Problem M6.1: Complex Pipelining Dependencies

Consider the following instruction sequence. An equivalent sequence of C-like pseudocode is also provided.

```
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;
……
```

Fill out the table below to identify all Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW) dependencies in the above sequence. Do not worry about memory dependencies for this question. The dependency between I2 and I3 is already filled in for you.

Current Instruction \ **Earlier (Older) Instruction**	I1	I2	I3	I4	I5	I6
I1	-					
I2		-				
I3		RAW	-			
I4				-		
I5					-	
I6						-

Problem M6.2: Out-of-order Scheduling

Ben Bitdiddle is adding a floating-point unit to the basic MIPS pipeline. He has patterned the design after the IBM 360/91's floating-point unit. His FPU has one adder, one multiplier, and one load/store unit. The *adder* has a four-cycle latency and is fully pipelined. The *multiplier* has a fifteen-cycle latency and is fully pipelined. Assume that loads and stores take 1 cycle (plus one cycle for the write-back stage for loads) and that we have perfect branch prediction.

There are 4 floating-point registers, **`F0-F3`**. These are separate from the integer registers. There is a single write-back port to each register file. In the case of a write-back conflict, the older instruction writes back first. Floating-point instructions (and loads writing floating point registers) must spend one cycle in the write-back stage before their result can be used. Integer results are available for bypass the next cycle after issue.

Ben is now deciding whether to go with (a) in-order issue using a scoreboard, (b) out-of-order issue, or (c) out-of-order issue with register renaming. His favorite benchmark is this DAXPY loop central to Gaussian elimination (Hennessy and Patterson, 291). The following code implements the operation Y=aX+Y for a vector of length 100. Initially R1 contains the base address for X, R2 contains the base address for Y, and F0 contains `a`. Your job is to evaluate the performance of the three scheduling alternatives on this loop.

```
   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 M6.2.A **In-order using a scoreboard**

Fill in the scoreboard in table M6.2-1 to simulate the execution of one iteration of the loop for in-order issue using a scoreboard. Keep in mind that, in this scheme, no instruction is issued that has a WAW hazard with any previous instruction that has not written back (as mentioned in the lecture slides). Recall the WB stage is only relevant for FP instructions (integer instructions can forward results). You may use ellipses in the table to represent the passage of time (to compress repetitive lines).

In steady state, how many cycles does each iteration of the loop take? What is the bottleneck?

Problem M6.2.B Out-of-order

Now consider a single-issue out-of-order implementation. In this scheme, the issue stage buffer holds multiple instructions waiting to issue. The decode stage can add up to one instruction per cycle to the issue buffer. The decode stage adds an instruction to the issue buffer if there is space and if the instruction does not have a WAR hazard with any previous instruction that has not issued or a WAW hazard with any previous instruction that has not written back. Assume you have an infinitely large issue buffer. Assume only one instruction can be dispatched from the issue buffer at a time.

Table M6.2-2 represents the execution of one iteration of the loop ***in steady state***. Fill in the cycle numbers for the cycles at which each instruction issues and writes back. The first row has been filled out for you already; please complete the rest of the table. Note that the order of instructions listed is not necessarily the issue order. We define cycle 0 as the time at which instruction I_1 is issued.

Draw arrows for the RAW, WAR, and WAW dependencies that are involved in the *critical path* of the loop in table M2.1-2. In steady state, how many cycles does each iteration of the loop take?

Problem M6.2.C Register Renaming

The number of registers specified in an ISA limits the maximum number of instructions that can be in the pipeline. This question studies register renaming to solve this problem. In this question, we will consider an ideal case where we have unlimited hardware resources for renaming registers.

Table M6.2-3 shows instructions from our benchmark for two iterations using the same format as Table M6.2-2. First, fill in the new register names for each instruction, where applicable. Since we have an infinite supply of register names, you should use a new register name each time a register is written (T0, T1, T2, etc). Keep in mind that after a register has been renamed, subsequent instructions that refer to that register need to refer instead to the new register name. You may find it helpful to create a rename table. Rename both integer and floating-point instructions.

Next, fill in the cycle numbers for the cycles at which each instruction issues and writes back. The decode stage can add up to one instruction per cycle to the re-order buffer (ROB). Assume that instruction I_2 was decoded in cycle 0, and cannot be issued until cycle 2. Also assume that you have an infinitely large ROB.

In steady state, how many cycles does each iteration of the loop take? What is the performance bottleneck?

Instr. Issued	Time (cycles)	Functional Unit Status					Floating Point 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

Table M6.2-1

	Time			Op	Dest	Src1	Src2
	Decode → Issue	Issued	WB				
I_1	-1	0	1	L.D	F2	R1	
I_2				MUL.D	F1	F2	F0
I_3				L.D	F3	R2	
I_4				ADD.D	F3	F1	F3
I_5				S.D		R2	F3
I_6				DADDUI	R1	R1	
I_7				DADDUI	R2	R2	
I_8				DSGTUI	R3	R1	
I_9				BEQZ		R3	

Table M6.2-2

	Time			Op	Dest	Src1	Src2
	Decode → Issue	Issued	WB				
I_1	-1	0	1	L.D	T0	R1	
I_2				MUL.D	T1	t0	F0
I_3				L.D	T2	R2	
I_4				ADD.D	T3		
I_5				S.D			
I_6				DADDUI			
I_7				DADDUI			
I_8				DSGTUI			
I_9				BEQZ			
I_1				L.D			
I_2				MUL.D			
I_3				L.D			
I_4				ADD.D			
I_5				S.D			
I_6				DADDUI			
I_7				DADDUI			
I_8				DSGTUI			
I_9				BEQZ			

Table M6.2-3

Problem M6.3: Out-of-Order Scheduling

This problem deals with an out-of-order single-issue processor that is based on the basic MIPS pipeline and has floating-point units. The FPU has one adder, one multiplier, and one load/store unit. The adder has a two-cycle latency and is fully pipelined. The multiplier has a ten-cycle latency and is fully pipelined. Assume that loads and stores take 1 cycle (plus one cycle for write-back for loads).

There are 4 floating-point registers, `F0-F3`. These are separate from the integer registers. There is a single write-back port to each register file. In the case of a write-back conflict, the older instruction writes back first. Floating-point instructions (including loads writing floating point registers) must spend one cycle in the write-back stage before their result can be used. Integer results are available for bypass the next cycle after issue.

To maximize number of instructions that can be in the pipeline, register renaming is used. The decode stage can add up to one instruction per cycle to the re-order buffer (ROB).

The instructions are committed in order and only one instruction may be committed per cycle. The earliest time an instruction can be committed is one cycle after write back.

For the following questions, we will evaluate the performance of the code segment in Figure M6.3-A.

I_1	`L.D`	`F1, 5(R2)`
I_2	`MUL.D`	`F2, F1, F0`
I_3	`ADD.D`	`F3, F2, F0`
I_4	`ADDI`	`R2, R2, 8`
I_5	`L.D`	`F1, 5(R2)`
I_6	`MUL.D`	`F2, F1, F1`
I_7	`ADD.D`	`F2, F2, F3`

Figure M6.3-A

Problem M6.3.A

For this question, we will consider an ideal case where we have unlimited hardware resources for renaming registers. Assume that you have an **infinitely large ROB**.

Your job is to complete Table M6.3-1. Fill in the cycle numbers when each instruction enters the ROB, issues, writes back, and commits. Also fill in the new register names for each instruction, where applicable. Since we have an infinite supply of register names, you should use a new register name each time a register is written (T0, T1, T2, … etc). Keep in mind that after a register has been renamed, subsequent instructions that refer to that register need to refer instead to the new register name.

	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				ADD.D			
I_4					ADDI			-
I_5					L.D			-
I_6					MUL.D			
I_7					ADD.D			

Table M6.3-1

Problem M6.3.B

For this question, assume that you have a **two-entry ROB**. An ROB entry can be reused **one cycle** after the instruction using it commits.

Your job is to complete Table M6.3-2. Fill in the cycle numbers when each instruction enters the ROB, issues, writes back, and commits. Also fill in the new register names for each instruction, where applicable.

	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				ADD.D			
I_4					ADDI			-
I_5					L.D			-
I_6					MUL.D			
I_7					ADD.D			

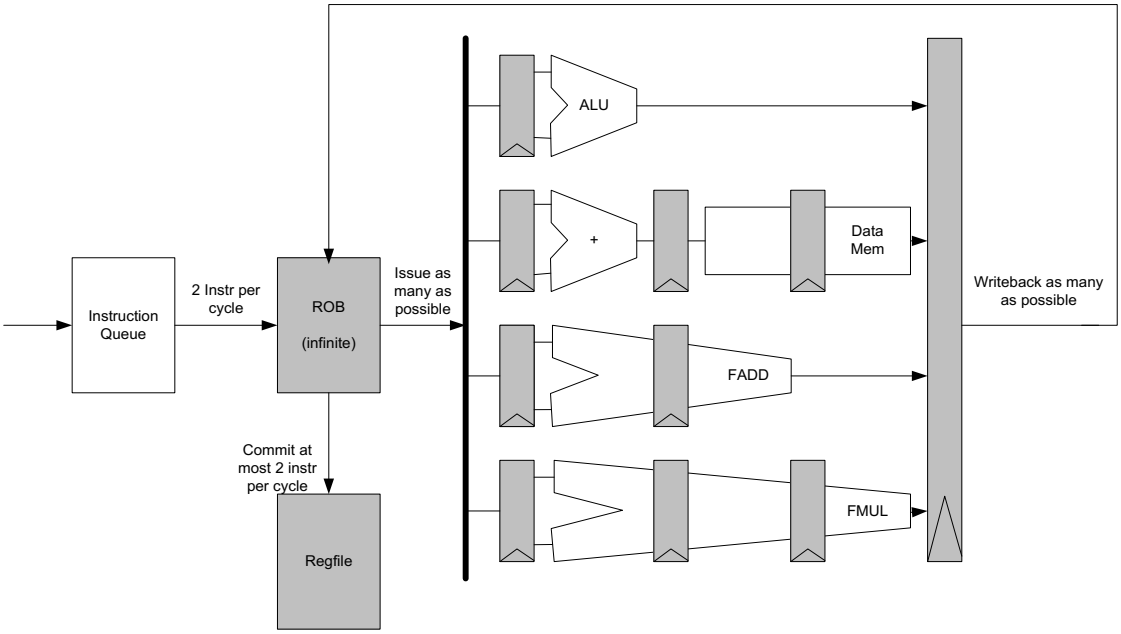
Table M6.3-2

Problem M6.4: Superscalar Processor

Consider the out-of-order, superscalar CPU shown in the diagram. It has the following features.

- Four fully-pipelined functional units: ALU, MEM, FADD, FMUL
- Instruction Fetch and Decode Unit that renames and sends 2 instructions per cycle to the ROB (assume perfect branch prediction and no cache misses)
- An unbounded length Reorder Buffer that can perform the following operations on every cycle.
  - Accept two instructions from the Instruction Fetch and Decode Unit
  - Dispatch an instruction to each functional unit including Data Memory
  - Let Write-back update an unlimited number of entries
  - Commit up to 2 instructions in-order
- There is no bypassing or short circuiting. For example, data entering the ROB cannot be passed on to the functional units or committed in the same cycle.

.



Now consider the execution of the following program on this machine using:

```
I1        loop:   LD F2, 0(R2)
I2                LD F3, 0(R3)
I3                FMUL F4, F2, F3
I4                LD F2, 4(R2)
I5                LD F3, 4(R3)
I6                FMUL F5, F2, F3
I7                FMUL F6, F4, F5
I8                FADD F4, F4, F5
I9                FMUL F6, F4, F5
I10               FADD F1, F1, F6
I11               ADD R2, R2, 8
I12               ADD R3, R3, 8
I13               ADD R4, R4, -1
I14               BNEZ R4, loop
```

Problem M6.4.A

Fill in the renaming tags in the following two tables for the execution of instructions I1 to I10. *Tags should not be reused.*

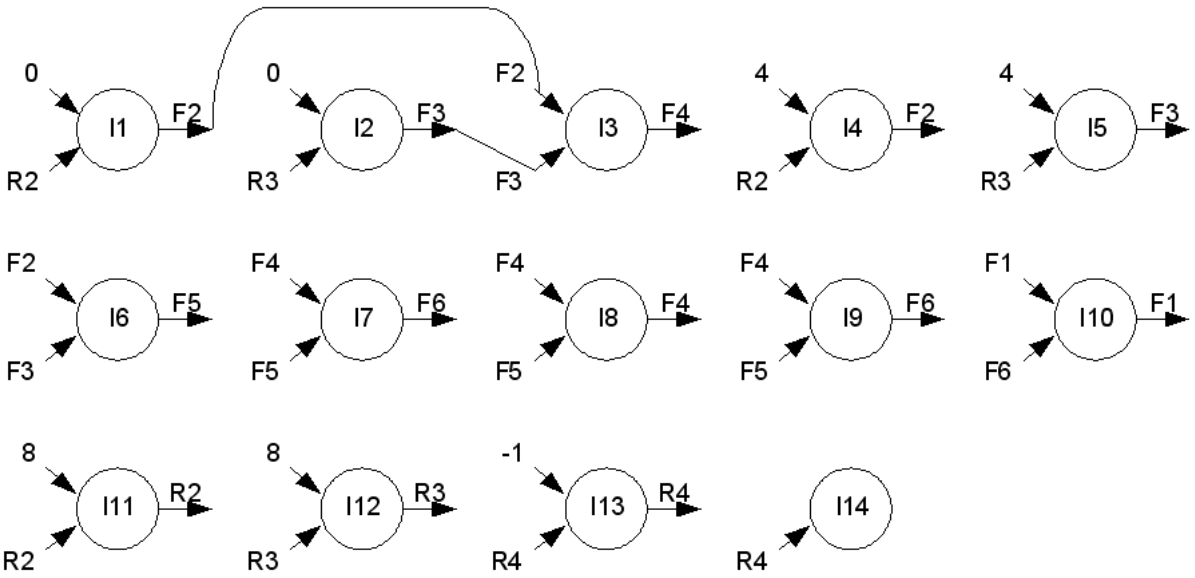
Instr #	**Instruction**	**Dest**	**Src1**	**Src2**
I1	LD F2, 0(R2)	T1	R2	0
I2	LD F3, 0(R3)	T2	R3	0
I3	FMUL F4, F2, F3			
I4	LD F2, 4(R2)		R2	4
I5	LD F3, 4(R3)		R3	4
I6	FMUL F5, F2, F3			
I7	FMUL F6, F4, F5			
I8	FADD F4, F4, F5			
I9	FMUL F6, F4, F5			
I10	FADD F1, F1, F6		F1	

Renaming table

	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10
R2										
R3										
F1										
F2	T1									
F3		T2								
F4										
F5										
F6										

Problem M6.4.B

Consider the execution of ***one*** iteration of the loop (I1 to I14). In the following diagram draw the data dependencies between the instructions after register renaming



Problem M6.4.C

The attached table is a data structure to record the times when some activity takes place in the ROB. For example, one column records the time when an instruction enters ROB, while the last two columns record, respectively, the time when an instruction is dispatched to the FU's and the time when results are written back to the ROB. This data structure has been designed to test your understanding of how a Superscalar machine functions.

Fill in the blanks in the last two columns up to slot T13 (you may use the source columns for book keeping).

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			F3	7	F4		
T4	**LD F2, 4(R2)**	2	C	2	R2		F2		
T5	**LD F3, 4(R3)**	3	C	3	R3		F3		
T6	**FMUL F5, F2, F3**	3					F5		
T7	**FMUL F6, F4, F5**	4					F6		
T8	**FADD F4, F4, F5**	4					F4		
T9	**FMUL F6, F4, F5**	5					F6		
T10	**FADD F1, F1, F6**	5					F1		
T11	**ADD R2, R2, 8**	6	R2	6	C	6	R2		
T12	**ADD R3, R3, 8**	6	R3	6	C	6	R3		
T13	**ADD R4, R4, -1**	7	R4	7	C	7	R4		
T14	**BNEZ R4, loop**	7			C	Loop			
T15	**LD F2, 0(R2)**	8	C	8			F2	10	14
T16	**LD F3, 0(R3)**	8	C	8			F3	11	15
T17	**FMUL F4, F2, F3**	9					F4		
T18	**LD F2, 4(R2)**	9	C	9			F2		
T19	**LD F3, 4(R3)**	10	C	10			F3		
T20	**FMUL F5, F2, F3**	10					F5		
T21	**FMUL F6, F4, F5**	11					F6		
T22	**FADD F4, F4, F5**	11					F4		
T23	**FMUL F6, F4, F5**	12					F6		
T24	**FADD F1, F1, F6**	12					F1		
T25	**ADD R2, R2, 8**	13			C	13	R2		
T26	**ADD R3, R3, 8**	13			C	13	R3		
T27	**ADD R4, R4, -1**	14			C	14	R4		
T28	**BNEZ R4, loop**	14			C	Loop			

Problem M6.4.D

Identify the instructions along the longest latency path in completing this iteration of the loop (up to instruction 13). Suppose we consider an instruction to have executed when its result is available in the ROB. How many cycles does this iteration take to execute?

Problem M6.4.E

Do you expect the same behavior, i.e., the same dependencies and the same number of cycles, for the next iteration? (You may use the slots from T15 onwards in the attached diagram for bookkeeping to answer this question). Please give a simple reason why the behavior may repeat, or identify a resource bottleneck or dependency that may preclude the repetition of the behavior.

Problem M6.4.F

Can you improve the performance by adding at most one additional memory port and an FP Multiplier? Explain briefly.

Yes / No

Problem M6.4.G

What is the minimum number of cycles needed to execute a typical iteration of this loop if we keep the same latencies for all the units but are allowed to use as many FUs and memory ports and are allowed to fetch and commit as many instructions as we want.

Problem M6.5: Register Renaming and Static vs. Dynamic Scheduling

The following MIPS code calculates the floating-point expression E = A * B + C * D, where the addresses of A, B, C, D, and E are stored in R1, R2, R3, R4, and R5, respectively:

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

Problem M6.5.A — Simple Pipeline

Calculate the number of cycles this code sequence would take to execute (i.e., the number of cycles between the issue of the first load instruction and the issue of the final store, inclusive) on a simple in-order pipelined machine that has no bypassing. The datapath includes a load/store unit, a floating-point adder, and a floating-point multiplier. Assume that loads have a two-cycle latency, floating-point multiplication has a four-cycle latency and floating-point addition has a two-cycle latency. Write-back for floating-point registers takes one cycle. Also assume that all functional units are fully pipelined and ignore any write-back conflicts. Give the number of cycles between the issue of the first load instruction and the issue of the final store, inclusive.

Problem M6.5.B — Static Scheduling

Reorder the instructions in the code sequence to minimize the execution time. Show the new instruction sequence and give the number of cycles this sequence takes to execute on the simple in-order pipeline.

Problem M6.5.C — Fewer Registers

Rewrite the code sequence, but now using only two floating-point registers. Optimize for minimum run-time. You may need to use temporary memory locations to hold intermediate values (this process is called register-spilling when done by a compiler). List the code sequence and give the number of cycles it takes to execute.

Problem M6.5.D **Register renaming and dynamic scheduling**

Simulate the effect of running the original code on a single-issue machine with register renaming and out-of-order issue. Ignore structural hazards apart from the single instruction decode per cycle. Show how the code is executed and give the number of cycles required. Compare it with results from the optimized execution in M2.4.B.

Problem M6.5.E **Effect of Register Spills**

Now simulate the effect of running the code you wrote in M2.4.C on the single-issue machine with register renaming and out-of-order issue from M2.4.D. Compare the number of cycles required to execute the program. What are the differences in the program and/or architecture that change the number of cycles required to execute the program? You should assume that all load instructions before a store must issue before the store is issued, and load instructions after a store must wait for the store to issue.

Problem M6.6: Register Renaming Schemes

This problem requires the knowledge of Handout Out-of-Order Execution with ROB and Lectures 8 and 9. Please, read these materials before answering the following questions.

Future File Scheme

In order to eliminate the step of reading operands from the reorder buffer in the decode stage, we can insert a second register file into the processor shown in Figure M6.6-A, called the *future file*. The future file contains the most up-to-date speculatively-executed value for a register, while the primary register file contains committed values. Each entry in the future file has a valid bit. A summary of the operations is given below.

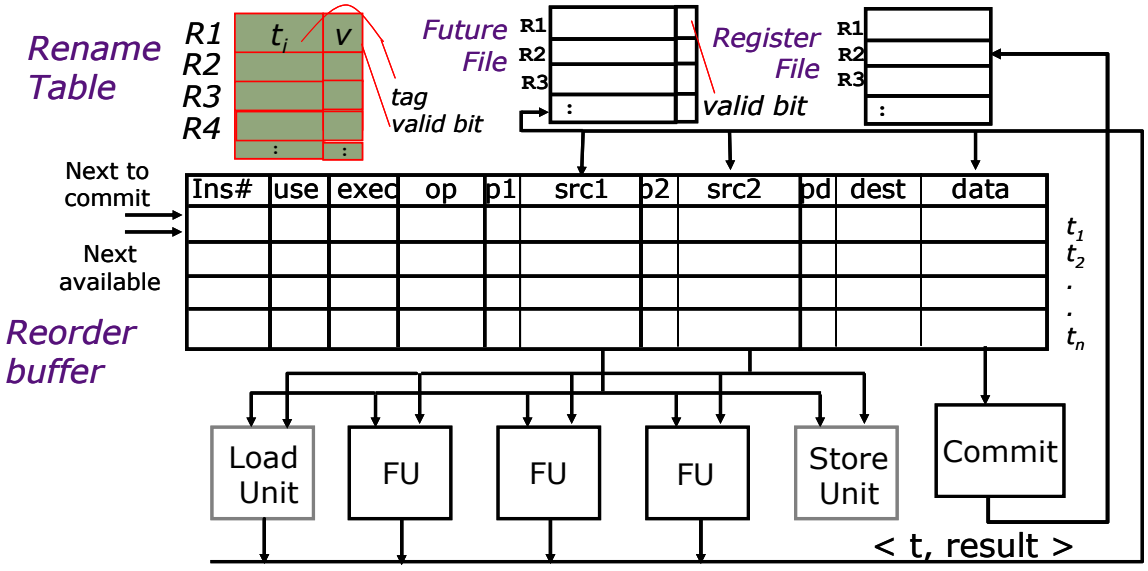


Figure M6.6-A

Only the decode and write-back stages have to change from the baseline implementation in Handout *Out-of-Order Execution with ROB* to implement the future file, as described below.

Decode: The Rename table, the register file, and the future file are read simultaneously. If the rename table has the valid bit set for an operand, then the value has not yet been produced and the tag will be used. Otherwise, if the future file has a valid bit set for its entry, then use the future file value. Otherwise, use the register file value. The instruction is assigned a slot in the ROB (the ROB index is this instruction's tag). If the instruction writes a register, its tag is written to the destination register entry in the rename table.

Write-Back: When an instruction completes execution, the result, if any, will be written back to the *data* field in the reorder buffer and the *pd* bit will be set. Additionally, any dependent instructions in the reorder buffer will receive the value. If the tag in the rename table for this register matches the tag of the result, the future file is written with the value and the valid bit on the rename table entry is cleared.

Problem M6.6.A **Finding Operands: Original ROB scheme**

Consider the original ROB scheme in Handout *Out-of-Order Execution with ROB*, and suppose the processor state is as given in Figure H4-A. Assume that the following three instructions enter the ROB simultaneously in a single cycle, and that no instruction commits or completes execution in this cycle. In the table below, write the contents of each instruction's source operand entries (either a register value or a tag t_1, t_2, etc., for both Src1 and Src2) and whether that entry came from the register file, the reorder buffer, the rename table or the instruction itself.

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

Problem M6.6.B **Finding Operands: Future File Scheme**

In the future file scheme, explain why an instruction entering the ROB will never need to fetch either of its operands from the ROB.

Problem M6.6.C **Future File Operation**

Describe a situation in which an instruction result is written to the ROB but might not be written to the future file. Provide a simple code sequence to illustrate your answer.