CS152 Computer Architecture and Engineering
Complex Pipelines, Out-of-Order Execution, and Speculation
Problem Set #3 (v1.1)

Assigned 2/24/2022 *Due March 8*

https://inst.eecs.berkeley.edu/~cs152/sp22

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

Assignments must be submitted through **Gradescope** by **11:59pm PST** on the specified due date. Refer to Piazza for the entry code to join the CS152 Gradescope. Late submissions will not be accepted, except for extreme circumstances and with prior arrangement.

Name:

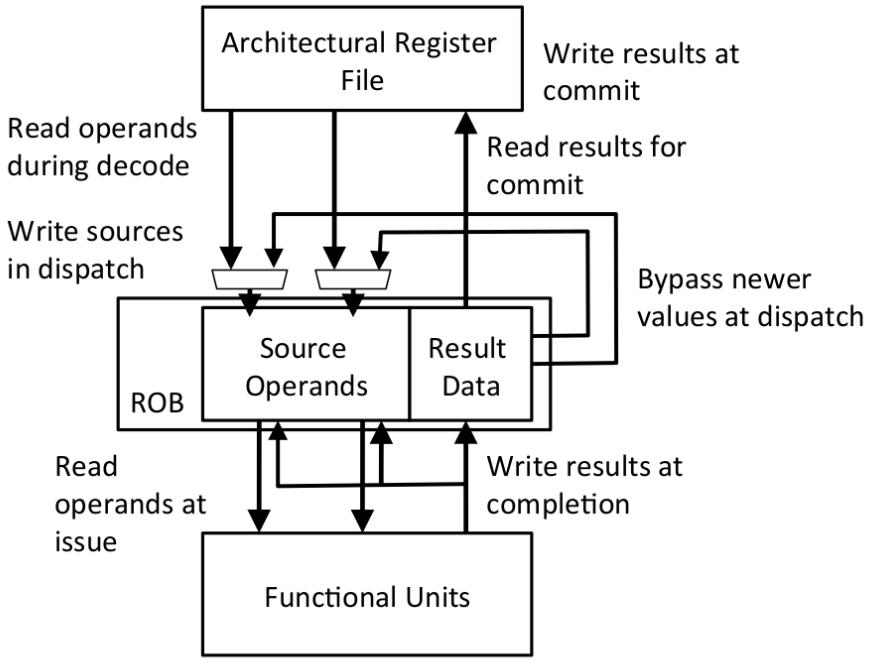
SID:

Problem 1: Out-of-Order Scheduling

This problem deals with an out-of-order single-issue processor that is based on the basic RISC-V pipeline and a floating-point unit. The FPU has one adder, one multiplier, and one load/store unit. The adder has a three-cycle latency and is fully pipelined. The multiplier has a six-cycle latency and is fully pipelined. Assume that stores take one cycle and loads take two cycles.

There are 31 **writable** integer registers (x1-x32) and 32 floating-point registers (f0-f31). 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 CPU uses a data-in-ROB design, so there is one rename register associated with each ROB entry. Functional units write back to the ROB upon completion. The functional units share a single write port to the ROB. In the case of a write-back conflict, the older instruction writes back first. 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.

Floating-point instructions (including loads writing floating-point registers) must spend one cycle in the writeback stage before their result can be used. Integer results are available for bypass the next cycle after issue and write back two cycles after issue.



For the following questions, we will evaluate the performance of the code segment below.

I_1	`fld f1, 0(x1)`
I_2	`fadd.d f2, f0, f1`
I_3	`fmul.d f3, f0, f2`
I_4	`addi x1, x1, 8`
I_5	`fld f1, 0(x1)`
I_6	`fadd.d f2, f1, f2`
I_7	`fmul.d f2, f2, f3`

A) For this part, consider an ideal case where we have an unlimited number of ROB entries.

In the table below, fill in the cycle number for 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 for each register that is written (P0, P1, …). Keep in mind that after a register has been renamed, subsequent instructions that refer to that register must refer to the new register name.

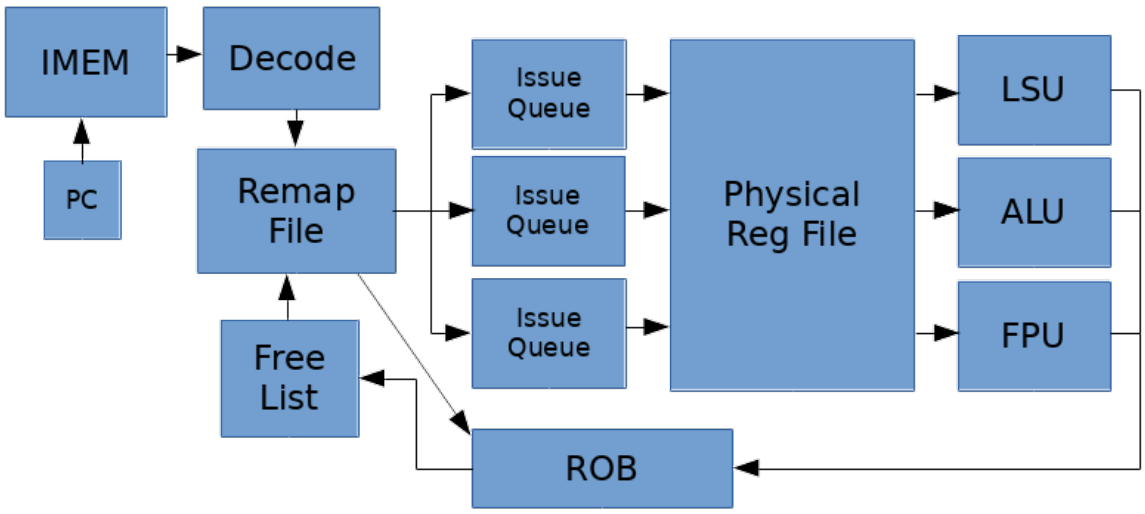
	Time				OP	Dest	Src1	Src2
	Enter ROB	Issue	WB	Commit				
I_1	-1	0	2	3	`fld`	P0	x1	-
I_2	0	3	6	7	`fadd.d`	P1	f0	P0
I_3	1				`fmul.d`			
I_4					`addi`			
I_5					`fld`			
I_6					`fadd.d`			
I_7					`fmul.d`			

B) For this part, consider a more realistic system with a four-entry ROB. An ROB entry can be used one cycle after the instruction using it commits. Fill in the table as you did in part A. If the instruction uses a source register that has already been retired, use the architectural name of the register.

	Time				OP	Dest	Src1	Src2
	Enter ROB	Issue	WB	Commit				
I_1	-1	0	2	3	`fld`	P0	x1	-
I_2	0	3	6	7	`fadd.d`	P1	f0	P0
I_3	1				`fmul.d`			
I_4					`addi`			
I_5					`fld`			
I_6					`fadd.d`			
I_7					`fmul.d`			

Problem 2: 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 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	P9
tail ->	0x8000101C	x2	P8
head ->	0x80001010	x6	P5
	0x80001014	x2	P11

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	
x1	P6	
x2	P2	
x3	P10	
x4	P7	
x5	P4	
x6	P13	
x7	P15	

B) When can a physical register be released and 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?

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		

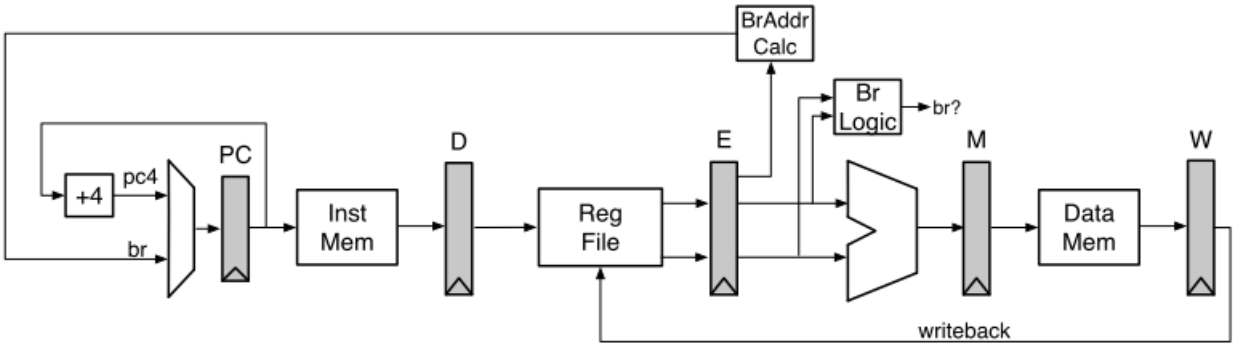
For the following instruction sequence, indicate which physical register gets assigned as the destination register and which physical register gets added to the free list on commit.

Instruction	Destination Register	Freed Register
`fld f2, 0(x3)`		
`fld f1, 0(x4)`		
`fmul.d f2, f2, f0`		
`fadd.d f1, f2, f1`		
`fsd f1, 0(x2)`		
`addi x4, x4, 8`		
`addi x3, x3, 8`		
`addi x2, x2, 8`		

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?

Problem 3: Pipelining with Branch Prediction

For this question, consider a fully bypassed 5-stage RISC-V processor. We have reproduced the pipeline diagram below (bypasses are not shown). Branches are resolved in the Execute Stage, and the Fetch Stage always speculates that the next PC is PC+4. For this problem, we will ignore unconditional jumps, and only concern ourselves with conditional branches.



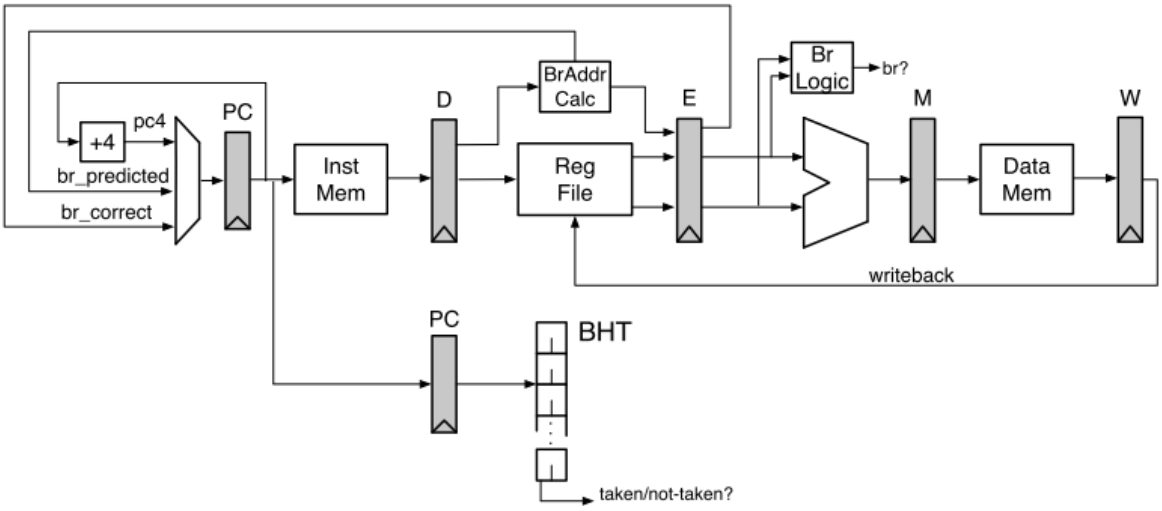
A) Fill in the following pipeline diagram using the code segment below. The first two instructions have been done for you.

Throughout this question, make sure you also show instructions that were speculated to be executed and then flushed (it would help to mark them explicitly) in the instruction/time diagrams, as they also consume pipeline resources.

```
0x2000: ori  x2, x0, -1
0x2004: addi x3, x0, -1
0x2008: beq  x2, x3, 0x2004
0x200c: lw   x5, 4(x6)
0x2010: xor  x7, x5, x7
0x2014: and  x5, x3, x7
0x2018: and  x3, x2, x3
```

PC	Instr	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13
0x2000	`ori`	F	D	X	M	W								
0x2004	`addi`		F	D	X	M	W							
0x2008	`beq`													

B) As you showed in the first parts of this question, branches in RISC-V can be expensive in a 5-stage pipeline. One way to help reduce this branch penalty is to add a Branch History Table (BHT) to the processor. This new proposed datapath is shown below:



The BHT has been added in the Decode Stage. The BHT is indexed by the PC register in the Decode Stage. Branch address calculation has been moved to the Decode Stage. This allows the processor to redirect the PC if the BHT predicts "Taken".

On a BHT mis-prediction, (1) the branch comparison logic in the Execute Stage detects mis-predicts, (2) kills the appropriates stages, and (3) starts the Instruction Fetch using the correct branch target (`br_correct`).

Remember the Fetch Stage is still predicting PC+4 every cycle, unless corrected by either the BHT in the Decode Stage (`br_predicted`) or by the branch logic in the Execute Stage (`br_correct`).

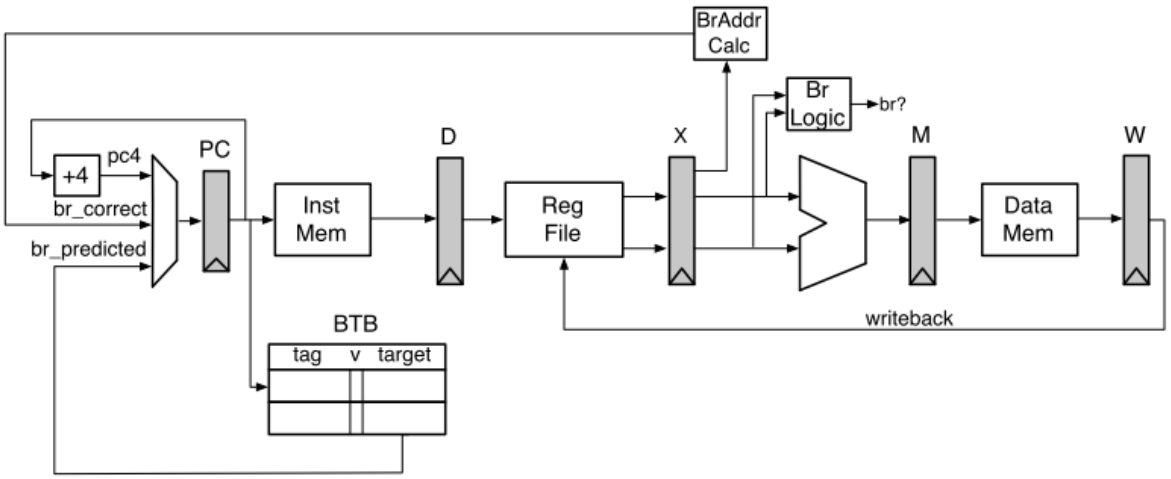
Using the code segment below, fill in the following pipeline diagram. Initially, the BHT counters are all initialized to "strongly-taken". The register x2 is initialized to 1, while the register x3 is initialized to 2. The first instruction has been done for you. It is okay if you do not use the entire table.

```
0x2000: lw   x7, 0(x6)
0x2004: addi x2, x2, 1
0x2008: bne  x2, x3, 0x2000
0x200c: sw   x7, 0(x6)
0x2010: add  x5, x5, x4
0x2014: sub  x7, x7, x5
```

PC	Instr	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13
0x2000	lw	F	D	X	M	W								
0x2004	addi		F	D	X	M	W							
0x2008	bne													

C) Unfortunately, while the BHT is an improvement, we still have to wait until we know the branch address to act on the BHT's prediction. We can solve this by using a two-entry Branch Target Buffer (BTB).

The new pipeline is shown below. For this question, we have removed the BHT and will only be using the BTB.



The BTB has been added in the Fetch Stage. The BTB is indexed by the PC register in the Fetch Stage. Branch address calculation has been moved back to the Execute Stage.

On a branch mis-prediction, (1) the branch comparison logic in the Execute Stage detects the mis-predict, (2) kills the appropriates stages, and (3) starts the Instruction Fetch using the correct branch target (`br_correct`).

Remember the Fetch Stage is still predicting PC+4 every cycle, unless either the BTB makes a prediction (has a matching and valid entry for the current PC) or the branch logic in the Execute Stage corrects for a branch mis-prediction (`br_correct`).

Using the code segment below (the exact same code from 4.B), fill in the following pipeline diagram. Upon entrance to this code segment, the register x2 is initialized to 1, while the register x3 is initialized to 2.

```
0x2000: lw   x7, 0(x6)
0x2004: addi x2, x2, 1
0x2008: bne  x2, x3, 0x2000
0x200c: sw   x7, 0(x6)
0x2010: add  x5, x5, x4
0x2014: and  x7, x5, x7
```

Initially, the BTB contains:

Tag	V	Target PC
0x2008	1	0x2000
0x201c	0	0x2010

(For simplicity, the tag is 32 bits, and we match the entire 32-bit PC register in the Decode Stage to verify a match). It is okay if you do not use the entire table.

PC	Instr	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13
0x2000	`lw`	F	D	X	M	W								
0x2004	`addi`		F	D	X	M	W							
0x2008	`bne`													

Problem 4: 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?

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?

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?

```
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
```

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

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?

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 $|x6 - x3| < 4$.

Problem 5: Branch Predictor Accuracy

For this problem, we are interested in the following code:

```
int array[N] = {…};
for (int i = 0; i < N; i++)
  if (array[i] != 0)
    array[i]++;
```

Using the disassembler, we get:

```
     li   a0, N
     la   a1, array
loop:
     lw   a2, 0(a1)
     beqz a2, endif
     addi a2, a2, 1
     sw   a2, 0(a1)
endif:
     addi a0, a0, -1
     addi a1, a1, 4
     bnez a0, loop
```

A) *Full BHT*

The processor that this code runs on uses a 512-entry branch history table (BHT), indexed by PC [10:2]. Each entry in the BHT contains a 2-bit counter, **initialized to the 01 state**.

Each 2-bit counter works as follows: the state of the 2-bit counter decides whether the branch is predicted taken or not taken, as shown in the table below. If the branch is actually taken, the counter is incremented (e.g., state 00 becomes state 01). If the branch is not taken, the counter is decremented. The counter saturates at 00 and 11 (a not-taken branch while in the 00 state keeps the 2-bit counter in the 00 state).

State	Prediction
00	Not taken
01	Not taken
10	Taken
11	Taken

If array = {1, 0, 0, -4, 0}, what is the prediction accuracy for the two branches found in the above code for five iterations of the loop, using the 512-entry BHT described above?

B) *Small BHT*

Now consider a BHT with only a single entry. That is, both branches will share the same counter. Now what will the prediction accuracy be for each branch? Assume we are using the same array, {1, 0, 0, -4, 0}.

C) *Static Hints*

For this question, assume that the compiler can specify statically which way the processor should predict the branch will go. If the processor sees a "branch-likely" hint from the compiler, it predicts the branch is taken and does NOT update the BHT with this branch (i.e., any branches the compiler can analyze do not pollute the BHT).

Which branches in the program, if any, can the compiler provides hints for? Assume the input array for the compiler's test runs varies widely and the compiler must be fairly confident in the accuracy of a static branch hint.