# A Simulator for Mano's Basic Computer

Armin Atarod

## 1. Introduction

The simulator is a program based on the book "Computer System Architecture" authored by M. Morris Mano. It has been developed using the "Ultralight" framework[1]. The backend of the program is written in C++ and the frontend is written in HTML/CSS/JavaScript.

## 2. Ultralight

Ultralight is a framework that enables developers to write desktop applications and games using C++ and a web core. The core uses the WebKit SDK (which powers Safari on Mac and IOS). The source can be built using CMake, which enables the program to be supported on all major platforms including Windows, Linux and MacOS.

The build command used for this program is:

```
cmake --build build --config Release --clean-first
```

## 3. GUI

The GUI is written in a single file named "app.html". The styling is located in the style tag in the header section of the HTML which is written in CSS.

In the body, there are 3 tables which (from left to right when rendered) display the assembly code, the memory data and the register values.

Under these tables there are 3 buttons on the left and 3 buttons on the right. The buttons on the left are used to assemble the code, load a txt, and save as a txt file. The buttons on the right are used to execute the next line, go back to the previous state and execute all lines.

[1] https://ultralig.ht/

The execute all button simply repeats the execute next function until the computer halts. The previous button restores the values the registers (and the memory) had before the last execution happened.

On the bottom of the page, there is a message log that is used to display the errors and success/finished messages when assembling the code or executing the lines.


A screenshot of the simulator

The memory table has $4096 = 2^{12}$ rows and the code table on the left has 5000 rows (the code might have many ORG instructions, so it should have more than 4096 rows).

The last line in the memory table that has been executed is highlighted in green. The scrollbar follows the highlighted line so that it always remains visible in the current view.

The register table displays the value of each register the computer has. The value of each register is written in front of it in binary, and on the right in HEX format. Below each number is the value it previously had before the last execution.

The values of the FGI, FGO and INPR registers are given by the user, so the user can type their values in the corresponding box.

All dimension values in the CSS section are given in the “vw” unit (viewport width), which means the content will always retain its position and width based on the current window width, regardless of how the window has been resized. If the height of the window becomes less than the

contents' height, a scrollbar will appear on the right which allows the user to navigate up and down the page.

```
<body>
    <div class="code">
        <table>
            <tr id="codeTableHeader">
                <th>Line</th>
                <th>Label</th>
                <th>Instruction</th>
                <th>Comment</th>
            </tr>
        </table>
    </div><!--
    --><script type="text/javascript">
        //Initialize the code table with 5000 rows
        var html = "";
        for (var i = 0; i < 5000; i++) {
            html += "<tr class=\"codeRow\" style=\"border-top: solid 0.001vw rgb(200, 200, 200);\">";
            html += "<td class=\"rowLine\">" + i.toString().toUpperCase() + "</td>";
            html += "<td class=\"rowLabel\"><input type=\"text\" class=\"rowLabelInput\"></td>";
            html += "<td class=\"rowInstruction\"><input type=\"text\" class=\"rowInstructionInput\"></td>";
            html += "<td class=\"rowComment\"><input type=\"text\" class=\"rowCommentInput\"></td>";
            html += "</tr>";
        }
        document.getElementById("codeTableHeader").insertAdjacentHTML("afterend", html);
    </script><!--
    --><div class="memory">
        <table>
            <tr id="memoryTableHeader">
                <th>Line</th>
                <th>Data</th>
            </tr>
        </table>
    </div><!--
    --><script type="text/javascript">
        //Initialize the memory table with 2^12 rows
        var html = "";
        for (var i = 0; i < 4096; i++) {
            html += "<tr class=\"memoryRow\" style=\"border-top: solid 0.001vw rgb(200, 200, 200);\">";
            html += "<td class=\"rowLine\">" + i.toString(16).toUpperCase() + "</td>";
            html += "<td class=\"rowData\"></td>";
            html += "</tr>";
        }
        document.getElementById("memoryTableHeader").insertAdjacentHTML("afterend", html);
    </script><!--
```

The beginning of the body, with the declaration and initialization of the code table and the memory table

```
</script><!--
--><div class="variables">
    <table>
        <tr style="color: rgb(255, 0, 0);">
            <td>IR</td>
            <td id="IR"></td>
            <td id="IRHEX"></td>
        </tr>
        <tr style="color: rgb(255, 0, 0);">
            <td></td>
            <td id="preIR"></td>
            <td id="preIRHEX"></td>
        </tr>
        <tr style="color: rgb(255, 0, 0);">
            <td>I</td>
            <td id="I"></td>
            <td id="IHEX"></td>
        </tr>
        <tr style="color: rgb(255, 0, 0);">
            <td></td>
            <td id="preI"></td>
            <td id="preIHEX"></td>
        </tr>
        <tr style="color: rgb(30, 120, 160);">
            <td>AC</td>
            <td id="AC"></td>
            <td id="ACHEX"></td>
        </tr>
        <tr style="color: rgb(30, 120, 160);">
            <td></td>
            <td id="preAC"></td>
            <td id="preACHEX"></td>
        </tr>
        <tr style="color: rgb(150, 90, 40);">
            <td>DR</td>
            <td id="DR"></td>
            <td id="DRHEX"></td>
        </tr>
        <tr style="color: rgb(150, 90, 40);">
            <td></td>
```

The declaration and initialization of the register table

```
</script>
<button style="background-color: rgb(10, 130, 10);" onclick="assemble()">Assemble</button>
<button style="background-color: rgb(30, 120, 160);" onclick="showLoadFile()">Load from txt</button>
<button style="background-color: rgb(30, 120, 160);" onclick="showSaveFile()">Save to txt</button>
<button style="background-color: rgb(150, 0, 0);" class="rightToLeft" onclick="executeAll()">Execute all</button>
<button style="background-color: rgb(200, 80, 0);" class="rightToLeft" onclick="executeNext()">Execute next</button>
<button style="background-color: rgb(150, 150, 100);" class="rightToLeft" onclick="previousState()">Previous</button>
<p id="log"></p>
<div style="width: 100%; height: 5vw;"></div>
```

The declaration and initialization of the buttons and the message log

At the end of the HTML code, there is an extra div which adds a padding of 5vw to the bottom of the page. This is to ensure a symmetrical distance from the top of the page and from the bottom of the page.

# 4. JavaScript/C++ binding

The binding of JavaScript and C++ is done with the help of Ultralight. The JavaScript functions that are to be called from C++ are defined in the OnDOMReady section of the MyApp.cpp file. In our case, these are assemble(), executeNext(), previousState(), showLoadFile(), and showSaveFile().

```
void MyApp::OnDOMReady(ultralight::View* caller, uint64_t frame_id, bool is_main_frame, const String& url) {
    auto scoped_context = caller->LockJSContext();
    JSContextRef ctx = (*scoped_context);
    JSObjectRef globalObj = JSContextGetGlobalObject(ctx);

    //Define the javascript functions that will be executed using c++

    JSStringRef name1 = JSStringCreateWithUTF8CString("assemble");
    JSObjectRef func1 = JSObjectMakeFunctionWithCallback(ctx, name1, assemble);

    JSObjectSetProperty(ctx, globalObj, name1, func1, 0, 0);

    JSStringRef name2 = JSStringCreateWithUTF8CString("executeNext");
    JSObjectRef func2 = JSObjectMakeFunctionWithCallback(ctx, name2, execute_next);

    JSObjectSetProperty(ctx, globalObj, name2, func2, 0, 0);

    JSStringRef name3 = JSStringCreateWithUTF8CString("previousState");
    JSObjectRef func3 = JSObjectMakeFunctionWithCallback(ctx, name3, previous_state);

    JSObjectSetProperty(ctx, globalObj, name3, func3, 0, 0);

    JSStringRef name4 = JSStringCreateWithUTF8CString("showLoadFile");
    JSObjectRef func4 = JSObjectMakeFunctionWithCallback(ctx, name4, show_load_file);

    JSObjectSetProperty(ctx, globalObj, name4, func4, 0, 0);

    JSStringRef name5 = JSStringCreateWithUTF8CString("showSaveFile");
    JSObjectRef func5 = JSObjectMakeFunctionWithCallback(ctx, name5, show_save_file);

    JSObjectSetProperty(ctx, globalObj, name5, func5, 0, 0);

    JSStringRelease(name1);
    JSStringRelease(name2);
    JSStringRelease(name3);
    JSStringRelease(name4);
    JSStringRelease(name5);
}
```

The OnDOMReady section with the 5 JavaScript functions linked to C++ functions

The scripts that are written in C++ and must be run in JavaScript also use a command given by Ultralight that runs the script globally in JavaScript.

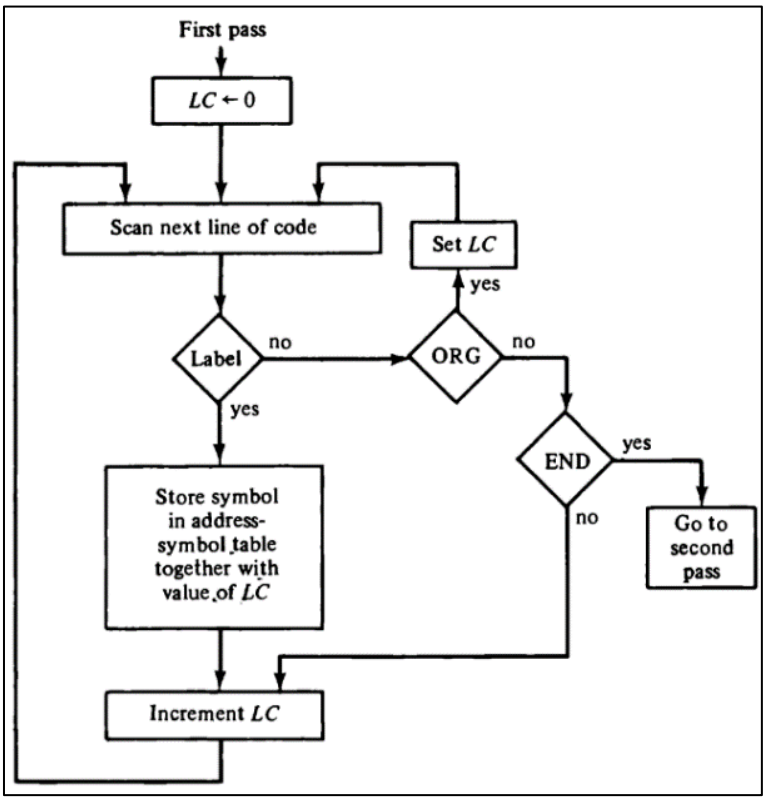
```
command = "log.innerHTML = 'File successfully saved.';log.style.color = 'rgb(10, 110, 10)';";
script = JSStringCreateWithUTF8CString(command.c_str());
JSEvaluateScript(ctx, script, 0, 0, 0, 0);
```

A JavaScript script run from the C++ side

# 5. The Assembler

The assembler function implements the procedure described in chapter six of the book. Each section of the code is explained in the comment(s) written above it. An error string is kept throughout the loop which stores the first error the assembler encounters. The assembler has two passes. In the first pass, the assembler finds the labels and puts them in a table called the Symbolic Address Table. In second pass, the assembler goes through each line of the code and converts the instruction into machine code that will be put in the memory table. After these two passes are finished, the machine code will be ready for execution.

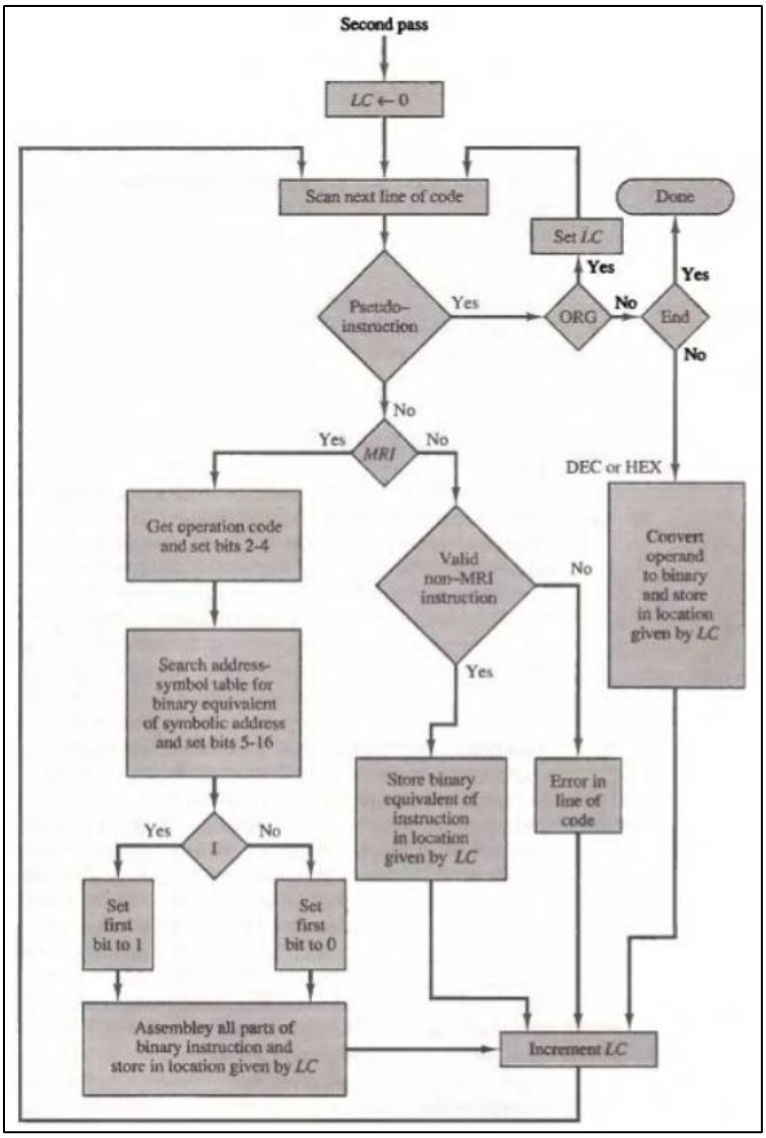
The first assembly will take some time (a few seconds), but the next assemblies will be almost instant. This is due to JavaScript taking a long time to write the data in the memory table.




The first pass of the assembler

```
//Run the first pass of the assembler + Error detection
int lc = -1;
for (int i = 0; i < 5000; i++) {
    lc++;
    if (!comments[i].empty()) {
        if (comments[i][0] != '/') {
            if (error_text.empty())
                error_text = "Line " + std::to_string(i) + ": Comments must start with \\'/\\'.";
            break;
        }
    }
    if (!labels[i].empty()) {
        if (labels[i].back() != ',') {
            if (error_text.empty())
                error_text = "Line " + std::to_string(i) + ": Labels must end with \\',\\'.";
            break;
        }
        if (instructions[i] == "END" || (instructions[i].size() > 3 && instructions[i].substr(0, 3) == "ORG")) {
            if (error_text.empty())
                error_text = "Line " + std::to_string(i) + ": ORG and END instructions must not have labels.";
            break;
        }
        else if (labels[i].size() > 4 || has_non_alphanumeric(labels[i].substr(0, labels[i].size() - 1))) {
            if (error_text.empty())
                error_text = "Line " + std::to_string(i) + ": Invalid label.";
            break;
```

Part of the first pass loop



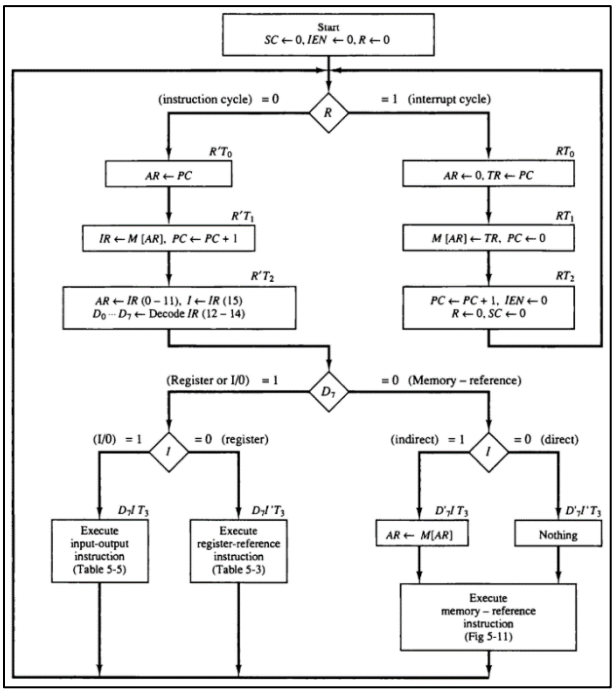

The second pass of the assembler

```cpp
//Run the second pass of the assembler + Error detection
lc = -1;
for (int i = 0; i < 5000; i++) {
    if (labels[i].empty() && instructions[i].empty())
        continue;
    lc++;
    if (instructions[i].size() < 3 || has_non_alphanumeric_or_space(instructions[i])) {
        if (error_text.empty())
            error_text = "Line " + std::to_string(i) + ": Invalid instruction.";
        break;
    }
    else {
        std::string instruction = instructions[i].substr(0, 3);
        if (instruction == "END")
            break;
        else if (lc >= 4096) {
            if (error_text.empty())
                error_text = "Line " + std::to_string(i) + ": LC exceeded 4095.";
            break;
        }
        else if (instruction == "ORG") {
            if (instructions[i].size() < 5 || instructions[i][3] != ' ' || has_non_alphanumeric(instructions[i].substr(4))) {
                if (error_text.empty())
                    error_text = "Line " + std::to_string(i) + ": Invalid ORG instruction";
                break;
            }
            else {
                int org_line = std::stoi(instructions[i].substr(4), nullptr, 16);
                lc = org_line - 1;
            }
        }
        else if (instruction == "HEX") {
            if (instructions[i].size() < 5 || instructions[i][3] != ' ') {
                if (error_text.empty())
                    error_text = "Line " + std::to_string(i) + ": Invalid HEX instruction.";
                break;
```

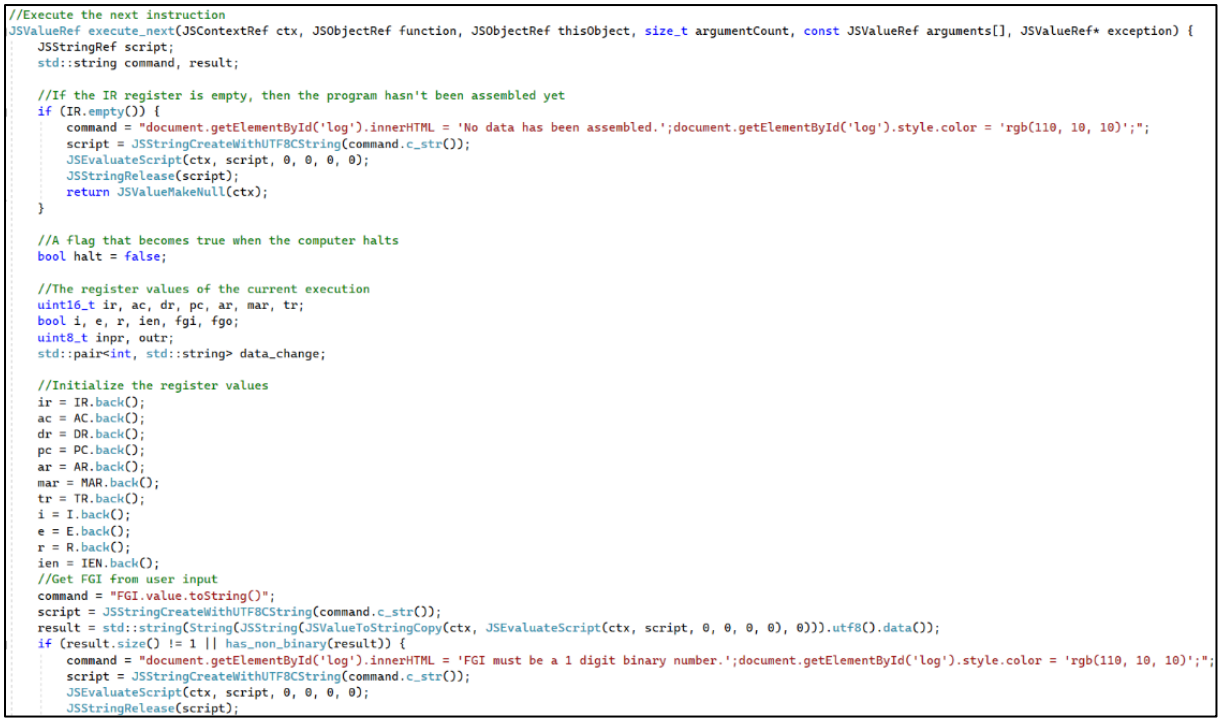
Part of the second pass loop

# 7. Instruction execution

The execution function implements the procedure described in chapter five of the book. In each cycle, the values of the 15 registers are stored in temporary variables, then the micro-operations are executed in each step, and then the final value of these temporary variables are stored in their corresponding vector. These vectors keep track of the changes each register has after each cycle. The details of this implementation are given in the comment(s) above each section of the code.

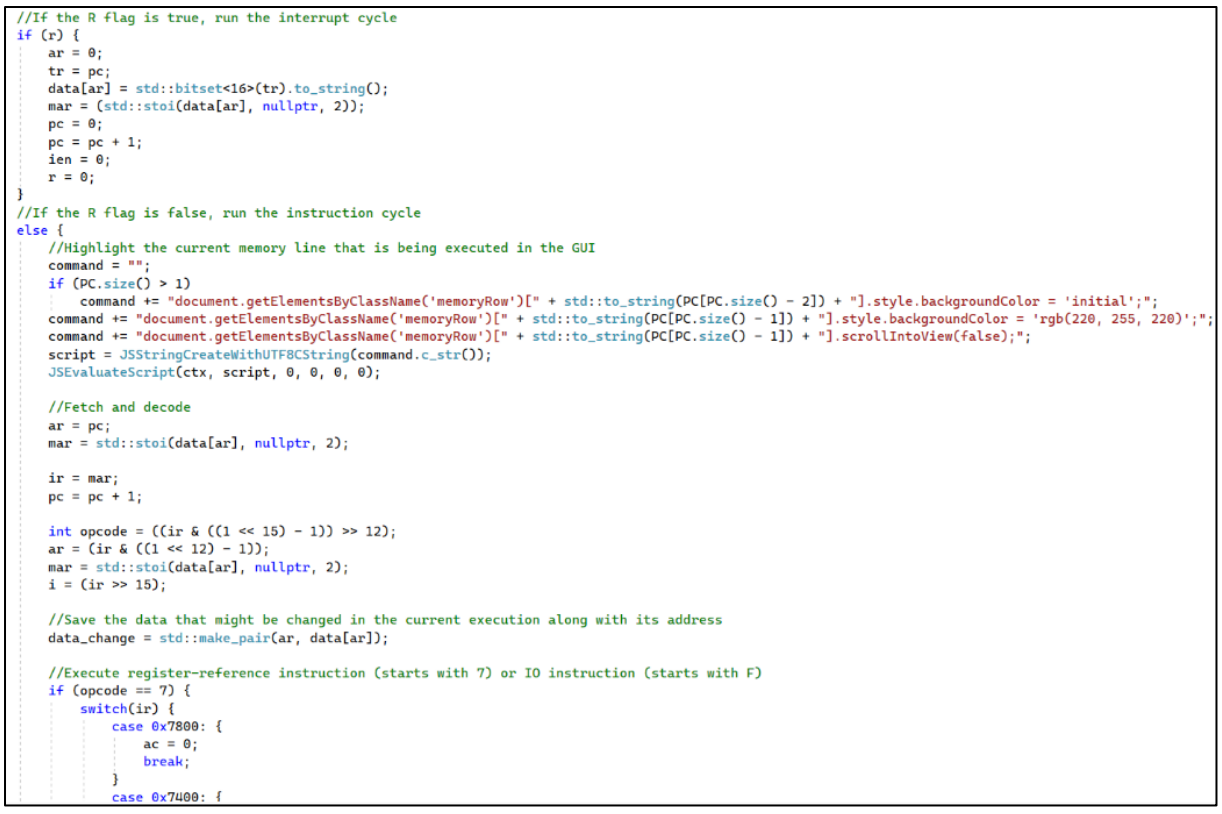



The computer execution cycle



```
//Execute the next instruction
JSValueRef execute_next(JSContextRef ctx, JSObjectRef function, JSObjectRef thisObject, size_t argumentCount, const JSValueRef arguments[], JSValueRef* exception) {
    JSStringRef script;
    std::string command, result;

    //If the IR register is empty, then the program hasn't been assembled yet
    if (IR.empty()) {
        command = "document.getElementById('log').innerHTML = 'No data has been assembled.';document.getElementById('log').style.color = 'rgb(110, 10, 10)';";
        script = JSStringCreateWithUTF8CString(command.c_str());
        JSEvaluateScript(ctx, script, 0, 0, 0, 0);
        JSStringRelease(script);
        return JSValueMakeNull(ctx);
    }

    //A flag that becomes true when the computer halts
    bool halt = false;

    //The register values of the current execution
    uint16_t ir, ac, dr, pc, ar, mar, tr;
    bool i, e, r, ien, fgi, fgo;
    uint8_t inpr, outr;
    std::pair<int, std::string> data_change;

    //Initialize the register values
    ir = IR.back();
    ac = AC.back();
    dr = DR.back();
    pc = PC.back();
    ar = AR.back();
    mar = MAR.back();
    tr = TR.back();
    i = I.back();
    e = E.back();
    r = R.back();
    ien = IEN.back();
    //Get FGI from user input
    command = "FGI.value.toString()";
    script = JSStringCreateWithUTF8CString(command.c_str());
    result = std::string(String(JSString(JSValueToStringCopy(ctx, JSEvaluateScript(ctx, script, 0, 0, 0, 0), 0))).utf8().data());
    if (result.size() != 1 || has_non_binary(result)) {
        command = "document.getElementById('log').innerHTML = 'FGI must be a 1 digit binary number.';document.getElementById('log').style.color = 'rgb(110, 10, 10)';";
        script = JSStringCreateWithUTF8CString(command.c_str());
        JSEvaluateScript(ctx, script, 0, 0, 0, 0);
        JSStringRelease(script);
```

The beginning of the execution function where the register vectors are initialized



```
//If the R flag is true, run the interrupt cycle
if (r) {
    ar = 0;
    tr = pc;
    data[ar] = std::bitset<16>(tr).to_string();
    mar = (std::stoi(data[ar], nullptr, 2));
    pc = 0;
    pc = pc + 1;
    ien = 0;
    r = 0;
}
//If the R flag is false, run the instruction cycle
else {
    //Highlight the current memory line that is being executed in the GUI
    command = "";
    if (PC.size() > 1)
        command += "document.getElementsByClassName('memoryRow')[" + std::to_string(PC[PC.size() - 2]) + "].style.backgroundColor = 'initial';";
    command += "document.getElementsByClassName('memoryRow')[" + std::to_string(PC[PC.size() - 1]) + "].style.backgroundColor = 'rgb(220, 255, 220)';";
    command += "document.getElementsByClassName('memoryRow')[" + std::to_string(PC[PC.size() - 1]) + "].scrollIntoView(false);";
    script = JSStringCreateWithUTF8CString(command.c_str());
    JSEvaluateScript(ctx, script, 0, 0, 0, 0);

    //Fetch and decode
    ar = pc;
    mar = std::stoi(data[ar], nullptr, 2);

    ir = mar;
    pc = pc + 1;

    int opcode = ((ir & ((1 << 15) - 1)) >> 12);
    ar = (ir & ((1 << 12) - 1));
    mar = std::stoi(data[ar], nullptr, 2);
    i = (ir >> 15);

    //Save the data that might be changed in the current execution along with its address
    data_change = std::make_pair(ar, data[ar]);

    //Execute register-reference instruction (starts with 7) or IO instruction (starts with F)
    if (opcode == 7) {
        switch(ir) {
            case 0x7800: {
                ac = 0;
                break;
            }
            case 0x7400: {
```

The main cycle where the IO cycle and Instruction cycle become separated

```
//Check the conditions for the R flag
r = ien & (fgo | fgi);
//If the computer has halted, then the PC register shouldn't be incremented
if (halt)
    pc = pc - 1;
//If the computer hasn't halted or the PC register has changed, store the new register values
//The PC register might change while the computer has halted when the Execute next button is pressed once more after it has halted
if (!halt || PC[PC.size() - 2] != PC.back()) {
    IR.push_back(ir);
    AC.push_back(ac);
    DR.push_back(dr);
    PC.push_back(pc);
    AR.push_back(ar);
    MAR.push_back(mar);
    TR.push_back(tr);
    I.push_back(i);
    E.push_back(e);
    R.push_back(r);
    IEN.push_back(ien);
    FGI.push_back(fgi);
    FGO.push_back(fgo);
    INPR.push_back(inpr);
    OUTR.push_back(outr);
    DATA_CHANGE.push_back(data_change);
}

//Update the register values in the GUI
refreshVariablesCommand(command);

script = JSStringCreateWithUTF8CString(command.c_str());
JSEvaluateScript(ctx, script, 0, 0, 0, 0);

//If the computer has halted, display a finish message
if (halt) {
    command = "log.innerHTML = 'Execution finished.';log.style.color = 'rgb(10, 110, 10)';";
    command += "finished = true;";
}
//If not, clear the message log
else
    command = "log.innerHTML = '';";
script = JSStringCreateWithUTF8CString(command.c_str());
JSEvaluateScript(ctx, script, 0, 0, 0, 0);
```

The end of the execution function where the new values are inserted into the register vectors and the computer halts and the program finishes

# 8. Previous state implementation

The previous state function is implemented using the register vectors. The last elements of the vectors get popped out, and the old last elements become the values of the registers. At most one line of data in the memory can be changed in each execution, so the old address-data pair that is kept in the DATA_CHANGE vector should replace the new data in the memory.

```
//Go to the previous state
JSValueRef previous_state(JSContextRef ctx, JSObjectRef function, JSObjectRef thisObject, size_t argumentCount, const JSValueRef arguments[], JSValueRef* exception) {
    JSStringRef script;
    std::string command, result;

    //If the IR vector (or any other register vector) has less than 3 elements, then a previous state doesn't exist
    if (IR.size() <= 2) {
        command = "document.getElementById('log').innerHTML = 'No previous state exists.';document.getElementById('log').style.color = 'rgb(110, 10, 10)';";
        script = JSStringCreateWithUTF8CString(command.c_str());
        JSEvaluateScript(ctx, script, 0, 0, 0, 0);
        JSStringRelease(script);
        return JSValueMakeNull(ctx);
    }

    //Clear the message log
    command = "document.getElementById('log').innerHTML = '';";
    //Remove the highlight from the current memory line
    command += "document.getElementsByClassName('memoryRow')[" + std::to_string(PC[PC.size() - 2]) + "].style.backgroundColor = 'initial';";

    //Remove the last element from each of the register vectors
    IR.pop_back();
    AC.pop_back();
    DR.pop_back();
    PC.pop_back();
    AR.pop_back();
    MAR.pop_back();
    data[DATA_CHANGE.back().first] = DATA_CHANGE.back().second;
    DATA_CHANGE.pop_back();
    TR.pop_back();
    I.pop_back();
    E.pop_back();
    R.pop_back();
    IEN.pop_back();
    FGI.pop_back();
    FGO.pop_back();
    INPR.pop_back();
    OUTR.pop_back();

    //Add the highlight to the previous memory line
    command += "document.getElementsByClassName('memoryRow')[" + std::to_string(PC[PC.size() - 2]) + "].style.backgroundColor = 'rgb(220, 255, 220)';";
    command += "document.getElementsByClassName('memoryRow')[" + std::to_string(PC[PC.size() - 2]) + "].scrollIntoView(false);";
    script = JSStringCreateWithUTF8CString(command.c_str());
    JSEvaluateScript(ctx, script, 0, 0, 0, 0);
```

Part of the previous state function with the vectors changing

# 9. Load from file

The load from txt button, opens a dialog box for the user to select a txt file and then loads the selected file onto the code table. To show the file selection dialog box, a command is run in the default shell of the operating system. After the dialog box opens, the user may either select a file or cancel. If the user cancels, the code table remains unchanged and the message log says the user has cancelled. If the user selects a file, the file will be decoded and displayed on the code table.

```cpp
//A function that executes a bash command in the OS and returns the result
std::string exec(const char* cmd) {
    std::string result = "";
    std::array<char, 128> buffer;
    FILE* pPipe = _popen(cmd, "r");
    if (pPipe == NULL)
        return "***Failed***";
    while (fgets(buffer.data(), 128, pPipe) != NULL)
        result += buffer.data();
    int endOfFileVal = feof(pPipe);
    int closeReturnVal = _pclose(pPipe);
    if (!endOfFileVal)
        return "***Failed***";
    //Remove the \n at the end
    result.pop_back();
     //For windows operating systems
    #if defined(_WIN32) || defined(_WIN64)
    if (result == "Cancel\n") //Remove the other \n at the end
        result.pop_back();
    else //Remove the OK and \n at the beginning
        result.erase(0, 3);
    //For linux operating systems
    #else
    if (result == "")
        return "Cancel";
    #endif
    return result;
}
```

The C++ function that executes the shell commands and returns the result

For Windows, the default shell is CMD and the dialog box is opened through a PowerShell command. So first a PowerShell script is written that opens a dialog box with only the .txt format being allowed, then a CMD command is executed that runs this script. A "-WindowStyle hidden" is added to the CMD command to hide the new window that opens along with PowerShell (although a brief flashing still occurs).

```powershell
[System.Reflection.Assembly]::LoadWithPartialName("System.windows.forms") | out-null

$OpenFileDialog = New-Object System.Windows.Forms.OpenFileDialog
$OpenFileDialog.initialDirectory = $initialDirectory
$OpenFileDialog.filter = "Text file (*.txt)|*.txt"
$OpenFileDialog.ShowDialog()
$OpenFileDialog.filename
```

The PowerShell script for opening a file selector dialog

The CMD command to run the PowerShell script:

```
powershell -WindowStyle hidden -executionpolicy bypass -file openfiledialog-txt.ps1
```

For Linux, Zenity (which is pre-installed on most standard Linux distributions) allows us to open a dialog box through the terminal, so no external script is needed.

The terminal command:

```
zenity --file-selection
```

After the address has been chosen, the lines of the file are read one by one. In each line, the comment section (if available) will be separated by the first slash (/) in the line. The label section (if available) will be separated by the first comma (,). The middle part will be the instruction itself. Before these separations happen, all illegal characters and double spaces and empty lines are removed. Then if the number of lines exceeds 5000, an error will be displayed; If not, the lines will be placed on the code table in the GUI.

## 10. Save to file

The save to txt button, opens a dialog box for the user to select a location for the file to be saved and then saves the code that is being displayed on the code table in the selected location.

The commands and functions that are used for opening the dialog box are similar to the commands used for loading a file in the previous section. The PowerShell script is slightly changed to open a save file dialog instead of an open file dialog and the Linux command has a "`--save`" added at the end for the same reason.

```
1  [System.Reflection.Assembly]::LoadWithPartialName("System.windows.forms") | out-null
2
3  $SaveFileDialog = New-Object System.Windows.Forms.SaveFileDialog
4  $SaveFileDialog.initialDirectory = $initialDirectory
5  $SaveFileDialog.filter = "Text file (*.txt)|*.txt"
6  $SaveFileDialog.CheckPathExists = $true
7  $SaveFileDialog.CreatePrompt = $true
8  $SaveFileDialog.OverwritePrompt = $true
9  $SaveFileDialog.ShowDialog()
10 $SaveFileDialog.FileName
```

The PowerShell script for opening a file save dialog

The content of the file that is being saved is the non-empty lines of the code table in the GUI. The label, instruction and comment sections are separated with the comma and slash characters and they are spaced using tab characters to ensure all sections are neatly positioned vertically under each other.

```
        ORG 1   /Origin of program set to 1
        LDA A   /Load A in AC
        ADD B   /Add B to AC
        STA SUM /Store A+B in SUM
        HLT     /Halt
A,      HEX F   /A = 15
B,      DEC -5  /B = -5
SUM,    HEX 0   /SUM is stored here
        END     /End of program
```

An example of a file saved by the program

# 11. Appacker for Windows

In Windows, the Release folder contains the main .exe file and the assets and .dll files the program requires to run. To pack all these files into a single executable, we use an open-source program called Appacker[2]. To pack an executable, open the program directory, specify the main .exe file, choose an icon and where it will be saved, and click pack. The result will be a single executable with the icon provided.


The single executable file for this simulator

[2] https://github.com/SerGreen/Appacker/releases