Computer Architecture Project 2 - Cache Simulation

Spring 2018, 10% of Course Grade

Completion Date: March 22 (But working on this before the midterm will help a lot!)
10% deduction for each week after that. Not accepted more than two weeks late.
email address to send it : hendrick@bu.edu AND kspalmer@bu.edu.
Please put "CS472"in the subject line to avoid it getting overlooked! No EXEs.

****
* **SHARE IDEAS, BUT NOT CODE. NOT *ONE* LINE! BASING YOUR WORK ON** *
* **SOMEONE ELSE'S WILL LIKELY GET YOU AN "F" FOR THE COURSE. DON'T USE** *
* **OTHERS WORK "TO GET AN IDEA OF HOW TO DO IT" OR FOR ANY OTHER** *
* **REASON. NO EXCUSES. NO EXCEPTIONS. NO POSTING SOLUTIONS, EVER!** *
****

1) OVERVIEW

You are to design and implement a software simulation of a cache memory subsystem. You can use any language. Note that any real cache is invisible to software. This will be a simulation of the way that a real cache works. Your "cache" will be a software structure/class that will contain information similar (but on a smaller and more simplified scale) to what would be in a real cache. Your "main memory" will be a 2K array. You can make it an array of integers or shorts (16-bits), but because it is simulating a byte-addressable memory, you won't be putting any value larger than 0xFF (255 decimal or 11111111 binary) in it.

You will provide for these two areas by defining an array for main memory and a structure/class/record (or array of these) for the cache. For example,

```
short Main_mem[2048];
```

So, in effect, your Main_mem array will be the "pretend" main memory area, your cache structure/class which you define will be the "pretend" cache, your program will pretend to be the intelligence of the memory sub-system, and requests that you type in will be the equivalent of a processor making requests of the memory subsystem.

You will implement a direct-mapped, write-back cache. The block size will be 16 bytes and you'll have 16 slots. Both of these numbers are smaller than a typical real cache, but make for a more manageable simulation. You should use a structure/class to manage each slot, where the structure consists of the appropriate information (e.g., a valid flag, tag, etc.) and you eventually have an array of these structures. (I'm not dictating that you MUST implement the cache in this exact way, but something close to this use of structures should make the most sense.)

2) INITIALIZATION

You MUST initialize your main memory (MM) in such a way that you'll be able to know whether or not you have the original value or not and also such that you'll be able to tell one byte from another. To achieve this, I want you to assign the value 0 to MM[0], 1 to MM[1], and so on until 0xFF to MM[0xFF]. 0xFF is the biggest value you can put in a byte, so then you should resume putting 0, 1, 2, and so on into MM elements x100, x101, and x102 (MM[x100] = 0, MM[x101] = 1, and MM[x102] = 2) until you've initialized the entire main memory array (MM[0x7FF] = 0xFF).

You should also initialize your cache to all zeros so that it will be easy (from a debugging and correcting point of view) to see which slots in the cache have something in them and which are still empty.

USE HEX FOR ALL ADDRESSES AND VALUES. This is primarily just an issue for input/output and an occasional constant. (A variable that has the decimal value 10 ALSO has the hexadecimal value 0A and the binary value 00001010 -- they are all the same number. It's just a matter of whether you input/output it as a decimal, hex or binary number.)

Using hex numbers will actually make your life easier. Until you perform a write to an address, you will know that a read of address x7AE should get the value xAE, a read of the address x422 should get the value x22, and so on. You wouldn't be able to immediately know if you had the correct value if you were typing in or printing out decimal values. (For example, x7AE is 1966 in decimal and xAE is x174 in decimal. It'll be obvious to you that address x7AE must contain xAE in it after Main_Mem has been initialized, but there'd be no way of telling that address 1966 decimal should have a decimal value of 174.)

Ideally, you can make an input file of the test case listed below and read that in, but if you aren't comfortable with that, feel free to again embed the data inside the program itself (even though that isn't good programming practice). Remember the lessons you learned in project one - you don't have to "convert" numbers to hex; just use the appropriate I/O or assignment statements.

3) BITWISE ANDS

I do NOT want you using the modulo function to extract the appropriate fields from the addresses. I want you using bitwise operations and shifts instead since they are much closer to what actually happens in hardware. Although you used bitwise ANDs and shifts in the first project, I offer the following example in C. It is based on a 32-bit address, a 32-byte block size (32 = 2 to the 5^{th}) and 8K slots (8K = 2 to the 13^{th}). This gives a 5-bit block offset in the least significant bits of the address, a 13-bit slot number in the next-least-significant bits and a 14-bit tag in the remaining most significant bits. You'll need to adjust the numbers for the specifics of this project, but the example combined with your experience in the first project should make it clear how to do that.

```
Block_offset = address & 0x0000001F;            /*Zero out all but the last 5 bits*/
Block_begin_addr = address & 0xFFFFFFE0; /*Zero out last 5 bits to find the block's first address*/
Tag = address >> 18;                            /* Shift off the block offset and slot # bits, moving the tag
                                                into the least significant bit positions */
Slot_num = (address & 0x0003FFE0) >> 5;      /* First zero out the tag and block offset bits, leaving
                                                behind just the bits for the slot number.  Then shift the
                                                slot number bits into the least significant bit positions.*/
```

Again, your project's characteristics differ from this example, so you'll have to change some of the values. (I want you to think this through and understand what you're doing rather than just copy something that I've done.) But this should be a big head start for you.

4) SUPPORTED OPERATIONS

You will need to support three types of requests from the screen once you have initialized your "main memory" and "cache" : Read byte, Write byte, and Display cache.

If a Read is requested, then you should ask what "address" in your "main memory" the Read is for. Note that references to a memory location 7ae, for example, are not to the physical memory location 7ae in your computer's memory, but rather to byte 7ae in your Main_mem array.

You should then simulate a real-life cache by checking to see if that address is in your cache. If it is, display the value you found for it in cache AND indicate that a cache hit took place. If not, go to your main memory instead, bring the entire block into your cache, display the value found AND indicate a cache miss. You **MUST** show whether it was a cache hit or miss and (on reads) what value you read out of the cache to receive full credit.

In other words, act like a cache acts. So, for example, the screen could look like the following, where the second and fourth lines are typed by the user (or provided by an input file or embedded in the program).

```
(R)ead, (W)rite, or (D)isplay Cache?
R
What address would you like read?
7ae
At that byte there is the value  ae (Cache Hit)
```

Writes would be very similar, except that the user must also indicate what the data to be written is. For example, (lines 2, 4, and 6 are typed by the user)

```
(R)ead, (W)rite, or (D)isplay Cache?
W
What address would you like to write to?
562
```

What data would you like to write at that address?
2f
Value 2f has been written to address 562. (Cache Hit)

(The previous two examples indicated a cache hit, but if there was a cache miss, you'd output "Cache Miss" instead of "Cache Hit" while still showing the value.) Note on writes: If a write is to be performed on a block that is not already in the cache, you must bring the entire block into the cache, perform the write and consider it a cache miss.

Display Cache will display the contents of each slot in the cache. For example, if the first few reads were for hex addresses 7A2, 2E, 2F and 3D5, those addresses would map to slot numbers A, 2, 2 and D, respectively. (Work out the numbers so you see why that is.) So you would display the following:

(R)ead, (W)rite, or (D)isplay Cache?
D

Slot	Valid	Tag	Data
0	0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1	0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2	1	0	20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
3	0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4	0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
5	0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
6	0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
7	0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
8	0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
9	0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
A	1	7	A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
B	0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
C	0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
D	1	3	D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
E	0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
F	0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

This does not include any information that might be needed to handle writes. If any such information is also needed in the cache, add it and show it when you perform a Display Cache.

Of course, as you perform more operations, more of the cache will be filled in.

Note that the data in the cache really is data, not addresses. We are not bringing addresses into the data part of the cache, but rather the values at those addresses. I've merely supplied a pattern to those values so that you (and I) can easily determine if the correct data is in the cache.

For example, at address 5AE the value is not 5AE it is AE. And AE is the value (along with the rest of the bytes in that block) that gets copied into the cache. Of course, your program shouldn't depend on that pattern, but actually get the value out of Main_mem otherwise it would break as soon as a new pattern was chosen.

IMPORTANT: Use the sequence just mentioned -- reads for addresses 7A2, 2e, 2f and 3d5 -- as a test of your simulation before you move on to the rest of the actual test you will turn in. If a Display Cache following those reads doesn't look EXACTLY like what is above (except for any extra information to handle writes that may or may not be needed), then you have a problem that you'll need to fix before moving on to the actual test to be turned in. (It's okay if the formatting is a little different, but otherwise your values should be consistent with the above.)

4) TESTING AND WHAT TO TURN IN

You should turn in a copy of your source code and also an output file which shows the results of the following test. (After you are finished debugging, generate your output file, which you can then turn in with the source. It doesn't matter how you generate the output file. Do a screen capture if you must.

IF THERE ARE ANY DISCREPANCIES BETWEEN YOUR SOURCE CODE AND THE OUTPUT, YOU WILL BE EXPECTED TO LET ME OR THE TA PERSONALLY TEST YOUR PROGRAM.

Don't bother with paper copies of your project. Just email the files to the addresses specified on page 1.

You should perform exactly the following sequence of tests on your cache simulator and turn in the output with your source code. Please no variations to cover up bugs in your program. :-)

Keep in mind that this is not a programming course, but one in Computer Architecture. You are assumed to already know programming before you entered this course. As a result, the focus of attention will not be on how clean your code is, but on how accurately it acts like a cache. If you perfectly program a flawed understanding of how caches work, you'll be graded according to your flawed understanding.

Note also that you have ONE chance to turn the program in. You should act as your own tester, making sure that the correct results are the ones you have in fact achieved. Don't expect to resubmit a program after the TA or I have pointed out its errors. If, however, you find errors before the TA has corrected your project, you are welcome to resubmit.

I may request a resubmission for errors not having to do with your understanding of caches, such as a failure to follow these directions (e.g., not following the test exactly, not using hex for the operations). For any of those resubmissions that are allowed, you will be deducted 10 points. Take pride in your work.

In the following test, I add some comments to provide clarity but, of course, you'll just enter the operations. Note that I've shown the numbers in hex. You must do the same.

Be sure that after each read you output what value your "simulated memory subsystem" is providing as a result of the read, either from your simulated cache or your simulated main memory. It should also indicate if the read resulted in a cache hit or miss.

(Feel free to make an input file out of this or build it into your program so you don't have to retype it each time.)

Operation	Comments
R	
5	
R	
6	
R	
7	
R	
14c	
R	
14d	
R	
14e	
R	
14f	
R	
150	
R	
151	
R	
3A6	
R	
4C3	

D	**Display** the Cache. Be sure that all the blocks you have brought in show up and be especially careful that ALL of the bytes for that block are in the cache.

W	Write the value 99 hex to address 14C. If the block is not in the cache, bring it in on
14C	writes as you would for a read.
99	
W	
63B	
7	

R
582

D **Display** the cache

R
348
R
3F

D **Display** the cache

R
14b
R
14c
R
63F
R
83

D **Display** the cache one last time, making sure that the correct blocks are there.