Computer Organization
End Semester Assignment

Rishit Gupta (2019091)

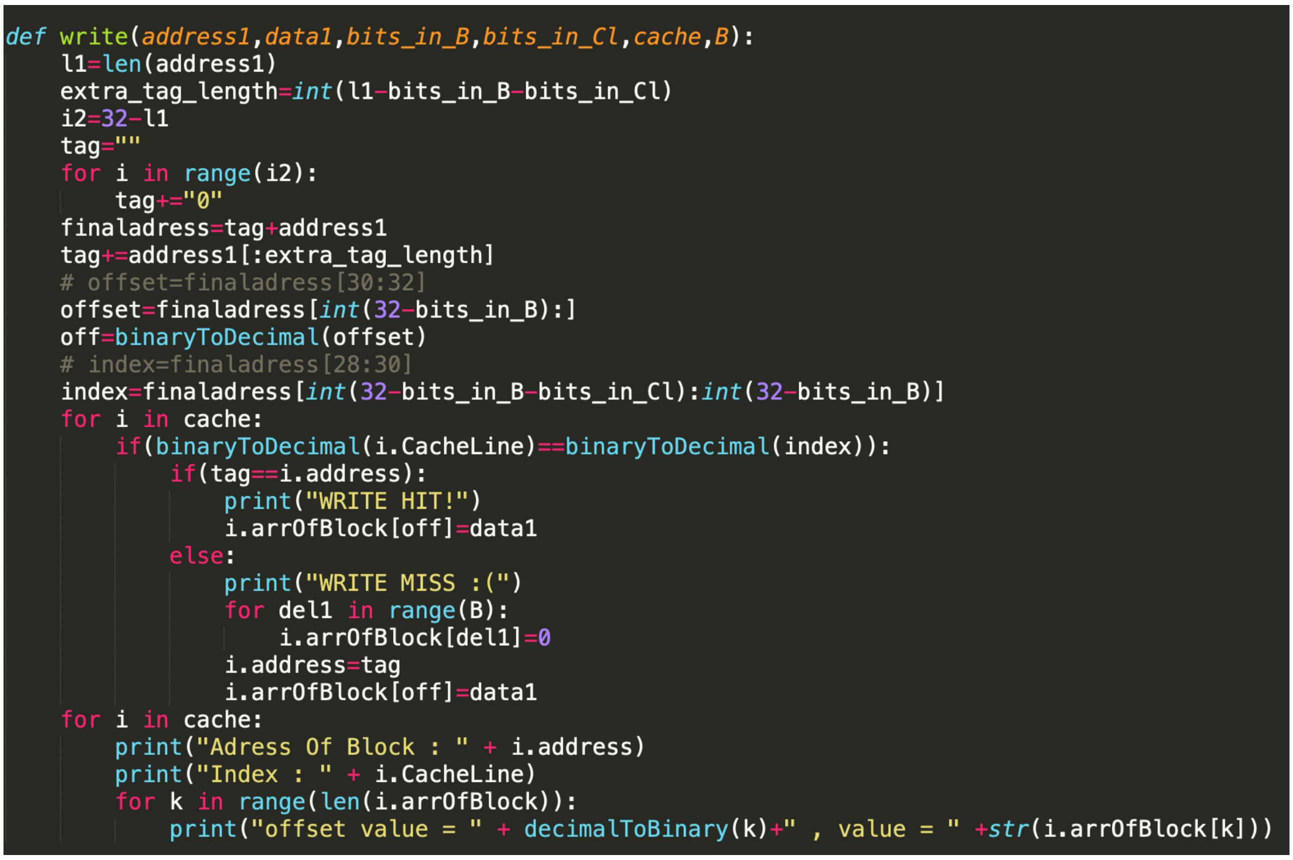
A cache of size S with CL as the number of cache lines and block size B is to be built. S, CL, and B are in powers of 2. Write a program that allows loading into cache and searching cache using:

1) *Direct Mapping :*

A direct-mapped cache is the simplest approach: each main memory address maps to exactly one cache block.
If the cache contains 2^k blocks, then the data at memory address i would go to cache block index

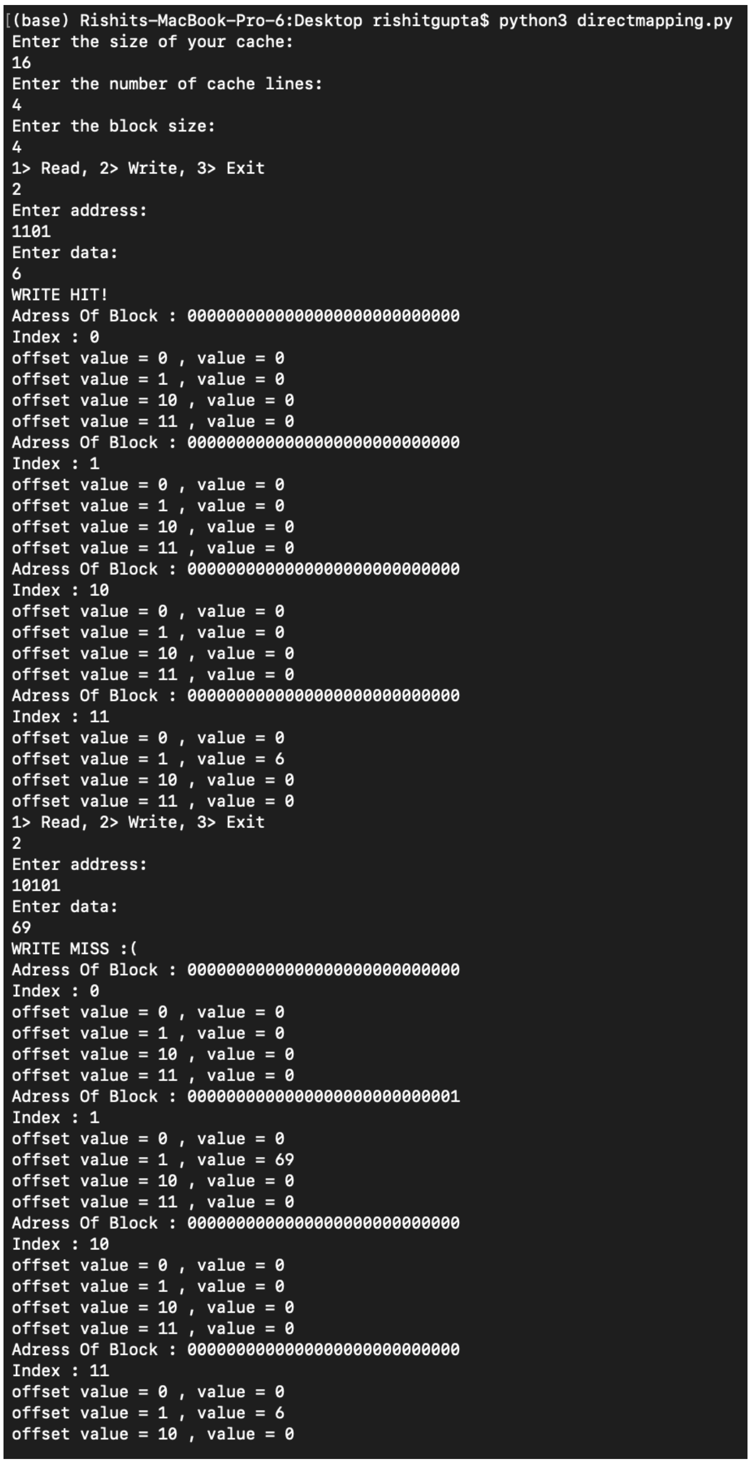
index = i mod 2^k

We need to add tags to the cache, which supply the rest of the address bits to let us distinguish between different memory locations that map to the same cache block. Read and write operations are implemented as shown below :

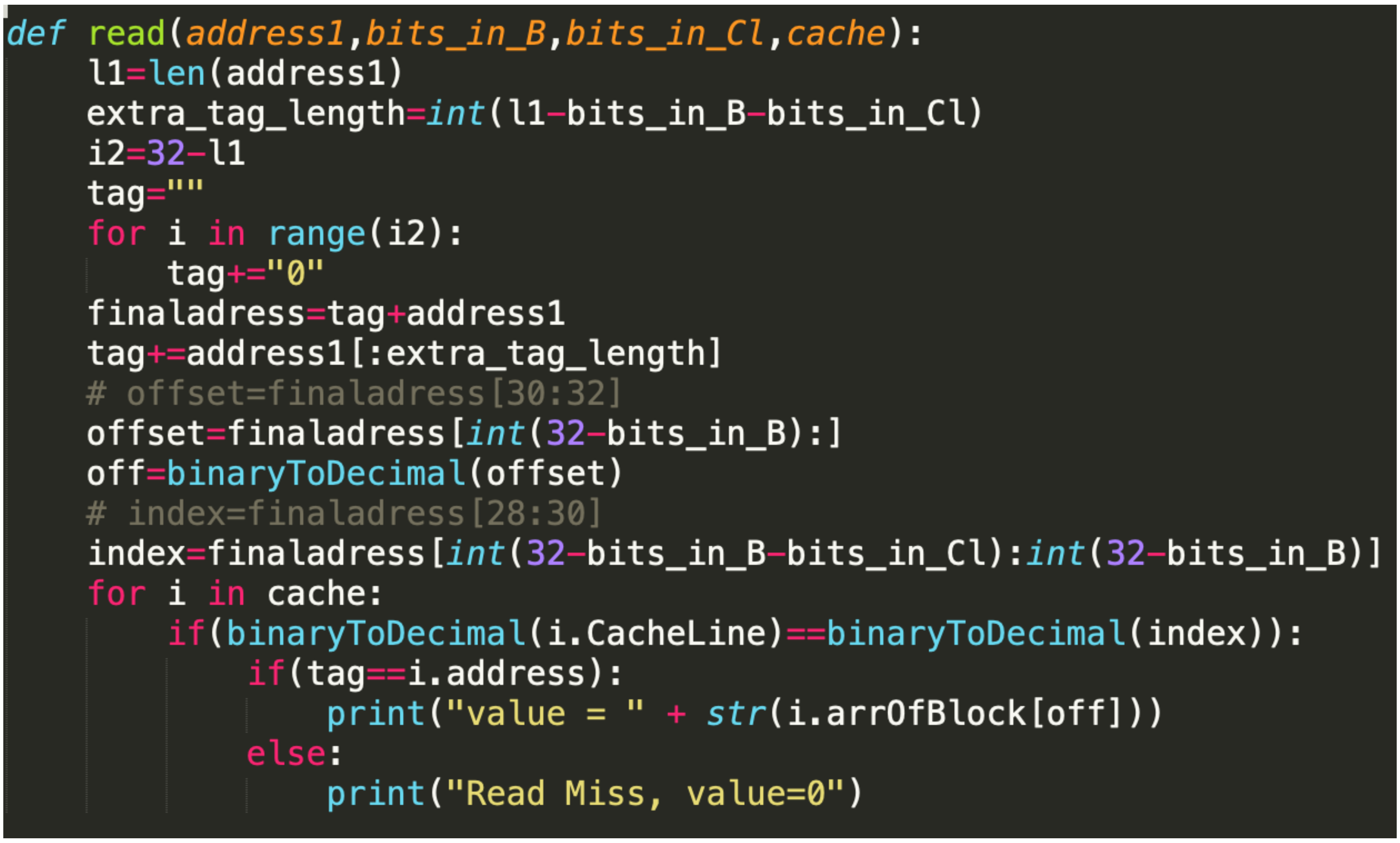


```python
def write(address1,data1,bits_in_B,bits_in_Cl,cache,B):
    l1=len(address1)
    extra_tag_length=int(l1-bits_in_B-bits_in_Cl)
    i2=32-l1
    tag=""
    for i in range(i2):
        tag+="0"
    finaladress=tag+address1
    tag+=address1[:extra_tag_length]
    # offset=finaladress[30:32]
    offset=finaladress[int(32-bits_in_B):]
    off=binaryToDecimal(offset)
    # index=finaladress[28:30]
    index=finaladress[int(32-bits_in_B-bits_in_Cl):int(32-bits_in_B)]
    for i in cache:
        if(binaryToDecimal(i.CacheLine)==binaryToDecimal(index)):
            if(tag==i.address):
                print("WRITE HIT!")
                i.arrOfBlock[off]=data1
            else:
                print("WRITE MISS :(")
                for del1 in range(B):
                    i.arrOfBlock[del1]=0
                i.address=tag
                i.arrOfBlock[off]=data1
    for i in cache:
        print("Adress Of Block : " + i.address)
        print("Index : " + i.CacheLine)
        for k in range(len(i.arrOfBlock)):
            print("offset value = " + decimalToBinary(k)+" , value = " +str(i.arrOfBlock[k]))
```

The above function first separates out the inputs in our 32 bit system (tag, block offset and cache line) . It first locates the cache line in which the write operation has to be made. We first search for the cache line in which the data is to be stored in our cache. If the tag address contained in the cache line is the same as that of our input, we do a write hit and we store our the data in the cache line. If the tag address contained in the cache is different , we do a replacement of the blocks in that particular cache line and store the data in our new block. This is known as a write miss.



```
[(base) Rishits-MacBook-Pro-6:Desktop rishitgupta$ python3 directmapping.py
Enter the size of your cache:
16
Enter the number of cache lines:
4
Enter the block size:
4
1> Read, 2> Write, 3> Exit
2
Enter address:
1101
Enter data:
6
WRITE HIT!
Adress Of Block : 0000000000000000000000000000
Index : 0
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 0000000000000000000000000000
Index : 1
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 0000000000000000000000000000
Index : 10
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 0000000000000000000000000000
Index : 11
offset value = 0 , value = 0
offset value = 1 , value = 6
offset value = 10 , value = 0
offset value = 11 , value = 0
1> Read, 2> Write, 3> Exit
2
Enter address:
10101
Enter data:
69
WRITE MISS :(
Adress Of Block : 0000000000000000000000000000
Index : 0
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 0000000000000000000000000001
Index : 1
offset value = 0 , value = 0
offset value = 1 , value = 69
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 0000000000000000000000000000
Index : 10
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 0000000000000000000000000000
Index : 11
offset value = 0 , value = 0
offset value = 1 , value = 6
offset value = 10 , value = 0
```

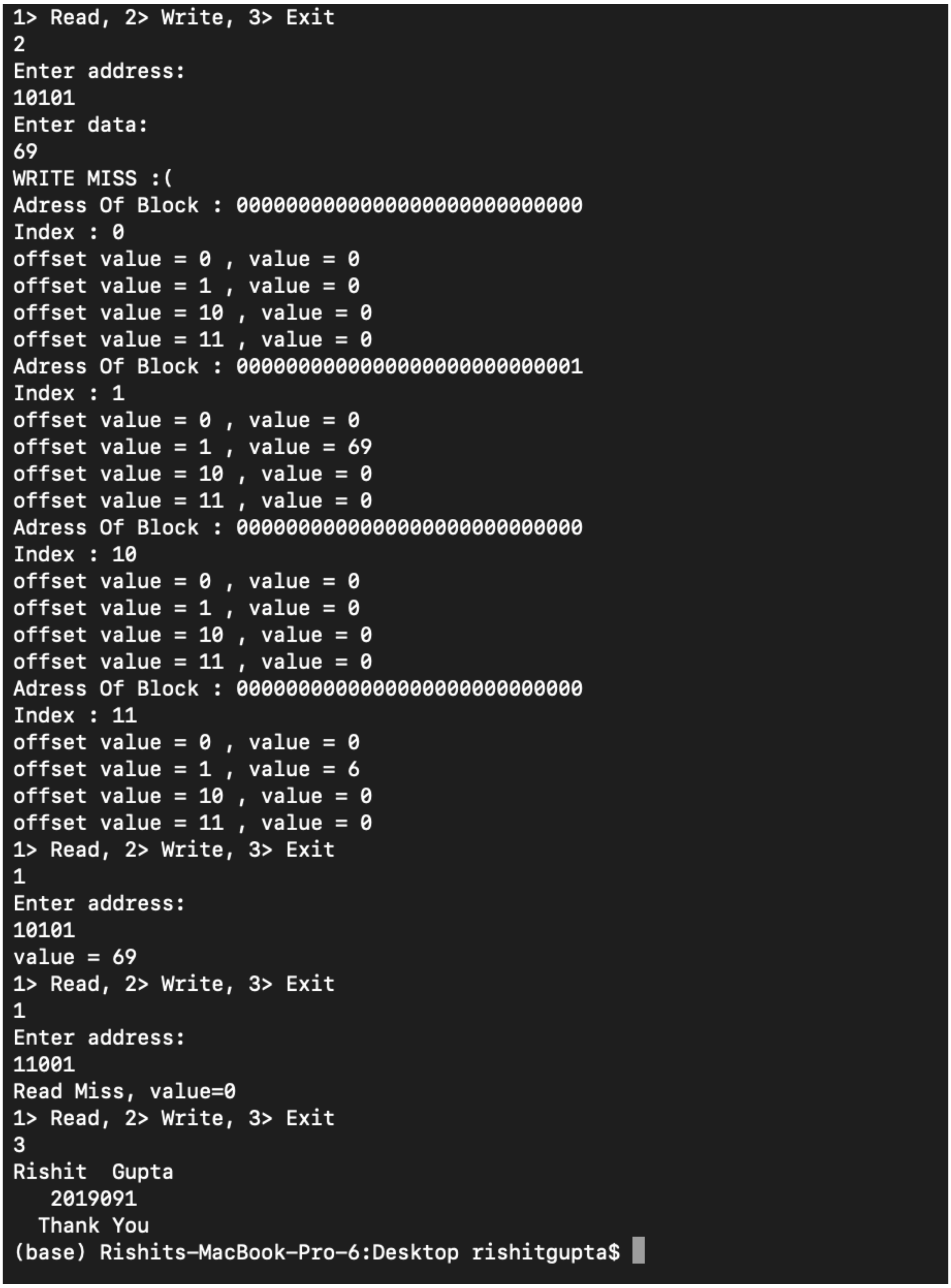


```python
def read(address1,bits_in_B,bits_in_Cl,cache):
    l1=len(address1)
    extra_tag_length=int(l1-bits_in_B-bits_in_Cl)
    i2=32-l1
    tag=""
    for i in range(i2):
        tag+="0"
    finaladress=tag+address1
    tag+=address1[:extra_tag_length]
    # offset=finaladress[30:32]
    offset=finaladress[int(32-bits_in_B):]
    off=binaryToDecimal(offset)
    # index=finaladress[28:30]
    index=finaladress[int(32-bits_in_B-bits_in_Cl):int(32-bits_in_B)]
    for i in cache:
        if(binaryToDecimal(i.CacheLine)==binaryToDecimal(index)):
            if(tag==i.address):
                print("value = " + str(i.arrOfBlock[off]))
            else:
                print("Read Miss, value=0")
```

The above method is used to read data in a Direct Mapping Cache. The user gives an address as input. The function returns the data at the address location given by the user. It first separates out the inputs in our 32 bit system(tag, block offset and cache line). We first locate the cache line belonging to the given address. If the block mentioned in our data is present in our cache line, we print the value present in the offer of the block in our cache. If the block which the address is referring too does not belong to our cache, we report a read miss.

I have maintained a cache array which contains objects of the block class.

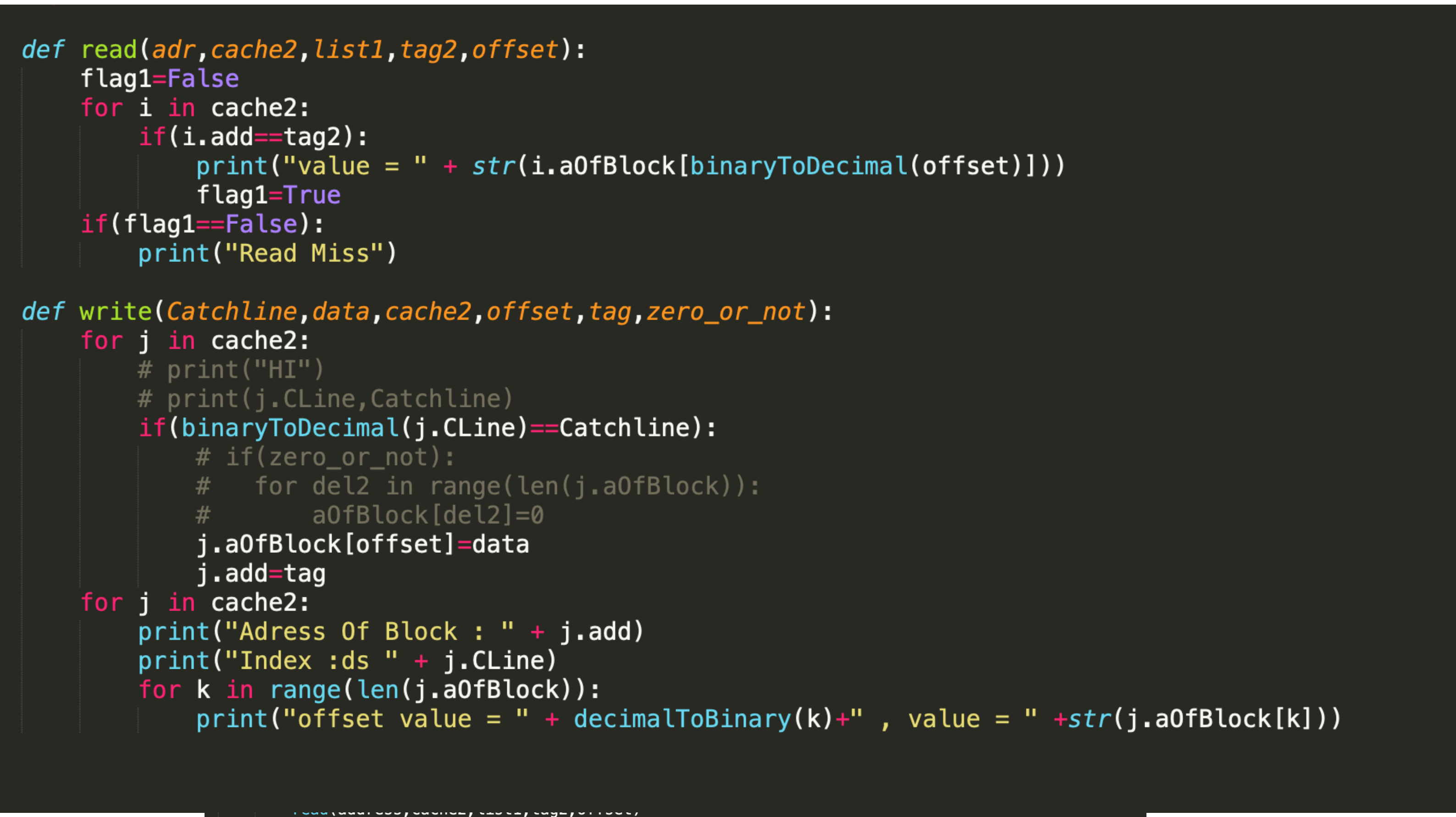
The drawback of directive mapping is that each block of the main memory maps to a fixed location in the cache; therefore, if two different blocks map to the same location in cache and they are continually referenced, the two blocks will be continually swapped in and out.



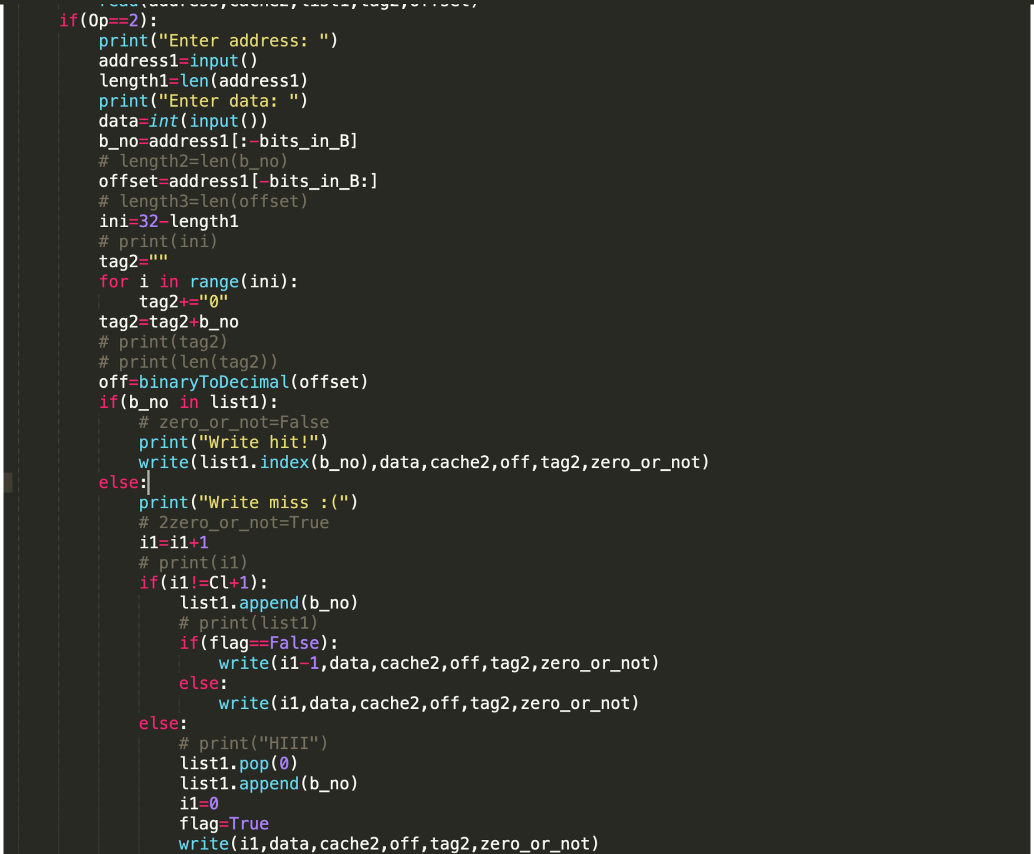
```
1> Read, 2> Write, 3> Exit
2
Enter address:
10101
Enter data:
69
WRITE MISS :(
Adress Of Block : 0000000000000000000000000000
Index : 0
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 0000000000000000000000000001
Index : 1
offset value = 0 , value = 0
offset value = 1 , value = 69
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 0000000000000000000000000000
Index : 10
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 0000000000000000000000000000
Index : 11
offset value = 0 , value = 0
offset value = 1 , value = 6
offset value = 10 , value = 0
offset value = 11 , value = 0
1> Read, 2> Write, 3> Exit
1
Enter address:
10101
value = 69
1> Read, 2> Write, 3> Exit
1
Enter address:
11001
Read Miss, value=0
1> Read, 2> Write, 3> Exit
3
Rishit  Gupta
   2019091
  Thank You
(base) Rishits-MacBook-Pro-6:Desktop rishitgupta$
```

2) *Associative Memory :*

A fully associative cache permits data to be stored in any cache block, instead of forcing each memory address into one particular block. Read and write operations are implemented as shown below :



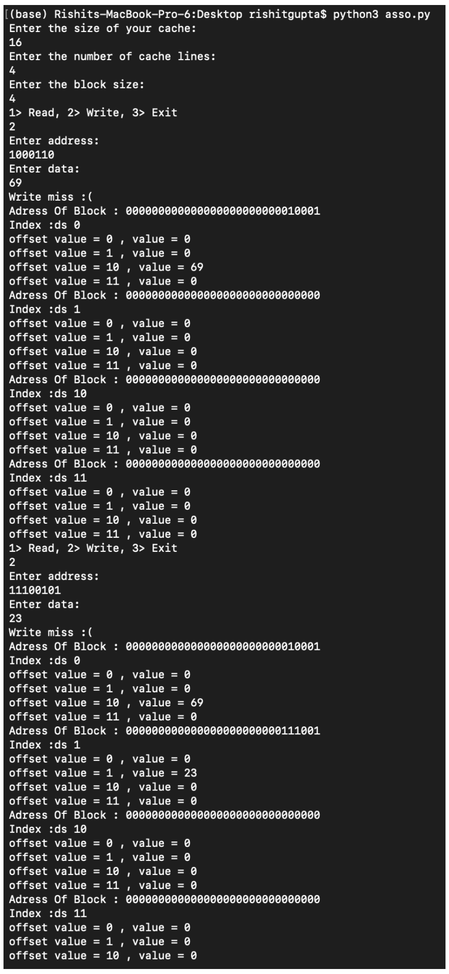
```python
def read(adr,cache2,list1,tag2,offset):
    flag1=False
    for i in cache2:
        if(i.add==tag2):
            print("value = " + str(i.aOfBlock[binaryToDecimal(offset)]))
            flag1=True
    if(flag1==False):
        print("Read Miss")

def write(Catchline,data,cache2,offset,tag,zero_or_not):
    for j in cache2:
        # print("HI")
        # print(j.CLine,Catchline)
        if(binaryToDecimal(j.CLine)==Catchline):
            # if(zero_or_not):
            #   for del2 in range(len(j.aOfBlock)):
            #       aOfBlock[del2]=0
            j.aOfBlock[offset]=data
            j.add=tag
    for j in cache2:
        print("Adress Of Block : " + j.add)
        print("Index :ds " + j.CLine)
        for k in range(len(j.aOfBlock)):
            print("offset value = " + decimalToBinary(k)+" , value = " +str(j.aOfBlock[k]))
```



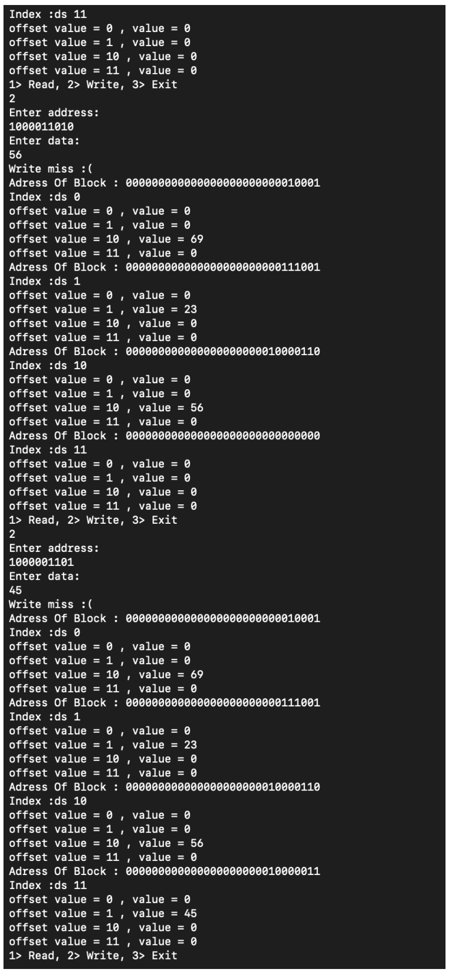
```python
if(Op==2):
    print("Enter address: ")
    address1=input()
    length1=len(address1)
    print("Enter data: ")
    data=int(input())
    b_no=address1[:-bits_in_B]
    # length2=len(b_no)
    offset=address1[-bits_in_B:]
    # length3=len(offset)
    ini=32-length1
    # print(ini)
    tag2=""
    for i in range(ini):
        tag2+="0"
    tag2=tag2+b_no
    # print(tag2)
    # print(len(tag2))
    off=binaryToDecimal(offset)
    if(b_no in list1):
        # zero_or_not=False
        print("Write hit!")
        write(list1.index(b_no),data,cache2,off,tag2,zero_or_not)
    else:
        print("Write miss :(")
        # 2zero_or_not=True
        i1=i1+1
        # print(i1)
        if(i1!=Cl+1):
            list1.append(b_no)
            # print(list1)
            if(flag==False):
                write(i1-1,data,cache2,off,tag2,zero_or_not)
            else:
                write(i1,data,cache2,off,tag2,zero_or_not)
        else:
            # print("HIII")
            list1.pop(0)
            list1.append(b_no)
            i1=0
            flag=True
            write(i1,data,cache2,off,tag2,zero_or_not)
```

In the write function, we write data into our cache lines with no restrictions. We keep writing in the cache lines serially until all the cache lines are used up. When the cache contains data in all its cache lines we follow the FIFO convention to write data in our cache lines.

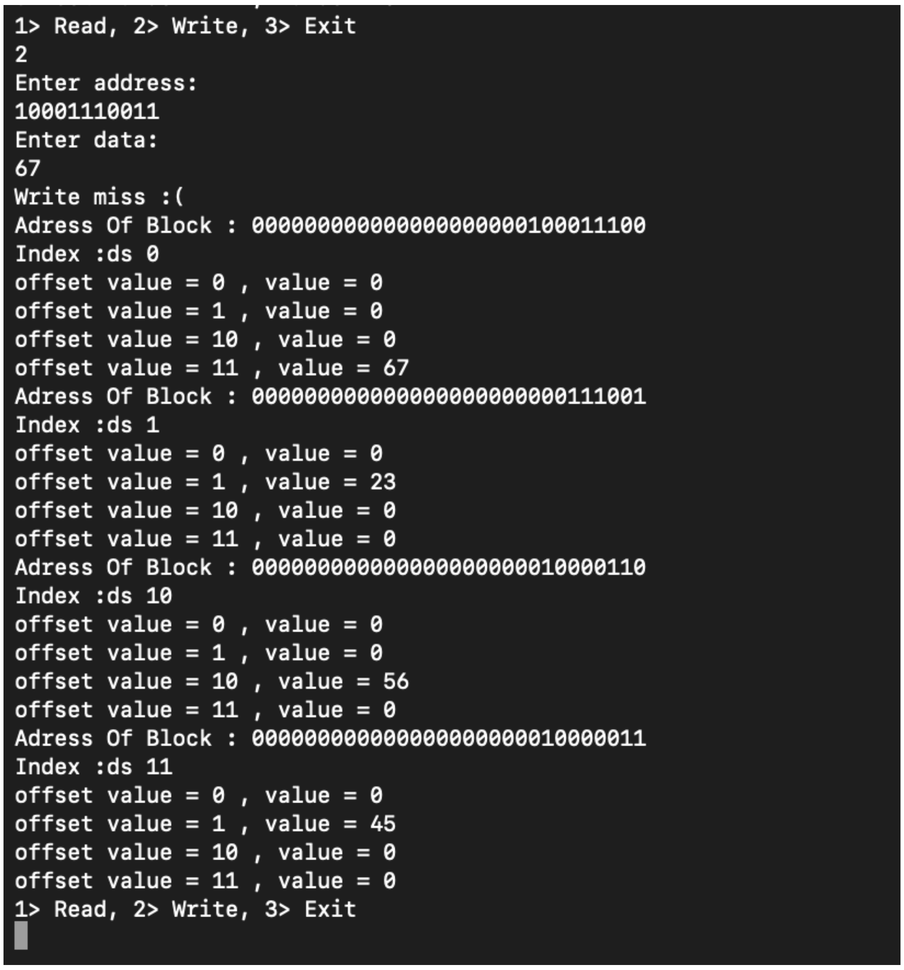
To read from the cache, we compare the address of our instruction with the address of each word in our block in our cache line. If we find the address of the instruction we are looking for in our cache, we print out the value of the data in that particular instruction.



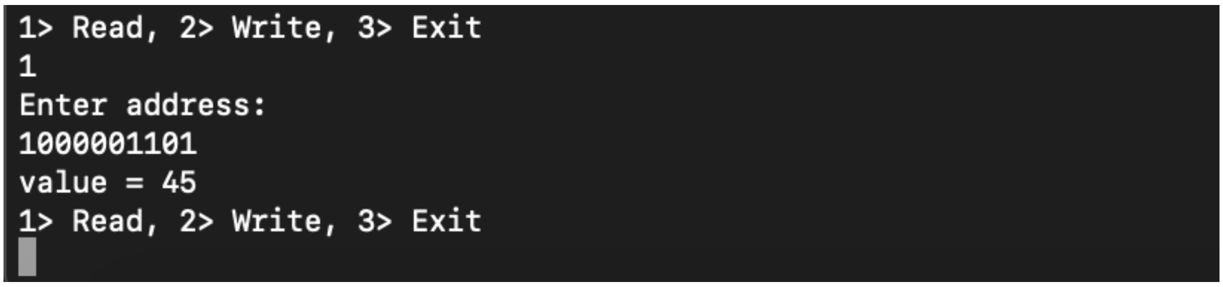
```
(base) Rishits-MacBook-Pro-6:Desktop rishitgupta$ python3 asso.py
Enter the size of your cache:
16
Enter the number of cache lines:
4
Enter the block size:
4
1> Read, 2> Write, 3> Exit
2
Enter address:
1000110
Enter data:
69
Write miss :(
Adress Of Block : 000000000000000000000000010001
Index :ds 0
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 69
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000000000
Index :ds 1
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000000000
Index :ds 10
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000000000
Index :ds 11
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
1> Read, 2> Write, 3> Exit
2
Enter address:
11100101
Enter data:
23
Write miss :(
Adress Of Block : 000000000000000000000000010001
Index :ds 0
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 69
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000111001
Index :ds 1
offset value = 0 , value = 0
offset value = 1 , value = 23
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000000000
Index :ds 10
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000000000
Index :ds 11
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
```



```
Index :ds 11
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
1> Read, 2> Write, 3> Exit
2
Enter address:
1000011010
Enter data:
56
Write miss :(
Adress Of Block : 000000000000000000000000010001
Index :ds 0
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 69
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000111001
Index :ds 1
offset value = 0 , value = 0
offset value = 1 , value = 23
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000010000110
Index :ds 10
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 56
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000000000
Index :ds 11
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
1> Read, 2> Write, 3> Exit
2
Enter address:
1000001101
Enter data:
45
Write miss :(
Adress Of Block : 000000000000000000000000010001
Index :ds 0
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 69
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000111001
Index :ds 1
offset value = 0 , value = 0
offset value = 1 , value = 23
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000010000110
Index :ds 10
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 56
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000010000011
Index :ds 11
offset value = 0 , value = 0
offset value = 1 , value = 45
offset value = 10 , value = 0
offset value = 11 , value = 0
1> Read, 2> Write, 3> Exit
```



```
1> Read, 2> Write, 3> Exit
2
Enter address:
10001110011
Enter data:
67
Write miss :(
Adress Of Block : 00000000000000000000100011100
Index :ds 0
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 67
Adress Of Block : 00000000000000000000000111001
Index :ds 1
offset value = 0 , value = 0
offset value = 1 , value = 23
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 00000000000000000000010000110
Index :ds 10
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 56
offset value = 11 , value = 0
Adress Of Block : 00000000000000000000010000011
Index :ds 11
offset value = 0 , value = 0
offset value = 1 , value = 45
offset value = 10 , value = 0
offset value = 11 , value = 0
1> Read, 2> Write, 3> Exit
```



```
1> Read, 2> Write, 3> Exit
1
Enter address:
1000001101
value = 45
1> Read, 2> Write, 3> Exit
```

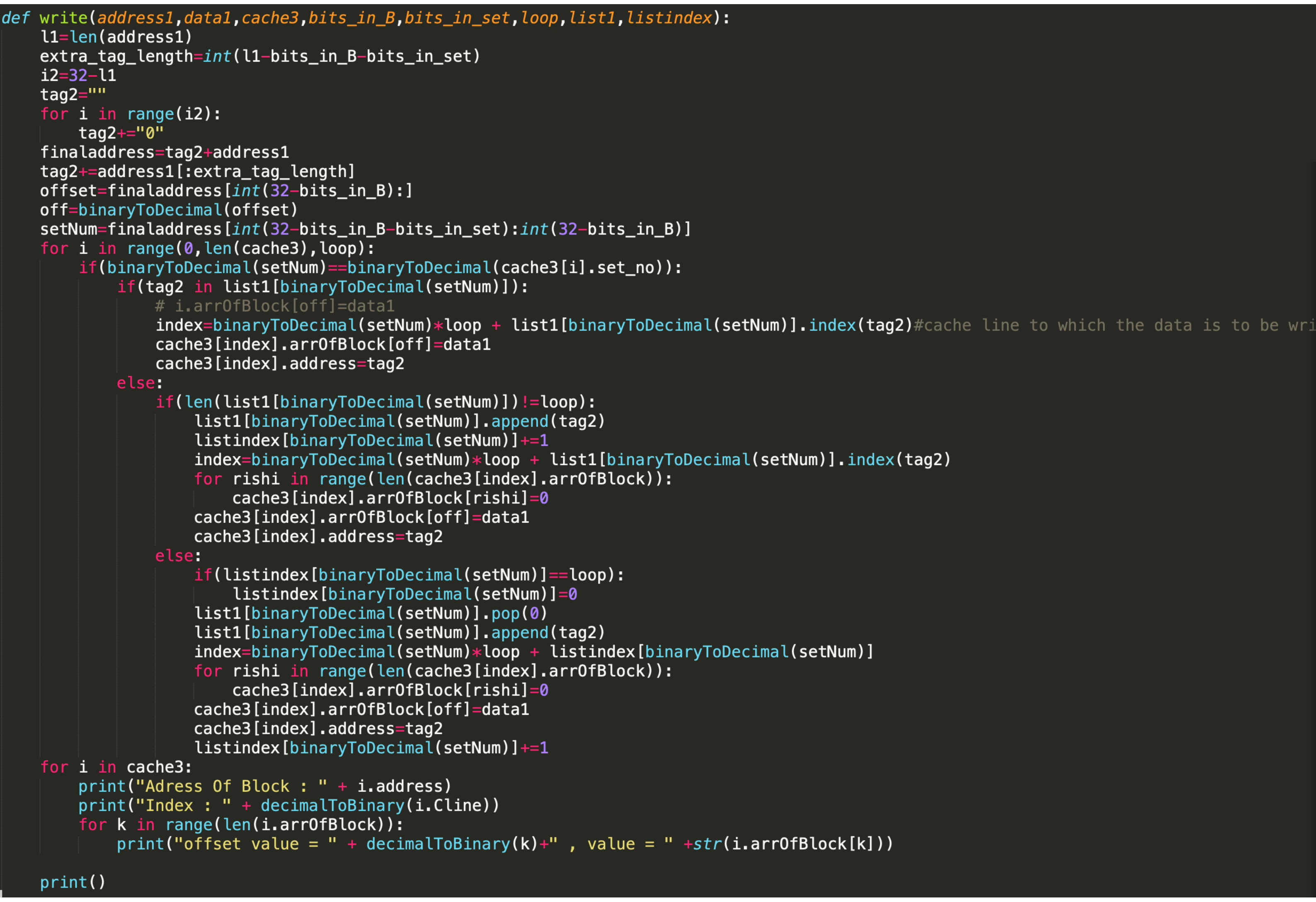
3) N-way Set Associative Memory

In an N-way Set Associative Memory, our cache is divided into n number of sets. Each set contains equal number of cache lines and each of them uses the Associative Memory Cache to load and search data. Read and write operations are implemented as shown below :

The above method is used to write data in a N-way set associative memory. The user gives the data and the address in which the data is to be stored as inputs.

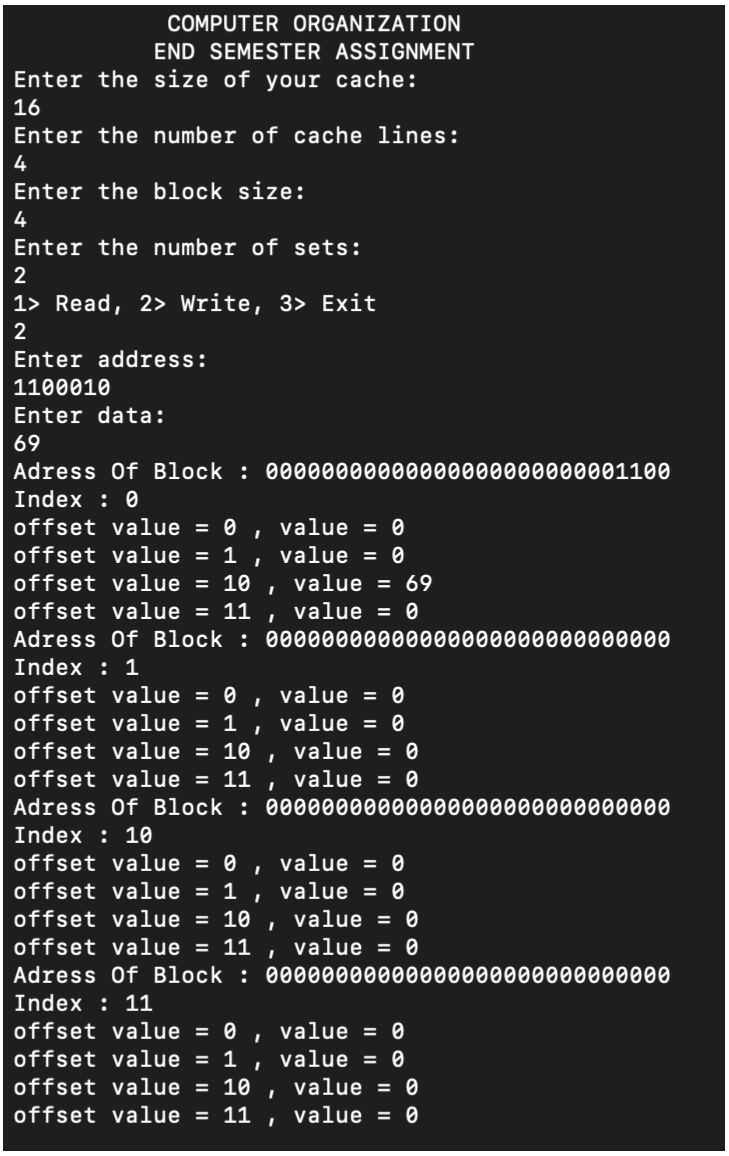
N-way Set Associative Memory Address Format

Tag	Set_Num	B_offset

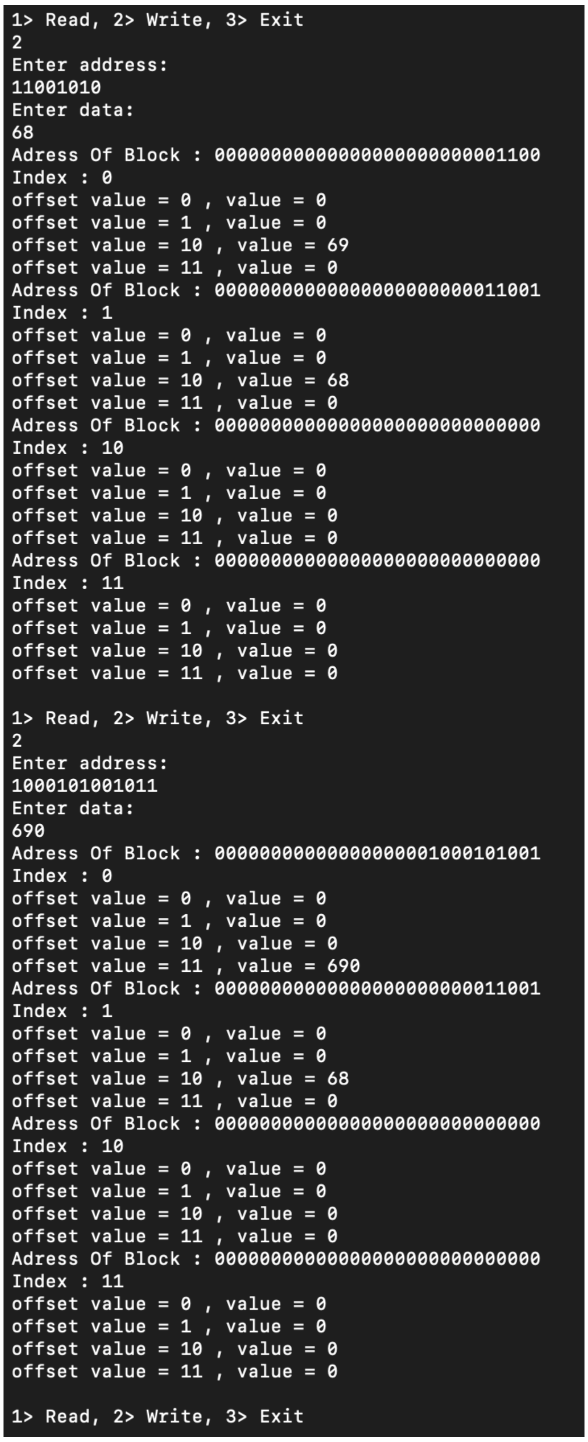


```
def write(address1,data1,cache3,bits_in_B,bits_in_set,loop,list1,listindex):
    l1=len(address1)
    extra_tag_length=int(l1-bits_in_B-bits_in_set)
    i2=32-l1
    tag2=""
    for i in range(i2):
        tag2+="0"
    finaladdress=tag2+address1
    tag2+=address1[:extra_tag_length]
    offset=finaladdress[int(32-bits_in_B):]
    off=binaryToDecimal(offset)
    setNum=finaladdress[int(32-bits_in_B-bits_in_set):int(32-bits_in_B)]
    for i in range(0,len(cache3),loop):
        if(binaryToDecimal(setNum)==binaryToDecimal(cache3[i].set_no)):
            if(tag2 in list1[binaryToDecimal(setNum)]):
                # i.arrOfBlock[off]=data1
                index=binaryToDecimal(setNum)*loop + list1[binaryToDecimal(setNum)].index(tag2)#cache line to which the data is to be wri
                cache3[index].arrOfBlock[off]=data1
                cache3[index].address=tag2
            else:
                if(len(list1[binaryToDecimal(setNum)])!=loop):
                    list1[binaryToDecimal(setNum)].append(tag2)
                    listindex[binaryToDecimal(setNum)]+=1
                    index=binaryToDecimal(setNum)*loop + list1[binaryToDecimal(setNum)].index(tag2)
                    for rishi in range(len(cache3[index].arrOfBlock)):
                        cache3[index].arrOfBlock[rishi]=0
                    cache3[index].arrOfBlock[off]=data1
                    cache3[index].address=tag2
                else:
                    if(listindex[binaryToDecimal(setNum)]==loop):
                        listindex[binaryToDecimal(setNum)]=0
                    list1[binaryToDecimal(setNum)].pop(0)
                    list1[binaryToDecimal(setNum)].append(tag2)
                    index=binaryToDecimal(setNum)*loop + listindex[binaryToDecimal(setNum)]
                    for rishi in range(len(cache3[index].arrOfBlock)):
                        cache3[index].arrOfBlock[rishi]=0
                    cache3[index].arrOfBlock[off]=data1
                    cache3[index].address=tag2
                    listindex[binaryToDecimal(setNum)]+=1
    for i in cache3:
        print("Adress Of Block : " + i.address)
        print("Index : " + decimalToBinary(i.Cline))
        for k in range(len(i.arrOfBlock)):
            print("offset value = " + decimalToBinary(k)+" , value = " +str(i.arrOfBlock[k]))

    print()
```

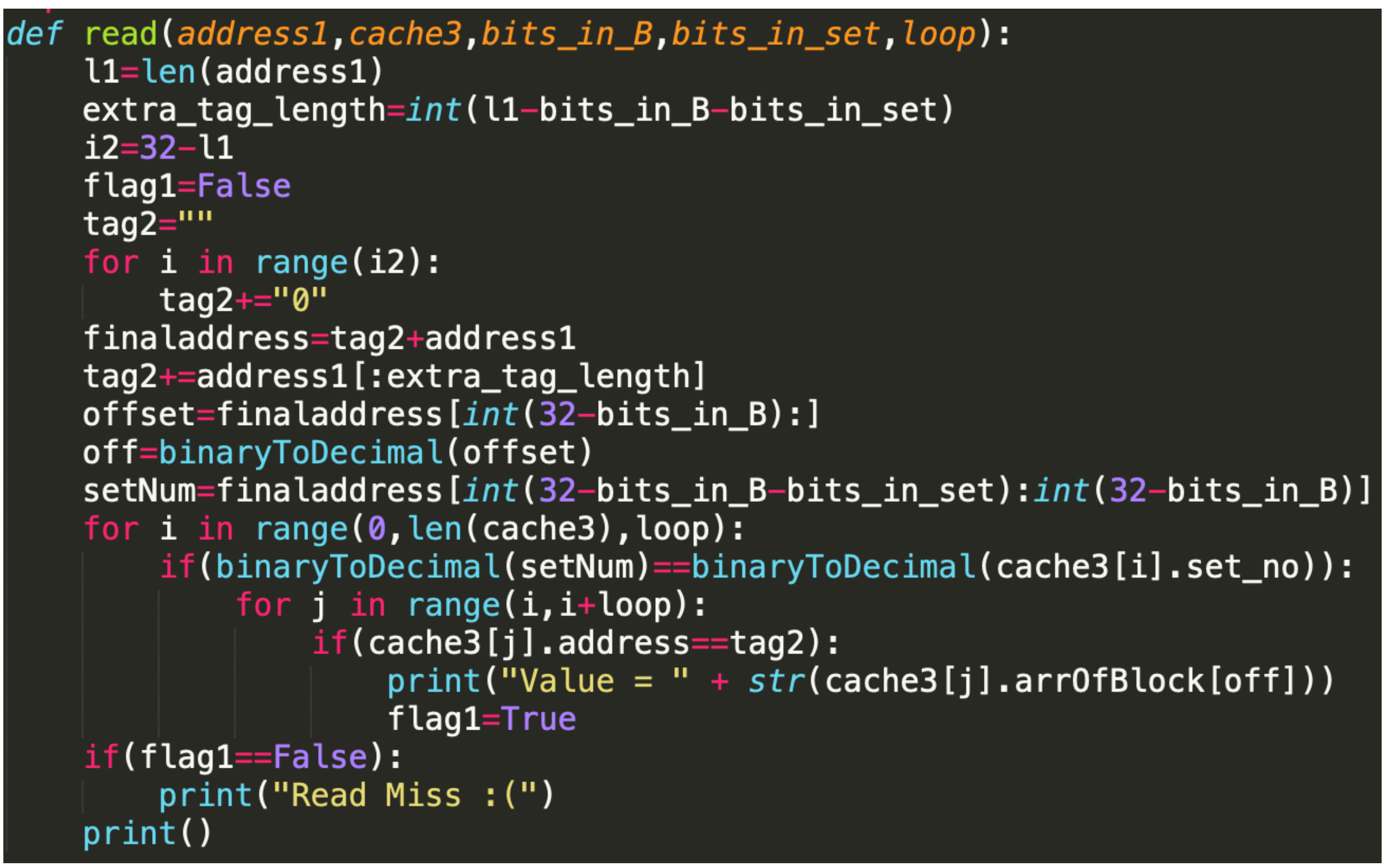
The above function first separates out the inputs in our 32 bit system.It first locates the set in which the write operation has to be made. We first search for the set in which the data is to be stored. If the address to which the data has to be written is already present in our cache, the data is overwritten or written in another word of the same block according to our input address. If it is not present in our cache, we store the data in some other cache line in our set. In case the set is already full, we follow the FIFO convention to add new data members in our cache. Since we our not maintaining a main memory the address which is removed loses its data. We print the our cache after every input which is given by the user



```
            COMPUTER ORGANIZATION
           END SEMESTER ASSIGNMENT
Enter the size of your cache:
16
Enter the number of cache lines:
4
Enter the block size:
4
Enter the number of sets:
2
1> Read, 2> Write, 3> Exit
2
Enter address:
1100010
Enter data:
69
Adress Of Block : 000000000000000000000000001100
Index : 0
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 69
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000000000
Index : 1
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000000000
Index : 10
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000000000
Index : 11
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
```

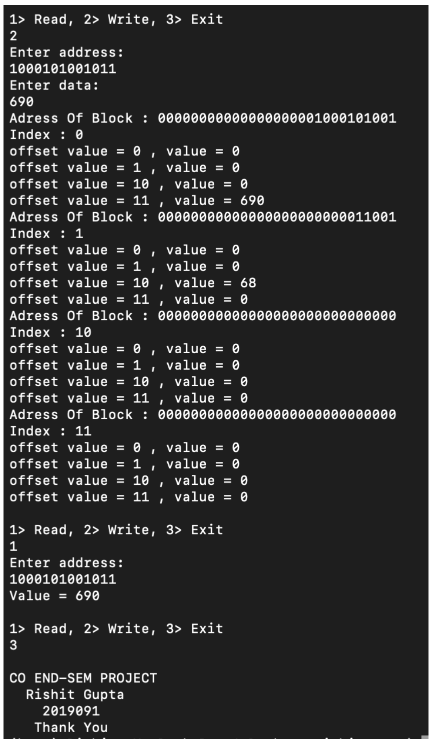


```
1> Read, 2> Write, 3> Exit
2
Enter address:
11001010
Enter data:
68
Adress Of Block : 000000000000000000000000001100
Index : 0
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 69
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000011001
Index : 1
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 68
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000000000
Index : 10
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000000000
Index : 11
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0

1> Read, 2> Write, 3> Exit
2
Enter address:
1000101001011
Enter data:
690
Adress Of Block : 000000000000000000001000101001
Index : 0
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 690
Adress Of Block : 000000000000000000000000011001
Index : 1
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 68
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000000000
Index : 10
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000000000
Index : 11
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0

1> Read, 2> Write, 3> Exit
```



```python
def read(address1,cache3,bits_in_B,bits_in_set,loop):
    l1=len(address1)
    extra_tag_length=int(l1-bits_in_B-bits_in_set)
    i2=32-l1
    flag1=False
    tag2=""
    for i in range(i2):
        tag2+="0"
    finaladdress=tag2+address1
    tag2+=address1[:extra_tag_length]
    offset=finaladdress[int(32-bits_in_B):]
    off=binaryToDecimal(offset)
    setNum=finaladdress[int(32-bits_in_B-bits_in_set):int(32-bits_in_B)]
    for i in range(0,len(cache3),loop):
        if(binaryToDecimal(setNum)==binaryToDecimal(cache3[i].set_no)):
            for j in range(i,i+loop):
                if(cache3[j].address==tag2):
                    print("Value = " + str(cache3[j].arrOfBlock[off]))
                    flag1=True
    if(flag1==False):
        print("Read Miss :(")
    print()
```

The above method is used to read data in a N-way set associative memory. The user gives an address as input. The function returns the data at the address location given by the user. It first separates out the inputs in our 32 bit system. It first locates the set from which the data has to be read. Once the set is located, a loop is run to check the address which matches that of the cache line in the given Set. If no data is found (the input address is not present in the cache), “Read Miss” is printed.



```
1> Read, 2> Write, 3> Exit
2
Enter address:
1000101001011
Enter data:
690
Adress Of Block : 000000000000000000001000101001
Index : 0
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 690
Adress Of Block : 000000000000000000000000011001
Index : 1
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 68
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000000000
Index : 10
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0
Adress Of Block : 000000000000000000000000000000
Index : 11
offset value = 0 , value = 0
offset value = 1 , value = 0
offset value = 10 , value = 0
offset value = 11 , value = 0

1> Read, 2> Write, 3> Exit
1
Enter address:
1000101001011
Value = 690

1> Read, 2> Write, 3> Exit
3

CO END-SEM PROJECT
  Rishit Gupta
    2019091
   Thank You
```