**UVSim**

Group C - Alexander Nin, David De Leo, Jess Smith, Zhihui Chen

Table of Contents

[Introduction 4](#_Toc173948119)

[User Stories & Use Cases 5](#_Toc173948120)

[User Stories 5](#_Toc173948121)

[Use Cases 6](#_Toc173948122)

[Functional Specifications 14](#_Toc173948123)

[Functional Requirements 14](#_Toc173948124)

[Non-Functional Requirements 15](#_Toc173948125)

[Class Descriptions 16](#_Toc173948126)

[UVSim Class 16](#_Toc173948127)

[Arithmetic Class 18](#_Toc173948128)

[Control Class 19](#_Toc173948129)

[IO Class 20](#_Toc173948130)

[MainWindow Class 21](#_Toc173948131)

[GUI Wireframes 24](#_Toc173948132)

[Unit Test Descriptions 25](#_Toc173948133)

[Application Instructions 28](#_Toc173948134)

[Preparation 28](#_Toc173948135)

[Load a Program 28](#_Toc173948136)

[Run a Program 28](#_Toc173948137)

[Save a Program 29](#_Toc173948138)

[Modify a Program 29](#_Toc173948139)

[Change Color Scheme 29](#_Toc173948140)

[Switch between Simulators 29](#_Toc173948141)

[Instruction Description 30](#_Toc173948142)

[Future Road Map 32](#_Toc173948143)

# Introduction

The UVSim project delivers a robust virtual machine environment designed to interpret BasicML, a straightforward machine language tailored for educational purposes. This simulator not only mimics a basic computer’s memory architecture with a 250-word memory simulation but also enhances user interaction through a comprehensive graphical user interface (GUI). The UVSim supports a variety of operations including arithmetic calculations, control flow commands, and input/output processes, making it an ideal tool for learning and experimenting with the fundamentals of machine language programming. Integrated features such as multi-file support, color theme customization, and an interactive editor further enrich the user experience, offering both novices and experienced programmers a versatile platform to develop and test machine language programs.

# User Stories & Use Cases

## User Stories

User Story 1:

As a computer science student,

I want to input two numbers, perform arithmetic operations (addition, subtraction, multiplication, division), and see the results,

so that I can understand how basic machine language operations manipulate data and apply these concepts to learn about computer architecture.

User Story 2:

As a computer science student,

I want to input a number, check if it is positive, negative, or zero, and then direct the flow of the program based on these conditions,

So that I can understand how conditional branching works in machine language.

## Use Cases

Use case 1: Adding numbers

Actor: UVSim system

System: Arithmetic unit

Goal: To add two numbers and update the accumulator with the result.

Steps:

1. System identifies the 'add' operation code.
2. System retrieves the number from the specified memory address.
3. System adds the number to the accumulator.
4. Result is stored in the accumulator.

Use case 2: Subtract numbers

Actor: UVSim system

System: Arithmetic unit

Goal: To subtract one number from another and update the accumulator with the result.

Steps:

1. System identifies the 'subtract' operation code.
2. System retrieves the number from the specified memory address.
3. System subtracts the number from the accumulator.
4. Result is stored in the accumulator.

Use case 3: Divide numbers

Actor: UVSim system

System: Arithmetic unit

Goal: To divide one number by another and update the accumulator with the result.

Steps:

1. System identifies the 'divide' operation code.
2. System retrieves the number from the specified memory address.
3. System divides the accumulator by the number.
4. Result is stored in the accumulator.

Use case 4: Multiply numbers

Actor: UVSim system

System: Arithmetic unit

Goal: To multiply two numbers and update the accumulator with the result.

Steps:

1. System identifies the 'multiply' operation code.
2. System retrieves the number from the specified memory address.
3. System multiplies the number by the accumulator.
4. Result is stored in the accumulator.

Use case 5: Read

Actor: User

System: Input/Output handler

Goal: To read input from the user and store it in memory.

Steps:

1. System prompts the user for input.
2. User enters data.
3. System stores the input in the specified memory location.

Use case 6: Write

Actor: UVSim system

System: Input/Output handler

Goal: To display a value from memory to the screen.

Steps:

1. System retrieves a value from a specified memory address.
2. System outputs the value to the screen.

Use case 7: Load

Actor: UVSim system

System: Memory management unit

Goal: To load a word from a specific memory location into the accumulator.

Steps:

1. System identifies the 'load' operation code.
2. System retrieves a word from the specified memory address.
3. System updates the accumulator with the retrieved value.

Use case 8: Store

Actor: UVSim system

System: Memory management unit

Goal: To store the value from the accumulator into a specified memory location.

Steps:

1. System identifies the 'store' operation code.
2. System writes the value of the accumulator to the specified memory address.

Use Case 9: Branch Actor: UVSim system

System: Control unit

Goal: To change the instruction pointer based on a specified condition or unconditionally.

Steps:

1. System identifies the 'branch' operation code.
2. The instruction pointer is updated to the specified memory address.
3. The system continues execution from the new instruction pointer location.

Use case 10: Halt

Actor: UVSim system

System: Control unit

Goal: To halt the virtual machine process and end program execution.

Steps:

1. Load program
2. Find address
3. Halt program process and exits program

Use Case 11: Edit Instruction

Actor: User

System: Instruction Editor

Goal: To modify an existing instruction in the program's memory.

Steps:

1. User selects the instruction to be edited.
2. User modified instruction.
3. System updates the instruction in the memory.

Use Case 12: Save Edited Instructions

Actor: User

System: File Management System

Goal: To save the current set of instructions in the memory to a new file.

Steps:

1. User edits the instructions.
2. User initiates the Save As operation.
3. System prompts the user to enter a filename for the new file.
4. User enters the filename and confirms the save operation.
5. System writes the instructions from memory to the new file.

Use Case 13: Change Color Theme

Actor: User

System: User Interface

Goal: To change the color theme of the program's GUI.

Steps:

1. User clicks the Change Color Theme option in the menu.
2. System displays the color palette for the user to choose a new color theme.
3. User selects a desired color theme.
4. System applies the new color theme to the GUI.

Use Case 14: Clear Output

Actor: User

System: User Interface

Goal: To clear the previous output in console.

Steps:

1. User clicks the Clear Output option.
2. System clears all text and data previously displayed in the console.

Use Case 15: Support GUI

Actor: User

System: Graphical User Interface

Goal: To interact with the program through a GUI.

Steps:

1. User launches the program, which starts with the GUI.
2. User interacts with the GUI elements such as buttons and input fields to perform operations (e.g., load, edit, save instructions).
3. System responds to each interaction by updating the GUI accordingly (e.g., displaying data, updating status).
4. User closes the GUI to end the session.

Use Case 16: Manage Multiple Files Simultaneously

Actor: User

System: User Interface and File Management System

Goal: To open, edit, and manage multiple files simultaneously within a single application instance.

Steps:

1. User opens multiple files using the menu.
2. System displays each file in a separate tab within the main application window.
3. User switches between tabs or windows to edit different files.
4. System keeps track of changes in each file independently.
5. User executes or saves files individually.

Use Case 17: Support Different Format

Actor: System

System: File and Memory Management System

Goal: To handle data files and internal memory with a new six-digit word format.

Steps:

1. System detects the format of the file being loaded (four or six digits) based on file metadata or initial inspection.
2. System processes file operations, ensuring all memory addresses and operations codes are valid and conform to the six-digit requirement.
3. System checks for overflow conditions and handles them as per the six-digit operation limits.

# Functional Specifications

## Functional Requirements

1. The system shall verify that input file has correct format.
2. The system shall be able to skip invalid or unused memory addresses.
3. The system shall verify that every command that will be run is a valid instruction.
4. The system shall be able to allow the user to manually alter the value of the accumulator and memory.
5. The system can allow for stepping through the program instructions.
6. The system shall display the address of the currently running memory.
7. The system shall verify that the user input has the correct format.
8. The system shall have 3 accumulators for system use.
9. The system shall be able to allow the user to reset the system.
10. Programs (saved in txt files) can be created and loaded through the application.
11. The system shall keep records of recently used files.
12. The system shall have an instruction that can convert a char input to a number in memory.
13. The system shall have an instruction that can convert a number in memory to a char output.
14. The system shall have an instruction that can print a null terminated string from memory using ascii values.
15. The system shall be able to allow the user to run the program by clicking a button.
16. When the system processes a 4- digit instruction, it should convert it to a 6-digit number.
17. The system shall ignore excess digits when processing instructions with more than 6 digits.
18. The system shall allow users to run multiple simulators simultaneously.
19. When a user runs multiple simulators at the same time, the system should be able to save their data separately.
20. The system shall and must have 250-word available memory.
21. The system shall prohibit the user from entering more than 250-word memory.
22. The system shall allow the user to modify commands directly in the interface.
23. The background color of the GUI shall be #4C721D.
24. The text color of the GUI shall be #FFFFFF.
25. The system shall allow the user to modify the color of the GUI.

## Non-Functional Requirements

1. Loading of locally-stored programs should be snappy (less than 2 seconds).
2. The system shall have to display the description, when the user hovers the mouse over the button.
3. The system shall be able to warn the user when the user needs to enter a number or has performed an incorrect operation.

# Class Descriptions

## UVSim Class

The core part of UVSim, which includes the most critical functions such as executing instructions and managing memory

Methods:

**loadProgram**: This method is responsible for loading a program from a specified file into the simulator’s memory. It parses and stores instructions from the file specified by the filename parameter. Once the program is loaded, it is stored in the simulator’s memory, ready for execution.

**getMemory**: This method retrieves the current state of the simulator’s memory, primarily for testing and debugging purposes. It returns a copy of the memory vector, allowing external components to inspect or verify the memory contents without altering the simulator’s state.

**setMemory**: This method sets a memory cell at a specified index to a new value. It is useful for testing and initialization. The index parameter specifies the memory cell to modify, and the value parameter is the new value to set. This method ensures the memory at the specified index is updated with the new value.

**clearMemory**: This method clears the simulator’s memory by setting all cells to zero. It ensures that the entire memory vector is reset, making it useful for resetting the simulator’s state between program executions or during initialization.

**getAccumulator**: This method accesses the current value of the simulator’s accumulator. It returns the accumulator’s current value, allowing external components to inspect or use it for calculations or conditions.

**setAccumulator**: This method sets the accumulator to a specified value. The num parameter is the new value for the accumulator. This is particularly useful for initialization or during testing phases to ensure the accumulator starts with a predetermined value.

**isHalted**: This method checks if the simulator’s execution has been halted. It returns a boolean value, true if the execution is halted and false otherwise, providing a simple way to check the run state of the simulator.

**setHalted**: This method sets the halted state of the simulator. The halt parameter is a boolean value that determines whether the simulator should continue execution or stop. This method is crucial for controlling the flow of program execution within the simulator.

**getInstructionPointer**: This method retrieves the current position of the instruction pointer, which indicates the next instruction to execute. It returns the current value of the instruction pointer, essential for debugging and execution flow control.

**setInstructionPointer**: This method sets the instruction pointer to a specified value. The ip parameter is the new value for the instruction pointer. This is useful for testing or when the program requires jumping to a different part of the memory during execution.

**run**: This method initiates the simulation by executing loaded instructions until the program is halted. It requires that a program is already loaded into memory and executes it until completion or until a halt condition is met.

**dump**: This method outputs the current state of the memory and registers for debugging purposes. It prints the state of memory and registers to the output, providing a snapshot of the simulator’s internals at any given point.

**load**: This method loads the content of a specific memory location into the accumulator. The operand parameter specifies the memory location from which to read. This updates the accumulator with the value found at the specified memory location.

**store**: This method stores a value into a specified memory location. The index parameter specifies where to store the value, and the word parameter is the value to be stored. This ensures the memory at the specified index is updated with the new value.

**fetch**: This method fetches the instruction from a specific memory location to be executed. The index parameter specifies the memory location from which to fetch. It returns the value of the fetched instruction, crucial for the execution phase of the simulation.

**execute**: This method decodes and executes the fetched instruction. The instruction parameter is the instruction code to execute. This method requires that the instruction be a valid opcode and carries out the operation specified by the instruction, affecting the state of the simulator accordingly.

## Arithmetic Class

Manages arithmetic operations within the UVSim, processing calculations and updates to the accumulator based on instruction codes.

Methods:

**add**: This method handles the addition of a specified value to the accumulator. The acc parameter represents the current value of the accumulator, and the value parameter is the value to be added. The method returns the updated accumulator value, reflecting the sum of the accumulator and the input value. This update is crucial for arithmetic operations within the simulator.

**subtract**: This method subtracts a given value from the accumulator. The acc parameter holds the current value of the accumulator, and the value parameter is the value to be subtracted. It returns the updated accumulator value, which is the difference between the accumulator and the input value. This method ensures that arithmetic operations involving subtraction are correctly handled within the simulator.

**multiply**: This method multiplies the accumulator by a given value. The acc parameter is the current value of the accumulator, and the value parameter is the multiplier. The method returns the updated accumulator value, reflecting the product of the accumulator and the multiplier. This operation is essential for performing multiplication within the simulator.

**divide**: This method divides the accumulator by a specified value. The acc parameter represents the current value of the accumulator, and the value parameter is the divisor. The method returns the updated accumulator value after division, ensuring the accumulator is updated to the quotient. This method is critical for handling division operations and includes a pre-condition that the divisor should not be zero to avoid a division by zero error.

**truncate**: This method truncates a number to four digits, which is used to keep accumulator values within operational limits. The num parameter is the number to truncate. It returns a truncated version of the number, ensuring that values exceeding the operational limits are appropriately scaled down to fit within the simulator’s capabilities.

## Control Class

Handles control flow operations such as branching based on the state of the accumulator and the instruction pointer.

Methods:

**branch**: This method directly sets the instruction pointer to a specified memory location. The operand parameter indicates the memory location to which the instruction pointer should be updated. This method is fundamental for controlling the flow of execution by allowing jumps to different parts of the program based on unconditional branch instructions.

**branchNeg**: This method allows the instruction pointer to branch to a specified location if the accumulator is negative. The operand parameter indicates where to branch, and instructionPointer and accumulator are used to determine if the branch should occur. If the accumulator is negative, the instruction pointer is updated to the operand location, facilitating conditional branching based on the state of the accumulator.

**branchZero**: This method branches to a specified location if the accumulator is zero. It uses the operand parameter to determine the branch location, and instructionPointer and accumulator to decide if the branch should occur. If the accumulator is zero, the instruction pointer is updated to the operand, supporting conditional control flow based on accumulator values.

## IO Class

Manages all input and output operations for the UVSim.

Methods:

**read**: This method manages the input operation, where it reads an integer from the user via the keyboard and stores it in a specified memory location within the simulator. The operand parameter is the memory location where the read value will be stored. This function returns the integer input by the user and ensures that the simulator’s memory is updated accordingly, allowing for user interaction during simulation runs.

**write**: This method is responsible for the output operation. It outputs a value from a specified memory location to the screen, allowing users to see the result of computations and memory manipulations. The operand parameter specifies the memory location from which to read the value, and the method outputs this value to the screen. This ensures that the simulator can communicate results and states back to the user effectively.

**promptFile**: This method prompts the user to enter a file name for the purpose of loading a program into the simulator. It returns the name of the file input by the user, facilitating the loading of different programs for simulation based on user input. This method is critical for initializing the simulation with user-specified programs.

## MainWindow Class

Provides a GUI for the UVSim, allowing users to interact with the simulator.

Methods:

**onButton1Clicked**: This method acts as an event handler for the “Load File” button within the GUI. It triggers the loading of a file into the simulator, ensuring that user-selected programs can be loaded and prepared for execution.

**onButton2Clicked**: This method is linked to the “Execute File” button. It handles the execution of the already loaded program, ensuring that the simulation starts and runs as expected. This method is crucial for transitioning from program loading to program execution.

**onButton3Clicked**: This method corresponds to the “Save As” button. It saves the current set of instructions from the simulator to a file, allowing users to export their programs or the current state of the simulator for later use or analysis.

**onButton4Clicked**: This event handler is associated with the “Clear Output” button. It clears the output area of the GUI, making it useful for resetting the display between simulation runs or when clearing previous results for clarity.

**setDefaultColors**: This method sets the default colors for the GUI components. It ensures that the initial visual state of the GUI is consistent and conforms to a standard aesthetic, enhancing usability and the visual experience of the user.

**changeColors**: This method changes the colors of the GUI components based on user choice. It allows for a customizable user interface where individuals can adjust the visual aspects of the GUI to their preferences, enhancing user interaction and satisfaction.

**onTextViewerTextChanged**: This event handler limits the number of lines that can be entered into the text viewer component of the GUI, ensuring that the input does not exceed predefined limits and maintains usability and performance standards.

**loadTextFile**: This method loads the content of a text file into the text viewer. The filename parameter specifies which file to load, facilitating the display of text file contents directly within the GUI.

**saveTextFile**: This method saves the content currently displayed in the text viewer to a file. The filename parameter specifies where to save the text, making it possible for users to export the contents of the text viewer for external use or storage.

**setTextFileTitle**: This method sets the title of the program within the GUI. It updates the display to reflect the current program or document title, helping users identify which files or programs are currently loaded or being edited.

**run**: This method initiates the simulation, executing loaded instructions until halted. It ensures that the program loaded into memory is executed, allowing users to observe and interact with the simulation process.

**getUserInput**: This method retrieves user input from the input field in the GUI. It collects input provided by the user, making it available for processing or interaction within the simulation.

**createTab**: This method adds a new tab to the GUI. It facilitates the organization of multiple documents or views within the simulator's interface, enhancing the user experience by providing a way to manage multiple tasks or views simultaneously.

**onTabChanged**: This event handler manages the changing of tabs in the GUI. It ensures that the display updates to reflect the content of the newly selected tab, maintaining the relevance and accuracy of the displayed information as the user navigates through different sections of the interface.

# GUI Wireframes

A screenshot of a computer

Description automatically generated

# Unit Test Descriptions

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Name | Reference | Description | Inputs | Expected Outputs | PASSED? |
| Set Memory | TestSetMemory | Test if memory can be set | register 0 = 1001 | register 0 = 1001 | PASS |
| Load Program | TestLoadProgram | Test if the file can be loaded successfully | test1.txt | register 0 = 1007 ... register 7 = 0000 | PASS |
| Read File Fail | TestLoadProgramFail | Test if a non-existing file reports an error | FAKE\_FILE.txt | error message | PASS |
| Accumulator | TestAccumulator | Test whether the accumulator can be set to a value | accumulator = 1002 | accumulator = 1002 | PASS |
| Halt | TestHalt | Test to see if halt works | N/A | halt program | PASS |
| Instruction Pointer | TestInstructionPointer | Test to see if the instruction pointer's value can be obtained | N/A | instructionPointer = 0 | PASS |
| Instruction Pointer Increment | TestInstructionPointerIncrement | Tests whether the value of the instruction pointer can be modified | instructionPointer = 1 | instructionPointer = 1 | PASS |
| Read | TestRead | Test that the input values are stored correctly | input = 7 | memory 10 = 7 | PASS |
| Read Fail | TestReadFail | Test to see if an error can be thrown if the input value is abnormal | input = a | error message | PASS |
| Write | TestWrite | Test to see if the value in memory can be read correctly | N/A | memory 10 = 9 | PASS |
| Load | TestLoad | Tests if the value can be read and added to the accumulator | N/A | accumulator = 9 | PASS |
| Store | TestStore | Test to see if the value in the accumulator can be saved to memory | N/A | memory 10 = 9 | PASS |
| Branch | TestBranch | Test to see if function can jump to the correct address | N/A | accumulator = 0 | PASS |
| Add | TestAdd | Test that the addition works and gives the correct answer | memory 10 = 9 | accumulator = 9 | PASS |
| Subtract | TestSubtract | Test that subtraction works and gives the correct answer | memory 10 = 9 | accumulator = -9 | PASS |
| Multiply | TestMultiply | Test that the multiplication works and gives the correct answer | memory 10 = 9; accumulator = 10 | accumulator = 90 | PASS |
| Divide | TestDivide | Test that the division works and gives the correct answer | memory 10 = 5; accumulator = 10 | accumulator = 2 | PASS |
| Branch Negative | TestBranchNeg | Tests if the function can be executed when the accumulator is negative | accumulator = -10 | accumulator = -10 | PASS |
| Branch Zero | TestBranchZero | Tests if the function can be executed when the accumulator is 0 | accumulator = 0 | accumulator = 0 | PASS |
| Divide Fail | TestDivideFail | Tests whether an error can be thrown when calculating division if the value is illegal | memory 10 = 0; accumulator = 0 | error message | PASS |

# Application Instructions

## Preparation

To compile and run UVSim, you will need a C++ compiler (such as g++ or clang) and QT6 (version 6.7.1 or later). Begin by cloning the repository or downloading the source code from the designated URL. You can use the following commands in your terminal to clone the repository and navigate into the project directory:

*git clone https://github.com/DavidADeLeo/CS2450*

*cd CS2450*

Once you have the source code, open QT Creator and load the CMakeLists.txt file from the project directory. To compile and execute the program, simply click on the green play button in QT Creator. This will build the UVSim application and run it, allowing you to interact with the simulator through its graphical user interface.

## Load a Program

To load a program into UVSim, begin by opening the File Menu and clicking on the Load File button located in the top left-hand corner of the interface. When prompted, navigate to and select the *.txt* file you wish to load and confirm your choice by hitting Enter. This action loads the selected program into the UVSim program for execution.

## Run a Program

To run a program, make sure you load or write a program. Then, click the Execute File button found within the UVSim interface. As the program runs, you can monitor the execution steps and instructions in the Instruments panel on the right-hand side of the GUI. Additionally, keep an eye on the Output window to follow prompts and see the results of the program as it processes.

## Save a Program

For saving a program, use the Save As button. When prompted, enter a name for your file and hit Enter to save the current state of the program to a file. This feature is essential for preserving your work or making backups of current settings and instructions.

## Modify a Program

To modify instructions within UVSim, edit the instructions directly in the Editor text area. Ensure that all changes conform to valid BasicML instructions. After making modifications, you can either save these changes using the Save As button or directly run the modified program by clicking the Execute File button. It is possible to execute changes without saving them first, which is useful for testing purposes.

## Change Color Scheme

If you wish to change the color scheme of the UVSim interface, click on the Change Colors button to open a color picker dialogue. Here, you can select your preferred background and text colors. Confirm your selections by hitting Enter to apply the new color scheme, enhancing the visual experience according to your preferences.

## Switch between Simulators

For changing between different simulators, simply click on the simulator number to toggle among available simulators. Each simulator can have distinct instructions and outputs, which can be viewed and edited independently. Run programs on any selected simulator by clicking the Execute File button. Changes to the simulator or its instructions can be saved using the Save As button. If needed, you can clear the output from a previous session by clicking the Clear Output button, which resets the output area for fresh execution results.

## Instruction Description

**READ (opcode 10)**: This instruction prompts the user to input an integer, which is then stored at the memory location specified by the operand.

**WRITE (opcode 11)**: This instruction outputs the value stored at the memory location specified by the operand.

**LOAD (opcode 20)**: The LOAD instruction transfers the content of a specific memory location (specified by the operand) into the accumulator.

**STORE (opcode 21)**: This instruction saves the value currently held in the accumulator into the memory location designated by the operand.

**ADD (opcode 30)**: The ADD instruction adds the value from a specified memory location (fetched through the operand) to the accumulator’s current value. The result of this addition is then stored back in the accumulator, updating it with the new sum.

**SUBTRACT (opcode 31)**: This operation subtracts the value at the memory location indicated by the operand from the accumulator’s current value. The result is stored back in the accumulator, effectively updating it with the new difference.

**DIVIDE (opcode 32)**: The DIVIDE instruction divides the accumulator’s current value by the value at the specified memory location. It updates the accumulator with the quotient of this division, handling division operations within the program.

**MULTIPLY (opcode 33)**: This instruction multiplies the accumulator’s current value by the value from the specified memory location. The product is then stored back in the accumulator, updating it with the result of the multiplication.

**BRANCH (opcode 40)**: The BRANCH instruction directly sets the instruction pointer to the memory location specified by the operand. This operation enables unconditional jumps within the program.

**BRANCHNEG (opcode 41)**: This instruction checks if the accumulator’s value is negative. If true, it sets the instruction pointer to the memory location specified by the operand, facilitating conditional branching based on negative results.

**BRANCHZERO (opcode 42)**: Similar to BRANCHNEG, the BRANCHZERO instruction checks if the accumulator’s value is zero. If it is, the instruction pointer is updated to the location specified by the operand, enabling conditional jumps based on zero results.

**HALT (opcode 43)**: The HALT instruction stops the program execution by setting the halted flag to true. This is a critical control operation that allows the program to terminate cleanly upon completion or when a stop condition is encountered.

**Error Handling (default case)**: If an unknown opcode is encountered, an error message is logged, and the program halts execution to prevent further erroneous operations.

# Future Road Map

In the future, we may add the following features to UVSim:

1. The instruction supports running line by line.
2. If running line by line, highlight the currently running line
3. Add Debug function.
4. Add highlighting function to instructions.
5. The total line of instructions is no longer limited.
6. Have a richer instruction set

A screenshot of a computer

Description automatically generated