Presented to Mr. Ronald Pascual,

Department of Computer Technology – College of Computer Studies

De La Salle University – Manila

Term 3, A.Y. 2023-2024

Full Association/Most Recently Used Cache Simulator

In Partial Fulfillment of the Course Requirements for

Introduction to Computer Organization and Architecture 2

(CSARCH2)

Submitted by:

Cipriaso, James Kielson S.

Demanalata, Ashantie Louize B.

Hilomen, Geo Brian P.

Santos, Andrea Li S.

Group 4

July 26, 2024

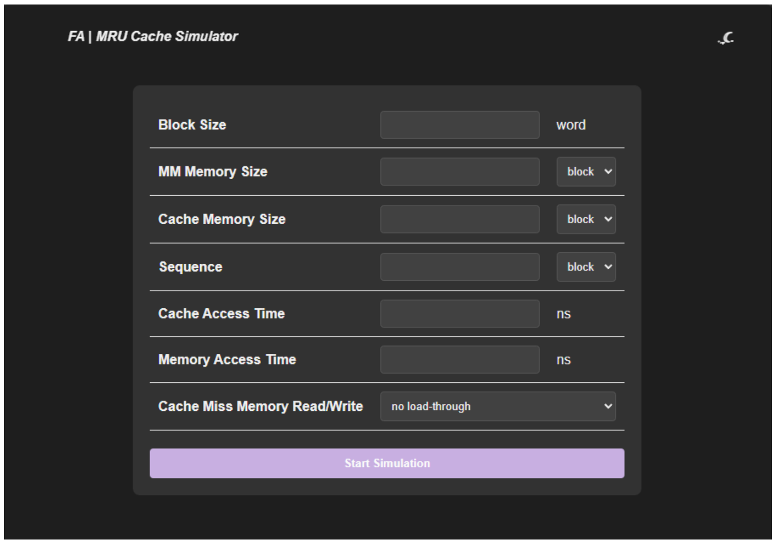
I. Introduction

In modern computing, Cache is a high-speed storage layer that allows for easier accessing of data by the use of the 'block' system where data is accessed by blocks, allowing for a more transient way of storage, making the handling of future data requests much faster. In this simulation project, the cache being simulated utilizes a Fully Associative cache mapping as well as a MRU (Most Recently Used) cache replacement algorithm.

Fully Associative cache mapping refers to a type of cache memory organization where placement of data blocks from the main memory can be done in any of the available cache lines. When the cache becomes full however, cache replacement algorithms are used in order to determine where to place the new incoming data which in this simulation uses MRU.

II. Methodology

A. Web Application Interface


FA | MRU Cache Simulator

Block Size		word
MM Memory Size		block
Cache Memory Size		block
Sequence		block
Cache Access Time		ns
Memory Access Time		ns
Cache Miss Memory Read/Write	no load-through	

Figure A.1: Web Application Interface

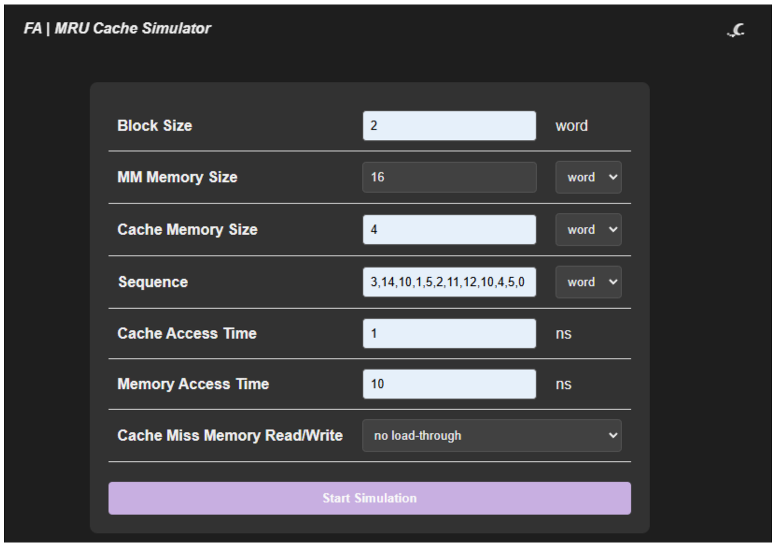
The web application for the cache simulation was entirely built on HTML, CSS, and Javascript. The design was kept simple for ease of use and includes a toggle button for light and dark mode for the viewing preference of the user. The simulation interface consists of forms to be inputted with data by the user. Once the information has been entered, the application will display the following outputs along with their respective Cache Snapshot, Final Cache State, and Sequence Hit/Miss Tables.

a. Input

Inputs	Expected Input
Block Size	Number of words per block.
MM Memory Size	Number of Blocks/Words.
Cache Memory Size	Number of Blocks/Words.
Sequence	Sequence of Blocks/Words to be accessed in main memory.
Cache Access Time	Time taken in accessing the cache in nanoseconds.
Memory Access Time	Time taken in accessing the memory in nanoseconds.
Cache Miss Memory (Read/Write)	Mode on how the cache should fetch or write data.

b. Output

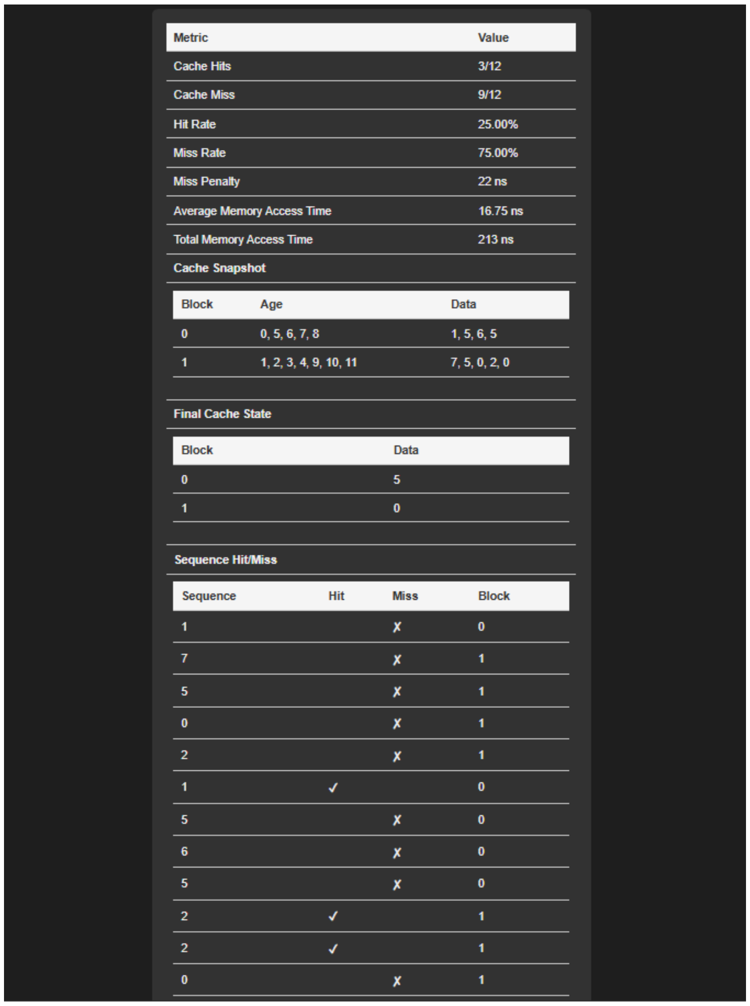
Outputs	Description
Cache Hits	Number of successful hits.
Cache Miss	Number of misses.
Miss Penalty	Time penalty received when there is a Cache Miss.
Miss Rate	Misses / Total Number of Accesses
Hit Rate	Hits / Total Number of Accesses
Average Memory Access Time	Memory Access Time - (Cache Hits / (Cache Hits + Cache Misses)) * Cache Access Time + (Cache Misses / (Cache Hits + Cache Misses)) * Miss Penalty
Total Memory Access Time	Cache Hits * Block Size * Cache Access Time + Cache Misses * (Cache Access Time + Memory Access Time * Block Size + Cache Access Time * Block Size



FA | MRU Cache Simulator

Block Size	2	word
MM Memory Size	16	word
Cache Memory Size	4	word
Sequence	3,14,10,1,5,2,11,12,10,4,5,0	word
Cache Access Time	1	ns
Memory Access Time	10	ns
Cache Miss Memory Read/Write	no load-through	

Figure A.2: Sample Inputs on Simulator Interface



Metric	Value
Cache Hits	3/12
Cache Miss	9/12
Hit Rate	25.00%
Miss Rate	75.00%
Miss Penalty	22 ns
Average Memory Access Time	16.75 ns
Total Memory Access Time	213 ns

Cache Snapshot

Block	Age	Data
0	0, 5, 6, 7, 8	1, 5, 6, 5
1	1, 2, 3, 4, 9, 10, 11	7, 5, 0, 2, 0

Final Cache State

Block	Data
0	5
1	0

Sequence Hit/Miss

Sequence	Hit	Miss	Block
1		✗	0
7		✗	1
5		✗	1
0		✗	1
2		✗	1
1	✓		0
5		✗	0
6		✗	0
5		✗	0
2	✓		1
2	✓		1
0		✗	1

Figure A.3: Sample Output

B. Web Application Programming

Onto the technical side of the web application, all the calculations for the simulator are done in the *script.js* file. This section will be following the step-by-step process for the simulation.

a. Input Checking

JavaScript

```
function checkInputs(bs,mms,mms_type,cms,cms_type,seq,seq_type,cat,mat) {
    const blockSeq = translateSeq(seq);
    const wordSeq = seq.split(",").map(index => {return
Number.parseInt(index)});
    // check for NaN inputs
    if(isNaN(bs)){return 'Error: Block Size input returns NaN';}
    if(isNaN(mms)){return 'Error: MM Memory Size input returns NaN';}
    if(isNaN(cms)){return 'Error: Cache Memory Size input returns NaN';}
    for(let i = 0; i < blockSeq.length; i++) {
        if(blockSeq[i] != blockSeq[i]) {return 'Error: Sequence elements input
returns NaN'}
    }
    if(isNaN(cat)){return 'Error: Cache Access Time input returns NaN';}
    if(isNaN(mat)){return 'Error: Memory Access Time input returns NaN';}

    // check valid blocks in sequence
    switch(mms_type + "-" + seq_type){
        case "block-block":
            if(Math.max.apply(null,blockSeq) > mms){return 'Error: Element/s in
Sequence exceeds MM Memory Size'}
            break;
        case "block-word":
            if(Math.max.apply(null,blockSeq) > mms*bs){return 'Error: Element/s
in Sequence exceeds MM Memory Size'}
            break;
        case "word-word":
            if(Math.max.apply(null,wordSeq) > mms){return 'Error: Element/s in
sequence exceeds MM Memory Size'}
            break;
        case "word-block":
            if(Math.max.apply(null,blockSeq) > mms/bs){return 'Error: Element/s
in sequence exceeds MM Memory Size'}
            break;
    }
    // check memory size
    if(mms_type === 'word'){
        if(mms % bs !== 0){return 'Error: MM Memory Size does not match Block
Size'}
    }
    if(cms_type === 'word'){
        if(cms % bs !== 0){return 'Error: Cache Memory Size does not match
Block Size'}
    }
    return '';
}
```

Figure: B.1: Code Snippet of Input Checking

The first part of the simulator code is the verification of user input. In this section of the program, the inputs are checked based on their expected data type and if their relation to the other inputs are logical and are compatible with the specifications needed for the simulation. In the Input Checking code, at the start of the function is NaN (Not a Number) input verification. This was added in order to not allow users to input a non-number character.

```
JavaScript
function checkInputs(bs,mms,mms_type,cms,cms_type,seq,seq_type,cat,mat) {
    const blockSeq = translateSeq(seq);
    const wordSeq = seq.split(",").map(index => {return
Number.parseInt(index)});
    // check for NaN inputs
    if(isNaN(bs)){return 'Error: Block Size input returns NaN';}
    if(isNaN(mms)){return 'Error: MM Memory Size input returns NaN';}
    if(isNaN(cms)){return 'Error: Cache Memory Size input returns NaN';}
    for(let i = 0; i < blockSeq.length; i++) {
        if(blockSeq[i] != blockSeq[i]) {return 'Error: Sequence elements input
returns NaN'}
    }
    if(isNaN(cat)){return 'Error: Cache Access Time input returns NaN';}
    if(isNaN(mat)){return 'Error: Memory Access Time input returns NaN';}
```

Figure B.1.1: Checking for NaN inputs

The next block of code checks whether the inputted Sequence falls within the boundaries of the inputted Memory Size. This checks for all combinations of word and block between the sequence and memory size.

```javascript
switch(mms_type + "-" + seq_type){
        case "block-block":
            if(Math.max.apply(null,blockSeq) > mms){return 'Error: Element/s in Sequence exceeds MM Memory Size'}
            break;
        case "block-word":
            if(Math.max.apply(null,blockSeq) > mms*bs){return 'Error: Element/s in Sequence exceeds MM Memory Size'}
            break;
        case "word-word":
            if(Math.max.apply(null,wordSeq) > mms){return 'Error: Element/s in sequence exceeds MM Memory Size'}
            break;
        case "word-block":
            if(Math.max.apply(null,blockSeq) > mms/bs){return 'Error: Element/s in sequence exceeds MM Memory Size'}
            break;
    }
```

Figure B.1.2: Checking Sequence and Memory Size compatibility

Finally in input checking, both the Cache Size and Memory Size are compared to the Block Size using the modulo operator in order to see if the block size and the cache/memory size matches each one.

```javascript
if(mms_type === 'word'){
        if(mms % bs !== 0){return 'Error: MM Memory Size does not match Block Size'}
    }
    if(cms_type === 'word'){
        if(cms % bs !== 0){return 'Error: Cache Memory Size does not match Block Size'}
    }
    return '';
```

Figure B.1.3: Comparison of Cache/Memory Size to Block Size

b. Sequence Translation to Word/Block

```
JavaScript
function translateSeq(seq, seq_type, bs) {
    return seq.split(",").map(index => {
        return seq_type === 'word' ? Math.floor(index / bs) :
Number.parseInt(index);
    });
}
```

Figure B.2: Translation of Sequence

Translation of Sequence allows for the conversion of Sequence values into their corresponding block indexes by dividing them by the block size and rounding down to the nearest integer. This is provided that the sequence was inputted in word format, otherwise, the inputted sequence will be parsed as integer and left as is.

c. Cache Checking

```
JavaScript
function contains(cache, block) {
    return cache.findIndex(item => item.block === block);
}

function checkFull(cache) {
    return cache.findIndex(item => item.block === undefined);
}
```

Figure B.3: Functions for checking the cache for block or empty slot respectively

These two functions are used to check the cache for either a specific block or if the cache has any empty slot left stored. Both of these functions are irreplaceable for the cache functionality as these functions determine the cache's actions and whether a cache transfer will miss or hit.

d. Cache Updating

```javascript
function updateCache(cache, cacheSize, block, index, recentIndex) {
    if (cache.length < cacheSize) {
        cache.push({ block: block, age: [index], data: [block] });
        return cache.length - 1;
    } else {
        cache[recentIndex].block = block;
        cache[recentIndex].age.push(index);
        cache[recentIndex].data.push(block);
        return recentIndex;
    }
}
```

Figure B.4: A function that removes or add a block to the cache

An important feature of the simulator as it is in charge of updating the cache storage. This function has an if-else statement that first checks if the contents of the cache is less than its size which it will then add a block. If the cache is full, it will replace the latest block following the MRU.

e. Miss Penalty Calculation

```javascript
function computeMissPenalty(cmm, cat, mat, bs) {
    switch(cmm) {
        case 'no-load':
            return cat + (bs * mat) + cat;
        case 'yes-load':
            return cat + mat + cat;
        case 'no-write':
            return cat + mat;
        case 'yes-write':
            return cat + (bs * mat) + cat + mat;
    }
}
```

Figure B.5: A function that computes the time penalty during cache miss

This function computes the time penalty when there is a cache miss. In Read, when there is a no-load-through miss, it can be calculated by Cache Access Time + (Block Size * Memory Access Time) + Cache Access Time, in load-through it is simply just Cache Access Time + Memory Access Time + Cache Access Time.

In Write, if there is a no-write-allocate, it is calculated by Cache Access Time + Memory Access Time, while in write-allocate it is Cache Access Time + (Block Size * Memory Access Time) + Cache Access Time + Memory Access Time.

f. Generating Tables

```JavaScript
function generateCacheSnapshotAll(cache) {
    let snapshot =
"<thead><tr><th>Block</th><th>Age</th><th>Data</th></tr></thead><tbody>";
    for (let i = 0; i < cache.length; i++) {
        snapshot += `<tr><td>${i}</td><td>${cache[i].age.join(',
')}</td><td>${cache[i].data.join(', ')}</td></tr>`;
    }
    return snapshot + "</tbody>";
}

function generateCacheSnapshotFinal(cache) {
    let snapshot =
"<thead><tr><th>Block</th><th>Data</th></tr></thead><tbody>";
    for (let i = 0; i < cache.length; i++) {
        snapshot += `<tr><td>${i}</td><td>${cache[i].data[cache[i].data.length
- 1]}</td></tr>`;
    }
    return snapshot + "</tbody>";
}

function generateSeqHitMiss(seq, hitMiss) {
    let snapshot =
"<thead><tr><th>Sequence</th><th>Hit</th><th>Miss</th><th>Block</th></tr></thea
d><tbody>";
    for (let i = 0; i < seq.length; i++) {
        snapshot += `<tr><td>${seq[i]}</td><td>${hitMiss[i].hit ? '✔' :
''}</td><td>${hitMiss[i].miss ? '✘' : ''}</td><td>${hitMiss[i].block !==
undefined ? hitMiss[i].block : ''}</td></tr>`;
    }
    return snapshot + "</tbody>";
}
```

Figure B.6: Three functions for generating table data

These three functions are used in order to generate table data in the simulation's interface. These tables generate a Cache Snapshot, Final State Cache Snapshot, and a table of the Cache's Hits and Misses respectively. They will then be added to the HTML page to be shown along with the other results.

g. Simulation Proper

This function combines every other function defined in the script. It handles user inputs using the checkInputs function defined earlier. Once the input validation is successful, it moves on to the next process where it simulates the cache mapping algorithm, Fully Associative Mapping, along with MRU (Most Recently Used) as its replacement algorithm. As the algorithm mostly handles block values, it converts word type inputs to their respective block values. After successfully simulating the cache, the function then computes for the required outputs such as Miss Penalty, Average Memory Access Time, Total Memory Access Time, etc. which are then presented as tables.

```
JavaScript
function simulateCache() {
    const bs = Number(document.forms['main-form']['bs'].value);
    const mms = Number(document.forms['main-form']['mms'].value);
    const mms_type = document.forms['main-form']['mms-type'].value;
    const cms = Number(document.forms['main-form']['cms'].value);
    const cms_type = document.forms['main-form']['cms-type'].value;
    const seq = document.forms['main-form']['seq'].value;
    const seq_type = document.forms['main-form']['seq-type'].value;
    const cat = Number(document.forms['main-form']['cat'].value);
    const mat = Number(document.forms['main-form']['mat'].value);
    const cmm = document.forms['main-form']['cmm'].value;
    let error = checkInputs(bs,mms,mms_type,cms,cms_type,seq,seq_type,cat,mat);
    document.getElementById('input-error').innerHTML = error;
    if(!error.includes("Error")){
        const cache = [];
        let cacheSize;
        if (cms_type === 'word') {
            cacheSize = Math.floor(cms / bs);
        } else {
            cacheSize = cms;
        }
```

Figure B.7.1: Fetching of Form Data and Input Checking

JavaScript

```javascript
const programFlow = translateSeq(seq, seq_type, bs);
        let hit = 0;
        let miss = 0;
        let recentIndex = 0;

        const hitMiss = [];

        // Simulate cache operations
        for (let i = 0; i < programFlow.length; i++) {
            const hitIndex = contains(cache, programFlow[i]);
            if (hitIndex !== -1) {
                hit++;
                recentIndex = hitIndex;
                cache[hitIndex].age.push(i);
                hitMiss.push({ hit: true, miss: false, block: hitIndex });
            } else {
                miss++;
                recentIndex = updateCache(cache, cacheSize, programFlow[i], i,
recentIndex);
                hitMiss.push({ hit: false, miss: true, block: recentIndex });
            }
        }
```

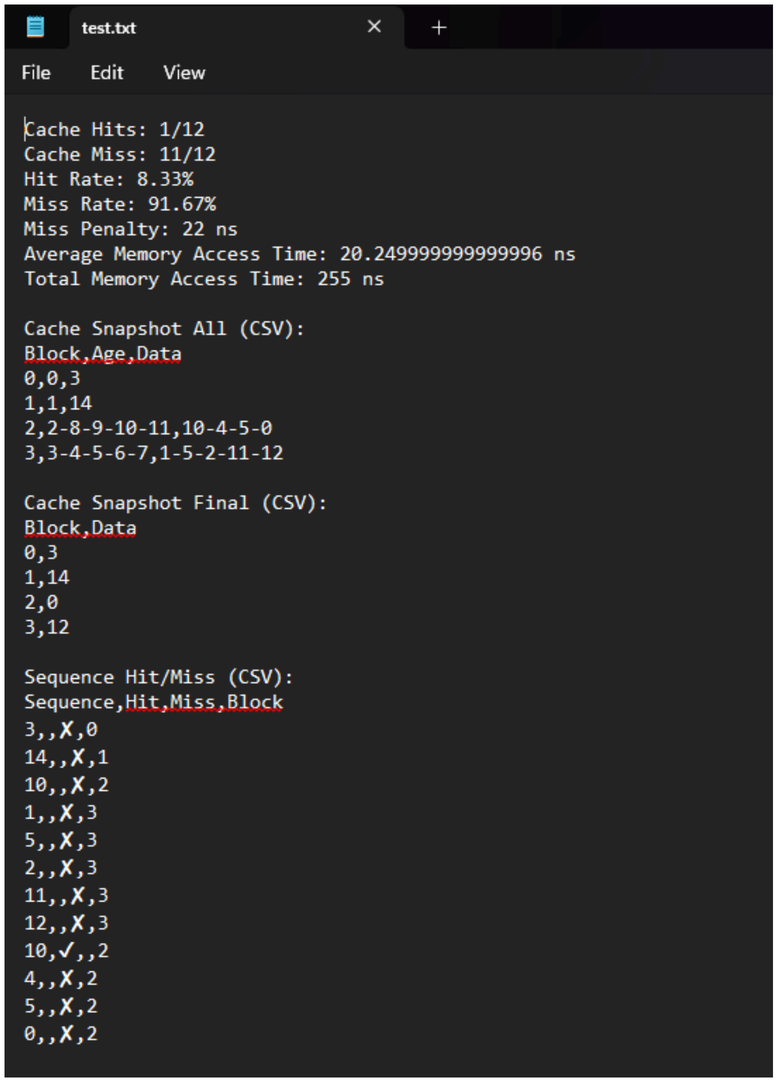
Figure B.7.2: Translation of Program Flow and simulation of Cache Operations

JavaScript
```javascript
const totalAccesses = hit + miss;
        const missPenalty = computeMissPenalty(cmm, cat, mat, bs);
        const aveMAT = (hit / totalAccesses) * cat + (miss / totalAccesses) *
missPenalty;
        const totalMAT = hit * bs * cat + miss * (cat + mat * bs + cat * bs);
        const hitRateFrac = `${hit}/${totalAccesses}`;
        const hitRatePercent = ((hit / totalAccesses) * 100).toFixed(2);
        const missRateFrac = `${miss}/${totalAccesses}`;
        const missRatePercent = ((miss / totalAccesses) * 100).toFixed(2);

        // Display results
        document.getElementById('res-cache-hits').innerHTML = hitRateFrac;
        document.getElementById('res-cache-miss').innerHTML = missRateFrac;
        document.getElementById('res-hit-rate-percent').innerHTML =
hitRatePercent + '%';
        document.getElementById('res-miss-rate-percent').innerHTML =
missRatePercent + '%';
        document.getElementById('res-miss-pen').innerHTML = missPenalty + '
ns';
        document.getElementById('res-ave-mat').innerHTML = aveMAT + ' ns';
        document.getElementById('res-total-mat').innerHTML = totalMAT + ' ns';

        // Display cache snapshots and hit/miss sequence
        document.getElementById('cache-all').innerHTML =
generateCacheSnapshotAll(cache);
        document.getElementById('cache-final').innerHTML =
generateCacheSnapshotFinal(cache);
        document.getElementById('seq-hit-miss').innerHTML =
generateSeqHitMiss(programFlow, hitMiss);

        document.querySelector('.output-parent').classList.add('active');
    } else {
        document.querySelector('.output-parent').classList.remove('active');
    }
    return false;
}
```
Figure B.7.3: Calculation and Displaying of Results

h. Results to File

JavaScript
```javascript
function outputToFile() {
    const filename = document.forms['out-to-file']['filename'].value;

    let content = `Cache Hits:
${document.getElementById('res-cache-hits').innerHTML}\n`;
    content += `Cache Miss:
${document.getElementById('res-cache-miss').innerHTML}\n`;
    content += `Hit Rate:
${document.getElementById('res-hit-rate-percent').innerHTML}\n`;
    content += `Miss Rate:
${document.getElementById('res-miss-rate-percent').innerHTML}\n`;
    content += `Miss Penalty:
${document.getElementById('res-miss-pen').innerHTML}\n`;
    content += `Average Memory Access Time:
${document.getElementById('res-ave-mat').innerHTML}\n`;
    content += `Total Memory Access Time:
${document.getElementById('res-total-mat').innerHTML}\n`;

    const snapshotAll = document.getElementById('cache-all').innerHTML;
    const snapshotFinal = document.getElementById('cache-final').innerHTML;
    const seqHitMiss = document.getElementById('seq-hit-miss').innerHTML;

    content += `\nCache Snapshot All (CSV):\n${snapshotAll.replace(/<\/tr>/g,
"\n").replace(/<\/t(h|d)><t(h|d)>/g, ",").replace(/, /g,
"-").replace(/<\/?[^>]+(>|$)/g, "")}\n`;
    content += `Cache Snapshot Final (CSV):\n${snapshotFinal.replace(/<\/tr>/g,
"\n").replace(/<\/t(h|d)><t(h|d)>/g, ",").replace(/<\/?[^>]+(>|$)/g, "")}\n`;
    content += `Sequence Hit/Miss (CSV):\n${seqHitMiss.replace(/<\/tr>/g,
"\n").replace(/<\/t(h|d)><t(h|d)>/g, ",").replace(/<\/?[^>]+(>|$)/g, "")}\n`;

    const blob = new Blob([content], { type: "text/plain;charset=utf-8" });
    const link = document.createElement("a");
    link.href = URL.createObjectURL(blob);
    link.download = `${filename}.txt`;
    link.click();

    return false;
}
```
Figure B.8: A function that concatenates data into a text file for outputting

Finally, this is a function that allows the web application to create a text file with the contents of the web application added inside. This allows the user to have a saved copy of the results of their simulation.

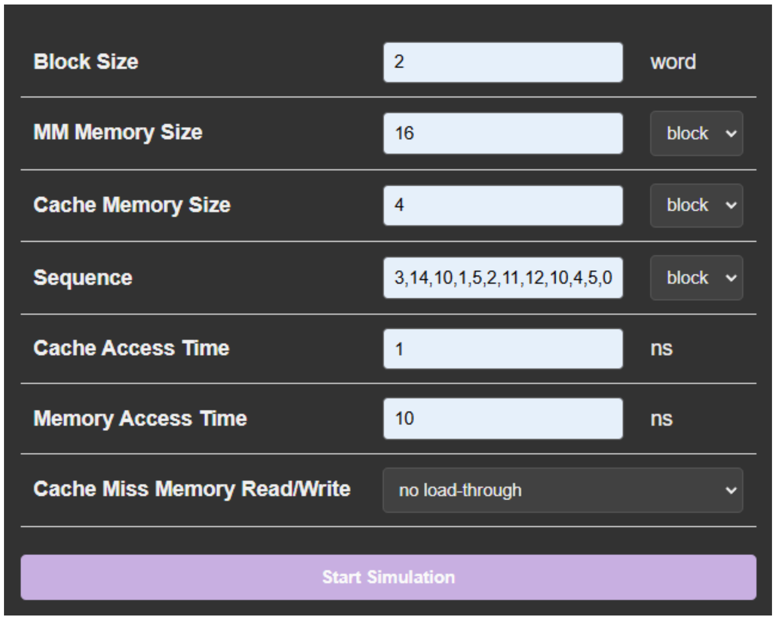


```
Cache Hits: 1/12
Cache Miss: 11/12
Hit Rate: 8.33%
Miss Rate: 91.67%
Miss Penalty: 22 ns
Average Memory Access Time: 20.249999999999996 ns
Total Memory Access Time: 255 ns

Cache Snapshot All (CSV):
Block,Age,Data
0,0,3
1,1,14
2,2-8-9-10-11,10-4-5-0
3,3-4-5-6-7,1-5-2-11-12

Cache Snapshot Final (CSV):
Block,Data
0,3
1,14
2,0
3,12

Sequence Hit/Miss (CSV):
Sequence,Hit,Miss,Block
3,,✗,0
14,,✗,1
10,,✗,2
1,,✗,3
5,,✗,3
2,,✗,3
11,,✗,3
12,,✗,3
10,✓,,2
4,,✗,2
5,,✗,2
0,,✗,2
```

Figure B.8.2: A text file containing the results of the simulation

III. Results and Discussion

Sample Simulation - Block Data Format/No-Load-Through

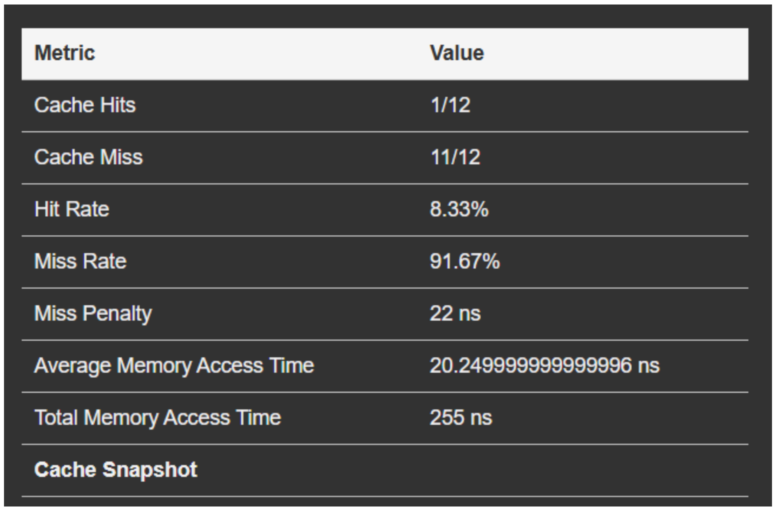


Block Size	2	word
MM Memory Size	16	block
Cache Memory Size	4	block
Sequence	3,14,10,1,5,2,11,12,10,4,5,0	block
Cache Access Time	1	ns
Memory Access Time	10	ns
Cache Miss Memory Read/Write	no load-through	

Figure A.2

In the program interface, there are 7 inputs shown that the user must type in for the simulator to work. Among these inputs, the MM Memory Size, Cache Memory Size, and Sequence are allowed to have either a word or block as their data type. Shown above is an example of an input for the simulator where the Block Size is 2 words, Main Memory size is 16 words, Cache Memory Size is 4 words, and the Sequence contains the values 3,14,10,1,5,2,11,12,10,4,5,0 where each value represents data to be accessed from the Main Memory.

In the above example, the simulation provided these results. From the table, it shows that the simulation accessed the cache 12 times. In Cache accessing, there is 1 Cache Hit and 11 Cache Misses, 8.33% and 91.67% of the data accessed respectively.



Metric	Value
Cache Hits	1/12
Cache Miss	11/12
Hit Rate	8.33%
Miss Rate	91.67%
Miss Penalty	22 ns
Average Memory Access Time	20.249999999999996 ns
Total Memory Access Time	255 ns
Cache Snapshot	

Figure 3.1: Result of the Simulation

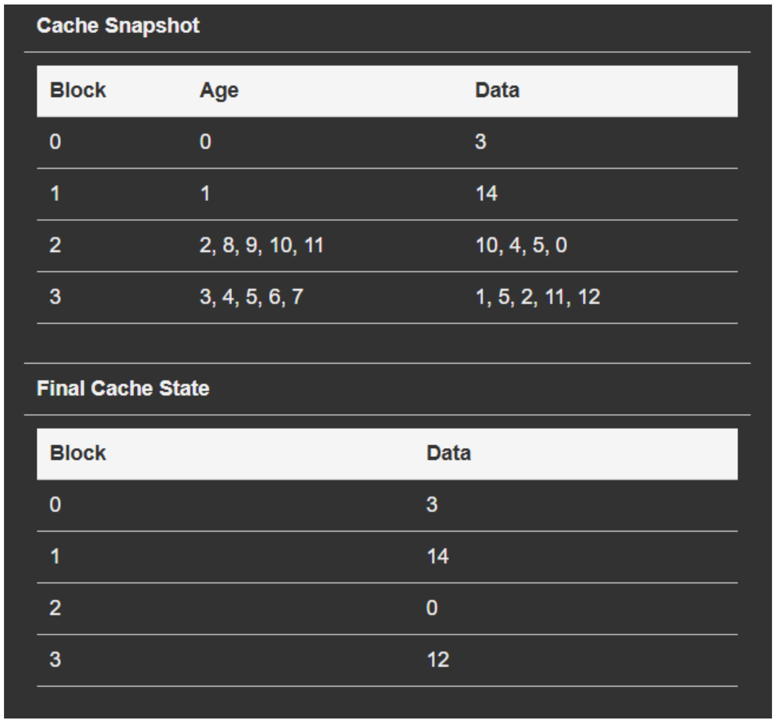
The Time Penalty provided was calculated by following the formula for no-load-through inputs which in this case is Cache Access Time + (Block Size * Memory Access Time) + Cache Access Time. The following table shows the different ways of calculating the Time Penalty depending on Cache's read and write

Cache Miss Type	Function	Formula
no-load-through	Read	Cache Access Time + (Block Size * Memory Access Time) + Cache Access Time
load-through	Read	Cache Access Time + Memory Access Time + Cache Access Time
no-write-allocate	Write	Cache Access Time + Memory Access Time
write-allocate	Write	Cache Access Time + (Block Size * Memory Access Time) + Cache Access Time + Memory Access Time

The inputted values for Cache Access Time is 1 ns while Memory Access Time is 10 ns. Following the no-load-through formula, the Time Penalty for Cache Miss is 22 ns. As for the Average Memory Access Time and Total Memory Access Time, both are calculated by following their respective formulas. As a result the Average and Total Memory Access Times of the simulation are 20.25 ns (rounded up) and 255 ns respectively.

Access Time	**Formula**
Average Memory Access Time	Memory Access Time - (Cache Hits / (Cache Hits + Cache Misses)) * Cache Access Time + (Cache Misses / (Cache Hits + Cache Misses)) * Miss Penalty
Total Memory Access Time	Cache Hits * Block Size * Cache Access Time + Cache Misses * (Cache Access Time + Memory Access Time * Block Size + Cache Access Time * Block Size

Other than simulation results, the simulation also shows a visualization of the Cache, its final state, and the process of how the cache checks the Sequence for hit and miss. Shown below are tables visualizing the cache memory, the Block number, Age of Data for MRU, and the data stored inside the blocks. The Block number specifies the index of block present per block size, the Age of Data shows how long the data has been in memory per access, and the data are stored to their respective blocks.



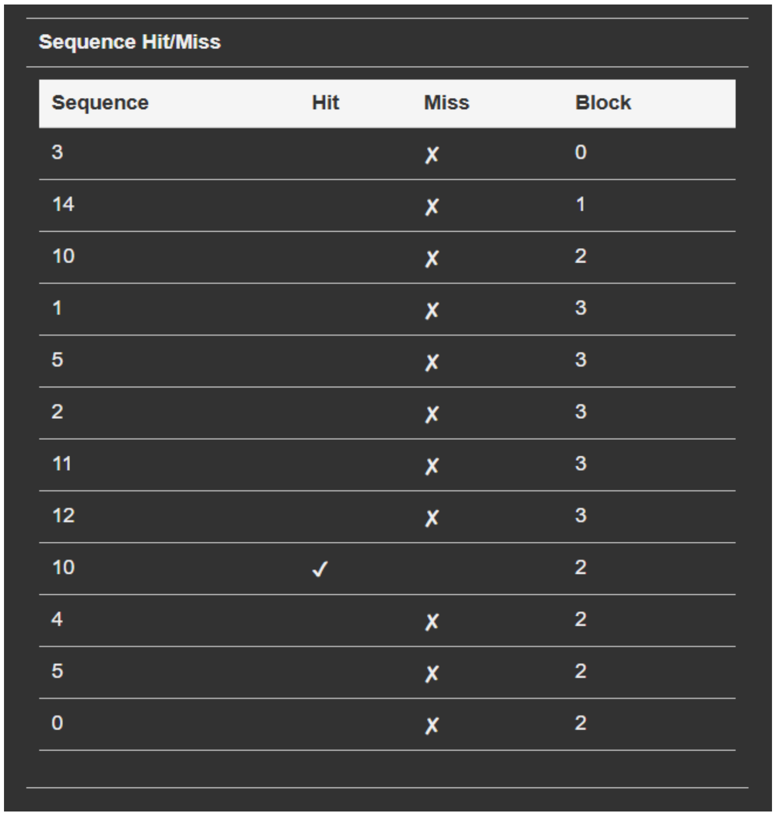
Cache Snapshot

Block	Age	Data
0	0	3
1	1	14
2	2, 8, 9, 10, 11	10, 4, 5, 0
3	3, 4, 5, 6, 7	1, 5, 2, 11, 12

Final Cache State

Block	Data
0	3
1	14
2	0
3	12

Figure 3.2: Cache Snapshot and Final State

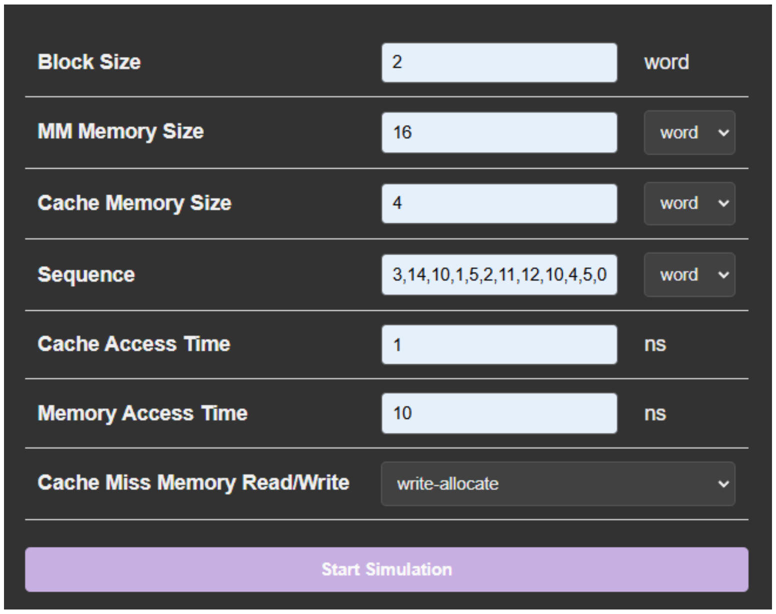


Sequence Hit/Miss

Sequence	Hit	Miss	Block
3		✗	0
14		✗	1
10		✗	2
1		✗	3
5		✗	3
2		✗	3
11		✗	3
12		✗	3
10	✓		2
4		✗	2
5		✗	2
0		✗	2

Figure 3.3: Sequence Process of Hit and Miss

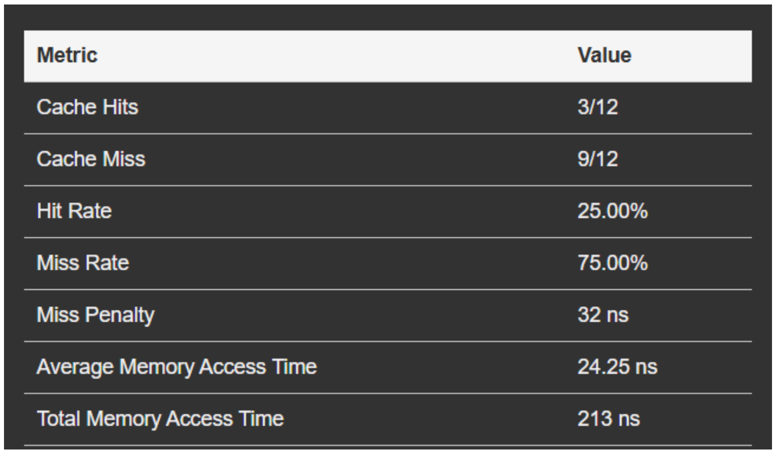
Sample Simulation - Word Data Format/Write-Allocate



Block Size	2	word
MM Memory Size	16	word
Cache Memory Size	4	word
Sequence	3,14,10,1,5,2,11,12,10,4,5,0	word
Cache Access Time	1	ns
Memory Access Time	10	ns
Cache Miss Memory Read/Write	write-allocate	

Figure 3.4: Similar Inputs but different data format and Cache Miss type

This simulation has similar inputs as the other simulation only this time, it has different data types for MM and Cache memory size, as well as the Program Flow which is formatted as word. The Cache Miss format has also been changed to Write-Allocate.



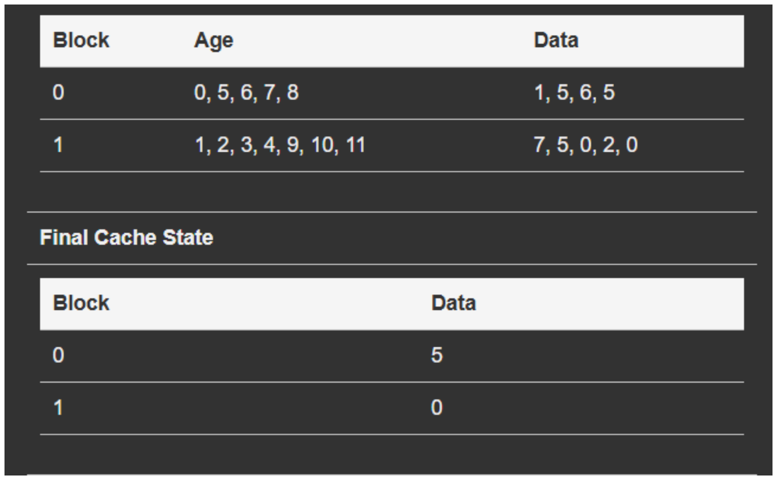
Metric	Value
Cache Hits	3/12
Cache Miss	9/12
Hit Rate	25.00%
Miss Rate	75.00%
Miss Penalty	32 ns
Average Memory Access Time	24.25 ns
Total Memory Access Time	213 ns

Figure 3.5: Results from the Word/Write-Allocate Simulation.

The number of Cache Hits in this simulation has increased to 3 even if it uses the same inputs as the Block/No-Load-Through simulation. This is because of the Sequence/Program Flow being converted into blocks prior to the simulation proper. It was converted by using the Sequence Translation of Word/Block function discussed in the Web Application Programming.

The Time Penalty also changed because of the Cache Miss format the simulation uses which in this case is Write-Allocate. Looking at the formulas from earlier, the Write-Allocate formula is similar to the formula of No-Load-Through with the difference being the addition of Memory Access Time for data writing. With the change of Time Penalty's value, number of Cache Hits and Cache Misses, the Average and Total Memory Access Time also changed to 24.25 ns and 213 ns respectively.

Finally, we have the table data of the simulation. Here we can see the data converted to block format, making the data stored wildly different to the data inputted in the simulation originally



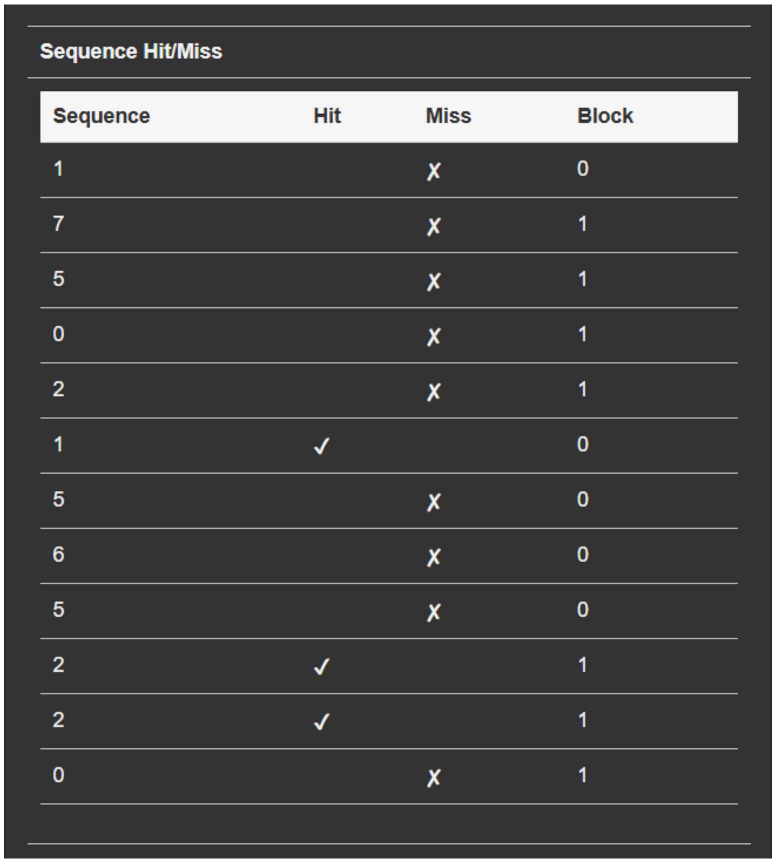
Block	Age	Data
0	0, 5, 6, 7, 8	1, 5, 6, 5
1	1, 2, 3, 4, 9, 10, 11	7, 5, 0, 2, 0

Final Cache State

Block	Data
0	5
1	0

Figure 3.6: Cache Snapshot and Final State from the Word/Write-Allocate Simulation

And shown below is the process by which each converted block is transferred to Cache. Due to the conversion of word to block format in Sequence, the number of Cache Hits increased.



Sequence Hit/Miss

Sequence	Hit	Miss	Block
1		✗	0
7		✗	1
5		✗	1
0		✗	1
2		✗	1
1	✓		0
5		✗	0
6		✗	0
5		✗	0
2	✓		1
2	✓		1
0		✗	1

Figure 3.6: Sequence Hit/Miss from the Word/Write-Allocate Simulation

IV. Conclusion

In conclusion, the simulation of a Fully Associative Cache utilizing a Most-Recently-Used algorithm for storing data has provided an insight into the performance and management of a cache storage system. The simulation was able to showcase the cache being Fully Associative, giving free reign over the placement of data in any cache line while using the Most Recently Used logic in order to free up space.

Also the Web Application proved to be easy to use, even allowing the user to switch between data formats and with it, a simple yet wonderful aesthetic design. The tabulation of data was also neat, tidy, yet brief, allowing the user to easily understand how the simulation process happens, how it looks inside the cache, and the sequence in which the data is stored to the cache.