Problem Set 4 Answer Guide

Problem 4.1: Superscalar Processor

Problem 4.1.A

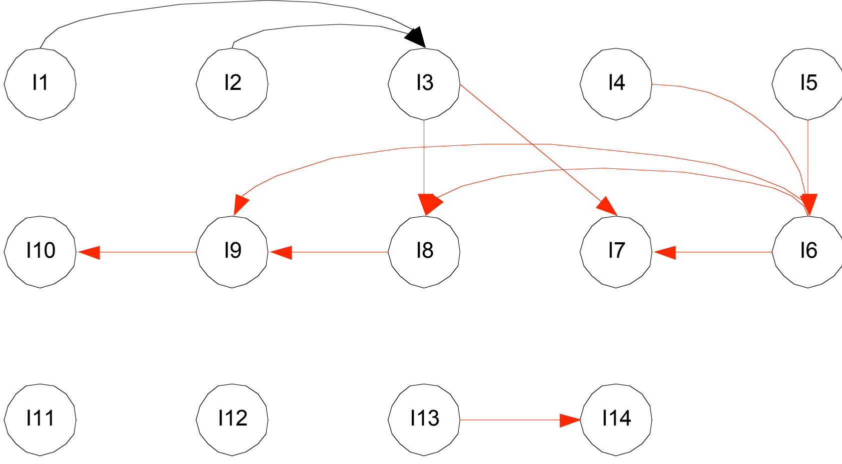
Fill in the renaming tags in the following two tables for the execution of instructions I1 to I10

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 4.1.B



Problem 4.1.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 4.1.D

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

27 cycles.

Problem 4.1.E

The behavior should repeat- should be obvious from the dependency graph (DAG) in Problem 4.1.D.

Problem 4.1.F

Yes/No ---

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 4.1.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. With each iteration depending only on the previous iteration for the FMULT on F1 value. Hence, the number of cycles is the latency of FMULT (3 + 1 cycle for writeback). At steady state, one iteration can complete every 4 cycles.

Problem 4.2: Register Renaming and Static vs. Dynamic Scheduling

Problem 4.2.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 (in cycle 1) until the issue of the final store instruction (in cycle 18) is 18 cycles, inclusive.

Problem 4.2.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 (in cycle 1) to the issue of the final store instruction (in cycle 15) is 15 cycles, inclusive. Using static scheduling has enabled us to reduce the execution time of the sequence by 17%.

Problem 4.2.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. However, if you assume a different latency in your solutions, that is acceptable as long as you state it. 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 4.2.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 (in cycle 1) to the issue of the final store instruction (in cycle 22) is 22 cycles, inclusive. Using only two floating-point registers results in a severe performance hit.

Problem 4.2.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 (in cycle 1) to the issue of the final store instruction (in cycle 17) 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 4.2.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 4.2.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 4.2.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 4.2.C, and much worse than 4.2.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 4.3: Branch Prediction

```
loop:
     LW   R4, 0(R3)
     ADDI R3, R3, 4
     SUBI R1, R1, 1
b1:
     BEQZ R4, b2
     ADDI R2, R2, 1
b2:
     BNEZ R1, loop
```

Figure 4.3-A: BP bits state diagram

Problem 4.3.A — Program

R2 contains the number of non-zero entries in the first n elements of array p.

Problem 4.3.B — 2-bit branch prediction

There are 7 mispredicts (shown in italics).

System State		Branch Predictor		Branch Behavior	
PC	R3/R4	b1 bits	b2 bits	Predicted	Actual
b1	4/1	**10**	10	N	N
b2	4/1	*10*	**10**	N	***T***
b1	8/0	**10**	*11*	N	***T***
b2	8/0	*11*	**11**	N	***T***
b1	12/1	**11**	*00*	N	N
b2	12/1	*10*	**00**	T	T
b1	16/0	**10**	*00*	N	***T***
b2	16/0	*11*	**00**	T	T
b1	20/1	**11**	*00*	N	N
b2	20/1	*10*	**00**	T	T
b1	24/0	**10**	*00*	N	***T***
b2	24/0	*11*	**00**	T	T
b1	28/1	**11**	*00*	N	N
b2	28/1	*10*	**00**	T	T
b1	32/0	**10**	*00*	N	***T***
b2	32/0	*11*	**00**	T	***N***

Table 4.3-1

Problem 4.3.C **Branch prediction with one global history bit**

There are 9 mispredicts (shown in italics).

System State			Branch Predictor				Behavior	
PC	R3/R4	history bit	b1 bits		b2 bits			
			set 0	set 1	set 0	set 1	Predicted	Actual
b1	4/1	**1**	10	**10**	10	10	N	N
b2	4/1	**0**	10	*10*	**10**	10	N	***T***
b1	8/0	**1**	10	**10**	*11*	10	N	***T***
b2	8/0	1	10	*11*	11	**10**	N	***T***
b1	12/1	1	10	**11**	11	*11*	N	N
b2	12/1	0	10	*10*	**11**	11	N	***T***
b1	16/0	1	10	**10**	*00*	11	N	***T***
b2	16/0	1	10	*11*	00	**11**	N	***T***
b1	20/1	1	10	**11**	00	*00*	N	N
b2	20/1	0	10	*10*	**00**	00	T	T
b1	24/0	1	10	**10**	*00*	00	N	***T***
b2	24/0	1	10	*11*	00	**00**	T	T
b1	28/1	1	10	**11**	00	*00*	N	N
b2	28/1	0	10	*10*	**00**	00	T	T
b1	32/0	1	10	**10**	*00*	00	N	***T***
b2	32/0	1	10	*11*	00	**00**	T	***N***

Table 4.3-2

Problem 4.3.D — Branch prediction with two global history bits

There are 7 mispredicts (shown in italics).

System State			Branch Predictor								Behavior	
PC	R3/R4	history	b1 bits				b2 bits					
		bits	set 00	set 01	set 10	set 11	set 00	set 01	set 10	set 11	Predicted	Actual
b1	4/1	**11**	10	10	10	**10**	10	10	10	10	N	N
b2	4/1	**01**	10	10	10	*10*	10	**10**	10	10	N	***T***
b1	8/0	**10**	10	10	**10**	10	10	*11*	10	10	N	***T***
b2	8/0	11	10	10	*11*	10	10	11	10	**10**	N	***T***
b1	12/1	11	10	10	11	**10**	10	11	10	*11*	N	N
b2	12/1	01	10	10	11	*10*	10	**11**	10	11	N	***T***
b1	16/0	10	10	10	**11**	10	10	*00*	10	11	N	***T***
b2	16/0	11	10	10	*00*	10	10	00	10	**11**	N	***T***
b1	20/1	11	10	10	00	**10**	10	00	10	*00*	N	N
b2	20/1	01	10	10	00	*10*	10	**00**	10	00	T	T
b1	24/0	10	10	10	**00**	10	10	*00*	10	00	T	T
b2	24/0	11	10	10	*00*	10	10	00	10	**00**	T	T
b1	28/1	11	10	10	00	**10**	10	00	10	*00*	N	N
b2	28/1	01	10	10	00	*10*	10	**00**	10	00	T	T
b1	32/0	10	10	10	**00**	10	10	*00*	10	00	T	T
b2	32/0	11	10	10	*00*	10	10	00	10	**00**	T	***N***

Table 4.3-3

Problem 4.3.E — Analysis I

The first thing to notice is that the more history bits we have, the longer it takes to get any correct prediction since we have to "train" the predictor. These start-up costs go up as the number of history bits increase.

Another thing to notice is that the single history bit does not help at all (even after we get into a steady-state phase). In both the single history bit and no history cases, the b2 branch is predicted correctly once we get past the start-up phase (since b2 is always taken). The single bit of history does not help since this history is too "nearsighted". The second history bit captures the alternating pattern of the b1 branch, and hence does not mispredict once it gets past the start-up phase. For large n then, the 2-bit history predictor is the best.

The final point of observation is that all the predictors mispredict the fall-through case (last b2 branch).

Problem 4.3.F — Analysis II

When the input is random, no prediction scheme will help predict whether b1 is taken or not. All three schemes will eventually predict b2 as always taken. However, the more history bits are used, the more sets need to be trained to predict the always taken for b2. Thus, the more history bits used, the more mispredicts of branch b2 will occur initially. The answer does not depend on the size of n. However, as n gets large, the start-up costs become insignificant among the three schemes.

The moral of the problem is: history bits are useful if there is a pattern among a sequence of branches. The longer this pattern is, the more history bits are needed to be able to recognize this pattern. If the pattern is not recognized, then global history bits can hurt because it take longer to train the branches that can be predicted correctly.

Problem 4.4: Managing Out-of-order Execution [? Hours]

Problem 4.4.A

Rename Table

R1	~~P1~~ ~~P4~~ P7
R2	~~P2~~ ~~P5~~ P8
R3	~~P3~~ P6
R4	P0

Physical Regs

P0	8016	p
P1	~~6823~~	~~p~~
P2	~~8000~~	~~p~~
P3	7	p
P4	0	p
P5	8004	p
P6	8	p
P7		
P8		
P9		

Free List

~~P4~~
~~P5~~
~~P6~~
~~P7~~
~~P8~~
P9
P1
P2
⋮

Reorder Buffer (ROB)

next to commit → (row 1); next available → (row 5)

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
~~x~~	x	lw	p	P2			r1	P1	P4
~~x~~	x	addi	p	P2			r2	P2	P5
x		beqz	p	P4					
x	x	addi	p	P3			r3	P3	P6
x		bne	p	P5	p	P0			
x		lw	p	P5			r1	P4	P7
x		addi	p	P5			r2	P5	P8
x		beqz		P7					