Chapter 3 sample problems part 2 (sections 3.6-3.12)

1. Given the following RISC-V code, and assuming loads/stores and integer (addiw, subiw, bne) take 1 cycle each, the fadd.d takes 2 cycles to compute and the fmult.d takes 3 cycles to compute, unroll the loop and schedule the code on a two-issue superscalar to remove as many stalls as possible using the restriction as described below. Assume one pipeline handles all int operations (including loads and stores) and the other handles all FP operations. Compute the CPI for each. Repeat this problem assuming either pipeline can handle any instruction (including two loads/stores in the same cycle). In your scheduling, make sure you fill the branch delay slot.

```
Loop:   fld     f1, 0(x1)
        fld     f2, 0(x2)
        fadd.d  f3, f1, f2
        fmult.d f5, f3, f4                  // f4 is a scalar
        fsd     f5, 0(x1)
        addiw   x1, x1, 8
        addiw   x2, x2, 8
        subiw   x3, x3, 1
        bne     x3, x0, Loop
```

Answer: **NOTE: assume "3 cycles to compute" means 3 cycles, not 2 plus forwarding to the MEM stage of the store (that is, the distance between fadd.d and fsd is at least 2 cycles and between fmult.d and fsd is at least 3 cycles) – note added 3/13/19**

```
Loop:   fld f1, 0(x1)
        fld f2, 0(x2)
        fld f6, 8(x1)
        fld f7, 8(x2)           fadd.d f3, f1, f2
        fld f10 16(x1)
        fld f11, 16(x2)         fadd.d f8, f6, f7
        fld f14, 24(x1)         fmult.d f5, f3, f4
        fld f15, 24(x2)         fadd.d f12, f10, f11
        addiw x1, x1, 32        fmult.d f9, f8, f4
        addiw x2, x2, 32        fadd.d f16, f14, 15
        fsd f5, -32(x1)         fmult.d f13, f12, f4
        subiw x3, x3, 1
        fsd f9, -24(x1)         fmult.d f17, f16, f4
        stall
        fsd f13, -16(x1)
        bne x3, x0, Loop
        fsd f17, -8(x1)
```

CPI = 17 cycles / 24 instructions = .708

The change here is that some of the load/store operations can be moved to the second pipeline to reduce the number of cycles slightly.

```
Loop:   fld f1, 0(x1)           fld f2, 0(x2)
        fld f6, 8(x1)           fld f7, 8(x2)
        fld f10, 16(x1)         fld f11, 16(x2)
        fld f14, 24(x1)         fadd.d f3, f1, f2
        fld f15, 24(x2)         fadd.d f8, f6, f7
        subiw x3, x3, 1         fadd.d f12, f10, f11
        fadd.d f16, f14, f15    fmult.d f5, f3, f4
        addiw x1, x1, 32        fmult.d f9, f8, f4
        addiw x2, x2, 32        fmult.d f13, f12, f4
```

```
                                 fmult.d f17, f16, f4
        fsd f5, -32(x1)
        fsd f9, -24(x1)
        fsd f13, -16(x1)         bne x3, x0, Loop
        fsd f17, -8(x1)
```

Notice how two additional loads occur in the first pipeline after we are able to start adding in the second. This is needed to help full some of the stalls. Further unrolling of the loop though won't help us fill any remaining empty slots. CPI = 14 cycles / 24 instructions = .583. Note that a CPI of .5 is optimal for a two-issue superscalar, so we came close.

2. Repeat #1 using the following code.

```
        Loop:   fld      f1, 0(x1)
                fmult.d  f3, f1, f2              // f2 is a scalar
                fld      f4, 0(x2)
                fadd.d   f5, f4, f3
                fsd      f5, 0(x1)
                addiw    x1, x1, 8
                addiw    x2, x2, 8
                bne      x2, x3, Loop
```

Answer:

```
Loop:   fld f1, 0(x1)
        fld f6, 8(x1)
        fld f10, 16(x1)          fmult.d f3, f1, f2
        fld f14, 24(x1)          fmult.d f7, f6, f2
        fld f4, 0(x2)            fmult.d f11, f10, f2
        fld f8, 8(x2)            fmult.d f15, f14, f2
        fld f12, 16(x2)          fadd.d f5, f4, f3
        fld f16, 24(x2)          fadd.d f8, f7, f3
        addiw x1, x1, 32         fadd.d f12, f11, f3
        addiw x2, x2, 32         fadd.d f16, f15, f3
        fsd f5, -32(x1)
        fsd f8, -24(x1)
        fsd f12, -16(x1)
        bne x2, x3, Loop
        fsd f16, -8(x1)
```

CPI = 15 cycles / 23 instructions = .652

```
Loop:   fld f1, 0(x1)            fld f6, 8(x1)
        fld f10, 16(x1)          fld f14, 24(x1)
        fld f4, 0(x2)            fmult.d f3, f1, f2
        fld f8, 8(x2)            fmult.d f7, f6, f2
        fld f12, 16(x2)          fmult.d f11, f10, f2
        fld f16, 24(x2)          fmult.d f15, f14, f2
        addiw x1, x1, 32         fadd.d f5, f4, f3
        addiw x2, x2, 32         fadd.d f8, f7, f3
                                 fadd.d f12, f11, f3
        fsd f5, -32(x1)          fadd.d f16, f15, f3
        fsd f8, -24(x1)
        fsd f12, -16(x1)         bne x2, x3, Loop
        fsd f16, -8(x1)
```

CPI = 13 cycles / 23 instructions = .565

3. Show how the code from problem #1 will execute on a 2-issue Tomasulo-style architecture with no restriction on which pair of instructions can be issued in one cycle. Use a table like that on slide 74 of the chapter 3 power point notes. Show two complete iterations of the loop. Include the fetch stage.

Instruction		Fetch	Issue	Execute	Write
fld	f1, 0(x1)	1	2	3	4
fld	f2, 0(x2)	1	2	3	5
fadd.d	f3, f1, f2	2	3	6	10
fmult.d	f5, f3, f4	2	3	11	18
fsd	f5, 0(x1)	3	4	19	n/a
addiw	x1, x1, 8	3	4	5	6
addiw	x2, x2, 8	4	5	6	7
subiw	x3, x3, 1	4	5	6	8
bne	x3, x0, Loop	5	6	9	n/a
fld	f1, 0(x1)	10	11	12	13
fld	f2, 0(x2)	10	11	12	14
fadd.d	f3, f1, f2	11	12	15	19
fmult.d	f5, f3, f4	11	12	20	27
fsd	f5, 0(x1)	12	13	28	n/a
addiw	x1, x1, 8	12	13	14	15
addiw	x2, x2, 8	13	14	15	16
subiw	x3, x3, 1	13	14	15	17
bne	x3, x0, Loop	14	15	18	n/a

Recall that we can only have 1 write per cycle, so some of the writes get postponed like the both subiw instructions.

4. Unroll and schedule the following code on a VLIW which has two load/store units, two pipelined FP units and an integer unit. Assume the fmult.d takes 5 cycles to compute. Fill the branch delay slot. Compute the CPI.

```
Loop:   fld      f0, 0(x1)
        fmult.d  f2, f0, f1          // f1 is a scalar
        fsd      f0, 0(x1)
        addiw    x1, x1, 8
        bne      x1, x2, Loop
```

Answer:

Load/store1	Load/store2	FP1	FP2	Int
fld f0, 0(x1)	fld f3, 8(x1)			
fld f5, 16(x1)	fld f7, 24(x1)			
fld f9, 32(x1)	fld f11, 40(x1)	fmult.d f2, f0, f1	fmult.d f4, f3, f1	
fld f13, 48(x1)	fld f15, 56(x1)	fmult.d f6, f5, f1	fmult.d f8, f7, f1	
fld f17, 64(x1)	fld f19, 72(x1)	fmult.d f10, f9, f1	fmult.d f12, f11, f1	
fld f21, 80(x1)	fld f23, 88(x1)	fmult.d f14, f13, f1	fmult.d f16, f15, f1	
fld f25, 96(x1)	fld f27, 104(x)	fmult.d f18, f17, f1	fmult.d f20, f19, f1	
fld f29, 112(x1)	fld f31, 120(x)	fmult.d f22, f21, f1	fmult.d f24, f23, f1	
fsd f2, 0(x1)	fsd f4, 8(x1)	fmult.d f26, f25, f1	fmult.d f28, f27, f1	
fsd f6, 16(x1)	fsd f4, 24(x1)	fmult.d f30, f29, f1	fmult.d f0, f31, f1	
fsd f10, 32(x1)	fsd f12, 40(x1)			
fsd f14, 48(x1)	fsd f16, 56(x1)			

fsd f18, 64(x1)	fsd f20, 72(x1)	addiw x1, x1, 128
fsd f22, -48(x1)	fsd f24, -40(x1)	
fsd f26, -32(x1)	fsd f28, -24(x1)	bne x1, x2, Loop
fsd f30, -16(x1)	fsd f0, -8(x1)	

Note the repeated use of f0 in the last fmult.d. Assuming 32 FP registers, we run out but luckily f0 will have already been stored and so we can reuse it at this point of the code. There are 50 instructions executed in 16 cycles with no stalls giving a CPI of 16 / 50 = .32 (slightly better than 3 instructions completing every cycle!)

5. Repeat #4 on the code from problem #1. In this problem, remember that the fadd.d takes 2 cycles and the fmult.d takes 3 cycles to compute. Assume there are 64 FP registers available.

Answer:

Load/store1	Load/store2	FP1	FP2	Int
fld f1, 0(x1)	fld f2, 0(x2)			
fld f6, 8(x1)	fld f7, 8(x2)			
fld f10, 16(x1)	fld f11, 16(x2)	fadd.d f3, f1, f2		
fld f14, 24(x1)	fld f15, 24(x2)	fadd.d f8, f6, f7		
fld f18, 32(x1)	fld f19, 32(x2)	fadd.d f12, f10, f11		
fld f22, 40(x1)	fld f23, 40(x2)	fadd.d f16, f14, f15	fmult.d f5, f3, f4	
fld f26, 48(x1)	fld f27, 48(x2)	fadd.d f20, f18, f19	fmult.d f9, f8, f4	
fld f30, 56(x1)	fld f31, 56(x2)	fadd.d f24, f22, f23	fmult.d f13, f12, f4	
fld f34, 64(x1)	fld f35, 64(x2)	fadd.d f28, f26, f27	fmult.d f17, f16, f4	
fsd f5, 0(x1)		fadd.d f32, f30, f31	fmult.d f21, f20, f4	
fsd f9, 8(x1)		fadd.d f36, f34, f35	fmult.d f25, f24, f4	
fsd f13, 16(x1)			fmult.d f29, f28, f4	
fsd f17, 24(x1)			fmult.d f33, f32, f4	
fsd f21, 32(x1)			fmult.d f37, f36, f4	
fsd f25, 40(x1)				subiw x3, x3, 1
fsd f29, 48(x1)				addiw x1, x1, 72
fsd f33, -16(x1)				bne x3, x0, Loop
fsd f37, -8(x1)				addiw x2, x2, 72

CPI = 18 / 49 = .367.

6. Slide 80 of the chapter 3 power point notes provides some comparisons between various approaches described in this chapter. Answer the following questions.
   a. Notice that there are no dynamic issue and dynamic scheduling superscalars without speculation. Why do we need speculation to have a successful dynamic issue/scheduled superscalar?
   b. The VLIW approach seems like it would surpass a hardware-based superscalar because the compiler can take its time to unroll loops and schedule the code. Yet VLIW has only been implemented in very few actual processors. Why do you suppose dynamic issue/scheduling is more popular?
   c. If we had a choice of doubling the system clock rate on a single pipeline or utilizing a superscalar, why is the superscalar the more common and practical approach?

Answer:
   a. Consider the code from problem #4 above. A loop consists of 5 instructions, including a branch. If issued on a dual-issue superscalar, the branch is issued every 3 cycles. But in Tomasulo, as seen in problem #3, a branch requires fetching, issuing and executing, so the result of the branch is not known until the end of cycle 3. Thus, we fetch, issue and begin

executing 3 cycles worth of a loop and then have to wait 3 additional cycles to start the next iteration. If branch speculation is accurate, and the target location is known at the end of the fetch stage of Tomasulo, the code from problem #4 would be able to start the next iteration 1 cycle after the branch, as shown in #4. The only problem is that the branch would be issued alone (the other instruction fetched would be the next sequential instruction and not issued because the branch was taken).

b. Personally, I like the VLIW approach. I find it more attractive to let the compiler arrange code to achieve a greater degree of parallelism than to rely on dynamic issue where there is limited time to schedule code. According to the authors though, there is only so much ILP available in any block of code and so loop unrolling is the only way to defeat this limitation. But an unrolled loop executing on a superscalar does not provide the performance increase that we find when executing dynamic code on a vector processor (see the last paragraph of section 3.7) where loop unrolling is automatically handled within the vector processor code itself and therefore compiler-based loop unrolling becomes unnecessary.

c. Because of cache access time, doubling the clock rate ultimately doesn't help us out as much as we would like. As seen with the MIPS R4000 pipeline, which was superpipelined to allow a faster clock rate, cache access was spread across 2 or 3 cycles which lengthened most of the stalls. The superscalar gives us some parallelism depending on how often we can issue 2 instructions per cycle. As seen in the earlier problems here, we have the potential for issuing 2 instructions per cycle at least half of the time, maybe more often.

7. Let's compare branch speculation approaches. Assume our architecture computes branch target addresses in the 3rd stage and branch conditions in the 4th stage. We can support a branch prediction buffer with a hit rate of 98% of all branches and a speculation accuracy of 95%. We can support a branch target buffer with a hit rate of 90% of all branches and a speculation accuracy of 90%. We can support a branch folding buffer with a hit rate of 80% of all branches and a speculation accuracy of 90%. A cache miss or miss-speculation has a penalty of 3 cycles to reach the branch condition and 1 extra cycle to modify the buffer being used. If a benchmark consists of 20% conditional branches, of which 70% are taken, and 3% unconditional branches, what is the overall penalty (per instruction, not per branch) for dealing with branches for each approach? Compare the 3 types of buffer and assume not taken.

Answer: Starting with assume not taken, if a branch is not taken there is no penalty, if it is an unconditional branch (always taken), we have a 2 cycle penalty as branch targets are determined in stage 3, and if we have a taken conditional branch, we have a 3 cycle penalty (conditions are determined in stage 4). As we are not using a buffer at all, there is no additional penalty for updating a buffer, and there are no buffer cache misses or miss-speculations. This gives us a penalty of 20% * 70% * 3 + 3% * 2 = .20 penalty per instruction.

With the branch prediction buffer, we still have to wait until stage 3 to determine where we are branching. For unconditional branches, the penalty is always 2 (we never miss-speculated since unconditional branches are always taken and on a miss, we don't need to update the buffer since we know the branch is always taken). For conditional branches, if found in the buffer and speculated correctly, the penalty is 2 and if either a buffer miss or miss-prediction, the penalty is 4 (determine the branch in stage 4 (3 cycles) plus 1 cycle to correct the prediction buffer). This gives us a penalty of 3% * 2 + 20% * 98% * 95% * 2 + 20% * 2% * 4 + 20% * 98% * 5% * 4 = .49 penalty per instruction. Notice this is over twice as bad as assuming not taken.

With the branch target buffer, we are fetching both the prediction and the target location. A buffer hit and correct speculation has a penalty of 0. The penalties are the same as above (4 cycles). Our penalty then is 3% * 0 + 20% * 90% * 90% * 0 + 20% * 10% * 4 + 20% * 90% * 10% * 4 = .15.

With branch unfolding, a buffer hit and correct speculation gives us a penalty of -1 whereas the penalties are the same as above. This gives us 3% * -1 + 20% * 80% * 90% * -1 + 20% * 20% * 4 + 20% * 80% * 10% * 4 = .05.

8. A single processor is executing three threads, labeled 1, 2 and 3. Assume a switch occurs between threads as follows: thread 1 gets 3 cycles of time, thread 2 gets 6 cycles of time, thread 3 gets 4 cycles of time, repeatedly. Each processor can support up to 4 instructions per cycle. Show how the three threads execute on each of the following processor types. Show 12 cycles worth of execution.
   a. no support for multi-threading where a switch between threads takes 2 cycles
   b. course-grained SMT where switches take 1 cycle
   c. fine-grained SMT where switches take 0 cycles
   d. full SMT

   Answer (- indicates an empty slot):

```
a. 1 1 1 -     b.  1 1 1 -     c.  1 1 1 -     d.  1 1 1 2
   s s s s         s s s s         2 2 2 2         2 2 2 2
   s s s s         2 2 2 2         2 2 - -         2 3 3 3
   2 2 2 2         2 2 - -         3 3 3 3         3 1 1 1
   2 2 - -         s s s s         1 1 1 -         2 2 2 2
   s s s s         3 3 3 3         2 2 2 2         2 2 3 3
   s s s s         s s s s         2 2 - -         3 3 1 1
   3 3 3 3         1 1 1 -         3 3 3 3         1 2 2 2
   s s s s         s s s s         1 1 1 -         2 2 3 3
   s s s s         2 2 2 2         2 2 2 2         3 3 1 1
   1 1 1 -         2 2 - -         2 2 - -         1 2 2 2
   s s s s         s s s s         3 3 3 3         2 2 2 3
```

9. It was easy to determine a pipeline's performance by counting stalls. With our out-of-order, multi-issue superscalar, performance is harder to determine using a formulaic approach (which is one reason why we've begun to move away from the strictly quantitative approach in this chapter). Stalls are now sometimes "hidden". What factors then are involved in the performance of a multi-issue superscalar? That is, under what circumstances does performance degrade? Rank them from largest impact to least impact.

   Answer: Many of our previous source of stalls are now taking place at reservation stations. This allows the fetch and issue portions of the pipeline can continue operating, at least for a time. Performance degradation occurs from these sources:
   a. Instruction cache misses cause the IIFU to stop fetching instructions – this is the largest impact because the miss penalty is many cycles long
   b. ROB and instruction queue buffers fill up – these stalls stop the fetch/issue stages, delay is based on when the filled buffer can remove some of its content
   c. Branch miss-speculation requires refilling the hardware units and ROB
   d. Limitation on the number of functional units and reservation statements (registers) – if the functional units are pipelined, they will typically not cause any stalls, but waiting instructions in reservation stations might if we only have a few reservation stations for any functional unit (for instance, imagine and FP adder with just two reservation stations)
   e. The single CDB only allows one instruction to send a result in any cycle