



TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

---

## Cache Memory Simulator

---

*Student :*

Stefania-Cristina MOZACU

*Teacher :*

Anca HANGAN

*Group :*

30434

December 16, 2025

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	Context and Motivation of the Project . . . . .	6
1.2	Purpose and Objectives of the Simulator . . . . .	7
1.3	Overview of the Cache Memory Concept . . . . .	7
<b>2</b>	<b>Project Proposal</b>	<b>9</b>
2.1	Objectives and Goals . . . . .	9
2.2	Functional Requirements . . . . .	9
2.3	Nonfunctional Requirements . . . . .	9
2.4	Project Timeline . . . . .	10
<b>3</b>	<b>Bibliographic Study</b>	<b>11</b>
3.1	Fundamental Concepts of Cache Memory . . . . .	11
3.2	Cache Types and Hierarchy (L1, L2, L3) . . . . .	11
3.3	Replacement and Write Policies . . . . .	12
3.4	Locality of Reference . . . . .	12
3.5	Cache Coherence and Consistency Protocols . . . . .	12
3.6	Cache Mapping Techniques (Direct-Mapped, Fully Associative, Set-Associative) . . . . .	13
<b>4</b>	<b>Analysis</b>	<b>15</b>
4.1	System Overview . . . . .	15
4.2	Functional Requirements and Mechanisms . . . . .	16
4.3	Nonfunctional Requirements and Mechanisms . . . . .	18
4.4	Use Case Diagram and Descriptions . . . . .	19
4.5	System Constraints and Limitations . . . . .	21
<b>5</b>	<b>Design</b>	<b>23</b>
5.1	Overview . . . . .	23
5.2	Package Diagram . . . . .	24
5.3	Class Diagram . . . . .	25
5.4	Component and Block Diagram . . . . .	27

5.5	Flow Chart . . . . .	28
5.6	Data Structure Design . . . . .	29
5.7	User Interface . . . . .	29
5.7.1	Configuration Panel . . . . .	30
5.7.2	Address Decode Panel . . . . .	31
5.7.3	Cache Visualisation . . . . .	31
5.7.4	Eviction Log and Statistics . . . . .	32
5.7.5	Execution Controls . . . . .	32
<b>6</b>	<b>Implementation</b>	<b>33</b>
6.1	IT Technology . . . . .	33
6.2	Algorithms and Simulation Scenarios . . . . .	34
6.2.1	Cache Mechanism . . . . .	34
6.2.2	Replacement Mechanism . . . . .	36
6.2.3	FIFO (First-In, First-Out) Policy . . . . .	37
6.2.4	LRU (Least Recently Used) Policy . . . . .	38
6.2.5	Random Replacement Policy . . . . .	38
6.2.6	Scenario 1: Matrix Traversal (Spatial Locality) . . . . .	39
6.2.7	Scenario 2: Random Access (No Locality) . . . . .	40
6.3	Data Input and Output . . . . .	41
6.4	Class Implementation Details . . . . .	42
6.5	Cache Mapping Implementation . . . . .	42
6.5.1	General k-Way Set-Associative Model . . . . .	42
6.5.2	Wrapper-Based Mapping Configuration . . . . .	43
6.5.3	Direct-Mapped Configuration ( $k = 1$ ) . . . . .	43
6.5.4	General Set-Associative Configuration ( $k > 1$ ) . . . . .	44
6.5.5	Fully Associative Configuration ( $k = \text{num\_blocks}$ ) . . . . .	44
6.6	Replacement Policy Implementation . . . . .	44
6.6.1	Policy Abstraction . . . . .	45
6.6.2	FIFO Policy Implementation . . . . .	45
6.6.3	LRU Policy Implementation . . . . .	45
6.6.4	Random Policy Implementation . . . . .	46

6.6.5	Integration with the Cache Core . . . . .	46
6.7	Simulation Engine and Execution Flow . . . . .	47
6.7.1	Cache Construction and Mapping Selection . . . . .	47
6.7.2	Address Sequence Preparation . . . . .	48
6.7.3	Execution Loop . . . . .	48
6.7.4	Step-by-Step and Full Execution . . . . .	49
6.7.5	Sequence Generation . . . . .	49
6.7.6	Execution Flow . . . . .	50
6.7.7	Core Simulation Logic . . . . .	51
6.8	Graphical User Interface Implementation . . . . .	53
6.8.1	UI Architecture and Responsibilities . . . . .	54
6.8.2	Event-Driven Execution Model . . . . .	54
6.8.3	Interface Layout . . . . .	54
6.8.4	Configuration Widgets and State Binding . . . . .	55
6.8.5	Cache Construction from UI Parameters . . . . .	55
6.8.6	Address Decode Visualization . . . . .	56
6.8.7	Cache Grid Rendering . . . . .	56
6.8.8	Simulation Control and Animation . . . . .	57
6.8.9	Statistics Display and Logging . . . . .	57
6.8.10	Integration with the Core Simulator . . . . .	58
6.9	Application Workflow and User Interface Walkthrough . . . . .	58
6.9.1	Initial Application State and Cache Configuration . . . . .	58
6.9.2	Input Validation and Error Handling . . . . .	60
6.9.3	Cache Size and Line Size Consistency . . . . .	60
6.9.4	Associativity and Block Divisibility . . . . .	61
6.9.5	Cache Hit Visualization . . . . .	62
6.9.6	Cache Miss Visualization . . . . .	62
6.9.7	Write Miss with Dirty Block (Write-Back Policy) . . . . .	63
6.9.8	Write Miss without Dirty Block . . . . .	64
6.9.9	Address Decoding and Tag-Index-Offset Visualization . . . . .	65
6.9.10	Result Export and Persistence . . . . .	66
6.9.11	Export of Simulation Results in JSON Format . . . . .	67

6.9.12	Export of Performance Charts as PDF . . . . .	68
6.10	Data Export and Visualization . . . . .	68
6.10.1	Statistics Collection . . . . .	68
6.10.2	Real-Time Visualization . . . . .	69
6.10.3	Aggregated Performance Display . . . . .	69
6.10.4	Data Export . . . . .	70
6.10.5	Design Rationale . . . . .	70
6.11	Error Handling and Input Validation . . . . .	70
6.11.1	User Interface Input Validation . . . . .	71
6.11.2	Cache Configuration Validation . . . . .	71
6.11.3	Address Validation and Normalization . . . . .	71
6.11.4	Runtime Safety in Simulation Execution . . . . .	72
6.11.5	RAM Boundary Checks . . . . .	72
6.11.6	Error Reporting and User Feedback . . . . .	72
6.11.7	Design Rationale . . . . .	72
6.12	Performance Considerations and Optimizations . . . . .	73
6.12.1	Step-Based Execution Model . . . . .	73
6.12.2	Efficient Data Structures . . . . .	73
6.12.3	Limited Cache and Memory Sizes . . . . .	73
6.12.4	Non-Blocking User Interface Updates . . . . .	74
6.12.5	Avoidance of Redundant Computation . . . . .	74
6.12.6	Trade-Offs and Limitations . . . . .	74
6.12.7	Potential Optimizations . . . . .	74
6.12.8	Design Rationale . . . . .	75
<b>7</b>	<b>Testing and Validation</b>	<b>76</b>
7.1	Testing Strategy . . . . .	76
7.2	Replacement Policy Validation . . . . .	76
7.3	Write Policy Testing . . . . .	77
7.4	Write-Miss Policy Validation . . . . .	77
7.5	Eviction and Write-Back to RAM . . . . .	78
7.6	Line Size and Address Alignment Testing . . . . .	78

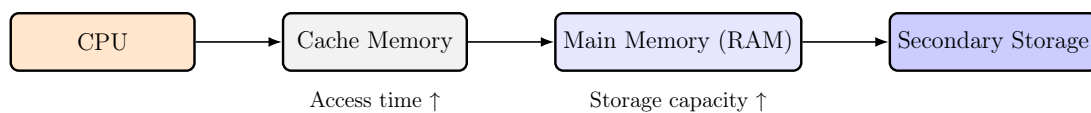
7.7	Simulation Engine Integration Tests . . . . .	78
7.8	Cache Reset Validation . . . . .	79
7.9	RAM Boundary and Error Handling Tests . . . . .	79
7.10	Parameterized Smoke Tests . . . . .	79
7.11	Randomized Stress Testing . . . . .	80
7.12	Validation Summary . . . . .	80
<b>8</b>	<b>Conclusion</b>	<b>81</b>

# 1 Introduction

## 1.1 Context and Motivation of the Project

In modern computer systems, the performance between the CPU and the main memory has become a major challenge to achieving high-speed execution. While processors can execute billions of instructions per second, depending on the frequency, the latency of main memory has not improved at the same speed, leading to a phenomenon known as the memory wall.

The cache memory is a small, high-speed storage unit located closer to the CPU, designed to temporarily store frequently used data and instructions [2]. By keeping the data closer to the processor, cache memory reduces the average time needed for data access and prevents the CPU from idling while waiting for information from the main memory [4].



**Figure 1:** Position of cache memory in the general memory hierarchy of a computer system.

Intuitively, we can imagine a student studying for the exam. The textbooks on the bookshelf represent the main memory, they contain all the necessary information but take time to reach. The notes kept on the desk, however, are immediately accessible and used repeatedly during study sessions. Considering this, the desk is the cache memory, storing the most relevant and frequently referenced materials. The cache memory organizes its contents for the CPU to have minimal delay, like the student organizes their desk to study efficiently.

This process is guided by the principles of temporal locality, the tendency to reuse recently accessed data, and spatial locality, the tendency to access data stored near recently accessed data [7]. To further improve efficiency, modern CPUs implement multi-level cache architectures, typically L1, L2, L3, each balancing access speed and capacity [3].

However, because cache operations occur transparently within the hardware, it can be challenging for engineers to visualize their behavior. The **Cache Memory Simulator** project was therefore developed to provide an interactive way to explore how cache configurations, mapping techniques, and replacement policies influence performance metrics such as hit rate and access latency.

## 1.2 Purpose and Objectives of the Simulator

The purpose of the project is to provide a practical and interactive tool to help people better understand how cache memory operates and how it influences the overall performance of a computer system. Although theoretical study offers fundamental principles, visualizing and experimenting with these help professionals better grasp the behavior of memory systems [3].

The main purpose of the simulator is to demonstrate how different cache configurations, mapping techniques, and replacement policies influence system performance. The simulator aims to replicate, in a simplified but accurate form, the internal processes of a CPU cache: how a data is stored, retrieved, and replaced based on user-defined configurations. It allows users to modify parameters such as:

- cache size and block size,
- mapping technique,
- replacement policy,
- and write policy.

## 1.3 Overview of the Cache Memory Concept

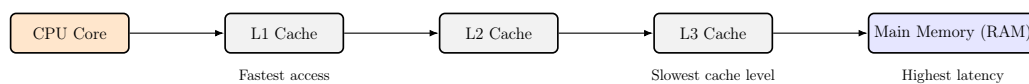
Cache memory is an intermediary storage between the CPU and main memory, designed to reduce data access latency and improve overall system throughput [2]. It stores copies of frequently accessed instructions and data so that future requests can be completed faster. The goal is to minimize the average time that the processor spends waiting for data or instructions, allowing it to maintain a high instruction execution rate.

When the CPU requests a piece of data, the cache controller first checks whether the data is already available in one of the cache levels. If it is found, the event is called a *cache hit*; otherwise, it is a *cache miss*, and the required data must be fetched from a slower level of memory [7]. The efficiency of a cache is measured by the hit rate, which is the ratio between the number of cache hits and the total number of memory accesses.

Modern processors use a multi-level cache hierarchy, typically organized into three main layers:

- **Level 1 Cache:** The smallest and fastest cache, located closest to the cores of the CPU. It is divided into separate instruction and data caches (L1i and L1d) to optimize parallel access. Access times are typically a few cycles,

- **Level 2 Cache:** Larger but slightly slower than L1, it serves as a secondary buffer that stores both instructions and data evicted from L1. L2 reduces the frequency of costly main memory accesses while maintaining high speed,
- **Level 3 Cache:** Shared among all CPU cores, acts like a large storage which contains data not found in L2. Although slower than the previous layers, it still significantly reduces main memory traffic [4].



**Figure 2:** Hierarchy of cache memory levels in relation to CPU and main memory [4].

In addition, cache memories use replacement policies to determine which block should be removed when the cache is full. The most common are:

- **Least Recently Used:** Replaces the block that has not been accessed for the longest time,
- **First-In-First-Out:** Removes the oldest block in the cache,
- **Random:** Chooses a random block to replace, which can be effective in certain workloads.

Finally, the interaction between cache and main memory is controlled through write policies. In a write-through policy, data is written to both cache and main memory simultaneously, ensuring consistency but increasing latency. In contrast, a write-back policy delays writing to main memory until the block is evicted, improving performance but increasing complexity [3].

Taking everything into account, cache memory improves system performance by exploiting temporal and spatial locality, organizing data across multiple hierarchical levels, and using intelligent mapping and replacement strategies.

## 2 Project Proposal

### 2.1 Objectives and Goals

The goal of this project is to develop an interactive **Cache Memory Simulator** in C that will help users understand how cache memory works under different configurations and conditions. The simulator will allow users to visualize cache behaviors, including the impact of different replacement policies, cache sizes, and write policies on system performance

The main objectives of the project are:

- To provide an interactive tool to visualize cache behavior in real-time and let users change cache policies and different configurations.
- To allow users to dynamically modify cache size, block size, replacement policy, and write policy.

### 2.2 Functional Requirements

The core features of the **Cache Memory Simulator** include:

- **Cache Configuration:** Users will be able to set cache size and block size.
- **Replacement Policies:** The simulator will allow users to choose from FIFO, LRU, and Random replacement policies.
- **Write Policies:** Users will have the option to choose between write-through and write-back policies.
- **Performance Metrics:** The simulator will track and display the hit rate and miss rate.
- **Interactive Interface:** The interface will allow users to visualize cache behavior, and modify parameters.

### 2.3 Nonfunctional Requirements

The simulator will meet the following nonfunctional requirements:

- **Usability:** The user interface should be intuitive, and the control of parameters and visualization of results should be easy to understand.

- **Portability:** The simulator will be developed in C to ensure compatibility with Windows and other platforms supporting .NET.
- **Scalability:** The simulator should handle larger cache configurations and complex cache systems.
- **Accuracy:** The results generated by the simulator must closely align with the expected behaviors of real-world cache systems.

## 2.4 Project Timeline

The project will be completed over a period of **13 weeks**, with the following milestones:

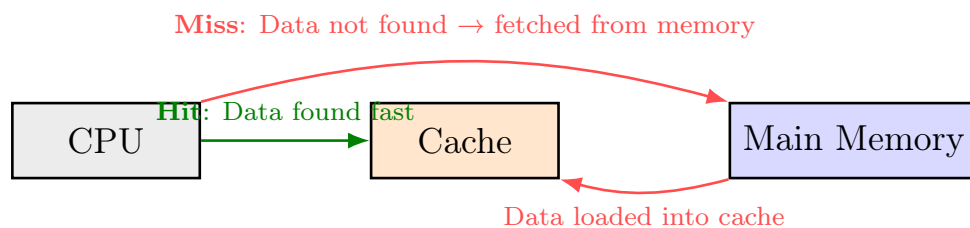
- **Weeks 1-2:** Research on cache memory, cache hierarchy (L1, L2, L3), replacement and write policies, and common simulation methods.
- **Weeks 3-4:** Focus on simulator design and user engagement. Define the user interaction flow, design the interface layout, and prototype the console-based UI to ensure intuitive control of cache parameters and visualization of results.
- **Weeks 5-6:** Implement the core functionality of the cache simulator, including cache initialization, memory access handling, and block organization. Begin basic functionality testing to verify correct reads and writes.
- **Weeks 7-8:** Integrate and test replacement policies (LRU, FIFO, Random) and write policies (write-through, write-back). Introduce runtime configuration for user-defined cache parameters. Start measuring hit and miss events.
- **Week 9-10:** Implement detailed performance metrics such as hit rate, miss rate, and average access latency. Finalize the statistics output format and verify accuracy through controlled test cases.
- **Week 11-12:** Complete the final report, integrate test results, and prepare the project for submission and presentation.
- **Week 13:** Have the project ready for presentation.

## 3 Bibliographic Study

### 3.1 Fundamental Concepts of Cache Memory

Cache memory is a small, fast storage area placed between the CPU and the main memory. Its role is to store copies of data and instructions that the processor is likely to reuse, so that future requests can be completed much faster [2, 4].

The idea is simple: if the processor has already used a piece of data once, there is a good chance that it will need it again soon. This is called temporal locality. Similarly, spatial locality is when the processor accesses one memory address, it will also probably need data from the nearby addresses as well [1]. By exploiting these properties, cache memory helps the processor spend less time waiting for data and instructions and more time doing computations.



**Figure 3:** Illustration of a cache hit versus cache miss, showing data access paths and latency differences.

Each cache is organized in blocks that hold small parts of the main memory. When the CPU requests data, the cache controller checks if that block already exists in cache hit; if not, it will be fetched from RAM, so cache miss. The efficiency of a cache is measured through the hit rate, which is the percentage of accesses that can be served directly from the cache [7].

### 3.2 Cache Types and Hierarchy (L1, L2, L3)

Modern processors use several levels of cache, organized to balance speed, size and cost [4]. The L1 cache is the smallest and fastest, located closest to the CPU core. It is usually split in two parts: one for instructions (L1i) and one for data (L1d), so the processor can fetch both in parallel. The L2 cache is larger but slightly slower and keeps data that no longer fits in L1. Finally, the L3 cache is shared among all cores and holds data that might still be reused before accessing main memory [3].

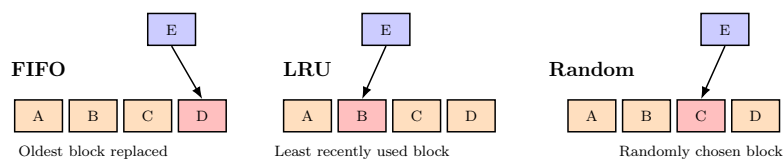
This hierarchy means that the most frequently used data is always closest to the processor, while less critical data moves further away, allowing a good balance between

performance and capacity.

### 3.3 Replacement and Write Policies

Since cache memory has limited space, it needs rules for deciding which data to remove when new information must be stored. These are known as replacement policies, the most common being: **Least Recently Used**, which removes the block that has not been accessed for the longest time; **First-In-First-Out**, which removes the oldest block in the cache; and **Random**, picks one at random, which can be easier to implement in hardware [1].

Caches also differ by how they handle write operations. In a write-through policy, data is written both to the cache and to the main memory at the same time, keeping them synchronized but slowing down the process. In a write-back policy, changes are written to memory only when that cache block is replaced, improving performance but requiring mechanisms to track modified data [3, 7].



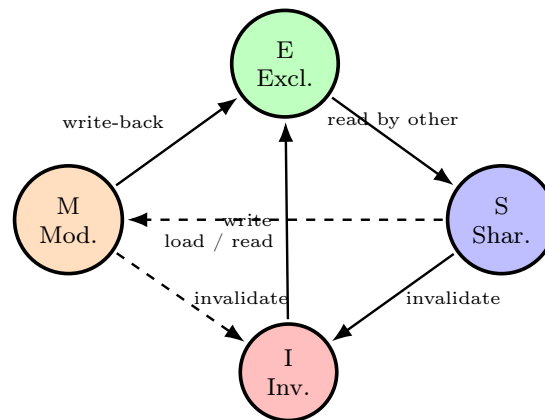
**Figure 4:** Comparison of cache replacement strategies: FIFO, Least Recently Used, and Random.

### 3.4 Locality of Reference

The concept of locality of reference lies at the heart of how caching works. It describes predictable access patterns in programs that make caching worthwhile. When a loop runs in a program, it repeatedly accesses the same instructions and variables. When data is stored sequentially in memory (like an array), accessing one element usually means the next few will also be used. Programs that are written with these patterns perform better on modern CPUs because they align naturally with how caches are designed [4].

### 3.5 Cache Coherence and Consistency Protocols

In multi-core processors, each core has its own cache. This creates a new problem, what happens when two cores access or modify the same piece of data? To prevent inconsistencies, systems use cache coherence protocols that synchronize caches automatically. Among the most common are the MSI, MESI, and MOESI protocols, which track whether a block is Modified, Exclusive, Shared, or Invalid [6, 5]. These states ensure that all cores have a consistent view of memory and avoid reading outdated data.



**Figure 5:** State transitions in the MESI cache coherence protocol [6].

However, keeping all caches coherent introduces extra communication and delays. Engineers must find a balance between consistency and speed, especially in multi-core and parallel system where data changes frequently. This trade-off is crucial in modern processors and directly influences scalability and overall performance.

### 3.6 Cache Mapping Techniques (Direct-Mapped, Fully Associative, Set-Associative)

To determine where a memory block is stored inside the cache, modern systems use several mapping techniques. These techniques define how main memory addresses are translated into cache locations, influencing both performance and hardware complexity [2, 7]. The three fundamental mapping types are: **direct-mapped**, **fully associative**, and **set-associative**.

#### Direct-Mapped Cache

In a direct-mapped cache, each block of main memory can be placed in exactly *one* specific cache line. The mapping is determined by taking the memory address modulo the number of cache lines. This system is simple and fast, but can lead to frequent conflicts when two memory addresses compete for the same line.

- **Advantages:** Fast lookup, easy hardware implementation.
- **Disadvantages:** High conflict miss rate.

#### Fully Associative Cache

In a fully associative cache, a memory block can be placed *anywhere* in the cache. There is no fixed position, so the cache controller must search all entries to find a matching

tag. Although extremely flexible, this method requires complex hardware and is expensive to scale.

- **Advantages:** Lowest conflict miss rate.
- **Disadvantages:** Slow lookup without parallel comparators; costly hardware.

### Set-Associative Cache

Set-associative caches combine the previous two methods. The cache is divided into multiple **sets**, each containing a fixed number of lines (the associativity). A memory block maps to a single set, but inside that set, it may be placed in any line.

Typical configurations include 2-way, 4-way, or 8-way associativity. Set-associative designs significantly reduce conflict misses while keeping hardware complexity manageable.

- **Advantages:** Good balance between performance and complexity.
- **Disadvantages:** Slightly slower than direct-mapped; still requires tag comparison within each set.

### Comparison

The choice of mapping technique affects how efficiently the cache can respond to varying memory access patterns:

- Direct-mapped is the fastest but suffers most from conflicts.
- Fully associative eliminates conflicts entirely but is expensive.
- Set-associative offers the best compromise and is used in nearly all modern CPUs.

Because of this, set-associative caches are standard in real-world systems—especially at L2 and L3 levels—while L1 caches often remain direct-mapped or low-associativity to ensure speed [4, 3].

## 4 Analysis

### 4.1 System Overview

The **Cache Memory Simulator** is a Python-based educational software tool designed to demonstrate the internal behavior and performance of cache memory systems in modern processors. Its main objective is to help users understand how cache configuration parameters such as size, block organization, associativity, replacement policies, and write policies affect the overall performance of memory access.

The simulator models the basic operations that take place inside a cache: when the CPU requests a piece of data, the program checks whether that data is already stored in cache (a *cache hit*) or if it must be fetched from the main memory (a *cache miss*). Each operation updates the total number of accesses, as well as the hit and miss rates, giving the user a clear picture of how efficient the cache is under different configurations.

The project is implemented in **Python**, using two main components:

- **Simulation Core:** Handles the logic of cache memory, including the organization of blocks, mapping functions, replacement policies (such as LRU, FIFO, or Random) and write policies (write-through and write-back).
- **Graphical Interface:** Built using the `tkinter` library, it provides an easy-to-use interface where users can enter parameters, start or stop the simulation, and view results in real time.

The interface allows the user to configure parameters such as cache size, block size, and replacement policy before starting the simulation. Once the program runs, the results are displayed dynamically, including the number of hits, misses, and the calculated hit rate.

The simulator focuses on clarity and usability rather than hardware-level precision. Some physical characteristics of real caches, like timing delays or coherence between multiple cache levels, are simplified to keep the system lightweight and easy to use. Even so, the project accurately captures the essential logic of how cache memory operates and how performance can vary depending on configuration.

The system's intended users are students and engineers studying computer architecture, as well as anyone wishing to experiment with caching principles in a controlled and visual environment. It provides an accessible way to explore how hardware design decisions translate into measurable performance outcomes.

## 4.2 Functional Requirements and Mechanisms

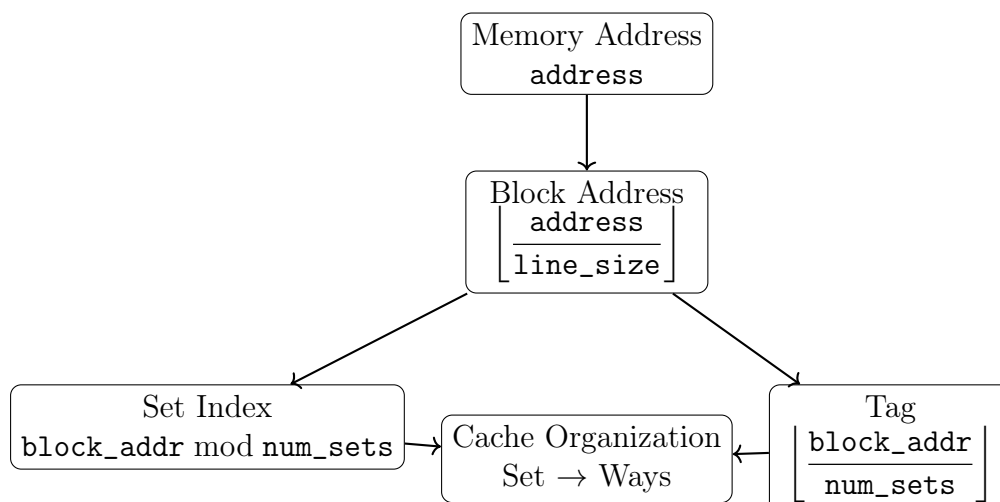
The main objective of the simulator is to model the functional behavior of a CPU cache interacting with main memory (RAM), with an emphasis on correctness, configurability, and observability of cache mechanisms. The simulator focuses on reproducing cache-level effects such as hits, misses, evictions, and write propagation, rather than cycle-accurate timing.

The main functional requirements of the system are described below.

- **Cache Configuration and Initialization:** The simulator allows the cache to be created and initialized with fully configurable parameters, including:
  - total number of cache blocks (`num_blocks`),
  - cache associativity,
  - cache line size (`line_size`),
  - replacement policy (LRU, FIFO, or Random),
  - write policy (write-back or write-through),

These parameters determine the internal organization of the cache, including the number of sets, the mapping of memory addresses, and the behavior on cache misses and evictions.

- **Address Mapping and Cache Organization:** Each memory address is mapped to a cache line using block-based addressing. The simulator computes the block address, set index, and tag based on the configured line size and number of sets. This mechanism ensures consistent behavior for both direct-mapped and set-associative cache organizations.



**Figure 6:** Address mapping and cache organization using block-based addressing

- **Cache Access Mechanism:** The core operation of the simulator is a generic cache access function that supports both read and write requests. For each access, the simulator determines whether the request results in a cache hit or a cache miss and returns detailed information about:
  - hit or miss outcome,
  - set index and way index (if applicable),
  - evicted cache block (if any),
  - main memory read and write activity.

This explicit return contract enables precise testing and analysis of cache behavior.

- **Replacement Policies:** When a cache miss occurs and a set is full, the simulator evicts a cache line according to the selected replacement policy:
  - LRU (Least Recently Used),
  - FIFO (First-In, First-Out),
  - Random.

For deterministic policies such as LRU and FIFO, the simulator guarantees predictable behavior to support reproducible testing.

- **Write Policies and Write-Miss Handling:** The simulator correctly implements both write-back and write-through policies:
  - In *write-back*, write hits mark cache lines as dirty and defer updates to main memory until eviction.
  - In *write-through*, every write hit immediately propagates the update to main memory.

Additionally, write misses are handled according to the selected write-miss policy:

- *write-allocate*, which loads the block into cache before writing,
  - *write-no-allocate*, which bypasses the cache and writes directly to main memory.
- **Eviction and Write-Back Mechanism:** When a dirty cache line is evicted under the write-back policy, the simulator performs a write-back to main memory at the base-aligned address of the cache block. This ensures that memory consistency is preserved across cache replacements.

- **Main Memory (RAM) Model:** A simplified RAM model is included, supporting explicit read and write operations. The RAM enforces address bounds and raises errors for invalid accesses, allowing the simulator to detect and handle out-of-range memory operations.
- **Simulation Control and Reset:** The simulator supports step-by-step execution of memory access sequences, enabling detailed inspection of cache state after each operation. A reset mechanism invalidates all cache lines and clears dirty bits, returning the cache to a clean initial state.
- **Observability and Testing Support:** Each cache access reports explicit memory read and memory write flags, making side effects on main memory observable. This design choice facilitates systematic unit testing and validation of all cache behaviors, including edge cases and policy interactions.

From a functional perspective, the simulator prioritizes correctness, transparency, and configurability. While it does not model hardware-level timing or pipelined execution, it accurately captures the logical behavior of cache memory systems and provides a controlled environment for studying the impact of cache parameters and access patterns on memory performance.

### 4.3 Nonfunctional Requirements and Mechanisms

Besides the main functionalities, the simulator must also ensure the program runs efficiently, is easy to use, and produces accurate results. These aspects are essential to make the project both technically correct and practical for educational use.

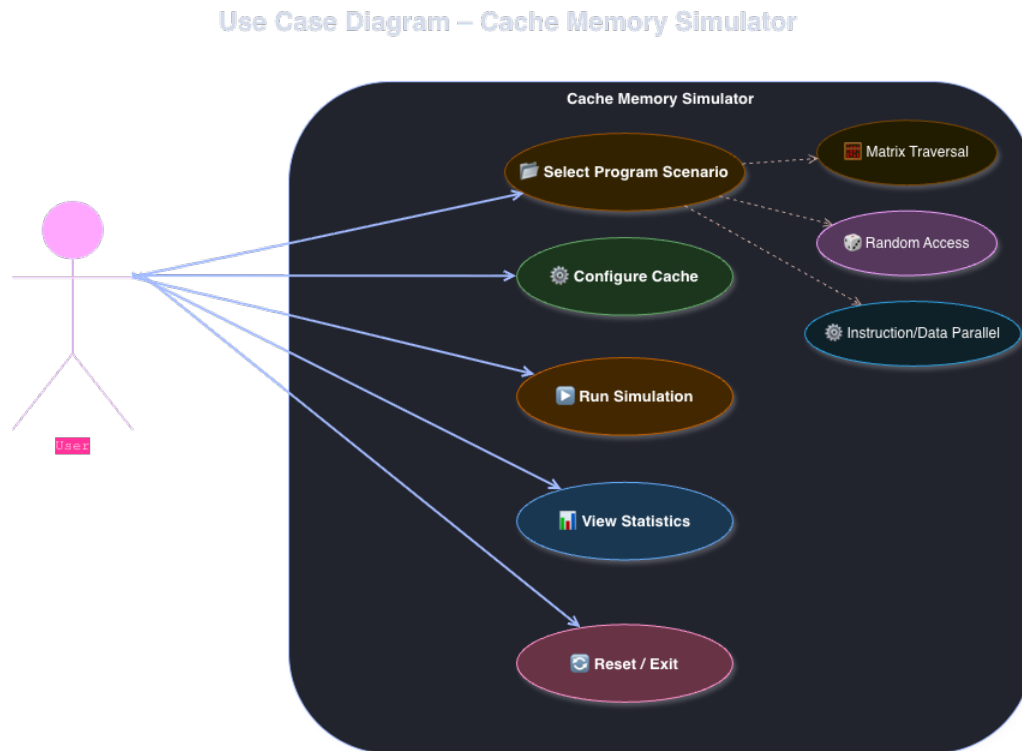
- **Usability:** The interface must be intuitive and simple to understand, even for users who are not familiar with cache architecture. Through `tkinter`, the simulator offers clear menus and buttons to configure cache parameters, select predefined programs, and start or stop the simulation. Results such as hit rate, miss rate, and access time are displayed live, allowing users to interpret outcomes immediately.
- **Performance and Efficiency:** The simulator should process long access sequences without noticeable delay. Since it is implemented in Python, the focus is on efficient use of data structures (lists, dictionaries, queues) to keep the simulation responsive. The goal is not to simulate hardware cycles in detail, but to ensure that results are updated smoothly and consistently.

- **Portability:** The program runs on any platform that supports Python 3 and `tkinter` (Windows, Linux, macOS). No additional installations or dependencies are required.
- **Scalability:** The code structure allows future extensions, such as adding multiple cache levels (L2, L3), multi-core support, or new replacement policies. Because each component (cache logic, GUI, and statistics) is modular, changes can be introduced easily without affecting the rest of the system.
- **Accuracy and Reliability:** Even though the simulator simplifies certain hardware details, it follows the same logical steps as a real cache controller. All operations: tag comparison, replacement decisions, and write policies, are implemented according to theoretical principles. The computed hit and miss rates should always match expected outcomes for the given access patterns.
- **User Feedback and Visualization:** During the simulation, visual feedback (colored indicators or counters) helps users distinguish between hits and misses. The simulator should remain responsive to user actions and allow configuration changes without restarting the application.
- **Maintainability:** The code is written in a clear, modular structure with comments that explain the main functions. This ensures that it can be easily modified, expanded, or integrated into other projects.

Overall, these nonfunctional requirements ensure that the simulator remains reliable, portable, and user-friendly.

## 4.4 Use Case Diagram and Descriptions

The use case diagram illustrates the interaction between the user and the main components of the Cache Memory Simulator. The system allows the user to configure cache parameters, select one of the predefined program scenarios, run simulations, and view performance statistics in real time.



**Figure 7:** Use case diagram of the Cache Memory Simulator.

Each use case represents a functional module of the application:

### 1. Configure Cache

The user can set parameters such as cache size, block size, associativity, replacement policy (LRU, FIFO, or Random), and write policy (write-through or write-back). This enables hands-on experimentation with different configurations.

### 2. Select Program Scenario

The simulator includes four predefined test programs, each demonstrating a different cache access pattern:

- **Matrix Traversal** – demonstrates spatial locality through sequential memory access,
- **Random Access** – represents non-localized memory access, leading to frequent cache misses,
- **Instruction/Data Parallel Access** – simulates the behavior of L1i and L1d caches operating in parallel.

### 3. Run Simulation

Once configured, the user runs the simulation to see how cache hits and misses occur over time. Each access request is evaluated by the simulator's core logic, and results are displayed in real time in the user interface.

### 4. View Statistics

The simulator calculates and displays performance metrics including the number of hits, misses, hit rate, miss rate, and average access time. These values are updated live and provide immediate visual feedback on cache efficiency.

### 5. Reset / Exit

The user can reset the current configuration to start a new simulation or exit the program. This clears all stored statistics and restores the default settings for the next experiment.

### Data Flow

The simulator reads the configuration parameters through the graphical interface, processes the generated address sequence, and displays live results to the user. These results can optionally be exported to a file for later analysis.

## 4.5 System Constraints and Limitations

Although the simulator provides a clear and interactive representation of cache operations, it also introduces certain simplifications compared to real hardware systems. These constraints are intentional, as the main objective of the project is educational visualization, not full-scale hardware emulation.

- **Timing and Latency:** The simulator does not model exact CPU clock cycles or bus communication delays. Instead, it focuses on logical correctness, whether data is found in cache or not.
- **Single-Core Model:** The current version simulates cache behavior for a single processor core. Multi-core and coherence mechanisms (such as MESI protocols) are discussed theoretically but not implemented.
- **Simplified Addressing:** Memory addresses are represented as integer values, without detailed byte-level or page-level management.

Despite these limitations, the simulator accurately reflects the logical behavior of

real cache systems. Future versions could extend the model to include multi-level cache hierarchies, multi-core support, and advanced visualization tools.

## 5 Design

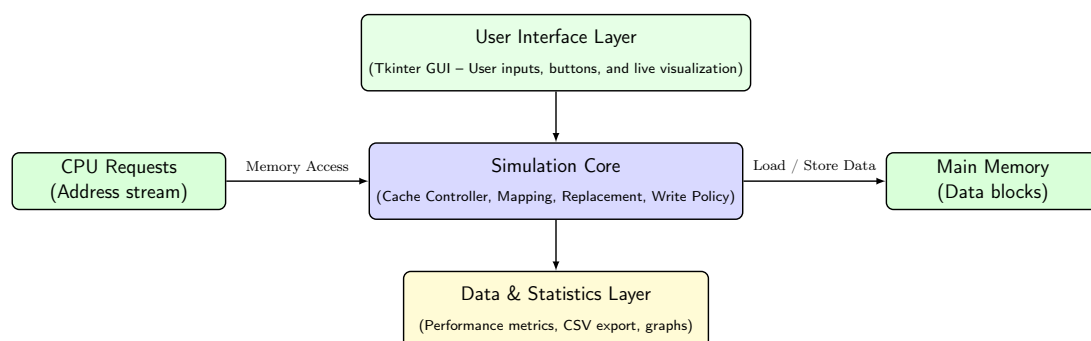
### 5.1 Overview

The design of the simulator follows a modular and layered structure to ensure clarity, scalability, and maintainability. Each major component is separated logically, allowing independent updates and clear communication between modules. This approach provides flexibility for future extensions such as multi-level caches or additional replacement policies.

The simulator is composed of three main subsystems:

- **User Interface Layer** – Handles user interaction and visualization. It manages input parameters such as cache size, replacement policy, and write mode, and displays results in real time.
- **Simulation Core** – Implements the internal behavior of the cache memory system. It manages memory access requests, determines cache hits and misses, applies mapping and replacement policies, and updates statistical metrics.
- **Data and Statistics Layer** – Collects, stores, and exports results. It provides computed statistics (hits, misses, hit rate) and optional data logging in CSV format.

The system uses a clear flow of communication: user input is processed by the interface, interpreted by the simulation core, and finally reflected visually through updated metrics and display elements. This separation of concerns also makes it easy to debug and extend the project.



**Figure 8:** Layered design overview of the Cache Memory Simulator showing the main system modules and data flow.

This layered design ensures that the simulator remains both educational and technically sound. The graphical interface focuses on usability and visualization, the

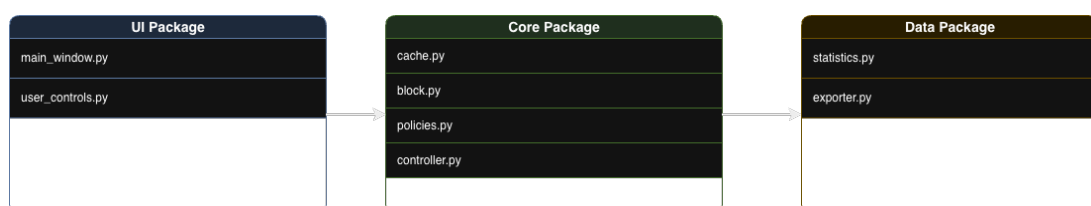
simulation core on correctness and logic, and the data layer on traceability and analysis. Together, these components form a cohesive framework that accurately models how cache memory operates in real systems.

## 5.2 Package Diagram

The internal structure of the Cache Memory Simulator is organized into a set of Python modules, grouped into packages. Each package corresponds to one of the main architectural layers described earlier — *UI*, *Core*, and *Data*.

- **UI** – Contains all elements related to the graphical interface, built with `tkinter`. It defines the main application window, parameter input forms, and real-time visual output. Key module: `ui/main_window.py`.
- **Core** – Implements the simulation logic, including cache organization, mapping, replacement, and write policy mechanisms. Core modules:
  - `core/cache.py` – Main cache structure and lookup logic.
  - `core/block.py` – Represents a single cache block (tag, valid bit, data).
  - `core/policies.py` – Contains replacement algorithms (FIFO, LRU, Random).
  - `core/controller.py` – Orchestrates memory access and connects the simulation core with the user interface.
- **Data** – Responsible for statistics collection and optional export. Contains modules for tracking hits, misses, and saving results in CSV format. Key module: `data/statistics.py`.

All packages interact in a top-down manner: user actions in the `ui` package trigger cache operations in the `core`, while results are logged and displayed through the `data` package. This separation provides a clear boundary between logic and visualization, making the system easily maintainable.



**Figure 9:** Package diagram of the Cache Memory Simulator.

### 5.3 Class Diagram

The internal structure of the simulator is represented through the UML class diagram shown below. This diagram provides a detailed view of the main software components, their attributes and methods, and how they interact within the system architecture. Each package (*UI*, *Core*, and *Data*) corresponds to a specific layer in the system, reflecting the modular design of the simulator.

The **UI package** contains the graphical interface classes responsible for user interaction and visualization. The `UIController` class manages the application window, collects user inputs, and communicates with the simulation core to start and control simulations.

The **Core package** implements the main cache logic. The `CacheSimulator` class acts as the central controller, coordinating data requests between the user interface and the cache itself. The `Cache` class manages cache blocks and replacement policies, while each `CacheBlock` stores metadata such as tag, valid bit, and data content. Replacement policies are implemented as separate subclasses (`LRUPolicy`, `FIFOPolicy`, `RandomPolicy`) that inherit from an abstract base class `ReplacementPolicy`, ensuring extensibility and clear separation of behaviors.

The **Data package** provides utility components for logging and result processing. The `Statistics` class records hit, miss, and access counts and computes the hit rate. The `Exporter` class allows the simulation data to be saved or visualized externally (as CSV files or plots).

This layered organization ensures that the graphical interface never directly manipulates low-level cache data structures, but instead communicates through the simulation core. Such a structure aligns with the **Model–View–Controller (MVC)** design pattern, improving maintainability and code reuse.



**Figure 10:** UML Class Diagram of the Cache Memory Simulator, showing relationships between the main classes across the UI, Core, and Data packages.

The relationships between these classes define a clear hierarchy of responsibility:

- The **UIController** interacts with **CacheSimulator** to start, stop, or reset simulations.
- The **CacheSimulator** delegates memory operations to the **Cache** class and gathers statistics through **Statistics**.
- The **Cache** manages multiple **CacheBlock** instances and applies a specific **ReplacementPolicy** strategy.
- The **Statistics** and **Exporter** classes support result analysis and external reporting.

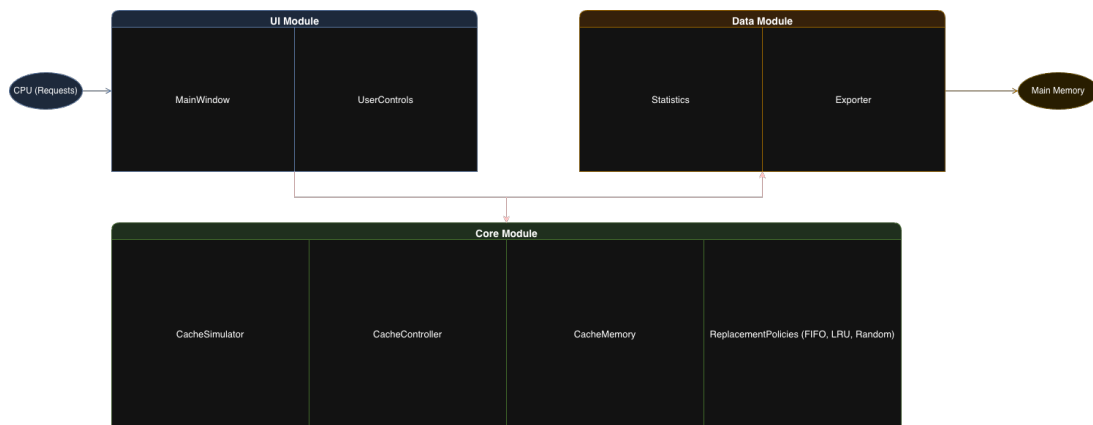
Overall, this class design emphasizes modularity and separation of concerns. Each class has a well-defined role, allowing new cache behaviors or visualization modules to be added in future versions without altering the simulator's core logic.

## 5.4 Component and Block Diagram

These diagrams illustrate how the main software components interact during runtime, showing the flow of control and data between modules.

### Component Diagram

The component diagram provides a high-level overview of the simulator's architecture. It shows how the system is divided into three distinct modules: the graphical interface (UI), the simulation core, and the data utilities; and how they communicate through interfaces.



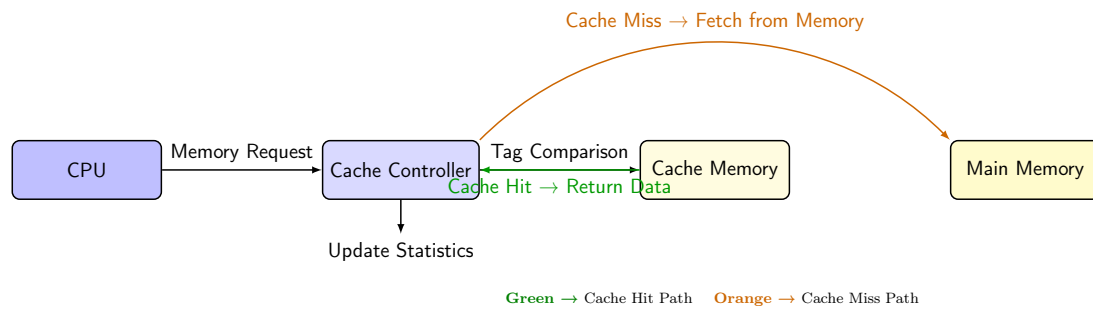
**Figure 11:** Component diagram of the Cache Memory Simulator, showing communication between major software modules.

Each component has a specific role within the system:

- **UI Module:** Provides user interaction, allowing cache parameters to be configured, simulation scenarios to be selected, and results to be visualized in real time.
- **Simulation Core:** Executes the cache operations, managing data access, replacement policies, and performance statistics.
- **Data and Export Module:** Handles performance metrics and allows results to be stored externally as CSV files or graphical outputs.

### Block Diagram

The block diagram provides a more detailed operational view of how cache accesses are processed internally by the simulator. It shows the flow of information between the CPU, cache controller, and main memory, illustrating what happens during a cache hit and a cache miss.



**Figure 12:** Block diagram showing the internal operational flow of cache access in the simulator.

### Legend:

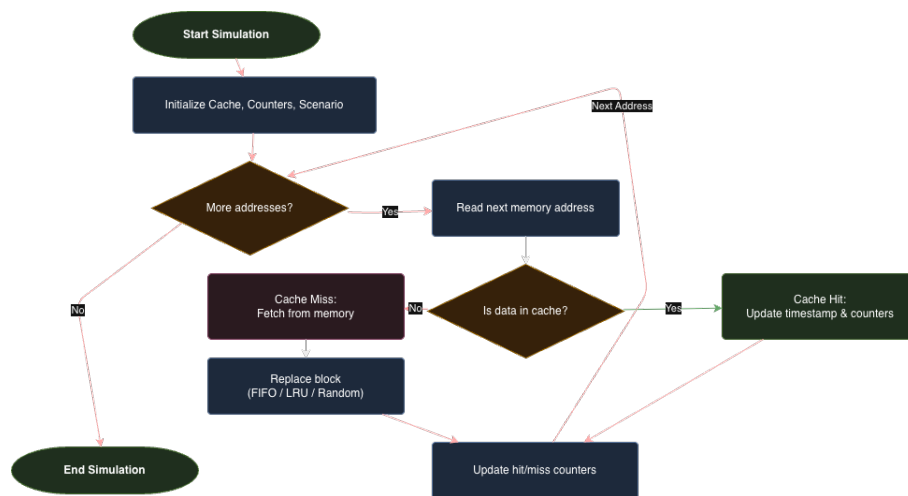
- **Blue blocks** – Processing components (CPU, Controller).
- **Yellow blocks** – Memory components (Cache, Main Memory).
- **Green arrow** – Cache hit path.
- **Orange arrow** – Cache miss path.

During operation:

1. The simulator sends a memory request to the *Cache Controller*.
2. The controller checks if the requested address exists in the *Cache Memory*.
3. If the block is found —> a **cache hit** occurs, and data is returned immediately.
4. If not found —> a **cache miss** occurs, and the block is fetched from *Main Memory* and stored in the cache.
5. Performance statistics such as total accesses, hits, and misses are updated dynamically.

## 5.5 Flow Chart

The flow chart below illustrates the sequence of operations during a cache simulation. It represents the decision logic used to determine whether a memory access results in a hit or a miss, and how replacement policies are applied accordingly.



**Figure 13:** Flow chart showing the main decision steps executed during cache simulation.

It clearly depicts how the simulator initializes, checks for cache hits, applies replacement policies on misses, and updates performance statistics dynamically.

## 5.6 Data Structure Design

The simulator uses object-oriented data structures that closely model real cache hardware. Each cache block is represented as an instance of the `CacheBlock` class, containing fields such as:

- **tag** – identifies which memory address the block corresponds to,
- **valid** – indicates whether the block contains valid data,
- **data** – stores the simulated content,
- **timestamp** – used for LRU replacement policy.

The cache itself is stored as a list (or 2D list for set-associative mapping) of these blocks. Hash maps are used to accelerate tag lookup and replacement selection. This design enables constant-time access checks and efficient block replacement decisions.

This structure ensures the simulator remains simple but logically faithful to real hardware implementations, making it easy to expand with additional policies or multi-level cache hierarchies in the future.

## 5.7 User Interface

The graphical user interface of the simulator is organised into four functional areas designed to make cache behaviour easy to observe and experiment with. The interface

allows users to configure the cache, visualise address decoding, inspect internal cache state, and control the execution of the simulation.

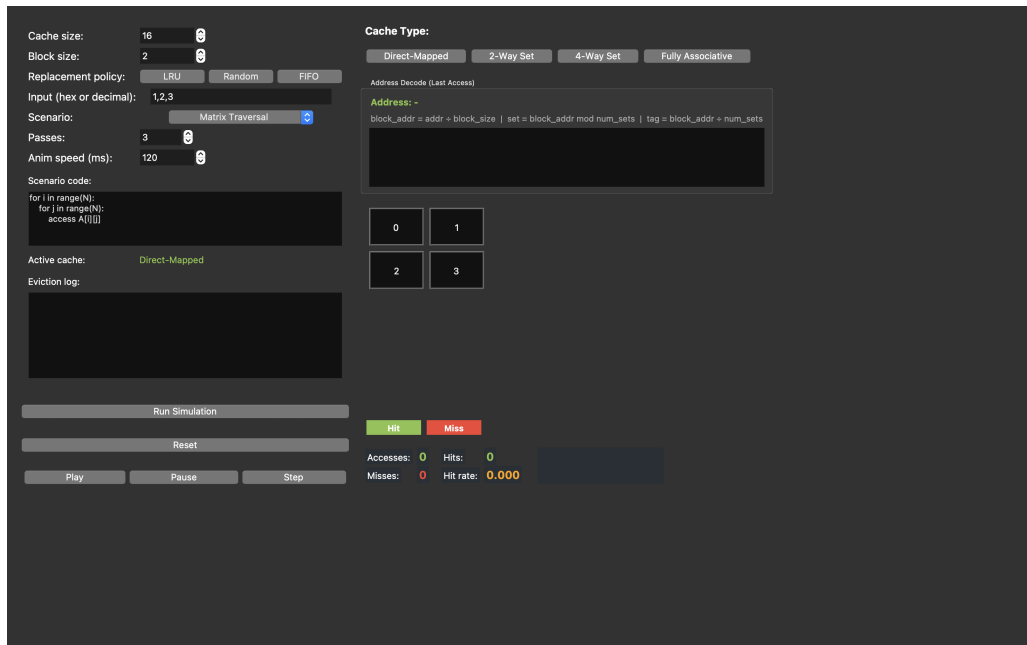


Figure 14: Simulator interface before running

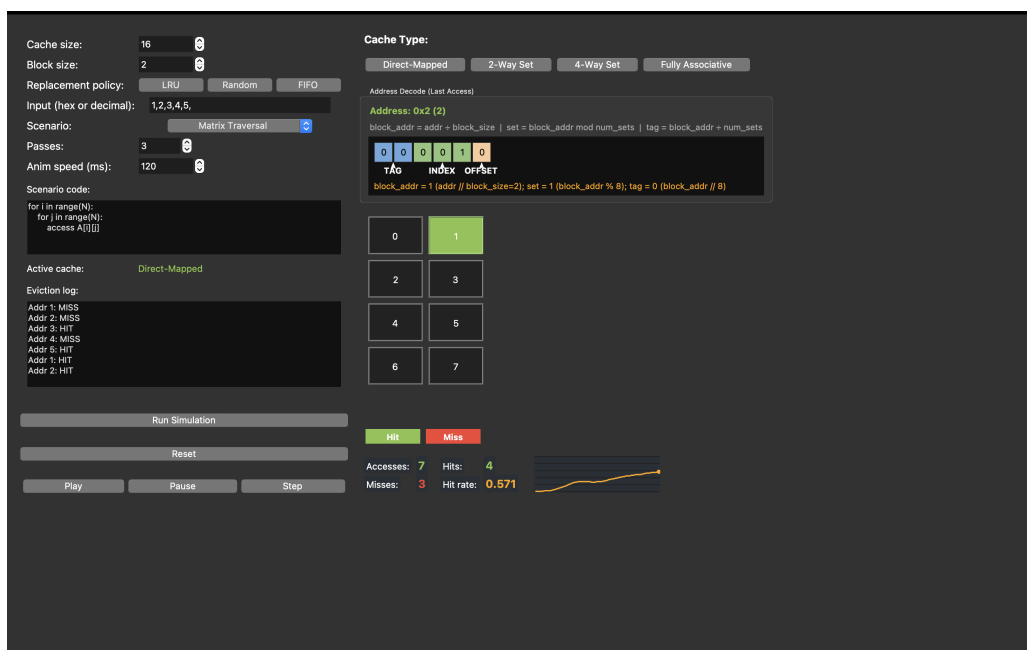


Figure 15: Simulator interface during running

### 5.7.1 Configuration Panel

On the left side of the interface, the configuration panel collects all parameters required to define a simulation:

- **Cache size**, **Block size**, and **Replacement policy** configure the structural properties of the cache.
- **Input (hex or decimal)** allows the user to enter a sequence of memory addresses that will be processed by the simulator.
- **Scenario** and the corresponding **Scenario code** field display the selected predefined workload pattern (e.g., *Matrix Traversal*, *String Processing*), together with pseudocode illustrating the memory access behaviour.
- **Passes** specifies how many times the selected scenario is executed.
- **Animation speed** controls the time delay between visual steps.

This part of the interface acts as a setup dashboard, enabling users to fully define both the cache model and the workload before starting the simulation.

### 5.7.2 Address Decode Panel

At the top centre of the interface, the Address Decode panel explains in detail how the most recently accessed address is interpreted by the cache.

The panel displays:

- the address in hexadecimal and decimal format,
- the formulas used to compute the block address, set index, and tag,
- a graphical highlighting of the **TAG**, **INDEX**, and **OFFSET** fields.

This component serves an educational purpose by showing how direct-mapped and set-associative caches break down memory addresses.

### 5.7.3 Cache Visualisation

The central region of the interface presents a grid-like representation of the cache. Each cell corresponds to a block or a set, depending on the selected associativity.

- Cells are coloured to show the status of the last access: green for *hit* and red for *miss*.
- When an address is processed, the simulator highlights the corresponding set, shows the stored tag, and performs any required replacement-policy updates (LRU, FIFO, or Random).

This visualisation provides an intuitive understanding of how data moves through the cache and how conflicts generate evictions.

#### 5.7.4 Eviction Log and Statistics

Below the configuration panel, the simulator maintains an **Eviction log** that records every access, including whether it resulted in a hit or a miss, and any evictions produced by set conflicts.

To the right of the interface, real-time statistics are displayed:

- total number of accesses,
- number of hits and misses,
- current hit rate.

A small time-series graph updates after each access, showing the evolution of the hit rate during execution. This helps users observe how locality patterns influence cache efficiency.

#### 5.7.5 Execution Controls

At the bottom of the interface, users can control the simulation through the following buttons:

- **Run Simulation** starts the full execution with animations enabled.
- **Reset** clears the cache and restores the initial interface state.
- **Play**, **Pause**, and **Step** allow fine-grained control of the simulation, enabling step-by-step inspection of each individual address access.

These controls transform the UI into a flexible educational tool suitable for demonstrations, debugging, and interactive exploration of cache behaviour.

## 6 Implementation

### 6.1 IT Technology

The main programming language used is **Python 3**, chosen for its simplicity, readability, and extensive support for both user interface development and algorithmic computation.

#### Programming Language: Python 3

Python provides a clear syntax and a wide range of libraries that make it ideal for implementing algorithms, graphical interfaces, and data analysis in a single environment. Its object-oriented structure allows modular implementation of cache components, such as cache blocks, mapping functions, and replacement policies.

#### Graphical Interface: Tkinter

The user interface is built using the `tkinter` library, which is part of the Python standard distribution. It allows rapid development of interactive graphical interfaces with buttons, labels, and input fields. Tkinter provides enough flexibility to visualize cache simulation results in real time, without requiring external frameworks.

#### Simulation and Data Handling Libraries

The simulator relies on Python's built-in data structures: `lists`, `dictionaries`, and `queues` for efficient memory access simulation and statistical computation. Additional libraries such as `matplotlib` or `pandas` will be used for graphical output and performance data analysis.

#### Version Control and Documentation

Version management and collaboration were performed using `GitHub`, while documentation was written in  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  using the `Overleaf` platform. This workflow ensures reproducibility, clean structure, and easy updates of both source code and documentation.

#### Reason for Technology Selection

The choice of Python and Tkinter was made because the focus of this project is educational visualization, not high-performance hardware emulation. Python allows fast prototyping and provides an accessible interface.

## 6.2 Algorithms and Simulation Scenarios

### 6.2.1 Cache Mechanism

The cache mechanism represents the core functional logic of the simulator and defines how each memory access is processed. Its role is to determine whether a requested address is already present in the cache (cache hit) or must be fetched from main memory (cache miss), while correctly applying mapping, replacement, and write policies.

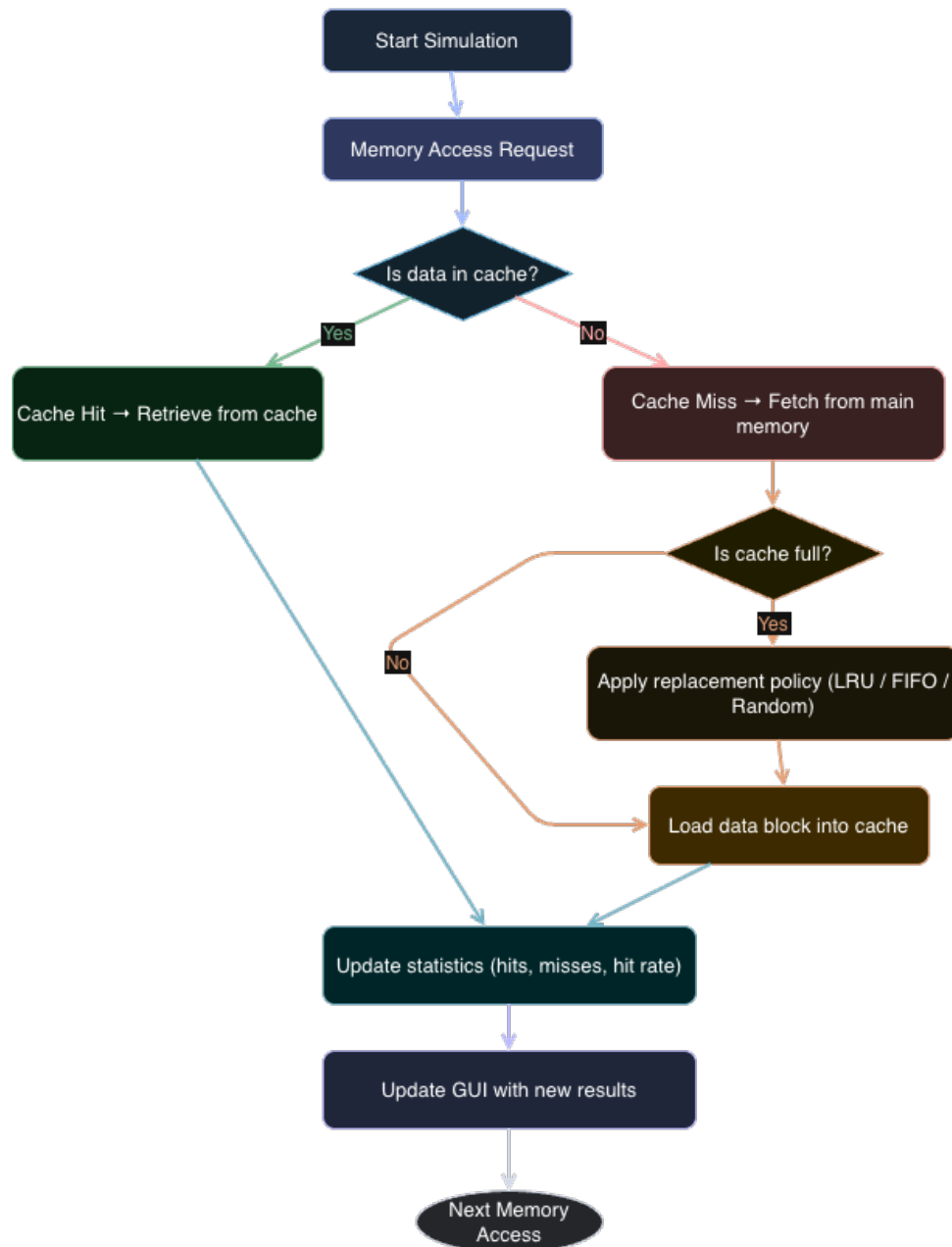
All memory operations are handled through a unified cache access routine, which mediates communication between the simulated CPU, the cache structure, and the RAM model.

#### Working Principle:

1. A memory address is issued by the simulation engine as a read or write request.
2. The cache decomposes the address into *block address*, *set index*, and *tag*, based on the configured line size and associativity.
3. The cache searches the selected set for a valid line with a matching tag:
  - if a match is found, the access is classified as a **cache hit**;
  - otherwise, a **cache miss** is generated.
4. On a cache miss, the simulator loads the required block from main memory and, if the set is full, evicts an existing cache line according to the selected replacement policy.
5. For write operations, the simulator applies the configured write policy (write-back or write-through) and write-miss policy (write-allocate or write-no-allocate), possibly marking cache lines as dirty or triggering immediate memory writes.

Each cache access produces explicit feedback regarding hit or miss status, evictions, and main memory activity. This design makes the impact of cache parameters—such as cache size, line size, associativity, and policy selection—directly observable through the resulting hit and miss rates.

### Cache Mechanism Flow



**Figure 16:** General flow of cache operation in the simulator.

The simulator reproduces the behavior of cache systems using simplified algorithmic models that emulate different memory access patterns. These algorithms are designed to demonstrate how cache efficiency depends on temporal and spatial locality, and how different access types affect the hit and miss rate.

Each scenario corresponds to a specific program written in C++ style pseudocode. The simulator converts these access sequences into numerical addresses and processes them

step by step, according to the selected cache configuration and replacement policy.

### 6.2.2 Replacement Mechanism

The replacement mechanism defines how the simulator selects a cache line to be evicted when a cache miss occurs and the target set is already full. This mechanism is essential for modeling realistic cache behavior, as it determines which data is discarded to make room for newly loaded blocks.

In the simulator, replacement decisions are handled independently of address mapping and are applied uniformly across all cache organizations (direct-mapped, set-associative, and fully associative).

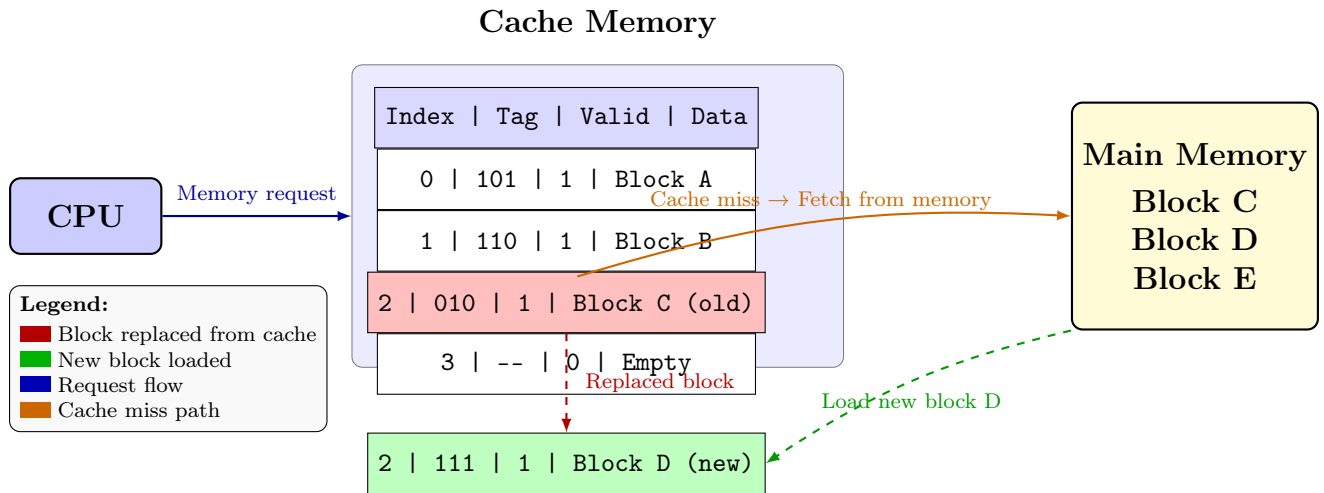
#### Working Principle:

1. When a cache miss occurs, the cache checks whether the target set contains any invalid (free) cache lines.
2. If a free line is available, the new block is loaded without requiring eviction.
3. If the set is full, the replacement mechanism selects one cache line for eviction based on the configured replacement policy.
4. If the evicted line is marked as dirty and the write-back policy is enabled, the simulator writes the block back to main memory before replacement.
5. The new block is then inserted into the cache, and internal replacement metadata is updated accordingly.

The simulator supports three replacement policies, each illustrating a different design trade-off:

- **FIFO (First-In, First-Out)** – evicts the cache line that has been present in the set for the longest time.
- **LRU (Least Recently Used)** – evicts the cache line that has not been accessed for the longest period.
- **Random** – evicts a randomly selected cache line within the set.

This modular replacement mechanism allows users to observe how different eviction strategies affect cache performance, conflict behavior, and overall hit and miss rates.



**Figure 17:** Cache replacement process showing how an old block is evicted and a new block is fetched from main memory.

### 6.2.3 FIFO (First-In, First-Out) Policy

The FIFO replacement policy evicts the cache line that has been resident in the cache for the longest time, independent of how frequently it has been accessed. This policy is simple to implement and highlights the limitations of replacement strategies that do not account for temporal locality.

#### Working Principle:

1. For each cache set, the simulator maintains an ordered structure representing the insertion order of cache lines.
2. On a cache miss:
  - if a free (invalid) cache line exists, the new block is placed into that line;
  - if the set is full, the oldest cache line (first inserted) is selected for eviction.
3. The new block is inserted at the end of the FIFO structure, and the order is updated accordingly.

The FIFO policy is implemented with minimal bookkeeping and provides deterministic eviction behavior, which is useful for controlled experiments and testing.

#### Listing 1: FIFO eviction selection (simplified)

```
def select_victim_fifo(queue):
    # queue stores way indices in insertion order
```

```
return queue.pop(0)
```

#### 6.2.4 LRU (Least Recently Used) Policy

The LRU replacement policy evicts the cache line that has not been accessed for the longest period of time. It exploits temporal locality by assuming that recently used data is more likely to be accessed again in the near future.

##### Working Principle:

1. For each cache set, the simulator tracks the access order of cache lines.
2. On every cache hit, the accessed cache line is marked as most recently used.
3. On a cache miss:
  - if a free cache line exists, the new block is loaded;
  - if the set is full, the least recently used cache line is selected for eviction.
4. Replacement metadata is updated after each access to preserve correct ordering.

LRU provides improved hit rates compared to FIFO for workloads with strong temporal locality, at the cost of additional bookkeeping.

##### Listing 2: LRU eviction update and selection (simplified)

```
def update_lru(lru_list, way):
    if way in lru_list:
        lru_list.remove(way)
    lru_list.append(way)

def select_victim_lru(lru_list):
    # least recently used is at the front
    return lru_list.pop(0)
```

#### 6.2.5 Random Replacement Policy

The Random replacement policy selects a cache line uniformly at random when an eviction is required. This policy has the lowest implementation complexity and avoids the overhead of maintaining access history.

### Working Principle:

1. On a cache miss:
  - if a free cache line exists, the new block is loaded;
  - if the set is full, one cache line is selected randomly and replaced.
2. No ordering or access history is maintained between accesses.

Although its performance is generally inferior to LRU, Random replacement can perform surprisingly well in highly irregular workloads and serves as a useful baseline for comparison.

**Listing 3:** Random eviction selection (simplified)

```

import random

def select_victim_random(num_ways):
    return random.randint(0, num_ways - 1)

```

#### 6.2.6 Scenario 1: Matrix Traversal (Spatial Locality)

This scenario models a sequential traversal of a two-dimensional matrix stored in row-major order. The generated access pattern emphasizes *spatial locality*, as consecutive memory accesses target adjacent addresses in main memory.

##### Access Pattern:

```

for (int i = 0; i < N; i++)
    for (int j = 0; j < M; j++)
        sum += A[i][j];

```

In the simulator, this scenario is **hardcoded** as a predefined memory access trace. The nested loop is converted into a fixed, linear sequence of addresses corresponding to contiguous matrix elements.

As a result, once a cache block is loaded, subsequent accesses to elements within the same cache line frequently result in cache hits. The first access to each cache line produces a miss, followed by multiple hits while iterating over neighboring elements.

This hardcoded scenario ensures deterministic behavior and allows users to clearly observe the effect of spatial locality without interference from dynamic or input-dependent variations.

### 6.2.7 Scenario 2: Random Access (No Locality)

This scenario simulates workloads with irregular memory access patterns, such as hash table lookups or unpredictable data queries. Addresses are selected pseudo-randomly from the available address space, resulting in minimal spatial and temporal locality.

#### Access Pattern:

```
for (int i = 0; i < N; i++) {  
    int index = rand() % N;  
    process(data[index]);  
}
```

In the simulator, this scenario is also **hardcoded** as a predefined access sequence. A pseudo-random generator is used to produce a fixed sequence of addresses before the simulation begins.

Because consecutive accesses are unlikely to target the same or neighboring cache lines, most memory requests result in cache misses. This scenario demonstrates how cache performance degrades when access patterns do not align with cache organization.

By hardcoding the access trace, the simulator guarantees reproducibility and allows direct comparison between different cache configurations and replacement policies under identical conditions.

#### Simulation Flow

Each algorithm generates a sequence of memory accesses. The simulator iterates through these accesses and determines, for each one:

1. if the requested address is present in the cache (hit),
2. if not, a miss occurs and the block is loaded from main memory,
3. if the cache is full, a replacement policy (LRU, FIFO, or Random) is applied,
4. statistics are updated after each access.

The results, including total accesses, hits, misses, and hit rate, are updated in real time on the user interface, providing a visual comparison between different scenarios. These algorithmic models allow the simulator to reproduce realistic cache behavior without requiring actual hardware execution.

## 6.3 Data Input and Output

The simulator operates on a well-defined flow of input data that describes cache configuration and memory access behavior, and produces structured output data that reflects cache performance. This separation between input and output enables systematic experimentation with different cache parameters and access patterns.

### Input Data

The input to the simulator consists of three main components:

- **Cache configuration parameters:** cache size (number of blocks), line size, associativity, replacement policy, write policy, and write-miss policy. These parameters are provided through the graphical user interface and directly determine the internal cache structure.
- **Simulation scenario:** one of the predefined, hardcoded access patterns (e.g., matrix traversal or random access), or a manually specified address sequence.
- **Memory access sequence:** a normalized list of memory addresses (and optional write operations) generated by the simulation engine before execution begins.

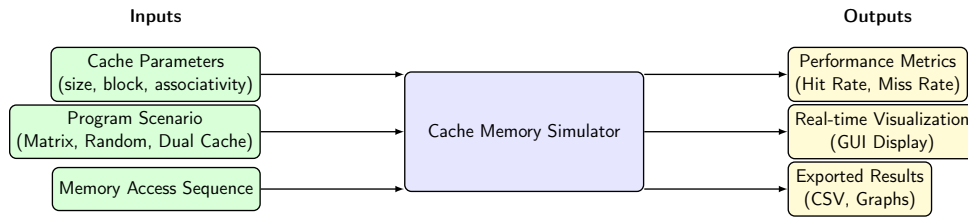
All input data is validated and normalized prior to execution to ensure compatibility with the selected cache configuration.

### Output Data

For each simulation run, the system produces detailed performance information derived from the execution of the access sequence:

- total number of processed memory accesses;
- number of cache hits and cache misses;
- calculated hit rate and miss rate;
- number of main memory read and write operations;
- information about evicted cache lines (when applicable).

These outputs are updated incrementally after each simulation step and are displayed in real time through the graphical interface. The same structured output data can also be used for logging, debugging, or offline analysis, enabling direct comparison between different cache configurations and scenarios.



**Figure 18:** Simplified black box representation of the Cache Memory Simulator, showing main inputs, outputs, and system boundary.

## 6.4 Class Implementation Details

## 6.5 Cache Mapping Implementation

Cache mapping defines how memory blocks are assigned to cache lines and how many candidate lines are examined on each access. In the simulator, all mapping strategies are implemented using a unified **k-way set-associative model**, parameterized by the associativity value selected by the user.

Instead of maintaining separate implementations for each mapping type, the simulator relies on a single generalized mechanism that covers:

- direct-mapped caches ( $k = 1$ ),
- set-associative caches ( $k > 1$ ),
- fully associative caches ( $k = \text{num\_blocks}$ ).

This design mirrors real hardware implementations and avoids code duplication while preserving conceptual clarity.

### 6.5.1 General k-Way Set-Associative Model

The cache is organized as a collection of sets, each containing  $k$  cache lines (ways), where  $k$  is the associativity. A memory block maps to exactly one set but may be stored in any of the ways belonging to that set.

The number of sets is computed as:

$$\text{num\_sets} = \frac{\text{num\_blocks}}{\text{associativity}}$$

Each memory address is decomposed into block-level components:

$$\text{block\_address} = \left\lfloor \frac{\text{address}}{\text{block size}} \right\rfloor$$

$$\text{set\_index} = \text{block\_address} \bmod \text{num\_sets}, \quad \text{tag} = \left\lfloor \frac{\text{block\_address}}{\text{num\_sets}} \right\rfloor$$

This decomposition is used uniformly for all cache configurations, ensuring consistent behavior across different associativity values.

### 6.5.2 Wrapper-Based Mapping Configuration

Mapping-specific configuration is handled by wrapper classes implemented in `k_associative_cache.py`. Each wrapper is responsible for:

- validating the selected associativity,
- computing the number of sets,
- initializing the core `Cache` object with the correct parameters,
- performing address decomposition for visualization purposes.

To guarantee correct mapping, the wrapper enforces divisibility between the total number of blocks and the associativity:

**Listing 4:** k-way block alignment

```
if num_blocks % associativity != 0:
    num_blocks += associativity - (num_blocks % associativity)
```

This adjustment ensures that all sets contain an equal number of ways.

### 6.5.3 Direct-Mapped Configuration ( $k = 1$ )

A direct-mapped cache is represented as a special case of the k-way model with associativity equal to one. In this configuration, each set contains exactly one cache line, and each memory block maps to a unique cache line.

The wrapper initializes the cache as follows:

**Listing 5:** Direct-mapped cache via k-way model

```
associativity = 1
self.cache = Cache(num_blocks=num_blocks,
                   block_size=block_size,
                   associativity=associativity,
                   replacement=replacement,
                   write_policy=write_policy)
```

Tag comparison is performed on a single cache line, and replacement is trivial, as there is no choice between multiple ways.

#### 6.5.4 General Set-Associative Configuration ( $k > 1$ )

For associativity values greater than one, the cache operates in set-associative mode. Each set contains exactly  $k$  ways, and tag comparison is limited to the lines within the selected set.

On a cache access:

- the set index is computed from the block address;
- all valid tags in the corresponding set are compared;
- if no match is found and the set is full, the replacement policy selects one of the  $k$  ways for eviction.

The replacement policy (FIFO, LRU, or Random) is applied strictly within the boundaries of the selected set.

#### 6.5.5 Fully Associative Configuration ( $k = \text{num\_blocks}$ )

A fully associative cache is implemented as the extreme case of the  $k$ -way model, where the associativity equals the total number of blocks. In this configuration, the cache contains a single set, and all cache lines belong to it:

$$\text{num\_sets} = 1$$

As a result, no index field is required, and all valid cache lines are considered during tag comparison. Replacement decisions rely entirely on the selected replacement policy.

This unified  $k$ -way approach allows the simulator to support all common cache organizations using a single, coherent implementation strategy.

### 6.6 Replacement Policy Implementation

Replacement policies determine which cache line is selected for eviction when a cache miss occurs and all lines in the target set are already valid. In the simulator, replacement logic is implemented as a separate, modular component, allowing eviction strategies to be changed without modifying the core cache access logic.

All replacement policies operate at the level of a single cache set and return the index of the cache line (way) that should be replaced.

### 6.6.1 Policy Abstraction

Replacement policies are implemented in the module `replacement_policies.py`. Each policy encapsulates its own internal state and exposes a common interface used by the cache core during eviction.

The cache core invokes the selected policy only when:

- a cache miss occurs, and
- the target set contains no invalid cache lines.

This separation ensures that replacement decisions are fully decoupled from address mapping and write handling.

### 6.6.2 FIFO Policy Implementation

The FIFO (First-In, First-Out) policy evicts the cache line that has been present in the set for the longest time. For each set, the policy maintains an ordered structure representing the insertion order of cache lines.

On eviction, the oldest entry is selected and removed from the queue. When a new block is inserted, its way index is appended to the end of the queue.

**Listing 6:** FIFO replacement policy (simplified)

```

class FIFOPolicy:
    def __init__(self):
        self.queue = []

    def select_victim(self):
        return self.queue.pop(0)

    def on_insert(self, way):
        self.queue.append(way)
  
```

This policy has minimal bookkeeping overhead and deterministic behavior, making it suitable for controlled experiments.

### 6.6.3 LRU Policy Implementation

The LRU (Least Recently Used) policy evicts the cache line that has not been accessed for the longest period of time. For each set, the policy maintains an ordered list of way indices based on access recency.

On every cache hit or insertion, the accessed way is moved to the end of the list,

marking it as most recently used. The least recently used entry is always located at the front of the list.

**Listing 7:** LRU replacement policy (simplified)

```

class LRUPolicy:
    def __init__(self):
        self.order = []

    def on_access(self, way):
        if way in self.order:
            self.order.remove(way)
        self.order.append(way)

    def select_victim(self):
        return self.order.pop(0)

```

This implementation directly reflects the conceptual definition of LRU and provides predictable eviction behavior for workloads with temporal locality.

#### 6.6.4 Random Policy Implementation

The Random replacement policy selects a cache line uniformly at random from the available ways in the target set. This policy maintains no access history and does not track insertion order.

On eviction, a random way index is generated and returned.

**Listing 8:** Random replacement policy (simplified)

```

import random

class RandomPolicy:
    def select_victim(self, num_ways):
        return random.randint(0, num_ways - 1)

```

Although Random replacement generally performs worse than LRU, it has negligible bookkeeping overhead and serves as a useful baseline for comparison.

#### 6.6.5 Integration with the Cache Core

During a cache miss, if the selected set is full, the cache core invokes the configured replacement policy to determine the victim cache line. If the evicted line is marked as dirty and the write-back policy is enabled, the cache triggers a write-back to main memory before replacing the line.

By isolating replacement logic into dedicated policy classes, the simulator achieves a clean and extensible design, allowing additional policies to be added with minimal changes to the overall system.

## 6.7 Simulation Engine and Execution Flow

The simulator includes a lightweight execution engine that mediates between the graphical user interface and the cache core. Its primary role is to translate user-provided configuration parameters (mapping, associativity, policies, and address sequences) into a concrete execution trace processed by the `CacheSimulator`.

The execution flow is deliberately split into two layers in order to separate configuration logic from cache execution logic.

The simulation engine is composed of the following core components:

- **Simulation wrapper** – responsible for reading UI input, configuring the cache (including associativity and policies), and generating or parsing memory access sequences;
- **CacheSimulator** – executes the actual cache accesses step by step and produces detailed results such as hit/miss information, evictions, and statistics.

### 6.7.1 Cache Construction and Mapping Selection

Cache construction is driven entirely by the associativity value selected by the user. Instead of relying on separate classes for each mapping type, the simulator uses a unified **k-way set-associative model**, where the mapping behavior emerges from the chosen associativity.

The simulation wrapper reads the associativity parameter from the user interface and initializes the cache accordingly:

**Listing 9:** Cache construction from UI parameters

```

def _create_cache_from_ui(self):
    assoc = int(self.ui.associativity.get())

    self.cache = Cache(
        num_blocks=self.ui.cache_size.get(),
        block_size=self.ui.block_size.get(),
        associativity=assoc,
        replacement=self.ui.replacement_policy.get(),
        write_policy=self.ui.write_policy.get(),
        write_miss_policy=self.ui.write_miss_policy.get()
  
```

)

This approach naturally supports:

- direct-mapped caches (`associativity = 1`),
- general set-associative caches (`associativity > 1`),
- fully associative caches (`associativity = num_blocks`).

All address decomposition (tag, set index, offset) is handled internally by the cache core, ensuring that execution logic remains independent of the specific mapping configuration.

### 6.7.2 Address Sequence Preparation

Before execution begins, the simulation wrapper prepares a concrete address sequence. This sequence may originate from:

- a predefined, hardcoded scenario (e.g., matrix traversal or random access), or
- a manually specified list of addresses entered by the user.

All addresses are normalized into a uniform representation before being passed to the simulator, ensuring compatibility across all cache configurations.

### 6.7.3 Execution Loop

The `CacheSimulator` executes the prepared address sequence in a strictly sequential manner. Each simulation step processes exactly one memory access:

**Listing 10:** Simulation execution step

```

address, is_write = self.sequence[self.index]

hit, set_idx, way_idx, evicted, mem_read, mem_write = \
    self.cache.access(address, is_write)

self.stats.record_access(hit, mem_read, mem_write)
self.index += 1
  
```

For each access, the simulator:

- determines hit or miss status,

- applies replacement and write policies when necessary,
- records memory reads and writes,
- updates global statistics.

The structured result of each step is returned to the user interface, which uses it to update the cache visualization and performance metrics in real time.

#### 6.7.4 Step-by-Step and Full Execution

The execution engine supports both incremental and full execution modes. Step-by-step execution is used for animation and educational inspection, while full execution allows the entire access sequence to be processed without intermediate visualization.

This dual execution model enables both interactive exploration and batch-style experimentation under identical cache configurations.

#### 6.7.5 Sequence Generation

The simulation engine supports two distinct sources of memory access sequences, allowing both controlled experimentation and manual testing.

##### Input Categories

- **Explicit address list:** a comma-separated list of memory addresses entered directly by the user through the graphical interface.
- **Scenario-based sequence:** predefined, hardcoded access traces that model specific memory access patterns, such as matrix traversal or random access.

##### Address Normalization

All input addresses are normalized before execution to ensure a uniform internal representation, independent of the format used by the user. The simulation engine converts each token into a hexadecimal address string that can be processed consistently by the cache core.

**Listing 11:** Address normalization routine

```

def _addr_to_hex_str(s):
    if s.startswith('0x'):
        return s[2:]
    if any(c in 'abcdefABCDEF' for c in s):
        return s
    return format(int(s, 10), 'x')

```

This normalization step allows decimal and hexadecimal inputs to be mixed freely without affecting correctness.

### Scenario-Based Sequence Generation

Scenario-based access patterns are generated explicitly by the simulation engine. Each scenario produces a fixed sequence of addresses before execution begins, ensuring deterministic and reproducible behavior.

**Listing 12:** Matrix traversal scenario generation

```

if name == 'Matrix Traversal':
    N = 8
    seq = [format(i * N + j, 'x')
            for i in range(N)
            for j in range(N)]
    return seq
  
```

The generated sequence is passed unchanged to the execution engine and processed step by step by the **CacheSimulator**. By hardcoding scenario generation and normalizing all addresses, the simulator guarantees compatibility across all cache configurations and enables direct comparison between different policies and associativity settings.

#### 6.7.6 Execution Flow

The execution flow defines how the prepared address sequence is processed by the simulation engine. Execution is coordinated by the **CacheSimulator**, which iterates over the address trace and applies cache operations sequentially.

Before execution begins, the simulation engine loads the normalized address sequence into the simulator. Each simulation step processes exactly one memory access, ensuring a clear and traceable execution model.

### Step-Based Execution

The core execution logic is implemented as a loop over the prepared sequence. For each entry, the simulator determines whether the operation represents a read or a write and forwards the request to the cache core:

**Listing 13:** Sequential execution of memory accesses

```

for address, is_write in self.sequence:
    hit, set_idx, way_idx, evicted, mem_read, mem_write = \
        self.cache.access(address, is_write)
    self.stats.record_access(hit, mem_read, mem_write)
  
```

Each invocation of `cache.access()` performs address mapping, hit/miss detection, replacement (if necessary), and write handling. The simulator then updates global statistics based on the returned information.

### Returned Execution Information

For every access, the simulator produces a structured result containing:

- the accessed memory address;
- hit or miss status;
- set index and way index (when applicable);
- information about evicted cache lines;
- main memory read and write activity.

This structured information is returned to the graphical user interface, which uses it to update the cache visualization, highlight hits and misses, and refresh performance metrics in real time.

By executing accesses sequentially and exposing detailed execution data, the simulator enables both interactive, step-by-step inspection and full batch execution under identical cache configurations.

#### 6.7.7 Core Simulation Logic

The core simulation logic is implemented by the `CacheSimulator` class, which acts as the central coordinator of execution. While wrapper classes are responsible for cache configuration and address decomposition, the `CacheSimulator` manages the execution order of memory accesses and maintains all global performance statistics.

Its design follows a strict step-based model, where each invocation corresponds to exactly one memory access, allowing precise inspection of cache behavior.

### Sequence Initialization

Before execution begins, the simulator loads a prepared sequence of memory accesses. Each entry consists of a memory address and a flag indicating whether the operation is a read or a write.

**Listing 14:** Loading the execution sequence

```
def load_sequence(self, addresses, writes=None):
    self.sequence = list(zip(addresses, writes))
    self.index = 0
```

This structure ensures that memory operations are processed in a deterministic order and enables both step-by-step and full execution modes.

### Step Execution

Each simulation step processes exactly one memory access. The simulator retrieves the next operation from the sequence and forwards it to the cache core for evaluation.

**Listing 15:** Single simulation step

```

address, is_write = self.sequence[self.index]

hit, set_index, way_index, evicted, mem_read, mem_write = \
    self.cache.access(address,
                       is_write,
                       self.write_miss_policy)

self.stats.record_access(hit, mem_read, mem_write)
self.index += 1

```

The `cache.access()` method encapsulates all low-level cache operations, including:

- address mapping to block address, set index, and tag;
- hit or miss detection within the selected set;
- invocation of the replacement policy when eviction is required;
- handling of write-back or write-through semantics;
- interaction with the RAM model through memory reads and writes.

The simulator itself remains agnostic to these internal details and operates solely on the structured result returned by the cache core.

### Statistics Management

After each access, the simulator updates global performance statistics. These statistics include the total number of accesses, cache hits and misses, hit rate, and the number of main memory read and write operations.

By centralizing statistics collection in the simulator rather than in the cache core, the design maintains a clear separation between functional behavior and performance accounting.

## Structured Step Result

Each execution step returns a structured dictionary that fully describes the outcome of the access:

**Listing 16:** Structured result returned by a simulation step

```

return {
    'address': address,
    'hit': hit,
    'set_index': set_index,
    'way_index': way_index,
    'stats': {
        'accesses': self.stats.accesses,
        'hits': self.stats.hits,
        'misses': self.stats.misses,
        'hit_rate': self.stats.hit_rate,
        'memory_reads': self.stats.memory_reads,
        'memory_writes': self.stats.memory_writes,
    },
    'evicted': evicted,
}

```

This structured output is consumed directly by the graphical interface, which uses it to update the cache visualization, highlight hit and miss events, display eviction information, and refresh performance metrics in real time.

## Design Rationale

By isolating execution control and statistics management inside the `CacheSimulator`, the simulation loop remains compact and readable. This design choice allows students to clearly trace how a memory address flows from user input, through cache mapping and replacement logic, and finally into main memory interactions, without obscuring the underlying mechanisms.

## 6.8 Graphical User Interface Implementation

The simulator includes a fully interactive graphical user interface implemented using the `Tkinter` library. The interface represents the control and visualization layer of the system and provides a clear separation between user interaction and the internal cache simulation logic.

Its primary purpose is to allow users to configure cache parameters, control simu-

lation execution, and observe cache behavior and performance metrics in real time. All computational logic is delegated to the simulation engine and cache core, while the UI focuses exclusively on interaction and visualization.

The interface is implemented as a single coordinating class, `UserInterface`, defined in `user_interface.py`. This class is responsible for constructing, managing, and updating all graphical components of the application.

### 6.8.1 UI Architecture and Responsibilities

The `UserInterface` class encapsulates all user-facing functionality and communicates with the simulation engine exclusively through well-defined method calls. It does not perform cache access, replacement, or memory operations internally.

The responsibilities of the UI include:

- collecting cache configuration parameters (cache size, block size, associativity, replacement policy, write policy, and write-miss policy);
- allowing the user to select predefined simulation scenarios or enter custom memory address sequences;
- instantiating and controlling the simulation engine;
- visualizing cache contents and address decomposition;
- displaying performance statistics and execution feedback.

This strict separation of concerns ensures that the graphical layer remains lightweight and that the functional cache model remains independently testable.

### 6.8.2 Event-Driven Execution Model

The interface follows an event-driven architecture, where user actions (button presses, parameter changes) trigger corresponding callback functions. These callbacks invoke simulation engine methods without blocking the graphical interface.

Simulation animation is implemented using Tkinter's event loop (`after()`), allowing continuous execution while keeping the UI responsive.

### 6.8.3 Interface Layout

The main window is divided into two primary regions:

- a **control panel** containing configuration widgets, scenario selection, execution controls, and textual logs;

- a **visualization panel** displaying the cache grid, address decomposition, and live performance statistics.

The application window is initialized programmatically:

**Listing 17:** Main UI window initialization

```

self.window = tk.Tk()
self.window.title("Cache Memory Simulator")
self.window.geometry("1080x1000")
self.window.configure(bg="#23967F")

```

This layout enables immediate visual feedback when parameters are modified.

#### 6.8.4 Configuration Widgets and State Binding

Cache parameters are exposed through interactive widgets bound to Tkinter variable objects. These variables provide automatic synchronization between the UI state and the simulation engine.

**Listing 18:** UI-bound configuration variables

```

self.cache_size = tk.IntVar(value=16)
self.block_size = tk.IntVar(value=2)
self.associativity = tk.IntVar(value=1)
self.address_width = tk.IntVar(value=6)
self.replacement_policy = tk.StringVar(value="LRU")
self.write_policy = tk.StringVar(value="write-back")
self.write_miss_policy = tk.StringVar(value="write-allocate")
self.scenario_var = tk.StringVar(value="Matrix Traversal")

```

When a simulation is started, the current values of these variables are read and forwarded to the simulation engine for cache construction.

#### 6.8.5 Cache Construction from UI Parameters

Unlike earlier designs that relied on separate wrapper classes for each mapping type, the current interface uses a unified **k-way set-associative model**. The selected associativity value directly determines the cache organization.

When the user starts a simulation, the UI constructs the cache through the simulation engine as follows:

**Listing 19:** Cache construction initiated by the UI

```

self.simulation = Simulation(self)
self.simulation.create_cache_from_ui()

```

This approach naturally supports:

- direct-mapped caches (`associativity = 1`);
- general set-associative caches (`associativity > 1`);
- fully associative caches (`associativity = num_blocks`).

### 6.8.6 Address Decode Visualization

One of the most educational components of the interface is the graphical address decoder. For each accessed address, the binary representation is split into *tag*, *set index*, and *offset* fields based on the current cache configuration.

The decoder renders each bit group using colored rectangles:

**Listing 20:** Address decode rendering

```

canvas.create_rectangle(x, y, x + box_w, y + box_h,
                        fill=color, outline="#222222")
canvas.create_text(x + box_w / 2, y + box_h / 2,
                  text=bit, font=(self.font_container, 10))

```

Color coding is used consistently:

- blue for tag bits;
- green for set index bits;
- orange for offset bits.

This visualization allows users to directly correlate address bits with cache mapping behavior.

### 6.8.7 Cache Grid Rendering

The cache contents are visualized as a grid of rectangular cells, where each cell corresponds to a cache line (or way within a set). The grid dimensions are dynamically derived from the number of blocks and associativity.

**Listing 21:** Cache grid cell creation

```

lbl = tk.Label(self.cache_display_frame,
               text=f"{i}",
               bg="#111111",
               fg="white")

```

After each simulation step, the grid is updated to reflect cache behavior:

- neutral or dark colors indicate valid but unaccessed lines;
- green highlights cache hits;
- red highlights cache misses.

**Listing 22:** Cache grid update on access

```
lbl.configure(bg="#8BC34A" if info.get("hit")
              else "#F44336")
```

### 6.8.8 Simulation Control and Animation

The interface supports step-by-step execution as well as continuous animation. Animation is implemented using Tkinter's scheduling mechanism:

**Listing 23:** Non-blocking animation scheduling

```
self._after_id = self.window.after(delay, self._animation_step)
```

Each animation step performs the following actions:

1. executes one simulation step via the `CacheSimulator`;
2. updates hit/miss statistics;
3. recolors the cache grid;
4. redraws the address decoder;
5. updates performance metrics displayed in the UI.

### 6.8.9 Statistics Display and Logging

The interface continuously displays global performance metrics, including:

- total number of accesses;
- number of cache hits and misses;
- hit rate;
- number of main memory reads and writes.

**Listing 24:** Live statistics update

```

self.stat_hit_rate.configure(
    text=f"{stats['hit_rate']:.3f}"
)
  
```

In addition, a textual log panel records significant events such as cache misses and eviction decisions, providing transparency into the internal behavior of the cache.

### 6.8.10 Integration with the Core Simulator

The UI does not implement any cache logic internally. Instead, it delegates all execution to the core simulator:

**Listing 25:** UI delegation to simulator

```

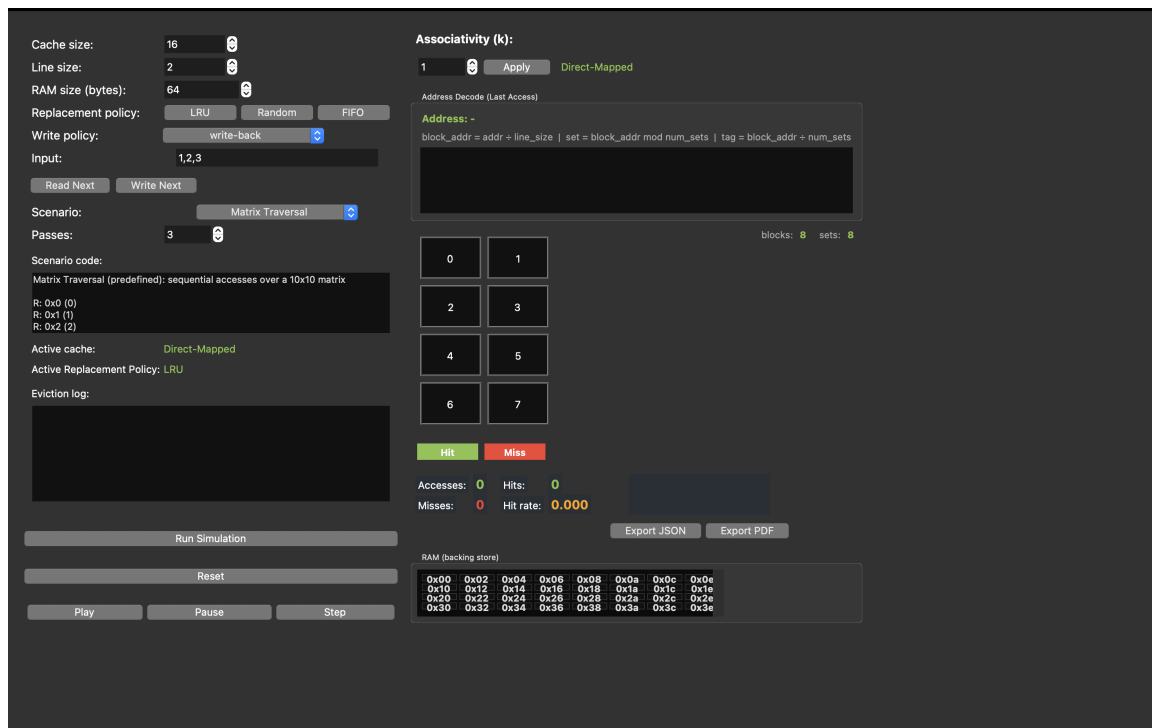
sim = self.simulation.simulator
sim.load_sequence(addresses)
self._running_sim = sim
  
```

This delegation ensures that visualization remains decoupled from cache computation and that the simulator can be tested independently of the UI.

## 6.9 Application Workflow and User Interface Walkthrough

### 6.9.1 Initial Application State and Cache Configuration

At startup, the cache simulator presents the user with a fully initialized graphical interface that exposes all configuration parameters required to define a cache memory system. Figure 19 illustrates the initial state of the application before any memory access is executed.



**Figure 19:** Initial state of the cache simulator interface, showing cache configuration controls and empty simulation state

The left side of the interface acts as a configuration and control panel. Here, the user specifies the structural parameters of the cache:

- **Cache size**, expressed as the total number of blocks;
- **Line size**, defining the number of bytes per cache block;
- **RAM size**, representing the size of the backing memory;
- **Associativity ( $k$ )**, which determines the cache mapping strategy;
- **Replacement policy** (LRU, FIFO, or Random);
- **Write policy** (write-back or write-through).

All configuration parameters are selected prior to simulation execution. Once applied, they determine the number of cache blocks, the number of sets, and the internal tag–index–offset decomposition used by the simulator.

In addition to cache parameters, the interface allows the user to define the memory access sequence. Two input mechanisms are supported:

- **Manual input**, where the user provides a comma-separated list of memory addresses;

- **Scenario-based input**, where predefined access patterns (such as matrix traversal or random access) automatically generate address sequences.

Before execution begins, the cache grid, statistics panel, eviction log, and RAM visualization are empty. This clearly separates configuration from execution and ensures that all observed behavior results exclusively from user-selected parameters and access patterns.

By explicitly requiring configuration to be completed before simulation, the interface reinforces the conceptual relationship between cache design choices and their runtime effects.

### 6.9.2 Input Validation and Error Handling

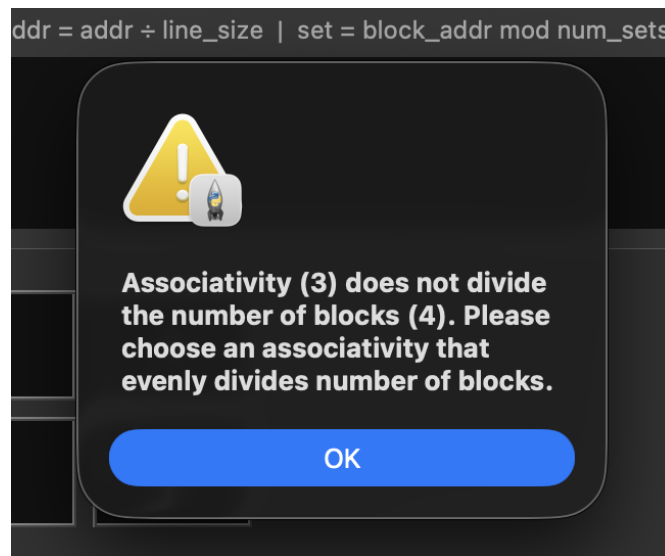
To ensure correctness and prevent undefined behavior, the simulator performs strict input validation at the graphical interface level. All cache parameters provided by the user are verified before any cache structure is instantiated or any simulation step is executed.

Validation is performed eagerly: whenever the user attempts to apply a cache configuration, the interface checks structural constraints and blocks execution if inconsistencies are detected.

### 6.9.3 Cache Size and Line Size Consistency

One fundamental constraint enforced by the simulator is that the cache size must be an exact multiple of the cache line size. This requirement ensures that the cache can be evenly divided into an integer number of blocks.

If incompatible values are selected, the simulator displays a modal error dialog and prevents the configuration from being applied, as shown in Figure 20.



**Figure 20:** Validation error when cache size is not a multiple of line size

This validation step guarantees that all subsequent computations involving block count, set count, and address decomposition remain well-defined.

#### 6.9.4 Associativity and Block Divisibility

For set-associative caches, an additional constraint must be satisfied: the associativity ( $k$ ) must evenly divide the total number of cache blocks. If this condition is violated, the cache cannot be partitioned into an integer number of sets.

When an invalid associativity value is selected, the simulator rejects the configuration and presents a descriptive error message to the user, illustrated in Figure 21.



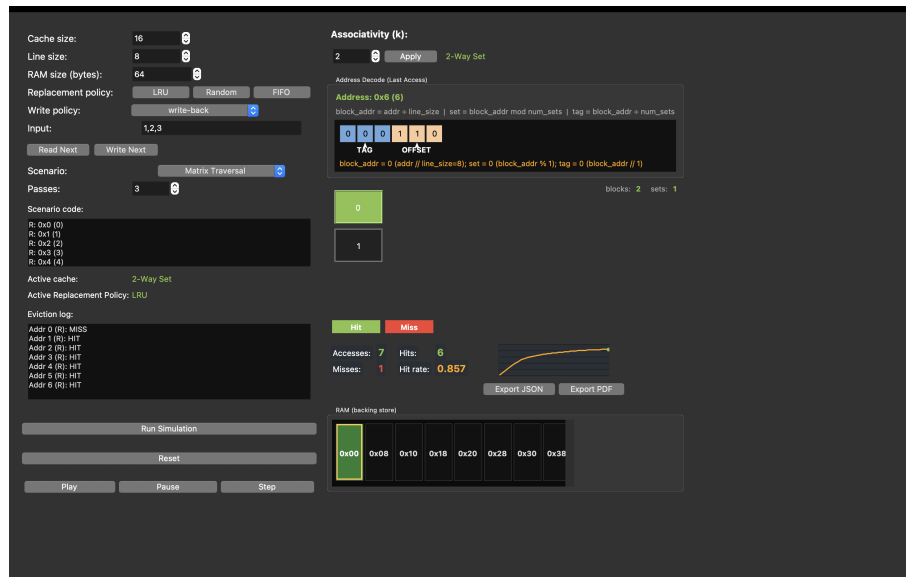
**Figure 21:** Validation error when associativity does not divide the number of cache blocks

By enforcing these constraints directly in the user interface, the simulator ensures that only valid cache configurations reach the core simulation logic. This design choice simplifies the implementation of the cache core and improves robustness by eliminating invalid states early in the execution flow.

### 6.9.5 Cache Hit Visualization

When a memory access targets a block that is already present in the cache and marked as valid, the simulator identifies the access as a **cache hit**. In this case, no data transfer from main memory is required.

Figure 22 illustrates a cache hit during simulation execution. The corresponding cache line is highlighted in green, indicating that the requested block was found in the cache.



**Figure 22:** Visualization of a cache hit: the requested block is found in the cache

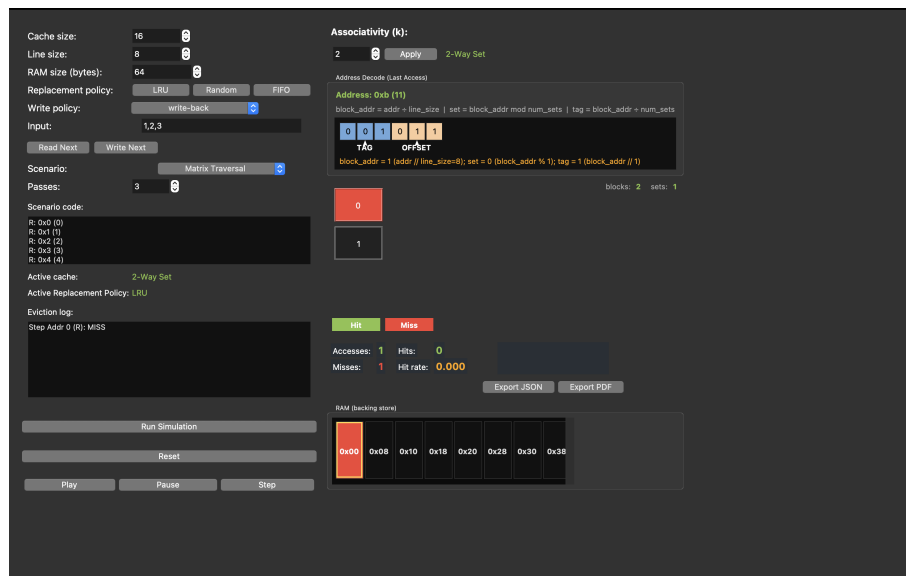
During a hit, the simulator updates only the replacement metadata (for example, recency information in the case of LRU), while the cache contents and main memory remain unchanged. This behavior reflects the low-latency access path of cache hits in real systems.

### 6.9.6 Cache Miss Visualization

If the requested memory block is not present in the cache, a **cache miss** occurs. In this situation, the simulator must fetch the block from main memory and insert it into the appropriate cache set.

Figure 23 shows a cache miss event. The accessed cache location is highlighted in red, indicating that the requested block was not found and that a memory fetch is

required.



**Figure 23:** Visualization of a cache miss triggering a block fetch from main memory

If the target set is already full, the simulator applies the selected replacement policy to evict an existing block before loading the new one. The miss event therefore represents the primary mechanism through which cache state evolves during execution.

### 6.9.7 Write Miss with Dirty Block (Write-Back Policy)

Under the **write-back** policy, write operations modify only the cache and defer updates to main memory until eviction. When a write miss occurs and an existing dirty block must be evicted, the simulator performs a write-back operation.

Figure 24 illustrates this scenario. The evicted cache block is marked as dirty, and a write-back to RAM is triggered before the new block is loaded.

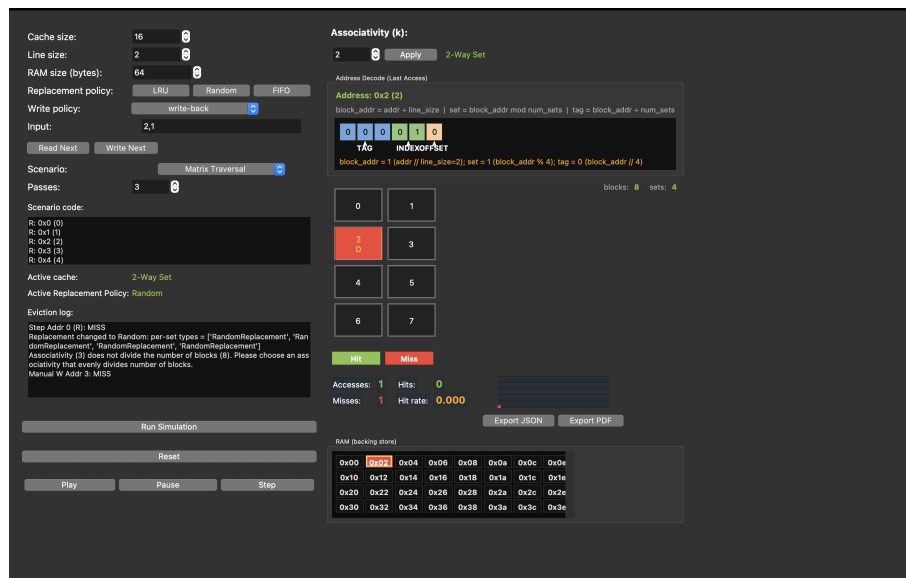


Figure 24: Write miss under write-back policy with eviction of a dirty block

This visualization highlights the delayed memory update characteristic of write-back caches and emphasizes the importance of the dirty bit in ensuring memory consistency.

### 6.9.8 Write Miss without Dirty Block

In contrast, when the evicted block is not marked as dirty, no write-back to main memory is required. This situation may occur either when the block has never been modified or when using a **write-through** policy.

Figure 25 shows a write miss where the evicted block is clean. The simulator replaces the cache line directly, without triggering a memory write for the evicted block.

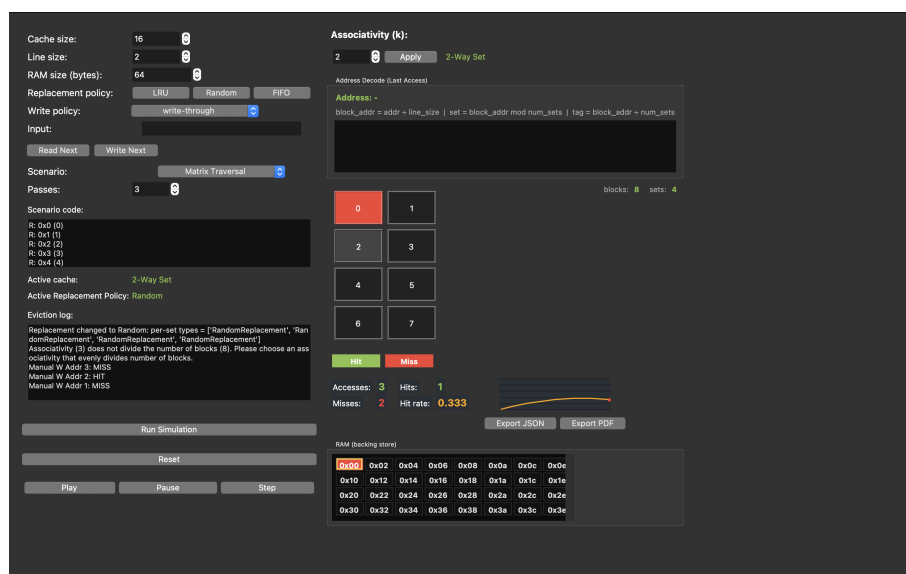


Figure 25: Write miss where the evicted block is clean and no write-back is required

By visually distinguishing dirty and non-dirty evictions, the simulator makes explicit the differences between write policies and their impact on memory traffic.

### 6.9.9 Address Decoding and Tag-Index-Offset Visualization

For each memory access, the simulator explicitly visualizes the address decoding process used internally by the cache. This step bridges the gap between abstract address calculations and their concrete effect on cache behavior.

Figure 26 illustrates the graphical address decoding panel displayed during simulation execution.



**Figure 26:** Graphical visualization of address decoding into Tag, Set Index, and Offset fields

The simulator converts each accessed memory address into its binary representation and decomposes it into three distinct fields:

- **Offset**, which selects the byte within a cache line;
- **Set index**, which determines the cache set to be accessed;
- **Tag**, which uniquely identifies the memory block within the selected set.

The size of each field is computed dynamically based on the active cache configuration:

$$\text{block\_addr} = \left\lfloor \frac{\text{address}}{\text{line\_size}} \right\rfloor, \quad \text{set\_index} = \text{block\_addr} \bmod \text{num\_sets}, \quad \text{tag} = \left\lfloor \frac{\text{block\_addr}}{\text{num\_sets}} \right\rfloor$$

Each portion of the address is highlighted using a distinct color in the interface, allowing the user to visually trace how individual bits contribute to cache lookup decisions.

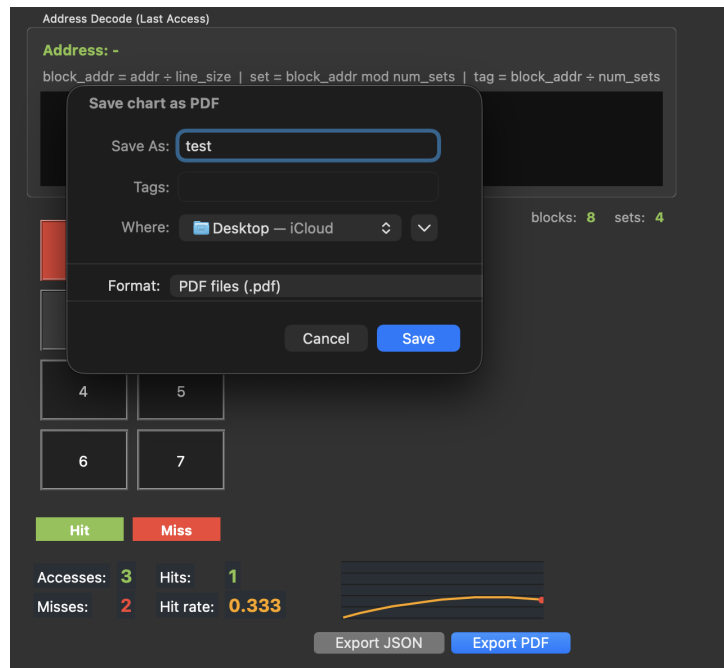
This visualization is updated at every simulation step and reflects the exact same computations performed by the cache core. As a result, users can directly correlate the displayed address fields with subsequent cache hit or miss outcomes.

By making address decomposition explicit, the simulator transforms a typically opaque hardware operation into an intuitive and observable process, significantly improving conceptual understanding of cache mapping and associativity.

### 6.9.10 Result Export and Persistence

To support offline analysis and reporting, the simulator provides a mechanism for exporting simulation results. At any point during or after execution, the user can save the current performance data in a portable format.

Figure 27 illustrates the export interface used to generate a PDF report containing the simulation results.



**Figure 27:** Export dialog for saving simulation results as a PDF document

The export functionality captures the current state of the statistics panel, including numerical performance metrics and the hit rate evolution graph. The resulting PDF file provides a static snapshot of the simulation outcome, which can be used for documentation, comparison between configurations, or submission as part of laboratory assignments.

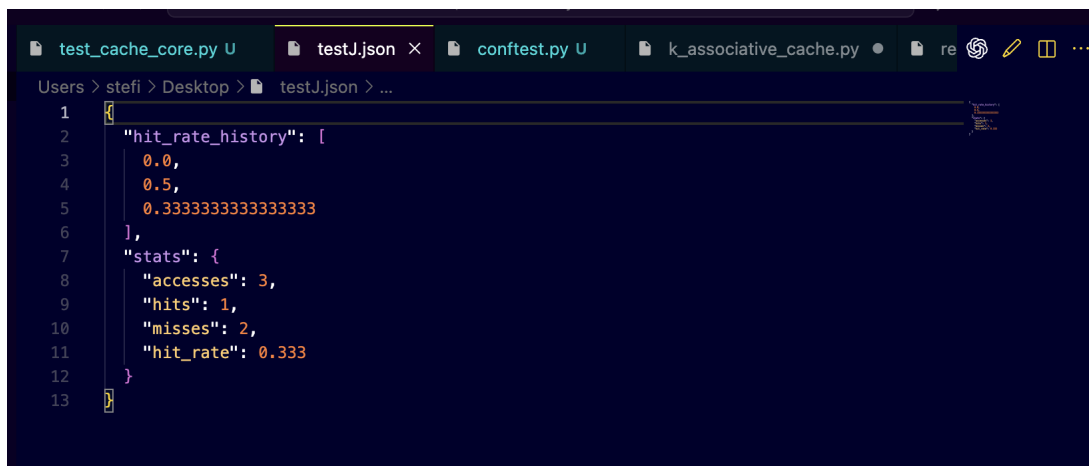
This feature decouples result analysis from the live simulation environment and allows users to preserve results without requiring repeated execution. By supporting standardized output formats, the simulator facilitates reproducibility and external evaluation of cache behavior.

The export mechanism operates independently of the simulation core, ensuring that saving results does not interfere with cache execution or state.

### 6.9.11 Export of Simulation Results in JSON Format

In addition to visual summaries, the simulator allows exporting raw execution data in a structured JSON format. This option is intended for programmatic analysis, reproducibility, and external processing of simulation results.

Figure 28 shows an example of a JSON file generated by the simulator after execution.



```

1  {
2    "hit_rate_history": [
3      0.0,
4      0.5,
5      0.3333333333333333
6    ],
7    "stats": {
8      "accesses": 3,
9      "hits": 1,
10     "misses": 2,
11     "hit_rate": 0.333
12   }
13 }

```

**Figure 28:** Exported JSON file containing hit rate history and aggregated statistics

The exported JSON document includes two main components:

- **Hit rate history**, which records the evolution of the hit rate after each memory access;
- **Aggregated statistics**, including total accesses, hits, misses, and final hit rate.

This structured representation mirrors the internal data maintained by the simulation engine and enables:

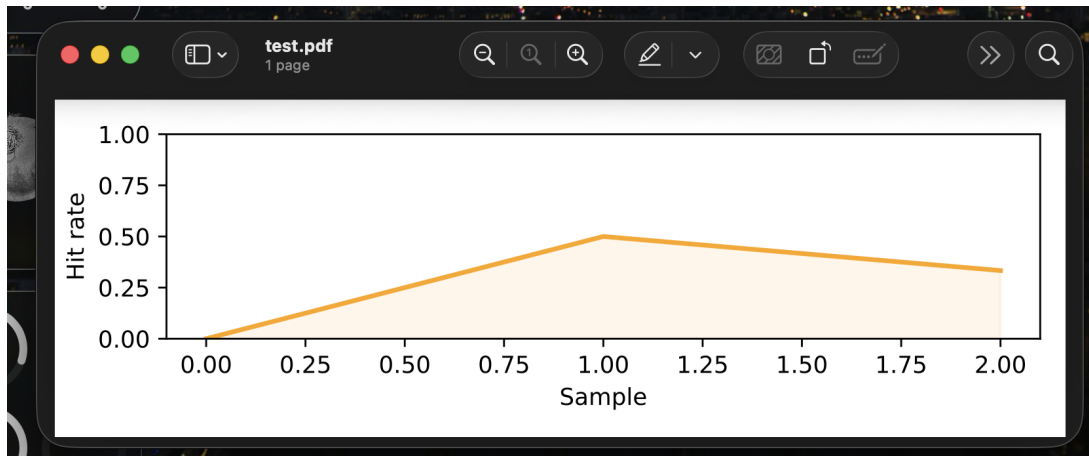
- post-processing using external scripts;
- comparison between multiple simulation runs;
- automated evaluation and testing.

By exporting results in a machine-readable format, the simulator supports reproducibility and facilitates integration with data analysis workflows.

### 6.9.12 Export of Performance Charts as PDF

For documentation and reporting purposes, the simulator also provides the ability to export the hit rate evolution graph as a PDF file. This export captures a visual summary of cache performance over time.

Figure 29 illustrates a PDF document generated from the statistics panel.



**Figure 29:** Exported PDF containing the hit rate evolution during simulation

The exported chart plots the hit rate after each memory access, providing a compact visual representation of cache efficiency. This is particularly useful for identifying warm-up behavior, steady-state performance, and the impact of different access patterns.

The PDF format ensures portability and preserves graphical quality, making it suitable for inclusion in reports, presentations, or laboratory submissions.

Exporting graphical results complements the numerical JSON output by offering an intuitive and easily interpretable summary of simulation behavior.

## 6.10 Data Export and Visualization

In addition to real-time visualization, the simulator provides structured access to performance data generated during execution. This data can be displayed dynamically through the graphical interface or exported for offline analysis and comparison between configurations.

The design separates *data collection*, *visualization*, and *export*, ensuring that performance metrics remain consistent regardless of how they are presented.

### 6.10.1 Statistics Collection

All performance-related data is collected centrally by the `CacheSimulator`. Statistics are updated after each memory access and reflect the global state of the simulation.

The following metrics are tracked:

- total number of memory accesses;
- number of cache hits and cache misses;
- hit rate and miss rate;
- number of main memory read operations;
- number of main memory write operations.

These values are stored in a dedicated statistics structure and updated incrementally during execution, ensuring low overhead and deterministic results.

### 6.10.2 Real-Time Visualization

The graphical user interface consumes the structured output returned by each simulation step and updates the visual components accordingly.

After every access, the UI:

- refreshes the cache grid to highlight hits and misses;
- updates the address decoder visualization;
- redraws numerical performance indicators (hits, misses, hit rate);
- appends textual messages to the execution log when evictions occur.

This step-by-step visualization allows users to directly correlate individual memory accesses with observed cache behavior and performance changes.

### 6.10.3 Aggregated Performance Display

In addition to per-step updates, the interface displays aggregated statistics that summarize the entire execution up to the current point. These values provide a high-level view of cache efficiency and allow users to compare different configurations or scenarios during repeated experiments.

Because the statistics are updated incrementally, the same execution can be paused, resumed, or replayed without losing consistency.

#### 6.10.4 Data Export

For offline analysis, the simulator supports exporting collected statistics after execution. Exported data can be used to compare cache configurations, replacement policies, or access patterns across multiple simulation runs.

The exported dataset typically includes:

- cache configuration parameters used during execution;
- total accesses, hits, and misses;
- hit and miss rates;
- memory read and write counts.

This separation between simulation and export logic ensures that data representation does not interfere with execution correctness.

#### 6.10.5 Design Rationale

By integrating both real-time visualization and structured data export, the simulator supports two complementary usage modes:

- interactive exploration, focused on understanding cache behavior at the level of individual accesses;
- analytical evaluation, focused on comparing performance metrics across different cache configurations.

This dual approach reinforces the educational purpose of the simulator while maintaining a clean and extensible architecture.

### 6.11 Error Handling and Input Validation

Robust error handling and input validation are essential to ensure correct simulation behavior and prevent invalid configurations from producing undefined results. The simulator performs validation at multiple levels, combining user interface checks with internal consistency guarantees enforced by the simulation engine and cache core.

The overall design goal is to fail early for invalid input while maintaining a responsive and informative user experience.

### 6.11.1 User Interface Input Validation

Basic validation is performed at the graphical user interface level to prevent the most common configuration errors. Input widgets restrict values to meaningful domains using `Tkinter` variable bindings and controlled selection elements (e.g., dropdown menus for policies).

The following constraints are enforced or assumed at the UI level:

- cache size and block size must be positive integers;
- associativity must be a positive integer;
- replacement, write, and write-miss policies must be selected from predefined options;
- scenario selection is restricted to predefined, supported access patterns.

By limiting user choices through the interface, many invalid states are prevented before the simulation engine is invoked.

### 6.11.2 Cache Configuration Validation

Additional validation is performed when constructing the cache from UI parameters. The simulation engine verifies that the selected configuration is internally consistent before execution begins.

In particular, the simulator ensures that:

- the number of cache blocks is divisible by the associativity value, guaranteeing an equal number of ways per set;
- block size is greater than zero and compatible with address decomposition;
- associativity does not exceed the total number of cache blocks.

When necessary, the cache wrapper aligns the number of blocks to satisfy set-associative constraints, ensuring predictable mapping behavior.

### 6.11.3 Address Validation and Normalization

All memory addresses are normalized before execution to ensure consistent internal representation. The simulation engine accepts both decimal and hexadecimal address formats and converts them into a uniform hexadecimal form.

Invalid or malformed address tokens are implicitly filtered by the normalization process, preventing runtime failures during execution.

This normalization step guarantees that address parsing errors do not propagate into the core simulation logic.

#### 6.11.4 Runtime Safety in Simulation Execution

During simulation, execution proceeds in a strictly controlled, step-based manner. Each step processes exactly one memory access, and internal indices are checked to prevent out-of-bounds access to the execution sequence.

The cache core enforces additional safety guarantees:

- cache set indices are computed modulo the number of sets;
- way indices are restricted to valid ranges within each set;
- memory read and write operations are issued only through the RAM abstraction.

These guarantees ensure that invalid memory access patterns cannot corrupt the simulator state.

#### 6.11.5 RAM Boundary Checks

The RAM model enforces strict bounds checking on all memory accesses. Any attempt to read or write outside the allocated memory range is detected and handled explicitly.

This prevents silent data corruption and reinforces the conceptual separation between cache behavior and main memory behavior.

#### 6.11.6 Error Reporting and User Feedback

Rather than terminating abruptly, the simulator reports configuration errors or invalid states through the graphical interface. Error conditions are communicated using clear visual feedback or log messages, allowing users to correct input parameters without restarting the application.

This approach maintains usability while ensuring correctness.

#### 6.11.7 Design Rationale

Error handling and validation logic are deliberately distributed across the system:

- the UI prevents invalid input whenever possible;
- the simulation engine enforces configuration consistency;

- the cache and RAM cores guarantee runtime safety.

This layered validation strategy improves reliability, simplifies debugging, and ensures that the simulator behaves predictably even when faced with incorrect or unexpected input.

## 6.12 Performance Considerations and Optimizations

Although the simulator is designed primarily for educational purposes rather than cycle-accurate performance evaluation, several design choices were made to ensure efficient execution and responsive interaction, even for longer memory access sequences.

The performance considerations focus on balancing clarity, correctness, and interactivity, rather than maximizing raw execution speed.

### 6.12.1 Step-Based Execution Model

The simulation engine operates using a step-based execution model, where each memory access is processed sequentially. This design enables precise visualization and detailed inspection of cache behavior, but also introduces overhead compared to batch execution.

To mitigate this overhead, the simulator minimizes per-step computation by delegating all low-level cache logic to the `Cache.access()` method and centralizing statistics updates in a lightweight structure.

### 6.12.2 Efficient Data Structures

The cache core relies primarily on Python's built-in data structures (lists and dictionaries) for representing cache sets, cache lines, and replacement metadata. These structures provide constant-time indexing and acceptable performance for the cache sizes used in educational scenarios.

Replacement policies are implemented using simple list-based bookkeeping, which keeps eviction decisions fast and predictable while remaining easy to understand.

### 6.12.3 Limited Cache and Memory Sizes

To preserve responsiveness of the graphical interface, the simulator intentionally limits cache and memory sizes to modest values. This prevents excessive rendering overhead when visualizing cache grids and address decoding.

By constraining problem size, the simulator maintains interactive frame rates even when executing step-by-step animations.

#### 6.12.4 Non-Blocking User Interface Updates

The graphical interface uses Tkinter's event scheduling mechanism (`after()`) to perform simulation steps without blocking the main event loop. This ensures that the interface remains responsive during continuous execution and that users can pause or stop the simulation at any time.

UI updates are limited to the components affected by each step, avoiding unnecessary redraws of static elements.

#### 6.12.5 Avoidance of Redundant Computation

Several computations, such as cache structure initialization, address bit-width calculations, and set layout derivation, are performed once at simulation start rather than during each access.

This reduces repeated overhead inside the main execution loop and simplifies the runtime behavior of the simulator.

#### 6.12.6 Trade-Offs and Limitations

Certain design trade-offs were made in favor of simplicity and educational clarity:

- replacement metadata is updated eagerly rather than lazily;
- address decoding is visualized at each step, even though this is not required for correctness;
- the simulator does not model timing effects such as latency, pipelining, or parallel memory accesses.

These choices ensure that the simulator remains transparent and easy to understand, at the cost of reduced scalability and timing realism.

#### 6.12.7 Potential Optimizations

Although not required for the current scope, several optimizations could be applied in future extensions:

- batching multiple simulation steps during continuous execution;
- decoupling visualization frequency from execution frequency;
- introducing optional lightweight profiling for replacement policy behavior;
- optimizing cache grid rendering for larger configurations.

### 6.12.8 Design Rationale

Overall, the simulator prioritizes correctness, determinism, and educational value over raw performance. The implemented optimizations are sufficient to support interactive exploration of cache behavior while keeping the system architecture clean, modular, and maintainable.

## 7 Testing and Validation

Testing plays a critical role in validating the correctness and robustness of the cache memory simulator. Given the complexity of cache behavior—replacement decisions, write semantics, and interaction with main memory—a comprehensive and systematic test strategy was required.

All tests were implemented as consolidated unit tests using the `pytest` framework and focus exclusively on the cache core, RAM model, and simulation engine. The graphical user interface is intentionally excluded from unit testing, as it does not contain functional cache logic.

### 7.1 Testing Strategy

The test suite follows three guiding principles:

- **Isolation:** tests target only the cache core, RAM, and simulator logic, without involving UI components;
- **Determinism:** wherever possible, tests are deterministic and assert exact expected behavior;
- **Coverage:** all major cache mechanisms and corner cases are exercised.

The tests are intentionally small and fast-running, making them suitable for continuous integration and repeated execution during development.

### 7.2 Replacement Policy Validation

Replacement policies are validated using controlled access sequences that fill a cache set and then trigger an eviction.

The test suite verifies correct behavior for all supported policies: **LRU**, **FIFO**, and **Random**.

#### LRU and FIFO determinism

For LRU and FIFO, the evicted block is deterministic. Tests assert that:

- LRU evicts the least recently used block;
- FIFO evicts the first inserted block.

### Random replacement

For the Random policy, determinism is not expected. Instead, tests assert that the evicted block belongs to the set of valid blocks present in the cache at the time of eviction.

These tests are executed across multiple cache configurations (num\_blocks, associativity, and line size) to ensure policy correctness under varied setups.

## 7.3 Write Policy Testing

The simulator supports both **write-back** and **write-through** policies. Write-hit behavior is tested explicitly.

### Write-back

On a write hit:

- the cache line's dirty bit must be set;
- no immediate memory write should be reported.

### Write-through

On a write hit:

- the dirty bit must remain unset;
- a memory write must be reported immediately.

The test suite inspects both the returned access flags and the internal state of the affected cache line.

## 7.4 Write-Miss Policy Validation

Both write-miss policies are validated: **write-allocate** and **write-no-allocate**.

### Write-allocate

On a write miss:

- a cache line is allocated;
- a memory read may occur;
- a memory write is reported.

## Write-no-allocate

On a write miss:

- no cache line is allocated;
- the write is forwarded directly to memory;
- the cache index for the write remains undefined.

These behaviors are verified independently of the selected write policy, ensuring consistent semantics.

## 7.5 Eviction and Write-Back to RAM

Eviction behavior under write-back is tested explicitly. When a dirty block is evicted:

- the simulator must report a memory write;
- the evicted block must be marked as dirty;
- the write-back must target the base-aligned address of the block.

This behavior is validated both at the cache level and through integration tests involving the `CacheSimulator` and the `RAM` model.

## 7.6 Line Size and Address Alignment Testing

To ensure correct block-based addressing, tests are included for configurations where `line_size > 1`. These tests verify that:

- evictions trigger write-back to the base address of the block;
- address alignment is preserved consistently across accesses.

This confirms that the simulator correctly handles block-granularity memory operations.

## 7.7 Simulation Engine Integration Tests

Integration tests validate the interaction between the cache core, RAM, and simulation engine. These tests load address sequences into the simulator and execute them step by step, asserting:

- correct hit/miss detection;
- correct memory read/write signaling;
- proper propagation of write-back effects into RAM.

This ensures that the simulator behaves consistently as a complete system, not just as isolated components.

## 7.8 Cache Reset Validation

The `reset()` operation is tested to ensure that it:

- clears all valid bits;
- clears all dirty bits;
- restores the cache to a clean initial state.

This is essential for repeated simulations and reproducibility of results.

## 7.9 RAM Boundary and Error Handling Tests

The RAM model enforces strict bounds checking. Tests confirm that:

- reading outside allocated memory raises `IndexError`;
- writing outside allocated memory raises `IndexError`.

This prevents silent memory corruption and ensures predictable behavior.

## 7.10 Parameterized Smoke Tests

A set of parameterized smoke tests exercises a small matrix of valid configurations across:

- cache sizes;
- associativities;
- line sizes.

These tests validate basic read and write semantics and ensure that no unexpected failures occur for supported configurations.

## 7.11 Randomized Stress Testing

Finally, a lightweight randomized stress test executes a short sequence of mixed read and write operations over random addresses. The goal is not formal verification, but rapid detection of unexpected exceptions or invalid states.

## 7.12 Validation Summary

Together, these tests provide strong confidence in the correctness of the cache simulator. They validate functional behavior, edge cases, and component interaction, while remaining fast, deterministic, and suitable for continuous integration.

This testing approach ensures that the simulator behaves predictably and correctly across all supported cache configurations and policies.

## 8 Conclusion

This project presented the design and implementation of an educational cache memory simulator that models the core mechanisms of modern CPU caches in a clear, interactive, and configurable manner. The simulator focuses on correctness and transparency rather than cycle-accurate timing, making it well suited for studying cache behavior and memory access patterns.

The implementation supports multiple cache organizations, including direct-mapped and  $k$ -way set-associative caches, along with configurable replacement policies (LRU, FIFO, Random) and write strategies (write-back and write-through, with write-allocate and write-no-allocate on write misses). By exposing these parameters through an interactive graphical interface, the simulator allows users to directly observe how architectural choices affect cache performance.

A clear separation of concerns was maintained throughout the system. The cache core, RAM model, simulation engine, and graphical interface are implemented as independent components with well-defined responsibilities. This modular architecture improves readability, testability, and maintainability, and allows the functional core of the simulator to be validated independently of the user interface.

The testing strategy provides strong confidence in the correctness of the implementation. Unit and integration tests validate replacement behavior, write semantics, eviction handling, RAM interaction, and error conditions across a wide range of configurations. Deterministic tests ensure predictable behavior, while lightweight randomized tests help detect unexpected corner cases.

From an educational perspective, the simulator succeeds in making abstract cache concepts tangible. Step-by-step execution, visual address decomposition, and real-time statistics enable intuitive exploration of locality, associativity, and replacement effects without requiring specialized hardware or low-level instrumentation.

While the simulator does not model detailed timing effects or microarchitectural optimizations, this limitation is intentional. The focus on clarity and conceptual correctness makes the tool suitable for coursework, self-study, and demonstration purposes in computer architecture and systems classes.

Overall, the project demonstrates how a carefully designed software model can effectively capture the essential behavior of cache memory systems, providing both a practical learning tool and a solid foundation for future extensions or more detailed architectural simulations.

## References

- [1] Zoltan Baruch. *AC – Memoria Cache*. Cluj-Napoca: Universitatea Tehnică din Cluj-Napoca, n.d. URL: [https://users.utcluj.ro/~baruch/book\\_ac/AC-Memoria-Cache.pdf](https://users.utcluj.ro/~baruch/book_ac/AC-Memoria-Cache.pdf).
- [2] GeeksforGeeks. *Cache Memory*. 2025. URL: <https://www.geeksforgeeks.org/computer-science-fundamentals/cache-memory/>.
- [3] GeeksforGeeks. *Cache Memory in Computer Organization*. 2025. URL: <https://www.geeksforgeeks.org/computer-organization-architecture/cache-memory-in-computer-organization/>.
- [4] Lenovo. *What Is Cache Memory? L1, L2, L3 Levels Guide*. n.d. URL: <https://www.lenovo.com/us/en/glossary/what-is-cache-memory/>.
- [5] P. Paschos et al. “Performance Analysis of Cache Coherence Protocols for Multi-core Architectures: A System Attribute Perspective”. In: *Journal of Parallel and Distributed Computing* (2016). URL: [https://www.researchgate.net/publication/310360949\\_Performance\\_Analysis\\_of\\_Cache\\_Coherence\\_Protocols\\_for\\_Multi-core\\_Architectures\\_A\\_System\\_Attribute\\_Perspective](https://www.researchgate.net/publication/310360949_Performance_Analysis_of_Cache_Coherence_Protocols_for_Multi-core_Architectures_A_System_Attribute_Perspective).
- [6] Stanford University. *Cache Coherence Lecture Notes, CS149: Parallel Computing*. 2020. URL: [https://gfxcourses.stanford.edu/cs149/fall20content/media/cachecoherence/10\\_coherence.pdf](https://gfxcourses.stanford.edu/cs149/fall20content/media/cachecoherence/10_coherence.pdf).
- [7] Wikipedia contributors. *CPU Cache*. 2025. URL: [https://en.wikipedia.org/wiki/CPU\\_cache](https://en.wikipedia.org/wiki/CPU_cache).