Name _____(answer key)_____

Computer Architecture and Engineering
CS152 Quiz #2
March 6th, 2012
Professor Krste Asanović

Name:__<ANSWER KEY>__

This is a closed book, closed notes exam.
80 Minutes
12 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 instruction ask you to explain your choice.

Writing name on each sheet	_____	2 Points
Question 1	_____	18 Points
Question 2	_____	29 Points
Question 3	_____	31 Points
TOTAL	_____	**80 Points**

Name _____(answer key)_____

Question 1: Three C's of Cache Misses (18 points)

Mark whether the following modifications to cache parameters will cause each of the categories to **increase, decrease**, or whether the modification will have **no effect**. You can assume the baseline cache is set associative. **Explain your reasoning** to receive credit.

Assume that in each case the other cache parameters (number of sets, number of ways, number of bytes/line) and the rest of the machine design remain the same.

	compulsory misses	**conflict misses**	**capacity misses**
increasing number of sets	no effect block size is constant	decreases more sets are available for data, so there is less of a chance for two ops to collide and evict one another	decreases capacity increases
increasing number of ways	no effect block size is constant	decreases there are more ways available for data to be placed into	decreases capacity increases
increasing number of bytes per line	decreases more data is brought in on a given cache miss	no effect associativity and number of sets is constant	decreases capacity increases

Question 2: Way-Predicting Cache Evaluation (29 points)

As a new hire for Caches-R-Us, you are tasked with evaluating a new cache design. To improve hit rates, your team has decided to use a two-way set-associative cache (it was formerly direct-mapped). Unfortunately, the *access time* has suffered. To fix this, you propose a "way-predicting" cache: on every cache access, you predict which way to select and read out the data in that way.

On a cache access, the prediction is used to route the data. If it is incorrect, there will be a delay as the correct way is accessed. If the desired data is not resident in the cache, it is like a normal cache miss. Figure 1 summarizes this process:

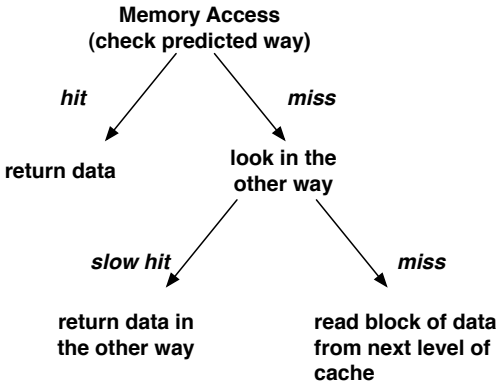

Figure 1: Way-prediction FSM

Since there are only two ways, only one bit will be used per prediction, and its value will directly correspond to the way. For this problem, you can ignore how predictions are generated. You can assume at the beginning of a cycle, the selected prediction is available, and determining the prediction is not on the critical path.

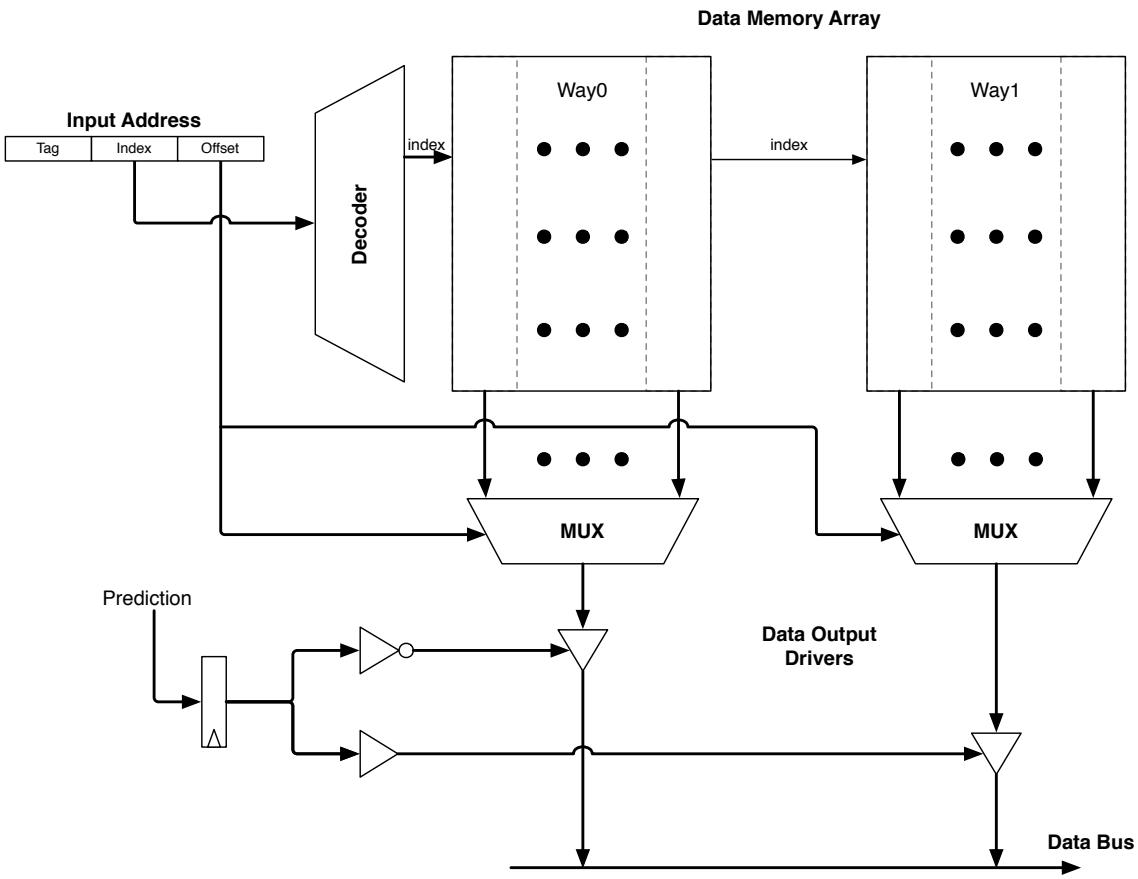

Figure 2. Way-Predicting 2-way Set-Associative Cache Datapath

Q2.A: Cache parameters (4 points)

Fill out the following table of cache parameters. The address size is 32 bits, the index size is 8 bits, and the block offset size is 4 bits (*note*: these parameters will hold for the rest of Question 2).

	Way-predicting 2-way set associative cache
Tag size (bits)	20 bits
Cache line size (bytes)	16 bytes
Number of sets	256 sets
Cache capacity (KB)	256*16*2=8192 (8 kB)

Table 1. Cache Parameters

Q2.B: Critical Paths (6 points)

Using Figure 2, Table 1, and Table 2, determine the cache access time (ps) for our way-predicting cache (i.e., the delay through the critical path). You can assume the prediction bit generation is off the critical path.

Component	Delay equation (ps)	Total (ps)
Decoder	20×(# of index bits) + 60	220
Memory array	$10\times\log_2$(# of rows) + $10\times\lfloor\log_2$(# of bits in a row)$\rfloor$+200	360
N-to-1 MUX	$50\times\log_2$ N + 200	300
Buffer driver	160	160
Data output driver	90×(associativity) + 100	280
Critical Path Delay		1160 ps

Table 2. Delay of each cache component

Buffer driver is not on critical path. # of bits in row includes across ways (so 16B*8b*2ways). -(1/2) pt for that. (if 350)

Notice how way prediction takes the tag check off the critical path. The inverting and non-inverting buffer drivers both have the same delay. You only need to worry about the case of a fast hit (cache hit with correct prediction).

Q2.C: Way-predicting I-Cache (5 points)

You realize that way-prediction could work well in the instruction cache. You propose a way-predicting I-cache where each set has its own prediction bit that records the last way that was accessed (for that set only).

For a cache with a LRU replacement policy (and the given parameters in Table 1), under what scenarios do you expect the proposed way prediction scheme to mispredict for an instruction cache?

This scheme does well for short loops that stay within the same cache line (temporal locality), and it works well for sequential code that steps through at unit-stride (spatial locality. It will only mispredict when it runs off the cache line). However, it does poorly with code that branches between two different ways in the same set.

-1 point if didn't say "code map to same set", since that would be the worst possible case (i.e., specified that the branch is to a section of code that is a multiple of 4kB away).

+2 if mentioned branching
+3 if mentioned same set
+1 for compulsory misses

Q2.D: Robust Prediction for the I-Cache (5 points)

A new proposal for predicting ways in I-caches is even more robust.. It combines three different techniques used in different scenarios: 1) when stepping sequentially through instructions on the same cache line, predict the same way, 2) when accessing the last instruction in a line, keep a prediction bit (one per line) to remember which way the next sequential instruction is in, and 3) for taken branches, add a prediction bit to the Branch History Table to point to the way that holds the target of the branch.

Explain all cases where this new prediction scheme will be incorrect (i.e., mispredict) for typical program behavior. Assume the cache still uses a LRU replacement policy.

Looking for at least two coherent, correct statements:

- cache miss, you haven't yet learned the prediction (i.e., where the next way is the first time you run off the new cache line; the first time you take a branch you don't know which way it's in)

- branch mispredict (i.e., the last iteration a loop) will send you to the wrong way

- aliasing in the branch predictor. Even if it predicts the branch path correctly, it may predict the wrong way (because two branches are sharing the same BHT entry).

+3 points for getting the first case.
+2 points for getting the second case.

4/5 points given for people that gave more than one situation, but it was just a variation of a single "category" (i.e., only listing different ways to mispredict a branch, but not covering cache misses or branch aliasing).

Q2.E: Way Prediction in the Data Cache (5 points)

Your boss also asks you to evaluate using a way-predicting 2-way set-associative cache as the data cache. The proposed prediction scheme for the data cache is to set the prediction bit to the last accessed way (each set has its own prediction bit that remembers the last way accessed). Explain how this technique will affect cache hit time compared to both a regular 2-way set-associative data cache and a direct-mapped cache.

Way prediction is faster than a 2-way set-associative cache (tag is not on critical path).

However, it is slower than direct-mapped caches because it still must select and drive out data from one of the ways.

Q2.F: Aliasing (4 points)

Assuming that the page size is 4KB for this machine, and the cache parameters from Table 1 still hold, and the cache is physically-tagged, virtually-indexed, does our way-predicting 2-way set-associative cache have a problem with aliasing? ***Describe*** why or why not.

Nope, there are no problems with aliasing. The number of bits for the page offset is the same as the number of bits for the index + number of bits for the block offset, meaning that each alias can only be found in a single set. Thus two aliases will index the same set, and share the same physical tag.

-3 for saying they will map to different indices (but recognize different indices would cause issues and mention a way it could be fixed)

-2 for wrong answer, but consistent with past answers?

-3 for saying “no”, but no backing this statement up.

Notice that way prediction has no effect on this.

Question 3: Virtual Memory and Big Pages (31 points)

For this problem, the machine you are studying uses a 2-level page table scheme. Also, the OS is smart enough to only allocate memory for the pages that it uses. Addresses in this machine are 32-bits long. The page offset is 12-bits in size. Both the level-1 page table index and level-2 page table index are 10 bits each.

10	10	12
L1 page idx	**L2 page idx**	**page offset**

Q3.A: Aggressor Code (4 points)

Provide a sequence of addresses (in hexadecimal) that a user program could issue to the memory system that would give the fastest growth in total physical memory usage for this system.

Step through addresses with a stride of 0x40_0000

0x00_0000
0x40_0000
0x80_0000
0xc0_0000

This is walking through the L1 page index, allocating a whole new L2 page for each memory access. After 1024 accesses, you have filled the L1 page table, allocated 1024 L2 page tables, and allocated 1024 data pages. If you step through on a L2 page idx unit-step, you would only allocated 1024 data pages and 1 L2 page table after 1024 accesses.

-2 points for stepping through at the L2 page index size.

Q3.B: Page Table Overhead (4 points)

As you learned in Lectures 8 and 9 (which covered address translation and virtual memory), the page tables themselves also reside in memory. Naturally, these page tables are "overhead" that is required to track all of a program's "user data". Describe the worst-case scenario in terms of memory devoted to "overhead" versus memory devoted to "user data". In other words, in what situation would the most overhead be allocated relative to user data (you do not have to write code, but be quantitative and exact).

Only using 1 user page. That requires one L1 page table and one L2 page table, so a ratio of 2:1 in overhead to user data.

-1 point: Most students said to have a filled L1 table (one page) by striding 4MB through memory -> that's 1 L1 + 1024 L2s + 1024 user pages -> ~1:1 ratio.
-1 point: suggested that you aren't counting the full 4kB page as being allocated if you touch 1 word.
-4 points: If student said to allocate all of memory. That's the answer to the next question.

Q3.C: Page Table Overhead Part II (4 points)

Now the other extreme: describe the best-case scenario in terms of the ratio between allocated "overhead" to allocated "user data" (minimizing overhead).

Allocate all of memory. one L1 page + 1024 L2 pages + 1M user pages -> ~1/1024 overhead to user pages.

Notice, we don't care *how much of the page is used*. If one word from the user page is referenced, the whole page must be allocated.

-1 point: if said to only have one L1 page, one L2 page, full L2 allocation (doesn't amortize out the single L1 table).

Q3.D: Memory System Performance (6 points)

Consider a memory system in which you have a direct-mapped TLB with 64 entries. Assume that page table entries are **not** allocated in the cache. If a user program performs sequential (unit-stride) 32-bit accesses through 4GB of memory, how many **TLB misses** will occur? How many **individual memory accesses to the page table** are required?

3 points:
1 million (2^20) TLB misses
two memory accesses for each miss (reading the L1 page, reading the L2 page. *Remember*: this is only because we artificially restricted the problem to never allow the caching of the page tables. In a real machine, the locality of our accesses should have the L1 and L2 page in the caches).

3 points:
Thus, 2 million accesses (2^21). full credit if answer was 2x the wrong answer given for the TLB misses.

TLB misses __ 1 million accesses (2^20)__

accesses __ 2 million accesses (2^21)__

Q3.E: Big Pages (4 points)

Big Page Address

10	10	12
L1 page idx	**page offset**	

Now let us consider adding "Big Pages" to the system. Normally, memory is allocated using the normal ("little") pages. Translating a virtual page number to a physical page number requires walking two levels of page tables to find the base address to the "little" page (you learned this in lecture). However, "Big Pages" allow programs to allocate enormous chunks of memory as a single, *big* page. The virtual page number of a "Big Page" is only 10 bits: the same, upper 10 bits used for the Level 1 page table index. The remaining 22 bits of the "Big Page" is used as the big page's offset. In this case, how **big is a "Big Page"** (how much memory can a program fit into a single "Big Page")? What is the **TLB reach** using only "Big Pages", for a TLB with 64 entries?

The big page has an offset of 22 bits, so it can address 2^22 bytes, or 4 MB. If the TLB holds 64 entries, each of 4MB, then the TLB reach is 256 MB.

"Big Page" size__4 MB__

TLB reach_256 MB__

Q3.F: Unit-Striding through Big Pages (6 points)

Because "Big Pages" only have enough space for the Level 1 page table index, translating from the "Big Page"'s virtual page number to its physical page number requires only looking at the level 1 page table. This means that the level 1 page table holds *either* the *base address* pointing to the Level 2 page table (for use by a small page) or it holds the *physical page number* for a "Big Page".

Let's return to Q3.D again, but now you use ***only*** "Big Pages" when allocating user data. If you are performing a series of sequential (unit-stride) accesses through 4 GB of memory, how many **TLB misses** occur? How many **individual memory accesses to the page table** occur? (TLB is still 64 entries and direct-mapped, and page table entries are still ***not*** stored in the cache).

Because there is greater TLB reach, there will be far fewer TLB misses (4GB/4MB). Also, since the L1 page table points directly to the base address of the big page, there is only 1 access per TLB miss required.

TLB misses __ 1024__

accesses __ 1024__

Q3.G: TLB Design for Big and Small Pages (3 points)

Consider a direct-mapped TLB design which holds 32 entries for small page address translations and 32 entries for "Big Page" address translations. In terms of area and power, is this design cheaper, equal to, or more expensive than a direct-mapped TLB with 64 entries of only small page address translations.. Only consider the design of the TLB itself in answering this question.

Tag is smaller

0 for saying equal bits
-2 for saying complex logic dominates the smaller tag.

END OF QUIZ