CSE 112
COMPUTER ORGANISATION

END SEMESTER CACHE ASSIGNMENT

By Yatharth Taneja (2019346)

INTRODUCTION

A cache is a hardware used by CPU to access data from the memory faster and reduce cost, energy, and time. In this project, a cache is build of the size of the user's choice and can use basic commands such as read and write on the basis of selection of mapping.

The three types of mapping included are:

1. Direct mapping
2. Associative mapping
3. N-way set Associative mapping

WORKING

The code is written in python and can be run from the terminal. The file can be run by the command py 2019346_YatharthTaneja_FinalAssignment.py . It asks the users for Input of Size of cache and block size and itself calculates the number of lines in the cache and main memory, and the indexes are assigned.

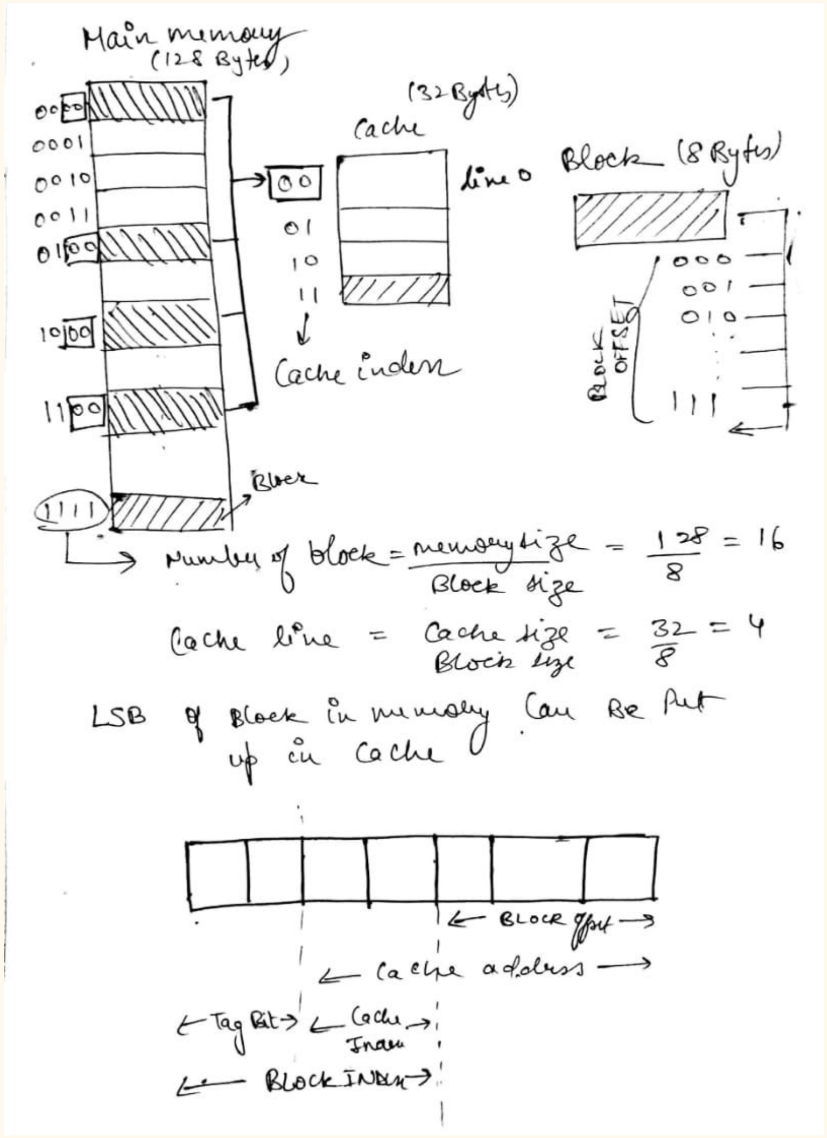
Next, it asks for what type of mapping we require and it can be selected as option 1,2,3 in the same order above. For N-way Associative it requires input N also.

There are 3 basic commands and the following syntax is to be followed.

1. read address
2. write address
3. exit

All the commands should be in lower case.

DIRECT MAPPING



In direct mapping, the cache indexes are compared with the LSB of the block indexes of the main memory. And only those blocks can be loaded into the particular cache line which has a common index

Here in this example, 0000,0100,1000,1100 can be loaded into the first line of cache

Hence the remaining 2 bit in the block index is called tag bits.

If 0000 is in the 1st line of cache and 1100 is requested to load then first the CPU will compare the cache bit i.e 00 and then compare the tag bit 00 and 11 since they are not the same will remove 0000 and load 1100 in the first line.

The complete address is of the size of BLOCK INDEX + BLOCK OFFSET.

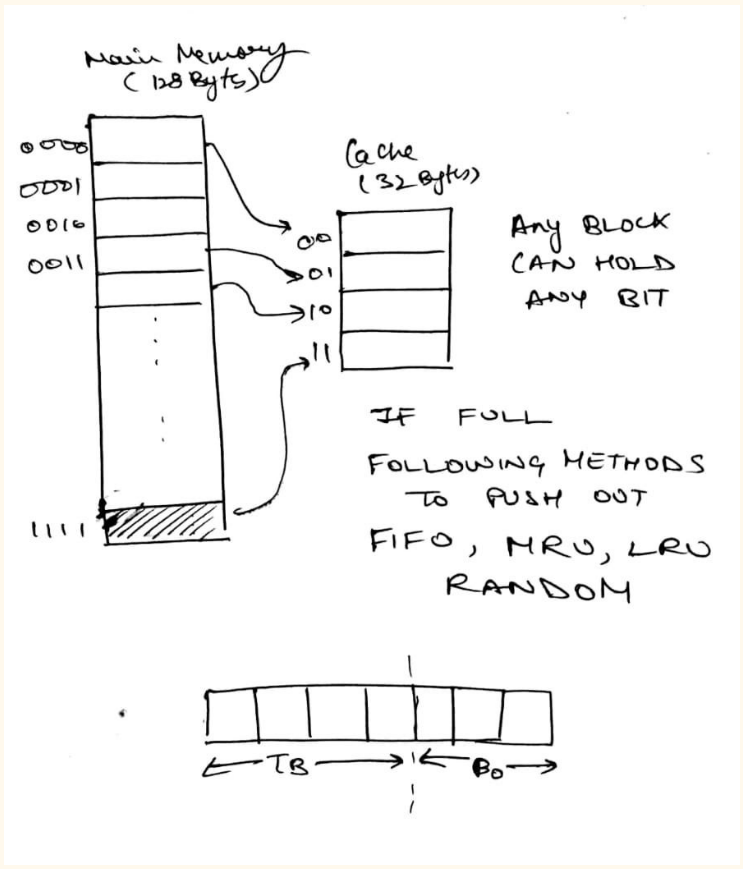
BLOCK OFFSET = log2(BLOCKSIZE)

So for the input address in the code, it searches the cache index and then compares the tag bits.

LOCALITY OF REFERENCE

The whole block is loaded hence if there is a different block offset but the tag and the cache index are the same it is a hit. As the whole corresponding block is in the cache

ASSOCIATIVE MAPPING



In associative mapping, we have the freedom to load any block index on any cache line.

Once the Cache is full there are 4 ways it discards a block whenever it needs to write a new block.

These methods are FIFO, Most recently used, Least recently used, and random.

In this project, I have used the FIFO method.

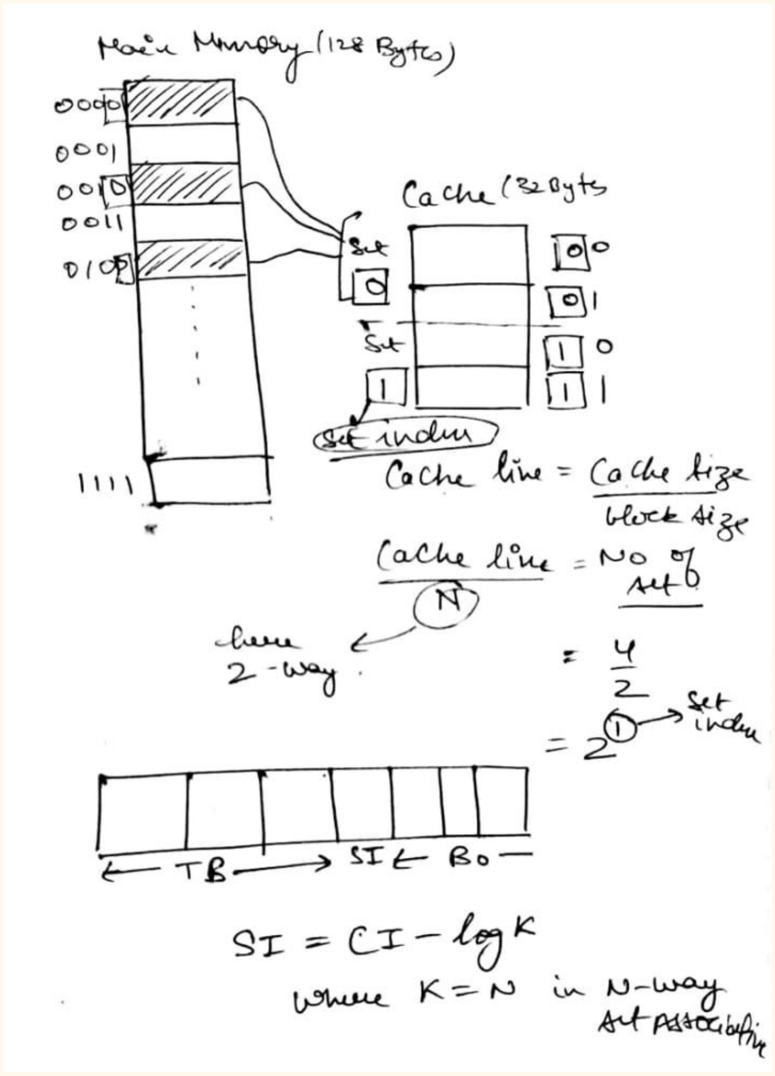
The address is simple

Tag Bits+ Block Offset

However, this method is really costly in real life.

In the code, it is simply implemented via a queue in the form of an array. The code outputs the address it is discarding at the intermediate step if there is a miss and the array is full. It simply searches for the element (tag bit) and then if found returns a hit or writes if a miss.

N-WAY SET ASSOCIATIVE MAPPING



It is a combination of the above two methods. And the approach is we divide Cache lines into number of sets.

Number of sets =cache line/N

Now similar to direct mapping the LSB of the block index is compared to the set index and the block can be loaded into that set which corresponds to the same index. However now in that set we have the flexibility of associative mapping and can place the block at the empty spot or Follow the FIFO approach.

The addressing is similar to direct mapping only difference is

Block index = Tagbit + Set index

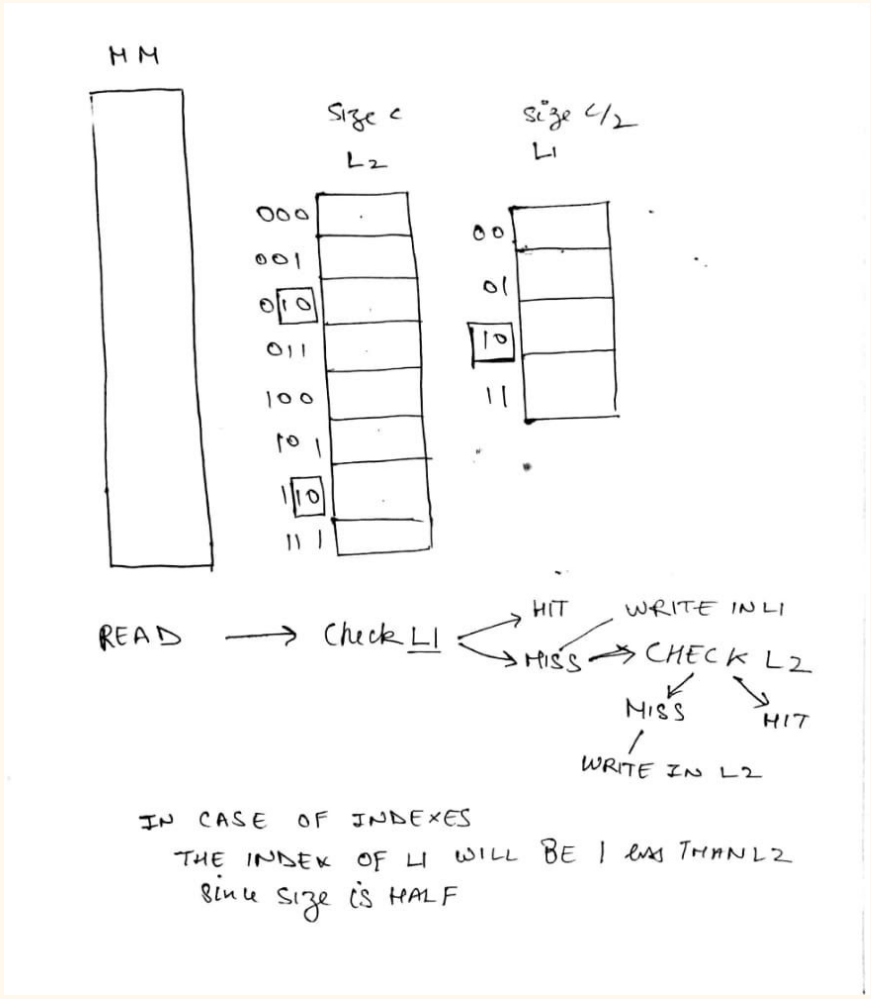
And set index= Cache index-log2(N)

In the code, I have made a 2D array of size [cache line/N][N].

If a user inputs an address to write it looks for the set index and goes to that set. And loads the array of N size where like associative mapping a queue is followed for searching and writing. If the block is already in the array it is a hit. Else it follows the FIFO approach and discards the first entry (if the array is full). The discarded entry is displayed along with the cache.

BONUS

For the bonus, another cache has to be made of half the size of the previous cache and all the concepts remain the same only there is an extra added level.



Hence whenever there is a read statement the program will first check-in L1 cache if it is a hit it will return a **hit**.

Else it will write in L1 and go to check L2 if there is a **hit** it will return hit else if there is a miss it will write in L2 also.

The indexing will be one less in case of L1

Eg Read address

0000111000 110 111 for a 16-bit machine with a block size of 8 hence

110 i.e 6 is cache index for L2 and 10 i.e 2 is cache index of L1

The same approach is followed while writing.

ASSUMPTIONS

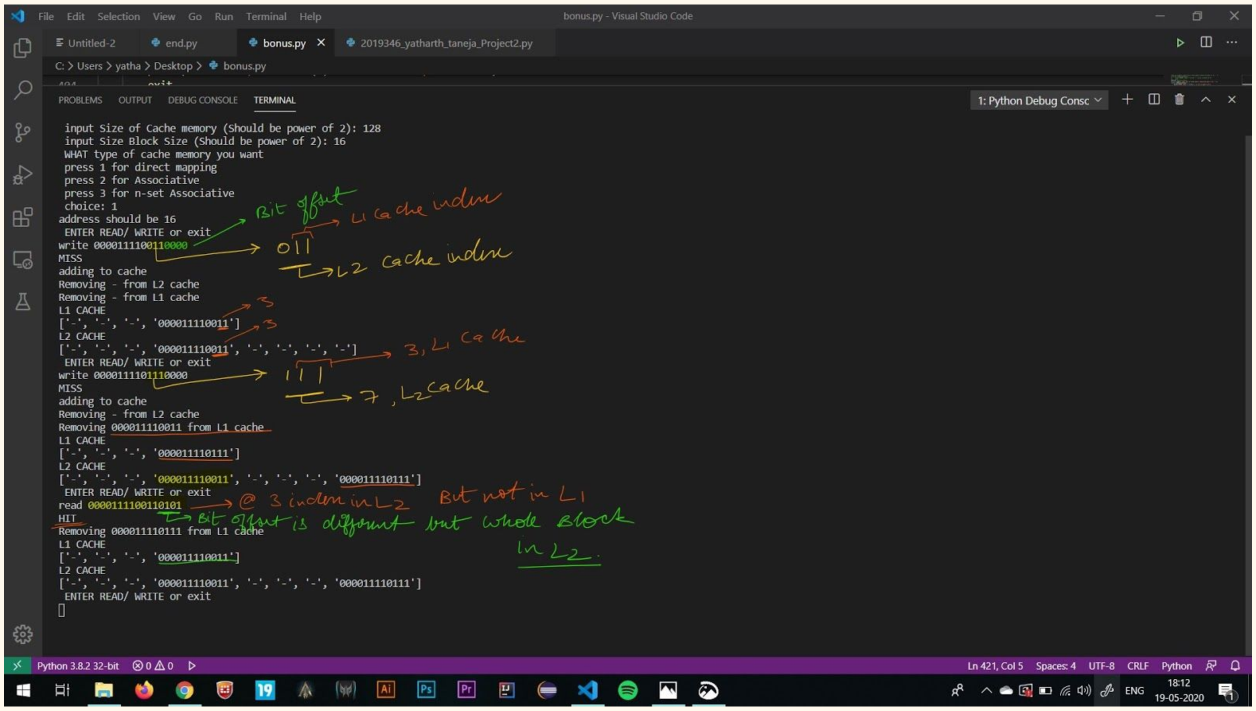
1. 1 word = 16-bit length hence the address will be 16 bit long.
2. The size of cache, block, N in set-associative all should be of the power of 2.
3. The cache should be significantly small than the main memory
4. Block size should also be smaller than the cache size significantly.
5. When a block is loaded all the block offsets are loaded for that block index hence it's a hit (Assumption in computer organization for data: Locality Of Reference)
6. read, write and exit statement are in lower case
7. All the sizes are in bytes.
8. The output displayed in the cache is the block index (tag bit + cache index) for clarity, not the whole address.
9. Indexing starts from 0.

ERROR HANDLING

1. Syntax error
2. Invalid address
3. The size cache, block, N in set-associative not in the power of 2.
4. Size of block exceeding
5. Size of cache bigger than the main memory

EXPLAINING OUTPUT

Direct mapping.



```
 input Size of Cache memory (Should be power of 2): 128
 input Size Block Size (Should be power of 2): 16
 WHAT type of cache memory you want
 press 1 for direct mapping
 press 2 for Associative
 press 3 for n-set Associative
 choice: 1
address should be 16
 ENTER READ/ WRITE or exit
write 0000111100110000
MISS
adding to cache
Removing - from L2 cache
Removing - from L1 cache
L1 CACHE
['-', '-', '-', '000011110011']
L2 CACHE
['-', '-', '-', '000011110011', '-', '-', '-', '-']
 ENTER READ/ WRITE or exit
write 0000111101110000
MISS
adding to cache
Removing - from L2 cache
Removing 000011110011 from L1 cache
L1 CACHE
['-', '-', '-', '000011110111']
L2 CACHE
['-', '-', '-', '000011110011', '-', '-', '-', '000011110111']
 ENTER READ/ WRITE or exit
read 0000111100110101
HIT
Removing 000011110111 from L1 cache
L1 CACHE
['-', '-', '-', '000011110011']
L2 CACHE
['-', '-', '-', '000011110011', '-', '-', '-', '000011110111']
 ENTER READ/ WRITE or exit
```

File Edit Selection View Go Run Terminal Help
bonus.py - Visual Studio Code
Untitled-2
end.py
bonus.py
2019346_yatharth_taneja_Project2.py
C: > Users > yatha > Desktop > bonus.py
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: Python Debug Consc
Bit offset
L1 cache index
011
L2 cache index
3
3
3, L1 cache
111
7, L2 cache
@ 3 index in L2
But not in L1
Bit offset is different but whole block in L2.
Python 3.8.2 32-bit
Ln 421, Col 5 Spaces: 4 UTF-8 CRLF Python
18:12
19-05-2020

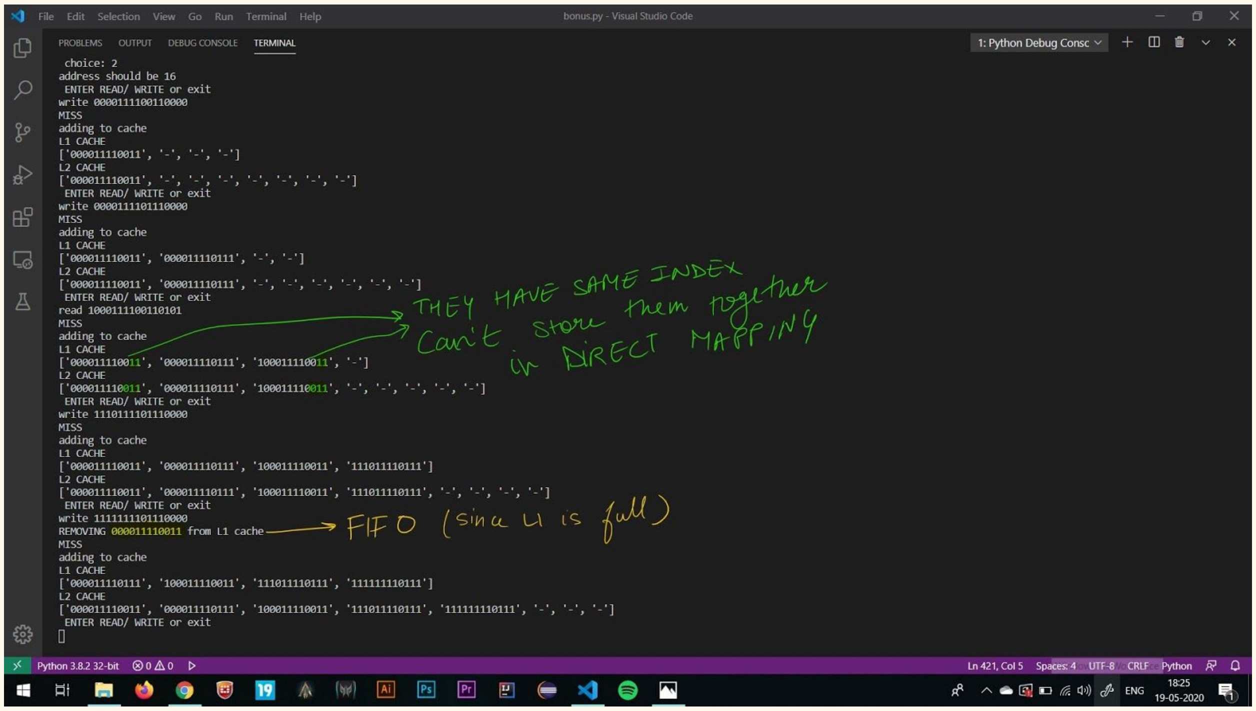
As you can see the address is 16 bit long . And the last 4 bits are block offset since block size is of 16 bits. The following LSB are cache index (next 2 for L1 and next 3 for L2 since L1 is of size 64 bits (4 cache lines) and L2 is of size 128 bits (8 cache lines)).

Hence when the first write statement is given the block index is written at the 3rd index of both cache L1 and L2 as 11 and 011 are 3 in decimal.

But for the next write statement, L1 has index 3 (11) and L2 has index 7 (111) hence it will write at the 7th index in L2 but will have to remove what was at L1 at index 3

Now if we'll read the address which is removed from L1, but it is at index 3 of L2. only we'll change the block offset. Now we get a Hit since it's block is already in the L2 cache (locality of reference). But it will write the block in L1 since it was removed.

Associative mapping



```
 choice: 2
address should be 16
 ENTER READ/ WRITE or exit
write 0000111100110000
MISS
adding to cache
L1 CACHE
['000011110011', '-', '-', '-']
L2 CACHE
['000011110011', '-', '-', '-', '-', '-', '-', '-']
 ENTER READ/ WRITE or exit
write 0000111101110000
MISS
adding to cache
L1 CACHE
['000011110011', '000011110111', '-', '-']
L2 CACHE
['000011110011', '000011110111', '-', '-', '-', '-', '-', '-']
 ENTER READ/ WRITE or exit
read 1000111100110101
MISS
adding to cache
L1 CACHE
['000011110011', '000011110111', '100011110011', '-']
L2 CACHE
['000011110011', '000011110111', '100011110011', '-', '-', '-', '-', '-']
 ENTER READ/ WRITE or exit
write 1110111101110000
MISS
adding to cache
L1 CACHE
['000011110011', '000011110111', '100011110011', '111011110111']
L2 CACHE
['000011110011', '000011110111', '100011110011', '111011110111', '-', '-', '-', '-']
 ENTER READ/ WRITE or exit
write 1111111101110000
REMOVING 000011110011 from L1 cache
MISS
adding to cache
L1 CACHE
['000011110111', '100011110011', '111011110111', '111111110111']
L2 CACHE
['000011110011', '000011110111', '100011110011', '111011110111', '111111110111', '-', '-', '-']
 ENTER READ/ WRITE or exit
```

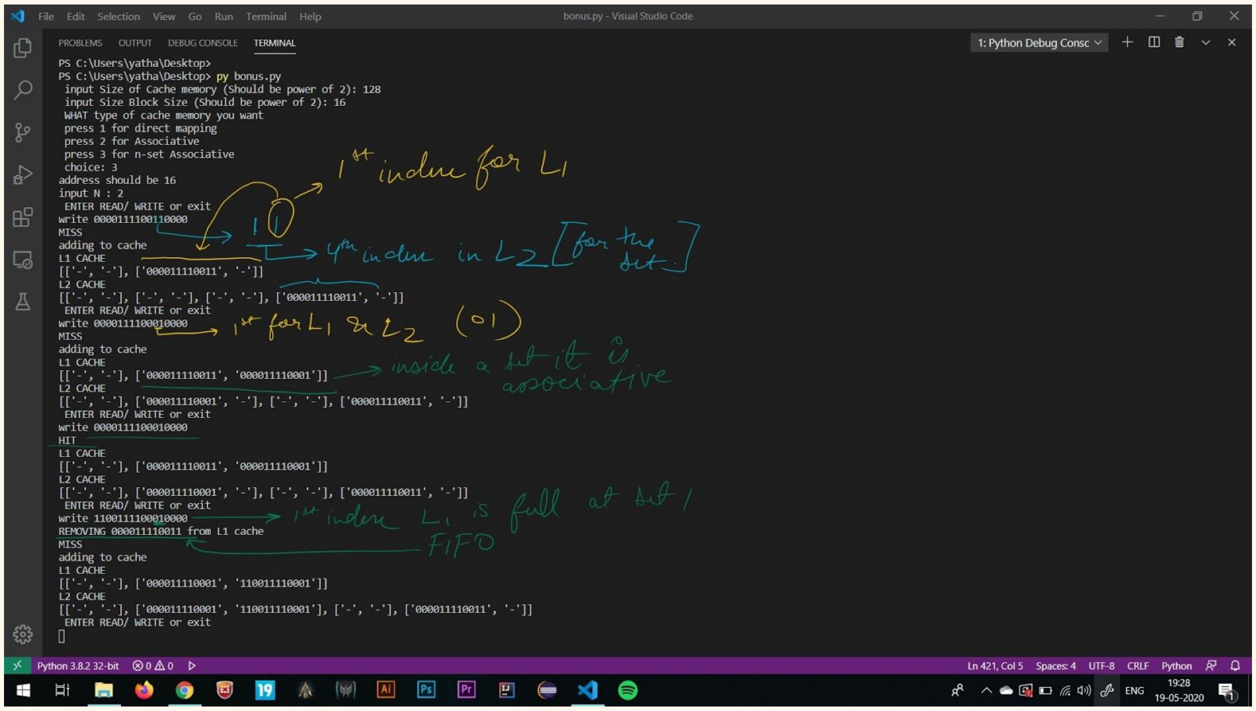
As you can see in the above example (The size of block and cache are the same from the direct mapping example) the block loaded at index 0 has the same cache index as loaded in index 2. This is not possible in direct mapping.

Since any block can be loaded anywhere. We have to decide which block to discard.

I'm following the FiFo approach as you can see when L1 is full and L2 is still empty. The new write block will go at index 4 of L2 and in L1 we'll discard the 0 index block and shift all the blocks one index backward and the new data will come at index 3.

A similar thing will happen once L2 cache is also filled and we'll follow FIFO.

Set Associative



```
PS C:\Users\yatha\Desktop>
PS C:\Users\yatha\Desktop> py bonus.py
 input Size of Cache memory (Should be power of 2): 128
 input Size Block Size (Should be power of 2): 16
 WHAT type of cache memory you want
 press 1 for direct mapping
 press 2 for Associative
 press 3 for n-set Associative
 choice: 3
address should be 16
input N : 2
 ENTER READ/ WRITE or exit
write 0000111100110000
MISS
adding to cache
L1 CACHE
[['-', '-'], ['000011110011', '-']]
L2 CACHE
[['-', '-'], ['-', '-'], ['-', '-'], ['000011110011', '-']]
 ENTER READ/ WRITE or exit
write 0000111100010000
MISS
adding to cache
L1 CACHE
[['-', '-'], ['000011110011', '000011110001']]
L2 CACHE
[['-', '-'], ['000011110001', '-'], ['-', '-'], ['000011110011', '-']]
 ENTER READ/ WRITE or exit
write 0000111100010000
HIT
L1 CACHE
[['-', '-'], ['000011110011', '000011110001']]
L2 CACHE
[['-', '-'], ['000011110001', '-'], ['-', '-'], ['000011110011', '-']]
 ENTER READ/ WRITE or exit
write 1100111100010000
REMOVING 000011110011 from L1 cache
MISS
adding to cache
L1 CACHE
[['-', '-'], ['000011110001', '110011110001']]
L2 CACHE
[['-', '-'], ['000011110001', '110011110001'], ['-', '-'], ['000011110011', '-']]
 ENTER READ/ WRITE or exit
```

1st index for L1
1 1
4th index in L2 [for the set]
1st for L1 & L2 (01)
inside a set it is associative
1st index L1 is full at set 1
FIFO
bonus.py - Visual Studio Code
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Python 3.8.2 32-bit
Ln 421, Col 5 Spaces: 4 UTF-8 CRLF Python
19:28 19-05-2020

This is a mixture of Both the above approaches. Here I have kept the size the same and made sets of 2. For the understanding purpose these sets are made visible in the cache.

Again last 4 bits are bit offset. But here we'll consider the set indexes instead of cache indexes.

Set indexes= log2=(cache line / n).

Hence 1 bit is for L1 cache from LSB after block offset and 2 bits for L2.

For the first write statement, it is 1 (index 1) for L1 hence we'll write in set 1, and inside a set is Associative mapping with FIFO.

While for the L2 it is 11 (index 3) hence we'll write in set 3. Again FIFO approach.

Now I gave an address in the write statement which is already in cache hence cache will give a hit as it is already there and it is done so that there is no overwriting data or excess use of space.

Now the next write statement has 01 and in which again 1 is index for L1 and 01 for L2. hence we'll write at index 1 of both cache sets. Now set with index 1 of L1 is full any other write coming to this set will remove the already inside block index (FIFO). Which is shown in the last write statement of the screenshot.