CSC 462/562 Computer Architecture
Homework #2 answer key

1. See separate spreadsheet.

2.
```
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. 2 stalls after second lw, 2 stalls after the third lw, 2 stalls after the addiw, and after the j the next 3 instructions are flushed even though two of them, addiw x2 and addiw x4 will both be re-fetched and executed, and anything fetched, and finally 3 instructions will be fetched and flushed after the last bne if the loop is repeated.
b. 2 stalls after the second lw, the 3 instructions after the first bne are all fetched and flushed, and the 3 instructions after the second bne are all fetched and flushed if the loop is repeated.
c. 2 stalls after the second lw (NOTE: even with forwarding, the result of x3 is needed in the ID stage, not the EX stage, so this results in a 2 cycle penalty), 1 stall after the third lw, 1 cycle of branch penalty after the j, and 1 cycle of branch penalty after the last bne
d. 2 stalls after the second lw, the third lw is fetched and flushed after 1 cycle, and 1 cycle after the second bne
e. There is insufficient instructions to fill all of the stalls but we can come very close. The two last two addiw's (of x2 and x4) can be moved to fill 2 of the stall/penalty slots. I move one into the RAW hazard slot after the second lw, and the other into the branch delay slot for the last bne. Within the else clause, the sw can be moved into the branch delay for the j instruction. This leaves 1 branch delay unfilled (the first bne), the RAW hazard slot after the lw in the if clause and a RAW hazard between the subiw and bne if the else clause executes. NOTE: you could move the lw x5 up IF we are guaranteed not to need x5 later in the program.
```
loop:   lw      x1, 0(x2)
        lw      x3, 0(x4)
        addiw   x2, x2, 4
        bne     x1, x3, else
        lw      x5, 0(x1)          // 1 stall after this lw
        addiw   x5, x5, 1
        j       next
        sw      x5, 0(x1)          // fills the branch delay of the j
else:   subiw   x6, x6, 1          // 1 stall here when else is executed
next:   bne     x6, x0, loop
        addiw   x4, x4, 4          // fills the branch delay of the bne
```

f. From part a, the loop takes 22 cycles. From part b, the loop takes 15 cycles. From part c, the loop takes 15 cycles. From part d, the loop takes 11 cycles. The scheduled code takes 11 cycles if the first bne is not taken and 9 cycles if it is.

3.

a. As any address retrieved in the MEM1 stage is used in either or both the EX and MEM2 stages but nowhere else, there will be nothing to forward from MEM1. We do need to be able to forward from EX and MEM2 to MEM1 though in the case where a later instruction loads a datum in the MEM1 stage and that address comes from the previous instruction's EX or MEM2 stage. We might need to forward from MEM1 to ID. All other forwarding remains as is (e.g., EX to EX, EX to ID, MEM2 to EX, MEM2 to MEM2 and MEM2 to ID)/

b. There will be no stalls introduced from adding the MEM1 stage because this stage is only producing addresses or data for the same instruction's EX or MEM2 stages. Similarly, an existing stall from a load to an ALU operation remains the same as we forward from MEM2 to the next instruction's EX stage. Loads and ALU operations to conditional branches previously had 2 and 1 cycle stalls but now these are 3 and 2 cycle stalls because the result from the ALU operation is now in stage 4 instead of 3 and the result from the load is now in stage 5 instead of 4. A new stall is introduced if a something fetched from memory in one instruction is used in one of the next two instructions as an address in its MEM1 stage. This creates either a 1 or 2 cycle stall as shown below.

```
IF      ID      M1      EX      M2      WB
        IF      ID      s       s       M1      EX      M2      WB

IF      ID      M1      EX      M2      WB
        IF      ID      s       M1      EX      M2      WB
```

c. We could have two instructions attempting to access the data cache at the same time. Consider the following sequence of instructions.

```
lw  x1, 0(x2)          IF      ID      M1      EX      M2      WB
add x3, x4, x5                 IF      ID      M1      EX      M2      WB
add  x6, x7, 0(x8)                     IF      ID      M1      EX      M2      WB
```

The lw is in MEM2 while the second add is in MEM1 at the same time and they are using the same L1 data cache, so the second add must stall because of this structural hazard.

4. Answers will vary depending on which version of the pipeline you are referencing. Obviously, we want the one with forwarding. Are branches computed in the ID stage or MEM stage? And should we take into account the load → branch and ALU → branch stalls that are not in figure C.22? Also note that there appear to be 4 forwarding conditions in table C.23 but 6 pathways in figure C.24. I will assume 4 conditions and 6 paths. The registers are NPC (IF/ID, ID/EX), IR (all 4 latches), A (ID/EX), B (ID/EX and EX/MEM), IMM (ID/EX), ALU output (EX/MEM, MEM/WB), LMD (MEM/WB). The number of registers will vary if you assume branches are computed in WB and handled in MEM. My count is 13 registers. There are 4 conditions for stalls (2 in C.22 plus the two dealing with branches), 4 conditions for forwarding and 6 forwarding paths. This gives me a cost of 4 * 13 + 2 * (4 + 4) + 1 * 6 = 74 units.

5. 18% conditional branches of which 30% use the new mode and 2% unconditional branches of which 70% use the new mode. The old mode, PC+offset, has a 1 cycle penalty if the branch

is taken (85% of the time, untaken branches have no penalty because we would have already fetched the next sequential instruction). So for this case, we have a penalty of .18 * .7 * .85 * 1 + .02 * .3 * 1 = .113. Using the new mode, 85% of conditional branches are taken and 100% of unconditional branches are taken, and in these cases, the penalty is 0. If the conditional branch is not taken (15% of the time), then the penalty is 1. This is .18 * .3 * .15 * 1 = .008. So, our overall penalty is .113 + .008 = .121

6. The benchmark is 41% loads, 12% stores, 31% ALU, 11% conditional and 5% unconditional branches. We have 1 cycle of stall from Load → ALU (which occurs 50% of all loads), 1 cycle of stall from ALU → branch (75% of all conditional branches), 2 cycles of stalls from Load → branch (10% of all conditional branches), and 1 cycle penalty from branch delays (100% of all branches). Without compiler optimization, we have all of these penalties giving us .41 * .50 * 1 + .11 * .75 * 1 + .11 * .10 * 2 + (.11 + .05) * 1 = .47. With compiler optimizations, we remove 80% of the Load → ALU stalls, 60% of the ALU → branch stalls, 65% of the load → branch stalls, and 65% of the branch delay penalties. This gives us .41 * .50 * .20 * 1 + .11 * .75 * .40 * 1 + .11 * .10 * 1.00 * 2 + (.11 + .05) * .35 * 1 = .15. So our two CPIs are 1.47 and 1.15. Using the optimizing compiler provides a 1.47 / 1.15 = 1.28 or 28% speedup.