Problem 1: Control-Flow, Loops and Loop Optimizations [40 points]

Consider the code depicted below for a function with integer local variables i and integer input parameters p and n. Assume that parameters p and n are assigned values by the calling context but the procedure uses them as local variables.

```c
int i, toggle;
01:    i = 0
02:    toggle = 0
03: L1: if i = n goto L5
04:    t0 = 1024 * i
05:    t1 = t0 + 0
06:    t2 = A[t1]
07:    t3 = t0 + 1
08:    t4 = A[t3]
09: if toggle = 0 goto L2
10:    toggle = 0
11:    t5 = t2 + t4
12: goto L3
13: L2: toggle = 1
14:    t5 = t2 - t4
15: goto L3
16: L3: t6 = t0 + 2
17: t7 = A[t6]
18: t8 = t0 + 3
19: t9 = A[t8]
20: t10 = t7 + t9
21: t11 = t5 + t10
22: t12 = p * i
23: t13 = 4 * t12
24: t14 = t11 / t13
25: t15 = 1024 * i
26: B[t15] = t14
27: t1 = i
28: t1 = i
29: goto L1
30: L5: return 0
```

For this code fragment determine:

a. [05 points] The Control Flow Graph (CFG).
b. [10 points] The dominator tree and the loops (identify the back edges).
c. [05 points] Use algebraic properties and copy propagation to improve the execution of the body of the loop(s) identified in (b).
d. [10 points] Based on the code obtained in (c) recognize loop induction variables and transform the code to eliminate them as much as possible. Indicate which variables are basic induction variables and which are derived induction variables.
e. [10 points] Discuss if there are opportunities for loop invariant code motion in the original code versus the transformed code obtained in (d). Arguing about the correctness of your transformations.
Problem 2: Branch Transformations [10 points]

Intermediate code generation phases of compiler are very sloppy in the way they generate conditional and unconditional gotos mostly for simplicity. For this reasons it is common for compiler to have a series of clean-up passes after intermediate code generation. In this exercise you will apply a specific transformation, called branch forwarding as illustrated in the figure below.

In this figure the conditional branch to L1 targets an unconditional goto instruction. In this case you can replace the conditional target branch with the target branch of the unconditional instruction thus saving the execution of an instruction.

```
01: instr
02: L0: if predicate goto L1
03: instr
04: goto L1
05: L1: goto L2
06: instr
```

```
01: instr
02: L0: if predicate goto L2
03: instr
04: goto L2
05: L1: goto L2    // to be removed?
06: instr
```

Questions:

a. [03 points] When is it legal to apply this transformation?

b. [04 points] Suggest a simple code transformation algorithm.

c. [03 points] Apply your algorithm to the code below.

```
01: if pred1 goto L2
02: L0: goto L2
03: instr1
04: if pred2 goto L1
05: goto L1
06: L1: goto L0
07: L2: instr2
08: instr3
09: instr4
10: goto L1
```
Problem 3: Iterative Data Flow Analysis [50 points]

In this problem you will develop and show the application of the Live Variable analysis problem to the code of a given procedure. The live variable analysis problem seeks to determine for each scalar variable (or temporary register) if its contents is live. This has the fundamental application in the context of register allocation since if a given variable is no longer live at a given point then there is no need to keep it in register any longer.

Formally, a variable $v$ is live at a execution point $p$ if either its value is used at $p$ or there exists an execution path from $p$ to $q$ along which the value of $v$ is not written and is used in $q$.

a. [10 points] Formalize the live variable problem as an iterative data-flow analysis problem showing the equations for the gen and kill abstractions as well as the initialization of the values for each basic block and statement. In this section you need to determine if this is a forward or backward data flow problem.

b. [25 points] Apply your data flow problem formalization to the procedure code depicted below showing the final result of the IN and OUT set of live variables for each basic block in the code.

c. [15 points] Consider that you only have 2 registers to implement the code in the body of the loop. Which set of live variables would you keep in registers and why?