**UVSim**

Spencer Rohwer

Brinley Crowther

Agustin Suarez Berois

Owen Rasor

# **Table of Contents**

**Introduction 2**

**Executive Summary 3**

**User Stories/Use Cases 4**

**Functional Specifications 9**

**Class Diagrams 11**

**GUI Wireframes 12**

**GUI Design Explained 13**

**Unit Test Descriptions 14**

**User Manual 16**

**Launching the App 16**

**Loading the Program 16**

**Running the Program 17**

**Console Input 17**

**Edit 18**

**Saving 18**

**Reset 19**

**Customizing the Interface 19**

**Running Multiple Programs 20**

**Example Workflow 21**

**Future Roadmap 22**

# Introduction

# Throughout the course of this semester, our team of four software engineering students collaborated on a comprehensive group project aimed at simulating a real-world software development experience. The objective of the project was to design, build, and iteratively improve a software system from initial concept to final implementation. Under the direction of an educational client, we developed *UVSim*—a virtual machine simulator designed to help computer science students better understand machine language and computer architecture.

# This project followed the complete software development lifecycle, progressing through key phases including requirements gathering, design documentation, prototype development, GUI implementation, testing, and iterative revisions. Each milestone introduced new challenges and expanding feature sets, simulating the evolution of client demands in real-world software engineering environments. Weekly team meetings, rigorous documentation, and collaborative coding were essential components of our workflow, ensuring all team members remained engaged and contributed meaningfully to the final deliverable.

# 

# Executive Summary

The UVSim project originated from a need expressed by an educational client to provide computer science students with an interactive and intuitive platform to learn machine-level programming. Our goal was to deliver a fully functional virtual machine capable of interpreting a simplified instruction set called BasicML, while offering students a hands-on experience manipulating memory, registers, and control flow. Early stages focused on developing a working command-line prototype capable of reading and executing BasicML programs. As the project progressed, we incorporated unit testing to ensure each function's reliability and correctness, as well as supporting documentation for usability and maintainability.

In response to evolving client requirements, UVSim underwent substantial enhancements. A graphical user interface was developed to improve usability, followed by the introduction of color customization features, file editing tools, and support for extended memory sizes and modernized six-digit word formats. The system now supports simultaneous editing and execution of multiple program files, bridging old and new formats seamlessly. Each milestone added complexity and demanded architectural adaptability, which our team addressed through careful design, modular class structures, and consistent feedback-driven iteration. UVSim is now a robust, user-friendly educational tool, and the final deliverable reflects the team’s collaborative effort, technical competency, and ability to meet changing stakeholder needs.

# User Stories/Use Cases

# User Stories

**Student:** As a student, I want to load and execute a BasicML program in the simulator so I can learn and understand how machine language instructions work.

**Instructor:** As an instructor, I want to track errors and executions so I can explain the behavior of a program and debug for my students.

# Use Cases

**UC1: Read word into memory**

**Actor:** User

**System:** UVSim

**Goal:** Store a four-digit number at a specified memory location

**Steps:**

1. User inputs a signed four-digit number

2. System validates the number

3. System checks if memory location specified is free

4. If valid, system stores the number at the memory location

**UC2: Write word to screen**

**Actor:** User

**System:** UVSim

**Goal:** Retrieve and display a word from a specified memory location

**Steps:**

1. User specifies a memory location

2. System validates the memory location

3. System retrieves data from the location

4. System displays the retrieved data

**UC3: Perform Arithmetic operation**

**Actor:** User

**System:** UVSim

**Goal:** Execute arithmetic operations using the accumulator

**Steps:**

1. System fetches the operation code

2. System retrieves the operand from the memory address

3. System validates the memory address and checks if occupied

4. System retrieves the word from the memory address

5. System performs arithmetic operation using the accumulator and the retrieve word

6. System updates the accumulator with the result

**UC4: Execute Branch Instruction**

**Actor:** User

**System:** UVSim

**Goal:** Update the program counter based on a branch instruction

**Steps:**

1. System fetches the branch instruction and operand

2. System validates the memory address

3. System updates the program counter based on the instruction

**UC5: Handle Errors**

**Actor:** User

**System:** UVSim

**Goal:** Detect and handle errors appropriately

**Steps:**

1. System detects an error during execution

2. System displays an error message

3. System halts execution or prompts user for input correction

**UC6: Dump Memory**

**Actor:** User

**System:** UVSim

**Goal:** Display all memory contents for inspection

**Steps:**

1. User issues a command to dump memory

2. System displays memory values in rows with their address

**UC7: Inspect Current Instruction**

**Actor:** User

**System:** UVSim

**Goal:** Display the current instruction, accumulator, and program counter

**Steps:**

1. User issues a command to inspect current state

2. System retrieves and displays:

● Decoded operation and operand of current instruction

● Current value of the accumulator

● Current program counter value

**UC8: Load program**

**Actor:** User

**System:** UVSim

**Goal:** Load a program from a text file into memory

**Steps:**

1. User specifies the input file containing BasicML program

2. System validates the file format

3. System loads the instructions into memory address

**UC9: Execute program**

**Actor:** User

**System:** UVSim

**Goal:** Execute the BasicML program sequentially

**Steps:**

1. System sets the program counter to 00

2. System fetches the instruction at current program counter

3. System decodes and executes the instruction

4. System updates the program counter

5. Steps repeat until HALT instruction is reached or error occurs

**UC10: Reset program**

**Actor:** User

**System:** UVSim

**Goal:** Reset simulator state

**Steps:**

1. User issues a reset command

2. System clears all memory contents

3. System resets the accumulator to zero

4. System resets the program counter to 00

**UC11: Save program**

**Actor:** User

**System:** UVSim

**Goal:** Save the current program state

**Steps:**

1. User issues a save command

2. System writes memory, accumulator, and program counter values to a file

**UC12: Log execution**

**Actor:** User

**System:** UVSim

**Goal:** Maintain an execution log for debugging and analysis

**Steps:**

1. System maintains a log during program execution

2. For each instruction, system records:

● Executed instruction and address

● Updated state of the accumulator and memory

3. System displays or saves the log upon program termination

# Functional Specifications

**Functional Requirements:**

UVSim shall:

1. Execute Basic Machine Language (BML) instructions sequentially.

2. Accept a properly formatted text file as user input and correctly parse and load BML programs.

3. Provide a text area in the GUI where users can input and edit program instructions.

4. Store and display memory addresses and values in a scrollable table, allowing users to inspect and modify memory contents.

5. Implement an Accumulator register to store intermediate values and perform arithmetic operations, updating the display accordingly.

6. Support conditional and unconditional branching instructions based on the Accumulator’s value and update execution flow accordingly.

7. Execute programs when the “Execute” button is pressed, running all instructions until a halt instruction is reached or an error occurs.

8. Provide a “Step” button to execute a single instruction at a time, updating the GUI with each step.

9. Detect and handle errors such as invalid instructions, memory overflows, and division by zero, displaying clear error messages.

10. Enable users to save programs as text files and reload them without modification or data loss.

11. Allow users to manually halt execution at any point via a dedicated “Quit” button.

12. Display the full memory state in an accessible format for user inspection, refreshing after every executed instruction.

13. Show the current Accumulator value dynamically, ensuring real-time updates during execution.

14. Maintain an execution log that records every executed instruction, including memory changes, for debugging purposes.

15. Accept user input when required by specific instructions, processing the input correctly and updating relevant components.

16. Switch between files, executing multiple files simultaneously.

17. Change the color scheme to any color desired via hex number.

18. Manage up to 250 lines in memory.

19. Convert BML programs of 4-digit words to 6-digit words.

20. Load files from any specified folder.

21. Edit files from original input file.

**Non-Functional Requirements**

1. The system shall provide a GUI designed using Kivy, ensuring clear navigation, responsive design, and accessibility.

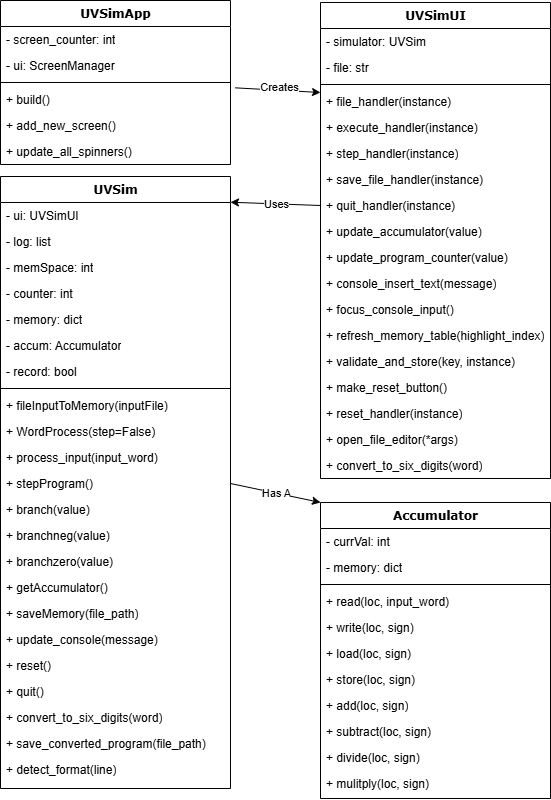
2. The system shall execute BML programs with an average response time of less than 500 milliseconds per instruction.

3. The system shall handle invalid inputs without crashing, ensuring error messages guide the user toward corrective actions.

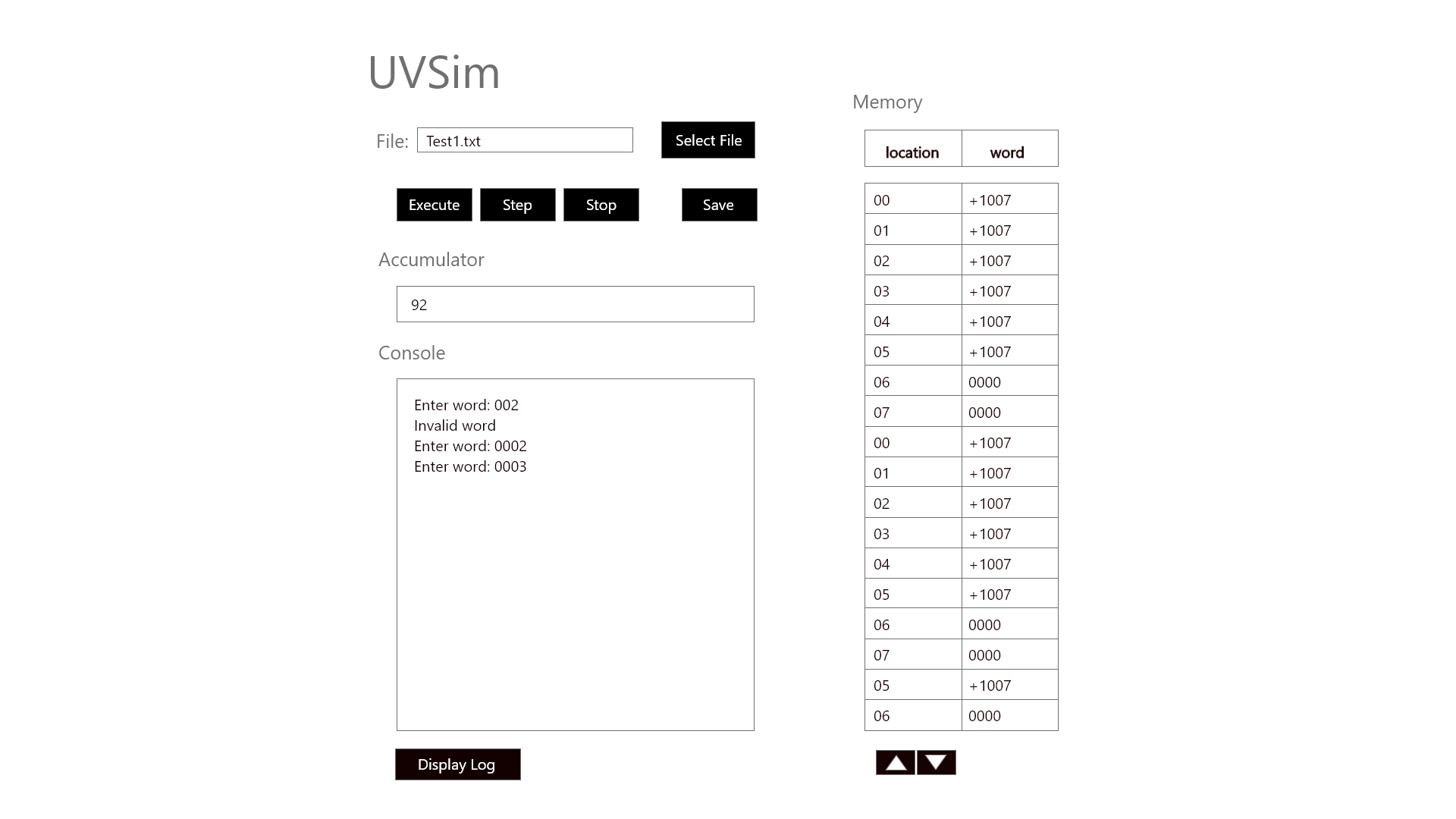
4. The system shall handle multiple files, executing each file in its own tab.

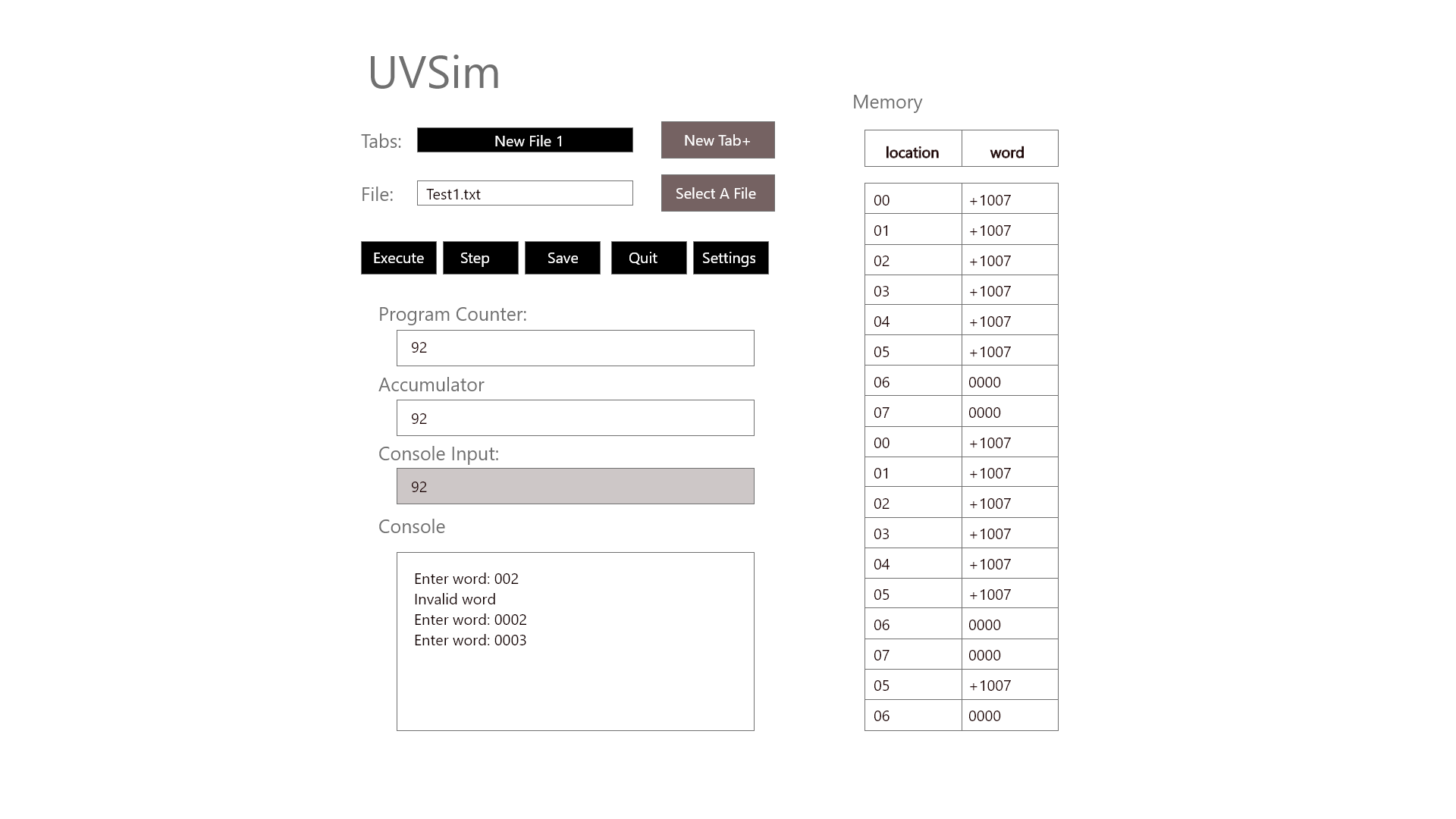
# 

# Class Diagrams



# GUI Wireframes

Original Wireframe:

Updated Wireframe:

## **GUI Design Explained**

**Left Panel – Controls and Display**

* **Tabs Dropdown (New File 1)**: Selects the currently active simulation screen.
* **New Tab+ Button**: Creates a new simulation tab/screen.
* **File Text Box (Test1.txt)**: Displays the name of the currently loaded file.
* **Select a File Button**: Opens a file chooser popup to load a .txt file into memory.
* **Execute Button**: Runs the entire loaded program until a halt instruction.
* **Step Button**: Executes one instruction at a time for debugging or controlled flow.
* **Save Button**: Opens a file save dialog to export the current memory state.
* **Quit Button**: Stops the simulation and exits back to the launcher (or app exit).
* **Settings Button**: Opens the color scheme configuration popup.
* **Program Counter Label & Box**: Displays the address of the instruction currently being executed.
* **Accumulator Label & Box**: Shows the current value in the accumulator register.
* **Console Input Label & Box**: Accepts user input when the program requests a word (e.g., via READ).
* **Console Output Section**: Displays logs, prompts, and program messages like input/output events and errors.

**Right Panel – Memory Table**

* **Memory Label**: Section header for memory visualization.
* **Table Header (location, word)**: Column labels for memory address and its corresponding value.
* **Memory Grid (e.g., 00, +1007)**: Editable memory slots showing address-value pairs; updated during program execution.

# Unit Test Descriptions

**test\_uvsim.py**

test\_default\_values: Verifies that UVSim initializes with default values: counter = 0, memSpace = 0, memory has 100 entries, and an Accumulator is instantiated.

test\_branch: Ensures that the branch instruction sets the program counter correctly and logs the jump.

test\_branchneg: Tests conditional branching to a memory location only if the accumulator’s value is negative.

test\_branchzero: Tests conditional branching to a memory location only if the accumulator’s value is zero.

test\_inspectCurrent: Validates that inspectCurrent accurately reflects the program's current state, including accumulator value and the latest action.

test\_saveMemory: Confirms that saveMemory writes the correct memory and accumulator state to a file and that the file contains expected content.

**test \_accumulator.py**

test\_default\_values: Verifies that the Accumulator initializes with currVal set to 0 and a memory of 100 entries initialized to "0000".

test\_read: Ensures that valid input strings (e.g., "1000", "+2000", "-3000") are properly formatted, stored in memory, and returned.

test\_write: Confirms that the write method retrieves and prints the correct value from memory to the console and returns it.

test\_load: Checks that the load method sets currVal to the correct integer value retrieved from memory.

test\_store: Verifies that the store method formats currVal correctly and writes it to the specified memory location.

test\_add: Tests whether the add method correctly adds memory values to the accumulator's currVal.

test\_subtract: Validates that the subtract method subtracts memory values from the accumulator’s currVal.

test\_divide: Ensures that the divide method performs correct integer division on currVal using the value in memory.

test\_multiply: Confirms that the multiply method updates currVal by multiplying it with values from memory.

# User Manual

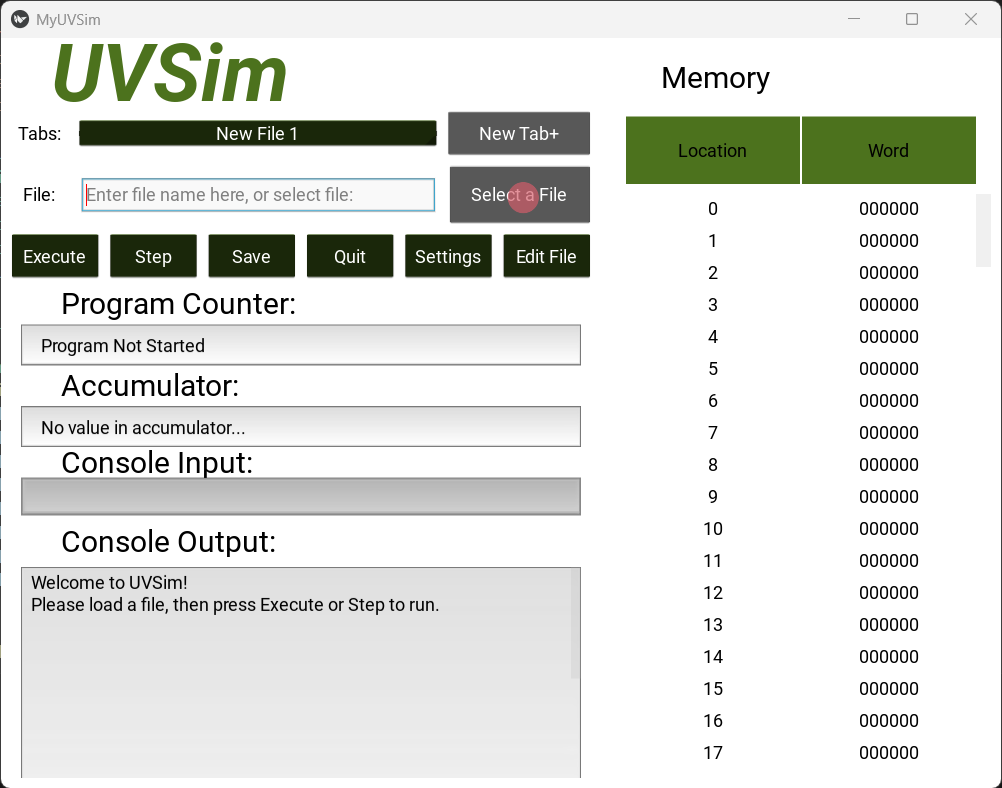
**Launching the App**

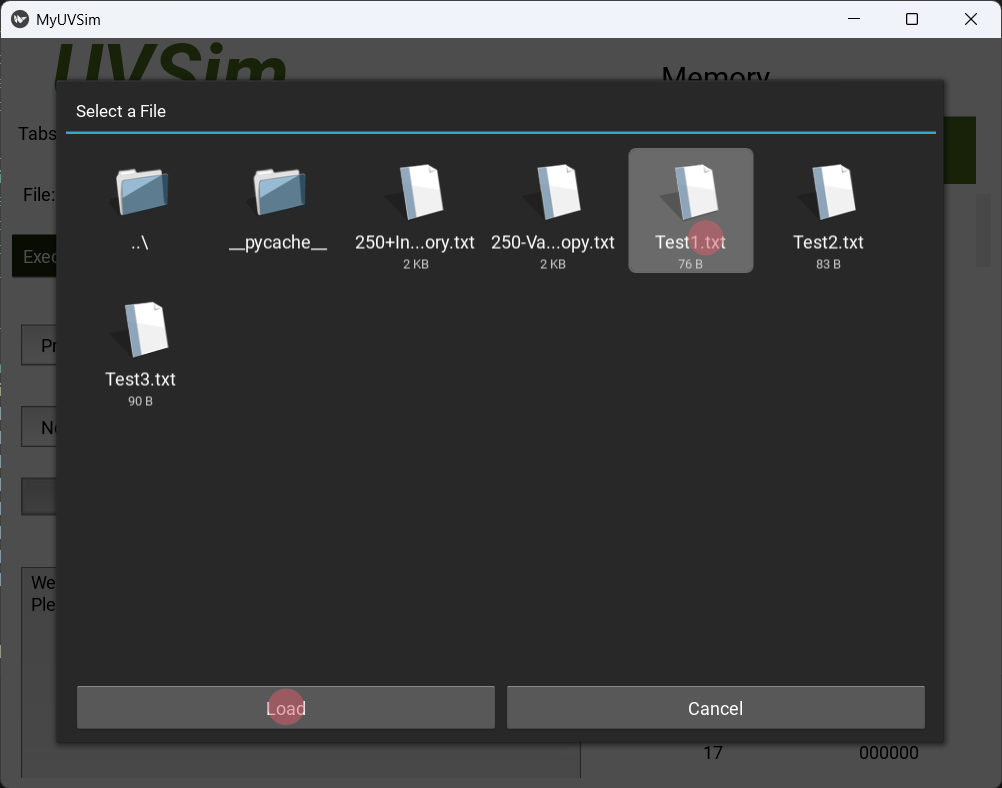
In the terminal, navigate to the folder that contains the necessary files.

Run:

**Loading the Program**

Option A: Type the file path in the **File:** field and press **Enter** on the keyboard.

Option B: Click “**Select a File”** to browse for .txt files.

In the directory window, highlight the desired file and click “**Load**”.

*Only .txt files*

*with 4 or 6-digit*

*BasicML words*

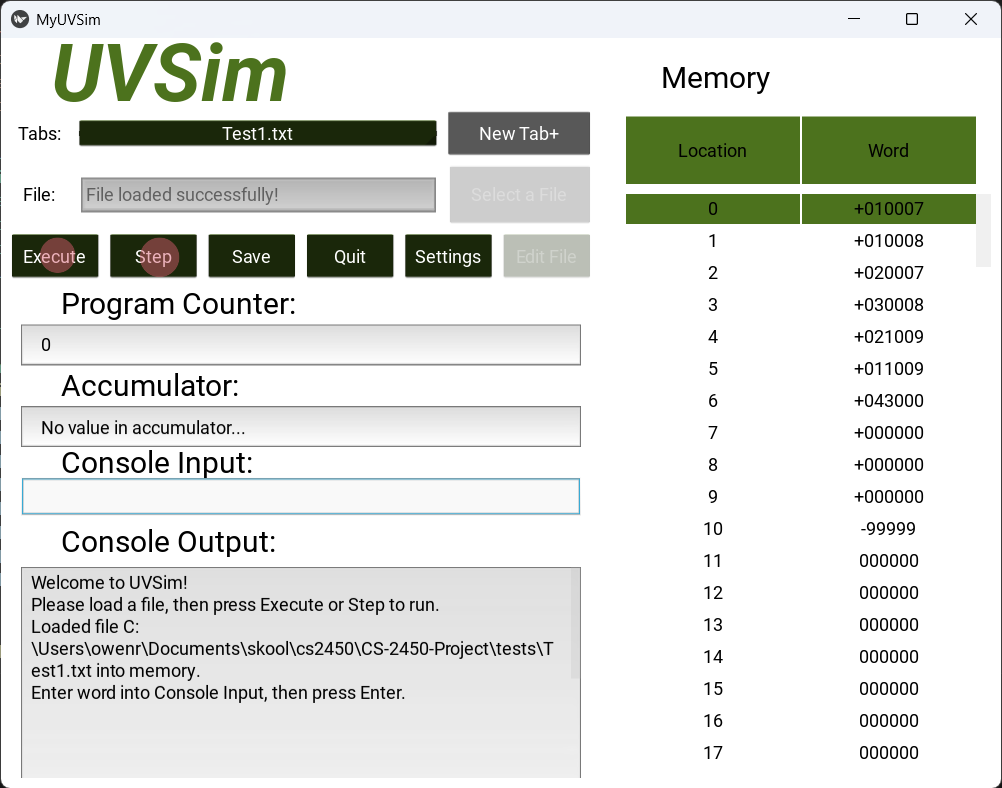
*are supported.*

*Max of 250*

*lines is allowed.*

**Running the Program**

Click “**Execute**” to run the entire program.

Click “**Step**” to execute one instruction at a time. You may hit “**Execute**” at any step to run the rest of the program.

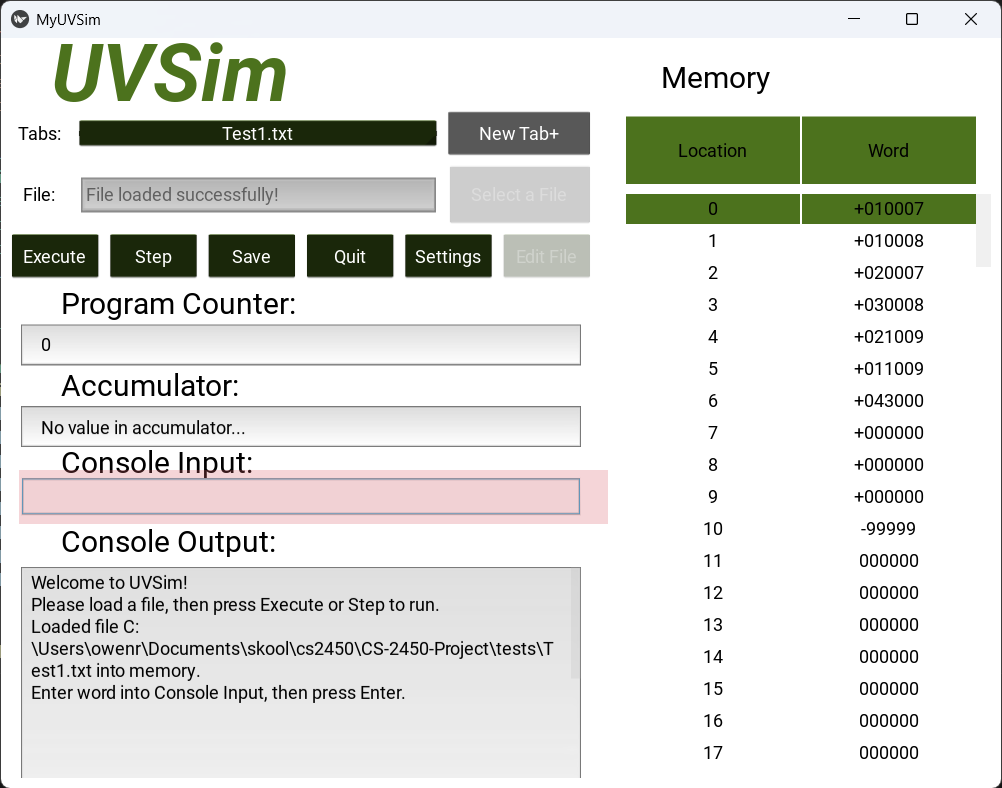
While stepping through, the current location in memory will be highlighted on the *Memory* table.

All program instructions and outputs will be displayed in the *Console Output* field.

The *Program Counter* and *Accumulator* fields will also be updated with each step.

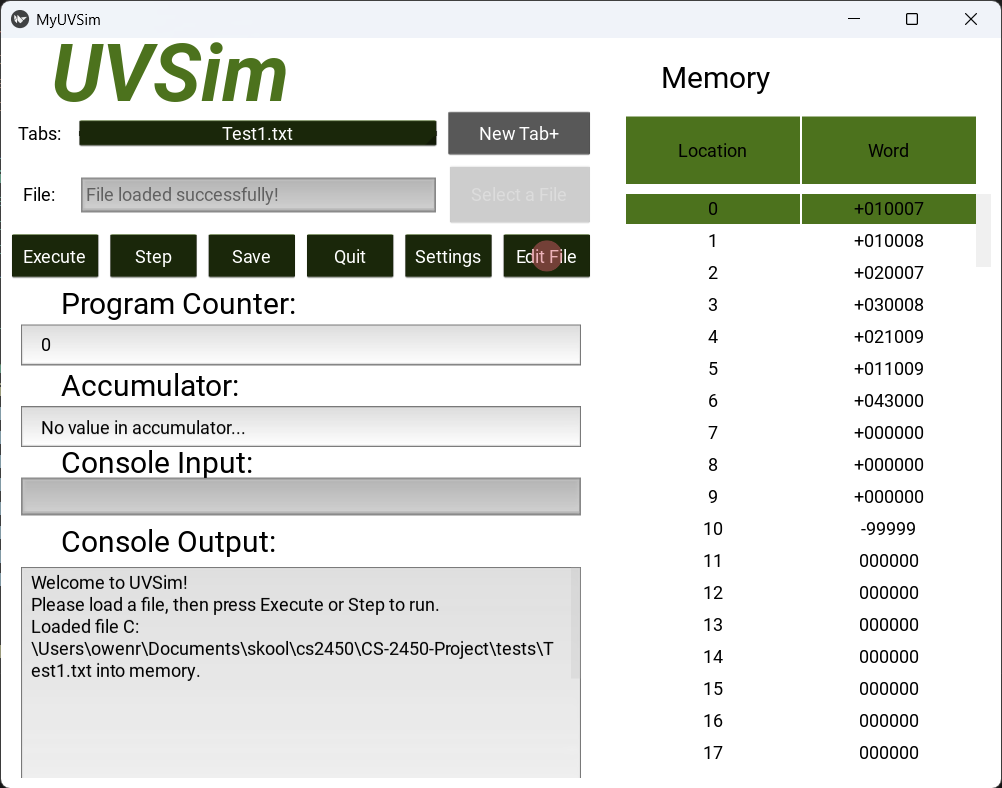
**Console Input**

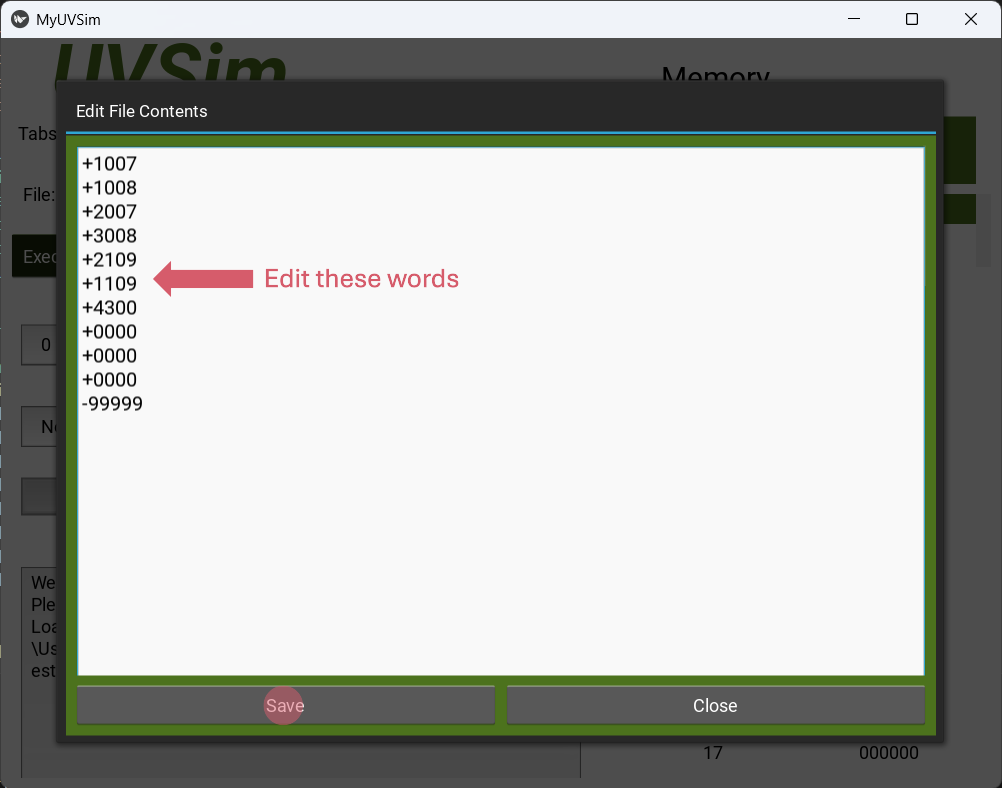
On a *Read* instruction, the console input becomes active.

Enter a 6-digit word (e.g., +123456 or -654321) and press **Enter** on the keyboard**.**

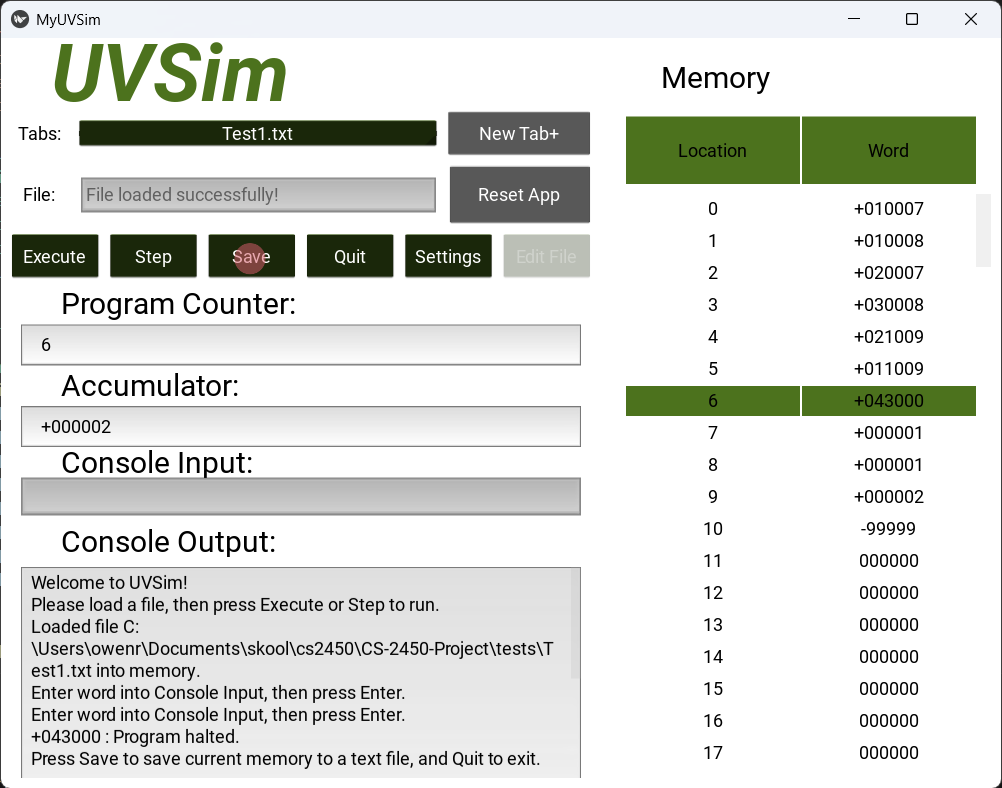
If input is valid, the word will be loaded into memory in the *Read* instruction’s specified location.

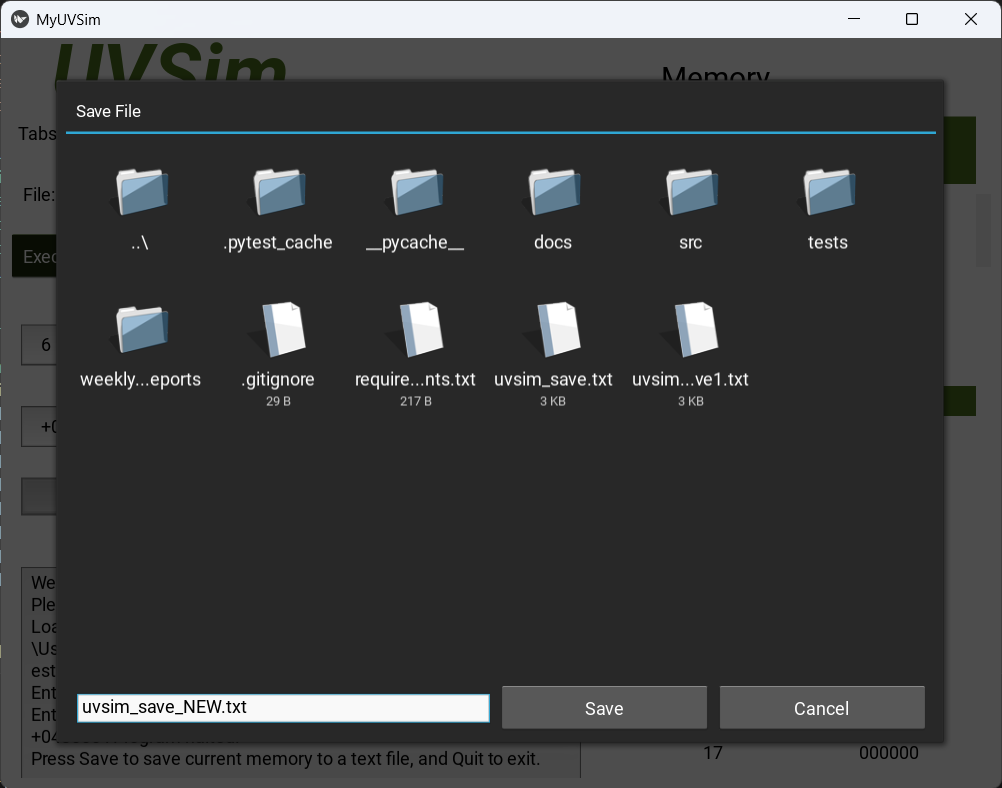
**Edit**

To edit the words in a loaded file, click “**Edit File**” to open file-editor.

Make changes, then click “**Save**” to update the memory.

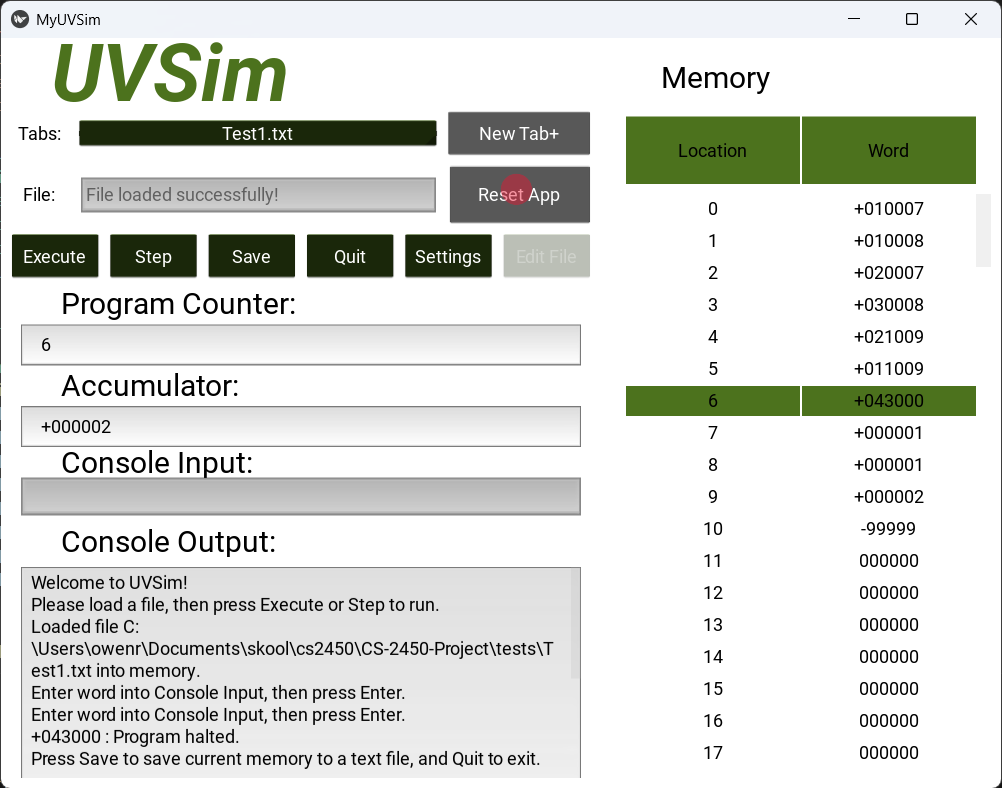
**Saving**

Click “**Save**” to export current memory and accumulator value to a .txt file.

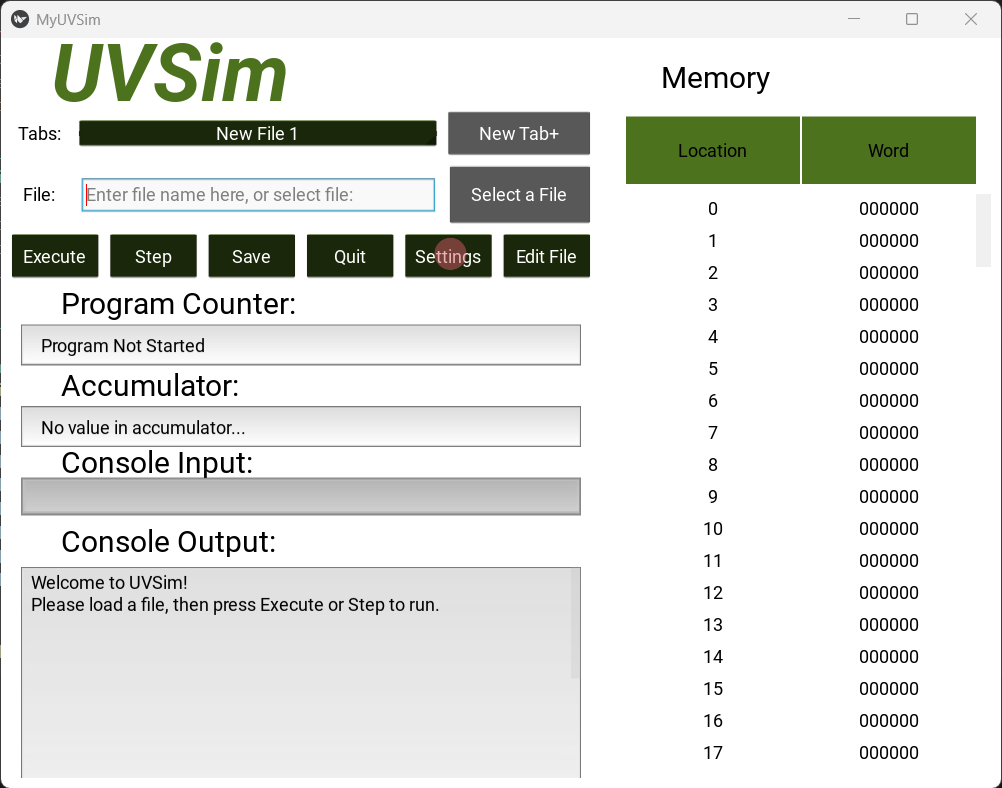
You can choose where to save and rename the file in the white text box.

**Reset**

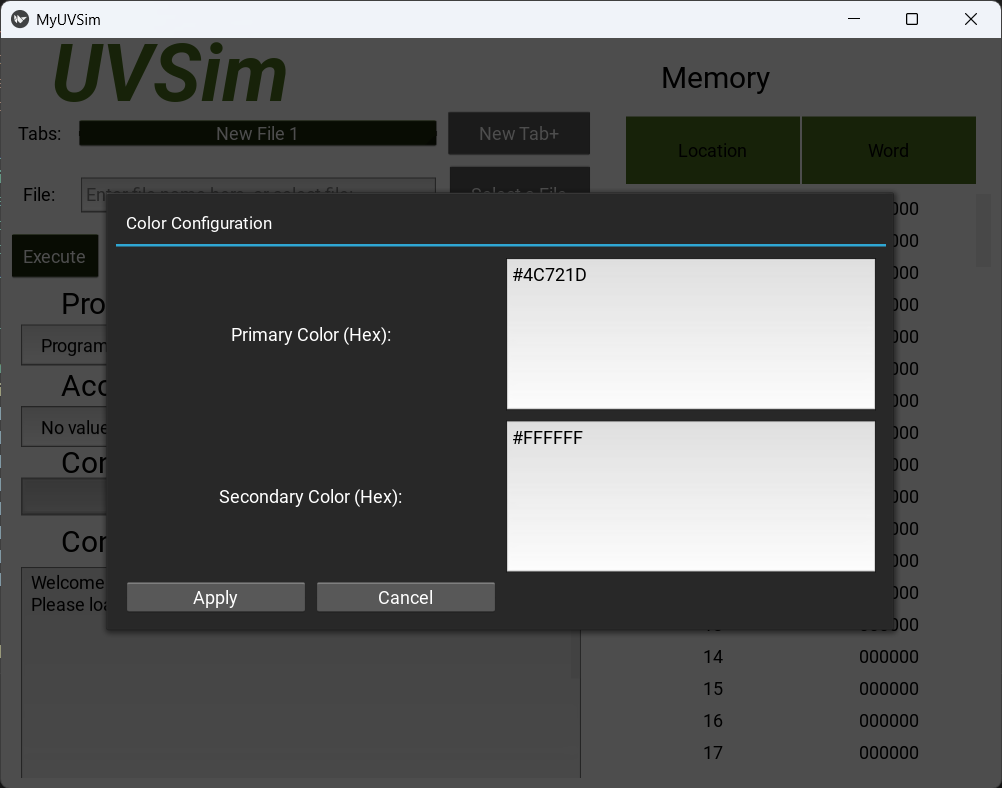
After the program halts, “**Select A File”** becomes “**Reset App**”.

Click it to reset memory, accumulator, and console for a fresh run.

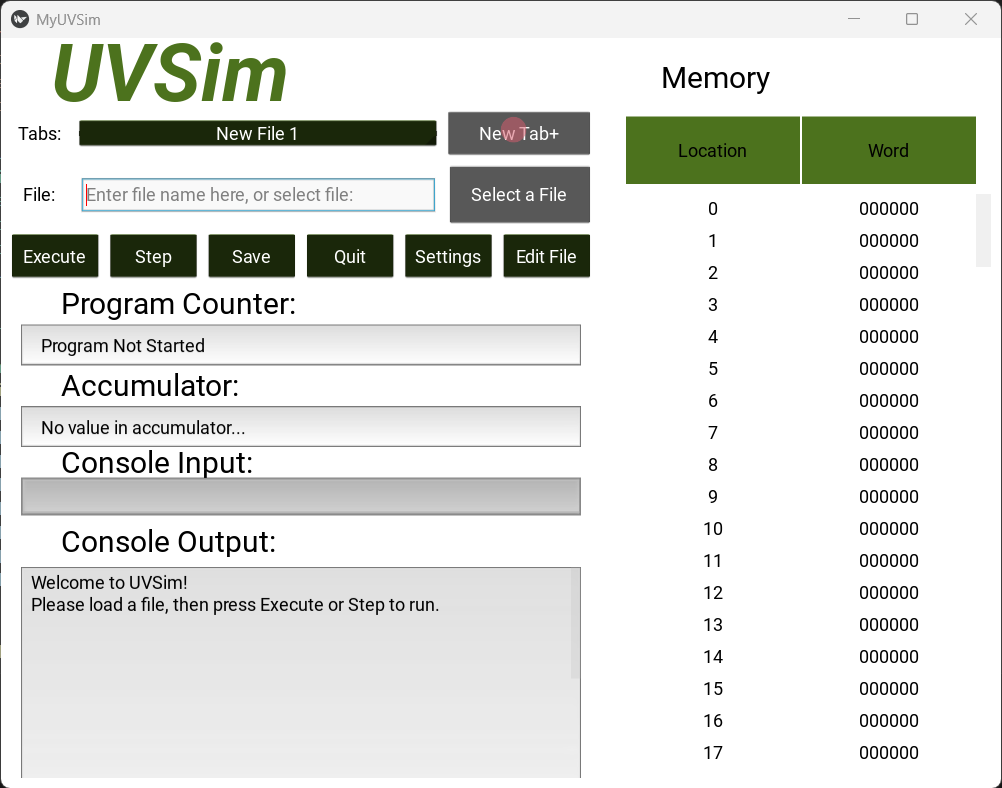
**Customizing the Interface**

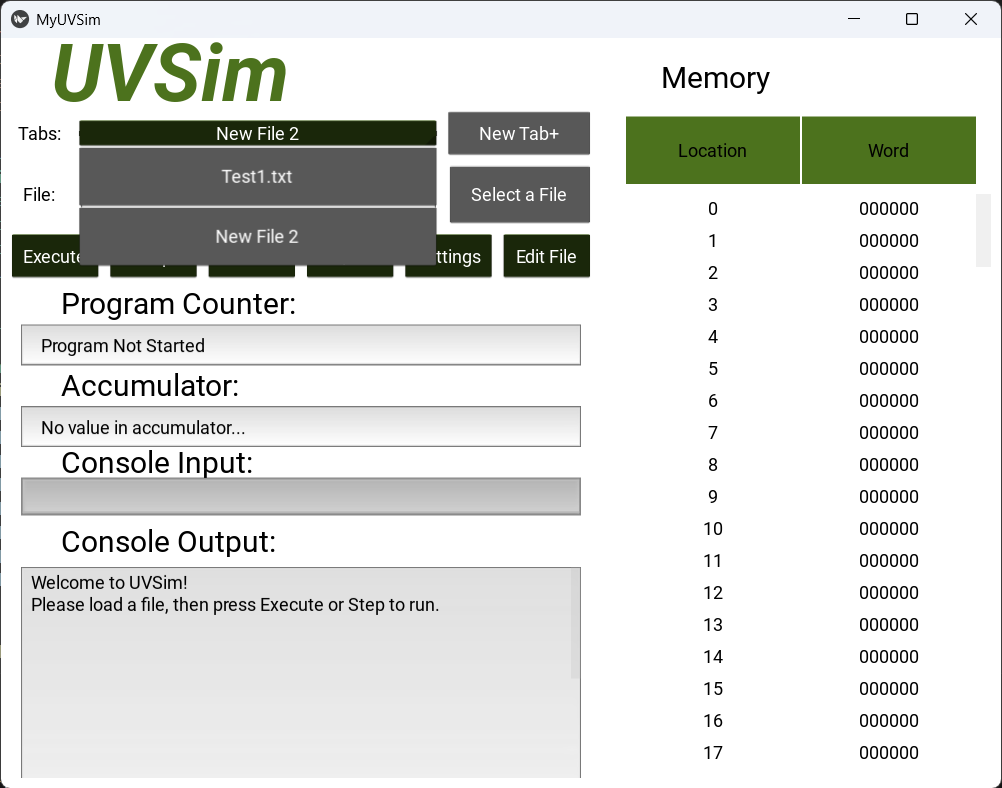
Click “**Settings**” to open the color configuration pop-up.

Modify *Primary* and *Secondary* hex colors.

Click “**Apply**” to update the app’s theme.

**Running Multiple Programs**

Click “**New Tab+**” to start another program in a new screen.

Use dropdown labeled “**Tabs:**”to switch between open files.

**Example Workflow**

1. Launch the app
2. Load file: click **Select a File** and choose *test1.txt*
3. Click **Execute**
4. Provide input when prompted
5. View output in **Output Console**
6. Click **Save** to export results
7. Click **Reset App** to re-run - OR - Click **Quit** to exit

# Future Road Map

The UVSim project has laid a solid foundation for simulating Basic Machine Language instructions through an interactive graphical interface powered by Kivy. Going forward, the development roadmap will focus on extending functionality, enhancing user experience, and promoting educational value for learners exploring low-level programming concepts.

The next phase of development will prioritize instruction set expansion. While the current implementation supports core arithmetic, memory, and control flow operations, adding support for additional opcodes (e.g., floating point math, bitwise logic) will broaden the simulator’s applicability and challenge students with more complex instruction types. Alongside instruction expansion, a feature to visualize the execution trace or instruction log in real-time will provide greater insight into program flow and debugging.

To enhance usability, the interface will undergo refinement. Planned improvements include adding breakpoints and introducing tooltips or hover descriptions for opcodes. A step-by-step tutorial or guided onboarding feature is also envisioned to help first-time users understand the simulator’s functionality.

For classroom use, future iterations will include a grading mode where instructors can preload a program and define expected memory states after execution. This feature would support automatic comparisons and scoring, making UVSim a viable educational assessment tool.

A longer-term goal is to migrate UVSim to a web-based environment using frameworks like Flask or FastAPI on the backend and React on the frontend. This transition would significantly improve accessibility by allowing users to run the simulator directly in their web browsers, eliminating the need for local Python installations. It would also open the door to collaborative features, cloud-based storage of simulations, and integration with online learning platforms.

As UVSim matures, it will grow from a niche simulation tool into a comprehensive educational platform for teaching computer architecture and machine-level thinking. Each step of the roadmap will be carefully considered with student usability, teaching utility, and technical robustness in mind.