# COMPUTER ORGANISATIONS

***Cache Assignment***

*Mansi Singhal*

*(2019370,Group 6)*

**Cache Memory** is a special very high-speed memory. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.

For this project, I have built a one-level cache on based of three mapping: Direct Mapping, fully associative mapping and n-way set associative mapping using **Java**.

The user is asked to put in three inputs first:

1. The bits of the physical address N.
2. The number of cache lines CL
3. The block size B

(where CL, and B are in powers of 2)

The cache is then initially assumed to be empty. Then input commands for read and write statements are taken.

Input format for read statement: N bit binary address (1100101 )

Input format for write statement: N bit binary address followed by data string (1100101 abcd)

Then the cache is built according to each input statement. Since this is a standalone cache so the only the data for the words with write statements will be shown in the cache. The rest of the word addresses will show data = null.

Assumptions

1. Input address must always be of N-bits and in binary.
2. The cache is initially empty.
3. For associative and set associative mappings, FIFO (First in, First out) algorithm of replacement is used.
4. CL, B and n (for set associative) are in powers of 2.
5. The data stored is of string format and there are no spaces in it.
6. The data for the addresses which are present in cache but have not been written in the input will be shown as null since we are not maintaining main memory.

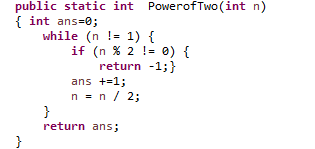
Error Handling

1. If the length of the input for read and write statements is more than 2 then an invalid statement error is thrown.
2. If the inputs of CL ,B and n are not powers of 2 then an Invalid input for dimensions of cache is thrown.

Explanation of the code

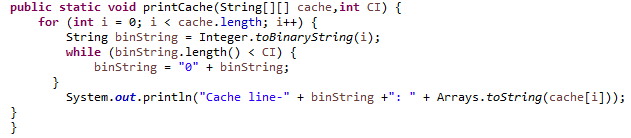
**Functions defined :**

1. PowerofTwo



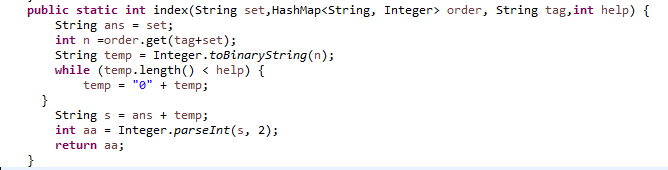
Returns the exponent for powers of 2. If the number supplied as input is not a power of 2 then returns

-1.

1. printCache

Prints the 2D array of cache.

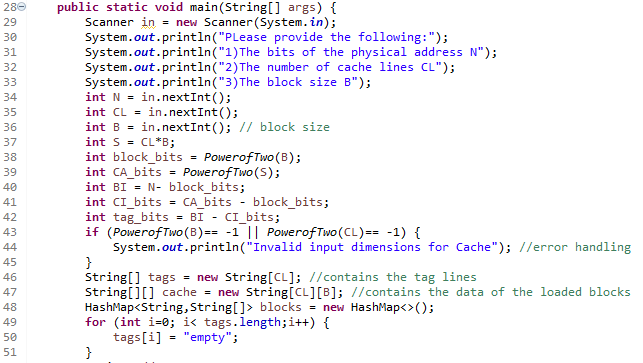
1. index (Only present for set associative code)



Returns the cache line number at which a particular input block is to be added.

**Variables and tables defined:**

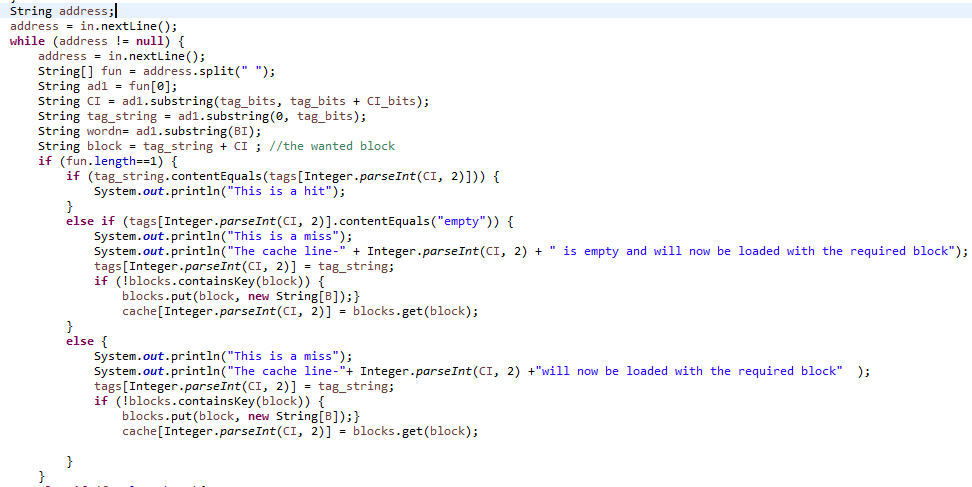
|  |  |  |
| --- | --- | --- |
| S No | Name | Description |
| Common for all three: | | |
| 1. | N | Number of bits in the address |
| 2. | CL | Number of cache lines |
| 3. | B | Number of blocks |
| 4. | S | Cache size (CL\*B) |
| 5. | block\_bits | Number of bits required to represent a word in a block |
| 6. | BI | Index from which word representation starts |
| 7. | cache | 2D Array storing the cache data, with rows representing the lines of cache |
| 8. | blocks | HashMap storing the blocks that have been used |
| 9. | address | String containing the input data |
| 10. | fun | Array containing data of address after splitting according to spaces |
| 11. | ad1 | String containing the address of the word which came as in input |
| 12. | data | String containing the data to be stored at the address |
| 13. | tag\_string | String containing the tag bits of the address |
| 14. | block | String containing the block bits of the address |
| 15. | wordn | String containing the word representation bits of the address |
| Direct Mapping: | | |
| 1. | CA\_bits | Number of bits required to represent the cache address |
| 2. | CI\_bits | Number of bits required to represent the cache line |
| 3. | tags | 1D Array which contains tag strings corresponding to the blocks that have been loaded in the respective cache lines |
| Fully associative Mapping: | | |
| 1. | tags | ArrayList which works as a queue to keep a check on which tag is to be deleted according to the FIFO logic |
| 2. | order\_store | The order of the blocks in cache is stored in this HashMap because the order in ArrayList keeps changing |
| Set Associative Mapping | | |
| 1. | set | It has a hashMap of sets with arraylists of every set working as a queue to keep a check on which tag is to be deleted according to the FIFO logic |
| 2. | order\_store | The order of the blocks in the set is stored in this HashMap because the order in ArrayList keeps changing |
| 3. | no\_sets | Number of sets that will be made |
| 4. | n | Set size (taken as input) |
| 5. | help | Number of bits of the cache index- number of set index bits |
| 6. | set\_bits | Number of bits required to represent the set |



Inputs for the cache dimensions are taken and then accordingly it is calculated that how many bits will represent what. For e.g., the exponent of Block size will give us the number of bits that are required for representing the word address in a block. Two arrays for tags and cache are also declared here. Along with the HashMap of the used blocks.

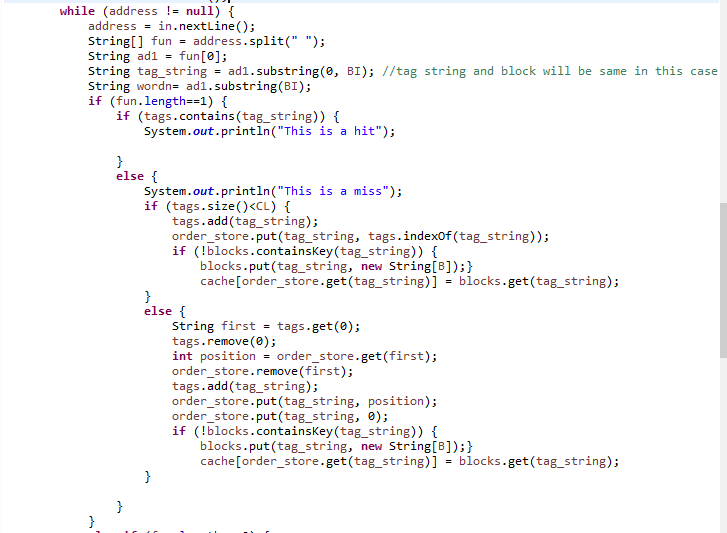
Then we take the string input. If the length of this string input is equal to 1. This is read statement, if it is 2 then it is a write statement, otherwise it is invalid and an error is thrown.

1. Direct Mapping



In case of Direct Mapping, inside both length ==1 and length ==2 conditions, we further have 3 conditions:

1. When the block is already loaded at the Cache line – We check this using the tag array that we are maintaining. If the tag string of the input is equal to the tag at the index of the cache line where the required address is supposed to go then this is a hit. We then update the value in case of write input and then display cache.
2. When no block is loaded at the cache line- then we create the block and load it in it cache array in the line it supposed to go to according to the Cache Address string with updated values according to input.
3. When some other block is present at the cache line – We check this using the tag array that we are maintaining. If the tag string of the input is not equal to the tag at the index of the cache line where the required address is supposed to go then this is a miss. We will replace the existing block at the line with the required block and update the data as well in case of write operation.
4. Associative Mapping



In case of fully associative mapping, , inside both length ==1 and length ==2 conditions, we further have 2 conditions:

1. If the block is already present in tags- then this is a hit and the cache will be shown after updating the data in case of write statements.
2. If the block is not already present – then there are further two possibilities:
   1. If all lines of cache are not yet filled- then the new block will be added in the next new line which is available in the cache.
   2. If all lines of cache are filled- then replacement will take place. So the first element of the arraylist will be removed since the arraylist is maintained according to the entry order of blocks. The order of this block will be checked from the order\_store HashMap and the new block will be added in it’s position in the cache and will also enter the arraylist as the latest element.

3 .N-way set associative mapping



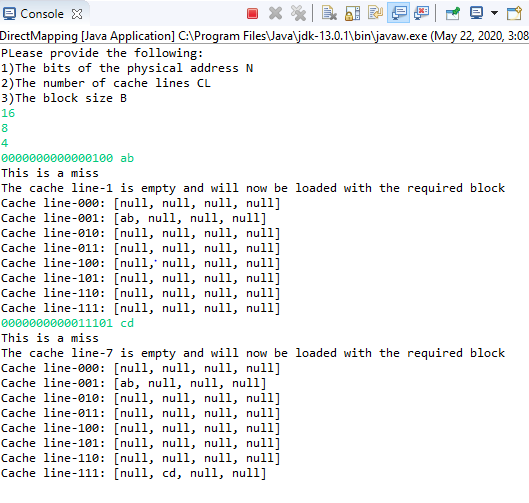
In case of fully associative mapping, , inside both length ==1 and length ==2 conditions, we firstly create a set if it doesn’t already exists. Then we further have two conditions:

1. If the set already contains the tag- then this is a hit and the cache is printed.
2. If the block is not already present in the set – then there are further two possibilities:
   1. If all lines of cache of a set are not yet filled- then the new block will be added in the next new line which is available in the cache set.
   2. If all lines of the cache set are filled- then replacement will take place. So the first element of the arraylist of the set will be removed since the arraylist is maintained according to the entry order of blocks in a set. The order of this block will be checked from the order\_store HashMap and the new block will be added in it’s position in the cache and will also enter the arraylist of the set as the latest element.

Working examples of the code

Here are some inputs and outputs for the three codes:

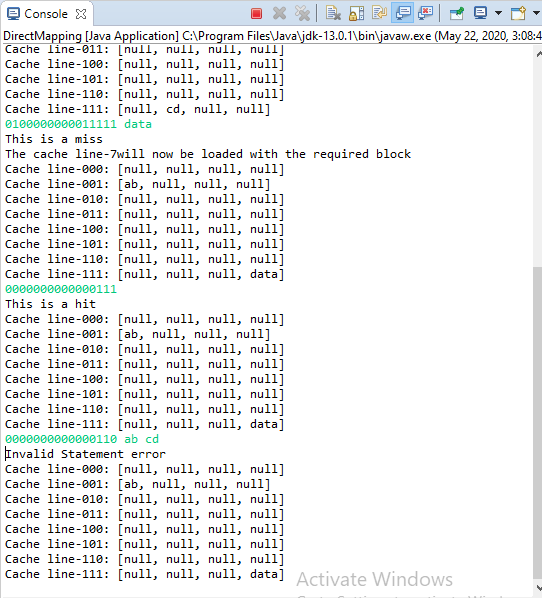
1. **Direct Mapping**



Explanation:

The address is of 16bits in this input.According to the other dimensions we can see that,the first 11 bits will represent the tagline in this, next 3 will show the cache index and last 2 will represent the word index.

For the first input “001” is the Cache index. So the block “00000000000001” will be loaded on the cache line “001” which is CL-1 with the data being added at “00” of the block. Similarly for the second input the block will be loaded on the line “111”which is CL-7.



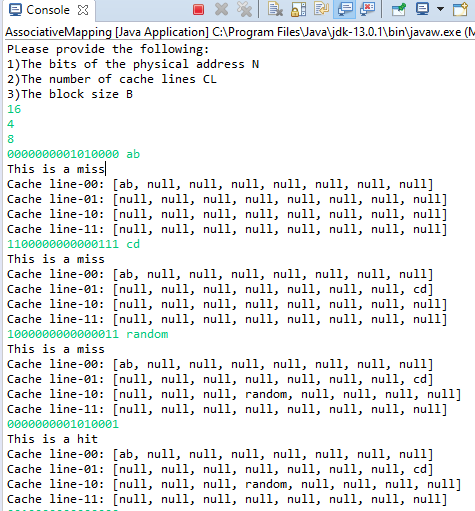
Explanation:

Now the input address is for a block which is supposed to be at “111” block, but there is already a previous block which is better at that line so replacement will take place and the new block will be placed on cache line “111” which is CL-7.

For the next statement, the requested block is already present at the cache line so this is a hit and Data at word “11” will be sent to CPU from the cache accordingly.

In the third input, the write statement has a space between the data, which is invalid according to the assumptions, so an error is thrown.

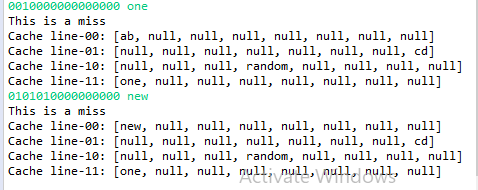
1. Fully associative Mapping



Explanation:

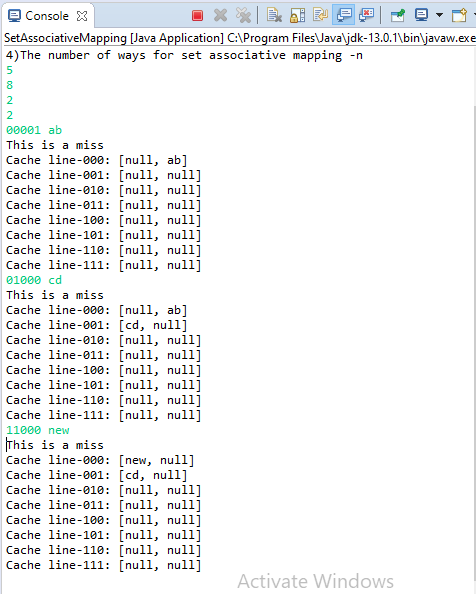
The address bits are 16, we have 4 number of cache lines and 8 blocks. So to represent the word we will need 3 bits (because 2^3=8) so the first 13bits will be the block (or tag) and rest will represent the word.

Since the cache is initially empty, the first three inputs which are of different blocks are loaded in order. Then for the fourth input the block is same as the first one, so it is a hit and the cache will return the data at word “001” to the CPU.



Explanation: Now one new block is called so the cache is full. So for the last input now we will have to replace one of the existing blocks from the cache, so since we are using FIFO logic, we will replace the very first block which was added and load the new block in place of it. And hence the initial block with “ab” as data at the first word is replaced with the new block with “new” as data.

1. **N-way set associative mapping**



In this input, we have a 5 bit address and 2 blocks so 1 bit will represent the word. Also, n is 2 and Cl is 8 so number of sets will be 4 (8/2). So we will need 2 bits for representation of set. So rest 2 bits work as the tag bits. In the sample input we first insert two blocks in the set “00”. And the one more block of the same set is requested to now we will have to replace according to fifo so the first block which entered with data “ab” at “1” will be replaced with the new block of the same set.(Set size was 2)