Coding Report

-Anirudh Saikrishnan(CS22Btech11004)

The program implements three different cache mapping strategies: direct-mapped, fully associative, and set-associative. The program reads cache configuration parameters and an access sequence from input files, simulates cache accesses based on the specified mapping strategy, and prints the results.

Key components of the coding approach

1. Data Structures used:
   The program defines three structures,
   - CacheConfig – Cache configuration parameters
   - Access – Mode + Address
   - CacheBlock – Flags for validity+dirtiness and the tag

2. File Handling
   The program uses utility functions `readCacheConfig` and `readAccessSequence` to take the input from the previously established files.

3. Direct Mapped Cache:
   - The `directMapped` function simulates a direct-mapped cache, a straightforward mapping where each memory block corresponds to a specific cache line.
   - The function begins by breaking down the memory address into relevant components: index, tag, and offset. It does so by calculating the required number of bits for offset and index from the cache configuration and performing the necessary right shift operations on the address.
   - For each access in the access sequence, the function checks if the corresponding cache line is occupied by the desired block by comparing the tags. If the cache line is occupied by the same block (hit), the function prints a hit message, including the address, index, and tag. If the cache line is not occupied by the desired block (miss), the function prints a miss message, updates the cache with the new block's tag, and continues.
   - The function does not take replacement policy into consideration since there is only one strategy for a direct mapped cache
   - The index is printed in the output

4. Fully Associative Mapped Cache:
   - The `fullyAssociative` function simulates a fully associative cache, where any memory block can map to any cache line. This function supports multiple replacement policies, namely FIFO, LRU, and RANDOM.
   - Like Direct mapping, the function begins by breaking down the address into relevant components, but this time, only into offset bits and tag since there is no fixed index for a block. It calculates the tag by using the necessary right shift operations again
   - **FIFO:** The function initializes a queue to keep track of the order in which the indices have been added/tampered with.

- **LRU:** The function initializes an array to keep track of the access order for each cache line, facilitating the identification of the least recently used block. It essentially counts the number of iterations since an index has been last accessed. Thus, the index with the largest value is the one that needs to be replaced.
- **RANDOM:** The function generates an index randomly and replaces the element present there in case the cache is full
- No index is printed since it has no real meaning in a fully associative cache

5. Set Associative Mapped Cache:

   The function 'setAssociative' uses very similar strategies to 'fullyAssociative' for all 3 replacement policies since in essence, once the set has been found, it's just a fully associative cache within that set.
   The Set number and the way are both printed in the output

6. Test Cases

   Very limited testing has been done on the code, not extending beyond a few random addresses strung together just to ensure input/output was functioning correctly.
   Most of the logic in the code stems from pure theoretical knowledge and remains untested.

7. Not much is done in the case of writes. Write throughs are not treated any differently since there is no connection to a main memory.
   In case of write backs, the dirty bit is set to one, but that's about it.