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 are not shown and neither are the lower level representations of the accesses to the parameters using the frame pointer and corresponding offsets as well as the address calculation for the accesses to the array ‘A’. Instead, and for symbolic variables such as ‘a’ we rely on the name of the variable itself to denote its address.

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.

Solution:

a) Figure below depicts the Control-Flow-Graph (CFG) for this section of code revealing 5 basic blocks and no nesting structure.

b) The frequency of accesses to each of the temporary variable is depicted on the RHS of the figure with the temporary variables 't7' and 't10' having 4 accesses, 't11' has 3 accesses and all the remaining variables with 2 accesses. As such, and given that all the basic blocks have the same weight, if we were to use 5 registers we would assign two of them to 't7', 't10' and 't11' and the remainder 2 to any three of the other variables.

c) The live ranges in terms of the instructions where each temporary variable is used is shown in the figure above.

d) Based on the live ranges in c) and the simplest of the two definitions of interference we would derive the interference graph depicted below. As can be seen there is one 5-clique involving nodes {t0, t1, t3, t7, t10} As such the minimum number of colors will be 5 (shown on the right).
Based on the register coloring above we can re-write the original code as shown below (i.e. left).

e) If we were short of one register, i.e., wanted to have just 4 registers, one possibility would be to not even map to register the value associated with ‘t3’ as it has a very short live range. In this case one would not even use a regular register but an auxiliary temporary register such as $at1 assembly register. The figure below on the right illustrates this scenario where and given the fact that this would be a clean register operation there is no need to save the value of the $at1 register.

f) There are various opportunities here for constant propagation and dead-code elimination. For instance, ‘t0’ is assigned the zero value and ‘t1’ the value 1. Neither of these temporaries is subsequently modified so their values can be propagated forward. Similarly, t7 is set to 1 but this value is never used and so the assignment on line 04 can be safely removed.

The resulting transformed code (before applying other transformations such as algebraic simplification and strength reduction) is as shown below on the left. On the right we depict the corresponding interference graph, which can now be colored using just 3 registers.
Problem 2: Control-Flow Graphs and Dominators [30 points]
Consider the three-address instructions sequence below.

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.

Solution:
(a) The CFG and the dominator tree are shown below.
(b) See figure below.
(c) By inspection there is no edge whose "head" dominates its "tail", i.e., there is no edge in the CFG whose basic block pointed to by the edge's head dominates the basic block at the tail of that same edge.
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.

(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 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?

Solution:

(a) Instruction dependence graph is shown below (left) where we have indicated the distance between the nodes.

(b) On the right (top) is a feasible schedule for this DAG.

(c) The length of the resulting schedule is really limited by the latency of the arithmetic operations, in particular the division which is 7 cycles long - in other words this computation is compute-bound. While adding an additional ALU will improve the execution schedule only slightly. Adding an additional load/store unit will improve the length of the execution marginally under the assumption that the bandwidth to memory is not exhausted.