Hashemite University

Faculty of Engineering and Technology

Computer Engineering Department

System Programming

Issa Raouf Qandah **2036177**

Hassan Taqi Eddin **2036057**

Ammar Rajeh Zeidan **1934002**

Thaer Mohammed Eid **2035027**

Dr. Islam Al-Malkawi

SIC / XE SIMULATOR

Table of Contents:

Introduction:

In this project we use C++ to implement the simulator for SIC/XE processor ,the input is the memory and the output the final state of the register and the modified memory location .

Introduction about the SIC and SIC/XE:

What is the difference between these two models :

- **The first different is: in the MEMORY where the SIC has a memory size =bytes =32K ,and the SIC/XE has a memory size =bytes=1 M.**

  But the similarity in MEMORY is : each of model has the memory consists of 8-bit byte, the WORD=3BYTES=24BIT and all address are byte addressable.

SIC	SIC/XE
• Memory consists of 8-bit bytes • A word is three bytes (i.e. 24-bits), addressed by lowers number byte • All address are byte addressable • System memory size = 2^{15} bytes (32K)	Same as SIC with the exception: System memory size = 2^{20} bytes (1 Megabyte)

The second different is :

A hypothetical computer that includes the hardware features most often found on real machines, the SIC has two versions :first one SIC the standard model , the second one is SIC/XE extension model .

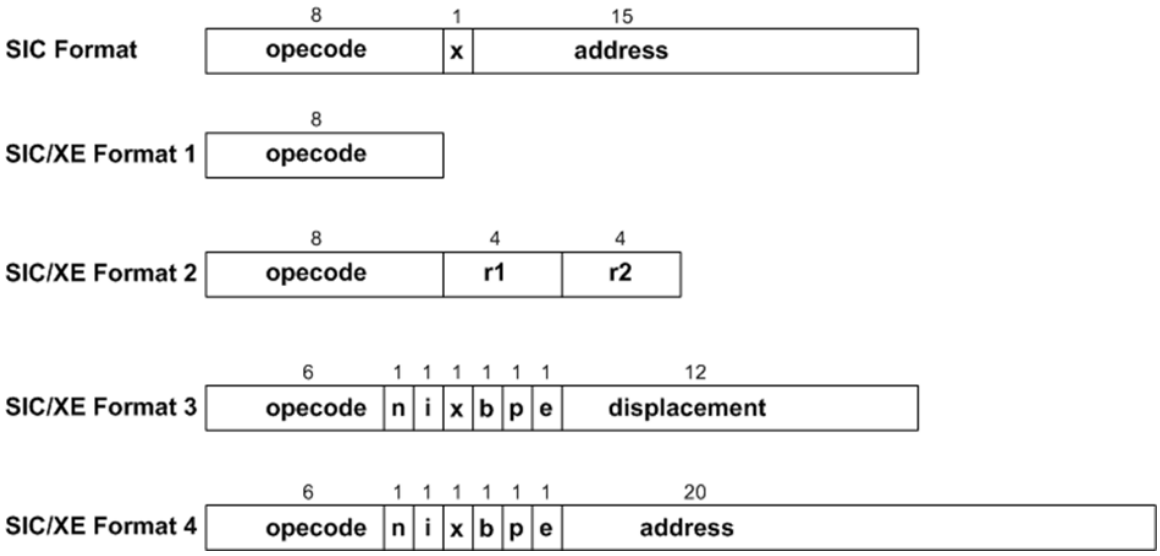
<table>
<tr><th>SIC</th><th>SIC/XE</th></tr>
<tr><td>• Five registers, each is 1-word (i.e. 24-bits)
• The registers are:
<table>
<tr><th>Reg</th><th>Num</th><th>Usage</th></tr>
<tr><td>A</td><td>0</td><td>Accumulator</td></tr>
<tr><td>X</td><td>1</td><td>Index Reg</td></tr>
<tr><td>L</td><td>2</td><td>Linkage Reg</td></tr>
<tr><td>PC</td><td>8</td><td>ProgCounter</td></tr>
<tr><td>SW</td><td>9</td><td>Status Reg</td></tr>
</table></td><td>• There are 4 additional registers
<table>
<tr><th>Reg</th><th>Num</th><th>Usage</th></tr>
<tr><td>B</td><td>3</td><td>Base Reg</td></tr>
<tr><td>S</td><td>4</td><td>GPR</td></tr>
<tr><td>T</td><td>5</td><td>GPR</td></tr>
<tr><td>F</td><td>6</td><td>FP Accumulator</td></tr>
</table></td></tr>
</table>

Note : any program written in SIC should run on SIC/XE.

- **The third different is: in DATA FORMATS where :**
  - **in the SIC are 24 bit integers in 2's complement ,8 bit character and the floating point are not supported .**
  - **but in SIC/XE supported two word floating point data 48 bit in the following format:**

S	Exponent	Fraction

Where : S bit[47] bit is a sign bit 0 for (+) and 1 for (-)



Note: ***SIC/XE formats 1 and 2 don't access memory and formats 3 is very similar to SIC formats .***

****What is the (n,I,x,b,p,e)??these are called FLAGS so now what is the flags bit description.effected on cc.**

FLAG Bit	Description
n,i	(n,i) = (0,1) → immediate addressing (n,i) = (1,0) → indirect addressing (n,i) = (0,0) or (1, 1) → simple addressing
x	x=1 indicates index addressing TA= (x) + address
p	p=1 indicates PC relative addressing TA = (PC) + disp -2048 ≤ disp ≤ 2047
b	b=1 indicates base-relative addressing TA= (B) + disp 0 ≤ disp ≤ 4095
e	Format-3 (e=0) or Format-4 (e=1)

- **Registers :**

• Mnemonic	Number	Special Use
A ; Accumulator	0	used for arithmetic operations
X : Index register	1	used for addressing
L : Linkage register	2	the jump to subroutine (JSUB) instruction stores the return address in this register
PC :Program counter	8	contains the address of the next instruction to be fetched for execution
SW : Status word	9	contains a variety of information, including a condition code.

Mnemonic	Number	Special Use
B	3	Base register, used for addressing
S	4	General purpose register
T	5	General purpose register
F	6	Floating point accumulator (48 bits)

The addressing mode categories:

1-Simple addressing mode
2-Indirect addressing mode
3- Immediate addressing mode

To determine the addressing mode :

check n , i flags using an if statement , then go through the x , b and p flags and do the needed calculations according to the following table:

Addressing	n i x b p e	Target Address	Operand	Notation
Simple	1 1 0 0 0 0	Disp	(TA)	op c
	1 1 0 0 0 1	Addr	(TA)	+op m
	1 1 0 0 1 0	(PC) + disp	(TA)	op m
	1 1 0 1 0 0	(b) + disp	(TA)	op m
	1 1 1 0 0 0	(X) + disp	(TA)	op c,X
	1 1 1 0 0 1	(X) + addr	(TA)	+op m,X
	1 1 1 0 1 0	(PC) + (X) +disp	(TA)	op m,X
	1 1 1 1 0 0	(B) + (X) + disp	(TA)	op m,X
	0 0 0 - - -	b/p/e/disp	(TA)	op m
	0 0 1 - - -	(X)+ b/p/e/disp	(TA)	op m,X
Indirect	1 0 0 0 0 0	disp	((TA))	op @c
	1 0 0 0 0 1	addr	((TA))	+op @m
	1 0 0 0 1 0	(PC) +disp	((TA))	op @m
	1 0 0 1 0 0	(B) + disp	((TA))	op @m

Immediate	0 1 0 0 0 0	disp	TA	op #c
	0 1 0 0 0 1	addr	TA	op #m
	0 1 0 0 1 0	(PC) +disp	TA	op #m
	0 1 0 1 0 0	(B)+disp	TA	op #m

TABLE: TA calculations and addressing modes of SIC/XE machine.

The body of the instructions code:

The opcode of each instruction will go through multiple "if statements" to compare it with the opcode's of the SIC/XE machine, each " if statement" will implement the calculations needed to execute the instruction and change the registers values and memory locations if needed .

1.Arithmetic instructions:

1.Addition (ADD) ,(AddR)
2.Subtraction (SUB),(SubR)
3.Multiplication(MUL)
4.Division (DIV)
5.Compasion (COMP)

Fetches a word from memory { m[TA], m[TA+1], m[TA+2] } and perform the corresponding arithmetic operation with the accumulator register (AX) , then it saves the value in the accumulator register.
In the case of the COMP instruction it adjusts the value of the CC according to the comparison result that is :
if A>m then SW="000001" && CC=0
if A=m then SW="000000" && CC=1
if A<m then SW="000002" && CC=-1 .

2. Register instructions:

- Arithmetic(ADDR, SUBR, MULR, DIVR,COMPR):
perform the arithmetic instruction between the two specified registers, R1, R2, and saves the result in R2.
In the case of the COMPR instruction it adjusts the value of the CC according to the comparison result.
- Move register (RMO): move the value of register R1 to register R2.

3. Jump Instructions:

- Jump unconditionally instruction (J): changes the value of the PC register to a specified byte of memory address, and jump to that address.
- Conditional jump instructions (JEQ, JGT, JLT, JSUB): checks the status of the CC, (<, =, >) and then changes the value of the PC register to a specified byte of memory address, and jump to that address accordingly.

4. (TIX):

increment the X register by one and then fetches a word of memory and compares that word with the X register.

5. (TIXR):

increment the X register by one and then fetches a word of memory and compares that word with the specified register by R1 .

6. Store instructions:

(STA, STB, STL, STS, STX, STX, STSW): saves the value of the specified register into a word in memory.
-(STCH):saves the rightmost byte of register A into a byte of memory.
-(STF): saves the value of the FP accumulator into five bytes of memory.

7. Logical instructions(AND, OR):

fetches a word from memory { m, m+1, m+2 } and perform the logical instruction with the accumulator register (AX) and saves the result in the accumulator register.

8. Load instructions:

-(LDA, LDB, LDL, LDS, LDX, LDT): fetches a word from memory { m[TA, m[TA+1], m[TA+2] } and then load their values to the specified register.

- (LDCH): fetches one byte of memory and load its value in the rightmost byte of the accumulator register.

9. Shift instructions:

(SHIRTL, SHIFTR): left circular shift n bits of the specified register and saves the new value back to the register.

10. Comp:

Retrieves a word from memory ,performs the corresponding arithmetic operation with the accumulator register (A), and then saves the outcome back to the accumulator register and modify the value of CC.

11. CLEAR:

clears the value of the specified register

Overview of the program in the provided code:

1. **Includes and librarys: The program starts with including necessary C++ standard libraries and using the std namespace.**
2. **Global Variables: Several global variables are declared, including strings to represent various registers (A, X, L, PC, etc.), opcode (op), flags, format, etc. These variables are used to store information about the current state of the simulation.**
3. **Classes: The program defines a Memory class that encapsulates the memory of the simulated system. It has methods to read and write memory locations, And enum class InstructionFormat is defined to represent different instruction formats.**
4. **Memory Object Creation: An instance of the Memory class is created with a size of 1MB.**
5. **Instructions Functions: There are several functions like checkhexa, addBinaryStrings, AND, Sub, Divide, shiftLeft ,ShiftRight ,Add ,Multiply ,Or ,ReplaceBits ,Read ,Write ,determine_reg ,Addleadzero1&2.These functions perform various operations on binary and hexadecimal strings and called inside Instruction() function in general to perform the operation.**
6. **Process Functions: These functions deals with the strings and binary (converting) and the whole code depends on them to process the input text file and save it in vector and read and compare it to find the opcode then calling the instruction() function to do the rest these functions are (parseHexString, convertToBinary, Readinput, valueToHexChar, hexCharToValue, addHexStrings, binaryToHex, compute_TA, instructions, determineInstructionFormat) .**

7. **Printing Functions: (printInstructionFormats , printMemory , writeRegiesters) These functions are responsible for printing various outputs. The printInstructionFormats function prints information about the parsed instructions and their formats. The printMemory function prints the memory contents to an output file ("output.txt") and console. The writeRegisters function writes register values to the output file.**

8. **Main Function: The main function is where the program execution starts. It creates an instance of the Memory class, writes initial values to specific memory locations, reads input from the "input.txt" file, processes instructions, prints instruction formats, writes register values to the output file, and prints memory contents.**

Overall, the program simulates the execution of instructions using different formats and maintains the state of registers, memory, and flags accordingly. It demonstrates the basic execution of SIC/XE instructions and provides outputs in both the console and an output file.

After the outline let's go deep in the code:

First: How the code works

The function Readdinput read the input file line by line to find the start and the last and store them in variables and checks if there is the symbol (!!) to consider it as comment and checks then if the line starts with hexadecimal digit to start store in a hexadecimal vector (hexNumber) after converting it from ascii to hexadecimal then it will transfer the content to another binary vector (binaryNumber) after converting it to binary and performing flipping the vector because last instruction is stored and should executed first.

After I have the binary vector filled with binary numbers as a strings then calling the function(printInstructionFormats) which then first call the function name (determineInstructionFormat) which parsing the binary vector elements (each line which is machine code) and determine the format, flags ,etc.

After the determination it fill the machine code in the memory and increase the value of the PC based on the size of the machine code and call the function (Compute_TA) that's computes the target address then call the function name (instructions) which take the determined flags, opcode, etc. and then execute them based on the determined instruction.

instructions function do modifying the values of the registers and memory locations and then either call another function to execute the instruction or execute it directly.

After we perform each function on the machine code in the vector, it returns the (enum) inside function called (printInstructionFormats),depending on the enum, which prints the format ,flags ,opcode ,etc.

After we perform these above functions on all the machine code inside the binaryNumber vector it call writeRegisters and PrintMemory functions that do print the values of the registers and memory and write them to output file (final values of the registers and memory locations).

Second: Libraries and classes

The code including necessary C++ standard libraries and using the std namespace and here are them :

The libraries:

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include <iomanip>
#include <bitset>
```

The classes:

```
//classes
class Memory {
private:
    vector<string> memory;
    const int ADDRESS_BITS = 20;
    const int ELEMENT_SIZE = 8;

public:
    Memory(int size) {
        memory.resize(size, "00000000"); // Initialize memory with 8-bit zeros
    }

    void write(const string& address, const  string& value)
    {


        int index = bitset<20>(address).to_ulong();
        if (value.size() != ELEMENT_SIZE)
        {
            cout << "Invalid value size." << endl;
            return;
        }

        memory[index] = value;
    }

    string read(const string& address) const
    {

        int index = std::bitset<20>(address).to_ulong();
        return memory[index];
    }
};
enum class InstructionFormat {
    StandardSIC,
    Format1,
    Format2,
    Format3,
    Format4
};
Memory memory(1024 * 1024);// Memory
```

Third: Global variables

All the necessary variables some of them used as registers or used in the instruction function are global variables to modify it anywhere in the code which is given temporary value and some of them used as flags and addresses, etc, and there is two vectors one of (binaryNumbers)them to store the machine codes and the other (porttext) to store when calling RD instruction which store id, and lines written in input text file and here is all of them :

```
using namespace std;
//Global_variables
string  op;
string format;
string N;//shift value
string n, i, x, b, p, e, flags;
string A = "000000000000000000000000";
string X = "000000000000000000000000";
string L = "000000000000000000000000";
string PC;
string TA ;
string SW = "000000000000000000000000";
string B = "000000000000000000000000";
string S = "000000000000000000000000";
string T = "000000000000000000000000";
string F = "000000000000000000000000";
string address;
string disp;
string newPC;
string PCstart;
string PClast;
string cc;
string TAplus1, TAplus2, TAplus3;
string r1;
string r2;
string ins_name;
string reg_name;
string id;
vector<string> binaryNumbers;// New vector to hold binary representations
vector<string> porttext;//TD,RD,WD
```

Forth: Functions called by instruction function

All of the coming functions are call either in the instruction function or used in another function which called by instruction function so basically they are doing the determined instruction (as we mention previously).

1. Checkhexa:

This function Checks if the character is a valid hexadecimal digit (0-9 or a-f)

```
bool checkhexa(char c)
{
    // Check if the character is a valid hexadecimal digit (0-9 or a-f)
    return (isdigit(c) || (tolower(c) >= 'a' && tolower(c) <= 'f'));
}
```

2. addBinaryStrings:

This function that takes two binary strings as input and returns their sum as a binary string. The function pads the shorter string with zeros and iterates through the strings, adding digits and considering carry. The resulting binary sum is returned.

```
string addBinaryStrings(const string& one, const string& two)
{
    int maxLength = max(one.length(), two.length());
    string adding1 = string(maxLength - one.length(), '0') + one;
    string adding2 = string(maxLength - two.length(), '0') + two;

    int carry = 0;
    string result;

    for (int i = maxLength - 1; i >= 0; --i) {
        int sum = (adding1[i] - '0') + (adding2[i] - '0') + carry;
        result = char(sum % 2 + '0') + result;
        carry = sum / 2;
    }

    if (carry) {
        result = '1' + result;
    }

    return result;
}
```

3. And :

This function that computes the bitwise AND operation between two binary strings of equal length. It checks for equal string lengths, and if they differ, outputs an error message and returns an empty string. For corresponding bits in the input strings, the function appends '1' to the result string only if both bits are '1'; otherwise, it appends '0'. The function then returns the resulting binary string, representing the outcome of the bitwise AND operation.

```
string AND(const string& binary1, const string& binary2)
{
    if (binary1.size() != binary2.size()) {
        cout << "Binary numbers must have the same length." << endl;
        return "";
    }

    string result;
    result.reserve(binary1.size());

    for (size_t i = 0; i < binary1.size(); ++i) {
        if (binary1[i] == '1' && binary2[i] == '1') {
            result += '1';
        }
        else {
            result += '0';
        }
    }

    return result;
}
```

4. Divide:

This function for binary division. It converts the binary inputs to decimal integers and performs division to find the quotient and remainder. However, it has limitations like potential loss of leading zeros and fixed 24-bit output length. The code could benefit from directly working with binary representation for accuracy and dynamic output length based on the quotient.

```
string Divide(const string& bin_dividend, const string& bin_divisor)
{
    int dividend = stoi(bin_dividend, nullptr, 2);
    int divisor = stoi(bin_divisor, nullptr, 2);

    if (divisor == 0) {
        return "Undefined";
    }

    int quotient = dividend / divisor;
    int remainder = dividend % divisor;

    string quotient_bin = "";
    while (quotient > 0) {
        quotient_bin = to_string(quotient % 2) + quotient_bin;
        quotient /= 2;
    }

    return quotient_bin.empty() ? "0" : quotient_bin;
}
```

5. ShiftLeft :

This function that rotates a binary number leftwards by a specified binary shift amount. It adjusts the length of both inputs to 24 bits, which may lead to inaccuracies and unnecessary computations. The improved version calculates the actual shift based on the binary number's length and performs a rotation without unnecessary padding or substring manipulations.

```
string shiftLeft(const string& binaryNumber, const string& shiftAmount)
{
    string addleading = binaryNumber;
    while (addleading.length() < 24) {
        addleading = "0" + addleading;
    }
    // Ensure binaryNumber and shiftAmount have 24 bits

    string paddedShiftAmount = shiftAmount;
    while (paddedShiftAmount.length() < 24) {
        paddedShiftAmount = "0" + paddedShiftAmount;
    }

    // Perform left rotation
    int shift = stoi(paddedShiftAmount, nullptr, 2) % 24;
    string rotatedBinary = addleading.substr(shift) + addleading.substr(0, shift);

    return rotatedBinary.substr(rotatedBinary.length() - 24); // Ensure the result is 24 bits
}
```

6. ShiftRight:

This function attempts to perform a right shift on a binary number, and performs a shifting without unnecessary padding adhering to standard behavior.

```
string ShiftRight(const string& binaryNumber, const string& shiftAmount) {
    // Ensure binaryNumber and shiftAmount have 24 bits
    string paddedBinaryNumber = binaryNumber;
    while (paddedBinaryNumber.length() < 24) {
        paddedBinaryNumber = paddedBinaryNumber[0] + paddedBinaryNumber;
    }

    string paddedShiftAmount = shiftAmount;
    while (paddedShiftAmount.length() < 24) {
        paddedShiftAmount = "0" + paddedShiftAmount;
    }

    // Perform right shift
    int shift = stoi(paddedShiftAmount, nullptr, 2) % 24;
    string shiftedBinary = paddedBinaryNumber.substr(0, 24 - shift);

    // Fill vacated positions with bits set to the leftmost bit
    char fillBit = shiftedBinary[0];
    while (shiftedBinary.length() < 24) {
        shiftedBinary = fillBit + shiftedBinary;
    }

    return shiftedBinary;
}
```

7. Add:

This function performs addition between two input binary strings, a and b, while managing carry values. It iterates through both strings in reverse, calculating the sum of corresponding bits and accumulating the result. The function returning the result.

```
string Add(string a, string b)
{
    string result;
    int carry = 0;

    int i = a.size() - 1;
    int j = b.size() - 1;

    while (i >= 0 || j >= 0 || carry) {
        int sum = carry;
        if (i >= 0)
            sum += a[i] - '0';
        if (j >= 0)
            sum += b[j] - '0';

        result += to_string(sum % 2);
        carry = sum / 2;

        i--;
        j--;
    }

    reverse(result.begin(), result.end());
    return result;
}
```

8. Multiply:

This function calculates the product of two binary strings, a and b, using binary multiplication. It iterates through b's bits in reverse and, if the current bit is '1', adds a shifted a to the result. Then, it appends '0' to a for the next bit. The result is adjusted to 24 bits, either by trimming or padding.

```
string Multiply(string a, string b) {
    string result = "0";

    for (int i = b.size() - 1; i >= 0; i--) {
        if (b[i] == '1') {
            string temp = a;
            result = Add(result, temp);
        }
        a += '0';
    }

    // Trim the result to 24 bits
    if (result.size() > 24) {
        result = result.substr(result.size() - 24);
    }
    else if (result.size() < 24) {
        string padding(24 - result.size(), '0');
        result = padding + result;
    }

    return result;
}
```

9. Or:

This function computes the bitwise OR operation between two binary strings, a and b. It ensures equal lengths by adding leading zeros, then iterates through the bits of both strings. If either bit at the same position is '1', it appends '1' to the result; otherwise, it appends '0'. The resulting binary string reflects the bitwise OR outcome.

```
string OR(string a, string b) {
    string result = "";

    int maxLength = max(a.size(), b.size());

    while (a.size() < maxLength)
    {
        a = "0" + a;
    }
    while (b.size() < maxLength) {
        b = "0" + b;
    }

    for (int i = 0; i < maxLength; i++)
    {
        if (a[i] == '1' || b[i] == '1')
        {
            result += "1";
        }
        else {
            result += "0";
        }
    }

    return result;
}
```

10. replaceBits:

This Function replace between given bits with another given bits with given index.

```
string replaceBits(const string& original, const string& replacement, size_t Index)
{
    //This Function replace between given bits with another givin bits with givin index
    if (Index + replacement.size() > original.size())
    {
        cout << "Replacement goes beyond the original string size." << endl;
        return original;
    }

    string modified = original;
    for (size_t i = 0; i < replacement.size(); ++i) {
        modified[Index + i] = replacement[i];
    }

    return modified;
}
```

11. Read:

This function opens a file defined by filename, searches for a specific label, and stores the subsequent lines in the content vector. It indicates errors if the file cannot be opened or the label is not found.

```
void Read(const string& filename, const string& label, vector<string>& content)
{
    ifstream inputFile(filename);

    if (!inputFile.is_open()) {
        cout << "Failed to open the file." << endl;
        return;
    }

    string line;
    bool foundId = false;

    while (getline(inputFile, line)) {
        if (line.find("!! I/O '" + label + "'") != string::npos)
        {
            foundId = true;
            while (getline(inputFile, line) && line.find("!!") != 0)
            {
                content.push_back(line);
            }

            break;
        }
    }

    inputFile.close();

    if (!foundId) {
        cout << "ID not found in the file." << endl;
    }
}
```

12. Write:

This function saves lines from a vector to a specified output filename. It opens the file, writes a header line, then iterates through the lines, writing each one. If any write operation fails, it throws a runtime_error. The function closes the file at the end.

```
void Write(const vector<string>& lines, const string& filename) {
    ofstream outputFile(filename);

    if (!outputFile.is_open()) {
        throw runtime_error("Failed to open the output file.");
    }
    outputFile << "!!I/O '" << id << "'" << endl;
    for (const string& line : lines) {
        outputFile << line << endl;

        if (!outputFile) {
            outputFile.close();
            throw runtime_error("Error writing to file.");
        }
    }

    outputFile.close();
}
```

13. determine_reg:

This function that takes a string R representing a register identifier and returns a reference to a specific register while updating the reg_name variable accordingly. This code can be improved by using an enumeration or map for more maintainable and concise register handling.

P1:

```
string& determine_reg(string R)
{
    if (R == "0000")
    {
        reg_name = "A";
        return A;
    }
    else if (R == "0001")
    {
        reg_name = "X";
        return X;
    }
    else if (R == "0010")
    {
        reg_name = "L";
        return L;
    }
    else if (R == "0011")
    {
        reg_name = "B";
        return B;
    }
    else if (R == "0100")
    {
        reg_name = "S";
        return S;
    }
```

P2:

```
    }
    else if (R == "0101")
    {
        reg_name = "T";
        return T;
    }
    else if (R == "0110")
    {
        reg_name = "F";
        return F;
    }
    else if (R == "1000")
    {
        reg_name = "PC";
        return PC;
    }
    else if (R == "1001")
    {
        reg_name = "SW";
        return SW;
    }

}
```

14. Addleadzero1:

This function processes a hexadecimal string to ensure it has a consistent format. It returns a modified version with a leading zero if the string has only one character, or the last two characters if the string is longer. This maintains the format of the hexadecimal value.

```
string Addleadzero1(const string& hexString)
{
    if (hexString.size() == 0) {
        return "00";
    }
    else if (hexString.size() == 1) {
        stringstream ss;
        ss << "0" << hexString;
        return ss.str();
    }
    else if (hexString.size() > 2) {
        return hexString.substr(hexString.size() - 2, 2);
    }
    return hexString;
}
```

15. Addleadzero2:

This function ensures a standardized 8-character format for a hexadecimal string: empty input results in "00000000", while strings with less than 8 characters are padded with zeros. Strings exceeding 8 characters are trimmed to the last 8, and strings with 8 characters or more are unchanged. This function offers consistent formatting for hexadecimal values.

```
string Addleadzero2(const string& hexString)
{
    if (hexString.size() == 0) {
        return "00000000";
    }
    else if (hexString.size() < 8) {
        stringstream ss;
        ss << setw(8) << setfill('0') << hex << hexString;
        return ss.str();
    }
    else if (hexString.size() > 8) {
        return hexString.substr(hexString.size() - 8, 8);
    }
    return hexString;
}
```

Fifth: Process functions

1) parseHexString :

This function processes a hexadecimal string to ensure consistency by performing several operations:

1.It converts the entire input hexString to lowercase using a loop that iterates through each character and appends its lowercase version to the processedString.
2.It removes spaces and underscores from the processedString using the erase and remove_if functions. The remove_if function uses a lambda function to check if each character is a valid hexadecimal character using the checkhexa function (presumably defined elsewhere).

The resulting processedString is returned, providing a version of the input hexadecimal string that is lowercase and free of spaces and underscores, which helps in standardizing the representation.

Overall, the function contributes to maintaining a consistent format for hexadecimal values by making them lowercase and removing unnecessary characters.

```
string parseHexString(const string& hexString)
{
    string processedString;

    // Convert the entire input string to lowercase
    for (char c : hexString) {
        processedString += tolower(c);
    }

    // Remove spaces and underscores from the processed string
    processedString.erase(remove_if(processedString.begin(), processedString.end(),
        [](char c) { return !checkhexa(c); }), processedString.end());

    return processedString;
}
```

2) converToBinary:

This function processes an input hexadecimal string by standardizing it through lowercasing and removing unnecessary characters. The size needed for the binary representation is determined by multiplying the processed hexadecimal string's length by 4, as each hexadecimal digit corresponds to 4 binary bits. The processed hexadecimal string is then converted to an unsigned long long integer.

A bitset called binaryNumber is created to store the decimal value in binary form, representing the processed hexadecimal value. The binary representation is obtained as a string by converting the binaryNumber bitset, with leading zeros added or excess bits trimmed to match the calculated size. The function returns the final binary representation string, which accurately represents the input hexadecimal value while maintaining a consistent binary format.

```
string convertToBinary(const string& hexString) {
    // Convert the hexadecimal number to binary using bitset
    string processedString = parseHexString(hexString);
    int binarySize = processedString.length() * 4;
    unsigned long long hexNumber = stoull(processedString, nullptr, 16);
    bitset<64> binaryNumber(hexNumber); // Assuming the number of hex digits can be up to 16 (64 bits)

    // Get the binary string with leading zeros to match the required size
    string binaryString = binaryNumber.to_string();
    binaryString = binaryString.substr(binaryString.size() - binarySize);

    return binaryString;
}
```

3) Readinput (The most important function):

This function is designed to systematically process input derived from a specified file path. It undertakes a thorough examination of each line present within the file, each line holding valuable data. Should the initial character of a line exhibit validity as a hexadecimal value, the function commences its operation. Employing a judicious strategy, it orchestrates the transformation of this value into lowercase form, concurrently eradicating any redundant spaces or underscores. This refined result is subsequently allocated a position within the hexNumbers array, accompanied by its meticulously refined binary counterpart within the binaryNumbers array.

The function, however, demonstrates an expanded cognitive scope. It searches for the "START" and "LAST" within the lines, not merely as textual artifacts, but as substantial markers. When the function encounters these keywords, it does not merely acknowledge their existence; rather, it exercises its authority to seize the subsequent hexadecimal value. This value, Previously unremarkable, the value goes through the same improvement process before being assigned to either PCstart or PClast, depending on the particular keyword found. These two values—PCstart and PClast—serve as important markers, highlighting specific moments in the larger program's operation.

Upon achieving its designated objectives, the function gracefully concludes its duties. It systematically concludes its interaction with the file by formally closing it. Before this closure, however, it takes a moment to exhibit its accomplishments. It disseminates a presentation of the meticulously refined hexadecimal values residing within the hexNumbers array, accompanied by their corresponding binary counterparts. This presentation bears an air of sophistication, assuring consistent formatting through the addition of leading zeros where necessary.

In simple terms, this function goes beyond just reading files. It acts like a careful organizer, making sure things are neat and accurate. It ensures that each hexadecimal value is clear and fits perfectly into its place in the bigger picture.

Here is the code :

P1:

```
void Readinput(const string& filePath)
{
    ifstream inputFile(filePath);
    if (!inputFile) {
        cout<< "Error opening the input file." << endl;
        return;
    }

    vector<string> hexNumbers;

    string line;

    while (getline(inputFile, line)) {
        // Check if the line starts with a valid hexadecimal character
        if (line.length() > 0 && checkhexa(line[0])) {
            string hexNumber = parseHexString(line);
            hexNumbers.push_back(hexNumber);

            // Convert the hexadecimal number to binary and store it in the binaryNumbers vector
            binaryNumbers.push_back(convertToBinary(hexNumber));
        }

        // Find "START" in the line and extract the hex number after it
        size_t startPos = line.find("START");
        if (startPos != string::npos) {
            string hexSubstring = line.substr(startPos + 5); // "START" has 5 characters
            string start = parseHexString(hexSubstring);
            int numHexDigits = start.length();
            cout << "Found START: " << hex << setw(6) << setfill('0') << start << " (Binary: " << convertToBinary(start) << ")" << endl;
            PCstart = convertToBinary(start);
            PC = PCstart;
        }
```

P2:

```
        // Find "LAST" in the line and extract the hex number after it
        size_t lastPos = line.find("LAST");
        if (lastPos != string::npos)
        {
            string hexSubstring = line.substr(lastPos + 4); // "LAST" has 4 characters
            string last = parseHexString(hexSubstring);
            int numHexDigits = last.length();
            cout << "Found LAST: " << hex << setw(6) << setfill('0') << last << " (Binary: " << convertToBinary(last) << ")" << endl;
            PClast = convertToBinary(last);
        }
    }

    inputFile.close(); // Close the input file after reading

    // Print the hexadecimal numbers and their binary representations from the hexNumbers vector
    cout << "Numbers:" << endl;
    for (string hexNumber : hexNumbers) {
        // Ensure at least 6 digits by adding leading zeroes
        cout << hex << setw(6) << setfill('0') << hexNumber << " (Binary: " << convertToBinary(hexNumber) << ")" << endl;
    }
}
```

4) ValueToHexChar :

This function is a convert to hexadecimal characters. It follows a straightforward pattern: if the value is between 0 and 9, it simply adds the value to '0' to get the right character. If the value is between 10 and 15, it does a similar thing with 'A', and makes a small adjustment based on the value. However, if the value doesn't fit within these ranges, it raises an error to handle invalid inputs.

```
char valueToHexChar(int value)
{
    if (value >= 0 && value <= 9)
    {
        return '0' + value;
    }
    else if (value >= 10 && value <= 15)
    {
        return 'A' + (value - 10);
    }
    else {
        throw runtime_error("Invalid hexadecimal value");
    }
}
```

5) AddHexStrings :

This function is simply converting hexadecimal characters back into numerical values. It follows a logical approach: if the character is between '0' and '9', it just subtracts '0' from the character to get the value. For characters 'A' to 'F', or 'a' to 'f', it does a similar subtraction while adding 10 to the result. This accounts for the fact that hexadecimal uses letters beyond '9' to represent values. If the character doesn't match any of these cases, it's considered invalid, and the function raises an error to handle such situations.

```
int hexCharToValue(char c)
{
    if (c >= '0' && c <= '9') {
        return c - '0';
    }
    else if (c >= 'A' && c <= 'F') {
        return 10 + (c - 'A');
    }
    else if (c >= 'a' && c <= 'f') {
        return 10 + (c - 'a');
    }
    else {
        throw runtime_error("Invalid hexadecimal character");
    }
}
```

6) AddHexStrings :

This function adds two hexadecimal strings together. It ensures equal length by adding zeros at the beginning and then goes through each digit, adding them along with any carry. The valueToHexChar and hexCharToValue above functions help with conversions

```
string addHexStrings(const string& hex1, const string& hex2)
{
    //Add hexa values
    int maxLength = max(hex1.length(), hex2.length());
    string paddedHexStr1 = string(maxLength - hex1.length(), '0') + hex1;
    string paddedHexStr2 = string(maxLength - hex2.length(), '0') + hex2;

    int carry = 0;
    string result;

    for (int i = maxLength - 1; i >= 0; --i) {
        int digitSum = hexCharToValue(paddedHexStr1[i]) + hexCharToValue(paddedHexStr2[i]) + carry;
        result = valueToHexChar(digitSum % 16) + result;
        carry = digitSum / 16;
    }

    if (carry) {
        result = valueToHexChar(carry) + result;
    }

    return result;
}
```

7) BinaryToHex:

The function converts a binary string into its hexadecimal representation. Initially, the binary string is converted to a decimal value, leveraging a bitset of up to 64 bits. Then this decimal value is transformed into a hexadecimal representation using a stringstream. The function returns the resulting hexadecimal string.

```
string binaryToHex(const string& binary)
{
    // binary to decimal
    bitset<64> bits(binary);
    unsigned long decimal = bits.to_ulong();

    // decimal to hexadecimal
    stringstream hexStream;
    hexStream << hex << uppercase << decimal;

    return hexStream.str();
}
```

8) Comput_TA:

Simply this important function calculates the Target Address (TA) based on provided flags and register values. It uses conditional statements to determine TA through different operations like addition and memory reads based on the value of the flags. The function handles various addressing modes and register combinations to ensure accurate TA computation for memory access operations.

P1:

```
612  void compute_TA(string flags, Memory& mem)
613  {
614
615      if (flags == "110000")
616          TA = disp;
617      else if (flags == "110001")
618          TA = address;
619      else if (flags == "110010")
620          TA = addBinaryStrings(newPC, disp);
621
622      else  if (flags == "110100")
623          TA = addBinaryStrings(B, disp);
624      else  if (flags == "111000")
625          TA = addBinaryStrings(X, disp);
626      else  if (flags == "111001")
627          TA = addBinaryStrings(X, address);
628      else  if (flags == "111010")
629          TA = addBinaryStrings(addBinaryStrings(newPC, X), disp);
630      else  if (flags == "111100")
631          TA = addBinaryStrings(addBinaryStrings(B, X), disp);
632      else  if (n + i + x == "000")
633          TA = address;
634      else if (n + i + x == "001")
635          TA = addBinaryStrings(X, address);
636      else  if (flags == "100000")
637      {
638          TA = disp;
639          TA = mem.read(TA);
640
641      }
```

P2:

```
641      }
642      else if (flags == "100001")
643      {
644
645          TA = address;
646          TA = mem.read(TA);
647      }
648      else if (flags == "100010")
649      {
650          TA = addBinaryStrings(PC, disp);
651          TA = mem.read(TA);
652      }
653      else if (flags == "100100")
654      {
655          TA = addBinaryStrings(B, disp);
656          TA = mem.read(TA);
657      }
658      else if (flags == "010000")
659          TA = disp;
660      else if (flags == "010001")
661          TA = address;
662      else if (flags == "010010")
663          TA = addBinaryStrings(PC, disp);
664      else if (flags == "010100")
665          TA = addBinaryStrings(B, disp);
666
667
668
669  }
```

9) Instructions:

This function is like a conductor for various actions in the program. It checks the instruction code (op) and other signals to decide what to do. It handles tasks such as math, memory, comparisons, and i/o operations. It updates registers, memory, and counters to perform these tasks. It also manages flags and status information to keep track of how things are going.

Note that this function is too long so there is no picture of it but here is but the next part include all the instructions that this function has

10) determineInstructionFormat :

The given code snippet performs the task of determining the format of an instruction based on a provided binary number and manipulating memory. It contains a series of conditions and operations to identify the instruction format. The code starts by checking the length of the binary number, then proceeds with different cases.

In the first paragraph, the code checks if the length of the binary number is 24 bits. If true, it proceeds to check whether the instruction format is a Standard SIC format (n=0, i=0) or a Format3 instruction format (n=1, i=1, e=0). Depending on the detected format, it extracts various fields from the binary number such as opcode, flags, and address, and computes target addresses and updates the memory accordingly.

The second paragraph of the code snippet deals with other possible instruction formats. It checks for Format1 instructions (8 bits), Format2 instructions (16 bits), and Format4 instructions (32 bits with e=1). For each format, it extracts relevant fields, computes necessary values like target addresses, and updates memory.

Throughout the code, functions like addBinaryStrings are used to perform binary arithmetic, compute_TA to calculate target addresses, and instructions to handle further processing.

Pictures:

```
InstructionFormat determineInstructionFormat(const string& binaryNumber, Memory& mem)
{
    if (binaryNumber.length() == 24)
    {
        // Check if it's a Standard SIC instruction format (size = 24 bits and n=0,i=0)
        if (binaryNumber[6] == '0' && binaryNumber[7] == '0') {
            op = binaryNumber.substr(0, 8);
            n = binaryNumber[6];
            i = binaryNumber[7];
            x = binaryNumber[8];
            b = binaryNumber[9];
            p = binaryNumber[10];
            e = binaryNumber[11];
            flags = binaryNumber.substr(6, 6);

            address = binaryNumber.substr(9, 15);
            newPC = addBinaryStrings(PC, "000000000000000000011");
            compute_TA(flags, mem);

            memory.write(PC, binaryNumber.substr(0, 8));
            PC = addBinaryStrings(PC, "000000000000000000001");
            memory.write(PC, binaryNumber.substr(8, 8));
            PC = addBinaryStrings(PC, "000000000000000000001");
            memory.write(PC, binaryNumber.substr(16, 8));
            PC = addBinaryStrings(PC, "000000000000000000001");
            instructions();
            return InstructionFormat::StandardSIC;
        }
        // Check if it's a Format 3 instruction format (size = 24 bits and n=1, i=1, e=0)
```

```
        else if ((binaryNumber[6] == '1' && binaryNumber[7] == '1' || binaryNumber[6] == '0' && binaryNumber[7] == '1' || binaryNumber[6] == '1' && binaryNumber[7] == '0') && binaryNumber[11] == '0') {
            op = binaryNumber.substr(0, 6) + "00";
            n = binaryNumber[6];
            i = binaryNumber[7];
            x = binaryNumber[8];
            b = binaryNumber[9];
            p = binaryNumber[10];
            e = binaryNumber[11];
            flags = binaryNumber.substr(6, 6);
            disp = binaryNumber.substr(12, 12);
            format = "3";
            newPC = addBinaryStrings(PC, "000000000000000000011");
            compute_TA(flags, mem);
            memory.write(PC, binaryNumber.substr(0, 8));
            PC = addBinaryStrings(PC, "000000000000000000001");
            memory.write(PC, binaryNumber.substr(8, 8));
            PC = addBinaryStrings(PC, "000000000000000000001");
            memory.write(PC, binaryNumber.substr(16, 8));
            PC = addBinaryStrings(PC, "000000000000000000001");
            instructions();
            return InstructionFormat::Format3;
        }
    }
    else if (binaryNumber.length() == 8) {
        // Check if it's a Format 1 instruction format (size = 8 bits and consists of just opcode)
        op = binaryNumber;
        memory.write(PC, binaryNumber.substr(0, 8));
        PC = addBinaryStrings(PC, "000000000000000001");
        instructions();
        return InstructionFormat::Format1;
    }
    else if (binaryNumber.length() == 16) {
```

```
    else if (binaryNumber.length() == 16) {
        // Check if it's a Format 2 instruction format (size = 16 bits and operands are just registers)
        op = binaryNumber.substr(0, 8);
        r1 = binaryNumber.substr(8, 4);
        r2 = binaryNumber.substr(12, 4);
        N = addBinaryStrings(r2, "0001");
        memory.write(PC, binaryNumber.substr(0, 8));
        PC = addBinaryStrings(PC, "0000000000000000001");
        memory.write(PC, binaryNumber.substr(8, 8));
        PC = addBinaryStrings(PC, "0000000000000000001");
        instructions();
        return InstructionFormat::Format2;
    }
    else if (binaryNumber.length() == 32 && binaryNumber[11] == '1') {
        // Check if it's a Format 4 instruction format (size = 32 bits and e=1)
        op = binaryNumber.substr(0, 6) + "00";
        n = binaryNumber[6];
        i = binaryNumber[7];
        x = binaryNumber[8];
        b = binaryNumber[9];
        p = binaryNumber[10];
        e = binaryNumber[11];
        flags = binaryNumber.substr(6, 6);
        address = binaryNumber.substr(12, 20);
        format = "4";

        newPC = addBinaryStrings(PC, "00000000000000000100");
        compute_TA(flags, mem);

        memory.write(PC, binaryNumber.substr(0, 8));
        PC = addBinaryStrings(PC, "0000000000000000001");
        memory.write(PC, binaryNumber.substr(8, 8));
        PC = addBinaryStrings(PC, "0000000000000000001");
        memory.write(PC, binaryNumber.substr(16, 8));
        PC = addBinaryStrings(PC, "0000000000000000001");
        memory.write(PC, binaryNumber.substr(24, 8));
        PC = addBinaryStrings(PC, "0000000000000000001");
        instructions();
        return InstructionFormat::Format4;
    }
```

Sixth:The instructions

1. **Add:**

   This instruction checks for the "ADD" operation (op == "00011000") in an instruction. If it's an ADD operation, it adds values from memory to the accumulator ("A") based on certain conditions.

```
void instructions()
{
    if (op == "00011000")//ADD
    {
        cout << "ADD,";
        if ((n + i) == "01")
        {
            A = addBinaryStrings(A, TA);
            A = addBinaryStrings(A, "00000000000000000000000");

        }
        else
        {
            TAplus1 = addBinaryStrings(TA, "00000000000000000001");
            TAplus2 = addBinaryStrings(TA, "00000000000000000010");
            TAplus3 = addBinaryStrings(TA, "00000000000000000011");
            A = addBinaryStrings(A, memory.read(TAplus2) + memory.read(TAplus1) + memory.read(TA));

        }


    }
```

2. **AddR:**

   This instruction executes the "ADDR" operation (op == "10010000"), adding the value in register R1 to the value in register R2, and then stores the result in R2.

```
    else if (op == "10010000")//ADDR
    {
        cout << "ADDR,";

        determine_reg(r2) = addBinaryStrings(determine_reg(r1), determine_reg(r2));


    }
```

3. **And**:

This function handles the "AND" operation. If the operation code is "01000000," it performs a bitwise AND operation between the accumulator ("A") and either the target address ("TA") or the sum of memory values from certain addresses. The outcome is then stored in the accumulator ("A").

```
else if (op == "01000000")//AND
{
    cout << "AND,";
    if ((n + i) == "01")
    {

        A = AND(A, TA);
        A = addBinaryStrings(A, "00000000000000000000000");
    }
    else {

        TAplus1 = addBinaryStrings(TA, "00000000000000000001");
        TAplus2 = addBinaryStrings(TA, "00000000000000000010");
        TAplus3 = addBinaryStrings(TA, "00000000000000000011");
        A = AND(A, memory.read(TAplus2) + memory.read(TAplus1) + memory.read(TA));
    }
```

4. **Clear**:

This function do "CLEAR" operation (op == "10110100"), this code sets the value in register R1 to zero.

```
else if (op == "10110100") //Clear
{
    cout << "CLEAR,";
    determine_reg(r1) = "00000000000000000000000";


}
```

5. **Comp**:

This function "COMP" operation (op == "00101000") is managed by this code section. It first prints "COMP," to indicate the operation. If the sum of the "n" and "i" flags is "01," the code compares the accumulator value (A) with the target address (TA). Depending on the comparison, the condition code (cc) is set to "1" if A equals TA, "-1" if A is less than TA, or "0" otherwise. Conversely, if the sum of the "n" and "i" flags is not "01," the code computes three addresses by adding constants to TA. It then compares A with the sum of memory values from these addresses. The condition code (cc) is updated similarly based on the comparison result. In summary, this code section performs a comparison operation between A and either TA or the sum of memory values, updating the condition code accordingly.

```
else if (op == "00101000")//Comp
{
    cout << "COMP,";
    if ((n + i) == "01")
    {
        if (A == TA)
        {
            cc = "1";
            SW = "00000000000000000000000";
        }
        else if (A < TA)
        {
            cc = "-1";
            SW = "00000000000000000000010";
        }
        else
        {
            cc = "0";
            SW = "00000000000000000000001";
        }

    }
    else
    {

        TAplus1 = addBinaryStrings(TA, "00000000000000000001");
        TAplus2 = addBinaryStrings(TA, "00000000000000000010");
        TAplus3 = addBinaryStrings(TA, "00000000000000000011");
        if (A == (memory.read(TAplus2) + memory.read(TAplus1) + memory.read(TA)))
        {
            cc = "1";
            SW = "00000000000000000000000";
        }
        else if (A < (memory.read(TAplus2) + memory.read(TAplus1) + memory.read(TA)))
        {
            cc = "-1";
            SW = "00000000000000000000010";
        }
        else
        {
            cc = "0";
            SW = "00000000000000000000001";
        }
    }
```

6. **CompR**:

   For the "CompR" operation (op == "10100000"), the code prints "COMPR," and then compares the values in registers r1 and r2. Depending on the comparison result, it updates the condition code (cc) with "1" if r1 equals r2, "-1" if r1 is less than r2, or "0".

```
else if (op == "10100000")//CompR
{
    cout << "COMPR,";

    if (determine_reg(r1) == determine_reg(r2))
    {
        cc = "1";
        SW = "00000000000000000000000";
    }
    else if (determine_reg(r1) < determine_reg(r2))
    {
        cc = "-1";
        SW = "00000000000000000000010";
    }
    else
    {
        cc = "0";
        SW = "00000000000000000000001";
    }


}
```

7. **Div**:

   For the "DIV" operation (op == "00100100"), this code prints "DIV," and performs division. If (n + i) equals "01," it divides the accumulator (A) by the target address (TA) and adds zeros to A. Otherwise, it divides A by the sum of memory values from certain addresses.

```
else if (op == "00100100")//DIV
{
    cout << "DIV,";
    if ((n + i) == "01")
    {

        A = Divide(A, TA);
        A = addBinaryStrings(A, "000000000000000000000000");
    }
    else
    {
        TAplus1 = addBinaryStrings(TA, "000000000000000001");
        TAplus2 = addBinaryStrings(TA, "000000000000000010");
        TAplus3 = addBinaryStrings(TA, "000000000000000011");

        A = Divide(A, (memory.read(TAplus2) + memory.read(TAplus1) + memory.read(TA)));

    }

}
```

8. **DivR**:

For the "DIVR" operation (op == "10011100"), this code prints "DIVR," and then divides the value in register r2 by the value in register r1, storing the result back in r2.

```
else if (op == "10011100")//DIVR
{
    cout << "DIVR,";

    determine_reg(r2) = Divide(determine_reg(r2), determine_reg(r1));



}
```

9. **Jump**:

For the "Jump" operation (op == "00111100"), this code prints "JUMP," and then sets the program counter (PC) to the value of the target address (TA), resulting in a jump to that memory location.

```
else if (op == "00111100")//Jump
{
    cout << "JUMP,";

    PC = TA;


}
```

10. **JEQ**:

For "JEQ" operation (op == "00110000"), this code prints "JEQ," and if the condition code (cc) is "1," it sets the program counter (PC) to the value of the target address (TA), leading to a jump if the condition is met.

```
else if (op == "00110000")//JEQ
{
    cout << "JEQ,";
    if (cc == "1")
        PC = TA;


}
```

11. JGT:

For "JGT" (op == "00110100"), this code prints "JGT," and if the condition code (cc) is "0" (greater than), it updates the program counter (PC) to the target address (TA) for a jump.

```
else if (op == "00110100")//JGT
{
    cout << "JGT,";
    if (cc == "0")
        PC = TA;
}
```

12. JLS:

For "JLT" (op == "00111000"), this code prints "JLT," and if the condition code (cc) is "-1" (less than), it updates the program counter (PC) to the target address (TA) for a jump.

```
}
else if (op == "00111000")//JLT
{
    cout << "JLT,";

    if (cc == "-1")
        PC = TA;


}
```

13. JSUB:

For "JSUB" (op == "01000000"), this code prints "JSUB," and then saves the current program counter in the linkage register (L) before updating the program counter (PC) to the target address (TA). This facilitates a subroutine call.

```
else if (op == "01000000")//JSUB
{
    cout << "JSUB,";

    L = PC;
    PC = TA;



}
```

14. RSUB:

For "RSUB" (op == "01001100"), this code prints "RSUB," and simply sets the program counter (PC) to the value stored in the linkage register (L). This action effectively returns from a subroutine to the address where it was called.

```
else if (op == "01001100")//RSUB
{
    cout << "RSUB,";

    PC = L;



}
```

15. LDA:

For "LDA" (op == "00000000"), this code prints "LDA," and then either directly loads the target address (TA) into the accumulator (A) or computes memory addresses to load data from, summing values if needed.

```
else if (op == "00000000")//LDA
{

    cout << "LDA,";
    if ((n + i) == "01")
    {

        A = TA;
        A = addBinaryStrings(A, "000000000000000000000000");
    }
    else
    {
        TAplus1 = addBinaryStrings(TA, "000000000000000000001");
        TAplus2 = addBinaryStrings(TA, "000000000000000000010");
        TAplus3 = addBinaryStrings(TA, "000000000000000000011");
        A = (memory.read(TAplus2) + memory.read(TAplus1) + memory.read(TA));

    }
}
```

16. LDB:

For "LDB" (op == "01101000"), this code prints "LDB," and then either directly loads the target address (TA) into the base register (B) or computes memory addresses to load data from, summing values if necessary.

```
else if (op == "01101000")//LDB
{

    cout << "LDB,";
    if ((n + i) == "01")
    {

        B = TA;
        B = addBinaryStrings(A, "00000000000000000000000");
    }
    else {
        TAplus1 = addBinaryStrings(TA, "00000000000000000001");
        TAplus2 = addBinaryStrings(TA, "00000000000000000010");
        TAplus3 = addBinaryStrings(TA, "00000000000000000011");
        B = (memory.read(TAplus2) + memory.read(TAplus1) + memory.read(TA));

    }
}
```

17. LDCH:

For "LDCH" (op == "01010000"), this code prints "LDCH," and then reads a character value from memory at the target address (TA) and places it in the appropriate portion of the accumulator (A) based on the instruction format.

```
else if (op == "01010000")//LDCH
{
    cout << "LDCH,";

    string temp;
    temp = memory.read(TA);
    if (format == "3")
    {
        A = replaceBits(A, temp, 15);
    }
    if (format == "4")
    {
        A = replaceBits(A, temp, 23);
    }

}
```

18. LDL:

For "LDL" (op == "00001000"), this code prints "LDL," and then either directly loads the target address (TA) into the linkage register (L) or computes memory addresses to load data from, summing values if needed.

```
else if (op == "00001000")//LDL
{
    cout << "LDL,";
    if ((n + i) == "01")
    {

        L = TA;
        L = addBinaryStrings(A, "00000000000000000000000");
    }
    else
    {
        TAplus1 = addBinaryStrings(TA, "00000000000000000001");
        TAplus2 = addBinaryStrings(TA, "00000000000000000010");
        TAplus3 = addBinaryStrings(TA, "00000000000000000011");
        L = (memory.read(TAplus2) + memory.read(TAplus1) + memory.read(TA));

    }


}
```

19. LDS:

For "LDS" (op == "01101100"), this code prints "LDS," and then either directly loads the target address (TA) into the source register (S) or calculates memory addresses to load data from, summing values as required.

```
else if (op == "01101100")//LDS
{
    cout << "LDS,";
    if ((n + i) == "01")
    {

        S = TA;
        S = addBinaryStrings(A, "00000000000000000000000");
    }
    else
    {
        TAplus1 = addBinaryStrings(TA, "00000000000000000001");
        TAplus2 = addBinaryStrings(TA, "00000000000000000010");
        TAplus3 = addBinaryStrings(TA, "00000000000000000011");
        S = (memory.read(TAplus2) + memory.read(TAplus1) + memory.read(TA));
    }


}
```

20. LDT:

For "LDT" (op == "01110100"), this code prints "LDT," and then either directly loads the target address (TA) into the task register (T) or calculates memory addresses to load data from, summing values as needed.

```
else if (op == "01110100")//LDT
{
    cout << "LDT,";
    if ((n + i) == "01")
    {

        T = TA;
        T = addBinaryStrings(A, "00000000000000000000000");
    }
    else
    {
        TAplus1 = addBinaryStrings(TA, "00000000000000000001");
        TAplus2 = addBinaryStrings(TA, "00000000000000000010");
        TAplus3 = addBinaryStrings(TA, "00000000000000000011");
        T = (memory.read(TAplus2) + memory.read(TAplus1) + memory.read(TA));
    }

}
```

21. LDX:

For "LDX" (op == "00000100"), this code prints "LDX," and then either directly loads the target address (TA) into the index register (X) or calculates memory addresses to load data from, summing values as needed.

```
else if (op == "00000100")//LDX
{
    cout << "LDX,";
    if ((n + i) == "01")
    {

        X = TA;
        X = addBinaryStrings(A, "00000000000000000000000");
    }
    else
    {
        TAplus1 = addBinaryStrings(TA, "00000000000000000001");
        TAplus2 = addBinaryStrings(TA, "00000000000000000010");
        TAplus3 = addBinaryStrings(TA, "00000000000000000011");
        X = (memory.read(TAplus2) + memory.read(TAplus1) + memory.read(TA));
    }

}
```

22. MUL:

For "MUL" (op == "00100000"), this code prints "MUL," and then either directly multiplies the accumulator (A) with the target address (TA) or calculates memory addresses to multiply with, updating A with the result.

```
else if (op == "00100000")//MUL
{
    cout << "MUL,";
    if ((n + i) == "01")
    {

        A = Multiply(A, TA);
        A = addBinaryStrings(A, "00000000000000000000000");
    }
    else
    {
        TAplus1 = addBinaryStrings(TA, "00000000000000000001");
        TAplus2 = addBinaryStrings(TA, "00000000000000000010");
        TAplus3 = addBinaryStrings(TA, "00000000000000000011");
        A = Multiply(A, (memory.read(TAplus2) + memory.read(TAplus1) + memory.read(TA)));

    }

}
```

23. MULR:

For "MULR" (op == "10011000"), this code prints "MULR," and then multiplies the value in register r2 by the value in register r1, updating r2 with the result.

```
else if (op == "10011000")//MULR
{
    cout << "MULR,";


    determine_reg(r2) = Multiply(determine_reg(r2), determine_reg(r1));



}
```

24. OR:

For "OR" (op == "01000100"), this code prints "OR," and then either performs a bitwise OR operation between the accumulator (A) and the target address (TA), or calculates memory addresses for OR operations and updates A accordingly.

```
else if (op == "01000100")//OR
{
    cout << "OR,";
    if ((n + i) == "01")
    {

        A = OR(A, TA);
        A = addBinaryStrings(A, "00000000000000000000000");
    }
    else
    {
        TAplus1 = addBinaryStrings(TA, "00000000000000000001");
        TAplus2 = addBinaryStrings(TA, "00000000000000000010");
        TAplus3 = addBinaryStrings(TA, "00000000000000000011");
        A = OR(A, (memory.read(TAplus2) + memory.read(TAplus1) + memory.read(TA)));

    }


}
```

25. ShiftL:

For "SHIFTL" (op == "10100100"), this code prints "SHIFTL," and then shifts the value in register r1 to the left by a specified number of positions (N).

```
}
else if (op == "10100100")//SHIFTL
{
    cout << "SHIFTL,";


    determine_reg(r1) = shiftLeft(determine_reg(r1), N);



}
```

26. ShiftR:

For "SHIFTR" (op == "10101000"), this code prints "SHIFTR," and then right-shifts the value in register r1 by a specified number of positions (N).

```
else if (op == "10101000")//SHIFTR
{

    cout << "SHIFTR,";


    determine_reg(r1) = ShiftRight(determine_reg(r1), N);




}
```

27. STA:

For "STA" (op == "00001100"), this code prints "STA," and then writes the content of the accumulator (A) into memory at three consecutive addresses derived from the target address (TA). The accumulator value is divided into three parts and stored accordingly.

```
else if (op == "00001100")//STA
{
    cout << "STA,";

    TAplus1 = addBinaryStrings(TA, "0000000000000000001");
    TAplus2 = addBinaryStrings(TA, "0000000000000000010");
    TAplus3 = addBinaryStrings(TA, "0000000000000000011");
    memory.write(TA, A.substr(16, 8));
    memory.write(TAplus1, A.substr(8, 8));
    memory.write(TAplus2, A.substr(0, 8));
}
```

28. STB:

For "STB" (op == "01111000"), this code prints "STB," and then writes the content of the base register (B) into memory at three consecutive addresses derived from the target address (TA). The base register value is divided into three parts and stored accordingly.

```
else if (op == "01111000")//STB
{
    cout << "STB,";

    TAplus1 = addBinaryStrings(TA, "0000000000000000001");
    TAplus2 = addBinaryStrings(TA, "0000000000000000010");
    TAplus3 = addBinaryStrings(TA, "0000000000000000011");
    memory.write(TA, B.substr(16, 8));
    memory.write(TAplus1, B.substr(8, 8));
    memory.write(TAplus2, B.substr(0, 8));
}
```

29. STCH:

For "STCH" (op == "01010100"), this code prints "STCH," and then writes the least significant byte (8 bits) of the accumulator (A) into memory at an address derived from the target address (TA).

```
else if (op == "01010100")//STCH
{
    cout << "STCH,";

    string TAplus1 = addBinaryStrings(TA, "0000000000000000001");
    string TAplus2 = addBinaryStrings(TA, "0000000000000000010");
    string TAplus3 = addBinaryStrings(TA, "0000000000000000011");
    memory.write(TA, A.substr(16, 8));

}
```

30. STL:

The code handles the "STL", (op == "00010100") assembly instruction. It prints "STL," and writes parts of string L to memory addresses based on TA and constants.

```
else if (op == "00010100")//STL
{
    cout << "STL,";


    TAplus1 = addBinaryStrings(TA, "0000000000000000001");
    TAplus2 = addBinaryStrings(TA, "0000000000000000010");
    TAplus3 = addBinaryStrings(TA, "0000000000000000011");
    memory.write(TA, L.substr(16, 8));
    memory.write(TAplus1, L.substr(8, 8));
    memory.write(TAplus2, L.substr(0, 8));
}
```

31. STS:

The code processes the "STS", (op == "01111100") assembly instruction. It prints "STS," and writes portions of string S to memory addresses calculated using TA and constant values.

```
else if (op == "01111100")//STS
{
    cout << "STS,";

    TAplus1 = addBinaryStrings(TA, "0000000000000000001");
    TAplus2 = addBinaryStrings(TA, "0000000000000000010");
    TAplus3 = addBinaryStrings(TA, "0000000000000000011");
    memory.write(TA, S.substr(16, 8));
    memory.write(TAplus1, S.substr(8, 8));
    memory.write(TAplus2, S.substr(0, 8));
}
```

32. STSW:

The code handles "STSW" assembly instruction (opcode "11101000"). It prints "STSW," and writes parts of string SW to memory addresses calculated using TA and fixed binary values.

```
else if (op == "11101000")//STSW
{
    cout << "STSW,";

    TAplus1 = addBinaryStrings(TA, "0000000000000000001");
    TAplus2 = addBinaryStrings(TA, "0000000000000000010");
    TAplus3 = addBinaryStrings(TA, "0000000000000000011");
    memory.write(TA, SW.substr(16, 8));
    memory.write(TAplus1, SW.substr(8, 8));
    memory.write(TAplus2, SW.substr(0, 8));
}
```

33. STT:

This code processes the "STT" assembly instruction (opcode "10000100"). It prints "STT," and writes segments of string T to memory addresses derived from TA using fixed binary values.

```
else if (op == "10000100")//STT
{
    cout << "STT,";

    TAplus1 = addBinaryStrings(TA, "00000000000000000001");
    TAplus2 = addBinaryStrings(TA, "00000000000000000010");
    TAplus3 = addBinaryStrings(TA, "00000000000000000011");
    memory.write(TA, T.substr(16, 8));
    memory.write(TAplus1, T.substr(8, 8));
    memory.write(TAplus2, T.substr(0, 8));
}
```

34. STX:

This code manages the "STX" assembly instruction (opcode "00010000"). It prints "STX," and writes segments of string X to memory addresses obtained from TA using fixed binary values.

```
else if (op == "00010000")//STX
{
    cout << "STX,";

    TAplus1 = addBinaryStrings(TA, "00000000000000000001");
    TAplus2 = addBinaryStrings(TA, "00000000000000000010");
    TAplus3 = addBinaryStrings(TA, "00000000000000000011");
    memory.write(TA, X.substr(16, 8));
    memory.write(TAplus1, X.substr(8, 8));
    memory.write(TAplus2, X.substr(0, 8));
}
```

35. SUB:

This code handles the "SUB" assembly instruction (opcode "00011100"). It subtracts values from the accumulator A based on certain conditions. After printing "SUB," it subtracts either the value at memory address TA plus constants or the sum of values from specific memory addresses (TA, TAplus1, TAplus2) from the accumulator A.

```
else if (op == "00011100")//SUB
{
    cout << "SUB,";
    if ((n + i) == "01")
    {

        A = Sub(A, TA);
        A = addBinaryStrings(A, "00000000000000000000000");
    }
    else
    {
        TAplus1 = addBinaryStrings(TA, "00000000000000000001");
        TAplus2 = addBinaryStrings(TA, "00000000000000000010");
        TAplus3 = addBinaryStrings(TA, "00000000000000000011");
        A = Sub(A, memory.read(TAplus2) + memory.read(TAplus1) + memory.read(TA));

    }


}
```

36. SUBR:

This code section handles the "SUBR" assembly instruction (opcode "10010100"). When encountering this instruction, it prints "SUBR," to the console. It performs subtraction on registers based on the provided r1 and r2 values. The result of subtracting the value in r1 from the value in r2 is then stored back in r2.

```
else if (op == "10010100")//SUBR
{
    cout << "SUBR,";
    determine_reg(r2) = Sub(determine_reg(r2), determine_reg(r1));


}
```

37. TIX:

The code snippet manages the "TIX" assembly instruction (opcode "00101100"). It prints "TIX," to the console and increments register X. Depending on whether (n + i) equals "01" or not, it either compares X and TA to update a condition code (cc) for equality, or it generates memory addresses, compares X with a sum of memory values, and adjusts cc accordingly. This code essentially handles the "TIX" operation, keeping track of register X and adjusting the condition code for comparisons.

```
else if (op == "00101100")//TIX
{
    cout << "TIX,";
    if ((n + i) == "01")
    {
        X = addBinaryStrings(X, "000000000000000000000001");
        if (X == TA)
        {
            cc = "1";
            SW = "000000000000000000000000";
        }
        else if (X < TA)
        {
            cc = "-1";
            SW = "000000000000000000000010";
        }
        else
        {
            cc = "0";
            SW = "000000000000000000000001";
        }

    }
    else
    {
        X = addBinaryStrings(X, "000000000000000000000001");
        TAplus1 = addBinaryStrings(TA, "000000000000000000000001");
        TAplus2 = addBinaryStrings(TA, "000000000000000000000010");
        TAplus3 = addBinaryStrings(TA, "000000000000000000000011");
        if (X == (memory.read(TAplus2) + memory.read(TAplus1) + memory.read(TA)))
        {
            cc = "1";
            SW = "000000000000000000000000";
        }
        else if (X < (memory.read(TAplus2) + memory.read(TAplus1) + memory.read(TA)))
        {
            cc = "-1";
            SW = "000000000000000000000010";
        }
        else
        {
            cc = "0";
            SW = "000000000000000000000001";
        }
    }
}
```

38. TIXR:

This code snippet manages the "TIXR" assembly instruction (opcode "10111000"), printing "TIXR," to the console. It increments register X and compares its value with a specified register r1. The condition code cc is then adjusted to reflect the comparison result, indicating equality ("1"), X being less ("-1"), or greater ("0") than the register value. The snippet focuses on updating X and the condition code based on comparisons with r1.

```
else if (op == "10111000")//TIXR
{

    cout << "TIXR,";
    X = addBinaryStrings(X, "000000000000000000000001");


    if (X == determine_reg(r1))
    {
        cc = "1";
        SW = "000000000000000000000000";
    }
    else if (X < determine_reg(r1))
    {
        cc = "-1";
        SW = "000000000000000000000010";
    }
    else
    {
        cc = "0";
        SW = "000000000000000000000001";
    }


}
```

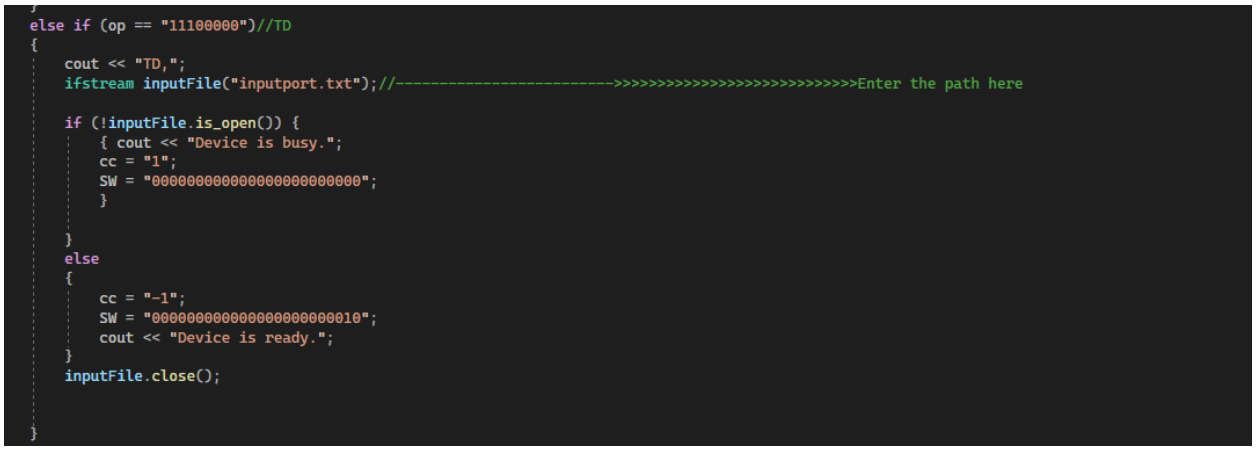
39. RMO:

This code handles the "RMO" assembly instruction (opcode "10101100"), printing "RMO," to the console. It assigns the value of one register r1 to another register r2, effectively copying the value between registers. The specific context of registers and variables is not provided within this snippet.

```
else if (op == "10101100")//RMO
{
    cout << "RMO,";
    determine_reg(r2) = determine_reg(r1);



}
```

40. TD:

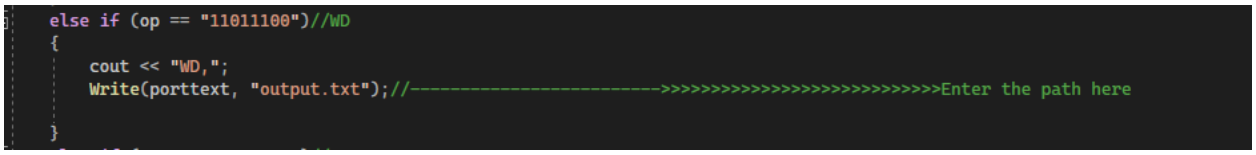
The provided code snippet manages the "TD" assembly instruction (opcode "11100000"). It begins by printing "TD," to the console and proceeds to attempt opening a file using a designated path. In the case that the file cannot be opened, the code outputs "Device is busy." and adjusts the condition code (cc) to "1". On the other hand, if the file can be successfully opened, the program prints "Device is ready." and sets cc to "-1". Once the checks are completed, the file is closed. In essence, this code segment primarily concerns itself with file accessibility, informing the status of the device and updating the condition code accordingly, while the specific file's context and variable details are not elaborated here.



```
else if (op == "11100000")//TD
{
    cout << "TD,";
    ifstream inputFile("inputport.txt");//------------------------>>>>>>>>>>>>>>>>>>>>>>>>>>>>Enter the path here

    if (!inputFile.is_open()) {
        { cout << "Device is busy.";
        cc = "1";
        SW = "00000000000000000000000";
        }

    }
    else
    {
        cc = "-1";
        SW = "00000000000000000000010";
        cout << "Device is ready.";
    }
    inputFile.close();


}
```

41. WD:

This code deals with the "WD" assembly instruction, (op == "11011100"). It prints "WD," to the console and uses the Write function to save the content of porttext to an output file with a specific path.



```
    else if (op == "11011100")//WD
    {
        cout << "WD,";
        Write(porttext, "output.txt");//------------------------>>>>>>>>>>>>>>>>>>>>>>>>>>>>Enter the path here

    }
```

42. RD:

This code snippet handles the "RD" assembly instruction, (op == "11011000"). It prints "RD," to the console and reads data from a file, either using the hexadecimal value of TA or memory content at TA, based on the (n + i) condition. The specific details of variables like id, porttext, and the Read function are not given in this snippet.

```
else if (op == "11011000")//RD
{
    cout << "RD,";
    if ((n + i) == "01")
    {
        id = TA;

        id = binaryToHex(id);
        id = Addleadzero1(id);

        Read("IO.txt", id, porttext);//------------------------>>>>>>>>>>>>>>>>>>>>>>>>>>>>Enter the path here
    }
    else
    {
        id = binaryToHex(memory.read(TA));
        id = Addleadzero1(id);
        Read("IO.txt", id, porttext);//------------------------>>>>>>>>>>>>>>>>>>>>>>>>>>>>Enter the path here
    }
}
```

Note: that WD it printed first before any other thing at the output file because it handled as a special case so when I call the instruction function with its opcode it calls another function which is print on the console and as discussed before in the flow of the code calling each instruction is before printing, so it was printed first.

Note: all these functions is called in the instructions function.

Seventh: Print functions

All these functions have one essential function which is printing but some of them do prints on the console for testing and others do print on the output file, although they are more than printing functions.

These functions are:

i. printInstructionFormats(Test console):

This function defines a testing function printInstructionFormats that takes a vector of binary numbers. It displays information about different instruction formats using a switch statement. For each binary number, it determines the format using the determineInstructionFormat function and memory. It then prints details based on the format, covering Standard SIC, Format 1, Format 2, Format 3, and Format 4. If the format is unrecognized, it prints "Invalid format". This function aids in testing instruction format determination and presentation.

```
void printInstructionFormats(const vector<string>& binaryNumbers)
{
    //This function only prints on console for testing
    cout << "-------------------------------------------------------------------------------------------------------------------------------------------------------" << endl;
    cout << "Instruction Formats:" << endl;
    for (const auto& binaryNumber : binaryNumbers)
    {

        InstructionFormat format = determineInstructionFormat(binaryNumber, memory);
        switch (format) {
        case InstructionFormat::StandardSIC:

            cout << "Standard SIC  (24 bits" << ", n =" << n << ", i =" << i << ",x=" << x << ", opcode = " << op << ",address=" << address << ", PC= " << PC << ", TA=" << TA << ")" << endl;
            break;
        case InstructionFormat::Format1:
            cout << "Format 1  (8 bits, opcode only" << ", opcode=" << op << ", PC=" << PC << ")" << endl;
            break;
        case InstructionFormat::Format2:
            cout << "Format 2  (16 bits, register operands," << " opcode = " << op << ", r1=" << r1 << ",r2=" << r2 << ", PC=" << PC << ")" << endl;
            break;
        case InstructionFormat::Format3:

            cout << "Format 3  (24 bits," << "opcode = " << op << ",n=" << n << ",i=" << i << ",x=" << x << ",b=" << b << ",p=" << p << ",e=" << e << ",disp=" << disp << ", PC=" << PC << ", TA=" << TA << ")" << endl;
            break;
        case InstructionFormat::Format4:
            cout << "Format 4  (32 bits," << "opcode=" << op << ",n=" << n << ",i=" << i << ",x=" << x << ",b=" << b << ",p=" << p << ",e=" << e << ",address=" << address << ", PC=" << PC << ", TA=" << TA << ")" << endl;
            break;
        default:
            cout << "Invalid format" << endl;
            break;
        }
    }
}
```

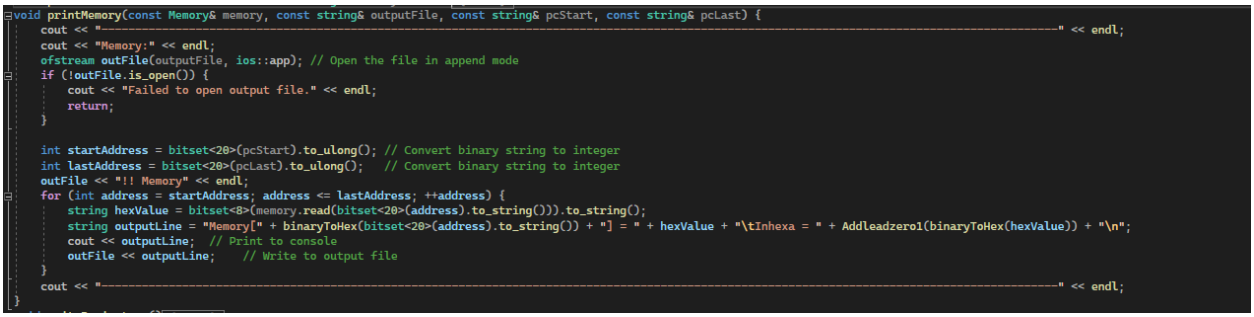
ii. Print memory:

This code defines a function called printMemory which serves to display and save memory contents. The function takes in a Memory object representing the memory structure, a string outputFile for saving the output, and two binary address strings pcStart and pcLast to specify the memory range.

Inside the function, the output file is opened in append mode, and if this operation fails, an error message is printed and the function returns. The provided binary address strings are converted into integer values to establish the range of memory addresses to work with.

The function then iterates through memory addresses within the specified range. It converts memory content into hexadecimal values and constructs output lines with proper formatting. These output lines are both displayed on the console and written to the output file.

In essence, the printMemory function ensures that memory contents are presented in a human-readable manner, both through console output and written to an output file. It handles necessary conversions and formatting for clarity.



```
void printMemory(const Memory& memory, const string& outputFile, const string& pcStart, const string& pcLast) {
    cout << "-------------------------------------------------------------------------------------------------------------------------------------------------------------------" << endl;
    cout << "Memory:" << endl;
    ofstream outFile(outputFile, ios::app); // Open the file in append mode
    if (!outFile.is_open()) {
        cout << "Failed to open output file." << endl;
        return;
    }

    int startAddress = bitset<20>(pcStart).to_ulong(); // Convert binary string to integer
    int lastAddress = bitset<20>(pcLast).to_ulong();   // Convert binary string to integer
    outFile << "!! Memory" << endl;
    for (int address = startAddress; address <= lastAddress; ++address) {
        string hexValue = bitset<8>(memory.read(bitset<20>(address).to_string())).to_string();
        string outputLine = "Memory[" + binaryToHex(bitset<20>(address).to_string()) + "] = " + hexValue + "\tInhexa = " + Addleadzero1(binaryToHex(hexValue)) + "\n";
        cout << outputLine;  // Print to console
        outFile << outputLine;    // Write to output file
    }
    cout << "-------------------------------------------------------------------------------------------------------------------------------------------------------------------" << endl;
}
```

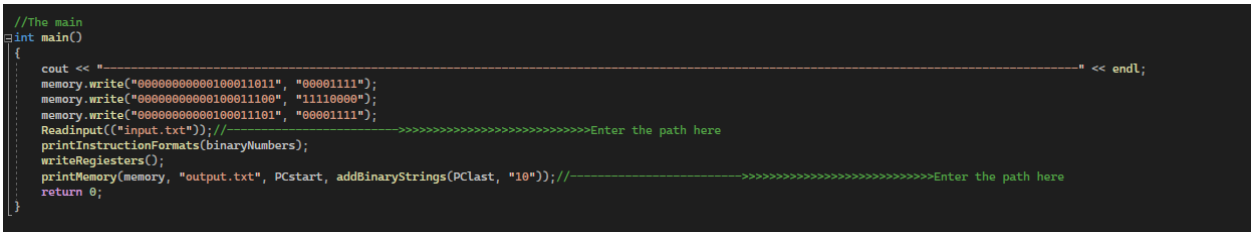
iii. WriteRegisters:

The writeRegisters function aims to save register values to an output file. It opens the file, writes a header, records each register's name, binary value, and leading-zero-padded hexadecimal value, then closes the file. This ensures readable storage of register data. Specifics like functions Addleadzero2, binaryToHex, and the output file path are used.

```
void writeRegiesters()
{
    ofstream outputFile("output.txt", ios::app);//------------------------>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>Enter the path here

    if (!outputFile.is_open()) {
        cerr << "Unable to open the output file." << endl;
        return;
    }

    outputFile << "!! Registers" << endl;
    outputFile << "A=" << A << "\t0x" << Addleadzero2(binaryToHex(A)) << endl;
    outputFile << "X=" << X << "\t0x" << Addleadzero2(binaryToHex(X)) << endl;
    outputFile << "L=" << L << "\t0x" << Addleadzero2(binaryToHex(L)) << endl;
    outputFile << "PC=" << PC << "\t\t0x" << Addleadzero2(binaryToHex(PC)) << endl;
    outputFile << "SW=" << SW << "\t0x" << Addleadzero2(binaryToHex(SW)) << endl;
    outputFile << "B=" << B << "\t0x" << Addleadzero2(binaryToHex(B)) << endl;
    outputFile << "S=" << S << "\t0x" << Addleadzero2(binaryToHex(S)) << endl;
    outputFile << "T=" << T << "\t0x" << Addleadzero2(binaryToHex(T)) << endl;
    outputFile << endl;
    outputFile.close();
}
```

iv. The main:

snippet demonstrates memory manipulation and data processing. The program begins with a separator of dashes and uses a simulated memory object for write operations. It references functions like Readinput(), printInstructionFormats(), writeRegisters(), and printMemory() for input/output tasks. The program's concise nature invites further exploration into its functions and purpose.



```
//The main
int main()
{
    cout << "-------------------------------------------------------------------------------------------------------------------------------------------------------" << endl;
    memory.write("0000000000100011011", "00001111");
    memory.write("0000000000100011100", "11110000");
    memory.write("0000000000100011101", "00001111");
    Readinput(("input.txt"));//------------------------>>>>>>>>>>>>>>>>>>>>>>>>>>>>Enter the path here
    printInstructionFormats(binaryNumbers);
    writeRegiesters();
    printMemory(memory, "output.txt", PCstart, addBinaryStrings(PClast, "10"));//------------------------>>>>>>>>>>>>>>>>>>>>>>>>>>>>Enter the path here
    return 0;
}
```

Eighth: Test plans

For all test plans was selected carefully to avoid repeating or to use them in another instruction at the same test considering that we could not use all the instructions in the code so we just used the necessary ones however the instructions that was not used was tested and another instruction similar to it was used.

Note: IO is constant input file for all tests

```
!! I/O 'F1'
This is a rainy day!!

!! I/O '05'
Renewable energy is energy generated from natural resources such as
sunlight, wind, rain, tides, and geothermal heat, which are renewable (naturally replenished).
```

1) First Test:

The input:

```
!! START = 00100
!! LAST = 0011C

!! CODE
!!      MSB_...._LSB

!! instructions
01_00_30
19_00_01
0F_20_03
6B_20_00
98_30
AC34
AC03
AC45
9C_00
E1_00_05
D9_00_05
DD_00_05
```

Console output:

```
-----------------------------------------------------------------------------------------------------------------------------------------------------------
Found START: 000100 (Binary: 0000000000100000000)
Found LAST: 00011c (Binary: 0000000000100011100)
-----------------------------------------------------------------------------------------------------------------------------------------------------------
Machine Codes:
010030 (Binary: 000000010000000000110000)
190001 (Binary: 000110010000000000000001)
0f2003 (Binary: 000011110010000000000011)
6b2000 (Binary: 011010110010000000000000)
009830 (Binary: 1001100000110000)
00ac34 (Binary: 1010110000110100)
00ac03 (Binary: 1010110000000011)
00ac45 (Binary: 1010110001000101)
009c00 (Binary: 1001110000000000)
e10005 (Binary: 111000010000000000000101)
d90005 (Binary: 110110010000000000000101)
dd0005 (Binary: 110111010000000000000101)
-----------------------------------------------------------------------------------------------------------------------------------------------------------
Instruction Formats:
LDA,Format 3  (24 bits,opcode = 00000000,n=0,i=1,x=0,b=0,p=0,e=0,disp=000000110000, PC=0000000000100000011, TA=000000110000)
ADD,Format 3  (24 bits,opcode = 00011000,n=0,i=1,x=0,b=0,p=0,e=0,disp=000000000001, PC=0000000000100000110, TA=000000000001)
STA,Format 3  (24 bits,opcode = 00001100,n=1,i=1,x=0,b=0,p=1,e=0,disp=000000000011, PC=0000000000100001001, TA=0000000000100001100)
LDB,Format 3  (24 bits,opcode = 01101000,n=1,i=1,x=0,b=0,p=1,e=0,disp=000000000000, PC=0000000000100001100, TA=0000000000100001100)
MULR,Format 2  (16 bits, register operands, opcode = 10011000, r1=0011,r2=0000, PC=0000000000100001110)
RMO,Format 2  (16 bits, register operands, opcode = 10101100, r1=0011,r2=0100, PC=0000000000100010000)
RMO,Format 2  (16 bits, register operands, opcode = 10101100, r1=0000,r2=0011, PC=0000000000100010010)
RMO,Format 2  (16 bits, register operands, opcode = 10101100, r1=0100,r2=0101, PC=0000000000100010100)
DIVR,Format 2  (16 bits, register operands, opcode = 10011100, r1=0000,r2=0000, PC=0000000000100010110)
TD,Device is ready.Format 3  (24 bits,opcode = 11100000,n=0,i=1,x=0,b=0,p=0,e=0,disp=000000000101, PC=0000000000100011001, TA=000000000101)
RD,Format 3  (24 bits,opcode = 11011000,n=0,i=1,x=0,b=0,p=0,e=0,disp=000000000101, PC=0000000000100011100, TA=000000000101)
WD,Format 3  (24 bits,opcode = 11011100,n=0,i=1,x=0,b=0,p=0,e=0,disp=000000000101, PC=0000000000100011111, TA=000000000101)
-----------------------------------------------------------------------------------------------------------------------------------------------------------
```

```
-------------------------------------
Memory:
Memory[100] = 00000001  Inhexa = 01
Memory[101] = 00000000  Inhexa = 00
Memory[102] = 00110000  Inhexa = 30
Memory[103] = 00011001  Inhexa = 19
Memory[104] = 00000000  Inhexa = 00
Memory[105] = 00000001  Inhexa = 01
Memory[106] = 00001111  Inhexa = 0F
Memory[107] = 00100000  Inhexa = 20
Memory[108] = 00000011  Inhexa = 03
Memory[109] = 01101011  Inhexa = 6B
Memory[10A] = 00100000  Inhexa = 20
Memory[10B] = 00000000  Inhexa = 00
Memory[10C] = 10011000  Inhexa = 98
Memory[10D] = 00110000  Inhexa = 30
Memory[10E] = 10101100  Inhexa = AC
Memory[10F] = 00110100  Inhexa = 34
Memory[110] = 10101100  Inhexa = AC
Memory[111] = 00000011  Inhexa = 03
Memory[112] = 10101100  Inhexa = AC
Memory[113] = 01000101  Inhexa = 45
Memory[114] = 10011100  Inhexa = 9C
Memory[115] = 00000000  Inhexa = 00
Memory[116] = 11100001  Inhexa = E1
Memory[117] = 00000000  Inhexa = 00
Memory[118] = 00000101  Inhexa = 05
Memory[119] = 11011001  Inhexa = D9
Memory[11A] = 00000000  Inhexa = 00
Memory[11B] = 00000101  Inhexa = 05
Memory[11C] = 11011101  Inhexa = DD
Memory[11D] = 00000000  Inhexa = 00
Memory[11E] = 00000101  Inhexa = 05
-------------------------------------
```

File output:

```
!!I/O '05'
Renewable energy is energy generated from natural resources such as
sunlight, wind, rain, tides, and geothermal heat, which are renewable (naturally replenished).

!! Registers
A=00000000000000000000001      0x00000001
X=00000000000000000000000      0x00000000
L=00000000000000000000000      0x00000000
PC=00000000000100011111        0x0000011F
SW=00000000000000000000000     0x00000000
B=00000000000100101100001      0x00000961
S=00000000000000000110001      0x00000031
T=00000000000000000110001      0x00000031

!! Memory
Memory[100] = 00000001  Inhexa = 01
Memory[101] = 00000000  Inhexa = 00
Memory[102] = 00110000  Inhexa = 30
Memory[103] = 00011001  Inhexa = 19
Memory[104] = 00000000  Inhexa = 00
Memory[105] = 00000001  Inhexa = 01
Memory[106] = 00001111  Inhexa = 0F
Memory[107] = 00100000  Inhexa = 20
Memory[108] = 00000011  Inhexa = 03
Memory[109] = 01101011  Inhexa = 6B
Memory[10A] = 00100000  Inhexa = 20
Memory[10B] = 00000000  Inhexa = 00
Memory[10C] = 10011000  Inhexa = 98
Memory[10D] = 00110000  Inhexa = 30
Memory[10E] = 10101100  Inhexa = AC
Memory[10F] = 00110100  Inhexa = 34
Memory[110] = 10101100  Inhexa = AC
Memory[111] = 00000011  Inhexa = 03
Memory[112] = 10101100  Inhexa = AC
Memory[113] = 01000101  Inhexa = 45
Memory[114] = 10011100  Inhexa = 9C
Memory[115] = 00000000  Inhexa = 00
Memory[116] = 11100001  Inhexa = E1
Memory[117] = 00000000  Inhexa = 00
Memory[118] = 00000101  Inhexa = 05
Memory[119] = 11011001  Inhexa = D9
Memory[11A] = 00000000  Inhexa = 00
Memory[11B] = 00000101  Inhexa = 05
Memory[11C] = 11011101  Inhexa = DD
Memory[11D] = 00000000  Inhexa = 00
Memory[11E] = 00000101  Inhexa = 05
```

2) Second test:

The input:

```
!! START = 00100
!! LAST = 0011E

!! CODE
!!      MSB_...._LSB

!! instructions
01_0F_30
51_00_12
69_0F_15
94_03
A0_30
E9_20_10
27_20_0A
A4_31
E1_00_F1
D9_00_F1
DD_00_F1
3D_20_0A
```

The console output:

```
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
Found START: 000100 (Binary: 00000000000100000000)
Found LAST: 00011e (Binary: 00000000000100011110)
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
Machine Codes:
010f30 (Binary: 000000010000111100110000)
510012 (Binary: 010100010000000000010010)
690f15 (Binary: 011010010000111100010101)
009403 (Binary: 1001010000000011)
00a030 (Binary: 1010000000110000)
e92010 (Binary: 111010010010000000010000)
27200a (Binary: 001001110010000000001010)
00a431 (Binary: 1010010000110001)
e100f1 (Binary: 111000010000000011110001)
d900f1 (Binary: 110110010000000011110001)
dd00f1 (Binary: 110111010000000011110001)
3d200a (Binary: 001111010010000000001010)
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
Instruction Formats:
LDA,Format 3  (24 bits,opcode = 00000000,n=0,i=1,x=0,b=0,p=0,e=0,disp=111100110000, PC=0000000000100000011, TA=111100110000)
LDCH,Format 3  (24 bits,opcode = 01010000,n=0,i=1,x=0,b=0,p=0,e=0,disp=000000010010, PC=0000000000100000110, TA=000000010010)
LDB,Format 3  (24 bits,opcode = 01101000,n=0,i=1,x=0,b=0,p=0,e=0,disp=111100010101, PC=0000000000100001001, TA=111100010101)
SUBR,Format 2  (16 bits, register operands, opcode = 10010100, r1=0000,r2=0011, PC=0000000000100001011)
COMPR,Format 2  (16 bits, register operands, opcode = 10100000, r1=0011,r2=0000, PC=0000000000100001101)
STSW,Format 3  (24 bits,opcode = 11101000,n=0,i=1,x=0,b=0,p=1,e=0,disp=000000010000, PC=0000000000100010000, TA=0000000000100011101)
DIV,Format 3  (24 bits,opcode = 00100100,n=1,i=1,x=0,b=0,p=1,e=0,disp=000000001010, PC=0000000000100010011, TA=0000000000000000100011101)
SHIFTL,Format 2  (16 bits, register operands, opcode = 10100100, r1=0011,r2=0001, PC=0000000000100010101)
TD,Device is ready.Format 3  (24 bits,opcode = 11100000,n=0,i=1,x=0,b=0,p=0,e=0,disp=000011110001, PC=0000000000100011000, TA=000011110001)
RD,Format 3  (24 bits,opcode = 11011000,n=0,i=1,x=0,b=0,p=0,e=0,disp=000011110001, PC=0000000000100011011, TA=000011110001)
WD,Format 3  (24 bits,opcode = 11011100,n=0,i=1,x=0,b=0,p=0,e=0,disp=000011110001, PC=0000000000100011110, TA=000011110001)
JUMP,Format 3  (24 bits,opcode = 00111100,n=0,i=1,x=0,b=0,p=1,e=0,disp=000000001010, PC=0000000000100101000, TA=0000000000100101000)
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
Memory:
Memory[100] = 00000001  Inhexa = 01
Memory[101] = 00001111  Inhexa = 0F
Memory[102] = 00110000  Inhexa = 30
Memory[103] = 01010001  Inhexa = 51
Memory[104] = 00000000  Inhexa = 00
Memory[105] = 00010010  Inhexa = 12
Memory[106] = 01101001  Inhexa = 69
Memory[107] = 00001111  Inhexa = 0F
Memory[108] = 00010101  Inhexa = 15
Memory[109] = 10010100  Inhexa = 94
Memory[10A] = 00000011  Inhexa = 03
Memory[10B] = 10100000  Inhexa = A0
Memory[10C] = 00110000  Inhexa = 30
Memory[10D] = 11101001  Inhexa = E9
Memory[10E] = 00100000  Inhexa = 20
Memory[10F] = 00010000  Inhexa = 10
Memory[110] = 00100111  Inhexa = 27
Memory[111] = 00100000  Inhexa = 20
Memory[112] = 00001010  Inhexa = 0A
Memory[113] = 10100100  Inhexa = A4
Memory[114] = 00110001  Inhexa = 31
Memory[115] = 11100001  Inhexa = E1
Memory[116] = 00000000  Inhexa = 00
Memory[117] = 11110001  Inhexa = F1
Memory[118] = 11011001  Inhexa = D9
Memory[119] = 00000000  Inhexa = 00
Memory[11A] = 11110001  Inhexa = F1
Memory[11B] = 11011101  Inhexa = DD
Memory[11C] = 00000000  Inhexa = 00
Memory[11D] = 11110001  Inhexa = F1
Memory[11E] = 00111101  Inhexa = 3D
Memory[11F] = 00100000  Inhexa = 20
Memory[120] = 00001010  Inhexa = 0A
```

File output:

```
!!I/O 'F1'
This is a rainy day!!

!! Registers
A=00000000010111100101110       0x00005E5E
X=00000000000000000000000       0x00000000
L=00000000000000000000000       0x00000000
PC=00000000000100101000         0x00000128
SW=00000000000000000000000      0x00000000
B=00000000000000000001100       0x0000000C
S=00000000000000000000000       0x00000000
T=00000000000000000000000       0x00000000
```

```
!! Memory
Memory[100] = 00000001  Inhexa = 01
Memory[101] = 00001111  Inhexa = 0F
Memory[102] = 00110000  Inhexa = 30
Memory[103] = 01010001  Inhexa = 51
Memory[104] = 00000000  Inhexa = 00
Memory[105] = 00010010  Inhexa = 12
Memory[106] = 01101001  Inhexa = 69
Memory[107] = 00001111  Inhexa = 0F
Memory[108] = 00010101  Inhexa = 15
Memory[109] = 10010100  Inhexa = 94
Memory[10A] = 00000011  Inhexa = 03
Memory[10B] = 10100000  Inhexa = A0
Memory[10C] = 00110000  Inhexa = 30
Memory[10D] = 11101001  Inhexa = E9
Memory[10E] = 00100000  Inhexa = 20
Memory[10F] = 00010000  Inhexa = 10
Memory[110] = 00100111  Inhexa = 27
Memory[111] = 00100000  Inhexa = 20
Memory[112] = 00001010  Inhexa = 0A
Memory[113] = 10100100  Inhexa = A4
Memory[114] = 00110001  Inhexa = 31
Memory[115] = 11100001  Inhexa = E1
Memory[116] = 00000000  Inhexa = 00
Memory[117] = 11110001  Inhexa = F1
Memory[118] = 11011001  Inhexa = D9
Memory[119] = 00000000  Inhexa = 00
Memory[11A] = 11110001  Inhexa = F1
Memory[11B] = 11011101  Inhexa = DD
Memory[11C] = 00000000  Inhexa = 00
Memory[11D] = 11110001  Inhexa = F1
Memory[11E] = 00111101  Inhexa = 3D
Memory[11F] = 00100000  Inhexa = 20
Memory[120] = 00001010  Inhexa = 0A
```

3) Third test:

The input:

```
!! START = 00100
!! LAST = 00115

!! CODE
!!      MSB_...._LSB

!! instructions
09_00_30
05_00_02
A8_10
AC_20
41_00_20
45_00_0A
B8_20
AC_03
AC_34
B4_30
```

Console output:

```
------------------------------------------------------------------------------------------------------------------------------------------------------
Found START: 000100 (Binary: 0000000000100000000)
Found LAST: 000115 (Binary: 0000000000100010101)
------------------------------------------------------------------------------------------------------------------------------------------------------
Machine Codes:
090030 (Binary: 000010010000000000110000)
050002 (Binary: 000001010000000000000010)
00a810 (Binary: 1010100000010000)
00ac20 (Binary: 1010110000100000)
410020 (Binary: 010000010000000000100000)
45000a (Binary: 010001010000000000001010)
00b820 (Binary: 1011100000100000)
00ac03 (Binary: 1010110000000011)
00ac34 (Binary: 1010110000110100)
00b430 (Binary: 1011010000110000)
------------------------------------------------------------------------------------------------------------------------------------------------------
Instruction Formats:
LDL,Format 3  (24 bits,opcode = 00001000,n=0,i=1,x=0,b=0,p=0,e=0,disp=000000110000, PC=0000000000100000011, TA=000000110000)
LDX,Format 3  (24 bits,opcode = 00000100,n=0,i=1,x=0,b=0,p=0,e=0,disp=000000000010, PC=0000000000100000110, TA=000000000010)
SHIFTR,Format 2  (16 bits, register operands, opcode = 10101000, r1=0001,r2=0000, PC=0000000000100001000)
RMO,Format 2  (16 bits, register operands, opcode = 10101100, r1=0010,r2=0000, PC=0000000000100001010)
AND,Format 3  (24 bits,opcode = 01000000,n=0,i=1,x=0,b=0,p=0,e=0,disp=000000100000, PC=0000000000100001101, TA=00000000000000000100000)
OR,Format 3  (24 bits,opcode = 01000100,n=0,i=1,x=0,b=0,p=0,e=0,disp=000000001010, PC=0000000000100010000, TA=00000000000000000001010)
TIXR,Format 2  (16 bits, register operands, opcode = 10111000, r1=0010,r2=0000, PC=0000000000100010010)
RMO,Format 2  (16 bits, register operands, opcode = 10101100, r1=0000,r2=0011, PC=0000000000100010100)
RMO,Format 2  (16 bits, register operands, opcode = 10101100, r1=0011,r2=0100, PC=0000000000100010110)
CLEAR,Format 2  (16 bits, register operands, opcode = 10110100, r1=0011,r2=0000, PC=0000000000100011000)
------------------------------------------------------------------------------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------------------------------------------------------------------------------
Memory:
Memory[100] = 00001001  Inhexa = 09
Memory[101] = 00000000  Inhexa = 00
Memory[102] = 00110000  Inhexa = 30
Memory[103] = 00000101  Inhexa = 05
Memory[104] = 00000000  Inhexa = 00
Memory[105] = 00000010  Inhexa = 02
Memory[106] = 10101000  Inhexa = A8
Memory[107] = 00010000  Inhexa = 10
Memory[108] = 10101100  Inhexa = AC
Memory[109] = 00100000  Inhexa = 20
Memory[10A] = 01000001  Inhexa = 41
Memory[10B] = 00000000  Inhexa = 00
Memory[10C] = 00100000  Inhexa = 20
Memory[10D] = 01000101  Inhexa = 45
Memory[10E] = 00000000  Inhexa = 00
Memory[10F] = 00001010  Inhexa = 0A
Memory[110] = 10111000  Inhexa = B8
Memory[111] = 00100000  Inhexa = 20
Memory[112] = 10101100  Inhexa = AC
Memory[113] = 00000011  Inhexa = 03
Memory[114] = 10101100  Inhexa = AC
Memory[115] = 00110100  Inhexa = 34
Memory[116] = 10110100  Inhexa = B4
Memory[117] = 00110000  Inhexa = 30
------------------------------------------------------------------------------------------------------------------------------------------------------
```

File output:

```
!! Registers
A=00000000000000000101010       0x0000002A
X=00000000000000000000010       0x00000002
L=00000000000000000110000       0x00000030
PC=00000000000100011000         0x00000118
SW=00000000000000000000010      0x00000002
B=00000000000000000000000       0x00000000
S=00000000000000000101010       0x0000002A
T=00000000000000000000000       0x00000000
```

```
!! Memory
Memory[100] = 00001001  Inhexa = 09
Memory[101] = 00000000  Inhexa = 00
Memory[102] = 00110000  Inhexa = 30
Memory[103] = 00000101  Inhexa = 05
Memory[104] = 00000000  Inhexa = 00
Memory[105] = 00000010  Inhexa = 02
Memory[106] = 10101000  Inhexa = A8
Memory[107] = 00010000  Inhexa = 10
Memory[108] = 10101100  Inhexa = AC
Memory[109] = 00100000  Inhexa = 20
Memory[10A] = 01000001  Inhexa = 41
Memory[10B] = 00000000  Inhexa = 00
Memory[10C] = 00100000  Inhexa = 20
Memory[10D] = 01000101  Inhexa = 45
Memory[10E] = 00000000  Inhexa = 00
Memory[10F] = 00001010  Inhexa = 0A
Memory[110] = 10111000  Inhexa = B8
Memory[111] = 00100000  Inhexa = 20
Memory[112] = 10101100  Inhexa = AC
Memory[113] = 00000011  Inhexa = 03
Memory[114] = 10101100  Inhexa = AC
Memory[115] = 00110100  Inhexa = 34
Memory[116] = 10110100  Inhexa = B4
Memory[117] = 00110000  Inhexa = 30
```

Forth test:(The one in resources)

Input file:

```
!!
!!
!!
!! This is a comment line


!! START = 00100
!! LAST = 00118

!! CODE
!!      MSB_...._LSB

!! instructions
03_20_18
6B_20_15
0B_20_12
6F_20_0F
77_20_0C
07_20_09
1B_20_06
2B_20_03
0F_20_03
00_00_01
```

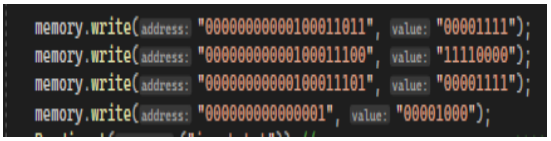
Console output:

```
------------------------------------------------------------------------------------------------------------------------------------------------------
Found START: 000100 (Binary: 0000000000100000000)
Found LAST: 000118 (Binary: 0000000000100011000)
------------------------------------------------------------------------------------------------------------------------------------------------------
Machine Codes:
032018 (Binary: 000000110010000000011000)
6b2015 (Binary: 011010110010000000010101)
0b2012 (Binary: 000010110010000000010010)
6f200f (Binary: 011011110010000000001111)
77200c (Binary: 011101110010000000001100)
072009 (Binary: 000001110010000000001001)
1b2006 (Binary: 000110110010000000000110)
2b2003 (Binary: 001010110010000000000011)
0f2003 (Binary: 000011110010000000000011)
000001 (Binary: 000000000000000000000001)
------------------------------------------------------------------------------------------------------------------------------------------------------
Instruction Formats:
LDA,Format 3  (24 bits,opcode = 00000000,n=1,i=1,x=0,b=0,p=1,e=0,disp=000000011000, PC=0000000000100000011, TA=0000000000100011011)
LDB,Format 3  (24 bits,opcode = 01101000,n=1,i=1,x=0,b=0,p=1,e=0,disp=000000010101, PC=0000000000100000110, TA=0000000000100011011)
LDL,Format 3  (24 bits,opcode = 00001000,n=1,i=1,x=0,b=0,p=1,e=0,disp=000000010010, PC=0000000000100001001, TA=0000000000100011011)
LDS,Format 3  (24 bits,opcode = 01101100,n=1,i=1,x=0,b=0,p=1,e=0,disp=000000001111, PC=0000000000100001100, TA=0000000000100011011)
LDT,Format 3  (24 bits,opcode = 01110100,n=1,i=1,x=0,b=0,p=1,e=0,disp=000000001100, PC=0000000000100001111, TA=0000000000100011011)
LDX,Format 3  (24 bits,opcode = 00000100,n=1,i=1,x=0,b=0,p=1,e=0,disp=000000001001, PC=0000000000100010010, TA=0000000000100011011)
ADD,Format 3  (24 bits,opcode = 00011000,n=1,i=1,x=0,b=0,p=1,e=0,disp=000000000110, PC=0000000000100010101, TA=0000000000100011011)
COMP,Format 3  (24 bits,opcode = 00101000,n=1,i=1,x=0,b=0,p=1,e=0,disp=000000000011, PC=0000000000100011000, TA=0000000000100011011)
STA,Format 3  (24 bits,opcode = 00001100,n=1,i=1,x=0,b=0,p=1,e=0,disp=000000000011, PC=0000000000100011011, TA=0000000000100011110)
LDA,Standard SIC  (24 bits, n =0, i =0,x=0, opcode = 00000000,address=000000000000001, PC= 0000000000100011110, TA=000000000000001)
------------------------------------------------------------------------------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------------------------------------------------------------------------------
Memory:
Memory[100] = 00000011  Inhexa = 03
Memory[101] = 00100000  Inhexa = 20
Memory[102] = 00011000  Inhexa = 18
Memory[103] = 01101011  Inhexa = 6B
Memory[104] = 00100000  Inhexa = 20
Memory[105] = 00010101  Inhexa = 15
Memory[106] = 00001011  Inhexa = 0B
Memory[107] = 00100000  Inhexa = 20
Memory[108] = 00010010  Inhexa = 12
Memory[109] = 01101111  Inhexa = 6F
Memory[10A] = 00100000  Inhexa = 20
Memory[10B] = 00001111  Inhexa = 0F
Memory[10C] = 01110111  Inhexa = 77
Memory[10D] = 00100000  Inhexa = 20
Memory[10E] = 00001100  Inhexa = 0C
Memory[10F] = 00000111  Inhexa = 07
Memory[110] = 00100000  Inhexa = 20
Memory[111] = 00001001  Inhexa = 09
Memory[112] = 00011011  Inhexa = 1B
Memory[113] = 00100000  Inhexa = 20
Memory[114] = 00000110  Inhexa = 06
Memory[115] = 00101011  Inhexa = 2B
Memory[116] = 00100000  Inhexa = 20
Memory[117] = 00000011  Inhexa = 03
Memory[118] = 00001111  Inhexa = 0F
Memory[119] = 00100000  Inhexa = 20
Memory[11A] = 00000011  Inhexa = 03
------------------------------------------------------------------------------------------------------------------------------------------------------
```

File output:

At first we filled some memory locations(TA) with some data




```
!! Registers
A=00000000000000000000001000      0x00000008
X=00001111111110000000001111      0x000FF00F
L=00001111111110000000001111      0x000FF00F
PC=00000000000100011110           0x0000011E
SW=00000000000000000000000001     0x00000001
B=00001111111110000000001111      0x000FF00F
S=00001111111110000000001111      0x000FF00F
T=00001111111110000000001111      0x000FF00F


!! Memory
Memory[100] = 00000011  Inhexa = 03
Memory[101] = 00100000  Inhexa = 20
Memory[102] = 00011000  Inhexa = 18
Memory[103] = 01101011  Inhexa = 6B
Memory[104] = 00100000  Inhexa = 20
Memory[105] = 00010101  Inhexa = 15
Memory[106] = 00001011  Inhexa = 0B
Memory[107] = 00100000  Inhexa = 20
Memory[108] = 00010010  Inhexa = 12
Memory[109] = 01101111  Inhexa = 6F
Memory[10A] = 00100000  Inhexa = 20
Memory[10B] = 00001111  Inhexa = 0F
Memory[10C] = 01110111  Inhexa = 77
Memory[10D] = 00100000  Inhexa = 20
Memory[10E] = 00001100  Inhexa = 0C
Memory[10F] = 00000111  Inhexa = 07
Memory[110] = 00100000  Inhexa = 20
Memory[111] = 00001001  Inhexa = 09
Memory[112] = 00011011  Inhexa = 1B
Memory[113] = 00100000  Inhexa = 20
Memory[114] = 00000110  Inhexa = 06
Memory[115] = 00101011  Inhexa = 2B
Memory[116] = 00100000  Inhexa = 20
Memory[117] = 00000011  Inhexa = 03
Memory[118] = 00001111  Inhexa = 0F
Memory[119] = 00100000  Inhexa = 20
Memory[11A] = 00000011  Inhexa = 03
```

Conclusion:

Certainly! In conclusion, the provided C++ program is a simulator for the SIC/XE architecture, a historical computer architecture used in mainframes. The program reads a list of hexadecimal instructions from an input file, processes them according to their opcode and instruction format, and simulates their execution while maintaining the state of registers, memory, and flags.

The program demonstrates several key concepts:

Instruction Formats: It handles various instruction formats, including Standard SIC, Format 1, Format 2, Format 3, and Format 4. Each format has different requirements for extracting opcode, registers, addresses, and other information.

Memory Simulation: The program simulates memory using a Memory class, allowing read and write operations to memory locations. This is crucial for storing and retrieving data during program execution.

Opcode Processing: Instructions are identified and processed based on their opcodes. The program performs different operations based on the opcode, such as addition, logical AND, shifting, etc.

Register Management: The simulator keeps track of registers (A, X, L, PC, etc.) and updates their values according to the executed instructions.

Output Generation: The program generates output in the form of printed instruction formats, memory contents, and register values. It writes some of this output to files and displays some on the console.

Hexadecimal and Binary Conversion: The program effectively converts hexadecimal input to binary and vice versa, which is a fundamental aspect of working with low-level computer systems.

The End