HW7.1. Filling the cache

Feel free to check out the guide that we have prepared to help you in this problem.

Consider an 11-bit addressable memory with the following contents: memory contents.txt

Your task would be to fill in the cache contents after a specific set of RISC-V instructions is run. Note that RISC-V is <u>little endian</u>.

For the valid and tag bits, write your answers in binary without the 0b prefix. For the data field, write your answers in hexadecimal without the 0x prefix.

As an example, let's say we have a 16-byte, 8-bytes per block, direct-mapped, write-back cache. When we execute $\frac{1}{2}$ $\frac{$

Row	Valid	Dirty	Tag	111	110	101	100	011	010	001	000
0	0	X	XXXXXXX	XX							
1	1	1	1100001	AD	E1	00	00	61	C0	61	C0

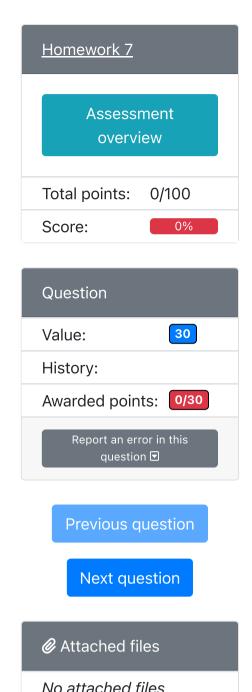
From the example, we observe the following:

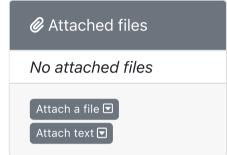
- 1. Since the memory address is 11 bits, and given the cache configuration stated earlier: Tag = 7 bits, Index = 1 bit, Offset = 3 bits. The 11-bit memory address is then partitioned accordingly, following the specified order (i.e. tag bits are the upper 7 bits, offset bits are the last 3 bits). Address 0x61C corresponds to index = 1.
- 2. Since the cache block at index 0 is not yet filled, valid bit = 0, contents are marked as X for every digit intended to be there. All the fields in the cache does not matter if the valid bit is 0 (i.e. no valid data has been loaded in that block yet), that's why we put X.
- 3. Even though we are just modifying a half-word (2 bytes) using sh at address 0x61C, we are getting 8 bytes from the memory (from offset 000 to offset 111) because the cache block size is 8 bytes. These 8 bytes correspond to bytes from address 0x618 to 0x61F (these addresses have the last 3 bits set to 000 and 111 respectively). Effectively, words from address 0x618 and 0x61C, from the provided text file, correspond to the 8 bytes that are loaded into the cache.
- 4. Following the little endian convention, the word at address $0 \times 61C$ is $0 \times ADE1B055$ (check the provided text file) where byte $0 \times AD$ corresponds to byte address $0 \times 61C + 3 = 0 \times 61F$. In the cache block, $0 \times AD$ will appear at the 111 column since the last 3 bits of $0 \times 61F$ is 111. The pattern then continues. The word at address 0×618 is $0 \times 61C061C0$ where leftmost (most significant) byte 0×61 corresponds to byte address $0 \times 618 + 3 = 0 \times 61B$. In the cache block, 0×61 will appear at the 0×61 column since the last 3 bits of $0 \times 61B$ is $0 \times 61B$.
- 5. Since we are storing a half-word from $\times 0$ (which is just 0×0000) at address $0 \times 61C$ (last 3 bits of the address is 100, so we are looking at that corresponding column), we overwrite the original 0×8055 byte and make it 0×0000 instead. We replaced two bytes starting from $0 \times 61C$ (the target address) and $0 \times 61C + 1 = 0 \times 61D$ (which corresponds to the 101 column). Since the data in the cache is not equal to the data in the original memory from the provided text file, dirty bit = 1 for that cache block.

Make sure you understood what is happening in the example and how the fields were filled in before proceeding.

Now it's your turn, let's say we run the following RISC-V code:

```
lb s0, 0x161(x0)
lh s1, 0x16A(x0)
lw s2, 0x61C(x0)
lb s3, 0x16B(x0)
lw s4, 0x298(x0)
sh x0, 0x61A(x0) # this is a store instruction
lh s6, 0x162(x0)
lb s7, 0x282(x0)
```





Let's say we have a <u>32-byte, 8-bytes per block, direct-mapped, write-back cache.</u> **The cache starts out empty.** If the corresponding cache block is invalid (i.e. no valid data was placed on the cache block), then put X corresponding to the number of digits intended to be there (similar to what was done in the example earlier).

Fill in the table below with the corresponding cache contents after the code above was run.

The rows correspond to the index. Again, <u>memory addresses are 11 bits</u>. You'll need to refer to the provided text file to know what the corresponding memory contents are.

Tip: It might be helpful to just consider the tag/index/offset bits during the code execution. The corresponding data can be filled in later since you can identify the bytes you need from the memory given the tag/index/offset bits.

Note: The textbox might be too short to contain the long tag bits, just make sure you are putting the correct number of bits and be careful when putting them in the table.

You might encounter a PrairieLearn bug that will prevent you clicking on a textbox (especially towards the rightmost columns). If this happens, just press Tab on your keyboard to access the next textbox.

Row	Valid	Dirty	Tag	111	110	101	100	011	010	001	000
0											
	2	0	2	2	•	•	?	?	3	3	?
1											
	•	•	•	•	•	•	•	•	?	?	?
2											
	?	2	2	?	?	?	?	?	0	0	?
3											
	•	•	•	•	•	•	?	•	?	•	?

Now let's say we replaced the direct-mapped cache with a <u>fully-associative cache with a</u>
<u>Least Recently Used (LRU) replacement policy with the same capacity and same write policy</u>

LRU means that the cache block that was accessed least recently would be the one that will be replaced if needed. For a fully-associative cache, block replacements will occur once the cache is full.

What would be the cache contents (again, the **cache also starts out empty**) after the same code was run?

Since a fully-associative cache does not have an index, fill up the cache contents starting from row 0 then going down, to be consistent with the staff solution.

Tip: Take note that the number of tag bits will change with the absence of the index bits.

Row	Valid	Dirty	Tag	111	110	101	100	011	010	001	000
0											
	•	•	?	?	•	?	•	•	•	?	?
1											
	•	•	•	•	?	•	•	•	•	•	?
2											
	?	•	•	•	?	?	?	9	?	?	•