Cache Memory Simulation

Miu Daria

Introduction

Context

A simulator for any type of memory can be a very useful tool in the learning process of the structure of computers.
This specific application can be used by those who are trying to understand the way data is written in the cache memory or read from the cache memory. Such an application can bring a more accessible approach to a topic that could be confusing when starting to learn about layers of computer memory.
To make things easier to understand the app will present a graphical user interface in which the user can introduce data such as the size of the cache memory, the length of addresses, the size of blocks, and the type of mapping. And then by the click of a button the user can watch how blocks of memory are searched, written, read, and every step is being explained on the screen.

Specifications

The cache memory simulator will be implemented using Java Programming Language. The IDE used for developing the project is IntelliJ IDEA. The graphical user interface will be developed using java swing.

Objective

The goal of this project is to design, model, and implement an interactive and educative program for simulating the functionality of cache memory. The application will retrieve data introduced by the user in the user interface. It will calculate the missing parameters: the number of blocks, index length, byte offset length, tag length. It will get the mapping type: set-associative or direct-mapped. It will display in 2 tables: the cache memory and the main memory.

Bibliographic study

Definition

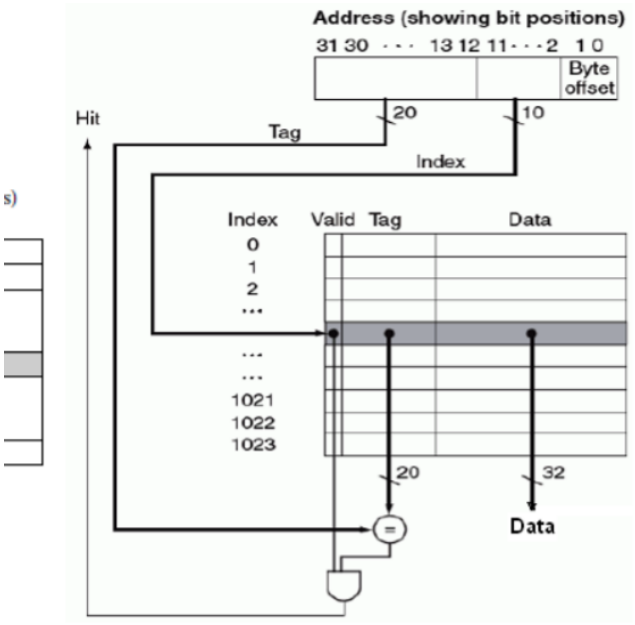
Cache memory consists of a small, fast memory that acts as a buffer for the DRAM memory. (The non-technical definition of cache is a safe place for hiding things.) The cache is built using a different memory technology, static random access memory (SRAM). SRAM is faster but less dense, and hence more expensive than DRAM. SRAM and DRAM are two layers of the memory hierarchy.

How it works

When the processor needs to read or write a location in the main memory, it first checks for a corresponding entry in the cache.
If the processor finds that the memory location is in the cache, a cache hit has occurred and data is read from the cache
If the processor does not find the memory location in the cache, a cache miss has occurred. For a cache miss, the cache allocates a new entry and copies in data from the main memory, then the request is fulfilled from the contents of the cache.

Cache mapping

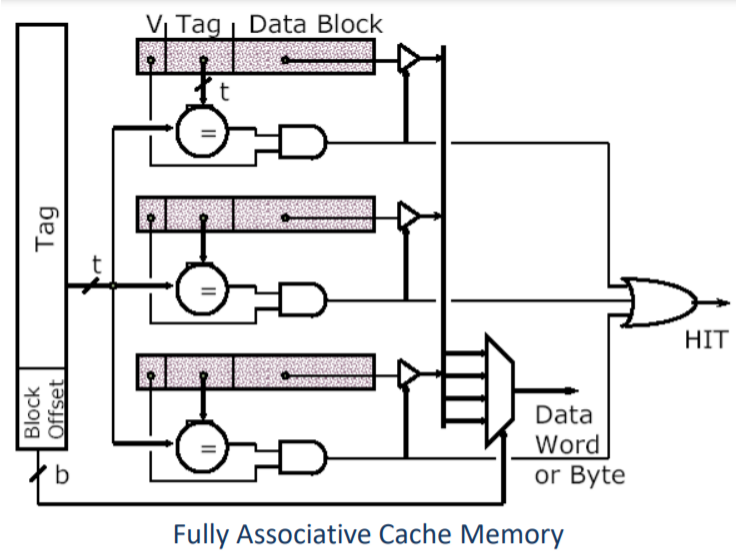
There are 3 different types of cache memory mapping:
• Direct Mapped Cache – each block has only one corresponding place in the cache



Direct Mapped Cache, 1024, 1-word blocks [1]

(Lecture 11 - Computer Architecture - Mihai Negru)

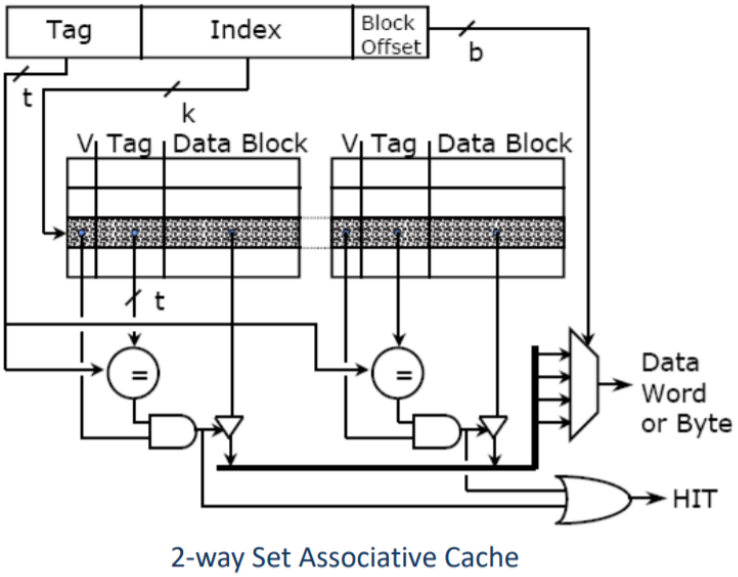
• Fully Associative Cache – a block can be placed anywhere in the cache



Fully Associative Cache Memory

(Lecture 11 - Computer Architecture - Mihai Negru)

• Set Associative Cache – a block can be placed in a restricted set of places in the cache. N sets in a cache N-way Set Associative



2-way Set Associative Cache

(Lecture 11 - Computer Architecture - Mihai Negru)

An ideal cache would anticipate all of the data needed by the processor and fetch it from the main memory ahead of time so that the cache has a zero miss rate.
The miss rate of a Direct Mapped Cache of size X is about the same as for a 2- to 4-way Set Associative cache of size X/2

Analysis,

The first step that the application performs is to calculate the missing parameters after the user introduces the data.
The user enters the following data: address size, cache size, block size.
Other needed information: number of bits for index, the byte offset, the number of bits for the tag.
To explain the mechanism of computation an example is provided below.

- address size : 32 bits
- cache size : 2 MB
- block size : 64 KB

=> to get the number of bits for index :
cache size / block size = 2^5 => 5 bits for index
=> to get the byte offset: block size = 2^16 => 16 bits for byte offset
=> to get the tag, compute how many bits remain : 32 - 5 - 16= 11 bits for tag.

The next step, after all the data is calculated is to initialize the objects of the application with the computed sizes.
Then a random content for the main memory is generated, and the selected operation is performed step by step using the algorithm described above.

Simulation of the Read Operation

For the read operation, the address from which the data should be read is retrieved from the user's input(in hexadecimal format). And then, the algorithm described below is performed.

Simulation of the Write Operation

For the write operation, some characteristics should be established: the write hit and the write miss policy. The application will use the write-back as a write hit policy and the write-allocate as a write miss policy. For the write back a flag bit named the dirty bit will be added in the cache representation(one bit for each cache entry. The dirty bit is 0 (or False) initially, and it becomes 1 (or True) when some byte in data contained in the cache entry is modified. When a cache entry is subject to eviction, if the dirty bit is 1, the data from the entry will be copied in the memory and dirty becomes 0 again.

Algorithm

An important aspect for both reading and writing operations is to establish the block replacement policy. The app uses LRU (least recently used) block replacement policy for associative type cache. For the direct-mapped cache, there is no need for a replacement policy since there will be only one block that can be replaced.

To implement the LRU:

- a hashMap that also keeps track of access order, and the least recently used element will always be the first element in the hashMap, so when eviction is needed the first element is removed.

To generate the memory:

- get the address size and the block size from user inputs and then create a hashMap with 2^address size/block size entries. Each entry will have as key a number and as value a list of block size bytes. For a better understanding see the example below. This will help for a better understanding and will keep the memory organized on blocks.

address length = 4 => 16 bytes
block size = 2 bytes

key value

0	1f 30
1	38 2a
2	...
3	...
4	...
5	...
6	...
7	...

To implement the fundamental algorithm:

- start from input address =>
- get parameters (tag, index, offset) =>
- go to set index =>
- check for the tag =>
- hit or miss =>
- if hit then read or write at the offset
- if miss look for a position with valid = 0 and place the block from memory in the cache; if there is no empty position and eviction is needed use the LRU and also check the dirty bit to see if it is necessary to write the evicted data back to memory.

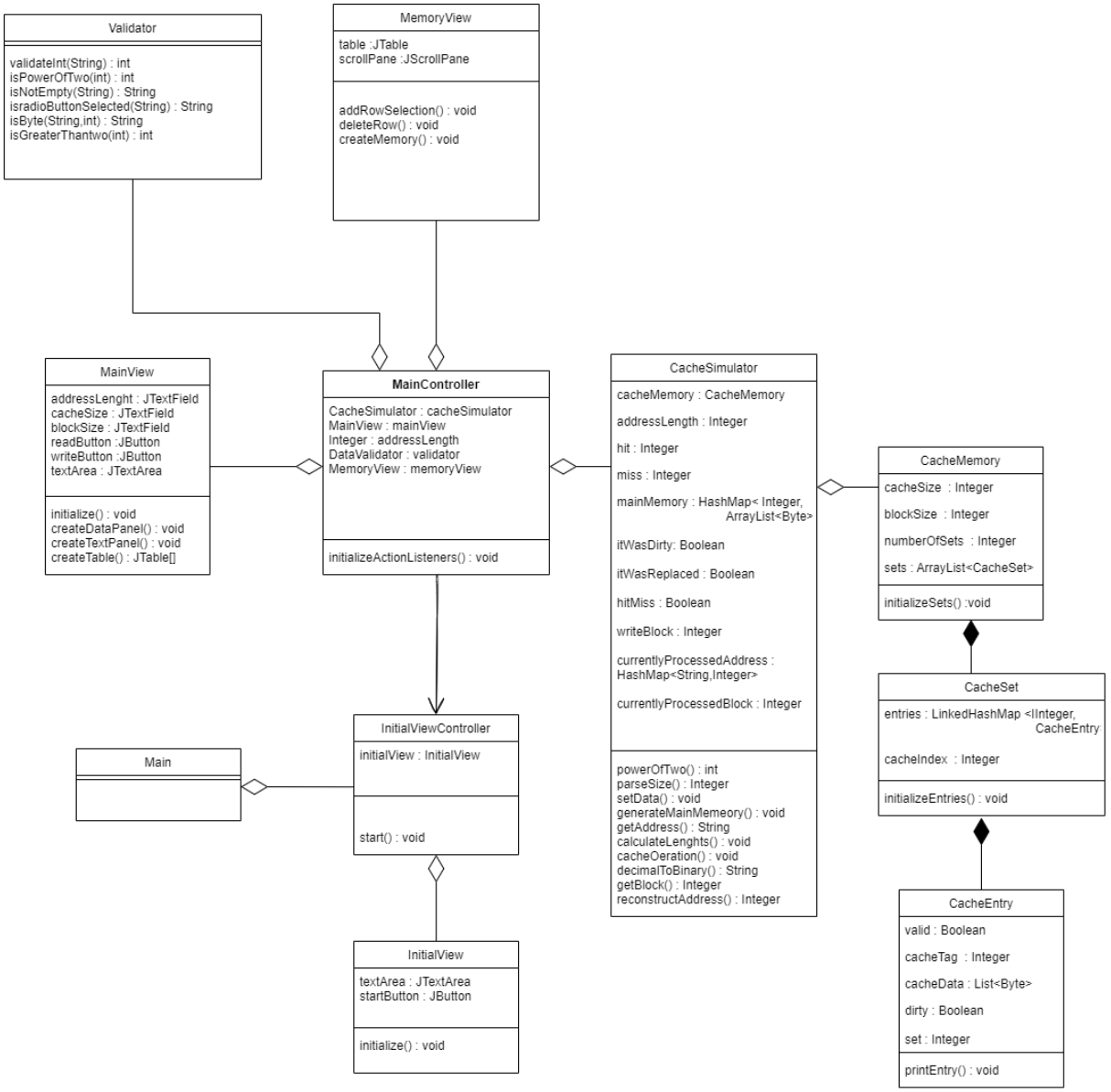
Data Structures

Data structures represent a way in which the data can be stored and organized so that it can be used efficiently and easily. The data structures used in this project are ArrayList, LinkedHashMap, and HashMap.

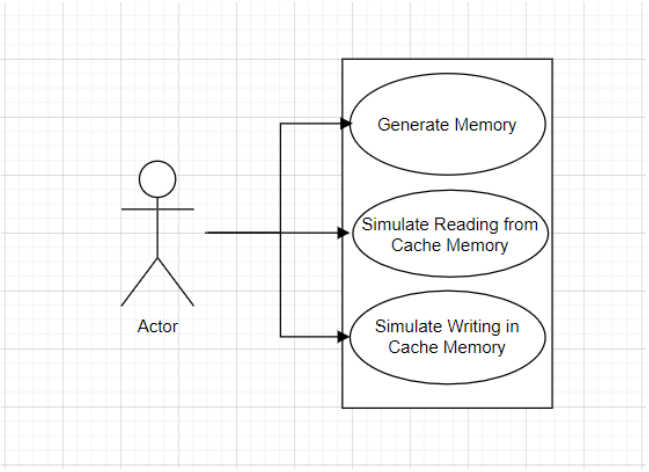
Design

UML Diagrams

Class Diagram & Class Design



Use Case Diagram & Use Case Scenarios



Generate Memory

Succesful scenario
step 1 : user introduces the parameters
step 2 : user presses the generate memory button
step 3 : a new window with a table containing the memory will pop-up
Unsuccessful scenario
step 1 : user introduces the parameters
step 2 : user presses the generate memory button
step 3 : some data is wrong (out of range or not a power of 2) and an error message will pop-up
step 4 : back to step 1

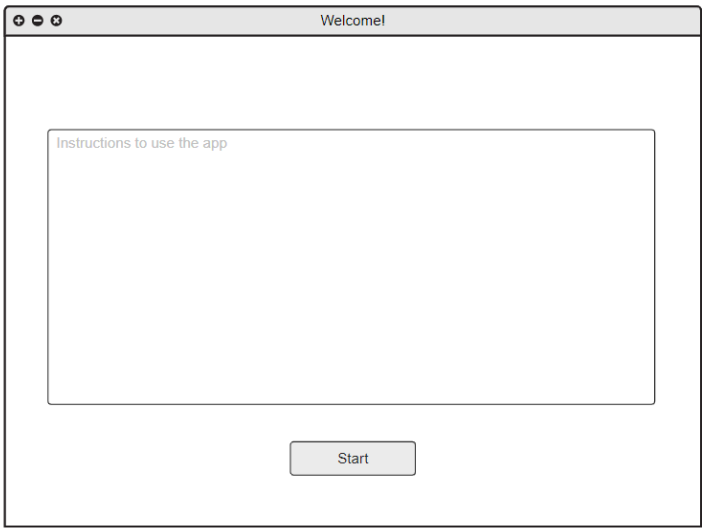
Simulate Reading

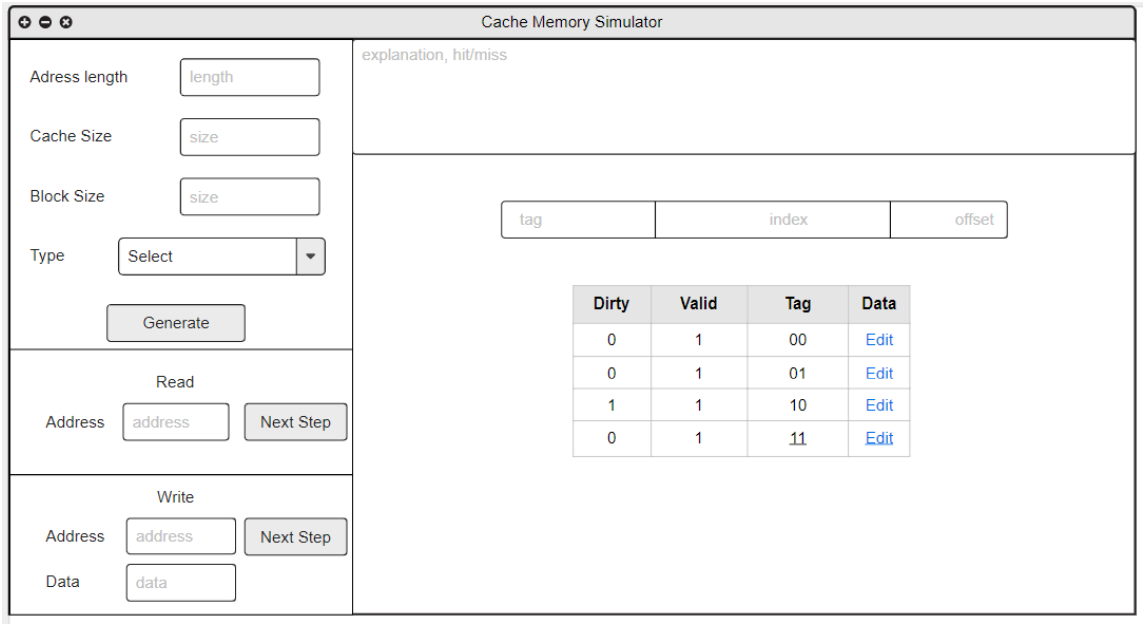
Succesful scenario
step 1 : generate memory successfully
step 2 : user introduces an address to read from
step 3 : click the “Read” button
step 4 : the simulation starts
step5 : the user sees the result and the steps are explained in a text box
Unsuccessful scenario
step 1 : generate memory successfully
step 2 : user introduces an address to read from
step 3 : click the “Read” button
step 4 : some data is wrong (out of range memory location or unknown characters) and an error message will pop-up
step 5 : go back to step 2

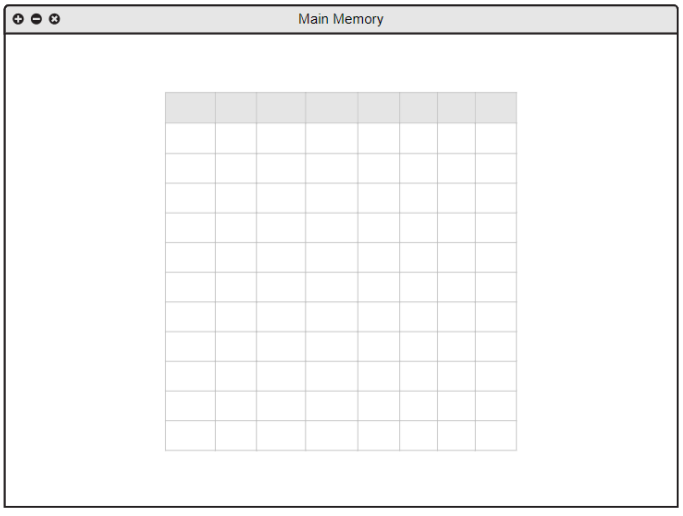
Simulate Writing

Succesful scenario
step 1 : generate memory successfully
step 2 : user introduces an address to write at and data to be written
step 3 : click the “Write” button
step 4 : the simulation starts
step5 : the user sees the result and the steps are explained in a text box
Unsuccessful scenario
step 1 : generate memory successfully
step 2 : user introduces an address to write at and data to be written
step 3 : click the”Write” button
step 4 : some data is wrong (out of range memory location or unknown characters) and an error message will pop-up
step 5 : go back to step 2

User Interface Design







Implementation

The CacheSimulator class contains the most important methods for the logic of the application.

The first important method is setData() in which the cache memory is initialized with the block size, cache size, type(direct, 2 way, 4 way) ; the address length is set, hit and miss is set to 0; the lengths for the tag, index, offset are also calculated. Also, all the sets of the cache are initialized with some default values.
The default values of each cache entry:

```
this.valid = false;
this.cacheTag = 999999;
this.cacheData = null;
this.dirty = false;
```

The method generateMemory() is the next method used. It generates a random memory organized by blocks. Lists of bytes of length block size are generated and added to a HashMap with the key corresponding to the block number.

Here it can be seen how a random array of bytes is generated, then this array of bytes will be added to a list and that list will be added to the HashMap.

```
byte[] randomBytes = new byte[(int) blocksize];
new Random().nextBytes(randomBytes);
Byte[] randomBytes2 = new Byte[(int) blocksize];
```

The first two methods described above had the role of configuring the system in which the cache will be simulated. The next methods are used for the processes of reading and writing data in and from the cache.

The method getAddress() is used to parse the addresses introduced by the user in hexa to binary strings.
This is how the transformation is done

```
String binAddr = Integer.toBinaryString(Integer.parseInt(address, radix: 16));
```

Then zeros are added to the front of the string if needed to reach the actual length of an address in the current configuration.

The method parseAddress() gets as a parameter the address in the binary string form and splits it into the tag, index, and offset by using the lengths for tag, index, offset calculated previously with the calculateLengths() method. The parts of the address are then added to a HashMap for an easier way to store them.

The method cacheOperation() is the most significant method, being the method used to read and write data in and from the cache. It has as parameters the index, the tag, the offset, and the actual address as a binary string, also it has a flag that represents if the operation that

needs to be performed is a write operation and a Byte write data that represents the byte that has to be written in memory in the case of a write operation.
The main idea is the following. First, the set from the wanted index is retrieved from the cache memory. Then the entries from that set are retrieved and also a copy of them because the entries are stored in a LinkedHashMap with the access order flag activated (that means that every put and get operation affects the order of the elements in the hash, which is useful for the LRU algorithm). Then the data from the wanted address is retrieved from the memory using the function getBlock() and a new cache entry is created. This cache entry is going to be inserted somewhere in the cache.
The next step is to iterate through all the elements of that set (can be 1, 2, or 4 elements). At each iteration check is that entry is valid, if so then check if that entry has the same tag as the one we are looking for, if so put that new cache entry in that spot and increase the hit rate, also if it is a write operation also update the byte from the “offset” with the “writeData”. It is needed to reenter the cache entry even if it is already there in order to trigger the reordering of the LinkedHashMap (for the LRU).
If the tag was not found the entry chosen to be replaced with the new cache entry is the first entry from the LinkedHashMap (that is the least recently used one). Also the miss count increases. If the block that has to be replaced has the dirty flag true, the block is written back to memory before being replaced.

The method getBlock() computes from which block of the main memory a certain address should be retrieved.

The method reconstructAddress() is used when writing back to memory a cache block that has the dirty bit ‘1’. From the tag and the index, the method computes the block in memory where the data has to be rewritten.

The initializeSets() and initializeEntries() methods from the CacheMemory and CacheSet classes initialize the memory with the right number of sets and entries and sets the default values.

Another important class of the project is the DataValidator class which is composed of methods for validating the data from the user. A more complex validating method is the isByte() method that checks if the data introduced for reading and for writing addresses and the writeData are in a correct form to be interpreted as bytes.

```
String possibleCharacters = new String( original: "abcdefABCDEF1234567890");
if ((s.length() > length)) {
    throw new RuntimeException("Value is grater than the capacity of the memory");
} if(StringUtils.isBlank(s)){
    throw new RuntimeException("Field is empty");
}
for (int i = 0; i < s.length(); i++) {
    if (possibleCharacters.indexOf(s.charAt(i)) == -1) {
        throw new RuntimeException("Wrong character");
    }
}
return s;
```

User Interface Implementation

The user interface consists of 3 views, one initial view, the main view of the application and the view with the random memory. The first one is controller by the InitialViewController, the second and the third ones are controlled by the MainController in which the action listeners for the buttons are implemented.

The generateButtonAction method calls functions from the CacheSimulator class (setData(), generateMemory() then calls the functions from the views that create the tables with the cache and the main memory according to the parameters introduced by the user. Also, the read, write and flush buttons are activated in this method, but only after a valid memory configuration is created.

The readButtonAction and writeButtonAction methods call functions from CacheSimulator (getCacheMemory(), getAddress(), parseAddress(), decimalToBinary()) then call methods from the view to set the new user interface view with the updated data. The updated data consists of adding new rows in the cache tables and explaining the steps of solving the actions in the text box area. The message to be displayed in the text area is also created in these methods by using a string builder to append all the necessar explanetion. Some important aspects to mention here are the calls to the CacheSimulator to get the hitMIss, itWasDirty, itWasReplaced flags that are modified in the cacheOperation() method and are used to display the correct explanation messages.
Another thing that should be mentioned is that in the writeButtonAction method there is also a portion of code for retrieving the writeData from the user in a byte format.

```
int i = Integer.decode("0x" + validator.isByte(mainView.getWriteData(), length: 2));
byte b = (byte) i;
Byte myByte = b;
```

In the MainView class the most significative methods are the ones for creating the cache tables:

```
private JTable[] createTable(CacheMemory cacheMemory, Integer type){
```

CreateTable method returns one or 2 or 4 tables depending on the mapping type of the cache. The tables are filled with default values then updated as the simulation proceeds

```
public void createCachePanel(CacheMemory cacheMemory, Integer type){
```

CreateCachePanel method attaches the tables generated with the method above to the JScrollPanes in order to display them.

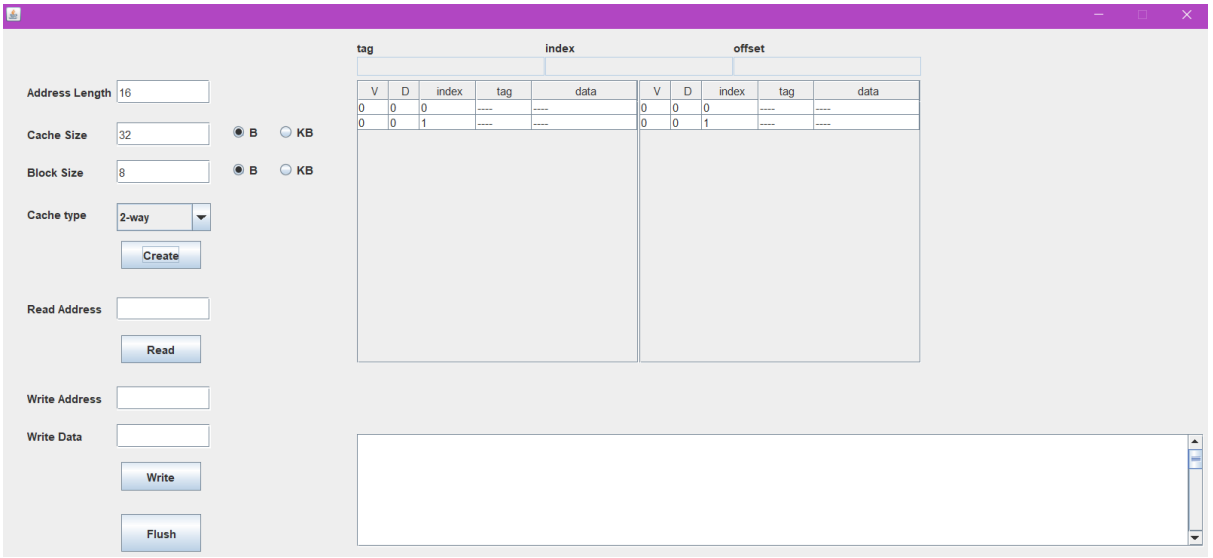
In the MemroyView class the method for creating the table is:

```
public void createMemory(HashMap<Integer, List<Byte>> memory){
```

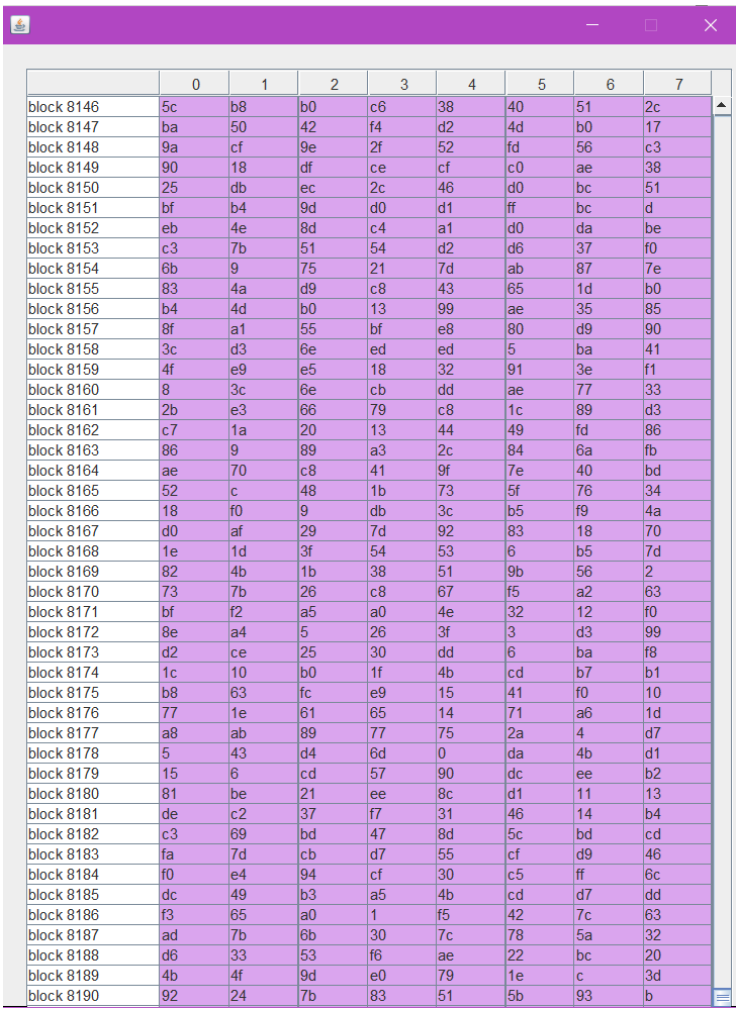
The method creates a table based on the memory randomly generated and organized in blocks.

Testing

For testing, I will present some screenshots from the user interface.
This is the initial view in which I introduced some sample data.

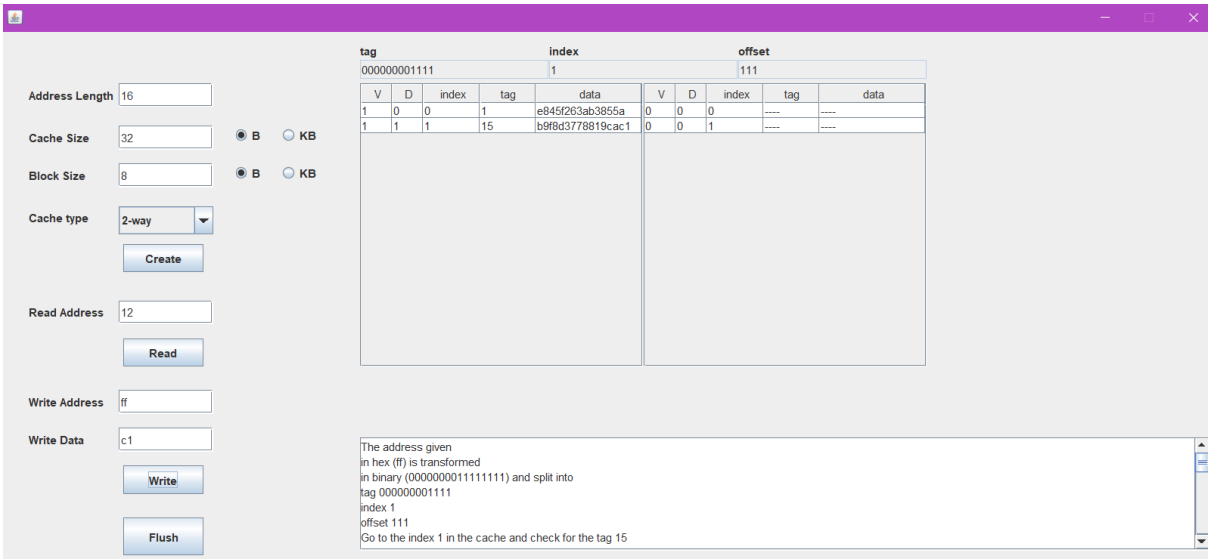


Here is the memory generated(the last rows of it).

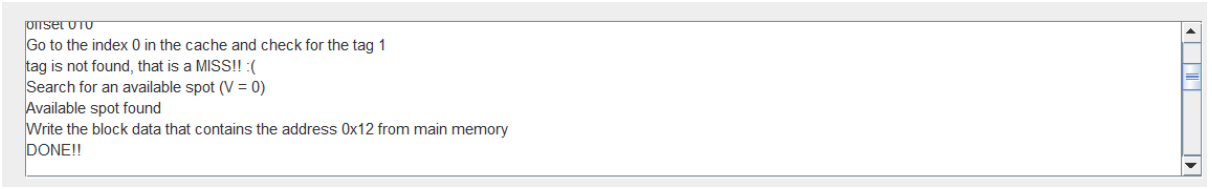


	0	1	2	3	4	5	6	7
block 8146	5c	b8	b0	c6	38	40	51	2c
block 8147	ba	50	42	f4	d2	4d	b0	17
block 8148	9a	cf	9e	2f	52	fd	56	c3
block 8149	90	18	df	ce	cf	c0	ae	38
block 8150	25	db	ec	2c	46	d0	bc	51
block 8151	bf	b4	9d	d0	d1	ff	bc	d
block 8152	eb	4e	8d	c4	a1	d0	da	be
block 8153	c3	7b	51	54	d2	d6	37	f0
block 8154	6b	9	75	21	7d	ab	87	7e
block 8155	83	4a	d9	c8	43	65	1d	b0
block 8156	b4	4d	b0	13	99	ae	35	85
block 8157	8f	a1	55	bf	e8	80	d9	90
block 8158	3c	d3	6e	ed	ed	5	ba	41
block 8159	4f	e9	e5	18	32	91	3e	f1
block 8160	8	3c	6e	cb	dd	ae	77	33
block 8161	2b	e3	66	79	c8	1c	89	d3
block 8162	c7	1a	20	13	44	49	fd	86
block 8163	86	9	89	a3	2c	84	6a	fb
block 8164	ae	70	c8	41	9f	7e	40	bd
block 8165	52	c	48	1b	73	5f	76	34
block 8166	18	f0	9	db	3c	b5	f9	4a
block 8167	d0	af	29	7d	92	83	18	70
block 8168	1e	1d	3f	54	53	6	b5	7d
block 8169	82	4b	1b	38	51	9b	56	2
block 8170	73	7b	26	c8	67	f5	a2	63
block 8171	bf	f2	a5	a0	4e	32	12	f0
block 8172	8e	a4	5	26	3f	3	d3	99
block 8173	d2	ce	25	30	dd	6	ba	f8
block 8174	1c	10	b0	1f	4b	cd	b7	b1
block 8175	b8	63	fc	e9	15	41	f0	10
block 8176	77	1e	61	65	14	71	a6	1d
block 8177	a8	ab	89	77	75	2a	4	d7
block 8178	5	43	d4	6d	0	da	4b	d1
block 8179	15	6	cd	57	90	dc	ee	b2
block 8180	81	be	21	ee	8c	d1	11	13
block 8181	de	c2	37	f7	31	46	14	b4
block 8182	c3	69	bd	47	8d	5c	bd	cd
block 8183	fa	7d	cb	d7	55	cf	d9	46
block 8184	f0	e4	94	cf	30	c5	ff	6c
block 8185	dc	49	b3	a5	4b	cd	d7	dd
block 8186	f3	65	a0	1	f5	42	7c	63
block 8187	ad	7b	6b	30	7c	78	5a	32
block 8188	d6	33	53	f6	ae	22	bc	20
block 8189	4b	4f	9d	e0	79	1e	c	3d
block 8190	92	24	7b	83	51	5b	93	b

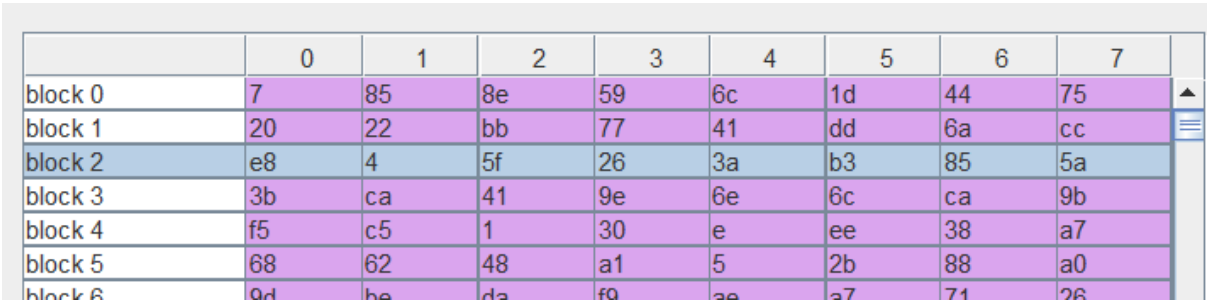
Here is the view after reading from address 0x12 and writing at address 0xff the byte c1. The byte is replaced only in the cache for now!!!



The rest of the explanation cannot be seen in the first screenshot for the writing operation.

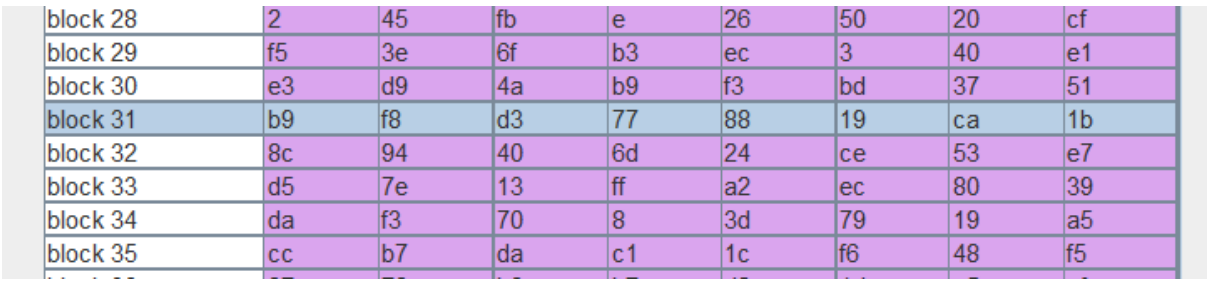


The block loaded at index 0 after the reading operation(view from the main memory)



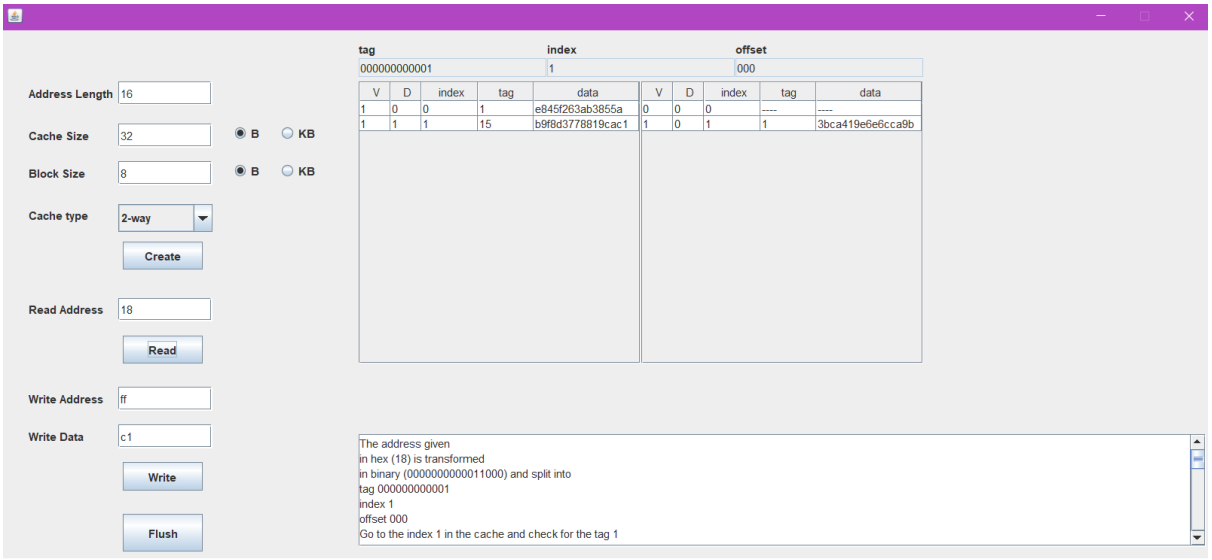
	0	1	2	3	4	5	6	7
block 0	7	85	8e	59	6c	1d	44	75
block 1	20	22	bb	77	41	dd	6a	cc
block 2	e8	4	5f	26	3a	b3	85	5a
block 3	3b	ca	41	9e	6e	6c	ca	9b
block 4	f5	c5	1	30	e	ee	38	a7
block 5	68	62	48	a1	5	2b	88	a0
block 6	9d	be	da	f9	ae	a7	71	26

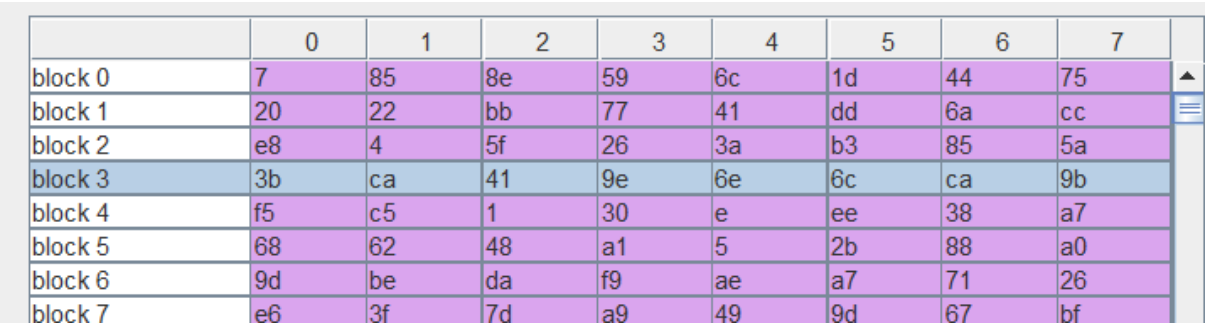
The block loaded at index 1 after the write operation(view from the main memory)



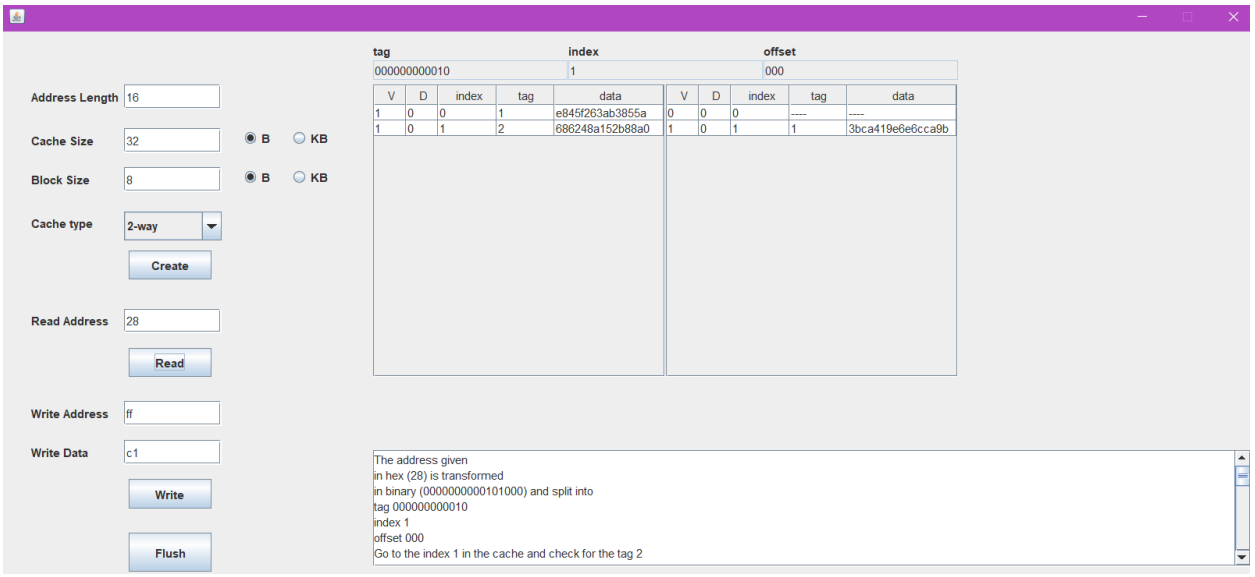
block 28	2	45	fb	e	26	50	20	cf
block 29	f5	3e	6f	b3	ec	3	40	e1
block 30	e3	d9	4a	b9	f3	bd	37	51
block 31	b9	f8	d3	77	88	19	ca	1b
block 32	8c	94	40	6d	24	ce	53	e7
block 33	d5	7e	13	ff	a2	ec	80	39
block 34	da	f3	70	8	3d	79	19	a5
block 35	cc	b7	da	c1	1c	f6	48	f5

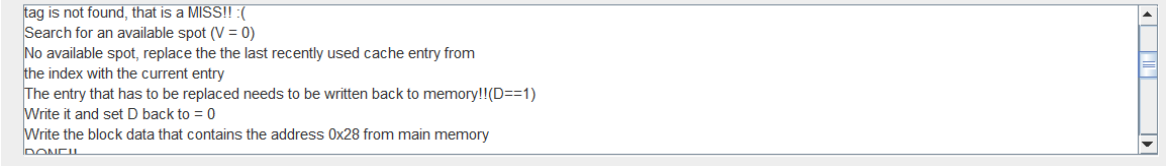
Do 2 more operations, both of them on the row with index 1 so that the write-back policy can also be illustrated.
Read from address 0x18 and then read from address 0x28. The second operation will generate the eviction of the data that was previously loaded from address ff, since that is the data that was least recently used.





	0	1	2	3	4	5	6	7
block 0	7	85	8e	59	6c	1d	44	75
block 1	20	22	bb	77	41	dd	6a	cc
block 2	e8	4	5f	26	3a	b3	85	5a
block 3	3b	ca	41	9e	6e	6c	ca	9b
block 4	f5	c5	1	30	e	ee	38	a7
block 5	68	62	48	a1	5	2b	88	a0
block 6	9d	be	da	f9	ae	a7	71	26
block 7	e6	3f	7d	a9	49	9d	67	bf





block 4	f5	c5	1	30	e	ee	38	a7
block 5	68	62	48	a1	5	2b	88	a0
block 6	9d	be	da	f9	ae	a7	71	26
block 7	e6	3f	7d	a9	49	9d	67	bf
block 8	c6	8a	50	31	e9	ca	f5	3b
block 9	58	ff	d7	cb	c1	42	f7	6f
block 10	2c	76	bc	6c	b2	54	ed	6a
block 11	97	c	19	6e	e7	aa	8d	e6
block 12	8	99	fd	b3	62	1d	7b	e6
block 13	52	79	33	98	c7	89	44	41
block 14	3f	71	a5	4f	2c	f1	36	1b
block 15	e2	cf	e0	e0	b0	9d	7e	6a
block 16	9c	de	d2	d0	c9	96	cb	a
block 17	c7	9c	47	32	8f	a4	6a	9
block 18	aa	88	9	cd	85	b1	30	18
block 19	84	35	9a	4	ae	6d	41	2
block 20	d7	6c	4a	1a	53	34	0	9e
block 21	6f	51	bd	68	d1	87	92	69
block 22	4a	43	b7	47	10	c	ea	66
block 23	0	3a	89	9e	dd	28	1	be
block 24	d2	16	9f	8e	cd	b9	d8	e8
block 25	4c	5e	81	4c	65	e9	77	17
block 26	7e	89	50	e4	a3	3	e2	72
block 27	aa	b1	86	2f	a6	b7	e1	57
block 28	2	45	fb	e	26	50	20	cf
block 29	f5	3e	6f	b3	ec	3	40	e1
block 30	e3	d9	4a	b9	f3	bd	37	51
block 31	b9	f8	d3	77	88	19	ca	c1
block 32	8c	94	40	6d	24	ce	53	e7

Conclusion

The goal of the project (creating a functional cache memory simulator with a friendly user interface) was reached successfully. The application can be used as a tool to understand the most basic operations (read, write) performed in a cache memory that supports direct mapping, 2 way, and 4-way mapping, and was an LRU update policy.

Working on the project was a challenge for me since I did not have a really defined knowledge about cache memories, even though have studied them last year. So, this project ended up being a great tool for me to learn about cache memories, and hopefully, the final application will be eventually helpful for others.

Bibliography

https://ict.iitk.ac.in/wp-content/uploads/CS422-Computer-Architecture-ComputerOrganizationAndDesign5thEdition2014.pdf
http://users.utcluj.ro/~negrum/wp-content/uploads/2020/02/ca/Lecture11%20-%20Memory%20Hierarchy.pdf
Memory Management Basic Management Techniques - Operating System Course 8 - Adrian Colesa
http://www.csit-sun.pub.ro/courses/cn2/Digital_design_book/Digital%20Design%20and%20Computer%20Architecture.pdf
https://www.geeksforgeeks.org/cache-memory-in-computer-organization/