CSE112 End Semester Assignment

Computer Organisation

Aditi Sejal

2029228

CO Group 3

Section B

1. Project Aim

To build a 2-level cache.
A 2-level standalone cache consisting of L1 cache at level 1 (higher level i.e. closer to CPU) of size CL/2 lines and L2 cache at level 2 (lower level i.e. closer to main memory) of size CL lines have been simulated using java code for 3 different mappings namely:

a) Direct Mapping
b) Fully Associative Mapping
c) N-way Set Associative Mapping

2. Project Interface

The project file is a .java file with a menu-based cache simulation providing options for 3 different mappings as listed above. The user is expected to provide the following inputs after choosing a mapping for the cache:

a) Size of the main memory
b) Number of lines in cache
c) Size of block

This is followed by a menu asking the user for the operation to be performed. Two main operations are available namely:

a) Read from cache: This option prompts the user for a specified number of bit-address following which the program proceeds to search the cache and output the data on the console while printing useful messages like the Cache Hit/Miss along the way.
b) Write to cache: This option prompts the user for a specified number of bit-address and an integer type data following which the program proceeds to load the data into cache at the given address while printing useful messages like Cache Hit/Miss and/or Movement/Replacement of blocks in cache along the way.

3. Project Workflow

The project workflow is decided as per the options and operations mentioned above chosen by the user. The common class is dmcache() which contains the properties for cache lines like tag, index, set, time last used, words in the block, valid bit, dirty bit etc. for use whenever and however required. The caches are made up as arrays/arraylists of objects of this class. The class has a show() function to print formatted outputs for the cache structure. The main class contains an interactive menu which operates differently for each mapping for cache as explained below.

3.1 Direct Mapping

In case of direct mapping, the given address is broken down into appropriate tag, index and block offset bits. The read and write operation then search the cache for given block number in specified index (index in the program has been implemented using index of an array of an object of cache lines) according to the rule in direct mapping. The exact flow of the program is explained through flowcharts below.

API Documentation

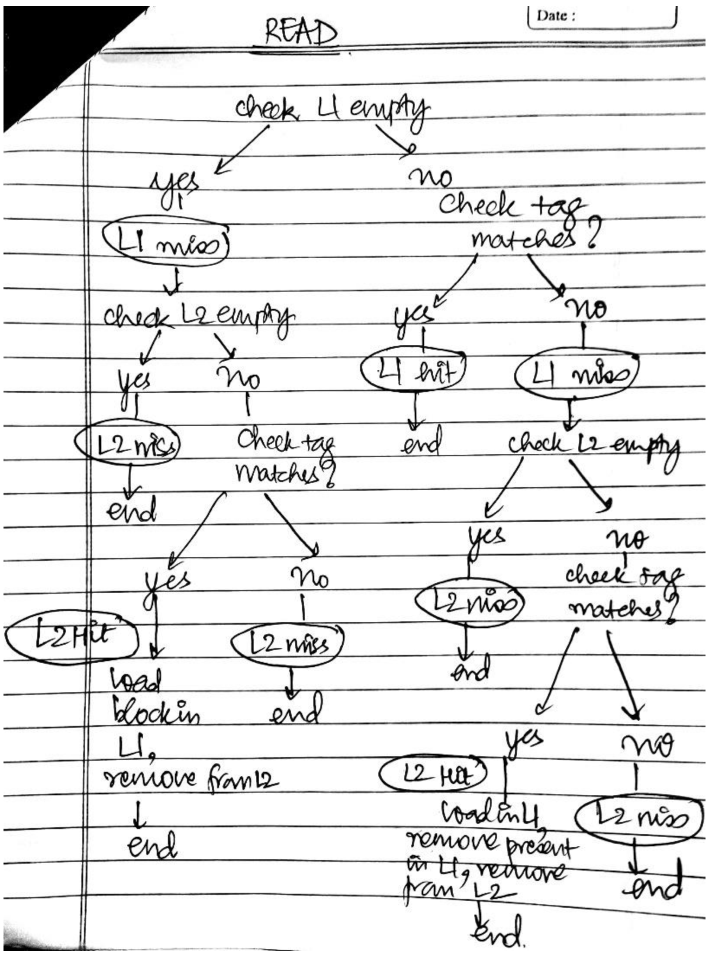
The Direct class in the AditiSejal_2019228_FinalAssignment.java file handles this mapping.

a) `Direct(long mm, long ccll, long bb)`
The constructor initialises all essential M, CL, S, B etc. values and creates an empty cache structure.

b) `public void l1printer()`
This function prints the cache structure of L1 cache.

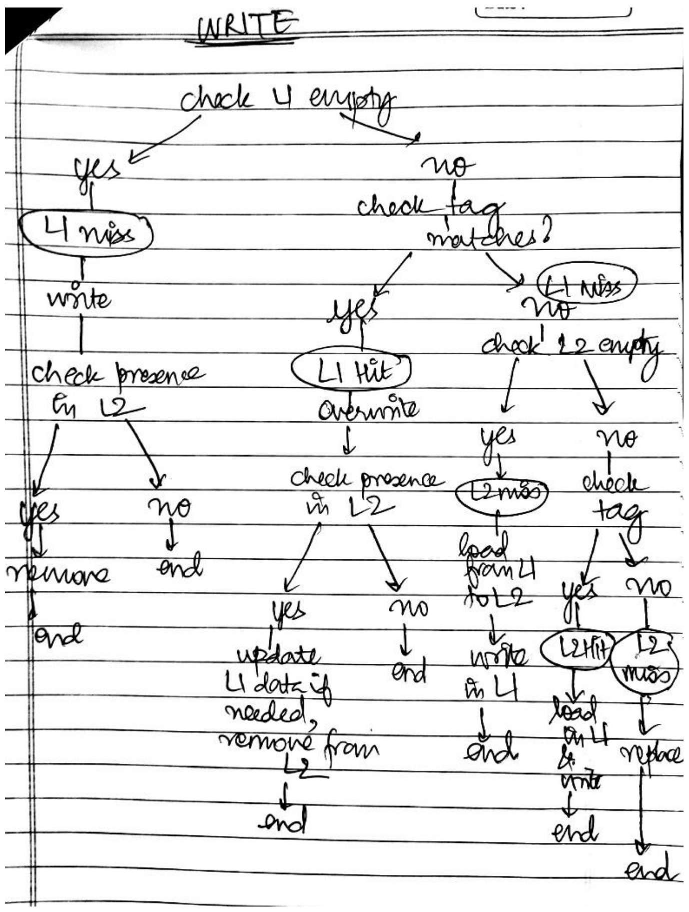
c) `public void l2printer()`
This function prints the cache structure of L2 cache.

d) `public void reader(String address)`
This function takes address of word as input and handles the read operation in direct mapped 2-level cache. It operates according to this flowchart. A read hit means that the block containing the word is already in the cache. If there occurs an L2 hit and a L1 miss, data is loaded into L1 which might cause replacement/movement of blocks for which appropriate messages are printed.



e) `public void writer(String address, int data)`

This function takes address of word and data to be written at that word as input and handles the write operation in direct mapped 2-level cache. It operates according to the flowchart. A write hit means that the block containing the word is already in the cache. If there occurs an L2 hit and a L1 miss, data is loaded into L1 which might cause replacement/movement of blocks for which appropriate messages are printed.



Outputs for this mapping of the cache structure follow the following format for a cache line:
<tag> *<index >* *<*w_i*.....*wi_{+k}*.....*w_{i+B-1}*>*

3.2 Associative Mapping

In case of associative mapping, the given address is broken down into appropriate tag and block offset bits. The read and write operation then search the cache for a match with given tag in cache L1 followed by L2 (according to the rule in associative mapping) and place the block in the first empty line found otherwise replacement is done. Least Recently Used i.e. LRU scheme is used in case of replacement (time is tracked through global time variable). The exact flow of the program is explained through flowcharts below.

API Documentation

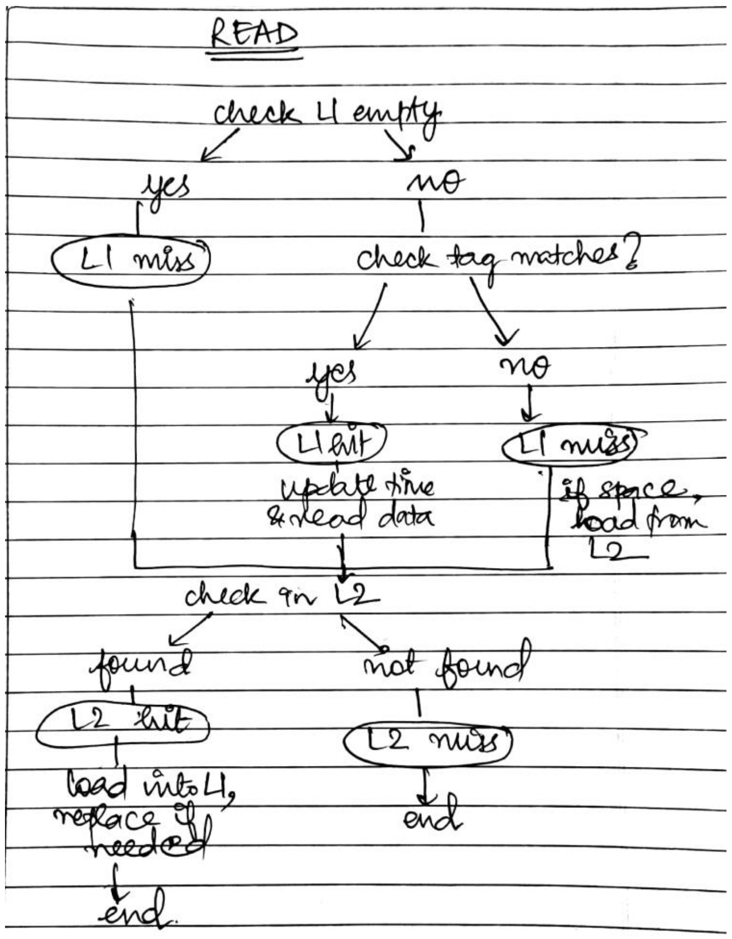
The Associative class in the AditiSejal_2019228_FinalAssignment.java file handles this mapping.

a) `Associative(long mm, long ccll, long bb)`
   The constructor initialises all essential M, CL, S, B etc. values and creates an empty cache structure.

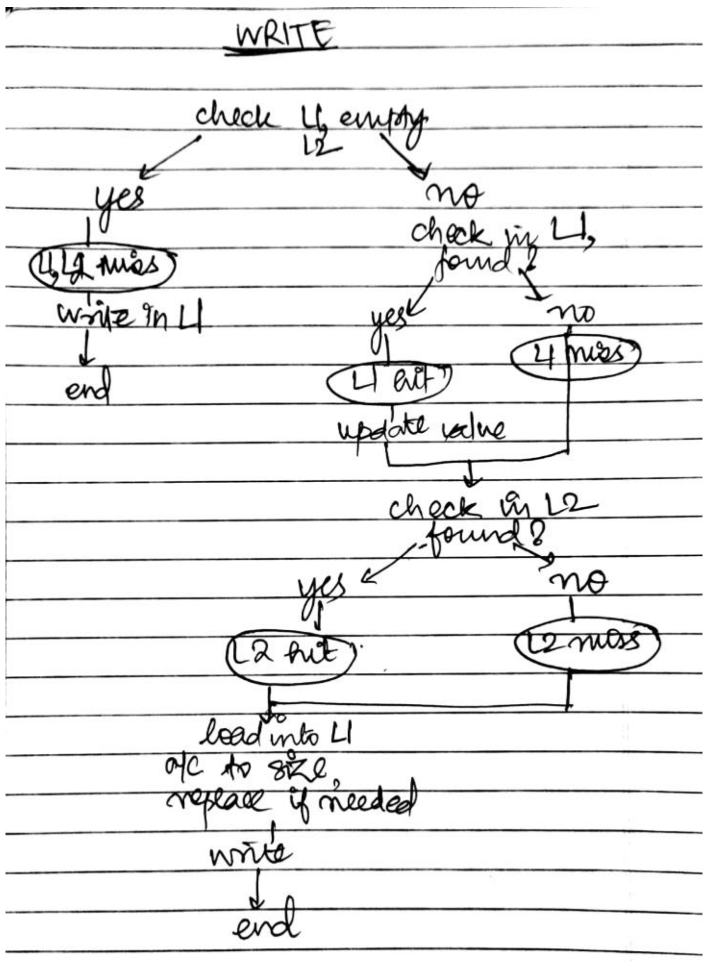
b) `public void l1printer()`
   This function prints the cache structure of L1 cache.

c) `public void l2printer()`
   This function prints the cache structure of L2 cache.

d) `public void reader(String address)`
   This function takes address of word as input and handles the read operation in fully associative memory mapped 2-level cache. It operates according to this flowchart. A read hit means that the block containing the word is already in the cache. If there occurs an L2 hit and a L1 miss, data is loaded into L1 which might cause LRU replacement/movement of blocks for which appropriate messages are printed.



e) `public void writer(String address, int data)`
This function takes address of word and data to be written at that word as input and handles the write operation in direct mapped 2-level cache. It operates according to the flowchart. A write hit means that the block containing the word is already in the cache. If there occurs an L2 hit and a L1 miss, data is loaded into L1 which might cause LRU replacement/movement of blocks for which appropriate messages are printed.



Outputs for this mapping of the cache structure follow the following format for a cache line:
<tag>   < last used time for block >   <w_i.....w_{i+k}.....w_{i+B-1}>

3.3 N-way Set Associative Mapping

In case of n-way set associative memory mapping, the given address is broken down into appropriate tag, set and block offset bits. The read and write operation then search the cache for given tag in specified set (cache lines have been implemented in the program using an array of arraylists) according to the rule in n-way set associative memory mapping.

For the read operation, first the set is found, then an iterative loop over the n lines in that set are read and data is fetched. This is done in L1 first followed by L2 in case of a miss. Replacements are done according to the Least Recently Used (LRU) scheme. Similarly, for the write operation.
For n = 1, the n-way set associative mapping resembles direct mapping.

API Documentation

The SetAssociative class in the AditiSejal_2019228_FinalAssignment.java file handles this mapping.

a) `SetAssociative(long mm, long ccll, long bb)`
The constructor initialises all essential M, CL, S, B etc. values and creates an empty cache structure.

b) `public void l1printer()`
This function prints the cache structure of L1 cache.

c) `public void l2printer()`
This function prints the cache structure of L2 cache.

d) `public void reader(String address)`
This function takes address of word as input and handles the read operation in n-way set associative memory mapped 2-level cache. A read hit means that the block containing the word is already in the cache. The search is first done setwise followed by iterative search over the n lines in the set. If there occurs an L2 hit and a L1 miss, data is loaded into L1 which might cause replacement/movement of blocks for which appropriate messages are printed.

f) `public void writer(String address, int data)`
This function takes address of word and data to be written at that word as input and handles the write operation in n-way set associative memory mapped 2-level cache. A write hit means that the block containing the word is already in the cache. The search is first done setwise followed by iterative search over the n lines in the set. If there occurs an L2 hit and a L1 miss, data is loaded into L1 which might cause LRU replacement/movement of blocks for which appropriate messages are printed.
Outputs for this mapping of the cache structure follow the following format for a cache line:
<tag> *<set>* *<*w_i*.....*wi_{+k}*.....*w_{i+B-1}*>*

API Documentation of common functions for all mapping types

a) `public static int powerofII(long n)`
   This function takes a long integer n as input (a power of 2) and returns its log base 2 i.e. the power to which 2 must be raised to get that n.

b) `public static String binaryconverter(int n, int len)`
   This function takes an integer n and desired length of its binary string as input and returns a string which is the binary form of n upto len bits.

c) `public static int todecimal(String num)`
   This function takes a string of binary form of a number and converts it to decimal form integer and returns this value.

4. <u>Assumptions and Constraints</u>

a) The two levels of the cache (L1 and L2) have been assumed to have sizes S/2 and S or number of cache lines CL/2 and CL respectively.
b) The two levels of the cache (L1 and L2) are assumed to have the same type of mapping (direct, associative or set-associative) for a cache mapping selection.
c) The two levels of the cache (L1 and L2) have been assumed to have the same size of blocks.
d) The two levels of the cache (L1 and L2) follow **exclusion policy** for maintenance of blocks within them. As a result, a block, if present in L1, can not be present in L2 simultaneously and vice-versa.
e) A unified standalone 2-level cache has been made which works without the intervention of main memory.
f) No type of main memory has been maintained. As a result, an address will be in either L1 or L2 only, failing which a miss is shown.
g) The word length has been assumed to be 16 bits.
h) The size of the main memory and size of the block has been asked in units of words following the same approach as in the tutorial for cache memory.
i) The addresses to be entered by the user have been assumed to be in **binary form** and correspond to a **word** in the memory. The memory system can thus be thought of as word addressable memory.
j) A **write allocate scheme with a write-back policy** has been followed in the cache memory system. This means that values are first brought into the cache (here represented by the read/write operations) and then updated. The updates are made in main memory (not maintained in this project) only on replacement.
k) Blocks have been assumed to have a valid bit to indicate the presence of valid data in them. When the valid bit is set to 0, blocks are considered empty while when the valid bit is set to 1, blocks are considered to contain valid information and thus not empty.
l) Blocks have been assumed to have a dirty bit to keep track of value updating of words. Before writing to the main memory, one can thus, interpret by looking at the

dirty bit whether the data in main memory and cache is same (dirty bit set to 0) or changed/updated (dirty bit set to 1) for a given word address.

m) The entire cache structure is printed on the console as output along with messages signifying the operations (move, replace block etc.) done when the number of cache lines in L2 cache are less than or equal to 64. Otherwise only messages regarding the operations done are printed. For error, if invalid choices are entered the program continues asking for another choice while for invalid data the program terminates.

n) For the cases of Associative Memory Mapping and n-way Set Associative Memory Mapping, the **LRU Replacement** Scheme (Least Recently Used) has been followed. The block which was operated on the earliest in the cache is replaced with a newer block.

o) All the write and read operations are done in the higher-level cache i.e. L1 cache and if block is not present in L1 during operation, the block is loaded into L1 from L2 cache (if present in L2). L2 thus is populated in cases of replacement or when L1 cache is full.

p) In the case of n-way Set Associative Memory Mapping, the number of cache lines in L1 must be greater than N. Mathematically**, CL > 2n**. This is because otherwise cache L1 becomes a fully associative memory. The problem here is that 1 is not an appropriate power of 2 and thus the number of sets cannot be decided correctly, giving errors for a 2-level cache.

q) Words are assumed to contain **integer-type data**.

r) The program works for up to 2^{32} words as main memory size.

s) Following the principle of spatial locality of reference, an entire block with all the words it contains is loaded into the cache and treated as full, valid data for all the words in the block. For example, if out of a block containing 4 words, data is written to one word, say the first word, then on loading into cache, the rest of the words in the block are assumed to have 0 as the valid data in them and subsequent reads/writes for these words would lead to a hit.

t) All inputs must be a power of 2 except the data to be written which can be any integer type.

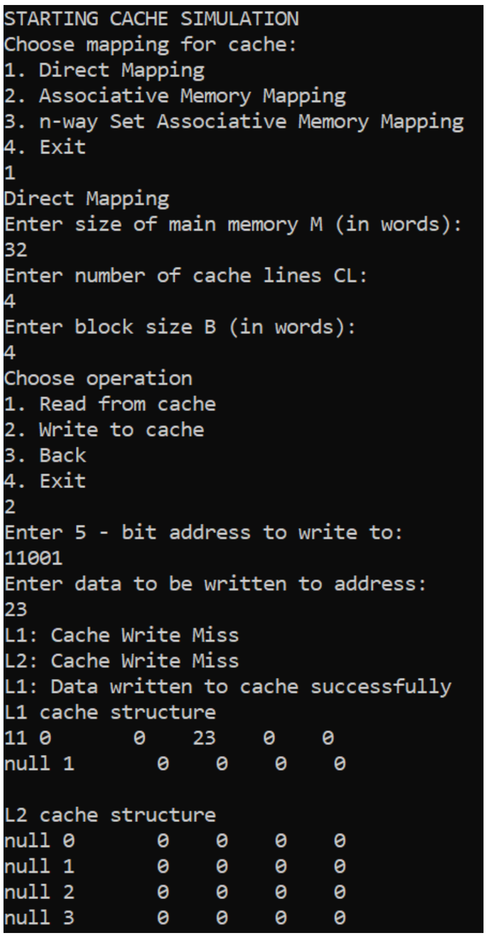
5. Project Execution Instructions

In the directory AditiSejal_2019228_FinalAssignment in command prompt, enter the following commands:

```
javac AditiSejal_2019228_FinalAssignment.java
java AditiSejal_2019228_FinalAssignment
```

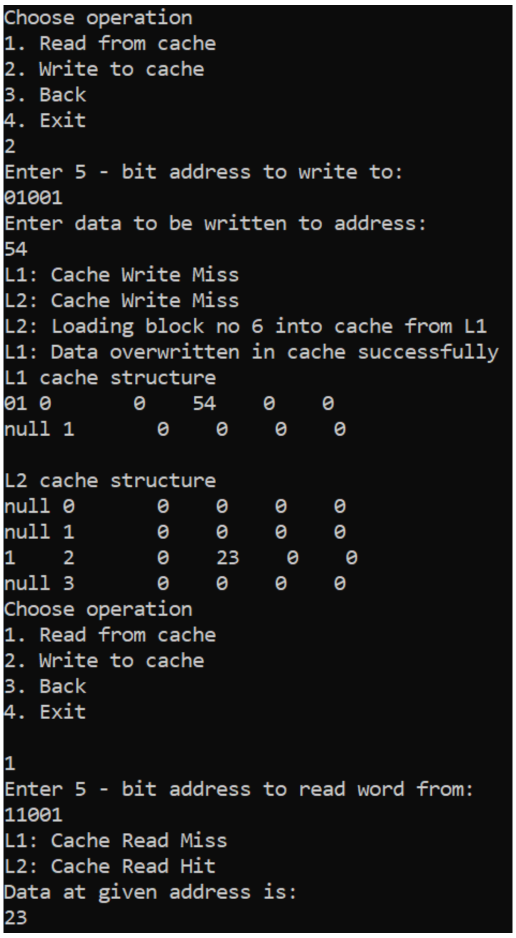
The cache simulation will start with a menu asking for a prompt of the type of mapping to be used for the cache. The user must enter a valid integer choice after selecting the same from the menu. Further prompts for inputs and outputs will guide the user through the program.

6. Project Sample Input, Output



```
STARTING CACHE SIMULATION
Choose mapping for cache:
1. Direct Mapping
2. Associative Memory Mapping
3. n-way Set Associative Memory Mapping
4. Exit
1
Direct Mapping
Enter size of main memory M (in words):
32
Enter number of cache lines CL:
4
Enter block size B (in words):
4
Choose operation
1. Read from cache
2. Write to cache
3. Back
4. Exit
2
Enter 5 - bit address to write to:
11001
Enter data to be written to address:
23
L1: Cache Write Miss
L2: Cache Write Miss
L1: Data written to cache successfully
L1 cache structure
11 0       0    23    0    0
null 1       0    0    0    0

L2 cache structure
null 0       0    0    0    0
null 1       0    0    0    0
null 2       0    0    0    0
null 3       0    0    0    0
```



```
Choose operation
1. Read from cache
2. Write to cache
3. Back
4. Exit
2
Enter 5 - bit address to write to:
01001
Enter data to be written to address:
54
L1: Cache Write Miss
L2: Cache Write Miss
L2: Loading block no 6 into cache from L1
L1: Data overwritten in cache successfully
L1 cache structure
01 0       0    54    0    0
null 1       0    0    0    0

L2 cache structure
null 0       0    0    0    0
null 1       0    0    0    0
1    2       0    23    0    0
null 3       0    0    0    0
Choose operation
1. Read from cache
2. Write to cache
3. Back
4. Exit

1
Enter 5 - bit address to read word from:
11001
L1: Cache Read Miss
L2: Cache Read Hit
Data at given address is:
23
```