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.**

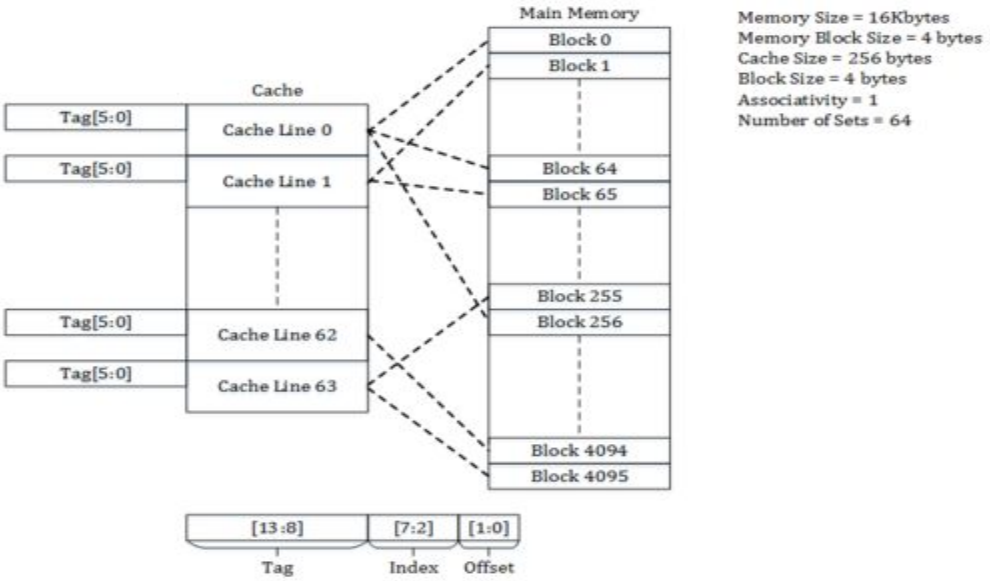
ERROR HANDLING

- **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.**

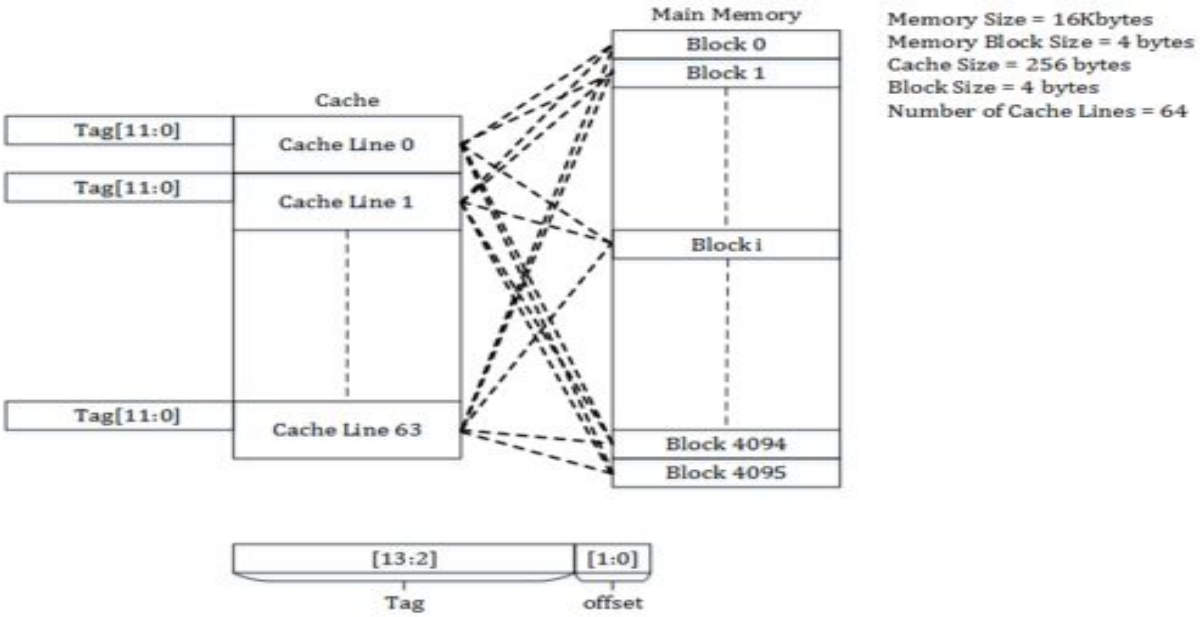
ANALYSIS OF THE CODE

- **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.**

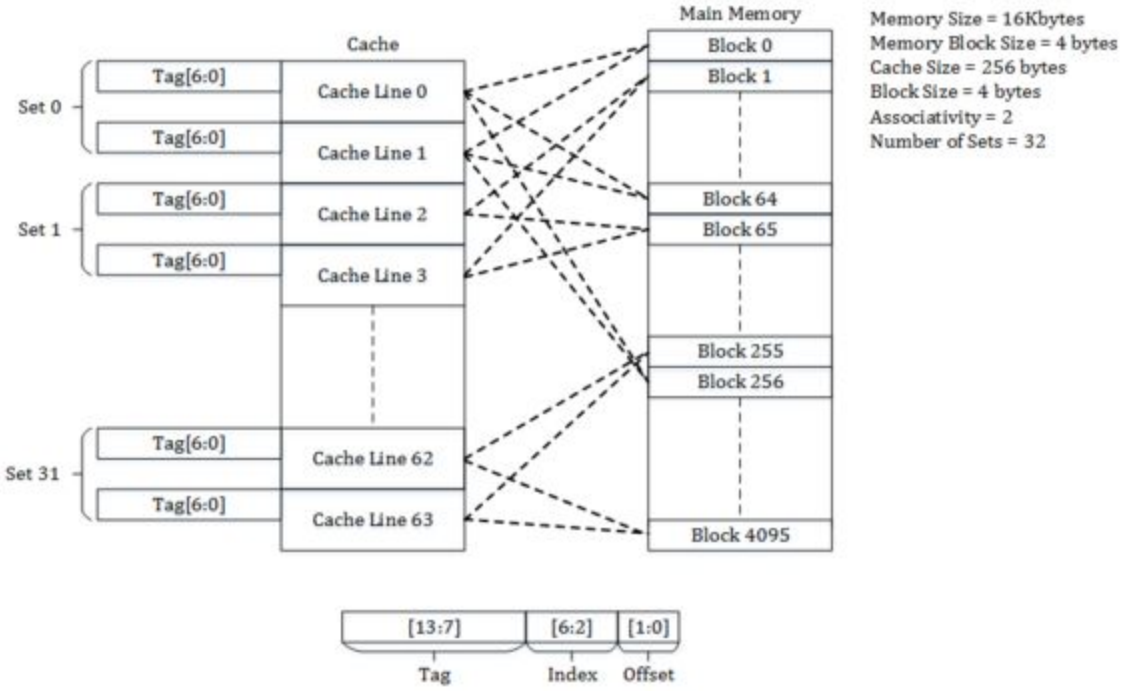
ANALYSIS OF CODE

- **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)*******************

```python
#DIRECT MAPPING
print()
print("                    ONE LEVEL CACHE // DIRECT MAPPING IMPLEMENTATION                                    ")
print()
print()

N=int(input("Enter main memory size: "))
print()
CL=int(input("Enter number of cache lines: "))
print()
B=int(input("Enter block size: "))
print()

cache_mem=[]
tag=[]

for i in range(CL):
    cache_mem.append([])
    tag.append(["NULL"])
for i in range(CL):
    for j in range(B):
        cache_mem[i].append("NULL")


def cachedisp():

    print("-----------------------------------------CACHE--MEMORY-----------------------------------------------")
    print()
    print()
    print("  LINE NO.              TAG                                    DATA")
    print()
    for i in range(CL):
        print("   "+str(i)+"                    ",end="")
        print(*tag[i],end="")
        print("                        ",end="")
        if tag[i][0]!="NULL":
            print("         ",end="")
            print(*cache_mem[i])
        else:
            print("         ",end="")
            print(*cache_mem[i])

def f():
```

```python
def f():

    inptype=input("write(w)/read(r): ")

    print()

    if inptype=='r':
        add=input("Enter address in binary form: ")

    else:
        add=input("Enter address in binary form: ")
        dataa=input("Enter data: ")

    print()

    if(len(add)!=len(bin(N)[2:])-1):
        print("INVALID ADDRESS")

    else:

        print("ADDRESS BREAKDOWN: ")
        print()

        word_no=int(add,2)
        print("Word No.: "+add+" ("+str(word_no)+")")

        block_offset=add[len(bin(N)[2:])-len(bin(B)[2:]):]
        print("Block Offset: "+block_offset+" ("+str(int(block_offset,2))+")")

        block_no=add[:len(bin(N)[2:])-len(bin(B)[2:])]
        print("Block No.: "+block_no+" ("+str(int(block_no,2))+")")

        line_no=block_no[len(block_no)-len(bin(CL)[2:])+1:]
        print("Line No.: "+line_no+" ("+str(int(line_no,2))+")")

        tag_no=block_no[:len(block_no)-len(bin(CL)[2:])+1]
        print("Tag :"+tag_no+" ("+str(int(tag_no,2))+")")
        print()

        line_index=int(line_no,2)

        print()
```

```
        if inptype=='w':

            if tag[line_index]==["NULL"] or tag[line_index][0]!=tag_no:

                print("ADDRESS NOT FOUND IN CACHE")
                print()
                print("LOADING DATA IN CACHE MEMORY")
                print()
                print("REPLACING DATA IN CACHE MEMORY AT LINE NO. "+str(line_index))
                print()
                print("LOADING BLOCK NO. "+str(int(block_no,2))+" IN CACHE MEMORY")
                print()
                print("DATA LOADED IN CACHE MEMORY")
                print()
                tag[line_index]=[tag_no]
                cache_mem[line_index]=["NULL"]*8
                cache_mem[line_index][int(block_offset,2)]=dataa
                print("WORD NO. "+str(word_no)+" UPDATED WITH GIVEN VALUE")

            elif tag[line_index][0]==tag_no:

                print("ADDRESS FOUND IN CACHE")
                print()
                cache_mem[line_index][int(block_offset,2)]=dataa
                print("WORD NO. "+str(word_no)+" UPDATED WITH GIVEN VALUE")

        else:

            if tag[line_index][0]==tag_no:

                print("CACHE HIT!!! ADDRESS FOUND")
                print()
                print("LOADING DATA FROM LINE NO. "+str(line_index)+" IN CACHE MEMORY")
                print()
                print("LOADING DATA FROM BLOCK NO. "+str(int(block_no,2))+" IN CACHE MEMORY")
                print()
                print("DATA: ",end=" ")
                print(str(cache_mem[line_index][int(block_offset,2)]))
                print()

            else:

                print("CACHE MISS!!! ADDRESS NOT FOUND")
                print()

    print()
    cachedisp()
    print()

cont='y'
while cont=='y':
    f()
    cont=input("continue? (y/n) ")
    print()
```

```
class BlockAddress:

    def __init__(self, blockno):
        self.blockno = blockno
        self.next = None
        self.prev = None

class CacheList:

    def __init__(self):
        self.head = None

    def push(self, new_data):
        new_node = BlockAddress(new_data)
        new_node.next = self.head

        if self.head is not None:
            self.head.prev = new_node
        self.head = new_node

    def lastNode(self, blockaddress):
        while(blockaddress.next is not None):
            blockaddress = blockaddress.next
        return blockaddress

    def getnode(self, blockaddress,x):
        while(blockaddress.blockno != x):
            blockaddress = blockaddress.next
        return blockaddress

    def deleteNode(self, dele):
        if self.head is None or dele is None:
            return
        if self.head == dele:
            self.head = dele.next
        if dele.next is not None:
            dele.next.prev = dele.prev
        if dele.prev is not None:
            dele.prev.next = dele.next


def cachedisp():

    print("-----------------------------------------CACHE--MEMORY-----------------------------------------------")
    print()
    print()
    print("  LINE NO.              TAG                 DATA")
    print()
    for i in range(len(block_address)):
        print("  "+str(i)+"                ",end="")
        print(block_address[i],end="                            ")
        print(*block_address1[block_address[i]])
    for j in range(len(block_address),CL):
        print("  "+str(j)+"                    ",end="")
        print("NULL",end="                 ")
        for i in range(B):
            print("NULL",end=" ")
        print()

print()

N=int(input("Enter main memory size: "))
print()
CL=int(input("Enter number of cache lines: "))
print()
B=int(input("Enter block size: "))
print()

block_address=[]
block_address1={}
address_list=CacheList()


def f():

    inptype=input("write(w)/read(r): ")

    print()

    if inptype=='r':
        add=input("Enter address in binary form: ")

    else:
```

```
    else:
        add=input("Enter address in binary form: ")
        dataa=input("Enter data: ")

    print()

    if(len(add)!=len(bin(N)[2:])-1):
        print("INVALID ADDRESS")

    else:

        print("ADDRESS BREAKDOWN: ")
        print()

        word_no=int(add,2)
        print("Word No.: "+str(add)+" ("+str(word_no)+")")

        block_offset=add[len(bin(N)[2:])-len(bin(B)[2:]):]
        print("Block Offset: "+str(block_offset)+" ("+str(int(block_offset,2))+")")

        block_no=add[:len(bin(N)[2:])-len(bin(B)[2:])]
        print("Tag: "+str(block_no)+" ("+str(int(block_no,2))+")")

        print()

        if inptype=='w':

            if block_no in block_address1:

                print("ADDRESS FOUND IN CACHE")
                print()
                print("WORD NO. "+str(word_no)+" UPDATED WITH GIVEN VALUE")
                block_address1[block_no][int(block_offset,2)]=dataa
                address_list.deleteNode(address_list.getnode(address_list.head,block_no))
                address_list.push(block_no)

            else:

                print("ADDRESS NOT IN CACHE")
                print()
                print("LOADING DATA IN CACHE MEMORY")
```

```
                else:

                    print("BLOCK NO.: "+address_list.lastNode(address_list.head).blockno)
                    del block_address1[address_list.lastNode(address_list.head).blockno]
                    address_list.deleteNode(address_list.lastNode(address_list.head))
                    block_address1[block_no]=["NULL"]*B
                    address_list.push(block_no)
                print()
                print("WORD NO. "+str(word_no)+" UPDATED WITH GIVEN VALUE")
                block_address1[block_no][int(block_offset,2)]=dataa

        else:

            if block_no in block_address1:

                print("CACHE HIT!!! ADDRESS FOUND IN CACHE")
                print()
                print("LOADING DATA FROM LINE NO. "+str(block_address.index(block_no))+" IN CACHE MEMORY")
                print()
                print("LOADING DATA FROM BLOCK NO. "+str(int(block_no,2))+" IN CACHE MEMORY")
                print()
                print("DATA: ",end=" ")
                print(block_address1[block_no][int(block_offset,2)])
                address_list.deleteNode(address_list.getnode(address_list.head,block_no))
                address_list.push(block_no)

            else:

                print("CACHE MISS!!! ADDRESS NOT FOUND IN CACHE")

    print()
    cachedisp()
    print()

cont='y'
while cont=='y':
    f()
    cont=input("continue? (y/n) ")
    print()
```

```
# n WAY SET ASSOCIATIVE
print()
print("                        ONE LEVEL CACHE // n WAY SET ASSOCIATIVE MAPPING IMPLEMENTATION                                  ")
print()
print()

def cachedisp():

    print("-----------------------------------------CACHE--MEMORY-----------------------------------------------")
    print()
    print()
    print("  LINE NO.           TAG                 SET NO.                    DATA")
    print()

    for i in range(CL//n):
        for j in range(n):
            print("  "+str((i)*n+j)+"              "+tag[i][j][0]+"            "+str(i)+"             ",end=" ")
            for k in range(B):
                print(cachetemp[i][j][k],end=" ")
            print()

print()

N=int(input("Enter main memory size: "))
print()
CL=int(input("Enter number of cache lines: "))
print()
B=int(input("Enter block size: "))
print()
n=int(input("Enter 'n': "))

print()

cache_mem=[]
cachetemp=[]
tag=[]
tagtemp=[]

for i in range(CL//n):
    cache_mem.append([])
    tag.append([])
    cachetemp.append([])
    tagtemp.append([])

for i in range(len(cache_mem)):
    for j in range(n):
        cache_mem[i].append(["NULL"]*B)
        tag[i].append(["NULL"])
        cachetemp[i].append(["NULL"]*B)
        tagtemp[i].append(["NULL"])

def f():

    inptype=input("write(w)/read(r): ")

    print()

    if inptype=='r':
        add=input("Enter address in binary form: ")

    else:
        add=input("Enter address in binary form: ")
        dataa=input("Enter data: ")

    print()

    if(len(add)!=len(bin(N)[2:])-1):
        print("INVALID ADDRESS")

    else:

        print("ADDRESS BREAKDOWN: ")
        print()

        word_no=int(add,2)
        print("Word No.: "+str(add)+" ("+str(word_no)+")")

        block_offset=add[len(bin(N)[2:])-len(bin(B)[2:]):]
        print("Block Offset: "+str(block_offset)+" ("+str(int(block_offset,2))+")")

        block_no=add[:len(bin(N)[2:])-len(bin(B)[2:])]
        print("Block no.: "+str(block_no)+" ("+str(int(block_no,2))+")")

        set_no=block_no[len(block_no)-len(bin(CL//n)[2:])+1:]
        print("Set no.: "+str(set_no)+" ("+str(int(set_no,2))+")")
```

```
    if inptype=='w':

        chk=0

        for i in range(len(tag[set_index])):

            if tag[set_index][i]==[tag_no]:
                chk=1
                break

        if chk==1:

            print("ADDRESS FOUND IN CACHE")
            print()
            print("WORD NO. "+str(word_no)+" UPDATED WITH GIVEN VALUE")
            cache_mem[set_index][i][int(block_offset,2)]=dataa
            cachetemp[set_index][cachetemp[set_index].index(cache_mem[set_index][i])][int(block_offset,2)]=dataa
            for i in range(len(cache_mem[set_index])):
                if tag[set_index][i]==[tag_no]:
                    temp1=cache_mem[set_index][i]
                    temp2=tag[set_index][i]
                    break

            for j in range(i+1,len(cache_mem[set_index])):
                cache_mem[set_index][j-1]=cache_mem[set_index][j]
                tag[set_index][j-1]=tag[set_index][j]

            cache_mem[set_index][-1]=temp1
            tag[set_index][-1]=temp2

        else:

            print("ADDRESS NOT FOUND IN CACHE")
            print()
            print("LOADING DATA IN CACHE MEMORY")
            print()
            print("REPLACING DATA IN CACHE MEMORY AT SET NO. ",end="")
            print(set_index)
            print()
            print("LOADING BLOCK NO. "+str(int(block_no,2))+" IN CACHE MEMORY")
            print()
            print("REPLACING DATA IN CACHE MEMORY AT LINE NO. ",end="")
```

```
        chk=0

        for i in range(len(tag[set_index])):

            if tag[set_index][i]==[tag_no]:
                chk=1
                break

        if chk==1:

            print("CACHE HIT!!! ADDRESS FOUND IN CACHE")
            print()
            print("LOADING DATA FROM CACHE MEMORY")
            print()
            print("LOADING DATA FROM SET NO. "+str(set_index)+" IN CACHE MEMORY")
            print()
            print("LOADING DATA FROM LINE NO. "+str(set_index*n+cachetemp[set_index].index(cache_mem[set_index][i]))+" IN CACHE MEMORY")
            print()
            print("DATA: ",end=" ")
            print(str(cache_mem[set_index][i][int(block_offset,2)]))

            for i in range(len(cache_mem[set_index])):
                if tag[set_index][i]==[tag_no]:
                    temp1=cache_mem[set_index][i]
                    temp2=tag[set_index][i]
                    break

            for j in range(i+1,len(cache_mem[set_index])):
                cache_mem[set_index][j-1]=cache_mem[set_index][j]
                tag[set_index][j-1]=tag[set_index][j]

            cache_mem[set_index][-1]=temp1
            tag[set_index][-1]=temp2

        else:

            print("CACHE MISS!!! ADDRESS NOT FOUND IN CACHE")

print()
cachedisp()
print()
```