedule you created in part (b)?

Question 6 (SSA): Convert the program below to SSA form. Show your work after each stage.

1. Add a start node containing initializations of all registers.
2. Draw the dominator tree.
3. Calculate dominance frontiers.
4. Insert φ-functions and rename temporaries to create SSA.

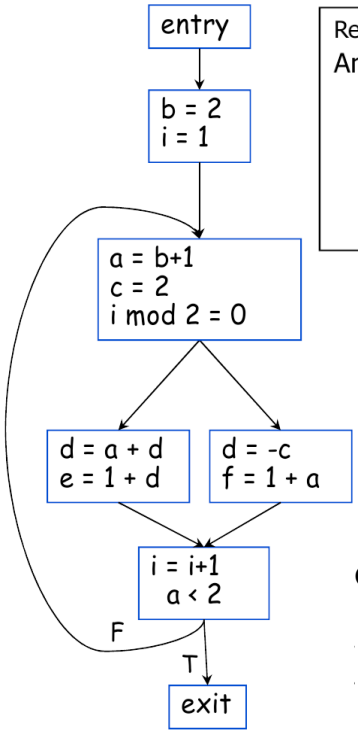


Question 7 (SSA): Here is a simple program that reads two numbers and prints their greatest common divisor as long as both numbers are positive.

```
x = read();
y = read();
while (x != y) {
        if (x > y) {
                x = x - y;
        } else { /* y > x */
                y = y - x;
        }
}
print(x);
```

Draw a control flow graph for this program (nodes = basic blocks, edges = possible control flow), using Static Single Assignment (SSA) form. You should include φ-functions where needed to merge different versions of the same variable. You should only include necessary φ-functions and not have extraneous ones scattered everywhere.
Hint: it might be easiest to sketch the flow graph first without converting it into SSA, then add the needed variable version numbers and φ-functions to get the final answer.

Question 8 (Optimization): What happens if you perform constant propagation followed by constant folding after the loop-invariant code motion in loop below?



Reminder:
An instruction is loop-invariant if, for each operand:
1. The operand is constant, OR
2. All definitions of that operand that reach the instruction are outside the loop, OR
3. There is exactly one in-loop definition of the operand that reaches the instruction, and that definition is loop invariant.

Reminder:
Move a loop-invariant x=... located in B, if
1. B dominates all exits of the loop
2. x is not defined anywhere else in the loop
3. B dominates all uses of x in the loop

Question 9 (Loop-Invariant Code Motion):

An instruction is loop-invariant if, for each operand:

1. The operand is constant, OR
2. All definitions of that operand that reach the instruction are outside the loop, OR
3. There is exactly one in-loop definition of the operand that reaches the instruction, and that definition is loop invariant.

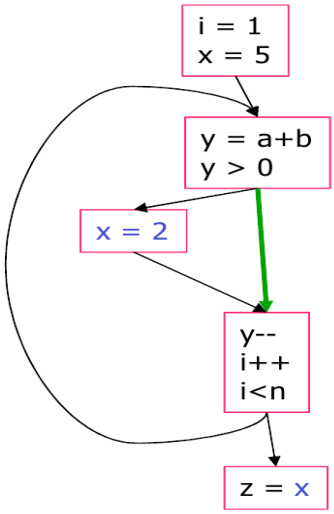
(a) How do we determine #2?

(b) For each loop-invariant instruction i: x=y+z in basic block B, we check that it satisfies the conditions:

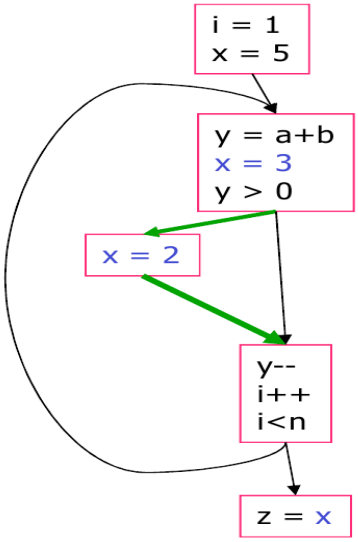
I. B dominates all exits of the loop
II. x is not defined anywhere else in the loop
III. B dominates all uses of x in the loop
(i.e. all uses of x in the loop can be reached only by i)

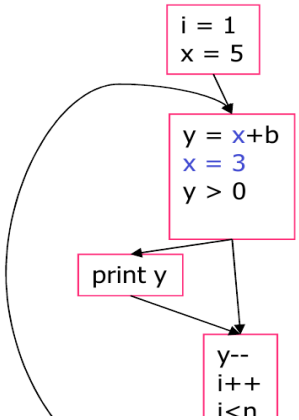
Now, for each CFG below, show why are these conditions necessary?

(I) B dominates all exits of the loop



(II) x is not defined anywhere else in the loop





(III) B dominates all uses of x in the loop

Question 10 (Optimization) You're the professor for a compiler construction class, and you have to write a final exam question on code optimization. Create a small code example for the exam which reduces to:

```
return 100
```

after the following optimizations are applied (only once) in this order:

1. Common Subexpression Elimination
2. Copy Propagation
3. Dead Code Elimination
4. Constant Folding
5. Constant Propagation
6. Constant Folding
7. Constant Propagation
8. Dead Code Elimination

Create the answer key by showing the code before and after each optimization step. Each optimization should have some real impact (i.e.: your initial program should not just be return 100!!). Hint: work backwards.