# Computer Organization Booth's Implementation Project

*Project Statement* - A cache of size S with CL as the number of cache lines and block size B is to be built. S, CL, and B are in powers of 2. Write a program that allows loading into cache and searching cache using:

1. Direct mapping
2. Associative memory
3. n-way set associative memory where n is a power of 2.

Any programming language (of your choice) can be used.

**What is cache?**

The time it takes to access the main memory is way more than the clock cycle of the CPU. So to reduce the number of times main memory is accessed we introduce RAM and cache in between the CPU and main memory. Cache is high speed memory but it is costly. Hence the size of cache is small in comparison to RAM and Main Memory.
The cache stores copies of the data that is frequently accessed from the Main Memory.

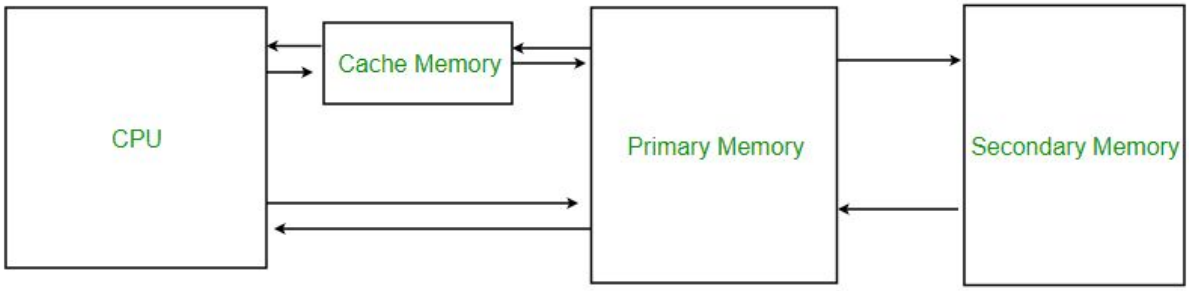


*Image source:* https://media.geeksforgeeks.org/wp-content/uploads/cache.png

There are three types of cache mappings to map address in Cache:

1. **Direct Mapping**
   In this mapping There is one particular cache line for each block of memory. If that cache line already has some data then that data is trashed and the new block is loaded in its place. The address is split into two pieces, there is an index piece and a tag piece. The higher significance bits are assigned as tag bits and lower significance ones are index bits. In these index bits the starting bits determine the cache line number and the end bits determine the word offset or the word in that particular block.

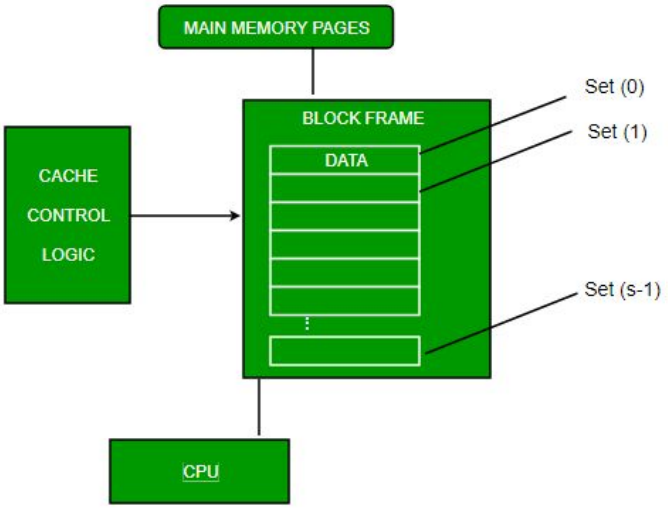


*Image source:* https://media.geeksforgeeks.org/wp-content/uploads/cache1.png

2. **Associative Mapping**
   In associative mapping any block of the main memory can go in any line of the cache. The address is divided in two parts. The startings bits determine tag and the ending bits determine the word offset or the word number in that block. If the cache gets full then lines are replaced according to Least Frequently used, Least recently used, Most Recently used, First in First out etc. In my project I have used the *Least Frequently Used* mechanism to determine the line that should be replaced in cache.

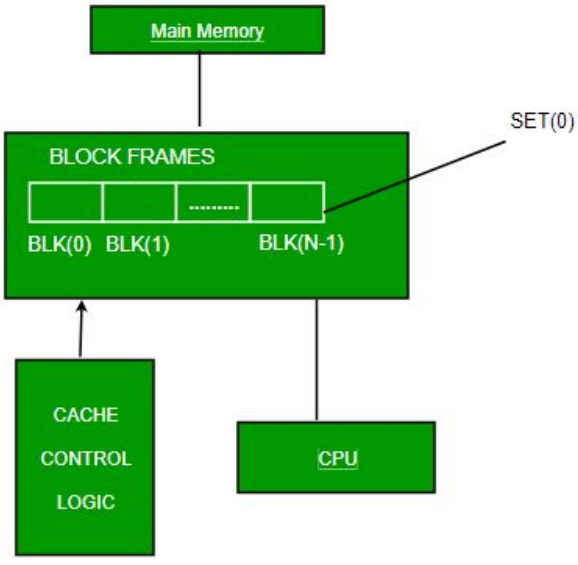


*Image source:* https://media.geeksforgeeks.org/wp-content/uploads/cache2.png

3. **K- Way Set Associative Mapping**
   It combines both direct and associative mapping techniques. The cache is divided into sets each of size K lines. These individual sets have associative mapping within them. The address is divided into three parts, the starting bits form the tag bits, the middle bots form the set index and the last bits form the word offset. When a particular set is filled lines are replaced according to various mechanisms similar to ones in associative mapping. In my project I have used the *Least Frequently Used* mechanism.

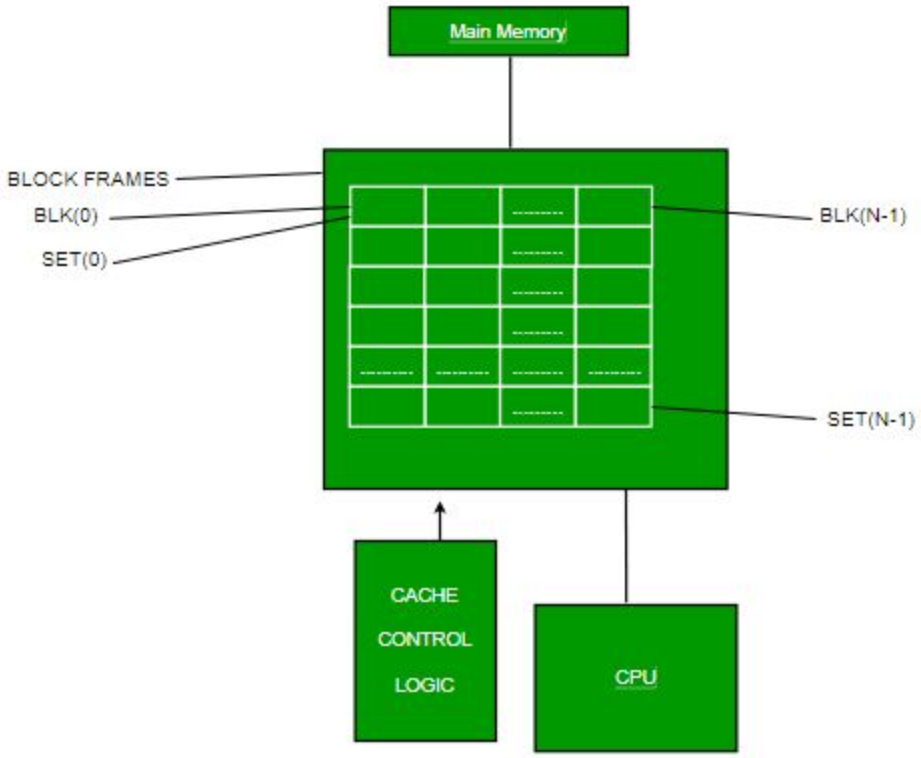


*Image source:* https://media.geeksforgeeks.org/wp-content/uploads/cache4.png

**Prerequisites for running the program**

- Python 3.3 must be present in the user's device
- PrettyTable 0.7.2 must be installed

**Assumptions**

- I have built a stand alone cache without any Main Memory.
- I have not taken cache size as input because cache size can be calculated by number cache lines and size of the block (by multiplying the two)
- I have added an extra input N which is the size of Main memory in powers of 2 and the number of bits there will be in the address

**Description of the Code**

There are Three classes in the code-

1. class DirectMapping
    a. def __init__(self,memorySize, blockOffsetBits, cacheIndexBits,dataType)
    In it I have initialised various parameters of a cache like tag bits and all. And also the build and empty cache which is a 2D list. The first element of the inner list is the tag and the rest of elements are the words or data.
    b. def read(self,address,other=-1)
    It first determines the tag bits, the line index and the word offset from the address and then goes to that particular line and matches the tag. If it's a hit then data is printed otherwise the line is replaced.
    c. def write(self,address,data,other=-1)
    It first determines the tag bits, the line index and the word offset from the address and then goes to that particular line and matches the tag. If it's a hit then data is written otherwise the line is replaced.
    d. def hitOrMiss(self,address)
    It determines through the address whether or not the particular address is present in the cache. It is called by the read and write functions.
    e. def printCache(self)
    It prints the cache in form of a table using pretty table module
    f. def returnBlock(self,address)
    It returns the cache line number the particular block addressed in the address is.

2. class AssociativeMapping
    a. def __init__(self,memorySize, blockOffsetBits, cacheIndexBits,dataType)
    In it I have initialised various parameters of a cache like tag bits and all. And also the build and empty cache which is a dictionary. The key in the dictionary is tag and the value of the dictionary is the list of words. The last element of the list is a counter that counts the number of times that particular line has been accessed to implement the LFR.
    b. def read(self,address,other=-1)
    It first determines the tag bits, the line index and the word offset from the address and then goes to that particular line and matches the tag. If it's a hit then data is printed otherwise the line is replaced by LFR.
    c. def write(self,address,data,other=-1)
    It first determines the tag bits, the line index and the word offset from the address and then goes to that particular line and matches the tag. If it's a hit then data is written otherwise the line is replaced by LFR.
    d. def hitOrMiss(self,address)
    It determines through the address whether or not the particular address is present in the cache. It is called by the read and write functions.
    e. def printCache(self)
    It prints the cache in form of a table using pretty table module

3. class SetAssociativeMapping
    a. def __init__(self,memorySize, blockOffsetBits, cacheIndexBits, kbits,dataType)
    In it I have initialised various parameters of a cache like tag bits and all. And also the build and empty cache which is a list of dictionaries. The elements of lists are various sets the cache has been divided to and the element is a dictionary of all the tags and the lists of worlds as values.
    b. def read(self,address,other=-1)
    It first determines the tag bits, the line index and the word offset from the address and then goes to that particular line and matches the tag. If it's a hit then data is printed otherwise the line is replaced.
    c. def write(self,address,data,other=-1)
    It first determines the tag bits, the line index and the word offset from the address and then goes to that particular line and matches the tag. If it's a hit then data is written otherwise the line is replaced.
    d. def hitOrMiss(self,address)
    It determines through the address whether or not the particular address is present in the cache. It is called by the read and write functions.
    e. def printCache(self)
    It prints the cache in form of a table using pretty table module
    f. def returnSet(self,address)

**Error Handled**

- If anywhere a wrong choice is entered from the ones expected
- If the size of main memory entered is less than the block size
- If cache size is greater than memory
- If cache is less than the block size
- If tag bits come out to be less than number of sets in Set Associative mapping (It can lead to duplicacy in tags in a particular set)
- If K is greater than the number of lines in cache in Set Associative Mapping

## Bonus part of assignment

*Project Statement* - Build a 2 level cache. The size of the level 1 cache is S/2 and the size of level 2 cache is S. All the other parameters are to be decided accordingly. This will also be a standalone cache without the intervention of the main memory.

**Multilevel Cache**
There are various methods of building multi level cache

1. **Inclusive Policy**
   In this policy all the blocks in the higher level cache are also present in the lower level cache. If there is Hit in L1 then data is read from L1 and returned to the processor. If data is not present in L1 but found in L2 then it is placed in L1 from L2. If it causes a block to be replaced in L1 then it is replaced. If data is not present in both L1 and L2 then it is placed in both of them after fetching from main memory. If this causes replacement of data in L2 then it is checked whether that line is present or not in L1 and appropriate action is taken to ensure that the Inclusion policy is not violated.

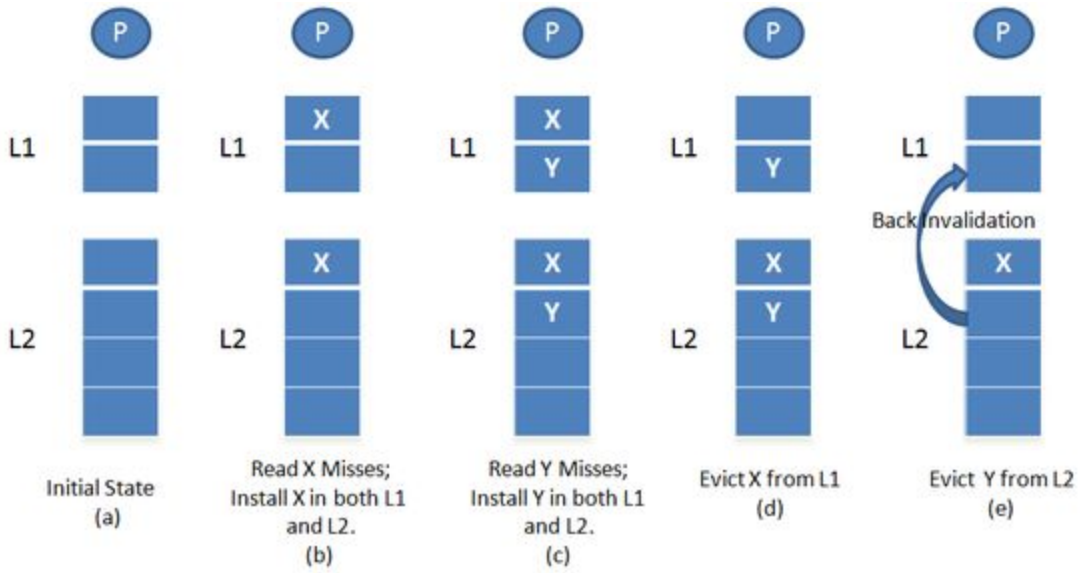


*Image source:* https://upload.wikimedia.org/wikipedia/commons/thumb/b/ba/Inclusive.png/550px-Inclusive.png

2. **Exclusive Policy**
   In this policy none of the blocks present in higher level cache are present in lower level cache. If the block to be read is found in L1 then it is read from there. If it is not found in L1 but is found in L2 then it is moved from L1 to L2 and if that causes a block to be evicted from L1 then that block is placed in L2. If the block is a miss in both L1 and L2 then it is placed in L1 after fetching it from the main memory. A block moves to L2 if it is evicted from L1.

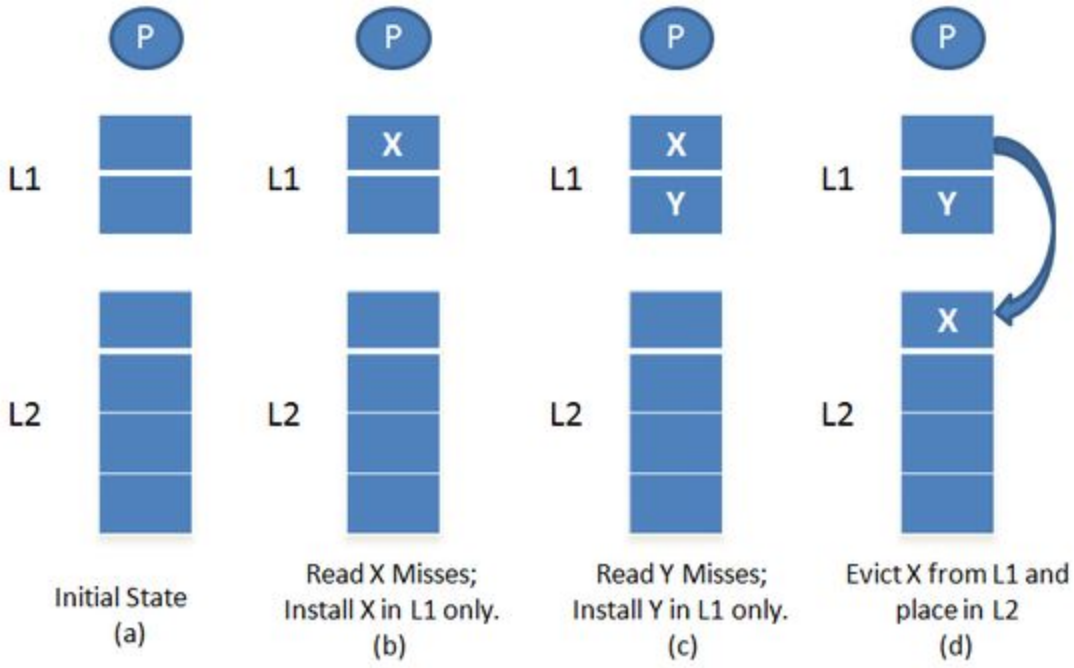


*Image source:https://upload.wikimedia.org/wikipedia/commons/thumb/e/e9/ExclusivePolicy.png/550px-ExclusivePolicy.png*

3. **NINE Policy**

It is neither inclusive nor exclusive of the blocks. The lower level cache may or may not have blocks of higher level. It is similar to Inclusion policy but there is no back validation when a block is evicted from L2. If there is Hit in L1 then data is read from L1 and returned to the processor. If data is not present in L1 but found in L2 then it is placed in L1 from L2. If it causes a block to be replaced in L1 then it is replaced. If data is not present in both L1 and L2 then it is placed in both of them after fetching from main memory.

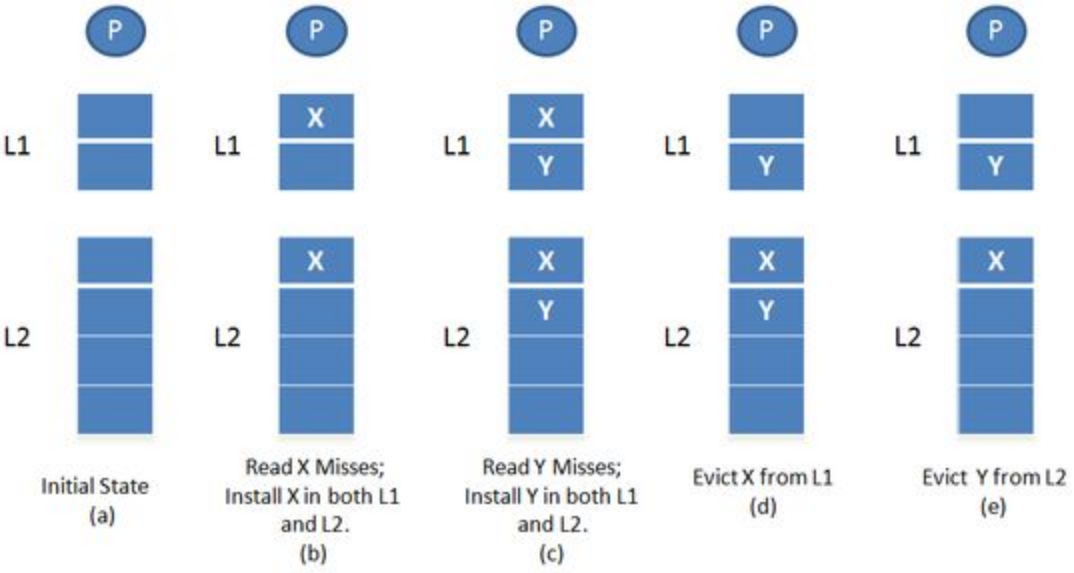


*Image source:* https://upload.wikimedia.org/wikipedia/commons/thumb/e/ec/NINE.png/550px-NINE.png

In my project I have used the *NINE policy* for implementing two level cache.

**Write Through and Write Back in Cache**

Various methods are followed for writing data in memory after writing it in cache so that data is consistent in memory and cache

1. **Write Through**
   In this method data is simultaneously updated in cache and in memory. So when we write new data in cache, we also update it in memory.
2. **Write Back**
   In this method data is only updated in main memory when that particular cache line is to be evicted. There is an extra bit that is stored with the cache line that tells whether or not that line was modified and if it was then its data is replaced in memory at the time of eviction.

In my project I have used the Write *Through* mechanism between the two caches and assumed the same for cache and memory though it does not matter how i write to memory because i am not maintaining one.

**Code**

I modified the code I had used for a single level cache to implement two level cache. I introduced a default parameter in all the read and write functions whose changed value would help me determine whether or not it is being called with a two level cache system. And accordingly I coded special If conditions if it.

**References**


https://www.youtube.com/watch?v=isbdr73Ymug - Understanding Set Associative Mapping - Computer Organization - Gate

https://www.youtube.com/watch?v=Pg6bqkekAXY&list=RDCMUC8FmKkoVFU20P6WnykizlUg&index=1 - Understanding Associative Mapping - Computer Organization - Gate

https://www.youtube.com/watch?v=pSarQQTJbDA&feature=youtu.be - Understanding Direct Mapping - Computer Organization - Gate

https://drive.google.com/file/d/1CXbmQ9lcKt-BbbK3SuS3yQEbBUPsD3MS/view - L11 - Cache and Main Memory

https://www.geeksforgeeks.org/write-through-and-write-back-in-cache/ - Write Through and Write Back in Cache

https://en.m.wikipedia.org/wiki/Cache_inclusion_policy - Inclusive and exclusive caches