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.