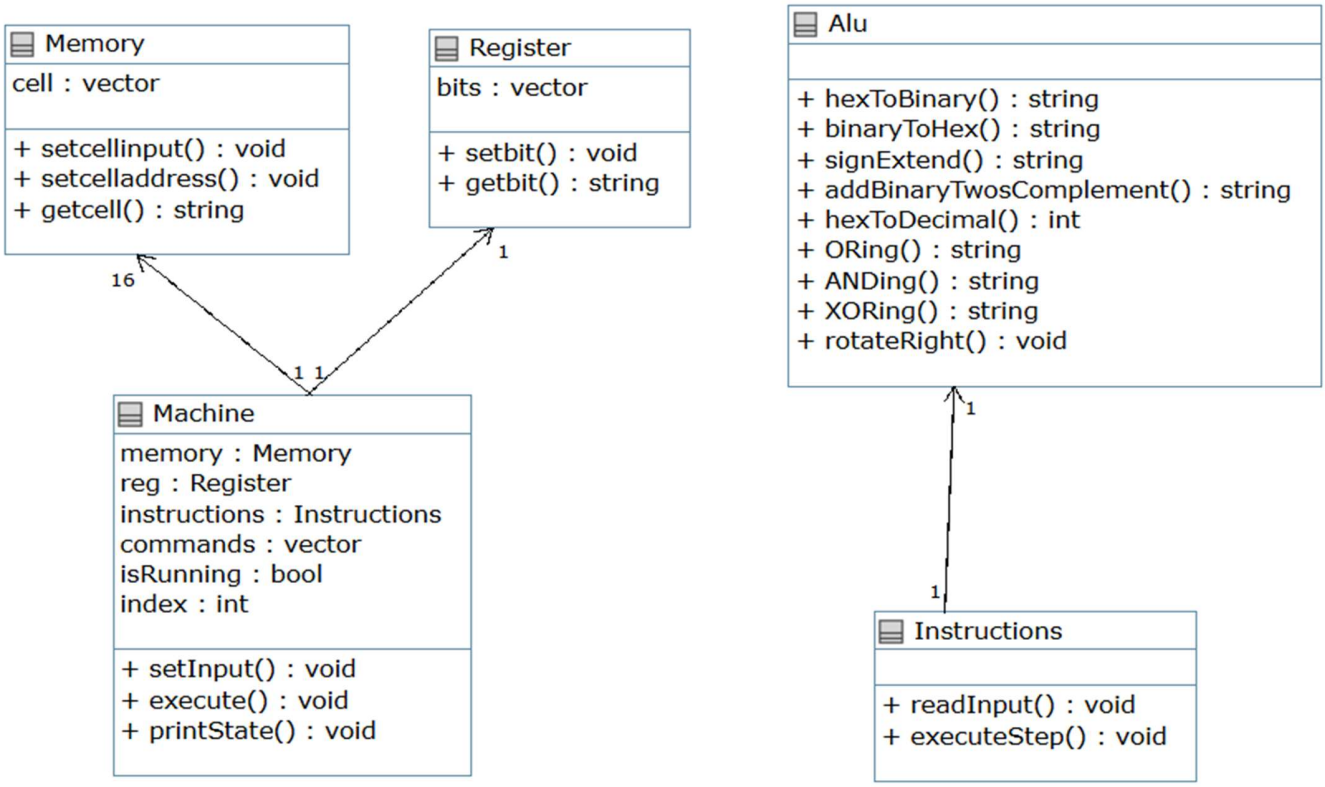
Faculty of computers and artificial intelligence

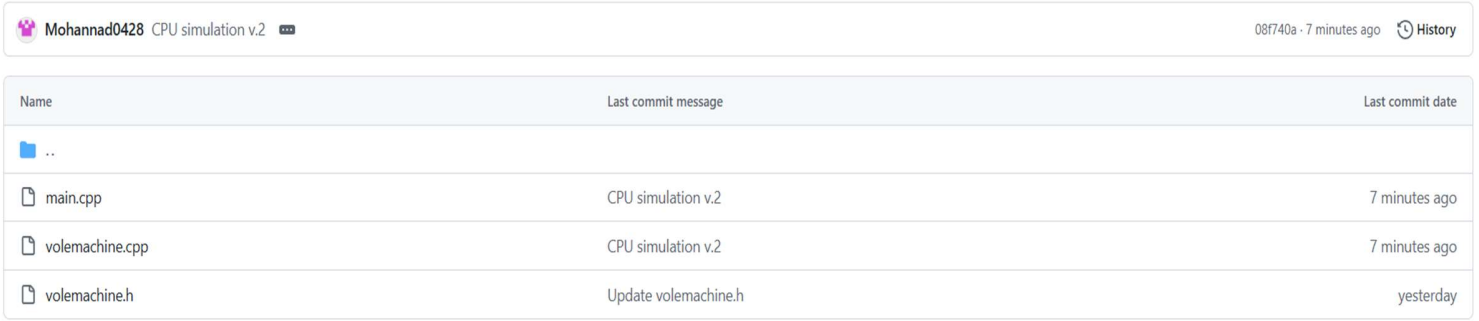
CS213 - Object Oriented Programming

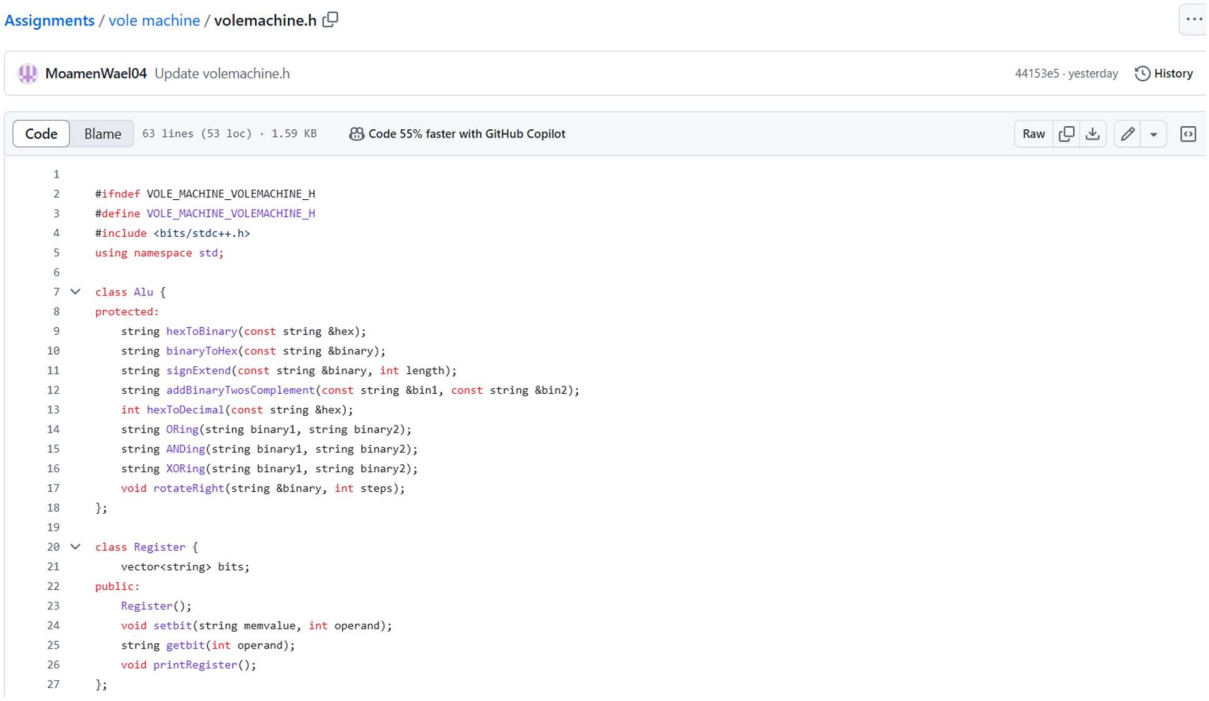
name	id	What he have done
Moamen wael	20230434	UML Design& Register class &Memory class
Mohanad essam	20230428	Alu class, Machine Class
Ahmed abdelsamea	20231010	Report& Instructions class

UML Design:


Memory
cell : vector
+ setcellinput() : void
+ setcelladdress() : void
+ getcell() : string
Register
bits : vector
+ setbit() : void
+ getbit() : string
Alu
+ hexToBinary() : string
+ binaryToHex() : string
+ signExtend() : string
+ addBinaryTwosComplement() : string
+ hexToDecimal() : int
+ ORing() : string
+ ANDing() : string
+ XORing() : string
+ rotateRight() : void
16
1
1
1
1
1
Machine
memory : Memory
reg : Register
instructions : Instructions
commands : vector
isRunning : bool
index : int
+ setInput() : void
+ execute() : void
+ printState() : void
Instructions
+ readInput() : void
+ executeStep() : void

GitHub Screenshot:






```
1
2   #ifndef VOLE_MACHINE_VOLEMACHINE_H
3   #define VOLE_MACHINE_VOLEMACHINE_H
4   #include <bits/stdc++.h>
5   using namespace std;
6
7   class Alu {
8   protected:
9       string hexToBinary(const string &hex);
10      string binaryToHex(const string &binary);
11      string signExtend(const string &binary, int length);
12      string addBinaryTwosComplement(const string &bin1, const string &bin2);
13      int hexToDecimal(const string &hex);
14      string ORing(string binary1, string binary2);
15      string ANDing(string binary1, string binary2);
16      string XORing(string binary1, string binary2);
17      void rotateRight(string &binary, int steps);
18  };
19
20  class Register {
21      vector<string> bits;
22  public:
23      Register();
24      void setbit(string memvalue, int operand);
25      string getbit(int operand);
26      void printRegister();
27  };
```

Code Blame 440 lines (403 loc) · 15.4 KB Code 55% faster with GitHub Copilot

```cpp
#include "volemachine.h"
// Alu Class Implementation
string Alu::hexToBinary(const string &hex) { string binary = "";
    for (char c: hex) {
        switch (toupper(c)) {
            case '0':
                binary += "0000";
                break;
            case '1':
                binary += "0001";
                break;
            case '2':
                binary += "0010";
                break;
            case '3':
                binary += "0011";
                break;
            case '4':
                binary += "0100";
                break;
            case '5':
                binary += "0101";
                break;
            case '6':
                binary += "0110";
                break;
            case '7':
                binary += "0111";
                break;
            case '8':
                binary += "1000";
                break;
            case '9':
                binary += "1001";
                break;
            case 'A':
                binary += "1010";
                break;
            case 'B':
                binary += "1011";
                break;
            case 'C':
                binary += "1100";
                break;
            case 'D':
                binary += "1101";
                break;
            case 'E':
                binary += "1110";
                break;
```

main Assignments / vole machine / main.cpp Go to file t

MoamenWael04 Add files via upload ee946f9 · 2 days ago History

Code Blame 13 lines (11 loc) · 242 Bytes Code 55% faster with GitHub Copilot Raw

```cpp
#include "volemachine.h"

int main() {
    Machine machine;
    string input;
    cout << "Please enter your command: \n";
    getline(cin, input);
    machine.setInput(input);
    machine.execute();
    machine.printState();

    return 0;
}
```

Code Explanation:

Vole Machine Simulator Report

Overview

The Vole machine is a simple computing architecture that operates on a set of instructions stored in memory. It uses a register file and an arithmetic logic unit (ALU) to execute instructions. This report provides a detailed explanation of the code structure and the algorithms implemented within the Vole machine.

Class Structure

The code consists of several classes, each responsible for a different aspect of the Vole machine:

- 1. Alu: This class handles arithmetic and logic operations.
- 2. Register: This class manages a collection of registers.
- 3. Memory: This class manages memory storage.
- 4. Instructions: This class combines ALU operations and instruction execution logic.
- 5. Machine: This is the main class that manages memory, registers, and instruction execution.

Algorithms and Functions

ALU Functions

- hexToBinary: Converts a hexadecimal string to its binary equivalent.
- binaryToHex: Converts a binary string to its hexadecimal equivalent.
- signExtend: Sign-extends a binary string to a specified length.
- addBinaryTwosComplement: Performs addition of two binary strings using 2's complement.
- hexToDecimal: Converts a hexadecimal string to a decimal integer.

Logical Operations

- ORing: Performs a bitwise OR operation between two binary strings.
- ANDing: Performs a bitwise AND operation, ensuring both strings are the same length before computation.
- XORing: Performs a bitwise XOR operation, again ensuring equal string lengths.

Register Class

- setbit: Sets a register bit at a specific index, ensuring the index is valid.
- getbit: Retrieves a register bit at a specified index, returning '0' for invalid indices.
- printRegister: Displays the current state of all registers.

Memory Class

- setcellinput: Reads input commands, ensuring they are properly formatted and fit within memory bounds.
- setcelladdress: Sets a memory cell at a specific address, checking for valid address ranges.
- getcell: Retrieves the value from a specified memory cell, returning '00' for invalid addresses.
- printMemory: Displays the current state of all memory cells.

Instructions Class

This class contains the casses

This class orchestrates instruction execution:

1. rotateRight: Rotates a binary string to the right by a specified number of steps.
2. readInput: Reads commands from memory and stores them for execution

Explanation of Cases in executeStep

- Case 1: Load from Memory to Register - Load the value from memory at the specified address into the designated register.
- Case 2: Directly Set Value to Register - Set the register directly with the value provided in the command.
- Case 3: Store from Register to Memory - Store the value from the specified register into memory at the given address. If the address is 0, output the value to the screen.
- Case 4: Copy Value Between Registers - Copy the value from one register to another.
- Case 5: Arithmetic Addition using 2's Complement - Add two register values using 2's complement addition, storing the result in the designated register.
- Case 7: Bitwise OR - Perform a bitwise OR operation between two registers and store the result in another register.
- Case 8: Bitwise AND - Perform a bitwise AND operation between two registers and store the result in another register.
- Case 9: Bitwise XOR - Perform a bitwise XOR operation between two registers and store the result in another register.
- Case 10: Rotate Right - Rotate the contents of a specified register to the right by a defined number of steps.
- Case 11: Conditional Branch - If a specific register's value equals the value in register 0, jump to the specified address; otherwise, proceed to the next instruction.
- Case 12: Halt Operation - Stop execution, signaling that no further instructions will be processed.
- Case 13: Conditional Branch - If the value in the specified register (operand) is greater than the value in register 0, jump to the address given by the operand. If not, move to the next instruction.

- Default: Invalid Operation Code - If an invalid opcode is encountered, execution halts, and an error message is displayed.

Machine Class

- setInput: Accepts a string input, populating memory cells with command data.
- execute: Continuously executes instructions until the machine halts or all commands have been processed.
- printState: Displays the current values in memory and registers, facilitating easy monitoring of the machine's state.

Conclusion

The Vole machine simulator is structured to handle various operations systematically. Each class is designed to manage specific tasks, allowing the overall system to function cohesively. The instruction execution process is straightforward, ensuring clarity and efficiency in operations. The design promotes easy expansion and modification for future enhancements or additional features.