CSCI 2500 — Computer Organization
Lab 10 (document version 1.0)

- This lab is due by the end of your lab session on Wednesday, November 13, 2019.
- This lab is to be completed **individually**. Do not share your code with anyone else.
- You **must** show your code and your solutions to a TA or mentor to receive credit for each checkpoint.
- Labs are available on Mondays before your lab sessions. Plan to start each lab early and ask questions during office hours, in the discussion forum on Submitty, and during your lab session.

1. **Checkpoint 1:** For the first checkpoint, download the `lab10.s` MIPS code. Walk through the code to understand what it does. Consider uncommenting the debugging blocks of code to display what is in memory address `$s0` at each loop iteration.

   Once you understand the code, comment out the debugging output for the time being. The problem with the given code is that the control overhead is too high, i.e., we load, add, and store, then jump to our latch basic block. Excluding the jumps, we have 25% overhead.

   To optimize this code, replicate the contents of the `loop_body` basic block three more times (for a total of four load/add/store combinations). Make sure you use the correct offsets and increment value in the `loop_latch` basic block. Verify that your code still works by again adding the debugging statements.

   Note that this technique is called *loop unrolling*.

2. **Checkpoint 2:** For the second checkpoint, note that we load our values exclusively into register `$t0`. Thinking back to Homeworks 2 and 4, there is no reason why we could not use more of the temporary registers. Make this adjustment in the given code by cycling through registers `$t1`, `$t2`, etc. And note by doing so, we effectively remove dependencies that are not true dependencies.

   This technique is simply called *register renaming*.

3. **Checkpoint 3:** For the third checkpoint, we can use a technique to *minimize stalls* by reordering our instructions. Note that we have a familiar (and problematic) pattern, i.e., a load followed by an add. The problem with this pattern is that it has to stall the pipeline since the data is not ready until the `MEM` stage, but we need that data prior to the ALU stage of the next instruction. Our instruction stream therefore requires a `nop` instruction between the load and the increment instructions. Reorder the instructions such that no such `nop` instruction is required.

   Finally, given all of the above optimizations, how much have all of these transformations reduced the runtime for this code sequence?