Computer System Architecture
6.823 Quiz #2
April 9th, 2021

Name: ________SOLUTIONS__________

90 Minutes
16 Pages

Notes:

- Not all questions are equally hard. Look over the whole quiz and budget your time carefully.
- Please state any assumptions you make, and show your work.
- Please write your answers by hand, on paper or a tablet.
- Please email all 16 pages of questions with your answers, including this cover page. Alternatively, you may email scans (or photographs) of separate sheets of paper. Emails should be sent to 6823-staff@csail.mit.edu
- Please ensure your name is written on every page you turn in.
- Do not discuss a quiz's contents with students who have not yet taken the quiz.
- Please sign the following statement before starting the quiz. If you are emailing separate sheets of paper, copy the statement onto the first page and sign it.

I certify that I will start and finish the quiz on time, and that
I will not give or receive unauthorized help on this quiz.

Sign here: ________________________________

Part A	________	30 Points
Part B	________	40 Points
Part C	________	30 Points
TOTAL	________	**100 Points**

Part A: Branch Prediction (30 points)

Ben Bitdiddle is trying to design a branch predictor for his processor. The C code is shown below:

```
for (int i = 0; i < 1000000; i++) {
      if (i % 2 == 0) { // branch B1
            // do something A
            ...
      }
      if (i % 5 == 0) { // branch B2
            // do something B
            ...
      }
}
```

The corresponding assembly code is shown below:

```
                  ADDI  R1, R0, 0
            LOOP: MODI  R2, R1, 2
0xc44:            BNE   R2, R0, M2        // branch B1
                  (do something A)
                  ...
            M2:   MODI  R2, R1, 5
0xc84:            BNE   R2, R0, END       // branch B2
                  (do something B)
                  ...

            END:  ADDI  R1, R1, 1
0xcc0:            BNE   R1, 1000000, LOOP     // branch LP
```

The MODI (modulo-immediate) instruction is defined as follows:
`MODI rd, rs, imm: rd <- rs mod imm`

Note that whenever the condition of the if-statement is met, we do **not** take the branch `BNE`.

Question 1 (3 points)

In steady state, how often are each of the three branches taken in the above code?

Keep in mind that when the if-condition is met, we do not take the BNE branch. Answers are in ratios.

B1: 1/2
B2: 4/5
LP:1

Question 2 (3 points)

Ben's first attempt is to design a static branch predictor. The static predictor predicts not taken for all forward branches, and taken for all backwards branches.

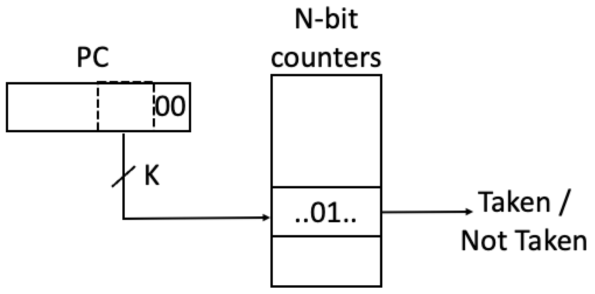
For each branch in Ben's code, what is the accuracy of this static predictor?

B1 and B2 are forward branches, and LP is a backwards branch.

B1: 1/2
B2: 1/5
LP: 1

Question 3 (9 points)

Ben tries to improve his prediction accuracy by designing a bimodal predictor.



The bimodal predictor consists of a table of **N**-bit counters. The predictor uses the **K** least significant bits of the PC in a word (not byte) address to index into the N-bit counters. The most significant bit of the counter is used to make a prediction for the branch (1 for Taken, 0 for Not Taken). If a branch is found to be taken, the counter is incremented by one (up to a maximum value of 2^N-1), and decremented by one (down to a minimum value of 0) otherwise. **Assume that all bits of the N-bit counters are initialized to zeros in the beginning.**

a) How many bits are needed so that the three branches (`B1`, `B2`, and `LP`) are mapped to distinct entries in the table of N-bit counters?

The three addresses are:
0xc44 = 0b1100 0100 0100
0xc84 = 0b1100 1000 0100
0xcc0 = 0b1100 1100 0000

Disregarding the least significant 2 bits, we need at least 5 bits to distinguish the three branches.

b) What is the minimum value of N required to achieve as good of an accuracy in steady state **for all branches** as the static predictor from Question 2?

Starting with 1 bit, we see that it will perform horribly for B1. The 1-bit predictor will always predict the previous branch direction, which is incorrect.

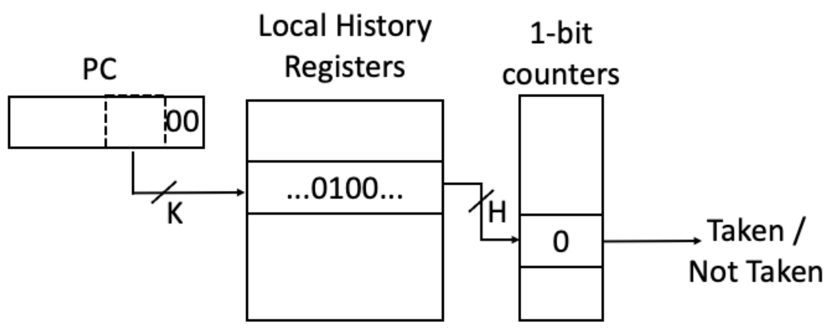
With 2 bits, the counters now have enough "stubbornness" to not immediately change its prediction upon mis-prediction.

c) Given your value of N, what is the accuracy of Ben's bimodal predictor for the three branches in steady state?
With 2 bits, we will have:

B1: 1/2 (will always predict not taken)
B2: 4/5 (will always predict taken)
LP: 1 (will always predict taken)

Question 4 (5 points)

Ben now considers a 2-level local history predictor design.



The **K** least significant bits of the PC in a word are now used to index into a table of local history registers. Each local history register stores **H** bits, and is used to keep track of the history of the PC. The H bits are used in turn to index into a table of 1-bit counters indicating the prediction (Taken if 1, Not taken if 0).

What is the minimum value of H such that all branches are predicted perfectly in steady state? Assume that the value of K is large enough that different branches are mapped to distinct local history registers.

Let's look at the patterns of each branch. We denote 1 as taken and 0 as not taken:

B1: 0101010101...
B2: 0111101111...
LP: 1111111111...

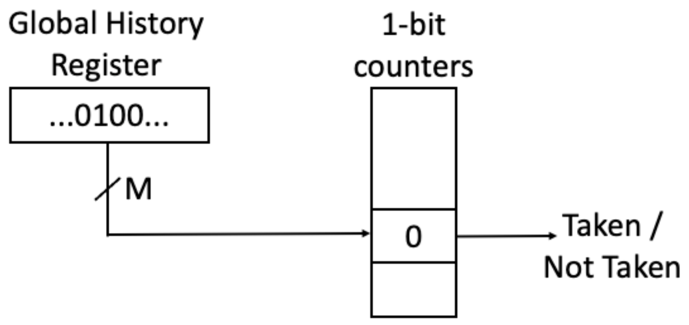
Note that we can safely disregard values of K from 1 to 4 since we need to predict not-taken after 4 taken branches for B2, whereas we need to predict taken for LP.

With 5 bits, we see that there can be no overlapping patterns between the three branches since the number of taken branches will differ for each branch (2 or 3 for B1, 4 for B2, and 5 for LP).

Thus, we need at least 5 bits to avoid aliasing.

Question 5 (5 points)

Ben now designs a global history predictor as follows.



The predictor consists of a single global history register of **M** bits that stores the outcomes of all branches in the program. Every time the processor encounters a branch, it shifts in a 1 to the history register from the right if the branch is taken, and a 0 if the branch is not taken. The global history register is used to index into a table of 1-bit counters indicating the prediction (Taken if 1, Not taken if 0).

Can this predictor achieve perfect prediction for all branches of Ben's code in steady state? Briefly explain your reasoning.

Notice from the previous question that there is a global pattern as well. If we take the LCM of the pattern lengths of B1, B2, and LP, we see that the same global pattern will repeat after 10 iterations. Since there are 3 branches per loop, 30 bits of global history is sufficient to capture all patterns and achieve perfect prediction.

Question 6 (5 points)

Ben is given the following piece of C code:

```
for (int i = 0; i < N; i++) {
     for (int j = 0; j < M; j++) { // branch LP
          ... // Do something
     }
}
```

Assume that M > 2. Ben wants to choose a branch predictor that maximizes the prediction accuracy for the inner loop (LP) of this code (assume that the outer loop is completely ignored by the branch predictor). Ben is given two choices: a bimodal predictor with 1-bit counters from Question 3, and a local history predictor from Question 4 with M-bit local histories. For what values of N will the local history predictor outperform the bimodal predictor? Assume that all structures for both predictors have their entries initialized to all zeros at the beginning of the outer loop.

Hint: First try to evaluate the two predictors with N = 1. What are their prediction accuracies?

Since LP has M iterations, it will repeatedly take M-1 taken branches and 1 not-taken branch.

First, let's look at the bimodal predictor. For every iteration of the outer loop, it will mis-predict the first taken branch and the last not-taken branch. Thus, its total prediction accuracy is:

BimodalAcc = (M-2)/M.

Now let's look at the local history predictor. Notice that the predictor has to go through 1 iteration of the outer loop just to fill in its M-bit history, during which it will always predict not taken. Then, it requires another iteration of the outer loop to train each of the M-bit patterns, again during which it always predicts not taken. Thus, the accuracy during the first 2 iterations is 2/(2*M)

After these 2 iterations, the local history predictor is fully trained and will predict all future branches perfectly. Thus, the overall accuracy is:

LocalHistAcc = (2 + (N-2)*M)/(M*N).

We need to solve LocalHistAcc > BimodalAcc for N, which gives us N > M+1.

Name ______________________________

Part B: Out-of-Order Processing (40 Points)

Question 1 (30 points)

This question uses the out-of-order machine described in the Quiz 2 Handout. We describe events that affect the initial state shown in the handout. Label each event with one of the actions listed in the handout. If you pick a label with a blank (_____), you also have to fill in the blank using the choices (i—vii) listed below. If you pick "R. Illegal action", state why it is an illegal action. If in doubt, state your assumptions.

Example: `I11` finishes and writes the result to the data field in its arithmetic reservation station entry.
Answer: (F): Write a speculative value using lazy data management.
(You can simply write "F".)

a) Instruction `I7` finishes, replaces `I10`'s tag `M1` with its result, and sets the `p1` bit.

(B, i): Satisfy a dependence on register value by bypassing a speculative value.

b) Assume Instruction `I7` finishes execution and commits. `I8` commits, and the speculative bit of entry 1 in the store buffer is cleared.

(P): Commit correctly speculated instruction, and mark lazily updated values as non-speculative.

c) Assume `A3` data becomes available. `I12` is issued and produces a segfault. Only entry `M3` and all following entries in the memory reservation station are cleared.

(R): The arithmetic reservation station entries that are younger than I12 must also be cleared.

d) Assume `A3` data becomes available and is found to be `1500`, and instruction `I12` encodes an offset of `0` (not shown in the handout). Instruction `I12` is issued, writes address `1500` into entry 2 of the load buffer, sets the corresponding valid bit, and reads the data from the matching store buffer entry 2.

(R): The store buffer entry 2 is a younger store than the load.

e) Assume `M3` data becomes available, `I14` is issued, and the branch is found to be predicted incorrectly and should have not been taken. The global history register is shifted to the right by one bit to recover the previous history.

(N): Abort speculative action and cleanup greedily managed values.

f) Assume Instruction `I17` is an `addi R5, R4, 1` instruction. `I17` enters the Register Rename and Allocation stage. It updates the rename table entry of `R5` to `A5`.

(G): Write speculative value using greedy value management

g) Assume `A3` data becomes available, and instruction `I16` encodes an offset of `0` (not shown in the handout). Instruction `I16` is issued and finds no matching loads in the load buffer.

(K, iv): Check the correctness of a speculation on memory address and find a correct speculation.

h) Assume Instruction `I19` is a branch instruction. When `I19` enters the decode stage, the branch predictor consults entry 7 and predicts the branch to be taken.

(E, iii): Satisfy a dependence on branch direction by speculation using a dynamic prediction.

i) Assume `M1` data becomes available. `I10` is issued, executes, and writes back its result to the `data` field of `A2` and sets the corresponding `pd` bit.

(R): The `A1` value is still not available, so the instruction cannot be issued.

j) Assume all instructions up to `I12` commit. `I13` commits, and overwrites register `R1` in the register file with the value from its `data` field.

(P): Commit correctly speculated instruction, and mark lazily updated values as non-speculative.

Question 2 (5 points)

Answer the following yes/no questions about our out-of-order machine with split reservation stations.

a) Do writes to the register file happen in program order? Yes

b) If a store finds a matching younger load in the load buffer, should the load and all following instructions be killed? Yes

c) Is the rename table always updated after an instruction commits? No

d) Are the 2-bit counters in the branch predictor updated lazily? Yes

e) Suppose our programs have 50% arithmetic and 50% memory instructions. Should both reservation stations be sized equally for maximum utilization? No (they can have different latencies)

Question 3 (5 points)

In our current implementation of reservation stations, at most one instruction is committed per cycle. Is there a way to provide higher commit throughput (commit more than one instruction/cycle) without making each of the reservation stations check more than one entry per cycle for commit? If so, briefly describe such a mechanism. If not, state your reasoning.

A simple implementation is to check the inum of the next-to-commit instructions in both reservation stations. If they have consecutive inums and are ready to commit, then we can safely commit both instructions. We can generalize this for N reservations stations as well.

Name ______________________________

Part C: Advanced Memory Operations (30 points)

For this part, please refer to the accompanying store sets Handout for a detailed explanation of how store sets work.

Consider the following C code:

```
int A[100];

for (int i = 0; i < 100; i++) {
     if (i % 2 == 0) {
          int j = long_latency_op1(); // Some long latency operation
          A[j] = x; // Store X
     }
     else {
          int j = long_latency_op2(); // Some long latency operation
          A[j] = y; // Store Y
     }
     int z = A[i]; // Load Z
     large_work(); // Large amount of work
}
```

Note that Store X, Store Y, and Load Z are the only instructions that access memory in this piece of code.

Ben runs the C code on his out-of-order processor which implements store sets to predict memory dependences as described in the store sets handout. Assume the following when evaluating how the above code executes on Ben's processor:

- *The value `j` is equal to `i` at every iteration of the loop. In addition, assume that the store address `A[j]` is available much later than the load address `A[i]`.*
- Because the stores depend on long-latency operations, the load is always ready to be issued before the store.
- All branches are predicted perfectly.
- The store sets of all loads are invalidated before the start of this code segment.
- The program performs a large amount of work at the end of the loop such that stores and loads from different iterations cannot be in flight at the same time.

Question 1 (3 points)

Out of the 100 iterations, how many times will the processor stall a load due to predicting a correct memory dependence via store sets?

First two iterations will be used to detect dependences between the two stores and the load. So answer is 98.

Now Ben runs the following C code on his machine:

```
int A[100];

for (int i = 0; i < 100; i++) {
      int x = foo();
      A[i] = x; // Store X
      ...
      int z = A[99 - i]; // Load Y
      ...
}
```

Store X and Load Y are the only instructions that access memory in this piece of code. Again, assume perfect branch prediction. Also assume that the processor has a large enough reorder buffer such that loads and stores from different iterations can be in flight simultaneously, and stores and loads are issued at roughly the same rate.

Question 2 (4 points)

Will our store set implementation predict a memory dependence between Store X and Load Y at some point? If so, what proportion of those predictions are false dependences (i.e., dependences that did not actually exist)? You don't need to give the exact answer, a rough proportion (within 5-10%) is fine. Justify your answer briefly.

Store sets will detect a dependence at the half-way point of the array (around A[49] or A[50]) due to loads and stores from different iterations being in flight at the same time. After that, all dependences it predicts will be false because the access streams of the stores and loads diverge to the lower and upper parts of the array. Thus, all of the dependences that store sets detects will be false dependences.

Question 3 (5 points)

Alyssa P. Hacker notes that Ben's store set implementation is always building store sets, but never forgetting useless ones. A simple solution would be to periodically clear all store sets, but Ben thinks this is too costly. Derive a simple mechanism that can remove stores from store sets that keep predicting a false dependence. *Hint: Think about how we design branch predictors.*

One ways is to keep a counter with some number of bits that keeps track of the "usefulness" of the detected dependence. The counter would sit alongside the store's SSIT entry, and decrement if it was part of a false dependence. When the usefulness counter reaches 0, we can evict the SSIT entry, thus removing the store from the store set. This is analogous to how we evict useless entries in a TAGE predictor.

Associative searches on store buffers are very expensive in terms of cycle time and power. Recall that when a load issues, it must perform an associative lookup on the store buffer to find any stores that it must forward data from.

With accurate memory dependence prediction, we can eliminate the store buffer completely. If we can detect that a load is dependent on an earlier store during decode, it can simply read the store's data input physical register instead of waiting for the store to write to value to its store buffer entry. This technique is called **Speculative Memory Bypassing (SMB)**.

For example,

```
SW   P3, 0(P4)  ; Store the value to memory
...
LW   P6, 0(P5)  ; Load the value from memory
```

Here, the stores and loads are denoted with the actual physical registers they read and write during execution (i.e., they are already renamed). Notice that if we predict that `Reg[P4] == Reg[P5]`, the `LW` can directly read the register `P3` and simply move the value to register `P6`. We call these *bypassing loads*. Loads that are predicted to not depend on a previous store (i.e., *non-bypassing loads*) can simply execute as before and read the data from the cache.

Since the physical register file now replaces the store buffer as a forwarding intermediary, we can eliminate the store buffer completely. Stores are now performed in order, when it is the store's turn to commit, updating the cache directly without an intermediary buffer.

We must also take care of two situations where we mis-predict the memory dependence: (1) A non-bypassing load that should have bypassed from a previous store, and (2) A bypassing load that should have read from the cache or bypassed from a different store. To preserve correct execution, every load is re-executed in program order when it is about to commit, always reading from the cache. This is called a *verification load*. If the original (speculative) load and the verification load produce different values, the load and all later instructions are discarded.

Ben modifies his processor to perform SMB and verification loads, allowing him to remove the store buffer entirely, and uses store sets to predict memory dependences.

Question 4 (4 points)

With Ben's modifications, do we still need any associative searches into the load buffer? Justify your reasoning.

No. Associative searches on the load buffer was needed because we needed to check if there was a younger load that was issued early that matches the store's address. This dependence will now be detected by the verification load, so no search is required. In fact, we can entirely remove the load buffer, calculating the effective load address at commit just like stores.

Question 5 (4 points)

If the majority of the program's loads are predicted as non-bypassing, are there any significant performance penalties associated with performing verification loads?

Yes. Re-executing all loads naively will consume twice the data cache bandwidth for non-bypassing loads (once for the original load, and once for verification load).

Question 6 (5 points)

Alyssa notes that the processor does not need to perform verification loads for all non-bypassing loads. If we can keep some information about previously committed stores, we can "filter" certain verification loads. Briefly describe the condition under which a verification load can be avoided.

We need to keep track of the last committed store to the same address as the load at the time of issue. Then, we check if there were any additional stores that have written to the same address that have committed. Note that this match need not be exact (we can deal with false positives -- e.g., using Bloom Filters).

Question 7 (5 points)

Explain a scenario where the processor with SMB (and no store buffer) would perform *better* than the original design (with load and store buffers, and store sets for memory dependence prediction). Then, explain a scenario where it would perform *worse*. State clearly any assumptions you make, and briefly justify your reasoning for each scenario.

There are multiple possible answers here. One possible answer is that we can reason about the store buffer as the bottleneck to the throughput of the processor. If there are a significant portion of stores in our instruction stream, then by Little's Law the store buffer will become the bottleneck as stores will be blocked in the front end of the pipeline waiting for an empty store buffer slot. In this case, the SMB design will outperform the original design.

The case where SMB would perform worse is when there are very few store-load pairs that have matching addresses. In this case we will consume much more of the data cache bandwidth (Question 5) which can bottleneck our processor. We also take the opposite of this case (lots of opportunities for bypassing, and our prediction is accurate) as the correct answer for when SMB would perform better.