Computer Architecture and Engineering
CS152 Quiz #3

March 22nd, 2012

Professor Krste Asanović

Name:__<ANSWER KEY>__

This is a closed book, closed notes exam.

80 Minutes

10 Pages

Notes:

- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz. If you have inadvertently been exposed to a quiz prior to taking it, you must tell the instructor or TA.
- You will get **no** credit for selecting multiple-choice answers without giving explanations if the instructions ask you to explain your choice.

Writing name on each sheet	________	1 Points
Question 1	________	30 Points
Question 2	________	12 Points
Question 3	________	9 Points
Question 4	________	28 Points
TOTAL	________	**80 Points**

Name __________(answer key)________

Question 1: Scheduling for Dual Issue, Out-of-order Processors (30 points)

The following question concerns the scheduling of floating-point code on a **dual issue** out-of-order processor. For this problem, we will deal with a processor that uses a split instruction window/ROB design, as shown in Figure 1.

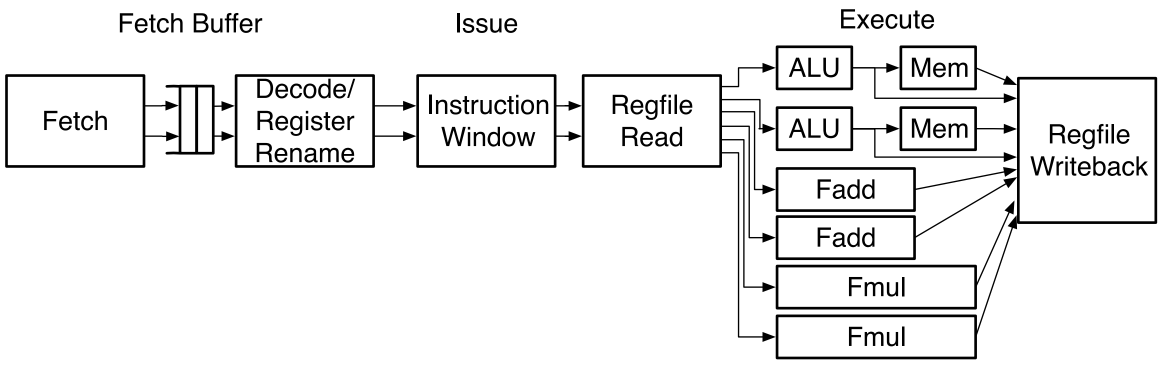

Figure 1. Dual Issue, Out-of-order Pipeline with split instruction window and ROB.

The processor contains the following stages:

- *Fetch* (F), *Decode/Rename* (D), *Issue* (I), *Regfile Read* (R), *Execute*(X1,X2,...), and *Regfile Writeback* (W)
- The *Execute* stage takes a variable number of cycles, depending on the instruction:
  - **one** cycle for ALU and branch operations (denoted as X1)
  - **two** cycles for memory operations (X1,X2, which includes the time in the ALU)
  - **three** cycles for floating-point add instructions (X1,X2,X3)
  - **four** cycles for floating-point multiply instructions (X1,X2,X3,X4)

You can assume that:

- All functional units are fully pipelined.
- There **is register renaming**.
- There are two register domains: a set for integer registers (`x1`,`x2`,...) and a set for floating-point registers (`f1`, `f2`, ...).
- The *Fetch* stage performs **perfect** branch prediction, and the fetch buffer can hold an infinite number of instructions.
- The *Issue* stage is a buffer of unlimited length that holds instructions waiting to begin execution (aka, the instruction window).
- An instruction will only exit the *Issue* stage if it does not cause any hazards and its operands will be ready by *Register Read* stage.
- **Two** instructions are fetched at a time.
- **Two** instructions are decoded and renamed at a time.
- **Up to two** instructions of any kind can be issued at a time, and if multiple instructions are ready, the **oldest** two will go first.
- An infinite number of instructions may write back to the register file simultaneously.

assumptions continued:

- There is no bypassing between functional units. All operand data is read from the register file, but the register file bypasses write values to the read ports.
- Store data is not needed until for address calculation and can be bypassed from the register file directly to the end of the (X1) stage.
- For the purposes of this question, treat stores as a single instruction that issues when both of its operands are ready (as opposed to the Lab 3 BOOM processor, which breaks stores into two separate micro-ops).

For this problem we will be describing the scheduling of the following RISC-V code:

```
Loop:
I0: flw    f1, 0(x1)
I1: flw    f2, 0(x2)
I2: fadd.s f3, f2, f3
I3: fmul.s f1, f1, f3
I4: fsw    f1, 0(x3)
I5: addi   x1, x1, 4
I6: addi   x2, x2, 4
I7: addi   x3, x3, 4
I8: addi   x4, x4, -1
I9: bne    x4, x0, loop
```

Instructions postfixed (*.s) denote instructions that affect single-precision floating point numbers. Assume the arrays accessed in this loop do not overlap.

Q1.A: Dual Issue Scheduling (28 points)

Complete Table 1 (found on the following page), indicating which stage each instruction is in for each cycle. Assume all register values are available at the start of the execution of the code and that the loop is taken.

The first three rows have been completed for you, and the fourth row has been started. It is okay if your instructions run off the right side of the table.

Q1.B: Dual Issue CPI (2 points)

For a sufficiently large number of iterations of the above loop, what CPI do you expect to achieve?

Write back on stores occurs on cycle 18 and 23, branches on 12 and 17, so 5 cycles for 10 instructions comes out to be 0.5 CPI.

0 points for saying "total cycles over total instructions."

CPI_0.5__

```
Loop:
I0: flw    f1, 0(x1)
I1: flw    f2, 0(x2)
I2: fadd.s f3, f2, f3
I3: fmul.s f1, f1, f3
I4: fsw    f1, 0(x3)
I5: addi   x1, x1, 4
I6: addi   x2, x2, 4
I7: addi   x3, x3, 4
I8: addi   x4, x4, -1
I9: bne    x4, x0, loop
```

Name __________(answer key)________

	Cycle Inst	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
flw	$I_{0,\text{iter }0}$	F	D	I	R	X1	X2	W																
flw	$I_{1,\text{iter }0}$	F	D	I	R	X1	X2	W																
fadd	$I_{2,\text{iter }0}$		F	D	I	I	I	R	X1	X2	X3	W												
fmu	$I_{3,\text{iter }0}$		F	D	I	I	I	I	I	I	I	R	X1	X2	X3	X4	W							
fsw	$I_{4,\text{iter }0}$			F	D	I	I	I	I	I	I	I	I	I	I	R	X1	X2	W					
addi	$I_{5,\text{iter }0}$			F	D	I	R	X1	W															
addi	$I_{6,\text{iter }0}$				F	D	I	R	X1	W														
addi	$I_{7,\text{iter }0}$				F	D	I	I	R	X1	W													
addi	$I_{8,\text{iter }0}$					F	D	I	R	X1	W													
bne	$I_{9,\text{iter }0}$					F	D	I	I	I	R	X1	W											
flw	$\mathbf{I_{0,\text{iter }1}}$						**F**	**D**	**I**	**R**	**X1**	**X2**	**W**											
flw	$I_{1,\text{iter }1}$						F	D	I	R	X1	X2	W											
fadd	$I_{2,\text{iter }1}$							F	D	I	I	I	R	X1	X2	X3	W							
fmu	$I_{3,\text{iter }1}$							F	D	I	I	I	I	I	I	I	R	X1	X2	X3	X4	W		
fsw	$I_{4,\text{iter }1}$								F	D	I	I	I	I	I	I	I	I	I	I	R	X1	X2	W
addi	$I_{5,\text{iter }1}$								F	D	I	R	X1	W										
addi	$I_{6,\text{iter }1}$									F	D	I	R	X1	W									
addi	$I_{7,\text{iter }1}$									F	D	I	I	R	X1	W								
addi	$I_{8,\text{iter }1}$										F	D	I	R	X1	W								
bne	$I_{9,\text{iter }1}$										F	D	I	I	I	R	X1	W						

Quite a few times more than 2 instructions want to issue! Notice that floating point is completely removed from critical path, and issue bandwidth is what this machine craves.

17 instructions, 28 points
-2 points for early issue
-2 for issuing 3+ instructions in same cycle
-2 for store not being bypassed (-1 second time mistake was made)
-3 points for store going too early (need to wait for fmul)

(generally, -2 for each incorrect instruction)

Name __________(answer key)________

Question 2: Out-of-order Machine Design (12 points)

An out-of-order superscalar processor uses a unified physical register file for register renaming and also separates the reorder buffer from the instruction window. During decode, instructions are allocated a slot in the reorder buffer, have their registers renamed, and then are placed in the instruction window to await issue into execution.

Each question was worth +4 points.

Part A) Should the number of **reorder buffer entries** be greater than, equal to, or less than the number of **instruction window entries**? Explain.

GREATER THAN

the ROB is tracking all in-flight instructions, whereas the instruction window is holding only instructions that have been decoded/renamed but not issued.

Part B) Should the number of **reorder buffer entries** be greater than, equal to, or less than the number of **physical registers**? Explain.

GREATER THAN

the ROB has an entry for all in-flight instructions, but not all instructions write to a destination (stores, branches), thus not all entries need a physical register allocated for them, so you can get away with having fewer physical registers than ROB entries.

Part C) Should the number of **instruction window entries** be greater than, equal to, or less than the number of **physical registers**? Explain.

LESS THAN

Since this is a unified physical register file design, the instruction window is holding instructions that have been decoded/renamed but not issued. However, nearly all instructions moving through the machine need a physical register to write to, so you would need fewer IW entries as you would need physical registers.

Question 3: Reclaiming Physical Registers (9 points)

In class, we stated that for a machine with a unified physical register file, we reclaim a physical register when the next writer of the same architectural register commits.

Q3.A: (4 points)

Describe what could go wrong if we instead reclaimed a physical register as soon as the writer commits. Write a short assembly code sequence and describe the events that would lead to incorrect execution.

Basically, the problem is we might reuse the physical register before all of the readers have been able to read out the correct value.

There are a couple ways this problem could manifest itself. Here's one simple example:

```
ADD x1, x0, x0                 -> ADD P1, r0, r0
.... < lots of instructions>
SUB x2, x1, x3                 -> SUB P2, P1, ...
```

or a more exact example:

```
ADD x1, x0, x0                 -> ADD P1, x0, x0
LW    x3, ....                 -> LW P3 (cache misses, add commits before the sub reads P1)
MUL x4, x0, x0                 -> MUL P1, x0, x0 (now that P1 has been freed, mul can have
                                      P1 allocated to it, and SUB could get the wrong value).
SUB x2, x1, x3                 -> SUB P2, P1, P3
```

From the above example, we can see that if the Add instruction (which is allocated physical register P1) puts P1 back on the free list once the Add commits, any instruction which reads P1 after the Add has committed could read the wrong value (especially if P1 was given away to a new instruction, because it had been added to the Free List).

Instead, P1 must stay around until the LAST reader commits (usually though, we wait for a new instruction to write to that same ISA destination).

Q3.B: (5 points)

Suppose we invented a scheme where we could reclaim a physical register as soon as the last reader of the value in the register was committed. We can identify the last reader by making a change to the ISA and having the compiler mark when a source register is last used. Would this work? If you answer yes, explain your reasoning. If you answer no, provide an example with a short code sequence to explain why it wouldn't work.

The answer depends on how you argued it: it could either be "yes" or "no" depending on your assumptions and point of view. We were only looking for *one* reason.

Here is a reason for Yes:

- The *soonest* you can free a register is after it is last read. Thus, if the compiler does a smart enough job, this scheme will actually work.

Note that compilers already perform their own register renaming, and already understand "liveness" of different registers, even across basic blocks. Many students tried to argue that a compiler wasn't capable of performing this scheme, but it actually already does!

Here are some reasons for No:

- On interrupts, we may want to read out the entire machine state, but if a register has been freed, we can no longer guarantee its value. Thus, an attempt at looking at the "ISA State" will involve many registers being in the "don't care" state.

- Malicious code could degrade the system by not having any "free" instructions in it. Or a compiler bug could hose the system by not performing a "free" in the right place. The ISA architect should "architect defensively" to make sure his processor isn't dead-on-arrival if a virus-writer wants to exploit an ISA design decision.

Question 4: Pipeline Widths (28 points)

We repeat the pipeline figure from lecture 11 below, which shows the major components of an out-of-order processor.

Because the major sections of the design (fetch, decode, execute, commit) are separated by buffers, we can independently change the peak instructions/cycle throughput of each section. In the following questions, we will increase the throughput of one section while keeping the others constant. Explain in each case a scenario where this might be advantageous, or where it might hurt performance if any? Assume the buffers between stages are large enough to not impact performance. *Hint: remember to consider that some stages may have to stall while waiting for resources to free up.*

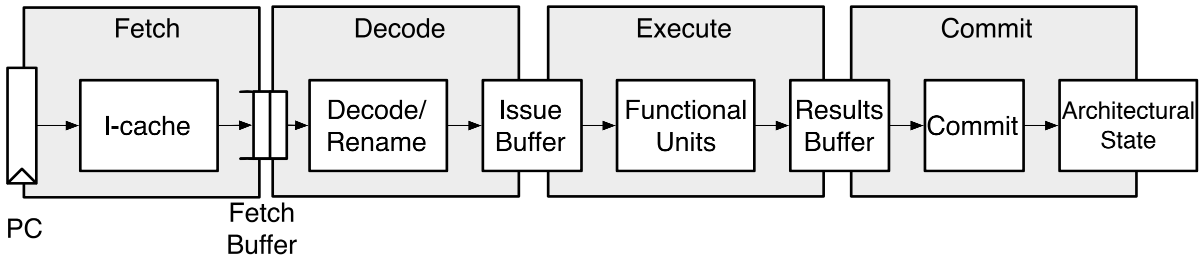

Figure 2. The major components of an out-of-order pipeline.

Yes, this question actually has 32 points allocated to it... so consider 4 points as extra credit!

Q4.A: Wider Fetch Stage Advantages (4 points)

If we double the fetch bandwidth relative to other sections (i.e., the pipeline can fetch twice as many instructions per cycle as it can decode), in what execution scenario might this be advantageous?

If branches were perfectly predicted and exceptions are rare and there are no I-cache misses, there would be little benefit to increasing fetch bandwidth as decode would be the bottleneck.

However, after an I-cache, branch mispredict or exception, we need to refill the fetch buffer and we can run ahead to prefetch I-cache misses, even if the decode stage is stalled.

So one scenario is an I-cache miss or branch mispredict, where the target code causes an I-cache miss.

+2 for "look ahead" without explanation or wrong explanation.
+2 for "fetch both sides of branch" - this requires more drastic changes than just widening fetch!

Q4.B: Wider Fetch Stage Disadvantages (4 points)

Describe a case where doubling the fetch bandwidth might make performance worse, or explain why it can never make performance worse?

If control flow is perfectly predicted, then no disadvantage (but then also no advantage!).

If control flow is mispredicted, then we could fetch ahead on the wrong path or cause I-cache misses that evict correct path instructions or cause bandwidth congestion in the memory system!

+2 if got something with branch mispredicts but not good explanation why.

Q4.C: Wider Decode Stage Advantages (4 points)

If we double the decode stage bandwidth compared to the fetch, execute, and commit stages, is there any execution scenario where this could improve performance? Explain.

Generally, the decode stage throughput will be constrained by either the fetch bandwidth or by the rate at which instructions commit and free up resources.

Even after a branch mispredict, decode will be constrained by fetch bandwidth along the correct path.

Wider decode would only help performance if fetch buffer had filled up because decode was previously stalled waiting for resources in the instruction window, ROB, or LDQ. But these are freed up only at the rate set by the Execute or Commit stages, so wider decode doesn't help!

Q4.D: Wider Decode Stage Disadvantages (4 points)

Is there any execution scenario where this might reduce performance? Explain.

As it cannot improve performance, it also cannot make matters worse by moving down a wrong path faster.

Q4.E: Wider Execute Stage Advantages (4 points)

If we double the execute stage bandwidth relative to the other stages is there any execution scenario where this could improve performance? Explain?

Yes, if we have many instructions waiting on one instruction, then a wider execute can help the machine move faster through sections with higher degrees of ILP.

Name __________(answer key)________

Q4.F: Wider Execute Stage Disadvantages (4 points)
Is there any execution scenario where this might reduce performance? Explain.

By executing some code farther, it might allow further speculation down an incorrect path, so it could cause extra wrong path cache misses and hence slow overall code performance down.

Q4.G: Wider Commit Stage Advantages (4 points)
If we double the commit stage bandwidth relative to the other stages is there any execution scenario where this could improve performance? Explain?

Yes, can free up resources faster after a critical instruction to allow the front-end to run ahead, especially code with a bursty commit pattern.

Also, different instructions use different resources (LSQ entries, branch tags, etc.), so the front-end might be blocked waiting on something like a memory re-order slot.

```
Example ROB
is entry busy? (0 = busy, 1 = not busy)
0 <- long latency memory  <<--- commit point is stuck here until load data comes back
1
1      (a bunch of ready to commit instructions)
1
1
0 <- long latency memory     <<--- decode is stalled waiting for this memory slot to free up
1
1
1
```

+4 if talked about needing to commit stores without load bypassing, even if this isn't a typical design point (Lab 3's BOOM is atypical, in this regard!).

Q4.H: Wider Commit Stage Disadvantages (4 points)
Is there any execution scenario where this might reduce performance? Explain.

If it allows faster progress down the wrong path, overall code performance could be slower! (like Q4.F).

END OF QUIZ