**2019112041 컴퓨터구조 정재욱**

**Report 3**

**(1) 시뮬레이터 출처**

출처 https://github.com/jaminthorns/cpu-cache-simulator

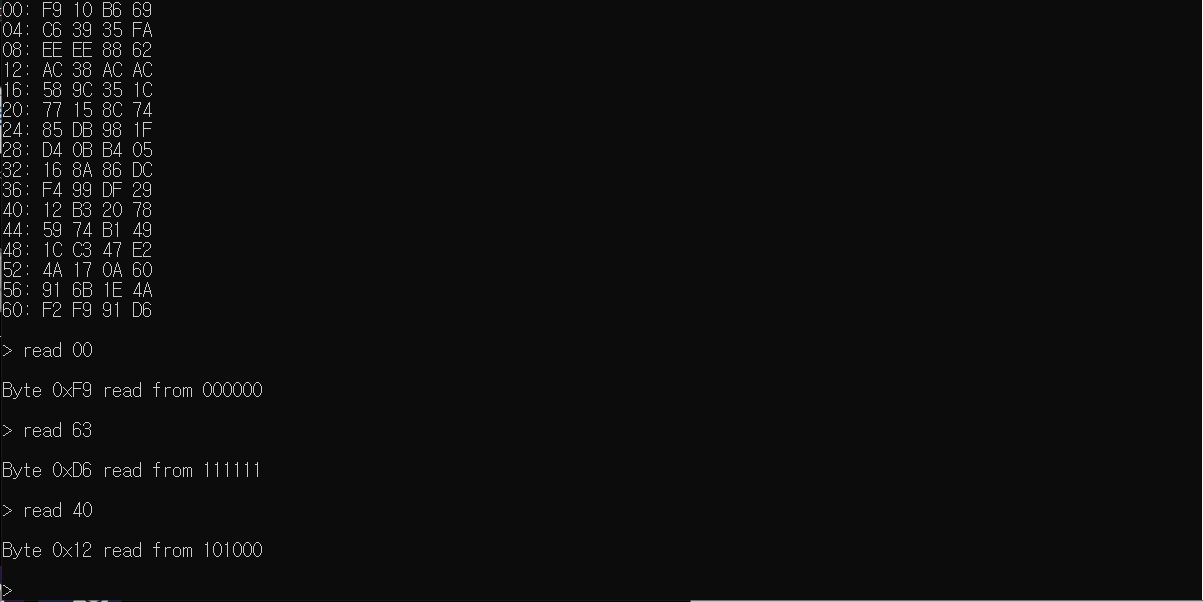
**(2) 시뮬레이터 분석**

시뮬레이터를 실행하는데 사용하는 명령어들

* **read ADDRESS** - read byte from memory
* **write ADDRESS BYTE** - write random byte to memory
* **randread AMOUNT** - read byte from random location in memory AMOUNT times
* **randwrite AMOUN**T - write random byte to random location in memory AMOUNTtimes
* **printcache START LENGTH** - print LENGTH lines of cache from START
* **printmem START LENGTH** - print LENGTH blocks of memory from START
* **stats** - print out hits, misses, and hit/miss ratio
* **quit** - quit the simulator



Printmem을 통해서 메모리에 있는 데이터를 출력했다. 세번째 줄을 보면 6 4 2 1 LRU WT가 있는 것을 볼 수 있는데, 이는 앞에서부터 차례로 메모리 크기, 캐시 크기, 블록 크기, associativity, replacement policy, write policy를 의미한다.



Read 명령어를 통해서 메모리에 저장된 데이터를 읽었다. 위에서 printmem을 통해서 확인한 메모리에 저장된 데이터와 같은 값이 나오는 것을 볼 수 있다.

read 00 명령어가 실행될 때, 캐시가 비어 있으므로 miss가 발생한다. 따라서 메모리로부터 데이터를 읽어온다. 또한 read 63, read 40이 실행될 때 역시 miss가 발생하므로 메모리로부터 데이터를 읽어온다.



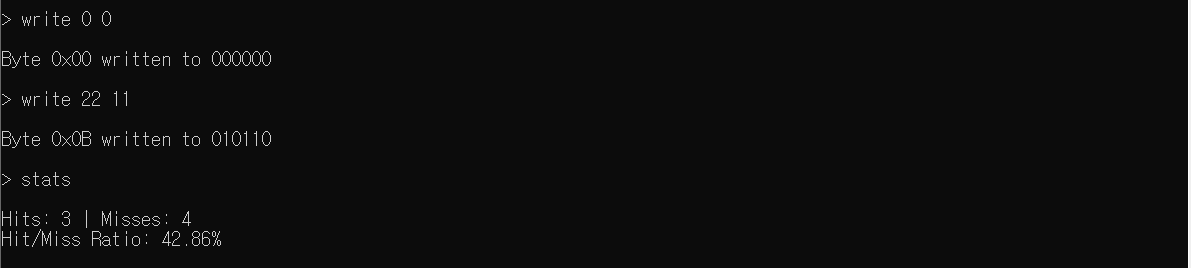
위에서 설명한 내용을 확인하기 위해서 printcache 명령어를 통해 캐시에 어떤 데이터가 있는지 확인해 보았다. 이전에 miss가 발생해 메모리로부터 데이터를 읽어오면서 캐시에 저장한 것을 볼 수 있다. 캐시를 살펴보면 read 명령어를 통해서 읽어온 데이터들이 포함된 블록이 캐시에 저장되어 있는 것을 볼 수 있는데, 00, 40, 60을 시작주소로 해서 한 블록의 데이터가 한 라인에 저장되어 있다.

Stats 명령어를 통해서 현재까지 실행된 시뮬레이터의 hit, miss hit ratio를 출력했는데 위에서 말한 것처럼 miss가 3번이 발생해 아직까지의 hit ratio는 0%인 것을 볼 수 있다.



이어서, read 명령어들을 통해 다시 한번 데이터들을 읽어보았다. 캐시에 시작주소가 0000인 블록이 저장되어 있고, 블록의 크기를 4바이트로 지정했으므로 메모리 주소가 3인 데이터는 캐시에 저장되어 있다. 따라서 cache hit가 발생된다. 메모리 주소가 42인 데이터도 마찬가지이다.

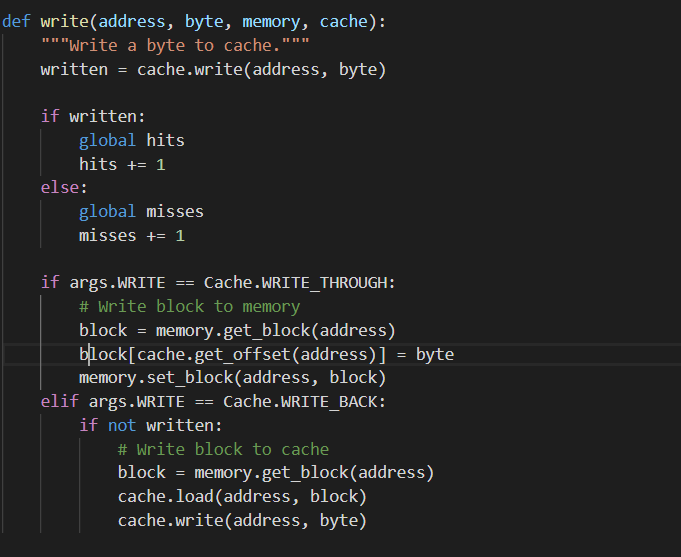
Read 명령어들을 실행한 이후, stats를 통해서 결과를 출력해보았다. Hit가 2번 발생했으므로 hit ratio가 40%로 바뀐 것을 볼 수 있다.



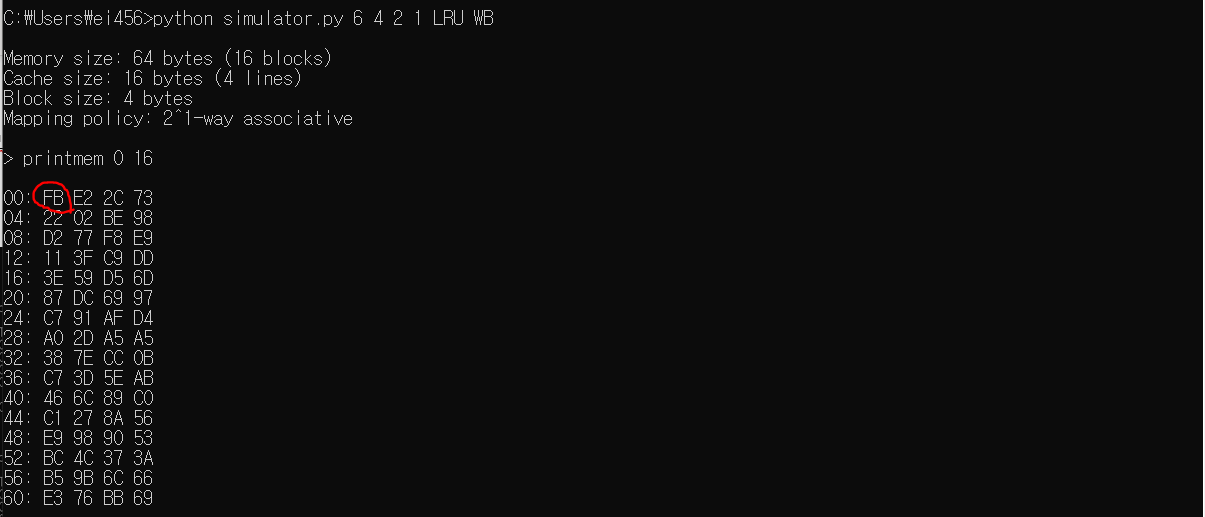
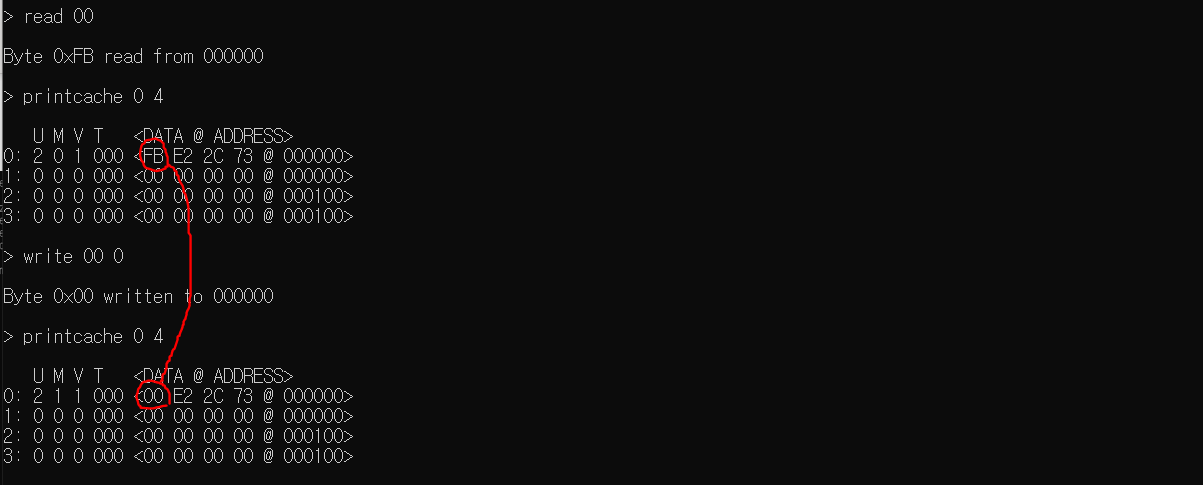
write 명령어들을 통해서 캐시에 없는 내용과 캐시에 있는 내용을 write 했다. Address가 0인 데이터는 캐시에 존재하므로 hit, 22인 데이터는 존재하지 않으므로 miss가 발생했다. 따라서 결과를 출력해보면 이전에 비해서 hit가 1 증가, miss가 1 증가되었다.

앞에서 write를 했으므로 메모리와 캐시의 데이터가 변경된 (write 22 11에서 11이 16진수로 0B) 것을 볼 수 있다. Write policy로는 write through를 선택했다. 따라서 write시 hit가 발생하더라도 캐시뿐만 아니라 메모리에도 수정을 해준 것을 볼 수 있다.



Cache simulator의 write 부분 코드이다. cache.write(address, byte) 함수는 cache에 메모리 주소가 address(매개변수)인 해당하는 데이터가 존재한다면 그 값을 byte(매개변수)로 바꾸고, 없다면 None을 return하는 함수이다. 중간 이후의 코드를 보면 WRITE\_BACK으로 policy를 설정하면 miss시에 메모리에 write를 하지 않고, cache에 해당하는 메모리의 블록을 저장한 뒤, write를 실행하는 반면, WRITE\_THROUGH시에는 miss나 hit와는 상관없이 메모리에만 write를 한다. 즉, miss가 발생하더라도 메모리에만 write를 하고, 캐시에는 저장하지 않도록 코드가 짜져 있었다. **그래서 위에서 설명한 예시에서 write 22 11를 통해 메모리에 write를 실행했음에도 불구하고 cache에 해당하는 데이터가 존재하지 않았다.**

Write policy를 write back으로 설정한 다음 실행해보았다. cache에 있는 데이터를 write하면, cache에는 write를 실행하지만 memory에는 값이 변경되지 않은 것을 볼 수 있다.



또한, cache에 없는 데이터를 write하면, write through와는 다르게 cache에 write한 데이터를 포함한 블록이 저장되는 것을 볼 수 있다.

**(3)**

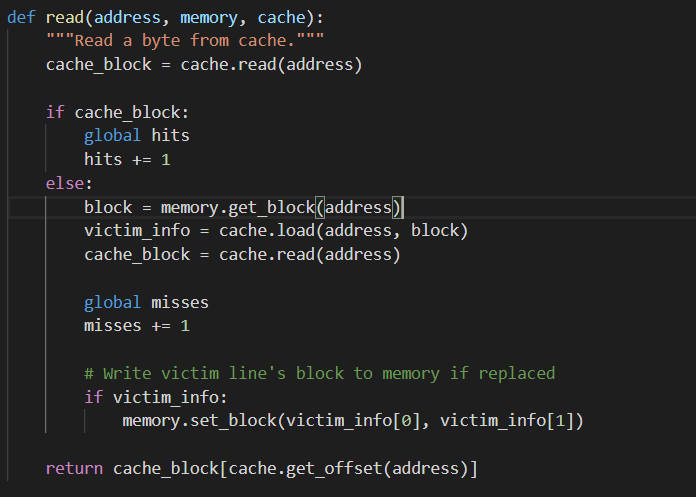
원래 시뮬레이터는 1-level cache였는데, 이를 L1 cache, L2 cache로 바꿈으로써 2-level cache로 upgrade 시켜 보았다.

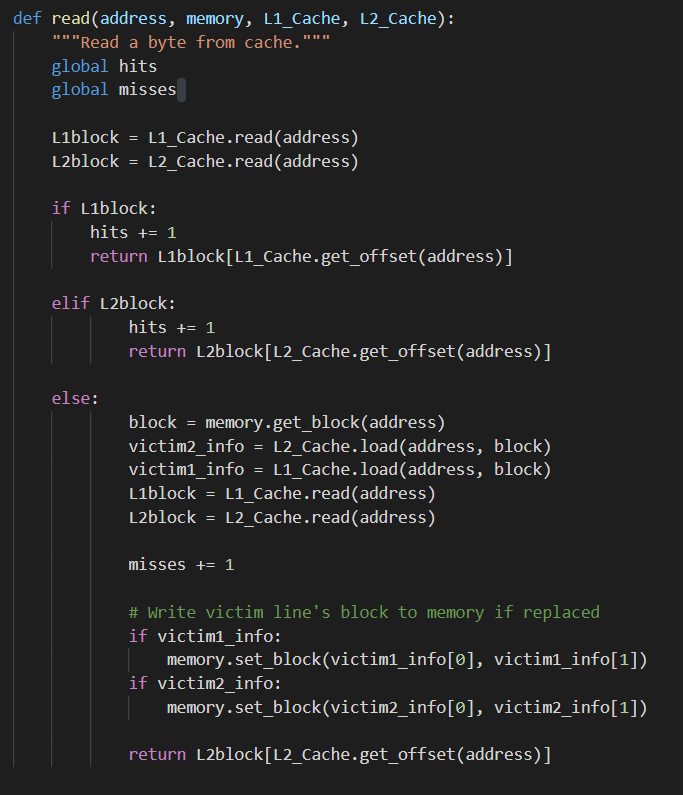
시뮬레이터를 실행하는데 사용하는 명령어들 (추가된 명령어들만)

(2)에서 사용한 명령어들과 같은 명령어를 사용하고 **printcache START LENGTH** 대신에

**printl1 START LENGTH** - print LENGTH lines of L1 cache from START

**printl2 START LENGTH** - print LENGTH lines of L2 cache from START





위에 있는 코드는 1-level cache의 코드이고, 아래에 있는 코드는 2-level cache의 코드이다. **cache.read(address) :** 캐시에 있는 데이터를 읽어오는 함수, hit시 메모리 블록, miss시 None return

**cache.load(address, block) :** 캐시로 메모리에 있는 블록을 로드하는 함수, victim은 로드로 인해서 캐시에서 삭제되는 데이터를 의미함, victim 주소, 데이터를 튜플로 return, no victim시 None return

**cache.get\_offset(address) :** offset을 구하는 함수

**memory.get\_block(address) :** address를 포함하는 메모리에 있는 블록을 리턴하는 함수

**L1\_Cache.load, L2\_Cache.load, L1\_Cache.read, L2\_Cache.read**는 위에서 설명한 함수와 똑같이 동작한다.

1-level cache과는 다르게 2-level cache 같은 경우 L1 cache에서 miss가 발생하더라도

**elif L2block:**

**hits += 1**

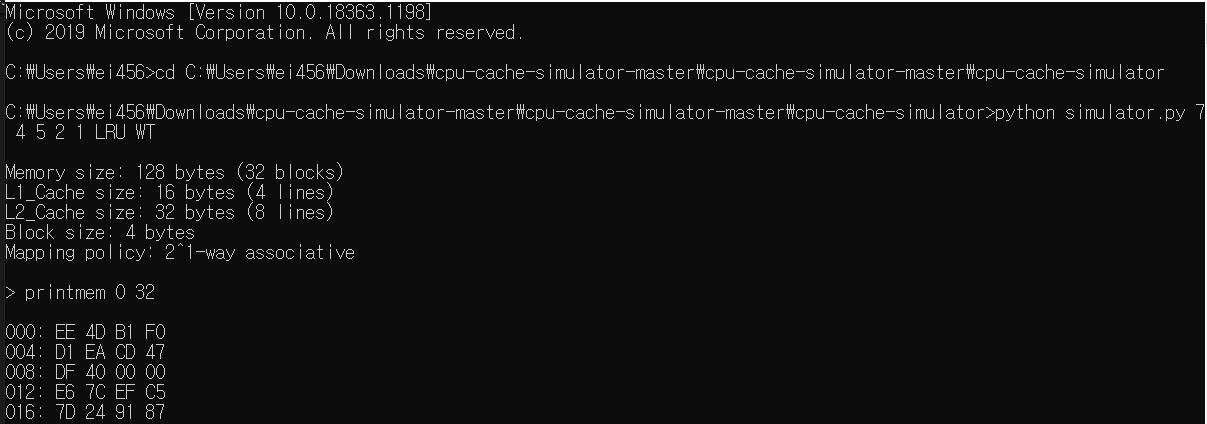
**return L2block[L2\_Cache.get\_offset(address)] # address에 존재하는 data를 return**

문을 통해서 L2 cache에서 hit가 발생한다면, 결국 hit로 처리될 수 있도록 upgrade 했다.

Else 문에서는 L1 cache, L2 cache에서 모두 miss가 발생한다면 메모리로부터 해당 address가 위치한 블록을 각각의 캐시에 로드하도록 만들었다.

위와 같이 upgrade 함으로써 hit ratio가 증가하는 방향으로 성능이 향상된다.

이를 확인하기 위해서 예시를 살펴보자.

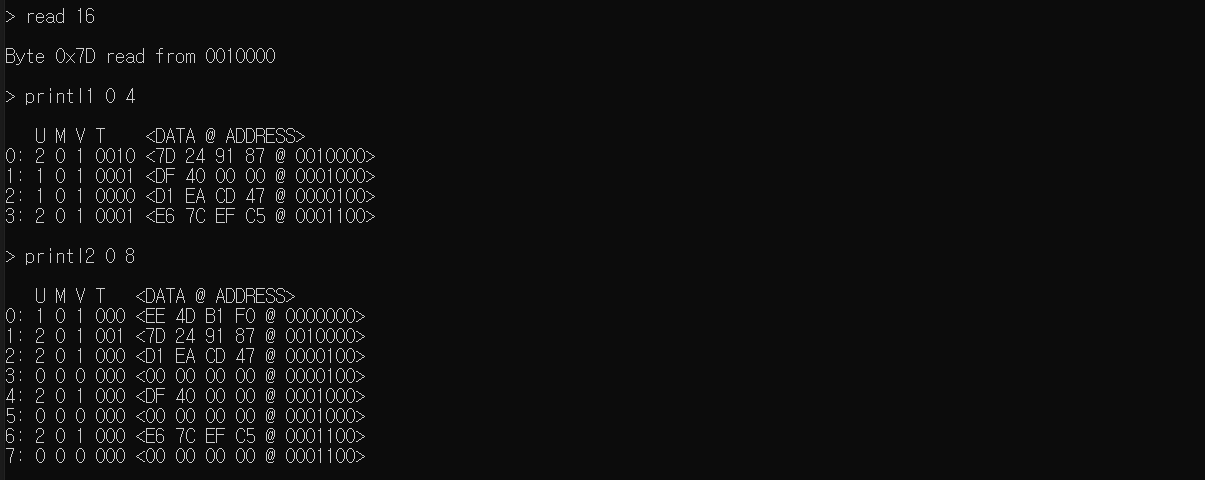


위와 같은 메모리, 캐시를 구성했다고 가정하자. 4~5번째 줄을 보면 **7 4 5 2 1 LRU WT**가 있는데 왼쪽부터 mem size, L1 size, L2 size, block size, mapping, replace policy, write policy이다. 아직까지 캐시는 비어있다.



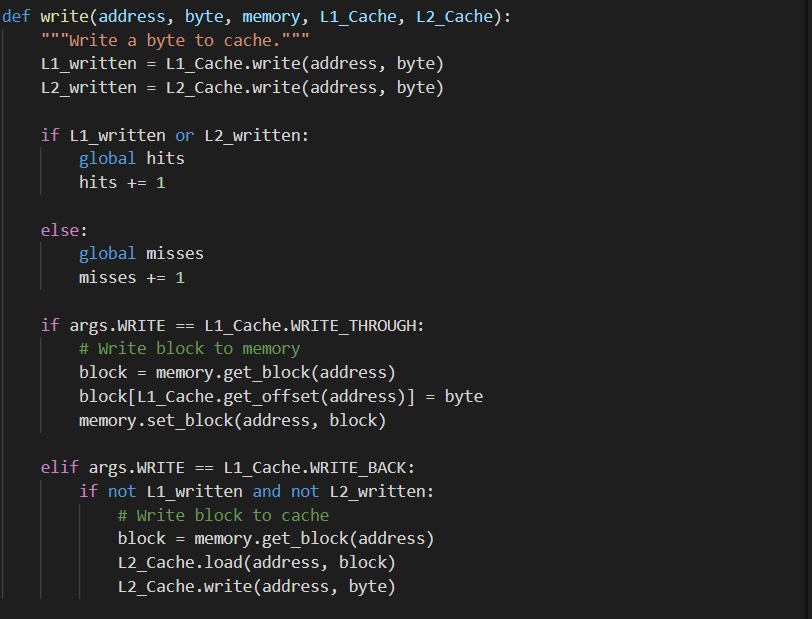


read 0 ~ read 12를 통해서 메모리의 첫번째 ~ 네번째 블록이 L1, L2 cache에 들어간 것을 볼 수 있다. 또한, 4번의 miss가 발생해서 hit ratio가 0%인 것을 볼 수 있다.



L1 cache가 꽉 찬 상태에서 read 16 명령어를 실행했다. 따라서 L1 cache에 있던 첫번째 블록**(EE 4D B1 F0)**이 read 16으로 인해 5번째 블록**(7D 24 91 87)**으로 대체(replace)되었다.

이후, read 0을 실행했다. L1 cache에는 주소가 0인 데이터가 존재하는 메모리상의 첫번째 블록**(EE 4D B1 F0)**이 존재하지 않지만, L2 cache에는 존재한다. 따라서 Hits가 1 증가한 것으로 보아, cache miss가 아니라 cache hit가 발생한 것을 stats 명령어를 통해서 볼 수 있다.



이는 2-level cache에서의 바뀐 write code이다. L1 cache나 L2 cache 둘 중 하나에서 hit가 발생한다면 hit로 처리되는 것을 볼 수 있다. WRITE\_BACK으로 write policy를 설정했을 경우, L1 cache, L2 cache에서 모두 miss가 발생한다면, L2 cache로 해당 메모리 블록을 로드한 뒤, write 작업을 실행한다.



전 페이지에 있는 예시에 이어서 write 0 44를 실행했는데, 이 명령어는 0번지에 44(16진수로 2C)를 write한다. 위에서 설명했듯이 L1 cache에는 첫번째 블록**(EE 4D B1 F0)** 이 없다. 그러나, L2 cache에는 해당하는 블록이 있으므로 stats을 통해서 결과를 출력해보면, hit가 1 증가한 것을 볼 수 있다. 또한, L2 cache와 메모리를 (맨 처음에 write through로 설정) 보면 write 0 44로 인해서 0번지에 있는 data가 EE에서 2C로 바뀌었다.