**CACHE**

PROBLEM STATEMENT

 A cache of size S with CL as the number of cache lines and block size B is to be built. S, CL, and B are in powers of 2. Write a program that allows loading into cache and searching cache using:

1. Direct mapping
2. Associative memory
3. n-way set associative memory where n is a power of 2.

SOLUTION

In this project I have designed an implementation of a cache which uses three different address mapping techniques to reach to a desirable address in the cache quickly. The project has three separate java codes each implementing a particular mapping technique to perform cache operations. In all the three program takes input of the length of physical address, block size, cache lines and set length (this is only an input for set-associative mapping). Then the program takes two binary strings as input in case of writing and takes one binary string in case of reading. Then in case of writing it writes the data in cache and displays the cache in the console. In case of reading it prints the data stored in the given address.

***Programming Language***: Java

ASSUMPTIONS

The following are the assumptions that we have taken while building Cache:

1. The physical address length has to be small than or equal to 32.
2. The data that can be written in cache could only be binary.
3. Every word in the cache is of 8 bits i.e one byte
4. All address related inputs are given in bytes.
5. The address input for reading or writing should be of the same length as the length entered for physical address
6. All the inputs of the program should be separated by colon”:”
7. The program will end when you write “STOP” in capitals.
8. Number of cache lines must be less than number of blocks

GETTING STARTED

Please follow these instructions to use the Cache:

1. Think about all the specification you want in your cache keeping in ming the above assumption.
2. Load the java program on to your compiler and hit the run button.
3. When the program will rum you will be prompted to enter the specifications (colon separated) in your java program.
4. Then hit enter, you will reach to the next line type two binary string colon separated to give a write command. In which first binary is the address to which you wish to write. The second string is the data you wish to write at that address.
5. But enter only one binary string ending with colon to read from the cache from the entered address.
6. After your command press enter and you will receive you output.
7. In case of read output will display “HIT” or “MISS”, also the data stored at that address. But if there is no data present ot the address it would display “null”.
8. In case of write the output will display “HIT” or “MISS”,and the contents of the clock you are trying to view in the cache.
9. Whenever there is a miss in read or write the program insert the reqired block and also displys a message about which block is getting inserted.
10. You can give as many read and write commands you want.But if you want the program to end you have to type “STOP”.

ERRORS HANDLED

The following are the errors that the assembler handles after reading the given Cache Code:

1. "PLEASE ENTER PHYSICAL ADDRESS LENGTH:BLOCK SIZE:CACHE LINE"
2. "ERROR:BLOCK SIZE CAN NOT BE GREATOR THAN MEMORY SIZE"
3. "ERROR:CACHE SIZE CAN NOT BE GREATOR THAN MEMORY SIZE"
4. "Data entered has exceeded the word length, hence can't write the data"

SUPPORT

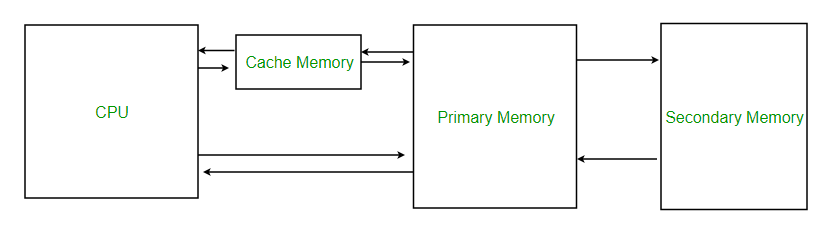
If you have any issues regarding this project, feel free to contact me.

Vani Sikka [vani19395@iiitd.ac.in](mailto:vani19395@iiitd.ac.in) (2019395)

**UNDERSTANDING THE CACHE**

**Cache Memory** is a special very high-speed memory. It is used to speed up and synchronizing with high-speed CPU. Cache memory is costlier than main memory or disk memory but economical than CPU registers. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.

Cache memory is used to reduce the average time to access data from the Main memory. The cache is a smaller and faster memory which stores copies of the data from frequently used main memory locations. There are various different independent caches in a CPU, which store instructions and data.



**UNDERSTANDING THE PROJECT**

The project has three parts hence each part in coded in a separate code file to avoid confusion. The following is the working all the code files explained in detail:

## **1.DIRECT MAPPING**

The code basically takes the basic specification of cache a input. From that input it makes a array of my defined class cache lines; which has two property tag and block which is also an array. This block array is of size entered by the user where every index of the block array represents the word number. When the user enters a command be it either read or write. The program slice the address into three parts the first LSB bits from 0 to n-w-p (where n is the exponent of the memory is the exponent of the block size and p is the exponent of the cache size) represent the tag. Then the Next LSB bits from n-w-p to n-w tell which cache line in the cache to check. Then the last MSB bits from n-w to the end of the address tells which word is to be accessed in the block. On slicing the address program go to the index which is the number specified the bits between n-w-p to n-w. At that index of the cache line program check that if the tag stored in that cache line is same as the tag in my address. If it is same program declare a HIT else a MISS. In case case of a HIT when the program goes to the block array of the cache line and displays the data at the index which is the number formed by n-w to end of address bits.

In case of a MISS program call the insert block function which first generates all the address that the given addresses block contains. Then check for all addresses if any of them have been updated (all updated address is stored in a HashMap to keep track of the address that get updated before their replacement) if yes the that word in the block gets the updated value if no then it retains its old value. Once I create this block, I assign it to the cache line represented n-w-p to n-w bits.

2.**Associative Mapping**

The code basically takes the basic specification of cache a input. From that input it makes an array of my defined class cache lines; which has two property tag and block which is also an array. This block array is of size entered by the user where every index of the block array represents the word number. When the user enters a command be it either read or write. The program slices the address into two parts the first LSB bits from 0 to n-w (where n is the exponent of the memory is the exponent of the block size) represent the tag. Then the last MSB bits from n-w to the end of the address tells which word is to be accessed in the block. Then I check each and every cache line if it has the address tag or not. If one of them have the tag same as address tag program declare a HIT else a MISS. In case of a HIT when the program goes to the block array of the cache line with matching tag and displays the data at the index which is the number formed by n-w to end of address bits.

In case of a MISS program call the insert block function which first generates all the address that the given addresses block contains. Then check for all addresses if any of them have been updated (all updated address is stored in a HashMap to keep track of the address that get updated before their replacement) if yes the that word in the block gets the updated value if no then it retains its old value. Once program create this block then program replace this block by the least frequently used block. Program obtains least frequently used block by the LRU it maintains.

3. **Set-Associative Mapping**

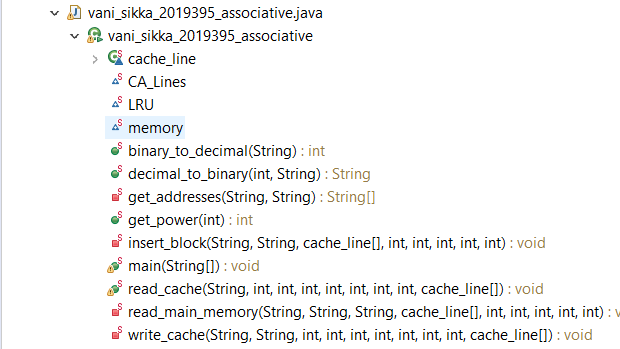
The code basically takes the basic specification of cache a input. From that input it makes an array of my defined class cache lines; which has two property tag and block which is also an array. This block array is of size entered by the user where every index of the block array represents the word number. When the user enters a command be it either read or write. The program slice the address into three parts the first LSB bits from 0 to n-w-p (where n is the exponent of the memory+

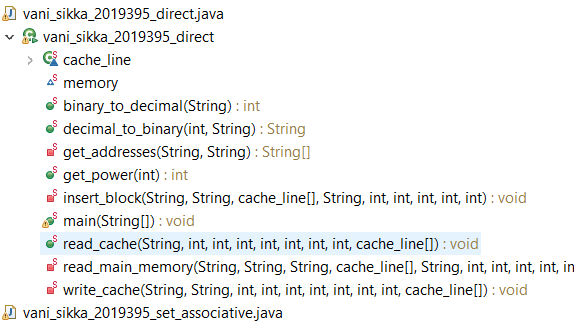
is the exponent of the block size and p is the exponent of the set size) represent the tag. Then the Next LSB bits from n-w-p to n-w tell the set number that address belong to. Then the last MSB bits from n-w to the end of the address tells which word is to be accessed in the block. On slicing the address program go to the index which is the number specified the bits between n-w-p to n-w. At that index of the cache line check all the tags of k- cache lines that are in the set. If one of them have the tag same as address tag program declare a HIT else a MISS. In case of a HIT when the program goes to the block array of the cache line with matching tag and displays the data at the index which is the number formed by n-w to end of address bits.

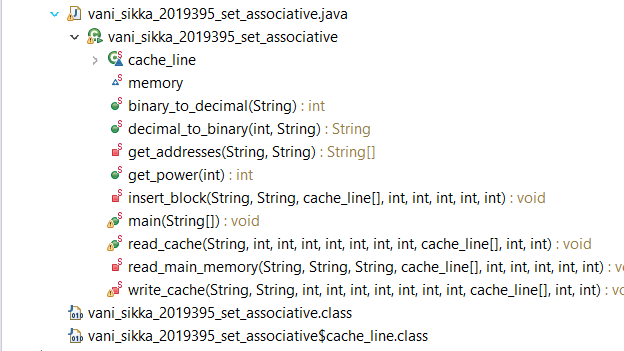
In case of a MISS program call the insert block function which first generates all the address that the given addresses block contains. Then check for all addresses if any of them have been updated (all updated address is stored in a HashMap to keep track of the address that get updated before their replacement) if yes the that word in the block gets the updated value if no then it retains its old value. Once program create this block then program replace this block at the stating of the set.

**LIST OF FUNCTIONS**

Herby, follows a list of all the functions used in the Java program for the Cache:







FUNCTIONS AND THEIR EXPLAINATION

Hereby, follows the list of functions used in the program with a quick explanation about them.   
The list is sorted in the order of the appearance of the functions in the code.

**static** **class** cache\_line

It is a class that represents every line in the cache. It has two parts to it .The tag part of the class is of string type storing the tag address present at that cache line. The second part is block i.e. an array strings where each index in array act as a word number and the data stored in that index is the data stored at that word number.

**static** Map<String,String> *memory*=**new** HashMap<>();

Memory is a HashMap which keeps a record of all the addresses which have been recently been updated. It is important because if the block gets removed then also the HashMap remembers the words that was updated in that replaced block.

**public** **static** **void** main(String[] args)

This function takes the cache specification as input. The then initialize the cache with those inputs. Then the function takes input from the user about their write or read commands. Depending upon what command the user has entered it calls write\_cache or read\_cache functions. It takes commands continuously till the time user writes “STOP”.

**public** **static** **void** print\_cache(cache\_line[] cache)

Function to print the entire cache.

**private** **static** **void** write\_cache

This function check if the data to be written is less than equal to eight bits; if not then it displays error. Then the function slices the input address according to the mapping rules. After slicing it searches in the cache using mapping specific technique. If the particular address is present in the cache then it declares “HIT” and displays the contents of the entire cache line where the address is found after writing in the cache. If the address is not present in the cache insert\_block function is called to insert the block which has that address and it also declares a “MISS”.

**public** **static** **void** read\_cache

Then the function slices the input address according to the mapping rules. After slicing it searches in the cache using mapping specific technique. If the particular address is present in the cache then it declares “HIT” and displays the contents stored in the address entered by the user. If the address is not present in the cache insert\_block function is called to insert the block which has that address and it also declares a “MISS”. Once the block is inserted then it displays the content stored in the given address.

**private** **static** **void** read\_main\_memory

This function is called when there is miss and a new block is to be inserted. Hence it displays the contents stored in the memory address if there is no address stored it displays null. Then it call the insert block function to insert block in the cache.

**private** **static** **void** insert\_block

The function calls get address function to get all the address present in the block. Then it iterates over all the function to see if any of the address has got updated recently by searching in the memory HashMap. If any of the address at that block has got updated then the updated value is stored at that address and if not updated then null is stored. Finally, when all the addresses has got their respective data the block is assigned at the cache line passed as parameter in the cache.

**private** **static** String[] get addresses

This function generates all the address that a block has. It does by finding all the binary strings that are possible of length of the wd\_no passed as the parameter. Then it makes an array if String where all the permutations succeeding the block number passed as parameter is stored. Hence this way the array contains all the address that a block has. This array is returned by the function.

**public** **static** String decimal\_to\_binary

The function returns the binary string of the same length as the word length of the passed integer.

**public** **static** **int** binary\_to\_decimal

it returns the integer value of the binary string passed as the parameter.

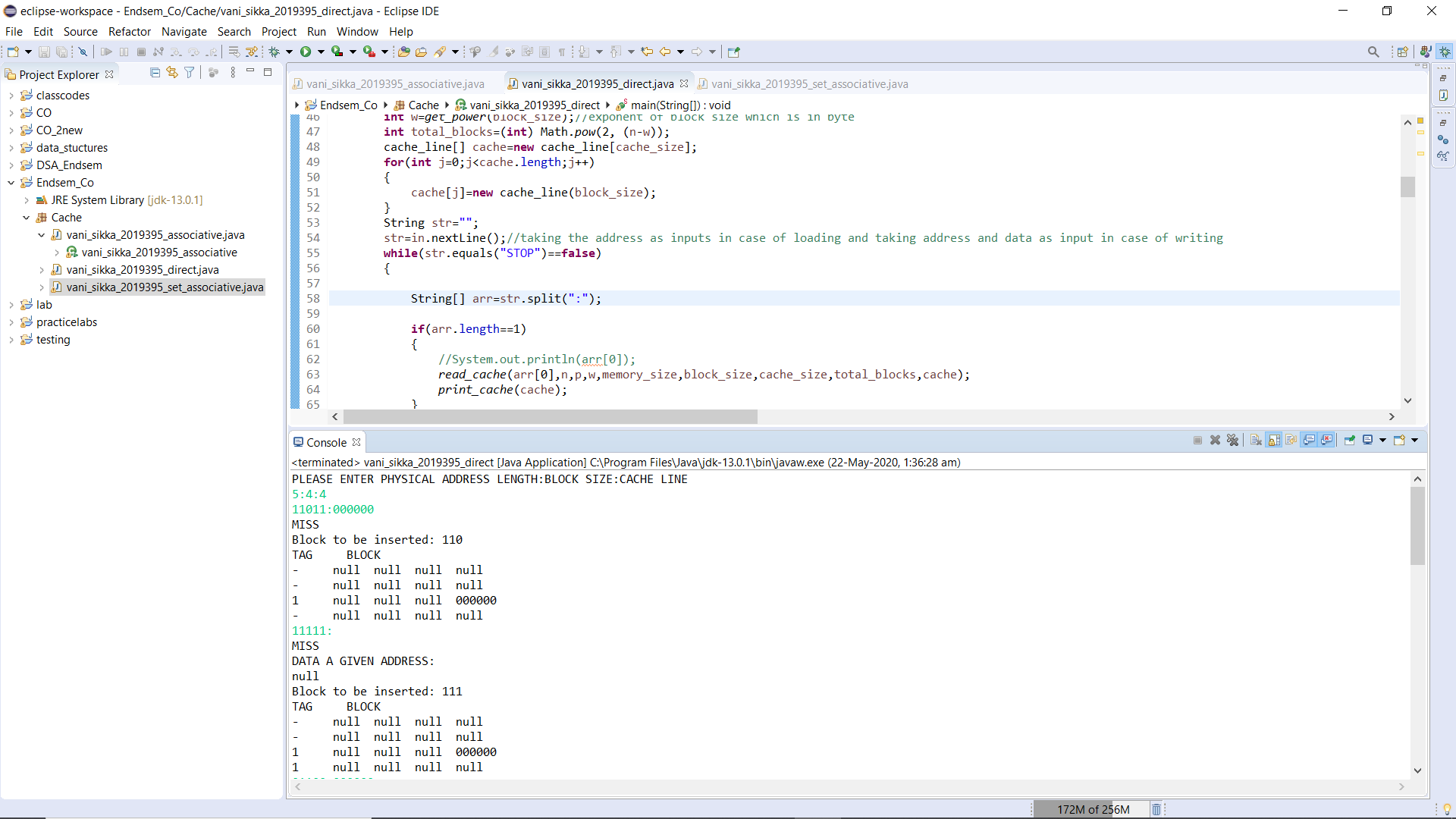
**public** **static** **int** get\_power

The function returns the exponent of 2 that equals to the number passed.

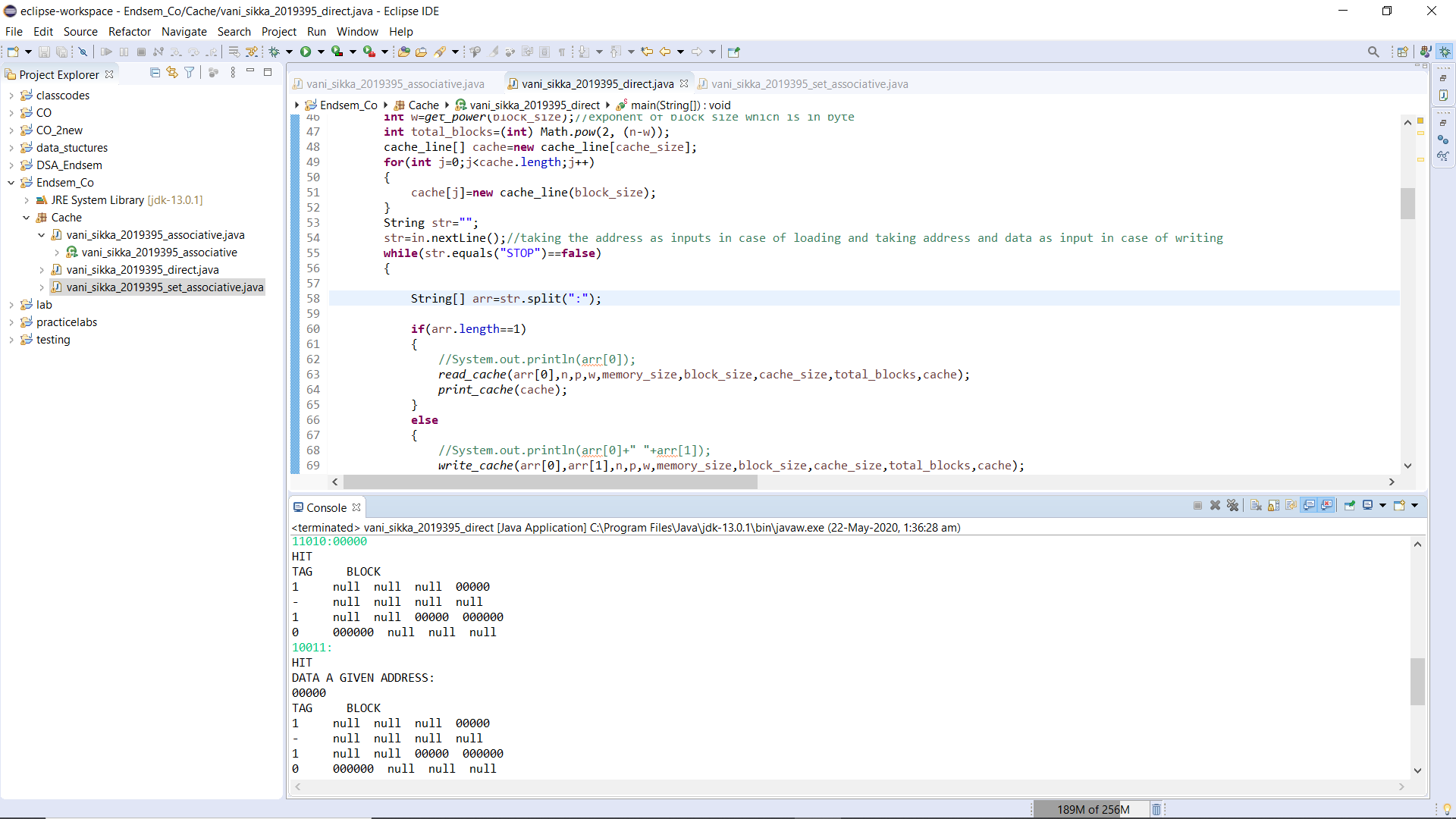
RUNNING THE CACHE

Herby are few pictures of the working of the cache with a short description followed for each one of them.

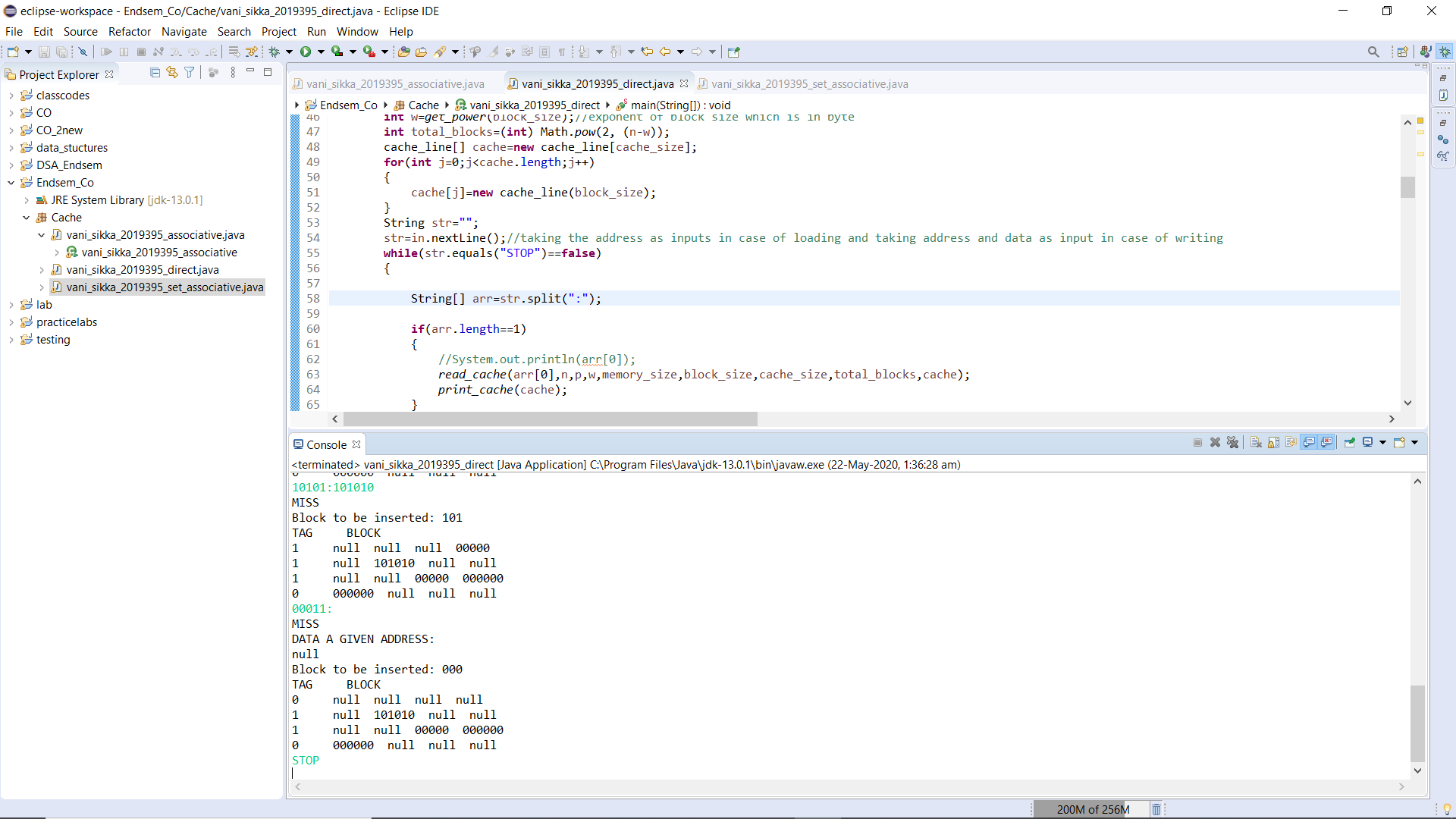
# DIRECT MAPPING



Response when initially the cache is empty i.e. writing and reading on address not present in the cache.

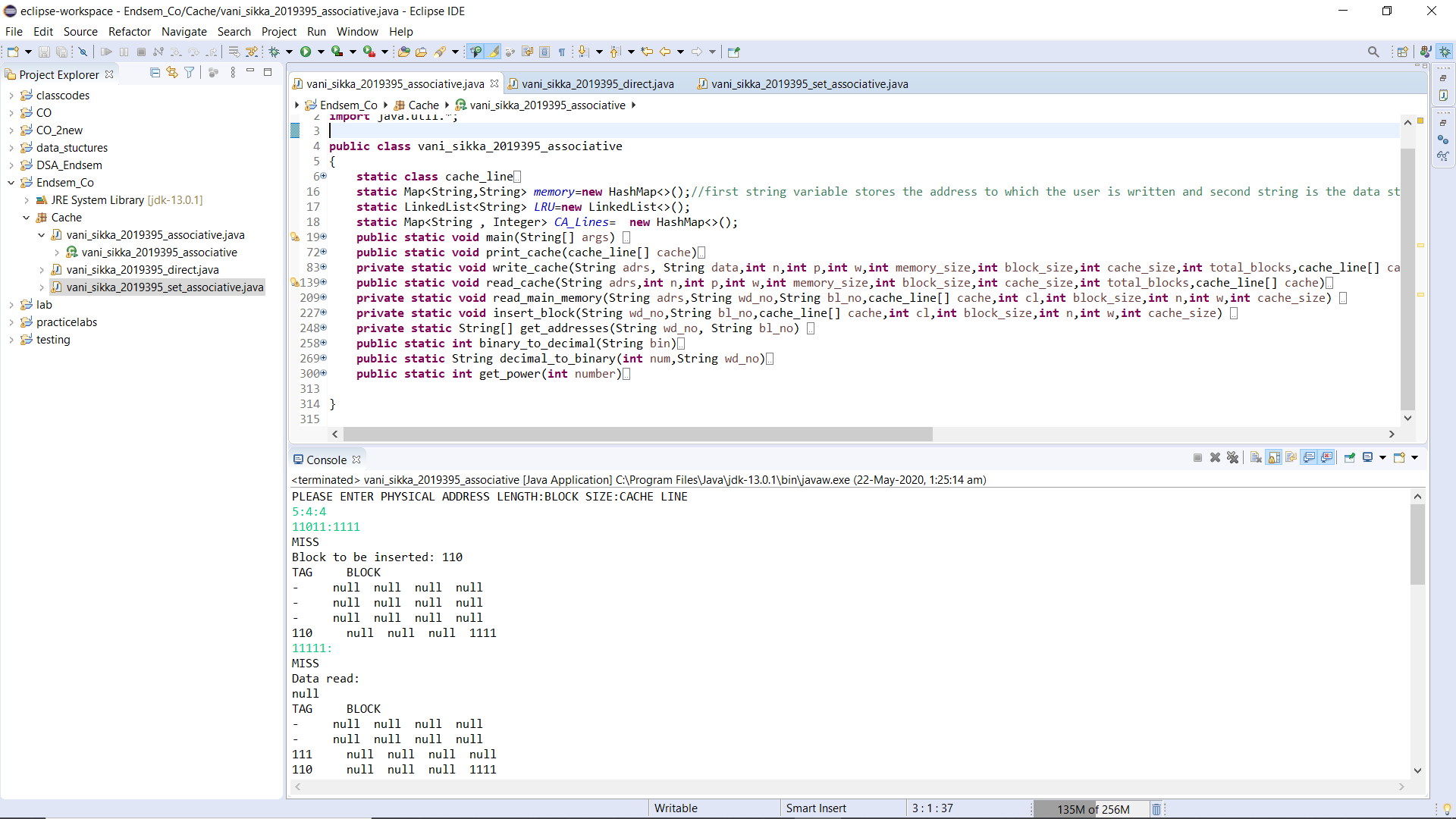


Writing and reading on address present in the cache.

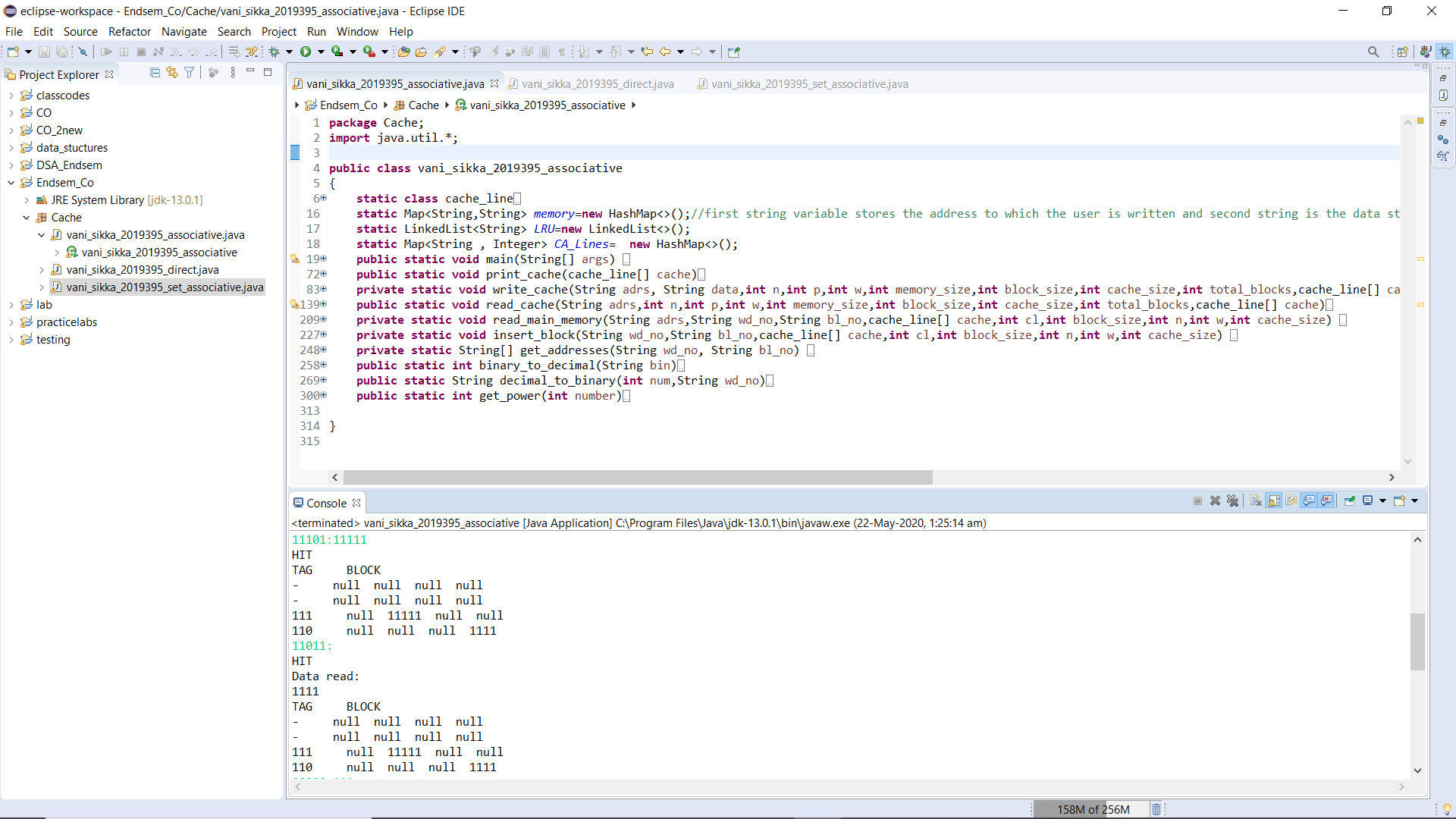


# Block at 00 location gets removed when cache line 00 was occupied by tag 1 and the required block is stored in the cache 00 and tag gets updated to 0.

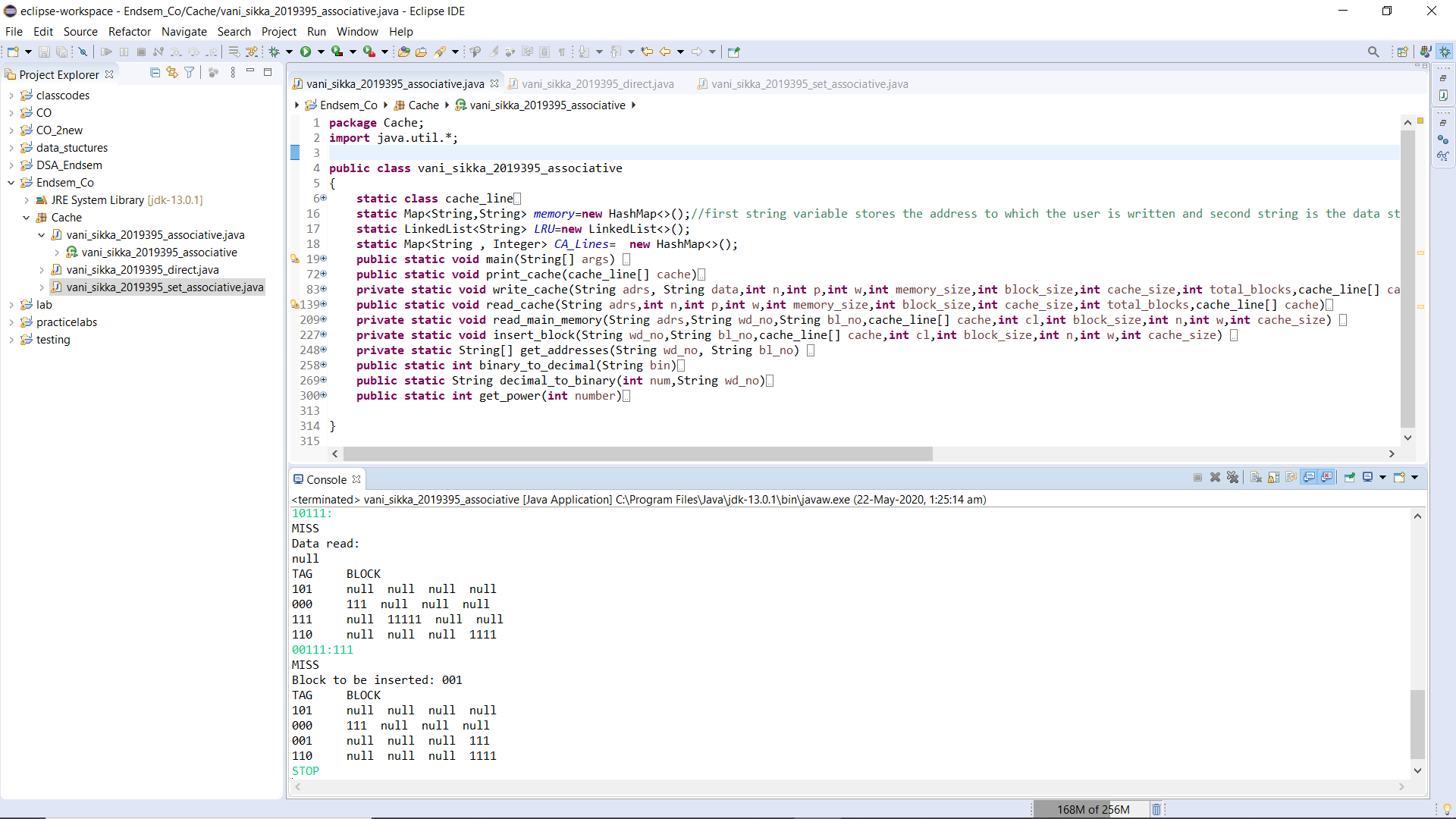
# ASSOCIATIVE MAPPING



Response when initially the cache is empty i.e. writing and reading on address not present in the cache.

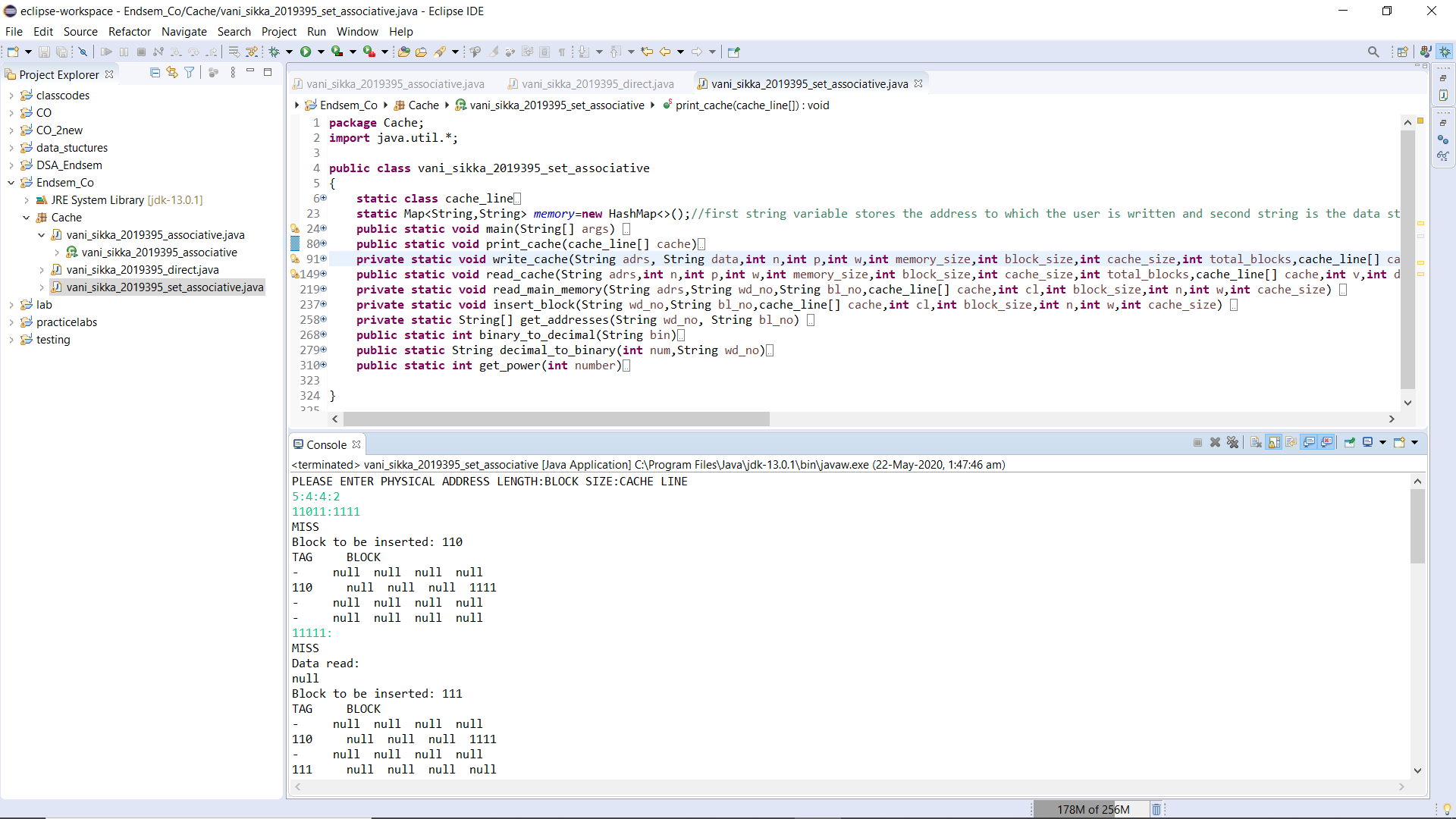


Writing and reading on address present in the cache.

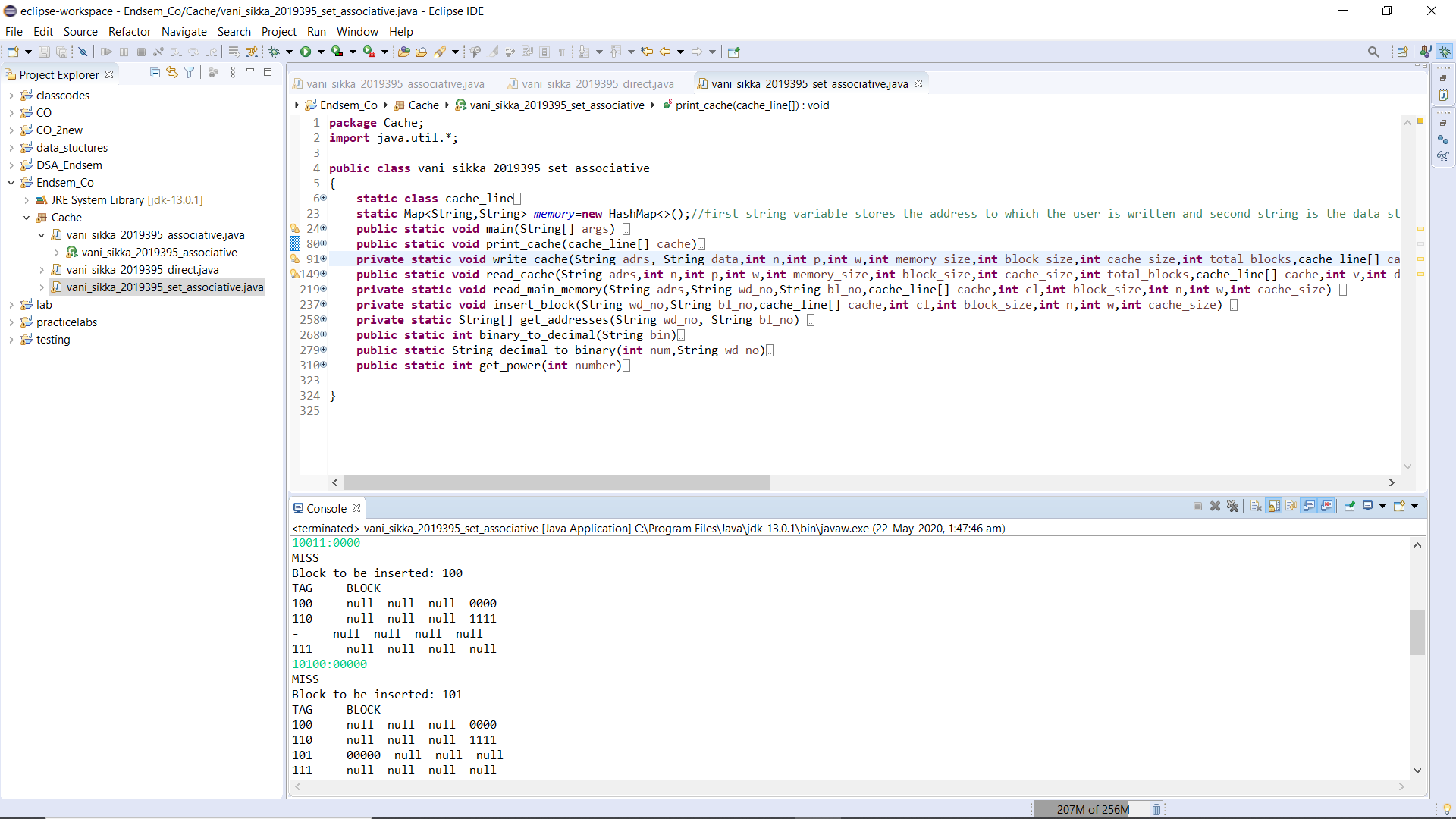


Least used block 111 gets removed when cache is full and the required block is stored in the cache. (Based on LRU function)

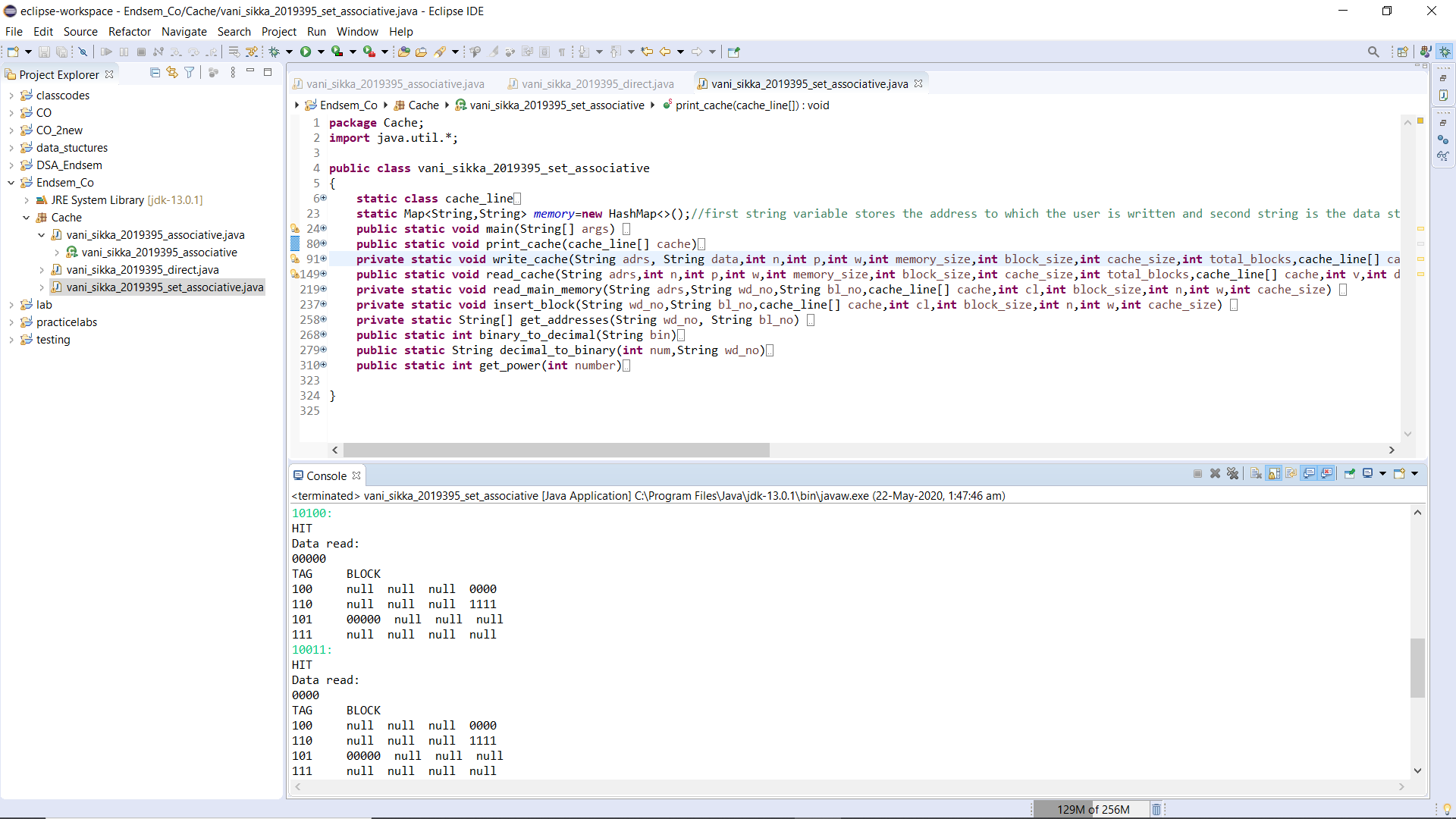
SET-ASSOCIATIVE MAPPING



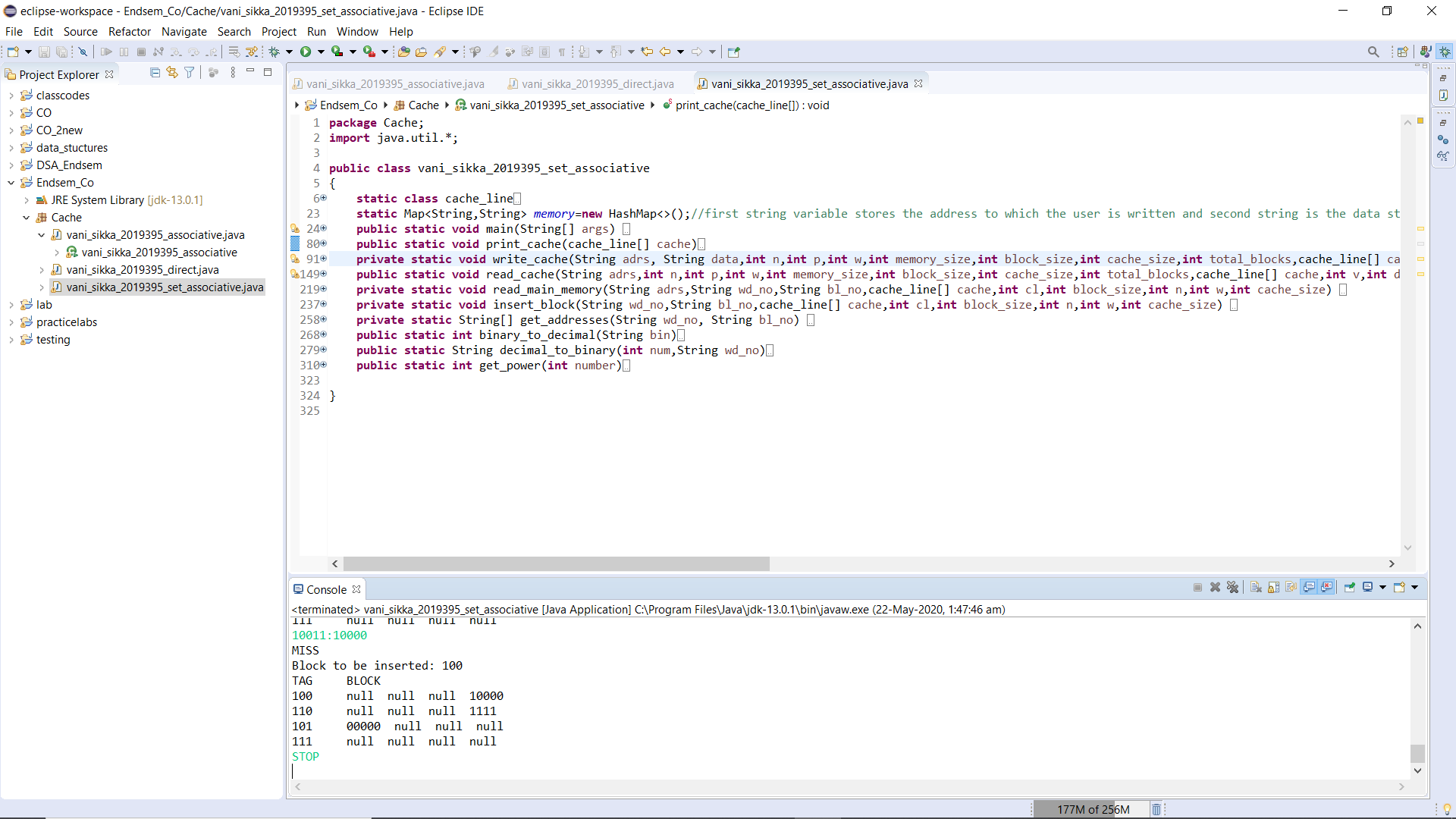
Response when initially the cache is empty i.e. writing and reading on address not present in the cache.



Filling of cache set wise.



Writing and reading on address present in the cache.



Replacement performed when cache is full