1. A programmer writes a program with all output directed to `stderr` and it runs rather slowly. He changes the program to direct all output to `stdout` (instead of to `stderr`) and it runs much faster. Why did he observe this difference in run times?

2. A mathematician has rewritten a single thread C program so that all the computations can be done in parallel (with some shared data) using 4 concurrent threads on a 4-cpu system. No one else is using the machine and he expects the new program to complete in one fourth the time of the old one. The new program is faster but only by about a factor of three (not four). Why has he achieved less speedup than he expected?

3. Rather than deal with the complexity of caches, why don’t hardware designers forget about slow dynamic RAM (50-100 nSec access time) and fabricate main memory entirely out of fast (5-6 nSec. access time) static RAM?

4. The canonical model of a process requires a minimum of three data segments: code, data, and stack. What is the function of each of these segments with regard to execution of the process? What permissions (read, write, execute) are usually put on each of the segments? What happens if you add multiple threads to the program?

5. What is the difference between a “ripple” adder and a “carry lookahead” adder? Which is best for performance and why?

6. Is there anything in the `hardware` which would restrict a programmer from having two threads, each running on its own CPU, writing the same global variable? Should there be?

7. What are relative speeds for accessing variables from registers, L1 cache, L2 cache, main memory (of DRAM), memory paged out to the swap device? Why is the L1 cache split into two separate sections (data and instructions)?

8. Assume that we have a standard 5-stage (IF, ID, EX, MEM, WB) integer pipeline MIPS CPU. Assume that all possible forwarding circuitry which we discussed is in place. Consider the following assembly program fragment:

   ```assembly
   lw $t1, 32($fp)
   lw $t2, 36($fp)
   add $t3, $t2, $t4
   sub $t6, $t1, $t5
   addi $t5, $t6, 128
   ```

   8a. Will the forwarding be able to eliminate all stalls due to RAW hazards which will occur in executing the above code? Why or why not? If not, could you eliminate the stall(s) by re-arranging the code (be specific)?

   8b. Will there be any stalls due to WAR hazards in executing the above code? Why or why not?
9. Assume that we have a standard 5-stage (IF, ID, EX, MEM, WB) integer pipeline MIPS CPU. Assume that branches are resolved at the end of the ID stage using an extra adder and comparator (as discussed in class). Consider the following assembly program fragment:

```assembly
lw $t5, 32($fp)
sw $t4, 36($fp)
add $t4, $t4, $t5
add $t1, $t2, $t3
beq $t0, $t1, target
nop
```

9a. Draw the timing diagram for execution of the above instructions (assuming that forwarding is in effect if needed) in the order written. When does the pipeline stall and why?

9b. Can you reorder the instructions and schedule the branch delay slot (i.e. remove the `nop` and replace it with one of the other instructions) in such a way as to execute without any stalls? If so, give the reordering and explain why it works.

10. How would you construct a pipelined fixed point (32-bit) multiplier for the EX stage in the MIPS CPU? How many stages would the multiplier have?

11. I/O operations (for example, to the disk) usually require two phases with an indeterminate amount of time in between. Why? How are interrupts and DMA transfers made use of in I/O so that CPU time is not wasted?

12. Sometimes large matrices must be accessed along both rows and columns (e.g. matrix multiplication). If matrices are stored “row-major” (as in C/C++) then access along columns can be expected to be much slower (why?). What precautions can be taken to make sure that column access is as fast as possible?

13. Explain the three main types of hazards which exist in pipelined architecture, and give some examples.