

**Project Report**

**Course Title:** Computer Architecture

**Course Code:** CSE 360

**Section No:** 01

**Submitted To:**

Nawab Yusuf Ali

Professor

Department of Computer Science & Engineering

**Submitted By:**

**Group-10**

|  |  |
| --- | --- |
| **Student Name** | **Student ID** |
| Md Saiful Islam | 2022-3-60-045 |
| Umme Mukaddisa | 2022-3-60-317 |

[**1. Title 3**](#_uoxzjy3j5fub)

[**2. Objective 3**](#_6ntri2w1vgo8)

[**3. Theory 4**](#_sb9r4e1eyaef)

[3.1. Key Concepts 4](#_yjncyo1aiiaf)

[3.2. Cache Parameters 4](#_uf16m2sdtso)

[3.3. Hit & Miss 4](#_rcgdf6uf3k2n)

[3.4 Cache Mapping Techniques 5](#_qxhj2z3slnci)

[**4. Design 6**](#_rjmrrreqo7in)

[4.1. Memory Hierarchy Architecture 6](#_poezl7j8nix)

[4.2. Configuration Summary 7](#_fvgntiz9agro)

[4.3. Address Breakdown 7](#_5suk26d4loqt)

[4.4. Flow Chart 8](#_vrn0r0l1hvn6)

[**5. Implementation 9**](#_90tzin2xpiht)

[5.1. Development Environment 9](#_pxjwb6363qgg)

[5.2. Data Structures 9](#_vm5o55wp56oi)

[5.3. Functional Modules 10](#_loo5asnqzmcx)

[**6. Test Run & Result 16**](#_dlfw87byhl90)

[6.1. Main menu: 16](#_mny6dbhk4npk)

[6.2. Simulating Mapping Technique 16](#_306hsvtil8ns)

[6.3. Comparison and analysis of all mapping 21](#_uzqey0kzlfyh)

[**8. Conclusion & Future Improvements 24**](#_ttk8e4efyjkf)

[8.1. Limitation 24](#_gq4805rynza)

[8.2. Future Improvements 24](#_6cpe97jkgtsj)

[8.2. Conclusion 24](#_k3wp0hagyvtz)

[**9. Bibliography 25**](#_b8inxrqncdvf)

# **1. Title**

Memory Hierarchy Simulation: Analyzing Cache Performance Using C Programming.

# 

# **2. Objective**

The project's objective is to build a simple simulator to better understand how a computer’s memory hierarchy works, especially focusing on two-level L1 & L2 caches & how they interact with main memory. The main idea is to see how different cache setups affect performance when handling various types of memory access.

The simulator will include three types of cache mapping:

* **Direct-mapped** – where each memory block has a fixed place in the cache.
* **Fully associative** – where blocks can go anywhere, and the system keeps track of the least recently used ones.
* **Set-associative** – A hybrid approach where blocks map to a specific set and are placed within available lines in that set.

We’ll test these methods using different access patterns (like random, sequential, and repeated), and measure things like hit rate, miss rate, and average memory access time. It’ll also show how memory addresses break down into tag, set, and offset, so it’s easier to see how cache hits and misses happen.

The goal isn’t just to build the tool, but to actually learn from it—seeing what works best under which conditions and understanding the trade-offs between performance, complexity, and design.

# **3. Theory**

Modern processors use a memory hierarchy to balance speed & cost, which includes registers, multiple levels of cache (L1, L2, L3), main memory (RAM), & secondary storage. Cache memory stores frequently accessed data to boost access speed & improve processor performance.

## **3.1. Key Concepts**

* Cache memory gives the fastest memory access, at the same time providing a large memory size at the price of less expensive types of semiconductors.
* Sits between normal main memory (slower & relatively larger) & CPU.
* May be located on the CPU chip or module
* The cache contains a copy of portions of main memory.
* When the processor attempts to read a word from memory, a check is made to determine if the word is in the cache.

If it exists there, the word is delivered to the processor. If not, a block of main memory consisting of some fixed number of words is read into the cache & then it is delivered to the processor.

## **3.2. Cache Parameters**

* **Cache Size (C)**: Total size of the cache in bytes.
* **Block Size (B)**: Size of one cache line.
* **Associativity (A)**: Number of blocks per set (For set-associative mapping).
* **Number of Sets (S)**: Number of sets in CM = Number of CM blocks/set size.

## **3.3. Hit & Miss**

* **Hit**: Requested data is found in the cache.
* **Miss**: Requested data is not found in the cache & must be fetched from main memory.

**Formulas:**

* Miss ratio + hit ratio = 1
* Effective Access Time: Te = H x Tc + (1 - H)(Tm + Tc),

Where,

Tc access time of cache memory

Tm access time of main memory

## **3.4 Cache Mapping Techniques**

**Direct-Mapped (One Address → One Line)**

* Simple: Cache Line = Address % Number of Lines
* **Pro:** Fast lookup
* **Con:** Frequent collisions

**Fully Associative (Any Block → Any Line)**

* Data can go anywhere; the entire cache must be searched
* Uses **LRU** for replacement
* **Pro:** Best space utilization
* **Con:** Complex and slower lookup

**Set-Associative (Middle Ground)**

* Cache is split into sets; each block maps to one set, but can go in any line within it
* Example: 4-way set-associative → 4 lines per set
* Pro: Balances speed and flexibility

# **4. Design**

## **4.1. Memory Hierarchy Architecture**

The simulator models a 3-level memory hierarchy with direct-mapped caches at both L1 and L2, followed by main memory. Each level is defined by its size, latency, and structure.

**L1 Cache (Level 1)**

* **Blocks:** 1
* **Block Size:** 16 bytes (4 words × 4 bytes)
* **Set:** 2-way
* **Access Time:** 1 cycle
* **Characteristics:**
  + Fastest and smallest cache level
  + Simple address mapping
  + Higher miss rate due to fixed block placement

**L2 Cache (Level 2)**

* **Blocks:** 64 (4× more than L1)
* **Block Size:** 16 bytes
* **Set:** 4-way
* **Access Time:** 10 cycles
* **Characteristics:**
  + Larger and slower than L1
  + Reduces miss rate significantly
  + Inclusive of L1 (contains all L1 data)

**Main Memory**

* **Model:** Byte-addressable array
* **Blocks:** 256
* **Block Size:** 16 bytes
* **Access Time:** 100 cycles
* **Characteristics:**
  + Simulates DRAM timing
  + Accessed only after L2 cache miss
  + Transfers data in cache block sizes

## **4.2. Configuration Summary**

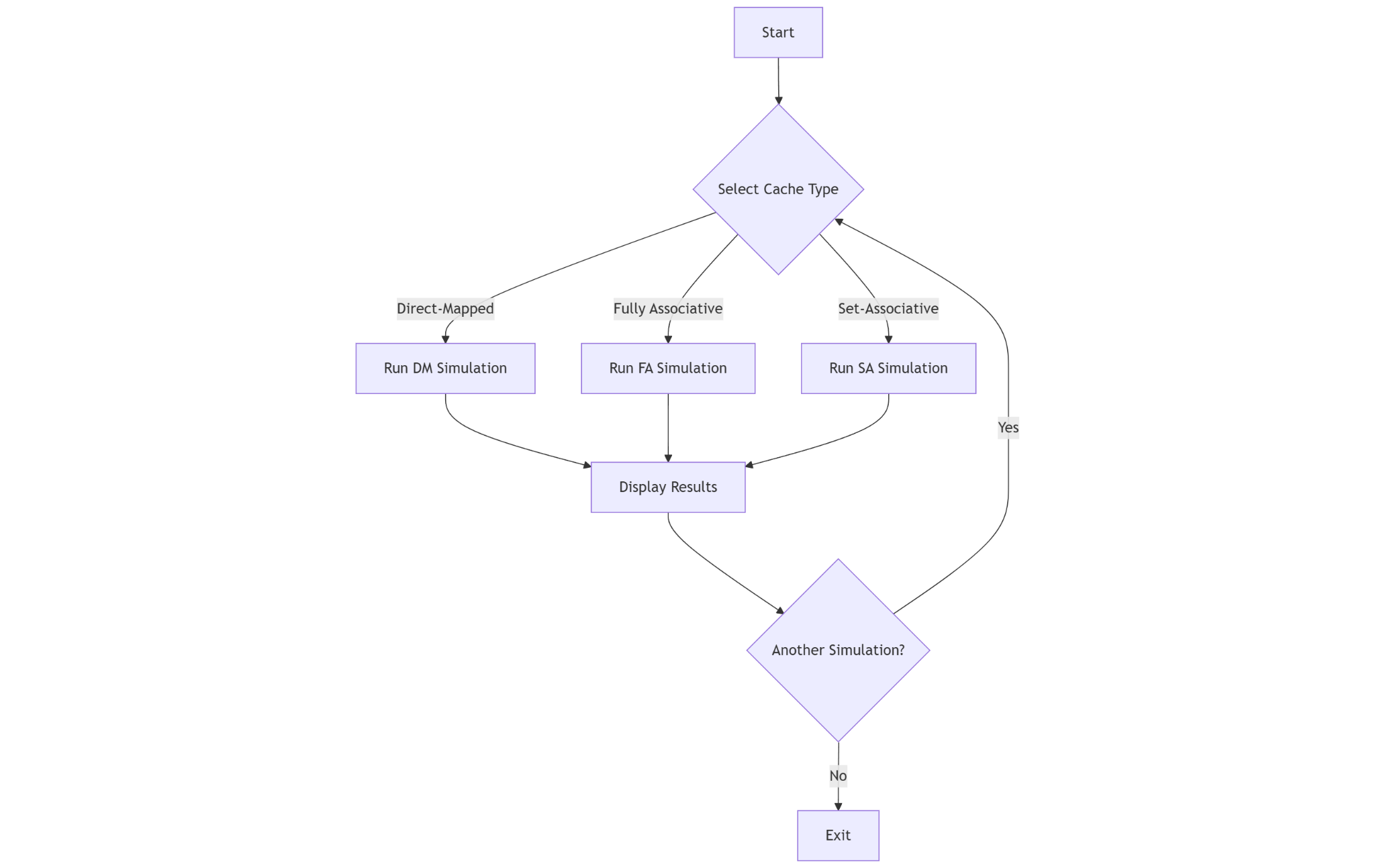
|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter** | **L1 Cache** | **L2 Cache** | **Main Memory** |
| Size | 256 bytes | 1 KB | 4 KB |
| Access Time | 1 cycle | 10 cycles | 100 cycles |
| Block Size | 16 bytes | 16 bytes | 16 bytes |
| Number of Blocks | 16 | 64 | 256 |

## **4.3. Address Breakdown**

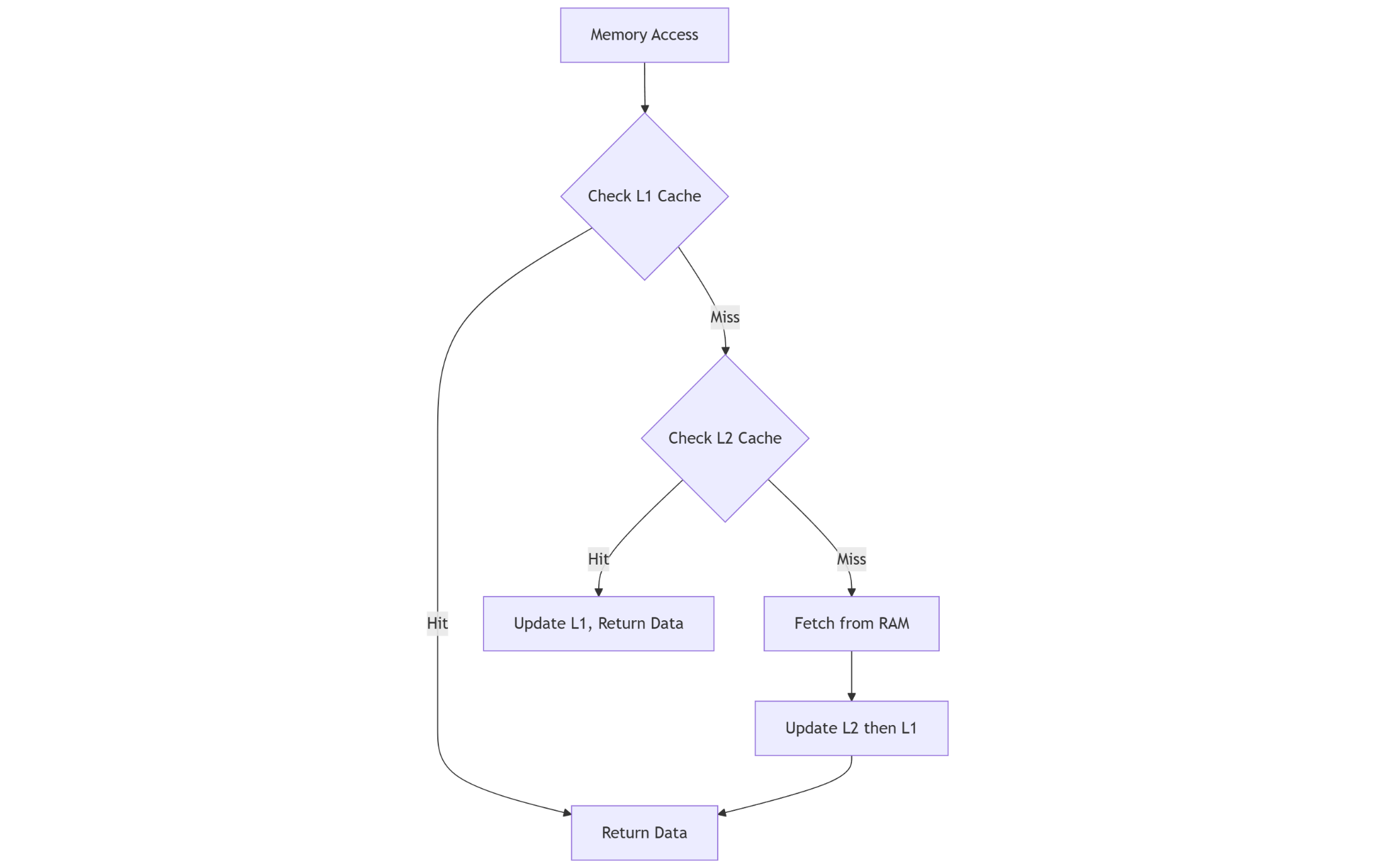
Each 32-bit memory address is divided into the following components:

* **Tag:** Uniquely identifies the memory block
* **Set Index:** Determines the cache set/block location
* **Block Offset (Word):** Selects the word within a block

## **4.4. Flow Chart**



**Main Program Flow**



**Cache Access Flow**

# **5. Implementation**

## **5.1. Development Environment**

* Programming Language: C
* Operating System: Windows 10
* Compiler: GCC (MinGW-w64)
* Development Tools: Codeblocks

## **5.2. Data Structures**

**// Cache line structure**

typedef struct {

int tag;

bool valid;

unsigned int address;

} CacheLine;

**// cache stat structure**

typedef struct {

int l1\_hits;

int l2\_hits;

int memory\_accesses;

long total\_cost;

float hit\_rate;

float avg\_access\_time;

} CacheStats;

// Structure to store hit address information

typedef struct {

unsigned int address;

int cache\_level; // 1 for L1 hit, 2 for L2 hit

} HitInfo;

## **5.3. Functional Modules**

// Initialize cache with invalid lines

void initializeCache(CacheLine \*cache, int size) {

for (int i = 0; i < size; i++) {

cache[i].valid = false;

cache[i].tag = -1;

cache[i].address = 0;

}

}

// Generate random memory addresses

void generateAddresses(unsigned int \*addresses, int numAccesses) {

for (int i = 0; i < numAccesses; i++) {

addresses[i] = (rand() % (ADDRESS\_SPACE / WORD\_SIZE)) \* WORD\_SIZE;

}

}

// Check if an address is in the cache

bool checkCache(CacheLine \*cache, int cacheSize, unsigned int address, int \*tag, int \*index) {

\*index = (address / (WORDS\_PER\_LINE \* WORD\_SIZE)) % cacheSize;

\*tag = address / (cacheSize \* WORDS\_PER\_LINE \* WORD\_SIZE);

return (cache[\*index].valid && cache[\*index].tag == \*tag);

}

// Update cache with new address

void updateCache(CacheLine \*cache, int index, int tag, unsigned int address) {

cache[index].valid = true;

cache[index].tag = tag;

cache[index].address = address;

}

//Cache Simulation Module

int cacheSimulation(int t)

{

int numAccesses;

int i;

// Statistics variables

int l1Hits = 0, l1Misses = 0;

int l2Hits = 0, l2Misses = 0;

long totalCycles = 0;

// Create cache structures

CacheLine l1Cache[L1\_SIZE];

CacheLine l2Cache[L2\_SIZE];

// Initialize caches

initializeCache(l1Cache, L1\_SIZE);

initializeCache(l2Cache, L2\_SIZE);

printf("Enter the number of memory access attempts to simulate: ");

scanf("%d", &numAccesses);

// Allocate memory for addresses

unsigned int \*addresses = (unsigned int \*)malloc(numAccesses \* sizeof(unsigned int));

// Create an array to store hit information

HitInfo \*hitInfoArray = (HitInfo \*)malloc(numAccesses \* sizeof(HitInfo));

int hitCount = 0;

// Generate random addresses

generateAddresses(addresses, numAccesses);

// Simulate memory accesses

for (i = 0; i < numAccesses; i++)

{

unsigned int address = addresses[i];

int tag, index;

bool l1Hit, l2Hit;

if(t==1)

{

// Print address breakdown with TAG/SET/WORD

printAddressBreakdown(address);

}

// Check L1 Cache

l1Hit = checkCache(l1Cache, L1\_SIZE, address, &tag, &index);

if(t==1) animateCheckL1();

if (l1Hit)

{

// L1 Cache hit

l1Hits++;

totalCycles += L1\_ACCESS\_COST;

// Record hit information

hitInfoArray[hitCount].address = address;

hitInfoArray[hitCount].cache\_level = 1; // L1 hit

hitCount++;

if(t==1)

{

printf("L1 CACHE HIT!\n");

displayCacheContents(l1Cache, L1\_SIZE, "L1 Cache");

if (hitCount > 0)

{

displayHitSummary(hitInfoArray, hitCount);

}

else

{

printf("No cache hits recorded during this simulation.\n");

}

printf("\nPress Enter to continue to next memory access...");

while (getchar() != '\n'); // Clear input buffer

getchar();

}

}

else

{

// L1 Cache miss

l1Misses++;

if(t==1)

{

printf("L1 CACHE MISS!\n");

// Check L2 Cache

usleep(500000);

}

l2Hit = checkCache(l2Cache, L2\_SIZE, address, &tag, &index);

if(t==1) animateCheckL2();

if (l2Hit)

{

// L2 Cache hit

l2Hits++;

totalCycles += L2\_ACCESS\_COST;

// Record hit information

hitInfoArray[hitCount].address = address;

hitInfoArray[hitCount].cache\_level = 2; // L2 hit

hitCount++;

if(t==1)

{

printf("\nL2 CACHE HIT!\n");

// Update L1 Cache

updateCache(l1Cache, (address / (WORDS\_PER\_LINE \* WORD\_SIZE)) % L1\_SIZE,

address / (L1\_SIZE \* WORDS\_PER\_LINE \* WORD\_SIZE), address);

printf("Data loaded from L2 to L1\n\n");

displayCacheContents(l2Cache, L2\_SIZE, "L2 Cache");

if (hitCount > 0)

{

displayHitSummary(hitInfoArray, hitCount);

}

else

{

printf("No cache hits recorded during this simulation.\n");

}

printf("\nPress Enter to continue to next memory access...");

while (getchar() != '\n'); // Clear input buffer

getchar();

}

}

else

{

// L2 Cache miss

l2Misses++;

totalCycles += MEMORY\_ACCESS\_COST;

if(t==1)

{

printf("L2 CACHE MISS!\n");

animateCheckMM();

}

// Update L1 and L2 Caches (inclusive cache policy)

updateCache(l1Cache, (address / (WORDS\_PER\_LINE \* WORD\_SIZE)) % L1\_SIZE,

address / (L1\_SIZE \* WORDS\_PER\_LINE \* WORD\_SIZE), address);

updateCache(l2Cache, (address / (WORDS\_PER\_LINE \* WORD\_SIZE)) % L2\_SIZE,

address / (L2\_SIZE \* WORDS\_PER\_LINE \* WORD\_SIZE), address);

if(t==1)

{

printf("Data loaded from Main Memory to L1 and L2.\n\n");

// Display cache contents after update

displayCacheContents(l1Cache, L1\_SIZE, "L1 Cache");

displayCacheContents(l2Cache, L2\_SIZE, "L2 Cache");

printf("\nPress Enter to continue to next memory access...");

while (getchar() != '\n'); // Clear input buffer

getchar(); // Wait for user input

}

}

}

}

// Calculate final statistics

float l1HitRatio = (float)l1Hits / numAccesses;

float l2HitRatio = (l1Misses > 0) ? (float)l2Hits / l1Misses : 0;

float amat = L1\_ACCESS\_COST + (1 - l1HitRatio) \* (L2\_ACCESS\_COST + (1 - l2HitRatio) \* MEMORY\_ACCESS\_COST);

// Display final statistics

clearScreen();

printf("Memory Hierarchy Simulation Complete\n");

printf("------------------------------------\n\n");

printf("Cache Architecture:\n");

printf("------------------\n");

printf(" Cache Policy: Inclusive (L2 contains all entries in L1)\n");

printf(" L1 Cache: %d sets, %d-byte lines (%d words per line)\n", L1\_SIZE, WORDS\_PER\_LINE \* WORD\_SIZE, WORDS\_PER\_LINE);

printf(" L2 Cache: %d sets, %d-byte lines (%d words per line)\n\n", L2\_SIZE, WORDS\_PER\_LINE \* WORD\_SIZE, WORDS\_PER\_LINE);

printf("Simulation Results:\n");

printf("------------------\n");

printf("Total memory accesses: %d\n\n", numAccesses);

printf("L1 Cache Statistics:\n");

printf(" Hits: %d (%.2f%%)\n", l1Hits, l1HitRatio \* 100);

printf(" Misses: %d (%.2f%%)\n\n", l1Misses, (1 - l1HitRatio) \* 100);

printf("L2 Cache Statistics:\n");

printf(" Hits: %d (%.2f%%)\n", l2Hits, l2HitRatio \* 100);

printf(" Misses: %d (%.2f%%)\n\n", l2Misses, (1 - l2HitRatio) \* 100);

printf("Performance Metrics:\n");

printf(" Total Cycle Cost: %ld cycles\n", totalCycles);

printf(" Average Memory Access Time (AMAT): %.2f cycles\n\n", amat);

// Display hit address summary

if (hitCount > 0)

{

displayHitSummary(hitInfoArray, hitCount);

}

else

{

printf("No cache hits recorded during this simulation.\n");

}

// Free allocated memory

free(addresses);

free(hitInfoArray);

printf("\n===============================================\n");

printf("Direct Mapping analysis complete.\n");

printf("Press Enter to return to main menu...");

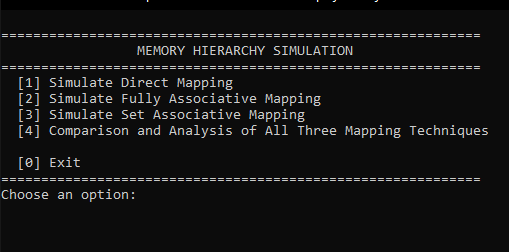
getchar();

getchar();

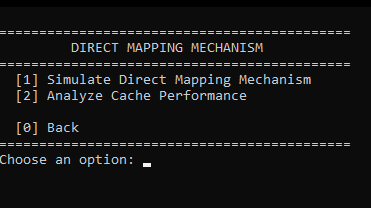
}

# **6. Test Run & Result**

## **6.1. Main menu:**

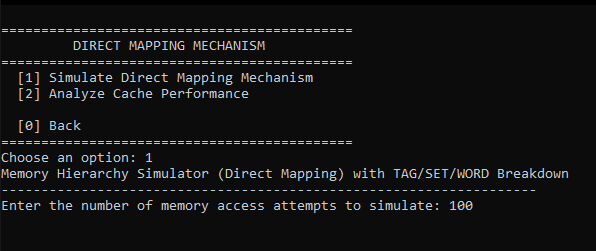


## **6.2. Simulating Mapping Technique**

****

#### 

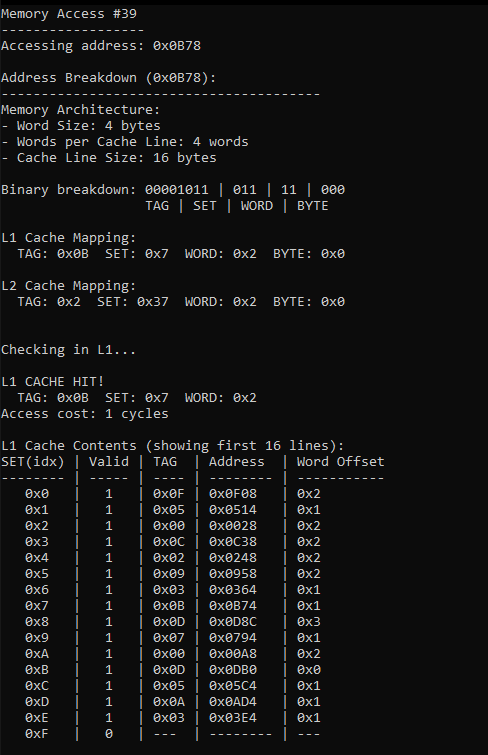
**Mapping Simulation(example: Direct Mapping)**



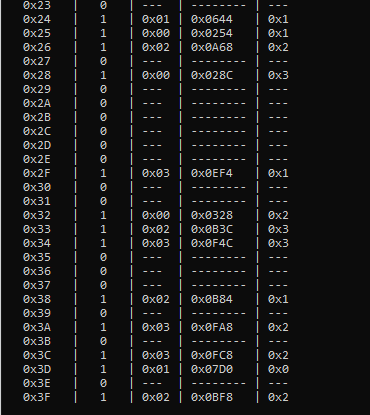
* **If a cache miss occurs in both L1 & L2:**

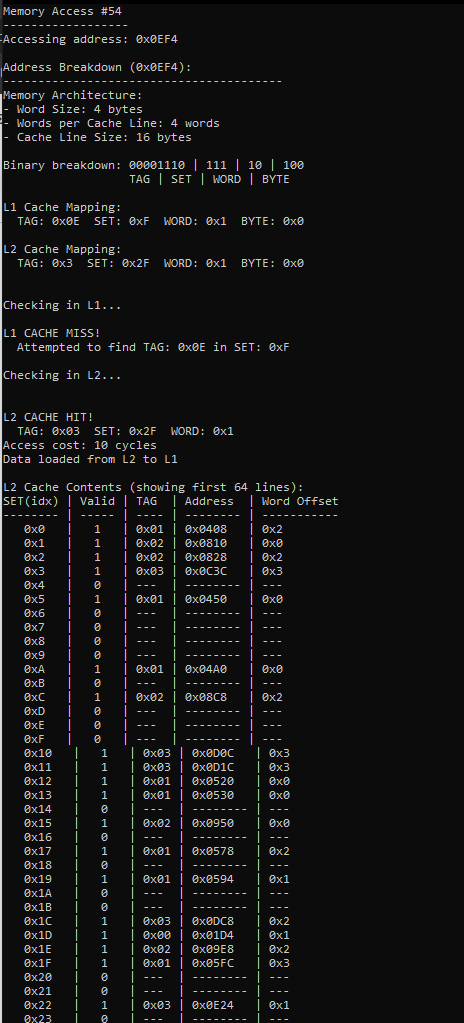
#### 

* **If a cache hit occurs in L1:**



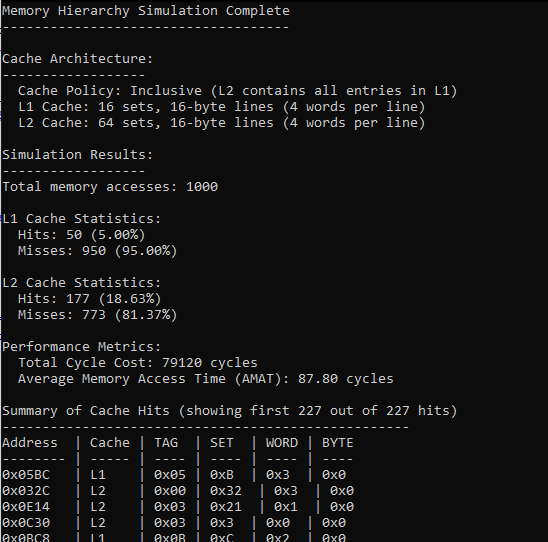
* **If a cache hit occurs in L2:**





**Cache Performance Analysis:**

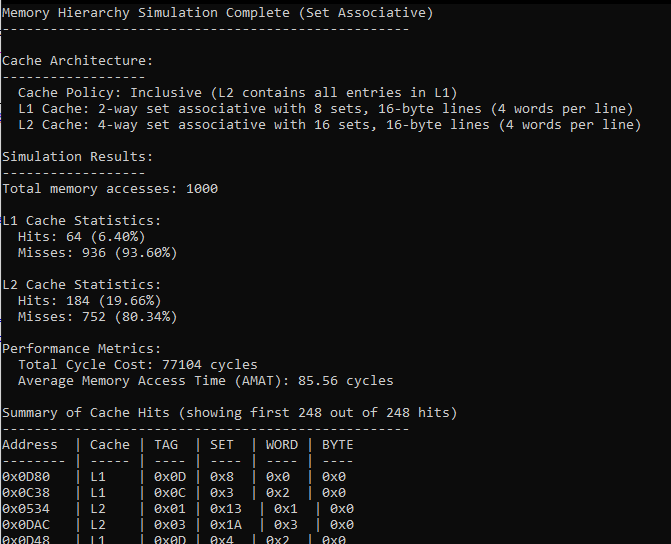
**Simulation Result for Direct Mapping:**



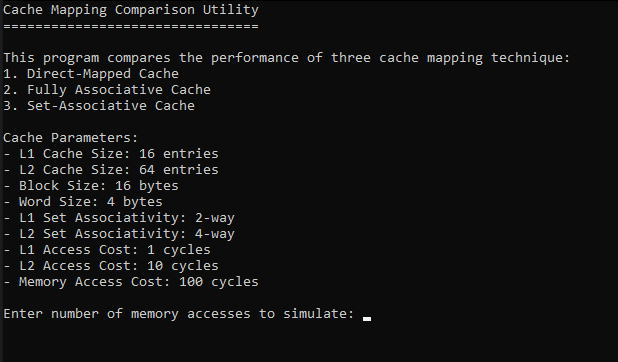
**Simulation Result for Associative Mapping:**

## 

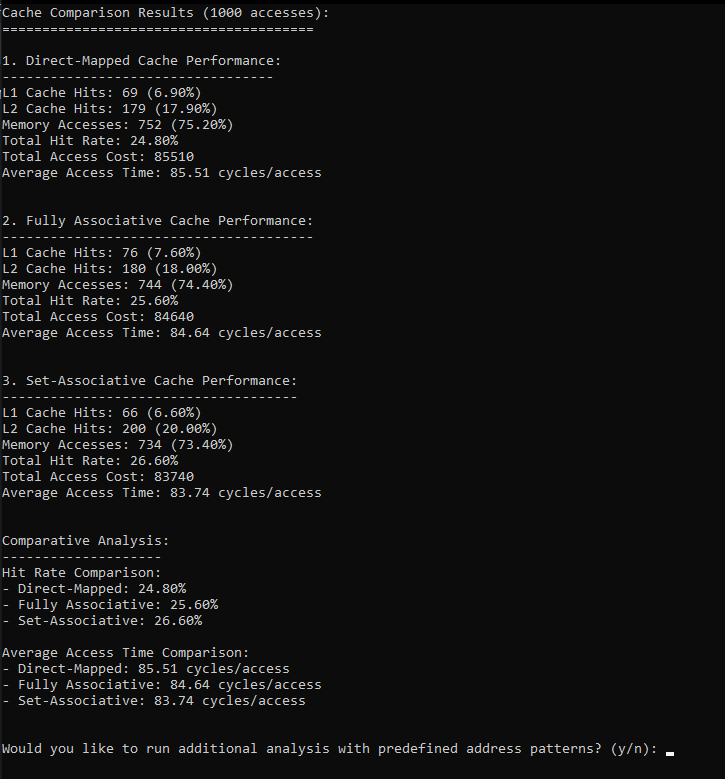
**Simulation Result for Set Associative Mapping:**



## **6.3. Comparison and analysis of all mapping**

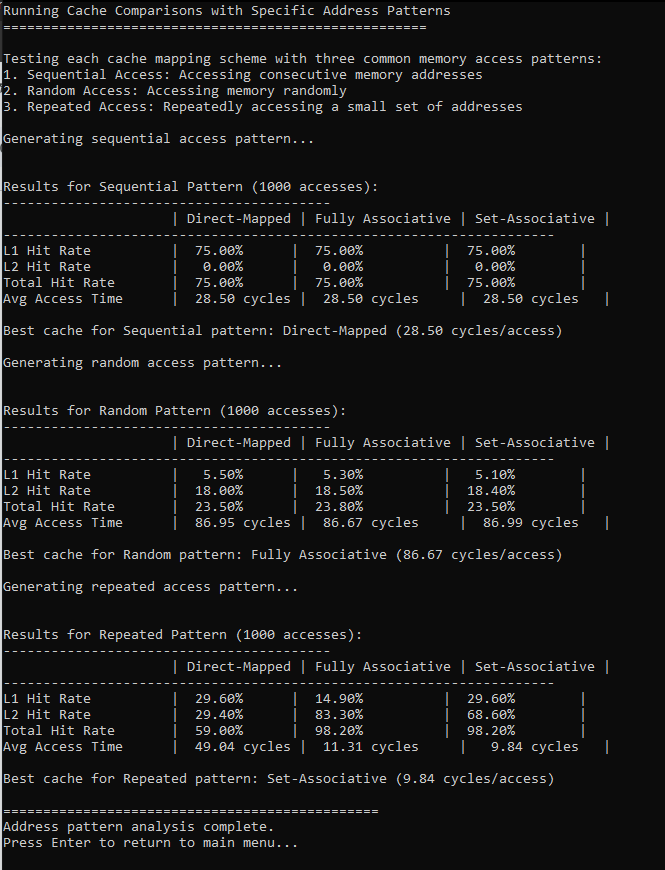


* **Analysis with random address:**



#### 

* **Analysis with predefined address patterns:**

****

# **8. Conclusion & Future Improvements**

## **8.1. Limitation**

* Fixed Block Size: The simulator uses a static block size (16 bytes), limiting exploration of spatial locality effects.
* Simplified Replacement Policy: Only direct replacement is implemented, missing advanced policies like LRU or FIFO.
* Single-Core Focus: The design does not account for multi-core cache coherence issues.

## **8.2. Future Improvements**

* Dynamic Replacement Policies
  + Integrate LRU, FIFO, and random replacement policies to study their impact on miss rates.
* Variable Block and Cache Sizes
  + Allow user-configurable block sizes to analyze trade-offs between miss rate and transfer overhead.
* Multi-Level Cache Simulation
  + Extend the hierarchy to include L3 cache and non-inclusive/exclusive policies.
* Real-World Workload Integration
  + Test with CPU trace files (e.g., SPEC benchmarks) for realistic performance evaluation.
* Graphical Visualization
  + Develop a Python/JavaScript GUI to dynamically display cache hits/misses and memory traffic.

## **8.2. Conclusion**

This project successfully simulated a memory hierarchy consisting of L1 and L2 caches along with main memory, allowing detailed analysis of cache behavior under various conditions. By implementing address breakdown, cache mapping techniques, and realistic latency values, the simulator provided insights into how memory access patterns affect performance metrics such as hit rate, miss rate, and average memory access time (AMAT).

Through experimentation with different access patterns—sequential, random, and repeated—the project demonstrated the importance of spatial and temporal locality in optimizing cache efficiency. The results highlight the trade-offs between cache size, speed, and complexity, which are essential considerations in modern processor design.

Overall, the simulator serves as an educational tool to better understand how caching works and how different architectural decisions can impact overall system performance.

# **9. Bibliography**

**Books**

1. Hennessy, J. L., & Patterson, D. A. (2017). Computer Architecture: A Quantitative Approach (6th ed.). Morgan Kaufmann.
2. Stallings, W. (2020). Computer Organization and Architecture: Designing for Performance (11th ed.). Pearson.

**Online Resources**

1. Wikipedia contributors. (2024). *CPU cache*. In *Wikipedia, The Free Encyclopedia*. Retrieved from<https://en.wikipedia.org/wiki/CPU_cache>

**Tutorials & References**

1. GeeksforGeeks. (2023). Cache Memory in Computer Organization. <https://www.geeksforgeeks.org/cache-memory/>
2. TutorialsPoint. (2022). Computer Architecture – Memory Hierarchy. <https://www.tutorialspoint.com/>