Appendix C sample problems (sections C.1-C.3)

NOTE: using landscape orientation because of the size of some of the spreadsheets

1. Given the RISC-V code below,
   a. show how the code executes on the original 5-stage pipeline assuming no forwarding, the MUX for the PC is in the MEM stage, and no scheduling; show the full loop plus the impact of the branch
   b. show how the code executes on the revised 5-stage pipeline assuming forwarding and branches determined in the ID stage, but no scheduling; show the full loop plus the impact of the branch
   c. schedule the code to remove as many stalls as possible. In the solution, indicate the timing diagram for each of the instructions in the loop plus the impact of the branch (that is, one more instruction afterward)
   d. assuming 1000 iterations of the loop, compute the speedup from b over a and c over a.

```
Loop: lw      x2, 0(x1)
      lw      x3, 0(x2)      // an indirect memory reference, R1 is a pointer
      addiw   x4, x3, 1
      sw      x4, 0(x2)
      addiw   x1, x1, 4
      subiw   x5, x5, 1
      bne     x5, x0, Loop
```

NOTE: I am using MM for the MEM stage to squeeze this onto the page, s for stall, fl for flush.

a.

lw x2, 0(x1)	IF	ID	EX	MM	WB																		
lw x3, 0(x2)		IF	s	s	ID	EX	MM	WB															
addiw x4, x3, 1					IF	s	s	ID	EX	MM	WB												
sw x4, 0(x2)								IF	s	s	ID	EX	MM	WB									
addiw x1, x1, 4											IF	ID	EX	MM	WB								
subiw x5, x5, 1												IF	ID	EX	MM	WB							
bne x5, x0, loop													IF	s	s	ID	EX	MM	WB				
next instr 1																IF	ID	EX	fl				
next instr 2																	IF	ID	fl				
next instr 3																		IF	fl				
lw x2, 0(x1)																			IF	ID	EX	MM	WB

Stalls arise for each RAW hazard, from the first lw to the second, from the second lw to the addiw, from the addiw to the sw, and from the subiw to the bne. The bne has a 3 cycle penalty in which all there instructions wind up being flushed.

b.

lw x2, 0(x1)	IF	ID	EX	MM	WB											
lw x3, 0(x2)		IF	ID	s	EX	MM	WB									
addiw x4, x3, 1			IF	s	ID	s	EX	MM	WB							
sw x4, 0(x2)					IF	s	ID	EX	MM	WB						
addiw x1, x1, 4							IF	ID	EX	MM	WB					
subiw x5, x5, 1								IF	ID	EX	MM	WB				
bne x5, x0, loop									IF	s	ID	EX	MM	WB		
next instr 1											IF	f				
lw x2, 0(x1)												IF	ID	EX	MM	WB

With forwarding, the lw → lw stall is reduced to 1 cycle as the address of the datum being fetched is available after the first lw's MEM stage and needs to be forwarded to the next lw's EX stage. The only other RAW hazards are from the second lw to the addiw and from the subiw to the bne (because the branch needs the result of the conditional in its ID stage). There is a 1-cycle branch penalty instead of 3 as in part a.

c. The scheduled code has moved the subiw as early as possible so that there is no hazard with the bne and to fill the stall between the two lw's, and the addiw for updating the byte offset of x1 up to fill the other stall after the lw. The store has been moved down to fill the branch delay slot, requiring that the offset be modified in the sw instruction. This code has no stalls at all.

```
Loop:   lw      x2, 0(x1)
        subiw   x5, x5, 1
        lw      x3, 0(x2)
        addiw   x1, x1, 4
        addiw   x4, x3, 1
        bne     x5, x0, Loop
        sw      x4, 0(x2)
```

d. The original pipeline without scheduling executed the first iteration of the loop in 19 cycles with the second iteration starting in cycle 19, thus each successive iteration completes 18 cycles after the previous iteration. The entire loop requires 999 * 18 + 19 = 18001 cycles. The revised pipeline requires 14 cycles to complete the first iteration of the loop with the second iteration starting in cycle 12, thus each successive iteration completes 12 cycles after the previous iteration. The entire loop requires 999 * 12 + 14 = 12002 cycles. This is a

speedup of 18001 / 12002 = 1.50 (50% speedup!) The scheduled code runs with no stalls or branch delays. These 7 instructions require 7 cycles once the pipeline has been filled. So we use n + k – 1 in which k = 5 and n = 7 * 1000 giving us 7004 cycles. This is a speedup of 18001 / 7004 = 2.57 or well over 150% speedup! Not bad for some minor hardware adjustments and using an optimizing compiler.

2. Do problem C.1 on page C-71 a, b and c, but assume branches are determined in the ID stage for a 1-cycle branch penalty. Add part d: schedule the code to remove all stalls and any branch delays assuming the pipeline in part c. NOTE: change the bnez to bne x4, x0, loop.

Answer:

a. x1: ld → addi
x1: addi → sd
x2: addi → sub
x2: addi → sd // this does not cause a hazard in our pipeline but could in others
x4: sub → bnez

b.

ld x1, 0(x2)	IF	ID	EX	MM	WB																	
addi x1, x1, 1		IF	s	s	ID	EX	MM	WB														
sd x1, 0(x2)					IF	s	s	ID	EX	MM	WB											
addi x2, x2, 4								IF	ID	EX	MM	WB										
sub x4, x3, x2									IF	s	s	ID	EX	MM	WB							
bne x4, x0, loop												IF	s	s	ID	EX	MM	WB				
next instr 1															IF	ID	EX	fl				
next instr 2																IF	ID	fl				
next instr 3																	IF	fl				
ld x1, 0(x2)																		IF	ID	EX	MM	WB

c.

ld x1, 0(x2)	IF	ID	EX	MM	WB									
addi x1, x1, 1		IF	ID	s	EX	MM	WB							
sd x1, 0(x2)			IF	s	ID	EX	MM	WB						
addi x2, x2, 4					IF	ID	EX	MM	WB					
sub x4, x3, x2						IF	ID	EX	MM	WB				
bne x4, x0, loop							IF	s	ID	EX	MM	WB		

next instr 1									IF	fl	fl	fl	fl	
ld x1, 0(x2)										IF	ID	EX	MM	WB

d. Scheduled code:

```
Loop:  ld     x1, 0(x2)
       addiw  x2, x2, 4
       subiw  x4, x3, x2
       addiw  x1, x1, 1
       bne    x4, x0, loop
       sd     x1,-4(x2)
```

We move the addiw of x2 to fill the RAW hazard delay between ld and addiw x1 and the subiw up to fill the RAW hazard delay between it and the bne (notice that because of forwarding, there is no need of a stall between the addiw of x2 and its use in subiw). We move the sd into the branch delay slot, adjusting the offset because x2 has already been incremented.

3. Given the following RISC-V code, assuming we have forwarding and a single cycle branch penalty. First explain where stalls arise and second schedule the code to remove all stalls.

```
Loop:  lw     x1, 0(x7)
       lw     x2, 0(x1)
       lw     x3, 0(x8)
       lw     x4, 0(x3)
       addw   x6, x2, x4
       sw     x6, 0(x2)
       addiw  x7, x7, 4
       addiw  x8, x8, 4
       subiw  x5, x5, 1
       bne    x5, x0, Loop
```

Answer: 1 stall arises after each of the first and third lw because of the data hazard loading the pointer and then loading the datum. 1 stall arises after the fourth lw because of the data hazard between loading the datum and using it in the addw. 1 stall arises between the subiw and bne because the bne needs the value in ID but its not available until after subiw EX. Finally, there is the branch delay slot. Scheduling the code to remove these stalls is done by first distributing the 2 lws of the pointers from the 2 lws of the data. Thus, we load into x1 and x3, and then use x1 and x3 to load into x2 and x4 respectively. We also move the subiw up to move it away from the bne and we can take any of the addiw or sw to move into the branch delay slot. There are several solutions. Here is mine.

```
Loop:  lw     x1, 0(x7)
       lw     x3, 0(x8)
```

```
lw      x2, 0(x1)
lw      x4, 0(x3)
subiw   x5, x5, 1
addw    x6, x2, x4
addiw   x7, x7, 4
addiw   x8, x8, 4
bne     x5, x0, Loop
sw      x6, 0(x2)
```

4. Let's compute the degradation of the RISC-V's pipeline performance due to realistic assumptions of hazard situations. A benchmark consists of 27% loads, 10% stores, 45% ALU operations, 15% conditional branches, 3 % unconditional branches. We use the version of RISC-V with forwarding and branches computed in the ID stage. Assume 10% of all loads/stores use indirect addressing so we have a Load followed immediately by a Load/Store. Assume 40% of all Loads have an ALU operation immediately afterward that uses the loaded value. Assume 60% of all conditional branches immediately follow an ALU operation that places a result in the register to be tested in the branch. Assume a further 10% of all conditional branches immediately follow a load of the register to be tested in the branch (note that this requires a 2-cycle stall because the datum is loaded in the MEM stage but needed in the branch's ID stage). An optimizing compiler can schedule code to eliminate 50% of the indirect memory reference stalls, 60% of all load-ALU stalls, fill 65% of the branch delay slots and fill 45% of the stalls between ALU-branch and 50% of one stall from the load-branch. Compare the difference between the unoptimized and optimized versions and the optimized and ideal (no stalls).

   Answer:
   Our source of stalls are load-ALU, ALU-branch, load-branch, and the branch delay slots.
   - Memory indirect stalls: .10 * (27% + 10%) * 1 = .037
   - Load-ALU: .40 * .27 * 1 = .108
   - ALU-branch: .60 * .15 * 1 = .09
   - Load-branch: .10 * .15 * 2 = .03
   - Branch delay: (.15 + .03) * 1 = .18
   - CPI Unoptimized = 1 + stalls = 1 + .037 + .108 + .09 + .03 + .18 = 1.445

   Compiler optimizations will resolve many of these
   - Memory indirect stalls: .037 * .5 = .0185 (we remove 50% of these)
   - Load-ALU: .108 * .4 = .0432 (we remove 60% of these)
   - ALU-branch: .09 * .55 = .0495
   - Load-branch: .03 * .5 = .015
   - Branch delay: (.15 + .03) * 1 * .35 = .063
   - CPI = 1 + stalls = 1 + .0185 + .0432 + .0495 + .015 + .063 = 1.1892

The ideal machine has a CPI of 1.0, the version with the optimizations is 1.1892 and the unoptimized is 1.445. The idea machine is 19% faster than the optimized machine, 44.5% faster than the unoptimized machine, and the optimized version is 1.445 / 1.1892 = 1.215 or about 22% faster than the unoptimized.

5. Sample problem 2 from the Appendix A problems asked what would happen if we exchange the EX and MEM stages in the 5-stage RISC-V architecture. Now that we have added a pipeline, let's reconsider this exchange from a performance and hardware point of view. This makes our pipeline now IF – ID – MEM – EX – WB allowing us to have register-memory operations (with the proviso that all memory references are now direct memory references and there are no base-displacement references). Assume branches are still computed in the ID stage.
    a. How would you set up forwarding (that is, from what stage to what stage)?
    b. In spite of the new forwarding, what new sources of stalls arise and what sources of stalls go away?

    Answer:
    a. We would forward MEM to MEM to allow for loads followed by stores as in

        ```
        lw      x2, 0(x3)
        sw      x2, 0(x4)
        ```

        We would forward EX to EX to allow for ALU operations that compute a value and use the value in the next ALU instruction. We would forward from EX to MEM so that a computed value could be stored, but as shown below, this will result in a stall even with the forwarding. Finally, we would forward MEM to ID and EX to ID for conditional branches even though these would still result in stalls.

    b. A load results in the value being available by the end of the load's 3rd stage so that it can be used in the next instruction if that next instruction is an ALU operation, so previous load → ALU stalls are removed:

        ```
        lw      x2, 0(x3)
        addiw   x2, x2, 1
        ```

        ALU operations that compute values that are then stored to memory are a new source of stall:

        ```
        addiw   x2, x1, 1
        sw      x2, 0(x3)
        ```

        because the addiw does not compute the addition until its 4th stage and the next instruction needs the datum in its 3rd stage. Another source of stall is an ALU whose result is used in a conditional branch:

        ```
        addi    x2, x3, x4
        bne     x2, x5, loop
        ```

        The addi does not have the add completed until the end of its 4th stage but bne needs the value ready at the beginning of its 2nd stage. This requires 2 cycles of stalls. Previously, a load followed by a conditional branch that used the loaded value required 2 cycles of stalls but now this is reduced to one:

```
lw      x2, 0(x3)
bne     x2, x4, next
```

because the load receives the datum in its 3rd stage so its available to the next instruction at the beginning of its third stage but is needed by the bne at the beginning of its second stage. Note that load to load and load to store do not cause stalls as they did not when the pipeline had EX before MEM. Previously though, the load of a pointer to be used in the next load operation caused a stall:

```
lw      x2, 0(x3)
lw      x4, 0(x2)
```

because the value of x2 was needed in the second lw's EX stage but not available until the end of the first lw's MEM stage. Now however, this is not the case because the EX stage follows the MEM stage. The two sets of timing diagrams are shown below.

```
lw      x2, 0(x3):    IF    ID    EX    MEM   WB
lw      x4, 0(x2):          IF    ID    s     EX    MEM   WB
```

becomes

```
lw      x2, 0(x3):    IF    ID    MEM   EX    WB
lw      x4, 0(x2):          IF    ID    MEM   EX    WB
```

Remember, we are no longer using EX to compute the effective address as all addresses are direct.

6. Do problem C.3 a, b and c on page C-73.

   Answer:

   a. The original machine had a clock cycle of 7 ns. Now we divide the fetch-execute cycle into five stages where the longest stage (MEM) is 2 ns, so we can retune the clock to 2 ns. However, the pipeline register delay is .1 ns, so the new machine's clock cycle time is 2.1 ns.
   b. The ideal CPI = 1. With 1 stall every 4 cycles, we add 1/4 = .25 to this, so the CPI is 1.25.
   c. CPU Execution Time = IC * CPI * Clock Cycle Time. In this case, IC remains the same whereas clock cycle time is 7 ns for the unpipelined machine and 2.1 ns for the pipelined machine and the CPI for the unpipelined machine is 1 and the pipelined machine is 1.25. This gives us a CPU execution time of IC * 1 * 7 ns = 7 ns * IC for the unpipelined machine and IC * 1.25 * 2.1 ns = 2.63 ns * IC for the pipelined machine. The pipelined machine is therefore 7 / 2.63 = 2.66 times faster.

7. Let's consider the following alternation to the 5-stage pipeline, similar to what we did in problem #5 above (and problem #2 from the sample problems in appendix A). NOTE: this question is a variation of question C.5 on pages C-73-C-75 but differs in a couple of ways. The new pipeline adds a second ALU and creates two "EX" stages called ALU1 and ALU2 as follows:

   IF – ID – ALU1 – MEM – ALU2 – WB

   ALU1 is solely used to compute the effective address for any memory reference while ALU2 is used as before to handle normal ALU operations. Because we can compute an effective address before the MEM stage, this allows us to fetch one datum from memory and then use it in the ALU to give us register-memory ALU operations while not having to impose the restriction from problem #5 in which all memory references must be direct (no base-displacement). Note that ALU1 is only used to compute addresses for memory references and

that ALU2 is used for all ALU operations only. We will assume the ID stage still handles branches. With this new version of the pipeline, we can accommodate ALU operations that retrieve one datum from memory and so we can remove some loads from our programs if the loaded datum is only used in one instruction.

a. What forwarding is needed in this new version of the pipeline?
b. What sources of stalls are removed and what new sources of stalls are introduced in spite of the forwarding from part a?
c. Given your answers to a and b, is this change worthwhile?

Answer:

a. ALU1 only computes effective addresses used in the same instruction's MEM stage, so there is never any forwarding from ALU1. MEM could fetch a datum to be used in a branch, as part of an effective address computation, to be stored in a later store instruction, or as part of a later ALU operation. This requires that MEM forward to each of ID, ALU1, MEM but not ALU2. The reason that we don't need a MEM → ALU2 forwarding is that the MEM stage precedes the ALU2 stage so can be forwarded either through the register file by storing the datum in the WB stage or forwarded to one of MEM, ALU1 or ID stages as necessary. The ALU2 stage may produce a datum that is used in a later branch, memory address, to be stored in memory, or used in a later ALU operation, so ALU2 must forward to each of ID, ALU1, MEM and ALU2.
b. Previously, 1 cycle stalls resulted from load → ALU, load → load/store (first load is of a pointer used by the second memory operation), ALU → branch, and load → branch (2 cycle stall). The first situation is no longer a problem as the load precedes the ALU within the same instruction or is an earlier instruction with forwarding of the loaded value to the ALU. The second source of stall (with pointers) still exists as the pointer is needed in the next instruction's ALU1 stage. The third and fourth situations result in 2 and 3 cycles of stalls respectively (MEM to ID requires 2 cycles of stall which is no different from before, ALU2 to ID requires 3 cycles of stall). New sources of stalls arise. The first is if a loaded value is part of an effective address of the next instruction (this is pretty much the same as load pointer → load datum). Another is if an ALU operation is computed a datum that will be used in a later effective address (e.g., addi x2, x3, x4 followed by lw x5, 4(x2)) – this situation will require a 2 cycle penalty, an ALU operation whose value is then stored in the next instruction.
c. To answer this question, we have to consider whether there are more load → ALU RAW hazards versus ALU and load operations whose results are used in a later effective address or branch. Since the ALU2 → branch and ALU2 → load/store are multiple cycle stalling situations, this revised pipeline is probably not worthwhile unless it reduces IC substantially.