Problem M2.16: 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;
......
```

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

Problem M2.17

```
    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 M2.17.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						

Problem M2.15: Out-of-Order Scheduling [? Hours]

Problem M2.15.A

	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 M2.15-1

Problem M2.15.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 M2.15-2

Problem M2.16: Superscalar Processor [? Hours]

Problem M2.16.A

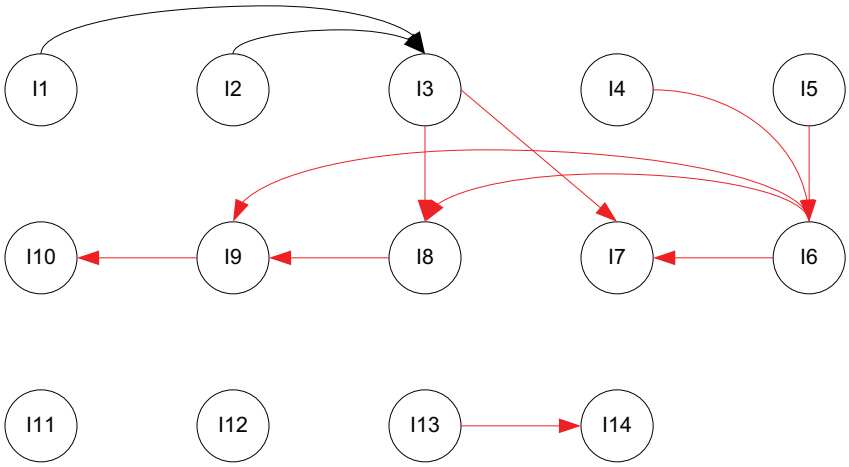
Fill in the renaming tags in the following two tables for the execution of instructions I1 to I10

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 M2.16.B



Problem M2.16.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 M2.16.D

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

27 cycles.

Problem M2.16.E

The behavior should repeat - should be obvious from the dependency graph (DAG) in Problem M2.16.D.

Problem M2.16.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 M2.16.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 M2.17: Register Renaming and Static vs. Dynamic Scheduling [? Hours]

Problem M2.17.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 M2.17.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 M2.17.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 M2.17.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 M2.17.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 M2.17.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 M2.17.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 M2.17.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 M2.17.C, and much worse than M2.17.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 M3.1: Register Renaming Schemes [? Hours]

Problem M3.1.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 M3.1.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 M3.1.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.

```
ADD  R1, R2, R3
SUB  R4, R5, R6
BEQ  R7, R8, L1     # Taken branch.
XOR  R9, R10, R11
ADD  R1, R5, R9
```

If all of the above instructions complete execution before the branch misprediction is detected, then the values of `R9` and `R1` in the future file will be the values produced by the `XOR` instruction and the second `ADD` instruction, respectively. However, because the branch was mispredicted, the `XOR` instruction and the second `ADD` instruction should never have been executed, the future file contains incorrect values for `R9` and `R1`. Either the correct values must be placed in the future file by some means, or the appropriate future file entries must be invalidated.

Problem M3.5: Fetch Pipelines [? Hours]

PC	PC Generation
F1	ICache Access
F2	
D1	Instruction Decode
D2	
RN	Rename/Reorder
RF	Register File Read
EX	Integer Execute

Problem M3.5.A — Pipelining Subroutine Returns

Immediately after what pipeline stage does the processor know that it is executing a subroutine return instruction?
D2

Immediately after what pipeline stage does the processor know the subroutine return address?
RF

How many pipeline bubbles are required when executing a subroutine return?
6

Problem M3.5.B — Adding a BTB

A subroutine can be called from many different locations and thus a single subroutine return can return to different locations. A BTB holds only the address of the last caller.

Problem M3.5.C — Adding a Return Stack

Normally, instruction fetch needs to wait until the return instruction finishes the RF stage before the return address is known. With the return stack, as soon as the return instruction is decoded in D2, instruction fetch can begin fetching from the return address. This saves 2 cycles.

A return address is pushed after a JAL/JALR instruction is decoded in D2. A return address is popped after a JR r31 instruction is decoded in D2.

Problem M3.5.D — Return Stack Operation

```
A: JAL B
A+1:
A+2:
…

B: JR r31
B+1:
B+2:
…
```

instruction	time→																	
A	PC	F1	F2	D1	D2	RN	RF	EX										
A+1		PC	F1	F2	D1	~~D2~~	~~RN~~	~~RF~~	~~EX~~									
A+2			PC	F1	F2	~~D1~~	~~D2~~	~~RN~~	~~RF~~	~~EX~~								
A+3				PC	F1	~~F2~~	~~D1~~	~~D2~~	~~RN~~	~~RF~~	~~EX~~							
A+4					PC	~~F1~~	~~F2~~	~~D1~~	~~D2~~	~~RN~~	~~RF~~	~~EX~~						
B						PC	F1	F2	D1	D2	RN	RF	EX					
B+1							PC	F1	F2	D1	~~D2~~	~~RN~~	~~RF~~	~~EX~~				
B+2								PC	F1	F2	~~D1~~	~~D2~~	~~RN~~	~~RF~~	~~EX~~			
B+3									PC	F1	~~F2~~	~~D1~~	~~D2~~	~~RN~~	~~RF~~	~~EX~~		
B+4										PC	~~F1~~	~~F2~~	~~D1~~	~~D2~~	~~RN~~	~~RF~~	~~EX~~	
A+1											PC	F1	F2	D1	D2	RN	RF	EX

Problem M3.5.E — Handling Return Address Mispredicts

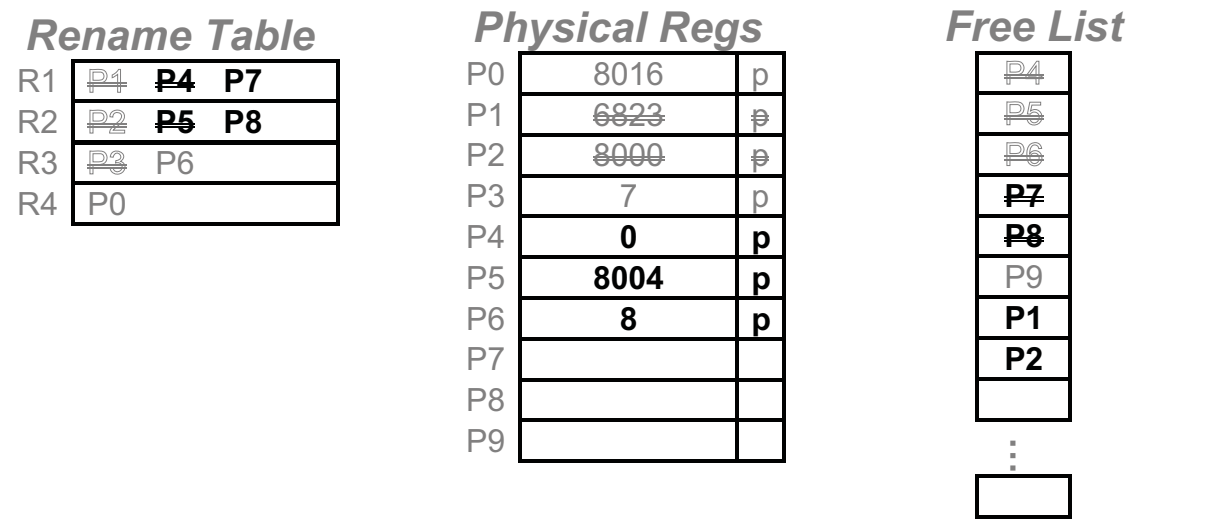
When a value is popped off the return stack after D2, it is saved for two cycles as part of the pipeline state. After the RF stage of the return instruction, the actual r31 is compared against the predicted return address. If the addresses match, then we are done. Otherwise we mux in the correct program counter at the PC stage and kill the instructions in F1 and F2. Depending on how fast the address comparison is assumed to be, you might also kill the instruction in D1. So there is an additional 2 or 3 cycles lost on a return mispredict.

Problem M3.5.F — Further Improving Performance

Ben should add a cache of the most recently encountered return instruction addresses. During F1, the contents of the cache are looked up to see if any entries match the current program counter. If so, then by the end of F1 (instead of D2) we know that we have a return instruction. We can then use the return stack to supply the return address.

Problem M3.6: Managing Out-of-order Execution [? Hours]

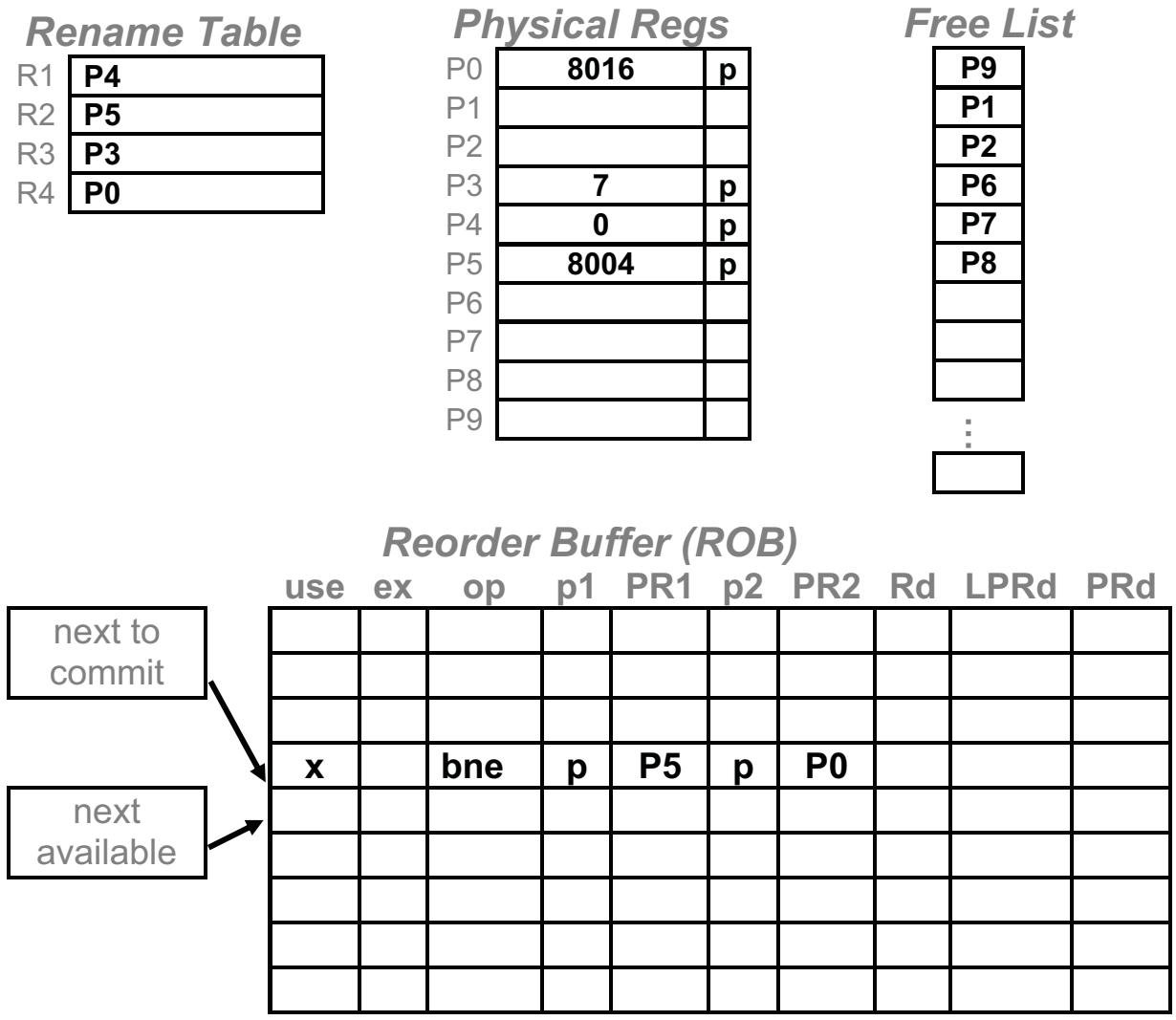
Problem M3.6.A



Reorder Buffer (ROB)

	use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
next to commit →	~~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
next available →	x		bne	**p**	P5	p	P0			
	x		**lw**	**p**	**P5**			**r1**	**P4**	**P7**
	x		**addi**	**p**	**P5**			**r2**	**P5**	**P8**
	x		**beqz**		**P7**					

Problem M3.6.B



Problem M3.6.C

Under what conditions, if any, might the loop execute at a faster rate on the in-order processor compared to the out-of-order processor?

If the out-of-order processor frequently mispredicts either of the branches, it is likely to execute the loop slower than the in-order processor. For this to be true, we must also assume that the branch misprediction penalty of the out-of-order processor is sufficiently longer than the branch resolution delay of the in-order processor, as is likely to be the case. The mispredictions may be due to deficiencies in the out-of-order processor's branch predictor, or the data-dependent branch may be fundamentally unpredictable in nature.

Under what conditions, if any, might the loop execute at a faster rate on the out-of-order processor compared to the in-order processor?

If the out-of-order processor predicts the branches with high enough accuracy, it can execute more than one instruction per cycle, and thereby execute the loop at a faster rate than the in-order processor.

Problem M3.7: Exceptions and Register Renaming [? Hours]

Problem M3.7.A — Recovering from Exceptions

By the definition of a precise exception, an exception that occurs in the middle of an x86 instruction should cause the machine state to revert to the state that previously existed right before the excepting instruction started executing. Thus a strategy to determine a precise state would be to take snapshots of the RAT only on x86 instruction boundaries (either when the last μop of an x86 instruction commits or right before the first μop of an x86 instruction is renamed).

Problem M3.7.B — Minimizing Snapshots

Ben is correct. Since an exception causes the machine to revert to the state found on an x86 instruction boundary, all the temporary state used by the μops does not need to be kept. Thus, the RAT only has to hold the rename mappings for the architectural registers, and not for T0-T7.

Problem M3.7.C — Renaming Registers

There must be at least 17 physical registers for the Bentium 4 to work properly. 16 registers are needed to hold the state of the machine at any given point in time (architectural and temporary register values), and an extra one is needed to rename an additional register using the given renaming algorithm to allow forward progress.

Problem M3.8: Multithreading [?? Hours]

Problem M3.8.A

Since there is no penalty for conditional branches, instructions take one cycle to execute unless there is a dependency problem. The following table summarizes the execution time for each instruction. From the table, the loop takes **104 cycles** to execute.

Instruction	Start Cycle	End Cycle
`LW R3, 0(R1)`	1	100
`LW R4, 4(R1)`	2	101
`SEQ R3, R3, R2`	101	101
`BNEZ R3, End`	102	102
`ADD R1, R0, R4`	103	103
`BNEZ R1, Loop`	104	104

Problem M3.8.B

If we have N threads and the first load executes in cycle 1, `SEQ`, which depends on the load, executes in cycle $2 \cdot N + 1$. To fully utilize the processor, we need to hide the 100-cycle memory latency, $2 \cdot N + 1 \geq 101$. The minimum number of thread needed is **50**.

Problem M3.8.C

	Throughput	**Latency**
Better	√	
Same		
Worse		√

Problem M3.8.D

In steady state, each thread can execute 6 instructions (`SEQ`, `BNEZ`, `ADD`, `BNEZ`, `LW`, `LW`). Therefore, to hide 99 cycles between the second `LW` and `SEQ`, a processor needs $\lceil 99/6 \rceil + 1 = 18$ threads.

Problem M3.9: Multithreaded architectures [?? Hours]

Problem M3.9.A

4, since the largest latency for any instruction is 4.

Problem M3.9.B

2/12 = 0.17 flops/cycle, on average we complete a loop every 12 cycles

Problem M3.9.C

Yes, we can hide the latency of the floating point instructions by moving the add instructions in between floating point and store instructions – we'd only need 3 threads. Moving the third load up to follow the second load would further reduce thread requirement to only 2.

Problem M3.10: Multithreading [?? Hours]

Problem M3.10.A

Fixed Switching: 6 Thread(s)

If we have N threads and L.D. executes in cycle 1, FADD, which depends on the load executes in cycle 2N + 1. To fully utilize the processor, we need to hide 12-cycle memory latency, $2N + 1 \geq 13$. The minimum number of thread needed is 6.

Data-dependent Switching: 4 Thread(s)

In steady state, each thread can execute 4 instructions (FADD, BNE, LD, ADDI). Therefore, to hide 11 cycles between ADDI and FADD, a processor needs $\lceil 11/4 \rceil + 1 = 4$ threads.

Problem M3.10.B

Fixed Switching: 2 Thread(s)

Each FADD depends on the previous iteration's FADD. If we have N threads and the first FADD executes in cycle 1, the second FADD executes in cycle 4N + 1. To fully utilize the processor, we need to hide 5-cycle latency, $4N + 1 \geq 6$. The minimum number of thread needed is 2.

Data-dependent Switching: 2 Thread(s)

In steady state, each thread can execute 4 instructions (FADD, BNE, LD, ADDI). Therefore, to hide 2 cycles between ADDI and FADD, a processor needs $\lceil 2/4 \rceil + 1 = 2$ threads.

Problem M3.10.C

Consider a **Simultaneous Multithreading (SMT)** machine with limited hardware resources. **Circle** the following hardware constraints that can limit the total number of threads that the machine can support. For the item(s) that you circle, **briefly describe** the minimum requirement to support **N** threads.

(A) Number of Functional Unit	Since not all the treads are executed in each cycle, the number of functional unit is not a constraint that limits the total number of threads that the machine can support.
(B) Number of Physical Registers	We need at least [N × (number of architecture registers) + 1] physical registers.
(C) Data Cache Size	This is for performance reasons.
(D) Data Cache Associatively	This is for performance reasons.

Problem M3.11: VLIW Programming [?? Hours]

Problem M3.11.A

To get 1 cycle per vector element performance, we need to use loop unrolling and software pipelining. The original loop is unrolled four times and software pipelined. Two registers (**F3** and **F7**) are used for saving partial sums, which are summed at the end.

At the start of the program n may be any value. By making successive checks and providing fix-up code, n can be guaranteed to be positive and a multiple of 4 by the prolog.

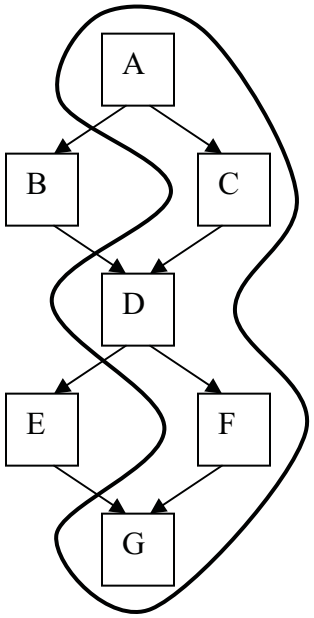
```
// R1 - points to X
// R2 - points to Y
// R5 - n
// F7 - result

    // clear partial sum registers
    MOVI2FP F3,R0
    MOVI2FP F7,R0

    // clear temporary registers used for multiply results
    MOVI2FP F2,R0
    MOVI2FP F6,R0
    MOVI2FP F10,R0
    MOVI2FP F14,R0

    // n must be greater than 0
    SGT     R3,R5,R0
    BEQZ    R3,end     // if !(n>0) goto end

    // n must be greater than 0
    ANDI  R3,R5,#3
    BEQZ  R3,prolog

    // (n>0) && ((n%4)!=0)
    SUB   R5,R5,R3
L1:
    L.S   F3,0(R1); L.S  F4,0(R2); SUBI R3,R3,#1
    MUL.S F3,F3,F4; ADDI R1,R1,#4;
    ADD.S F7,F7,F3; ADDI R2,R2,#4; BNEZ R3,L1

    BEQZ  R5,end

    // (n>=4) && ((n%4)==0)
prolog:
    L.S  F0, 0(R1); L.S  F1, 0(R2); SUBI R5,R5,#4
    L.S  F4, 4(R1); L.S  F5, 4(R2); ADDI R1,R1,#16
    L.S  F8,-8(R1); L.S  F9, 8(R2); ADDI R2,R2,#16
    L.S F12,-4(R1); L.S F13,-4(R2); BEQZ R5,epilog

    L.S  F0, 0(R1); L.S  F1, 0(R2); MUL.S  F2, F0, F1; SUBI R5,R5,#4
    L.S  F4, 4(R1); L.S  F5, 4(R2); MUL.S  F6, F4, F5; ADDI R1,R1,#16
    L.S  F8,-8(R1); L.S  F9, 8(R2); MUL.S F10, F8, F9; ADDI R2,R2,#16
    L.S F12,-4(R1); L.S F13,-4(R2); MUL.S F14,F12,F13; BEQZ R5,epilog

loop:
    L.S  F0, 0(R1); L.S  F1, 0(R2); MUL.S  F2, F0, F1; ADD.S F3,F3, F2; SUBI R5,R5,#4
```

```
    L.S  F4, 4(R1); L.S  F5, 4(R2); MUL.S  F6, F4, F5; ADD.S F7,F7, F6; ADDI R1,R1,#16
    L.S  F8,-8(R1); L.S  F9, 8(R2); MUL.S F10, F8, F9; ADD.S F3,F3,F10; ADDI R2,R2,#16
    L.S F12,-4(R1); L.S F13,-4(R2); MUL.S F14,F12,F13; ADD.S F7,F7,F14; BNEZ R5,loop

epilog:
    MUL.S  F2, F0, F1; ADD.S F3,F3, F2
    MUL.S  F6, F4, F5; ADD.S F7,F7, F6
    MUL.S F10, F8, F9; ADD.S F3,F3,F10
    MUL.S F14,F12,F13; ADD.S F7,F7,F14

    ADD.S F3,F3, F2
    ADD.S F7,F7, F6
    ADD.S F3,F3,F10
    ADD.S F7,F7,F14

    ADD.S F7,F7,F3

end:
```

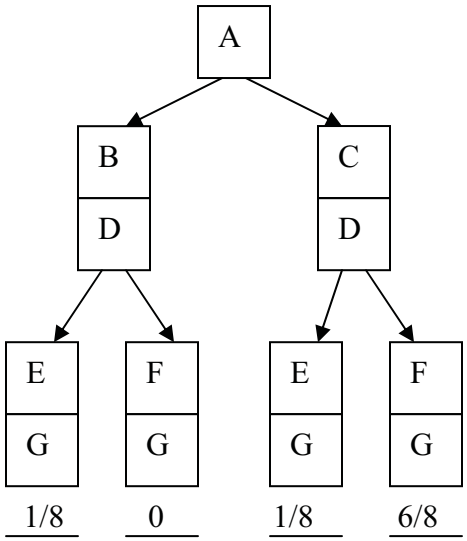
Problem M3.12: Trace Scheduling

Problem M3.12.A

Program's control flow graph



Decision tree



Problem M3.12.B

```
ACF:    ld   r1, data
        div  r3, r6, r7 ;; X <- V2/V3
        mul  r8, r6, r7 ;; Y <- V2*V3
D:      andi r2, r1, 3  ;; r2 <- r1%4
        bnez r2, G
A:      andi r2, r1, 7  ;; r2 <- r1%8
        bnez r2, E
B:      div  r3, r4, r5 ;; X <- V0/V1
E:      mul  r8, r4, r5 ;; Y <- V0*V1
G:
```

Problem M3.12.C

Assume that the load takes x cycles, divide takes y cycles, and multiply takes z cycles. Approximately how many cycles does the original code take? (ignore small constants)
x+max(y,z)

Approximately how many cycles does the new code take in the best case? **max(x,y,z)**

Problem M3.13: VLIW machines [?? Hours]

Problem M3.13.A

See **Table M3.13-1** on the next page.

Problem M3.13.B

12 cycles, 2/12=0.17 flops per cycle

Problem M3.13.C

3 instructions, because there are 5 memory ops and 5 ALU ops, and we can only issue 2 of them per instruction. (OR 4 instructions, because the slowest operation has a 4-cycle latency.)

Here is the resulting code.

add r1, r1, 4	add r2, r2, 4	ld f1, 0(r1)	ld f2, 0(r2)		fmul f4, f2, f1
add r3, r3, 4	add r4, r4, -1	ld f3, -4(r3)	st f4, -8(r1)	fadd f5, f4, f3	
	bnez r4, loop		st f5, -12(r3)		

for a particular instruction, white background corresponds to first iteration of the loop, grey background to the second iteration, yellow background to third, and blue to fourth. Note, one does not need to write the code to get an answer, because it's just a question of how many instructions are needed to express all the operations.

Problem M3.13.D

2/3=0.67 flops per cycle, 4 iterations at a time.

ALU1	ALU2	MU1	MU2	FADD	FMUL
`add r1, r1, 4`	`add r2, r2, 4`	`ld f1, 0(r1)`	**`ld f2, 0(r2)`**		
`add r3, r3, 4`	`add r4, r4, -1`	`ld f3, 0(r3)`			
					`fmul f4, f2, f1`
			`st f4, -4(r1)`	**`fadd f5, f4, f3`**	
	`bnez r4, loop`	**`st f5, -4(r3)`**			

Table M3.13-1: VLIW Program

Problem M3.13.E

We would need 5 instructions to execute two iterations and we would get 4/5=0.8 flops/cycle.

Problem M3.13.F

Same as above - 0.8 flops/cycle. We are fully utilizing the memory units, so we can't execute more loops/cycle.

Problem M3.13.G

No. We need to unroll the loop once to have an even number of memory ops. Use of the rotating registers would not allow us to squeeze in more memory ops per iteration, so we'd still need 5 instructions.

Problem M3.13.H

This is actually rather tricky. The correct answer is 5, because without interlocks, we can use the registers just as values come in for them, using the execution units to "store" the loops. The intuitive answer is 100 though.

Problem M3.13.I

There are approximately 100 instructions required, because maximum latency will be 100 cycles.

Problem M3.14: VLIW & Vector Coding [?? Hours]

Ben Bitdiddle has the following C loop, which takes the absolute value of elements within a vector.

```
for (i = 0; i < N; i++) {
    if (A[i] < 0)
        A[i] = -A[i];
}
```

Problem M3.14.A

```
; Initial Conditions:
;     R1 = N
;     R2 = &A[0]


      SGT R3, R1, R0
      BEQZ R3, end                                  ; R3 = (N > 0) | special case N ≤ 0
loop: LW R4, 0(R2)       | SUBI R1, R1, #1        ; R4 = A[i] | N--
      SLT R5, R4, R0     | ADDI R2, R2, #4        ; R5 = (A[i] < 0) | R2 = &A[i+1]
      BEQZ R5, next      |                        ; skip if (A[i]≥0)
      SUB R4, R0, R4     |                        ; A[i] = -A[i]
      SW R4, -4(R2)      |                        ; store updated value of A[i]
next: BNEZ R1, loop      |                        ; continue if N > 0
end:
```

Average Number of Cycles: ½ × (6 + 4) = 5

```
; SOLUTION #2

      SGT R3, R1, R0
      BNEZ R3, end                                  ; R3 = (N > 0) | special case N ≤ 0
loop: LW R4, 0(R2)       | SUBI R1, R1, #1        ; R4 = A[i] | N--
      SLT R5, R4, R0     | ADDI R2, R2, #4        ; R5 = (A[i] < 0) | R2 = &A[i+1]
      BNEZ R5, next      | SUB R4, R0, R4         ; skip if (A[i]≥0) | A[i] = -A[i]
      SW R4, -4(R2)      |                        ; store updated value of A[i]
next: BNEZ R1, loop      |                        ; continue if N > 0
end:
```

Average Number of Cycles: ½ × (5 + 4) = 4.5

NOTE: Although this solution minimizes code size and average number of cycles per element for this loop, it causes extra work because it subtracts regardless of whether it has to or not.