YASIR USMANI
AI22BTECH11031

COMPUTER ARCHITECTURE
CACHE MISS SIMULATOR - LAB 6

Introduction

The design and execution of a C++ cache simulation programme are described in this article. Based on the selected cache settings from the "cache.config" file and the access sequence from the "cache.access" file, the programme attempts to replicate cache behavior. Numerous cache settings are supported by the simulation, including write policy, associativity, block size, cache size, and replacement policy.

SUPPORTED PARTS

PART - 1: YES

PART - 2: YES

PART - 3: YES

PART - 4: YES

IMPLEMENTATION

The simulation is written in C++ and has many functions to translate binary to decimal, binary to hexadecimal, and binary to hexadecimal addresses. The main() functions, which receives the cache configuration, performs the access sequence, and outputs data about cache hits and misses, handles the core simulation logic.

CACHE CONFIGURATION

The "cache.config" file, which contains information about the cache's size, block size, associativity, replacement policy, and write policy, is read to determine the cache

configuration. Variables like cacheSize, blockSize, associativity, policy, and wbPolicy hold the configurations. These settings are used by the programme to set the simulation behavior and initialize the cache.

ACCESS CONFIGURATION

The "cache.access" file is where the access sequence, which includes the read (R) and write (W) operations and their related memory locations, is read. Every access is processed by the programme, which then decides if it is a cache hit or miss. It also takes into account the write policy (Write-Back or Write-Through) that is set in the cache configuration while performing write operations.

CACHE SIMULATION

Three replacement policies are supported by the simulation: RANDOM, LRU, and FIFO. Tags are stored in the cache using a two-dimensional array (idx), and replacement policy management is handled by extra arrays (lruClock for LRU). The software determines the index and tag based on the memory address and determines if the access is successful or unsuccessful.

RESULTS

Each access is reported by the programme, together with details on the memory address, set/index, hit/miss ratio, and cached tag. The output is displayed using the format that the assignment instructions provide.

CONCLUSION

Using the parameters and access sequence that are supplied, the cache simulation programme correctly simulates cache behavior. It manages various write rules and replacement strategies while offering information on cache hits and misses. It is simple to extend and modify the modular architecture to accommodate new features or combinations.

AREAS FOR IMPROVEMENT

There is room for improvement even if the existing solution satisfies the assignment criteria. To make the code more efficient and readable, it might be optimized. To increase the program's resilience, more error handling and input validation might be included.

TESTING

UNIT TESTING : To guarantee accuracy, every function—including conversion functions and cache simulation logic—was separately put through a unit test.

BOUNDARY TESTING : To verify robustness, the code was evaluated using the highest and lowest values that could be used for associativity, block size, and cache size.

SCENARIO TESTING : Different block sizes, associativities, replacement policies, and cache and block sizes were among the scenarios used to evaluate the algorithm.

RANDOM TESTING : To verify that the simulation is accurate and flexible, random test scenarios were created.