CSC 462/562 Computer Architecture
Homework #5 answer key

1. NOTE: the bne uses x4, we cannot adjust x4
   a. Loop: fld f1, 0(x1)
   fld f2, 0(x2)
   fld f6, 8(x1)
   fld f7, 8(x2) fadd.d f3, f1, f2
   fld f4, 0(x3)
   fld f9, 8(x3) fadd.d f8, f6, f7
   addiw x1, x1, 16 fsub.d f5, f3, f4
   addiw x2, x2, 16
   addiw x3, x3, 16 fsub.d f10, f8, f9
   fsd f5, 0(x4)
   addiw x4, x4, 16
   bne x4, x5, Loop
   fsd f10, -16(x4)
   CPI = 13 / 17 = .765
   b. Loop: fld f1, 0(x1)
   fld f2, 0(x2)
   fld f6, 8(x1)
   fld f7, 8(x2) fadd.d f3, f1, f2
   fld f4, 0(x3)
   fld f9, 8(x3) fadd.d f8, f6, f7
   fld f11, 16(x1) fsub.d f5, f3,f 4
   fld f12, 16(x2)
   fld f13, 16(x3) fsub.d f10, f8, f9
   fsd f5, 0(x4) fadd.d f14, f11, f12
   addiw x1, x1, 24
   fsd f10, 8(x4) addiw x4, x4, 24
   fsub.d f15, f14, f13
   addiw x3, x3, 24
   bne x4, x5, Loop
   fsd f15, -16(x4) addiw x2, x2, 24
   CPI = 16 / 23 = .696 (notice this is not as good as part a)

c. Loop:

fld f1, 0(x1)	fld f2, 0(x2)
fld f4, 0(x3)	fld f6, 8(x1)
fld f7, 8(x2)	fadd.d f3, f1, f2
fld f9, 8(x3)	fld f11, 16(x1)
fld f12, 16(x2)	fadd.d f8, f6, f7
fld f14, 16(x3)	fsub.d f5, f3, f4
addiw x1, x1, 24	fadd.d f13, f11, f12
fadd.d f14, f11, f12	fsub.d f10, f8, f9
fsd f5, 0(x4)	addiw x2, x2, 24
addiw x3, x3, 24	fsub.d f15, f13, f14
fsd f10, 16(x4)	addiw x4, x4, 24
bne x4, x5, Loop	
fsd f15, -8(x4)	

CPI = 13 / 24 = .542

2. a.

lw	x1, 0(x2)	IF	ID	EX	MM	WB																		
lw	x3, 0(x4)	IF	ID	EX	MM	WB																		
addw	x5, x1, x3		IF	s	ID	EX	MM	WB																
subiw	x5, x5, 1		IF	s	ID	s	EX	MM	WB															
sw	x5, 0(x6)				IF	s	ID	EX	MM	WB														
addiw	x2, x2, 4				IF	s	ID	EX	MM	WB														
addiw	x4, x4, 4						IF	ID	EX	MM	WB													
addiw	x6, x6, 4						IF	ID	EX	MM	WB													
subiw	x7, x7, 1							IF	ID	EX	MM	WB												
bne	x7, x0, Loop							IF	s	s	ID	EX	MM	WB										
lw	x1, 0(x2)										IF/fl	IF	ID	EX	MM	WB								
lw	x3, 0(x4)										IF/fl	IF	ID	EX	MM	WB								

a.

Instruction	Fetch	Issue	Execute	Write	Commit
lw x1, 0(x2)	1	2	3	4	5
lw x3, 0(x4)	1	2	3	5	6
addw x5, x1, x3	2	3	6	7	8
subiw x5, x5, 1	2	3	8	9	10

sw x5, 0(x6)	3	4	10		11
addiw x2, x2, 4	3	4	8	10	12
addiw x4, x4, 4	4	5	10	11	13
addiw x6, x6, 4	4	5	11	12	14
subiw x7, x7, 1	5	6	12	13	15
bne x7, x0, Loop	5	6	14		16

The pipeline completes 1 iteration in 13 cycles although the next iteration is able to start 2 cycles earlier, so 11 cycles per loop iteration. In Tomasulo, having only 2 int units prevents some of the addiw's from executing. The bne computes the new PC in cycle 14, so 14 cycles before the next loop iteration can begin, although in fact the bne commits at cycle 16.

3. Notice both pairs of fld are using the same source register, x1 and x2, so we have to sequentialize those in the Read Op stages. Also remember that writes are sequentialized.

a.

Instruction		Fetch	Issue	Read Op	Execute	Write
fld	f1, 0(x1)	1	2	3	4	5
fld	f2, 8(x1)	1	2	4	5	6
fmul.d	f3, f1, f2	2	3	7	8	11
fld	f4, 0(x2)	2	6*	7**	8	9
fld	f5, 8(x2)	3	7	8	9	10
fadd.d	f6, f4, f5	3	8	11	12	14
fsub.d	f7, f3, f6	4	8	15	16	18
fsd	f7, 0(x1)	4	10***	19	20	n/a
addiw	x1, x1, 16	5	6	8****	9	12
addiw	x2, x2, 16	5	6	9*****	10	13

* - both load/store units are busy until cycle 6, ** - fld can read x2 while fmul.d is reading f1 and f2
**** - two reads per cycle, has to wait until cycle 8, ***** - two reads per cycle

b.

Instruction		Fetch	Issue	Execute	Write
fld	f1, 0(x1)	1	2	3	4
fld	f2, 8(x1)	1	2	3	5
fmul.d	f3, f1, f2	2	3	6	9
fld	f4, 0(x2)	2	3	5*	6
fld	f5, 8(x2)	3	4	6	7
fadd.d	f6, f4, f5	3	4	8	10

fsub.d	f7, f3, f6	4	5	11	13
fsd	f7, 0(x1)	4	5	14	n/a
addiw	x1, x1, 16	5	6	7	8
addiw	x2, x2, 16	5	6	9	11**

* - cannot execute until first fld is done executing, ** - cannot write at 10 because fadd.d is writing

4.

a.

Load/store1	Load/store2	FP1	integer/branch
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	
fld f10, 24(x1)	fld f11, 24(x2)	fmult.d f6, f4, f5	
fld f13, 32(x1)	fld f14, 32(x2)	fmult.d f9, f7, f8	
fld f16, 40(x1)	fld f17, 40(x2)	fmult.d f12, f10, f11	
fsd f3, 0(x3)		fmult.d f15, f13, f14	
fsd f6, 8(x3)		fmult.d f18, f16, f17	
fsd f9, 16(x3)			addiw x3, x3, 48
fsd f12, -24(x3)			addiw x1, x1, 48
fsd f15, -16(x3)			bne x3, x4, Loop
fsd f18, -8(x3)			addiw x2, x2, 48

NOTE: we do not need the second FP unit at all for this code

CPI = 12 / 28 = .429 (ideal CPI for this VLIW architecture is .200)

5. In a typical block of code, branches occur every 6-8 instructions. In a multi-issue superscalar, this means a branch may be reached within 2-3 cycles of time. Looking at the Tomasulo architecture, it would take at least 3 cycles before the branch decision is known (1 cycle to fetch, 1 cycle to issue, 1 cycle to execute). Without speculation, every branch would postpone the next instruction fetch so after every 2-3 cycles of running non-branch instructions, there would be a 2-3 cycle stall, on average. Assume taken: wait until the branch target location is computed and immediately branch there. There is no penalty if the branch is taken (most branches are) but if the branch is not taken, the instructions erroneously fetched must be flushed. In a Tomasulo-style architecture, we do this by flushing the reorder buffer, and in a pipeline, we do this by flushing the earlier portions of the pipeline. Assume not taken: always fetch the next sequential instructions and continue executing until the branch's condition is determined and if taken, flush as described above. Branch prediction buffer: fetch a prediction of whether to branch or not. If the prediction is to branch, branch as soon as the target location is known. If the prediction is to not branch, continue. Once the condition is evaluated, if incorrect, flush. Note that if incorrect (or a buffer miss), the buffer must be updated which adds another cycle of penalty. Branch target buffer: same as the prediction buffer except that the prediction also comes with the last

branch location used so, if predicted as a branch, branch immediately to that location. This reduces the delay when branches are predicted (no longer have to wait until the branch location is computed) while having the same penalty has a branch prediction buffer. The only downside to this versus the prediction buffer is that the address takes up a lot more space than the prediction bit or bits and so you either need a much larger buffer or can store fewer predictions resulting in more misses. Branch unfolding: similar to the branch target buffer but also included in the buffer is the predicted next instruction. In this way, if the branch is predicted as taken, not only branch to the target location (actually the target location + 1 instruction) but also place the predicted next instruction right into the IR. This has the same penalty as the target and prediction buffers but on an accurate prediction, the "penalty" is -1 cycles instead of 0. The drawback here is that this buffer must be even larger as it must store the prediction bit(s), the target address and an instruction. For a 2-issue superscalar, I would use the branch target buffer as it has the best combination of benefits with small penalties. For VLIW, since that is compiler based, I would rely on the compiler to schedule branch delay slots if possible and utilize assume not taken.

6.

a. Branch predictions are retrieved and applied in F1 (branch prediction and target fetched), both a hybrid branch predictor and indirect predictor buffer are accessed in F3 if the branch target buffer access in F1 was a miss, the hybrid branch predictor result is applied in F3 and the indirect predictor result is applied in F4, also a return stack is accessed and used in F4 if the current instruction is a return; branch conditions and locations are determined in ALU pipe 0 in stages Ex1 and Ex2 (they are known by the end of Ex2)
b. ALU operations and loads are handled in the Integer execute and load-store unit and made available for later instructions by the end of the Ex2 stage and are needed entering the Ex1 stage of the next instruction, so this would incur a 1 cycle stall unless the two instructions are issued at the same time, then it is a 2 cycle stall. FP results are handled in the Floating Point Execute unit and made available at the end of the F5 stage (the last pipeline stage) and are needed heading into the F2 stage by the next instruction resulting in a 3 cycle penalty.
c. 2 (dual issue) maximum but if two sequential instructions in the instruction stream have a structural hazard (use the same functional unit), only the first instruction is issued in that cycle and the other is forced to wait
d. TLB/cache misses cause a delay in filling the instruction queue which, if empty, cause a lengthy delay (whatever the L1 cache miss penalty is). Branches, as noted in part a, are determined in the Ex2 stage, which is stage 8 of the pipeline, so the branch penalty for miss-predictions is 7 cycles but keep in mind that, as this is a dual issue superscalar, that could be as many as 14 incorrectly fetched instructions. Data hazards result in a 1 or 2 cycle penalty (or as many as 3 for FP). The book mentions the structural hazard as noted in c, but as at least one instruction is issued, this is not really a source of stall. I ranked them in this order because the average amount of "wasted work" (as per figure 3.36) because of branch mispredictions is around 10% (which I assume means 10% of 1 cycle, or an impact to CPI of about .1) while the average number of stalls because of cache misses (as per figure 3.37) ranges from about .1 cycle to as many as 8. Data hazards have a 1-3 cycle stall (see the answer to part b) but most of these can be hidden by the issue stage.