```
#Lakshmi Katravulapalli
#Cache Simulation Project Code

import sys

# Variables to maintain the simulation statistics
Hit = 0
Miss = 0
reads = 0
writes = 0


def update_lru(address):
  # Logic for updating LRU policy
  # getting the set index
  set_idx = (address // BLOCK_SIZE) % NUM_SETS
  #  getting the tag
  tag = address // BLOCK_SIZE
  # Checking if the tag already in the set of the cache, if it is there we have to
update the lru_position by making it the most recently used
  if tag in lru_position[set_idx][1]:
    lru_position[set_idx][0].remove(tag)
    lru_position[set_idx][0].append(tag)
    lru_position[set_idx][1][tag] = len(lru_position[set_idx][0]) - 1
  # if not there making the most recently used
  else:
    lru_position[set_idx][0].append(tag)
    lru_position[set_idx][1][tag] = len(lru_position[set_idx][0]) - 1


def simulate_access(op, address):
  # Getting tag and set_index
  set_idx = (address // BLOCK_SIZE) % NUM_SETS
  tag = address // BLOCK_SIZE
  #  variable 'found' is used to check if the tag is found in the set.
  found = False
  global reads, writes
  # variable 'reads' contain no. of reads from the memory, 'writes' contain no. of
writes to the memory,
  for i in range(len(tag_array[set_idx])):
    # Go through each block in the set and check if the tag is found.
    if tag == tag_array[set_idx][i]:
      # If found increase Hit count, and make found=True
      global Hit
      found = True
      Hit += 1
      # If LRU replacement policy update the LRU position.
      if is_lru:  # LRU policy is chosen
        update_lru(address)

      if op == 'W' and WB == True:
        # If the policy is write-back then make it dirty.
        dirty[tag] = True
      elif op == 'W':
        # If the policy is write through update write's count
        writes+= 1

  if not found:
    # If not found We increase the miss count and allocate the new block
```

```
    global Miss
    Miss += 1
    if len(tag_array[set_idx]) == ASSOC:
      # We check if there is no space in the set we have to evict the block using
the replacement policy
      if is_lru:
        # if LRU then we evict the least recently used tag.
        evicted = lru_position[set_idx][0].pop(0)
        # evict the block
        tag_array[set_idx].remove(evicted)
        del lru_position[set_idx][1][evicted]
        # if the evicted block is dirty we have to write to the memory
        if evicted in dirty:
          del dirty[evicted]
          # writes to memory increase
          writes+= 1
      elif is_fifo:
        # if FIFO then we evict the block that entered first.
        evicted = tag_array[set_idx].pop(0)
        # if the evicted block is dirty we have to write to the memory
        if evicted in dirty:
          del dirty[evicted]
          # writes to memory increase
          writes+= 1
      elif is_lifo:
        # if LIFO then we evict the block that entered last.
        evicted = tag_array[set_idx].pop()
        # if the evicted block is dirty we have to write to the memory
        if evicted in dirty:
          del dirty[evicted]
          # writes to memory increase
          writes+= 1
    # We allocate the block for new tag
    tag_array[set_idx].append(tag)
    # If LRU we update the LRU
    if is_lru:  # LRU policy is chosen
      update_lru(address)
    if op == 'W' and WB == True:
      # If the policy is write-back then make it dirty.
      dirty[tag] = True
    elif op == 'W':
      # If the policy is write through update write's count
      writes+= 1
    # If it is miss then the read from memory occurs for both write and read
operations
    reads+=1


if __name__ == "__main__":
  # ./SIM <CACHE_SIZE> <ASSOC> <REPLACEMENT> <WB> <TRACE_FILE>
  arguments=sys.argv[1:]
  # Getting the Cache size
  CACHE_SIZE=int(arguments[0])
  # GEtting the Associativity of the cache
  ASSOC =int(arguments[1])
  if ASSOC==0:
    print("Associativity shouldn't be 0")

  BLOCK_SIZE = 64
```

```
  is_lru=False
  is_fifo=False
  is_lifo=False
  NUM_SETS = CACHE_SIZE // (BLOCK_SIZE * ASSOC)  # Configure number of sets
  # Getting the replacement policy 0 for LRU, 1 for FIFO, 2 for LIFO.
  if arguments[2]=='0':
    is_lru=True
  elif arguments[2]=='1':
    is_fifo=True
  elif arguments[2]=='2':
    is_lifo=True
  # Getting the write policy option
  #  1 for writeback and 0 for write through
  WB=bool(int(arguments[3]))

  # tag_array represents the cache
  tag_array = [[] for _ in range(NUM_SETS)]

  # lru_position holds the tag values in least recently to most recent order.
  lru_position = [[[], {}] for _ in range(NUM_SETS)]
  # dirty dictionary holds if the block is dirty or not.
  dirty = {}
  with open(arguments[4], 'r') as file:
    for line in file:
      op, address = line.split()
      address = int(address, 16)
      simulate_access(op, address)

# Print out the statistics
print(f"Hits: {Hit}")
print(f"Misses: {Miss}")
print(f"Reads: {reads}")
print(f"Writes: {writes}")
print(f"Miss ratio: {Miss/(Miss+Hit)}")
```