Computer Organization and Assembly Language Programming

Project 2

Memory Hierarchy Simulator

Dr. Cherif Salama

Dana Alkhouri - 900214106

Mostafa Elfaggal - 900211128

Raef Hany - 900213194

Department of Computer Science and Engineering, The American University in Cairo

Description:

In this project, we worked on implementing the memory hierarchy simulator. The main programming language used is Java script for simplicity and efficiency. In addition, it will be easier to implement the UI. We implemented two bonus features, a UI and set/fully associative caching.

The project is placed on GitHub at the following link:

https://github.com/MostafaBelo/Assembly_Project2

Design Decisions

We are starting by explaining the main file we needed in the project, which is Data.js. It starts with implementing a class called **CACHE** that represents a cache memory system simulation. The cache class has several properties that allow us to implement functionalities and calculate different things that are required; the following are the properties of the cache:

a. Address_size: The memory address size in bits is 24 by default. (input)
b. S: represents the size of the cache in bytes. (input)
c. L: represents the cache line size in the block. (input)
d. C: represents the number of cache lines. (input)
e. Associativity: represents the associativity of the cache. We are implementing direct mapping, set, and full associativity. (input)
f. MissPenalty & HitPenalty: time penalties for cache misses and hits. (input)

On the other part, we are calculating simulation outputs that are changed based on the access of the cache, and they are traced. The following are the outputs:

a. Access: it calculates the number of times we accessed the cache.

b. Hits: counts the number of hits.

c. Miss: counts the number of misses.

d. Memory_time: it counts the time taken to perform operations on the cache from reading and writing.

e. lastAccess: to keep track of the last cache access.

The functions that were used to modify and calculate the variables are:

Function Name	**Functionality**
setPenalty(missPenalty = 120, hitPenalty = 1)	Set miss and hit penalties
isIndexInCache(index, isBin = true)	Checks if a given index is in the cache
isvalid(index, tag)	Checks if the entry at a given index contains the given tag while it is valid or not
getJ(index, tag)	Used to compute the last access, it computes the location within a given index (inside the set) at which the given index and tag are
getRandomInt(min, max)	Used for random placement within a set, computes a random integer from min(inclusive) to max(exclusive)
setAddressSize(size = 24)	Sets the memory address size
get(index)	Obtains the set with the given index, an array of objects. Each object is a line containing valid and tag keys.
read(address)	Attempts to access the given address, updates the hits and calls write in case of a miss
getHitsRatio()	Returns the ratio between hits and total accesses
getMissessRatio()	Returns the ratio between misses and total accesses
getAMAT()	Returns the ratio between total memory time and total number of accesses
write(tag, index, offset)	Handles the process of writing data into the cache, taking

	into consideration the misses, and updating it in the cache's data (validity and tag)
init(S = 0, L = 0, associativity = 1)	Initializes the cache with new parameters.

There is also the parser.js, which handles breaking down the input access sequence into an easy-to-use array of numbers, where each number is a byte address to be accessed. The dataconversions.js is another file that contains helpful side functions used to convert a decimal number to binary. This is used when trying to access an element to convert the address to binary for easier splitting.

The other two main files besides data.js are gui.js and flow.js. These files handle the UI and the flow of the program in general. These files have been mostly copied from our first project since they are greatly similar. The flow breaks down the process into executeStart, which initializes everything. ExecuteNext is called from the UI and handles calling the read with the correct parsed address from the parser. And finally, executeStop which stops all execution.

As for the gui.js, it handles connecting all operations of the backend and the flow to the UI. For that, it allows opening a file within which the access sequence is. It allows for starting the execution, going to the next access, or falling through all executions, as well as stopping execution. All of which simply link to the flow. The gui.js also reads the inputs, outputs, and properties of the cache in the details sections. Finally, the gui.js shows the cache's current state in the bottom section. For that, it views it as a table where every row is a set, while every two columns are a line/bank. Every line has a column for validity and another for the tag. The UI also highlights the last access, which colors the correct line, in red should it have been a miss or in

green if it was a hit. From the details section, one can also specify an address of the set one wishes to highlight for easier readability.

Lastly, there is a test.js file that contains some of our internal testing, especially bug testing. And the main.js file, which ties everything together by creating and linking the necessary objects and classes.

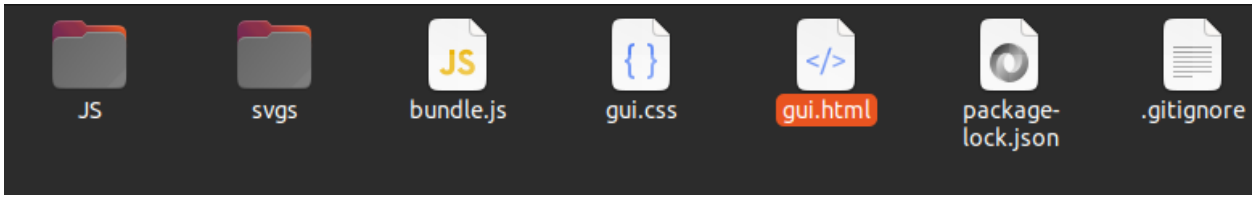
The program is done in JS, which is compiled with Webpack into a bundle.js file, which is injected into an HTML file. The entire program is run by simply opening the gui.html file in a browser.

Known Bugs:

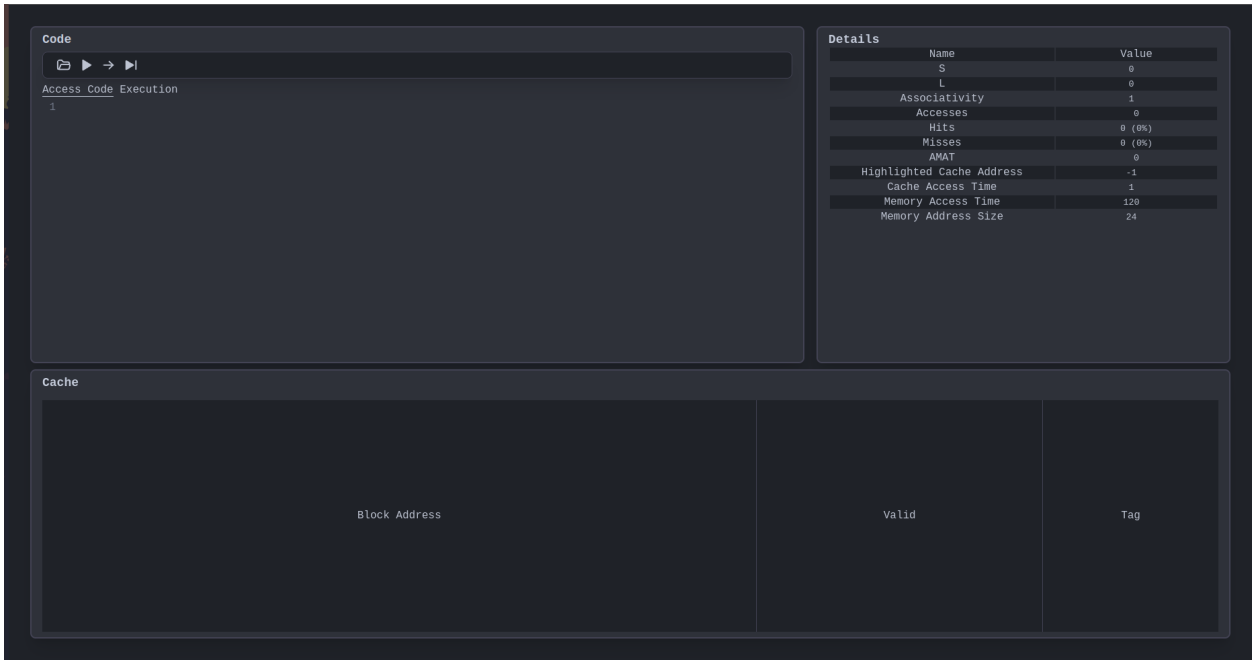
None so far

User Guide:

From the project folder, head to the source folder and open the “gui.html” in a browser of your choice. (The project has only been tested on google chrome, though it should work elsewhere)



The page should look as follows once opened



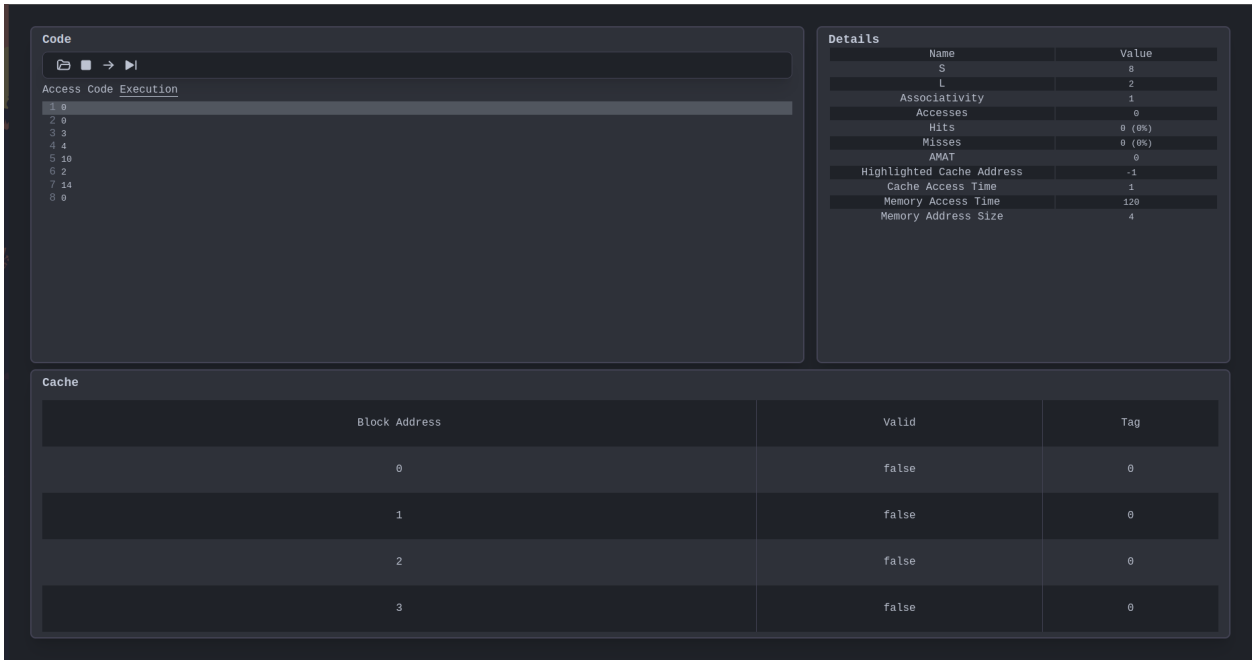
The upper left section is where the access code is written. The upper right section is where details about the program exist. At the bottom, there is the cache.

In the code section, you can write the code manually or open a file to edit/run. Code is written in the format specified with addresses(decimal) separated by commas. You may have white spaces or new lines to ease readability, as our parser ignores them. Pick the open file icon at the top left (the first icon on the left under the title code) to open a file.

The details section on the right is where you may specify some inputs to the program or where you might find some outputs. You may change the S, L, and Associativity fields as they act as inputs. S is the size of the cache, L is the line size, and associativity is the degree of associativity. To have a direct mapping cache, set the associativity to 1. To have a fully associative system, set the associativity to S/L. A set associative system is when associativity is a number in between. The Accesses, Hits, Misses, and AMAT(Average Memory Access Time) fields are outputs.

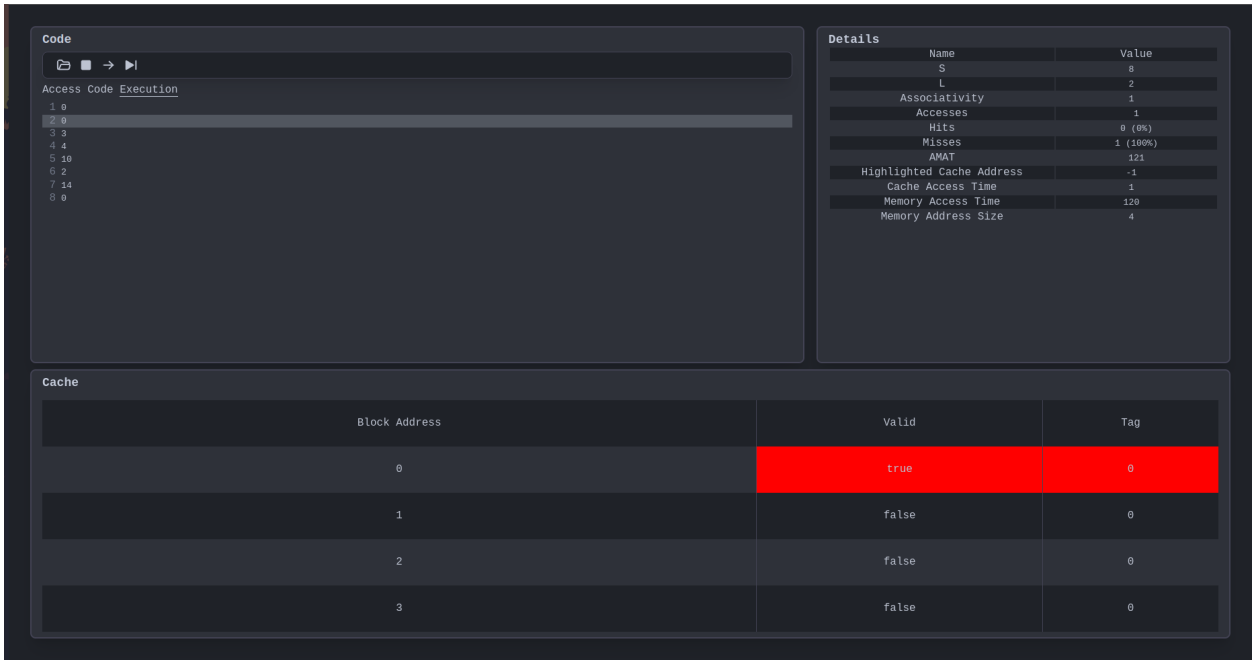
Accesses are the total number of accesses that have been executed. Hits are the total number of hits performed, followed by the percentage of hits between parenthesis. Misses are the total number of misses performed, followed by the percentage of misses between parenthesis. AMAT is the average memory access time in cycles. Highlighted Cache Address, Cache Access Time, Memory Access Time, and Memory Address size are other allowed inputs. Highlighted Cache address allows you to specify an address of a set to highlight in the cache for better readability (to dehighlight all cells, set this field to -1). Cache Access Time is the time in cycles needed to access the cache, which is also the hit penalty. Memory Access Time is the time in cycles needed to access the memory, which yields the miss penalty when added to the Cache Access Time. Finally, change the memory address size to your liking by adjusting the last field in the details section.

Once you have written some access code and adjusted the details section, start the program by clicking the play button at the top of the code section. This will move you to the execution tab and start execution.

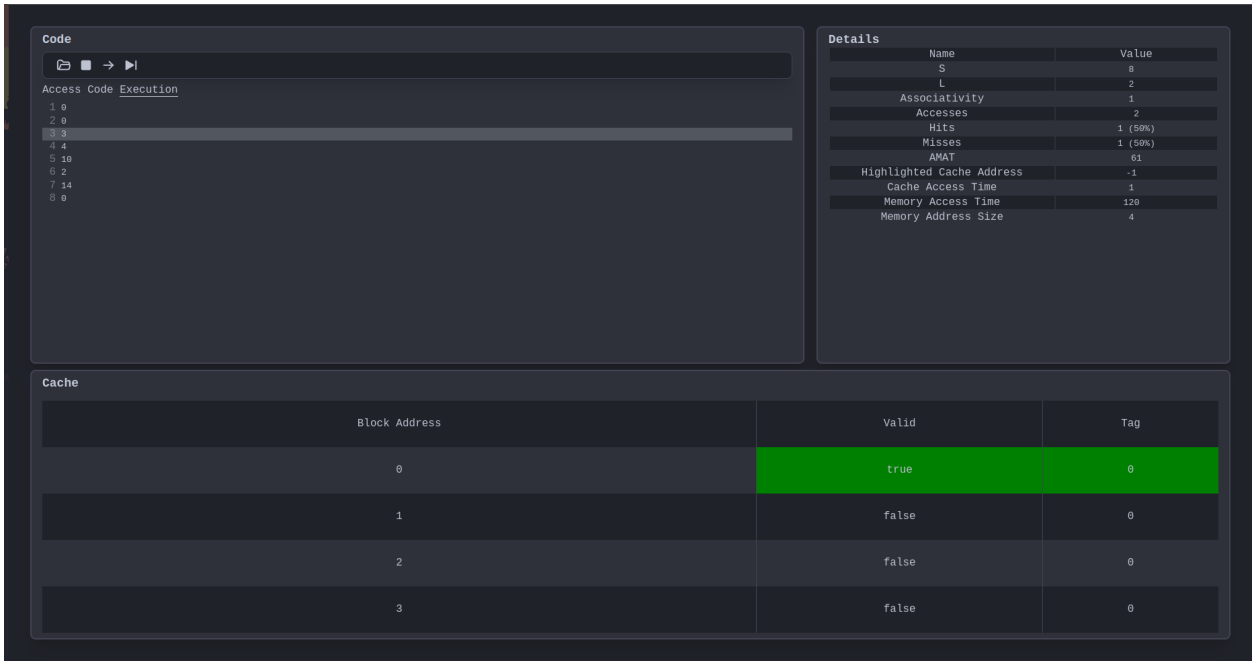


The highlighted line is the one that will be executed. To execute the following line, press the next arrow, the third icon from the left at the top of the code section. To execute all accesses until the end, press the fallthrough button, the fourth icon from the left at the top of the code section. To stop execution, press the stop button at the top of the code section.

While executing, when a miss takes place, the corresponding line will be highlighted in red.

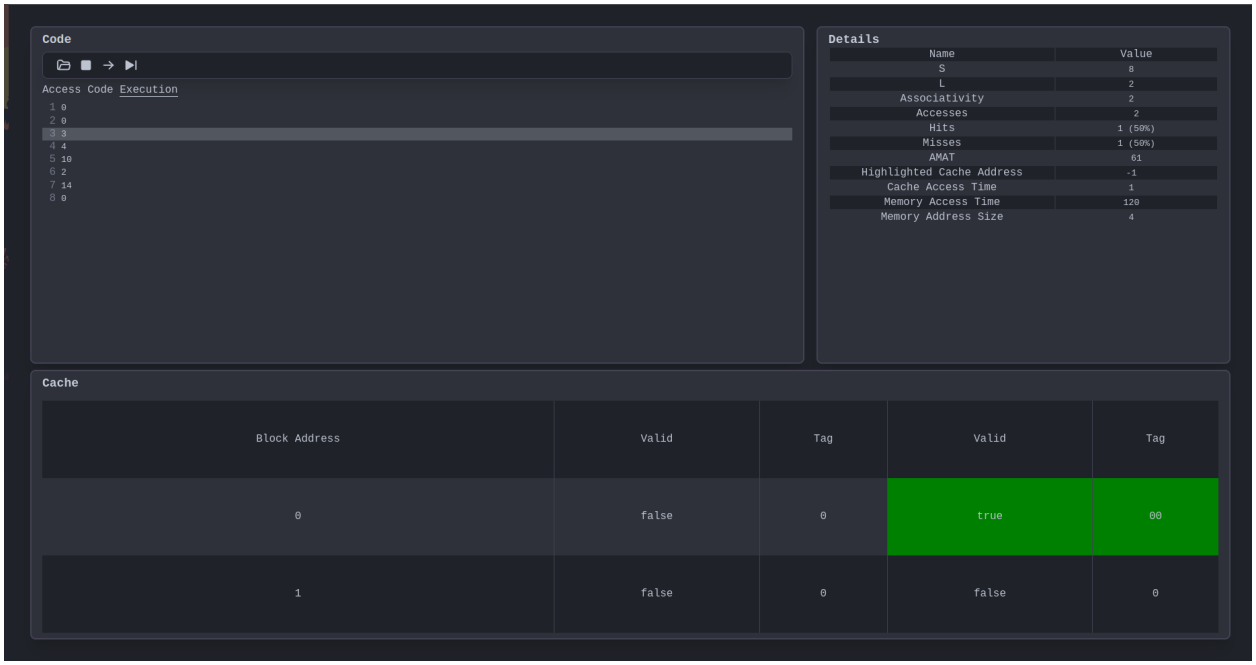


Conversely, the corresponding line is highlighted in green when a hit takes place.



Notice that the data of every line aren’t shown as they are not relevant, and only the valid and tag parts are shown.

Here is an example of how set associative looks like.



Tests:

Our first test is the one shown in the guide. The file is provided as a text file in the tests folder. Notice that the text file contains some extra information at the end about the details section (this part is to be removed from the code section before running the test; it is only there for reference). The details section is to be set to S=8, L=2, Associativity=2, Memory Address Size = 4. The same outcomes would have been reached if the Memory Address Size was anything greater than 4. You may also tweak the access times and the S and L, but will reach different results. Note since our system uses a random placement system when the set is full, you may have different results from the one shown between every run and another. The access sequence is:

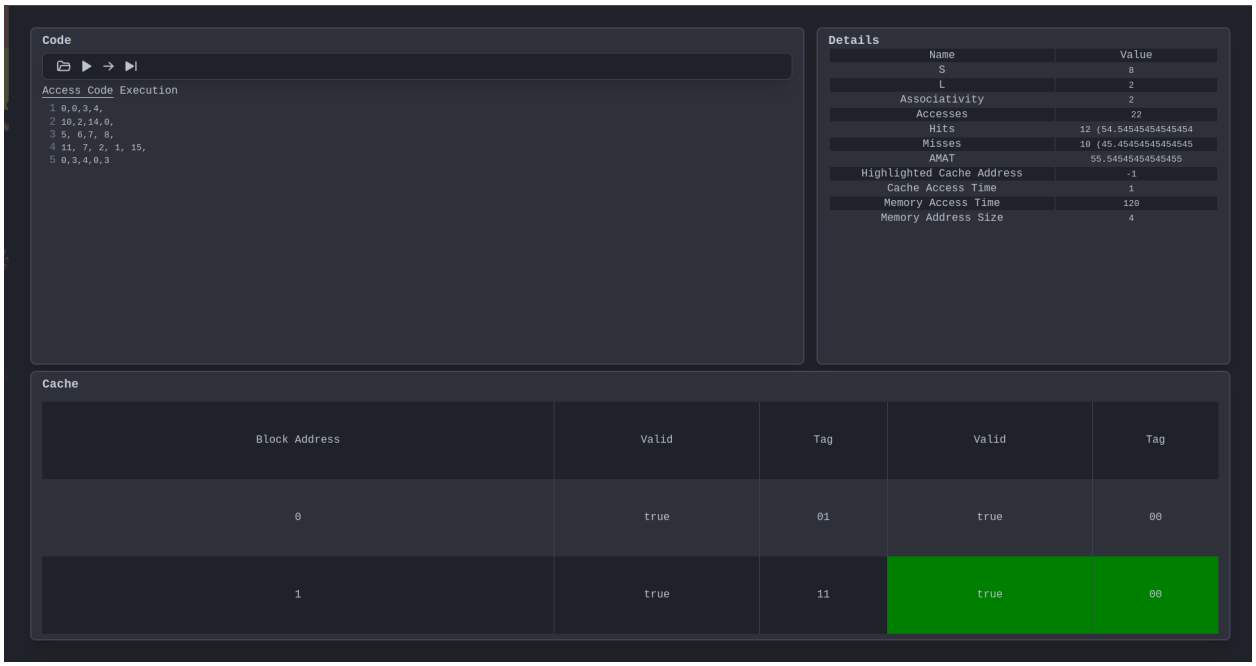
“0,0,3,4,

10,2,14,0,

5, 6,7, 8,

11, 7, 2, 1, 15,

0,3,4,0,3”



For the second sequence:

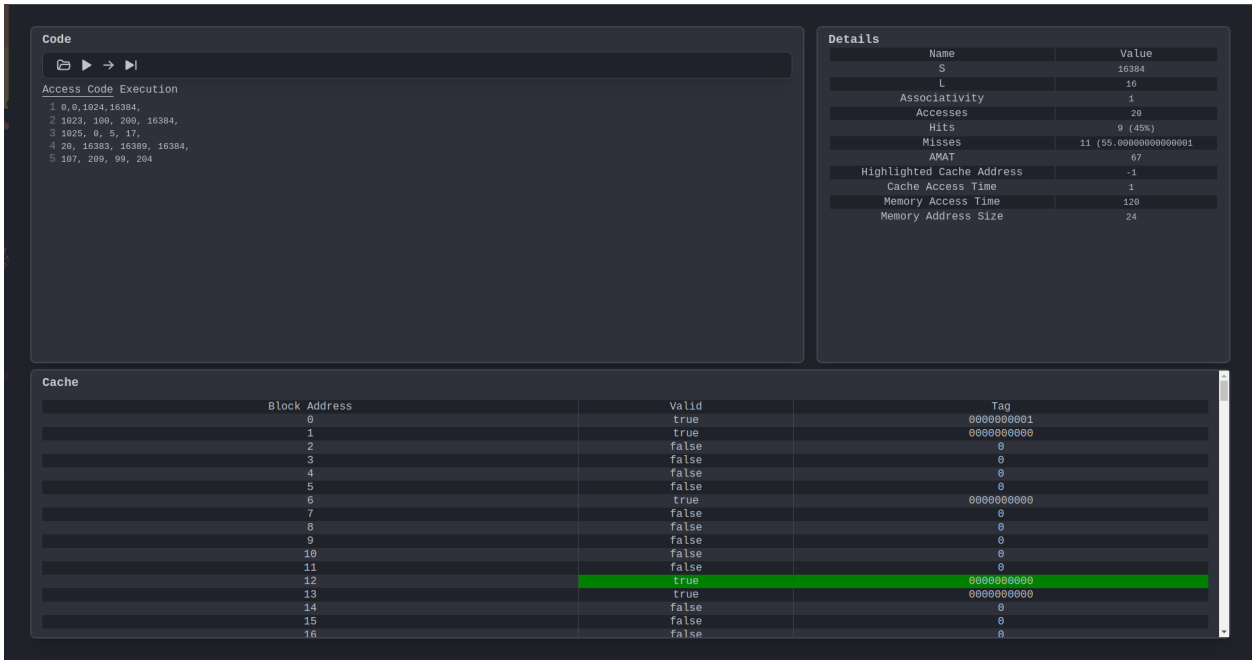
“0,0,1024,16384,

1023, 100, 200, 16384,

1025, 0, 5, 17,

20, 16383, 16389, 16384,

107, 209, 99, 204”


Code
Access Code Execution
1 0,0,1024,16384,
2 1023, 100, 200, 16384,
3 1025, 0, 5, 17,
4 20, 16383, 16389, 16384,
5 107, 209, 99, 204
Details
Name Value
S 16384
L 16
Associativity 1
Accesses 20
Hits 9 (45%)
Misses 11 (55.00000000000001
AMAT 67
Highlighted Cache Address -1
Cache Access Time 1
Memory Access Time 120
Memory Address Size 24
Cache
Block Address Valid Tag
0 true 0000000001
1 true 0000000000
2 false 0
3 false 0
4 false 0
5 false 0
6 true 0000000000
7 false 0
8 false 0
9 false 0
10 false 0
11 false 0
12 true 0000000000
13 true 0000000000
14 false 0
15 false 0
16 false 0