Computer System Architecture
6.823 Quiz #1
October 15th, 2021

Name: ____________________________

This is a closed book, closed notes exam.
80 Minutes
16 Pages (+2 Scratch)

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.
- Show your work to receive full credit.
- 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.
- Pages 17 and 18 are scratch pages. Use them if you need more space to answer one of the questions, or for rough work.

Part A	________	30 Points
Part B	________	35 Points
Part C	________	35 Points
TOTAL	________	**100 Points**

Part A: Caches and Virtual Memory (30 Points)

Ben Bitdiddle is building a processor that uses 20-bit virtual addresses and 20-bit physical addresses. The processor has a single data cache with the following parameters:

- 8 sets
- 2-way set-associative
- 256-byte cache blocks

Ben's system uses a page-based virtual memory system with 1KB pages. Ben is wondering whether to use a virtually addressed (i.e., virtually indexed and virtually tagged) or a physically addressed (i.e., physically indexed and physically tagged) cache.

Question 1 (4 points)

Consider the virtual address VA = 0x06823 (0b0000 0110 1000 0010 0011). In the diagrams below, mark the locations where data corresponding to this virtual address can possibly reside for virtually addressed and physically addressed caches. If the data can reside in all entries, simply write "all entries" as your answer.

Page offset: bottom 10 bits (bits 0 to 9)
Index: bits 8 to 10
Thus, the top bit of the index may change depending on the VA->PA mapping.

Virtually addressed cache		
Index	**Way 0**	**Way 1**
0	X	X
1		
2		
3		
4		
5		
6		
7		

Physically addressed cache		
Index	**Way 0**	**Way 1**
0	X	X
1		
2		
3		
4	X	X
5		
6		
7		

Name ____________________________

Question 2 (4 points)

Now consider *physical* address PA = 0x06823 (0b0000 0110 1000 0010 0011). In the diagrams below, mark the locations where data corresponding to this physical address can possibly reside. If the data can reside in all entries, simply write "all entries" as your answer.

Notice that this is the dual of Question 1 (top bit of index can change for VA cache).

Virtually addressed cache		
Index	**Way 0**	**Way 1**
0	X	X
1		
2		
3		
4	X	X
5		
6		
7		

Physically addressed cache		
Index	**Way 0**	**Way 1**
0	X	X
1		
2		
3		
4		
5		
6		
7		

Question 3 (4 points)

Recall that in a virtually addressed cache, two virtual addresses that map to the same physical address can reside in two different cache sets, which can result in stale data. Ben wants to solve this problem for his virtually addressed cache by adding a physically addressed inclusive L2 cache. Alongside the physical tag, the L2 cache also stores the virtual tag of the data (i.e., the tag used by the virtually addressed L1 cache).

If an access misses in the L1 and hits in the L2, the virtual tag of the request is compared with that of the L2 cache line. If they do not match, the L1 traverses all of its entries and invalidates those that have a matching virtual tag.

Does this solve the problem for Ben's processor? Explain briefly.

No! Suppose two virtual addresses VA1 and VA2 have the same virtual tags but different indices, and map to the same physical address. Suppose VA1 data is installed in the L1 and L2. Then, upon a miss in the L1 for VA2, the L2 will have a virtual tag match with the entry for VA1, thus allowing the two addresses to co-exist in the cache.

Question 4 (5 points)

Alyssa P. Hacker thinks that Ben's solution is inefficient because the L1 has to check all tag entries for possible aliasing, which becomes more expensive as the L1 size grows.

Consider 2-way L1 caches of different sizes (not just the 8-set design from previous questions). What is the minimum fraction of L1 cache entries that must be checked to resolve aliasing using the scheme from Question 3? Give your answer as a fraction of total cache entries.

Two different cases:

- 4 sets of less: No aliasing occurs, so no check needed.
- 8 sets of more: 1/4 of the entries have to be checked since the bottom 2 bits of the index are fixed.

Ben now wants his system to support small (1KB) and large (64KB) pages. To accelerate address translation, Ben adds a single TLB that contains mappings for both small and large pages. The TLB is 2-way set-associative with 4 sets, as shown below:

Index	**Way 0**				**Way 1**			
	V	**L**	**Tag**	**PPN**	**V**	**L**	**Tag**	**PPN**
0								
1								
2								
3								

Each TLB entry contains the following fields:

- V: Valid bit
- L: Large bit, which is set to 1 if the entry is for a large page
- Tag: The upper 8 bits of the virtual address is stored as the tag
- PPN: The physical page number corresponding to the virtual address

A TLB lookup works in the following 2-step procedure:

1. First, we try to match small pages by using bits 10 to 11 of the virtual address as the index into the TLB, and the remaining upper bits (bits 12 to 19) as the tag. A small page match occurs if the tag matches and the L bit is not set.
2. If there are no small page matches, bits 10 to 15 are set to zero and the same indexing and tag scheme is used as in step 1. A large page match occurs if the tag matches and the L bit is set.

If there are no small or large page matches, the TLB lookup results in a miss. The Memory Management Unit walks the page table, finds the VPN-to-PPN mapping, and populates the appropriate TLB entry.

Question 5 (3 points)

What is the reach of the TLB (i.e., the total amount of virtual address space the TLB can hold translations for)?

Large pages can only be mapped to the first set, so the reach is:
(64KB + 3*1KB) * (2 ways) = 134KB.

Question 6 (5 points)

Ben decides to divide up his virtual address space into two halves, where the upper half of the address space only uses 64KB pages, and the lower half only uses 1KB pages. Ben thinks that using this address division can help make TLB lookups more efficient.

Given this scheme, can you make any changes to the TLB lookup procedure to make TLB accesses more efficient? Briefly explain your reasoning.

Yes. Instead of the two-step lookup procedure, the TLB can use the MSB of the address to distinguish between large- and small-page lookups. If the MSB is zero, we only look for small pages, and if the MSB is one, we only look for large pages.

Question 7 (5 points)

Alyssa notes that Ben's scheme is too restrictive. She instead suggests that the upper half of the address space be used as a *preferred* address space for large pages. Processes can have large and small pages anywhere in their address space, but they should try to allocate small pages in the lower half and large pages in the upper half whenever possible.

Given Alyssa's alternate scheme, can you still make changes to the TLB lookup procedure to make TLB accesses more efficient? Briefly explain your reasoning.

Yes. We can swap the lookup procedure depending on the MSB. If the MSB is one, we *first* lookup for large pages and then look for small pages if there are none. If MSB is zero, we follow the same procedure as before.

Name ____________________________

Part B: Out-of-Order Processor (35 points)

Question 1 (25 points)

This question uses the out-of-order machine described in the Quiz 1 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: I5 commits and writes R2 in the register file.
Answer: (P): Commit correctly speculated instruction, and mark lazily updated values as non-speculative.
(You can simply write "P".)

a) Assume instruction I5 commits. Instruction I6 finishes execution and commits, and the valid bit of the rename table entry for register R3 is cleared.

   (R): The rename table entry for R3 does not match the tag of I5, so the rename table should not be altered.

b) Instruction I9 causes an exception. All instructions following ROB entry T7 are cleared from the ROB, and the ROB is drained until it becomes empty.

   (M): Abort speculative action and cleanup lazily managed values

c) Instruction I6 finishes execution. The result of the divide is written to the data field of T4.

   (F): Write a speculative value using lazy data management

d) Assume T5 becomes available. Instruction I9 is issued because both of its source operands are present.

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

e) Assume I18 is decoded and found to be a conditional branch. The local history branch predictor predicts taken, and instruction I20 is fetched from the calculated target address.

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

f) Assume instruction I16 has a destination register R2. When I16 is allocated, its ROB entry tag T14 is written to the R2 entry of the rename table and its valid bit is set.

(G): Write a speculative value using greedy data management

g) Assume T9 becomes available. Instruction I12 is issued, and the branch is found to have not been taken. The rename table is restored to the snapshot that was taken when I12 was allocated.

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

h) Assume all instructions up to I11 commit. I12 is issued, executes, and commits. Upon commit, the corresponding prediction counter entry for the branch is updated.

(J, iii): Non-speculatively update a prediction on branch direction

i) Assume all instructions up to I7 commit. I8 commits, and since the rename table entry for its destination register does not contain its tag, it does not write its data value to the register file.

(R): Every instruction should write its result to the architectural register file when it commits, regardless of the rename table.

j) Assume T8 becomes available. Instruction I11 is issued the cycle after I6 finishes execution because the processor has a single unpipelined divider.

(A, vii): Satisfy a dependence on functional unit by stalling

Name ____________________________

Question 2 (5 points)

Is the local history branch predictor able to recover from branch misprediction perfectly? If so, briefly explain why. If not, describe a scenario in which perfect misprediction recovery is impossible with the described recovery mechanism.

No, it is not perfect. A simple case is when there are enough branches in flight such that we overflow the local history table entry with speculative predictions.

Question 3 (5 points)

Ben Bitdiddle profiles the execution of a wide variety of programs with the out-of-order processor and discovers the following: Each instruction on average takes 5 cycles to be fetched, decoded, and allocated into the ROB. It then spends 4 cycles on average in the 10-entry ROB before being issued. After being issued, it takes 3 cycles on average to finish. The instruction then spends an average of 4 cycles waiting in the ROB until commit, and the entry it used to occupy becomes available for another instruction 2 cycles later.

Assume perfect branch prediction. What is the expected average throughput of the machine?

Average length of occupancy: $4 + 3 + 4 + 2 = 13$ cycles
Number of ROB entries: 10 entries
Average throughput = 10/13 instructions/cycle

Name ___________________________

Part C: Branch Prediction (35 points)

Ben Bitdiddle is designing a branch target buffer (BTB) for a 2-wide superscalar processor with a complex pipeline with the following stages:

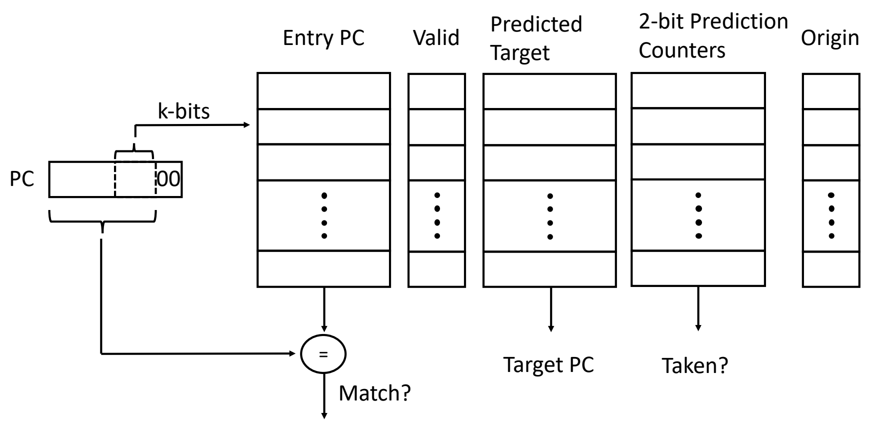
A	Address (PC) generation
F1	Instruction Fetch Stage 1
F2	Instruction Fetch Stage 2
F3	Instruction Fetch Stage 3
B	Branch Address Calculation / Begin Decode
D	Complete Decode
J	Steer Instructions to Functional Units
R	Register File Read
E	Execute
⋮	Remainder of execution pipeline

The processor has the following characteristics:

- The machine fetches two consecutive instructions per cycle starting from the *fetch address*. We will refer to the two instructions fetched in a single cycle as an *instruction bundle*. Note that the PC of the first instruction of an instruction bundle is the fetch address.
- At the beginning of the A stage, the BTB immediately begins processing to generate the target PC of potential branch(es) in the instruction bundle.
- The branch target address is known at the end of the B stage
- The branch condition is known at the end of the E stage

Because we are fetching two instructions per cycle, either (or both) of the two instructions can be a branch, requiring a prediction from the BTB. Thus, Ben designs a direct-mapped, tagged BTB that generates a target address for the *entire instruction bundle* in a single cycle. In his design, the BTB is looked up with fields in the fetch address (i.e., the PC of the first instruction in the bundle).

Recall from lecture that a tagged BTB is just like a cache, where fields of the PC are used to index into the structure and conduct a tag match. Each entry in Ben's BTB has a corresponding target PC for the given entry PC, and a 2-bit saturating counter that predicts whether the branch is taken based on the uppermost bit. We define a BTB lookup as a *hit* if the Entry PC tag matches and the prediction counter predicts taken, and a *miss* otherwise. The figure below shows a diagram of Ben's BTB that has 2^k entries:



A BTB hit indicates one of the two instructions in the instruction bundle is predicted as a taken branch, and the processor fetches the next instruction bundle from the target PC. A BTB miss indicates that no branch is found within the instruction bundle, and the processor simply fetches the next sequential instruction bundle.

The BTB's target PC is updated if a branch is found to be taken and it does not match the target PC of the instruction in the E stage. The prediction counter is incremented if the branch is found to have been taken in the E stage, and decremented if not taken. Note that if two *taken* branches both try to update the BTB in the E stage, the first instruction wins because it is earlier in program order.

The BTB also records an additional 1-bit field *origin* that records which among the two instructions in the instruction bundle was the taken branch. The origin is set to 0 if the first instruction is found to have been taken later down the pipeline, and 1 if the same condition holds for the second instruction. Upon a BTB hit where the origin field is 0, the processor kills the second instruction in the bundle, since the instruction is predicted to be in a wrong path of execution.

Question 1 (4 points)

Fill in the following table to indicate how many instructions can possibly be killed **at most** for each given scenario. The rows indicate possible BTB lookup results (hit or miss), and the columns indicate actual branch outcomes. Assume that the branch target upon a BTB hit is always correct. Recall that Ben's BTB is able to produce a target PC in a single cycle. Make sure to include the second instruction in the instr. bundle which can be killed!

BTB lookup	**Max instructions killed if branch outcome is**	
	Taken	**Not taken**
Hit	1	17
Miss	17	0

Question 2 (6 points)

Ben is interested in figuring out what happens to his single-cycle BTB when the instruction bundle consists of two branch instructions. Consider the following loop written in C:

```
for (int i = 1; i < 1000000; i++) {
      if (i & 1 == 0) { // branch B1
            // do something A
            ...
      }
      else if ((i+1) & 3 == 0) { // branch B2
            // do something B
            ...
      }
      else {
            // do something C
            ...
      }
}
```

Following is the corresponding assembly code:

```
0x400:      LOOP: ADDI  R1, R1, 1
0x404:            ADDI  R2, R2, 1
0x408:            ANDI  R3, R1, 1         // i & 1
0x40C:            ANDI  R4, R2, 3         // (i+1) & 3
0x410:            BEQ   R3, R0, M1        // branch B1
0x414:            BEQ   R4, R0, M2        // branch B2
                  (do something C)
                  ...

0x608:            BNE   R1, 999999, LOOP
0x60C:            J     END
0x610:      M1:   (do something A)
                  ...

0x808:            BNE   R1, 999999, LOOP
0x80C:            J     END
0x810:      M2:   (do something B)
                  ...

0xA0C:            BNE   R1, 999999, LOOP
0xA10:      END:
```

Assume the following:

- `R1` and `R2` are initialized to 0 and 1, respectively, before the first iteration of the loop.
- All BTB entries are zeroed before the start of the loop.
- The BTB has enough entries such that no other branches alias with the fetch address `0x410`, and there are no other branches other than those shown in assembly.
- By the time the processor loops back to address `0x410`, branch instructions `B1` and `B2` have committed.

Calculate how often Ben's BTB will correctly predict the branch target for branches `B1` and `B2` in *steady state*.

a) `B1`

Note that B1 and B2 have the following branch pattern (1 = taken, 0 = not taken):

B1: 1 0 1 0 1 0 1 0 1 0 1 0 ...
B2: 0 1 0 0 0 1 0 0 0 1 0 0 ...

Blue indicates the repeating pattern. Since the 2-bit counter will eventually be trained to always predict taken, we only need to figure out whether the branch targets match. Notice that the branch target will ping-pong between M1 and M2. So for every 4 iterations, the BTB target will be M1, M1, M2, M1, which means B1 will be correct 1/4 of the times.

b) `B2`

From the above reasoning, we can see that B2 is always mispredicted in steady state.

Question 3 (4 points)

How would you fix the assembly code above to improve the prediction accuracy of Ben's BTB for branches `B1` and `B2`? You don't have to write down the code. Simply describe the change to the code at a high level.

We can have B1 and B2 reside in different instruction bundles (e.g., insert a NOP in between) such that they use different BTB entries, which will improve prediction accuracy for B1(it will predict always predict not taken, which will be correct half the time) and B2 (it will also predict always not taken, and be correct half of the time).

Question 4 (4 points)

Suppose the compiler organizes the code such that all branch target addresses are 8-byte aligned (i.e., the bottom three bits are zero), and the processor starts execution from an 8-byte aligned address. What is the maximum utilization of Ben's BTB in this scenario? Explain briefly.

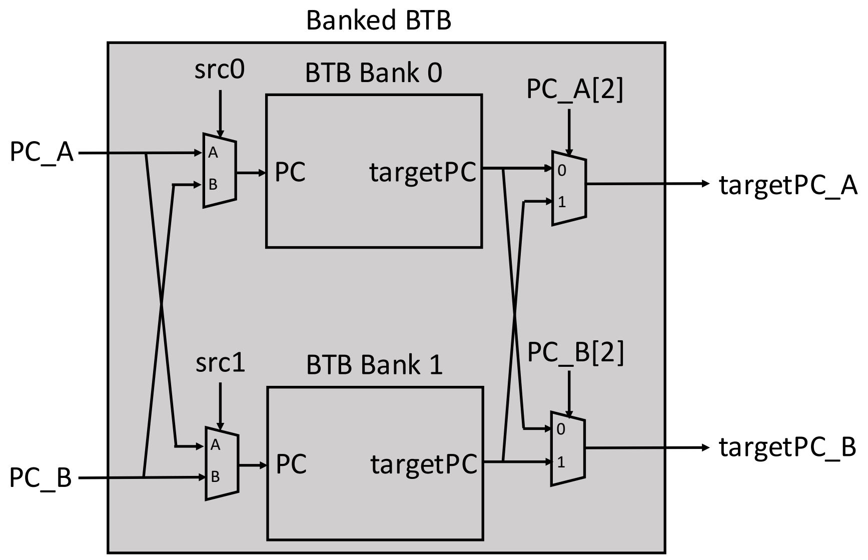
We will only use half of the BTB entries at most since we will only ever use the even entries of the BTB (entry 0, 2, 4, ...).

Ben now considers a *banked BTB* design. In this design, the BTB is split into two smaller structures that we call *banks*, each of which holds half of the BTB entries. The banked BTB now has two input ports A and B that each accept a PC for BTB lookup, and two output ports that each correspond to the target PC of the corresponding input (`targetPC_A` for input `PC_A`, and `targetPC_B` for input `PC_B`).

BTB entries are divided into the two banks as follows:

- Bank 0 holds BTB entries corresponding to instructions at even word locations, i.e., 0x0, 0x8, 0x10, 0x18, etc.
- Bank 1 holds BTB entries corresponding to instructions at odd word locations, i.e., 0x4, 0xC, 0x14, 0x1C, etc.

At every cycle, the banked BTB is accessed at the beginning of the A stage by feeding the PC of the first and second instruction to ports A and B respectively. Because each bank can independently process a single BTB lookup, two PCs that look up different banks can be executed in parallel in a single cycle. However, if the two lookups go to the same bank, they will be serialized across two cycles. The diagram below shows a possible implementation of the banked BTB:



Question 5 (4 points)

Is there ever a situation where the banked BTB takes two cycles to generate a target PC in Ben's processor? If so, describe the scenario. If not, briefly explain your reasoning.

No. Since every instruction bundle consists of instructions contiguous in memory, the two instruction PCs will always use different banks.

Question 6 (4 points)

Consider again the assembly code from Question 3:

```
0x400:     LOOP: ADDI  R1, R1, 1
0x404:           ADDI  R2, R2, 1
0x408            ANDI  R3, R1, 1       // i & 1
0x40C:           ANDI  R4, R2, 1       // (i+1) & 3
0x410:           BEQ   R3, R0, M1      // branch B1
0x414:           BEQ   R4, R0, M2      // branch B2
                 (do something C)
                 ...

0x608:           BNE   R1, 999999, LOOP
0x60C:           J     END
0x610:     M1:   (do something A)
                 ...

0x808:           BNE   R1, 999999, LOOP
0x80C:           J     END
0x810:     M2:   (do something B)
                 ...

0xA0C:           BNE   R1, 999999, LOOP
0xA10:     END:
```

Assume the following (same as in Question 3, except for the third bullet point):

- `R1` and `R2` are initialized to 0 and 1, respectively, before the first iteration of the loop.
- All BTB entries are zeroed before the start of the loop.
- The BTB has enough entries such that no other branches alias with the branches at addresses `0x410` and `0x414`, and there are no other branches other than those shown in assembly.
- By the time the processor loops back to address `0x410`, branch instructions `B1` and `B2` have committed.

Calculate how often the banked BTB will correctly predict the branch target for branches `B1` and `B2` in *steady state*.

a) B1

Now B1's behavior is captured independently, so it is predicted (not taken) correctly half the time.

b) `B2`

Note that a "prediction" on B2 when B1 is taken is meaningless since it's as if the branch never executed in the first place. Given this, the prediction accuracy is 1/2.

If you assumed the question was asking how often the BTB *bank* predicted correctly, the answer would be 3/4. We gave full points for this answer.

Question 7 (4 points)

Can the banked BTB ever achieve *worse* overall prediction accuracy than Ben's original BTB? Assume that both BTBs have an equal number of total BTB entries, and unlike in Question 4, don't assume that all the branch targets are 8-byte aligned. If it is possible, describe a scenario where this can happen. If not, briefly explain your reasoning.

Yes. Consider N branches all at 8-byte aligned addresses, but the fetch address for half of them is odd (i.e., they are fetched as the second instruction in the instruction bundle). Then, a N-entry BTB of original design can store the target address for all these branches assuming no aliasing, whereas the banked BTB can only store half.

Question 8 (5 points)

We define a *basic block* as a piece of straight-line code that has the following properties:

- Single-entry: No instruction other than the first one in a basic block is a target of any branches or jumps in the program.
- Single-exit: The basic block is terminated by a branch or a jump, with no other branches in-between.

Assume that every basic block in Ben's program is composed of 6 instructions. Ben now wants to design an 8-wide superscalar processor, which motivates Ben to design a BTB that predicts *two* target PCs for a given instruction bundle: one for the basic block that is the target of the current instruction bundle, and the second for the basic block that is the target of the branch instruction in the *first* basic block. Given the two target PCs, the processor fetches 6 instructions from the first basic block, and two from the second.

Ben finds that the overall prediction accuracy for each basic block upon a BTB hit is 90%. What is the average number of useful instructions fetched upon a BTB hit?

Average number of useful instructions from 1st BB: 0.9 * 6 = 5.4
Average number of useful instructions from 2nd BB: 0.9^2 * 2 = 1.62

Thus, the BTB fetches 7.02 useful instructions on average.

Name ____________________________

Scratch Space

Use these extra pages if you run out of space or for your own personal notes. We will not grade this unless you tell us explicitly in the earlier pages.

Name ______________________________

Scratch Space