CS 250 Spring 2017 Homework 10 SOLUTION and Grading Guide
GTAs: Please enter scores into Blackboard by Tuesday, April 18, 2017

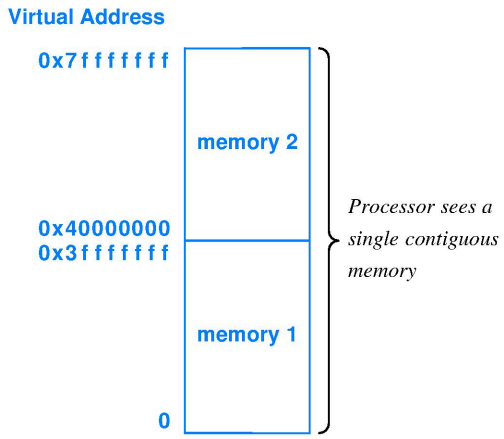
Maximum score = 15

1. Consider the circuit for a 4 KB, 2-way set associative write-back cache using LRU replacement and having 16-byte blocks that is for a computer using 32-bit addresses that address 32-bit words in main memory.
   a. **[3 pts., 1 each for each field]** What are the sizes of the tag, index, and block offset fields in bits?
   Answer: The tag is leftover bits, and the number of blocks is not given, so start with the offset.
   Lines are 16 bytes and the question statement does nothing to suggest that memory is not byte-addressed, so we need 4 offset bits because they can point to any one of the 16 bytes within a block. Next, cache size is 4 KB, which refers to storage area for the blocks. So, 4 KB at 16 bytes/block gives 256 blocks, which are configured at 2 blocks per set, implying there are $128=2^7$ indices for 7 index bits. Thus 32 – 7 – 4 = 21 tag bits.
   Thus, tag size = 21 bits, index size = 7 bits, and offset size = 4 bits for a total of 32.

   b. How many blocks are there in the cache?
   Answer: 256 blocks, see part (a) for analysis.

   c. **[1 pt, either final answer receives credit]** How many bytes of overhead are there in the cache? What percentage of all memory bits in the cache are overhead bits?
   Answer: The overhead bits for each block and are: tag field bits, valid bit, dirty bit (because this is a write-back cache), and any bit(s) used to support the replacement policy (algorithm) required because this cache is 2-way set associative.
   The design of the circuit to support LRU replacement in a 2-way set can be as simple as a single bit of storage to track which one of the two blocks in the set was least recently used. If we design that way then:
   Total overhead bits per block is 21 + 1 + 1 + 1/2 = 23.5 bits. There are 256 blocks, so there are (23.5*256)/8 = 752 bytes of overhead. The overhead percentage is then 752 bytes / (752 + 4096) bytes = 15.5%.
   If instead the design uses 1 bit per block to track block use, then the overhead is (24*256)/8 = 768 bytes, and the percentage is 768 bytes / (768 + 4096) bytes = 15.8%.

2. **[2pts.; 1 point each for identifying the correct module and for stating the correct offset]** Given the following virtual address space



and the C program

```
char c;
char *p;
p = (char *)1073741826;
c = *p;
```

Which memory module will be referenced, and where will the referenced byte be located within the module?

Answer: The pointer p is given the value 0x40000002 (find a decimal to hex converter on the internet, or crunch it yourself). This address is in module 2 and at byte offset 2 within that module.

3. **[3pts., 1 pt. each bold entry in the answer table]** Assuming a page size of 8 Kbytes what is the page number in hexadecimal and offset in hexadecimal for addresses 0x00000FFF, 0x00001FFF, 0x00002002, and 0xFFFFDFFD? Pad as needed with leading zeros to obtain hexadecimal representations for your answers.

   Answer: An 8 Kbyte page contains 2^{13} bytes, so the offset field is 13 bits, the given addresses are 8 hexadecimal digits, thus 32-bit addresses, so the page number is the most significant 19 bits. Splitting the given addresses into these two fields and writing the result as two hexadecimal numbers, with the fields padded with zero bits to obtain a multiple of 4 bits in each case we have

Given address	Binary address showing split	Page number	Offset
0x00000FFF	00000000000000000000 0111111111111	0x00000	**0x0FFF**
0x00001FFF	00000000000000000000 1111111111111	0x00000	0x1FFF
0x00002002	00000000000000000001 0000000000010	**0x00001**	0x0002
0xFFFFDFFD	11111111111111111110 1111111111101	**0x7FFFE**	0x1FFD

4. **[3 pts., 1 pt. each for each answer in red below]** Assume a virtual memory with a 13-bit address space organized into 8 pages. Physical DRAM memory implements addresses 0x000 through 0xFFF.

Page number	Page frame
0	3

1	1
2	Not present in DRAM memory
3	Not present in DRAM memory
4	2
5	Not present in DRAM memory
6	0
7	Not present in DRAM memory

In the table below. for each listed virtual address, fill in the corresponding physical address or note if a page fault would occur.

Answer: Pages are consecutive chunks of addresses, so if there are 8 pages in this 13-bit address space then we need 3 bits to denote which page, i.e., for the tag field, which will be the 3 high order bits. Now, in a virtual memory system, when a page is brought from, say, disk, into main memory, that page can be placed anywhere in the physical memory. Why do we allow anywhere? First, because we can. The page table above, which is not complex, implies that there is no restriction on the page frame number at which a given page can reside. Second, because allowing any page to reside at any page frame means that the page replacement policy has total freedom to make the best choice of which page to eject from among all the pages in memory when a page must be ejected. Such page placement freedom means the system has one less obstacle to achieving its best effort level of reduction of transfers between main memory and disk, each of which we know is a performance-crushing disaster. Such placement freedom also is the key characteristic of a *fully associative cache*.

For a fully associative caching system there is no point to identifying the set into which a given block goes because a block can be placed anywhere. Consequently there is no Index field of bits in such a system. Therefore, after 3 bits to identify the page, this leaves 10 bits for the offset within a page. Thus, a page in this system holds the contents of 1K of addresses. So the given 12-bit-addressed physical memory, i.e., 4K locations, holds these 1K pages at page frame address ranges 0x000 – 0x3FF (Page Frame 0), 0x400 – 0x7FF (Page Frame 1), 0x800 – 0xBFF (Page Frame 2), and 0xC00 – 0xFFF (Page Frame 3 and the final frame because 0xFFF is the "top" of this memory). This physical memory can hold 4 pages at a time.
The page location information in the table given in the problem statement can now be converted into the contents of the TLB as follows:

Page number	Starts at frame physical address
0	0xC00
1	0x400
2	Not present in DRAM memory
3	Not present in DRAM memory
4	0x800
5	Not present in DRAM memory
6	0x000
7	Not present in DRAM memory

To translate a virtual address to a physical address, split the virtual address into a page number (high order 3 bits), look up where in main memory that page number is and (1) declare a Page Fault if the page is not in main memory or (2) add the Offset field (low order 10 bits in this problem) to the starting frame address given in the TLB. **This yields a final answer of:**

Virtual address (13 bits shown in hex with leading zero padding)	Corresponding physical address or Page fault
0x0000 → page 000 = 0, offset = 0x000	0xC00 + 0x000 = **0xC00**
0x0E90 → page 011 = 3, offset = 0x290	3 → not present = **Page fault**
0x03FF → page 000 = 0, offset = 0x3FF	0xC00 + 0x3FF = **0xFFF**
0x0400 → page 001 = 1, offset = 0x000	0x400 + 0x000 = **0x400**

5. **[3 pts., 1 pt. each for each answer in red below]** In Question 4 we assumed that the given page table did not change as the CPU tried to access each of the 4 virtual addresses. An actual page table would dynamically update per its design as the CPU accessed successive virtual addresses.
So, let's assume that the CPU has accessed the four virtual addresses in the table of Question 4 in the order shown, that is, in the order of reading down the column of the table. Further, let's assume that any page faults that you identified in answering Question 4 are the only page faults that occurred when the CPU accessed these four addresses in order (depending on the replacement algorithm this need not be the case). Finally, let's assume that we do not know the page replacement algorithm used by this virtual memory system design nor do we know which accesses in Question 4 were loads and which were stores. Finally, a complete page table must at least have bits for each page to indicate Valid and Dirty (other bits may be needed to support access permissions and the page replacement algorithm, but let's not worry about access permission and we do not know the replacement algorithm, so we will ignore those bit(s) as well).

Fill in the empty page table below to show its contents after the fourth access. If there is more than one possibility for a translation entry, list all possibilities. Use the following definitions for the Valid and Dirty bits: 0 = characteristic not true, 1= characteristic is true, X = table entry is correct with either a 0 or 1 for this bit, and ? = "given the limited degree of knowledge about this virtual memory system design, the value of this bit must be only one of 0 or 1 but the actual value cannot be determined."

Page	Starts at frame physical address	Present	Dirty
0			
1			
2			
3			
4			
5			
6			
7			

Answer: The contents of a page table is updated on a page fault (miss) to reflect the presence of the new page in DRAM. When a miss occurs, a page must be brought in to DRAM and,

perhaps a page must be moved out. Physical DRAM in this problem contains 4 page frames, and the initial page table shows that all four frames are occupied. We do not know much about the metadata in the page table initially. This following table shows everything known at the start, before making the first of the 4 translations:

Page number	Starts at frame physical address	Present	Dirty
0	0xC00	1	?
1	0x400	1	?
2	Not present in DRAM memory	0	X
3	Not present in DRAM memory	0	X
4	0x800	1	?
5	Not present in DRAM memory	0	X
6	0x000	1	?
7	Not present in DRAM memory	0	X

Now comes the first translation of a CPU access to virtual address 0x0000, with page number of 0, so this is a hit. The translation of page 0 to page frame 3 at physical address 0xC00 is used but not changed as a result of performing the translation. The Present (valid) bit remains set. We don't know whether this access is a read or a write (LOAD or STORE) so we cannot improve our knowledge of the Dirty bit value. Thus this Hit access to the table for an unknown access type (read or write?) results in no change to the table contents.

Now access 0x0E90, which is a virtual address within the range for page number 3. The page table says page number 3 is "Not present in DRAM memory." We have a page fault and all physical memory page frames are in use. A page must be ejected from DRAM. Also, the question statement says that this is the only page fault that will occur. To ensure this, we must eject a page from DRAM that is not used in the remaining two address translations. This excludes the virtual page numbers 0 (from access at 0x03FF) and 1 (access at 0x0400). Thus, the candidates to eject from DRAM are pages that reside in other frames than 0xC00 and 0x400, i.e., frames 0x800 and 0x000 contain pages that could be ejected without causing an additional page fault. So, now the table looks like the following.

The translation for virtual page 3 has been updated to show that page has been placed in either frame 0x800 or 0x000. Its metadata become Present=1 and Dirty=? (unknown becauseit is unknown whether this access is a write or not). For the two pages that could have been ejected from DRAM, 4 or 6, both of their translations have been updated to show their possible ejection ("Replaced or still at 0x800" and "Replaced or still at 0x000"). Their metadata has been adjusted to conform to their uncertain state by setting Present=?. Nothing new has been learned about their Dirty status, so that indication remains marked ?.

Page number	Starts at frame physical address	Present	Dirty
0	0xC00	**1**	?
1	0x400	1	?
2	Not present in DRAM memory	0	**X**
3	**0x800 or else 0x000**	1	?
4	Replaced or still at 0x800	?	?
5	Not present in DRAM memory	0	X

6	Replaced or still at 0x000	?	?
7	Not present in DRAM memory	0	X

The remaining translations of 0x03FF and 0x0400 are hits, are unknown as to access type (read or write), and so have no effect on the contents of the page table. The above is the final answer.