***DOCUMENTATION***

***CACHE – PROJECT***

**Introduction to the Objective of the Program:**

The objective of the program is to create a cache structure with **Direct Mapping**, **Associative Mapping** and **Set-Associative Mapping.** The cache Memory is easy to access and is time efficient. So, the computer looks for the address required in the **cache Memory** first and then if the Address is missing then it searches the address in the **Main Memory**. This increases the efficiency and the time to access data decreases.

**Functions used**

1. **to\_bin ():**

This function is used to take hexadecimal addresses in form of string and then return the binary form of the same. It takes two string in form of input one the hexadecimal address in the empty string.

Example if to\_bin (FFFF) is called, it returns 1111111111111111 as an output

1. **to\_dec ():**

This function is used to calculate the decimal form of a string entered in binary form. It takes the binary form a number in input and returns the decimal form of the number.

Example if to\_dec (111) is called it returns 7 as an output.

1. **power\_2 ():**

This function is used to calculate the power of 2 in the particular number. It takes an integer as input and returns the power of 2 in that number.

Example if power\_2 (16) Is called then it would return 4 as an output.

1. **Search ():**

This function takes an array structure and tag bit as an input and return -1 if the tag bit is not present in the array otherwise the line number where it is present.

Some of the inbuilt functions such as **int ()**, **str ()** and functions related to the **Module Math** are used in this program.

**ASSUMPTIONS:**

* It is assumed that the address is of **16 Bit.**
* The input of address of each word in taken in **Hexadecimal Base.**
* All inputs are assumed to be in the **power of two** starting from 2.
* It is assumed that there is **no main memory**.
* It is assumed that data to be stored is always greater than or equal to 0. **Data>=0.**
* 1 word = 1Byte

**INPUT FORMAT**

The input format encounters two types of input:

1. The “**READ**” type of input
2. The “**WRITE**” type of input

**The READ type:**

The read type of statement helps the user to read the contents of a particular address.

This type of statement is followed by the followed by a hexadecimal address representing the address to be read and the give the data (if any data is stored) to the user.

Example:

**READ FFFF**

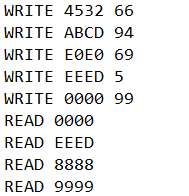
When this statement is encountered by the program, then it returns the data stored at FFFF in the cache (if any).

**The WRITE type:**

The write type of statement helps the user to input the data in cache at the specified address. This type of statement is followed by the address where the data is to be stored and the respective data to be stored. Example:

**WRITE FFFF 120**

When this type of statement is encountered the program reads the address where the data is to be stored and the data. It reaches the location where data is to be placed and stores the data if it is empty, else replaces the data with the earlier stored data.



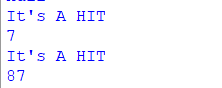
**OUTPUT FORMAT**

The output format consists of mainly three important types of statements:

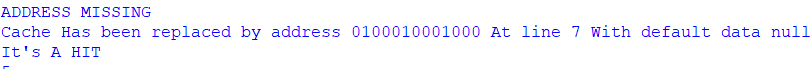
1. **The HIT STATEMENT**
2. **THE MISS STATEMENT**
3. **THE REPLACEMENT STATEMENT**

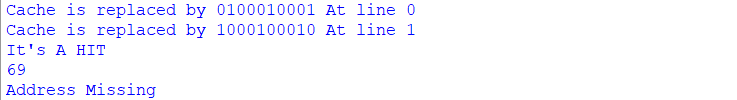
**The Hit Statement:**

This statement can be encountered by the user when the address that is made read by the user is present in the cache and this is followed by the data present in that address.



**The Miss Statement:**

This type of statement is encountered when the address input by the user is not present in present in the cache and this is followed by the **replacement statement** with shows that the structure in cache is being replaced by the **tag\_bit and** the default data which is null. 



The output also show the numbers of tagbits, offsetbits (Word), the number of line bits for each type of Mapping.

**The Working of the Program:**

1. **DIRECT – Mapping**

This mapping is a mapping where each of the block has a unique position that is decided by the tagbit, the line bit and the offset bit of the address of the word. Let us assume that address is of 16 bits then there are 2^16 combinations or Main Memory availabe with us.

The program takes the input of the number of cache lines and size of a block .

Let us suppose that the user inputs 2^p as number of cache lines and 2^x as the size if the block.

The total number of blocks is 2^16/2^x i.e 2^16-x therefore these 2^16-x bits represent the blocknumbers. So out of 16 bits 16-x represents the block number and the rest of the bits i.e the x bits represent the offset(word) bit.

Now 2^16-x/2^p represent the number of blocks in each of the i.e 16-x-p bits represent the number of blocks that can be in a single line. The least significant p bits out of the 16-x bits represents the line bit.

Example 1000100010001000. Lines=2^4, size of each block=2^3

So the bits 16-3= 13 1000100010001 represent the block number

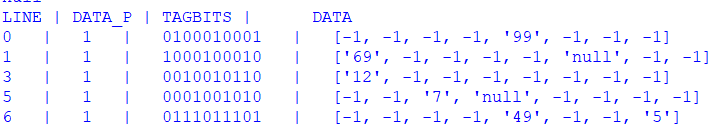
16-3-4=9 100010001 represents the tagbits

Least significant 4 16-3=13 bits represent the line number 0001 and 000 represent the word number.

The program when takes the input from the user, If the statement is “WRITE” , then the program reaches the particular address and then if the tagbit of that location is empty then it stores the data, If not then it replaces the tagbit as well as the data stored in that line and then shows the tagbit by which it is being replaced.

If the statement is a “READ” statement, then it reaches the location i.e th line of that unique block and then it checks the data as well as the tagbits. If the tagbit is empty then it means that the data for that address is not shown and it is a miss. If it is not empty, then it checks both the tagbit and the data of that bit and if either if them is not same then it again shows the missig output.

In each case whenever there is a miss, the program stores the value null in that location along with the tagbit as there is no role of mainmemory in this program and if the same address is read again then it shows the data as null for that address. Then the program finally shows the strucutre with the lines only where the data is present. The Data array shows the history and present of the data present in that line (Pae



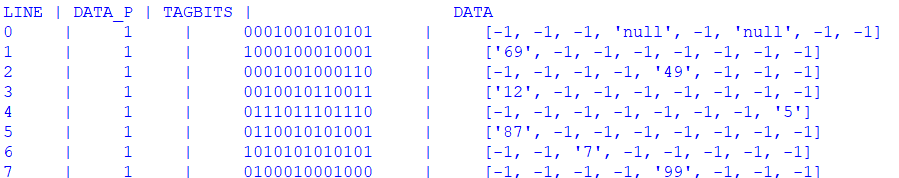
1. **Fully Associative Mapping:**

This type of mapping is based on the assumption that any block number can be placed in any of the lines from the cache. When all the lines of the cache is full then the replacement technique that Is used is FIFO which means that the program will start replacing the lines that came first and continue so after with the same. When the program encounter the

“WRITE” statement , then it follows the same procedure as described in the direct mapping with the difference that there is no particular line for any block and replacement technique followed is FIFO. When “READ” statement is acquired then it follows the same procedure as described in the Direct mapping and If it is a miss then it adds the data with default value null in the cache structure.

The main memory has 2^16 addresses. Let the number of lines be 2^p and the block size be 2^x. Then the number of the blocks is 2^16/2^x i.e 2^16-x.

So this means that the 16-x bits represent the number of blocks and the block number and the x bits represent the offset (Word) bits. The output format show all the lines with data in them along with the tag bit and the line number.



1. **N-Way-set Associative Mapping:**

The type of mapping can be seen as the combination of both the Direct Mapping and Fully Associative Mapping. The number of lines are divided in n set (that User Inputs) and there is a specific block where the set belongs.

Number of lines = Total number of line let’s say 2^X

Number of sets = 2^y

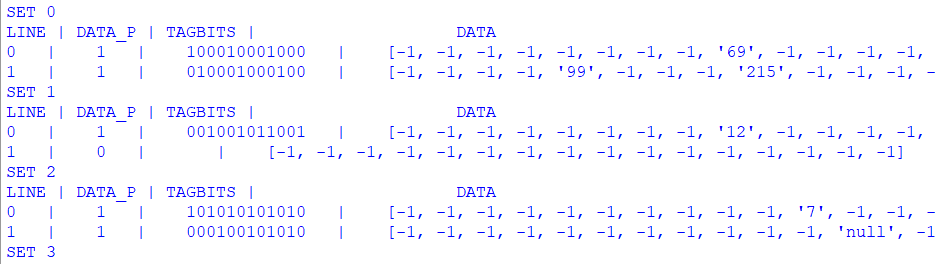
Number of lines in each set = 2^X/2^y i.e 2^X-y id the number of lines in each set.

The set number of any specific block is = block number mod number of set

It can be seen as a combination of both the mapping as within a set , a block can be placed in any of the lines in it which is a property of the Fully Associative Mapping and Direct Mapping as there is a particular set for a particular block number.

Example it is a 2-Way set Associative Mapping with 8 lines and block size 32, then number of line in each set would be 4 and the set of each block would be block number mod 2

Here is the output format of this type of mapping. The output shows the set number and the lines present in each set along with the history and the present data and the tagbits present in the line in each set.



**Error Reporting:**

Following are the errors that can be encountered by the user:

* “NOT A VALID INPUT BLOCK LENGTH CANNOT BE NEAGTIVE”. This Statement can be encountered byb the users when the size of the block enetered is greater than the total Main Memory size.
* “THE INPUTS ARE NOT VALID TAGBITS CANNOT BE NEGATIVE”. This statement can be encountered by the users when the number of blocks to be stored are less than the number of lines present in the cache structure.
* “INVALID INPUT NUMBER OF LINES LESS THAN THE NUMBER OF SETS”. This type of error can be encountered by the user when there are less lines than the number of set as it is not possible to arrange them in the given number of set.
* “INVALID ADDRESS INPUT BITS GREATER THAN 16”. This type of error suggests that the address entered by the user in the file is greater than the 16 bit address(MAX).