Problem M3.5: 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	ICache Access
F2	
D1	Instruction Decode
D2	
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 M3.5.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 M3.5.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?

Problem M3.5.C Adding a Return Stack

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 M3.5.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~~									

Problem M3.5.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 M3.5.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 M3.6: 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: ~~P1~~. The rename table snapshots and other bookkeeping information for branch misprediction recovery are not shown.

Problem M3.6.A

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**.

Rename Table

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

Physical Regs

P0	8016	p
P1	6823	p
P2	8000	p
P3	7	p
P4		
P5		
P6		
P7		
P8		
P9		

Free List

~~P4~~
~~P5~~
~~P6~~
P7
P8
P9
⋮

Reorder Buffer (ROB)

next to commit →

next available →

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

Problem M3.6.B

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 beqz 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 Table

R1			
R2			
R3			
R4			

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5		
P6		
P7		
P8		
P9		

Free List

⋮

Reorder Buffer (ROB)

next to commit

next available

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

Problem M3.6.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?

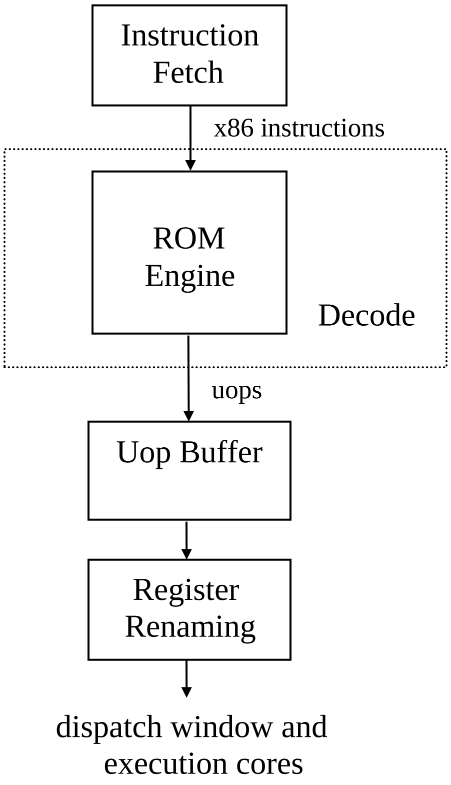
Problem M3.7: Exceptions and Register Renaming

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 (µops) generated by the microcode engine. The microcode engine is the decode unit and is used to generate µops for all the x86 instructions. For example, the following register-memory x86 instruction might be translated into the following RISC-like µops:

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

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 M3.7.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 M3.7.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 M3.7.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.