

DOCUMENTATION

COMPUTER ORGANISATION

END SEM ASSIGNMENT

ONE LEVEL CACHE IMPLEMENTATION

Name : ABHISHEK CHATURVEDI

Roll no. : 2019401

Group no. : 3

ASSIGNMENT OVERVIEW

- Implementation of one 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).
- 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:
 - Cache memory 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.
- 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.

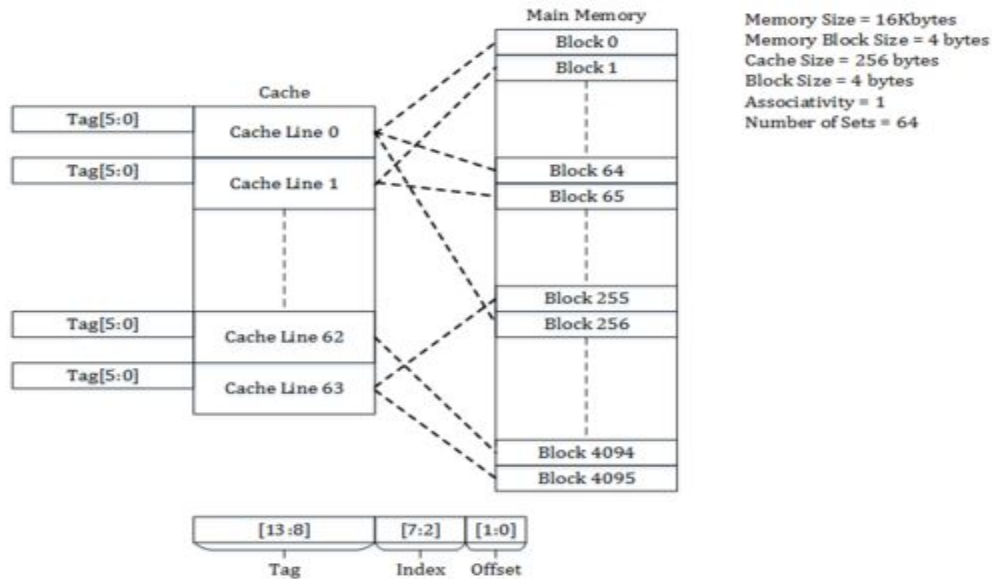
ERROR HANDLING

- 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 cache cannot be equal to or greater than the main memory size, if this is not the case then the program gives inbuilt python error.

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, the line no. and the tag.
- 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.
- The address' tag bits are compared to the tag bits at the cache line no.
- If the two tags match then it is a hit else it is a miss.
- 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.
- The program simply compares the tag of the given address and the tag of the line no. where block is to be placed.
- If the two tags match it is a hit and if it is a write operation then value is written at block offset in the block and if it is a miss and if it is a write operation then the block in the cache line is removed and the new block is placed in the line and then the value is written at the address.
- If it is a hit and if it is a read operation then value at block offset of the block is returned.

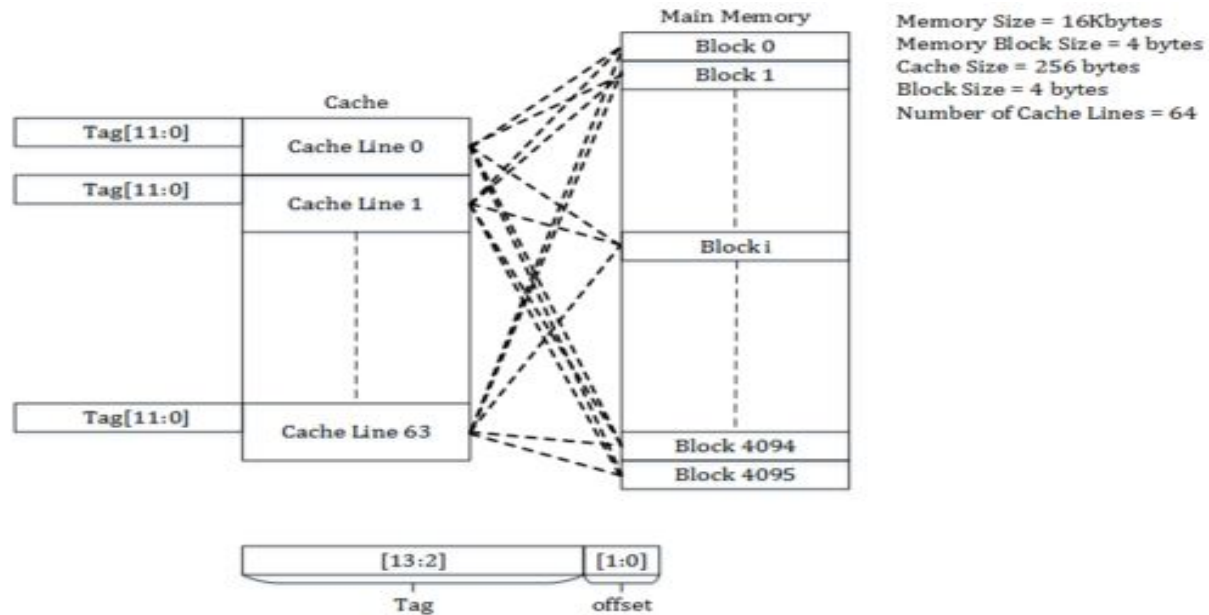
ANALYSIS OF THE CODE

- 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 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)

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 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 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 cache memory 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 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.

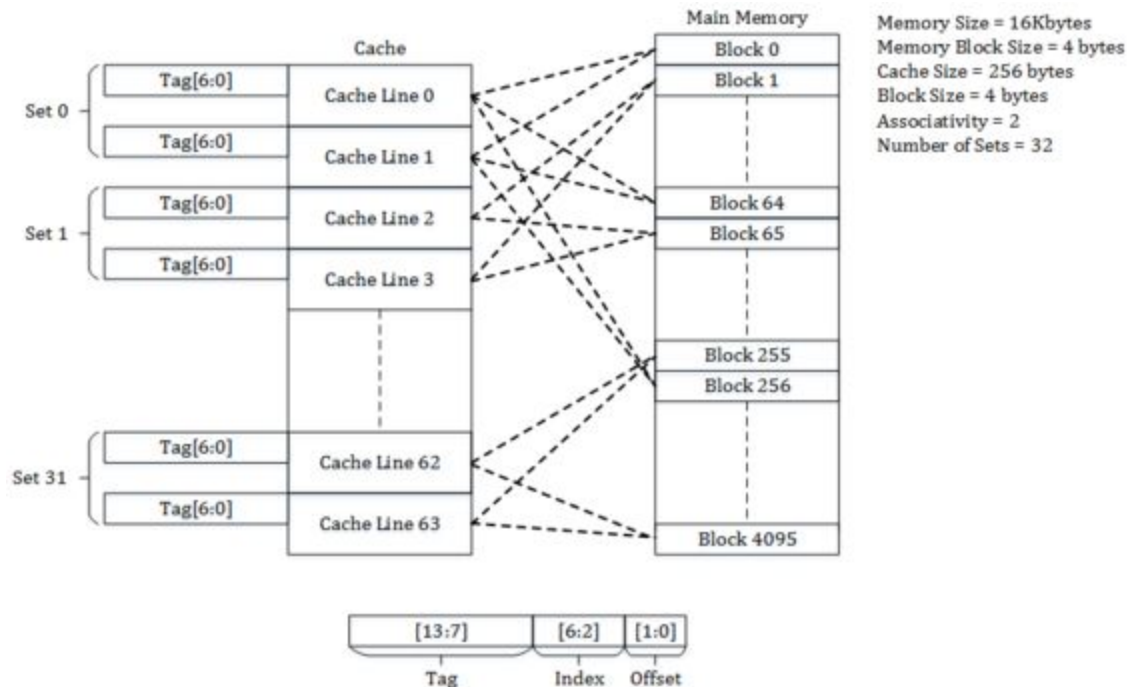
ANALYSIS OF CODE

- 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 block 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 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 cache memory is divided in sets each containing n cache lines.
- Therefore, the number of sets in the cache memory is equal to quotient when (number of cache lines) 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 and tag.
- Then the program iterates over all the cache lines in the set(derived from the address) and compares the tag bits.
- If the tag bits of address and tag bits of the 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 it is a miss 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

- Two arrays(list in python) are used as data structures for storing tag values and blocks separately.
- Since the program only compares all the tags in a given set it takes linear time i.e, $O(n)$. (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 complexity of the program is $O(n)$.
- 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

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

```

1  #DIRECT MAPPING
2  print()
3  print("                ONE LEVEL CACHE // DIRECT 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
14 cache_mem=[]
15 tag=[]
16
17 for i in range(CL):
18     cache_mem.append([])
19     tag.append(["NULL"])
20 for i in range(CL):
21     for j in range(B):
22         cache_mem[i].append("NULL")
23
24
25 def cachedisp():
26
27     print("-----CACHE--MEMORY-----")
28     print()
29     print()
30     print("        LINE NO.            TAG                DATA")
31     print()
32     for i in range(CL):
33         print(" "+str(i)+"",end="")
34         print(*tag[i],end="")
35         print(" ",end="")
36         if tag[i][0]!="NULL":
37             print(" ",end="")
38             print(*cache_mem[i])
39         else:
40             print(" ",end="")
41             print(*cache_mem[i])
42
43 def f():

```

```

44
45     inptype=input("write(w)/read(r): ")
46
47     print()
48
49     if inptype=='r':
50         add=input("Enter address in binary form: ")
51
52     else:
53         add=input("Enter address in binary form: ")
54         dataa=input("Enter data: ")
55
56     print()
57
58     if(len(add)!=len(bin(N)[2:])-1):
59         print("INVALID ADDRESS")
60
61     else:
62
63         print("ADDRESS BREAKDOWN: ")
64         print()
65
66         word_no=int(add,2)
67         print("Word No.: "+add+" (" +str(word_no)+")")
68
69         block_offset=add[len(bin(N)[2:])-len(bin(B)[2:]):]
70         print("Block Offset: "+block_offset+" (" +str(int(block_offset,2))+")")
71
72         block_no=add[:len(bin(N)[2:])-len(bin(B)[2:])]
73         print("Block No.: "+block_no+" (" +str(int(block_no,2))+")")
74
75         line_no=block_no[len(block_no)-len(bin(CL)[2:])+1:]
76         print("Line No.: "+line_no+" (" +str(int(line_no,2))+")")
77
78         tag_no=block_no[:len(block_no)-len(bin(CL)[2:])+1]
79         print("Tag : "+tag_no+" (" +str(int(tag_no,2))+")")
80         print()
81
82         line_index=int(line_no,2)
83
84         print()

```

```

86     if inptype=='w':
87
88         if tag[line_index]!='NULL' or tag[line_index][0]!=tag_no:
89
90             print("ADDRESS NOT FOUND IN CACHE")
91             print()
92             print("LOADING DATA IN CACHE MEMORY")
93             print()
94             print("REPLACING DATA IN CACHE MEMORY AT LINE NO. "+str(line_index))
95             print()
96             print("LOADING BLOCK NO. "+str(int(block_no,2))+" IN CACHE MEMORY")
97             print()
98             print("DATA LOADED IN CACHE MEMORY")
99             print()
100            tag[line_index]=[tag_no]
101            cache_mem[line_index]='NULL'*B
102            cache_mem[line_index][int(block_offset,2)]=dataa
103            print("WORD NO. "+str(word_no)+" UPDATED WITH GIVEN VALUE")
104
105            elif tag[line_index][0]==tag_no:
106
107                print("ADDRESS FOUND IN CACHE")
108                print()
109                cache_mem[line_index][int(block_offset,2)]=dataa
110                print("WORD NO. "+str(word_no)+" UPDATED WITH GIVEN VALUE")
111
112            else:
113
114                if tag[line_index][0]==tag_no:
115
116                    print("CACHE HIT!!! ADDRESS FOUND")
117                    print()
118                    print("LOADING DATA FROM LINE NO. "+str(line_index)+" IN CACHE MEMORY")
119                    print()
120                    print("LOADING DATA FROM BLOCK NO. "+str(int(block_no,2))+" IN CACHE MEMORY")
121                    print()
122                    print("DATA: ",end=" ")
123                    print(str(cache_mem[line_index][int(block_offset,2)]))
124                    print()
125
126                else:
127
128                    print("CACHE MISS!!! ADDRESS NOT FOUND")

```

```

129            print("WORD NO. "+str(word_no)+" UPDATED WITH GIVEN VALUE")
130
131            else:
132
133                if tag[line_index][0]==tag_no:
134
135                    print("CACHE HIT!!! ADDRESS FOUND")
136                    print()
137                    print("LOADING DATA FROM LINE NO. "+str(line_index)+" IN CACHE MEMORY")
138                    print()
139                    print("LOADING DATA FROM BLOCK NO. "+str(int(block_no,2))+" IN CACHE MEMORY")
140                    print()
141                    print("DATA: ",end=" ")
142                    print(str(cache_mem[line_index][int(block_offset,2)]))
143                    print()
144
145                else:
146
147                    print("CACHE MISS!!! ADDRESS NOT FOUND")
148                    print()
149
150            print()
151            cachedisp()
152            print()
153
154            cont='y'
155            while cont=='y':
156                f()
157                cont=input("continue? (y/n) ")
158                print()

```

```

42
43 class BlockAddress:
44
45     def __init__(self, blockno):
46         self.blockno = blockno
47         self.next = None
48         self.prev = None
49
50 class CacheList:
51
52     def __init__(self):
53         self.head = None
54
55     def push(self, new_data):
56         new_node = BlockAddress(new_data)
57         new_node.next = self.head
58
59         if self.head is not None:
60             self.head.prev = new_node
61         self.head = new_node
62
63     def lastNode(self, blockaddress):
64         while(blockaddress.next is not None):
65             blockaddress = blockaddress.next
66         return blockaddress
67
68     def getNode(self, blockaddress, x):
69         while(blockaddress.blockno != x):
70             blockaddress = blockaddress.next
71         return blockaddress
72
73     def deleteNode(self, dele):
74         if self.head is None or dele is None:
75             return
76         if self.head == dele:
77             self.head = dele.next
78         if dele.next is not None:
79             dele.next.prev = dele.prev
80         if dele.prev is not None:
81             dele.prev.next = dele.next
82
83
84 def cachedisp():

```

```

83
84 def cachedisp():
85
86     print("-----CACHE--MEMORY-----")
87     print()
88     print()
89     print(" LINE NO. TAG DATA")
90     print()
91     for i in range(len(block_address)):
92         print(" "+str(i)+" ", end="")
93         print(block_address[i], end=" ")
94         print("block_address1[block_address[i]]")
95     for j in range(len(block_address), CL):
96         print(" "+str(j)+" ", end="")
97         print("NULL", end=" ")
98     for i in range(B):
99         print("NULL", end=" ")
100     print()
101
102 print()
103
104 N=int(input("Enter main memory size: "))
105 print()
106 CL=int(input("Enter number of cache lines: "))
107 print()
108 B=int(input("Enter block size: "))
109 print()
110
111 block_address=[]
112 block_address1={}
113 address_list=CacheList()
114
115
116 def f():
117
118     inptype=input("write(w)/read(r): ")
119
120     print()
121
122     if inptype=='r':
123         add=input("Enter address in binary form: ")
124
125     else:

```

```

124
125     else:
126         add=input("Enter address in binary form: ")
127         dataa=input("Enter data: ")
128
129     print()
130
131     if(len(add)!=len(bin(N)[2:])-1):
132         print("INVALID ADDRESS")
133
134     else:
135
136         print("ADDRESS BREAKDOWN: ")
137         print()
138
139         word_no=int(add,2)
140         print("Word No.: "+str(add)+" (" +str(word_no)+")")
141
142         block_offset=add[len(bin(N)[2:])-len(bin(B)[2:]):]
143         print("Block Offset: "+str(block_offset)+" (" +str(int(block_offset,2))+")")
144
145         block_no=add[:len(bin(N)[2:])-len(bin(B)[2:])]
146         print("Tag: "+str(block_no)+" (" +str(int(block_no,2))+")")
147
148         print()
149
150         if inptype=='w':
151
152             if block_no in block_address1:
153
154                 print("ADDRESS FOUND IN CACHE")
155                 print()
156                 print("WORD NO. "+str(word_no)+" UPDATED WITH GIVEN VALUE")
157                 block_address1[block_no][int(block_offset,2)]=dataa
158                 address_list.deleteNode(address_list.getnode(address_list.head,block_no))
159                 address_list.push(block_no)
160
161             else:
162
163                 print("ADDRESS NOT IN CACHE")
164                 print()
165                 print("LOADING DATA IN CACHE MEMORY")
166                 print()

```

```

166
167
168     else:
169
170         print("BLOCK NO.: "+address_list.lastNode(address_list.head).blockno)
171         del block_address1[address_list.lastNode(address_list.head).blockno]
172         address_list.deleteNode(address_list.lastNode(address_list.head))
173         block_address1[block_no]=["NULL"]*B
174         address_list.push(block_no)
175         print()
176         print("WORD NO. "+str(word_no)+" UPDATED WITH GIVEN VALUE")
177         block_address1[block_no][int(block_offset,2)]=dataa
178
179     else:
180
181         if block_no in block_address1:
182
183             print("CACHE HIT!!! ADDRESS FOUND IN CACHE")
184             print()
185             print("LOADING DATA FROM LINE NO. "+str(block_address.index(block_no))+ " IN CACHE MEMORY")
186             print()
187             print("LOADING DATA FROM BLOCK NO. "+str(int(block_no,2))+ " IN CACHE MEMORY")
188             print()
189             print("DATA: ",end=" ")
190             print(block_address1[block_no][int(block_offset,2)])
191             address_list.deleteNode(address_list.getnode(address_list.head,block_no))
192             address_list.push(block_no)
193
194         else:
195
196             print("CACHE MISS!!! ADDRESS NOT FOUND IN CACHE")
197
198     print()
199     cachedisp()
200     print()
201
202     cont='y'
203     while cont=='y':
204         f()
205         cont=input("continue? (y/n) ")
206         print()
207

```



```

1  # n WAY SET ASSOCIATIVE
2  print()
3  print("                ONE LEVEL CACHE // n WAY SET ASSOCIATIVE MAPPING IMPLEMENTATION")
4  print()
5  print()
6
7  def cachedisp():
8
9      print("-----CACHE--MEMORY-----")
10     print()
11     print()
12     print("    LINE NO.        TAG            SET NO.            DATA")
13     print()
14
15     for i in range(CL//n):
16         for j in range(n):
17             print("    "+str(i)*n+j+"            "+tag[i][j][0]+"            "+str(i)+"            ",end=" ")
18             for k in range(B):
19                 print(cachetemp[i][j][k],end=" ")
20             print()
21
22     print()
23
24     N=int(input("Enter main memory size: "))
25     print()
26     CL=int(input("Enter number of cache lines: "))
27     print()
28     B=int(input("Enter block size: "))
29     print()
30     n=int(input("Enter 'n': "))
31
32     print()
33
34     cache_mem=[]
35     cachetemp=[]
36     tag=[]
37     tagtemp=[]
38
39     for i in range(CL//n):
40         cache_mem.append([])
41         tag.append([])
42         cachetemp.append([])
43         tagtemp.append([])

```

```

44
45     for i in range(len(cache_mem)):
46         for j in range(n):
47             cache_mem[i].append(["NULL"]*B)
48             tag[i].append(["NULL"])
49             cachetemp[i].append(["NULL"]*B)
50             tagtemp[i].append(["NULL"])
51
52     def f():
53
54         inptype=input("write(w)/read(r): ")
55
56         print()
57
58         if inptype=='r':
59             add=input("Enter address in binary form: ")
60
61         else:
62             add=input("Enter address in binary form: ")
63             dataa=input("Enter data: ")
64
65         print()
66
67         if(len(add)!=len(bin(N)[2:])-1):
68             print("INVALID ADDRESS")
69
70         else:
71
72             print("ADDRESS BREAKDOWN: ")
73             print()
74
75             word_no=int(add,2)
76             print("Word No.: "+str(add)+" (" +str(word_no)+)")")
77
78             block_offset=add[len(bin(N)[2:])-len(bin(B)[2:]):]
79             print("Block Offset: "+str(block_offset)+" (" +str(int(block_offset,2))+)")")
80
81             block_no=add[:len(bin(N)[2:])-len(bin(B)[2:])]
82             print("Block no.: "+str(block_no)+" (" +str(int(block_no,2))+)")")
83
84             set_no=block_no[len(block_no)-len(bin(CL//n)[2:])+1:]
85             print("Set no.: "+str(set_no)+" (" +str(int(set_no,2))+)")")
86

```

```

92
93     if inptype=='w':
94
95         chk=0
96
97         for i in range(len(tag[set_index])):
98
99             if tag[set_index][i]==[tag_no]:
100                 chk=1
101                 break
102
103         if chk==1:
104
105             print("ADDRESS FOUND IN CACHE")
106             print()
107             print("WORD NO. "+str(word_no)+" UPDATED WITH GIVEN VALUE")
108             cache_mem[set_index][i][int(block_offset,2)]=dataa
109             cachetemp[set_index][cachetemp[set_index].index(cache_mem[set_index][i])][int(block_offset,2)]=dataa
110             for i in range(len(cache_mem[set_index])):
111                 if tag[set_index][i]==[tag_no]:
112                     temp1=cache_mem[set_index][i]
113                     temp2=tag[set_index][i]
114                     break
115
116             for j in range(i+1,len(cache_mem[set_index])):
117                 cache_mem[set_index][j-1]=cache_mem[set_index][j]
118                 tag[set_index][j-1]=tag[set_index][j]
119
120             cache_mem[set_index][-1]=temp1
121             tag[set_index][-1]=temp2
122
123         else:
124
125             print("ADDRESS NOT FOUND IN CACHE")
126             print()
127             print("LOADING DATA IN CACHE MEMORY")
128             print()
129             print("REPLACING DATA IN CACHE MEMORY AT SET NO. ",end="")
130             print(set_index)
131             print()
132             print("LOADING BLOCK NO. "+str(int(block_no,2))+" IN CACHE MEMORY")
133             print()
134             print("REPLACING DATA IN CACHE MEMORY AT LINE NO. ",end="")

```

```

178
179     chk=0
180
181     for i in range(len(tag[set_index])):
182
183         if tag[set_index][i]==[tag_no]:
184             chk=1
185             break
186
187     if chk==1:
188
189         print("CACHE HIT!!! ADDRESS FOUND IN CACHE")
190         print()
191         print("LOADING DATA FROM CACHE MEMORY")
192         print()
193         print("LOADING DATA FROM SET NO. "+str(set_index)+" IN CACHE MEMORY")
194         print()
195         print("LOADING DATA FROM LINE NO. "+str(set_index*n+cachetemp[set_index].index(cache_mem[set_index][i]))+" IN CACHE MEMORY")
196         print()
197         print("DATA: ",end=" ")
198         print(str(cache_mem[set_index][i][int(block_offset,2)]))
199
200     for i in range(len(cache_mem[set_index])):
201         if tag[set_index][i]==[tag_no]:
202             temp1=cache_mem[set_index][i]
203             temp2=tag[set_index][i]
204             break
205
206     for j in range(i+1,len(cache_mem[set_index])):
207         cache_mem[set_index][j-1]=cache_mem[set_index][j]
208         tag[set_index][j-1]=tag[set_index][j]
209
210     cache_mem[set_index][-1]=temp1
211     tag[set_index][-1]=temp2
212
213     else:
214
215         print("CACHE MISS!!! ADDRESS NOT FOUND IN CACHE")
216
217     print()
218     cachedisp()
219     print()

```