Problem 1: Register Allocation [50 points]

Consider the sequence of 3-address (assembly-like) instructions depicted below corresponding to the code of a procedure with local variables 'a'. The procedure also accesses a global array variable 'A'. For simplicity, the prologue and epilogue code for this procedure is not shown and neither are the lower level representations of the accesses to the parameters using the frame pointer and corresponding offset as well as the address calculation for the accesses to the array ‘A’.

```
01:   t0 = 0
02:   t1 = 1
03:   t3 = a
04:   t7 = 1
05:   t10 = 0
06:   if (t3 > 0) goto L1
07:   t5 = t1 * 2
08:   t6 = A[t5]
09:   t7 = t6 * 2
10:   t8 = t7 * 4
11:   goto L2
12:   L1: t9 = t0 + 1
13:   t10 = A[t9]
14:   t11 = t10 * 2
15:   goto L3
16:   L2: t9 = t1 + t8
17:   t10 = A[t9]
18:   L3: t12 = t7 + t11
19:   return t12
```

Questions:

(a) [05 points] Determine the Control-Flow Graph (CFG) corresponding to this assembly identifying the various basic blocks and the instructions in each basic block.

(b) [10 points] Using the simple top-down register allocation determine which variable should be kept in registers assuming that you only have 5 available registers.

(c) [10 points] Determine for each variable 't0' through 't12' during which instructions is each of them alive. A variable is said to be alive at a specific point in the execution, if there is a possible use of that variable after this execution point. Consider the instruction where the last use of a variable occurs as the last instruction of the live range of that variable. Notice that a given variable may have several live range sections. This information is key for the global register allocator.

(d) [10 points] Determine the interference graphs of the live ranges (web) for the various temporary variables in this program for the simplest definition of instruction interference described in class. Argue that the resulting graph cannot be colored using 5 registers. Using the graph-coloring heuristic algorithm described in class, try to color this graph using successively larger number of registers. What is the minimum number of registers you can color either of the interference graphs found earlier? You do not have to show the intermediate coloring algorithm steps but show the live ranges of each variable. Also, show the modified assembly code using the
physical registers instead of the temporary variables. Ignore in this representation the access to local variables and parameters of the procedure.

(e) [10 points] If you were short of one register for the temporary variables which variable would you spill to memory and why? Show the resulting modified assembly code with the spilling instructions and the modified instructions using the physical register instead of the temporary variables.

(f) [05 points] Suggest source code transformations that would reduce the number of required registers for this code. Argue that your transformations are correct, i.e., the transformed code yields the same results as the original un-transformed code.
Problem 2: Control-Flow Graphs and Dominators [30 points]
Consider the three-address instructions sequence below.

```plaintext
01:    i = 0
02:    a = 0
03:    b = p1
04:    if (a < b) goto L2
05:    L1:  A[a] = 1
06:    A[b] = 0
07:    c = a + b
08:    d = c + 1
09:    A[d] = 2
10:    if (d > a) goto L4
11:    L3:  A[d] = 0
12:    c = 0
13:    d = 0
14:    i = i + 1
15:    goto L4
16:    L2:  i = i + 1
17:    if (i == p2) goto L5
18:    L4:  if (i != p2) goto L3
19:    goto L1
20:    L5:  i = i + 1
21:    return
```

Questions:
(a) [10 points] Determine the Control-Flow Graph (CFG) corresponding to this assembly identifying the various basic blocks and the instructions in each basic block.
(b) [10 points] Determine the dominator tree for the various basic block of the corresponding CFG.
(c) [10 points] Determine the back-edges (if any) and the corresponding natural loops of this code.
Problem 3: Instruction Scheduling [20 points]
Consider the sequence of MIPS-like instructions depicted below. Under the assumption that you are targeting a computing architecture with one (1) integer arithmetic and logic unit with a 1 clock cycle latency for arithmetic and logic operations (except for multiplication and division with a latency of 4 and 7 cycles respectively) and one (1) memory units with a load (read) latency of 2 clock cycles and write latency of 1 clock cycle answer the following questions. Neither of these units is pipelined.

(1)   lw   $t0, 4($fp)
(2)   lw   $t1, 8($fp)
(3)   lw   $t2, 12($fp)
(4)   addu $t0, $v0, 4
(5)   addu $t4, $t2, $t0
(6)   addu $t1, $v1, 8
(7)   addu $t3, $t1, 1
(8)   mul   $t2, $t0, $t1
(9)   div   $t1, $t1, $t3
(10)  sw   $t0, 0($gp)
(11)  sw   $t1, 4($gp)
(12)  sw   $t2, 8($gp)
(13)  sw   $t3, 12($gp)
(14)  sw   $t4, 16($gp)

Questions:

(a)  [10 points] Derive the instruction dependence graph (DAG) for this sequence of instructions, indicating the latency for each data dependence as the weight of the edges between nodes of this graph that represent the instructions.

(b)  [05 points] Derive a feasible schedule for a computer architecture with a two non-pipelined arithmetic functional unit and a single pipelined memory unit with the latency parameters outlined above. A non-pipelined unit does not allow the issue of independent instructions at every clock cycle.

(c)  [05 points] Would an additional arithmetic logic unit or the conversion to a pipelined arithmetic unit or the inclusion of an additional memory (possibly pipelined) unit reduce the number of clock cycles required to execute these instructions? Why or why not? What does this tell you about this sequence of instructions?