Problem M8.1: Fetch Pipelines

Ben is designing a deeply-pipelined, single-issue, in-order MIPS processor. The first half of his pipeline is as follows:

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

There are no branch delay slots and currently there is **no** branch prediction hardware (instructions are fetched sequentially unless the PC is redirected by a later pipeline stage). Subroutine calls use **JAL/JALR** (jump and link). These instructions write the return address (PC+4) into the link register (r31). Subroutine returns use **JR r31**. Assume that PC Generation takes a whole cycle and that you cannot bypass anything into the end of the PC Generation phase.

Problem M8.1.A

Pipelining Subroutine Returns

Immediately after what pipeline stage does the processor know that it is executing a subroutine return instruction? Immediately after what pipeline stage does the processor know the subroutine return address? How many pipeline bubbles are required when executing a subroutine return?

Problem M8.1.B Adding a BTB

Louis Reasoner suggests adding a BTB to speed up subroutine returns. Why doesn't a standard BTB work well for predicting subroutine returns?

Instead of a BTB, Ben decides to add a return stack to his processor pipeline. This return stack records the return addresses of the *N* most recent subroutine calls. This return stack takes no time to access (it is always presenting a return address).

Explain how this return stack can speed up subroutine returns. Describe when and in which pipeline stages return addresses are pushed on and popped off the stack.

Problem M8.1.D

Return Stack Operation

Fill in the pipeline diagram below corresponding to the execution of the following code on the return stack machine:

```
A: JAL B
A+1: A+2:
...
B: JR r31
B+1: B+2:
...
```

Make sure to indicate the instruction that is being executed. The first two instructions are illustrated below. The crossed out stages indicate that the instruction was killed during those cycles.

instruction					time→									
A	PC	F1	F2	D1	D2	RN	RF	EX						
A+1		PC	F1	F2	D1	-D2	RN	RF	EX_					
					, and the second							, and the second		

Problem M8.1.E

Handling Return Address Mispredicts

If the return address prediction is wrong, how is this detected? How does the processor recover, and how many cycles are lost (relative to a correct prediction)?

Problem M8.1.F

Further Improving Performance

Describe a hardware structure that Ben could add, in addition to the return stack, to improve the performance of return instructions so that there is usually only a one-cycle pipeline bubble when executing subroutine returns (assume that the structure takes a full cycle to access).

Problem M8.2: Managing Out-of-order Execution

This problem investigates the operation of a superscalar processor with branch prediction, register renaming, and out-of-order execution. The processor holds all data values in a **physical register file**, and uses a **rename table** to map from architectural to physical register names. A **free list** is used to track which physical registers are available for use. A **reorder buffer (ROB)** contains the bookkeeping information for managing the out-of-order execution (but, it does not contain any register data values).

When a branch instruction is encountered, the processor predicts the outcome and takes a snapshot of the rename table. If a misprediction is detected when the branch instruction later executes, the processor recovers by flushing the incorrect instructions from the ROB, rolling back the "next available" pointer, updating the free list, and restoring the earlier rename table snapshot.

We will investigate the execution of the following code sequence (assume that there is **no** branch-delay slot):

```
loop: lw r1, 0(r2) # load r1 from address in r2
    addi r2, r2, 4 # increment r2 pointer
    beqz r1, skip # branch to "skip" if r1 is 0
    addi r3, r3, 1 # increment r3
skip: bne r2, r4, loop # loop until r2 equals r4
```

The diagram for Question M3.5.A on the next page shows the state of the processor during the execution of the given code sequence. An instance of each instruction in the loop has been issued into the ROB (the beqz instruction has been predicted not-taken), but none of the instructions have begun execution. In the diagram, old values which are no longer valid are shown in the following format: The rename table snapshots and other bookkeeping information for branch misprediction recovery are not shown.

Assume that the following events occur in order (though not necessarily in a single cycle):

- **Step 1.** The first three instructions from the next loop iteration (lw, addi, beqz) are written into the ROB (note that the bne instruction has been predicted taken).
- **Step 2.** All instructions which are ready after Step 1 execute, write their result to the physical register file, and update the ROB. Note that this step only occurs **once**.
- Step 3. As many instructions as possible commit.

Update the diagram below to reflect the processor state after these events have occurred. Cross out any entries which are no longer valid. Note that the "ex" field should be marked when an instruction executes, and the "use" field should be cleared when it commits. Be sure to update the "next to commit" and "next available" pointers. If the load executes, assume that the data value it retrieves is 0.

Ren	ame 1	Table	Ph	ysical R	egs	F	ree L	ist
R1 📮	4 P4		P0	8016	р		P4	
R2 ₽			P1	6823	р		P5	
R3 ₽			P2	8000	р		PS	
R4 P			P3	7	р		P7	
			P4				P8	
			P5				P9	
			P6					
			P7					
			P8					
			P9				:	
					•			Ī

	Reorder Buffer (ROB)										
	_	use	ex	ор	p1	PR1	p2	PR2	Rd	LPRd	PRd
next to	\longmapsto	Χ		lw	р	P2			r1	P1	P4
commit		Χ		addi	р	P2			r2	P2	P5
	•	Χ		begz		P4					
		Χ		addi	р	P3			r3	P3	P6
next		Χ		bne		P5	р	P0			
available	\longrightarrow										
	•										
											_

Assume that after the events from Question M3.6.A have occurred, the following events occur in order:

- **Step 1.** The processor detects that the beqz instruction has mispredicted the branch outcome, and recovery action is taken to repair the processor state.
- **Step 2.** The begz instruction commits.
- **Step 3.** The correct next instruction is fetched and is written into the ROB.

Fill in the diagram below to reflect the processor state after these events have occurred. Although you are not given the rename table snapshot, you should be able to deduce the necessary information from the diagram from Question M3.6.A. You do not need to show invalid entries in the diagram, but be sure to fill in all the fields which have valid data, and update the "next to commit" and "next available" pointers. Also make sure that the free list contains all available registers.

Rename Tab	ole	Phys P0 P1 P2 P3 P4 P5 P6 P7 P8 P9	sical	Reg	S		Fre	ee List	
	use	order op	But p1			-	Rd	LPRd	PRd
next to commit									
next available									

Problem M8.2.C

Consider (1) a single-issue, in-order processor with no branch prediction and (2) a multiple-issue, out-of-order processor with branch prediction. Assume that both processors have the same clock frequency. Consider how fast the given loop executes on each processor, assuming that it executes for many iterations.

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?

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?

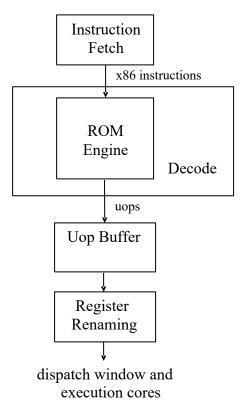
Problem M8.3: Exceptions and Register Renaming

Ben Bitdiddle has decided to start Bentel Corporation, a company specializing in high-end x86 processors to compete with Intel. His latest project is the Bentium 4, a superscalar, out-of-order processor with register renaming and speculative execution.

The Bentium 4 has 8 architectural registers (EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI). In addition, the processor provides 8 internal registers T0-T7 not visible to the ISA that can be used to hold intermediary values used by micro-operations (μ ops) generated by the microcode engine. The microcode engine is the decode unit and is used to generate μ ops for all the x86 instructions. For example, the following register-memory x86 instruction might be translated into the following RISC-like μ ops:

$$\begin{array}{ccc} ADD \; R_d, \, R_a, \, offset(R_b) & \rightarrow & LW & T0, \, offset(R_b) \\ & & ADD \; R_d, \, R_a, \, T0 \end{array}$$

All 16 µop-visible registers are renamed by the register allocation table (RAT) into a set of physical registers (P0-Pn). There is a separate shadow map structure that takes a snapshot of the RAT on a speculative branch in case of a misprediction. The block diagram for the front-end of the Bentium 4 is shown below:



Note: The decode block is actually replicated in the Bentium 4 in order to decode multiple instructions per cycle (not shown in the diagram).

Problem M8.3.A

Recovering from Exceptions

For the Bentium 4, if an x86 instruction takes an exception before it is committed, the machine state is reset back to the precise state that existed right before the excepting instruction started executing. This instruction is then re-executed after the exception is handled. Ben proposes that the shadow map structure used for speculative branches can also be used to recover a precise state in the event of an exception. Specify a strategy that can be implemented for taking the least number of snapshots of the RAT that would still allow the Bentium 4 to implement precise exception handling.

Problem M8.3.B

Minimizing Snapshots

Ben further states that the shadow map structure does not need to take a snapshot of all the registers in the Bentium 4 to be able to recover from an exception. Is Ben correct or not? If so, state which registers do not need to be recorded and explain why they are not necessary, or explain why all the registers are necessary in the snapshot.

Problem M8.3.C

Renaming Registers

Assume that the Bentium 4 has the same register renaming scheme as the Pentium 4. What is the minimum number of physical registers (P) that the Bentium 4 must have to allow register renaming to work? Explain your answer.

Problem M8.4: Out-of-order Execution (Spring 2014 Quiz 2, Part C)

In this problem, we are going to update the state of the processor when different events happen. You are given an out-of-order processor in some initial state, as described by the registers (renaming table, physical registers, and free list), one-bit branch predictor, and re-order buffer. Your job is to show the changes that occur when some event occurs, starting from the <u>same initial state except where noted</u>. For partial credit, <u>briefly</u> describe what changes occur.

Problem M8.4.A

Show the state of the processor if the first load completes (but does not commit).

00: LD R1, 0(R2)
04: ADD R3, R1, R4
08: ADD R2, R1, R2
0c: BGEZ R4, A
10: LD R3, 0(R2)
A: 14: SUB R1, R3, R2
18: ADD R4, R3, R1

BRANCH PREDICTOR						
00	1					
01	0					
10	1					
11 0						

RENAMING TABLE						
R1	P4					
R2	P6					
R3	P5					
R4	Р3					

PHYS. REG. FILE							
P0	(R1)	р					
P1	(R2)	р					
P2	(R3)	р					
Р3	(R4)	р					
P4							
P5							
P6							
P7							

_	
	FREE LIST
	P7

				R	E-ORDER B	UFFER (ROE	3)			
Next to commit	Use?	Ex	Op	P1	PR1	P2	PR2	Rd	LPRd	PRd
	Х		LD	р	P1			R1	PO	P4
	X		ADD		P4	р	Р3	R3	P2	P5
	Х		ADD		P4	р	P1	R2	P1	P6
Next available	X		BGEZ	р	Р3					

FREE LIST

Р7

Problem M8.4.B

Show the state of the processor after the next instruction is issued.

00: LD R1, 0(R2)
04: ADD R3, R1, R4
08: ADD R2, R1, R2
0c: BGEZ R4, A
10: LD R3, 0(R2)
A: 14: SUB R1, R3, R2
18: ADD R4, R3, R1

BRANCH PREDICTOR					
00	1				
01	0				
10	1				
11	0				

RENAMING TABLE						
R1	P4					
R2	P6					
R3	P5					
R4	Р3					

PH	YS. REG. FII	LE
P0	(R1)	р
P1	(R2)	р
P2	(R3)	р
Р3	(R4)	р
P4		
P5		
P6		
P7		

1										
				R	E-ORDER B	UFFER (ROE	3)			
Next to commit	Use?	Ex	Ор	P1	PR1	P2	PR2	Rd	LPRd	PRd
	X		LD	р	P1			R1	PO	P4
	Х		ADD		P4	р	P3	R3	P2	P5
	Х		ADD		P4	р	P1	R2	P1	P6
Next available	Х		BGEZ	р	P3					
→ Next available										

Problem M8.4.C

From the state at the end of Question 2, as the <u>next action</u> can the processor issue (not execute) another instruction?

In one or two sentences, what does this say about our design? How can we improve it?

FREE LIST

Р7

Show the state of the processor if the first LD triggers a page fault and after abort finishes.

00: LD R1, 0(R2)
04: ADD R3, R1, R4
08: ADD R2, R1, R2
0c: BGEZ R4, A
10: LD R3, 0(R2)
A: 14: SUB R1, R3, R2
18: ADD R4, R3, R1

Op

LD

ADD ADD

BGEZ

P1

р

Use?

Χ

Χ

Next to commit

Next available

Ex

BRANCH PREDICTOR			
00	1		
01	0		
10	1		
11	0		

RENAMING TABLE		
R1	P4	
R2	P6	
R3	P5	
R4	Р3	

RE-ORDER BUFFER (ROB)

P2

PR1

Р1

Ρ4

Р4

Р3

PH	YS. REG. FII	LE
PO	(R1)	р
P1	(R2)	р
P2	(R3)	р
Р3	(R4)	р
P4		
P5		
P6		·
P7		

	P5			
	P6			
	P7			
PR2		Rd	LPRd	PRd
		R1	P0	P4
9		R3	P2	P5
² 1		R2	P1	P6

Problem M8.5 (Spring 2015 Quiz 2, Part B)

You are given an out-of-order processor that

- Issues at most one instruction per cycle
- Commits at most one instruction per cycle
- Uses an unified physical register file

Problem M8.5.A

Consider the following code sequence:

<u>Addr</u>						
(0x24)		lw	r2,	(r4)	, #0	
(0x28)		addi	r2,	r2,	#16	
(0x2C)		lw	r3,	(r4)	, #4	
(0x30)		blez	r3,	L1		
(0x34)		addi	r4,	r2,	#8	
(0x38)		mul	r1,	r2,	r3	
(0x3C)		addi	r3,	r2,	#8	
(0x40)	L1:	add	r2,	r1,	r3	
	(0x24) (0x28) (0x2C) (0x30) (0x34) (0x38) (0x3C)	(0x24) (0x28) (0x2C) (0x30) (0x34) (0x38) (0x3C)	(0x24) lw (0x28) addi (0x2C) lw (0x30) blez (0x34) addi (0x38) mul (0x3C) addi	(0x24) lw r2, (0x28) addi r2, (0x2C) lw r3, (0x30) blez r3, (0x34) addi r4, (0x38) mul r1, (0x3C) addi r3,	(0x24) lw r2, (r4) (0x28) addi r2, r2, (0x2C) lw r3, (r4) (0x30) blez r3, L1 (0x34) addi r4, r2, (0x38) mul r1, r2, (0x3C) addi r3, r2,	(0x24) lw r2, (r4), #0 (0x28) addi r2, r2, #16 (0x2C) lw r3, (r4), #4 (0x30) blez r3, L1 (0x34) addi r4, r2, #8 (0x38) mul r1, r2, r3 (0x3C) addi r3, r2, #8

Assume the branch instruction (blez) is not taken. 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.

Older Instruction

	10	I1	12	13	14	15	16	17
10	-							
I1	WAW RAW	-						
12			-					
13				-				
14					-			
15						1		
16							-	
17								-

Younger Instruction In Problems M8.5.B to M8.5.D, you should update the state of the processor when different events happen. The starting state in each question is the same, and the event specified in each question is the ONLY event that takes place for that question. The starting state is shown in the different structures: renaming table, physical registers, free list, two-bit branch predictor, global history buffer, and reorder buffer (ROB).

Note the following conventions:

- The valid bit for any entry is represented by "1".
- The valid bit can be cleared by crossing it out.
- In the ROB, the "ex" field should be marked with "1" when an instruction starts execution, and the "use" field should be cleared when it commits. Be sure to update the "next to commit" and "next available" pointers, if necessary.
- Fill out the "after" fields in all the tables. Write new values in these boxes if the values change due to the event specified in the question. You do not have to repeat the values if they do not change due to the event.

In Questions 2 through 4, we will use the same code sequence as in Question 1:

	Addr						
ΙO	(0x24)		lw	r2,	(r4)	,	#0
I1	(0x28)		addi	r2,	r2,		#16
I2	(0x2C)		lw	r3,	(r4)	,	#4
I3	(0x30)		blez	r3,	L1		
I4	(0x34)		addi	r4,	r2,	# 8	}
I5	(0x38)		mul	r1,	r2,	r3	3
I6	(0x3C)		addi	r3,	r2,	# 8	}
I7	(0x40)	L1:	add	r2,	r1,	r3	}

The starting state of the processor is as follows:

- Instructions I0-I4 are already in the ROB.
- I0 (lw) has already finished execution.
- I1 (addi) and I2 (lw) have started executing but have not finished yet.
- I3 (blez) has been predicted to be Not-Taken by the branch predictor.
- I5 (mul) has completed the decode stage.
- I6 (addi) has completed the Fetch Stage.
- The next PC is set to 0x40, which is the PC of I7 (add).

The following figure shows the starting state of the processor. Suppose the decoded instruction I5 (mul) is now inserted into the ROB. Update the diagram to reflect the processor state after this event has occurred.

Prediction Counter					
Index	Index Before Afte				
000	11				
001	00				
010	11				
011	01				
100	10				
101	11				
110	01				
111	00				

Fetche	ed Inst. Queue
PC	Inst.
0x3C	I6 (addi)

Decoded Inst. Queue
Inst.
I5 (mul)

Branch Global History		
Before	After	
0010110		

Next PC to be fetched		
Before	After	
0x40		

Rename Table (Latest)		
Name Before Afte		After
R1	P0	
R2	P5	
R3	P6	
R4	P7	

Renam	Valid	
(Snaps	1	
Name	Before	After
R1	P0	
R2	P5	
R3	P6	
R4	Р3	

Physical Registers				
Name	Value	Valid		
P0	45	1		
P1	2	1		
P2	-3	1		
Р3	100	1		
P4	20	1		
P5				
P6				
P7				
P8				
P9				
P10				

1
Free List
P8
P9
P10

Reorder Buffer (ROB)]				
use	ex	ор	p1	PR1	p2	PR2	Rd	LPRd	PRd		
1	1	lw	1	Р3			r2	P1	P4		
1	1	addi	1	P4			r2	P4	P5	`	Next to
1	1	lw	1	Р3			r3	P2	P6	1	commit
1		blez		P6							
1		addi		P5			r4	Р3	Р7	1	NI4
										\leftarrow	Next available
											available
										1	
										1	
										1	

Start from the same processor state, shown below. Suppose now I1 (addi) has completed execution. Commit as many instructions as possible. Update the diagram to reflect the processor state after I1 execution completes and as many instructions as possible have committed. Again, assume no other events take place.

Prediction Counter		
Index	Before	After
000	11	
001	00	
010	11	
011	01	
100	10	
101	11	
110	01	
111	00	

Fetched Inst. Queue		
PC	Inst.	
0x3C	I6 (addi)	

Decoded Inst. Queue	
Inst.	
I5 (mul)	

Branch Global History		
Before	After	
0010110		

Next PC to be fetched		
Before	After	
0x40		

Rename Table (Latest)					
Name	Name Before After				
R1	P0				
R2	P5				
R3	P6				
R4	P7				

R2	P5				
R3	P6				
R4	R4 P7				
·					
Renam	e Table	Valid			
	e Table shot 1)	Valid 1			
		Valid 1 After			

Р5

Р6

Р3

R2

R3

R4

Physical Registers					
Name	Value	Valid			
P0	45	1			
P1	2	1			
P2	-3	1			
Р3	100	1			
P4	20	1			
P5					
P6					
P7					
P8					
P9					
P10					

Free List
P8
P9
P10

Reorder Buffer (ROB)											
use	ex	ор	p1	PR1	p2	PR2	Rd	LPRd	PRd		
1	1	lw	1	Р3			r2	P1	P4	<	77
1	1	addi	1	P4			r2	P4	P5		Next to
1	1	lw	1	Р3			r3	P2	P6		commit
1		blez		P6							
1		addi		P5			r4	Р3	P7		Next
										\leftarrow	available
											available
·											

Start from the same processor state, shown below. Suppose instruction I2 (lw) triggers an ALU overflow exception. Restore the architectural and microarchitectural state to recover from misspeculation. The exception handler for the processor is at address 0x8C (control is transferred to the exception handler after recovery). You do not need to worry about the number of cycles taken by recovery. Show the processor state after recovery.

Prediction Counter					
Index	Before	After			
000	11				
001	00				
010	11				
011	01				
100	10				
101	11				
110	01				
111	00				
	1				

Fetched Inst. Queue			
PC	Inst.		
0x3C	I6 (addi)		

Decoded Inst. Queue			
Inst.			
I5 (mul)			

Branch Global History				
Before After				
0010110				

Next PC to be fetched			
Before After			
0x40			

Rename Table (Latest)						
Name	Name Before After					
R1	P0					
R2	P5					
R3	P6					
R4	P7					

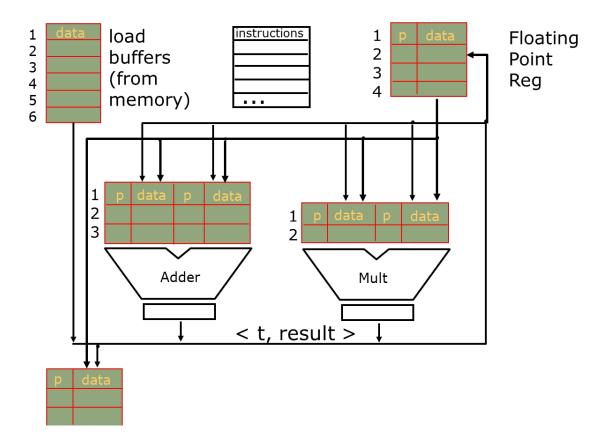
Renam	Valid		
(Snaps	(Snapshot 1)		
Name	Name Before		
R1	R1 P0		
R2	P5		
R3	R3 P6		
D/I	מם		

	sical Register	5		
Name	Value	Valid		
P0	45	1		
P1	2	1		
P2	-3	1		
Р3	100	1		
P4	20	1		
P5				
P6				
P7				
P8				
P9				
P10				

Free List				
P8				
P9				
P10				

Reorder Buffer (ROB)							1				
use	ex	ор	p1	PR1	p2	PR2	Rd	LPRd	PRd		
1	1	lw	1	Р3			r2	P1	P4	<	
1	1	addi	1	P4			r2	P4	P5	,	Next to
1	1	lw	1	Р3			r3	P2	P6		commit
1		blez		P6							
1		addi		P5			r4	Р3	P7		
										\leftarrow	Next
											available

Problem M8.6: Out-of-order Processor Design (Spring 2014 Quiz 2, Part D)



You are designing an out-of-order processor similar to the IBM 360/91 Tomasulo design shown above. This design distributes the re-order buffer around the processor, placing entries near their associated functional units. In such a design, the distributed ROB entries are called "reservation stations". Entries are allocated when the instruction is decoded and freed when the instruction is dispatched to the functional unit.

Your design achieves an average throughput of 1.5 instructions per cycle. Two-thirds of instructions are adds, and one-third are multiplies. The latency of each instruction type *from allocation to completion* is 5 cycles for adds and 14 cycles for multiplies.

Type of operation	Fraction of instructions	Average latency
Add	2/3	5
Multiply	1/3	14

The adder and multiplier are each fully pipelined with full bypassing. *Once an instruction is dispatched to the FU*, the adder takes 2 cycles and the multiplier takes 5 cycles.

Throughput	Add latency	Multiply latency
1.5	2	5

Problem M8.6.A

How many entries are in use, on average, in the reservation station at each functional unit (adder, multiplier) in the steady state? Assume there are infinite entries available if needed. What is the average latency of an instruction in this machine? For partial credit, feel free to give any formulae you believe may be important to answer this question.