Last updated:
03/11/2016

Problem M2.0: Complex Pipelining Dependencies

```
I1:  L.D     F1, 0 (R1)         ;    F1 = *r1;
I2:  MUL.D   F2, F0, F2         ;    F2 = F0*F2;
I3:  ADD.D   F1, F2, F2         ;    F1 = F2 + F2;
I4:  L.D     F2, 0 (R2)         ;    F2 = *r2;
I5:  ADD.D   F3, F1, F2         ;    F3 = F1 + F2;
I6:  S.D     F3, 0 (R3)         ;    *r3 = F3;
……
```

		Earlier (Older) Instruction					
		I1	I2	I3	I4	I5	I6
Current Instruction	I1	-					
	I2	-	-				
	I3	WAW	RAW	-			
	I4	-	WAW/WAR	WAR	-		
	I5	-	-	RAW	RAW	-	
	I6	-	-	-	-	RAW	-

Problem M2.1: Out-of-order Scheduling [? Hours]

```
          loop:
      I1        L.D     F2, 0(R1)    ;load X(i)
      I2        MUL.D   F1, F2, F0   ;multiply a*X(i)
      I3        L.D     F3, 0(R2)    ;load Y(i)
      I4        ADD.D   F3, F1, F3   ;add a*X(i)+Y(i)
      I5        S.D     F3, 0(R2)    ;store Y(i)
      I6        DADDUI  R1, R1, 8    ;increment X index
      I7        DADDUI  R2, R2, 8    ;increment Y index
      I8        DSGTUI  R3, R1, 800  ;test if done
      I9        BEQZ    R3, loop     ;loop if not done
```

Problem M2.1.A — In-order using a scoreboard

Each loop takes 28 cycles. The bottleneck is the long latency of the FP functional units.

Instr. Issued	Time (cycles)	Functional Unit Status					Registers Reserved for Writes
		Int	Load (1)	Adder (4)	Multiplier (15)	WB	
I_1	0		F2				F2
	1					F2	F2
I_2	2				F1		F1
I_3	3		F3		F1		F1,F3
	4				F1	F3	F1,F3
	...						
	16				F1		F1
	17					F1	F1
I_4	18			F3			F3
	...						
	21			F3			F3
	22					F3	F3
I_5	23						
I_6	24	R1					
I_7	25	R2					
I_8	26	R3					
I_9	27						

Table M2.1-1

Problem M2.1.B Out-of-order

The arrows show hazards that slow down the loop.Again, 28 cycles are required for each iteration. Out-of-order issue doesn't give any wins as we still must wait for the RAW hazard between I1/I2, I2/I4 and I4/I5, the WAW hazard between I3/I4, as well as the WAR hazard between I5/I7.

	Time			Op	Dest	Src1	Src2
	Decode →Issue	Issued	WB				
I_1	-1	0	1	`L.D`	F2	R1	
I_2	0	2	17	`MUL.D`	F1	F2	F0
I_3	1	3	4	`L.D`	F3	R2	
I_4	5	18	22	`ADD.D`	F3	F1	F3
I_5	6	23		`S.D`		R2	F3
I_6	7	8		`DADDUI`	R1	R1	
I_7	24	25		`DADDUI`	R2	R2	
I_8	25	26		`DSGTUI`	R3	R1	
I_9	26	27		`BEQZ`		R3	

Table M2.1-2

Problem M2.1.C **Register Renaming**

Thanks to register re-renaming, we can eliminate the WAW hazard between I3/I4 and the WAR hazard between I5/I7, and we can decode an instruction every cycle. Thus, instructions I7, I8, and I9 can be issued without stalling on I5 and we can issue a loop every 9 cycles (and complete the previous iteration of the loop every nine cycles). A fully pipelined multiplier is necessary to allow a new multiply instruction to be issued every 9 cycles.

In reality, it turns out that the single-issue and single-writeback restrictions introduce structural conflicts that don't allow the loop to settle in a 9-cycle period. A rough simulation suggests that a loop completes in a 10, 9, 8, 9, ... cycle pattern.

	Time			Op	Dest	Src1	Src2
	Decode →Issue	Issued	WB				
I_1	-1	0	1	L.D	T0	R1	
I_2	0	2	17	MUL.D	T1	T0	F0
I_3	1	3	4	L.D	T2	R2	
I_4	2	18	22	ADD.D	T3	T1	T2
I_5	3	23		S.D		T3	R2
I_6	4	5		DADDUI	T4	R1	
I_7	5	6		DADDUI	T5	R2	
I_8	6	7		DSGTUI	T6	T4	
I_9	7	8		BEQZ		T6	
I_1	8	9	10	L.D	T7	T4	
I_2	9	11	26	MUL.D	T8	T7	F0
I_3	10	12	13	L.D	T9	T5	
I_4	11	27	31	ADD.D	T10	T8	T9
I_5	12	32		S.D		T10	T5
I_6	13	14		DADDUI	T11	T4	
I_7	14	15		DADDUI	T12	T5	
I_8	15	16		DSGTUI	T13	T11	
I_9	16	17		BEQZ		T13	

Table M2.1-3

Problem M2.2: Out-of-Order Scheduling [? Hours]

Problem M2.2.A

This question is similar to Problem M3.1.C with shorter latency for the FPU.

	Time				OP	Dest	Src1	Src2
	Decode → ROB	Issued	WB	Committed				
I_1	-1	0	1	2	`L.D`	T0	R2	-
I_2	0	2	12	13	`MUL.D`	T1	T0	F0
I_3	1	**13**	**15**	**16**	`ADD.D`	**T2**	**T1**	**F0**
I_4	**2**	**3**	**4**	**17**	`ADDI`	**T3**	**R2**	**-**
I_5	**3**	**4**	**5**	**18**	`L.D`	**T4**	**T3**	**-**
I_6	**4**	**6**	**16**	**19**	`MUL.D`	**T5**	**T4**	**T4**
I_7	**5**	**17**	**19**	**20**	`ADD.D`	**T6**	**T5**	**T2**

Table M2.2-1

Problem M2.2.B

(This is NOT a unified register file design. The register names (T0, T1, …etc) in the renaming table refer to the ROB tags. Since we have a two-entry ROB, we should only use T0 and T1 for the renaming.)

	Time				OP	Dest	Src1	Src2
	Decode → ROB	Issued	WB	Committed				
I_1	-1	0	1	2	`L.D`	T0	R2	-
I_2	0	2	12	13	`MUL.D`	T1	T0	F0
I_3	3	**13**	**15**	**16**	`ADD.D`	**T0**	**T1**	**F0**
I_4	**14**	**15**	**16**	**17**	`ADDI`	**T1**	**R2**	**-**
I_5	**17**	**18**	**19**	**20**	`L.D`	**T0**	**T1**	**-**
I_6	**18**	**20**	**30**	**31**	`MUL.D`	**T1**	**T0**	**T0**
I_7	**21**	**31**	**33**	**34**	`ADD.D`	**T0**	**T1**	**F3**

Table M2.2-2

Problem M2.3: Superscalar Processor [? Hours]

Problem M2.3.A

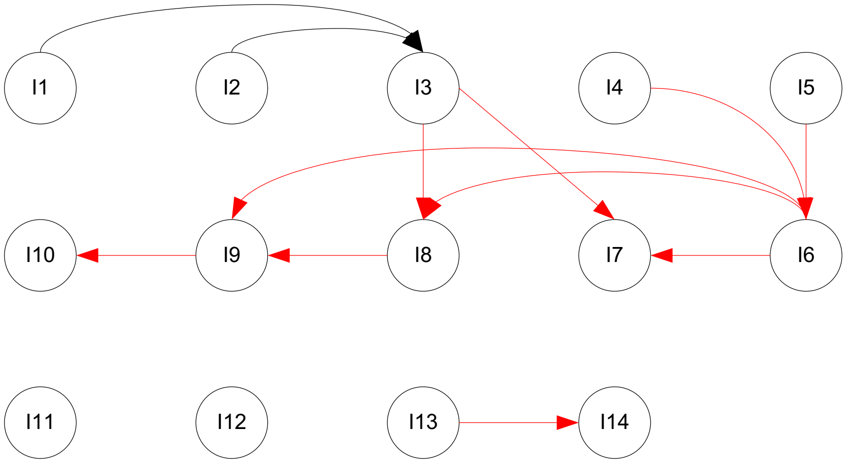
Fill in the renaming tags in the following two tables for the execution of instructions I1 to I10

Instr #	**Instruction**	**Dest**	**Src1**	**Src2**
I1	LD F2, 0(R2)	T1	R2	0
I2	LD F3, 0(R3)	T2	R3	0
I3	FMUL F4, F2, F3	T3	T1	T2
I4	LD F2, 4(R2)	T4	R2	4
I5	LD F3, 4(R3)	T5	R3	4
I6	FMUL F5, F2, F3	T6	T4	T5
I7	FMUL F6, F4, F5	T7	T3	T6
I8	FADD F4, F4, F5	T8	T3	T6
I9	FMUL F6, F4, F5	T9	T8	T6
I10	FADD F1, F1, F6	T10	F1	T9

Renaming table

	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10
R2										
R3										
F1										T10
F2	T1			T4						
F3		T2			T5					
F4			T3					T8		
F5						T6				
F6							T7		T9	

Problem M2.3.B



Problem M2.3.C

See the following table.

Slot	Instruction	Cycle instruction entered ROB	Argument 1		Argument 2		dst	*Cycle dispatched*	*Cycle written back to ROB*
			src1	cycle available	Src2	cycle available	dst reg		
T1	**LD F2, 0(R2)**	1	C	1	R2	1	F2	2	6
T2	**LD F3, 0(R3)**	1	C	1	R3	1	F3	3	7
T3	**FMUL F4, F2, F3**	2	F2	6	F3	7	F4	8	12
T4	**LD F2, 4(R2)**	2	C	2	R2	2	F2	4	8
T5	**LD F3, 4(R3)**	3	C	3	R3	3	F3	5	9
T6	**FMUL F5, F2, F3**	3	F2	8	F3	9	F5	10	14
T7	**FMUL F6, F4, F5**	4	F4	12	F5	14	F6	15	19
T8	**FADD F4, F4, F5**	4	F4	12	F5	14	F4	15	18
T9	**FMUL F6, F4, F5**	5	F4	18	F5	14	F6	19	23
T10	**FADD F1, F1, F6**	5	F1	5	F6	23	F1	24	27
T11	**ADD R2, R2, 8**	6	R2	6	C	6	R2	7	9
T12	**ADD R3, R3, 8**	6	R3	6	C	6	R3	8	10
T13	**ADD R4, R4, -1**	7	R4	7	C	7	R4	9	11
T14	**BNEZ R4, loop**	7	R4	11	C	Loop			
T15	**LD F2, 0(R2)**	8	C	8	R2	9	F2	10	14
T16	**LD F3, 0(R3)**	8	C	8	R3	10	F3	11	15
T17	**FMUL F4, F2, F3**	9	F2	14	F3	15	F4	16	20
T18	**LD F2, 4(R2)**	9	C	9	R2	9	F2	12	16
T19	**LD F3, 4(R3)**	10	C	10	R3	10	F3	13	17
T20	**FMUL F5, F2, F3**	10	F2	16	F3	17	F5	18	22
T21	**FMUL F6, F4, F5**	11	F4	20	F5	22	F6	23	27
T22	**FADD F4, F4, F5**	11	F4	20	F5	22	F4	23	26
T23	**FMUL F6, F4, F5**	12	F4	26	F5	22	F6	27	31
T24	**FADD F1, F1, F6**	12	F1	27	F6	31	F1	32	35
T25	**ADD R2, R2, 8**	13	R2	13	C	13	R2	14	16
T26	**ADD R3, R3, 8**	13	R3	13	C	13	R3	15	17
T27	**ADD R4, R4, -1**	14	R4	14	C	14	R4	16	18
T28	**BNEZ R4, loop**	14			C	Loop			
T29									

Problem M2.3.D

I5, I6, I7, I8, I9, I10 (see registers in blue in previous table)

27 cycles.

Problem M2.3.E

The behavior should repeat - should be obvious from the dependency graph (DAG) in Problem M2.3.D.

Problem M2.3.F

Yes

An extra FP multiplier does not really help, because All FMUL instructions execute as soon as operands are ready. But an extra memory port helps, because dispatch of I4, I5 was delayed waiting for memory port.

Problem M2.3.G

The answer is 4 cycles.

Since the integer index/counter additions are relatively short, they can proceed to generate values for different loop iterations and load all values from memory saving them to renamed registers. After a large number of iterations, many iterations of the loop will be running in parallel. Hence, the number of cycles is the latency of FMUL (3 + 1 cycle for write-back). In steady state, one iteration can complete every 4 cycles.

Problem M2.4: Register Renaming and Static vs. Dynamic Scheduling

Problem M2.4.A Simple Pipeline

The following table shows the cycles in which instructions are decoded, issued, and written back. It starts with cycle 0 in which the first load has been decoded (and thus has just entered the issue stage). It is assumed that all instructions prior to the first load have already been completed. Although not shown below, there is a buffer that holds instructions that are waiting in the issue stage. Since there is no bypassing, an instruction must complete the write-back stage before a dependent instruction can issue. For example, as shown in the table, the second load is issued in cycle 2, executes for 2 cycles, and is written back in cycle 4. Thus, any instruction that depends on the load can issue no earlier than cycle 5.

	Decoded Instruction (Enters Issue)	**Issued Instruction (Enters Execute)**	**WB Cycle For Issued Instruction**
0	`L.S F0, 0(R1)`	Stall	
1	`L.S F1, 0(R2)`	`L.S F0, 0(R1)`	3
2	`MUL.S F0, F0, F1`	`L.S F1, 0(R2)`	4
3	`L.S F2, 0(R3)`	Stall	
4	`L.S F3, 0(R4)`	Stall	
5	`MUL.S F2, F2, F3`	`MUL.S F0, F0, F1`	9
6	`ADD.S F0, F0, F2`	`L.S F2, 0(R3)`	8
7	`S.S F0, 0(R5)`	`L.S F3, 0(R4)`	9
8		Stall	
9		Stall	
10		`MUL.S F2, F2, F3`	14
11		Stall	
12		Stall	
13		Stall	
14		Stall	
15		`ADD.S F0, F0, F2`	17
16		Stall	
17		Stall	
18		`S.S F0, 0(R5)`	

The number of cycles from the issue of the first load instruction until the issue of the final store instruction is 18 cycles, inclusive.

Problem M2.4.B **Static Scheduling**

The new code sequence is given below. Originally there were two stall cycles after the second load instruction. Now these cycles will be filled by the third and fourth load instructions. The remaining instructions cannot be reordered due to data dependencies (except for the two multiply instructions, although doing that would hurt performance).

```
L.S       F0, 0(R1)
L.S       F1, 0(R2)
L.S       F2, 0(R3)
L.S       F3, 0(R4)
MUL.S     F0, F0, F1
MUL.S     F2, F2, F3
ADD.S     F0, F0, F2
S.S       F0, 0(R5)
```

The following table shows the cycles in which the instructions in the above sequence are decoded, issued, and written back.

	Decoded Instruction (Enters Issue)	**Issued Instruction (Enters Execute)**	**WB Cycle For Issued Instruction**
0	`L.S F0, 0(R1)`	Stall	
1	`L.S F1, 0(R2)`	`L.S F0, 0(R1)`	3
2	`L.S F2, 0(R3`	`L.S F1, 0(R2)`	4
3	`L.S F3, 0(R4)`	`L.S F2, 0(R3)`	5
4	`MUL.S F0, F0, F1`	`L.S F3, 0(R4)`	6
5	`MUL.S F2, F2, F3`	`MUL.S F0, F0, F1`	9
6	`ADD.S F0, F0, F2`	Stall	
7	`S.S F0, 0(R5)`	`MUL.S F2, F2, F3`	11
8		Stall	
9		Stall	
10		Stall	
11		Stall	
12		`ADD.S F0, F0, F2`	14
13		Stall	
14		Stall	
15		`S.S F0, 0(R5)`	

The number of cycles from the issue of the first load instruction to the issue of the final store instruction is 15 cycles, inclusive. Static scheduling has enabled us to reduce the execution time of the sequence by 17%.

Problem M2.4.C Fewer Registers

The new code sequence using only two floating-point registers is shown below. It is assumed that R6 holds the address of a memory location that can be used to store temporary values.

```
L.S       F0, 0(R1)
L.S       F1, 0(R2)
MUL.S     F0, F0, F1
L.S       F1, 0(R3)
S.S       F0, 0(R6)
L.S       F0, 0(R4)
MUL.S     F0, F0, F1
L.S       F1, 0(R6)
ADD.S     F0, F0, F1
S.S       F0, 0(R5)
```

The following table shows the cycles in which the instructions in the above sequence are decoded, issued, and written back. For this problem, a store instruction is needed in the middle of the instruction sequence in order to spill a register. Although not explicitly stated in the problem, stores have the same latency as loads (two cycles), since they use the same functional unit. Because the result of the store is not needed for several cycles after it completes (when the load restores the spilled value), it would take a very long latency for store instructions in order to delay the last load. We don't have to worry about WAR hazards in the above sequence because instructions are issued in-order. Note that we can no longer execute the four original loads in sequence as in M3.4.B because of the lack of available registers. We can, however, execute the third load before saving the intermediate value from the first MUL instruction.

	Decoded Instruction (Enters Issue)	**Issued Instruction (Enters Execute)**	**WB Cycle For Issued Instruction**
0	`L.S F0, 0(R1)`	Stall	
1	`L.S F1, 0(R2)`	`L.S F0, 0(R1)`	3
2	`MUL.S F0, F0, F1`	`L.S F1, 0(R2)`	4
3	`L.S F1, 0(R3)`	Stall	
4	`S.S F0, 0(R6)`	Stall	
5	`L.S F0, 0(R4)`	`MUL.S F0, F0, F1`	9
6	`MUL.S F0, F0, F1`	`L.S F1, 0(R3)`	8
7	`L.S F1, 0(R6)`	Stall	
8	`ADD.S F0, F0, F1`	Stall	
9	`S.S F0, 0(R5)`	Stall	
10		`S.S F0, 0(R6)`	
11		`L.S F0, 0(R4)`	13
12		Stall	
13		Stall	
14		`MUL.S F0, F0, F1`	18
15		`L.S F1, 0(R6)`	17
16		Stall	
17		Stall	
18		Stall	
19		`ADD.S F0, F0, F1`	21
20		Stall	
21		Stall	
22		`S.S F0, 0(R5)`	

The number of cycles from the issue of the first load instruction to the issue of the final store instruction is 22 cycles, inclusive. The use of only two floating-point registers results in a severe performance hit.

Problem M2.4.D Register renaming and dynamic scheduling

The table below shows the cycles in which the instructions in the original code sequence are decoded, issued, and written back on the single-issue machine with register renaming and out-of-order issue. The table also contains the rename table for the architectural registers.

	Decoded/Renamed Instruction (Enters Issue)	Rename				Issued Instruction (Enters Execute)	WB Cycle For Issued Instruction
		F0	F1	F2	F3		
0	`L.S T0, 0(R1)`	T0				Stall	
1	`L.S T1, 0(R2)`	T0	T1			`L.S T0, 0(R1)`	3
2	`MUL.S T2, T0, T1`	T2	T1			`L.S T1, 0(R2)`	4
3	`L.S T3, 0(R3)`	T2	T1	T3		Stall	
4	`L.S T4, 0(R4)`	T2	T1	T3	T4	`L.S T3, 0(R3)`	6
5	`MUL.S T5, T3, T4`	T2	T1	T5	T4	`MUL.S T2, T0, T1`	9
6	`ADD.S T6, T2, T5`	T6	T1	T5	T4	`L.S T4, 0(R4)`	8
7	`S.S T6, 0(R5)`	T6	T1	T5	T4	Stall	
8						Stall	
9						`MUL.S T5, T3, T4`	13
10						Stall	
11						Stall	
12						Stall	
13						Stall	
14						`ADD.S T6, T2, T5`	16
15						Stall	
16						Stall	
17						`S.S T6, 0(R5)`	

The number of cycles from the issue of the first load instruction to the issue of the final store instruction is 17 cycles, inclusive. This is one cycle better than executing this code on an in-order machine but not quite as good as the performance of the optimized code in M2.4.B, which only required 15 cycles. The difference in performance between the statically scheduled code and the dynamically scheduled code can be attributed to the fact that only a single instruction can be decoded at a time, which limits the hardware's ability to find independent instructions to issue. The optimized version of the code from M2.4.B executing on this machine would not improve in performance over executing on an in-order machine – it would still take 15 cycles.

Note, that in cycle 5, we would get better performance if we issued the final load instruction rather than the MUL instruction. The machine doesn't know that, so it issues the instruction that entered the ROB first.

Problem M2.4.E Effect of Register Spills

The table below shows the cycles in which the instructions in the original code sequence are decoded, issued, and written back on the single-issue machine with register renaming and out-of-order issue.

	Decoded/Renamed Instruction (Enters Issue)	Rename		Issued Instruction (Enters Execute)	WB Cycle For Issued Instruction
		F0	F1		
0	`L.S T0, 0(R1)`	T0		Stall	
1	`L.S T1, 0(R2)`	T0	T1	`L.S T0, 0(R1)`	3
2	`MUL.S T2, T0, T1`	T2	T1	`L.S T1, 0(R2)`	4
3	`L.S T3, 0(R3)`	T2	T3	Stall	
4	`S.S T2, 0(R6)`	T2	T3	`L.S T3, 0(R3)`	6
5	`L.S T4, 0(R4)`	T4	T3	`MUL.S T2, T0, T1`	9
6	`MUL.S T5, T4, T3`	T5	T3	Stall	
7	`L.S T6, 0(R6)`	T5	T6	Stall	
8	`ADD.S T7, T5, T6`	T7	T6	Stall	
9	`S.S T7, 0(R5)`	T7	T6	Stall	
10				`S.S T2, 0(R6)`	12
11				`L.S T4, 0(R4)`	13
12				`L.S T6, 0(R6)`	14
13				Stall	
14				`MUL.S T5, T4, T3`	18
15				Stall	
16				Stall	
17				Stall	
18				Stall	
19				`ADD.S T7, T5, T6`	21
20				Stall	
21				Stall	
22				`S.S T7, 0(R5)`	24

It now takes 22 cycles between issue of the first load instruction and issue of the last store instruction. That is the same performance as M2.4.C, and much worse than M2.4.D.

We managed to execute two instructions out of order, but we still couldn't beat the in-order performance. The problem lies with the fact that we had to wait for the first store to issue before we could continue with the program. This is directly linked to having only two registers, thus having to store intermediate values.

Problem M2.5: Register Renaming Schemes

Problem M2.5.A Finding Operands: Original ROB scheme

Instruction	Src1 value	Regfile, ROB, rename table, or instruction?	Src2 value	Regfile, ROB, rename table, or instruction?
`sub r5,r1,r3`	1	Regfile	t_2	Rename table
`addi r6,r2,4`	2	Regfile	4	Instruction
`andi r7,r4,3`	4	ROB	3	Instruction

Problem M2.5.B Finding Operands: Future File Scheme

A source register operand for an instruction *I* can be in one of the following three possible states.

1. It can be produced by a previous instruction that has not yet completed, in which case *I* will get the tag from the rename table.
2. It can be produced by a previous instruction that has completed execution but has not yet written back to the register file. However, the previous instruction will have written the value to the future file in this case, so *I* can obtain the value from that structure.
3. It can be produced by a previous instruction that has committed its value to the register file, in which case *I* can simply read the value from the regfile.

None of the above scenarios requires *I* to fetch an operand from the ROB.

Problem M2.5.C Future File Operation

An example code sequence is:

```
LD   R2, 0(R1)
ADDI R3, R2, 1
SUB  R4, R3, R5
ADD  R3, R4, R6
```

An instruction result will be written to the ROB but not the future file if a subsequent instruction has been decoded and writes to the same destination register. To illustrate with the given example, since instruction decode occurs in order, the `ADD` instruction will be decoded after the `ADDI` instruction. Thus, the entry for `R3` in the rename table will contain a tag for the `ADD` instruction after all of the above instructions have been decoded. Now suppose that the `ADDI` instruction completes execution after the `ADD` instruction is decoded. Because the tag for `R3` will not match the tag for the `ADDI` instruction, the result of that instruction will not be written back to the future file, but it will be written back to the ROB.