Homework 1 Solution (CS 4240 Compilers & Interpreters, spring 2019)

Question 1: Intermediate Representations [15 points]

Show how the following source code fragment can be translated to three-address code. Explain your answer in a few sentences. Assume standard rules of operator precedence/associativity as in C & Java.

```
while ( i < 100 ) {
    if (c[i] == 0) then
        a[i] = a[i] * (x + y) + b[i] * (x + y + 1);
    else
        d[i] = (a[i] + b[i]) * (x + y) - c[i] / (x + y);
    i = i + 1;
}
```

Comments on Q1:

By translating a program written in a high-level language to an intermediate represe ntation form, we break down the format of the given program into a simpler form, m aking it easier to optimize the code. It is also important to preserve the behavior of t he given program when doing translation to IR.

To translate expressions with more than two binary operators involved, temporary v ariables should be used in order to store intermediate values. It is important to preser ve the operator precedence and associativity of the source language.

Many students made mistakes in translating branches properly to intermediate representation, checking inverse conditions in branch operations. Also, some students had trouble translating the looping logic of the while loop to IR.

The expression (x+y) occurs multiple times throughout the given program. Some students stored the calculated value of (x+y) to a temporary variable before entering the if-else structure and reused it whenever (x+y) was needed, which is a clever thing to do.

Next page contains a sample model answer for Q1 from one of the students, Mike Lewis.

Source	Tiger IR
`while ( i < 100 ) {`	`check:`
	`brgeq, end, i, 100`
`if (c[i] == 0) then`	`array_load, t0, c, i`
	`brneq, after_if, t0, 0`
`a[i] = a[i] * (x + y) + b[i] * (x + y + 1);`	`array_load, t1, a, i`
	`add, t2, x, y`
	`mult, t3, t1, t2`
	`array_load, t4, b, i`
	`add, t5, x, y`
	`add, t6, t5, 1`
	`mult, t7, t6, t3`
	`array_store, t7, a, i`
`else`	`goto after_else`
	`after_if:`
`d[i] = (a[i] + b[i]) * (x + y) - c[i] / (x + y);`	`array_load, t8, a, i`
	`array_load, t9, b, i`
	`add, t10, t8, t9`
	`add, t11, x, y`
	`mult, t12, t10, t11`
	`array_load, t13, c, i`
	`add, t14, x, y`
	`div, t15, t13, t14`
	`sub t16, t12, t15`
	`array_store, t16, d, i`
	`after_else:`
`i = i + 1;`	`add, i, i, 1`
`}`	`goto check`
	`end:`

The process for converting a source language into three-address is to take each statement in the source and split it into its component sub-operations, storing intermediate values in temporary registers if necessary. For example, a statement such as `u = v * w * x * y * z` can be split into four separate multiplication operations:

- `t0 = v * w`
- `t1 = t0 * x`
- `t2 = t1 * y`
- `u = t2 * z`

The result of these smaller operations is the same as the original source version, but the size of each operation is smaller, allowing for easy analysis and execution.

Similar techniques are used for control flow blocks, using conditional and unconditional branch instructions to route the execution of the program between various sections as dictated by the source.

Question 2: Control Flow Graphs, Dominators, Reaching Definitions [40 points]
Consider the Tiger-IR fragment shown below. Please refer to the Tiger-IR Reference Manual at the end of this document. Note that **puti** prints to standard output, and **geti** reads from standard input.

- **a)** Show a control flow graph for this IR code, and define the basic blocks that you assumed (can be minimal or maximal with respect to IR instructions). **[5 points]**
- **b)** Construct the dominator tree for the CFG that you produced in part a). **[5 points]**
- **c)** For each def in the IR program, list all the uses that it reaches. Show the output of performing dead (useless) code elimination, using the basic reaching definitions algorithm that does not remove any branches. Explain your answer in a few sentences. **[20 points]**
- **d)** Discuss in a few sentences if there are additional dead instructions that were not identified by the basic algorithm, and if they can be identified as such by using other techniques that you've learned in class. **[10 points]**

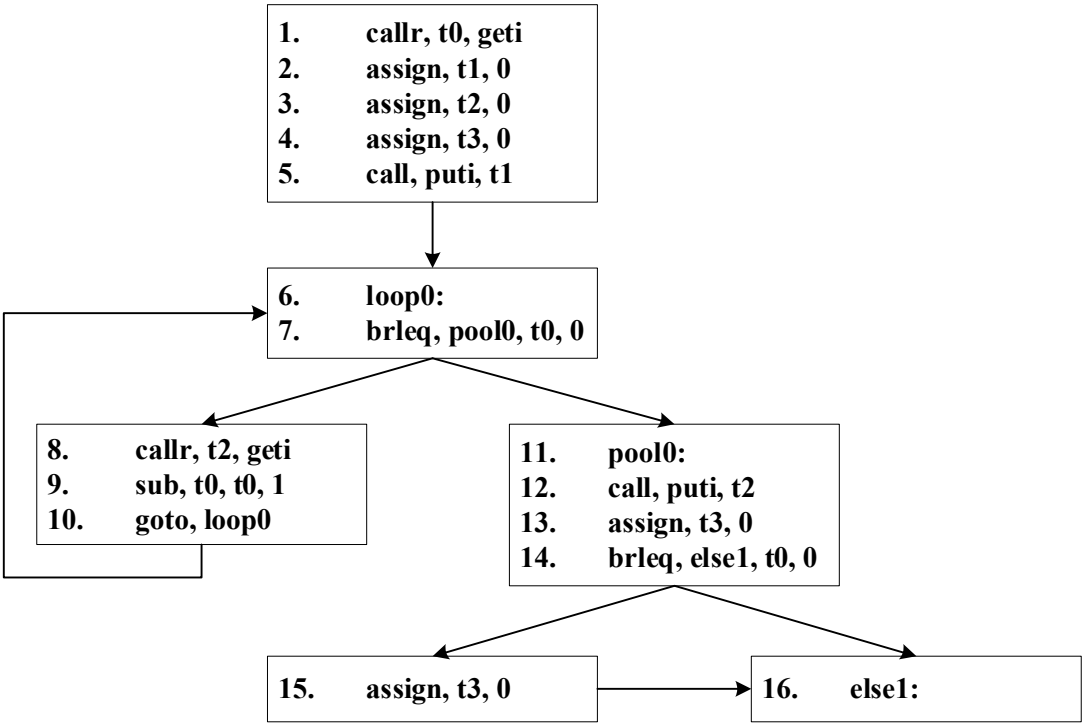
IR:

1. **callr, t0, geti**
2. **assign, t1, 0**
3. **assign, t2, 0**
4. **assign, t3, 0**
5. **call, puti, t1**
6. **loop0:**
7. **brleq, pool0, t0, 0**
8. **callr, t2, geti**
9. **sub, t0, t0, 1**
10. **goto, loop0**
11. **pool0:**
12. **call, puti, t2**
13. **assign, t3, 0**
14. **brleq, else1, t0, 0**
15. **assign, t3, 0**
16. **else1:**

Solution and Comments

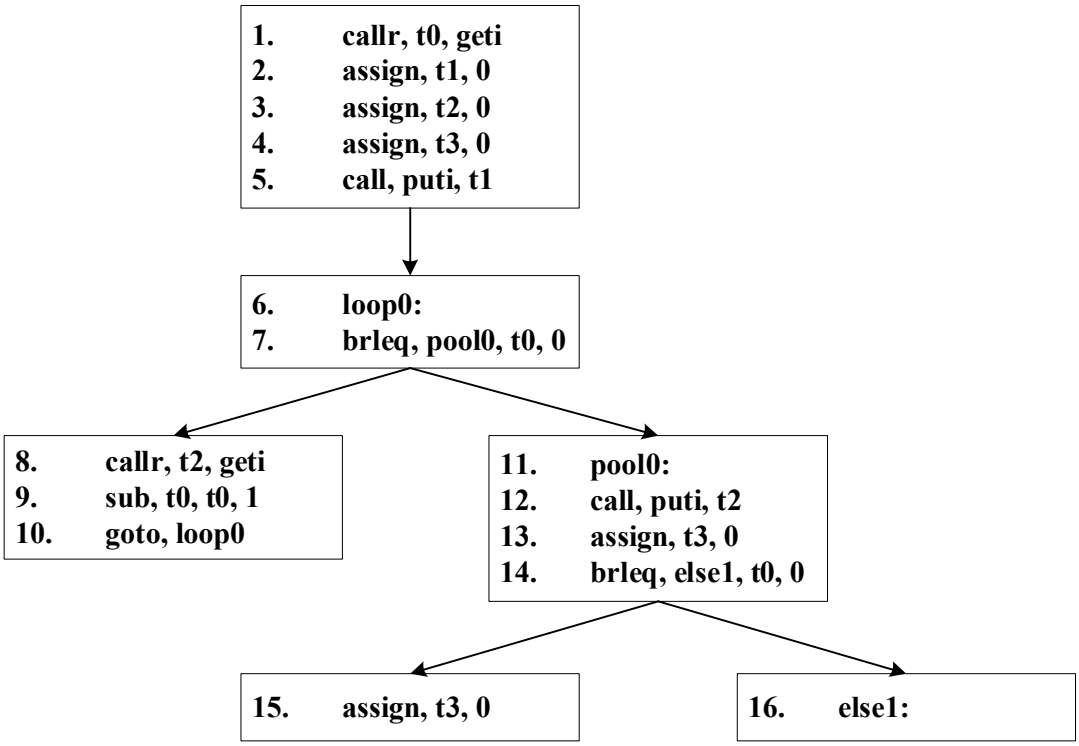
a)

Assuming maximal basic blocks:



Comments: Most of the students forgot to define which kind of basic blocks they are assuming.

b)



c)

Line of def	Lines of uses reached
1	7, 9, 14
2	5
3	12
4	
8	12
9	7, 9, 14
13	
15	

Output of dead code elimination:

1. **callr, t0, geti**
2. **assign, t1, 0**
3. **assign, t2, 0**
4.
5. **call, puti, t1**
6. **loop0:**
7. **brleq, pool0, t0, 0**
8. **callr, t2, geti**
9. **sub, t0, t0, 1**
10. **goto, loop0**
11. **pool0:**
12. **call, puti, t2**
13.
14. **brleq, else1, t0, 0**
15.
16. **else1:**

Explanation:

First, mark all critical instructions including function calls, branches, jumps and labels. Instructions at lines 1, 5, 6, 7, 8, 10, 11, 12, 14, 16 are marked in this step. Then, mark non-dead instructions by walking backwards from critical instructions following use-def relations. Instructions at lines 2, 3, 9 are marked in this step. Finally, remove all unmarked instructions (instructions at lines 4, 13, 15).

Comments: The def of t0 in line 9 should be able to reach the use of t0 in line 9. A lot of the students did not include the 2 key points in their explanations: critical instructions and backward marking.

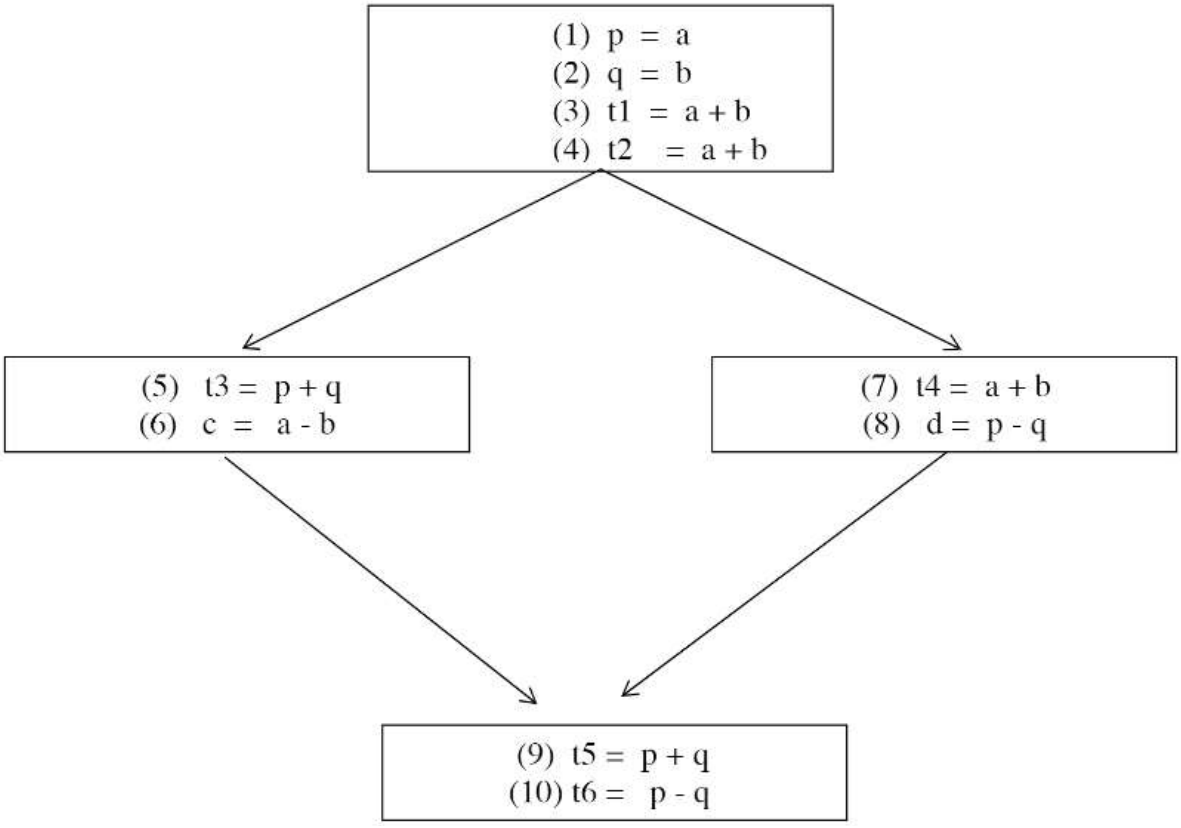
d)

There is no further dead instruction that could be identified by other techniques learned in this class so far. However, with a more sophisticated algorithm that takes branches into consideration (on page 545 of the textbook), line 14 can be eliminated because no other instruction is control dependent on it.

Question 3: **Value Numbering & Redundancy Elimination [45 points]**

Consider the following CFG with the IR code shown in each basic block. Please answer the following questions. Show value numbers assigned to variables and expressions.

- **a)** Show the redundant expression computation detected by the Local Value Numbering (LVN) technique only (limited to basic block by basic block analysis). Show the transformed code generated after the redundancy is removed. Introduce additional temporaries to hold the sub-expression results if necessary. **[15 points]**
- **b)** Repeat the above step for Superlocal Value Numbering (SVN) working on the scope of Extended Basic Block (EBB). How many additional redundancies are detected and eliminated by SVN compared to LVN? Explain your answer. **[15 points]**
- **c)** Detect and eliminate the redundancies that are left out (if any) by SVN in the CFG. Again, introduce necessary temporaries to hold the sub-expression results and show the transformed code which should have no remaining redundancies. Explain your answer. **[15 points]**



Comments on Q3.a)

Local Value Numbering(LVN) operates on each basic block, using a separate value number table for each basic block. However, many students used a single value number table for all four

blocks. In the three basic blocks excluding the first basic block, value numbers assigned to variables a and p should be different when LVN is applied. Same holds for variables b and q.

Since LVN operates on a single basic block, the only redundant calculation detected is (a+b) on line (4). Variable t1, which already contains the calculated value of (a+b) can be reused on line (4).

Comments on Q3.b)

Superlocal Value Numbering(SVN) expands the scope of value numbering from a basic block to an extended basic block(EBB). SVN maintains a separate value number table for each control path in an EBB. By using SVN, two more redundancies are detected ((5), (7)) and can be replaced with a reference to t1.

Some students combined all four basic blocks to a single Extended Basic Block(EBB), but by definition of EBBs, the first three blocks are in a common EBB, and the last basic blocks forms a separate EBB on its own.

Comments on Q3.c)

The expressions (a-b) and (p-q) are contain same values, and are both computed twice (redundant) in both of the possible control paths.

Also, line(9) also contains redundant computation, and can be replaced with a reference to a variable which already holds the needed value. The redundancy is detectable by human observation, but Dominator Value Numbering could also detect the redundancy. Since the first basic block dominates the final basic block, the final basic block can use a subset of its dominator's value number table where the keys or the variables which are part of a key in the table are not redefined before reaching the final basic block.

Some students used variables c or d in line(10) to reduce redundant computation, but using only c or only d generates some issues. Variables c and d are only initialized and available in one of the two possible control paths. Using a phi-function or introducing a new temporary variable was necessary to reduce redundant computation in line(10).

Next two pages contains a sample model answer for Q3 from one of the students, Sang-Chan Kim.

Question 3

a)

Basic Blocks	Code	X	VN(X)
1	1) p = a 2) q = b 3) t1 = a + b ~~4) t2 = a + b~~ **t2 = t1**	a	1
		p	1
		b	2
		q	2
		$a^1 + b^2$	3
		t1	3
		t2	3
2	5) t3 = p + q 6) c = a – b	p	1
		q	2
		$p^1 + q^2$	3
		t3	3
		a	4
		b	5
		$a^4 – b^5$	6
		c	6
3	7) t4 = a + b 8) d = p - q	a	1
		b	2
		$a^1 + b^2$	3
		t4	3
		p	4
		q	5
		$p^4 – q^5$	6
		d	6
4	9) t5 = p + q 10) t6 = p - q	p	1
		q	2
		$p^1 + q^2$	3
		t5	3
		$p^1 – q^2$	4
		t6	4

b) The two extended basic blocks in this problem are EBB1 = {BB1, BB2, BB3} and EBB2 = {BB4}

For EBB1, execute two LVNs. Execute LVN on a merge of BB1 and BB2 and another on a merge of BB1 and BB3.

Basic Blocks	Code	X	VN(X)
<BB1, BB2>	(1) p = a (2) q = b (3) t1 = a + b (4) ~~t2 = a + b~~ **t2 = t1** (5) ~~t3 = p + q~~ **t3 = t1** (6) c = a – b	a	1
		p	1
		b	2
		q	2
		$a^1 + b^2$	3
		t1	3
		t2	3
		$p^1 + q^2$	3
		t3	3

		$a^1 - b^2$	4
		c	4

Basic Blocks	Code	X	VN(X)
<BB1, BB3>	(1) p = a (2) q = b (3) t1 = a + b (4) ~~t2 = a + b~~ **t2 = t1** (7) ~~t4 = a + b~~ **t4 = t1** (8) d = p - q	a	1
		p	1
		b	2
		q	2
		$a^1 + b^2$	3
		t1	3
		t2	3
		t4	3
		$p^1 - q^2$	4
		d	4

For EBB2, execute LVN on a single BB4 block.

Basic Blocks	Code	X	V(N)
<BB4>	(9) t5 = p + q (10) t6 = p - q	p	1
		q	2
		$p^1 + q^2$	3
		t5	3
		$p^1 - q^2$	4
		t6	4

By executing SVN, we've reduced **two more** redundancies in total. The two correspond to Line 5 and Line 7 in the given original code.

c) There are four redundancies that are left out by SVN. In the original code, Line 6, 8, and 10 compute the same value (a – b == p – q). Declare a new temporary variable **t7** such that **t7 = a – b** in **BB1**.

(6) c = a -b ➔ c = t7

(8) d = p – q ➔ d = t7

(10) t6 = p – q ➔ t6 = t7

Lastly, Line 9 is also a redundancy not eliminated by SVN. Change Line 9 such that

(9) t5 = p + q ➔ t5 = t1