Chapter 3 sample problems part 1 (sections 3.1-3.5 and appendix C.7)

1. Given the following RISC-V FP code and assuming the latencies from figure C.28 (page C-47) and assuming a 1 cycle branch penalty but assuming the store and the fmult.d can enter the MEM/WB stages at the same time
    a. describe the stalls that arise
    b. schedule the code to remove as many stalls as possible
    c. unroll the loop a sufficient number of times and schedule the code to remove all stalls
    d. compute the speedup of the unrolled and scheduled code over the original

```
loop:   fld      f1, 0(x1)
        fmult.d  f3, f1, f2          // f2 is a scalar as in doing
        fsd      f3, 0(x1)           // x[i] = x[i] * c;
        addiw    x1, x1, 8
        bne      x1, x2, loop
```

    Answer:

    a. 1 stall after the fld because of the RAW hazard on loading f1, 5 stalls after the fmult.d because of the latency for the multiplication (note that a 6th stall would normally be needed if we could not permit the structural hazard of fmult.d and fsd entering the MEM and WB stages at the same time), 1 stall after the addiw, and the branch delay slot = 8 total.
    b. We move the addiw up and the fsd down but that's all we can do

```
loop:   fld      f1, 0(x1)
        addiw    x1, x1, 8
        fmult.d  f3, f1, f2
        // 4 stalls here
        bne      x1, x2, loop
        fsd      f3, -8(x1)
```

    The scheduled code has 4 stalls and no branch delay instead of 7 stalls and one branch delay from the original code

    c. By unrolling the loop, we will have several fld instructions to fill the RAW hazard stall after the first fld. We can also move the addiw up before the fsd to fill one stall after the fmult.d, but we need more, so we unroll the loop. We need a total of 6 cycles' worth of instructions between each fmult.d and its fsd, so we unroll the loop for a total of 6 iterations (the 6th instruction is the addiw). In scheduling the code, we move one fsd beneath the bne to fill the branch delay slot.

```
loop:   fld      f1, 0(x1)
        fld      f4 8(x1)
        fld      f6, 16(x1)
        fld      f8, 24(x1)
        fld      f10, 32(x1)
        fld      f12, 40(x1)
        fmult.d  f3, f1, f2
        fmult.d  f5, f4, f2
        fmult.d  f7, f6, f2
        fmult.d  f9, f8, f2
        fmult.d  f11, f10, f2
        fmult.d  f13, f12, f2
        addiw    x1, x1, 48
        fsd      f3, -48(x1)
        fsd      f5, -40(x1)
        fsd      f7, -32(x1)
```

```
        fsd     f9, -24(x1)
        fsd     f11, -16(x1)
        bne     x1, x2, loop
        fsd     f13, -8(x1)
```

d. The original code executes 5 instructions in 13 cycles for a CPI of 13 / 5 = 2.6. The revised code has a CPI of 1.0 (no stalls or penalties). The speedup is therefore 2.6 / 1.0 = 2.6.

2. Repeat #1 on the following code.

```
loop:   fld     f1, 0(x1)
        fld     f2, 0(x2)
        fmult.d f3, f1, f2
        fsd     f3, 0(x3)
        addiw   x1, x1, 8
        addiw   x2, x2, 8
        addiw   x3, x3, 8
        bne     x1, x4, loop
```

answer:

a. 1 stall after the second fld, 5 stalls after the fmult.d (note that there would be a 6th stall if the fmult.d and fsd couldn't enter the MEM/WB stages at the same time), and the branch delay.

b. We can remove almost all of the needed stalls by simple scheduling, but not quite all of them.

```
loop:   fld     f1, 0(x1)
        fld     f2, 0(x2)
        addiw   x1, x1, 8
        fmult.d f3, f1, f2
        addiw   x2, x2, 8
        addiw   x3, x3, 8
        //  2 stalls
        bne     x1, x4, loop
        fsd     f3, -8(x3)
```

c. Unroll the loop for 3 total iterations to fill the needed stalls.

```
loop:   fld     f1, 0(x1)
        fld     f2, 0(x2)
        fld     f4, 8(x1)
        fld     f5, 8(x2)
        fld     f7, 16(x1)
        fld     f8, 16(x2)
        fmult.d f3, f1, f2
        fmult.d f6, f4, f5
        fmult.d f9, f7, f8
        addiw   x1, x1, 24
        addiw   x2, x2, 24
        addiw   x3, x3, 24
        fsd     f3, -24(x3)
        fsd     f6, -16(x3)
        bne     x1, x4, loop
        fsd     f9, -8(x3)
```

d. The original code has 8 instructions which take 15 cycles to complete or a CPI of 15 / 8 = 1.875, the unrolled and scheduled code has 16 instructions which take 16 cycles to complete for a CPI of 1.0. The speedup is 1.875 / 1.0 = 1.875.

3. Explain the complications in compiler-based loop unrolling and scheduling.

   Answer: First, the compiler must determine the number of iterations required. This involves a little math with respect to understanding the latencies of RAW hazards because of floating point operations and determining the maximum latency. Second, the compiler must determine how many "neutral" instructions exist that can be moved to fill stalls to reduce that "longest latency" (and other RAW hazard stalls and branch delay slots). For instance, in comparing sample problem 1 to problem 2, both used fmult.d's which have 5 cycles of stalls between the fmult.d and fsd, but because there were more neutral instructions in problem 2, the compiler needed fewer iterations of an unrolled loop to fill the stalls. Third, the compiler must assign registers for each unrolled iteration. This is no problem as long as there are a sufficient number of registers available, but longer duration FP operations like FP division may result in not having enough registers to unroll a loop a sufficient amount of time to remove all stalls. Fourth, the compiler needs to alter the displacements on memory references for loads/stores that take place after the register has been manipulated by an addiw or subiw. This ensures that the proper array value is accessed in memory for the load/store operations. Finally, not shown in these example problems is that the compiler must add code to handle extra loop iterations if the number of unrolled iterations is not a factor of the total number of iterations. For instance, sample problem 2 unrolled the loop for 3 total iterations but if the loop executed 50 times then the compiler must insert "start-up" or "wrap-up" code to handle the leftover iterations.

4. Recall that a simple 1-bit branch predictor has a higher miss-prediction rate than a 2-bit predictor because, upon reaching a branch, the 1-bit predictor would be set to the last result of the branch, which is probably "not taken" but most branches dealing with loops are taken. But note that conditional branches are not just used for loops; they are also used for if-else statements. The behavior of a conditional branch for an if-else statement will likely be different than that of a loop (that is, the branch may not be taken nearly as often in an if-else statement). A difference between a conditional branch for a loop and a conditional branch for a selection statement is the offset: in a loop the offset is negative to move backward in the program while in a selection statement it is positive to branch around the if clause (the branch around the else clause is an unconditional branch). We modified our 5-stage pipeline earlier so that branches were handled in the ID stage. If we provide our 5-stage pipeline with a prediction buffer, we are fetching that prediction in the IF stage. Let's assume we have time during the IF stage to perform the branch location computation (take the offset from the instruction, sign extend it and add it to the PC). Before the IF stage ends, we modify the PC to be PC + 4 if the prediction is that the branch is not taken and modify the PC to PC + Offset if the prediction is taken. This worked well if we had accurate predictions and had buffer hits (miss-predictions and buffer misses both resulted in a 2 cycle penalty, one to determine the result of the branch, one to update the buffer). Let's make a slight modification to this strategy since negative offsets are for loops and branches for loops are usually taken. The process now works as follows. In the IF stage, fetch the instruction and the branch prediction. If the instruction is a branch whose offset is negative or if the offset is positive and the branch is predicted as taken, sign extend the offset and add it the PC and fetch the next instruction from that location. If the offset is positive and the prediction is to not branch, or if the prediction buffer results in a miss, or the instruction is not a branch, use PC + 4 for the next instruction fetch. In effect, we are always branching for loops and using the prediction for other branches. If we predict correctly, the penalty is 0. On a miss-prediction, we do not modify the buffer, so the penalty is 1 instead of 2 (1 cycle because we know the result in the ID stage, but we do not bother to update the prediction buffer). Assume that branches for loops are taken 99% of the time and that our prediction for selection statements is 75% accurate. If a benchmark consists of 3% unconditional branches and 17% conditional branches where half of those branches are for loops and the other half for selection

statements, and where the prediction buffer has a hit rate of 90%, compare the performance of this new scheme to using the branch prediction buffer without this distinction between negative and positive branches, to one in which we do not use branch prediction but instead the compiler successfully schedules an instruction into the branch delay slot 70% of the time and to one that has no branch handling scheme at all.

Answer: For no scheme at all, there is always a 1 cycle penalty for any branch, leading to a CPI of 1 + (17% + 3%) * 1 = 1.2. For a scheme where the compiler successfully fills the branch delay slot 70% of the time, we have 1 + (17% + 3%) * .30 * 1 = 1.06. The original branch prediction buffer has a 0 cycle penalty for unconditional branches and for conditional branches found in the prediction buffer that were accurately predicted, and a 2 cycle penalty for buffer misses and miss-predictions. This gives us a CPI of 1 + .17 * (.10 + .50 * .01 + .50 * .25) * 2 = 1.078. The new scheme has a 0 cycle penalty for unconditional branches and a 0, 1 or 2 cycle penalty for conditional branches depending on if accurately predicted, miss-predicted or miss-predicted selection statements, or a prediction buffer miss. This gives us a CPI of 1 + .17 * (.50 * (.01 * 1 + .99 * 0) + .50 * (.90 * (.25 * 2 + .75 * 0) + .10 * 2) + .03 * 0 = 1.056. The new scheme provides a slight improvement over having the compiler fill the branch delay slot and a larger improvement (about 2%) over the original prediction buffer.

5. Provide an example of code in which the result of one condition can be used to accurately predict the result of a second condition.

   Answer: A simple example comes from a programmer using two if statements rather than an if-else statement:

   ```
   if(grade >= 60) pass++;
   if(grade < 60) fail++;
   ```

   The second condition will only be true if the first is false. If a compiler was programmed to, it could rewrite this as an if-else statement with one condition, so this is perhaps not the best example. A similar but better example is shown below.

   ```
   if(age>=21) adult=true;
   if(state==“KY”) resident=true;
   if(adult&&resident) canVote=true;
   ```

   Here, we could combine the first two conditions and eliminate the third entirely but doing so would mean that we would not have values for `adult` and `resident`, which we may need at some future point in the code. So here, the third condition is true only if the first two conditions are true allowing a correlated branch predictor to provide superior performance.

6. The iCore 7 920 includes a “loop exit” predictor. Research this idea. What is a loop exit predictor? How does it differ from an ordinary branch prediction buffer? Provide an example of code that would defeat the loop exit predictor.

   Answer: Most branch prediction buffers store multiple prediction bits (correlating bits, history bits) and the predicted destination address (the target address’ offset). In addition to these bits to help improve the prediction, the loop exit predictor also stores a *predicted iteration count*. As it is expected that the loop will execute some number of times, this expected number is recorded. An extra loop iteration counter (register) counts each time the branch is taken. With each fetch of the branch instruction, the iteration counter is incremented and compared against the predicted count. On reaching the predicted count, the hardware assumes the branch is not taken. If there is a miss-prediction, the expected iteration counter is updated along with the prediction bits. While most for-loops generally lead to a very accurate prediction, the following inner loop would cause problems for this approach because the inner loop’s “expected iteration count” keeps changing.

```
for(i=1;i<n;i++)
        for(j=0;j<i;j++)
```

7. For the MIPS R4000 pipeline, assume we implement a branch target buffer to predict branches. Compare the performance of this approach to assuming branches are not taken. Use a benchmark with 15% conditional branches and 4% unconditional branches where 85% of conditional branches are taken. Assume the compiler can schedule one neutral instruction in the branch delay slot 50% of the time. When switching to the target buffer, assume the compiler does not schedule any instruction into a branch delay slot and assume an accuracy of 90% and a buffer hit rate of 95%.

   Answer: For the original MIPS R4000, if branches are not taken, the pipeline continues as normal with no penalty. On taken branches, the penalty is either 3 cycles or 2 cycles depending on whether an instruction was successfully scheduled in the first branch delay slot (branches are determined in stage 4 of the MIPS R4000, so a 3 cycle penalty). With the new approach, the prediction information is fetched in the IF stage, but not received back until the IS stage (because instruction fetches take 2 cycles), so accurate predictions have a 1 cycle penalty while inaccurate predictions are not determined until the EX stage (3 cycles) but have an additional 2-cycle penalty to modify the buffer (again, cache accesses are 2 cycles). Buffer misses have the same penalty.

   For the current MIPS R4000 using assume-not-taken and scheduling
   Conditional branches: .15 * .85 * .5 * 2 + .15 * .85 * .5 * 3 (half have a scheduled instruction in the branch delay slot, no penalty when not taken) = .319
   Unconditional branches: .04 * .5 * 2 + .04 * .5 * 3 (always taken) = .1
   Total penalty = .319 + .1 = .419
   For the revised pipeline using branch prediction and no scheduling
   Conditional branches: .15 * .90 *.95 * 1 + .15 * .10 * .95 * 5 + .15 * .05 * 5 = .237
   Unconditional branches: .04 * 1 (never miss-predicted) = .04
   Total penalty = .237 + .04 = .277
   Speedup = .419 / .277 = 1.513 or more than a 50% improvement.

8. Given the following RISC-V FP code and using the latencies from figure C.28 (page C-47), show how the code executes on the Scoreboard as presented in Appendix A.8, a Tomasulo-style processor as covered in section 3.4 and 3.5, and the FP pipeline from Appendix C. Show, using a table, when each instruction reaches each of the stages of the given architecture (fetch, issue, read operands, execute, write result). Make the following assumptions:
   - for the scoreboard, an instruction is issued as soon as a functional unit is available but registers are not read until both operands are available and only one functional unit can read operands in any cycle but both operands are read in that cycle
   - for the Tomasulo architecture, only one result can be forwarded at a time but all waiting reservation stations can read a forwarded datum if they are waiting for it, given both data, the functional unit begins executing at the start of the next cycle
   - in both architectures, there are two load/store units, two integer ALU units, a single non-pipelined FP adder and a single non-pipelined FP multiplier
   - for the pipeline, an ALU operation and a store can enter MEM or WB in the same cycle
   - only one instruction can be issued per cycle

   ```
   fld     f1, 0(x1)
   fadd.d  f2, f1, f3
   fld     f4, 0(x2)
   fmult.d f5, f2, f4
   fld     f6, 8(x2)
   ```

```
fsub.d    f7, f5, f6
fsd       f7, 8(x2)
```

Answer:
Scoreboard example:

	Fetch	Issue	Read Ops	Execute	Write Result
fld	1	2	3	4	5
fadd.d	2	3	6	7	11
fld	3	4	5	6	7
fmult.d	4	5	12	13	20
fld	5	6*	7	8	9
fsub.d	6	12**	21	22	26
fsd	7***	14	27	28	n/a

* - the third fld can be issued at cycle 6 because the first fld has left its functional unit at the end of cycle 5
** - because there is only one, non-pipelined FP adder, the fsub.d cannot be issued until the fadd.d exits
*** - recall that this architecture had a separate fetch stage which fetched instructions and placed them into an instruction buffer. Even though the fsub.d is stalled before entering its issue stage, we are still able to fetch instructions until that buffer becomes full. So we can fetch the fsd in stage 7, where it sits behind the fsub.d waiting to be issued.

Tomasulo example:

	Fetch	Issue	Execute	Write Result
fld	1	2	3	4
fadd.d	2	3	5	9
fld	3	4	5*	6
fmult.d	4	5	10	17
fld	5	6	7	8
fsub.d	6	7**	18	22
fsd	7	8	23	n/a

* - the fld can execute at the same time as the fadd.d because they are in different functional units
** - because of reservation stations, the FP adder can accept this instruction immediately, unlike in the scoreboard, but the fsub.d must still wait until the adder has completed its addition before execution may begin (in this case, the fsub.d has to wait longer anyway because it is waiting for the fmult.d to write its result, thus it doesn't start executing until cycle 18)

FP pipeline

	IF	ID	EX	MEM	WB
fld	1	2	3	4	5
fadd.d	2	3	5	9	10
fld	3	5*	6	7	8
fmult.d	5*	6	9**	16	17
fld	6	7	9***	10	11
fsub.d	7	9	16	20	21
fsd	9	16	17	20	n/a

* - the fld stalls in the IF stage because the fadd.d has stalled in the ID stage, in cycle 5, the fadd.d moves into the FP adder's execute stage (A1) allowing the fld to move to ID and the fmult.d to be fetched
** - recall that we are using forwarding so the fadd.d's result is forwarded out of A4 directly to the fmult.d's M1, this happens at cycle 9

*** - since the fmult.d is stalled in its ID stage, the fld cannot move to its EX stage until cycle 9, and similarly, the fsub.d stalls it ints ID stage in cycle 9 so the fsd cannot move to its ID stage until cycle 16

Because of forwarding, the final result being stored in memory is sent to memory earlier than using Tomasulo's architecture. Even if the fsd were stalled an extra cycle because of two instructions entering the MEM stage in the same cycle, it still outperforms Tomasulo (in this example, see #9).

9. Redo #8 using the following loop. Show two complete iterations of the loop. Assume for the FP pipeline that branches are handled in the ID stage and do not move beyond the ID stage (use n/a to indicate that the instruction is no longer moving through the pipeline).

```
loop:  fld     f1, 0(x1)
       fmult.d f3, f1, f2
       fsd     f3, 0(x1)
       addiw   x1, x1, 8
       bne     x1, x4, loop
```

Answer:
Scoreboard example:

	Fetch	Issue	Read Ops	Execute	Write Result
fld	1	2	3	4	5
fmult.d	2	3	6	7	14
fsd	3	4	15	16	n/a
addiw	4	5	6	7	8
bne	5	6	9	10	n/a
fld*	11	12	13	14	15
fmult.d	12	15**	16	17	24
fsd	13	16***	25	26	n/a
addiw	14	17****	18	19	20
bne	15	18	21	22	n/a

* - after the bne, the Scoreboard would retrieve incorrect instructions until the end of cycle 10. At that point, the incorrectly fetched instructions would be flushed. The bne would not be fetched until cycle 11.
** - the second fmult.d cannot be issued until the first fmult.d exits the FP * unit.
*** - note that even if the fmult.d could have been issued earlier than cycle 15, this second fsd could not be issued until cycle 16 because the second fld and first fsd are still using the two load/store units, preventing this instruction from being issued until cycle 16
**** - since the previous two instructions stalled at the issue stage, the addiw, even though it was fetched in cycle 14, cannot be issued until cycle 17

Tomasulo example:

	Fetch	Issue	Execute	Write Result
fld	1	2	3	4
fmult.d	2	3	5	12
fsd	3	4	13	n/a
addiw	4	5	6	7
bne	5	6	8	n/a
fld*	9	10	11	13**
fmult.d	10	11***	14	21
fsd	11	12****	22	n/a

	addiw	12	13	14	15
	bne	13	14	16	n/a

* - the second fld is not fetched until after the branch result is known

** - the second fld cannot write its result in cycle 12 as the fmult.d is writing during that cycle

*** - the second fmult.d can be issued to the single FP multiplier even though the multiplier is still executing the first fmult.d because the second one is stored in a reservation station

**** - the second fsd can be issued at time 12 even though both load/store units are in use because of reservation stations

FP pipeline	IF	ID	EX	MEM	WB
fld	1	2	3	4	5
fmult.d	2	3	5	12	13
fsd	3	5*	6	12	n/a
addiw	5	6	12**	13	14
bne	6	13***	n/a		
fld	14****	15	16	17	18
fmult.d	15	16	18	25	26
fsd	16	18	19	25	n/a
addiw	18	19	20	21	22
bne	19	21	n/a		

* - the fsd is stalled in the IF stage while the fmult.d is stalled in the ID stage waiting for the fld to forward its results (at the end of cycle 4), notice also that the fsd is allowed to enter the MEM stage along with the fmult.d

** - the addiw must stall in its ID stage while the fsd is stalled in its EX stage waiting for the fmult.d to forward its results, which happens at the end of cycle 11, so both the fsd and addiw can move forward in cycle 12

*** - while the bne is stalled in the IF stage until cycle 12 because of waiting for the addiw and fsd to move forward because of their stalls, the bne must stall 1 more cycle due to a RAW hazard between the addiw and bne, so the bne does nto move into its ID stage until cycle 13

**** - anything fetched after cycle 6 is flushed, after cycle 13, so we don't fetch the second iteration's fld until cycle 14

In this example, we see Tomasulo improving over both the scoreboard and the pipeline. Tomasulo writes its result in cycles 13 and 22 while the Scoreboard does so in cycles 16 and 26 and the pipeline in cycles 12 and 25.

10. Do problem 3.7 on pages 268-269.

Answer:	Original	Rewritten
fld	f2, 0(Rx)	T9, 0(Rx)
fmult.d	f5, f0, f2	T10, f0, T9
fdiv.d	f8, f0, f2	T11, f0, T9
fld	f4, 0(Ry)	T12, 0(Ry)
fadd.d	f6, f0, f4	T13, f0, T12
fadd.d	f10, f8, f2	T14, T11, T9
fsd	f4, 0(Ry)	T15, 0(Ry)

Notice that the store is not of f4 because that causes a WAR hazard with the first fadd.d