Appendix C sample problems (sections C.4-C.6)

1. Assume an instruction mix of 15% conditional branches, 1% unconditional branches, 84% all others, and 60% of the conditional branches are taken. We have a 4-stage pipeline where branch target locations are computed in the 3rd stage and branch conditions in the 4th stage. Assume no other sources of pipeline stalls. Should we implement "assume taken" or "assume not taken" for this machine? For "assume not taken", even unconditional branches are assumed not taken.

   Answer:
   Assume taken branch penalty = .15 * .60 * 2 + .15 * .40 * 3 + .01 * 2 = .38
   That is, of the 15% conditional branches, if they are taken, we accrue a 2 cycle penalty because we know the branch location in the 3rd stage, and if we find that the branch is not taken (4th stage), we wind up with a 3 cycle penalty. For unconditional branches, we always have a 2 cycle penalty.

   Assume not taken = .15 * .60 * 3 + .15 * .40 * 0 + .01 * 3 = .30
   Here, if we assume not taken, then conditional branches that are not taken have no penalty but conditional branches taken and unconditional branches always have a 3 cycle penalty.

   So assume not taken is better for this machine, much like RISC-V, but unlike MIPS R4000.

2. Do problem C.7 on page C-75 – C-76. For both parts, determine which processor is faster and by how much.

   Answer: 5-stage pipeline has a 1 ns clock cycle and experiences a stall every 5 instructions. 12-stage pipeline has a clock cycle time of .6 ns and experiences three stalls in every 8 instructions. 20% of all instructions are branches with a miss-prediction rate of 5%.
   a. Taking into account only data hazards, the 5-stage pipeline's CPU execution time is IC * (1 + 1 / 5) * 1 ns = 1.2 ns * IC while the 12-stage pipeline's execution time is IC * (1 + 3 / 8) * .6 ns = .825 ns * IC, so the 12-stage pipeline is 1.2 / .825 = 1.455 times faster.
   b. A branch miss-prediction yields a 2-cycle penalty for the 5-stage pipeline and a 5-cycle penalty for the 12-stage pipeline. The CPIs are now 1 + 1 / 5 + .20 * .05 * 2 = 1.22 for the 5-stage pipeline and 1 + 3 / 8 + .20 * .05 * 5 = 1.425 for the 12-stage pipeline. CPU Execution Time for the 5-stage pipeline = IC * 1.22 * 1 ns = 1.22 ns * IC. CPU Execution Time for the 12-stage pipeline = IC * 1.425 * .6 ns = .855 ns * IC. So the 12-stage pipeline is still faster by 1.22 / .855 = 1.427 times faster.

3. Examine the RISC-V FP pipeline. A new source of structural hazards arises because of the variable length of the EX stages –two or more instructions could collide trying to enter the MEM stage in the same cycle.
   a. List the new sources of structural hazards by specifying for a pair of instructions, when they each exit the ID stage to cause a collision in the MEM stage.

b. One possible solution to this type of structure hazard is to let one instruction bypass the MEM stage and move directly to the WB stage. This works, for instance, when one instruction is a load-store and the other is an FP operation. However, this brings about another structural hazard. Why?

Answer:

a. There are no sources of structural hazard between a pair of integer operations as there is no variability in their length. The sources will arise either if we have one integer and one FP operation, as listed below, or two FP operations, as listed below.

   Integer-FP: integer is in ID while FP is in A4, M7 or in the 25th cycle of a DIV, these combinations mean that the FP operation precedes the int operation by 3 (FP add), 6 (FP multiply) or 24 (FP division).

   FP-FP: FP multiply occurs 3 cycles before an FP add, or an FP divide occurs 18 cycles before an FP multiply or 21 cycles before an FP add.

b. If an instruction bypasses MEM to avoid a structural hazard with an FP operation, it could collide with yet another instruction, this time in the WB stage. Consider the following instruction sequence.

add.d f1, f2, f3	IF	ID	A1	A2	A3	A4	MM	WB
instruction 1		IF	ID	EX	MM	WB		
add x1, x2, x3			IF	ID	EX	MM	WB	
add x4, x5, x6				IF	ID	EX	MM	WB

   If we allow the last add to bypass MEM to avoid a structural hazard with the add.d, then we have a structural hazard with the second to last add. So we have to avoid this type of situation.

4. Given the following RISC-V code, use the revised pipeline with the FP units to show the timing. Assume forwarding is available. NOTE: two instructions cannot be allowed to enter the MEM stage at the same even if one instruction does nothing during the MEM stage.

```
fld     f0, 0(x3)
fadd    f1, f0, f2
fmul    f3, f1, f4
addi    x3, x3, 8
fsd     f3, 0(x3)
```

Answer:

fld f0, 0(x3)	IF	ID	EX	MM	WB													
fadd f1, f0, f2		IF	ID	s	A1	A2	A3	A4	MM	WB								
fmul f3, f1, f4			IF	s	ID	s	s	s	M1	M2	M3	M4	M5	M6	M7	MM	WB	
addi x3, x3, 8					IF	s	s	s	ID	EX	MM	WB						
fsd f3, 0(x3)									IF	ID	EX	s	s	s	s	s	MM	WB

There is a stall between the load and add as usual. There are 3 cycles of stalls between the add and multiply because the multiply must wait for the add to produce its result and forward it to M1. There are 4 cycles of stalls between the multiply the store because of the latency of computing the multiply (7 total cycles) and a 5th stall added so that the store does not enter the MEM stage at the same time the multiply does. Notice that the multiply uses WB and not MEM while store uses MEM but not WB. We could redo the hardware so that the multiply skips the MEM stage and goes right into the WB stage so that the store is not stalled one extra cycle. But this is a problematic idea because the hardware not only has to detect this structural hazard but also detect that multiply doesn't use MEM and store doesn't use WB.

5. Repeat #4 on the following code assuming forwarding is available and that branches are handled in the ID stage. Assume that a store and an FP operation can enter both the MEM and WB stages simultaneously. Show the first full iteration plus the first instruction of the second iteration. Next, schedule the code to remove as many stalls as possible and show the revised timing diagram.

```
loop:  ld      f0, 0(x1)
       ld      f1, 0(x2)
       add.d   f2, f0, f1
       sd      f2, 0(x2)
       addi    x1, x1, 8
       addi    x2, x2, 8
       bne     x1, x3, loop
```

ld f0, 0(x1)	IF	ID	EX	MM	WB												
ld f1, 0(x2)		IF	ID	EX	MM	WB											
add.d f2, f0, f1			IF	ID	s	A1	A2	A3	A4	MM	WB						
sd f2, 0(x2)				IF	s	ID	EX	s	s	MM	WB						
addi x1, x1, 8						IF	ID	s	s	EX	MM	WB					
addi x2, x2, 8							IF	s	s	ID	EX	MM	WB				
bne x1, x3, loop										IF	s	ID	EX	MM	WB		
next instruction												IF	f				
ld f0, 0(x1)													IF	ID	EX	MM	WB

The scheduled code and timing diagram is shown below. Notice the sd reaches the MEM stage at the same time the add.d does. Although this structural hazard requires additional hardware, we make the assumption that it is ok because the add.d does nothing in its MEM stage while the sd does nothing in its WB stage. If this structural hazard is not permitted, we would have to rearrange the code with some stalls because the sd, being in the branch delay slot, cannot be stalled by 1 cycle. It would have to be moved before the bne which would involve stalling it because of not having enough distance from the add.d and we would not have anything in the branch delay slot!

ld f0, 0(x1)	IF	ID	EX	MM	WB							
ld f1, 0(x2)		IF	ID	EX	MM	WB						
addi x1, x1, 8			IF	ID	EX	MM	WB					
add.d f2, f0, f1				IF	ID	A1	A2	A3	A4	MM	WB	
addi x2, x2, 8					IF	ID	EX	MM	WB			
bne x1, x3, loop						IF	ID	EX	MM	WB		
sd f2, 0(x2)							IF	ID	EX	MM	WB	
ld f0, 0(x1)								IF	ID	EX	MM	WB

6. Why are precise exceptions harder to maintain in the FP pipeline than the 5-stage pipeline? In the 5-stage pipeline, all instructions exit the pipeline in the order they were fetched. No exceptions arise in the WB stage so we can handle exceptions in that stage. For any instruction, a vector of information is passed down the pipeline with it: its PC value, if it raised an exception, and what type of exception arose and in which stage. The WB stage can look at an instruction and see if it raised an exception. If so, it saves that instruction's PC value, the pipeline is flushed, the exception is handled, and the pipeline restarts with the saved PC value. Therefore, if a later instruction raised an exception earlier in the pipeline (see below), the exceptions are still handled in order – in fact, it is possible that by handling the earlier instruction's exception that the later exception will not re-arise when the pipeline restarts. Below, the latter instruction (addiw) causes an exception earlier in time than the earlier instruction.

lw	IF	ID	EX	MEM*	WB	
addiw		IF*	ID	EX	MEM	WB

* indicates where the exception arose. Assume the lw causes a memory violation whereas the addiw is a page fault.

When we move to the FP pipeline, an instruction may have left the pipeline by the time an earlier instruction which raised an exception reaches the WB stage. Come up with an example demonstrating how exception handling with out of order completion will not cause a problem, and an example where it does cause a problem.

Answer: The following code is equivalent to the C code *x = y * z; where *x is a pointer whose address is stored in 0(x2) and y and z are already loaded into F2 and F0 respectively.

```
fmult.d      f4, f2, f0
lw           x1, 0(x2)
fsd          f4, 0(x1)
```

First, load the pointer from memory (lw) before storing the result of the multiplication (fsd). The lw exits the pipeline before the fmult.d completes. What if the fmult.d raises an overflow exception in M7? lw will have already left the pipeline. Now the lw is not impacted by the result of the fmult.d, so handling the exception of the fmult.d after lw exits the pipeline is not a problem (because we did not alter memory, and we will restart the pipeline with the mul, so we would repeat the lw). But what if the lw was a store instruction instead? Now we are impacting memory. Consider this sequence:

```
fmult.d     f4, f2, f0
fsd         f10, 0(x2)
fsd         f4, 0(x1)
```

We wind up altering memory by storing f10 to 0(x2) in spite of the exception raised by the fmult.d. We will examine some solutions to this problem later in the semester.

7. An examination of figure C.34 (page C-56) shows that by far, the greatest reason for stalls for the FP SPEC benchmarks is because of FP divides. Would this argue then that we should pipeline the FP divider like we have pipelined the FP adder and multiplier?

   Answer: No, if you look at the figure, there are two sources of divide stalls, the divide itself and the structural hazard for not having a pipelined divider. The divide itself causes RAW hazards, for instance between divide and store or between divide and using the result in another FP operation. The lengthy latency is the source of most of the stalls. If you look at the structural hazard stalls because of the lack of a pipelined divide unit, you see values ranging from 0.0 to 2.0 indicating that most of these benchmarks do not have divide operations within a distance that would cause a stall because the divider is not pipelined. Although divides are rare and the structural hazard penalty is small, it actually might be worth implementing a pipelined divider because the hardware to pipeline a functional unit is small and for reasons we cover in chapter 3.

8. Given the following code, show how it executes on the MIPS R4000 pipeline assuming forwarding is available from EX to EX and from DS to EX, and a 3-cycle branch penalty. Branches are computed in EX so there is no RAW hazard stall needed between the second addiw and the bne.

```
loop:   lw      x2, 0(x1)
        lw      x3, 0(x2)
        addiw   x3, x3, 1
        sw      x3, 0(x2)
        addiw   x1, x1, 4
        bne     x1, x5, loop
```

   Answer:

lw x2, 0(x1)	IF	IS	RF	EX	DF	DS	TC	WB													
lw x3, 0(x2)		IF	IS	RF	s	s	EX	DF	DS	TC	WB										
addiw x3, x3, 1			IF	IS	s	s	RF	s	s	EX	DF	DS	TC	WB							
sw x3, 0(x2)				IF	s	s	ID	s	s	RF	EX	DF	DS	TC	WB						
addiw x1, x1, 4							IF	s	s	ID	RF	EX	DF	DS	TC	WB					

bne x1, x5, loop										IF	ID	RF	EX	DF	DS	TC	WB				
next instr 1											IF	ID	RF	fl							
next instr 2												IF	ID	fl							
next instr 3													IF	fl							
lw x2, 0(x1)														IF	IS	RF	EX	DF	DS	TC	WB

Two stalls are needed between the first and second lw's because the first fetch retrieves a pointer to the datum, two stalls are needed between the second lw and the addiw but no stalls are needed between the addiw and the sw of the computed value. The branch penalty amounts to 3 cycles of wasted instruction fetches.

9. Let's compare the 5-stage RISC-V pipeline (the revised version with forwarding and branches computed in the ID stage) to the MIPS R4000 pipeline. First, being a superpipelined processor, the MIPS R4000 clock speed is 60% faster than the RISC-V clock speed. The ideal CPI of both is 1 and the IC will be the same for any benchmark. The only other difference is in the number of stalls. We will make the following assumptions:
   - Instruction mix of 33% load, 38% ALU, 10% store, 15% conditional branch, 4% unconditional branch
   - 40% of all ALU operations use a value loaded in the immediately preceding instruction, and no loads use indirect memory reference
   - 75% of all conditional branches are taken and no conditional branches have a RAW hazard with a preceding instruction
   - An optimizing compiler can schedule 85% of the load → ALU hazards away for RISC-V and 85% of the load → ALU 1st cycle penalties for MIPS R4000 but does not attempt to remove any other data hazard stalls
   - An optimizing compiler can fill the branch delay slot for RISC-V code 80% of the time but the optimizing compiler for MIPS R4000 assumes taken in 100% of the cases and does not attempt to schedule a branch delay slow

   If the code is optimized, which processor is faster?

   Answer: Compute CPU Execution Times for both processors = IC * CPI * Clock Cycle Time. IC remains the same, CPI = 1 + stalls. Clock Cycle Time for MIPS R4000 is 1.6 that of RISC-V. Use 1.6 for RISC-V's CCT and 1 for MIPS R4000's CCT. Now, compute stalls.

   For RISC-V: 1 cycle penalty for every RAW hazard that cannot be scheduled away and for any unfilled branch delay slot, 38% * 40% * 15% = .023. The branch delay slots unfilled = (15% + 4%) * 20% = .038. Therefore, the CPI for RISC-V = 1 + .023 + .038 = 1.061.

   For MIPS R4000, RAW hazards amount to 2 cycle stalls (if not scheduled) or a 1 cycle stall (if scheduled), 38% * 40% * 15% * 2 + 38% * 40% * 85% * 1 = .175. Branch penalty is 3 cycles if branch is taken, 2 cycles if branch is not taken. 75% of all conditional branches are taken (100% of all unconditional branches are taken), or 15% * 75% * 3 + 15% * 25% * 2 + 4% * 3 = .533. MIPS R4000 CPI = 1 + .175 + .533 = 1.708.

   CPU Execution Time RISC-V = IC * 1.061 * 1.6 = 1.698 * IC
   CPU Execution Time MIPS R4000 = IC * 1.708 * 1 = 1.708 * IC

RISC-V is slightly faster (1.708 / 1.698 = 1.006 or about .6% faster). MIPS R4000 would be faster if the compiler was allowed to schedule an instruction into the branch delay slot (see figure C.40 on page C-60).