EAST WEST UNIVERSITY

Department of Computer Science & Engineering

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**

2. Objective **3**

3. Theory **4**

3.1. Key Concepts 4

3.2. Cache Parameters 4

3.3. Hit & Miss 4

3.4 Cache Mapping Techniques 5

4. Design **6**

4.1. Memory Hierarchy Architecture 6

4.2. Configuration Summary 7

4.3. Address Breakdown 7

4.4. Flow Chart 8

5. Implementation **9**

5.1. Development Environment 9

5.2. Data Structures 9

5.3. Functional Modules 10

6. Test Run & Result **16**

6.1. Main menu: 16

6.2. Simulating Mapping Technique 16

6.3. Comparison and analysis of all mapping 21

8. Conclusion & Future Improvements **24**

8.1. Limitation 24

8.2. Future Improvements 24

8.2. Conclusion 24

9. Bibliography **25**

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

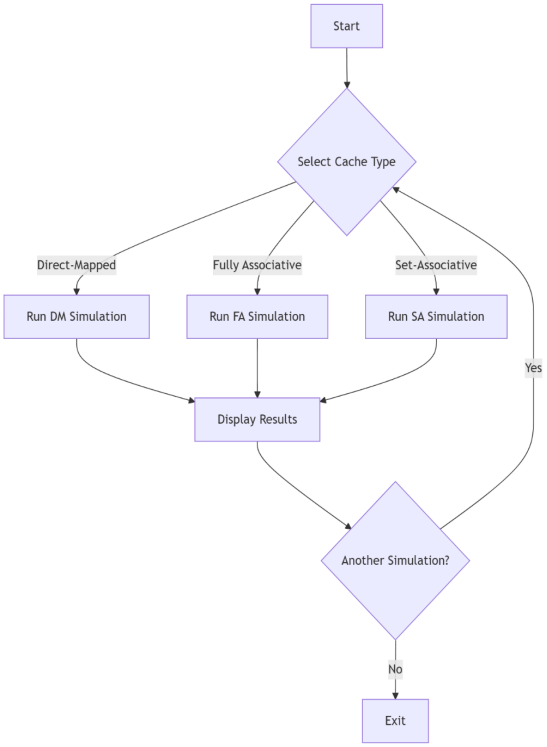
4.3. Address Breakdown

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

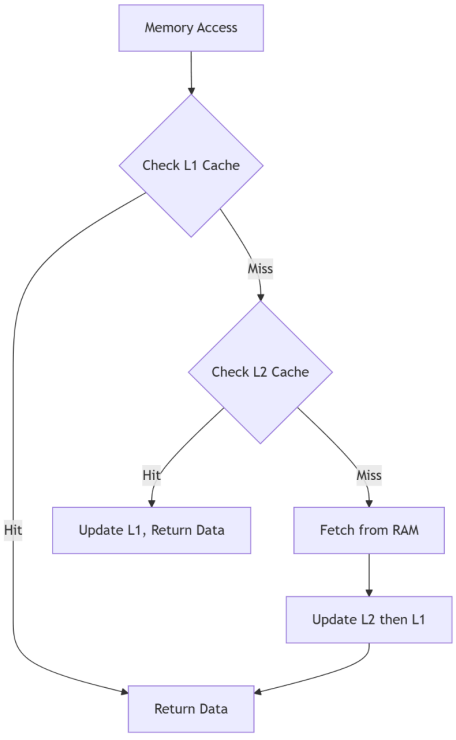
- **Tag:** Uniquely identifies the memory block
- **Set Index:** Determines the cache set/block location
- **Block Offset (Word):** Selects the word within a block

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);
```

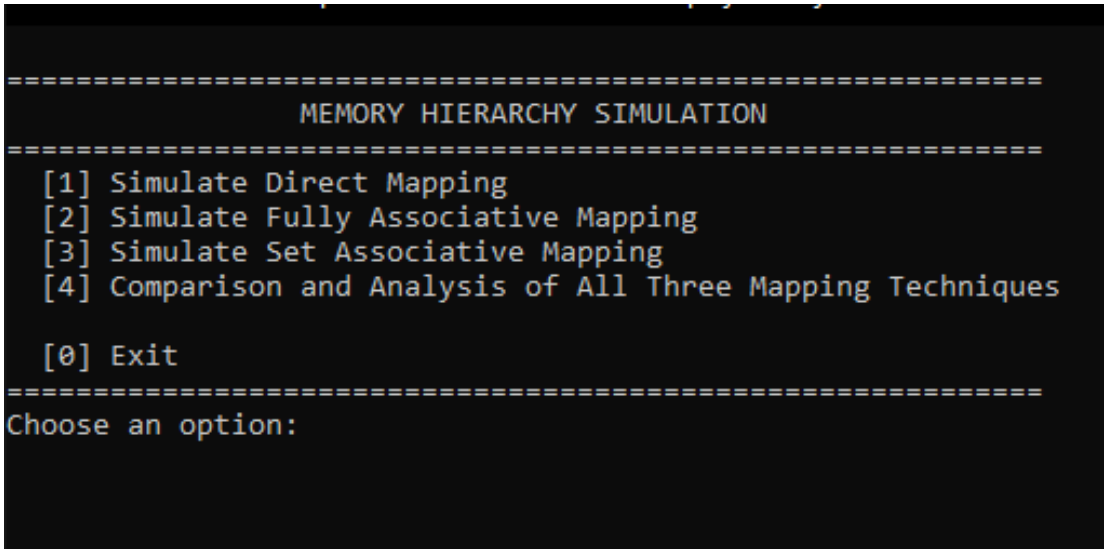
```
}


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");
```

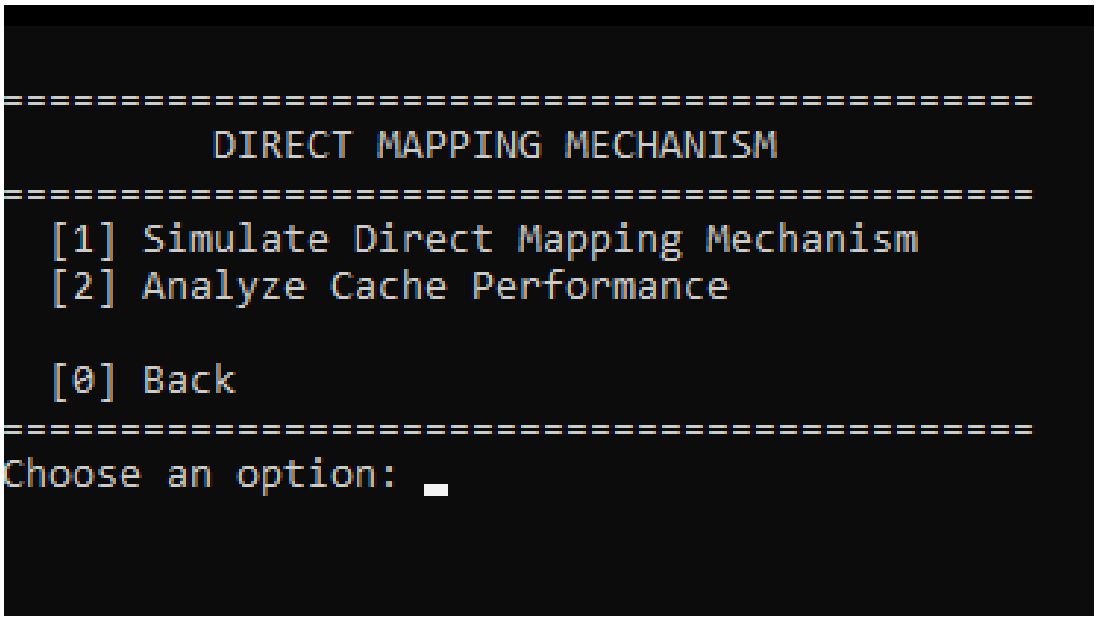
```
    printf("-----------------------------------\n\n");
    printf("Cache Architecture:\n");
    printf("------------------\n");
    printf("  Cache Policy: Inclusive (L2 contains all entries in L1)\n");
    printf("  L1 Cache: %d sets, %d-byte lines (%d words per line)\n", L1_SIZE, WORDS_PER_LINE *
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)

```
==========================================
        DIRECT MAPPING MECHANISM
==========================================
  [1] Simulate Direct Mapping Mechanism
  [2] Analyze Cache Performance

  [0] Back
==========================================
Choose an option: 1
Memory Hierarchy Simulator (Direct Mapping) with TAG/SET/WORD Breakdown
------------------------------------------------------------------
Enter the number of memory access attempts to simulate: 100
```

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

```
Memory Access #1
------------------
Accessing address: 0x069C

Address Breakdown (0x069C):
----------------------------------------
Memory Architecture:
- Word Size: 4 bytes
- Words per Cache Line: 4 words
- Cache Line Size: 16 bytes

Binary breakdown: 00000110 | 100 | 11 | 100
                  TAG | SET | WORD | BYTE

L1 Cache Mapping:
  TAG: 0x06  SET: 0x9  WORD: 0x3  BYTE: 0x0

L2 Cache Mapping:
  TAG: 0x1  SET: 0x29  WORD: 0x3  BYTE: 0x0


Checking in L1...

L1 CACHE MISS!
  Attempted to find TAG: 0x06 in SET: 0x9

Checking in L2...

L2 CACHE MISS!
  Attempted to find TAG: 0x01 in SET: 0x29

Accessing Main Memory.....

Access cost: 100 cycles
Data loaded from Main Memory to L1 and L2.
```

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

```
Memory Access #39
------------------
Accessing address: 0x0B78

Address Breakdown (0x0B78):
----------------------------------------
Memory Architecture:
- Word Size: 4 bytes
- Words per Cache Line: 4 words
- Cache Line Size: 16 bytes

Binary breakdown: 00001011 | 011 | 11 | 000
                  TAG | SET | WORD | BYTE

L1 Cache Mapping:
  TAG: 0x0B  SET: 0x7  WORD: 0x2  BYTE: 0x0

L2 Cache Mapping:
  TAG: 0x2  SET: 0x37  WORD: 0x2  BYTE: 0x0


Checking in L1...

L1 CACHE HIT!
  TAG: 0x0B  SET: 0x7  WORD: 0x2
Access cost: 1 cycles

L1 Cache Contents (showing first 16 lines):
SET(idx) | Valid | TAG  | Address  | Word Offset
-------- | ----- | ---- | -------- | -----------
   0x0   |   1   | 0x0F | 0x0F08   | 0x2
   0x1   |   1   | 0x05 | 0x0514   | 0x1
   0x2   |   1   | 0x00 | 0x0028   | 0x2
   0x3   |   1   | 0x0C | 0x0C38   | 0x2
   0x4   |   1   | 0x02 | 0x0248   | 0x2
   0x5   |   1   | 0x09 | 0x0958   | 0x2
   0x6   |   1   | 0x03 | 0x0364   | 0x1
   0x7   |   1   | 0x0B | 0x0B74   | 0x1
   0x8   |   1   | 0x0D | 0x0D8C   | 0x3
   0x9   |   1   | 0x07 | 0x0794   | 0x1
   0xA   |   1   | 0x00 | 0x00A8   | 0x2
   0xB   |   1   | 0x0D | 0x0DB0   | 0x0
   0xC   |   1   | 0x05 | 0x05C4   | 0x1
   0xD   |   1   | 0x0A | 0x0AD4   | 0x1
   0xE   |   1   | 0x03 | 0x03E4   | 0x1
   0xF   |   0   | ---  | -------- | ---
```

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

```
Memory Access #54
------------------
Accessing address: 0x0EF4

Address Breakdown (0x0EF4):
----------------------------------------
Memory Architecture:
- Word Size: 4 bytes
- Words per Cache Line: 4 words
- Cache Line Size: 16 bytes

Binary breakdown: 00001110 | 111 | 10 | 100
                  TAG | SET | WORD | BYTE

L1 Cache Mapping:
  TAG: 0x0E  SET: 0xF  WORD: 0x1  BYTE: 0x0

L2 Cache Mapping:
  TAG: 0x3  SET: 0x2F  WORD: 0x1  BYTE: 0x0


Checking in L1...

L1 CACHE MISS!
  Attempted to find TAG: 0x0E in SET: 0xF

Checking in L2...


L2 CACHE HIT!
  TAG: 0x03  SET: 0x2F  WORD: 0x1
Access cost: 10 cycles
Data loaded from L2 to L1

L2 Cache Contents (showing first 64 lines):
SET(idx) | Valid | TAG  | Address  | Word Offset
-------- | ----- | ---- | -------- | -----------
   0x0   |   1   | 0x01 | 0x0408   | 0x2
   0x1   |   1   | 0x02 | 0x0810   | 0x0
   0x2   |   1   | 0x02 | 0x0828   | 0x2
   0x3   |   1   | 0x03 | 0x0C3C   | 0x3
   0x4   |   0   | ---  | -------- | ---
   0x5   |   1   | 0x01 | 0x0450   | 0x0
   0x6   |   0   | ---  | -------- | ---
   0x7   |   0   | ---  | -------- | ---
   0x8   |   0   | ---  | -------- | ---
   0x9   |   0   | ---  | -------- | ---
   0xA   |   1   | 0x01 | 0x04A0   | 0x0
   0xB   |   0   | ---  | -------- | ---
   0xC   |   1   | 0x02 | 0x08C8   | 0x2
   0xD   |   0   | ---  | -------- | ---
   0xE   |   0   | ---  | -------- | ---
   0xF   |   0   | ---  | -------- | ---
   0x10  |   1   | 0x03 | 0x0D0C   | 0x3
   0x11  |   1   | 0x03 | 0x0D1C   | 0x3
   0x12  |   1   | 0x01 | 0x0520   | 0x0
   0x13  |   1   | 0x01 | 0x0530   | 0x0
   0x14  |   0   | ---  | -------- | ---
   0x15  |   1   | 0x02 | 0x0950   | 0x0
   0x16  |   0   | ---  | -------- | ---
   0x17  |   1   | 0x01 | 0x0578   | 0x2
   0x18  |   0   | ---  | -------- | ---
   0x19  |   1   | 0x01 | 0x0594   | 0x1
   0x1A  |   0   | ---  | -------- | ---
   0x1B  |   0   | ---  | -------- | ---
   0x1C  |   1   | 0x03 | 0x0DC8   | 0x2
   0x1D  |   1   | 0x00 | 0x01D4   | 0x1
   0x1E  |   1   | 0x02 | 0x09E8   | 0x2
   0x1F  |   1   | 0x01 | 0x05FC   | 0x3
   0x20  |   0   | ---  | -------- | ---
   0x21  |   0   | ---  | -------- | ---
   0x22  |   1   | 0x03 | 0x0E24   | 0x1
   0x23  |   0   | ---  | -------- | ---
```

```
   0x23  |   0   | ---  | -------- | ---
   0x24  |   1   | 0x01 | 0x0644   | 0x1
   0x25  |   1   | 0x00 | 0x0254   | 0x1
   0x26  |   1   | 0x02 | 0x0A68   | 0x2
   0x27  |   0   | ---  | -------- | ---
   0x28  |   1   | 0x00 | 0x028C   | 0x3
   0x29  |   0   | ---  | -------- | ---
   0x2A  |   0   | ---  | -------- | ---
   0x2B  |   0   | ---  | -------- | ---
   0x2C  |   0   | ---  | -------- | ---
   0x2D  |   0   | ---  | -------- | ---
   0x2E  |   0   | ---  | -------- | ---
   0x2F  |   1   | 0x03 | 0x0EF4   | 0x1
   0x30  |   0   | ---  | -------- | ---
   0x31  |   0   | ---  | -------- | ---
   0x32  |   1   | 0x00 | 0x0328   | 0x2
   0x33  |   1   | 0x02 | 0x0B3C   | 0x3
   0x34  |   1   | 0x03 | 0x0F4C   | 0x3
   0x35  |   0   | ---  | -------- | ---
   0x36  |   0   | ---  | -------- | ---
   0x37  |   0   | ---  | -------- | ---
   0x38  |   1   | 0x02 | 0x0B84   | 0x1
   0x39  |   0   | ---  | -------- | ---
   0x3A  |   1   | 0x03 | 0x0FA8   | 0x2
   0x3B  |   0   | ---  | -------- | ---
   0x3C  |   1   | 0x03 | 0x0FC8   | 0x2
   0x3D  |   1   | 0x01 | 0x07D0   | 0x0
   0x3E  |   0   | ---  | -------- | ---
   0x3F  |   1   | 0x02 | 0x0BF8   | 0x2
```

Cache Performance Analysis:

Simulation Result for Direct Mapping:

```
Memory Hierarchy Simulation Complete
------------------------------------

Cache Architecture:
-------------------
  Cache Policy: Inclusive (L2 contains all entries in L1)
  L1 Cache: 16 sets, 16-byte lines (4 words per line)
  L2 Cache: 64 sets, 16-byte lines (4 words per line)

Simulation Results:
-------------------
Total memory accesses: 1000

L1 Cache Statistics:
  Hits: 50 (5.00%)
  Misses: 950 (95.00%)

L2 Cache Statistics:
  Hits: 177 (18.63%)
  Misses: 773 (81.37%)

Performance Metrics:
  Total Cycle Cost: 79120 cycles
  Average Memory Access Time (AMAT): 87.80 cycles

Summary of Cache Hits (showing first 227 out of 227 hits)
---------------------------------------------------------
Address  | Cache | TAG  | SET  | WORD | BYTE
-------- | ----- | ---- | ---- | ---- | ----
0x05BC   | L1    | 0x05 | 0xB  | 0x3  | 0x0
0x032C   | L2    | 0x00 | 0x32 | 0x3  | 0x0
0x0E14   | L2    | 0x03 | 0x21 | 0x1  | 0x0
0x0C30   | L2    | 0x03 | 0x3  | 0x0  | 0x0
0x0BC8   | L1    | 0x0B | 0xC  | 0x2  | 0x0
```

Simulation Result for Associative Mapping:

```
Memory Hierarchy Simulation Complete (Fully Associative)
--------------------------------------------------------

Cache Architecture:
-------------------
  Cache Policy: Inclusive (L2 contains all entries in L1)
  L1 Cache: Fully associative with 16 entries, 16-byte lines (4 words per line)
  L2 Cache: Fully associative with 64 entries, 16-byte lines (4 words per line)

Simulation Results:
-------------------
Total memory accesses: 1000

L1 Cache Statistics:
  Hits: 74 (7.40%)
  Misses: 926 (92.60%)

L2 Cache Statistics:
  Hits: 167 (18.03%)
  Misses: 759 (81.97%)

Performance Metrics:
  Total Cycle Cost: 77644 cycles
  Average Memory Access Time (AMAT): 86.16 cycles

Summary of Cache Hits (showing first 241 out of 241 hits)
---------------------------------------------------------
Address  | Cache | TAG  | SET  | WORD | BYTE
-------- | ----- | ---- | ---- | ---- | ----
0x0E40   | L1    | 0x0E | 0x4  | 0x0  | 0x0
0x0150   | L1    | 0x01 | 0x5  | 0x0  | 0x0
0x0138   | L2    | 0x00 | 0x13 | 0x2  | 0x0
0x0A28   | L1    | 0x0A | 0x2  | 0x2  | 0x0
0x03F4   | L2    | 0x00 | 0x3F | 0x1  | 0x0
0x0A80   | L1    | 0x0A | 0x8  | 0x0  | 0x0
0x0870   | L2    | 0x02 | 0x7  | 0x0  | 0x0
0x098C   | L2    | 0x02 | 0x18 | 0x3  | 0x0
```

Simulation Result for Set Associative Mapping:

```
Memory Hierarchy Simulation Complete (Set Associative)
------------------------------------------------------

Cache Architecture:
-------------------
  Cache Policy: Inclusive (L2 contains all entries in L1)
  L1 Cache: 2-way set associative with 8 sets, 16-byte lines (4 words per line)
  L2 Cache: 4-way set associative with 16 sets, 16-byte lines (4 words per line)

Simulation Results:
-------------------
Total memory accesses: 1000

L1 Cache Statistics:
  Hits: 64 (6.40%)
  Misses: 936 (93.60%)

L2 Cache Statistics:
  Hits: 184 (19.66%)
  Misses: 752 (80.34%)

Performance Metrics:
  Total Cycle Cost: 77104 cycles
  Average Memory Access Time (AMAT): 85.56 cycles

Summary of Cache Hits (showing first 248 out of 248 hits)
---------------------------------------------------------
Address  | Cache | TAG  | SET  | WORD | BYTE
-------- | ----- | ---- | ---- | ---- | ----
0x0D80   | L1    | 0x0D | 0x8  | 0x0  | 0x0
0x0C38   | L1    | 0x0C | 0x3  | 0x2  | 0x0
0x0534   | L2    | 0x01 | 0x13 | 0x1  | 0x0
0x0DAC   | L2    | 0x03 | 0x1A | 0x3  | 0x0
0x0D48   | L1    | 0x0D | 0x4  | 0x2  | 0x0
```

6.3. Comparison and analysis of all mapping

```
Cache Mapping Comparison Utility
================================

This program compares the performance of three cache mapping technique:
1. Direct-Mapped Cache
2. Fully Associative Cache
3. Set-Associative Cache

Cache Parameters:
- L1 Cache Size: 16 entries
- L2 Cache Size: 64 entries
- Block Size: 16 bytes
- Word Size: 4 bytes
- L1 Set Associativity: 2-way
- L2 Set Associativity: 4-way
- L1 Access Cost: 1 cycles
- L2 Access Cost: 10 cycles
- Memory Access Cost: 100 cycles

Enter number of memory accesses to simulate:
```

- **Analysis with random address:**

```
Cache Comparison Results (1000 accesses):
=========================================

1. Direct-Mapped Cache Performance:
----------------------------------
L1 Cache Hits: 69 (6.90%)
L2 Cache Hits: 179 (17.90%)
Memory Accesses: 752 (75.20%)
Total Hit Rate: 24.80%
Total Access Cost: 85510
Average Access Time: 85.51 cycles/access


2. Fully Associative Cache Performance:
---------------------------------------
L1 Cache Hits: 76 (7.60%)
L2 Cache Hits: 180 (18.00%)
Memory Accesses: 744 (74.40%)
Total Hit Rate: 25.60%
Total Access Cost: 84640
Average Access Time: 84.64 cycles/access


3. Set-Associative Cache Performance:
-------------------------------------
L1 Cache Hits: 66 (6.60%)
L2 Cache Hits: 200 (20.00%)
Memory Accesses: 734 (73.40%)
Total Hit Rate: 26.60%
Total Access Cost: 83740
Average Access Time: 83.74 cycles/access


Comparative Analysis:
--------------------
Hit Rate Comparison:
- Direct-Mapped: 24.80%
- Fully Associative: 25.60%
- Set-Associative: 26.60%

Average Access Time Comparison:
- Direct-Mapped: 85.51 cycles/access
- Fully Associative: 84.64 cycles/access
- Set-Associative: 83.74 cycles/access


Would you like to run additional analysis with predefined address patterns? (y/n): _
```

- **Analysis with predefined address patterns:**

```
Running Cache Comparisons with Specific Address Patterns
==================================================

Testing each cache mapping scheme with three common memory access patterns:
1. Sequential Access: Accessing consecutive memory addresses
2. Random Access: Accessing memory randomly
3. Repeated Access: Repeatedly accessing a small set of addresses

Generating sequential access pattern...


Results for Sequential Pattern (1000 accesses):
---------------------------------------
                    | Direct-Mapped | Fully Associative | Set-Associative |
-------------------------------------------------------------------------
L1 Hit Rate         |  75.00%       |  75.00%           |  75.00%         |
L2 Hit Rate         |   0.00%       |   0.00%           |   0.00%         |
Total Hit Rate      |  75.00%       |  75.00%           |  75.00%         |
Avg Access Time     |  28.50 cycles |  28.50 cycles     |  28.50 cycles   |

Best cache for Sequential pattern: Direct-Mapped (28.50 cycles/access)

Generating random access pattern...


Results for Random Pattern (1000 accesses):
---------------------------------------
                    | Direct-Mapped | Fully Associative | Set-Associative |
-------------------------------------------------------------------------
L1 Hit Rate         |   5.50%       |   5.30%           |   5.10%         |
L2 Hit Rate         |  18.00%       |  18.50%           |  18.40%         |
Total Hit Rate      |  23.50%       |  23.80%           |  23.50%         |
Avg Access Time     |  86.95 cycles |  86.67 cycles     |  86.99 cycles   |

Best cache for Random pattern: Fully Associative (86.67 cycles/access)

Generating repeated access pattern...


Results for Repeated Pattern (1000 accesses):
---------------------------------------
                    | Direct-Mapped | Fully Associative | Set-Associative |
-------------------------------------------------------------------------
L1 Hit Rate         |  29.60%       |  14.90%           |  29.60%         |
L2 Hit Rate         |  29.40%       |  83.30%           |  68.60%         |
Total Hit Rate      |  59.00%       |  98.20%           |  98.20%         |
Avg Access Time     |  49.04 cycles |  11.31 cycles     |   9.84 cycles   |

Best cache for Repeated pattern: Set-Associative (9.84 cycles/access)

==============================================
Address pattern analysis complete.
Press Enter to return to main menu...
```

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/