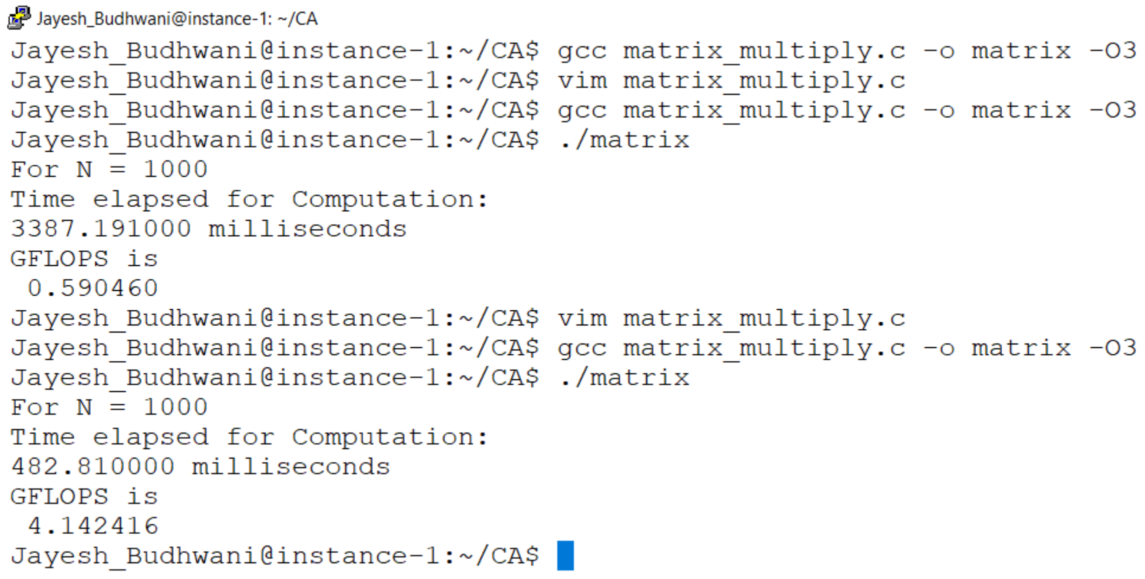
Assignment 4

Problem 2: Matrix Operation and Looping order

A. Looping Order

This part define the working of **matrix multiplication** using different orderings of i, j, and k loops. For this two orderings were used **ijk and ikj**, First ijk was used, and then ikj was used and as we can see that ordering **ijk took 3387.191** milliseconds and ordering **ikj took 482.81** milliseconds.



```
Jayesh_Budhwani@instance-1: ~/CA
Jayesh_Budhwani@instance-1:~/CA$ gcc matrix_multiply.c -o matrix -O3
Jayesh_Budhwani@instance-1:~/CA$ vim matrix_multiply.c
Jayesh_Budhwani@instance-1:~/CA$ gcc matrix_multiply.c -o matrix -O3
Jayesh_Budhwani@instance-1:~/CA$ ./matrix
For N = 1000
Time elapsed for Computation:
3387.191000 milliseconds
GFLOPS is
 0.590460
Jayesh_Budhwani@instance-1:~/CA$ vim matrix_multiply.c
Jayesh_Budhwani@instance-1:~/CA$ gcc matrix_multiply.c -o matrix -O3
Jayesh_Budhwani@instance-1:~/CA$ ./matrix
For N = 1000
Time elapsed for Computation:
482.810000 milliseconds
GFLOPS is
 4.142416
Jayesh_Budhwani@instance-1:~/CA$
```

The answers to the questions are

1. **Ordering 2** is better as it takes less time.
2. By the above experiment, we can say that for a 1000 x 1000 matrix the time required is 3387 milliseconds for ordering 1 and 482 for ordering 2 thus we can say that for execution time as a parameter the ordering 2 is better.
3. Among the two, Ordering 1 performs the worst.

B. Matrix transpose:

In this problem **blocking-based matrix transpose** needs to be done which is **cache-friendly.** Matrix transpose and experiments with different values of ‘n‘ and ‘blocksize’ were conducted. Firstly we need to create the algorithm for cache-friendly matrix transpose which uses blocksize as a parameter.

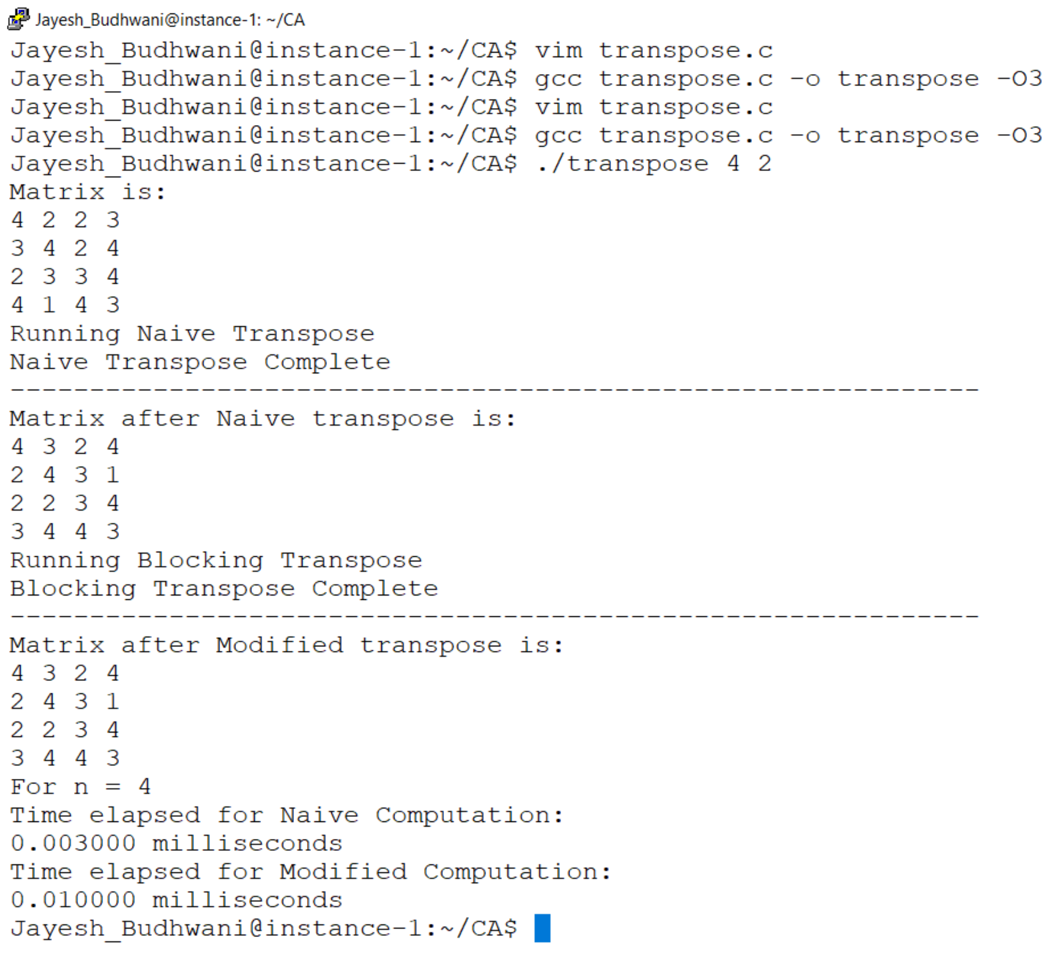
Creating the algorithm: The standard algorithm uses **4 loops**. Two loops for traversing the matrix in block form and the other two for traversing the one block at a time and transposing the block.

Algo:

Input: source (matrix pointer), destination (matrix pointer), n (no. of rows and columns), blocksize

```
1. For x from 0 to blocksize
2. For y from 0 to blocksize
3. For i from 0 to x
4. For j from 0 to y
5. destination[j+i*n] = source[i+j*n]
```

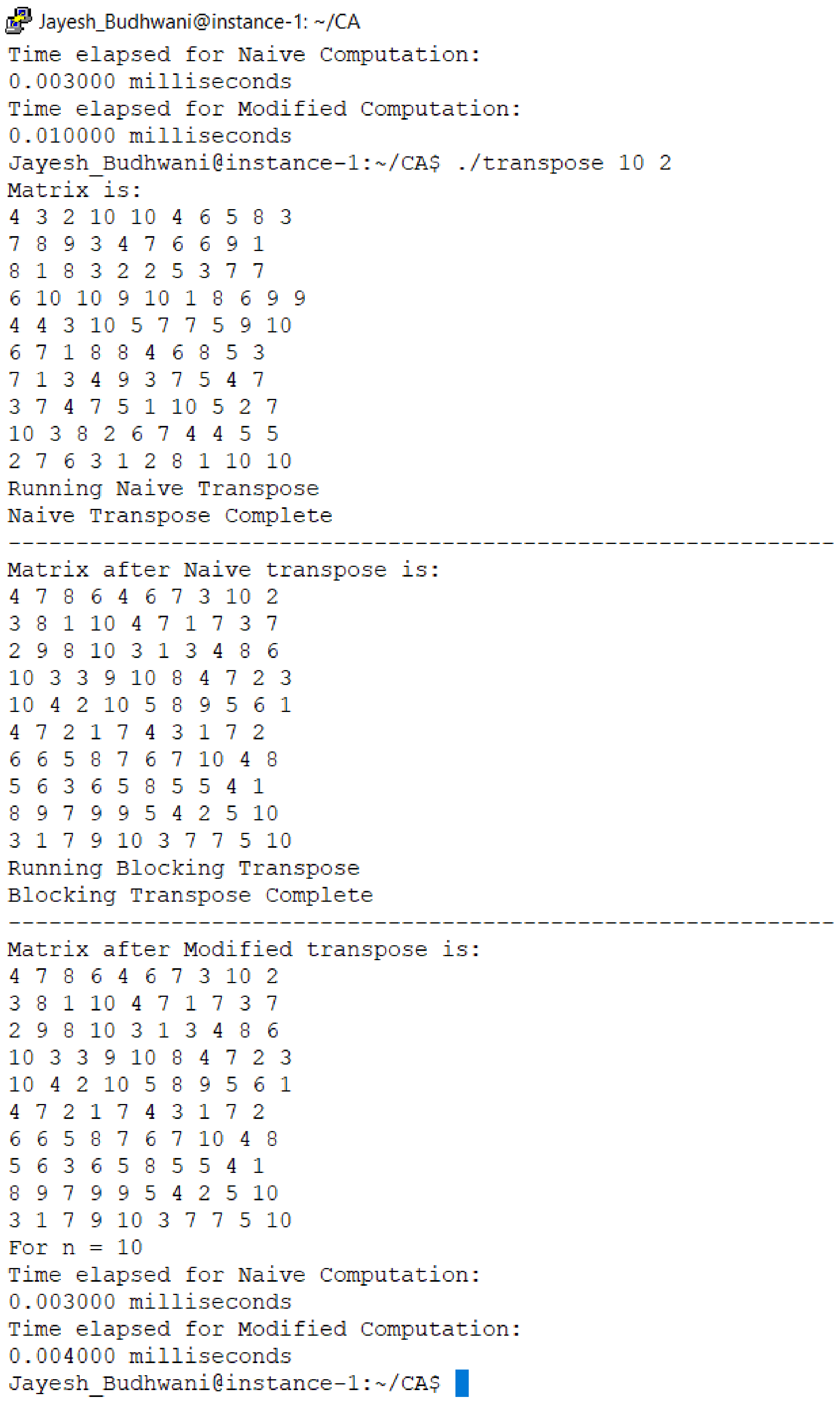
Testing on data: For testing the matrices were printed to check the accuracy of code.
The first test was for **n = 4, blocksize = 2**



Jayesh_Budhwani@instance-1: ~/CA

```
Jayesh_Budhwani@instance-1:~/CA$ vim transpose.c
Jayesh_Budhwani@instance-1:~/CA$ gcc transpose.c -o transpose -O3
Jayesh_Budhwani@instance-1:~/CA$ vim transpose.c
Jayesh_Budhwani@instance-1:~/CA$ gcc transpose.c -o transpose -O3
Jayesh_Budhwani@instance-1:~/CA$ ./transpose 4 2
Matrix is:
4 2 2 3
3 4 2 4
2 3 3 4
4 1 4 3
Running Naive Transpose
Naive Transpose Complete
------------------------------------------------------------
Matrix after Naive transpose is:
4 3 2 4
2 4 3 1
2 2 3 4
3 4 4 3
Running Blocking Transpose
Blocking Transpose Complete
------------------------------------------------------------
Matrix after Modified transpose is:
4 3 2 4
2 4 3 1
2 2 3 4
3 4 4 3
For n = 4
Time elapsed for Naive Computation:
0.003000 milliseconds
Time elapsed for Modified Computation:
0.010000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$
```

The second test was for **n = 10, blocksize = 2**



```
Jayesh_Budhwani@instance-1: ~/CA
Time elapsed for Naive Computation:
0.003000 milliseconds
Time elapsed for Modified Computation:
0.010000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$ ./transpose 10 2
Matrix is:
4 3 2 10 10 4 6 5 8 3
7 8 9 3 4 7 6 6 9 1
8 1 8 3 2 2 5 3 7 7
6 10 10 9 10 1 8 6 9 9
4 4 3 10 5 7 7 5 9 10
6 7 1 8 8 4 6 8 5 3
7 1 3 4 9 3 7 5 4 7
3 7 4 7 5 1 10 5 2 7
10 3 8 2 6 7 4 4 5 5
2 7 6 3 1 2 8 1 10 10
Running Naive Transpose
Naive Transpose Complete
--------------------------------------------------------------
Matrix after Naive transpose is:
4 7 8 6 4 6 7 3 10 2
3 8 1 10 4 7 1 7 3 7
2 9 8 10 3 1 3 4 8 6
10 3 3 9 10 8 4 7 2 3
10 4 2 10 5 8 9 5 6 1
4 7 2 1 7 4 3 1 7 2
6 6 5 8 7 6 7 10 4 8
5 6 3 6 5 8 5 5 4 1
8 9 7 9 9 5 4 2 5 10
3 1 7 9 10 3 7 7 5 10
Running Blocking Transpose
Blocking Transpose Complete
--------------------------------------------------------------
Matrix after Modified transpose is:
4 7 8 6 4 6 7 3 10 2
3 8 1 10 4 7 1 7 3 7
2 9 8 10 3 1 3 4 8 6
10 3 3 9 10 8 4 7 2 3
10 4 2 10 5 8 9 5 6 1
4 7 2 1 7 4 3 1 7 2
6 6 5 8 7 6 7 10 4 8
5 6 3 6 5 8 5 5 4 1
8 9 7 9 9 5 4 2 5 10
3 1 7 9 10 3 7 7 5 10
For n = 10
Time elapsed for Naive Computation:
0.003000 milliseconds
Time elapsed for Modified Computation:
0.004000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$
```

As we can see from the results of the tests, the algorithm is working accurately.

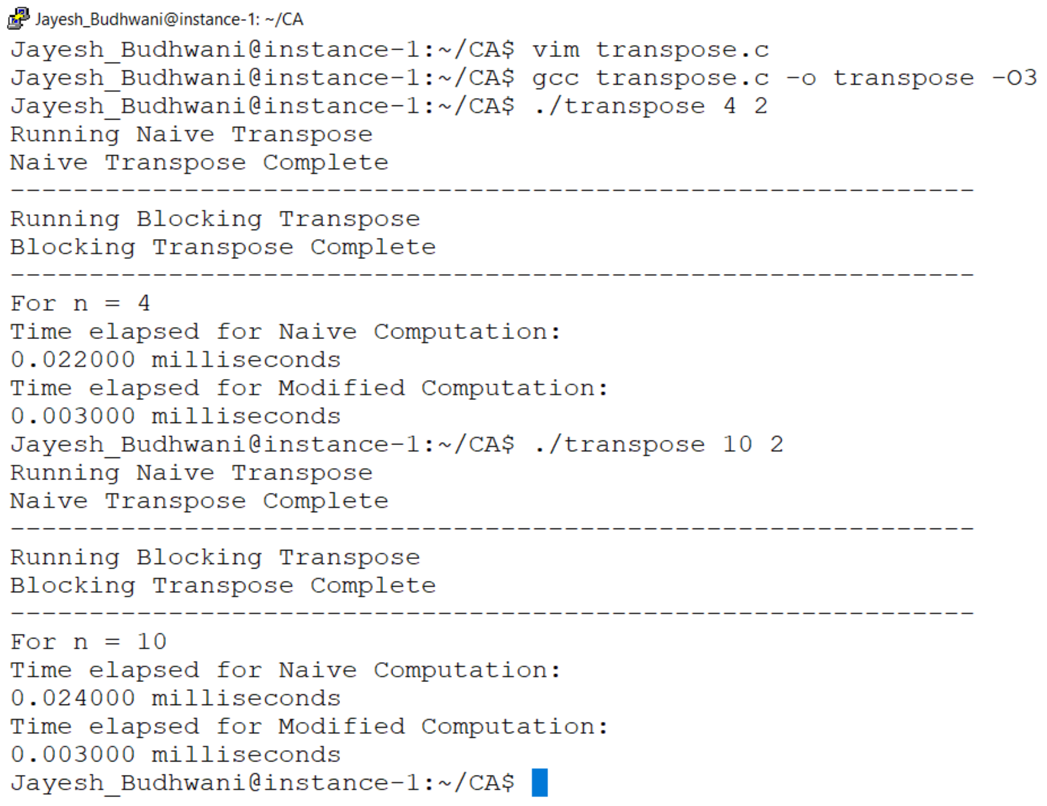
Setting 1: Changing Array/Matrice Sizes
For this setting the **block size is fixed to 20** and the Matrix size is changed, increased gradually, and since the Matrice is of larger order the values of the matrices and the transpose are not printed.
The below table summarises the setting results for different values of n which is the size of matrice.

S.No.	n	blocksize	Time for Naive transpose (ms)	Time for blocking transpose (ms)
1.	100	20	0.089000 milliseconds	0.077000 milliseconds
2.	1000	20	8.954000 milliseconds	8.528000 milliseconds
3	2000	20	57.740000 milliseconds	34.685000 milliseconds
4.	5000	20	437.795000 milliseconds	224.124000 milliseconds
5.	10000	20	2195.960000 milliseconds	933.780000 milliseconds

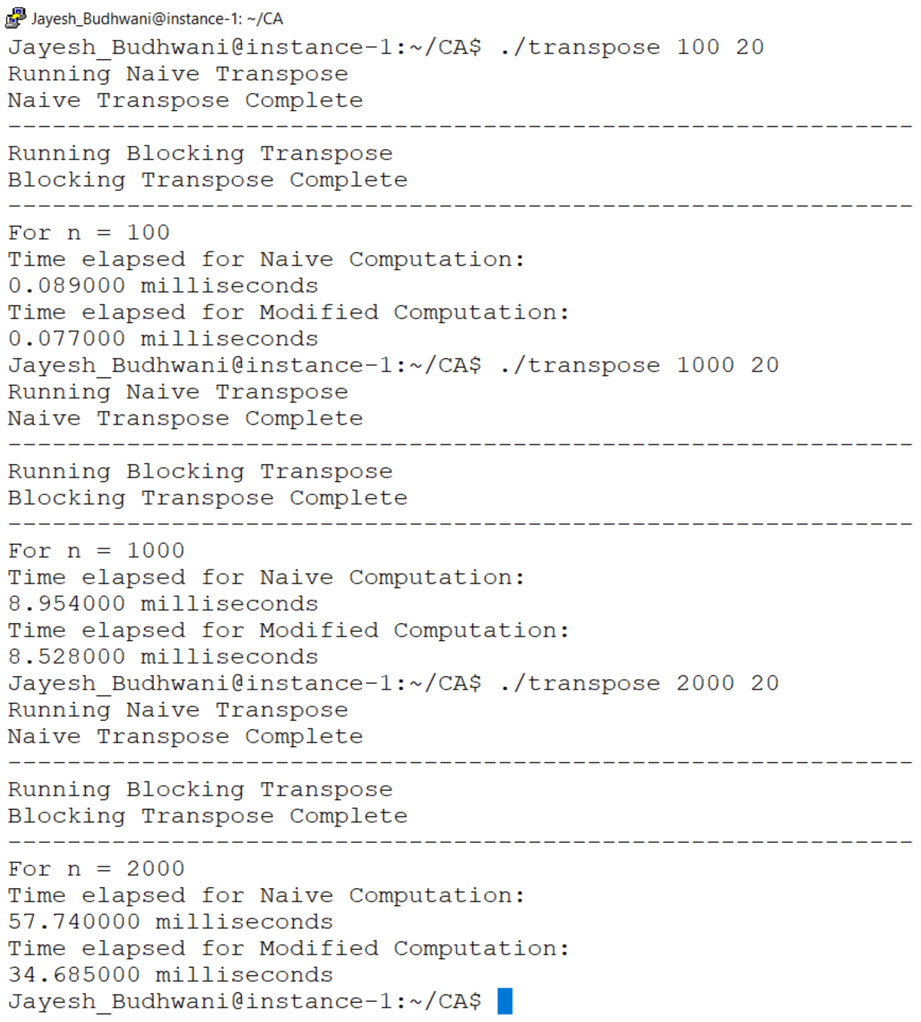
Answer 1:
As we can see from data that at every value of n the value of blocking transpose is better than that of nonblocking. But when we set the value n less than 100 at some instances where the value of n was 10 or less than 10 the blocking transpose was taking more time than nonblocking.

Answer 2:
This is because when the size of the matrice increases and more elements are present in the block then more hit rate is observed, this is because when the size of the matrice increases more blocks are created and when there are more blocks then there are more elements that belong to the blocks and thus there is more hit rate since all the elements can reside in the cache at the same time and thus the execution time reduces. Also when the block size reaches the size of the cache then the highest hit rate is observed and the code is the fastest at that time.

Outputs:

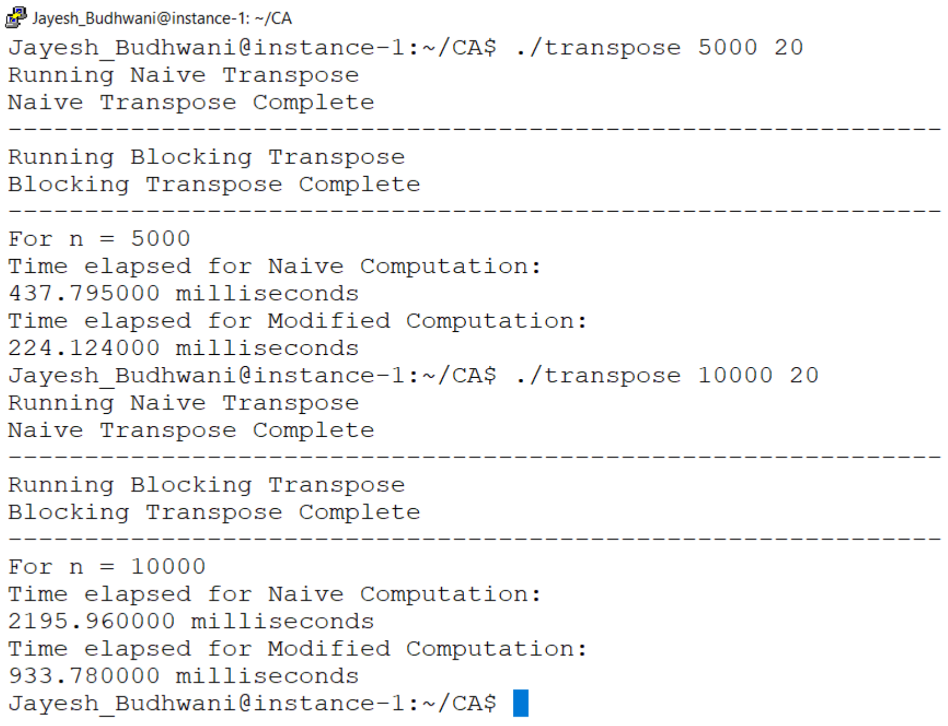


```
Jayesh_Budhwani@instance-1:~/CA$ vim transpose.c
Jayesh_Budhwani@instance-1:~/CA$ gcc transpose.c -o transpose -O3
Jayesh_Budhwani@instance-1:~/CA$ ./transpose 4 2
Running Naive Transpose
Naive Transpose Complete
-------------------------------------------------------------
Running Blocking Transpose
Blocking Transpose Complete
-------------------------------------------------------------
For n = 4
Time elapsed for Naive Computation:
0.022000 milliseconds
Time elapsed for Modified Computation:
0.003000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$ ./transpose 10 2
Running Naive Transpose
Naive Transpose Complete
-------------------------------------------------------------
Running Blocking Transpose
Blocking Transpose Complete
-------------------------------------------------------------
For n = 10
Time elapsed for Naive Computation:
0.024000 milliseconds
Time elapsed for Modified Computation:
0.003000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$
```



Jayesh_Budhwani@instance-1: ~/CA

```
Jayesh_Budhwani@instance-1:~/CA$ ./transpose 100 20
Running Naive Transpose
Naive Transpose Complete
-------------------------------------------------------------
Running Blocking Transpose
Blocking Transpose Complete
-------------------------------------------------------------
For n = 100
Time elapsed for Naive Computation:
0.089000 milliseconds
Time elapsed for Modified Computation:
0.077000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$ ./transpose 1000 20
Running Naive Transpose
Naive Transpose Complete
-------------------------------------------------------------
Running Blocking Transpose
Blocking Transpose Complete
-------------------------------------------------------------
For n = 1000
Time elapsed for Naive Computation:
8.954000 milliseconds
Time elapsed for Modified Computation:
8.528000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$ ./transpose 2000 20
Running Naive Transpose
Naive Transpose Complete
-------------------------------------------------------------
Running Blocking Transpose
Blocking Transpose Complete
-------------------------------------------------------------
For n = 2000
Time elapsed for Naive Computation:
57.740000 milliseconds
Time elapsed for Modified Computation:
34.685000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$
```



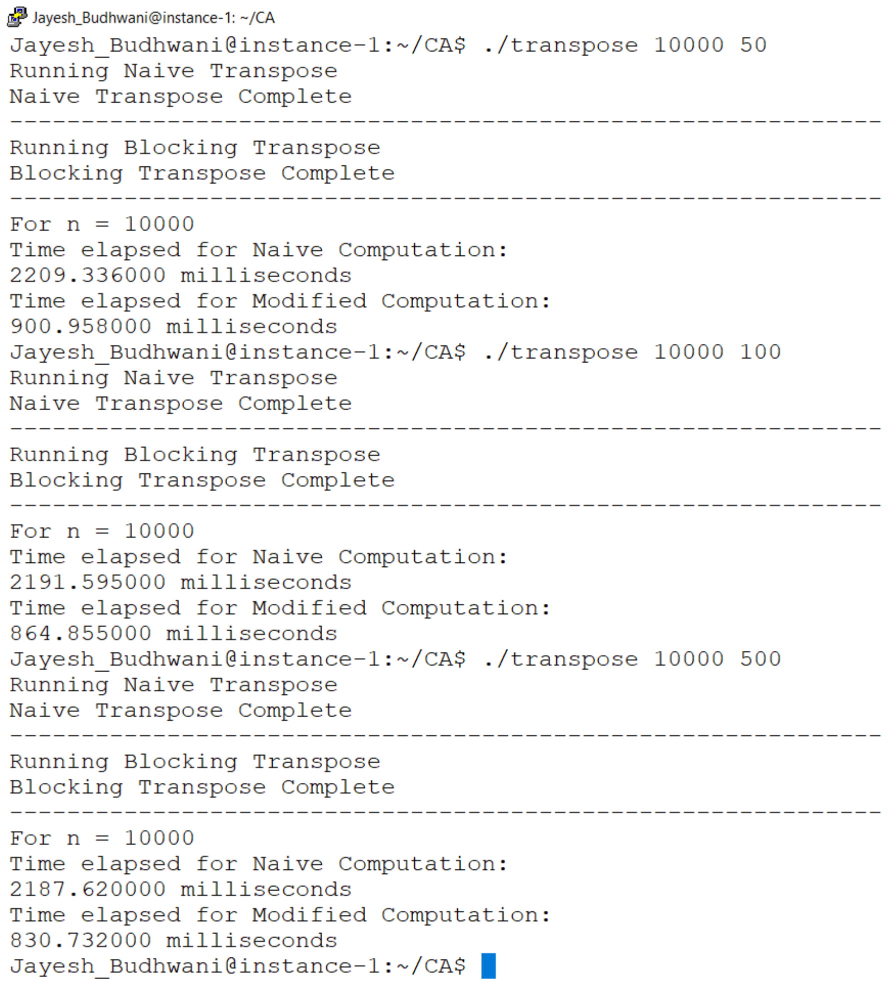
Jayesh_Budhwani@instance-1: ~/CA

```
Jayesh_Budhwani@instance-1:~/CA$ ./transpose 5000 20
Running Naive Transpose
Naive Transpose Complete
-------------------------------------------------------------
Running Blocking Transpose
Blocking Transpose Complete
-------------------------------------------------------------
For n = 5000
Time elapsed for Naive Computation:
437.795000 milliseconds
Time elapsed for Modified Computation:
224.124000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$ ./transpose 10000 20
Running Naive Transpose
Naive Transpose Complete
-------------------------------------------------------------
Running Blocking Transpose
Blocking Transpose Complete
-------------------------------------------------------------
For n = 10000
Time elapsed for Naive Computation:
2195.960000 milliseconds
Time elapsed for Modified Computation:
933.780000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$
```

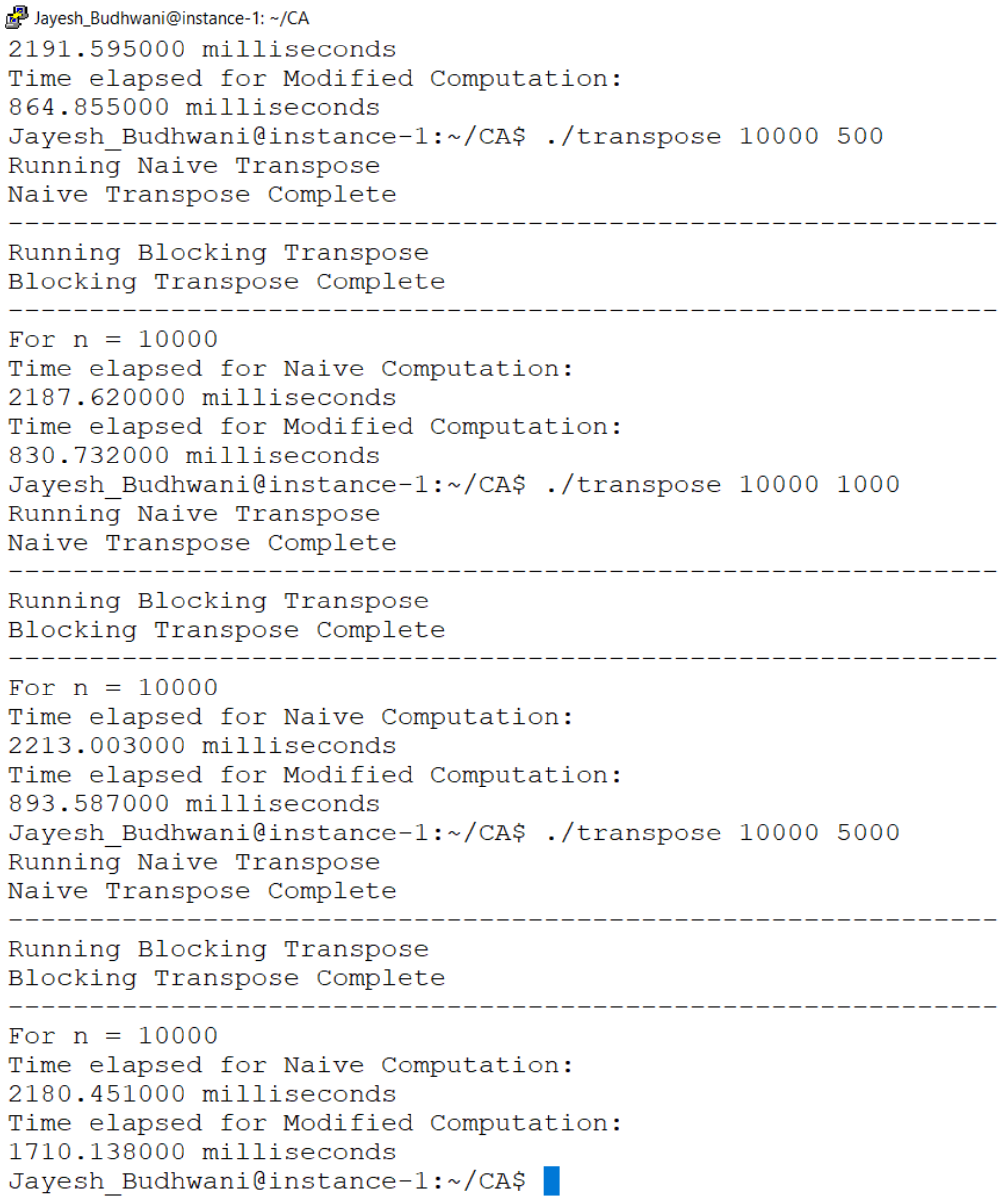
Setting 2: Changing blocksize
For this setting the **Matrice/array size is fixed to 10000** and the blocksize is changed and increased gradually, since the Matrice is of larger order the values of the matrices and the transpose are not printed.
The below table summarises the setting results for different values of block size.

S.No.	n	blocksize	Time for Naive transpose (ms)	Time for blocking transpose (ms)
1.	10000	50	2209.336000 milliseconds	900.958000 milliseconds
2.	10000	100	2191.595000 milliseconds	864.855000 milliseconds
3	10000	500	2187.620000 milliseconds	830.732000 milliseconds
4.	10000	1000	2213.003000 milliseconds	893.587000 milliseconds
5.	10000	5000	2180.451000 milliseconds	1710.138000 milliseconds

Answer 3: In the above setting and summarised results we can see that with increasing blocksize the time for execution in case of blocking version of cache first decreases till blocksize = 500 and after that as the block size increases the execution time also increases. This is because as block size increases more elements are present in the block and when the block size is less than the cache size there would be more locality and thus more hit and hence time decreases means faster execution. But as soon as the block size increase more than the cache size the extra number of elements will not fit in the block and thus there would be more miss hence the execution time increases and is thus slower.
Outputs:



```
Jayesh_Budhwani@instance-1: ~/CA
Jayesh_Budhwani@instance-1:~/CA$ ./transpose 10000 50
Running Naive Transpose
Naive Transpose Complete
-------------------------------------------------------------
Running Blocking Transpose
Blocking Transpose Complete
-------------------------------------------------------------
For n = 10000
Time elapsed for Naive Computation:
2209.336000 milliseconds
Time elapsed for Modified Computation:
900.958000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$ ./transpose 10000 100
Running Naive Transpose
Naive Transpose Complete
-------------------------------------------------------------
Running Blocking Transpose
Blocking Transpose Complete
-------------------------------------------------------------
For n = 10000
Time elapsed for Naive Computation:
2191.595000 milliseconds
Time elapsed for Modified Computation:
864.855000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$ ./transpose 10000 500
Running Naive Transpose
Naive Transpose Complete
-------------------------------------------------------------
Running Blocking Transpose
Blocking Transpose Complete
-------------------------------------------------------------
For n = 10000
Time elapsed for Naive Computation:
2187.620000 milliseconds
Time elapsed for Modified Computation:
830.732000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$
```



```
Jayesh_Budhwani@instance-1: ~/CA
2191.595000 milliseconds
Time elapsed for Modified Computation:
864.855000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$ ./transpose 10000 500
Running Naive Transpose
Naive Transpose Complete
-------------------------------------------------------------
Running Blocking Transpose
Blocking Transpose Complete
-------------------------------------------------------------
For n = 10000
Time elapsed for Naive Computation:
2187.620000 milliseconds
Time elapsed for Modified Computation:
830.732000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$ ./transpose 10000 1000
Running Naive Transpose
Naive Transpose Complete
-------------------------------------------------------------
Running Blocking Transpose
Blocking Transpose Complete
-------------------------------------------------------------
For n = 10000
Time elapsed for Naive Computation:
2213.003000 milliseconds
Time elapsed for Modified Computation:
893.587000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$ ./transpose 10000 5000
Running Naive Transpose
Naive Transpose Complete
-------------------------------------------------------------
Running Blocking Transpose
Blocking Transpose Complete
-------------------------------------------------------------
For n = 10000
Time elapsed for Naive Computation:
2180.451000 milliseconds
Time elapsed for Modified Computation:
1710.138000 milliseconds
Jayesh_Budhwani@instance-1:~/CA$
```

All the files that are matrix_multiply.c, transpose.c, and exercise2.txt are provided in the zip.

Problem 1: RISC-V Code

The Riscv code for the given c code is provided in the cache.s file.

Common Questions/Answers:

1. Cache blocks can be of different sizes like 4 bytes, 8 bytes, 64 bytes, etc, and venus provides us the option to change the block size according to user requirements. Currently, the cache block is 4 bytes.
2. If the step size is ‘n’ that means we can have ‘n’ different array elements that are being accessed, now if the block size is 4 bytes then 1 element can reside in the block and if the block size is 16 bytes 4 elements can reside in a block. This means for 16 there are 4 consecutive accesses where the first one would be a miss and the rest 3 would be a hit, because of the first miss.
3. If the cache block size is ‘n’ bytes and no. of blocks are ‘m’ total space provided by the cache is ‘n*m’ bytes. Like 4 bytes block size and 4 blocks mean 16 bytes of total cache memory.
4. If the number of blocks is ‘n’ then for each value that has the data%n value same we can say those elements map to the same block.
   Ex: let n = 4 so for the data value of 1 and 5, 1%4 and 5%4 gives the same result of 1 and thus they are both mapped to the same block. And thus we can say that for data values in the interval of ‘n‘ that is no. of block the values reside in the same block.
5. Venus provides us the option to change the associativity as per requirements currently there is no associativity that we are using in venus.
6. As we have seen that if the number of blocks is ‘n’ then the block will map the value block%n.

Setting 1:

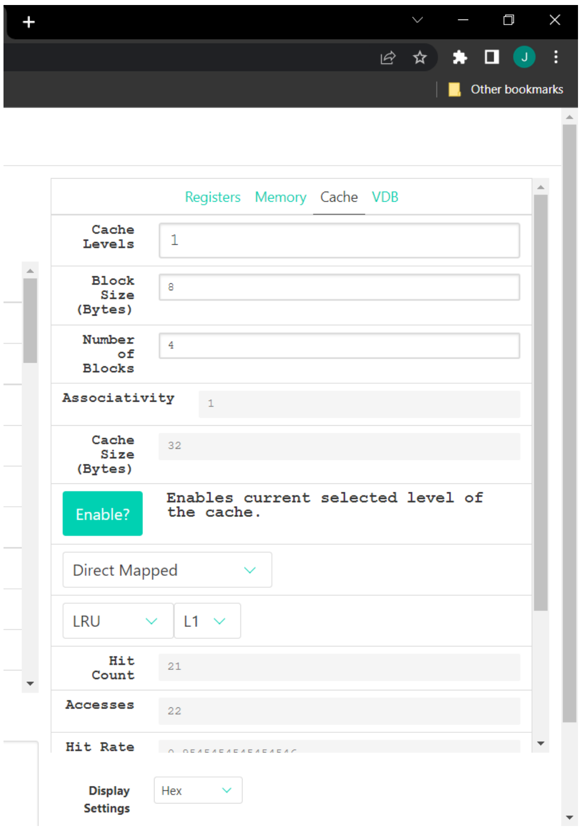
Program Parameters: (set these by initializing the registers in the code)

- Array Size: 128 (bytes)
- Step Size: 8
- Rep Count: 4
- Option : 0

Cache Parameters: (set these in the Cache tab)

- Cache Levels: 1
- Block Size: 8
- Number of Blocks: 4
- Enable?: Should be green
- Placement Policy: Direct Mapped
- Associativity: 1 (Venus won’t let you change this with your placement
- policy, why?)
- Block Replacement Policy: LRU

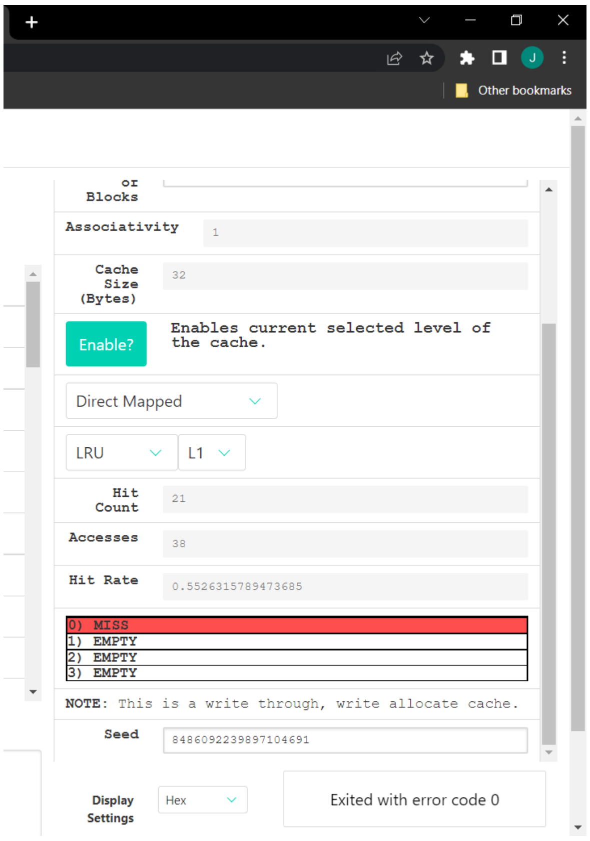
In this setting we are using option 0 which accesses the element 0 of the array again and again so the first time it accesses will be a cache miss rest will be a hit. Array size in bytes = 128 bytes, this means 128/4 elements are present in the array which will be 32 but the step size is 8 so 32/8 = 4 access will be there in one reparation. And the total access will be around 20.
Miss rate will be 1 rest will be hit.
The below picture shows the output of the above setting.


Registers Memory Cache VDB

Cache Levels	1
Block Size (Bytes)	8
Number of Blocks	4
Associativity	1
Cache Size (Bytes)	32
Enable?	Enables current selected level of the cache.
Direct Mapped	
LRU	L1
Hit Count	21
Accesses	22
Hit Rate	0.9545454545454546

Display Settings: Hex

The above setting with option = 1 is shown below.



of Blocks	
Associativity	1
Cache Size (Bytes)	32
Enable?	Enables current selected level of the cache.
Direct Mapped	
LRU	L1
Hit Count	21
Accesses	38
Hit Rate	0.5526315789473685

```
0) MISS
1) EMPTY
2) EMPTY
3) EMPTY
```

NOTE: This is a write through, write allocate cache.

Seed: 8486092239897104691

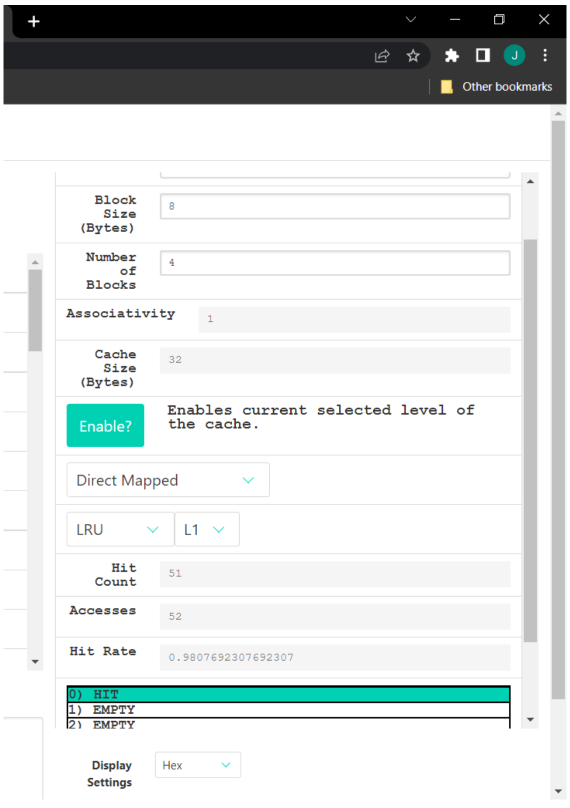
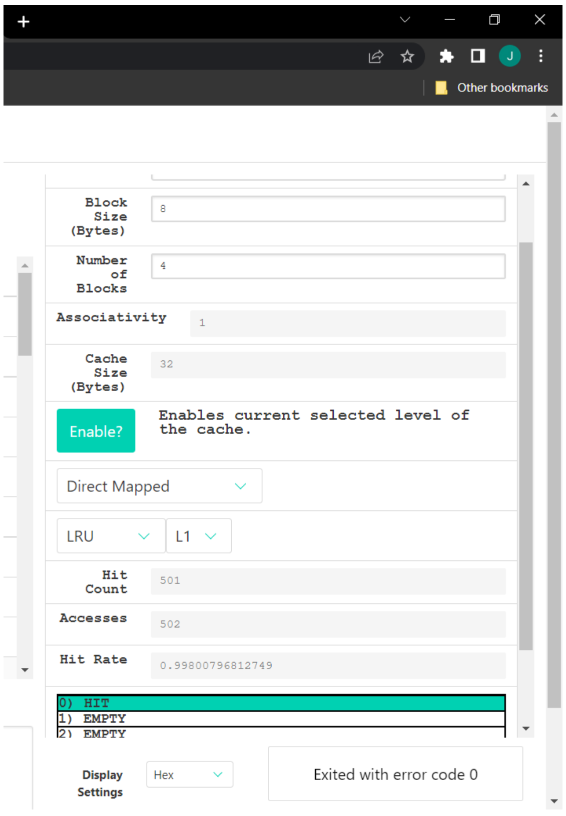
Display Settings: Hex

Exited with error code 0

Answer 1:
When we access the data it gets stored in the cache at the block-**data1%n** where n is no. of blocks and if the size of the block is more we can say that more data can be stored in that block simultaneously now the access of the first element will be a miss but the subsequent access of elements in that block would be a hit.
Now when another data data2 comes and it also has the same value as previous data that is **data1%n = data2%n** then this data will overwrite the previous data1 and thus we can see that data values can be remaining in the same block or can be replaced with another data which can lead to miss or hit.
In the current scenario when the option is 0 we are only accessing the first element of the array and thus the first access would be the only miss and the rest of the access would be a hit.

Answer 2:
Step-Size and Cache Size since the cache size is 32 bytes and the step size is 8 that means 8*4 = 32 bytes can be stored at a time.

Answer 3:
In the current scenario since the option is 0 there is a hit rate of 0.95. Now if reparations increase the hit rate will also increase. But increasing block size or blocks or cache size with the same reparations does not change the hit rate.

Reparations = 10	Repetitions = 100
	

Reparations = 10:

Block Size (Bytes): 8
Number of Blocks: 4
Associativity: 1
Cache Size (Bytes): 32
Enable? Enables current selected level of the cache.
Direct Mapped
LRU
L1
Hit Count: 51
Accesses: 52
Hit Rate: 0.9807692307692307

```
0) HIT
1) EMPTY
2) EMPTY
```

Display Settings: Hex

Repetitions = 100:

Block Size (Bytes): 8
Number of Blocks: 4
Associativity: 1
Cache Size (Bytes): 32
Enable? Enables current selected level of the cache.
Direct Mapped
LRU
L1
Hit Count: 501
Accesses: 502
Hit Rate: 0.99800796812749

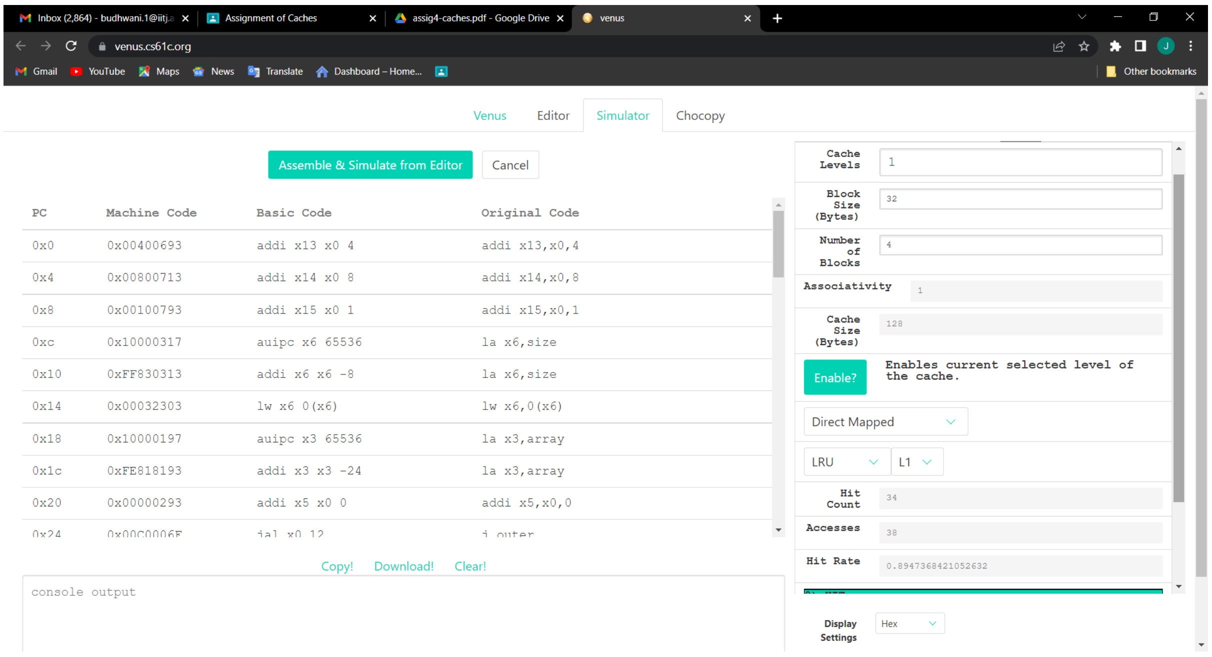
```
0) HIT
1) EMPTY
2) EMPTY
```

Display Settings: Hex

Exited with error code 0

But if the option is 1 then we can say that **step size and cache block size** are the parameters this is because if the step size is ‘n’ and cache block size is ‘m’ bytes then m/4 (1 element = 4 bytes) elements can reside in cache now if **n = m/4** we can say all elements of a step can reside in a cache block.
Ex: step size = 8, cache block size = 32, no. of elements that can be put in a cache block = 32/4 = 8 which is equal to step size. The hit rate will be 86%.

Another solution can be **Array Size and cache size**. When both of these are equal highest hit rate is observed.
As shown below.



Venus Editor Simulator Chocopy

Assemble & Simulate from Editor Cancel

PC	Machine Code	Basic Code	Original Code
0x0	0x00400693	addi x13 x0 4	addi x13,x0,4
0x4	0x00800713	addi x14 x0 8	addi x14,x0,8
0x8	0x00100793	addi x15 x0 1	addi x15,x0,1
0xc	0x10000317	auipc x6 65536	la x6,size
0x10	0xFF830313	addi x6 x6 -8	la x6,size
0x14	0x00032303	lw x6 0(x6)	lw x6,0(x6)
0x18	0x10000197	auipc x3 65536	la x3,array
0x1c	0xFE818193	addi x3 x3 -24	la x3,array
0x20	0x00000293	addi x5 x0 0	addi x5,x0,0
0x24	0x00C0006F	jal x0 12	j outer

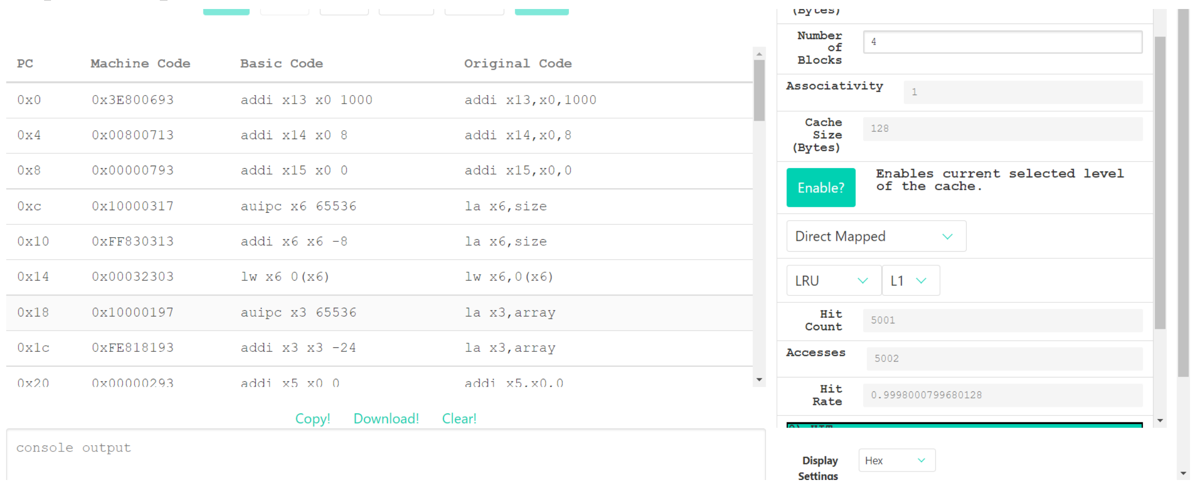
Copy! Download! Clear!

```
console output
```

Cache Levels	1
Block Size (Bytes)	32
Number of Blocks	4
Associativity	1
Cache Size (Bytes)	128
Enable?	Enables current selected level of the cache.
Direct Mapped	
LRU	L1
Hit Count	34
Accesses	38
Hit Rate	0.8947368421052632
Display Settings	Hex

Answer 4:

For option 0 **rep count = 1000.**



PC	Machine Code	Basic Code	Original Code
0x0	0x3E800693	addi x13 x0 1000	addi x13,x0,1000
0x4	0x00800713	addi x14 x0 8	addi x14,x0,8
0x8	0x00000793	addi x15 x0 0	addi x15,x0,0
0xc	0x10000317	auipc x6 65536	la x6,size
0x10	0xFF830313	addi x6 x6 -8	la x6,size
0x14	0x00032303	lw x6 0(x6)	lw x6,0(x6)
0x18	0x10000197	auipc x3 65536	la x3,array
0x1c	0xFE818193	addi x3 x3 -24	la x3,array
0x20	0x00000293	addi x5 x0 0	addi x5,x0,0

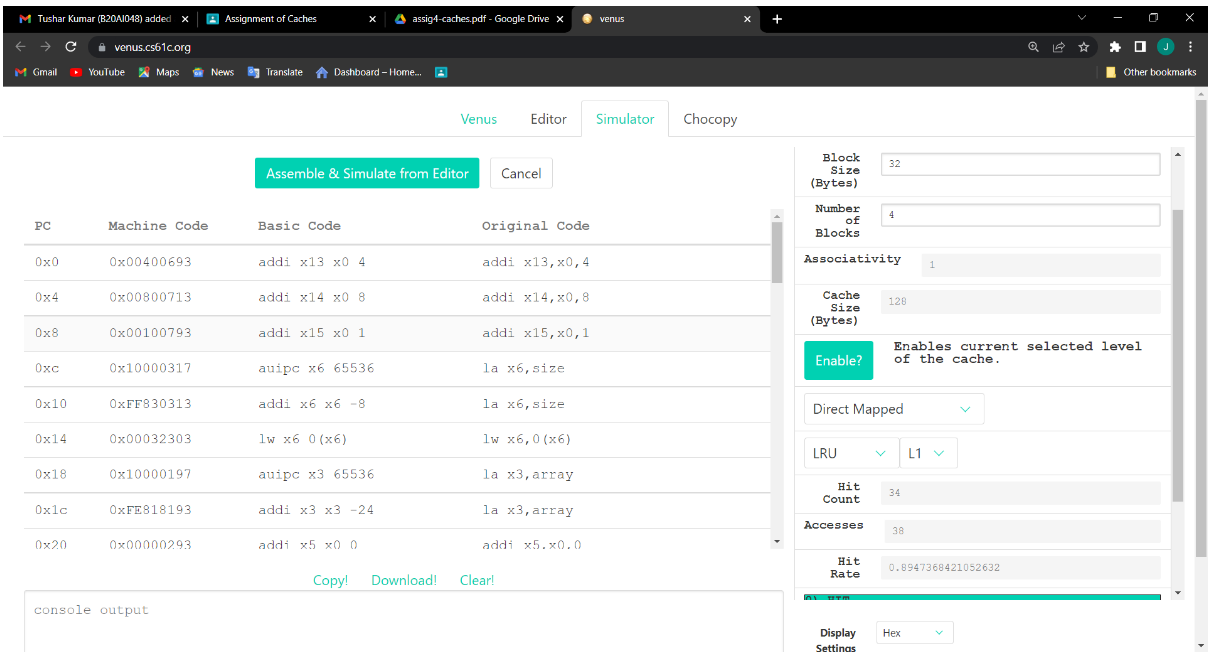
Copy! Download! Clear!

```
console output
```

Number of Blocks	4
Associativity	1
Cache Size (Bytes)	128
Enable?	Enables current selected level of the cache.
Direct Mapped	
LRU	L1
Hit Count	5001
Accesses	5002
Hit Rate	0.9998000799680128
Display Settings	Hex

For option 1 **cache block size to 32 bytes** increasing the rep counts.

We can modify the cache block size to get the highest value of the hit rate. This is because when the cache size is more than 8 bytes we can accommodate more elements and thus more hit rate. But in the current setting, we are using option = 0, which always uses the first element of the array so there will always be a hit rate of greater than 0.98. Further increase in hit rate can be observed using higher rep counts as in question 3.



PC	Machine Code	Basic Code	Original Code
0x0	0x00400693	addi x13 x0 4	addi x13,x0,4
0x4	0x00800713	addi x14 x0 8	addi x14,x0,8
0x8	0x00100793	addi x15 x0 1	addi x15,x0,1
0xc	0x10000317	auipc x6 65536	la x6,size
0x10	0xFF830313	addi x6 x6 -8	la x6,size
0x14	0x00032303	lw x6 0(x6)	lw x6,0(x6)
0x18	0x10000197	auipc x3 65536	la x3,array
0x1c	0xFE818193	addi x3 x3 -24	la x3,array
0x20	0x00000293	addi x5 x0 0	addi x5,x0,0

```
console output
```

Block Size (Bytes)	32
Number of Blocks	4
Associativity	1
Cache Size (Bytes)	128
Enable?	Enables current selected level of the cache.
Placement	Direct Mapped
Replacement / Level	LRU / L1
Hit Count	34
Accesses	38
Hit Rate	0.8947368421052632
Display Settings	Hex

Setting 2:

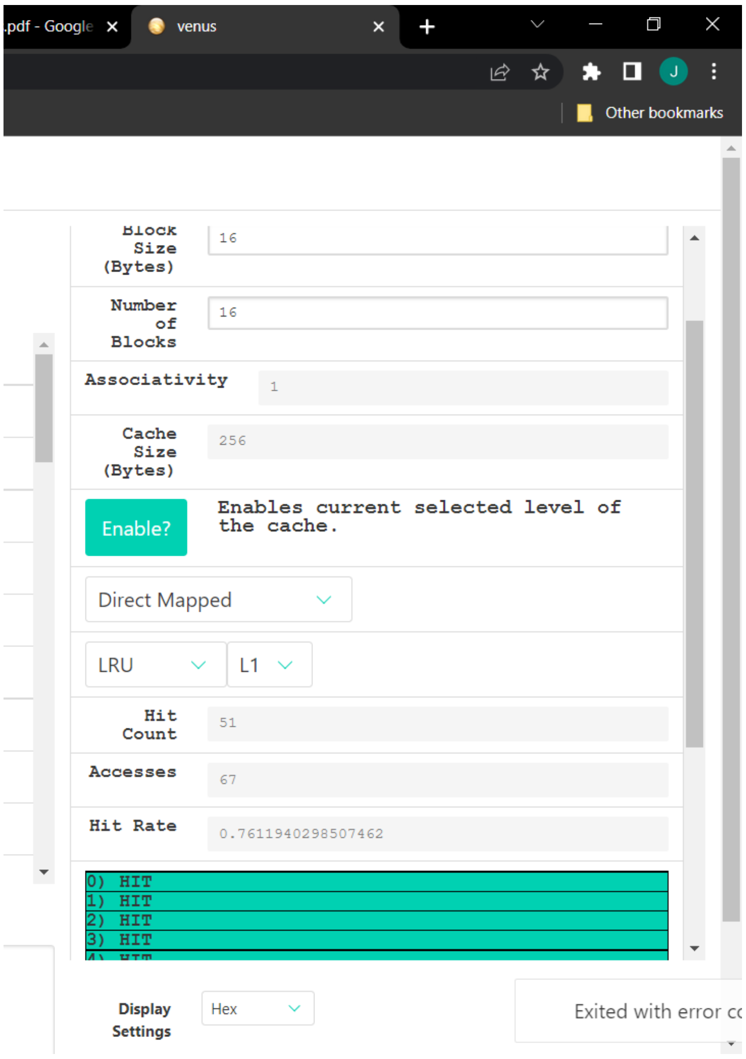
Program Parameters: (set these by initializing the registers in the code)

- Array Size: 256 (bytes)
- Step Size: 2
- Rep Count: 1
- Option: 1

Cache Parameters: (set these in the Cache tab)

- Cache Levels: 1
- Block Size: 16
- Number of Blocks: 16
- Enable?: Should be green
- Placement Policy: N-Way Set Associative
- Associativity: 4
- Block Replacement Policy: LRU

For the given setting with direct-mapped cache we observe a hit rate of 0.75.



Block Size (Bytes) 16

Number of Blocks 16

Associativity 1

Cache Size (Bytes) 256

Enable? Enables current selected level of the cache.

Direct Mapped

LRU L1

Hit Count 51

Accesses 67

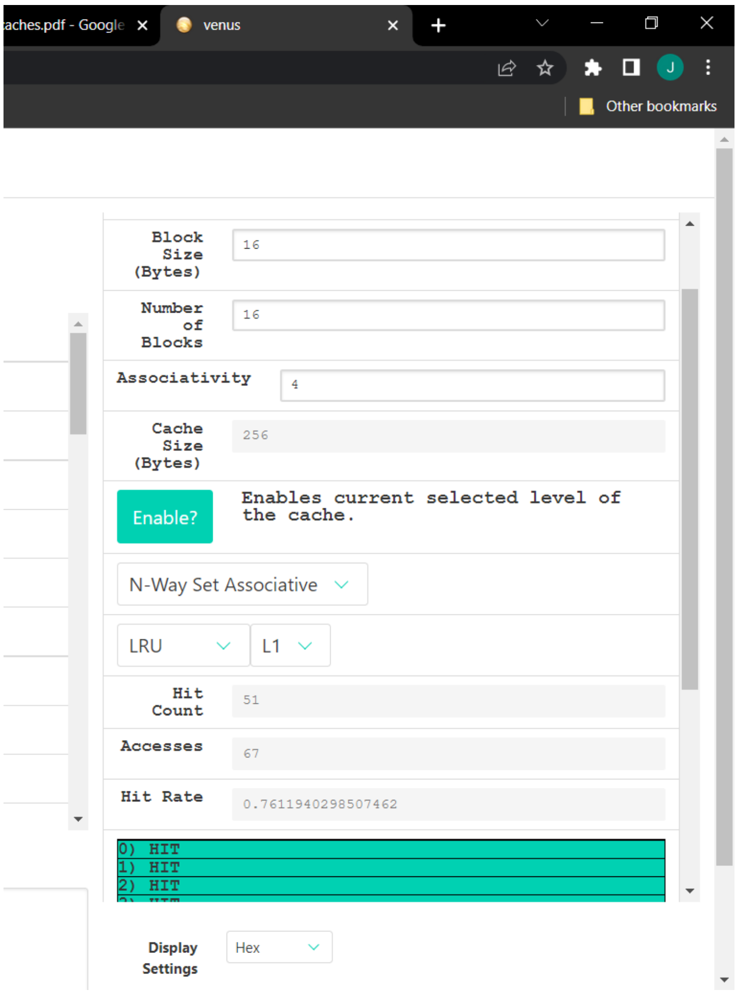
Hit Rate 0.7611940298507462

```
0) HIT
1) HIT
2) HIT
3) HIT
```

Display Settings Hex

Exited with error c

With a 4-way set-associative cache we will observe a hit rate of 0.75.



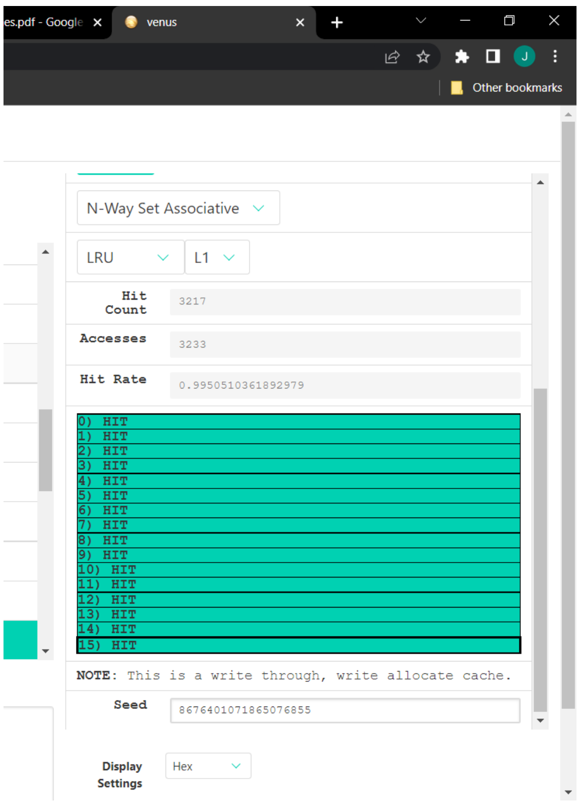
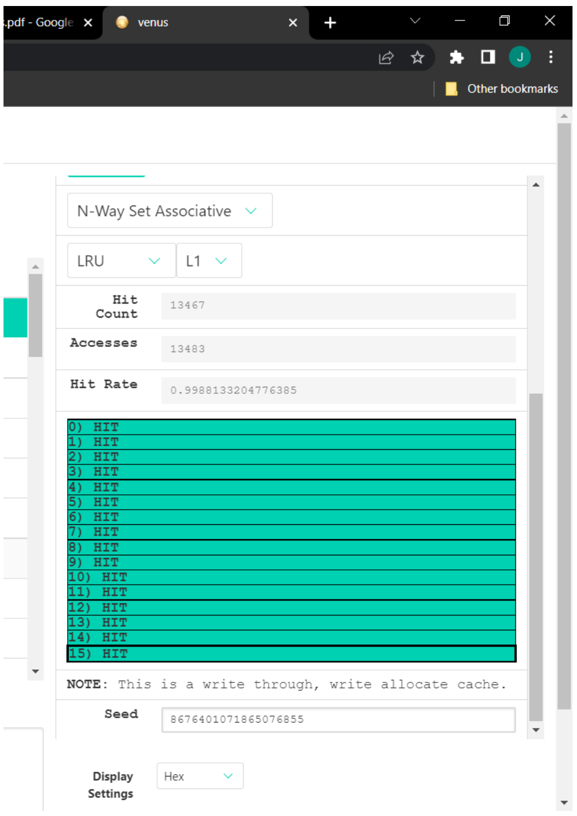
Block Size (Bytes): 16
Number of Blocks: 16
Associativity: 4
Cache Size (Bytes): 256
Enables current selected level of the cache.
N-Way Set Associative
LRU
L1
Hit Count: 51
Accesses: 67
Hit Rate: 0.7611940298507462

```
0) HIT
1) HIT
2) HIT
```

Answer 1:
For each iteration of the inner loop there are, 67 memory access as the size of the array is 256/4 = 64 since the size is given in bytes and each element is 4 bytes.

Answer 2:
MHHH-MHHH-MHHH is the pattern and the smallest pattern is MHHH.

Answer 3:
In both the below cases with rep count 100 and 1000 the hit rate is same which is around 0.98.

rep count = 100	rep count = 1000
	

rep count = 100:

```
N-Way Set Associative
LRU    L1
Hit Count   3217
Accesses    3233
Hit Rate    0.9950510361892979
0) HIT
1) HIT
2) HIT
3) HIT
4) HIT
5) HIT
6) HIT
7) HIT
8) HIT
9) HIT
10) HIT
11) HIT
12) HIT
13) HIT
14) HIT
15) HIT
NOTE: This is a write through, write allocate cache.
Seed   8676401071865076855
Display Settings   Hex
```

rep count = 1000:

```
N-Way Set Associative
LRU    L1
Hit Count   13467
Accesses    13483
Hit Rate    0.9988133204776385
0) HIT
1) HIT
2) HIT
3) HIT
4) HIT
5) HIT
6) HIT
7) HIT
8) HIT
9) HIT
10) HIT
11) HIT
12) HIT
13) HIT
14) HIT
15) HIT
NOTE: This is a write through, write allocate cache.
Seed   8676401071865076855
Display Settings   Hex
```

Setting 3:
Program Parameters: (set these by initializing the registers in the code)

- Array Size: 128 (bytes)
- Step Size: 1
- Rep Count: 1
- Option : 0

Cache Parameters: (set these in the Cache tab)

- Cache Levels: 2

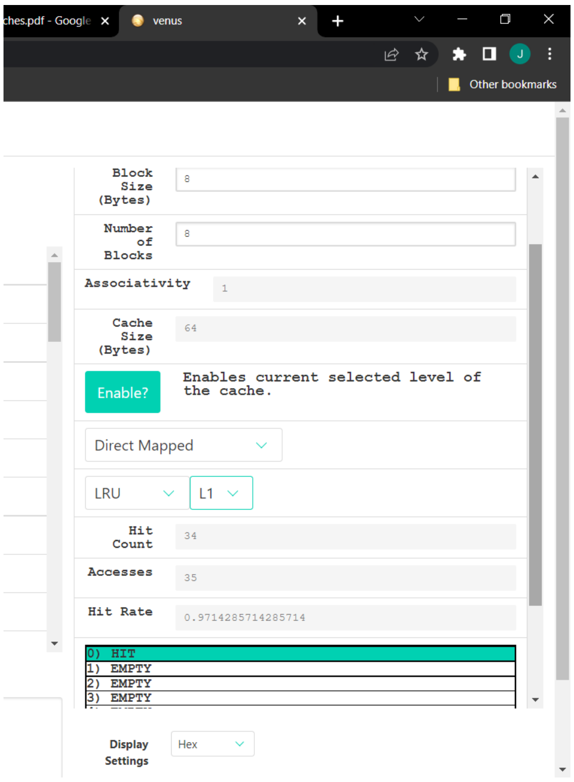
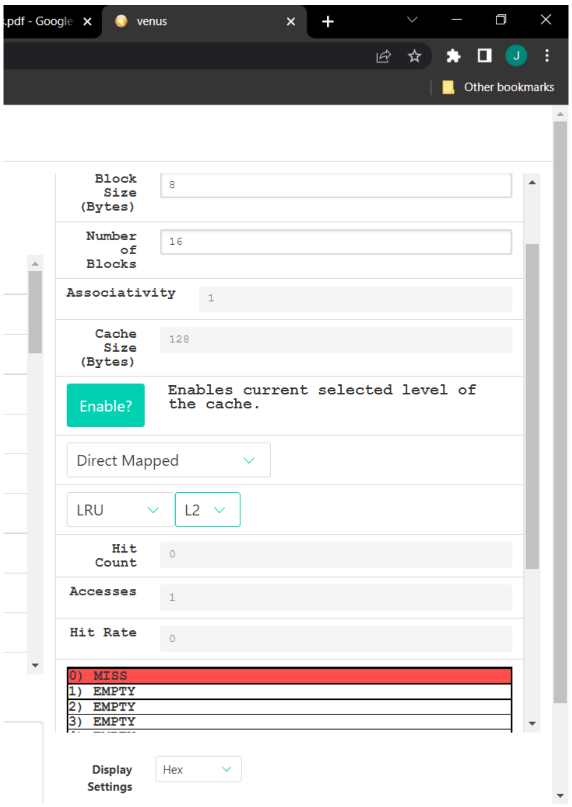
NOTE: Make sure the following parameters are for the L1 cache! (Select L1 in the dropdown right next to the replacement policy)

- Block Size: 8
- Number of Blocks: 8
- Enable?: Should be green
- Placement Policy: Direct Mapped
- Associativity: 1
- Block Replacement Policy: LRU

NOTE: Make sure the following parameters are for the L2 cache! (Select L2 in the dropdown right next to the replacement policy)

- Block Size: 8
- Number of Blocks: 16
- Enable?: Should be green
- Placement Policy: Direct Mapped
- Associativity: 1
- Block Replacement Policy: LRU

For the above setting the output are

L1:	L2:
	

L1:

Block Size (Bytes) 8
Number of Blocks 8
Associativity 1
Cache Size (Bytes) 64
Enable? Enables current selected level of the cache.
Direct Mapped
LRU L1
Hit Count 34
Accesses 35
Hit Rate 0.9714285714285714

```
0) HIT
1) EMPTY
2) EMPTY
3) EMPTY
```

Display Settings Hex

L2:

Block Size (Bytes) 8
Number of Blocks 16
Associativity 1
Cache Size (Bytes) 128
Enable? Enables current selected level of the cache.
Direct Mapped
LRU L2
Hit Count 0
Accesses 1
Hit Rate 0

```
0) MISS
1) EMPTY
2) EMPTY
3) EMPTY
```

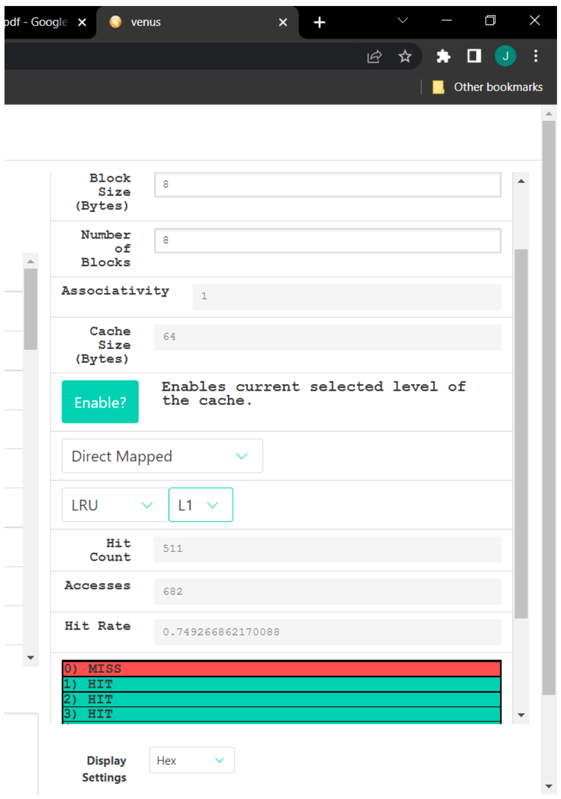
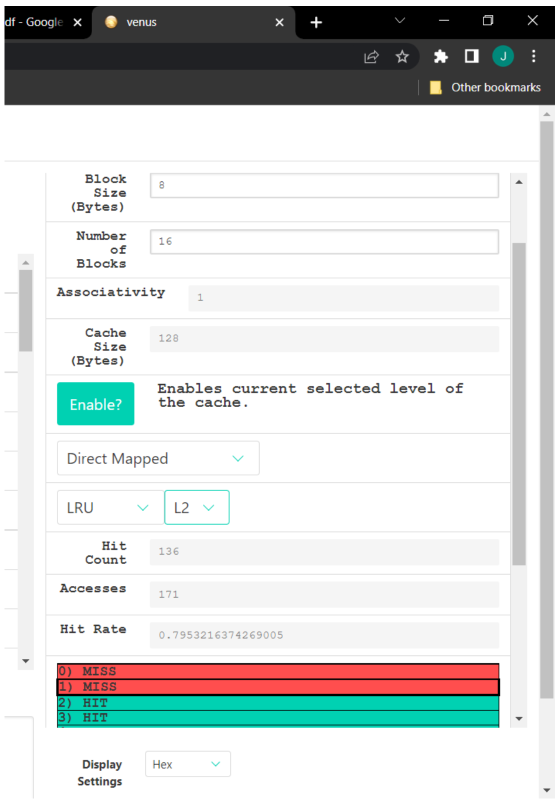
Display Settings Hex

Answer 1:
L1 0.99
L2 0.00
Overall 0.99

Answer 2:
L1 Access-131 hit-130 miss-1

Answer 3:
L2 Access-1 hit-0 miss-1

Answer 4:
For option = 0 there is not any parameter whose change in value will increase the hit rate in L2. Hence when the option changes to 1 we can increase the hit rate of L2 by changing the rep count.
Rep count = 10 and option = 1 (But in this case hit rate of L1 also changes)

L1:	L2:
 Block Size (Bytes) 8 Number of Blocks 8 Associativity 1 Cache Size (Bytes) 64 Enable? Enables current selected level of the cache. Direct Mapped LRU L1 Hit Count 511 Accesses 682 Hit Rate 0.749266862170088 0) MISS 1) HIT 2) HIT 3) HIT Display Settings Hex	 Block Size (Bytes) 8 Number of Blocks 16 Associativity 1 Cache Size (Bytes) 128 Enable? Enables current selected level of the cache. Direct Mapped LRU L2 Hit Count 136 Accesses 171 Hit Rate 0.7953216374269005 0) MISS 1) MISS 2) HIT 3) HIT Display Settings Hex

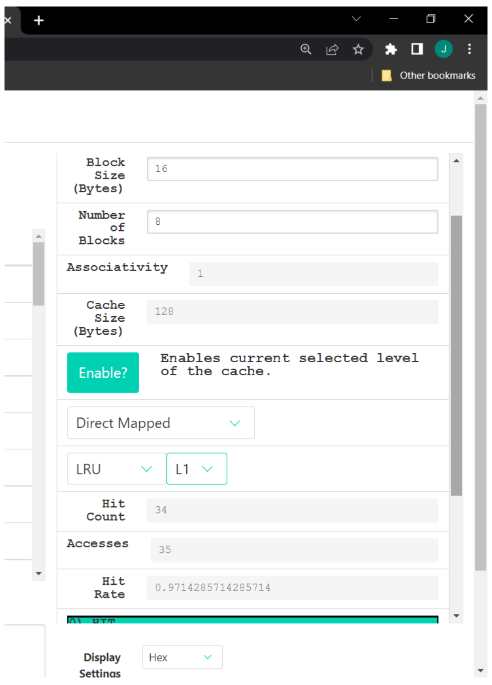
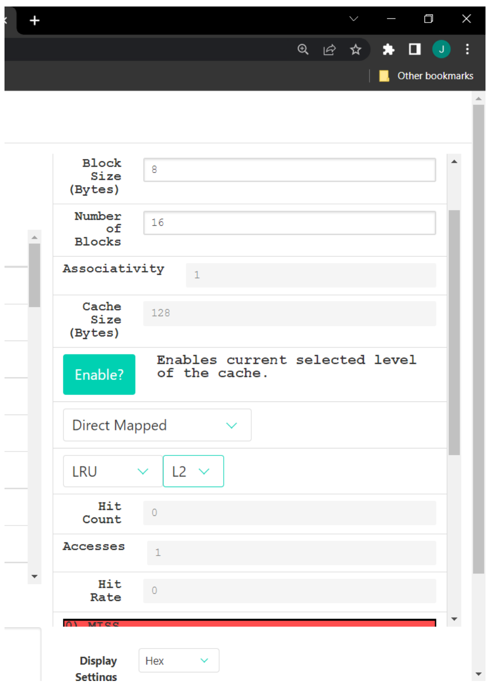
Answer 5:

= = = = (This is when option = 0)

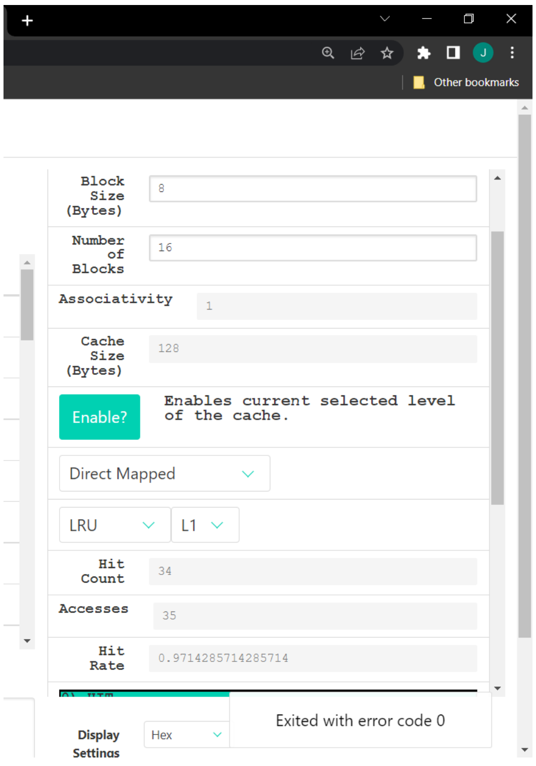
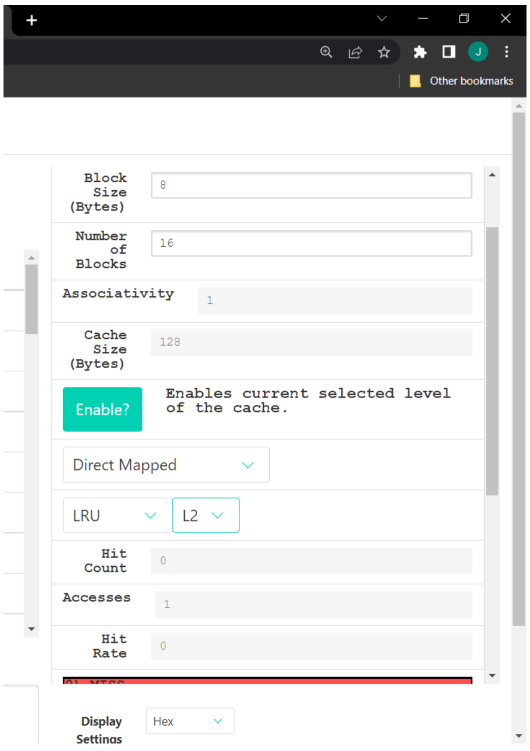
= - + - (This is observed for option = 1 where in first case block were increased to 16 and in second case block size was increased to 16)

For option = 0

Case 1: block size in L1 increased to 16

L1:	L2:
 Block Size (Bytes) 16 Number of Blocks 8 Associativity 1 Cache Size (Bytes) 128 Enable? Enables current selected level of the cache. Direct Mapped LRU L1 Hit Count 34 Accesses 35 Hit Rate 0.9714285714285714 0) HIT Display Settings Hex	 Block Size (Bytes) 8 Number of Blocks 16 Associativity 1 Cache Size (Bytes) 128 Enable? Enables current selected level of the cache. Direct Mapped LRU L2 Hit Count 0 Accesses 1 Hit Rate 0 0) MISS Display Settings Hex

Case 2: blocks in L1 increased to 16

L1:	L2:
	

L1:

Block Size (Bytes): 8
Number of Blocks: 16
Associativity: 1
Cache Size (Bytes): 128
Enable? Enables current selected level of the cache.
Direct Mapped
LRU | L1
Hit Count: 34
Accesses: 35
Hit Rate: 0.9714285714285714

```
0) HIT
```

Display Settings: Hex
Exited with error code 0

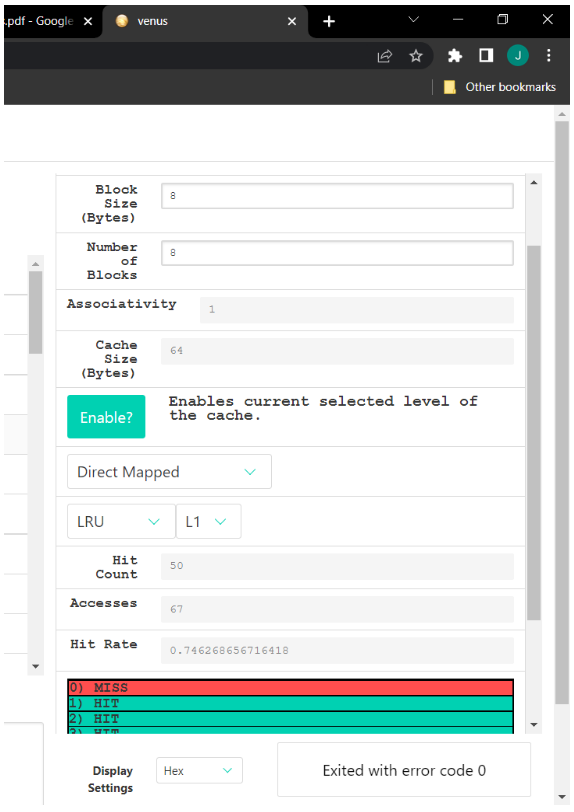
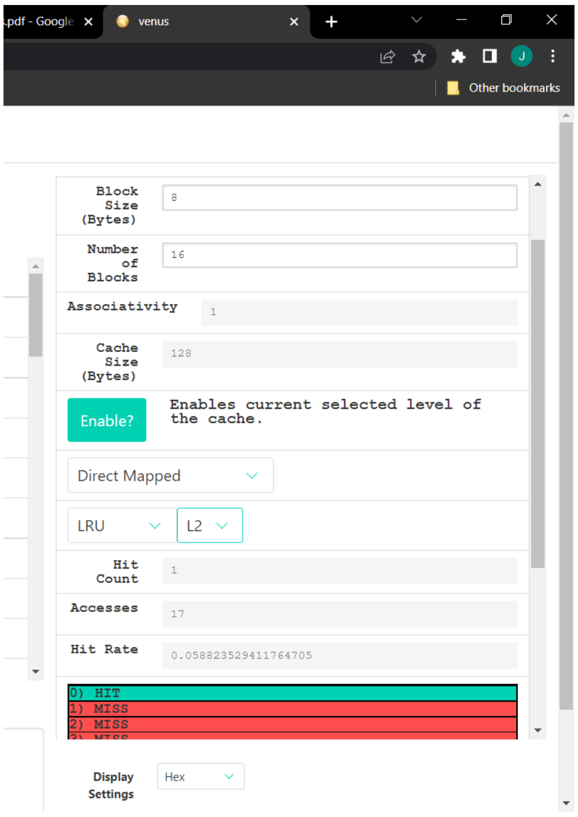
L2:

Block Size (Bytes): 8
Number of Blocks: 16
Associativity: 1
Cache Size (Bytes): 128
Enable? Enables current selected level of the cache.
Direct Mapped
LRU | L2
Hit Count: 0
Accesses: 1
Hit Rate: 0

```
0) MISS
```

Display Settings: Hex

Original with option = 1

L1:	L2:
	

L1:

Block Size (Bytes): 8
Number of Blocks: 8
Associativity: 1
Cache Size (Bytes): 64
Enable? Enables current selected level of the cache.
Direct Mapped
LRU | L1
Hit Count: 50
Accesses: 67
Hit Rate: 0.746268656716418

```
0) MISS
1) HIT
2) HIT
3) HIT
```

Display Settings: Hex
Exited with error code 0

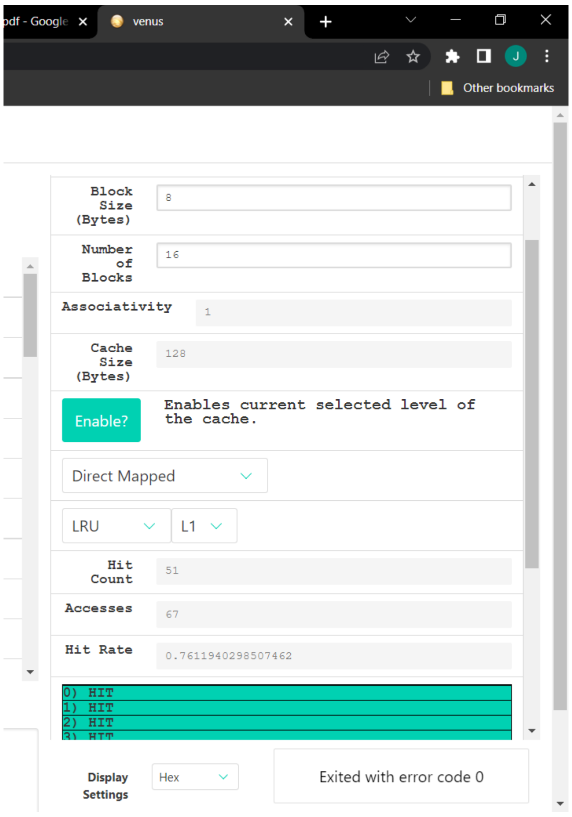
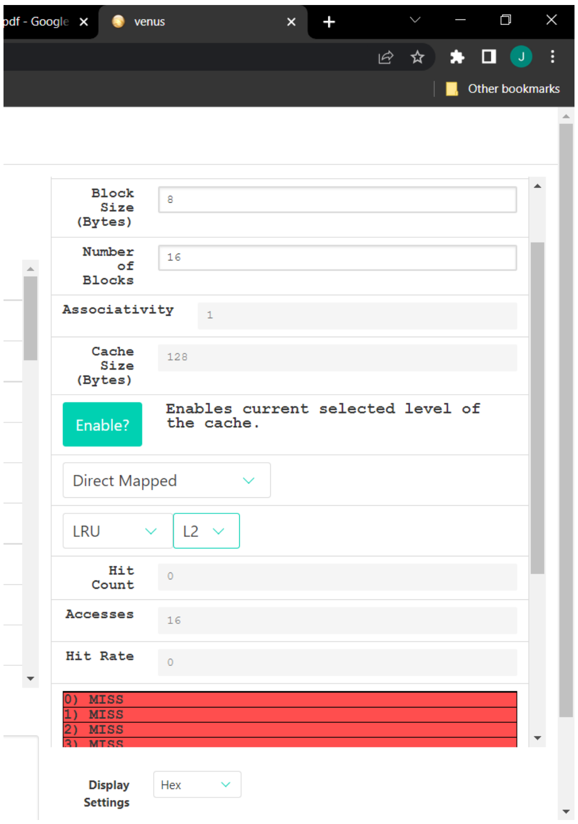
L2:

Block Size (Bytes): 8
Number of Blocks: 16
Associativity: 1
Cache Size (Bytes): 128
Enable? Enables current selected level of the cache.
Direct Mapped
LRU | L2
Hit Count: 1
Accesses: 17
Hit Rate: 0.058823529411764705

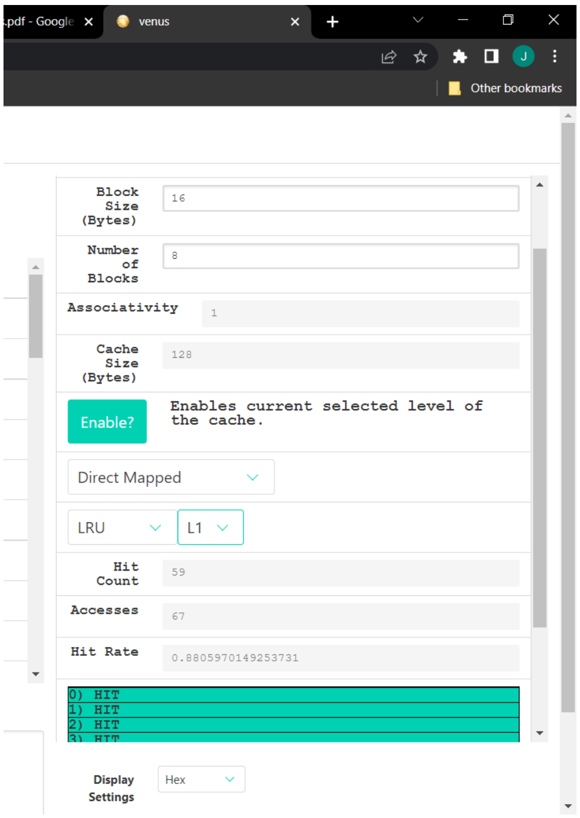
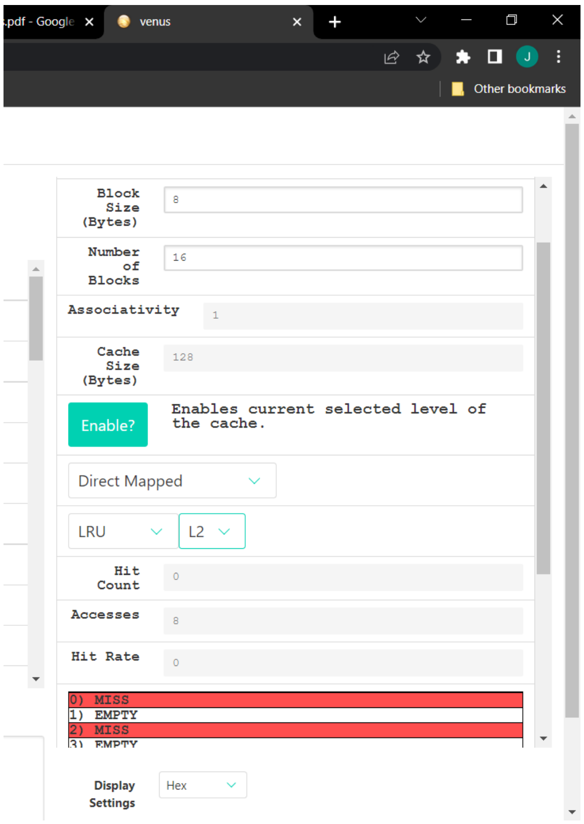
```
0) HIT
1) MISS
2) MISS
3) MISS
```

Display Settings: Hex

Case 1: block in L1 increased to 16

L1:	L2:
 pdf - Google · venus Other bookmarks Block Size (Bytes) 8 Number of Blocks 16 Associativity 1 Cache Size (Bytes) 128 Enable? Enables current selected level of the cache. Direct Mapped LRU · L1 Hit Count 51 Accesses 67 Hit Rate 0.7611940298507462 0) HIT 1) HIT 2) HIT 3) HIT Display Settings · Hex Exited with error code 0	 pdf - Google · venus Other bookmarks Block Size (Bytes) 8 Number of Blocks 16 Associativity 1 Cache Size (Bytes) 128 Enable? Enables current selected level of the cache. Direct Mapped LRU · L2 Hit Count 0 Accesses 16 Hit Rate 0 0) MISS 1) MISS 2) MISS 3) MISS Display Settings · Hex

Case 2: block size in L1 increased to 16

L1:	L2:
 .pdf - Google · venus Other bookmarks Block Size (Bytes) 16 Number of Blocks 8 Associativity 1 Cache Size (Bytes) 128 Enable? Enables current selected level of the cache. Direct Mapped LRU · L1 Hit Count 59 Accesses 67 Hit Rate 0.8805970149253731 0) HIT 1) HIT 2) HIT 3) HIT Display Settings · Hex	 .pdf - Google · venus Other bookmarks Block Size (Bytes) 8 Number of Blocks 16 Associativity 1 Cache Size (Bytes) 128 Enable? Enables current selected level of the cache. Direct Mapped LRU · L2 Hit Count 0 Accesses 8 Hit Rate 0 0) MISS 1) EMPTY 2) MISS 3) EMPTY Display Settings · Hex