Portland State University
Maseeh College of Engineering and Computer Science
Electrical and Computer Engineering Department
ECE485 - Microprocessor System Design
Winter 2024

Cache Simulator Project

Final Report

[TEAM 4] Wa'el Al Kalbani, Mohammad Hasan, Yousef Alothman, Mohamed Gnedi
Instructor: **Dr. Yuchen Huang**
TA: **Sonali Fernado**
Date: March 1st, 2024

Overview

This project involves creating a simulation for a split L1 cache system in a 32-bit multiprocessor setup, using a MESI for cache coherence. It involves designing instruction and data caches, tracking usage statistics, simulating cache-L2 interactions, and supporting multiple operational modes. The outcome includes a detailed report and roles such as coding the base program, checking calculations and modifying code, Program Pseudocode and working on the cache simulations.

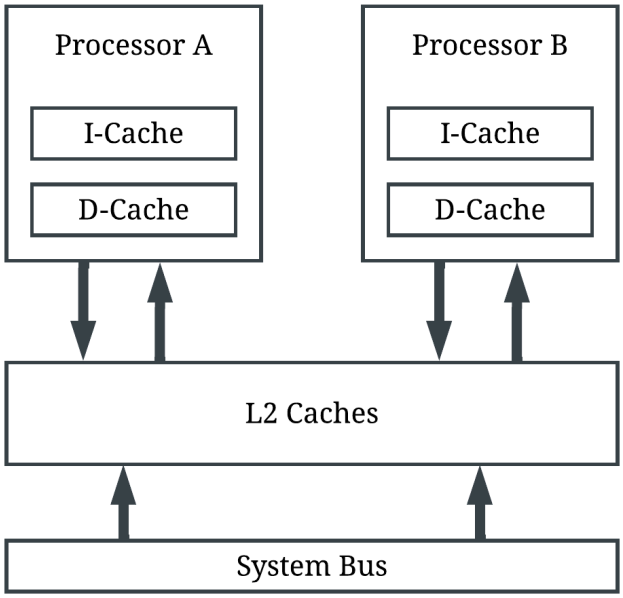

Figure 1: Program block diagram

Trace File: The simulator accepts a trace file as input, which contains a sequence of memory access operations. Each line in the trace file represents a single operation, coded as a numerical value followed by a hexadecimal memory address.

Simulation Mode: Users can select a simulation mode at the start of the program. The modes include:

Mode_0: Basic simulation with a summary of usage statistics.
Mode_1: Detailed simulation including cache coherence messages and more granular operational logs.
Mode_2: Exit the simulation

Table of Content

List of Tables

List of Figures

1. Introduction

In this project, we embark on designing and simulating a sophisticated cache system for a 32-bit multicore processor architecture. The aim is to create a Level 1 (L1) cache configuration, optimized for performance and coherence across multiple processors using the MESI protocol. Our approach involves meticulously configuring instruction and data caches for optimal efficiency, while also ensuring a seamless interaction with the Level 2 cache.

MESI Protocol

The MESI protocol is a widely used cache coherency protocol that ensures data consistency among caches in a multiprocessor system. The protocol defines four states that a cache line can be in: Modified (M), Exclusive (E), Shared (S), and Invalid (I). These states indicate the current status of the data in the cache line and dictate how caches interact with each other when accessing or modifying data.

Modified (M): This state indicates that the cache line has been modified and differs from the corresponding line in main memory. Any modifications made to this line must be written back to memory before the line can be replaced or invalidated.

Exclusive (E): This state indicates that the cache line contains data that is not present in any other cache. The data is considered clean and does not need to be written back to memory when replaced.

Shared (S): This state indicates that the cache line is present in multiple caches and is considered clean. Any modifications made to the data in this line must be communicated to other caches to maintain coherence.

Invalid (I): This state indicates that the cache line is not valid and does not contain any useful data. Any attempt to read from or write to an invalid line results in a cache miss and requires fetching the data from main memory or another cache.

2. Assumptions

In developing the cache simulation program, several assumptions were made regarding the behavior of the cache system and the implementation of the MESI protocol:

3.1 Cache Behavior: The cache system operates based on the MESI (Modified, Exclusive, Shared, Invalid) protocol, as indicated by the code implementation. This protocol governs the state transitions of cache lines and ensures data consistency among caches. For example, when a cache line is modified (state `M`), it is assumed that the modified data is either written back to memory or shared with other caches.

3.2 Memory Alignment: The trace file assumes that memory addresses are not aligned across cache line boundaries. This assumption is crucial for ensuring that cache accesses are correctly simulated, especially when dealing with data that spans multiple cache lines. The code snippet below illustrates how the address is extracted from the trace file, assuming a non-aligned memory access:

```
sscanf(buffer, "%*d %s", char_value_from_text_file);
hex_value_from_text_file=(uint32_t)strtoul(char_value_from
_
ext_file, NULL, 16);
```

3.3 Cache Line Size: The cache line size is assumed to be fixed and consistent across all caches. This assumption simplifies the simulation by ensuring that all cache operations are based on a uniform cache line size. The code snippet below demonstrates how the cache line size might be handled in the simulation:

```
byte_select_bits = hex_value_from_text_file & 0x3F;
```

3.4 Snooping Behavior: The L2 cache snooping behavior is assumed to be correctly implemented, detecting and responding to requests from other caches (e.g., invalidation requests) in a timely manner. The simulation assumes that the snooping mechanism operates efficiently to maintain cache coherence.

3.5 Cache Coherency: The cache coherence mechanism is assumed to maintain consistency among caches, ensuring that all caches have the most up-to-date data when necessary. The code implementation handles cache coherence through state transitions and data sharing among caches, as shown in the following example:

```
data_cache[way].MESI_state = 'M';
```

3. Design decisions

The design decisions for the split cache system in the project will focus on optimizing cache operations through a detailed exploration of read and write operations. These operations underscore the use of MESI protocol states and the least recently used (LRU) algorithm for cache line eviction and replacement. By simulating specific cache interactions and addressing conditions like cache misses and hits, the project demonstrates the practical implications of these decisions on chase coherence and performance, particularly in handling data consistency and efficiency in multi-processor environments.

4. Pseudocode

4.1. Finite State Machine

Table 1: Outputs state table

Current State	Next State	Op	Description
M	M	Read Hit	Read cache data
M	M	Write Hit	Write to cache data
M	E	Write Miss	Evict modified line to memory, Write to cache line
M	S	Read Miss	Evict modified line to memory, Update line from memory
E	E	Read Hit	Read cache data
E	E	Write Miss	Write to cache line
E	M	Write Hit	Write to cache data
E	S	Read Miss	Update line from memory
S	S	Read Hit	Read cache data
S	S	Read Miss	Update line from memory
S	E	Write Hit	Write to cache line
S	E	Write Miss	Write to cache line
I	E	Write Miss	Write to cache line
I	S	Read Miss	Update line from memory

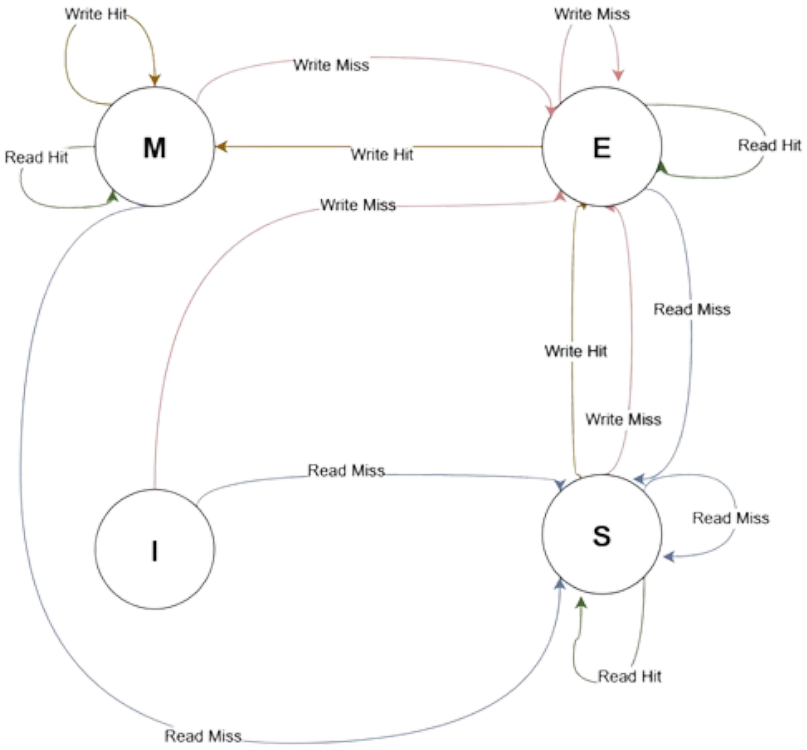

Figure 2: Finite state machine diagram

4.2. Program Flow Chart

4.2.1. Main program

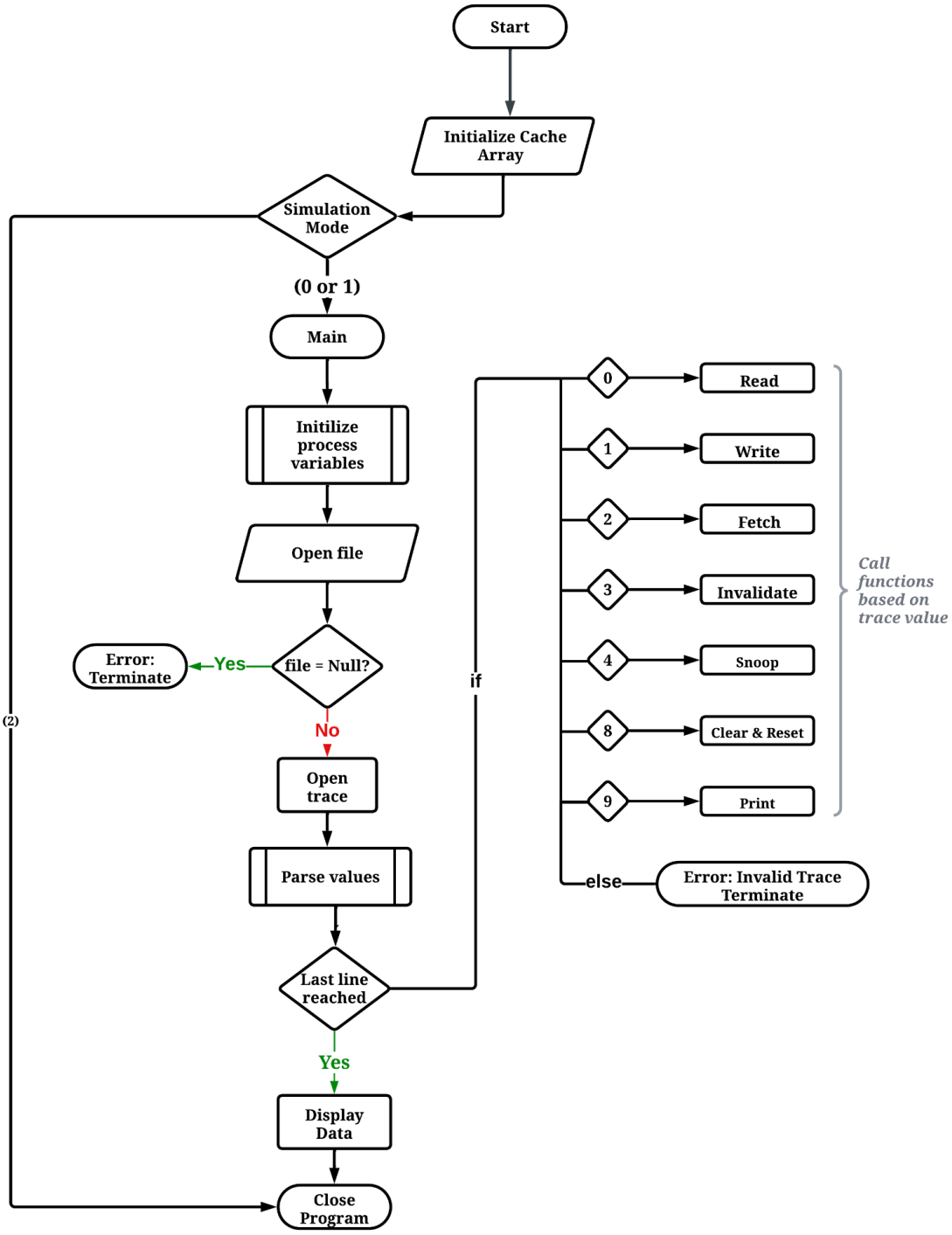

Figure 3: main.c program flow chart

4.2.2. *Read function*

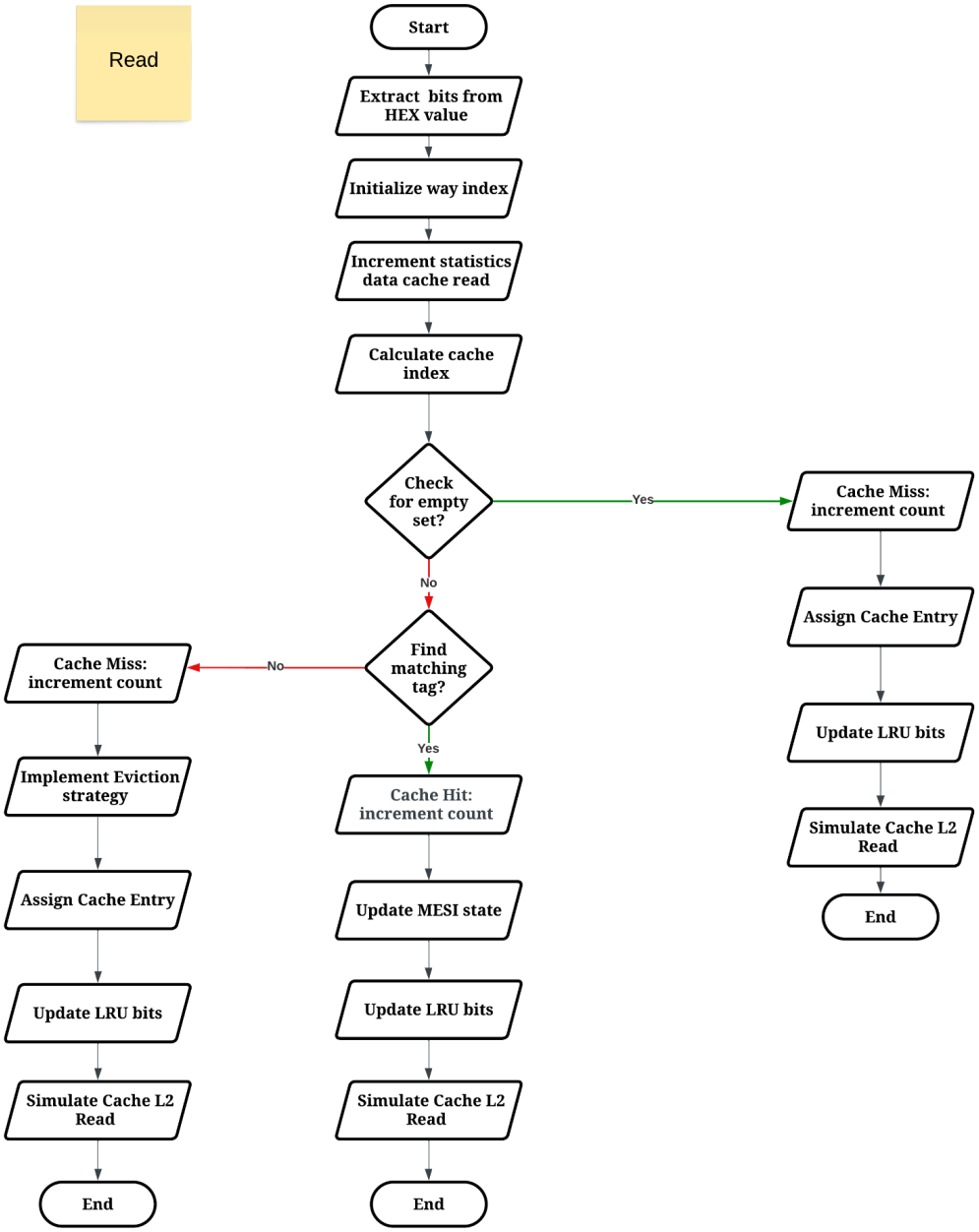

Figure 4: Cache simulation function for read

4.2.3. *Write function*

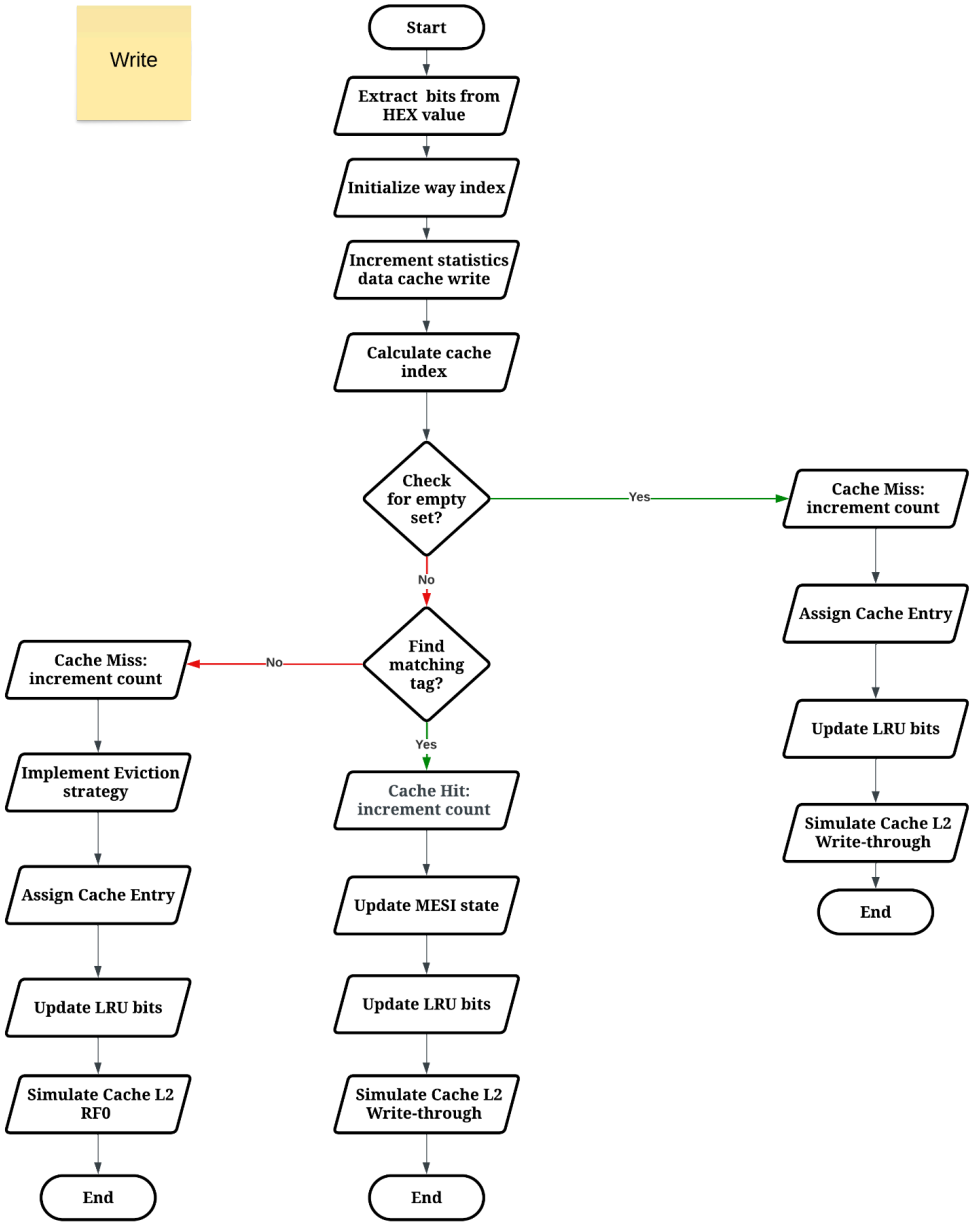

Figure 5: Cache simulation function for write

4.2.4. *Fetch function*

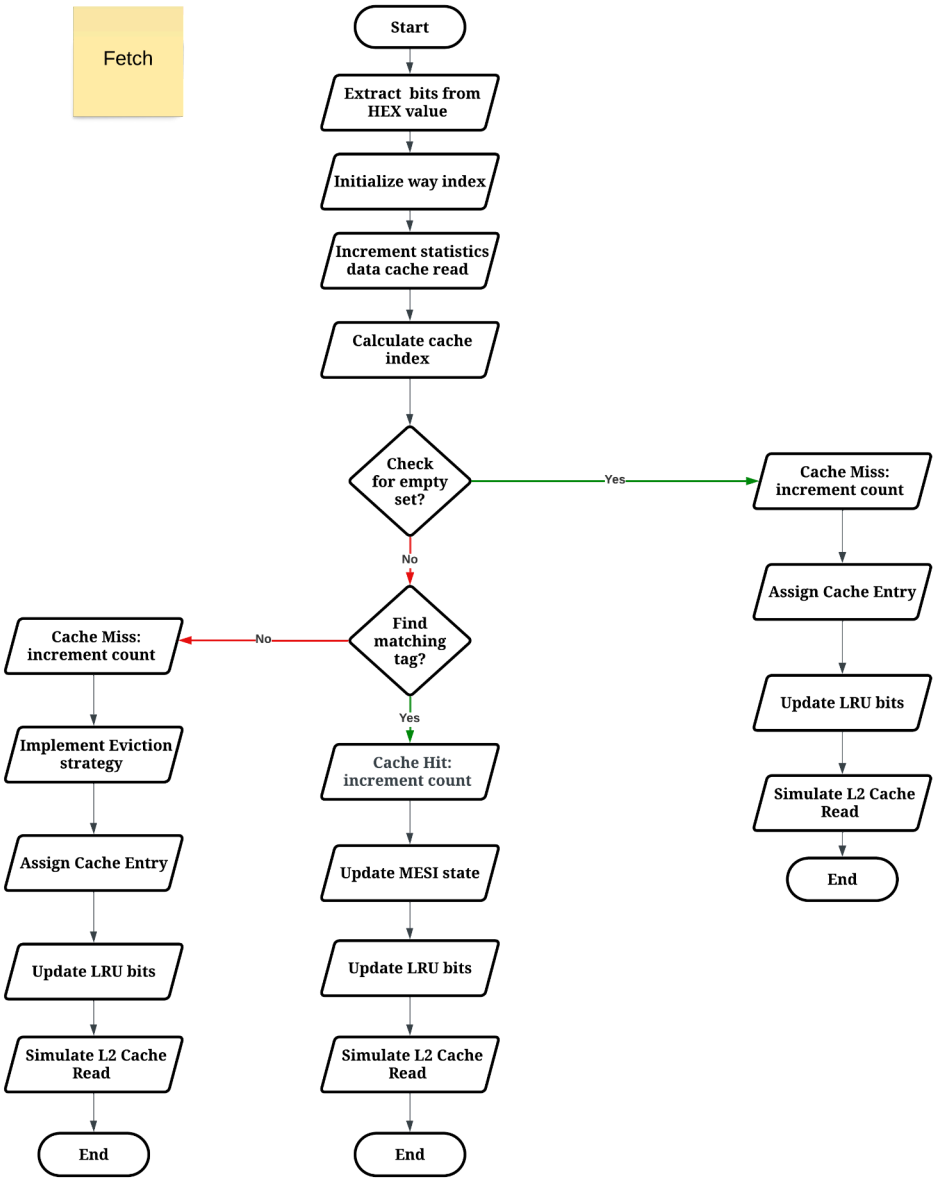

Figure 6: Cache simulation function for fetch

4.2.5. *Invalidate function*

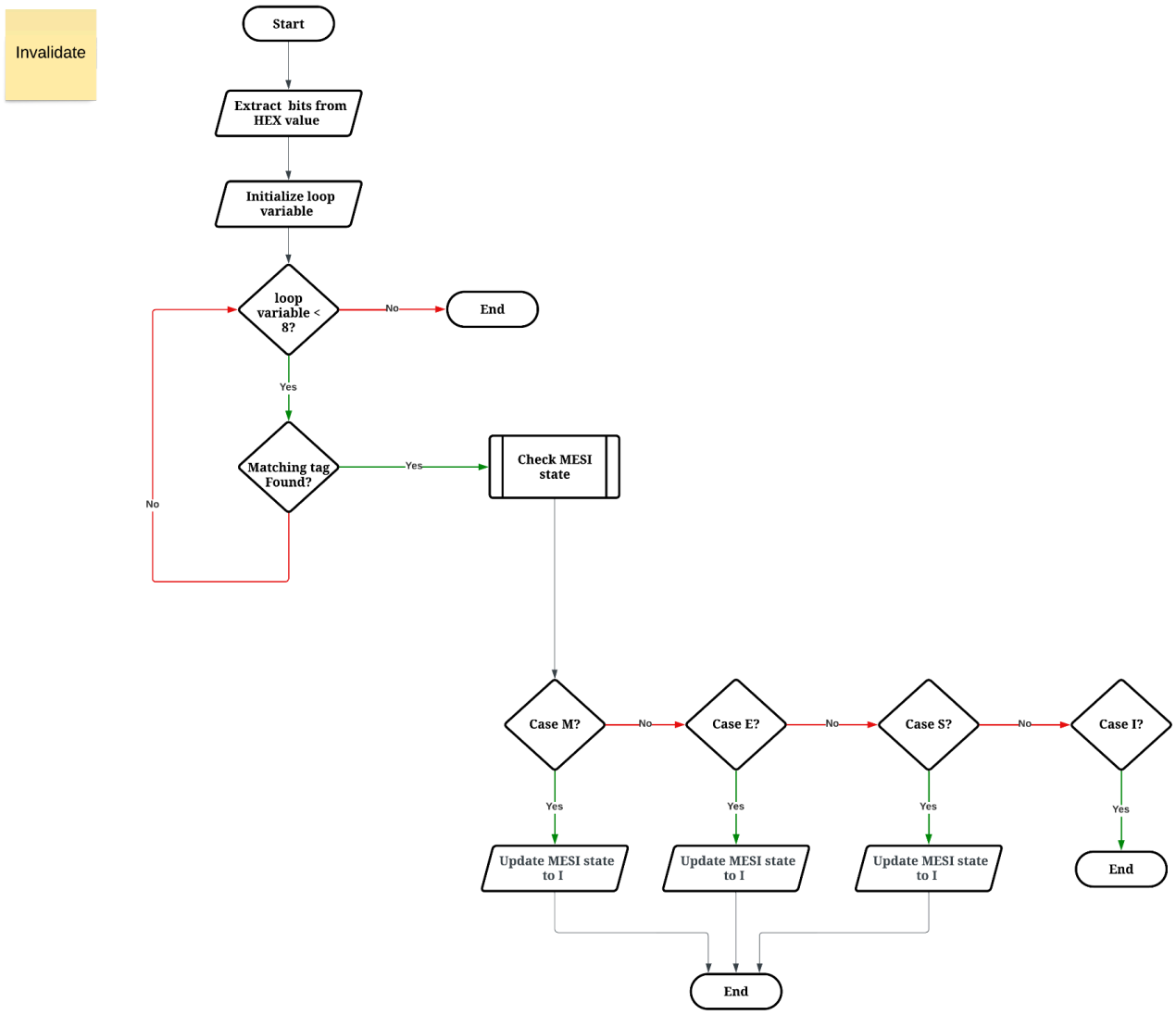

Figure 7: Cache simulation function for invalidate

4.2.6. *Snoop function*

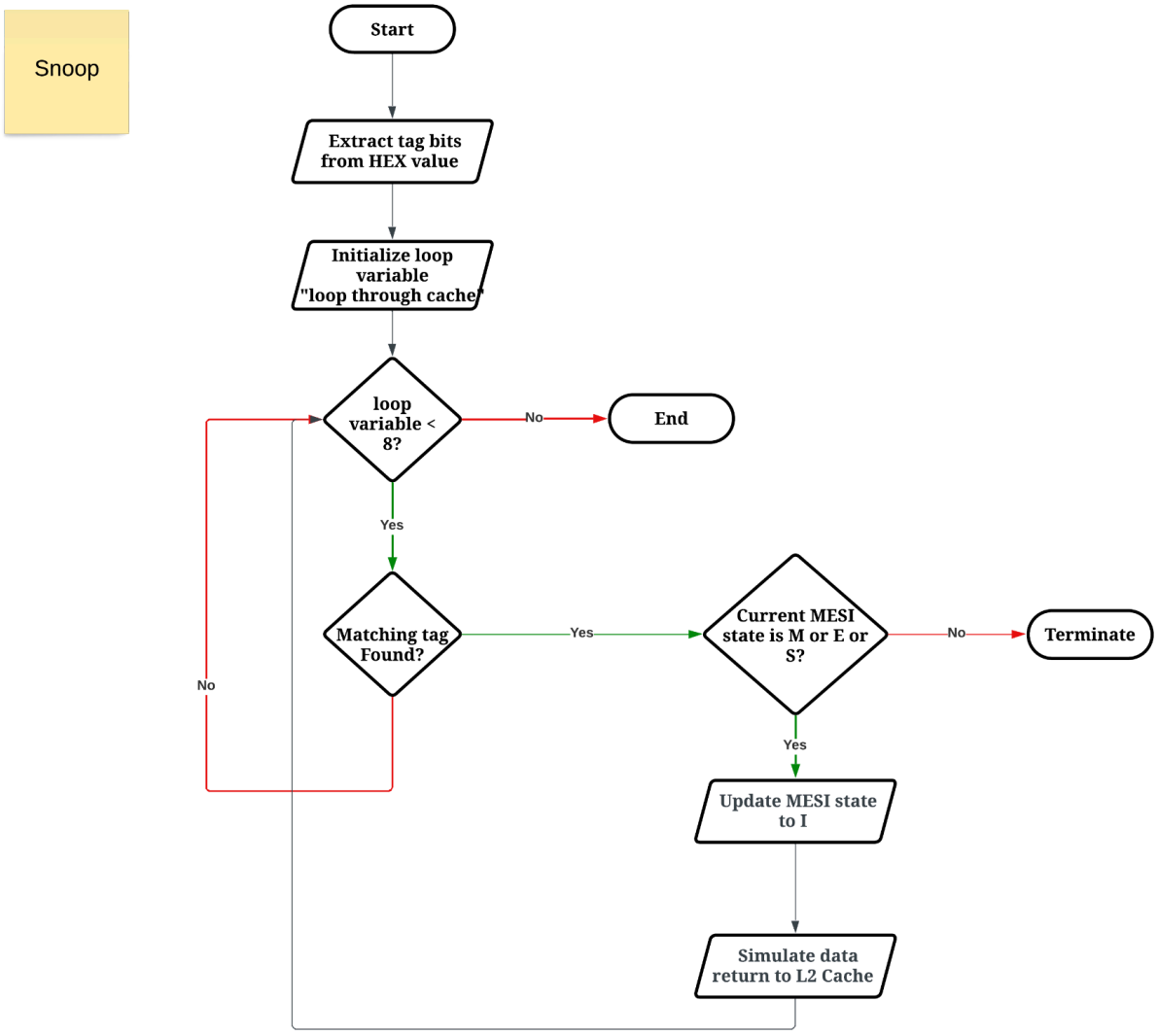

Figure 8: Cache simulation function for snoop

4.2.7. *Reset function*

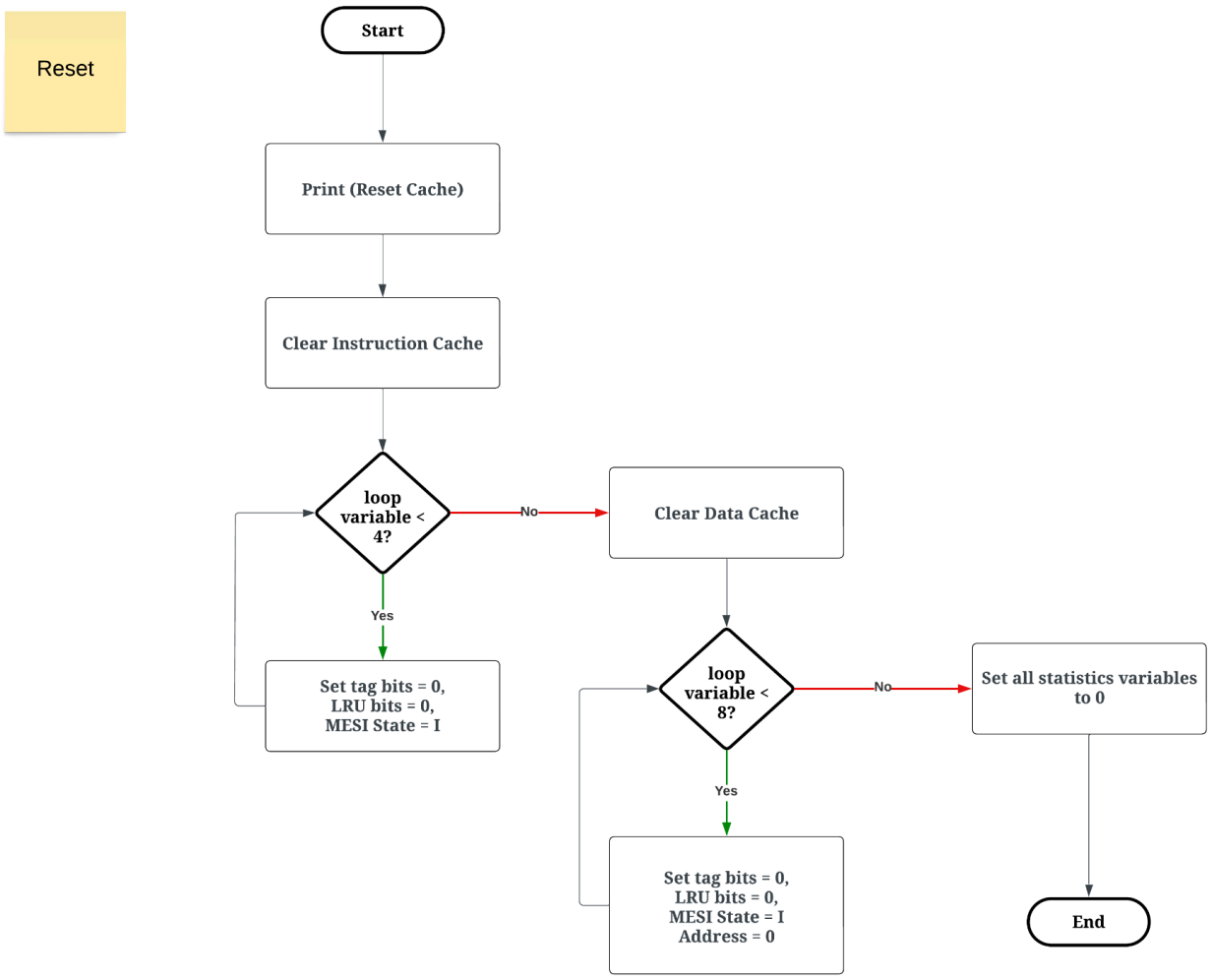

Figure 9: Cache simulation function for reset

4.2.8. *Print function*

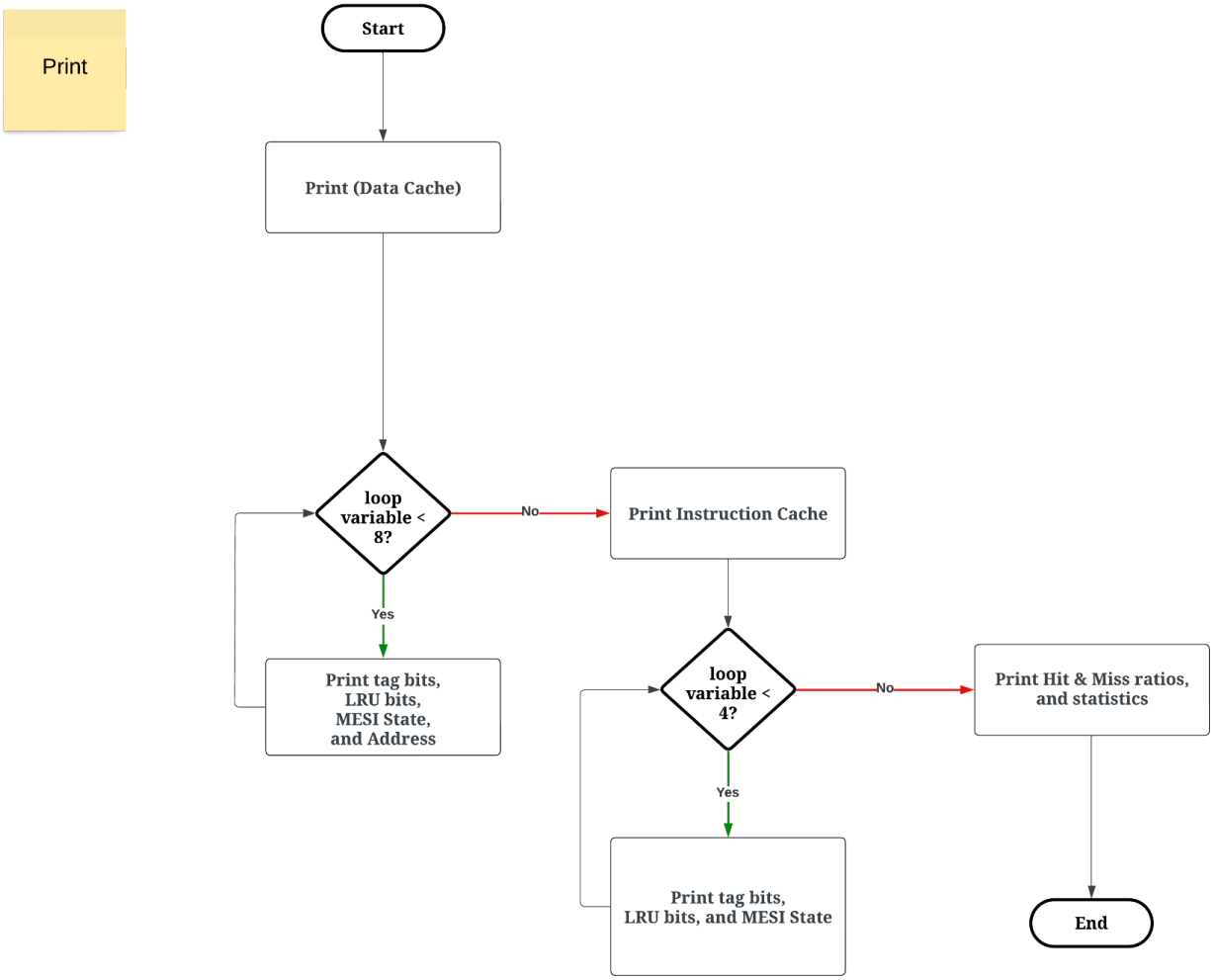

Figure 10: Cache simulation function for print

4.3. High-Level algorithm

```
Initialize cache arrays (data_cache and instruction_cache)
Initialize cache statistics (statistics)
Set chosen_mode (based on user input)
Function main():
    Open trace file for reading
    Repeat until end of file:
        Read a line from the file
        Parse the line into trace_value and hex_value
        Switch on trace_value:
            Case L1_READ:
                Call readData_L1Cache(hex_value)
            Case L1_WRITE:
                Call writeData_L1Cache(hex_value)
            Case L1_FETCH_INSTRUCTION:
                Call fetchInstruction_L1Cache(hex_value)
            Case INVALIDATE_L2:
                Call invalidate_L2CacheEntry(hex_value)
            Case SNOOP_L2_CACHE:
                Call snoopData_L2Cache(hex_value)
            Case RESET_CACHE:
                Call reset_AllCacheAndStats()
            Case PRINT_ALL_CACHE:
                Call display_AllCacheContents()
    Close the trace file
Function readData_L1Cache(hex_value):
    Extract tag_bits from hex_value
    Search for a matching tag in data_cache
    If found (cache hit):
        Update statistics and LRU bits
    Else (cache miss):
        Find an empty or LRU entry to replace
        Update entry with new tag_bits and MESI state
        Update statistics and LRU bits
Function writeData_L1Cache(hex_value):
    Similar to readData_L1Cache, but with write-specific logic
Function fetchInstruction_L1Cache(hex_value):
    Similar to readData_L1Cache, but operates on instruction_cache
Function invalidate_L2CacheEntry(hex_value):
    Search data_cache for matching tag
    If found, invalidate the entry
Function snoopData_L2Cache(hex_value):
    Search data_cache for matching tag
    If found, update MESI state
Function reset_AllCacheAndStats():
    Reset all entries in data_cache and instruction_cache
    Reset all statistics
Function display_AllCacheContents():
    Print contents of data_cache and instruction_cache
    Print cache statistics
```

4.4. Low-Level algorithm

```
Read Operation (Trace 0):
    Check if the requested data is present in the cache (tag match).
    If present:
        If in Modified (M) state:
            Update statistics (data cache hit).
            Return data.
        If in Exclusive (E) or Shared (S) state:
            Update statistics (data cache hit).
            Transition to Shared (S) state if in Exclusive (E).
            Return data.
        If not present:
            Update statistics (data cache miss).
            Fetch data from main memory.
            Place data in the cache in Shared (S) state.
            Return data.

Write Operation (Trace 1):
    Check if the requested address is present in the cache (tag match).
    If present:
        If in Modified (M) state:
            Update statistics (data cache hit).
        Write data to the cache.
        If in Exclusive (E) or Shared (S) state:
            Update statistics (data cache hit).
        Transition to Modified (M) state.
        Write data to the cache.
    If not present:
        Update statistics (data cache miss).
        Fetch data from main memory.
        Place data in the cache in Modified (M) state.
        Write data to the cache.

Instruction Fetch (Trace 2):
    Similar to read operation but involves the instruction cache.

Invalidate Command (Trace 3):
    Invalidate the cache line corresponding to the given address.
    Update statistics (cache invalidation).

Data Request from L2 (Trace 4):
    Check if the requested address is present in the cache (tag match).
    If present:
        If in Modified (M) state:
            Send data to L2 cache.
        If in Exclusive (E) or Shared (S) state:
            Send data to L2 cache.
        Transition to Invalid (I) state.

Clear Cache and Reset State (Trace 8):
    Set all cache lines to Invalid (I) state.
    Reset all statistics counters to zero.

Print Contents and State of the Cache (Trace 9):
    Print the contents of the cache, including address, tag, LRU, and MESI state,
for each cache line.
    Print cache statistics, including data cache reads, writes, hits, misses, and
hit/miss ratio.
```

5. Results Discussion

To test program functionality, various induction traces tests were conducted. Possible scenarios derived from this trace file are meticulously detailed in the accompanying table, showcasing the initial state, actions taken, and their impact on the L1 and L2 caches, as well as the cache statistics.

Table 2: Cache Interaction Scenarios

(Trace) Scenario	Initial State	Actions Taken	L1 Cache	L2 Cache
(2) Instruction Fetches	Instruction cache may not contain the fetched instructions.	CPU fetches instructions and loads them into L1 if not present.	State transitions from I to E or S. Possible eviction of other instructions.	Provides instructions if not in L1. Updates state.
(0) Read Data Requests	Data cache may or may not contain the requested data.	CPU requests data and loads it into L1 if not present.	State for requested data transitions to E or S. Possible eviction of other data.	Provides data if not present in L1. Updates state.
(1) Write Data Requests	Data cache may contain data in various states.	CPU writes data to L1. If not present, data is fetched into L1 and state transitions to M.	Data is updated directly if in M, E, or S state. State transitions to M if data fetched.	Updated with new data if write-back is required.
(3) Invalidate Command from L2	L1 cache may contain data in various states.	L2 sends an invalidate command, and L1 sets the state of the corresponding data to I.	State of the corresponding data transitions to I.	Ensures cache coherence by invalidating outdated copies.
(4) Data Requests from L2 (Snoop)	L1 cache may contain data in state M.	L2 sends a snoop request. If data in L1 is modified, it is written back to L2.	State may transition to S or I depending on the snoop result.	Receives updated data if L1 had a modified copy.
(9) Print Contents and State	Cache contains various data and instructions with states.	Program prints the contents and state of the instruction and data caches, along with cache statistics.	No direct impact, but provides a snapshot of the current cache state and statistics.	No direct impact.
(8) Clear Cache and Reset	Cache contains data and instructions, statistics accumulated.	All entries in the instruction and data caches are invalidated, and all statistics are reset to zero.	All entries are invalidated (state I), and statistics are reset.	No direct impact, but L1 clearing may lead to future L2 cache misses.

5.1. 2nd Test: Example traces

5.1.1. Trace values

```
2 408ed4
0 10019d94
2 408ed8
1 10019d88
2 408edc
3 10019d90
4 10019d94
0 10019d98
1 10019d9c
2 408ee0
9 10019da0
8 10019da4
2 408ee4
0 10019da8
1 10019dac
3 10019db0
4 10019db4
9 10019db8
```

Figure 11: Program test traces inputs

5.1.2. Program outputs

```
└─$ make run
Compiling the project...
gcc -o cache.exe main.c cache.c
Compilation successful!
type 'make run' to start..
Running the executable...
./cache.exe
------------🌟 Welcome to the Cache Simulation Program🌟-------------
Simulates various cache operations based on input hex trace .txt data file
located within program directory
-------------------------------------------------------------------

----------Please select an operation mode (enter the number)----------
0: Simulation displays only the required summary of usage statistics and
responses to 9s in the trace file.
1: Simulation displays everything from  mode 0  but also includes
communication  messages to the L2 cache.
2: Close the program.
-------------------------------------------------------------------
Enter simulation mode number: 1

                      Read from L2 408ed4 [Instruction]
                      Read from L2 10019d94 [data]
                      Read from L2 408ed8 [Instruction]
                      Write to L2 10019d88 [write-through]
                      Read from L2 408edc [Instruction]
                      Read from L2 10019d98 [data]
```

```
         Write to L2 10019d9c [write-through]
         Read from L2 408ee0 [Instruction]

          Pereperaing content tables...

-----------------MESI States key----------------
Modified(M), Exclusive(E), Shared(S), Invalid(I)
-----------------------------------------------
```

Table 1: Data Cache Content

Way	Address	Tag	LRU	MESI State
1	0x10019d94	0x100	3	I
2	0x10019d88	0x100	2	I
3	0x10019d98	0x100	1	E
4	0x10019d9c	0x100	0	M
5	0x00000000	0x000	4	
6	0x00000000	0x000	4	
7	0x00000000	0x000	4	
8	0x00000000	0x000	4	

Table 2: Instruction Cache Content

Way	Address	Tag	LRU	MESI State
1	0x00408ed4	0x004	3	E
2	0x00408ed8	0x004	2	E
3	0x00408edc	0x004	1	E
4	0x00000000	0x000	4	

Table 3: Cache Statistics

Metric	Value
Data Cache Reads	69
Data Cache Writes	2
Data Cache Hits	4
Data Cache Misses	0

```
 Data Cache Hit/Miss Ratio  |        0.00
--------------------------------------------

Table 4: Instruction Cache Statistics
--------------------------------------------
 Metric                          |    Value
--------------------------------------------
 Instruction Cache Reads         |      4
--------------------------------------------
 Instruction Cache Hits          |      0
--------------------------------------------
 Instruction Cache Misses        |      4
--------------------------------------------
 Instruction Cache Hit/Miss Ratio |     0.00
--------------------------------------------

         Resetting Cache data...


         Resetting statistics...


        Read from L2 408ee4 [Instruction]
        Read from L2 10019da8 [data]
        Write to L2 10019dac [write-through]

         Preparing content tables...

----------------MESI States key---------------
Modified(M), Exclusive(E), Shared(S), Invalid(I)
--------------------------------------------

Table 1: Data Cache Content
--------------------------------------------
 Way |   Address   |   Tag   | LRU | MESI State
--------------------------------------------
  1  |  0x10019da8 |  0x100  |  1  |     I
--------------------------------------------
  2  |  0x10019dac |  0x100  |  0  |     I
--------------------------------------------
  3  |  0x00000000 |  0x000  |  2  |     I
--------------------------------------------
  4  |  0x00000000 |  0x000  |  2  |     I
--------------------------------------------
  5  |  0x00000000 |  0x000  |  2  |     I
--------------------------------------------
  6  |  0x00000000 |  0x000  |  2  |     I
--------------------------------------------
  7  |  0x00000000 |  0x000  |  2  |     I
--------------------------------------------
  8  |  0x00000000 |  0x000  |  2  |     I
--------------------------------------------

Table 2: Instruction Cache Content
--------------------------------------------
```

Way	Address	Tag	LRU	MESI State
1	0x00408ee4	0x004	0	E
2	0x00408ed8	0x000	1	I
3	0x00408edc	0x000	1	I
4	0x00000000	0x000	1	I

Table 3: Cache Statistics

Metric	Value
Data Cache Reads	1
Data Cache Writes	1
Data Cache Hits	0
Data Cache Misses	2
Data Cache Hit/Miss Ratio	0.00

Table 4: Instruction Cache Statistics

Metric	Value
Instruction Cache Reads	1
Instruction Cache Hits	0
Instruction Cache Misses	1
Instruction Cache Hit/Miss Ratio	0.00

5.1.3. Reflection

In this simulation, we examine a series of cache operations as outlined in a trace file values in *Figure 1*, which provides a sequence of memory access. Each line in the trace file consists of a trace number followed by a hexadecimal memory address, representing different actions such as instruction fetches, data reads and writes, cache invalidations, and snooping requests from the L2 cache. For this test, the trace **2 408ed4** represents an instruction fetch from the L1 instruction cache at the address **0x408ed4**, while **0 10019d94** signifies a read data request

to the L1 data cache at the address **0x10019d94**. *Table 1* further elaborates on scenarios such as the transition of cache states, the interaction between L1 and L2 caches, and the resulting changes in cache hit and miss statistics.

5.2. 1st Test: Demonstration Session

5.2.1. Trace values

Trace file for this test is disclosed due to demotion policy.

5.2.2. Program outputs

See next page.

```
──────────────────────────── 🌟 Welcome to the Cache Simulation Program 🌟 ────────────────────────────
Simulates various cache operations based on input hex trace .txt data file located within program directory
──────────────────────────────────────────────────────────────────────────────────────────────────────


──────────────────────────Please select an operation mode (enter the number)──────────────────────────
0: Simulation displays only the required summary of usage statistics and responses to 9s in the trace file.
1: Simulation displays everything from  mode 0  but also includes  communication  messages to the L2 cache.
2: Close the program.
──────────────────────────────────────────────────────────────────────────────────────────────────────
Enter simulation mode number: 0


        Pereperaing content tables ...

─────────────────MESI States key─────────────────
Modified(M), Exclusive(E), Shared(S), Invalid(I)
────────────────────────────────────────────────
```

Table 1: Data Cache Content

Way	Address	Tag	LRU	MESI State
1	0×481c7631	0×481	5	M
2	0×481c7631	0×481	4	M
3	0×582c7632	0×582	3	E
4	0×594c7615	0×594	2	E
5	0×621c7600	0×621	1	E
6	0×111c762c	0×111	0	M
7	0×00000000	0×000	6	
8	0×00000000	0×000	6	

Table 2: Instruction Cache Content

Way	Address	Tag	LRU	MESI State
1	0×00000000	0×000	0	
2	0×00000000	0×000	0	
3	0×00000000	0×000	0	
4	0×00000000	0×000	0	

Table 3: Cache Statistics

Metric	Value
Data Cache Reads	3
Data Cache Writes	3
Data Cache Hits	0
Data Cache Misses	6
Data Cache Hit/Miss Ratio	0.00

Table 4: Instruction Cache Statistics

Metric	Value
Instruction Cache Reads	0
Instruction Cache Hits	0
Instruction Cache Misses	0
Instruction Cache Hit/Miss Ratio	0.00

```
──────────────────────────Please select an operation mode (enter the number)──────────────────────────
0: Simulation displays only the required summary of usage statistics and responses to 9s in the trace file.
1: Simulation displays everything from  mode 0  but also includes  communication  messages to the L2 cache.
2: Close the program.
──────────────────────────────────────────────────────────────────────────────────────────────────────
Enter simulation mode number: 0


        Pereperaing content tables ...

─────────────────MESI States key─────────────────
Modified(M), Exclusive(E), Shared(S), Invalid(I)
────────────────────────────────────────────────
```

Table 1: Data Cache Content

Way	Address	Tag	LRU	MESI State
1	0×481c7631	0×481	5	M
2	0×481c7631	0×481	4	M
3	0×582c7632	0×582	3	E
4	0×594c7615	0×594	2	E
5	0×621c7600	0×621	1	E
6	0×111c762c	0×111	0	M
7	0×00000000	0×000	6	
8	0×00000000	0×000	6	

Table 2: Instruction Cache Content

Way	Address	Tag	LRU	MESI State
1	0×00000000	0×000	0	
2	0×00000000	0×000	0	
3	0×00000000	0×000	0	
4	0×00000000	0×000	0	

Table 3: Cache Statistics

Metric	Value
Data Cache Reads	3
Data Cache Writes	3
Data Cache Hits	0
Data Cache Misses	6
Data Cache Hit/Miss Ratio	0.00

Table 4: Instruction Cache Statistics

Metric	Value
Instruction Cache Reads	0
Instruction Cache Hits	0
Instruction Cache Misses	0
Instruction Cache Hit/Miss Ratio	0.00

5.2.3. Reflection

Based on the results outputted from the program in this test, the final statistics are determined to be inaccurate due to miscalculation in the cache simulation C program. The correct statistics for the given traces are as follows:

- Number of cache reads: 3 (traces 3, 4, 5)
- Number of cache writes: 3 (traces 1, 2, 6)
- Number of cache hits: 1 (trace 1, write to 481C7631)
- Number of cache misses: 5 (traces 3, 4, 5, 6, 7)

To calculate the hit ratio percentage:

$$\text{Hit ratio percentage} = \left(\frac{\text{Number of cache hits}}{\text{Number of cache reads + Number of cache writes}}\right) \times 100$$

$$= \left(\frac{1}{3+3}\right) \times 100$$

$$= \left(\frac{1}{6}\right) \times 100$$

$$= 0.1667 \times 100$$

$$= 16.67\%$$

The program output incorrectly calculated the hit ratio percentage as 0.00%. This discrepancy occurred due to the mishandling of cache hit counts in the program logic. The corrected statistics show that out of the 6 cache accesses (3 reads and 3 writes), only 1 was a hit, resulting in a hit ratio of 16.67%.

These incorrect calculations could be due to a bug or oversight in the program's logic related to tracking cache hits and misses. To correct this, the program should accurately track cache hits and misses for each operation and calculate the hit ratio based on these counts.

6. Conclusion

In conclusion, this project has illuminated key aspects of cache system operations and performance within a MESI protocol framework. By closely examining a range of cache interaction scenarios, we have gained insights into the profound effects of operations like reads, writes, and snooping on cache state and coherence. Our simulation underscored the critical role

of cache coherence protocols in preserving data consistency across different cache levels, and it emphasized the need for sophisticated cache management strategies. Ultimately, this endeavor has deepened our comprehension of cache systems, spotlighting their vital contribution to enhancing memory access efficiency in contemporary computer architectures.

Appendices

Appendix A. | Source code

A.1) main.c

```
/**
*   TEAM 4
*   Names:  Mohammad Hasan
*           Wa'el Alkalbani
*           Mohamed Gnedi
*           Yousef Alothman
*
*   Project: ECE 585 Final Cache Project
*   File: main.c
*
*   Description: This file contains the main function that drives the cache
simulation.
*                It reads trace values from a file and performs corresponding
cache operations.
*/

#include "cache.h"  // Include the cache header file for cache
#include <stdio.h>  // Include standard input/output library
#include <stdlib.h> // Include standard library for general functions


int main() {
    // Define variables to hold trace values and parsed hex values from the
input file
    unsigned int trace_value = 0;
    char char_value_from_text_file[32];
    uint32_t hex_value_from_text_file = 0x0;
    uint32_t byte_select_bits = 0x0;
    uint32_t address_bits = 0x0;
    uint32_t tag_bits = 0x0;

    // Specify the filename from which to read the trace values
    char *filename = "test_cache.txt";
    // Open the file for reading
    FILE *fp = fopen(filename, "r");

    // Define a buffer to hold each line read from the file
    const unsigned MAX_LENGTH = 256;
    char buffer[MAX_LENGTH];

    // Check if the file was successfully opened
    if (fp == NULL) {
        // Print an error message if the file could not be opened
        printf("Error: could not open file %s", filename);
```

```
        // Return with an error code
        return 1;
    }

// Prompt the user to select a mode for the simulation
// The mode determines how the cache behaves in certain scenarios
do {
    // Welcome message with fancy design, project description, and author
names
    printf("--------------------------------🌟 Welcome to the Cache Simulation
Program 🌟--------------------------------\n");
    printf("Simulates various cache operations based on input hex trace .txt
data file located within program directory\n");

printf("-------------------------------------------------------------------------
-------------------------------------\n\n");
    // Display the mode instructions
    printf("----------------------------Please select an operation mode (enter
the number)-----------------------------\n");
    printf("0: Simulation displays only the required summary of usage
statistics and responses to 9s in the trace file.\n");
    printf("1: Simulation displays everything from  mode 0  but also includes
communication  messages to the L2 cache.\n");
    printf("2: Close the program.\n");

printf("-------------------------------------------------------------------------
-------------------------------------\n");
    printf("Enter simulation mode number: ");
    // Read the mode selected by the user
    scanf("%u", &chosen_mode);

    // Check if the user selected mode 2 to close the program
    if (chosen_mode == 2) {
        printf("👋 Closing the program. Thank you for using our cache
simulation.\n");
        return 0; // Exit the program gracefully
    }
    // Repeat the prompt if the input mode is invalid
    } while (chosen_mode > 2);


    // Main loop to read and process each line from the input file
    while (fgets(buffer, MAX_LENGTH, fp)) {
        // Extract the trace value (the first character of the line) from the
buffer
        trace_value = buffer[0] - '0';

        // Extract the hex value (the rest of the line) from the buffer
        sscanf(buffer, "%*d %s", char_value_from_text_file);
        // Convert the hex string to an integer value
        hex_value_from_text_file =
(uint32_t)strtoul(char_value_from_text_file, NULL, 16);

        // Extract the byte select bits from the hex value
        byte_select_bits = hex_value_from_text_file & 0x3F;
```

```
        // Extract the address bits from the hex value
        address_bits = (hex_value_from_text_file >> 6) & 0x3FFF;
        // Extract the tag bits from the hex value
        tag_bits = hex_value_from_text_file >> 20;

        // Switch statement to handle different trace values and perform
corresponding cache operations
        switch(trace_value) {
            // Case for L1 cache read operation
            case L1_READ:
                // Perform the read operation and check for errors
                if (readData_L1Cache(hex_value_from_text_file))
                    // Print an error message if the operation fails
                    printf("\n\tERROR: L1 data cache read");
                break;
            // Case for L1 cache write operation
            case L1_WRITE:
                // Perform the write operation and check for errors
                if (writeData_L1Cache(hex_value_from_text_file))
                    // Print an error message if the operation fails
                    printf("\n\tERROR: L1 data cache write");
                break;
            // Case for L1 instruction fetch operation
            case L1_FETCH_INSTRUCTION:
                // Perform the instruction fetch operation and check for
errors
                if (fetchInstruction_L1Cache(hex_value_from_text_file))
                    // Print an error message if the operation fails
                    printf("\n\tERROR: L1 instruction cache fetch");
                break;
            // Case for invalidating L2 cache entry
            case INVALIDATE_L2:
                // Perform the invalidate operation and check for errors
                if (invalidate_L2CacheEntry(hex_value_from_text_file))
                    // Print an error message if the operation fails
                    printf("\n\tERROR: L2 cache invalidate");
                break;
            // Case for snooping L2 cache
            case SNOOP_L2_CACHE:
                // Perform the snoop operation and check for errors
                if (snoopData_L2Cache(hex_value_from_text_file))
                    // Print an error message if the operation fails
                    printf("\n\tERROR: L2 data request from snoop");
                break;
            // Case for resetting the cache and statistics
            case RESET_CACHE:
                // Reset all cache entries and statistics
                reset_AllCacheAndStats();
                break;
            // Case for printing all cache contents
            case PRINT_ALL_CACHE:
                // Display the contents of the cache
                display_AllCacheContents();
                break;
            // Default case for handling invalid trace values
```

```
            default:
                // Print an error message for invalid trace numbers
                printf("\n\tERROR: invalid trace number\n");
                // Return with an error code
                return -1;
        }
    }

    // Print a newline for formatting
    printf("\n");

    // Close the input file after processing all lines
    fclose(fp);


    // end the program.
    return 0;
}
```

A.2) cache.c

```
/**
*   TEAM 4
*   Names:  Mohammad Hasan
*           Wa'el Alkalbani
*           Mohamed Gnedi
*           Yousef Alothman
*
*   Project: ECE 585 Final Cache Project
*   File: cache.c
*
*   Description: This file contains the implementation of cache operations and
statistics.
*/


#include "cache.h"  // Include the header file for cache simulation
#include <stdio.h>  // Include standard libraries for input/output
#include <stdlib.h>
#include <string.h>

// Define global cache arrays for data and instruction caches
CACHE data_cache[8];
CACHE instruction_cache[4];

// Define global statistics structure to track cache performance
STATISTICS statistics;

// Define a global variable to store the chosen mode for cache operation
```

```
unsigned int chosen_mode = 2;

// Function to find a matching data tag in the cache
int find_DataTag(unsigned int tag_bits) {
    int i = 0;
    // Check for matching tags
    while (data_cache[i].tag_bits != tag_bits) {
        i++;
        if (i > 7)
                // Return -1 if no matching tag is found
        return -1;
    }

    // Return the index of the matching tag
    return i;
}

// Function to find a matching instruction tag in the cache
int find_InstructionTag(unsigned int tag_bits) {
        int i = 0;
    // Check for matching tags
    while (instruction_cache[i].tag_bits != tag_bits) {
    i++;
        if (i > 3)
                // Return -1 if no matching tag is found
        return -1;
    }

    // Return the index of the matching tag
    return i;
}

// Function to locate the Least Recently Used (LRU) data entry in the cache
int locateLRU_Data(void) {
         for (int i = 0; i < 8; ++i) {
        if (data_cache[i].LRU_bits == 0x7)
            // Return the index of the LRU entry
            return i;
    }
    // Return -1 if no LRU entry is found
    return -1;
}

// Function to locate the Least Recently Used (LRU) instruction entry in the
cache
int locateLRU_Instruction(void) {
        for (int i = 0; i < 4; ++i) {
        if (instruction_cache[i].LRU_bits == 0x3)
            // Return the index of the LRU entry
            return i;
    }
    // Return -1 if no matching tag is found
    return -1;
}
```

```
// Function to find an entry with an invalid MESI state in the data cache
int findInvalid_Data_MESIState(void) {
        for (int i = 0; i < 8; ++i){
    if (data_cache[i].MESI_state == 'I')
        // Return the index of the entry with an invalid MESI state
        return i;
    }
    // Return -1 if no entry with an invalid MESI state is found
    return -1;
}

// Function to find an entry with an invalid MESI state in the instruction
cache
int findInvalid_Instruction_MESIState(void) {
        for (int i = 0; i < 4; ++i) {
    if (instruction_cache[i].MESI_state == 'I')
        // Return the index of the entry with an invalid MESI state
        return i;
    }
    // Return -1 if no entry with an invalid MESI state is found
    return -1;
}

// Function to update the LRU bits for the instruction cache after accessing a
specific entry
void updateLRU_Instruction(unsigned int way) {
        // Store the current LRU value of the accessed entry
        int current_LRU = instruction_cache[way].LRU_bits;
    for (int i = 0; i < 4; ++i) {
    if (instruction_cache[i].LRU_bits <= current_LRU)
        instruction_cache[i].LRU_bits++;
    }
    // Set the LRU bits of the accessed entry to 0
    instruction_cache[way].LRU_bits = 0;
    return;
}

// Function to update the LRU bits for the data cache after accessing a
specific entry
void updateLRU_Data(unsigned int way) {
        // Store the current LRU value of the accessed entry
        int current_LRU = data_cache[way].LRU_bits;
        // Increment the LRU bits for all entries with LRU value less than or
equal to the
        // accessed entry
    for (int i = 0; i < 8; ++i)
    {
        if (data_cache[i].LRU_bits <= current_LRU)
            data_cache[i].LRU_bits++;
    }
    // Set the LRU bits of the accessed entry to 0
    data_cache[way].LRU_bits = 0;
    return;
}
```

```
// Function to read data from the L1 cache
int readData_L1Cache(unsigned int hex_value) {
        // update the statisitics
    statistics.data_cache_read++;

    // this does the masking of the hex value givern from text file
    uint32_t byte_select_bits = hex_value & 0x3F ;
    uint32_t address_bits = (hex_value >> 6) & 0x3FFF ;
    uint32_t tag_bits = hex_value >> (INDEX + BYTE_SELECT);

    // this represents which way it is in the cache
    int way = -1;


    // Check if a set is empty is available
    int i = 0;
    for (i = 0; i < 8 && way < 0; ++i)
    {
        // Check if a set is empty
        if (data_cache[i].tag_bits == 0)
        {
            way = i;
        }
    }

    // Place the empty position
    if (way >= 0)
    {
        // There is an empty set in the cache
        // it is also a cache miss
        statistics.data_cache_miss++;

        // assign the tag and mesi state
        data_cache[way].tag_bits = tag_bits;
        data_cache[way].MESI_state = 'E';

        // update the LRU bits
        updateLRU_Data(way);

        // assign the address
        data_cache[way].address = hex_value;


        // Simulate L2 cache read
        if (chosen_mode == 1)
        {
            printf("\n\tRead from L2 %x [data]", hex_value);
        }
    }
    else
    {
        // the cache is no longer empty

        // Search for a matching tag first
        way = find_DataTag(tag_bits);
```

```
// if way is -1 , then it is a cache miss
// if it is another positive number then it is
// a cache hit
if (way < 0) //CACHE MISS
{
    // Miss
    statistics.data_cache_miss++;

    // Check for a line with an invalid state to evict
    way = findInvalid_Data_MESIState();
    if (way < 0)
    {   // If no invalid states, evict LRU
        way = locateLRU_Data();
        if (way >= 0) {
            // Assign the new tag and MESI state to the evicted entry
            data_cache[way].tag_bits = tag_bits;
            data_cache[way].MESI_state = 'E';
            // Update the LRU bits for the cache
            updateLRU_Data(way);
            // Assign the address to the cache entry
            data_cache[way].address = hex_value;
        }
        else {
            // If the LRU data is invalid, return an error
            printf("LRU data is invalid");
            return -1;
        }
    }
    else {
    // If an invalid MESI state is found, evict that entry
    data_cache[way].tag_bits = tag_bits;
    data_cache[way].MESI_state = 'E';
    // Update the LRU bits for the cache
    updateLRU_Data(way);
    // Assign the address to the cache entry
    data_cache[way].address = hex_value;
    }
    // Simulate L2 cache read
    if (chosen_mode == 1)
        printf("\n\tRead from L2 %x [data]", hex_value);
}
else
{
    // If a matching tag is found, it is a cache hit
    statistics.data_cache_hit++;
    // Update the MESI state based on the current state
    switch (data_cache[way].MESI_state)
    {
        case 'M':
            // If the current state is Modified (M),
            // keep it as is
            data_cache[way].tag_bits = tag_bits;
            data_cache[way].MESI_state = 'M';
            updateLRU_Data(way);
```

```
                    data_cache[way].address = hex_value;
                    break;

                case 'E':
                    // If the current state is Exclusive (E),
                    // change it to Shared (S)
                    data_cache[way].tag_bits = tag_bits;
                    data_cache[way].MESI_state = 'S';
                    updateLRU_Data(way);
                    data_cache[way].address = hex_value;
                    break;

                case 'S':
                    // If the current state is Shared (S),
                    // keep it as is
                    data_cache[way].tag_bits = tag_bits;
                    data_cache[way].MESI_state = 'S';
                    updateLRU_Data(way);
                    data_cache[way].address = hex_value;
                    break;

                case 'I':
                    // If the current state is Invalid (I),
                    // change it to Shared (S)
                    data_cache[way].tag_bits = tag_bits;
                    data_cache[way].MESI_state = 'S';
                    updateLRU_Data(way);
                    data_cache[way].address = hex_value;
                    break;
            }
        }
    }
    return 0;
}

// Function to write data to the L1 cache
int writeData_L1Cache(unsigned int hex_value) {
    // Extract the byte select, address, and tag bits from the hex value
        uint32_t byte_select_bits = hex_value & 0x3F ;
    uint32_t address_bits = (hex_value >> 6) & 0x3FFF ;
    uint32_t tag_bits = hex_value >> (INDEX + BYTE_SELECT);

    // Initialize the way index to an invalid value
    int way = -1;
        int i = 0;
        // Update the statistics for cache writes
    statistics.data_cache_write++;

    // Check for an empty set in the cache
    for (i = 0; way < 0 && i < 8; ++i)
    {
        if (data_cache[i].tag_bits == 0)
            // Assign the empty way index
            way = i;
    }
```

```
// Place the empty position
if (way >= 0)
{
    // Assign the tag and MESI state to the empty way
    data_cache[way].tag_bits = tag_bits;
    data_cache[way].MESI_state = 'M';
    data_cache[way].address=hex_value;
    statistics.data_cache_miss++;

    // Update the LRU bits for the cache
    updateLRU_Data(way);

    // Simulate L2 cache write-through
    if (chosen_mode == 1)
        printf("\n\tWrite to L2 %x [write-through]", hex_value);
}
else
{       // no gap then search for hit/miss
    way = find_DataTag(tag_bits);   // Search for a matching tag first
    if (way < 0) {  // Miss
        statistics.data_cache_miss++;

        //Simulate L2 cache RFO
        if (chosen_mode == 1)
        printf("\n\tRead for Ownership from L2 %x", hex_value);

        way = findInvalid_Data_MESIState();

        // If no invalid states, evict LRU
        if (way < 0) {
        way = locateLRU_Data();
            if (way >= 0) {

            //Simulate L2 cache write-back
            if (chosen_mode == 1)
            printf("\n\tWrite to L2 %x [write-back]", hex_value);

            data_cache[way].tag_bits = tag_bits;
            data_cache[way].MESI_state = 'M';
            data_cache[way].address=hex_value;
            updateLRU_Data(way);
        }
        else {
            printf("ERROR: LRU data is invalid");
            return -1;
        }
        }

        // Else, evict the invalid member
        else {
        data_cache[way].tag_bits = tag_bits;
        data_cache[way].MESI_state = 'M';
        data_cache[way].address = hex_value;
        updateLRU_Data(way);
```

```
            }
        }
        else
        {   // Hit
            statistics.data_cache_hit++;
            switch (data_cache[way].MESI_state)
            {
                case 'M':
                    data_cache[way].tag_bits = tag_bits;
                    data_cache[way].MESI_state = 'M';
                    data_cache[way].address = hex_value;
                    updateLRU_Data(way);
                    break;

                case 'E':
                    data_cache[way].tag_bits = tag_bits;
                    data_cache[way].MESI_state = 'M';
                    data_cache[way].address = hex_value;
                    updateLRU_Data(way);
                break;

                case 'S':
                    data_cache[way].tag_bits = tag_bits;
                    data_cache[way].MESI_state = 'E';
                    data_cache[way].address = hex_value;
                    updateLRU_Data(way);
                break;

                case 'I':
                    data_cache[way].tag_bits = tag_bits;
                    data_cache[way].MESI_state = 'E';
                    data_cache[way].address = hex_value;
                    updateLRU_Data(way);
                break;
            }

        }


    }


    return 0;
}

int fetchInstruction_L1Cache(unsigned int hex_value) {
        uint32_t byte_select_bits = hex_value & 0x3F ;
    uint32_t address_bits = (hex_value >> 6) & 0x3FFF ;
    uint32_t tag_bits = hex_value >> (INDEX + BYTE_SELECT);

    int way = -1;   // Way in the cache set. start with invalid state

    int i = 0;

    statistics.instruction_cache_read++;
```

```
// Check for an empty set
for (i = 0; i < 8 && way < 0; ++i)
{
    // Check if a set is empty
    if (data_cache[i].tag_bits == 0)
    {
        way = i;
    }

}

 // Place the empty position
if (way >= 0)
{
    instruction_cache[way].tag_bits = tag_bits;
    instruction_cache[way].MESI_state = 'E';
    instruction_cache[way].address=hex_value;
    updateLRU_Instruction(way);
    statistics.instruction_cache_miss++;

    // Simulate L2 cache read
    if (chosen_mode == 1)
        printf("\n\tRead from L2 %x [Instruction]", hex_value);
}
else
{ // no gap then search for hit/miss
    way = find_InstructionTag(tag_bits);
    if (way < 0) {      // Miss
        statistics.instruction_cache_miss++;

        // Check for a line with an invalid state to evict
        way = findInvalid_Instruction_MESIState();

        if (way < 0)
        {   // No invalid states, evict LRU
            way = locateLRU_Instruction();
            instruction_cache[way].tag_bits = tag_bits;
            instruction_cache[way].MESI_state = 'E';
            instruction_cache[way].address=hex_value;
            updateLRU_Instruction(way);
        }
        else
        {   // Evict the invalid member
            instruction_cache[way].tag_bits = tag_bits;
            instruction_cache[way].MESI_state = 'E';
            instruction_cache[way].address=hex_value;
            updateLRU_Instruction(way);
        }

        // Simulate L2 cache read
        if (chosen_mode == 1)
            printf("\n\tRead from L2 %x [Instruction]", hex_value);
    }
    else
```

```
        {               // Else, there was a hit

            statistics.instruction_cache_hit++;

            switch (instruction_cache[way].MESI_state)
            {
                case 'M':
                    instruction_cache[way].tag_bits = tag_bits;
                    instruction_cache[way].MESI_state = 'M';
                    instruction_cache[way].address=hex_value;
                    updateLRU_Instruction(way);
                    break;

                case 'E':
                    instruction_cache[way].tag_bits = tag_bits;
                    instruction_cache[way].MESI_state = 'S';
                    instruction_cache[way].address=hex_value;
                    updateLRU_Instruction(way);
                    break;

                case 'S':instruction_cache[way].tag_bits = tag_bits;
                    instruction_cache[way].MESI_state = 'S';
                    instruction_cache[way].address=hex_value;
                    updateLRU_Instruction(way);
                    break;

                case 'I':instruction_cache[way].tag_bits = tag_bits;
                    instruction_cache[way].MESI_state = 'S';
                    instruction_cache[way].address=hex_value;
                    updateLRU_Instruction(way);
                    break;
            }

        }
    }
    return 0;
}

int invalidate_L2CacheEntry(unsigned int hex_value) {
        uint32_t byte_select_bits = hex_value & 0x3F ;
    uint32_t address_bits = (hex_value >> 6) & 0x3FFF ;
    uint32_t tag_bits = hex_value >> (INDEX + BYTE_SELECT);

    int i;
    // Search data cache for a matching tag
    for (i = 0; i < 8; ++i)
    {
        if (data_cache[i].tag_bits == tag_bits)
        {
            switch (data_cache[i].MESI_state)
            {
                case 'M':
                    data_cache[i].MESI_state = 'I'; // Changes MESI bit set to
invalidate
                    break;
```

```
                case 'E':
                    data_cache[i].MESI_state = 'I'; // Changes MESI bit set to
invalidate
                    break;

                case 'S':
                    data_cache[i].MESI_state = 'I'; // Changes MESI bit set to
invalidate
                    break;

                case 'I':
                    return 0;           // Do nothing, already invalid

                default:
                    return -1;          // Non-MESI state
            }
        }
    }


    return 0;
}

int snoopData_L2Cache(unsigned int hex_value) {
    unsigned int tag_bits = hex_value >> (BYTE_SELECT + INDEX);  // Extract
tag bits from hex value
    int i = 0;

    // Iterate over data cache of other processors to find matching tags
    for (i = 0; i < 8; ++i)
    {
        if (data_cache[i].tag_bits == tag_bits)
        {  // If tag bits match

            if
            (
                (data_cache[i].MESI_state == 'M') ||
                (data_cache[i].MESI_state == 'E') ||
                (data_cache[i].MESI_state == 'S')
            )
            {
                data_cache[i].MESI_state = 'I';
            }
            else
            {
                return 0;
            }

            // If in chosen mode, simulate data return to L2 cache
            if (chosen_mode == 1)
                printf("\n\tL2 Cache Receives Data: %x", hex_value); //
Display the hex value sent back to L2 cache

        }
```

```
    }
    return 0;
}

void reset_AllCacheAndStats(void) {
         int i;
    printf("\n\t Resetting Cache data...\n\n");
    // Clear the instruction cache
    for (i = 0; i < 4; ++i)
    {
        instruction_cache[i].tag_bits = 0;
    instruction_cache[i].LRU_bits = 0;
    instruction_cache[i].MESI_state = 'I';
    }

    // Clear the data cache
    for (i = 0; i < 8; ++i)
    {
        data_cache[i].tag_bits = 0;
    data_cache[i].LRU_bits = 0;
    data_cache[i].MESI_state = 'I';
        data_cache[i].address = 0;
    }

    // Reset all statistics
    printf("\n\t Resetting statistics...\n\n");
    statistics.data_cache_hit = 0;
    statistics.data_cache_miss = 0;
    statistics.data_cache_read = 0;
    statistics.data_cache_write = 0;
    statistics.data_cache_hit_ratio = 0.0;
    statistics.instruction_cache_hit = 0;
    statistics.instruction_cache_miss = 0;
    statistics.instruction_cache_read = 0;
    statistics.instruction_cache_hit_ratio = 0.0;

    return;
}

void display_AllCacheContents(void) {
        int i;

    printf("\n\n\t Pereperaing content tables...\n\n");
    printf("-----------------\033[1mMESI States
key\033[0m---------------\n");
    printf("Modified(M), Exclusive(E), Shared(S), Invalid(I)\n");
    printf("-----------------------------------------------\n\n");
    printf("\033[3mTable 1: Data Cache Content\033[0m\n");
    printf("-----------------------------------------------\n");
    printf(" \033[1mWay |   Address   |   Tag   | LRU | MESI State
\033[0m\n");
    printf("-----------------------------------------------\n");
    for (i = 0; i < 8; ++i) {
        printf("  %d  |  0x%08x |  0x%03x  |  %u  |     %c      \n",
           i + 1, data_cache[i].address, data_cache[i].tag_bits,
```

```
data_cache[i].LRU_bits, data_cache[i].MESI_state);
        printf("-----------------------------------------------\n");
    }

    printf("\n\033[3mTable 2: Instruction Cache Content\033[0m\n");
    printf("-----------------------------------------------\n");
    printf(" \033[1mWay |   Address   |   Tag   | LRU | MESI State
\033[0m\n");
    printf("-----------------------------------------------\n");
    for (i = 0; i < 4; ++i) {
        printf("  %d  |  0x%08x |  0x%03x  |  %u  |     %c      \n",
           i + 1, instruction_cache[i].address,
instruction_cache[i].tag_bits, instruction_cache[i].LRU_bits,
instruction_cache[i].MESI_state);
        printf("-----------------------------------------------\n");
    }


    // Calculate data hit ratios
    if (statistics.data_cache_miss == 0) {
        statistics.data_cache_hit_ratio = 0;
    }
    else
    {
        statistics.data_cache_hit_ratio = (float)statistics.data_cache_hit /
(float)statistics.data_cache_miss;
    }

    // Calculate instruction hit ratios
    if (statistics.instruction_cache_miss == 0) {
        statistics.instruction_cache_hit_ratio = 0;
    }
    else
    {
        statistics.instruction_cache_hit_ratio =
(float)statistics.instruction_cache_hit /
(float)statistics.instruction_cache_miss;
    }

    printf("\n\033[3mTable 3: Cache Statistics\033[0m\n");
    printf("-----------------------------------------------\n");
    printf(" \033[1mMetric                      |       Value      \033[0m\n");
    printf("-----------------------------------------------\n");
    printf(" Data Cache Reads           |         %u        \n",
statistics.data_cache_read);
    printf("-----------------------------------------------\n");
    printf(" Data Cache Writes          |         %u        \n",
statistics.data_cache_write);
    printf("-----------------------------------------------\n");
    printf(" Data Cache Hits            |         %u        \n",
statistics.data_cache_hit);
    printf("-----------------------------------------------\n");
    printf(" Data Cache Misses          |         %u        \n",
statistics.data_cache_miss);
    printf("-----------------------------------------------\n");
```

```
    printf(" Data Cache Hit/Miss Ratio  |        %.2f       \n",
statistics.data_cache_hit_ratio);
    printf("-----------------------------------------------\n");

    printf("\n\033[3mTable 4: Instruction Cache Statistics\033[0m\n");
    printf("-----------------------------------------------\n");
    printf(" \033[1mMetric                          |    Value
\033[0m\n");
    printf("-----------------------------------------------\n");
    printf(" Instruction Cache Reads          |      %u      \n",
statistics.instruction_cache_read);
    printf("-----------------------------------------------\n");
    printf(" Instruction Cache Hits           |      %u      \n",
statistics.instruction_cache_hit);
    printf("-----------------------------------------------\n");
    printf(" Instruction Cache Misses         |      %u      \n",
statistics.instruction_cache_miss);
    printf("-----------------------------------------------\n");
    printf(" Instruction Cache Hit/Miss Ratio |     %.2f     \n",
statistics.instruction_cache_hit_ratio);
    printf("-----------------------------------------------\n");

}
```

A.3) cache.h

```
/**
*   TEAM 4
*   Names:  Mohammad Hasan
*           Wa'el Alkalbani
*           Mohamed Gnedi
*           Yousef Alothman
*
*   Project: ECE 585 Final Cache Project
*   File: cache.h
*
*   Description: This header file contains the definitions of the cache
structure,
*                statistics structure, operation enums, and function
prototypes
*                used in the cache simulation.
*/

// Include guard to prevent multiple inclusions of this header file
#ifndef CACHE_H
#define CACHE_H

#include <stdint.h> // Include the standard integer library for fixed-width
integer types

// Define constants for the cache configuration
#define TAG_BITS 12         // Number of bits used for the tag
#define INDEX 14            // Number of bits used for the index
#define BYTE_SELECT 6       // Number of bits used for byte selection
```

```c
// Define a structure to hold cache statistics
typedef struct my_stat {
    unsigned int data_cache_hit;          // Number of data cache hits
    unsigned int data_cache_miss;         // Number of data cache misses
    unsigned int data_cache_read;         // Number of data cache reads
    unsigned int data_cache_write;        // Number of data cache writes
    float data_cache_hit_ratio;           // Ratio of data cache hits to
misses
    unsigned int instruction_cache_hit;   // Number of instruction cache hits
    unsigned int instruction_cache_miss;  // Number of instruction cache
misses
    unsigned int instruction_cache_read;  // Number of instruction cache reads
    float instruction_cache_hit_ratio;    // Ratio of instruction cache hits
to misses
} STATISTICS;

// Define a structure to represent a cache line
typedef struct my_cache {
    unsigned int tag_bits;     // Tag bits of the cache line
    unsigned int LRU_bits;     // Least Recently Used bits for cache
replacement
    char MESI_state;           // MESI state of the cache (Modified, Exclusive,
Shared, Invalid)
    unsigned char data[64];    // Data stored in the cache line
    unsigned int address;      // Address of the cache line
} CACHE;

// Define an enumeration for cache operations
typedef enum ops {
    L1_READ = 0,               // Read from L1 cache
    L1_WRITE = 1,              // Write to L1 cache
    L1_FETCH_INSTRUCTION = 2, // Fetch instruction from L1 cache
    INVALIDATE_L2 = 3,         // Invalidate L2 cache entry
    SNOOP_L2_CACHE = 4,        // Snoop L2 cache
    RESET_CACHE = 8,           // Reset cache
    PRINT_ALL_CACHE = 9,       // Print all cache contents
} OPERATIONS;


// Declare external variables for cache arrays and statistics
extern CACHE data_cache[8];           // Array for data cache lines
extern CACHE instruction_cache[4];    // Array for instruction cache lines
extern STATISTICS statistics;         // Cache statistics
extern unsigned int chosen_mode;      // Mode chosen for cache operation


// Function prototypes for cache operations
int find_DataTag(unsigned int tag_bits);
int find_InstructionTag(unsigned int tag_bits);
int locateLRU_Data(void);
int locateLRU_Instruction(void);
int findInvalid_Data_MESIState(void);
int findInvalid_Instruction_MESIState(void);
void updateLRU_Instruction(unsigned int way);
```

```
void updateLRU_Data(unsigned int way);
int readData_L1Cache(unsigned int hex_value);
int writeData_L1Cache(unsigned int hex_value);
int fetchInstruction_L1Cache(unsigned int hex_value);
int invalidate_L2CacheEntry(unsigned int hex_value);
int snoopData_L2Cache(unsigned int hex_value);
void reset_AllCacheAndStats(void);
void display_AllCacheContents(void);

#endif // CACHE_H
```

A.4) Makefile

```
all:
    @echo "Compiling the project..."
    gcc -o cache.exe main.c cache.c
    @echo "Compilation successful!"
    @echo "type 'make run' to start.."
run: all
    @echo "Running the executable..."
    ./cache.exe
clean:
    @echo "Cleaning up..."
    rm -f cache.exe
    @echo "Clean up complete!"
.PHONY: all run clean
```

A.3) test_cache.txt

```
2 408ed4
0 10019d94
2 408ed8
1 10019d88
2 408edc
3 10019d90
4 10019d94
0 10019d98
1 10019d9c
2 408ee0
9 10019da0
8 10019da4
2 408ee4
0 10019da8
1 10019dac
3 10019db0
4 10019db4
9 10019db8
```

Appendix B. | Design Log

Below is a task a location for the project with (time in hours):

——----------February 29th, 2024——----------

- ➔ Wa'el Alkalbani (4)
  - ◆ Understand the project requirements and write a pseudocode
  - ◆ Sketched a design flowchart using Lucid
- ➔ Mohammad Hasan (4)
  - ◆ Started coding and completed main.c program to open txt traces file, read, and close.
  - ◆ Discussed the program with team

——----------March 2nd, 2024——----------

- ➔ Wa'el Alkalbani (2) & Mohammad Hasan (2)
  - ◆ We discussed the program again and broke down the the design to understood:
    - There are two different caches: data cache, and instruction cache
    - The program needs to call functions based on trace value:
      - 0 → Read
      - 1 → Write
      - 2 → Instruction Fetch
      - 3 → Invalidate
      - 4 → Snoop
      - 8 → Clear & Reset
      - 9 → Print
- ➔ Mohammad Hasan (3)
  - ◆ Debug code and check calculations to modify code
- ➔ Wa'el Alkalbani (3)
  - ◆ Complete the cache simulation coding
  - ◆ Split the code into different files: cache.c cache.h and kept main.c to call functions
  - ◆ Completed the executions files and makefile. Also added a menu for simulation modes: 0, 1 and 2 (which ends the program)

——----------March 5th, 2024——----------

- ➔ Yousef Alothman (5)
  - ◆ Started report decorations
  - ◆ Completed, overview, introduction, discussion, and conclusion
  - ◆ Worked on the flow chart of the program
- ➔ Wa'el Alkalbani (1)
  - ◆ Generate example traces file and tested various scenarios for the program
  - ◆ Communicated with Yousef to document testing results

——----------March 11th, 2024——----------

- ➔ Wa'el Alkalbani (2)
  - ◆ Finished off the assumption section and added the MESI protocols to introduction

—----------March 12th, 2024—----------

- ➔ Mohamed Gnedi (5)
  - ◆ Completed Main program flowchart
  - ◆ Created flowcharts for cache and program commands
  - ◆ Reviewed main cache program to match the flowcharts
  - ◆ Included the MESI Finite State Machine
  - ◆ Added a description of the state transitions and operations
- ➔ Wa'el Alkalbani (2.5)
  - ◆ Work with Youeef to update flowchart
  - ◆ Discussed cache simulation code and sub-flow charts with Mohammed
  - ◆ Write up an abstract (overview)
  - ◆ Submit final report and project source code files
- ➔ Yousef Alothman (2)
  - ◆ Worked with Wael with flowchart.
  - ◆ Add more information into the conclusion.
  - ◆ Rewrote the conclusion and added some more information.

Appendix C. | Specifications and Requirements

ECE 485/585
Winter 2024
Final Project Description

Your team is responsible for the design and simulation of a split L1 cache for a new 32-bit processor which can be used with up to three other processors in a shared memory configuration. The system employs a MESI protocol to ensure cache coherence.

Your L1 instruction cache is four-way set associative and consists of 16K sets and 64-byte lines. Your L1 data cache is eight-way set associative and consists of 16K sets of 64-byte lines. The L1 data cache is write-back using write allocate and is write-back except for the first write to a line which is write-through. Both caches employ LRU replacement policy and are backed by a shared L2 cache. The cache hierarchy employs inclusivity.

Describe and simulate your cache in Verilog, C, or C++. If using Verilog, your design does not need to be synthesizable. Your simulation does not need to be clock accurate and does not need to store/retrieve data.

Maintain and report the following key statistics of cache usage for each cache and display them upon completion of execution of each trace:

- Number of cache reads
- Number of cache writes
- Number of cache hits
- Number of cache misses
- Cache hit ratio

In order to maintain inclusivity and implement the MESI protocol the L1 caches may have to communicate with the shared L2 cache. To simulate this, you should display the following messages (where <address> is a hexadecimal address).

- Return data to L2 <address>
    - In response to a 4 in the trace file your cache should signal that it's returning the data for that line (if present and modified)
- Write to L2 <address>
    - This operation is used to write back a modified line to L2 upon eviction from the L1 cache. It is also used for an initial write through when a cache line is written for the first time so that the L2 knows it's been modified and has the correct data
- Read from L2 <address>
    - This operation is used to obtain the data from L2 on an L1 cache miss
- Read for Ownership from L2 <address>
    - This operation is used to obtain the data from L2 on an L1 cache write miss

Modes

Your simulation must support at least two modes (without the need to recompile). In the first mode, your simulation displays only the required summary of usage statistics and responses to 9s in the trace file and nothing else. In the second mode, your simulation should display everything from mode 0 but also display the communication messages to the L2 described above (and nothing else). You may choose to have additional modes or conditional compilation that display debug information but these should not be present in the version that you demonstrate for the final project demo. Your simulation should not require recompilation to change the name of the input trace file.

Project report

You must submit a brief report that includes relevant internal design documentation, assumptions and design decisions, the source code for your cache, any associated modules used in the validation of the cache (including the testbench if using Verilog), and your simulation results along with the usage statistics.

Traces

Your testbench must read cache accesses/events from a text file of the following format. You should not make any assumptions about alignment of memory addresses. You can assume that memory references do not span cache line boundaries.

```
n address
```

Where n is

0 read data request to L1 data cache
1 write data request to L1 data cache
2 instruction fetch (a read request to L1 instruction cache)
3 invalidate command from L2
4 data request from L2 (in response to snoop)
8 clear the cache and reset all state (and statistics)
9 print contents and state of the cache (allow subsequent trace activity)

The address will be a hex value. For example:

```
2 408ed4
0 10019d94
2 408ed8
1 10019d88
2 408edc
```

When printing the contents and state of the cache in response to a 9 in the trace file, use a concise but readable form that shows only the valid lines in the cache along with way and appropriate state and LRU bits.

Grading

The project is worth 100 points. Your grade will be based upon:

- External specification
- Completeness of the solution (adherence to the requirements above)
- Correctness of the solution
- Quality and readability of the project report (e.g. specifications, design decisions)
- Validity of design decisions
- Quality of implementation
- Structure and clarity of design
- Readability
- Maintainability
- Testing
- Presentation of results
- Project Roles & Responsibility document: listing and describing each team member's contribution to the project

Roles and Responsibility of Each Member in the Project

Each project team member needs to contribute to the project. In the final project material submission, in addition to submitting the final project specification and all source code, each team needs to also submit a Project Roles & Responsibility document. In this Roles & Responsibility document, you need to list who did what for the project for every member of the project team. All team members shall work together to define and agree on the roles and responsibility of each team member for the project and the amount of work distributed to each team member shall be fairly even, thus ensuring every team member contribute to a significant portion for the project. It is the student's job to actively and regularly participate in team meetings, work with team members collaboratively and actively contributes to the project. At the final project presentation, TA and/or instructor will review and ask questions to every student on the project team on the individual student's contribution to the project.