Problem 4. Having fallen in love with the IBM360 early in the course, you've analyzed a dynamic instruction trace of the EGGSELL spreadsheet running your Valentine's Day order list on your MIPS machine in order to consider resurrecting some aspects of the 360. You find that the benchmark executed 1,000,000 instructions in 2,200,000 cycles and that instruction frequencies were:

Arithmetic	50%
Branch	20%
Load	20%
Store	10%

Under more careful analysis you find that 25% of the loads are used to add a value to a single register. Putting these two important discoveries together, you have decided to add a LADD instruction to your old flame MIPS machine. The instruction

```
LADD rt, rs, offset
```

has the RTL semantics

```
REG[rt] := REG[rt] + MEM[REG[rs] + offset]
```

Having also become an expert in micro-architecture, you believe that you can support this instruction without increasing the clock cycle time of your MIPS.

Assuming you can pull this off, how low must be the CPI of the new machine for this enhancement to improve performance of the application?

Give a couple of reasons why you expect the CPI to increase with this enhancement.

Give a brief sketch of how you might modify the microarchitecture of the basic MIPS pipelined datapath to support this instruction without severely impacting the cycle time.

What changes are required to make hazard resolution work properly? What are likely to be the aspects that make it difficult to maintain the cycle time?

Initial CPI = 2.2 LADD eliminates 5% of instructions.
At same CT, new CPI must be < 2.2/.95 ~ 2.3

In a general sense you might imagine that doing the same work in fewer instructions might raise the CPI, but that doesn't answer the question. They might pipeline just as well as the old instructions, maintaining the same CPI and just improving execution time. The reason that almost works is that instructions dependent on the LADD will stall because you cannot forward the value till later. However, such an instruction would otherwise be dependent on the ADD following the load. More accurate is that the load and the add can no longer be separated by independent instructions. The new stall is when the data operand of the LADD itself (not the address operand) is dependent on a previous instruction. Now the LADD will stall even though the LOAD portion could go forward. Also, data misses will be amortized over few instructions.

There are two good design solutions. The "Stanford MIPS style" option is to add a sixth stage between MEM and writeback to do that ADD. The data operand, which is carried to MEM for the STORE, must be carried to the ADD stage.
Note that adding a stage does not increase the CPI if no LADDs were used. It simply means that more values will be forwarded, since WB is further delayed. The change required is another level of forwarding. This requires an additional set of data wires along the length of the datapath and widens the forwarding mux. The logic for determining the mux selects is hardly any worse. LADDs cannot forward except from the ADD stage. The increase in cycle time is due to widening each bit slice of the datapath and the additional steering logic on the mux.

The "Berkeley/Sun RISC/Sparc style" would be to split the LADD at the decode stage into three micro ops. The first is essentially a load, the second is a NOP, and the third is the ADD. Observe that the third brings the data operand into the EX stage as the loaded value is produced from the MEM stage. No new wiring is required. We simply feed the loaded value back. Of course, this is no faster than issuing two instruction – and it prevents the compiler from filling the load delay slot, but it does reduce code size. It has no impact on the cycle time.