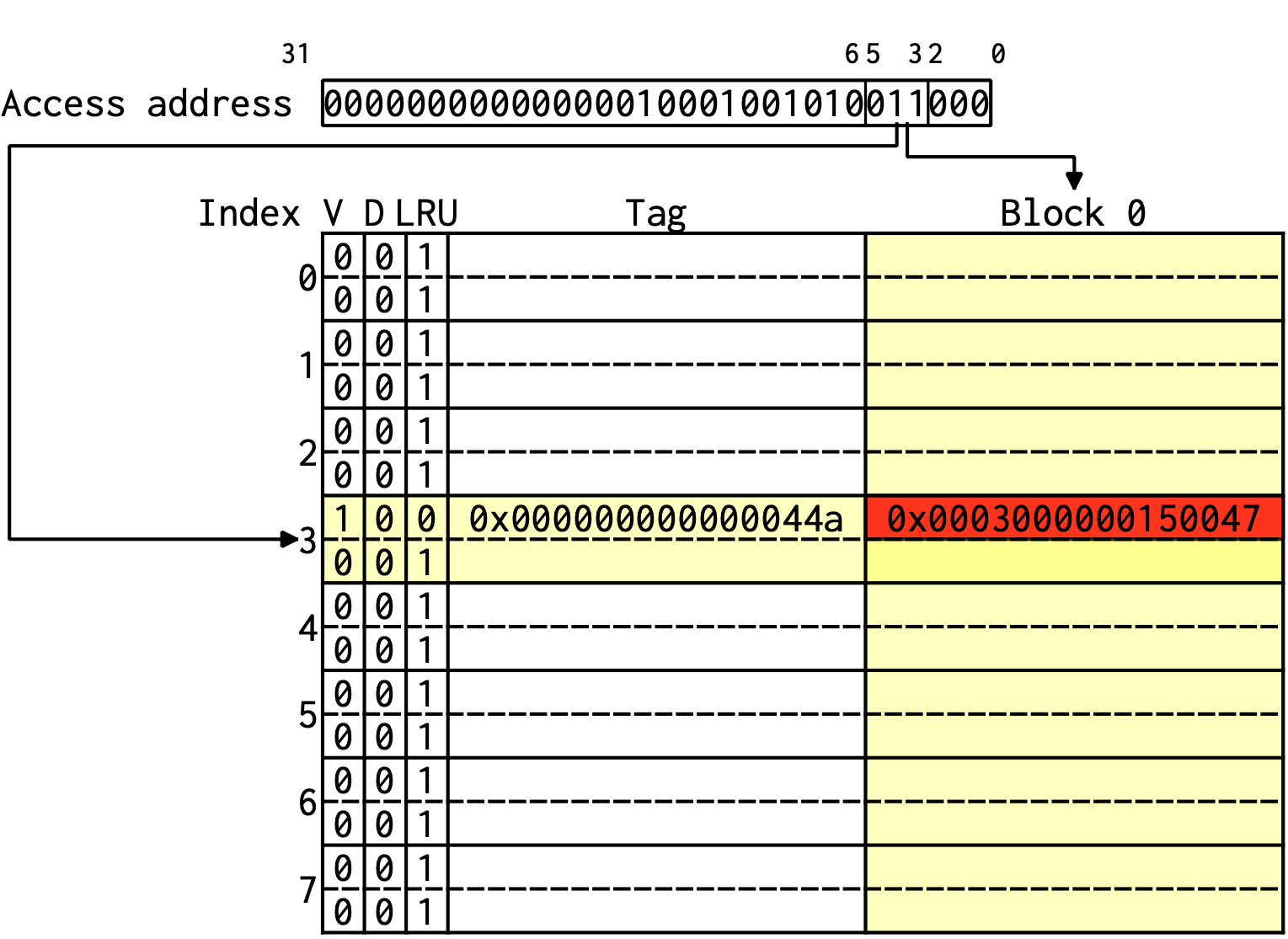
1. (30 points) Please run the simulation to find the following cases. For each case, please show the screenshots of (i)the load/store instruction at which you find the case, (ii)accessed memory address, and (iii)cache states before and after that load/store instruction. Also, please give a simple explanation of what happened for each case. You may follow the example question we made to answer the following three cases.

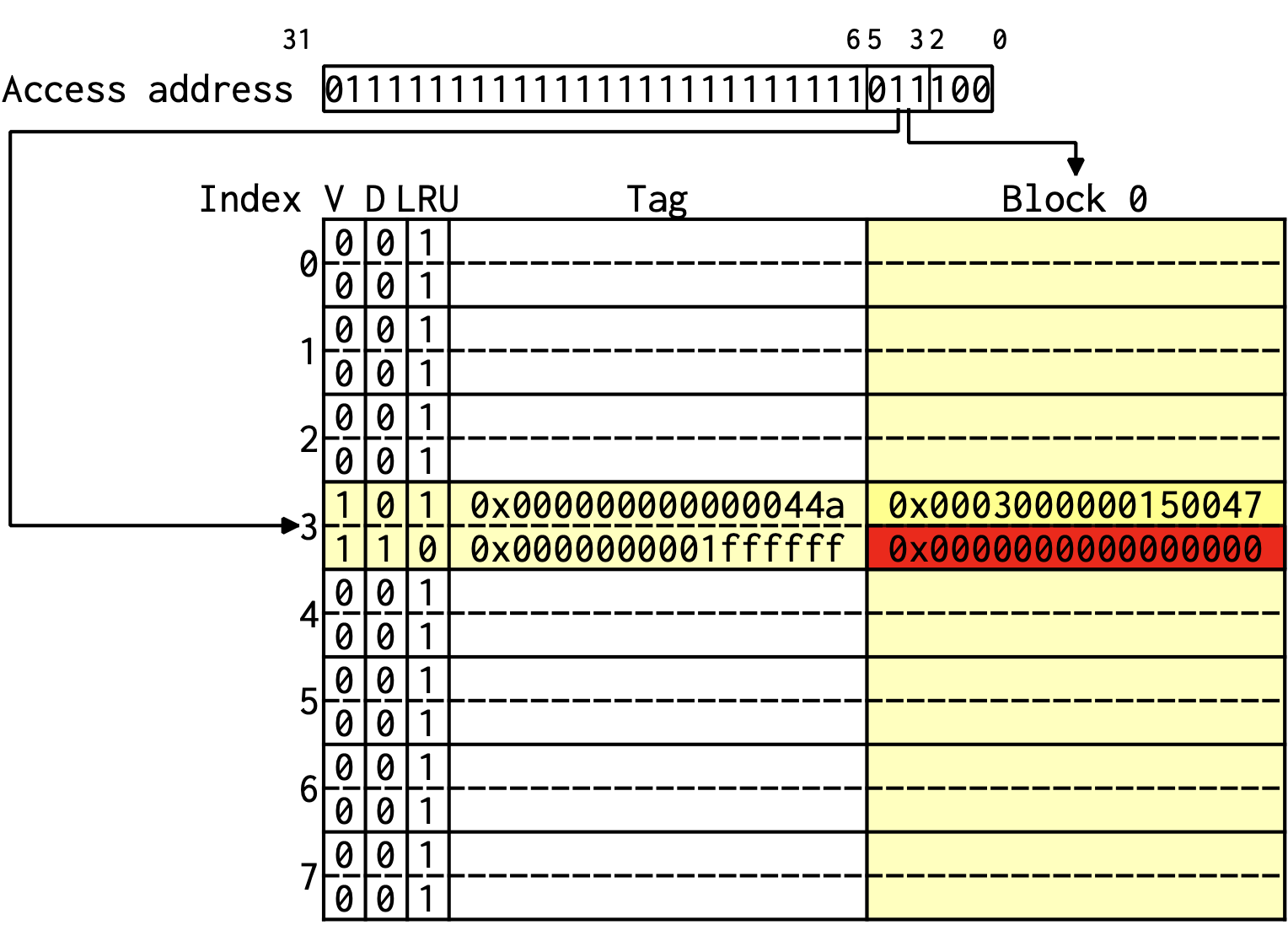
1. (10 points) One case of cache insertion at an index (a set) with exactly one way already occupied. Don’t forget to explain the change of LRU bits of both ways in that set.



**Image1.1.1: the store instruction I find the case**

****

**Image1.1.2: cache state before the instruction**



**Image1.1.3:accessed memory address & cache state after the instruction**

**Explanation:**

**With write allocate, we fetch the block on a write miss.**

**After the write miss at index 3, the valid bit(V) and dirty bit(D) are set to 1, and the LRU bit will become 0 to represent that the block is the most recently used, just like what Image1.1.3 shows.**

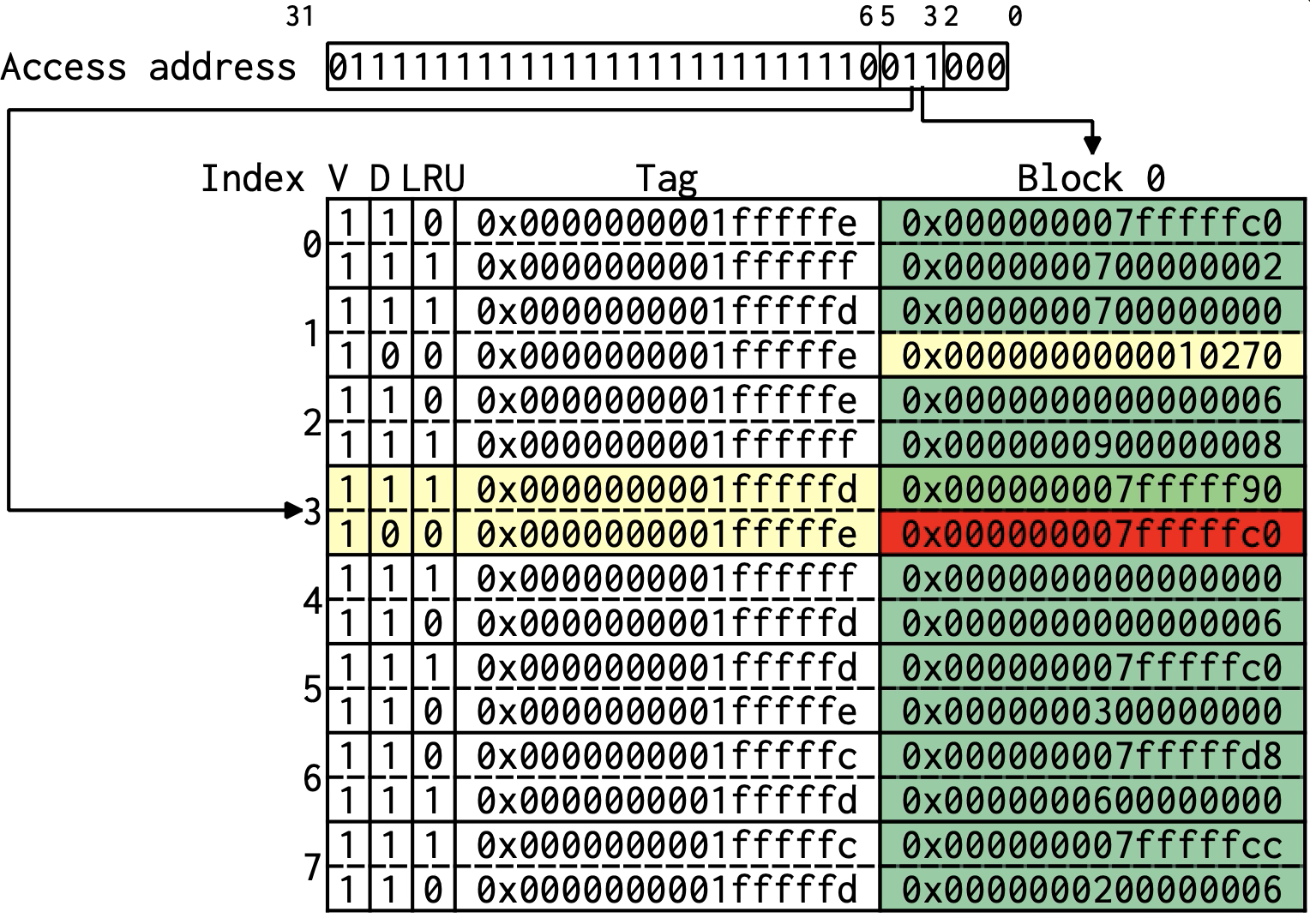
**What’s more, by Image1.1.2 and 1.1.3, we can see that in the 4th set(with index3), there’s exactly one way already occupied.**

**After the store instruction, the other way is occupied.**

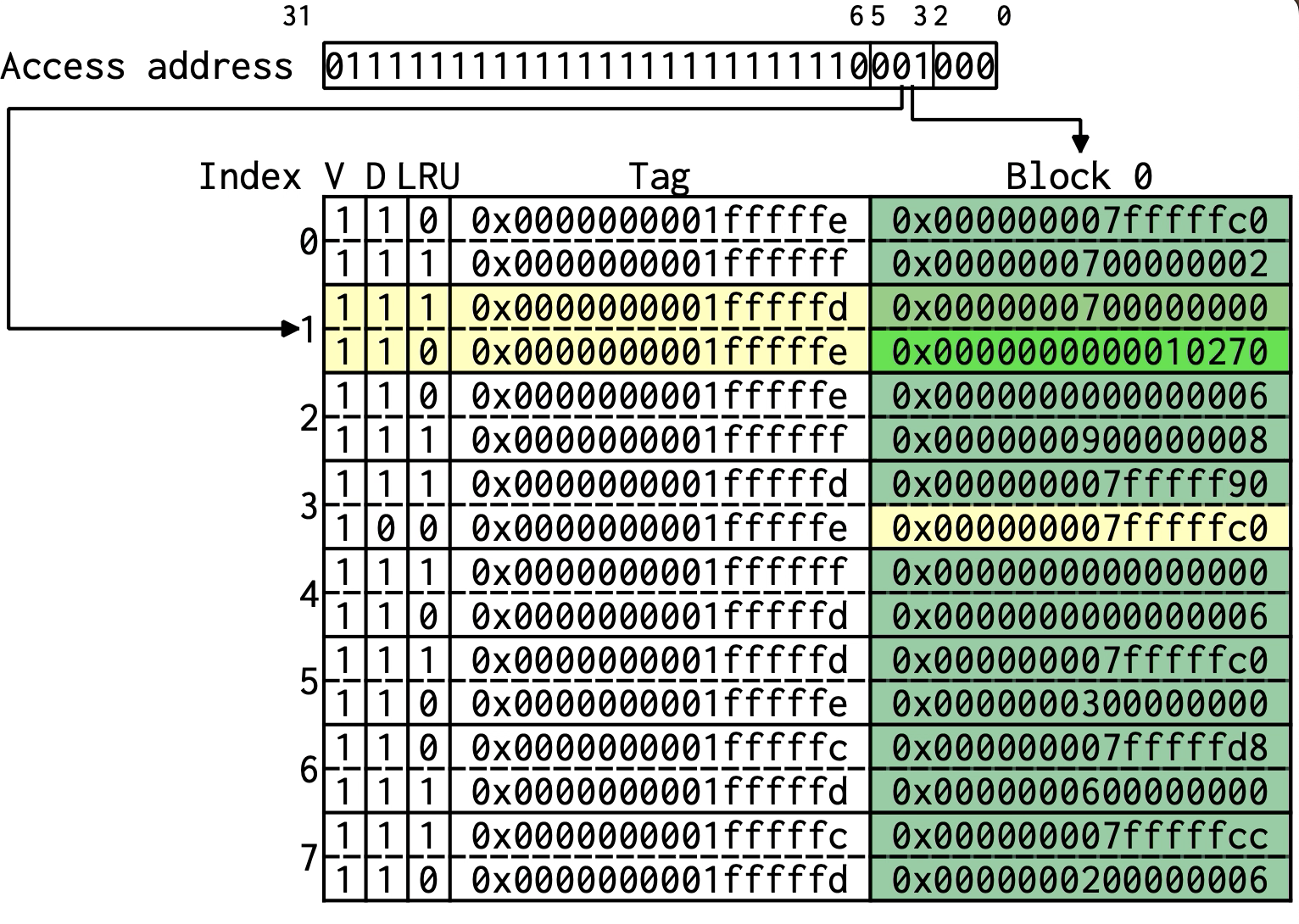
1. (10 points) One case of a hit that makes a block become dirty. (Not dirty before that hit)



**Image1.2.1: the store instruction I find the case**



**Image1.2.2: cache state before the instruction**

****

**Image1.2.3:accessed memory address & cache state after the instruction**

**Explanation:**

**With write allocate, we fetch the block on a write hit.**

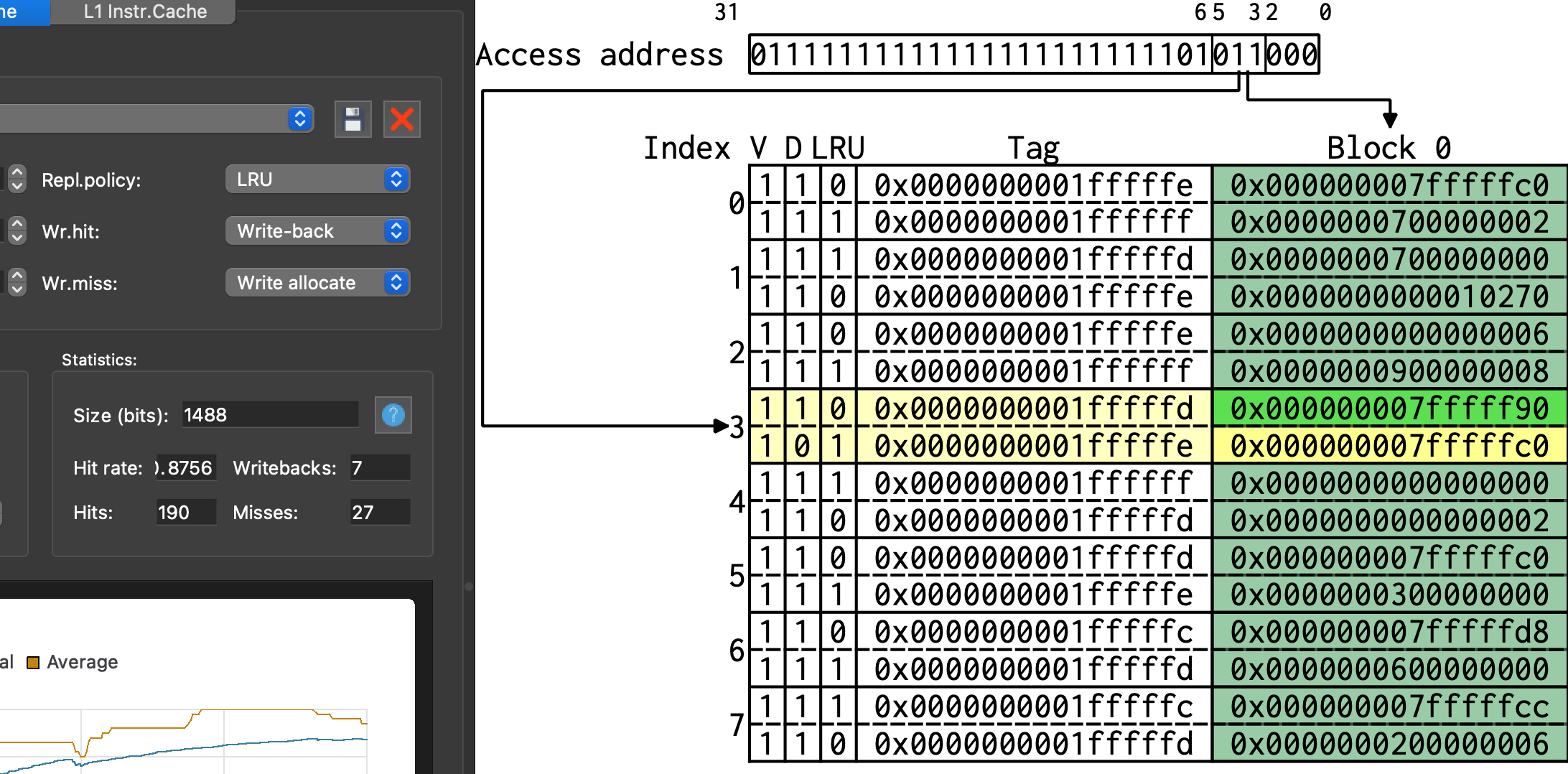
**After the write hit at 2nd set(with index 1), the dirty bit(D) is set to 1, just like what Image1.2.2 and 1.2.3 show.**

**As a result, we can see that the write hit makes a block becomes dirty.**

1. (10 points) One case of a replacement with write-back needed.

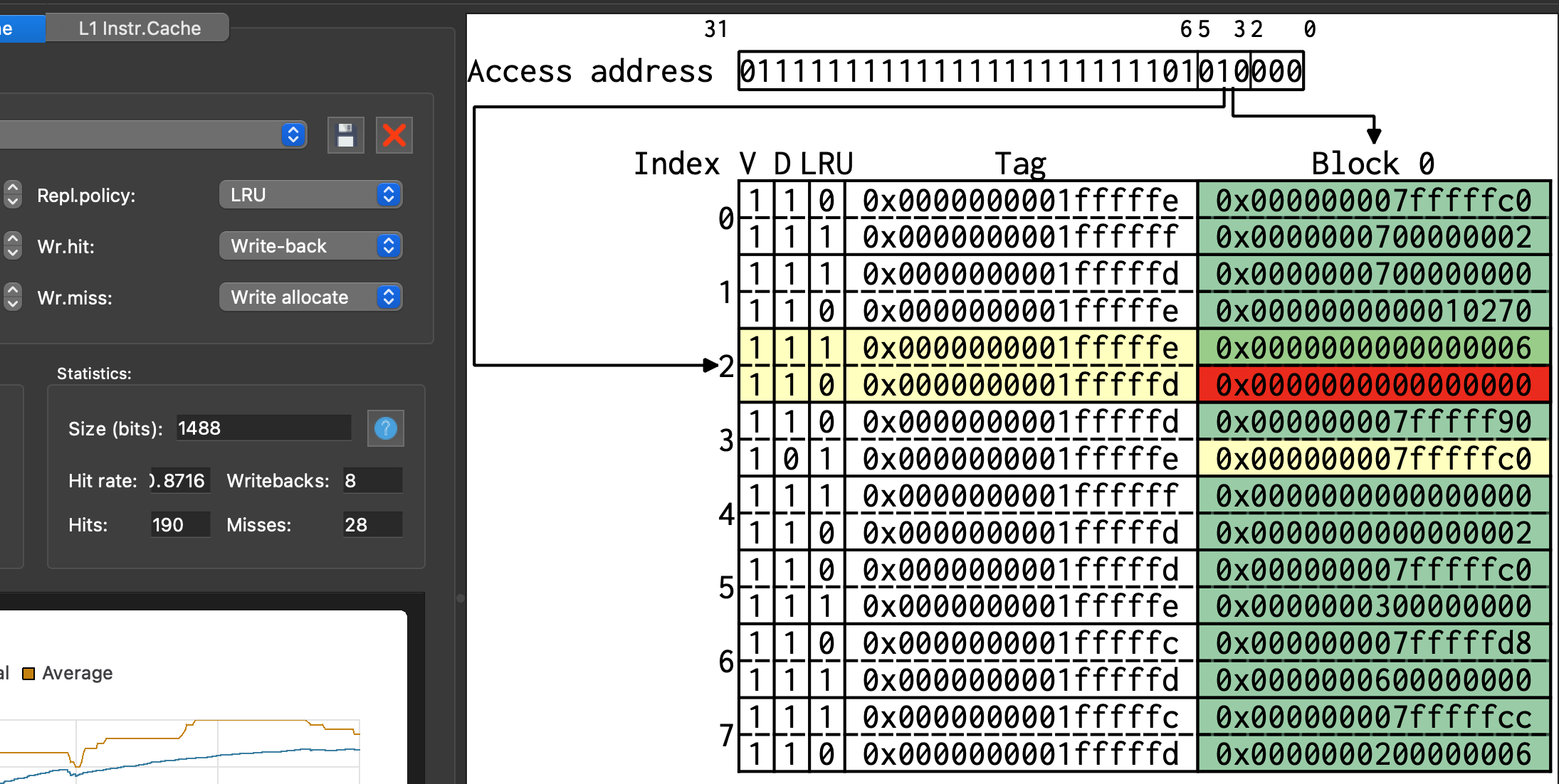


**Image1.3.1: the store instruction I find the case**





**Image1.3.2: cache state before the instruction**





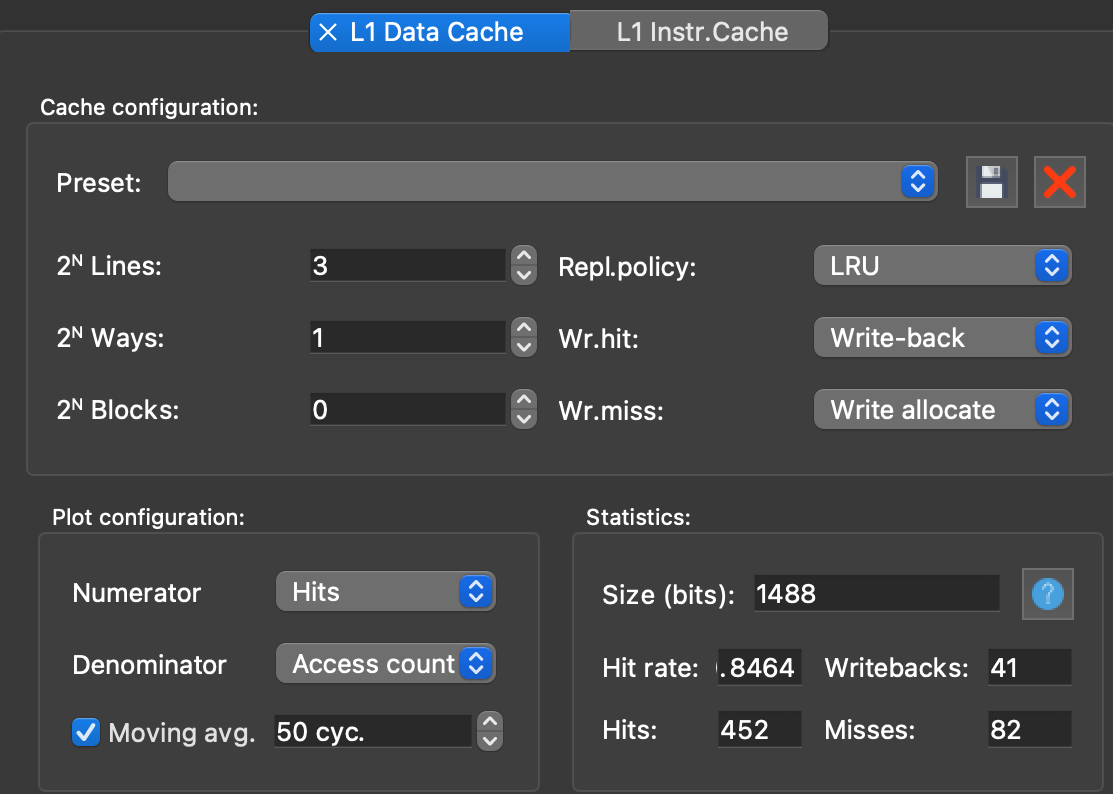
**Image1.3.3:accessed memory address & cache state after the instruction**

**Explanation:**

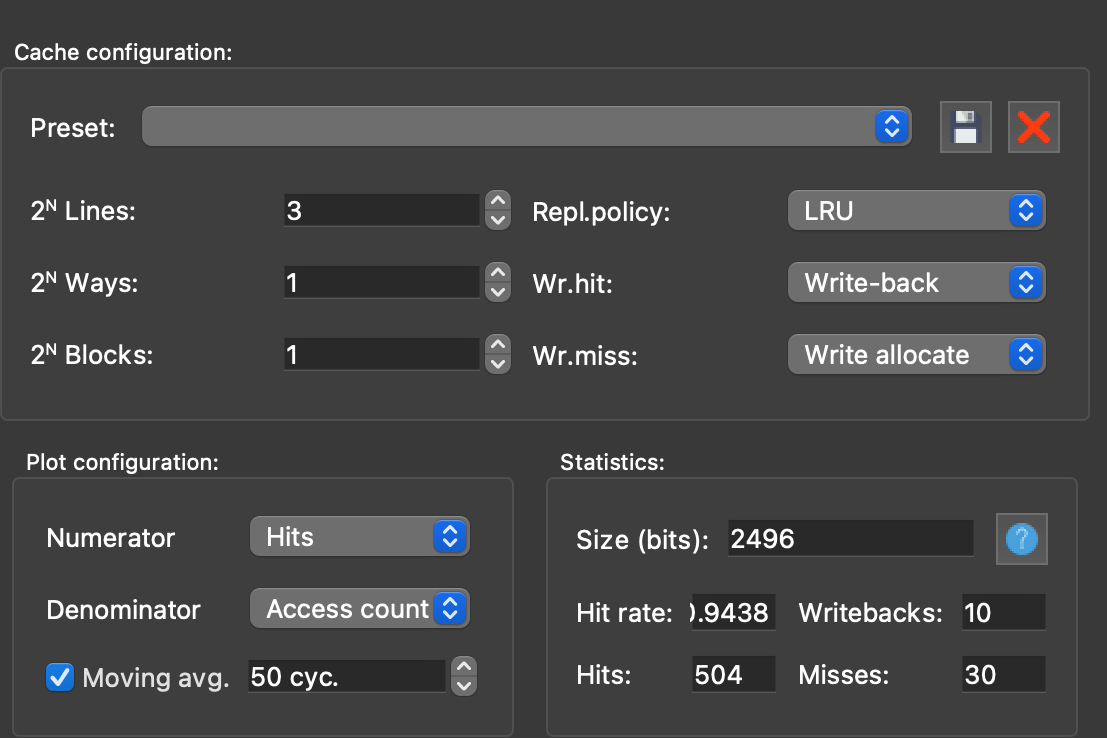
**By Image1.3.2 and 1.3.3, we can see that the 28th miss happened after the store instruction, and the LRU bit becomes 0 (LRU bit = 1, originally).**

**What’s more, since the dirty bit for the original data is 1, we know that we need to write back the old data to the memory and replace the block with new data.**

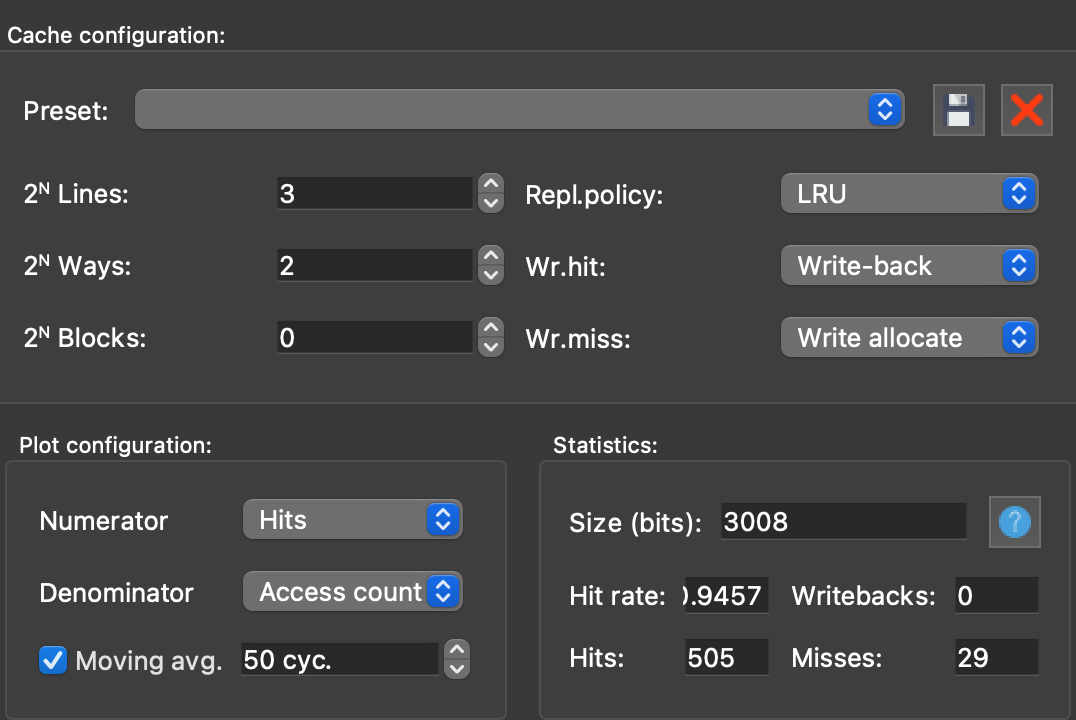
2. (10 points) Improve the hit rate by designing your own cache. You may adjust the associativity or the cache size. Please give some screenshots to show (i) the cache you design and (ii) the improvement of the hit rate. A brief discussion about why the cache you design makes the hit rate higher is also needed. Note that if you can’t get a better hit rate somehow, a reasonable discussion is still needed. Otherwise, you can use the given ELF example that a higher hit rate is definitely possible.



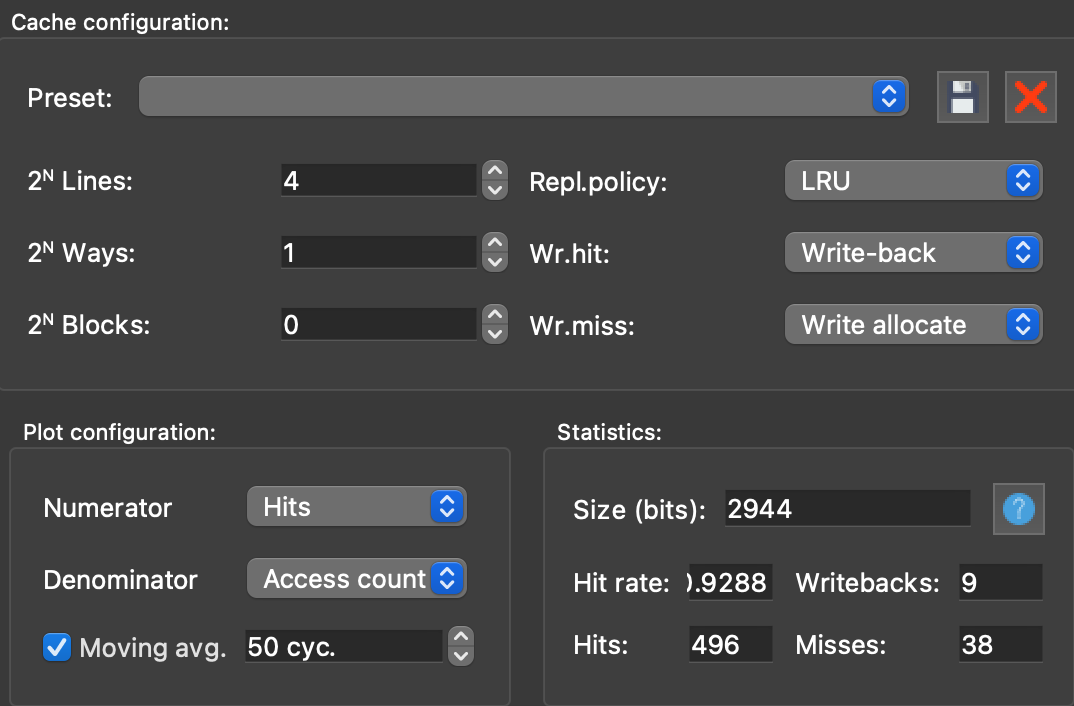
**Image2.1: the original cache 🡪84.6%**

****

**Image2.2: 3-1-1 🡪94.3%**



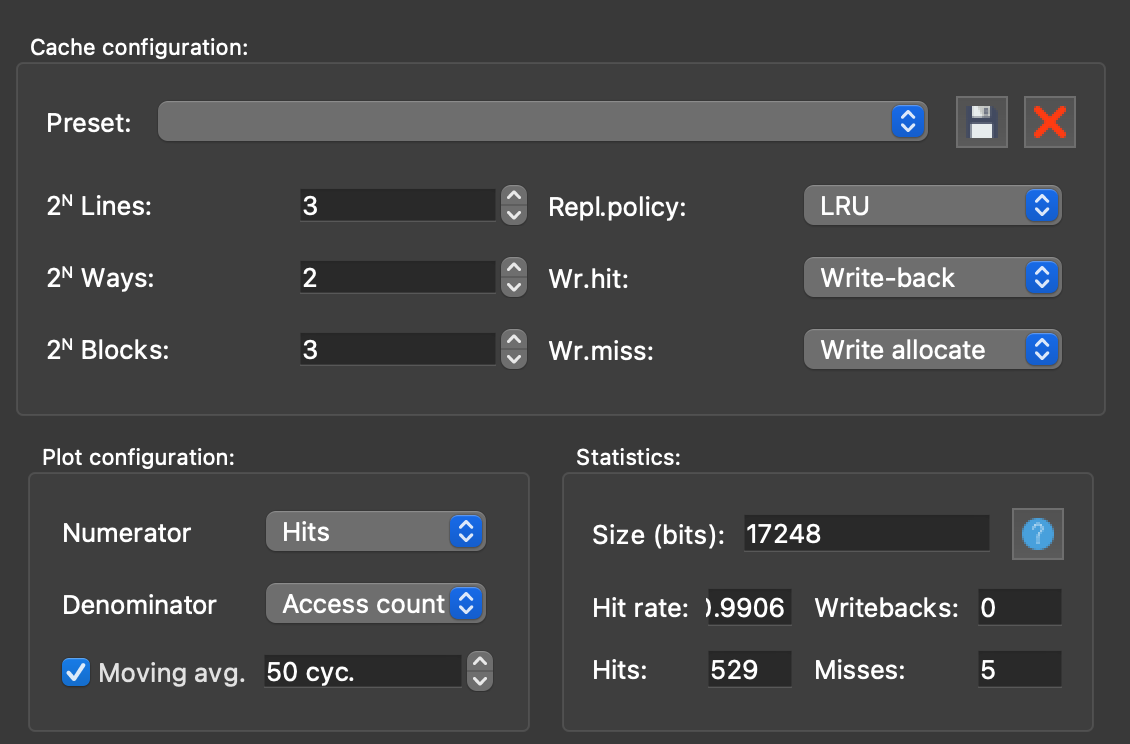
**Image2.3: 3-2-0🡪94.6%**



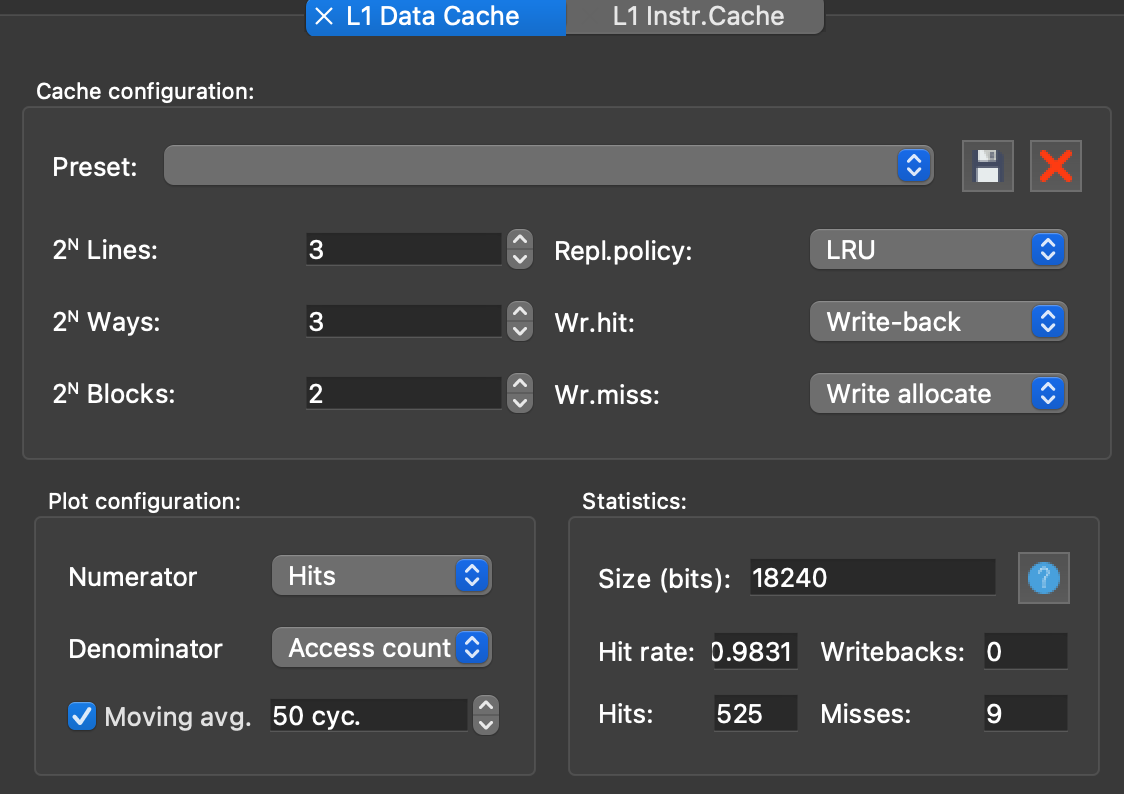
**Image2.4: 4-1-0 🡪92.9%**

**By image 2.1~2.4, we can see that increase the number of ways and blocks are better than increase the one of lines.**

**As a result, I will focus on increase the former two aspect instead of increasing the one of lines to enhance the Hit rate to near 100% later(image 2.5 and image 2.6)**



**Image2.5: 3-2-3 🡪99%**



**Image2.6: 3-3-2 🡪98%**

**By image2.5 and 2.6, we can see that the performance of increasing the number of blocks is better than increase that of ways.**

**Therefore, the cache show by image 2.5 is my final cache.**

**In my opinion, my coding style contribute to this situation since I use nearly all of the registers in my code and I use stack to store the data before entering a loop or a function almost every time; as a result, increasing the amount of the ways or the blocks is better!**

**And since the registers I used for store instruction when storing data into the stack are the same type(e.g. temporary register or save register), which means their addresses have little difference, it’s better to increase the number of blocks than the ways.**

3. (20 points) Suppose a cache receives the following memory addresses for accessing 32-bit single words: (Note: Memory addresses are to the bytes.)

0x74, 0x3c, 0xa4, 0x70, 0x34, 0x08, 0xc4, 0xb8, 0x94, 0xb8

For each reference, identify (i) the index, (ii) the tag, and (iii) whether it is a hit or a miss, for each of the following caches. Assume each cache is initially empty.

1. (10 points) A direct-mapped cache with 1-word blocks and a total size of 8 words.

**The memory address is 8bit**

**8/1 = 8 block 🡪 3 bit index,**

**1word/block=4 byte block🡪 2 bit offset**

**8–3–2=3🡪 3 bit tag**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Word addr** | **Binary addr** | **Tag** | **Index** | **Hit/Miss** |
| **0x74** | **011 101 00** | **011** | **101** | **Miss** |
| **0x3c** | **001 111 00** | **001** | **111** | **Miss** |
| **0xa4** | **101 001 00** | **101** | **001** | **Miss** |
| **0x70** | **011 100 00** | **011** | **100** | **Miss** |
| **0x34** | **001 101 00** | **001** | **101** | **Miss** |
| **0x08** | **000 010 00** | **000** | **010** | **Miss** |
| **0xc4** | **110 001 00** | **110** | **001** | **Miss** |
| **0xb8** | **101 110 00** | **101** | **110** | **Miss** |
| **0x94** | **100 101 00** | **100** | **101** | **Miss** |
| **0xb8** | **101 110 00** | **101** | **110** | **Hit** |

1. (10 points) A 2-way set associative cache with 2-word blocks and a total size of 16 words.

**16/2 = 8 block,8 block / 2 way = 4 sets 🡪 2bit index**

**2word/block = 8 byte block🡪 3 bit offset**

**8–2–3=3🡪 3 bit tag**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Word addr** | **Binary addr** | **Tag** | **Index** | **Hit/Miss** |
| **0x74** | **011 10 100** | **011** | **10** | **Miss** |
| **0x3c** | **001 11 100** | **001** | **11** | **Miss** |
| **0xa4** | **101 00 100** | **101** | **00** | **Miss** |
| **0x70** | **011 10 000** | **011** | **10** | **Hit** |
| **0x34** | **001 10 100** | **001** | **10** | **Miss** |
| **0x08** | **000 01 000** | **000** | **01** | **Miss** |
| **0xc4** | **110 00 100** | **110** | **00** | **Miss** |
| **0xb8** | **101 11 000** | **101** | **11** | **Miss** |
| **0x94** | **100 10 100** | **100** | **10** | **Miss** |
| **0xb8** | **101 11 000** | **101** | **11** | **Hit** |

4. (10 points) Consider the Hamming single error correcting (SEC) code.

(a)  (4 points) Let A = 10011100 be an 8-bit data. Show how to get a 12-bit Hamming SEC code word B for A.

**8bit 🡪 2^n >= n + 8 + 1, n = 4 🡪 4 parity bits**

**\_ \_ 1 \_ 0 0 1 \_ 1 1 0 0**

**\_ \_ 1 \_ 0 0 1 \_ 1 1 0 0 🡪 p1 = 1**

**\_ \_ 1 \_ 0 0 1 \_ 1 1 0 0 🡪 p2 = 1**

**\_ \_ 1 \_ 0 0 1 \_ 1 1 0 0 🡪 p4 = 1**

**\_ \_ 1 \_ 0 0 1 \_ 1 1 0 0 🡪 p8 = 0**

**B = 11 1 1 001 0 1100**

1. (4 points) Suppose C is obtained by inverting the fifth bit from the left of B and hence has a single bit error. Show how to find and correct the single bit error in C.

**B = 1111 0010 1100 🡪 C = 1111 1010 1100**

**1111 1010 1100 🡪 H = \_ \_ \_ 1**

**1111 1010 1100 🡪 H = \_ \_ 0 1**

**1111 1010 1100 🡪 H = \_ 1 0 1**

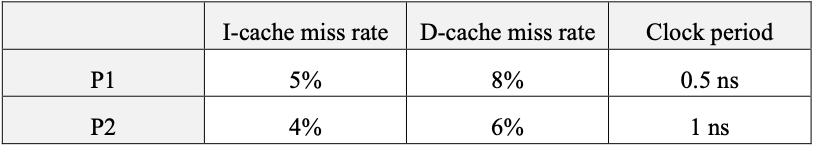
**1111 1010 1100 🡪 H = 0 1 0 1**

**0101 🡪 5(decimal) 🡪 the fifth bit**

1. (2 points) What is the minimum number of parity bits required to protect a 256-bit data using the Hamming SEC code?

**2^n >= n + 256 + 1 = n + 2^8 + 1 🡪 n = 9 🡪 9 parity bits**

5. (10 points) The following table gives some information for each of the instruction and data caches attached to each of the two processors, P1 and P2.



Assume each processor has a base CPI of 1 without any memory stalls. Each access to the instruction or data memory takes 50 ns. And 40% of all instructions have data accesses. Note that a base CPI of 1 means that an instruction completes in one clock cycle, unless either the instruction access or the data access causes a cache miss.

1. (8 points) What is the total CPI for P1 and P2?

**AMAT = Hit Time + Miss Rate \* Miss Penalty**

**P1 :**

**Hit Time = 1 (clock/instruction) \* 0.5 (ns/clock) = 0.5ns/instruction**

**AMAT = 0.5ns + 50ns\*0.05 + 50ns\*0.08\*0.4**

**= 0.5 + 2.5 + 1.6 = 4.6ns**

Note: I-cache miss may happen in every instructions!!Remember not to multiply 0.4

**CPI = (4.6ns /instruction) / (0.5ns/clock) = 9.2clock/instruction**

**P2:**

**Hit Time = 1 (clock/instruction) \* 1 (ns/clock) = 1ns/instruction**

**AMAT = 1ns + 50ns\*0.04 + 50ns\*0.06\*0.4**

**= 1 + 2 + 1.2 = ns**

**CPI = (4.2ns /instruction) / (1ns/clock) = 4.2clock/instruction**

(b)  (2 points) Which processor is faster if they have the same amount of instructions?

**Time cost per instruction:**

**P1: 9.2\*0.5 ns = 4.6ns**

**P2:4.2\*1ns = 4.2ns**

**4.2 < 4.6, 4.6-4.2 = 0.4**

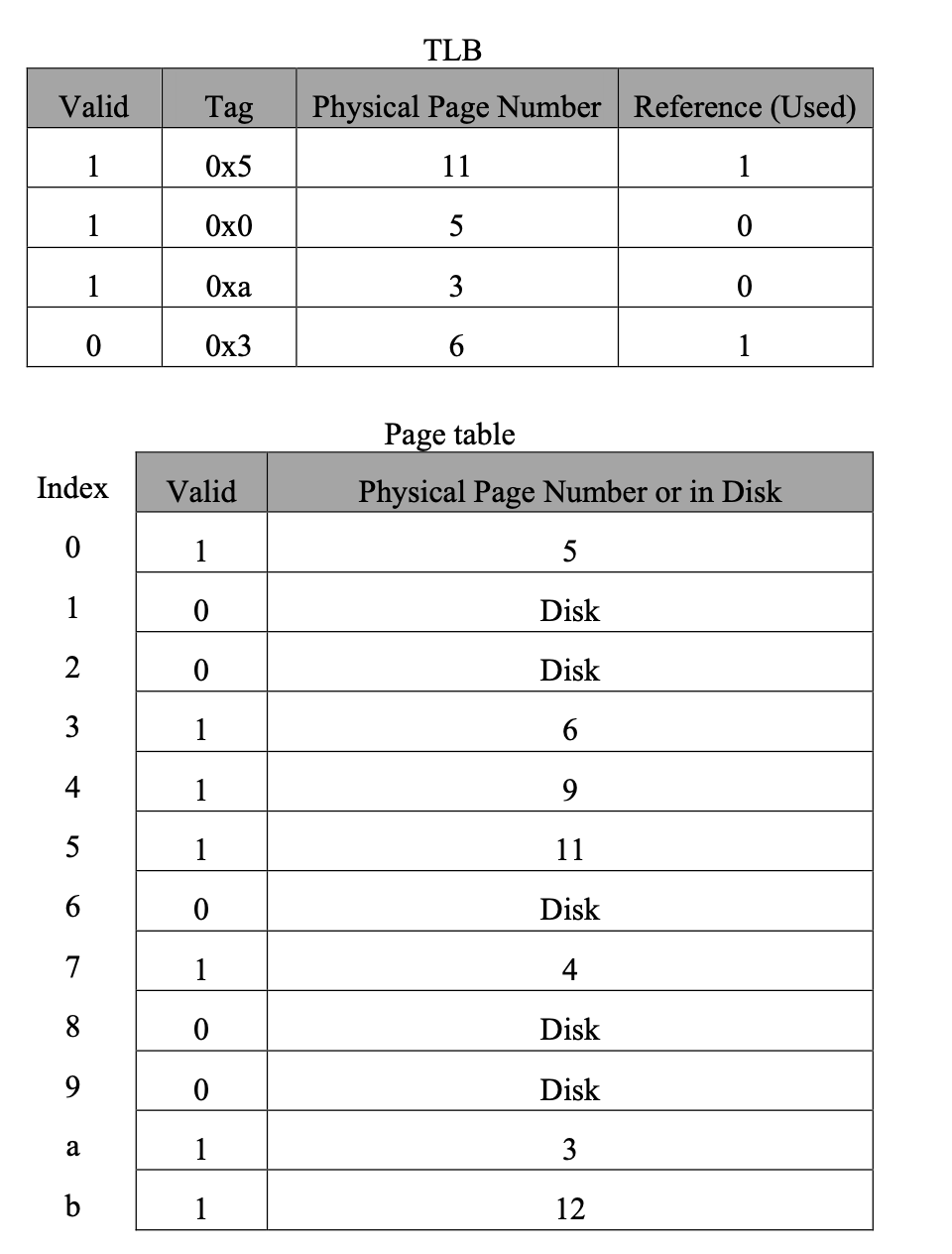
**P2 is faster than P1 by 0.4ns**

6. (20 points) The following data constitute a stream of virtual byte addresses generated by a processor.

0x4a90, 0x210c, 0xa3b5, 0x0e18, 0x904f

With 4 KB pages, a **four-entry fully associative TLB**, and **approximate LRU replacement**, 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.

The initial TLB and page table states are as below.



For each access shown above, list whether (1) the access is a hit or miss in the TLB, (2) whether the access causes a page fault, and (3) the updated state of the TLB.

|  |  |
| --- | --- |
| **Page number(4bit)** | **Offset(12bit)** |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **address** | **Hit/Miss**  **Page Fault** | **TLB** | | | | |
| **Valid** | **Tag** | **PPN** | **reference** | **Replacement**  **pointer** |
| **0x4a90**  **(0100 1010**  **1001 0000)** | **Miss in**  **TLB**  **Hit in**  **PT** | **1** | **0x5** | **11** | **1** | **v** |
| **1** | **0x0** | **5** | **0** |  |
| **1** | **0xa** | **3** | **0** |  |
| **1** | **0x4** | **9** | **1** |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **address** | **Hit/Miss**  **Page Fault** | **TLB** | | | | |
| **Valid** | **Tag** | **PPN** | **reference** | **Replacement**  **pointer** |
| **0x210c**  **(0010 0001**  **0000 1100)** | **Miss in**  **TLB and**  **PT**  **Page fault** | **1** | **0x5** | **11** | **0** |  |
| **1** | **0x2** | **1**  **(random**  **Choose)** | **1** |  |
| **1** | **0xa** | **3** | **0** | **v** |
| **1** | **0x4** | **9** | **1** |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **address** | **Hit/Miss**  **Page Fault** | **TLB** | | | | |
| **Valid** | **Tag** | **PPN** | **reference** | **Replacement**  **pointer** |
| 0xa3b5  **(1010 0011**  **1011 0101)** | **Hit in TLB** | **1** | **0x5** | **11** | **0** |  |
| **1** | **0x2** | **1** | **1** |  |
| **1** | **0xa** | **3** | **1** | **v** |
| **1** | **0x4** | **9** | **1** |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **address** | **Hit/Miss**  **Page Fault** | **TLB** | | | | |
| **Valid** | **Tag** | **PPN** | **reference** | **Replacement**  **pointer** |
| 0x0e18  **(0000 1110**  **0001 1000)** | **Miss in**  **TLB**  **Hit in PT** | **1** | **0x0** | **5** | **1** |  |
| **1** | **0x2** | **1** | **1** | **v** |
| **1** | **0xa** | **3** | **0** |  |
| **1** | **0x4** | **9** | **0** |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **address** | **Hit/Miss**  **Page Fault** | **TLB** | | | | |
| **Valid** | **Tag** | **PPN** | **reference** | **Replacement**  **pointer** |
| 0x904f  **(1001 0000**  **0100 1111)** | **Miss in**  **TLB and PT**  **Page fault** | **1** | **0x0** | **5** | **1** |  |
| **1** | **0x2** | **1** | **0** |  |
| **1** | **0x9** | **2**  **(random**  **Choose)** | **1** |  |
| **1** | **0x4** | **9** | **0** | **v** |