Group 2

Mohanad Abouserie - 900213567

Amr Abdelbaky - 900204834

Mahmoud Nour - 900202978

A brief description of the implementation:

To begin with, we take as an input the total cache size, the cache line size, and the number of cycles needed to access the cache. We also take as an input a text file, which includes a sequence of memory addresses to be accessed. The memory addresses are in decimal format, and they represent the location of the byte to be accessed.

The implementation is divided as follows:
A struct named CacheLine, which has three variables, namely the valid bit, the tag value, and the vector of the bytes (offsets) for this line.
A class named CacheSimulator which basically abstracts the whole algorithm
A run_code function which is responsible for taking the input, validating it, and running the simulation.

In what follows, we'll look inside the CacheSimulator class to uncover the core implementation of the algorithm.

The class has the following as its attributes:

```
vector<vector<CacheBlock>> cache;
 unsigned int cacheSize;
 unsigned int lineSize;
 unsigned int accessCycles;
 unsigned int accesses;
 unsigned int hits;
 unsigned int misses;
 unsigned int numLines;
 unsigned int indexBits;
 unsigned int offsetBits;
 unsigned int tagBits;
```

We implemented the cache as a vector of type CacheLine. CacheLine consists of valid bit, tag, and vector of the bytes (offsets) of the line.

A constructor is called with the initialization of the class. The constructor uses the three inputs of the program (cache size, line size, and access cycles), which are eventually passed as parameters to the constructor, to initialize the attributes of the class, as well as resize the cache accordingly.
Three helper functions namely getIndex, getTag, and getOffset use the address to extract the Index, tag, and offset bits.

A simulate function loops over each address in the file. In each loop, it calls the accessMemory function, then recalculates the hit ratio, miss ratio, and AMAT, and finally outputs the following as required:

- Cache state after every access
- Total number of accesses
- Hit ratio
- Miss ratio
- Average memory access time (AMAT)

The accessMemory function in the provided code is responsible for simulating memory access in the cache. It performs the following steps:
First, it increments the accesses counter to keep track of the total number of memory accesses. Next, it calculates the index, tag, and offset of the given memory address using the getIndex(), getTag(), and getOffset() helper functions, respectively. After that, it searches for the offset within the vector cache[index].offsets to check if the accessed offset is already present in the cache line. If the cache line is valid (cache[index].valid is true), the tag matches (cache[index].tag is equal to tag), and the offset is found (it != cache[index].offsets.end()), it indicates a cache hit. In this case, the hits counter is incremented. However, if any of these conditions fail, it means there is a cache miss. The misses counter is incremented, and the function proceeds to update the cache line. If the cache line is already valid and the tag does not match (cache[index].valid && cache[index].tag != tag), it means that the cache line needs to be overwritten. In this case, the offsets vector of the cache line is cleared to make room for the new offset. Afterwards, the cache line is marked as valid (cache[index].valid = true), the tag is updated (cache[index].tag = tag), and the accessed offset is added to the offsets vector of the cache line.