De La Salle University - Manila

Term 3, A.Y. 2023 - 2024

In partial fulfillment

of the course in

Introduction to Computer Organization and Architecture 2 (S12)

Group 6

Submitted by:

Ang, Czarina Damienne N.

Esteban, Janina Angela M.

Herrera, Diego Martin D.

Lim, Jannica Allison S.

Submitted to:

Mr. Ronald Pascual

July 31, 2024

TABLE OF CONTENTS

I. INTRODUCTION

The website created demonstrates how a block-set-associative cache mapping simulator can store data using the Most Recently Used algorithm or MRU. The group utilized vanilla HTML and CSS for the front-end development of the page and JavaScript for the back-end. For deployment, Render was used.

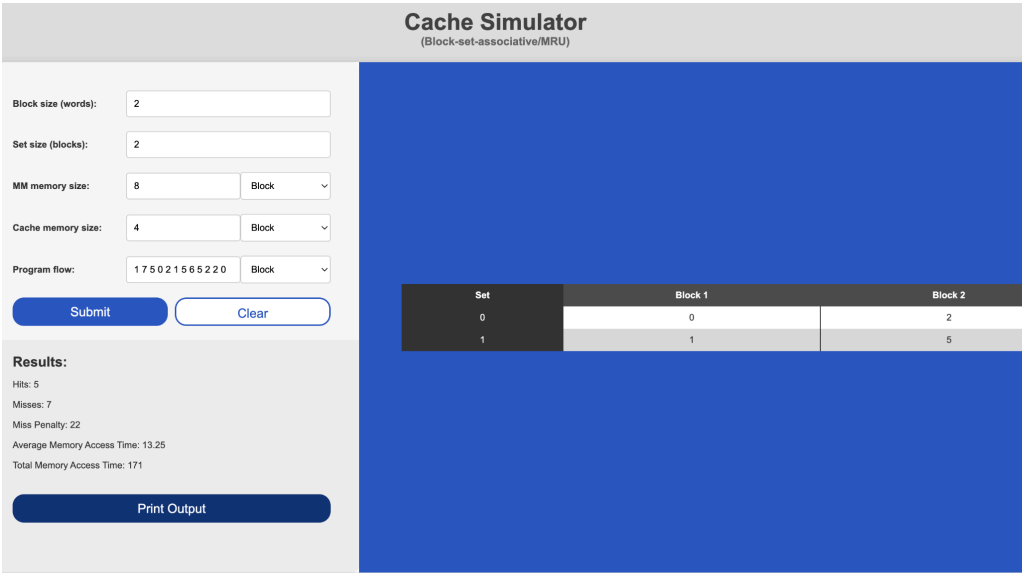
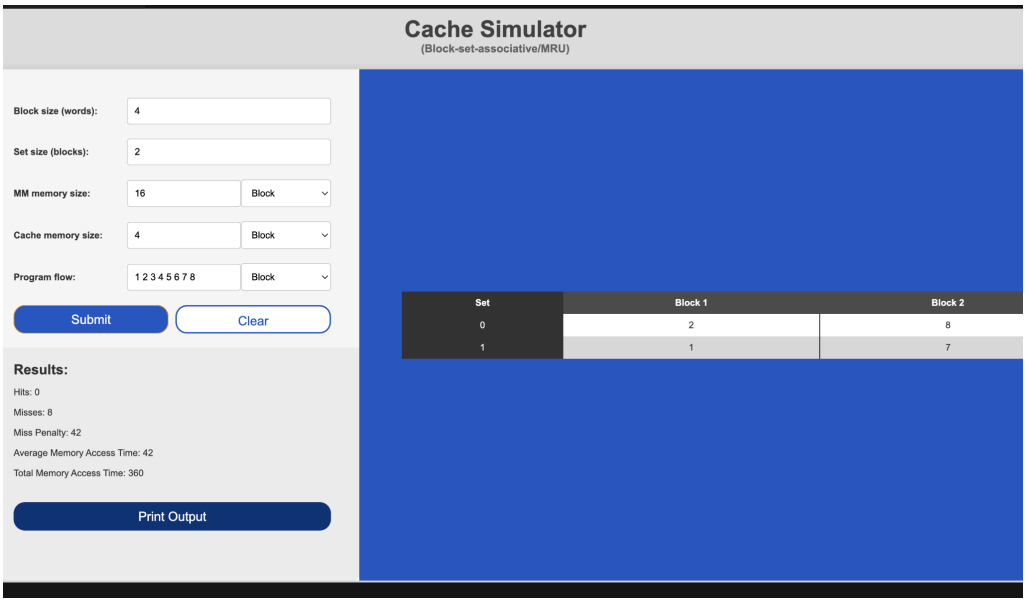
For the first part of the JavaScript code, ‘pow2(x) is used to check if a number ‘x’ is a power of 2 and verifies it by making sure it is greater than 0. The ‘convbin(dec, bits)’ function converts a decimal number ‘dec’ to its binary representation, ensuring the result has a minimum length of ‘bits’. It adds leading zeroes if there are such cases and slices off excess bits to ensure they fit the bit width.

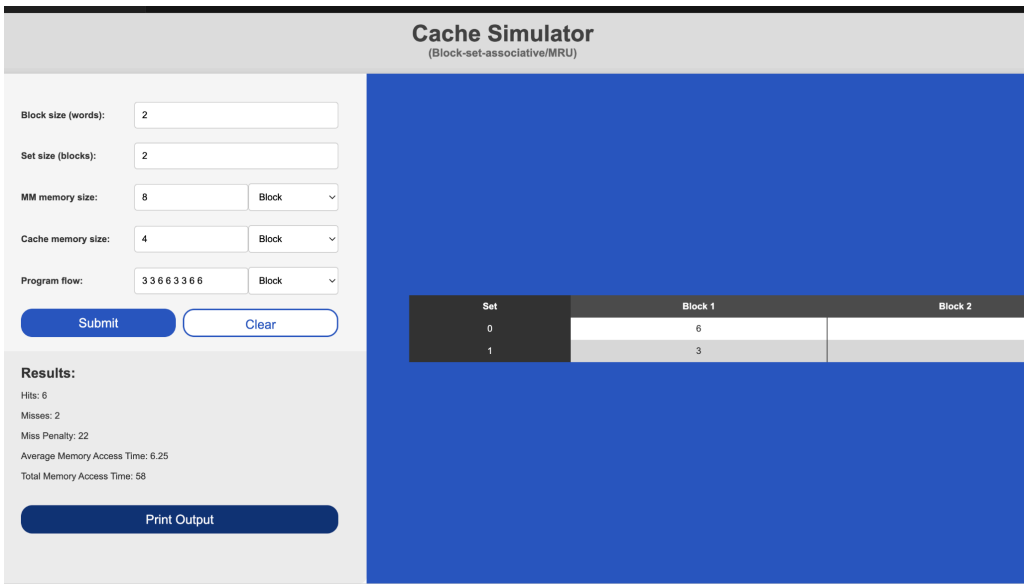
The ‘getset(bin, addnum)’ function extracts the set index from the binary address ‘bin’. The ‘addum’ array specifies the length of the tag and set index portions. On the other hand, the ‘gettag(bin, tag)’ extracts the tag from the binary address using the length specified by ‘tag’. Similar to ‘convbin(dec, bits)’, this function also slices the binary address. The difference is that it tries to obtain the tag portion, which is important for determining the specific block within the set.

The website also uses a different set of script functions for HTML. The ‘alertError(meesage)’ displays an error message by setting the inner text of an HTML element with an ID “error-message” to the provided ‘message’ and making the message visible to the users. The ‘downloadText()’ gathers all the results from the ache simulation and formats them into a text string. It includes all details such as hits, misses, miss penalty, average memory access time, total memory access time, and the cache's current state.

II. TEST CASES

A. Normal Case

1	 Cache Simulator (Block-set-associative/MRU) Block size (words): 2 Set size (blocks): 2 MM memory size: 8 Block Cache memory size: 4 Block Program flow: 1 7 5 0 2 1 5 6 5 2 2 0 Block Submit Clear Results: Hits: 5 Misses: 7 Miss Penalty: 22 Average Memory Access Time: 13.25 Total Memory Access Time: 171 Print Output Set Block 1 Block 2 0 0 2 1 1 5
2	 Cache Simulator (Block-set-associative/MRU) Block size (words): 4 Set size (blocks): 2 MM memory size: 16 Block Cache memory size: 4 Block Program flow: 1 2 3 4 5 6 7 8 Block Submit Clear Results: Hits: 0 Misses: 8 Miss Penalty: 42 Average Memory Access Time: 42 Total Memory Access Time: 360 Print Output Set Block 1 Block 2 0 2 8 1 1 7

3	 Cache Simulator (Block-set-associative/MRU) Block size (words): 2 Set size (blocks): 2 MM memory size: 8 Block Cache memory size: 4 Block Program flow: 3 3 6 6 3 3 6 6 Block Submit Clear Results: Hits: 6 Misses: 2 Miss Penalty: 22 Average Memory Access Time: 6.25 Total Memory Access Time: 58 Print Output Set Block 1 Block 2 0 6 1 3
4	 Cache Simulator (Block-set-associative/MRU) Block size (words): 8 Set size (blocks): 4 MM memory size: 128 Word Cache memory size: 8 Block Program flow: 5 6 7 8 9 10 11 12 Block Submit Clear Results: Hits: 0 Misses: 8 Miss Penalty: 82 Average Memory Access Time: 82 Total Memory Access Time: 712 Print Output Set Block 1 Block 2 Block 3 Block 4 0 6 8 10 12 1 5 7 9 11

5

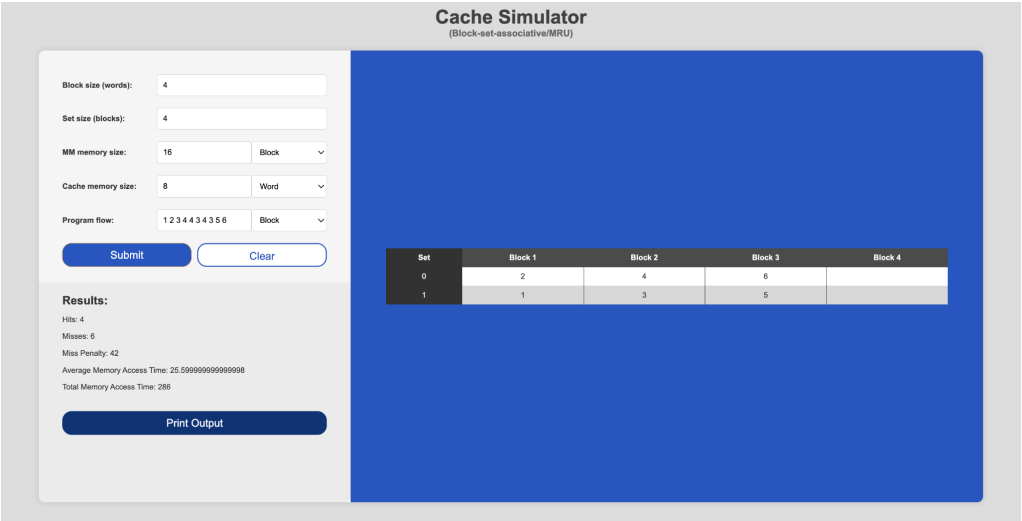

Cache Simulator
(Block-set-associative/MRU)
Block size (words): 4
Set size (blocks): 4
MM memory size: 128 Word
Cache memory size: 16 Word
Program flow: 5 1 2 4 7 5 7 8 0 44 55 Block
Submit
Clear
Results:
Hits: 2
Misses: 10
Miss Penalty: 42
Average Memory Access Time: 35.166666666666664
Total Memory Access Time: 458
Print Output
Set | Block 1 | Block 2 | Block 3 | Block 4
0 | 4 | 8 | 0 | 44
1 | 5 | 1
2 | 2 | 66
3 | 7 | 55

6


Cache Simulator
(Block-set-associative/MRU)
Block size (words): 4
Set size (blocks): 4
MM memory size: 128 Word
Cache memory size: 32 Word
Program flow: 11 22 33 44 55 11 22 Word
Submit
Clear
Results:
Hits: 2
Misses: 5
Miss Penalty: 42
Average Memory Access Time: 30.285714285714285
Total Memory Access Time: 233
Print Output
Set | Block 1 | Block 2 | Block 3 | Block 4
0 | 11 | 33
1 | 22 | 44 | 55
2
3
4
5
6
7

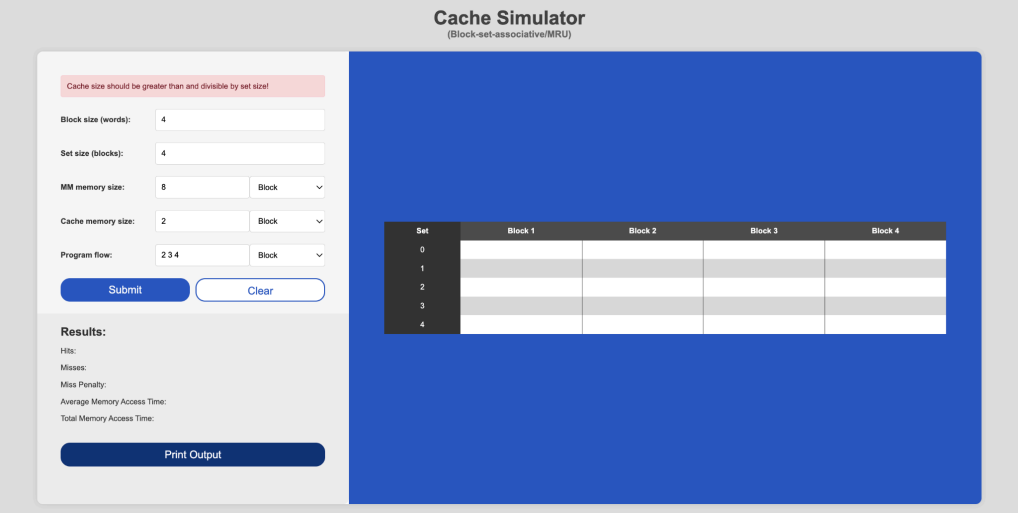
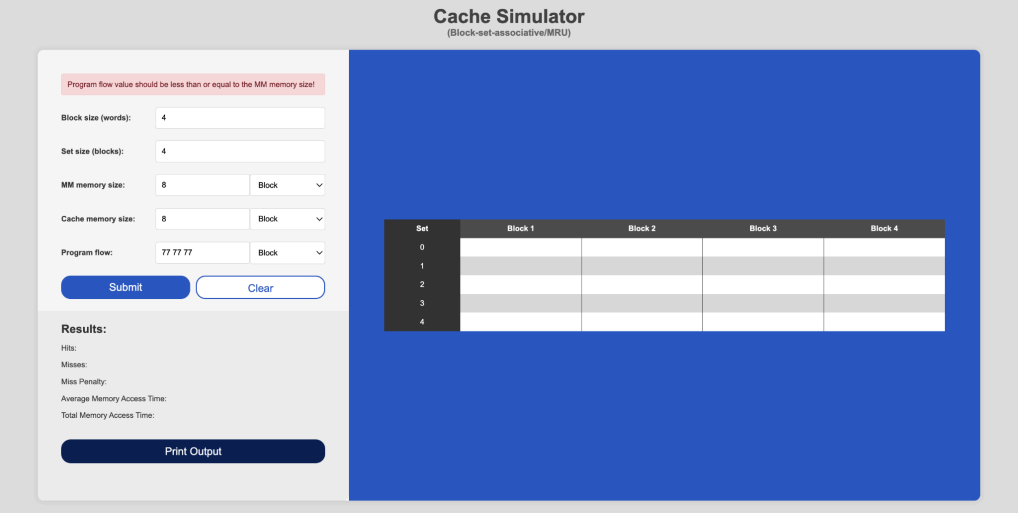
7


Cache Simulator
(Block-set-associative/MRU)
Block size (words): 4
Set size (blocks): 4
MM memory size: 8 Block
Cache memory size: 64 Word
Program flow: 1 2 3 4 5 6 7 8 0 Word
Submit
Clear
Results:
Hits: 0
Misses: 9
Miss Penalty: 42
Average Memory Access Time: 42
Total Memory Access Time: 405
Print Output
Set | Block 1 | Block 2 | Block 3 | Block 4
0 | 1 | 2 | 3 | 0
1 | 4 | 5 | 6 | 7
2 | 8
3
4
5
6
7
8
9
10
11
12
13
14
15

8	 Cache Simulator (Block-set-associative/MRU) Block size (words): 4 Set size (blocks): 4 MM memory size: 16 Block Cache memory size: 8 Block Program flow: 34 3 1 2 4 7 8 9 11 12 Word Submit Clear Results: Hits: 0 Misses: 10 Miss Penalty: 42 Average Memory Access Time: 42 Total Memory Access Time: 450 Print Output Set Block 1 Block 2 Block 3 Block 4 0 34 3 1 11 1 4 7 12
9	 Cache Simulator (Block-set-associative/MRU) Block size (words): 4 Set size (blocks): 4 MM memory size: 16 Block Cache memory size: 8 Word Program flow: 1 2 3 4 4 3 4 3 5 6 Block Submit Clear Results: Hits: 4 Misses: 6 Miss Penalty: 42 Average Memory Access Time: 25.599999999999998 Total Memory Access Time: 286 Print Output Set Block 1 Block 2 Block 3 Block 4 0 2 4 6 1 1 3 5
10	 Cache Simulator (Block-set-associative/MRU) Block size (words): 4 Set size (blocks): 4 MM memory size: 256 Word Cache memory size: 8 Block Program flow: 2 3 5 6 3 2 4 6 Word Submit Clear Results: Hits: 3 Misses: 5 Miss Penalty: 42 Average Memory Access Time: 26.625 Total Memory Access Time: 237 Print Output Set Block 1 Block 2 Block 3 Block 4 0 2 3 1 5 6 4

B. Invalid Input

1	
2	

3	
4	

III. ANALYSIS

The javascript file, *cachesim.js*, contains the cache simulator algorithm. It has four helper functions: *pow2()*, *convbin()*, *getset()*, and *gettag()*. *pow2()* requires a numerical input and checks if the input is a power of 2.

```
function pow2(x){
    return x > 0 && (x & (x - 1)) === 0;
}
```

convbin() has two parameters, where the first parameter is the decimal and the second one is the number of bits needed. It converts the decimal input to binary according to the number of bits stated in the parameter. If the converted binary value's bits are less than the number of bits asked by the second parameter, it pads the binary string with zeros.

```
function convbin(dec, bits){ //covert to binary
    let bin = dec.toString(2);

    while (bin.length < bits){
        bin = '0' + bin;
    }

    if (bin.length > bits){
        bin = bin.slice(-bits);
    }

    return bin;
}
```

getset() requires a binary string and an array that has the values of tag, set, and word. This function gets the set bits of the binary value.

```
function getset(bin, addnum){ //get the set from address
    let first = 0;
    let s = '';

    addnum.forEach(len => {
        let last = first + len;
        s = bin.slice(first, last);
        first = last;
    });

    return s;
}
```

gettag() needs a binary string and the number of tag bits. This function gets the tag bits from the binary value.

```
function gettag(bin, tag){ //get the tag from address
    let first = 0;
    let tt = '';

        let last = first + tag;
        tt = bin.slice(first, last);
        first = last;

    return tt;
}
```

The file has an event handler for when the submit button is pressed. First, it gets the inputs from the text areas. There are three error handlers to make sure the user inputs correct values. First one checks if block size, set size, MM memory size, and cache memory size is a power of two. Second error checker checks if all input values are greater than 0. The last error handler checks if the cache memory size is divisible by set size or cache memory size is greater than set size. If these conditions are violated, it shows an error message and clears the text areas of the potentially incorrect values.

```
document.getElementById("sub").onclick = function(event){
    event.preventDefault();
    block = parseInt(document.getElementById("block").value);
    set = parseInt(document.getElementById("set").value);
    mmn = parseInt(document.getElementById("mmn").value);
    mm = document.getElementById("mm").value;
    cmn = parseInt(document.getElementById("cmn").value);
    cm = document.getElementById("cm").value;
    pmn = document.getElementById("pmn").value;
    pf = document.getElementById("pf").value;
```

```
if(!pow2(block) || !pow2(set) || !pow2(mmn) || !pow2(cmn)){
    alertError("Inputs should be a power of 2!");
    document.getElementById("block").value = '';
    document.getElementById("set").value = '';
    document.getElementById("mmn").value = '';
    return;
}

if(block <= 0 || set <= 0 || mmn <= 0 || cmn <= 0){
    alertError("Inputs should be greater than 0!");
    document.getElementById("block").value = '';
    document.getElementById("set").value = '';
    document.getElementById("mmn").value = '';
    document.getElementById("cmn").value = '';
    return;
}

if(cmn % set || cmn < set){
    alertError("Cache size should be greater than and divisible by set size!");
    document.getElementById("set").value = '';
    document.getElementById("cmn").value = '';
    return;
}
```

pfnarr is an array that contains the blocks or addresses from the program flow. It then checks if the individual block values are less than the MM memory size. If the condition is violated, then it prints out an error message and clears the array and asks the user for a new input.

```
pfnarr = pmn.split(' ').map(Number); // to store in array

for (let i = 0; i < pfnarr.length; i++) {
    let btest = (mm === "bmm") ? mmn * block : mmn;

    if (btest < pfnarr[i]) {
        alertError("Program flow value should be less than or equal to the MM memory size!");
        document.getElementById("pmn").value = '';
        pfnarr = [];
        btest = 0;
        return;
    }
}
```

Next, it enables the print output button to allow the user to export it as a text file. The following blocks of code computes the number of set, tag, and word bits needed. To compute for set, it checks if the user inputs the value in blocks or words. If it is in blocks, it divides cache memory size by set size and stores it into *numset*, which is the total number of sets in the cache. To get the number of set bits, binary logarithm is applied to the quotient of cache memory size divided by set size. This value is stored in *fset*. However, if it is in words, *numset* is equal to the quotient of cache memory size divided by block size. *fset* would be equal to the binary logarithm of *numset* divided by set size. Number of word bits is computed by applying binary logarithm to the block size. To get tag, it checks if the MM memory size is inputted in block or word. If in word, *tag* would be equal to binary logarithm of MM memory size minus set minus word. If in block, *tag* would be equal to the binary logarithm of the product of MM memory size multiplied by block size, then is subtracted to set and word. The number of tag and set bits are then pushed to the array *addnum*, which is needed for function *getset()*.

```
document.getElementById("print").disabled = false;
    let numset; //total sets
    let inds = 0;
    let word;
    let fset;
    let tag;
    let addval;
    let totb;
    let s;
    let addnum = []; //stores tag and set
    let pfpairs = []; //to store value and its set

    if(cm === "cmb"){
        numset = cmn / set;
        fset = Math.log2((cmn / set));
    }
    else if(cm === "cmw"){
        numset = cmn / block;
        fset = Math.log2(((cmn / block) / set));
    }

    word = Math.log2(block);

    if(mm === "wmm"){
        tag = Math.log2(mmn) - fset - word;
    }
    else if(mm === "bmm"){
        tag = Math.log2((mmn * block)) - fset - word;
    }

    addnum.push(tag);
    addnum.push(fset);
```

It will then manipulate the program flow inputs and solve for its set number to know where to store these values in the cache. Initially, it will check if the program flow is in blocks or words (or addresses). If in words, it adds all tag, set, and word and stores it in *totb*, which is the total number of bits. *pfpairs* is an array that contains the value, its set bits, and tag bits. This was done by looping through each value stored in *pfnarr*. While each value is being looped, it converts the value to binary using function *convbin()* and stores it in *addval*, it then gets the set number in binary through function *getset()* and stores it in *s*. This will be converted to a decimal by using *parseInt()* and stores it in *setNumber*. To get the tag bits, function *gettag()* will be called and stores the binary value in *t*. However, if it is in blocks, *totb* will contain the sum of *tag* and *fset*. Moreover, while each value in *pfnarr* is being looped, it gets the binary value and stores it in *addval*, and *setNumber* stores the set number, which can be computed by finding the modulo of the value from the program flow and *numset* or the total number of sets in the cache. *t* stores the tag bits, which is found by using *gettag()* with parameters *addval* and *tag*.

```
//to check if word or block
if (pf === "pfw"){
    totb = tag + fset + word;
    pfpairs = pfnarr.map(num => {
        addval = convbin(num, totb);
        s = getset(addval, addnum);
        setNumber = parseInt(s, 2);
        t = gettag(addval, tag);
        return [num, setNumber, t];
    });
}
else if (pf === "pfb"){
    numset = cmn / set;
    totb = tag + fset;
    pfpairs = pfnarr.map(num => {
        addval = convbin(num, totb);
        let setNumber = num % numset;
        t = gettag(addval, tag);
        return [num, setNumber, t];
    });
}
```

To place the values from *pfpairs* in the cache, arrays *cache* and *lastind* are created. *cache* is a 2D array whose numbers and rows are dictated by *numset* and *set*, respectively. To initialize, *numset* and *set* are used as lengths and each value in the array is **null**. Meanwhile, *lastind* is a 1D array whose size is dictated by *numset* and is filled with **-1**. *cache* stores the value in correct order with respect to its computed *setNumber*. *lastind* stores the last index for each set. Each block number would be stored according to the last set used. For example, if the most recently used set is set 2 and at block 1, the value stored in *lastind[2]* would be 1.

```
let j;
let hit = 0;
let miss = 0;
let value;
let tagval;
//stores the value computed value
let cache = Array.from({ length: numset }, () => Array.from({ length: set }, () => null));
let lastind = Array(numset).fill(-1); //stores the last index used
```

The MRU or Most Recently Used algorithm is used when a set and its blocks have values, and the current program flow needs to be stored in that same set, the current program flow will replace a stored program flow according to the most recently used block in that set. To apply this replacement algorithm, *pfpairs* will be looped for each value. *pfpairs[0]* or *value* stores the program flow inputs, *pfpairs[1]* or *inds* stores its set number, and *pfpairs[2]* or *tagval* stores its tag bits. Another loop is used to access each block in a set. While looping, there are three possible cases that could happen. One is if the value stored in the cache matches a value in the program flow. If this happens, it will increment the hit counter and store the last block in *lastind[inds]*. Another is if the location needed in the cache has a **null** value stored, it would store the program flow in that set and block, records and store the last used block in *lastinds* and increments the miss counter. The last case is if every location in the cache has a value and none of it matches the current program flow, it would store the current program flow in the last used block of the current program flow's *setNumber* and increment the miss counter.

```
pfpairs.forEach(function(pair){ //mru
    value = pair[0]; //get val
    inds = pair[1]; //get set
    tagval = pair[2]; //get tag

    for(j = 0; j < cache[inds].length; j++){ //j is block
        if (cache[inds][j] === value){ //if value is found
            lastind[inds] = j;
            hit++;
            break;
        }
        else if(cache[inds][j] === null){  //if value is null (start)
            cache[inds][j] = value;
            lastind[inds] = j;
            miss++;
            break;
        }
    }

        if (cache[inds].every(val => val != value)){ //if all have value, store in last index
            cache[inds][lastind[inds]] = value;
            miss++;
        }
});
```

Lastly, to solve for the cache and memory access times, hits, misses, and miss penalty, the program assumes that the cache access time *catn* is 1 and memory access time *matn* is 10. Miss penalty is solved by adding the cache memory access time to itself and adding the sum to the product of block size and memory access time. The average memory access time is computer by getting the quotient of hits and the number of program flow inputs. This quotient is multiplied to the cache memory access time. The product is added to the quotient of miss and number of program flow inputs multiplied by miss penalty. To get total memory access time, the product of block, hit, and cache memory access time is added to the product of miss, block, and sum of cache and memory access time. This sum is then added to the product of miss and cache access time.

```
catn = 1;
matn = 10;

let missp = catn + (block * matn) + catn;
let avemmtime = (hit/pfnarr.length)*catn + (miss/pfnarr.length)*missp;
let totmmtime = (hit*block*catn) + (miss*block*(catn+matn)) + (miss*catn);
```

These results are displayed on the website by editing the text contents. Function *populateTable()* helps with the visualization of the cache, creating a table whose number of rows are equal to *numset* and number of columns are equal to *set*. After, it removes any visible error message if there is any.

```
    document.getElementById("hits").textContent = hit;
    document.getElementById("misses").textContent = miss;
    document.getElementById("missp").textContent = missp;
    document.getElementById("avemmtime").textContent = avemmtime;
    document.getElementById("totmmtime").textContent = totmmtime;

    function populateTable(cache) {
        const tableBody = document.getElementById("cacheBody");
        tableBody.innerHTML = "";

        const blockNumbers = cache[0].length;

        const header = document.createElement("tr");
        header.innerHTML = `<th>Set</th>${Array.from({ length: blockNumbers }, (_, i) => `<th>Block ${i + 1}</th>`).join("")}`;
        const tableHead = document.querySelector("#cacheTable thead");
        tableHead.innerHTML = "";
        tableHead.appendChild(header);

        cache.forEach((set, index) => {
            const row = document.createElement("tr");
            row.innerHTML = `<td>${index}</td>${set.map(block => `<td>${block !== null ? block : ''}</td>`).join("")}`;
            tableBody.appendChild(row);
        });
    }


    populateTable(cache); //to display table values

const errorMessage = document.getElementById("error-message");
errorMessage.innerText = '';
errorMessage.classList.remove("visible");
```

To export the results to a text file, function *downloadText()* aids with placing the results to the text file. This is done by getting the text contents of each result and adding them to variable *text*. An event listener is added to handle the logic if the button print is clicked. When clicked, it calls the function *downloadText()* and creates a text file that will be automatically downloaded to the user's device.

```
function downloadText() {
    const hits = document.getElementById("hits").textContent;
    const misses = document.getElementById("misses").textContent;
    const missp = document.getElementById("missp").textContent;
    const avemmtime = document.getElementById("avemmtime").textContent;
    const totmmtime = document.getElementById("totmmtime").textContent;

    let text = `Results:\n\n`;
    text += `Hits: ${hits}\n`;
    text += `Misses: ${misses}\n`;
    text += `Miss Penalty: ${missp}\n`;
    text += `Average Memory Access Time: ${avemmtime}\n`;
    text += `Total Memory Access Time: ${totmmtime}\n\n`;

    text += `Cache State:\n\n`;

    const table = document.getElementById("cacheTable");
    const headers = table.querySelectorAll("thead th");

    let headerText = '';
    const colWidths = [6, 10, 10, 10, 10];
    headers.forEach((header, index) => {
        const headerContent = header.textContent.trim();
        if (index === 0) {
            headerText += headerContent.padEnd(colWidths[index] + 3);
        } else {
            headerText += headerContent.padEnd(colWidths[index]);
        }
    });
    text += headerText + '\n'; //header row
    text += '-'.repeat(headerText.length) + '\n'; // line separator
```

```
    const rows = table.querySelectorAll("tbody tr");
    rows.forEach((row) => {
        const cells = row.querySelectorAll("td");
        let rowText = '';
        cells.forEach((cell, cellIndex) => {
            let cellContent = cell.textContent.trim() === '' ? '---' : cell.textContent;
            if (cellIndex === 0) {
                rowText += cellContent.padEnd(colWidths[cellIndex]) + '| ';
            } else {
                const padding = (colWidths[cellIndex] - cellContent.length) / 2;
                rowText += cellContent.padStart(Math.floor(padding) + cellContent.length).padEnd(colWidths[cellIndex]);
            }
        });
        text += rowText.trim() + '\n'; // row content
    });

    return text;
}
```

```
document.getElementById("print").addEventListener("click", function() {
    const textContent = downloadText();
    const blob = new Blob([textContent], { type: 'text/plain' });
    const url = URL.createObjectURL(blob);

    const a = document.createElement("a");
    a.href = url;
    a.download = "cachesimresult.txt";  // name of the file
    a.click();

    URL.revokeObjectURL(url);
});
```

A clear button is created for when the user wants to clear their inputs. When this button is clicked, the event handler will clear all the text areas and return the drop down to “blocks.” It also clears all the variables and arrays to prevent mistakes for the next run. It also returns the visual table on the frontend to its default form and disables the print button.

```
document.getElementById("clear").onclick = function(event){
    event.preventDefault();
    document.getElementById("block").value = '';
    document.getElementById("set").value = '';
    document.getElementById("mmn").value = '';
    document.getElementById("cmn").value = '';
    document.getElementById("pmn").value = '';
    document.getElementById("mm").value = "bmm";
    document.getElementById("cm").value = "cmb";
    document.getElementById("pf").value = "pfb";
    block = 0;
    set = 0;
    mmn = 0;
    cmn = 0;
    pmn = 0;
    pfnarr = [];
    addnum = [];
    pfpairs = [];
    inds = 0;
    hit = 0;
    miss = 0;
    value = 0;
    numset = 0;
    document.getElementById("hits").textContent = '';
    document.getElementById("misses").textContent = '';
    document.getElementById("missp").textContent = '';
    document.getElementById("avemmtime").textContent = '';
    document.getElementById("totmmtime").textContent = '';
    document.getElementById("print").disabled = true;
```

```
const defaultRow = `
    <tr>
                    <th>Set</th>
                    <th>Block 1</th>
                    <th>Block 2</th>
                    <th>Block 3</th>
                    <th>Block 4</th>
                </tr>
`;

const defaultBod = `
    <tr><td>0</td><td></td><td></td><td></td><td></td></tr>
    <tr><td>1</td><td></td><td></td><td></td><td></td></tr>
    <tr><td>2</td><td></td><td></td><td></td><td></td></tr>
    <tr><td>3</td><td></td><td></td><td></td><td></td></tr>
    <tr><td>4</td><td></td><td></td><td></td><td></td></tr>
`;
document.getElementById('head').innerHTML = defaultRow;
document.getElementById('cacheBody').innerHTML = defaultBod;
```

IV. CONCLUSION

Caches allow for the bridging between the main memory and CPU. It allows for data that is frequently accessed to be more readily available. That is why using the block-set-associative mapping method in tandem with the MRU algorithm is an efficient method in mapping the cache. Block-set-associative is the midpoint of directly mapped and fully associative which balances both methods with simplicity while also reducing the misses in the cache. The simulator showcases this mapping method alongside the MRU replacement algorithm to fully illustrate the different process that goes on within the cache. Overall, the simulation project was helpful in illustrating how the cache works with different test conditions.