User Story

User Story 1:

As an instructor, I want to provide CS students with a machine language simulation tool, so that they can experiment with low-level programming concepts without needing physical hardware.

User Story 2:

As a CS student, I want to run my BasicML program in the UVSim so I can understand how assembly-level instructions interact with CPU and memory.

Use Cases

Use Case 1: File Input by User

Actor: User
System: File handling and loader subsystem
Goal: Load a program file into memory for execution or editing
Steps:

1. Launch program.
2. Prompt user to select a file from any directory.
3. Receive file selection input.
4. Validate file existence and format (4-digit or 6-digit words).
5. Confirm file contains no more than 100 lines (old format) or 250 lines (new format).
6. Read file contents word by word (5 characters for old, 6 for new format).
7. Parse and store valid words into sequential memory addresses.
8. If invalid format or too many lines, raise error and re-prompt.
9. Display file contents in GUI for user inspection and editing.

Use Case 2: Execute BRANCH (Opcode 40 / 040 for new format)

Actor: Instruction execution unit
System: Program counter and control flow logic
Goal: Jump unconditionally to a specific memory address
Steps:

1. Parse opcode from instruction (2-digit or 3-digit opcode depending on format).

2. Identify operand (target address, supporting 2- or 3-digit addresses).
3. Verify target address within valid range (00-99 old, 000-249 new).
4. Set program counter to target address.
5. Continue execution from new memory location.

Use Case 3: Execute READ (Opcode 10 / 010 for new format)

Actor: Instruction execution unit
System: Input handling and memory writing logic
Goal: Store user input into specified memory location
Steps:

1. Parse opcode from instruction.
2. Identify target memory address from operand.
3. Prompt user for input.
4. Receive input from console.
5. Store input value in identified memory address.
6. Increment program counter.

Use Case 4: Execute HALT (Opcode 43 / 043 for new format)

Actor: Instruction execution unit
System: Program state and control logic
Goal: End program execution
Steps:

1. Parse opcode from instruction.
2. Set halted flag to True.
3. Output message to console showing halt location and accumulator value.
4. Stop instruction cycle.

Use Case 5: Execute BRANCHNEG (Opcode 41 / 041 for new format)

Actor: Instruction execution unit
System: Program counter and accumulator logic
Goal: Branch to a new address if accumulator is negative
Steps:

1. Parse opcode from instruction.
2. Identify operand (target address).
3. Check if accumulator < 0.
4. If true, set program counter to operand.
5. Else, increment program counter.

Use Case 6: Execute BRANCHZERO (Opcode 42 / 042 for new format)

Actor: Instruction execution unit
System: Program counter and accumulator logic
Goal: Branch to a new address if accumulator is zero
Steps:

1. Parse opcode from instruction.
2. Identify operand (target address).
3. Check if accumulator == 0.
4. If true, set program counter to operand.
5. Else, increment program counter.

Use Case 7: Execute LOAD (Opcode 20 / 020 for new format)

Actor: Instruction execution unit
System: Memory management and code processor
Goal: Successfully load a value into the accumulator
Steps:

1. Parse function code.
2. Identify target memory address from operand.
3. Fetch value from identified memory address.
4. Copy fetched value into accumulator register.
5. Increment program counter.

Use Case 8: Execute STORE (Opcode 21 / 021 for new format)

Actor: Instruction execution unit
System: Memory management subsystem
Goal: Store the value in the accumulator into memory
Steps:

1. Parse function code.
2. Identify target memory address from operand.
3. Copy value from accumulator to memory at identified address.
4. Increment program counter.

Use Case 9: Execute ADD (Opcode 30 / 030 for new format)

Actor: Instruction execution unit
System: Arithmetic logic unit (ALU)
Goal: Add a value from memory to the accumulator
Steps:

1. Parse function code.
2. Identify memory address from operand.
3. Fetch value from memory.

4. Add value to accumulator.
5. Store result in accumulator.
6. Increment program counter.

Use Case 10: Execute SUBTRACT (Opcode 31 / 031 for new format)

Actor: Instruction execution unit
System: Arithmetic logic unit (ALU)
Goal: Subtract a memory value from the accumulator
Steps:

1. Parse function code.
2. Identify memory address from operand.
3. Fetch value from memory.
4. Subtract value from accumulator.
5. Store result in accumulator.
6. Increment program counter.

Use Case 11: Execute DIVIDE (Opcode 32 / 032 for new format)

Actor: Instruction execution unit
System: Arithmetic logic unit (ALU)
Goal: Divide the accumulator by a value from memory
Steps:

1. Parse function code.
2. Identify memory address from operand.
3. Fetch value from memory.
4. If value ≠ 0, divide accumulator by value.
5. Store result in accumulator.
6. If value == 0, raise divide-by-zero error and halt.
7. Increment program counter unless halted.

Use Case 12: Execute MULTIPLY (Opcode 33 / 033 for new format)

Actor: Instruction execution unit
System: Arithmetic logic unit (ALU)
Goal: Multiply accumulator by a value from memory
Steps:

1. Parse function code.
2. Identify memory address from operand.
3. Fetch value from memory.
4. Multiply value by accumulator.
5. Store result in accumulator.
6. Increment program counter.

Use Case 13: Convert 4-Digit File to 6-Digit File Format

Actor: User
System: File conversion utility within the application
Goal: Convert an existing 4-digit word format file into the new 6-digit word format file for extended memory and functionality
Steps:

1. User opens the conversion tool from the application GUI or menu.
2. Prompt user to select a 4-digit format file from any directory.
3. Validate the selected file contains only valid 4-digit words and does not exceed 100 lines.
4. Read each line of the file sequentially.
5. For each line:
   - If the first two digits match a valid function opcode, convert using the 0XX0XX format (e.g., 1007 → 010007).
   - Otherwise, treat as numerical value and convert using the 00XXXX format (e.g., 5555 → 005555).
6. Reject the file if any invalid or ambiguous lines are detected.
7. Prompt user to save the new 6-digit file in a user-chosen directory with a new filename.
8. Save the converted file respecting the 250-line max and six-digit word format.
9. Confirm conversion success and optionally open the new file in the editor.

Use Case 14: Color Scheme Customization

Actor: User
Goal: Customize the color scheme of the application
Steps:

1. Launch the application.
2. Navigate to color settings (through GUI or a config file).
3. Select a primary and an off-color from a color picker or by entering RGB/Hex values.
4. Apply changes instantly or restart the app to see changes.
5. Ensure readability of text with the selected color scheme.

Use Case 15: Load and Save Files from Custom Directories

Actor: User
Goal: Load and save files from user-specified directories
Steps:

1. Launch the program.
2. Use the "Open" button to navigate and load files from any directory.
3. Edit the file as necessary.
4. Save the file to any directory or under a new name using the "Save" button.

Functional specifications

Functional:

1. The system shall display a ‘load program file’ button.

2. The system GUI shall exhibit a primary color, used as the main background color. The primary color shall default to UVU green (Hex# 4C721D).

3. The system GUI shall exhibit a secondary color, used for clickable buttons and text. The secondary color shall default to white (Hex# FFFFFF).

4. The system colors shall be user-configurable via a configuration file or in-app option without recompilation.

5. The system shall allow users to open a text file when the ‘load program file’ button is clicked. The text file shall be imported via a user-chosen directory.

6. The system shall raise an error if one or more words in the text file are not valid 4- or 6-digit signed integers.

7. The system shall load the contents of the chosen file into editable memory via the GUI, allowing user review before execution.

8. The system shall allow a user to make changes to their file inside the GUI including cut, copy, paste, add, delete, and edit operations.

9. The system GUI shall allow a user to save their file to a user-chosen directory with optional renaming.

10. The program shall begin operation at the first location in memory (000) when the ‘run program’ button is clicked.

11. The system’s memory access shall be restricted to valid address space (000–249).

12. The system shall display a ‘run program’ button, allowing an open file to be executed.

13. The system shall open a user-input popup when a READ command (010XXX) is encountered in the program.

14. The system shall read the contents of the input field in the input popup into memory when the user clicks the ‘submit’ button in the popup window.

15. The system’s execution shall support various 6-digit opcodes within the range 010–043:

15a. Opcode 010: Read a word from the keyboard into a specific location in memory

15b. Opcode 011: Write a word from a specific location in memory to screen

15c. Opcode 020: Load a word from a specific location in memory into the accumulator

15d. Opcode 021: Store a word from the accumulator into a specific location in memory

15e. Opcode 030: Add a word from a specific location in memory to the word in the accumulator

15f. Opcode 031: Subtract a word from a specific location in memory from the word in the accumulator

15g. Opcode 032: Divide the word in the accumulator by a word from a specific location in memory

15h. Opcode 033: Multiply a word from a specific location in memory to the word in the accumulator

15i. Opcode 040: Branch to a specific location in memory

15j. Opcode 041: Branch to a specific location in memory if the accumulator is negative

15k. Opcode 042: Branch to a specific location in memory if the accumulator is zero

15l. Opcode 043: HALT: stop the program

16. The system shall display an accumulator value, with the accumulator being a memory register with the same size and functions of the other memory registers.

17. The system shall reset the accumulator to 0 and memory to all zeros when a new file is loaded in.

18. The system shall end program execution when a HALT command (043XXX) is encountered.

19. The system shall display the contents of memory (up to 250 entries) in a scrollable, editable table format, confirming when a program is successfully loaded.

20. The system shall display output in a separate text area when the program executes a WRITE instruction (011XXX).

21. The system shall prevent the user from running the program if no file is loaded or if the file contains format errors.

22. The system shall allow the user to close the application in the GUI.

23. The system shall initialize all memory cells (000–249) and the accumulator to 0 when a new file is loaded in.

24. The system shall reject malformed instructions (such as: invalid characters, unsupported opcodes, or incorrect word length).

25. The system shall raise an error on division by zero.

26. The system’s accumulator and register values exceeding ±999999 shall wrap using modulo 1,000,000.

27. The system shall display error messages coinciding with errors that might be encountered (ex. bad words in a file, missing words, trying to access out-of-range memory).

28. The system shall support multiple files open at once via GUI tabs or windows. 4-digit files shall be automatically converted to 6-digit format before editing or running.

29. The system shall only allow files with words of consistent size, either 4-digit or 6-digit words.

Non-functional:

1. The system shall be built using tiered architecture.

2. The system shall avoid crashes through input validation and exception handling.

3. The system shall make it clear to the user if the file chosen isn’t of the correct format (e.g., wrong length, illegal characters, too many lines).

4. The system shall be compatible with Python 3+.

5. The system shall be compatible with Windows and macOS.

6. The system shall execute the program and remain responsive (≤ 3 seconds for 250 instructions).

Class Diagram

```
+---------------------------------------------+
|                 ColorConfig
+---------------------------------------------+
| - filename: str
| - primary: str
| - off: str
+---------------------------------------------+
| + __init__(filename="colors.json")
| + is_valid_hex(color: str): bool
| + load_colors(): void
+----------------------------------------------+
                 ▲
                 | used by
+---------------------------------------------------------+
|                      UVSimGUI
+---------------------------------------------------------+
| - root: Tk
| - sim: UVSim
| - load: Load_Program
| - colors: ColorConfig
| - memory_entries: list[Entry]
| - output_lines: list[str]
| - file_loaded: bool
| - run_button: Button
| - mem_canvas: Canvas
| - accumulator_var: StringVar
+----------------------------------------------------------+
| + __init__(root: Tk)
| + build_widgets(): void
| + load_file(): void
| + save_file(): void
| + save_memory_entry(index: int, value: str)
| + sync_entries_to_memory(): bool
| + run_program(): void
| + step_through_program(): void
| + show_read_popup(operand: int): int
| + update_memory_display(): void
| + update_output(): void
| + clear_output(): void
| + open_file_in_new_window(): void
+-----------------------------------------------------------+
▲                           ▲                            ▲
|                           |                             |
uses|                       has-a                       uses|
|                           |                                |
+-----------------------+  +-----------------------------------+  +-------------------------------------+
|         UVSim            |     Load_Program              |          Convert
+------------------------+  +-----------------------------------+  +-----------------------------------+
```

```
| - cpu: CPU                    | - memory: Memory              | + convert_to_int(word): int or False
| - memory: Memory      +-------------------------------------------+    +--------------------------------+
+--------------------------+     | + __init__(): void
| + run(): void                  | + load_program(filename: str): void
+---------------------------+    +---------------------------------------------+

              ▲
              | has-a
              |
+-----------------------------------------------------------+
|                      CPU
+-----------------------------------------------------------+
| - accumulator: int
| - program_counter: int
| - halted: bool
+-------------------------------------------------------------+
| + __init__()
| + execute(instruction: int, memory: Memory): void
+--------------------------------------------------------------+
              ▲
             uses
              |
+-----------------------------------------+
|                  Memory
+------------------------------------------+
| - memory: list[MemoryRegister]
| - read_callback
| - waiting_for_input: bool
+-------------------------------------------+
| + __init__()
| + write(address: int, value: int): void
| + read(address: int): int
+--------------------------------------------+
              ▲
              has-a
              |
+---------------------------+
|    MemoryRegister
+----------------------------+
| - value: int
+---------------------------+
| + __init__(value=0)
| + set(value: int): void
| + get(): int
| + __repr__(): str
+--------------------------+
      ▲
      | called by
      |
+----------------+
|   main.py
```

```
+----------------+
| + main(): void
+-----------------+
```

Class Descriptions

ColorConfig

- Loads GUI color themes from a colors.json file.
- Validates hex format.
- Allows customization of primary and off UI colors.

UVSimGUI

- Main GUI class that handles user interaction.
- Manages widgets, program loading, file handling, memory view, and execution.
- Coordinates between Load_Program, UVSim, and ColorConfig.

Load_Program

- Responsible for loading instruction files into memory.
- Parses .txt files and handles file format validation.

Convert

- Static class or helper for converting between formats.
- Changes strings into integers by striping the whitespace.

UVSim

- Simulator core.
- Ties together CPU and memory.
- Executes the program loaded in memory via CPU instructions.

CPU

- Core processor emulation.
- Handles opcodes, accumulator logic, branching, and program counter.
- Executes instructions by operating on Memory.

Memory

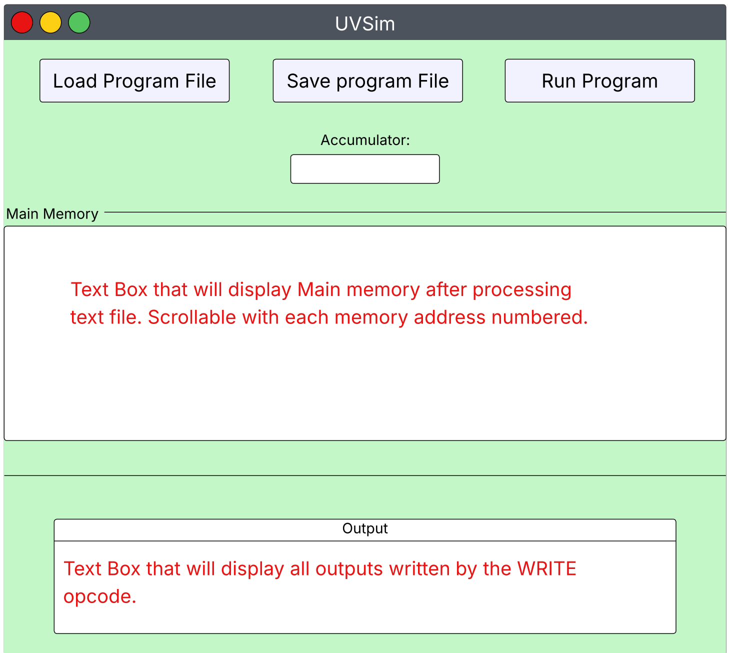
- Stores all instructions and values.
- Offers read and write operations with safety checks and callback support.

MemoryRegister

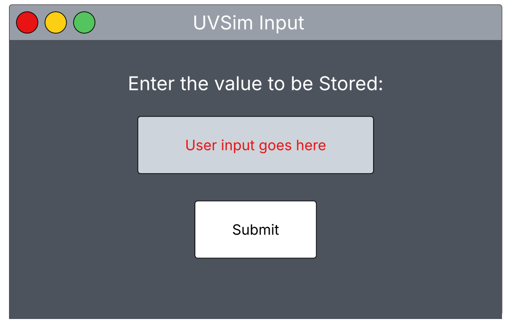
- Represents a single memory cell (integer value).
- Provides getter/setter and string conversion.

main.py

- Entry point for launching the GUI application.



Pop Up Window: Pops up to ask user for READ input.



Unit Tests - Group B : UnitTests						
	A	B	C	D	E	F
1	**Name**	**Description**	**Use Case(s)**	**Inputs**	**Expected Outputs**	**Conclusion (How we know it works)**
2	test_file_open	Tests whether a file can be read by the program.	Retrieving a file necessary for program operation	Test1.txt' (assumes this file exists), and 'qwertyuiop.txt' (assumes this file does not exist)	No output if it succeeds, raises OS error if the file does not exist	We are able to run the program and verify that it works. The test also passes when attempting to open a file that exists.
3	test_program_init	Tests that a correctly formatted file is properly read into memory.	Retrieving a file necessary for program operation	Any existing file that is formatted correctly (in this case uses 'Test1.txt' and 'Test2.txt'	Compares values in memory with expected values, fails if values do not match	Test passes, and test print output in the main program displays expected values in memory.
4	test_negative_values	Tests that negative values are added (subtracted) correctly in the accumulator.	When negative values are input into memory.	100 into accumulator, values of -30 and -50	Accumulator = 20	Accumulator successfully contains the value '20' after negative values are input. (100-30-50 = 20)
5	test_subtract_negative	Tests that subtracting negatives leads to adding in the accumulator.	Program encounters a 'subtract negative' situation	100 into accumulator, values of -30 and -50	Accumulator = 180	Accumulator contains expected value after subtracting -30 and -50 from 100.
6	test_bad_word_format	Verifies that improper words cannot be written into memory, and checks that the program cannot run if memory somehow does contain improper text.	Bad values are written to memory.	Write "TEST" into memory, run a program containing "TEST" in memory	Type errors in both cases.	The tests return errors when attempting to write invalid words into memory, and when trying to run a file containing invalid words.
7	test_improper_word_in_file	Verifies that invalid words contained in the designated file raise errors and are not allowed by the program.	Invalid words exist in the text file.	A text file containing too many characters, and a text file containing a word that should not be recognized by the program.	Value errors in both cases.	The test returns value errors when attempting to read in a file containing words of the wrong length or words that cannot be recognized.
8	test_branch	Test that makes sure branch moves pointer to designated part in memory	Branch to specific part in memory	4005, 05 for branch function	Program counter moves to 05	Test function asserts that counter is at memory location 05 after using BRANCH 4005
9	test_branch_executes_until_halt	Test that makes sure program still runs after branching.	Branch to specific part in memory	4001, 01 for branch function, 4300 for HALT function	Program moves counter to 01 then Halts	The HALT successfully passes after branching. test_branch verifies the branch moves the counter.
10	test_write_prints_hello_world	Test that proves the program can create basic programs such as printing Hello World.	Print Hello World to console	"Hello World" into memory.	Program prints "Hello World" from memory to console	Program prints 'Hello World" from memory to console.
11	test_input_then_print_hello_world	Test that shows user inputs are correctly saved into memory and can be printed to console.	Print Hello World to console	User inputs 'Hello World" into memory using Read.	Program prints "Hello World" from memory to console	Program prints 'Hello World" from memory to console.
12	test_load_and_store_value	Checks that load and store are working as expected.	Load and Store Data to memory and accumulator.	Inputs 42 at memory position 10, stores 42 into accumulator, which then loads value into memory position 11.	Accumulator = 42, Memory[11] = 42	The accumulator and memory [11] can only = 42 if the LOAD and STORE opcodes are working properly.
13	test_invalid_address_raises	Checks that program catches invalid addresses and raises and IndexErrorl.	Load and Store Data memory and accumulator.	Input: 99 at memory location 200	IndexError	200 is out of range, and so the program showing an IndexError means it is handling memory correctly.
14	test_multiply_two_values	Verifies that the Multiply opcode is working and produces correct output.	Utilize Multiplication Opcode	Inputs: 6 in Accumulator, 7 to Memory	Accumulator = 42	7 * 6 = 42, verifies the correct output and memory location (accumulator) of the multiply opcode.
15	test_multiplication_with_zero	Further functionality test of the multiply opcode with zero.	Utilize Multiplication Opcode	Inputs: 0 in Accumulator, 5 to Memory	Accumulator = 0	5 * 0 = 0, verfieis the correct output of the multiply opcode.
16	test_division_two_values	Verifies that division opcode is working and produces correct output.	Utlizie Division Opcode	Inputs: 42 in accumulator, 7 to memory	Accumulator = 6	42 / 7 = 6, verifies the correct output of the division opcode
17	test_division_with_zero	Checks that trying to divide accumulator by zero produces a ZeroDivisionError.	Utilize DIvision Opcode	Inputs: 10 in accumulator, 0 to memory.	ZeroDivisionError	A ZeroDivisionError proves that the program is not moving ahead with erroneous computations.
18	test_memory_write_and_read	Verifies that memory can be written to and read from, and that invalid accesses raise errors.	Writing and reading data to/from memory.	write(10, 1234), then read(10); Write (150, 9999) (will raise IndexError)	Read from address 10 returns 1234; writing to 150 raises IndexError.	Successfully reads and writes within bounds; error is raised when accessing out-of-bounds memory, confirming memory limits are enforced.
19	test_load_store	Verifies that the CPU can load data from memory into the accumulator and store it back into another memory location.	Data transfer between memory and CPU	memory[5] = 4321; LOAD from 2005, STORE to 2106	accumulator = 4321, memory[6] = 4321	The CPU correctly transfers data between memory and accumulator.
20	test_add_subtract	Checks that ADD and SUBTRACT opcodes update the accumulator as expected.	Arithmetic operations in CPU	acc = 100, memory[1] = 50, memory[2] = 25	accumulator = 125 after ADD then SUBTRACT	The accumulator holds the correct result after sequential arithmetic.
21	test_multiply_divide	Verifies correct functionality of MULTIPLY and DIVIDE opcodes.	Arithmetic operations in CPU	acc = 10, memory[3] = 5, memory[4] = 2	accumulator = 25 after multiply/divide	Confirms correct execution of compound arithmetic logic.
22	test_branch_operations	Tests conditional and unconditional branch instructions.	Instruction control flow	acc = -1, then 0, then 5; various BRANCH, BRANCHNEG, BRANCHZERO opcodes	program_counter = 7, 9, 2, 1 accordingly	Asserts that program counter updates correctly based on accumulator state.
23	test_halt	Ensures HALT opcode stops execution.	End of program execution	4300	cpu.halted = True	Confirms that the halt flag is set when HALT is executed.
24	test_bad_opcode	Ensures an invalid opcode raises an appropriate error.	Error handling for invalid instructions	ex. 5555	ValueError	Program correctly identifies and rejects unsupported opcodes.
25	test_bad_word_length	Validates that too-long memory words raise errors.	Memory validation	memory.write(5, 12345)	ValueError	Writing a 5-digit word fails validation, as expected.

UVSim Software Simulator

User Manual

1. Introduction

UVSim is a Python-based software simulator designed to execute programs written in BasicML—a simplified educational machine language. This application provides a graphical user interface (GUI) using the tkinter library and supports both legacy and modern instruction formats. The simulator enables file loading, editing, execution, and debugging of machine code programs.

2. System Requirements

- **Python Version:** Python 3.x (Tkinter is typically included)
- **Operating System:** Cross-platform (Windows, macOS, Linux)
- **Libraries:** Only tkinter (included with Python)

3. Launching the Application

To launch UVSim:

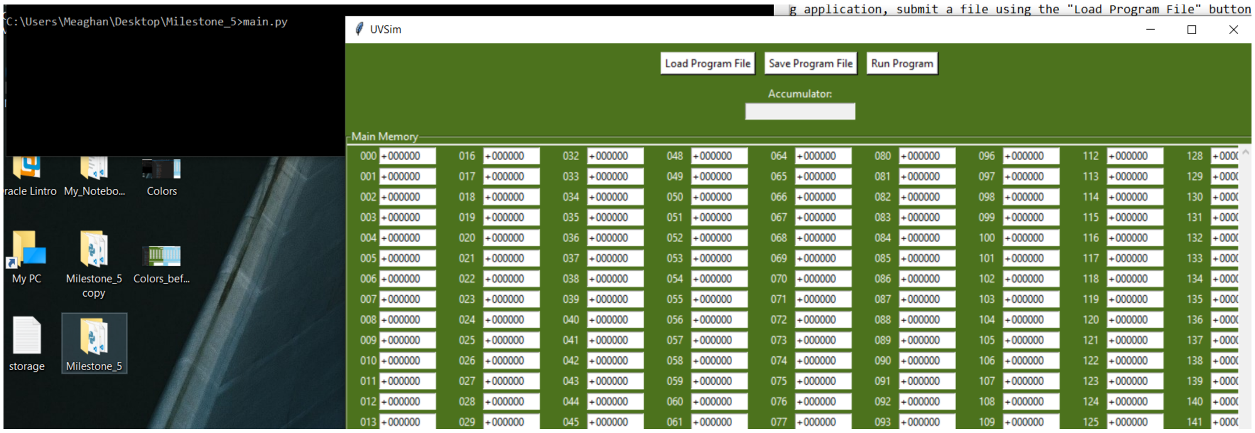
1. Open a terminal or command prompt.
2. Navigate to the directory where main.py is located.
3. Run the following command:

```
python3 main.py
```

4. Application Overview

Upon launching, the main interface presents several buttons and a program editor.

- **Load Program File** – Load a .txt file with BasicML instructions.
- **Run Program** – Execute the loaded instruction set.
- **Save Program File** – Save current instructions to a file.
- **Instruction Editor** – Edit instructions directly in the GUI.
- **Popup Windows** – Separate file tabs allow multiple files to be open at once.



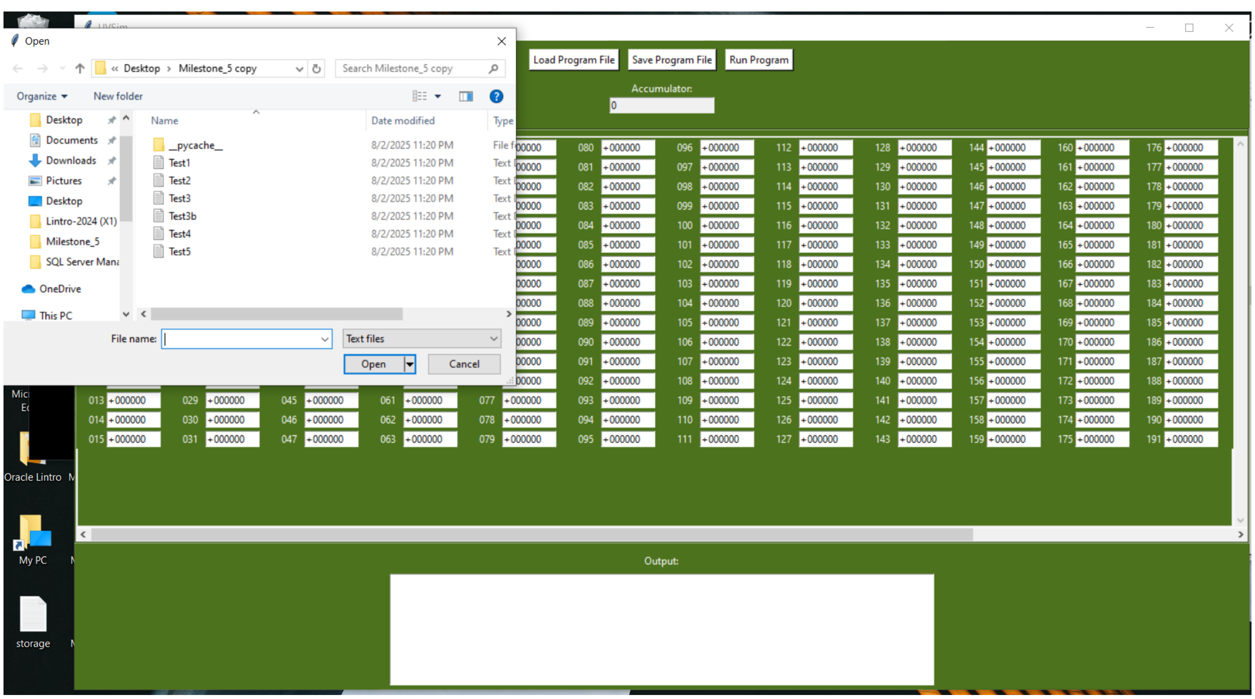
5. How to Use UVSim

5.1 Loading and Running Programs

Test3b.txt

```
+2020
+3021
+2127
+1127
+3222
+2127
+1127
+3323
+2127
+1127
+3224
+2127
+1127
+3025
+2127
+1127
+3126
+2127
+1127
+4300
+1111
+2222
+1111
+0444
+0002
-1000
+1000
```

1. Click **Load Program File** to import a .txt file.
2. Click **Run Program** to begin execution.
3. For input instructions (opcode READ), a pop-up will request user input. Enter the value and click **Submit**.
4. Upon HALT, the program stops and returns output.
5. To execute a new file, repeat the above process.



5.2 Editing Instructions in GUI

- Add, delete, cut/copy/paste, and modify lines directly.
- File must not exceed the memory limit:
  - Legacy format: max 100 lines (00–99)
  - New format: max 250 lines (000–249)

6. File Format Specifications

6.1 BasicML File Structure

Test3b.txt

```
+2020
+3021
+2127
+1127
+3222
+2127
+1127
+3323
+2127
+1127
+3224
+2127
+1127
+3025
+2127
+1127
+3126
+2127
+1127
+4300
+1111
+2222
+1111
+0444
+0002
-1000
+1000
```

- Plain text (.txt) file
- One signed integer instruction per line
- Must include a HALT instruction:
  - Legacy: +4300
  - New: +043000

6.2 Formats

- **Legacy (4-digit)**: +1020, -3001, etc.
- **New (6-digit)**: +010035, +030250, etc.
- File must be **consistently formatted** with all lines of equal digit length.

7. BasicML Instruction Set

7.1 Legacy Opcodes (4-digit)

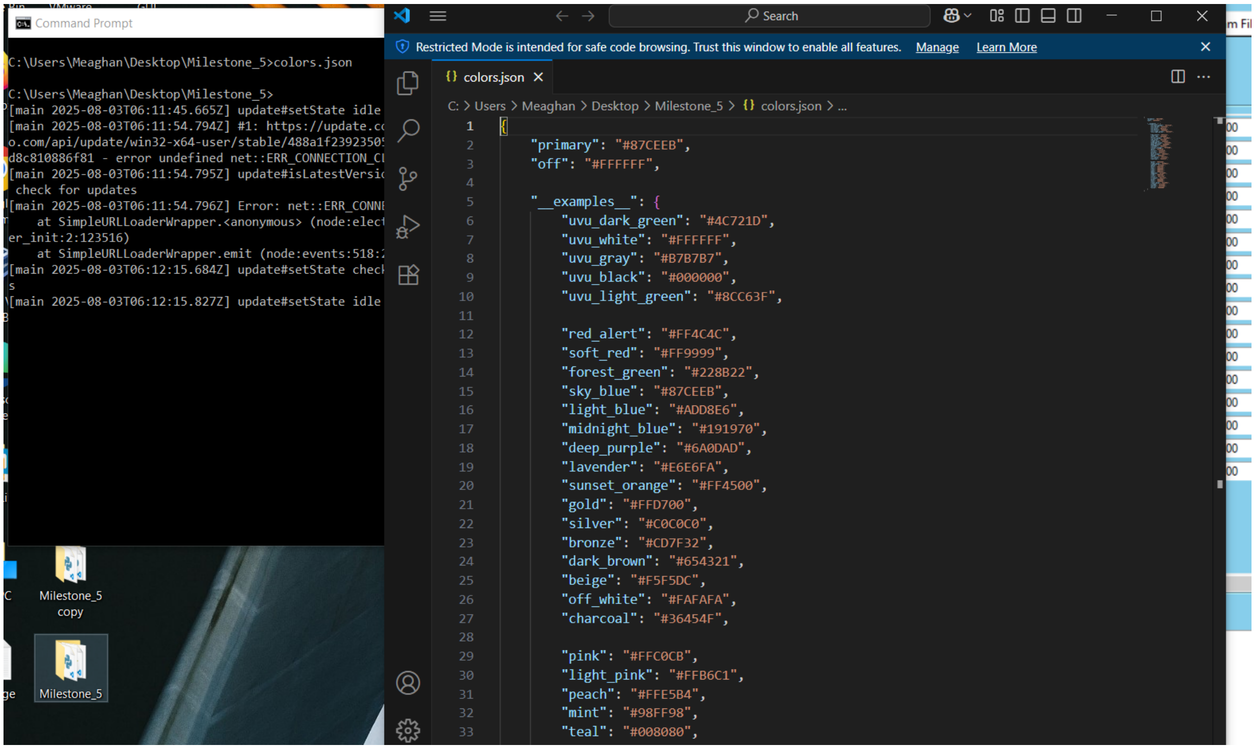
Opcode	Operation	Description
10	READ	Input a value to memory
11	WRITE	Output a value from memory

Opcode	Operation	Description
20	LOAD	Load memory value into accumulator
21	STORE	Store accumulator into memory
30	ADD	Add memory value to accumulator
31	SUBTRACT	Subtract memory value from accumulator
32	DIVIDE	Divide accumulator by memory value
33	MULTIPLY	Multiply accumulator by memory value
40	BRANCH	Jump to memory location
41	BRANCHNEG	Jump if accumulator is negative
42	BRANCHZERO	Jump if accumulator is zero
43	HALT	End program execution

7.2 New Opcodes (6-digit)

Opcode	Operation
010	READ
011	WRITE
020	LOAD
021	STORE
030	ADD
031	SUBTRACT
032	DIVIDE
033	MULTIPLY
040	BRANCH
041	BRANCHNEG
042	BRANCHZERO
043	HALT

8. Customizing the Interface Colors



The program's color scheme can be customized by editing the colors.json file:

1. Navigate to the file location and open colors.json.
2. Edit the primary and off color hex codes:

```
{
  "primary": "#4C721D",
  "off": "#FFFFFF"
}
```

3. Hex code format must start with # followed by six characters (0–9, A–F).
4. Save changes and re-launch the program.
5. If errors appear, ensure the hex codes are valid.

9. File Conversion and Compatibility

- Legacy 4-digit files are automatically converted to 6-digit instructions internally.
- The simulator supports up to **250** memory addresses for 6-digit files.
- Avoid mixing instruction formats within a single file.

10. Error Handling and Debugging

If errors occur:

- The application halts and reports the issue.
- A detailed error log is saved in a .txt file.
- Example issues:
    - Invalid opcode
    - Improper formatting
    - Instruction exceeding memory limit

11. Example Instruction Breakdown

4-Digit Format

- +1235

- 12 → Opcode
- 35 → Memory Location

6-Digit Format

- +010035
  - 010 → Opcode
  - 035 → Memory Location

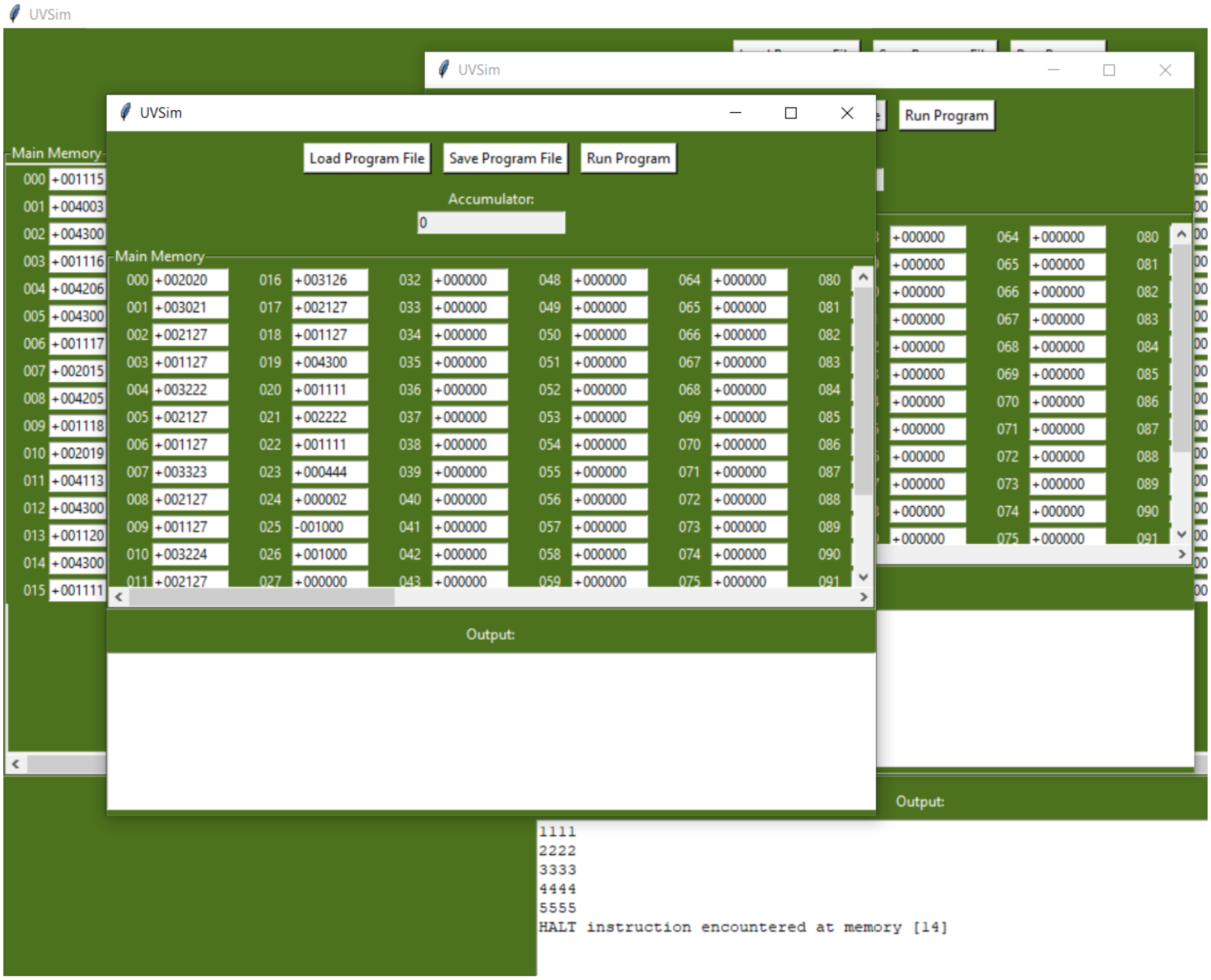
12. Testing Instructions

To test your code:

1. Create a .txt file with valid instructions.
2. Load it in UVSim.
3. If no errors occur, the code executes.
4. If errors occur, check the log file and correct any formatting issues.

13. Notes

- You can load multiple program files simultaneously via GUI tabs.
- Only **one program** can be executed at a time.



14. Support

For bug reports or feature requests, please submit issues to the GitHub Repository.