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 little endian.

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 `sh x0 0x61C(x0)`, the cache (which started out empty) would look like the following: *(This is already a hint. Check the memory contents provided earlier and check for the correctness of this example)*

Row	Valid	Dirty	Tag	111	110	101	100	011	010	001	000
0	0	X	XXXXXXX	XX	XX	XX	XX	XX	XX	XX	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 `0x61C` is `0xADE1B055` (check the provided text file) where byte `0xAD` corresponds to byte address `0x61C + 3 = 0x61F`. In the cache block, `0xAD` will appear at the `111` column since the last 3 bits of `0x61F` is `111`. The pattern then continues. The word at address `0x618` is `0x61C061C0` where leftmost (most significant) byte `0x61` corresponds to byte address `0x618 + 3 = 0x61B`. In the cache block, `0x61` will appear at the `011` column since the last 3 bits of `0x61B` is `011`.

5. Since we are storing a half-word from `x0` (which is just `0x0000`) at address `0x61C` (last 3 bits of the address is `100`, so we are looking at that corresponding column), we overwrite the original `0xB055` byte and make it `0x0000` instead. We replaced two bytes starting from `0x61C` (the target address) and `0x61C + 1 = 0x61D` (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 32-byte, 8-bytes per block, direct-mapped, write-back cache. **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, memory addresses are 11 bits. 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
0	1 ✓100%	0 ✓100%	010100 ✓100%	B0 ✓100%	BA ✓100%	CA ✓100%	FE ✓100%	DE ✓100%	AD ✓100%
1	1 ✓100%	0 ✓100%	001011 ✓100%	12 ✓100%	34 ✓100%	56 ✓100%	78 ✓100%	00 ✓100%	AB ✓100%
2	0 ✓100%	X ✓100%	XXXXXX ✓100%	XX ✓100%	XX ✓100%	XX ✓100%	XX ✓100%	XX ✓100%	XX ✓100%

Homework 7

Assessment overview

Total points: 30/100

Score: 30%

Question

Value: 30

History: 30

Awarded points: 30/30

Report an error in this question

Previous question

Next question

Attached files

No attached files

Attach a file

Attach text

3	1 ✓100%	1 ✓100%	110000 ✓100%	AD ✓100%	E1 ✓100%	B0 ✓100%	55 ✓100%	00 ✓100%	00 ✓100%

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

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
0	1 ✓100%	0 ✓100%	00101100 ✓100%	FF ✓100%	FF ✓100%	FF ✓100%	FF ✓100%	87 ✓100%	65 ✓100%
1	1 ✓100%	0 ✓100%	01010000 ✓100%	B0 ✓100%	BA ✓100%	CA ✓100%	FE ✓100%	DE ✓100%	AD ✓100%
2	1 ✓100%	1 ✓100%	11000011 ✓100%	AD ✓100%	E1 ✓100%	B0 ✓100%	55 ✓100%	00 ✓100%	00 ✓100%
3	1 ✓100%	0 ✓100%	01010011 ✓100%	CA ✓100%	FE ✓100%	D0 ✓100%	0D ✓100%	CA ✓100%	FE ✓100%

Try a new variant

Correct answer

Let's say we have a 32-byte, 8-bytes per block, direct-mapped, write-back cache. **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, memory addresses are 11 bits. You'll need to refer to the provided text file to know what the corresponding memory contents are.

Row	Valid	Dirty	Tag	111	110	101	100	011	010	001	000
0	`1`	`0`	`010100`	`B0`	`BA`	`CA`	`FE`	`DE`	`AD`	`BE`	`EF`
1	`1`	`0`	`001011`	`12`	`34`	`56`	`78`	`00`	`AB`	`CD`	`EF`
2	`0`	`X`	`XXXXXX`	`XX`	`XX`	`XX`	`XX`	`XX`	`XX`	`XX`	`XX`
3	`1`	`1`	`110000`	`AD`	`E1`	`B0`	`55`	`00`	`00`	`61`	`C0`

Parse the addresses first into bits and divide them into Tag/Index/Offset. Track what is happening every instruction.

```
lb s0, 0x161(x0) # 001011 00 001
```

Tag = 001011 at Index 00 (Row 0). Valid bit = 1. Dirty bit = 0 (this is a load). Get words from address 0x160 and 0x164 (covers bytes at address 0x160-0x167). 8-byte data (starting from offset 111 down to 000) would be 0xFFFFFFFF87654321.

```
lh s1, 0x16A(x0) # 001011 01 010
```

Tag = 001011 at Index 01 (Row 1). Valid bit = 1. Dirty bit = 0 (this is a load). Get words from address 0x168 and 0x16C (covers bytes at address 0x168-0x16F). 8-byte data (starting from offset 111 down to 000) would be 0x1234567800ABCDEF.

```
lw s2, 0x61C(x0) # 110000 11 100
```

Tag = 110000 at Index 11 (Row 3). Valid bit = 1. Dirty bit = 0 (this is a load). Get words from address 0x618 and 0x61C (covers bytes at address 0x618-0x61F). 8-byte data (starting from offset 111 down to 000) would be 0xADE1B05561C061C0.

```
lb s3, 0x16B(x0) # 001011 01 011
```

Tag = 001011 at Index 01 (Row 1). The same tag exists on Row 1 already. This is a hit.

```
lw s4, 0x298(x0) # 010100 11 000
```

Tag = 010100 at Index 11 (Row 3). This is a new tag. It replaces the one in Row 3 before. Get words from address 0x298 and 0x29C (covers bytes at address 0x298-0x29F). 8-byte data (starting from offset 111 down to 000) would be 0xCAFED00DCAFEBABE.

```
sh x0, 0x61A(x0) # 110000 11 010            # this is a store instruction
```

Tag = 110000 at Index 11 (Row 3). This was in the cache before but got replaced by the previous instruction, now we load it again. Get words from address 0x618 and 0x61C (covers bytes at address 0x618-0x61F). 8-byte data (starting from offset 111 down to 000) would be 0xADE1B05561C061C0. Next, we write 0x0000 starting at offset 010. Thus, the data would be 0xADE1B055000061C0 instead. Dirty bit = 1.

```
lh s6, 0x162(x0) # 001011 00 010
```

Tag = 001011 at Index 00 (Row 0). The same tag exists on Row 0 already. This is a hit.

```
lb s7, 0x282(x0) # 010100 00 010
```

Tag = 010100 at Index 00 (Row 0). This is a new tag. It replaces the one in Row 0 before. Get words from address 0x280 and 0x284 (covers bytes at address 0x280-0x287). 8-byte data (starting from offset 111 down to 000) would be 0xB0BACAFEDEADBEEF.

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

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.

Row	Valid	Dirty	Tag	111	110	101	100	011	010	001	000
0	1	0	00101100	FF	FF	FF	FF	87	65	43	21
1	1	0	01010000	B0	BA	CA	FE	DE	AD	BE	EF
2	1	1	11000011	AD	E1	B0	55	00	00	61	C0
3	1	0	01010011	CA	FE	D0	0D	CA	FE	BA	BE

Parse the addresses first into bits and divide them into Tag/Offset (note that there's no Index since this is fully-associative configuration). Track what is happening every instruction.

```
lb s0, 0x161(x0) # 00101100 001
```

Tag = 00101100. Put that in Row 0. Valid bit = 1. Dirty bit = 0 (this is a load). Get words from address 0x160 and 0x164 (covers bytes at address 0x160-0x167). 8-byte data (starting from offset 111 down to 000) would be 0xFFFFFFFF87654321.

```
lh s1, 0x16A(x0) # 00101101 010
```

Tag = 00101101. This is a new tag, put that in Row 1. Valid bit = 1. Dirty bit = 0 (this is a load). Get words from address 0x168 and 0x16C (covers bytes at address 0x168-0x16F). 8-byte data (starting from offset 111 down to 000) would be 0x1234567800ABCDEF.

```
lw s2, 0x61C(x0) # 11000011 100
```

Tag = 11000011. This is a new tag, put that in Row 2. Valid bit = 1. Dirty bit = 0 (this is a load). Get words from address 0x618 and 0x61C (covers bytes at address 0x618-0x61F). 8-byte data (starting from offset 111 down to 000) would be 0xADE1B05561C061C0.

```
lb s3, 0x16B(x0) # 00101101 011
```

Tag = 00101101. The same tag exists on Row 1 already. This is a hit.

```
lw s4, 0x298(x0) # 01010011 000
```

Tag = 01010011. This is a new tag, put that in Row 3. Get words from address 0x298 and 0x29C (covers bytes at address 0x298-0x29F). 8-byte data (starting from offset 111 down to 000) would be 0xCAFED00DCAFEBABE.

```
sh x0, 0x61A(x0) # 11000011 010            # this is a store instruction
```

Tag = 11000011. The same tag exists in Row 2 already. This is a hit. Next, we write 0x0000 starting at offset 010. Thus, the data would be 0xADE1B055000061C0 instead. Dirty bit = 1.

```
lh s6, 0x162(x0) # 00101100 010
```

Tag = 00101100. The same tag exists on Row 0 already. This is a hit.

```
lb s7, 0x282(x0) # 01010000 010
```

Tag = 01010000. This is a new tag but since the cache is already full, we need to replace a block. This replaces the one in Row 1 because that's the least recently used (LRU) block/row. Get words from address 0x280 and 0x284 (covers bytes at address 0x280-0x287). 8-byte data (starting from offset 111 down to 000) would be 0xB0BACAFEDEADBEEF.

Submitted answer 8 **correct: 100%**
Submitted at 2022-10-29 03:26:50 (PDT)
show

Submitted answer 7 **partially correct: 89%**
Submitted at 2022-10-29 03:24:41 (PDT)
show

Submitted answer 6 **partially correct: 80%**
Submitted at 2022-10-29 03:22:19 (PDT)
show

Show/hide older submissions