

# **DOCUMENTATION**

## **COMPUTER ORGANISATION**

### **END SEM ASSIGNMENT (BONUS TASK)**

#### **TWO LEVEL CACHE IMPLEMENTATION**

**Name : ABHISHEK CHATURVEDI**

**Roll no. : 2019401**

**Group no. : 3**

# **ASSIGNMENT OVERVIEW**

- Implementation of two level cache for direct, fully associative and n way set associative mapping which allows searching and loading data into cache memory.
- Programming language: Python 3
- The algorithm used for replacement is LRU (Least Recently Used).
- The algorithm used for cache inclusion policy is NINE(Neither inclusive nor exclusive)
- **Inputs:**
  - Main memory size
  - No. of cache lines
  - Block size
  - 'n' (only for n way set associative mapping)
  - Read(r) or write(w) operation
  - Address (if read)/ address and data(if write)
- **Outputs:**
  - Both level one and level two cache after each operation.
  - **Write operation:**
    - Address breakdown depending on the type of mapping (block no., set no., line no., tag, word no.)
    - Address found/not found in cache
    - If the address is found then its location in cache(line no., set no. etc.)
    - If the address is not found then location in cache memory where data is to be loaded(line no., set no. etc.) and the block no. which is to be loaded
    - Message that the word no. at the given address is updated with input data
  - **Read operation:**
    - Address breakdown depending on the type of mapping ( i.e. block no., set no., line no., tag, word no.)
    - Cache hit/miss i.e. address found/not found
    - Data/value at the address
    - Location in cache memory where data was found depending on the type of mapping( i.e. line no., set no. etc.)

## **ASSUMPTIONS**

- The main memory size and block size are in terms of number of words they can store.
- Number of cache lines in input is for level one cache.
- The block size of first and second level cache are assumed to be equal.
- The type of mapping for the level one and level two cache is assumed to be the same.
- The value of 'n' in n way set associative mapping is assumed to be equal for both the levels of cache.
- Word size length is assumed to be 32 bits (1Word=4Bytes) but the program will run successfully for even 16 bits or 64 bits and more.
- The main memory size, block size, number of cache lines and 'n' are all assumed in powers of 2, i.e. the minimum value of these variables can be 2.
- Size of level two cache is twice the size of level one cache.
- Initially all the values in tag and cache lines are initialised with "NULL" string.

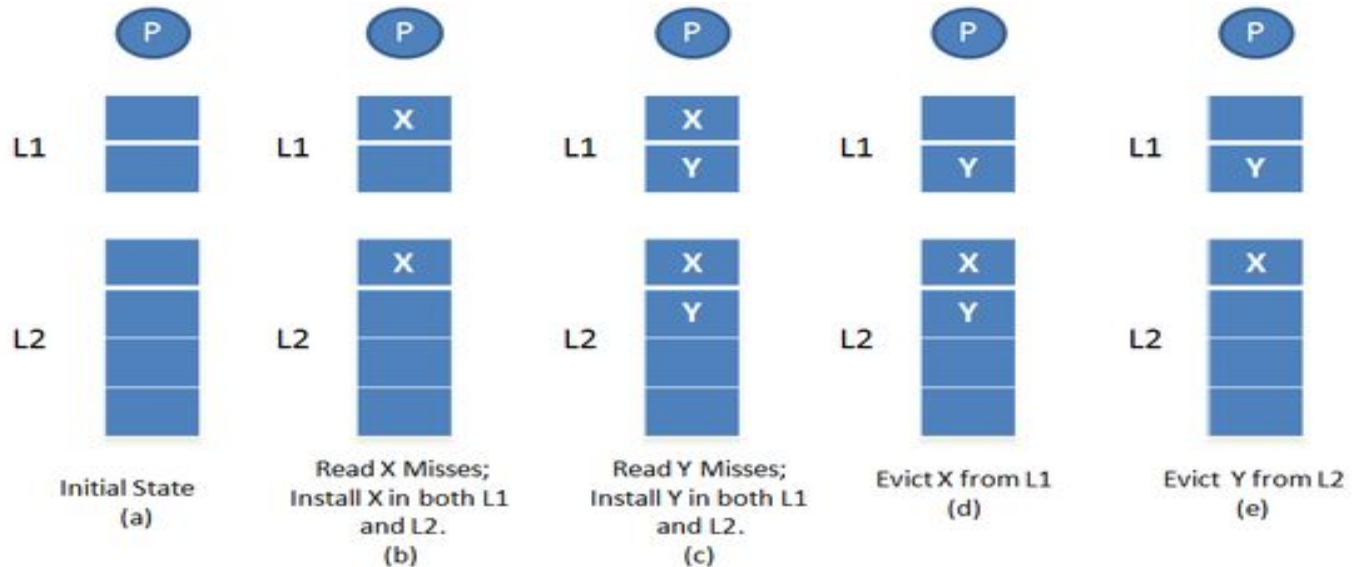
## **ERROR HANDLING**

- If the length of address is not equal to the number of bits required to express the main memory size in power of 2 then the program gives output "INVALID ADDRESS" and asks for input again.
- The size of either of the caches cannot be equal to or greater than the main memory size, if this is not the case then the program gives inbuilt python error.

## **NINE INCLUSION POLICY**

- In this type of policy if there is a write operation then the block is searched for in level one and level two cache and if the block is found in both the levels then the given data is updated at the address in both the cache levels.
- If the block is present in level one cache only then data at the given address is written in level one cache only and the block is not loaded in level two cache.
- If the block is not found in level one cache but it is present in level two cache then that block is loaded in level one cache from level two cache before the data is written at the address in both the cache levels.
- If the block is not found in any level of cache then the block is placed in both cache levels and then the data is updated at the address.

- In this policy there is no connection between the two levels of cache at the time of eviction of a block i.e, the removal of a block in one cache level is independent of the other.
- If a block is to be evicted from second level then there is no back invalidation like in case of inclusive policy i.e, a block is removed from level two cache only even if that block is present in the first level cache.

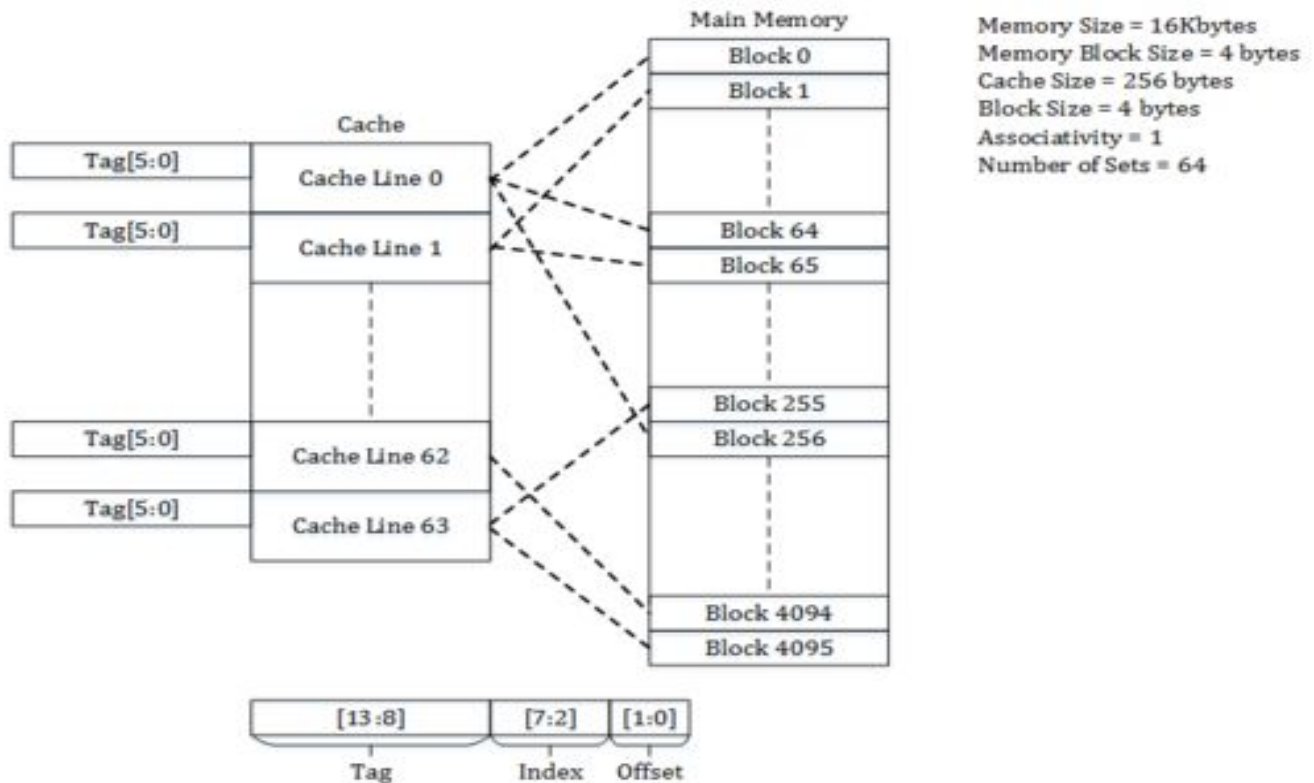


## **DIRECT MAPPING**

### **CONCEPT:**

- In this type of mapping each block of main memory maps into only one specific cache line in the cache memory.
- The physical address is split up into three parts i.e, the block offset(same for both cache), the line no.(different for two levels of cache) and the tag(different for two levels of cache).
- The block offset is the index of the word in the block.
- The line no. bits and tag bits make up the block no. bits and this block no. is used to determine the line no. where this block will be placed in cache memory.
- The line no. is equal to (block no.(in decimal)) modulo (no. of cache lines) i.e, the remainder when block no. is divided by the no. of cache lines.
- Firstly, the address' tag bits are compared to the tag bits at the cache line number in level one cache.
- If the two tags match then it is a hit and data is returned.

- If tag bits don't match in level one tags then level two tag bits are compared to address tag bits and if the tag bits match it is a hit otherwise miss.
- If it is a hit in level two cache then the block is loaded in level one cache also before the data at the address is returned.
- If it is a miss then the block is brought from main memory and loaded in both level one and level two cache.
- If it is a miss and the cache line is already occupied by another block then that block is evicted out and the current block is placed in the cache line.
- If it is a miss and the cache line is empty then the block is simply added to the cache line.



## **ADVANTAGES**

- This mapping is faster since tags are compared at a particular line no. in cache and there is no need to search through the whole cache.
- The replacement policy is simple and easy to implement.
- Cheap hardware implementation.

## **DISADVANTAGES**

- Low cache hit rate.

## **WORKING OF THE PROGRAM**

- The program divides the physical address into tag, line no. and block offset.
- Firstly, the program simply compares the tag of the given address and the tag of the line no. where block is to be placed in level one cache.
- If the tag bits match and it is a write operation then data is written at the address and then the tag bits of line no. in level two cache is compared if the block is present then value is also written at the address in level two cache.
- If the tag bits match in level one cache then data at the address is returned.
- If the tag bits do not match in level one cache then level two cache's line no. tag bits are compared and if the tag bits match and it is write operation then the block is loaded in level one cache also and data at the address is updated in both the levels of cache.
- If tag bits match and it is read operation data at the block offset in the block is returned.
- If the tag bits don't match even in level two cache and it is a write operation then the block is loaded in both levels of cache and then data at the address is written.

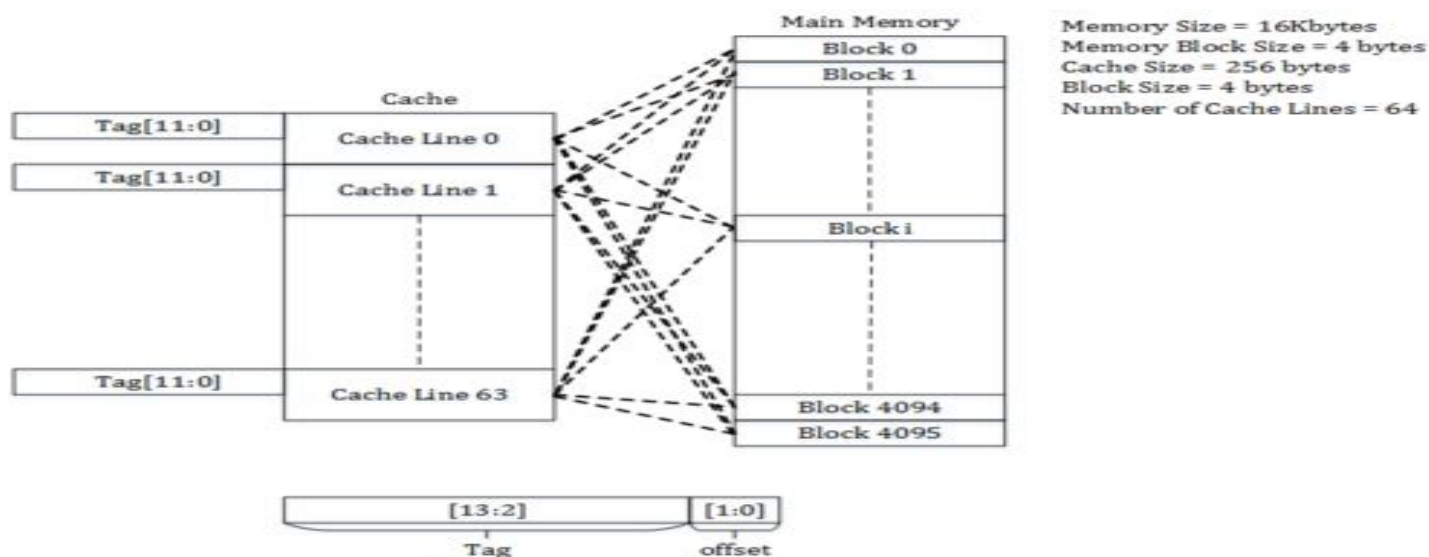
## **ANALYSIS OF THE CODE**

- Two arrays(list in python) are used as data structures for storing tag values and blocks separately.
- Since the program only compares the tags at a given line no. it takes constant time i.e,  $O(1)$ .
- The program also removes and places a new block and writes an address which is done in constant time  $O(1)$ .
- Therefore, the overall time complexity of the program is  $O(1)$ .
- The space complexity of the program is  $O(n)$  since tag array and cache memory are arrays of size  $n$ . ( $n$ =no. of cache lines in level two cache)

# **FULLY ASSOCIATIVE MAPPING**

## **CONCEPT:**

- In this type of mapping a block from main memory can be mapped to any line in cache memory given that the line is free.
- The physical address is split into only two parts i.e, the tag or block no. and block offset.
- The block offset is the index of the word in the block.
- The tag of the given address is compared to the tag of every cache line in level one cache one by one until the tags match or all the cache line's tag is compared.
- If there is a match of tags then it is a hit and the data at the block offset is returned.
- If the tags do not match then the address tag is compared to the tag of every line in level two cache.
- If the tags do not match in even level two cache then the block is placed in both level one and level two caches and then the data at the block offset in the block is returned.
- If the tags match in level two cache then the block is loaded in level one cache also before the data at the address is returned.
- If there is no matching of tags and there is an empty cache line then the block is simply added to cache memory.
- If there is no matching of tags and all the cache lines are occupied then a block is evicted from the cache memory and a new block is placed in the cache memory.
- The block which is to be evicted is decided on the basis of replacement policy (LRU in this case).



## **ADVANTAGES**

- Better hit rate.
- Since there is full flexibility of storing a block in any cache line it ensures complete utilisation of cache.
- A wide variety of replacement algorithms can be used for replacement.

## **DISADVANTAGES**

- Since we need to compare the tag of each cache line the search and placement policy is slow.
- Expensive hardware implementation since a lot of comparisons are involved.

## **WORKING OF THE CODE**

- The program firstly divides the physical address into two parts i.e, block offset and tag/block no.
- Then the program iterates over all the cache lines in the level one cache and compares the tag bits.
- If the tag bits of address and tag bits of a cache line match it is a hit and the data at the block offset in the block is returned in read operation and in write the given data is written at the block offset in the block.
- If the tag bits match in level one cache and it is write operation then the tag bits in level two cache are compared to address tag bits and if the tag bits match then data is updated at the address in level two cache also.
- If the tag bits do not match in level one cache then the tag bits in level two cache are compared.
- If tag bits in level two cache match then the block is loaded in level one cache and then data is written at the address in both the levels of cache if it is write operation.
- If the tag bits do not match even in level two cache and it is a write input then the block is uploaded in both level one and level two cache and then data is updated at the address.
- If it is a miss and it is write operation and all the cache lines are filled then a block is evicted from the cache memory based on LRU replacement algorithm and the new block is placed in the cache memory and the given data is written at block offset of the block.

## **ANALYSIS OF CODE**

- The data structures used are a dictionary(hashmap in other common languages) and a queue implemented using a doubly linked list.
- The queue is used for maintaining the order for replacement in LRU algorithm. It has most recently used blocks in the front and least recently used block in the rear.

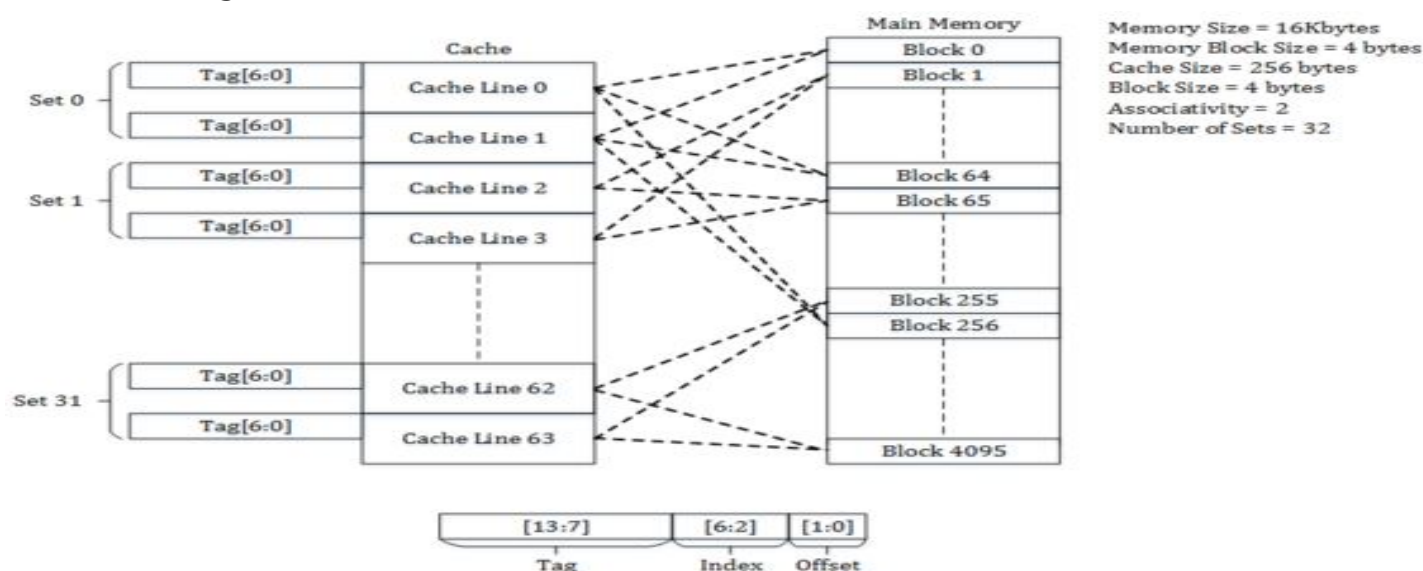


- Since the program looks up for a block no. in hashmap/dictionary it is done in constant time of  $O(1)$ .
- Also deleting/removing and adding/replacing a block in cache memory takes constant time  $O(1)$ .
- Therefore, the overall time complexity of the program is  $O(1)$ .
- The space complexity of the program is  $O(n)$  because there can be  $n$  blocks in the cache memory. ( $n$ =no of cache lines in level two cache)

## N WAY SET ASSOCIATIVE MAPPING

### CONCEPT

- This type of mapping is a hybrid of fully associative mapping and direct mapping.
- In this type of mapping the level one and level two caches are divided into sets each containing  $n$  cache lines.
- Therefore, the number of sets in each cache level is equal to quotient when (number of cache lines in that cache) is divided by ( $n$ ).
- A block from the main memory can map in a specific set only but at any line in that set given that the line is free.
- The physical address is divided into three parts namely tag, set number and block offset.
- The block offset is the index of the word in the block.
- The tag bits of the address are compared to tag bits to each cache line in the set (derived from set bits in the address) until there is a match of tag bits or every cache line's tag is compared.
- If the tag bits match it is a hit otherwise miss.



## **ADVANTAGES**

- There is a wide variety of replacement policies which could be used.

## **DISADVANTAGES**

- It does not utilise all the available cache lines so the cache misses are more.

## **WORKING OF THE CODE**

- The program firstly divides the physical address into three parts i.e, block offset, set number(different for the two cache levels) and tag(different for two levels of cache).
- Then the program iterates over all the cache lines in the set(derived from the address) in the level one cache and compares the tag bits.
- If the tag bits of address and tag bits of a cache line match in level one cache it is a hit and the data at the block offset in the block is returned in read operation and in write the given data is written at the block offset in the block.
- If tag bits do not match in level one cache then the tag bits of cache in lines of the set of level two cache are compared.
- If the tag bits match and it is a write operation then the block is loaded into level one cache and then the given data is written at the address in both level one and level two cache.
- If the tag bits do not match then it is a miss and in write operation the block is placed in both first and second level cache and then data is updated at the given address in both the cache.
- If tag bits do not match and it is a write operation and all the cache lines in the set are filled then a block is evicted from the set based on LRU replacement algorithm and the new block is placed in the cache memory and the given data is written at block offset of the block.
- If it is a miss and it is a write operation and all the cache lines in the set are not filled then the block is simply added to the set and the given data is written at block offset of the block.

## **ANALYSIS OF CODE**

- Four arrays(list in python) are used as data structures for storing tag values and blocks separately for level one and level two cache.
- Since the program only compares all the cache line's tags in a given set of both the cache levels it takes linear\*linear=square time i.e,  $O(n^2)$ . (n=no. of cache lines in the set)
- The program also removes and places a new block and writes an address which is done in constant time  $O(1)$ .
- Therefore, the overall time complexity of the program is  $O(n^2)$ .

- The space complexity of the program is  $O(n)$  since tag array and cache memory are arrays of size  $n$ . ( $n$ =no. of cache lines)

## **REFERENCES:**

- [https://en.wikipedia.org/wiki/Cache\\_placement\\_policies](https://en.wikipedia.org/wiki/Cache_placement_policies)
- [https://en.wikipedia.org/wiki/Cache\\_inclusion\\_policy#NINE\\_Policy](https://en.wikipedia.org/wiki/Cache_inclusion_policy#NINE_Policy)
- [https://en.wikipedia.org/wiki/Cache\\_replacement\\_policies#Least\\_recently\\_used\\_\(LRU\)](https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_recently_used_(LRU))

**\*\*\*\*\* (screenshots of the codes is attached below) \*\*\*\*\***

```

1 #TDM
2 print()
3 print("                TWO LEVEL CACHE // DIRECT MAPPING IMPLEMENTATION                ")
4 print()
5 print()
6
7 def L1disp():
8     print("-----LEVEL --1--CACHE--MEMORY-----")
9     print()
10    print()
11    print("    LINE NO.          TAG          DATA")
12    print()
13    for i in range(CL):
14        print(" "+str(i)+"          ",end="")
15        print(*tag1[i],end="")
16        print("          ",end="")
17        for j in range(B):
18            print(l1[i][j],end=" ")
19        print()
20
21 def L2disp():
22     print("-----LEVEL --2--CACHE--MEMORY-----")
23     print()
24     print()
25     print("    LINE NO.          TAG          DATA")
26     print()
27     for i in range(2*CL):
28         print(" "+str(i)+"          ",end="")
29         print(*tag2[i],end="")
30         print("          ",end="")
31         for j in range(B):
32             print(l2[i][j],end=" ")
33         print()
34
35 print()
36
37 N=int(input("Enter main memory size: "))
38 print()
39 CL=int(input("Enter number of cache lines: "))
40 print()
41 B=int(input("Enter block size: "))
42 print()
43

```

```

47 L2=[]
48 tag1=[]
49 tag2=[]
50
51 for i in range(2*CL):
52     if i<CL:
53         L1.append(["NULL"]*B)
54         tag1.append(["NULL"])
55         L2.append(["NULL"]*B)
56         tag2.append(["NULL"])
57     else:
58         L2.append(["NULL"]*B)
59         tag2.append(["NULL"])
60
61
62 def f():
63
64     inptype=input("write(w)/read(r): ")
65
66     print()
67
68     if inptype=='r':
69         add=input("Enter address in binary form: ")
70
71     else:
72         add=input("Enter address in binary form: ")
73         dataa=input("Enter data: ")
74
75     print()
76
77     if(len(add)!=len(bin(N)[2:])-1):
78         print("INVALID ADDRESS!")
79
80     else:
81
82         print("ADDRESS BREAKDOWN: ")
83         print()
84
85         word_no=int(add,2)
86         print("Word No.: "+add+" (" +str(word_no)+")")
87
88         block_offset=add[len(bin(N)[2:])-len(bin(B)[2:]):]
89         print("Block Offset: "+block_offset+" (" +str(int(block_offset,2))+"")

```

```

113     if tag2[line_index2]==[] or tag2[line_index2][0]!=tag_no2:
114
115         print("ADDRESS NOT FOUND IN CACHE")
116         print()
117         print("LOADING DATA IN CACHE MEMORY")
118         print()
119         print("REPLACING DATA IN CACHE MEMORY AT LINE NO. "+str(line_index1)+" IN L1")
120         print()
121         print("REPLACING DATA IN CACHE MEMORY AT LINE NO. "+str(line_index2)+" IN L2")
122         print()
123         print("LOADING BLOCK NO. "+str(int(block_no,2))+" IN L1 AND L2")
124         print()
125         print("DATA LOADED IN CACHE MEMORY")
126         print()
127         print("REPLACED ADDRESS IN L1: ",end="")
128
129         if tag1[line_index1]=="NULL":
130             print("Line was empty, no block found!")
131         else :
132             print("BLOCK NO. "+tag1[line_index1][0]+line_no1)
133
134         print("REPLACED ADDRESS IN L2: ",end="")
135
136         if tag2[line_index2]=="NULL":
137             print("Line was empty, no block found!")
138         else :
139             print("BLOCK NO.: "+tag2[line_index2][0]+line_no2)
140
141         tag1[line_index1]=[tag_no1]
142         tag2[line_index2]=[tag_no2]
143         L1[line_index1]="NULL"*B
144         L2[line_index2]="NULL"*B
145         L1[line_index1][int(block_offset,2)]=dataa
146         L2[line_index2][int(block_offset,2)]=dataa
147
148         print()
149         print("DATA LOADED IN L1 AND L2")
150         print()
151         print("UPDATING THE GIVEN ADDRESS WITH GIVEN VALUE IN CACHE MEMORY")
152         print()
153         print("WORD NO. "+str(word_no)+" UPDATED IN L1 AND L2")
154

```

```

209     print()
210     print("WORD NO. "+str(word_no)+" UPDATED IN L2 ALSO")
211     L2[line_index2][int(block_offset,2)]=dataa
212
213     else:
214
215         if (tag1[line_index1]==[] or tag1[line_index1][0]!=tag_no1):
216
217             if tag2[line_index2]==[] or tag2[line_index2][0]!=tag_no2:
218
219                 print("CACHE MISS!!!")
220                 print("ADDRESS NOT FOUND IN CACHE MEMORY!")
221
222             else:
223
224                 print("CACHE HIT!!!")
225                 print()
226                 print("ADDRESS FOUND IN CACHE IN L2 AT LINE NO. "+str(line_index2))
227                 print()
228                 print("DATA: ",end="")
229                 print(L2[line_index2][int(block_offset,2)])
230
231             else:
232
233                 print("CACHE HIT!!!")
234                 print()
235                 print("ADDRESS FOUND IN L1 AT LINE NO. "+str(line_index1))
236                 print()
237                 print("DATA: ",end="")
238                 print(L1[line_index1][int(block_offset,2)])
239
240     print()
241     l1disp()
242     print()
243     l2disp()
244     print()
245
246     cont='y'
247     while cont=='y':
248         f()
249         cont=input("continue? (y/n) ")
250         print()

```



```

2 print()
3 print("          TWO LEVEL CACHE // FULLY ASSOCIATIVE MAPPING IMPLEMENTATION")
4 print()
5 print()
6
7 class BlockAddress:
8
9     def __init__(self, blockno):
10         self.blockno = blockno
11         self.next = None
12         self.prev = None
13
14 class CacheList:
15
16     def __init__(self):
17         self.head = None
18
19     def push(self, new_data):
20         new_node = BlockAddress(new_data)
21         new_node.next = self.head
22
23         if self.head is not None:
24             self.head.prev = new_node
25         self.head = new_node
26
27     def lastNode(self, blockaddress):
28         while(blockaddress.next is not None):
29             blockaddress = blockaddress.next
30         return blockaddress
31
32     def getNode(self, blockaddress, x):
33         while(blockaddress.blockno != x):
34             blockaddress = blockaddress.next
35         return blockaddress
36
37     def deleteNode(self, dele):
38         if self.head is None or dele is None:
39             return
40         if self.head == dele:
41             self.head = dele.next
42         if dele.next is not None:
43             dele.next.prev = dele.prev
44         if dele.prev is not None:

```



```

66     block_address1.append("NULL")
67
68 def L1disp():
69     print("-----LEVEL-1--CACHE--MEMORY-----")
70     print()
71     print()
72     print(" LINE NO.          TAG          DATA")
73     print()
74     for i in range(CL):
75         print(" "+str(i)+"          ",end="")
76         print(block_address1[i],end="")
77         print("          ",end="")
78         if block_address1[i] in BA1:
79             for j in range(B):
80                 print(BA1[block_address1[i]][j][0],end=" ")
81         else:
82             for j in range(B):
83                 print("NULL",end=" ")
84         print()
85
86 def L2disp():
87     print("-----LEVEL-2--CACHE--MEMORY-----")
88     print()
89     print()
90     print(" LINE NO.          TAG          DATA")
91     print()
92     for i in range(2*CL):
93         print(" "+str(i)+"          ",end="")
94         print(block_address2[i],end="")
95         print("          ",end="")
96         if block_address2[i] in BA2:
97             for j in range(B):
98                 print(BA2[block_address2[i]][j][0],end=" ")
99         else:
100             for j in range(B):
101                 print("NULL",end=" ")
102         print()
103
104 def f():
105
106     inptype=input("write(w)/read(r): ")
107
108     print()

```

```

103
104 def f():
105
106     inptype=input("write(w)/read(r): ")
107
108     print()
109
110     if inptype=='r':
111         add=input("Enter address in binary form: ")
112
113     else:
114         add=input("Enter address in binary form: ")
115         dataa=input("Enter data: ")
116
117     print()
118
119     if(len(add)!=len(bin(N)[2:])-1):
120         print("Invalid address")
121
122     else:
123
124         print("ADDRESS BREAKDOWN: ")
125         print()
126
127         word_no=int(add,2)
128         print("Word No.: "+str(add)+" (" +str(word_no)+)")")
129
130         block_offset=add[len(bin(N)[2:])-len(bin(B)[2:]):]
131         print("Block Offset: "+str(block_offset)+" (" +str(int(block_offset,2))+)")")
132
133         block_no=add[:len(bin(N)[2:])-len(bin(B)[2:])]
134         print("Tag: "+str(block_no)+" (" +str(int(block_no,2))+)")")
135
136         print()
137
138         if block_no in BA1 and inptype=='r':
139
140             print("CACHE HIT!!! ADDRESS FOUND")
141             print()
142             print("LOADING DATA FROM L1")
143             print()
144             print("LOADING DATA FROM LINE NO. "+str(block_address1.index(block_no))+ " IN L1")
145             print()

```

```

160     print()
161     BA1[block_no][int(block_offset,2)]=[dataa]
162     print("WORD NO. "+str(word_no)+" UPDATED IN L1")
163     address_list1.deleteNode(address_list1.getnode(address_list1.head,block_no))
164     address_list1.push(block_no)
165
166     if block_no in BA2:
167
168         print("ADDRESS FOUND IN CACHE IN L2 ALSO")
169         print()
170         print("ADDRESS FOUND AT LINE NO. "+str(block_address2.index(block_no))+" IN L2")
171         print()
172         print("UPDATING THE GIVEN ADDRESS WITH GIVEN VALUE IN CACHE MEMORY")
173         print()
174         BA2[block_no][int(block_offset,2)]=[dataa]
175         print("WORD NO. "+str(word_no)+" UPDATED IN L2")
176         address_list2.deleteNode(address_list2.getnode(address_list2.head,block_no))
177         address_list2.push(block_no)
178
179     elif block_no in BA2 and block_no not in BA1 and inptype=='w':
180
181         print("ADDRESS FOUND IN L2")
182         print()
183         print("LOADING DATA FROM L2 INTO L1")
184         print()
185         print("LOADING DATA FROM LINE NO. "+str(block_address2.index(block_no))+" IN L2")
186         print()
187         print("REPLACING DATA IN L1 AT LINE NO. ",end="")
188
189         if len(BA1)<CL:
190             print(len(BA1))
191             block_address1[len(BA1)]=block_no
192         else:
193             print(block_address1.index(address_list1.lastNode(address_list1.head).blockno))
194             block_address1[block_address1.index(address_list1.lastNode(address_list1.head).blockno)]=block_no
195             print()
196             print("LOADING BLOCK NO. "+str(int(block_no,2))+" IN L1")
197             print()
198
199             print("REPLACED ADDRESS IN L1: ",end="")
200
201         if len(BA1)<CL:
202

```

```

283         BA1[block_no]=["NULL"]*B
284         address_list1.push(block_no)
285
286         print("REPLACED ADDRESS IN L2: ",end="")
287
288         if len(BA2)<2*CL:
289
290             BA2[block_no]=["NULL"]*B
291             print("Line was empty, no block found!")
292             address_list2.push(block_no)
293
294         else:
295
296             print("BLOCK NO.: "+address_list2.lastNode(address_list2.head).blockno)
297             del BA2[address_list2.lastNode(address_list2.head).blockno]
298             address_list2.deleteNode(address_list2.lastNode(address_list2.head))
299             BA2[block_no]=["NULL"]*B
300             address_list2.push(block_no)
301
302         print()
303         print("DATA LOADED IN L1 AND L2")
304         print()
305         print("UPDATING THE GIVEN ADDRESS WITH GIVEN VALUE IN CACHE MEMORY")
306         print()
307         print("WORD NO. "+str(word_no)+" UPDATED WITH GIVEN VALUE IN L1 AND L2")
308         BA1[block_no][int(block_offset,2)]=[dataaa]
309         BA2[block_no][int(block_offset,2)]=[dataaa]
310
311         else:
312             print("CACHE MISS!!!")
313             print("ADDRESS NOT IN CACHE MEMORY")
314
315         print()
316         l1disp()
317         print()
318         l2disp()
319
320     cont='y'
321     while cont=='y':
322         f()
323         print()
324         cont=input("continue? (y/n) ")
325         print()

```

```

2 print()
3 print("                TWO LEVEL CACHE // n WAY SET ASSOCIATIVE MAPPING IMPLEMENTATION                ")
4 print()
5 print()
6
7 N=int(input("Enter main memory size: "))
8 print()
9 CL=int(input("Enter number of cache lines: "))
10 print()
11 B=int(input("Enter block size: "))
12 print()
13 n=int(input("Enter 'n': "))
14 print()
15
16 L1=[]
17 L1temp=[]
18 L2=[]
19 L2temp=[]
20 tag1=[]
21 tag1temp=[]
22 tag2=[]
23 tag2temp=[]
24
25 for i in range((2*CL)//n):
26     if i<CL//n:
27         L1.append([])
28         L1temp.append([])
29         tag1.append([])
30         tag1temp.append([])
31         L2.append([])
32         L2temp.append([])
33         tag2.append([])
34         tag2temp.append([])
35     else:
36         L2.append([])
37         L2temp.append([])
38         tag2.append([])
39         tag2temp.append([])
40
41 for i in range(len(L2)):      # Initialising each line in each set with empty adress in L2
42     for j in range(n):
43         L2[i].append(["NULL"]*B)
44         L2temp[i].append(["NULL"]*B)

```



```

85
86 def f():
87
88     inptype=input("write(w)/read(r): ")
89
90     print()
91
92     if inptype=='r':
93         add=input("Enter address in binary form: ")
94
95     else:
96         add=input("Enter address in binary form: ")
97         dataa=input("Enter data: ")
98
99     print()
100
101     if(len(add)!=len(bin(N)[2:])-1):
102         print("INVALID ADDRESS!")
103
104     else:
105
106         print("ADDRESS BREAKDOWN: ")
107         print()
108
109         word_no=int(add,2)
110         print("Word No.: "+str(add)+" (" +str(word_no)+")")
111
112         block_offset=add[len(bin(N)[2:])-len(bin(B)[2:]):]
113         print("Block Offset: "+str(block_offset)+" (" +str(int(block_offset,2))+")")
114
115         block_no=add[:len(bin(N)[2:])-len(bin(B)[2:])]
116         print("Block no.: "+str(block_no)+" (" +str(int(block_no,2))+")")
117
118         set_no1=block_no[len(block_no)-len(bin(CL//n)[2:])+1:]
119         print("Set 1 no.: "+str(set_no1)+" (" +str(int(set_no1,2))+")")
120
121         set_no2=block_no[len(block_no)-len(bin((2*CL)//n)[2:])+1:]
122         print("Set 2 no.: "+str(set_no2)+" (" +str(int(set_no2,2))+")")
123
124         tag_no1=block_no[:len(block_no)-len(bin(CL//n)[2:])+1]
125         print("Tag1: "+str(tag_no1)+" (" +str(int(tag_no1,2))+")")
126
127         tag_no2=block_no[:len(block_no)-len(bin((2*CL)//n)[2:])+1]

```

```

135     if inptype=='w':
136
137         chk=0
138
139         for i in range(len(tag1[set_index1])):
140
141             if tag1[set_index1][i]==[tag_no1]:
142                 chk=1
143                 break
144
145         if chk==1:
146
147             print("ADDRESS FOUND IN CACHE")
148             print()
149             print("ADDRESS FOUND IN L1")
150             print()
151             print("ADDRESS FOUND IN SET NO. "+str(set_index1)+" IN L1")
152             print()
153             print("ADDRESS FOUND IN LINE NO. "+str(set_index1*n+L1temp[set_index1].index(L1[set_index1][i]))+" IN L1")
154             print()
155             print("UPDATING THE GIVEN ADDRESS WITH GIVEN VALUE IN CACHE MEMORY")
156             print()
157             print("WORD NO. "+str(word_no)+" UPDATED WITH GIVEN DATA IN L1")
158             L1[set_index1][i][int(block_offset,2)]=[dataa]
159             L1temp[set_index1][L1temp[set_index1].index(L1[set_index1][i])][int(block_offset,2)]=[dataa]
160
161             for i in range(len(L1[set_index1])):
162                 if tag1[set_index1][i]==[tag_no1]:
163                     temp1=L1[set_index1][i]
164                     temp2=tag1[set_index1][i]
165                     break
166
167             for j in range(i+1,len(L1[set_index1])):
168                 L1[set_index1][j-1]=L1[set_index1][j]
169                 tag1[set_index1][j-1]=tag1[set_index1][j]
170
171             L1[set_index1][-1]=temp1
172             tag1[set_index1][-1]=temp2
173
174             flag=0
175
176             for i in range(len(tag2[set_index2])):

```

```

180         break
181
182     if flag==1:
183
184         print("ADDRESS FOUND IN L2 ALSO")
185         print()
186         print("ADDRESS FOUND IN SET NO. "+str(set_index2)+" IN L2")
187         print()
188         print("ADDRESS FOUND IN LINE NO. "+str(set_index2*n+L2temp[set_index2].index(L2[set_index2][i]))+" IN L2")
189         print()
190
191         for i in range(len(L2[set_index2])):
192             if tag2[set_index2][i]==[tag_no2]:
193                 temp1=L2[set_index2][i]
194                 temp2=tag2[set_index2][i]
195                 break
196
197         for j in range(i+1,len(L2[set_index2])):
198             L2[set_index2][j-1]=L2[set_index2][j]
199             tag2[set_index2][j-1]=tag2[set_index2][j]
200         print("UPDATING THE GIVEN ADDRESS WITH GIVEN VALUE IN CACHE MEMORY")
201         print()
202         print("WORD NO. "+str(word_no)+" UPDATED WITH GIVEN DATA IN L2 ALSO")
203         L2[set_index2][-1]=temp1
204         tag2[set_index2][-1]=temp2
205         L2[set_index2][-1][int(block_offset,2)]=[dataaa]
206         L2temp[set_index2][L2temp[set_index2].index(L2[set_index2][i])][int(block_offset,2)]=[dataaa]
207
208     else:
209
210         flag=0
211
212         for i in range(len(tag2[set_index2])):
213
214             if tag2[set_index2][i]==[tag_no2]:
215                 flag=1
216                 break
217
218         if flag==1:
219
220             print("ADDRESS FOUND IN CACHE")
221             print()
222             print("ADDRESS FOUND IN L2")

```



```

295     print("WORD NO. "+str(word_no)+" UPDATED WITH GIVEN VALUE IN L1 AND L2")
296     L1temp[set_index1][L1temp[set_index1].index(L1[set_index1][i])][int(block_offset,2)]=[dataaa]
297
298     else:
299
300         print("ADDRESS NOT FOUND IN CACHE")
301         print()
302         print("LOADING DATA INTO L1 AND L2")
303         print()
304         print("REPLACING DATA IN L1 AT SET NO. ",end="")
305         print(set_index1)
306         print()
307         print("REPLACING DATA IN L1 AT LINE NO. ",end="")
308
309         flag=0
310
311         for i in range(len(L1[set_index1])):
312
313             if L1[set_index1][i]==[["NULL"]]*B:
314                 L1[set_index1][i]=[["NULL"]]*B
315                 L1temp[set_index1][i]=[["NULL"]]*B
316                 L1[set_index1][i][int(block_offset,2)]=[dataaa]
317                 L1temp[set_index1][i][int(block_offset,2)]=[dataaa]
318                 tag1[set_index1][i]=[tag_no1]
319                 tag1temp[set_index1][i]=[tag_no1]
320                 print(str(set_index1*n+i))
321                 print()
322                 print("REPLACED ADDRESS IN L1: ")
323                 print("Line was empty, no block found!")
324                 print()
325                 print("DATA LOADED IN L1")
326                 print()
327                 flag=1
328                 break
329
330         if flag==0:
331
332             temp=tag1[set_index1][0]
333             L1temp[set_index1][L1temp[set_index1].index(L1[set_index1][0])]=[["NULL"]]*B
334             L1temp[set_index1][L1temp[set_index1].index(L1[set_index1][0])][int(block_offset,2)]=[dataaa]
335             tag1temp[set_index1][tag1temp[set_index1].index(tag1[set_index1][0])]=[tag_no1]
336
337             for i in range(1,len(L1[set_index1])):

```

```

353 print()
354 print("REPLACING DATA IN L2 AT LINE NO. ",end="")
355
356 flag=0
357
358 for i in range(len(L2[set_index2])):
359
360     if L2[set_index2][i]==[["NULL"]]*B:
361         L2[set_index2][i]=[["NULL"]]*B
362         L2temp[set_index2][i]=[["NULL"]]*B
363         L2[set_index2][i][int(block_offset,2)]=dataaa
364         L2temp[set_index2][i][int(block_offset,2)]=dataaa
365         tag2[set_index2][i]=tag_no2
366         tag2temp[set_index2][i]=tag_no2
367         print(str(set_index2*n+i))
368         print()
369         print("REPLACED ADDRESS IN L2: ")
370         print("Line was empty, no block found!")
371         print()
372         print("DATA LOADED IN L2")
373         print()
374         print("UPDATING THE GIVEN ADDRESS WITH GIVEN VALUE IN CACHE MEMORY")
375         print()
376         print("WORD NO. "+str(word_no)+" UPDATED WITH GIVEN VALUE IN L1 AND L2")
377         flag=1
378         break
379
380 if flag==0:
381
382     temp=tag2[set_index2][0]
383     L2temp[set_index2][L2temp[set_index2].index(L2[set_index2][0])]=[["NULL"]]*B
384     L2temp[set_index2][L2temp[set_index2].index(L2[set_index2][0])][int(block_offset,2)]=dataaa
385     tag2temp[set_index2][tag2temp[set_index2].index(tag2[set_index2][0])]=tag_no2
386
387     for j in range(1,len(L2[set_index2])):
388         L2[set_index2][j-1]=L2[set_index2][j]
389         tag2[set_index2][j-1]=tag2[set_index2][j]
390
391     L2[set_index2][-1]=[["NULL"]]*B
392     L2[set_index2][-1][int(block_offset,2)]=dataaa
393     tag2[set_index2][-1]=tag_no2
394     print(str(set_index2*n+L2temp[set_index2].index(L2[set_index2][i])))
395     print()

```

```

453     print()
454     print("ADDRESS FOUND IN L2")
455     print()
456     print("LOADING DATA FROM L2 INTO L1")
457     print()
458     print("LOADING DATA FROM SET NO. "+str(set_index2)+" IN L2")
459     print()
460     print("LOADING DATA FROM LINE NO. "+str(set_index2*n+L2temp[set_index2].index(L2[set_index2][i]))+" IN L2")
461     print()
462
463     for i in range(len(L2[set_index2])):
464         if tag2[set_index2][i]==[tag_no2]:
465             temp1=L2[set_index2][i]
466             temp2=tag2[set_index2][i]
467             break
468
469     for j in range(i+1,len(L2[set_index2])):
470         L2[set_index2][j-1]=L2[set_index2][j]
471         tag2[set_index2][j-1]=tag2[set_index2][j]
472
473     L2[set_index2][-1]=temp1
474     tag2[set_index2][-1]=temp2
475     print("DATA: ",end=" ")
476     print(str(*L2[set_index2][-1][int(block_offset,2)]))
477
478
479     else:
480
481         print("CACHE MISS!!!")
482         print("ADDRESS NOT FOUND IN CACHE")
483
484     print()
485     L1disp()
486     print()
487     L2disp()
488     print()
489
490     cont='y'
491     while cont=='y':
492         f()
493         cont=input("continue? (y/n) ")
494         print()

```