CS152 Computer Architecture and Engineering

Complex Pipelines, Out-of-Order Execution, and Speculation

Assigned 03/05/2026	Problem Set #3, Version (1.0)	*Due March 18 @ 11:59:59PT*

https://inst.eecs.berkeley.edu/~cs152/sp26/

The problem sets are intended to help you learn the material, and we encourage you to collaborate with other students and to ask questions in discussion sections and office hours to understand the problems. However, each student must turn in their own solution to the problems.

The problem sets also provide essential background material for the exam and the midterms. The problem sets will be graded primarily on an effort basis, but if you do not work through the problem sets yourself you are unlikely to succeed on the exam or midterms!

By grading primarily on an effort basis, we mean that we will award significant partial credit for demonstrating your understanding of the problem and concepts at hand. As long as reasonable assumptions and explanations are provided, we will lean towards awarding credit.

We will distribute solutions to the problem set after the deadline to give you feedback.

Assignments must be submitted through Gradescope by **11:59:59pm PT** on the specified due date. *Box/clearly mark all solutions that don't involve filling in a figure/table. Only boxed/clearly marked solutions and filled in figures/tables will be considered for grading.*

Name: ____

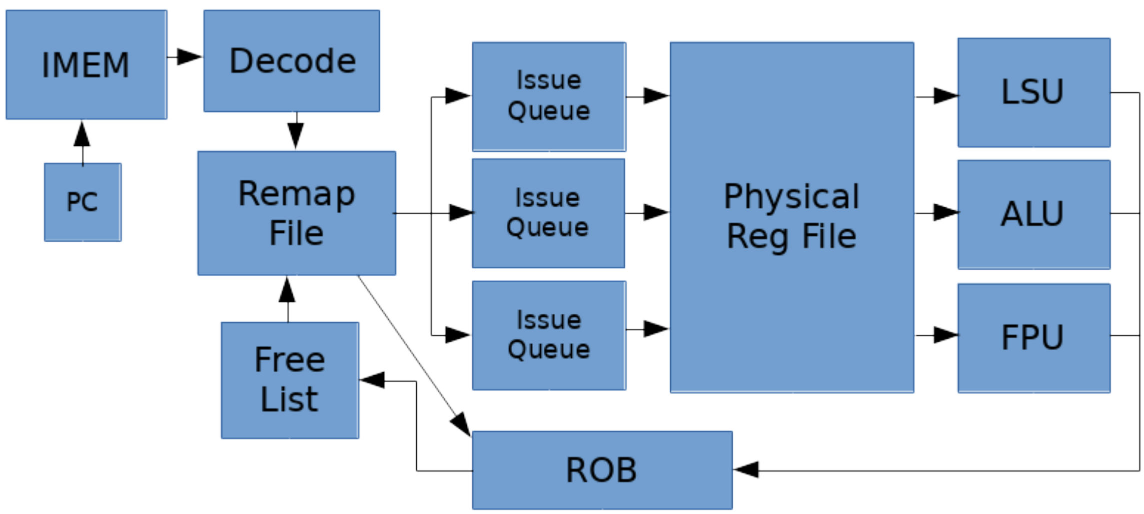
SID: ____

Collaborators (Name, SID):

____

Problem 1: Unified Physical Register Files

In this problem, we will consider an out-of-order CPU design using a unified physical register file. All data, both retired and inflight, are kept in the same physical register file. The pipeline contains a remap file that is indexed by the architectural register number and stores the physical register number the architectural register maps to. The physical register file contains the register data and a bit indicating whether the data is valid or not. The pipeline also contains a free list, which is a FIFO queue containing the physical register numbers that are not yet mapped to architectural registers. On issue, the current mappings of the destination register and two source registers are read from the remap file and stored in the ROB. The head of the free list is then popped off and written to the entry for the destination architectural register in the remap file. On a branch mispredict or exception, the remap file can be restored by going backwards through the ROB and restoring the old physical register mappings.



A) Consider a system with eight architectural registers, sixteen physical registers, and a four-entry circular ROB. The following table shows the ROB when an exception occurs in the instruction indicated in bold.

	ROB PC	Arch. Register	Old Phys. Register
	0x80001008	x1	P3
tail ->	0x8000101C	x2	P5
head ->	0x80001010	x6	P8
	0x80001014	x2	P14

The left column of the following table shows the state of the remap file when the exception is detected. Fill out the right column to show the restored state.

Arch Reg	Current State	Restored State
x0	P1	P1
x1	P6	P3
x2	P2	P14
x3	P10	P10
x4	P7	P7
x5	P4	P4
x6	P13	P13
x7	P15	P15

Note: The head represents the oldest instruction and the tail represents the youngest instruction in the ROB.

B) When can a physical register be released and put back on the free list?

There are two situations in which a physical register can be put back on the free list.

1) When handling an exception, or killing instructions on an incorrectly predicated path, the destination physical registers of younger misspeculated instructions are returned to the free list as the remap file is being regenerated.

2) When an instruction commits, the old physical register for its destination architectural register can be put back on the free list

C) How many physical registers must there be so that the pipeline never stalls due to lack of physical registers in the free list?

The number of physical registers should be the number of architectural registers plus the number of ROB entries.

Under this condition, there will always be a free physical register if there is a free ROB entry, as each ROB entry can only be assigned 0 or 1 physical destination registers. The pipeline would run out of ROB slots as soon or sooner than it would run out of physical registers

D) Here are some of the initial register mappings and the free list for a RISC-V OoO CPU with a unified physical register file containing both integer and floating-point registers.

Arch Register	Phys Register		Free List
f0	P6		P8
f1	P9		P20
f2	P3		P10
x2	P5		P21
x3	P13		P17
x4	P11		

For the following instruction sequence, indicate which physical register gets assigned as the destination register and which physical register gets added to the free list on commit.

Instruction	Destination Register	Freed Register
`fld f2, 0(x3)`	P8	P3
`fld f1, 0(x4)`	P20	P9
`fsd f1, 0(x2)`	-	-
`fadd.d f1, f2, f1`	P10	P20
`fmul.d f2, f2, f0`	P21	P8
`addi x4, x4, 8`	P17	P11
`addi x3, x3, 8`	P3	P13
`addi x2, x2, 8`	P9	P5

E) If we wanted to implement register renaming in a superscalar OoO core that can issue two instructions per cycle, what would we have to change?

The danger is that the two instructions we issue in the same cycle have a hazard between them. In that case, renaming them independently would cause wrong labels to be assigned. For example, in case of a RAW hazard, the destination register of the first instruction need to be the same physical register as the read register of the second instruction. We need to add a bypass to make this happen because in the same cycle that the second instruction reads the rename table, the first instruction has not updated it yet. The same is true of WAW hazards. If the second instruction writes to the same architectural register as the first, its previous physical destination register must be the physical destination register used by the first, not the current entry in the remap table.

Problem 2: Load/Store Speculation

A) Suppose we want to execute stores out-of-order. Could there be an issue if we allow stores to write to the cache even when there are uncommitted instructions before them in program order?

Yes, executing stores ahead of other instructions and writing to the cache before commit can lead to the following issues.

1. Imprecise exceptions: If an instruction before the store in program order has an exception, you have to roll back the architectural state. This is hard to do if you've already modified the cache contents. The same is true if the store is the result of a mispredicted branch.

2. Memory ordering: If there are load or store instructions to the same address before the completed store in program order, writing to the cache on completion will change the program's behavior from what would happen if executed in-order.

B) Suppose we bypass load values from a speculative store buffer. If the load address hits in both the store buffer and the cache, which one should we use: the data forwarded from the store buffer or the data from the cache?

If the store buffer hit corresponds to a store instruction that comes before the load in program order, we should take the store buffer data. If the store instruction comes after the load instruction in program order, we should take the cache data.

C) Suppose that we want loads and stores to execute out-of-order with respect to each other. Under what circumstances in the code below can we execute instruction 5 before executing any others? Assume this datapath implements register renaming.

```
1. add x1, x1, x2
2. sw  x5, (x2)
3. lw  x6, (x8)
4. sw  x5, (x6)
5. lw  x8, (x3)
6. add x8, x8, x8
```

We can only execute instruction 5 before the others if the address it is loading from does not overlap the addresses being stored to in instructions 2 and 4. So $|x2 - x3| \geq 4$ and $|x6 - x3| \geq 4$.

D) Under what circumstances can we execute instruction 4 in the code above before executing any others?

Instruction 4 has a RAW hazard with instruction 3 because it depends on the value of register x6 that is being loaded by instruction 3. Therefore, it cannot execute first. It does not have any dependencies on any other instruction, since stores do not change architectural state until they are committed, so it can execute as soon as instruction 3 completes.

E) Now assume that we execute instruction 5 before all other instructions, but instruction 5 causes an exception (e.g., page fault). We want to provide precise exceptions in this processor. What happens with instructions 1, 2, 3, 4, and 6 before execution switches to the OS handler? What should happen if instructions, 1, 2, 3, or 4 also raise an exception?

To provide precise exceptions, we have to execute and commit all instructions prior to instruction 5 before switching to the OS handler. Instruction 6 must be killed (thus not commit) because it's after 5. If instructions 1, 2, 3, or 4 also raise an exception, the earliest instruction to have an exception should take precedent, and all later instructions should be flushed from the pipeline.

F) How can we always be able to execute loads and stores out of order before their addresses are known? What is the downside and how is it handled? Specifically, assume that we executed instruction 5 before instruction 4, but then realized that $|x6 - x3| < 4$.

We can speculatively assume that addresses of all loads and stores are non-overlapping and issue them before knowing their addresses. Once addresses become known, if we realize that we shouldn't have reordered some loads and stores, we have to terminate the ones we shouldn't have executed as well as any further instructions that depend on them. In the example, we have to terminate instruction 5 as well as 6 once we figure out that $|x6- x3| < 4$. Once instruction 4 completes, we then re-execute instructions 5 and 6.