Name _____

Computer Architecture and Engineering
CS152 Quiz #3
March 28th, 2016
Professor George Michelogiannakis

Name: _____<ANSWER KEY>_____

This is a closed book, closed notes exam.
80 Minutes
13 pages

Notes:

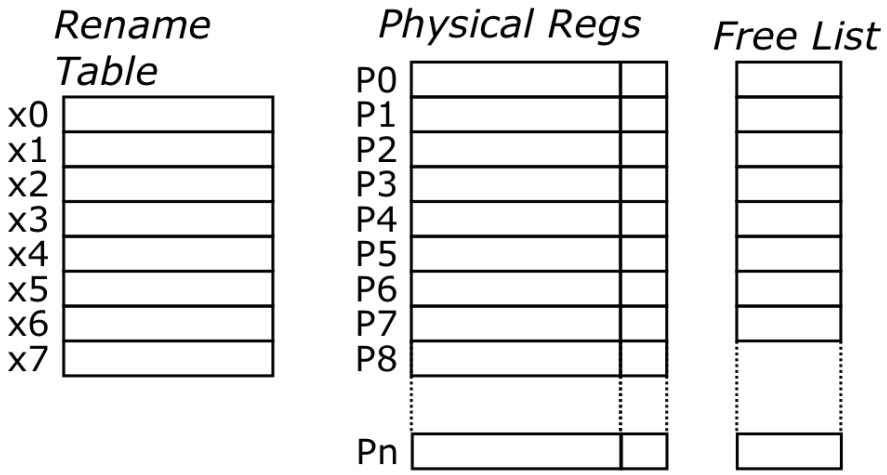
- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not taken the quiz. If 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 Point
Question 1	_____	30 Points
Question 2	_____	32 Points
Question 3	_____	21 Points
Question 4	_____	16 Points
TOTAL		

Name _____

Question 1: Register Renaming [30 points]

For this question, we will use the following code to understand how out-of-order processors behave. We will assume an architecture with a unified register file that both floating point and integer ISA registers map to ("x" and "f" registers). This architecture also has a rename table, a reorder buffer, and a free list. As a reminder, you can find the block diagram below. The rename table maps both integer and floating point registers in the same way. There are 16 physical registers (n=16). "LPRd" refers to the old label of the destination register, and "PRd" the new one. You can assume the branch is not taken, and that we issue one instruction per cycle unless the question says otherwise. "Use" refers to whether that entry in the ROB is valid, and "ex" whether that instruction is executing.



ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

```
loop:
  (1) faddi f1, f5, 5
  (2) sub   x6, x5, x4
  (3) flw   f1, 0(x6)
  (4) fmul  f3, f1, f4
  (5) fmul  f9, f9, f8
  (6) fadd  f4, f2, f3
  (7) fcvt  x6, f3   // Convert f3 to an integer and store in x6
  (8) fsw   f4, 0(x6)
  (9) bne   x7, x0, loop // Assume not taken
  (10)faddi f4, f4, 8
  (11)faddi f6, f6, 8
```

Name _____

Q1.A Hazards [4 points]

How many RAW hazards are there in the above segment? Write your answer in the form of "(I) -> (J)" to show a dependency between instruction I and J (J reads, I writes).

(2)->(3), (3)->(4), (4)->(6), (7)->(8), (4)->(7), (6)->(10), (6)->(8)
There is no RAW hazard between instruction 8 and 10 because stores don't write to a register.

Q1.B Register Renaming and Data Hazards [6 points]
Show how the processor would perform register renaming in the given code sequence such as to maximize performance by minimizing stalls. Denote renamed registers with P starting with index 1 (i.e., P1). Non-renamed registers can keep their ISA register name. What kinds of hazards does register renaming help with? Explain.

Register renaming eliminates WAW and WAR dependencies that would otherwise cause stalls.

```
loop:
  (1) fadd  P1, f5, 5
  (2) sub   P2, x5, x4
  (3) flw   P3, 0(P2)
  (4) fmul  P4, P3, f4
  (5) fmul  P5, f9, f8
  (6) fadd  P6, f2, P4
  (7) fcvt  P7, P4
  (8) fsw   P6, 0(P7)
  (9) bne   x7, x0, loop
  (10)faddi P8, P6, 8
  (11)faddi P9, f6, 8
```

Name _____

Q1.C Register renaming [10 points]

Complete the following table, assuming that both `x` registers and `f` registers are renamed from the same pool of unified physical registers. The initial map table and free list is given below. Assume that the free list has 16 slots, is organized as a FIFO – a physical register is popped off the top of list when allocated, and a physical register is added back to the bottom of the list when reclaimed. Register x0 is tied to 0 and does not get renamed. For each instruction, label the following:

- which physical register gets assigned to the instruction as a destination
- upon commit, which physical register gets added back to the free list.

If a register wasn't previously mapped, you can leave "freed register" empty.

Architectural Register	Physical Register
f2	P11
f4	P14
f5	P5
f6	P12
f8	P7
f9	P16
x4	P3
x5	P15
x7	P13

Free List
P1
P2
P8
P4
P10
P6
P9

Instruction	ISA Destination Register	Physical Destination Register	Freed Register
`fadd f1, f5, 5`	f1	P1	
`sub x6, x5, x4`	x6	P2	
`flw f1, 0(x6)`	f1	P8	P1
`fmul f3, f1, f4`	f3	P4	
`fmul f9, f9, f8`	f9	P10	P16
`fadd f4, f2, f3`	f4	P6	P14
`fcvt x6, f3`	x6	P9	P2
`fsw f4, 0(x6)`	N/A	N/A	N/A
`bne x7, x0, loop`	N/A	N/A	N/A
`faddi f4, f4, 8`	f4	P1	P6
`faddi f6, f6, 8`	f6	P16	P12

Q1.D Number of Architectural Registers [6 points]

For an ISA with 32 integer registers and 32 floating point registers, how many architectural registers would you provide and why? Argue for what is a reasonable number of architectural registers such that the number of architectural registers will not be a bottleneck, while not providing unnecessarily

many architectural registers. Choose among these options (#PRs = number of physical registers, #ARs = number of architectural registers):

#PRs = (½) * #ARs #PRs = #ARs **#PRs = 2 * #ARs**

What can happen if there are not enough physical registers?

If there aren't enough physical registers, this will manifest in an instruction accessing the free list to acquire a new label and failing to find a new one. This means that the instruction won't be able to issue even though functional units may be free and dependencies (hazards) may not exist.

As we saw in the previous code sequence, a sequence of instructions may continuously operate in a few architectural registers, but new physical registers need to be continuously allocated. This can keep going on a large number of registers. Therefore, we need more physical registers than architectural registers.

Q1.E Superscalar Register Renaming [4 points]

Now lets assume that we want to apply register renaming in a processor that issues two instructions per cycle instead of one. Do we need to do anything differently in how we assign physical registers to instructions, or does the methodology you used so far in this question suffice?

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 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.

Name _____

Question 2: Execution Progress and Branch Prediction [32 points]

For this question, we will examine the progress of execution of the following code:

loop:

```
(1)add   x1, x1, x2
(2)sw    x5, (x2)
(3)addi  x6, x6, x6
(4)beq   x1, x2, loop
(5)sw    x7, (x2)
(6)fadd  f8, f8, f8
(7)beq   x8, x7, loop
(8)lw    x9, (x2)
```

To begin with, we will use a processor that has the following ROB structure, and issues one instruction per cycle. Instructions get written into the ROB at decode time (at the end of the decode cycle the instruction is in the ROB). Instructions write their output in the physical register file when the data is generated, and other instructions directly get that data from the physical register file when needed. The ROB does not contain a "data" field. Stores calculate their address as an integer operation. Branches are also considered integer operations and take two cycles to execute. You can assume integer operations take two cycles to execute, and floating point four cycles. You can also assume one integer and one floating point functional unit, both fully pipelined that latch their operands on their first pipeline stage. "Use" means that that corresponding ROB entry is valid and "exec" means the instruction is executing. "Pd" is 1 when the instruction completes.

Ins #	Use?	Exec?	Op	P1	Src1	P2	Src2	Pd	Dest

Q2.A Register Renaming [5 points]

As described so far, this processor does not perform any register renaming. Is performance going to be affected for this code segment compared to using register renaming? Explain

Yes because there are WAR and WAW hazards that without register renaming will cause stalls. Also, there are potential hazards across iterations of the same loop

Name _____

Q2.B Branch Prediction Effect [4 points]
Lets initially assume there is no branch prediction. What impact does this have on this code segment and why? How many cycles penalty for each branch misprediction at minimum?

No branch prediction means that by the time we issue a branch, we have to wait for it to complete before issuing the next instruction. So there will be a bubble after each branch that the processor with perfect branch prediction will avoid. So a minimum of two cycle penalty per branch.

Q2.C Branch Prediction [8 points]
Now we want to add branch prediction in this processor and keep issuing and executing instructions while branches execute. Lets assume that we design a simple predictor that always predicts that a branch is not taken. A branch resolves and figures out if it was mispeculated when it would commit and leave the ROB. First describe how to handle mispeculations in this processor. Then show the state of the ROB as soon as the first branch of our code (number 4) completes and the processor realizes that it mispeculated, and what it should do to recover.

When a branch is predicted, instructions below it still get issued. If the branch completes and we realize it was mispredicted, all instructions below it need to be erased from the ROB and their data not allowed to change architectural state (i.e., their instructions must not commit).

Ins #	Use?	Exec?	Op	P1	Src1	P2	Src2	Pd	Dest
4	1	1	beq	1	x1	1	x2	1	
5	1	1	sw	1	x2	1	x7	0	
6	1	1	fadd	1	f8	1	f8	0	
7	1	0	beq	1	x8	1	x7	0	

All instructions before the branch would have committed by the cycle the branch completes execution. The branch takes two full execution cycles to realize it is not taken. Therefore, two instructions after it are executing (5 and 6), and one (7) has been issued but didn't get the chance to execute yet.

Name _____

Q2.D Functional Units [3 points]
What would change in the above ROB state if the processor had a single functional unit for both integer and floating point operations and that unit has four pipeline stages?

We would have more instructions in the ROB because the branch takes longer to complete, so instructions 7 and 8 will be in the ROB in the table of the previous question.

Q2.E Mispeculation with Stores [8 points]
Now lets assume we want stores to commit out of program order. In other words, we now let stores commit even if there are uncommitted instructions in the ROB that come before them in program order.
Does this create any issue with branch prediction? If so, propose a way to solve it and describe how your solution would work, especially with subsequent loads.

The problem now becomes that uncommitted stores may later need to be flushed because of a misspeculated branch, but by that time they may already have changed memory contents. One solution is to add a speculative store buffer that holds data of speculatively-executed stores. When the store commits it sends its data to the buffer. When that store no longer is speculatively-executed (previous branches complete execution), data in the speculative store buffer are no longer marked as speculative, and can be sent to memory if there is no older store also in the store buffer. Loads first have to get their data from the speculative store buffer if they can, otherwise from memory.

Name _____

Q2.F Precise Exceptions [4 points]

Assume that the first branch (instruction number 4) generates an exception when it is decoded (e.g., bad operand). Will that exception be precise? In 1-3 sentences, how can you modify this processor to provide precise exceptions?

That exception will not be precise because the previous instruction (number 3) will be executing and thus won't get the change to write its result. We can make the exception wait before being serviced until all previous instructions have completed and committed.

Name _____

Question 3: Loads and Stores, Exceptions [21 points]

For this question, we will consider this code:

```
(1)add   x1, x1, x2
(2)sw    x5, (x2)
(3)lw    x6, (x8)
(4)sw    x5, (x6)
(5)lw    x9, (x3)
(6)add   x9, x9, x9
```

Q3.A Reordering [4 points]
Suppose that we want to allow out of order load and store execution. Under what circumstances in the code above can we execute instruction 5 before executing any others? Explain

The address of instruction 5 (x3) must be different than the previous two stores. So x3 != x6 and x3 != x2

Q3.B Reordering Continued [4 points]
Same question as Q3.A (above), but for instruction 4. Why do loads impose or do not impose a reordering constraint?

There are two factors here. First, there is the store on instruction (2). So x6 != x2. However, there is also a RAW hazard that prevents instruction 4 from executing before 3. Loads do not change values of memory locations and thus do not impose constraints on the order of stores or other loads. In other words, loads impose constraints only for RAW hazards.

Name _____

Q3.C Out of Order Loads and Store [6 points]
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

We can speculatively assume that addresses of all loads and stores are different and issue them before knowing their addresses. Once addresses become known, if we realize that we shouldn't had 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. Once instruction 4 completes, we then re-execute instructions 5 and 6.

Q3.D Exceptions [7 points]
Now lets 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.

Name _____

Question 4: Potpourri [16 points]

Q4.A [4 points] Do you think exceptions cause more of a performance penalty in out-of-order or in-order processors and why?

Out of order because they switch to the OS handler which causes the ROB to be clear of the program's instructions. Since the benefit of out-of-order execution depends on having enough candidate instructions in the ROB to choose from and instruction issue can take a few cycles to re-fill the ROB with adequate candidates to use all functional units, out-of-order processors take a larger performance hit.

Q4.B [4 points] Does a cache miss cause a larger performance penalty for an in-order processor or an out-of-order processor?

In-order processor because it has no choice but to wait for the cache to be refilled. An out-of-order processor can execute other instructions as long as it has space in its ROB.

Name _____

Q4.C [4 points] What is the challenge in a superscalar out-of-order processor that fetches four instructions in one cycle, if the program contains a taken branch every two instructions?

In this case, it will fetch two branches every cycle. This requires the capability for two branch predictions per cycle, as well as fetching from two different instruction addresses per cycle.

Q4.D [4 points] Suppose we bypass load values from the speculative store buffer. Suppose that the load address hits three times in the store buffer for the following stores in program order (note that the load is younger than a and b but older than c):

(a) oldest non-speculative store
(b) middle speculative store (its execution depends on a branch that was predicted and not yet completed)
(c) youngest speculative store that is younger than the load

Which store's value should the load access? Is it possible for (c) to be non-speculative while (b) is still speculative?

It should return the youngest store's data that is older than the load. In this case, b. (c) cannot be speculative because the same branch that makes (b) speculative is also making (c) speculative.