Ezeuko Emmanuel – MZQ5FK

Homework 4 – Cache

1. Assuming a direct-mapped cache with one-word blocks and a total size of 16 blocks, list if each reference is a hit or a miss assuming the cache is initially filled with word address 0, 1, 2, . . . 15 memory data. Show the state of the cache after the last reference. Calculate the hit rate for this reference string. (Correct answer: hit rate = 4/12)

Solution

Cache Configuration:

- **Cache size**: 16 blocks.
- **Block size**: 1 word per block.
- **Address format**: Each word address has a unique block in the cache (direct-mapped).
- The cache is initially filled with word addresses 0 to 15.

Initial cache content (word addresses):
Blocks:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

Accessing the sequence:

- Word address sequence: 1, 18, 2, 3, 4, 20, 5, 21, 33, 34, 1, 4
- For each access, determine whether it is a **hit** or **miss**, and update the cache as needed.

- The block address column contains list of block address we need to access.
- The cache index refers to the cache index of block = block address % 16.
- Hit = block found
- Miss = block not found
- Cache state after access shows the block address stored in the cache.

For a 16-block direct mapped cache, will have **16 sets or cache line**
we use the last four bits as the set index and the remaining bits before them as the tag. Which is represented in the example below for an address e.g **18**.
Set index =>
18 % 16 = 2
18 => 01 **0010**
It belongs to set index 2

To calculate the tag we divide the address by the set = 16:
18 / 16 = 1
18 => 01 0010
The tag is 1.

addr	Tag	**Cache index**	**Hit/Miss**	**Cache state after access**
1	00	**0001**	hit	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
18	01	**0010**	miss	[0, 1, 18, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
2	00	**0010**	miss	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
3	00	**0011**	hit	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
4	00	**0100**	hit	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
20	01	**0100**	miss	[0, 1, 2, 3, 20, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
5	00	**0101**	hit	[0, 1, 2, 3, 20, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
21	01	**0101**	miss	[0, 1, 2, 3, 20, 21, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
33	10	**0001**	miss	[0, 33, 2, 3, 20, 21, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
34	10	**0010**	miss	[0, 33, 34, 3, 20, 21, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

1	00	**0001**	miss	[0, 1, 34, 3, 20, 21, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
4	00	**0100**	miss	[0, 1, 34, 3, 4, 21, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

Hit Rate Calculation:

- **Total accesses**: 12
- **Hits block address** : 4 (1, 3, 4, 5)
- **Misses**: 8

Hit Rate:

Hit Rate = $\frac{total\ hit}{total\ access} = \frac{\mathbf{4}}{\mathbf{12}}$

2. Now, assuming a direct-mapped cache with two-word blocks and a total size of 8 blocks, list if each reference is a hit or a miss assuming the cache is initially empty. Show the state of the cache after the last reference.
Calculate hit rate for this reference string. (Correct answer: hit rate = 1/12)

Solution

In a two-word cache block, the address is used to align with 8 bytes, for a total of 8 blocks. This means to get the cache index or set index:

Set index = $(\frac{\text{address}}{\text{block size}})\ \%\ \text{set}$

Tag = $\frac{\text{address}}{(\text{set} * \text{block size})}$

No of word per block = 2

Set = 8 blocks which can be represented with 3 bits

E.g

Set index = 21 **=>** $\frac{21}{2} \% 8$ **= 2**

Tag = = 21 **=>** $\frac{\mathbf{21}}{(\mathbf{8} * \mathbf{2}\,)}$ **= 1**

i.e For the 8 block set, we use the next 3 bits after the LSB to represent it.

21 = 1**010**1 here **010** is the set index, tag = 1

The last bit of the address is not used, the set index started after the LSB

addr	**Tag**	**Set index**	**Hit/ Miss**	**Cache state after access**
1	00	**000**	miss	[(0, 1), (), (), (), (), (), (), ()]
18	01	**001**	miss	[(0, 1), (18,19), (), (), (), (), (), ()]
2	00	**001**	miss	[(0, 1), (2,3), (), (), (), (), (), ()]
3	00	**001**	hit	[(0, 1), (2,3), (), (), (), (), (), ()]
4	00	**010**	miss	[(0, 1), (2,3), (4,5), (), (), (), (), ()]
20	01	**010**	miss	[(0, 1), (2,3), (20,21), (), (), (), (), ()]
5	00	**010**	miss	[(0, 1), (2,3), (4,5), (), (), (), (), ()]

21	01	**010**	miss	[(0, 1), (2,3), (20,21), (), (), (), (), ()]
33	10	**000**	miss	[(32,33), (2,3), (20,21), (), (), (), (), ()]
34	10	**001**	miss	[(32,33), (34,35), (20,21), (), (), (), (), ()]
1	00	**000**	miss	[(0,1), (34,35), (20,21), (), (), (), (), ()]
4	00	**010**	miss	[(0,1), (34,35), (4,5), (), (), (), (), ()]

Hit rate = 1 / 12

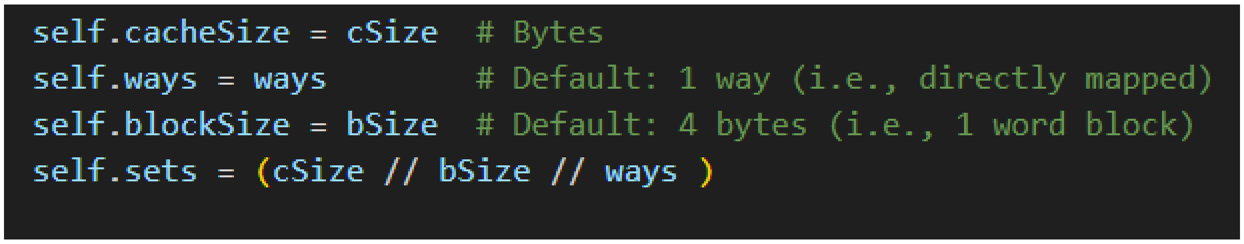
Description of functions in code

To calculate the set, we use the formula:

Set = $\frac{\text{cache size}}{\text{block size} * \text{no of ways}}$

Total cache size divided by block size, gives the total number of blocks in the cache.

Then dividing with the number of ways gives the total set in the cache.



```
self.cacheSize = cSize  # Bytes
self.ways = ways        # Default: 1 way (i.e., directly mapped)
self.blockSize = bSize  # Default: 4 bytes (i.e., 1 word block)
self.sets = (cSize // bSize // ways )
```

1. find_set (address):

This function calculates the **set index** based on the string address using the following formula we derived in the solution above:

Set index =$(\frac{\text{address}}{\text{block size}})\ \%\ \text{set}$

E.g assume cache address = 21 in byte address, for a 8 block set, and block size is 2 bytes. For the 8 block set, we use the next 3 bits after the LSB to represent it.

Set index = 21 **=>** $\frac{21}{2}\ \%\ 8$ **=** **2**

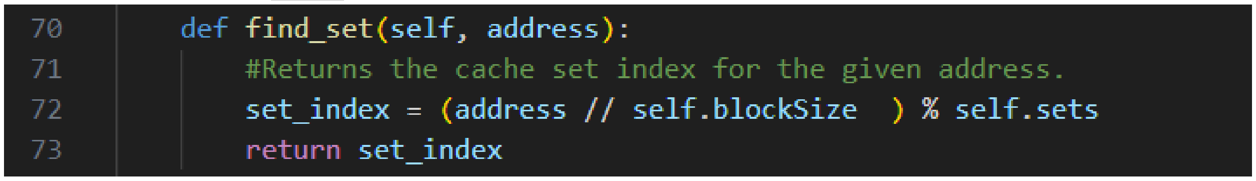
1**010**1 => **2**

Tag = = 21 **=>** $\frac{\mathbf{21}}{(\mathbf{8} * \mathbf{2})}$ **= 1**

21 = 1**010**1 here **010** is the set index, tag = 1

Note, the address and block size are in the same unit, whether bytes or word. For the code file they are in bytes.

In the formula, we first divide the address by the block size. This returns the block number where the address falls in, then multiplying it with the modulus set (%set) makes sure it doesn't exceed the set limit.



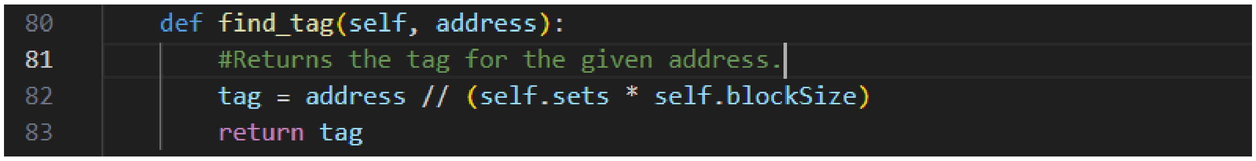
```python
    def find_set(self, address):
        #Returns the cache set index for the given address.
        set_index = (address // self.blockSize  ) % self.sets
        return set_index
```

2. find_tag (address):

The **tag** is the part of the address that helps us identify if the data stored in the cache block is the correct one for the given cache address. It's calculated by dividing the cache address by the number of sets multiplied by the block size. This will return all the bits after the set bits. The formula was derived above.

$$\text{Tag} = \frac{\text{address}}{(\text{set} * \text{ block size})}$$

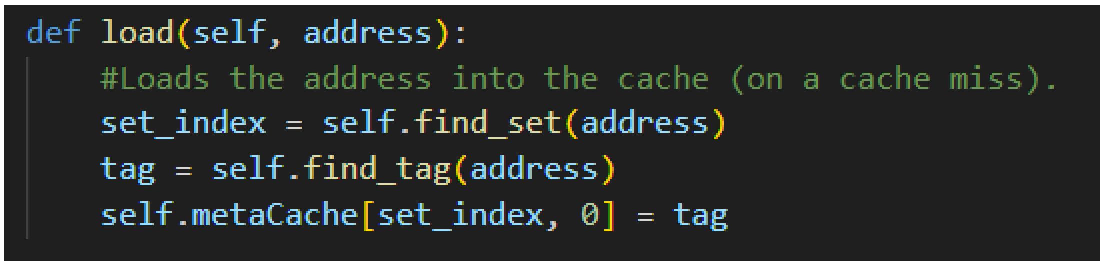
In the formula we convert the address to byte address by multiplying with four, since the block size is in bytes.



```python
    def find_tag(self, address):
        #Returns the tag for the given address.
        tag = address // (self.sets * self.blockSize)
        return tag
```

3. load(address):

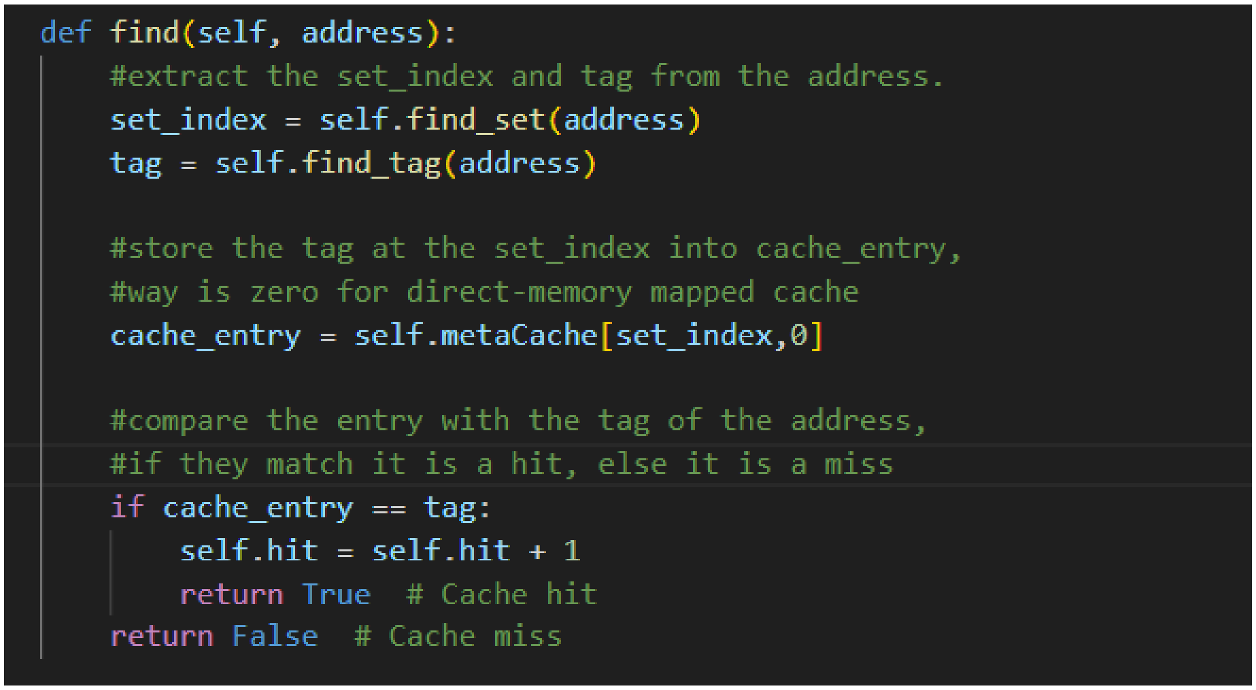
When there's a cache miss, this function loads the data from memory into the cache. It updates the cache block at the appropriate set with the new tag and data. We first extract the set and tag of the address. Then we store the tag in the Meta cache at the set index, here we have set the ways to 0, this is for direct-mapped cache, since direct-mapped cache have only one way.



```python
def load(self, address):
    #Loads the address into the cache (on a cache miss).
    set_index = self.find_set(address)
    tag = self.find_tag(address)
    self.metaCache[set_index, 0] = tag
```

4. Find (address):

This function checks if the data for the given address is currently in the cache by comparing the tag and set index. We first retrieve the set index and set tag, then check for the entry in the meta-cache , if its available then it’s a hit, and we increment our hit count else it’s a miss. If it’s a miss, the load is called to load the data from memory.



```python
def find(self, address):
    #extract the set_index and tag from the address.
    set_index = self.find_set(address)
    tag = self.find_tag(address)

    #store the tag at the set_index into cache_entry,
    #way is zero for direct-memory mapped cache
    cache_entry = self.metaCache[set_index,0]

    #compare the entry with the tag of the address,
    #if they match it is a hit, else it is a miss
    if cache_entry == tag:
        self.hit = self.hit + 1
        return True  # Cache hit
    return False  # Cache miss
```

Verification code output for direct memory

```
$ python3 sim_dm.py first.trace 16 1 4
main.py trace cacheSize(Bytes) #ofWays blockSize(Bytes)
first.trace
Processing your program trace, progress so far = 0 %
0 address

set and tag of 0x0 is 0 0
address 0x0 CACHE MISS. Loading from memory.
4 address

set and tag of 0x4 is 1 0
address 0x4 CACHE MISS. Loading from memory.
8 address

set and tag of 0x8 is 2 0
address 0x8 CACHE MISS. Loading from memory.
12 address

set and tag of 0xc is 3 0
address 0xc CACHE MISS. Loading from memory.
16 address

set and tag of 0x10 is 0 1
address 0x10 CACHE MISS. Loading from memory.
12 address
```

```

set and tag of 0xc is 3 0
address 0xc CACHE HIT. Good Job.
16 address

set and tag of 0x10 is 0 1
address 0x10 CACHE HIT. Good Job.
60 address

set and tag of 0x3c is 3 3
address 0x3c CACHE MISS. Loading from memory.
total cache misses 6
miss_rate 0.75
hit_rate 0.25
Finished processing your program trace, progress = 100.0 %
```

```
$ python3 sim_dm.py first.trace 16 1 8
main.py trace cacheSize(Bytes) #ofWays blockSize(Bytes)
first.trace
Processing your program trace, progress so far = 0 %
0 address

set and tag of 0x0 is 0 0
address 0x0 CACHE MISS. Loading from memory.
4 address

set and tag of 0x4 is 0 0
address 0x4 CACHE HIT. Good Job.
8 address

set and tag of 0x8 is 1 0
address 0x8 CACHE MISS. Loading from memory.
12 address

set and tag of 0xc is 1 0
address 0xc CACHE HIT. Good Job.
16 address

set and tag of 0x10 is 0 1
address 0x10 CACHE MISS. Loading from memory.
12 address

set and tag of 0xc is 1 0
address 0xc CACHE HIT. Good Job.
16 address

set and tag of 0x10 is 0 1
address 0x10 CACHE HIT. Good Job.
60 address

set and tag of 0x3c is 1 3
address 0x3c CACHE MISS. Loading from memory.
total cache misses 4
miss_rate 0.5
hit_rate 0.5
Finished processing your program trace, progress = 100.0 %
```

```
$ python3 sim_dm.py pingpong.trace 16 1 4
main.py trace cacheSize(Bytes) #ofWays blockSize(Bytes)
pingpong.trace
Processing your program trace, progress so far = 0 %
0 address

set and tag of 0x0 is 0 0
address 0x0 CACHE MISS. Loading from memory.
16 address

set and tag of 0x10 is 0 1
address 0x10 CACHE MISS. Loading from memory.
0 address

set and tag of 0x0 is 0 0
address 0x0 CACHE MISS. Loading from memory.
16 address

set and tag of 0x10 is 0 1
address 0x10 CACHE MISS. Loading from memory.
0 address

set and tag of 0x0 is 0 0
address 0x0 CACHE MISS. Loading from memory.
16 address

set and tag of 0x10 is 0 1
address 0x10 CACHE MISS. Loading from memory.
0 address

set and tag of 0x0 is 0 0
address 0x0 CACHE MISS. Loading from memory.
16 address

set and tag of 0x10 is 0 1
address 0x10 CACHE MISS. Loading from memory.
total cache misses 8
miss_rate 1.0
hit_rate 0.0
Finished processing your program trace, progress = 100.0 %
```

PART 2 – set associative cache

2. Below is a sequence of twelve 32-bit memory address references given as word addresses. 1, 18, 2, 3, 4, 20, 5, 21, 33, 34, 1, 4. Note that word and byte addresses are different. The word address appended by offset (two zeros) gives you the byte address. In other words, word addresses are multiples of 4. For example, if the word address is"1", the equivalent byte address is"4" (100). Assuming a set-associative cache with two ways, two-word blocks and a total size of 8 blocks, list if each reference is a hit or a miss assuming the cache is initially empty (assume LRU replacement). Show the state of the cache after the last reference. Calculate the hit rate for this reference string. (Correct answer: hit rate = 5 / 12)

Part 2: set associative Cache with Two-word Blocks (8 blocks total)

1. **Cache Configuration:**

2-way set-associative: Each set has 2 ways (slots).
2-word blocks: Each block holds 2 words.
8 blocks

Mode; Least recently used cache.

Calculate the cache size:

1 word = 4 bytes

1 block **= 2 words = 8 bytes**

8 block cache = 8 * 8 = **64 bytes**

The cache is a **64** byte cache

No of ways = **2**

Set = $\frac{\text{cache size}}{\text{block size} * \text{no of ways}}$

Set = $\frac{64}{8 * 2}$ = 4 sets

Set index = $(\frac{\text{address}}{\text{block size}})\ \%\ \text{set}$

Tag = $\frac{\text{address}}{(\text{set} * \text{ block size})}$

e.g **37 => for a two way, two word per block, and a total of 8 blocks**

Set index = Set index = $(\frac{37}{2})\ \%\ 4$ = $\frac{5}{2}$ = 2

Tag = $\frac{37}{(4 * 2)}$ = 4

In binary

37 => 100**10**1

The set index is **10b.**

Tag = 100b

addr	Tag	Index	Hit/miss	Cache state after access
1	000	00	miss	{(0,1)} {} {} {}
18	010	01	miss	{(0,1)} {(18,19)} {} {}
2	000	01	miss	{(0,1)} {(18,19),(2,3)} {} {}
3	000	01	hit	{(0,1)} {(18,19),(2,3)} {} {}
4	000	10	miss	{(0,1)} {(18,19),(2,3)} {(4,5)} {}
20	010	10	miss	{(0,1)} {(18,19),(2,3)} {(4,5),(20,21)} {}
5	000	10	hit	{(0,1)} {(18,19),(2,3)} {(4,5),(20,21)} {}
21	010	10	hit	{(0,1)} {(18,19),(2,3)} {(4,5),(20,21)} {}
33	100	00	miss	{(0,1),(32,33)} {(18,19),(2,3)} {(4,5),(20,21)} {}
34	100	01	miss	{(0,1),(32,33)} {(34,35),(2,3)} {(4,5),(20,21)} {}
1	000	00	hit	{(0,1),(32,33)} {(34,35),(2,3)} {(4,5),(20,21)} {}
4	000	10	hit	{(0,1),(32,33)} {(34,35),(2,3)} {(4,5),(20,21)} {}

Hit Rate Calculation:

- Total accesses: 12
- Hits: 5
- Misses: 7

Hit Rate = 5 / 12

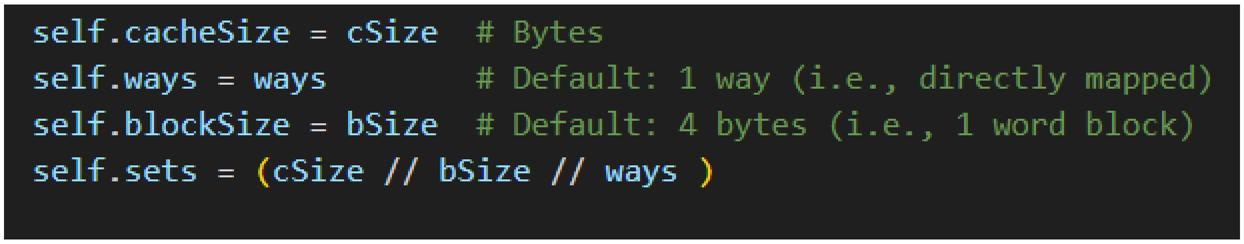
Description of functions

To calculate the set, we use the formula:

Set = $\dfrac{\text{cache size}}{\text{block size} * \text{no of ways}}$

Total cache size divided by block size, gives the total number of blocks in the cache.

Then dividing with the number of ways gives the total set in the cache.



```
self.cacheSize = cSize  # Bytes
self.ways = ways        # Default: 1 way (i.e., directly mapped)
self.blockSize = bSize  # Default: 4 bytes (i.e., 1 word block)
self.sets = (cSize // bSize // ways )
```

1. find_set (address):

This function calculates the **set index** based on the string address using the following formula we derived in the solution above:

Set index = $(\dfrac{\text{address}}{\text{block size}})\ \%\ \text{set}$

E.g assume cache address = 21 in byte address, for a 8 block set, and block size is 2 bytes. For the 8 block set, we use the next 3 bits after the LSB to represent it.

Set index = 21 **=>** $\dfrac{21}{2}\ \%\ 8$ **= 2**

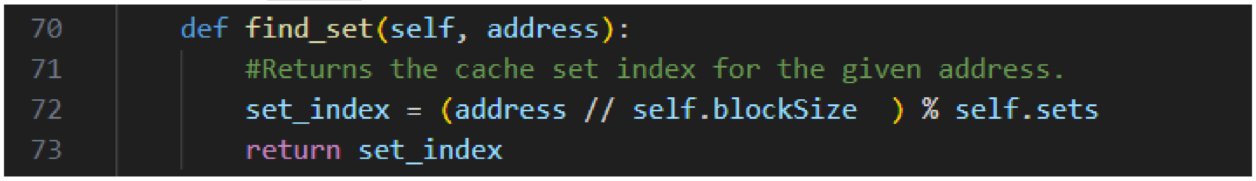
1**010**1 => **2**

Tag = = 21 **=>** $\dfrac{\mathbf{21}}{(\mathbf{8} * \mathbf{2})}$ **= 1**

21 = 1**010**1 here **010** is the set index, tag = 1

Note, the address and block size are in the same unit, whether bytes or word. For the code file they are in bytes.

In the formula, we first divide the address by the block size. This returns the block number where the address falls in, then multiplying it with the modulus set (%set) makes sure it doesn't exceed the set limit.



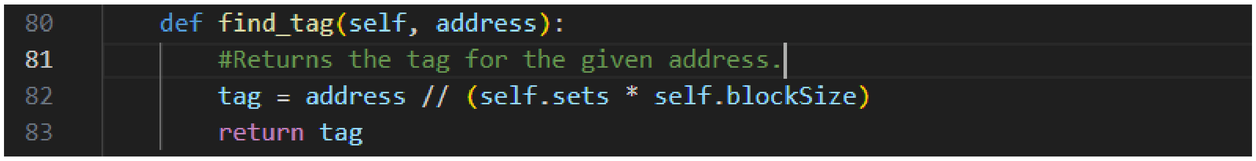
```python
    def find_set(self, address):
        #Returns the cache set index for the given address.
        set_index = (address // self.blockSize  ) % self.sets
        return set_index
```

2. find_tag (address):

The **tag** is the part of the address that helps us identify if the data stored in the cache block is the correct one for the given cache address. It's calculated by dividing the cache address by the number of sets multiplied by the block size. This will return all the bits after the set bits. The formula was derived above.

$$\text{Tag} = \frac{\text{address}}{(\text{set} * \text{ block size})}$$

In the formula we convert the address to byte address by multiplying with four, since the block size is in bytes.



```python
    def find_tag(self, address):
        #Returns the tag for the given address.
        tag = address // (self.sets * self.blockSize)
        return tag
```

3. load(address):

When there's a cache miss, this function loads the data from memory into the cache. It updates the cache block at the appropriate set with the new tag and data. We first extract the set and tag of the address. Then we store the tag in the Meta cache at the set index,

For a set associative mapping it is a little different from direct-mapped cache.

For set-assocative mapping using LRU. We need to store a pointer to the least recently used block for each set so that on every load they are replaced if the set is not yet filled. In the code it is stored in the `self.cache[set_index, 0, 0]` **`for each of the sets.`** Then we store the incoming address within the metacache at the set index, on the LRU pointer index. Then increment the pointer. We use %self.ways after the increment , this is to ensure that the pointer over wraps and starts from zero when it gets to the last block.

```python
def load(self, address):
        #extract the set_index and tag from the address
        set_index = self.find_set(address)
        tag = self.find_tag(address)

        #for LRU, store a pointer to the least recently
        #used block of the set, in an arbitrarily location
        #here we are storing the pointer for each set in the cache
        #at address self.cache[set_index, 0, 0]
        lru = self.cache[set_index, 0, 0]

        #when the LRU has not been incremented, numpy
       numpy initialize them to -1 instead of 0,
        #so, rewrite to zero if it is -1.
        if(lru == -1):
            lru =0
        #load the new tag into the set at the LRU pointer
        #and increment the pointer to the next least recently
        # used tag in the set
        self.metaCache[set_index, lru] = tag
        self.cache[set_index, 0, 0] = (lru + 1 ) % self.ways
```

4. find (address):

This function checks if data for a given address exists in the cache by examining both the tag and set index. First, it retrieves the set index and tag from the address. If an entry with a matching tag is found in the cache set, it counts as a "hit," and we increment the hit count. If no match is found, it’s a "miss," and we call load() to fetch data from memory.

For a Set-Associative Mapping Using LRU: After extracting the set index and tag, the function iterates through the set's ways to look for a matching tag. If no match is found, it’s a miss, and the address is loaded. If a match (hit) is found, the hit count is incremented. Next, the LRU pointer, which identifies the next entry to replace, is updated. If the LRU pointer is set to -1, it’s reset to zero. The order of tags in the set is then shuffled to mark the hit tag as the most recently used (MRU), ensuring the cache reflects recent usage patterns.

```
    def find(self, address):
        #extract the set_index and tag from the address.
        set_index = self.find_set(address)
        tag = self.find_tag(address)

        #loop through the ways in the set, search for a match
        for i in range(self.ways):
            #if there is a match, then it is a hit.
            if(tag == self.metaCache[set_index,i]):
                #increment hit count
                self.hit = self.hit + 1

                #we need to shuffle the ways, whenever we get a hit
                #by moving the tag we hit, as the Most Recently Used(MRU).

                #extract the set_index pointer and make sure it is not pointing
                #to -1
                lru = self.cache[set_index, 0, 0]
                if(lru == -1):
                    lru =0
```

example shuffling

1 2 3 **4** 5 6 7 **8** 9

Assume the LRU pointer is pointing to 8, and we found tag 4. Note: since the pointer is moving in the forward direction, the LRU is 8, which is the data to replace next and 7 is the Last recently used.

So we want to move tag 4, as the last recently used tag by replacing it with seven so it becomes.

1 2 3 5 6 7 4 8 9 => note the order of replacement is 8, 9, 1, 2, 3, and so on.

To implement this in the code, we loop between 4 and 8. That is the tag index we found and the LRU pointer. Then remove 4, and push them backwards, after which we replace 4 behind 8.

1 2 3 4 5 6 7 **8** 9

A second situation occurs where we the tag we found is 8, and 8 is also at the LRU. We simply increment the pointer to 9.

1 2 3 4 5 6 7 8 **9**

In the code below, if there is a hit and the LRU pointer is also pointing on the tag, that is, if the next tag in the LRU pointer to replace is the current tag we found (i == LRU), simply increment the LRU pointer.

Else shuffle . we shuffle only address between the found tag and the LRU pointer e.g

If we have 1 2 3 4 5 6 7 8 9

And we hit tag 5 , while the LRU is pointing to 9. We simply shuffle only 5 6 7 8 9 by moving 5 forward to replace 8. And other address downwards.

```
   #if the tag is the least recently used tag, then LRU is pointing to i,
                # simply increment LRU to point to the next tag.

                if (i  == lru):
                    lru = (lru + 1)% self.ways
                    self.cache[set_index, 0, 0] = lru
                else:
  #else shuffle the tags, remove the tag we found and bring other tags backwards
                    while (i  != lru):

                        self.metaCache[set_index, i % self.ways] = 
self.metaCache[set_index, (i + 1)% self.ways]
                        i = (i + 1)% self.ways
#once we are done with shuffling, add our tag before the pointer.
                   self.metaCache[set_index, (lru-1)% self.ways] = tag

                return True  # Cache hit
        return False  # Cache miss
```

so within the while loop, we keep moving the address downwards to create space for our hit using :

```
self.metaCache[set_index, i % self.ways] =  self.metaCache[set_index, (i + 1)%
self.ways]
```

after the shuffle, we move the tag forward just before the LRU pointer.

```
self.metaCache[set_index, (lru-1)% self.ways] = tag
```

then we return true.

Verification code output for set associative memory

```
$ python3 sim_sa.py pingpong.trace 16 2 4
pingpong.trace
Processing your program trace, progress so far = 0 %
set and tag of 0x0 is 0 0
address 0x0 CACHE MISS. Loading from memory.
set and tag of 0x10 is 0 8
address 0x10 CACHE MISS. Loading from memory.
set and tag of 0x0 is 0 0
address 0x0 CACHE HIT. Good Job.
set and tag of 0x10 is 0 8
address 0x10 CACHE HIT. Good Job.
```

```
set and tag of 0x0 is 0 0
address 0x0 CACHE HIT. Good Job.
set and tag of 0x10 is 0 8
address 0x10 CACHE HIT. Good Job.
set and tag of 0x0 is 0 0
address 0x0 CACHE HIT. Good Job.
set and tag of 0x10 is 0 8
address 0x10 CACHE HIT. Good Job.
total cache misses 2
miss_rate 0.25
hit_rate 0.75
Finished processing your program trace, progress = 100.0 %
```

```
$ python3 sim_sa.py test_q.trace 64 2 4
test_q.trace
Processing your program trace, progress so far = 0 %
set and tag of 0x0 is 0 0
address 0x0 CACHE MISS. Loading from memory.
set and tag of 0x4 is 1 0
address 0x4 CACHE MISS. Loading from memory.
set and tag of 0x24 is 1 1
address 0x24 CACHE MISS. Loading from memory.
set and tag of 0x10 is 4 0
address 0x10 CACHE MISS. Loading from memory.
set and tag of 0x4 is 1 0
address 0x4 CACHE HIT. Good Job.
set and tag of 0x44 is 1 2
address 0x44 CACHE MISS. Loading from memory.
set and tag of 0x24 is 1 1
address 0x24 CACHE MISS. Loading from memory.
set and tag of 0x1c is 7 0
address 0x1c CACHE MISS. Loading from memory.
set and tag of 0x0 is 0 0
address 0x0 CACHE HIT. Good Job.
set and tag of 0x24 is 1 1
address 0x24 CACHE HIT. Good Job.
set and tag of 0x30 is 4 1
address 0x30 CACHE MISS. Loading from memory.
set and tag of 0x10 is 4 0
address 0x10 CACHE HIT. Good Job.
total cache misses 8
miss_rate 0.6666666666666666
hit_rate 0.33333333333333337
Finished processing your program trace, progress = 100.0 %
```

```
$ python3 sim_sa.py sa.trace  64 2 8
sa.trace
Processing your program trace, progress so far = 0 %
set and tag of 0x4 is 0 0
address 0x4 CACHE MISS. Loading from memory.
set and tag of 0x48 is 1 2
address 0x48 CACHE MISS. Loading from memory.
set and tag of 0x8 is 1 0
address 0x8 CACHE MISS. Loading from memory.
set and tag of 0xc is 1 0
address 0xc CACHE HIT. Good Job.
set and tag of 0x10 is 2 0
address 0x10 CACHE MISS. Loading from memory.
set and tag of 0x50 is 2 2
address 0x50 CACHE MISS. Loading from memory.
set and tag of 0x14 is 2 0
address 0x14 CACHE HIT. Good Job.
set and tag of 0x54 is 2 2
address 0x54 CACHE HIT. Good Job.
set and tag of 0x84 is 0 4
address 0x84 CACHE MISS. Loading from memory.
set and tag of 0x88 is 1 4
address 0x88 CACHE MISS. Loading from memory.
set and tag of 0x4 is 0 0
address 0x4 CACHE HIT. Good Job.
set and tag of 0x10 is 2 0
address 0x10 CACHE HIT. Good Job.
total cache misses 7
miss_rate 0.5833333333333334
hit_rate 0.4166666666666663
Finished processing your program trace, progress = 100.0 %
```

```
$ python3 sim_dm.py dm_a.trace 64 1 4
main.py trace cacheSize(Bytes) #ofWays blockSize(Bytes)
dm_a.trace
Processing your program trace, progress so far = 0 %
0 address

set and tag of 0x0 is 0 0
address 0x0 CACHE MISS. Loading from memory.
4 address

set and tag of 0x4 is 1 0
address 0x4 CACHE MISS. Loading from memory.
8 address

set and tag of 0x8 is 2 0
address 0x8 CACHE MISS. Loading from memory.
12 address

set and tag of 0xc is 3 0
address 0xc CACHE MISS. Loading from memory.
16 address

set and tag of 0x10 is 4 0
address 0x10 CACHE MISS. Loading from memory.
20 address

set and tag of 0x14 is 5 0
address 0x14 CACHE MISS. Loading from memory.
```

```
24 address

set and tag of 0x18 is 6 0
address 0x18 CACHE MISS. Loading from memory.
28 address

set and tag of 0x1c is 7 0
address 0x1c CACHE MISS. Loading from memory.
32 address

set and tag of 0x20 is 8 0
address 0x20 CACHE MISS. Loading from memory.
36 address

set and tag of 0x24 is 9 0
address 0x24 CACHE MISS. Loading from memory.
40 address

set and tag of 0x28 is 10 0
address 0x28 CACHE MISS. Loading from memory.
44 address

set and tag of 0x2c is 11 0
address 0x2c CACHE MISS. Loading from memory.
48 address

set and tag of 0x30 is 12 0
address 0x30 CACHE MISS. Loading from memory.
52 address

set and tag of 0x34 is 13 0
address 0x34 CACHE MISS. Loading from memory.
56 address

set and tag of 0x38 is 14 0
address 0x38 CACHE MISS. Loading from memory.
60 address

set and tag of 0x3c is 15 0
address 0x3c CACHE MISS. Loading from memory.
4 address

set and tag of 0x4 is 1 0
address 0x4 CACHE HIT. Good Job.
72 address

set and tag of 0x48 is 2 1
address 0x48 CACHE MISS. Loading from memory.
8 address

set and tag of 0x8 is 2 0
address 0x8 CACHE MISS. Loading from memory.
12 address

set and tag of 0xc is 3 0
address 0xc CACHE HIT. Good Job.
16 address

set and tag of 0x10 is 4 0
address 0x10 CACHE HIT. Good Job.
80 address

set and tag of 0x50 is 4 1
address 0x50 CACHE MISS. Loading from memory.
20 address

set and tag of 0x14 is 5 0
address 0x14 CACHE HIT. Good Job.
84 address

set and tag of 0x54 is 5 1
address 0x54 CACHE MISS. Loading from memory.
```

```
132 address

set and tag of 0x84 is 1 2
address 0x84 CACHE MISS. Loading from memory.
136 address

set and tag of 0x88 is 2 2
address 0x88 CACHE MISS. Loading from memory.
4 address

set and tag of 0x4 is 1 0
address 0x4 CACHE MISS. Loading from memory.
16 address

set and tag of 0x10 is 4 0
address 0x10 CACHE MISS. Loading from memory.
total cache misses 24
miss_rate 0.8571428571428571
hit_rate 0.1428571428571429
Finished processing your program trace, progress = 100.0 %
```

```
$ python3 sim_dm.py dm_b.trace 64 1 8
main.py trace cacheSize(Bytes) #ofWays blockSize(Bytes)
dm_b.trace
Processing your program trace, progress so far = 0 %
4 address

set and tag of 0x4 is 0 0
address 0x4 CACHE MISS. Loading from memory.
72 address

set and tag of 0x48 is 1 1
address 0x48 CACHE MISS. Loading from memory.
8 address

set and tag of 0x8 is 1 0
address 0x8 CACHE MISS. Loading from memory.
12 address

set and tag of 0xc is 1 0
address 0xc CACHE HIT. Good Job.
16 address
```

```
set and tag of 0x10 is 2 0
address 0x10 CACHE MISS. Loading from memory.
80 address

set and tag of 0x50 is 2 1
address 0x50 CACHE MISS. Loading from memory.
20 address

set and tag of 0x14 is 2 0
address 0x14 CACHE MISS. Loading from memory.
84 address

set and tag of 0x54 is 2 1
address 0x54 CACHE MISS. Loading from memory.
132 address

set and tag of 0x84 is 0 2
address 0x84 CACHE MISS. Loading from memory.
136 address

set and tag of 0x88 is 1 2
address 0x88 CACHE MISS. Loading from memory.
4 address

set and tag of 0x4 is 0 0
address 0x4 CACHE MISS. Loading from memory.
16 address

set and tag of 0x10 is 2 0
address 0x10 CACHE MISS. Loading from memory.
total cache misses 11
miss_rate 0.9166666666666666
hit_rate 0.08333333333333337
Finished processing your program trace, progress = 100.0 %
```