CSC 462/562 Computer Architecture
Homework #2: Due Wednesday, February 6

This assignment covers appendix C.1-C.3. Answer question 1 and four of the remaining five problems in all. Word process your answers (a spreadsheet can be used for #1). Submit by hardcopy if possible.

1. Given the following RISC-V code

```
loop:   lw      x1, 0(x2)
        lw      x3, 0(x4)
        add     x5, x1, x3
        lw      x6, 0(x5)
        subiw   x6, x6, 1
        sw      x6, 0(x5)
        addiw   x2, x2, 4
        addiw   x4, x4, 4
        bne     x5, x7, loop
```

   a. Show the timing diagram for how one full iteration of this code will execute on the original 5-stage pipeline with no scheduling and assuming branches result in an updated PC in the MEM stage. Show the first full iteration plus the first lw of the second iteration.
   b. Repeat part a assuming the revised 5-stage pipeline with forwarding and branches determined in the ID stage.
   c. Assuming the revised 5-stage pipeline, schedule the code to remove all stalls and fill the branch delay slot.

2. Given the following RISC-V code:

```
loop:   lw      x1, 0(x2)
        lw      x3, 0(x4)
        bne     x1, x3, else
        lw      x5, 0(x1)
        addiw   x5, x5, 1
        sw      x5, 0(x1)
        j       next
else:   subiw   x6, x6, 1
next:   addiw   x2, x2, 4
        addiw   x4, x4, 4
        bne     x6, x0, loop
```

   This code contains a loop as well as an if-else statement.
   a. Using the original 5-stage pipeline (no forwarding, branches determined in the MEM stage), and assuming the first bne is not taken but the bottom bne (for the loop) is taken, state where and how many stalls arise (you do not need to provide a timing diagram).
   b. Repeat part a assuming the first bne is taken.
   c. Repeat part a using the revised 5-stage pipeline with forwarding and branches determined in the ID stage.
   d. Repeat part c assuming the first bne is taken.
   e. Attempt to schedule the code to remove as many stalls as possible.
   f. Compare the efficiency of your code form part e to those in a-d.

3. In the sample problems for appendix C, we looked at a couple of modifications to the pipeline to accommodate register-memory ALU operations (in one case, we reversed the order of the EX and MEM stages, in the other, we added another ALU stage). Another change we could make to the RISC-V 5-stage pipeline is to add a second MEM stage. The new pipeline stages might look like this: `IF – ID – MEM1 – EX – MEM2 – WB` For all of our current RISC-V instructions, the pipeline works the same as before except that nothing happens in the MEM1 stage. Three changes however are made available. First, we add register-memory ALU operations whereby one datum is retrieved from memory in the MEM1 stage and used in the EX stage (in such a case, nothing happens in the MEM2 stage as we saw with all ALU operations before). Second, we add an indirect addressing mode for loads/stores in which the pointer address is fetched in MEM1 and used to load or store the datum in MEM2. Third, we add a MOVE instruction that fetches a datum in MEM1 and stores it to another location in MEM2. Note that because there is no preceding EX stage for MEM1, the only addressing mode available in MEM1 is direct memory referencing. For MEM2, we can use base-displacement.
    a. What forms of forwarding are needed to accommodate this new version of the pipeline?
    b. How does this change to the pipeline impact existing stalls and will this create any new stalls?
    c. This new pipeline also brings about a new structural hazard. What is it and why?

4. Let's compute the cost of changing our RISC-V processor from non-pipelined to pipelined. The cost is based on three things: the number of latches (registers) used to prevent a value from moving from one stage to the next, the forwarding logic and forwarding pathways, and the stall logic. Assume that each register latch costs 4 units, the logic for each possible forwarding path costs 2 units, the logic for each stall costs 2 units, and each possible forwarding path costs 1 unit. What is the total cost? HINT: refer to figures C.20, C.22, C.23 and C.24.

5. RISC-V use PC-relative branches meaning that the branch target location is always PC + offset where the offset is part of the instruction. So for instance, J Top is not PC ← Top but PC ← PC + Top where Top is a distance from the current instruction to the label in the code. The advantage of this is that the distance (displacement) is usually fairly small, keeping the number of bits down in the instruction. Consider changing the RISC-V pipeline so that there are two types of branches, absolute and PC-relative. Absolute branches do not require an addition and so we can just send the address as fetched from the instruction cache right to the instruction cache to fetch the next instruction. If we can decode enough of the instruction in the IF stage to know it is an absolute branch, we can send the address right to the cache immediately before the IF stage ends and thereby have a 0-cycle branch penalty. Assume 30% of conditional branches and 70% of unconditional branches can use this new type of branch. Also assume an instruction mix of 18% conditional branches, 2% unconditional branches, and 80% of all other instructions. Finally, assume 85% of all conditional branches are taken. What is the average branch penalty per instruction? NOTE: absolute conditional branches that are not taken have a 1-cycle penalty because the PC+4 address should have been used rather than the absolute address, and PC-relative branches that are taken also have a 1-cycle penalty because they are determined in the ID stage.

6. Let's compare an optimized versus non-optimized program on the RISC-V pipeline. Make the following assumptions:

- 50% of all load operations are followed by an ALU operation that uses the loaded value but the compiler is able to schedule away 80% of these stalls
- 75% of conditional branches immediately follow an ALU operation that computes the condition but the compiler is able to schedule away 60% of these stalls
- 10% of conditional branches immediately follow a load that loads the result of a condition being tested and the compiler is not set up to schedule this form of stall
- the compiler can fill the branch delay slot 65% of the time

Test this on a benchmark with 41% loads, 12% stores, 31% ALU operations, 11% conditional branches and 5% unconditional branches. Assume no other source of stalls/penalties. Compute the speedup from optimization (recall that the ideal CPI is 1, we add to this the stalls).