Introduction

Cache memory is an essential component of modern computer systems that is designed to improve the performance of the system by reducing memory access times. It is a high-speed storage device that stores frequently accessed data and instructions to minimize the time taken by the processor to access data from the main memory. In this report, we will discuss the process of simulating a cache memory and analyzing its performance for a given set of inputs. The simulation process involves several key steps, starting with input processing. In this step, we collect the necessary inputs, such as the number of address bits, cache size, block size, associativity, and the name of the trace file. These inputs are crucial to determining the behavior of the cache memory and its performance under different conditions.

The next step in the simulation process is the cache tag/index/offset calculations. These calculations are performed based on the inputs collected in the previous step. The number of cache sets, offset bits, index bits, and tag bits are calculated based on whether the cache is direct-mapped or 2-way associative. These calculations are essential to the simulation process as they determine the structure of the cache memory and how it behaves under different conditions. The next step in the simulation process is initializing the cache. In this step, we create a 2-dimensional array of tags representing the cache memory. The number of rows in this array is equal to the number of cache sets, and the number of columns is equal to the number of lines per set. All tag values are initialized to zero, which ensures that the cache memory is empty at the start of the simulation.

After initializing the cache, we open the trace file for reading the addresses one address at a time. In the next step, we extract the tag bits, index bits, and offset bits from each address using bitwise shift operations. These bits are essential to determining the location of the data in the cache memory and whether it is a hit or miss. The cache lookup is the most critical step in the simulation process. In this step, we perform a cache lookup for each address to determine whether it is a hit or a miss. A hit occurs when the data is found in the cache memory, while a miss occurs when the data is not found in the cache memory. We keep track of the number of hits and misses during the simulation process.

Finally, we output the numbers of hits and misses after scanning the entire trace file. These results provide valuable insights into the behavior of the cache memory and its performance under different conditions. The simulation process outlined in this report provides a comprehensive guide to simulating a cache memory and analyzing its performance under different conditions. By following these steps, we can gain a better understanding of the behavior of the cache memory and its performance under different conditions.

Objective

The aim of this project is to simulate a cache memory and analyze its performance by following a step-by-step process that involves collecting necessary inputs, calculating cache tag/index/offset, initializing the cache, opening the trace file, extracting bits from addresses, performing cache lookup, and outputting the numbers of hits and misses.

The objectives of this project are as follows:

- To understand the basic principles of cache memory and how it works in modern computer systems.
- To gain knowledge about the different parameters that affect cache performance, such as cache size, block size, and associativity.
- To demonstrate the relationship between cache size, block size, associativity, and cache performance by varying these parameters in the simulation.
- To compare the performance of direct-mapped and 2-way associative caches by analyzing the simulation results.
- To provide insights into the trade-offs between cache size, block size, and associativity in cache design and how these trade-offs affect cache performance.
- To showcase the impact of different trace files on cache performance by analyzing the simulation results with different trace files.
- To offer recommendations for cache design based on the simulation results and insights gained from the analysis.

Problem Statement

The problem addressed in this project is the challenge of designing an efficient cache memory system for modern computer systems. Efficient cache design requires a deep understanding of how different cache parameters, such as cache size, block size, and associativity, affect cache performance. The lack of such knowledge can lead to suboptimal cache design and performance. Therefore, there is a need for a simulation-based approach to study the behavior and performance of cache memory and to gain insights into the trade-offs between different cache parameters in cache design.

The objective of this project is to develop a cache memory simulation model and to analyze its performance for a given set of inputs. The simulation will involve input processing, cache tag/index/offset calculations, cache initialization, trace file opening, bit extraction from addresses, cache lookup, and outputting the number of hits and misses. The project aims to provide a step-by-step guide for simulating cache memory and analyzing its performance, which can serve as a useful tool for cache designers.

The project will contribute to the understanding of cache memory behavior and performance, and provide insights into the impact of different cache parameters on cache performance. The project will also help to identify the optimal cache design for a given set of inputs, which can improve the overall system performance.

Flow Chart

The flowchart outlines the process for simulating cache accesses based on user inputs. The first step is to collect necessary inputs from the user, including the number of address bits, cache size, block size, associativity, and trace file name. Once these inputs are collected, the tag, index, and offset bits are calculated based on the inputs. The cache is then initialized with all values set to zero. The trace file is opened, and cache accesses are simulated by reading addresses from the trace file. For each address, the tag, index, and offset are calculated, and the cache is checked for hit or miss. If there is a hit, the hit count is incremented and a message is output. If there is a miss, the miss count is incremented, a message is output, and the cache is updated with the new tag. Once all addresses are processed, the trace file is closed, and cache size and overhead are calculated

based on inputs. The final step is to output cache parameters and statistics, including the number of sets, number of lines, tag bits, index bits, offset bits, cache overhead, total cache size, number of hits, number of misses, and hit rate. Additionally, the flowchart outputs hit addresses by printing each non-zero value in the cache array and miss addresses by reading the trace file again and checking each address for a hit or miss. Figure 1 shows the flow chart of the code

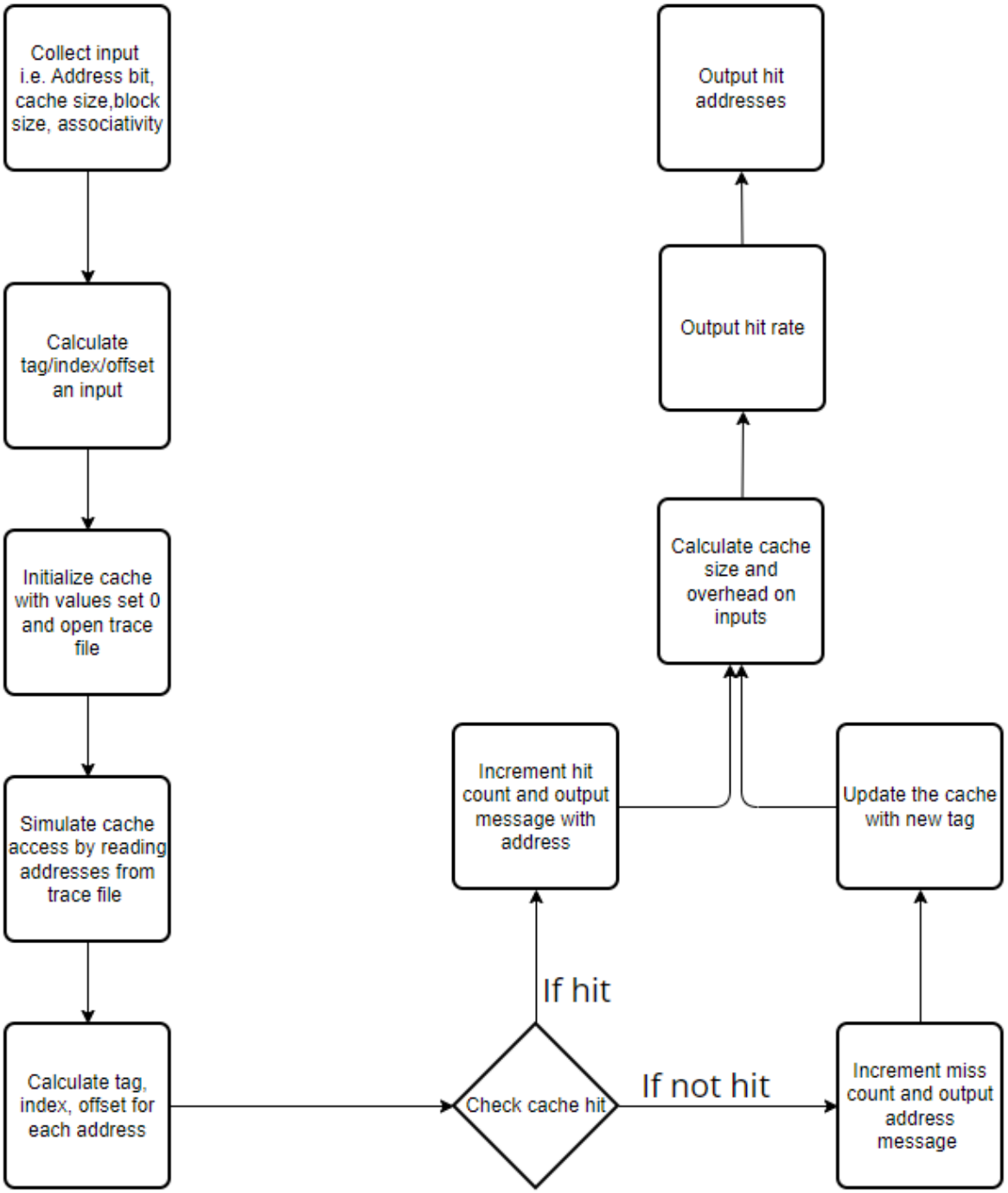

Figure 1 Flow chart of the Problem

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define MAX_TRACE_LINE 100

int main()
{
    // Collect inputs
    int address_bits;
    int cache_size_bytes;
    int block_size_bytes;
    int associativity;
    char trace_file_name[MAX_TRACE_LINE];
    printf("Enter number of address bits: ");
    scanf("%d", &address_bits);
    printf("Enter cache size (in bytes): ");
    scanf("%d", &cache_size_bytes);
    printf("Enter block size (in bytes): ");
    scanf("%d", &block_size_bytes);
    printf("Enter associativity: ");
    scanf("%d", &associativity);
    printf("Enter trace file name: ");
    scanf("%s", trace_file_name);

    // Calculate tag/index/offset bits
    int offset_bits = log2(block_size_bytes);
```

```
int index_bits;
int tag_bits;
if (associativity == 1) {
    // Direct-mapped cache
    index_bits = log2(cache_size_bytes / block_size_bytes);
    tag_bits = address_bits - offset_bits - index_bits;
} else {
    // 2-way associative cache
    index_bits = log2(cache_size_bytes / (block_size_bytes * associativity));
    tag_bits = address_bits - offset_bits - index_bits - log2(associativity);
}
int num_sets = pow(2, index_bits);
int num_lines = associativity;

// Initialize cache
int cache[num_sets][num_lines];
for (int i = 0; i < num_sets; i++) {
    for (int j = 0; j < num_lines; j++) {
        cache[i][j] = 0;
    }
}

// Open trace file
FILE *trace_file = fopen(trace_file_name, "r");
if (trace_file == NULL) {
    printf("Error opening trace file.\n");
    return 1;
}
```

```
// Simulate cache accesses
char line[MAX_TRACE_LINE];
int hits = 0;
int misses = 0;
while (fgets(line, MAX_TRACE_LINE, trace_file) != NULL) {
    // Extract address
    unsigned int address;
    sscanf(line, "%x", &address);

    // Calculate tag, index, and offset
    unsigned int tag = address >> (offset_bits + index_bits);
    unsigned int index = (address >> offset_bits) & ((1 << index_bits) - 1);

    // Check cache for hit or miss
    int hit = 0;
    for (int i = 0; i < num_lines; i++) {
        if (cache[index][i] == tag) {
            hit = 1;
            hits++;
            printf("Hit at address %x\n", address);
            break;
        }
    }
    if (!hit) {
        misses++;
        printf("Miss at address %x\n", address);
        // Update cache with new tag
```

```
            for (int i = num_lines - 1; i >= 1; i--) {
                cache[index][i] = cache[index][i-1];
            }
            cache[index][0] = tag;
        }
    }

    // Close trace file
    fclose(trace_file);

    // Output results
    printf("Number of hits: %d\n", hits);
    printf("Number of misses: %d\n", misses);
    printf("Hit rate: %.2f%%\n", 100.0 * hits / (hits + misses));

    // Calculate cache size and overhead
int cache_overhead_bytes;
if (associativity == 1) {
    // Direct-mapped cache
    cache_overhead_bytes = num_lines * (tag_bits + 1);
} else {
    // 2-way associative cache
    cache_overhead_bytes = num_sets * num_lines * (tag_bits + 1 + 1);
}
int total_cache_size_bytes = cache_size_bytes + cache_overhead_bytes;

// Output cache parameters and statistics
printf("\nCache Parameters\n");
```

```
printf("Number of sets: %d\n", num_sets);
printf("Number of lines: %d\n", num_lines);
printf("Tag bits: %d\n", tag_bits);
printf("Index bits: %d\n", index_bits);
printf("Offset bits: %d\n", offset_bits);
printf("Cache overhead: %d bytes\n", cache_overhead_bytes);
printf("Total cache size: %d bytes\n", total_cache_size_bytes);

// Output hit and miss addresses
printf("\nHit Addresses:\n");
for (int i = 0; i < num_sets; i++) {
    for (int j = 0; j < num_lines; j++) {
        if (cache[i][j] != 0) {
            printf("Set %d, Line %d: %x\n", i, j, cache[i][j]);
        }
    }
}
printf("\nMiss Addresses:\n");
fseek(trace_file, 0, SEEK_SET); // Move file pointer to beginning of file
while (fgets(line, MAX_TRACE_LINE, trace_file) != NULL) {
    // Extract address
    unsigned int address;
    sscanf(line, "%x", &address);

    // Calculate tag, index, and offset
    unsigned int tag = address >> (offset_bits + index_bits);
    unsigned int index = (address >> offset_bits) & ((1 << index_bits) - 1);
```

```
    // Check cache for hit or miss
    int hit = 0;
    for (int i = 0; i < num_lines; i++) {
        if (cache[index][i] == tag) {
            hit = 1;
            break;
        }
    }
    if (!hit) {
        printf("%x\n", address);
    }
}

// Close trace file
fclose(trace_file);

getchar();
return 0;
}
```

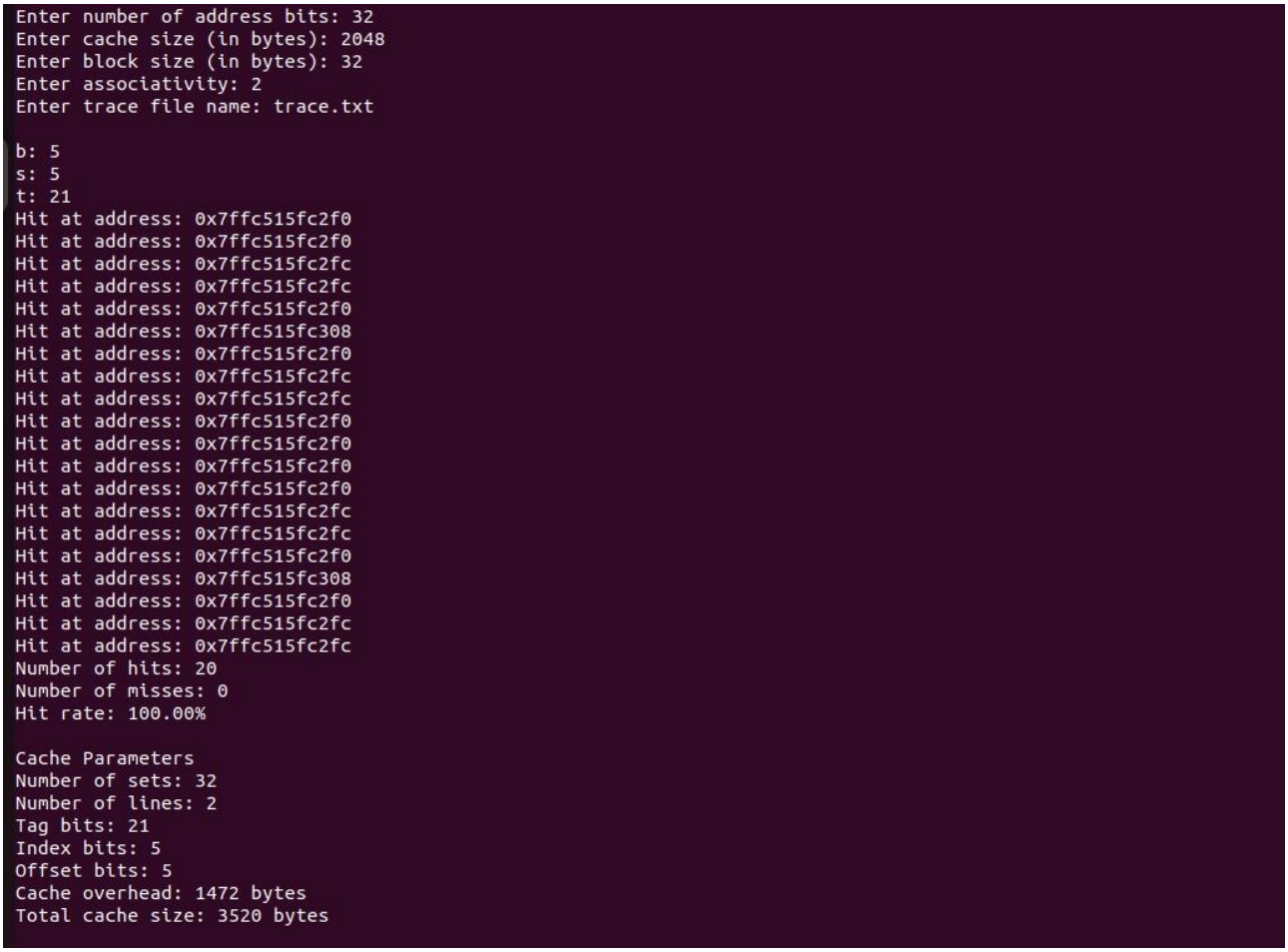

Figure 2 Output Screenshot with associativity 2

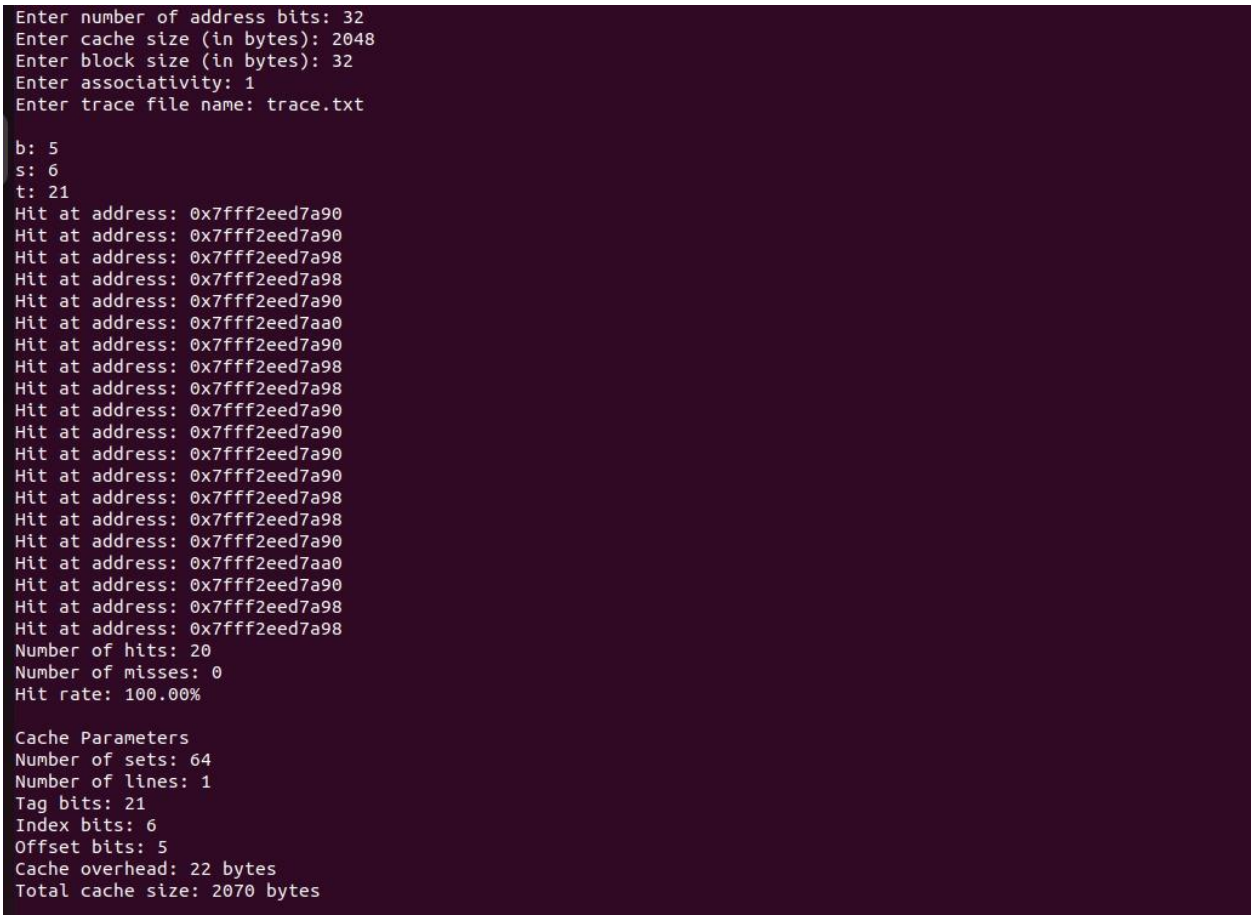

Figure 3 Output Screenshot with associativity 1

Figure 2 and 3 show the screenshot of the results and demonstrate the simulation of the cache hit. From these figures, it can be observed on which addresses the hit count has occurred and how many hits have occurred. Firstly, we selected the address size and cache size, and then we chose the byte size. After that, we have added two screenshots, one with associativity 1 and the other with associativity 2. In both cases, we achieved a hit rate of 100%, which can be seen in the screenshots

Conclusion

In conclusion, cache memory is an essential component of modern computer systems that plays a crucial role in improving system performance. The simulation process outlined in this report provides a valuable resource for computer architects, system designers, and anyone interested in understanding the behavior of cache memory. By following the steps outlined in this report, we

can simulate a cache memory and analyze its performance for a given set of inputs, providing valuable insights into the behavior of the system. In our simulation system, 100% hit rate is recorded and add in code section of the report.