UVSimulator

Contributors: John Christianson, Josh Gimenes, Myles Mangum, Brandon Fitzgerald

Table of Contents

Introduction

A key part of Computer Science Education is understanding computer architecture and low-level programming. To enhance students' learning of these topics, virtual simulations of basic computers and their components can teach core concepts without getting bogged down by real-world technicalities. In order to assist with this concept, over the last few months our team has been developing UVSim. UVSim is a simple simulation of a fairly basic computer with only a few sets of instructions. The minimal variability, however, is paired with a high level of customization and interactability that is focused on the learning and accommodations of potential students.

The current version of UVSim has all the logic in the backend to allow students to test their own code and files. There is also a decent amount of customization, including allowing multiple files, changing the colors of the application, writing and saving new files, and visualization of the virtual memory. These tools enable students to test their knowledge of computer logic and understand how their code interacts with computer concepts such as accumulators and program counters. We are presenting this project to you to show you how we have already started to create this great application, and where we would like to take it in the future with you. The rest of this document includes requirements of the software, ideas for future improvements, instructions on how to run the application, UML diagrams of the codebase, and all unit tests run to verify the application's functionality.

Requirements and Use Cases

Use Cases

1. Load Program File
Actor: Student at the keyboard
System: ProgramLoader + Memory
Goal: Populate memory locations 000-NNN with the words from the selected text file.
Prompt the user for a filename.
Open the file and read each line.
Convert each line to a MemoryRegister word; ignore blank lines and comments.
Display to uthe ser what lines have been ignored.
Write the word into the next memory cell.
Close the file; return success or error (file not found etc.).

2. Validate Memory Address
Actor: Memory class internals
System: Memory bounds checker
Goal: Prevent illegal access outside 0-99.
Receive a read(addr) or write(addr,val) request.
If addr < 0 or > 99, throwan exception.
Otherwise, continue with the read or write.

3. Fetch Instruction
Actor: CPU fetch-decode-execute loop
System: UVCpu
Goal: Retrieve the next instruction word for decoding.
Read memory at programCounter.
Split the four-digit word into opcode (first 2) and operand (last 2).
Increment programCounter to point to the subsequent cell.

4. WRITE (11)
Actor: CPU
System: Console handler
Goal: Display a word from memory.
Fetch value from operand address.
Print value with sign and leading zeros.

5. LOAD (20)
Actor: CPU
System: Register logic
Goal: Move a word from memory into the accumulator.
Fetch value at operand address.
Copy value into the accumulator register.

6. STORE (21)
Actor: CPU
System: Register logic
Goal: Save the accumulator back to memory.
Read the current accumulator.
Write it to the operand address in memory.

7. ADD (30)
Actor: CPU arithmetic unit
System: ALU emulation
Goal: Update the accumulator with addition.
Fetch operand value.
Compute acc = acc + operand.
If |acc| > 999999, flag overflow, cut off the highest digits until no longer overflowing.
Otherwise, store the result in the accumulator.

8. SUBTRACT (31)
Actor: CPU arithmetic unit
System: ALU emulation
Goal: Update the accumulator with subtraction.
Fetch operand value.
Compute acc = acc - operand.
If |acc| > 999999, flag overflow, cut off the highest digits until no longer overflowing.
Otherwise, store the result in the accumulator.

9. DIVIDE (32)
Actor: CPU arithmetic unit
System: ALU emulation
Goal: Divide the accumulator by the operand.
Fetch operand value.
If operand == 0, print divide by zero error, HALT.
Else compute integer division, store in the accumulator.

10. MULTIPLY (33)
Actor: CPU arithmetic unit
System: ALU emulation
Goal: Multiply the accumulator by a word from memory.
Fetch operand value from memory.
Compute acc = acc * operand.
If |acc| > 999999, flag overflow, cut off the highest digits until no longer overflowing.
Otherwise, store the result in the accumulator.

11. BRANCH (40)
Actor: CPU control unit
System: Program counter manager
Goal: Unconditionally alter sequential flow.
Set programCounter = operand.

12. BRANCHNEG (41)
Actor: CPU control unit
System: Program counter manager
Goal: Alter sequential flow if accumulator is negative.
If acc < 0, set programCounter = operand; else continue sequentially.

13. BRANCHZERO (42)
Actor: CPU control unit
System: Program counter manager
Goal: Alter sequential flow if accumulator is zero.
If acc == 0, set programCounter = operand; else continue sequentially.

14. HALT (43)
Actor: CPU loop
System: UVCpu
Goal: End program execution cleanly.
Detect opcode 43.
Print "Program halted normally."
Break out of the main loop and exit to OS.

15. Change GUI Colors
Actor: Student using the program
System: UVSimGUI
Goal: Change the background colors of the GUI based on the user's preference when requested
Customize button is selected
Another menu displaying primary and secondary color choices is provided.
One of these buttons is selected
A display of all colors available is presented for the user's choice.
A new color is chosen and confirmed
Panels flagged and stored as either "Primary" or "Secondary" will be iterated through to modify based on the user's choice.

16. Display User's Code Visually
Actor: User opening a file
Goal: Display chosen files by the user in a readable format.
Selects "Open Program File"
GUI will open up a secondary window displaying possible selections
The user selects and confirms a file
GUI will send the file to the backend as well as display the contents on the screen.

17. Edit files before running code
Actor: User making edits
System: UVSimGUI
Goal: Change the inputted code while the software is running.
“Program” panel is selected
A cursor appears, implying editability
A keystroke is inputted
The GUI displays keystrokes in the text box
18. Save edits to the computer
Actor: User making edits
System: UVSimGUI
Goal: Maintain edits made to files between sessions in the software
“Save As” is clicked
A secondary window opens up
A file name is inputted
Select “OK” to confirm
When confirmed, the GUI will store the new file

19. Convert file format to 6-digit words
Actor: User with old file formats
System: UVSimGUI
Goal: Convert older version files into the newer format
4->6 button is clicked
The GUI displays a secondary window with available files for conversion.
The selected file is chosen
The window then prompts for the file to be saved as another name.
For conversion, the software will identify any 4-digit valid words, and convert them to 6-digit.
If there is an invalid line or one that is already 6-digit, conversion is skipped.
The file is saved, and the secondary window is closed

20. Clear output from UI
Actor: User who has too many prompts in the output window
System: UVSimGUI
Goal: Clear the output window for visibility
The “Clear” button is selected
The text field of the output object is cleared

21. View multiple tabs
Actor: User testing multiple programs
System: UVSimGUI / FileTab
Goal: Allow the user to flip between files within the GUI
“Open Program File” is chosen
“Open Program File” is chosen for a second time
Labels identifying the file names are displayed in the “Program” window
The first file is selected by the user
The tab highlights and fills the “Program” window with the file content

Functional Requirements

1: The system will handle and display the write operation (11)
2: The system will handle and prompt the user with the read operation (10)
3: The system shall load program instructions from text files into memory. (20)
4: The system shall ADD a word from memory together with the word in the accumulator and leave it in the accumulator when the opcode is 30
5: The system shall SUBTRACT a word from memory with the word in the accumulator and leave it in the accumulator when the opcode is 31
6: The system shall DIVIDE a word from the accumulator with a word in memory and leave it in the accumulator when the opcode is 32
7: The system shall MULTIPLY a word from the accumulator with a word in memory and leave it in the accumulator when the opcode is 33
8: The system shall handle BRANCH operation with opcodes 40, 41, 42
9: The system shall HALT the program with opcode 43
10: The system shall have an accumulator register to temporarily hold an integer
11: The system will detect overflow of +999999 and -999999 and cut off the highest digit
12: The system will take in a 6-digit number and interpret it by separating into opcode and operand
13: The system shall validate each line of the text file for any syntax errors
14: The system shall store strings 6 digits long with a + or - in front into memory ex: +000000 (21)
15: The system will store the memory in a 250-length array
16: The system shall display an error message if the program contains over 250 instructions
17: The system will handle dividing by zero exceptions by halting the system
18: The system will display the memory in a distinct section in the GUI as addresses and values

19: The system shall have a customize button that brings up a window with options to select colors
20: The system shall let you change the background color to another color by clicking "Primary Color"
21: The system shall let you change the button color to another color by clicking "Secondary color"
22: The system shall remember your color preferences
23: The system shall save any text within the program text box to a file named by the user
24: The system shall open .txt files from anywhere on the user's computer to be displayed in the program text box
25: The system shall be able to open multiple text file windows at the same time
26: The system shall allow editing of files in the UVSim graphical user interface

Nonfunctional Requirements

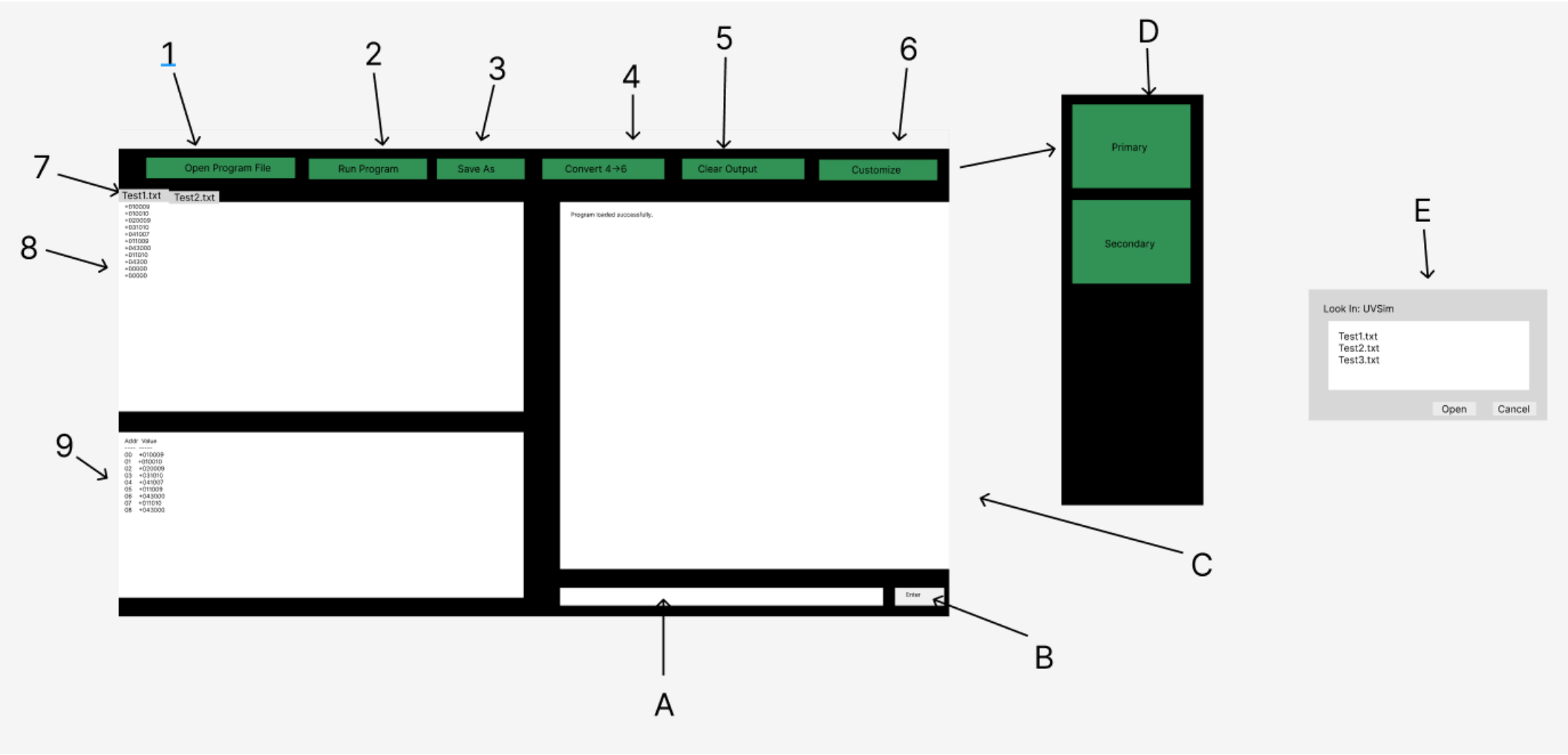
1: The system shall run on a machine with JDK 24 installed
2: The system shall complete operations within 1 second
3: The system will label all GUI widgets
4: The system will handle all error messages by notifying the user without crashing

Application Design and Architecture

GUI Wireframe and Description

To reference throughout our design of the project and user interface, we have created a detailed wireframe with descriptions of how different buttons and sections should behave and present to the user.

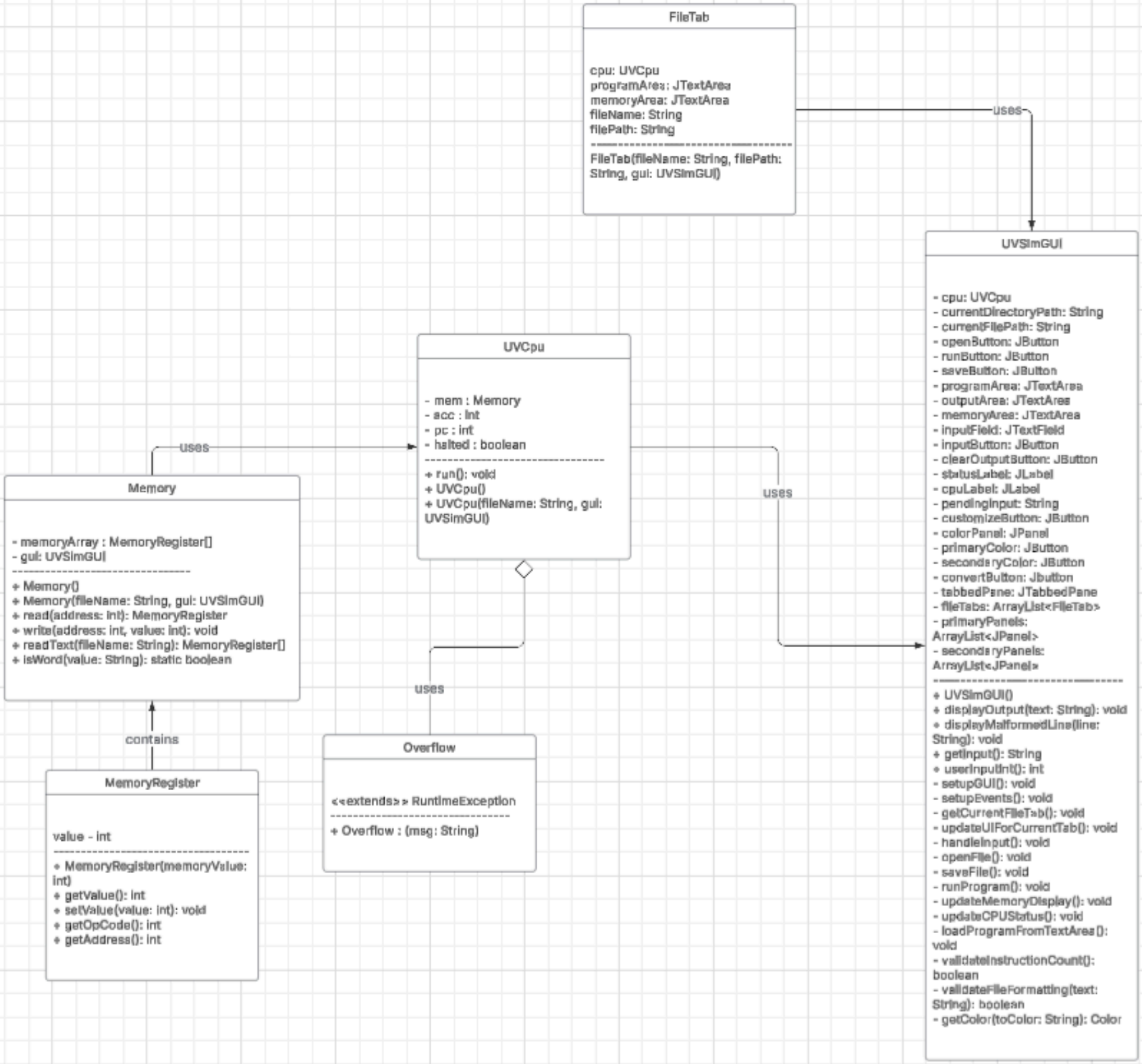
- **1:** Selecting this button will open up a window to select a text file for loading into memory. See the text box on the right for more info.
- **2:** When a file is loaded, this button will be pressable. On press, the program will run all instructions until a halt command. To the right is the name of the file loaded.
- **3:** Selecting this button will allow you to save a file and rename it to your choice.
- **4:** Selecting this allows the user to choose a file for format conversion from the old 4-word style to the new 6-word style.
- **5:** Selecting this button will clear out the output screen to clean up the visuals of it
- **6:** This allows the user to change the colors of the program. There is a primary and a secondary color that can be changed
- **7:** Different tabs can be selected for the user to view any opened file.
- **8:** This text area will display the text file selected with Open Program Files.
- **9:** This is a visual representation of the Memory within the UVSim CPU.
- **A:** This is where input of an integer, -999999 to 999999 only, will be placed. After something is inputted, the Enter button is able to be pressed.
- **B:** Press this button when the desired input is complete
- **C:** Any user prompts or errors with file parsing will display in this section
- **D:** This is the customization menu once “Customize” is pressed. When a new color is selected, after confirmation in the color selector, the color will change.
- **E:** When Open Program File or Convert is selected, a file explorer will open up. Select the file from the list, or access the correct file from the explorer, and press Open to load


1
2
3
4
5
6
7
8
9
A
B
C
D
E
Open Program File
Run Program
Save As
Convert 4→6
Clear Output
Customize
Test1.txt
Test2.txt
Program loaded successfully.
Addr Value
Enter
Primary
Secondary
Look In: UVSim
Test1.txt
Test2.txt
Test3.txt
Open
Cancel

Class Diagram

The backend of the current application is comprised of four different components: CPU, memory, memory registers, and overflow detection. With OOP in mind, the different objects have been divided into their non-simulated counterparts. The Memory class holds individual memory registers defined by MemoryRegister. This memory is included with a custom exception class to comprise the UVCpu. The CPU itself also holds other pieces, such as an accumulator and a pointer to the memory, which can be represented by a basic integer.

The frontend, or GUI, of the application is mostly maintained by the class UVSimGUI. The GUI is built in Swing, a GUI toolkit written in Java. The number of available components and customization with Swing has allowed for a nuanced and rapid implementation of our desired design over the last few months. To enable users to tab between files, a class called FileTab contains all the necessary parts for each individual file, which is then managed by the GUI. On the next page is more details about how the different components interact behind the scenes.


FileTab
cpu: UVCpu
programArea: JTextArea
memoryArea: JTextArea
fileName: String
filePath: String
FileTab(fileName: String, filePath: String, gui: UVSimGUI)
uses
UVSimGUI
- cpu: UVCpu
- currentDirectoryPath: String
- currentFilePath: String
- openButton: JButton
- runButton: JButton
- saveButton: JButton
- programArea: JTextArea
- outputArea: JTextArea
- memoryArea: JTextArea
- inputField: JTextField
- inputButton: JButton
- clearOutputButton: JButton
- statusLabel: JLabel
- cpuLabel: JLabel
- pendingInput: String
- customizeButton: JButton
- colorPanel: JPanel
- primaryColor: JButton
- secondaryColor: JButton
- convertButton: Jbutton
- tabbedPane: JTabbedPane
- fileTabs: ArrayList<FileTab>
- primaryPanels: ArrayList<JPanel>
- secondaryPanels: ArrayList<JPanel>
+ UVSimGUI()
+ displayOutput(text: String): void
+ displayMalformedLine(line: String): void
+ getInput(): String
+ userInputInt(): int
- setupGUI(): void
- setupEvents(): void
- getCurrentFileTab(): void
- updateUIForCurrentTab(): void
- handleInput(): void
- openFile(): void
- saveFile(): void
- runProgram(): void
- updateMemoryDisplay(): void
- updateCPUStatus(): void
- loadProgramFromTextArea(): void
- validateInstructionCount(): boolean
- validateFileFormatting(text: String): boolean
- getColor(toColor: String): Color
UVCpu
- mem : Memory
- acc : int
- pc : int
- halted : boolean
+ run(): void
+ UVCpu()
+ UVCpu(fileName: String, gui: UVSimGUI)
uses
uses
Memory
- memoryArray : MemoryRegister[]
- gui: UVSimGUI
+ Memory()
+ Memory(fileName: String, gui: UVSimGUI)
+ read(address: int): MemoryRegister
+ write(address: int, value: int): void
+ readText(fileName: String): MemoryRegister[]
+ isWord(value: String): static boolean
uses
contains
MemoryRegister
value - int
+ MemoryRegister(memoryValue: int)
+ getValue(): int
+ setValue(value: int): void
+ getOpCode(): int
+ getAddress(): int
Overflow
<<extends>> RuntimeException
+ Overflow : (msg: String)

Future Road Map

As UVSim continues to grow, there are several directions we'd like to explore in future versions. With additional development time or follow-on projects, these are some of the improvements we'd aim to implement:

1. **Built-In Tutorials**

UVSim is already a great way for students to understand how opcodes and operands work at a low level. To take that further, we'd like to create a built-in tutorial system. A guided, interactive experience that walks students through small example programs step by step. It would explain each instruction and show how it changes the accumulator and memory in real time.

2. **Visual Diagram**

To help students understand control flow, we're considering a live visual diagram that tracks how a program executes. It would highlight jumps, loops, and branches as they happen, giving students a better sense of how instructions affect the program counter and overall logic.

3. **Web/Mobile Versions**

Right now, UVSim runs as a desktop Java application. In the future, we'd love to create mobile and browser-based versions of the app. This would let students run UVSim on phones, tablets, or directly in a browser. Making it more accessible across platforms and devices.

4. **Auto-Saving**

Currently, users can save their programs manually, but if their system crashes mid-session, they'd lose all progress. Adding an auto-save or crash recovery feature would prevent that, helping users avoid data loss during unexpected interruptions.

5. **More Export Options**

At the moment, UVSim only supports exporting files as '.txt'. We'd like to support other formats like JSON or CSV, which could make it easier for instructors to analyze program behavior or integrate UVSim outputs with grading tools or other systems.

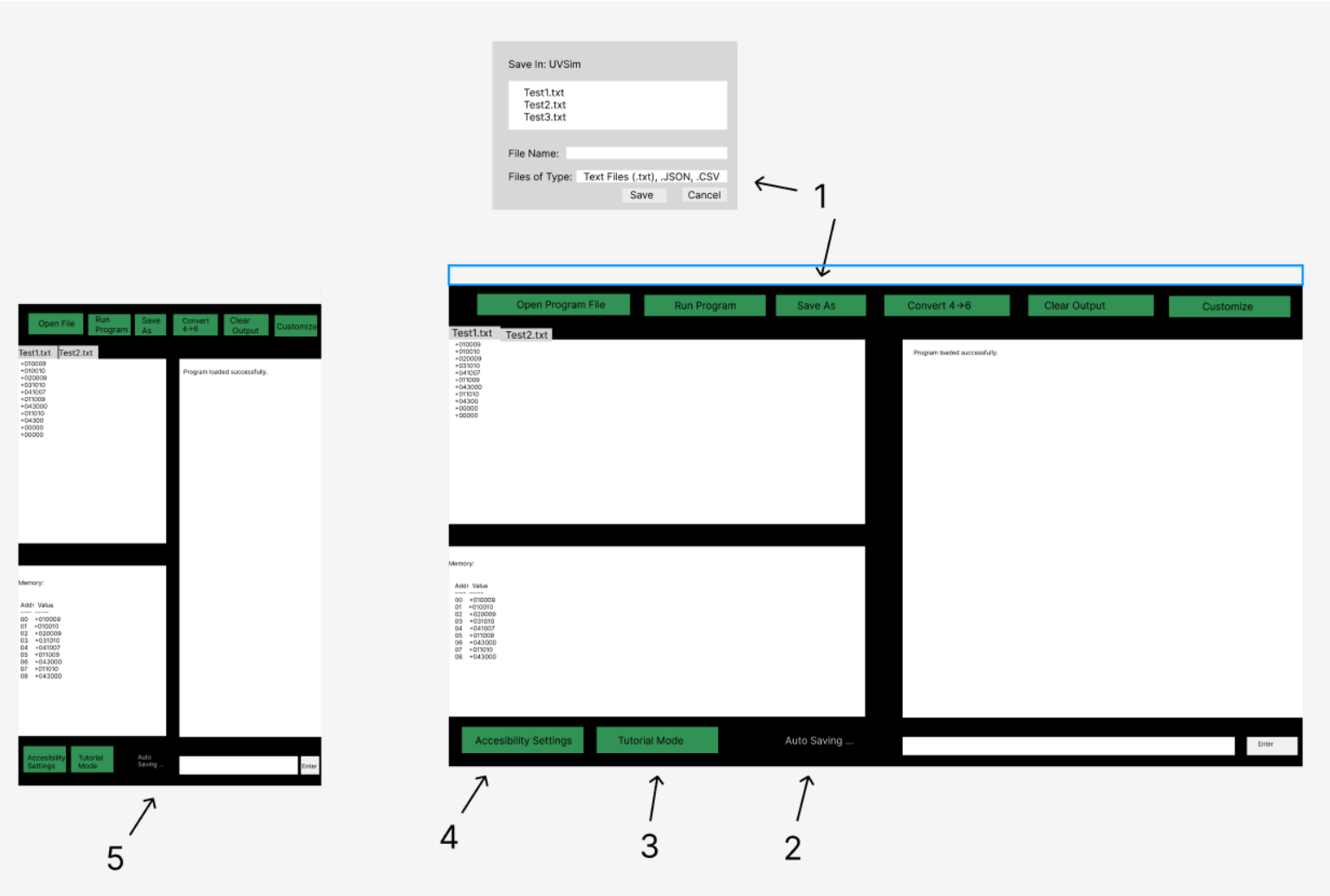
6. **Accessibility Settings**

UVSim could benefit from better accessibility support. Some ideas we've discussed include high-contrast themes, larger default font sizes, built-in dark mode, and visual filters for users prone to migraines or seizures. These settings would make UVSim more inclusive and easier to use for a wider range of students.

Future Visual Mock-Up

In order to visualize our plans better, we have created a wireframe with some of the different possibilities for the future. Each feature is numbered and described below:

1. When saving, you would now get the option to save as a .JSON or .CSV
2. Auto Save Feature to prevent crashing when editing a file
3. Selecting Tutorial Mode would bring up a guide to walk you through every button and screen (exactly like how the wireframe is explaining this now)
4. These will be the accessibility settings where you can adjust visuals/colors
5. Possible design for a mobile version



Application Instructions

Setup

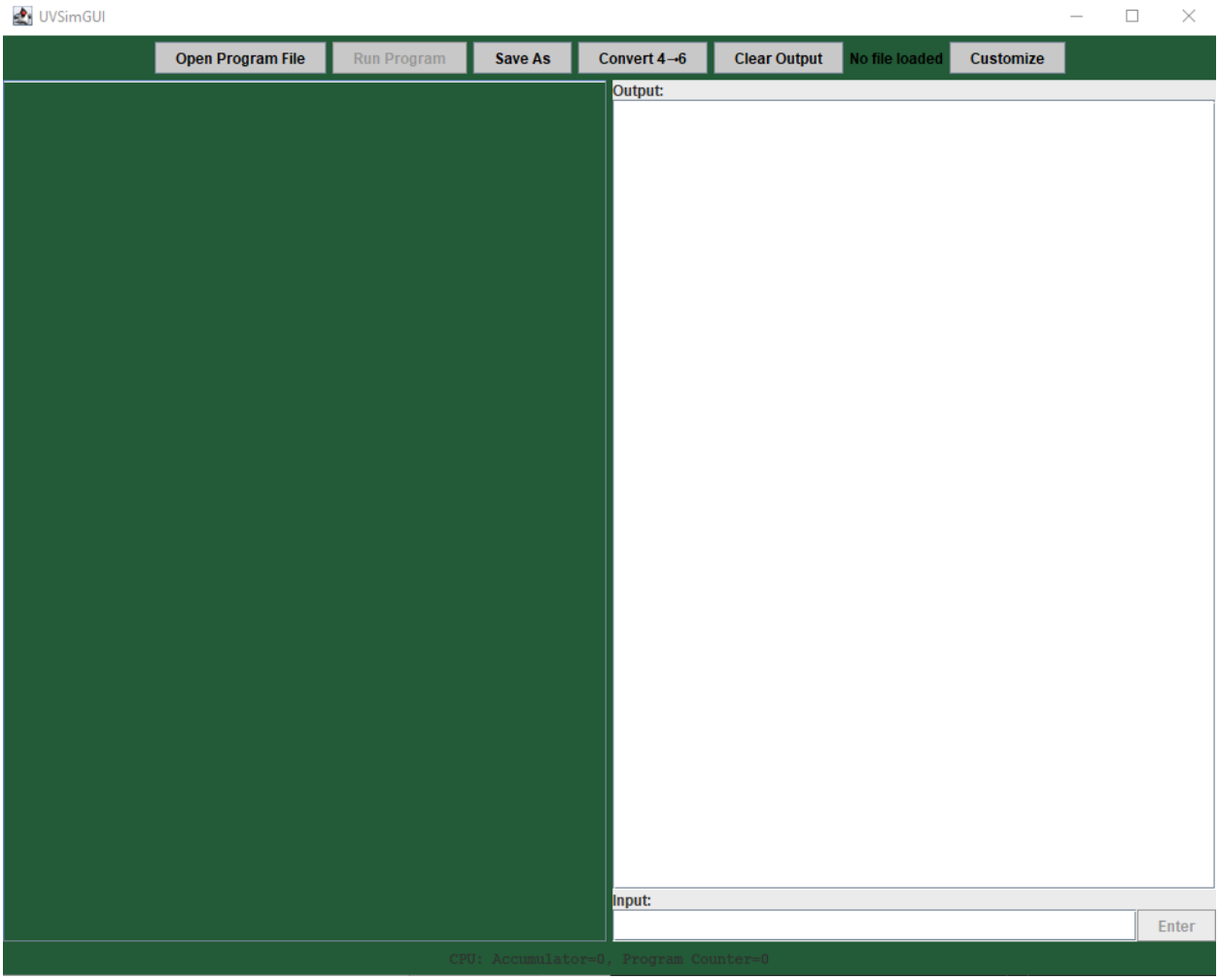
PREREQUISITES:

- Java JDK 8 or higher
- JUnit 4 (for testing)

COMPILATION: The application has been precompiled into a jar file that is within the repo. The file named "UVSim.jar" is what you need to run the program.

RUNNING THE APPLICATION:

1. Ensure **UVSim.jar** and any **.txt** BasicML files are in the same folder.
2. Run the program:
    - Double-click **UVSim.jar**, or
    - Use terminal: **java -jar UVSim.jar**
3. The GUI will launch.



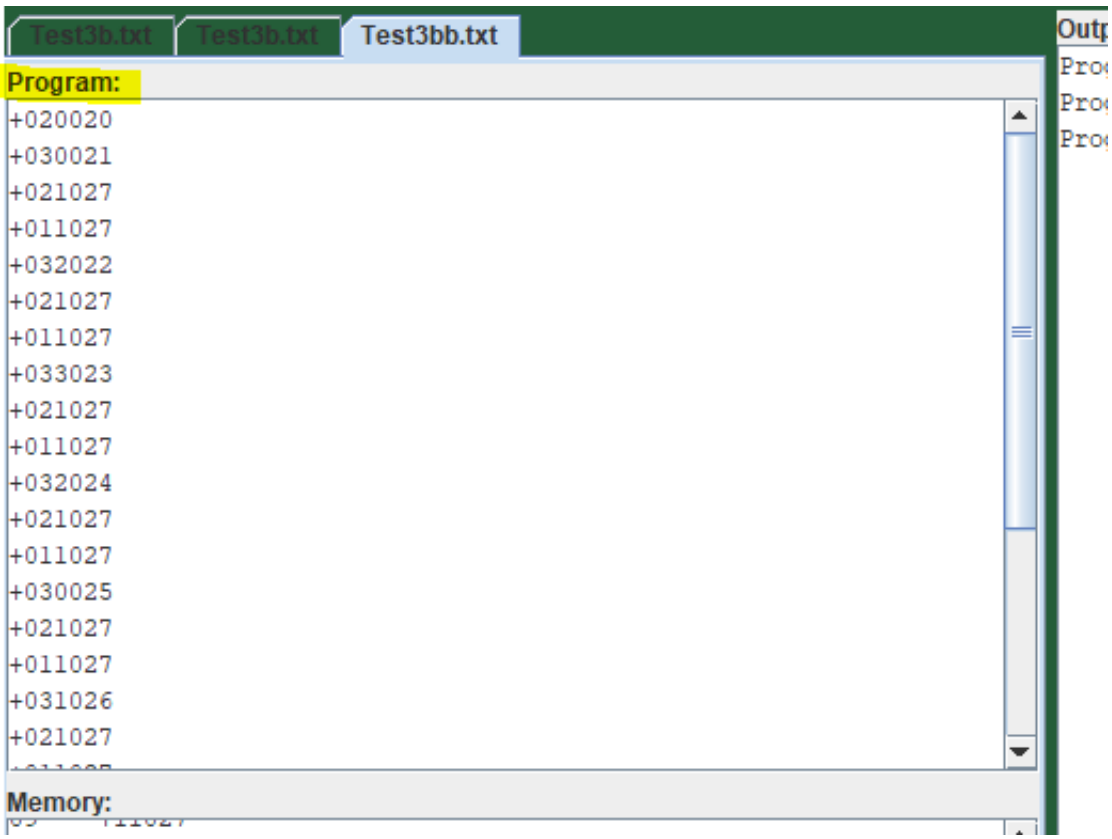
GUI Overview

Top Buttons

- **Open Program File**: Load a .txt BasicML file from any file system.
- **Run Program**: Execute loaded or edited instructions via file selected by user.
- **Convert 4->6**: Converts a .txt file with 4 digit words to one with 6 digits instead. Select the file that you would like to change, and then after selected save it as any other name to convert it. To run the converted file, you will need to reopen the file with the Open button.
- **Save As**: Save the current program (with format validation).
- **Clear Output**: Clears the output pane.
- **Customize**: Opens color selection tools (primary/secondary).

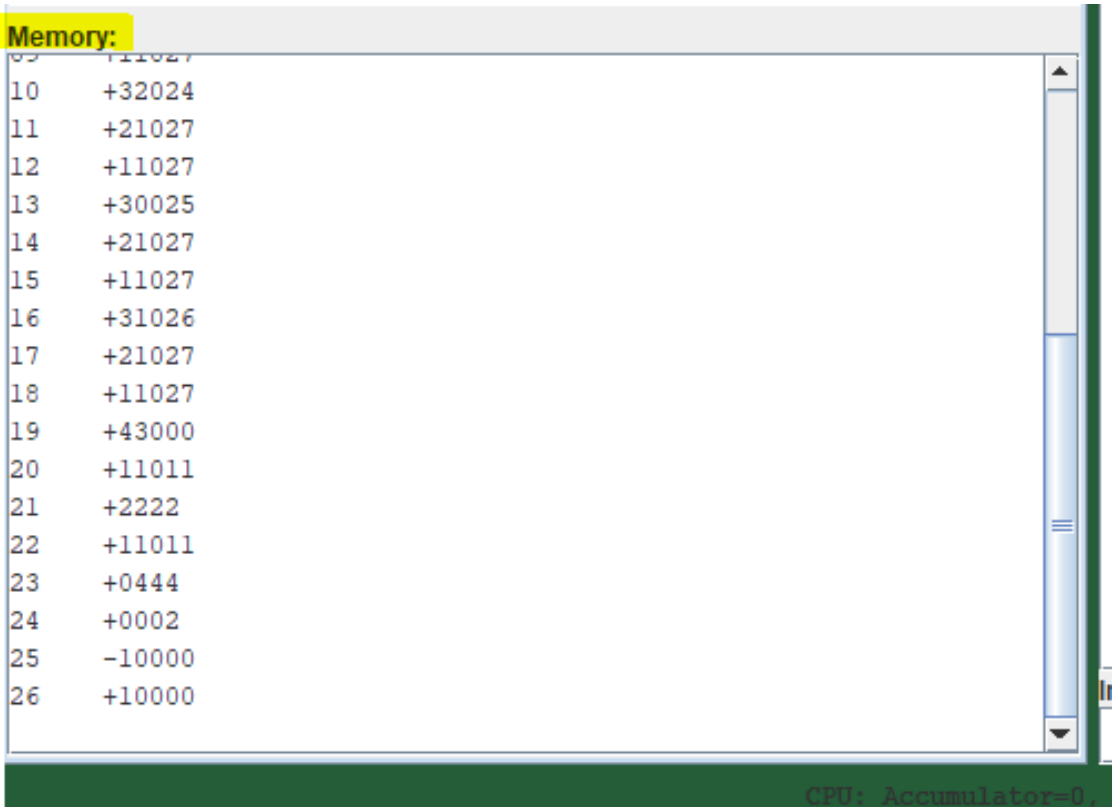
Program Pane

This is displayed on the top left-hand side of the GUI, after a file has been selected. Within this specific pane, the file is editable before running the program. To save the edits, select the "Save As" button in the top section of the GUI. In addition to allowing edits, the user can also open multiple files at one time. To swap between files, there are labels above the program pane that can be selected. On selection, the chosen file will be displayed.



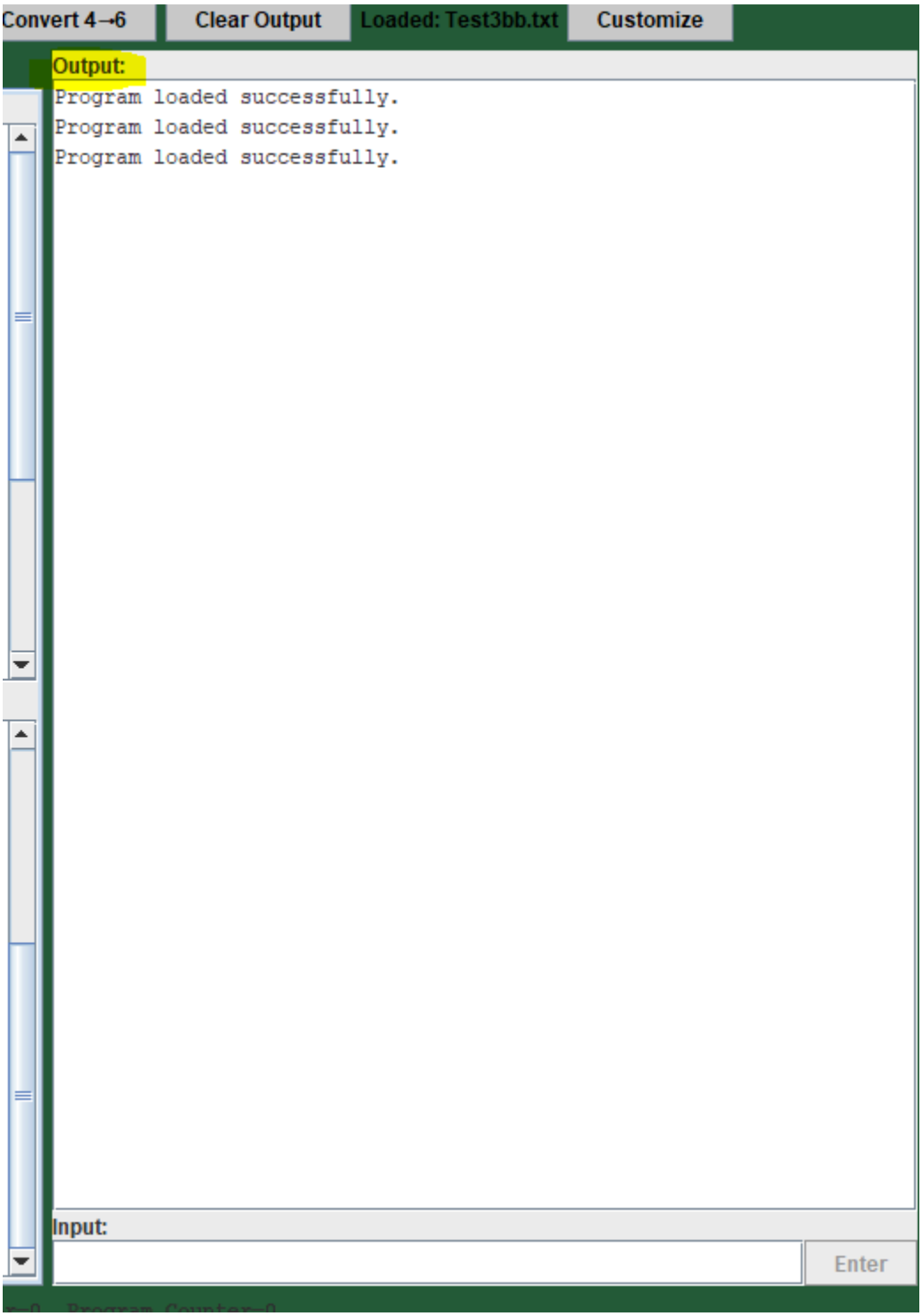
Memory Pane

This is displayed on the bottom left-hand side of the GUI, after a file has been selected. It is a visual representation of the virtual memory, storing all the code above it. As the program runs, the memory will update in real time based on any inputs of the user.



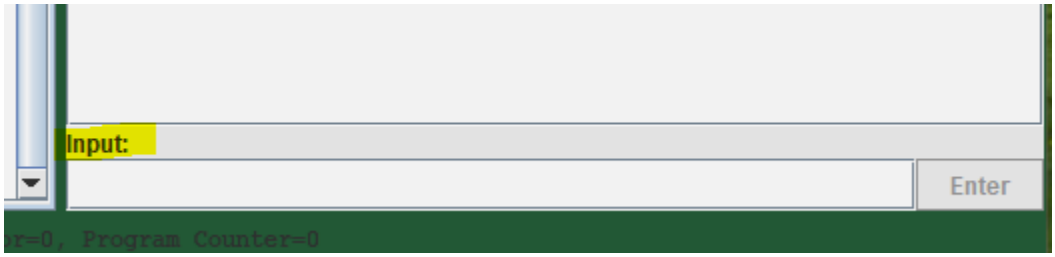
Output Pane

This is displayed on the right-hand side of the GUI. Displays output from write instructions and other system messages.



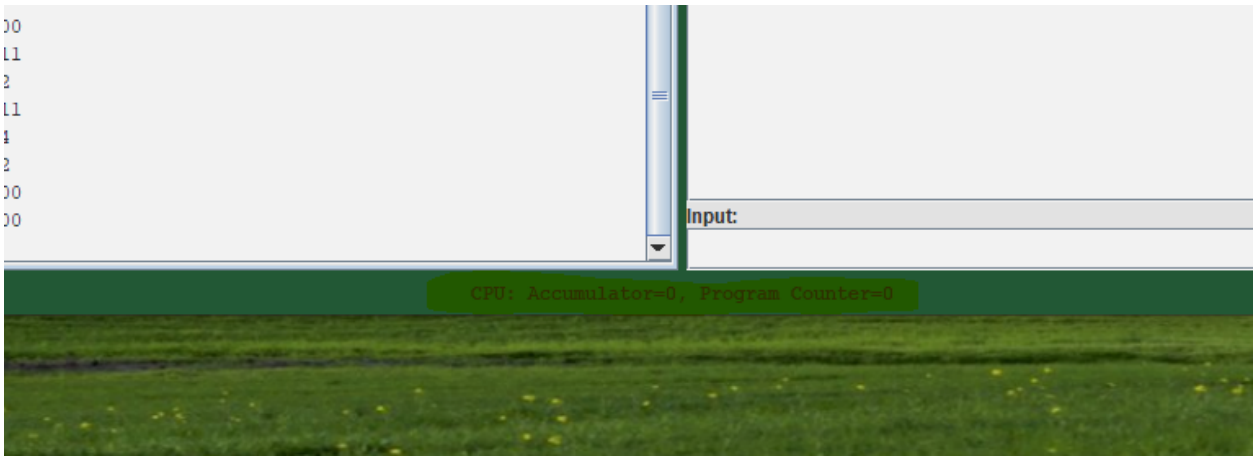
Input Field

Underneath the GUI is the area where users can input based on what their code actually does. Prompts for user input will display above in the output pane.



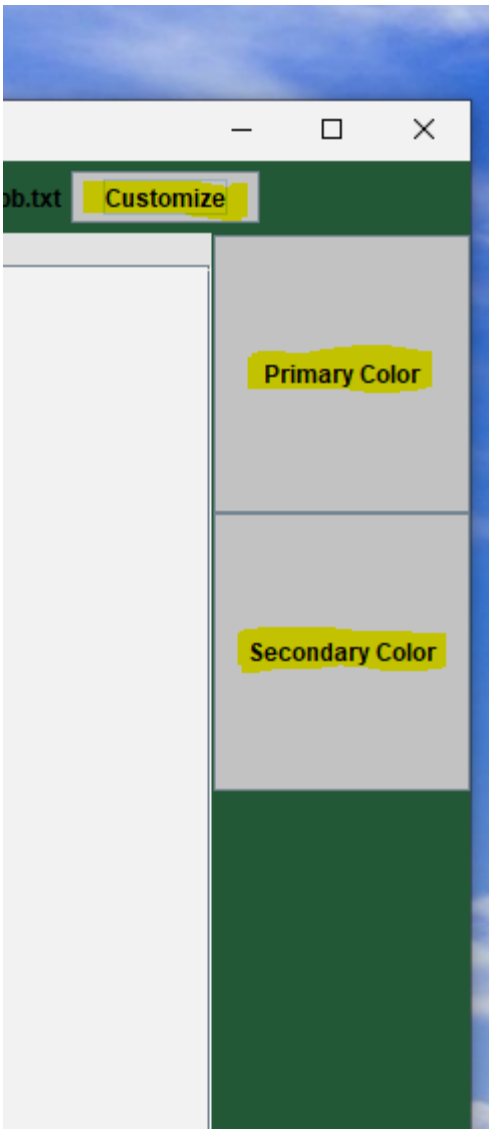
CPU Status

Shows the current accumulator and the program counter values in real time.



Customize Pane

On the far right of the top panel is a button labeled "Customize." On selection, the GUI will extend to have two additional buttons, explained below



- **Primary Color**: Background blank space can be modified to any selected color based on the choice of the user
- **Secondary Color**: All buttons' background color can be modified based on the choice of the user. When these buttons are selected, a secondary window will display with several ways for the user to change the color of their GUI The color will change as soon as the choice is confirmed in this secondary window by the user. The new preference will be kept between launches of the GUI, so the user doesn't need to select their preference every time.

Current Unit Tests

Name	Description	Inputs	Outputs	Succeed/Fail
testLoadOperation	Writes numbers to memory, tests for correct value	2005, 4300, 1234	1234	Succeed if True
testLoadNegativeValue	Tests a negative value	2005, 4300, -5678	-5678	Succeed if True
testStoreOperation	Tests that a value gets stored after commands	2105, 4300, 9876	9876	Succeed if True
testStoreNegativeValue	Tests that a negative value gets stored after commands	2105, 4300, -4321	-4321	Succeed if True
testWriteOperation	Tests that a number gets written to console after commands	1105, 4300, 1234	1234	Succeed if True
testWriteNegativeValue	Tests that a negative number gets written to console after instructions	1105, 4300, -9876	-9876	Succeed if True
testBranchOperation	Tests that a branch happens at branch instruction	4005, 2099, 4300	6	Succeed if True and cpu is halted
testBranchNegativeWhenAccumulatorNegative	Tests that the branch happens when accumulator is negative	4105, 2099, 4300, -100	6	Succeed if True and cpu is halted
testBranchNegativeWhenAccumulatorPositive	Tests that a branch doesn't happen when accumulator is positive	4105, 4300, 2099, 100	2	Succeed if True and cpu is halted

Name	Description	Inputs	Outputs	Succeed/Fail
testBranchZeroWhenAccumulatorZero	Tests that a branch happens when Accumulator is zero	4205, 2099, 4300, 0	6	Succeed if True and cpu is halted
testBranchZeroWhenAccumulatorNonZero	Tests that a branch doesn't happen when Accumulator is zero	4205, 4300, 2099, 42	2	Succeed if True and cpu is halted
testHaltOperation	Tests that the halt instruction halts the program	4300, 2099	1	Succeed if True and cpu is halted
testInvalidMemoryAccess	Tests that loading a memory address that hasn't been written to yet loads a 0 into acc	2099, 4300	0	Succeed if True
testProgramCounterIncrement	Asserts pc is 3 after instructions, and acc is 300	2005, 3006, 4300, 100, 200	3, 300	Succeed if True
testAdditionPositive	Tests to see if cpu addition is the same as expected addition	3005, 4300, 10, 5	15	Succeed if True
testAdditionNegative	Tests to see if cpu negative addition is the same as expected negative addition	3005, 4300, -20, 10	-10	Succeed if True
testSubtractionPositive	Tests to see if cpu subtraction is the same as expected subtraction	3105, 4300, 4, 10	6	Succeed if True
testSubtractionNegativeResult	Tests to see if cpu negative subtraction is the same as expected negative subtraction	3105, 4300, 20, 5	-15	Succeed if True

Name	Description	Inputs	Outputs	Succeed/Fail
testDivisionResult	Tests to see if cpu division is the same as expected division	3205, 4300, 2, 10	5	Succeed if True
testDivisionByZeroThrows	Tests that an error is thrown when attempting to divide by zero	3205, 4300, 0, 10	fail	Succeed if Error thrown
testMultiplicationPositive	Tests that the cpu multiplies correctly when both multipliers are positive	3305, 4300, 4, 3	12	Succeed if True
testMultiplicationByZero	Tests that the cpu multiplies correctly when multiplicand is zero	3305, 4300, 0, 25	0	Succeed if True
read	Tests the read function in the memory class accesses the correct index	N/A	1009, 0	Succeed if True
write	Tests the write function in the memory class goes to the correct index	5,555	5555	Succeed if True
readTextFail	Checks to see if memory is still empty if bad file is given	Test0.txt	[0, 100 times]	If memory is equal to an empty array of 100 zeros