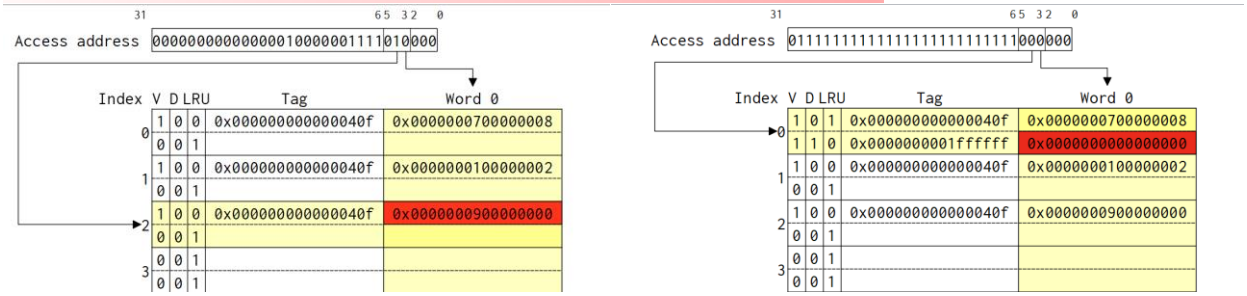
**Department of Computer Science**
**National Tsing Hua University**
**CS4100 Computer Architecture**
**Spring 2023 Homework 6**

Deadline: 2023/06/09 23:59

1. (15 points) Please run the simulation to find the following cases. For (a)~(c) case, please show the screenshots of (i) the load/store instruction at which you find the case, (ii) accessed memory address, and (iii) cache states before and after that load/store instruction. Also, please give a simple explanation of what happened for each case. You may follow the example question we made to answer the following cases.

   (a) (3 points) One case of cache insertion at an index (a set) with exactly one way already occupied. Don't forget to explain the change of LRU bits of both ways in that set.

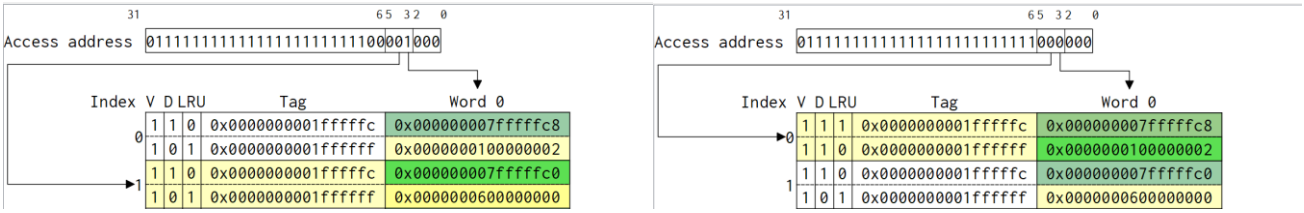
```
10344:          fcc43823          sd x12 -48 x8          MEM
```




With write allocate, we fetch the block on a write miss and there is an empty block in Set 0. After the write misses at index 0, the valid bit (V) and the dirty bit (D) will be 1, and the LRU bit will become 0 to represent that the block is the most recently used. In addition, the LRU bit of the originally existing data block will become 1 to represent that the block is not the most recently used data block.

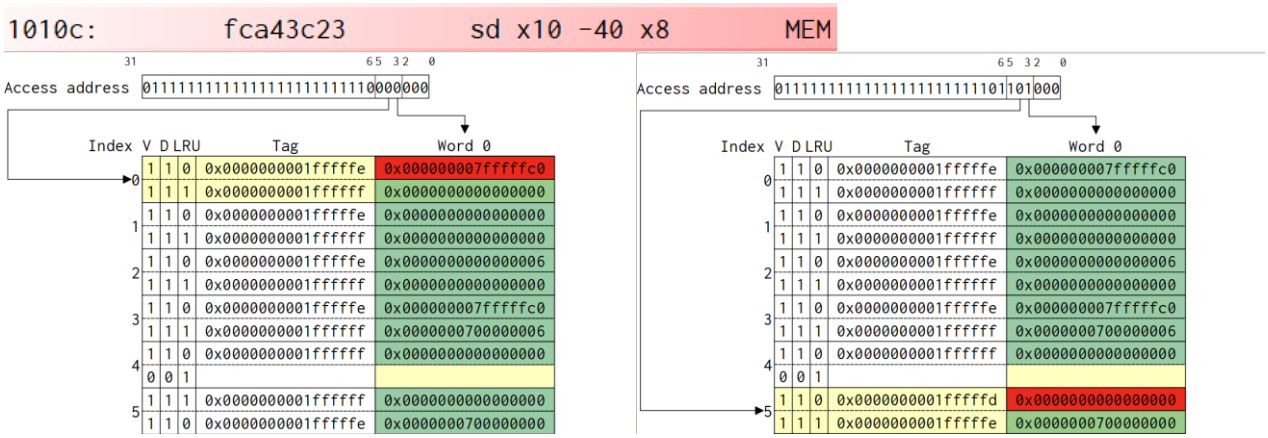
   (b) (3 points) One case of a hit that makes a block become dirty. (Not dirty before that hit)

```
100dc:          00e7a023          sw x14 0 x15          MEM
```



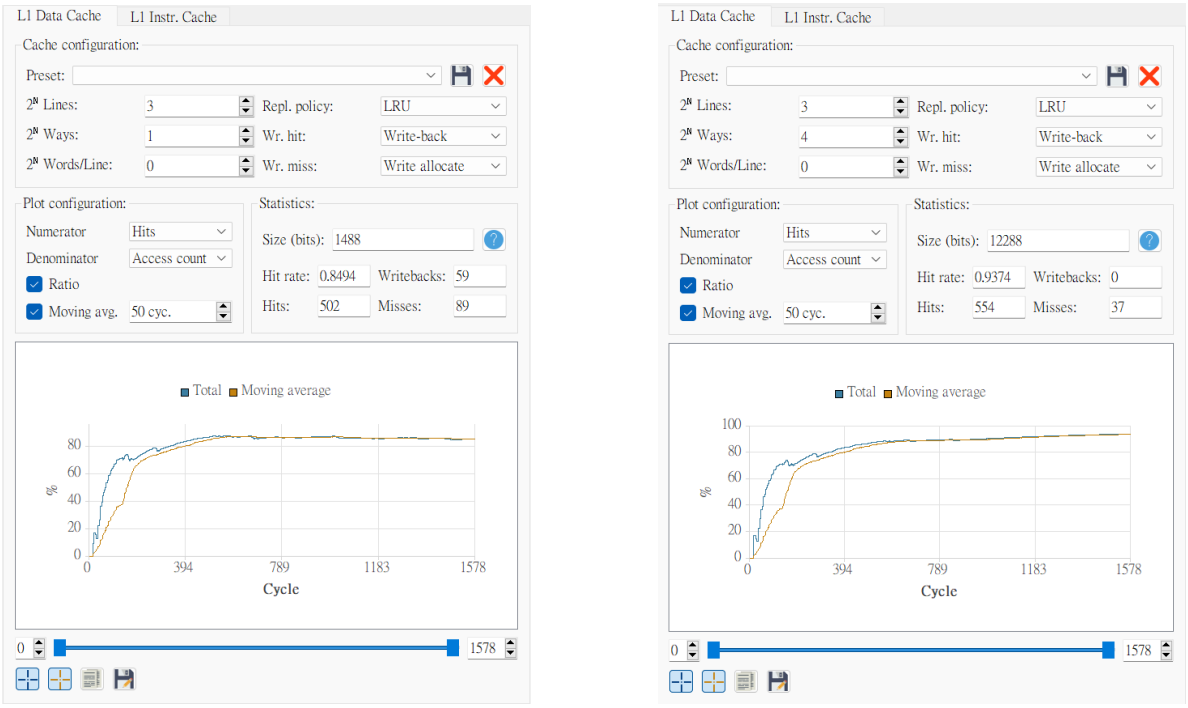

Because of write hit, we write into the block directly without write allocate. After the write hits at index 0, the dirty bit (D) will be 1, and the LRU bit will become 0 to represent that the block is the most recently used.

(c) (3 points) One case of a replacement with write-back needed.




Set 5 is full and the least recently use block with dirty bit 1 should be replace. Therefore, write-back is needed. With write allocate, we fetch the block on a write miss for replacement. After the write misses at index 5, the dirty bit (D) will be 1, and the LRU bit will become 0 to represent that the block is the most recently used.

(d) (6 points) Improve the hit rate by designing your own cache. You may adjust the associativity or the cache size. Please give some screenshots to show (i) the cache you design and (ii) the improvement of the hit rate. A brief discussion about why the cache you design makes the hit rate higher is also needed.




In order to reduce conflict misses, I set a higher associativity (Ways) of the cache. From the pictures above, we can see that the hit rate rises from 84.94% to 93.74%.

2. (12 points) Consider a 10-bit data, $D = \{d1, d2, d3, d4, d5, d6, d7, d8, d9, d10\}$, to be protected by using a Hamming single error correcting code with an additional parity bit, $p11$, so that double errors can be detected.

   (a) (2 points) Let the encoded codeword $C = \{p1, p2, d1, p4, d2, d3, d4, p8, d5, d6, d7, d8, d9, d10, p11\}$. Show the encoding process of $D$ =1001011001 to obtain the encoded codeword $C$.

| |
|---|
| Set p1 to make bits 1, 3, 5, 7, 9, 11, 13 even parity. |
| 1-1-001-011001- |
| Set p2 to make bits 2, 3, 6, 7, 10, 11, 14 even parity. |
| 111-001-011001- |
| Set p4 to make bits 4, 5, 6, 7, 12, 13, 14 even parity. |
| 1110001-011001- |
| Set p8 to make bits 8, 9, 10, 11, 12, 13, 14 even parity. |
| 11100011011001- |
| Set p11 to make all bits even parity |
| 111000110110010 |

C = 111000110110010

   (b) (2 points) Assume no error occurs. Show the decoding process of $C$ (i.e., locating/correcting the Screenshots Explanation Instruction: Cache states before and after the write miss 3 single error, detecting double errors, or confirming that there is no error; you have to verify if the result is right or wrong) from (a).

| |
|---|
| Check bits 1, 3, 5, 7, 9, 11, 13 if they are even parity. |
| Yes => H = $h_8h_4h_2h_1$ = ---0 |
| Check bits 2, 3, 6, 7, 10, 11, 14 if they are even parity. |
| Yes => H = $h_8h_4h_2h_1$ = --00 |
| Check bits 4, 5, 6, 7, 12, 13, 14 if they are even parity. |
| Yes => H = $h_8h_4h_2h_1$ = -000 |
| Check bits 8, 9, 10, 11, 12, 13, 14 if they are even parity. |
| Yes => H = $h_8h_4h_2h_1$ = 0000 |
| Check all the bits if they are even parity. (By XOR) |
| Yes => $h_{11}$ = 0 |

H is 0 and h11 is 0. Therefore, there is no error.

(c) (2 points) From (a), suppose $d5$ of $C$ is inverted. Show the decoding process.

d5 is bit 9.

| |
|---|
| Check bits 1, 3, 5, 7, 9, 11, 13 if they are even parity. |
| No => H = $h_8h_4h_2h_1$ = ---1 |
| Check bits 2, 3, 6, 7, 10, 11, 14 if they are even parity. |
| Yes => H = $h_8h_4h_2h_1$ = --01 |
| Check bits 4, 5, 6, 7, 12, 13, 14 if they are even parity. |
| Yes => H = $h_8h_4h_2h_1$ = -001 |
| Check bits 8, 9, 10, 11, 12, 13, 14 if they are even parity. |
| No => H = $h_8h_4h_2h_1$ = 1001 |
| Check all the bits if they are even parity. (By XOR) |
| No => $h_{11}$ = 1 |

8 + 1 = 9. H is not 0 and h11 is 1. Therefore, there is a correctable single error of bit 9.

(d) (2 points) From (a), suppose $p1$ and $d8$ of $C$ are inverted. Show the decoding process.

p1 is bit 1 and d8 is bit 12.

| |
|---|
| Check bits 1, 3, 5, 7, 9, 11, 13 if they are even parity. |
| No => H = $h_8h_4h_2h_1$ = ---1 |
| Check bits 2, 3, 6, 7, 10, 11, 14 if they are even parity. |
| Yes => H = $h_8h_4h_2h_1$ = --01 |
| Check bits 4, 5, 6, 7, 12, 13, 14 if they are even parity. |
| No => H = $h_8h_4h_2h_1$ = -101 |
| Check bits 8, 9, 10, 11, 12, 13, 14 if they are even parity. |
| No => H = $h_8h_4h_2h_1$ = 1101 |
| Check all the bits if they are even parity. (By XOR) |
| Yes => $h_{11}$ = 0 |

H is not 0 and h11 is 0. Therefore, there are double errors.

(e) (2 points) From (a), suppose $p1$, $d8$, and $p11$ of $C$ are inverted after the encoding. Show the decoding process.

p1 is bit 1, d8 is bit 12, and p11 is bit 15.

| |
|---|
| Check bits 1, 3, 5, 7, 9, 11, 13 if they are even parity. |
| No => H = $h_8h_4h_2h_1$ = ---1 |
| Check bits 2, 3, 6, 7, 10, 11, 14 if they are even parity. |
| Yes => H = $h_8h_4h_2h_1$ = --01 |
| Check bits 4, 5, 6, 7, 12, 13, 14 if they are even parity. |
| No => H = $h_8h_4h_2h_1$ = -101 |
| Check bits 8, 9, 10, 11, 12, 13, 14 if they are even parity. |
| No => H = $h_8h_4h_2h_1$ = 1101 |
| Check all the bits if they are even parity. (By XOR) |
| No => $h_{11}$ = 1 |

8 + 4 + 1 = 13. H is not 0 and h11 is 1. Therefore, there is a correctable error. However, it is not true.

(f) (2 points) What is the minimum number of parity bits required to protect 128-bit data using the same method to construct a SECDED code?

In order to ensure that each data bit is covered by at least two parity bits, the minimum number of parity bits should be 8 ($2^7$ = 128, 0 ~ 7) + 1 (an additional parity bit) = 9.

3. (12 points) For a direct-mapped cache design with a 14-bit memory address, the following bit fields of the address are used to access the cache.

| Field | Tag | Index | Block Offset | Byte Offset |
|---|---|---|---|---|
| Bits | 13:9 | 8:4 | 3:2 | 1:0 |

(a) (2 points) What is the cache block size (in bytes)?

4 (byte offset) * 4 (block offset) = 16 bytes.

(b) (2 points) How many blocks does the cache have?

$2^5$ = 32 blocks.

(c) (2 points) What is the total cache size (in bytes)?

16 (block size) * 32 (# of blocks) = 512 bytes.

(d) (2 points) What is the actual SRAM memory size used to implement this cache in bytes? Please also consider the valid bit, dirty bit, and tag bits for each block.

valid bit + dirty bit + tag bits = 7 bits.

7 / 8 * 32 + 512 = 28 + 512 = 540 bytes.

(e) (2 points) With the same cache size in (c), derive the bit fields of the address for a 2-way set associative cache design.

| Field | Tag | Index | Block Offset | Byte Offset |
|---|---|---|---|---|
| Bits | 13:8 | 7:4 | 3:2 | 1:0 |

(f) (2 points) From (e), what is the actual SRAM memory size used to implement this cache in bytes? Please also consider the valid bit, dirty bit, and tag bits for each block; an extra reference bit for each set.

32 blocks / 2 ways = 16 sets

valid bit + dirty bit + tag bits = 8 bits.

8 / 8 * 32 + 512 + 1 / 8 * 16 = 546 bytes.

4. (8 points) Consider the cache architecture of a specific processor that its base CPI without memory stalls is 1. Suppose that the L-1 miss rate is 5%. The main memory access delay is 200 cycles. There are several options of L-2 and L-3 caches in the following table.

| Level | Name (Placement) | Access Cycle | Local Miss Rate |
|---|---|---|---|
| L-2 | L2-DM (Direct Mapped) | L2 access cycle = 16 | L2 Miss Rate = 4.0% |
| L-2 | L2-4WAY (4-way Set Associative) | L2 access cycle = 20 | L2 Miss Rate = 3.5% |
| L-3 | L3-8WAY (8-way Set Associative) | L3 access cycle = 50 | L3 Miss Rate = 2.0% |

Please note the following definitions:

- Local Miss Rate: the fraction of references to one level of a cache that miss. That is, the number of misses in a cache divided by the total number of memory accesses to that cache.
- Global Miss Rate: the fraction of references that miss in all levels of a multilevel cache. That is, the number of misses in the cache divided by the total number of memory accesses generated by the CPU.

(a) (5 points) Calculate the effective CPI for each possible multilevel (i.e., 2-level or 3-level) cache design. Which one provides the best performance based on the effective CPI?

| | |
|---|---|
| L2-DM | 1 + 0.05 * 16 + 0.05 * 0.04 * 200 = 2.2 CPI |
| L2-4WAY | 1 + 0.05 * 20 + 0.05 * 0.035 * 200 = 2.35 CPI |
| L2-DM + L3-8WAY | 1 + 0.05 * 16 + 0.05 * 0.04 * 50 + 0.05 * 0.04 * 0.02 * 200 = 1.908 CPI |
| L2-4WAY + L3-8WAY | 1 + 0.05 * 20 + 0.05 * 0.035 * 50 + 0.05 * 0.035 * 0.02 * 200 = 2.0945 CPI |

L2-DM + L3-8WAY provides the best performance based on the effective CPI.

(b) (3 points) From (a), consider the multilevel cache design with the best performance. Suppose you want to further reduce the effective CPI to 1.8 or lower by improving the L-2 cache with fewer access cycles (assuming the local miss rate remains the same). What are the maximum access cycles of this improved L-2 cache to meet this requirement?

1 + 0.05 * n + 0.05 * 0.04 * 50 + 0.05 * 0.04 * 0.02 * 200 <= 1.8 CPI, n <= 13.84

The maximum access cycles are 13.

5. (23 points) Consider that a computer P uses a 128-byte 2-way set associative cache as a primary cache. The cache adopts the write-back and write-allocate policies. Each block in the cache has 16 bytes. The physical byte address of the computer has 12 bits. The block replacement policy is LRU (Least-Recently Used).

   Suppose the cache receives the following memory accesses. Show the content of the cache after each memory access. Indicate if it is a read/write miss/hit for each access. Also, indicate if there occurs a block replacement, write allocate, or write back.

| No. | Read or Write | Address |
|---|---|---|
| 1 | Read | `0x340` |
| 2 | Read | `0x000` |
| 3 | Read | `0x1d8` |
| 4 | Write | `0x354` |
| 5 | Read | `0xa61` |
| 6 | Write | `0xa61` |
| 7 | Read | `0x3ec` |
| 8 | Read | `0xa62` |
| 9 | Read | `0x3ea` |
| 10 | Read | `0x422` |

Suppose initially the cache has one entry as shown in the following table. For each block (way), the **V** denotes the valid bit (V=1 indicates a valid block; V=0 or a blank field indicates an invalid block), and the **D** is the dirty bit (D=1 indicates a dirty block). For each set, the **R** denotes the reference bit (R=0 indicates that Way 0 is the least recently used between the two ways in the same set; R=1 indicates that Way 1 is the least recently used).

Additionally, **`Mem[0x000-00f]`** represents a data block of 16 bytes from **`Mem[0x000]`** to **`Mem[0x00f]`**. Note that the empty tag for the existing entry in Set 0 is also for you to fill in.

| | | Way 0 | | | | Way 1 | | | |
|---|---|---|---|---|---|---|---|---|---|
| | R | V | D | Tag | Data | V | D | Tag | Data |
| Set 0 | 1 | 1 | 0 | $000000_2$ | `Mem[0x000-00f]` | | | | |
| Set 1 | | | | | | | | | |
| Set 2 | | | | | | | | | |
| Set 3 | | | | | | | | | |

Read `0x340`: read miss, no block replacement, no write allocate, no write back

| | | Way 0 | | | | Way 1 | | | |
|---|---|---|---|---|---|---|---|---|---|
| | R | V | D | Tag | Data | V | D | Tag | Data |
| Set 0 | 0 | 1 | 0 | $000000_2$ | `Mem[0x000-00f]` | 1 | 0 | $001101_2$ | `Mem[0x340-34f]` |
| Set 1 | | | | | | | | | |
| Set 2 | | | | | | | | | |
| Set 3 | | | | | | | | | |

Read `0x000`: read hit, no block replacement, no write allocate, no write back

| | | Way 0 | | | | | Way 1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | R | V | D | Tag | Data | V | D | Tag | Data |
| Set 0 | 1 | 1 | 0 | $000000_2$ | Mem[0x000-00f] | 1 | 0 | $001101_2$ | Mem[0x340-34f] |
| Set 1 | | | | | | | | | |
| Set 2 | | | | | | | | | |
| Set 3 | | | | | | | | | |

Read `0x1d8`: read miss, no block replacement, no write allocate, no write back

| | | Way 0 | | | | | Way 1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | R | V | D | Tag | Data | V | D | Tag | Data |
| Set 0 | 1 | 1 | 0 | $000000_2$ | Mem[0x000-00f] | 1 | 0 | $001101_2$ | Mem[0x340-34f] |
| Set 1 | 1 | 1 | 0 | $000111_2$ | Mem[0x1d0-1df] | | | | |
| Set 2 | | | | | | | | | |
| Set 3 | | | | | | | | | |

Write `0x354`: write miss, no block replacement, write allocate, no write back

| | | Way 0 | | | | | Way 1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | R | V | D | Tag | Data | V | D | Tag | Data |
| Set 0 | 1 | 1 | 0 | $000000_2$ | Mem[0x000-00f] | 1 | 0 | $001101_2$ | Mem[0x340-34f] |
| Set 1 | 0 | 1 | 0 | $000111_2$ | Mem[0x1d0-1df] | 1 | 1 | $001101_2$ | Mem[0x350-35f] |
| Set 2 | | | | | | | | | |
| Set 3 | | | | | | | | | |

Read `0xa61`: read miss, no block replacement, no write allocate, no write back

| | | Way 0 | | | | | Way 1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | R | V | D | Tag | Data | V | D | Tag | Data |
| Set 0 | 1 | 1 | 0 | $000000_2$ | Mem[0x000-00f] | 1 | 0 | $001101_2$ | Mem[0x340-34f] |
| Set 1 | 0 | 1 | 0 | $000111_2$ | Mem[0x1d0-1df] | 1 | 1 | $001101_2$ | Mem[0x350-35f] |
| Set 2 | 1 | 1 | 0 | $101001_2$ | Mem[0xa60-a6f] | | | | |
| Set 3 | | | | | | | | | |

Write `0xa61`: write hit, no block replacement, no write allocate, no write back

| | | Way 0 | | | | | Way 1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | R | V | D | Tag | Data | V | D | Tag | Data |
| Set 0 | 1 | 1 | 0 | $000000_2$ | Mem[0x000-00f] | 1 | 0 | $001101_2$ | Mem[0x340-34f] |
| Set 1 | 0 | 1 | 0 | $000111_2$ | Mem[0x1d0-1df] | 1 | 1 | $001101_2$ | Mem[0x350-35f] |
| Set 2 | 1 | 1 | 1 | $101001_2$ | Mem[0xa60-a6f] | | | | |
| Set 3 | | | | | | | | | |

Read `0x3ec`: read miss, no block replacement, no write allocate, no write back

| | Way 0 | | | | | Way 1 | | | |
|---|---|---|---|---|---|---|---|---|---|
| | R | V | D | Tag | Data | V | D | Tag | Data |
| Set 0 | 1 | 1 | 0 | $000000_2$ | Mem[0x000-00f] | 1 | 0 | $001101_2$ | Mem[0x340-34f] |
| Set 1 | 0 | 1 | 0 | $000111_2$ | Mem[0x1d0-1df] | 1 | 1 | $001101_2$ | Mem[0x350-35f] |
| Set 2 | 0 | 1 | 1 | $101001_2$ | Mem[0xa60-a6f] | 1 | 0 | $001111_2$ | Mem[0x3e0-3ef] |
| Set 3 | | | | | | | | | |

Read `0xa62`: read hit, no block replacement, no write allocate, no write back

| | Way 0 | | | | | Way 1 | | | |
|---|---|---|---|---|---|---|---|---|---|
| | R | V | D | Tag | Data | V | D | Tag | Data |
| Set 0 | 1 | 1 | 0 | $000000_2$ | Mem[0x000-00f] | 1 | 0 | $001101_2$ | Mem[0x340-34f] |
| Set 1 | 0 | 1 | 0 | $000111_2$ | Mem[0x1d0-1df] | 1 | 1 | $001101_2$ | Mem[0x350-35f] |
| Set 2 | 1 | 1 | 1 | $101001_2$ | Mem[0xa60-a6f] | 1 | 0 | $001111_2$ | Mem[0x3e0-3ef] |
| Set 3 | | | | | | | | | |

Read `0x3ea`: read hit, no block replacement, no write allocate, no write back

| | Way 0 | | | | | Way 1 | | | |
|---|---|---|---|---|---|---|---|---|---|
| | R | V | D | Tag | Data | V | D | Tag | Data |
| Set 0 | 1 | 1 | 0 | $000000_2$ | Mem[0x000-00f] | 1 | 0 | $001101_2$ | Mem[0x340-34f] |
| Set 1 | 0 | 1 | 0 | $000111_2$ | Mem[0x1d0-1df] | 1 | 1 | $001101_2$ | Mem[0x350-35f] |
| Set 2 | 0 | 1 | 1 | $101001_2$ | Mem[0xa60-a6f] | 1 | 0 | $001111_2$ | Mem[0x3e0-3ef] |
| Set 3 | | | | | | | | | |

Read `0x422`: read miss, block replacement, no write allocate, write back

| | Way 0 | | | | | Way 1 | | | |
|---|---|---|---|---|---|---|---|---|---|
| | R | V | D | Tag | Data | V | D | Tag | Data |
| Set 0 | 1 | 1 | 0 | $000000_2$ | Mem[0x000-00f] | 1 | 0 | $001101_2$ | Mem[0x340-34f] |
| Set 1 | 0 | 1 | 0 | $000111_2$ | Mem[0x1d0-1df] | 1 | 1 | $001101_2$ | Mem[0x350-35f] |
| Set 2 | 1 | 1 | 0 | $010000_2$ | Mem[0x420-42f] | 1 | 0 | $001111_2$ | Mem[0x3e0-3ef] |
| Set 3 | | | | | | | | | |

6. (10 points) A byte-addressable virtual memory system has the following characteristics:
   - Each virtual address is 16 bits.
   - Each physical address is 14 bits.
   - The page size is 512 bytes.
   - Each process has a one-level page table, and each page table entry (PTE) is 4 bytes.
   - The TLB is directed-mapped and has 16 entries

   (a) (2 points) How many bits are required for the page offset?
   512 bytes = $2^9$ bytes => 9 bits are required.

   (b) (2 points) What is the maximum number of virtual pages a process could have?
   16 bits – 9 bits = 7 bits => $2^7$ = 128 => A process could have a maximum of 128 virtual pages.

   (c) (2 points) How many physical pages are there?
   14 bits – 9 bits = 5 bits => $2^5$ = 32 => There are 32 pages.

   (d) (2 points) What is the maximum page table size for a process?
   128 virtual pages * 4 bytes / PTE = 512 bytes
   => The maximum page table size for a process is 512 bytes.

   (e) (2 points) Assuming each entry of the TLB has a valid bit, a dirty bit, a reference bit, a tag, and a physical page number, what is the total number of bits in the TLB?
   1 (valid) + 1 (dirty) + 1 (reference) + 7 (tag = virtual page number) + 5 (physical page number) = 15
   => 15 (bits / entry) * 16 (entries) = 240 bits
   => The total number of bits in the TLB is 240.

7. (20 points) The following addresses constitute a stream of virtual byte addresses generated by a processor.

`0x5368, 0x02c3, 0x434b, 0x6812, 0xaf50`

Assume that the main memory has adequate space to accommodate the requested physical page brought from the disk without page replacement. Therefore, you can choose any page number as long as it is not used in the page table. Suppose that a page has 4KB. A four-entry fully associative TLB with the approximate LRU replacement policy (see the appendix) is utilized.
The initial TLB and page table states are listed below. Suppose the ***replacement pointer*** points to the first block.

| Valid | Tag | Physical Page Number | Reference (Used) | |
|---|---|---|---|---|
| 1 | 0x4 | 6 | 1 | ← Replacement Pointer |
| 1 | 0x1 | 2 | 0 | |
| 1 | 0xa | 3 | 1 | |
| 0 | 0x3 | 5 | 0 | |

| Index | Valid | Physical Page Number or in Disk |
|---|---|---|
| 0 | 0 | Disk |
| 1 | 1 | 2 |
| 2 | 0 | Disk |
| 3 | 1 | 5 |
| 4 | 1 | 6 |
| 5 | 1 | 11 |
| 6 | 1 | 7 |
| 7 | 0 | Disk |
| 8 | 0 | Disk |
| 9 | 0 | Disk |
| a | 1 | 3 |
| b | 0 | Disk |

For each access shown above, list whether the access is a hit or miss in the TLB, whether the access causes a page fault, and the updated state of the TLB.

Appendix: An approximate LRU can be implemented as follows:

- A ***used bit*** (initial 0) is associated with every block.
- When a block is accessed (hit or miss), its used bit is set to 1.
- On a replacement, a ***replacement pointer*** circularly scans through the blocks to find a block with its used bit = 0 to replace.
  - The replacement pointer points to the first block initially, and scans circularly through the blocks in a set during the operation.
- Along the way, the replacement pointer also resets the encountered used bits from 1 to 0.
  - Alternatively, if on an access, all other used bits in a set are 1, they are reset to 0 except the bit of the block that is accessed.

Page offset = 12 bits.

`0x5368`:

TLB miss, no page fault

| Valid | Tag | Physical Page Number | Reference (Used) | |
|---|---|---|---|---|
| 1 | 0x4 | 6 | 1 | ← Replacement Pointer |
| 1 | 0x1 | 2 | 0 | |
| 1 | 0xa | 3 | 1 | |
| 1 | 0x5 | 11 | 1 | |

`0x02c3`:

TLB miss, page fault

| Valid | Tag | Physical Page Number | Reference (Used) | |
|---|---|---|---|---|
| 1 | 0x4 | 6 | 0 | |
| 1 | 0x0 | 1 | 1 | |
| 1 | 0xa | 3 | 1 | ← Replacement Pointer |
| 1 | 0x5 | 11 | 1 | |

| Index | Valid | Physical Page Number or in Disk |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 1 | 2 |
| 2 | 0 | Disk |
| 3 | 1 | 5 |
| 4 | 1 | 6 |
| 5 | 1 | 11 |
| 6 | 1 | 7 |
| 7 | 0 | Disk |
| 8 | 0 | Disk |
| 9 | 0 | Disk |
| a | 1 | 3 |
| b | 0 | Disk |

`0x434b`:

TLB hit, no page fault

| Valid | Tag | Physical Page Number | Reference (Used) | |
|---|---|---|---|---|
| 1 | 0x4 | 6 | 1 | |
| 1 | 0x0 | 1 | 0 | |
| 1 | 0xa | 3 | 0 | ← Replacement Pointer |
| 1 | 0x5 | 11 | 0 | |

`0x6812`:

TLB miss, no page fault

| Valid | Tag | Physical Page Number | Reference (Used) | |
|---|---|---|---|---|
| 1 | 0x4 | 6 | 1 | |
| 1 | 0x0 | 1 | 0 | |
| 1 | 0x6 | 7 | 1 | |
| 1 | 0x5 | 11 | 0 | ← Replacement Pointer |

`0xaf50`:

TLB miss, no page fault

| Valid | Tag | Physical Page Number | Reference (Used) | |
|---|---|---|---|---|
| 1 | 0x4 | 6 | 1 | ← Replacement Pointer |
| 1 | 0x0 | 1 | 0 | |
| 1 | 0x6 | 7 | 1 | |
| 1 | 0xa | 3 | 1 | |