UVSim

Andrew Buckland, Jaykant Lekhi, Jonah Cragun, Zac Peterson

UVU

CS 2450

Professor Burtt

9/9/24

Table of Contents

Introduction

UVSim is a simple virtual machine used to help users execute their own machine language basicML code. It contains a CPU, register, and main memory. An accumulator – a register into which information is put before the UVSim uses it in calculations or examines it in various ways. All the information in the UVSim is handled in terms of words. A **word** is a signed six-digit decimal number, such as +123400, -567800. The UVSim is equipped with a 250-word memory, and these words are referenced by their location numbers 000, 01, ..., 249. The BasicML program must be loaded into the main memory starting at location 00 before executing. Each instruction written in BasicML occupies one word of the UVSim memory (instruction are signed six-digit decimal number). We shall assume that the sign of a BasicML instruction is always plus, but the sign of a data word may be either plus or minus. Each location in the UVSim memory may contain an instruction, a data value used by a program or an unused area of memory. The first three digits of each BasicML instruction are the operation code specifying the operation to be performed.

The project's success relied heavily on understanding and balancing priorities. Effective communication among the team members ensured that each aspect of the project was managed efficiently, for development to deployment. The team's ability to address risks and adapt to challenges was crucial in delivering a functional and educational tool.

The development of UVsim was a collaborative and iterative process, involvement of team members and dedication of the entire team. By integrating diverse skills and utilizing appropriate resources and tools, the team creates a valuable resource for learning computer architecture and programming principles.

We would like to extend our sincere gratitude to everyone who supported and contributed to this project, including our educators and peers. Your feedback, guidance, and encouragement were invaluable in helping us achieve our educational goals and deliver a functional and innovative tool. Thank you for your collaboration and support throughout this journey.

User Stories

User Story 1

- As a student, I want to execute machine language programs so that I can learn the language BasicML.

User Story 2

- As an educational company, I want to have good learning tools so that our students can be taught effectively.

Use Cases

Use Case 1

- **ACTOR**: ADD algorithm
- **SYSTEM**: Arithmetic System
- **GOAL**: Add the number from the instructions provided memory address to the number that is currently in the accumulator

STEPS:

1. Get number in provided memory address
2. Add the number to the number in the accumulator

Use Case 2

- **ACTOR**: SUBTRACT algorithm
- **SYSTEM**: Arithmetic System
- **GOAL**: Subtract the number in the provided memory address from the number in the accumulator.

STEPS:

1. Get number in provided memory address
2. Subtract the number from the number in the accumulator

Use Case 3

- **ACTOR**: BRANCH algorithm
- **SYSTEM**: Control System

- **GOAL**: Branch to the instruction at the provided memory location

STEPS:

1. Set the current pointer equal to the provided memory address.

Use Case 4

- **ACTOR**: Branch Negative
- **SYSTEM**: Control System
- **GOAL**: Branch to the instruction at the provided memory location if the accumulator is negative.

STEPS:

1. Check if the provided memory address (br_target) is within the valid range.
2. If the accumulator is negative, set the current pointer to the provided memory address.
3. If the accumulator is not negative, increment the current memory address and set it as the current pointer.

Use Case 5

- **ACTOR**: Branch Zero
- **SYSTEM**: Control System
- **GOAL**: Branch to the instruction at the provided memory location if the accumulator is zero.

STEPS:

1. Check if the provided memory address (br_target) is within the valid range.
2. If the accumulator is zero, set the current pointer to the provided memory address.
3. If the accumulator is not zero, increment the current memory address and set it as the current pointer.

Use Case 6

- **ACTOR**: Store Algorithm
- **SYSTEM**: Memory Management System
- **GOAL**: Store the value in the accumulator to a specific memory location.

STEPS:

1. Check if the provided memory address (mem_addr) is within the valid range.
2. Store the value in the accumulator at the specified memory address.

Use Case 7

- **ACTOR**: Read Algorithm
- **SYSTEM**: I/O System
- **GOAL**: Read in a value from the UI into a specified location in memory.

STEPS:

1. Check that memory location is within a valid range.
2. Create input dialogue box.
3. Wait for user input.

4. Store value into specified memory location after the enter key has been read.

Use Case 8

- **ACTOR**: Write Algorithm
- **SYSTEM**: I/O System
- **GOAL**: Write value to the console from specified memory location.

STEPS:

1. Check that memory location is within a valid range.
2. Get value from specified memory address.
3. Write value to the terminal.

Use Case 9

- **ACTOR**: Halt Algorithm
- **SYSTEM**: Control System
- **GOAL**: Stop the program from running.

STEPS:

1. Return a value greater than the size of addressable memory.

Use Case 10

- **ACTOR**: Load Algorithm
- **SYSTEM**: Memory Management System

- **GOAL**: Load the value from a specific memory location into the accumulator.

STEPS:

1. Check if the provided memory address(mem_addr) is within the valid range.
2. Load the value from the specified memory address into the accumulator

Use Case 11

- **ACTOR**: Divide Algorithm
- **SYSTEM**: Arithmetic System
- **GOAL**: Divide the number in the accumulator by the number at the provided memory address.

STEPS:

1. Get the number from the provided memory address.
2. If the number is zero, throw a division by zero error.
3. Divide the number in the accumulator by the number from the provided memory address.

Use Case 12

- **ACTOR**: Multiply Algorithm
- **SYSTEM**: Arithmetic System
- **GOAL**: Multiply the number in the accumulator by the number at the provided memory address.

STEPS:

1. Get the number from the provided memory address.
2. Multiply the number in the accumulator by the number from the provided memory address.

Use Case 13

- **ACTOR**: Read File
- **SYSTEM**: Input System
- **GOAL**: Read in a file and store the instructions

STEPS:

1. Check if the file exists. if not, throw an error.
2. Read in an input file and collect each line.
3. During each iteration of line collection, confirm that the line is a valid input.
4. Store the valid input into the memory array.

Use Case 14

- **ACTOR**: User
- **SYSTEM**: UI System
- **GOAL**: Pick a file to be read

STEPS:

1. Click import instructions.
2. Click file to be imported.
3. Click open.

Use Case 15

- **ACTOR**: User

- **SYSTEM**: UI System

- **GOAL**: Save instructions to a file

STEPS:

1. Click save instructions to file.
2. Find directory where you want to save.
3. Name the file.
4. Click Save.

Use Case 16

- **ACTOR**: User

- **SYSTEM**: UI System

- **GOAL**: Edit currently saved instructions

STEPS:

1. Double click an instruction.
2. Edit the instruction to a 6-digit signed number.
3. Click enter.

Use Case 17

- **ACTOR**: User

- **SYSTEM**: UI System

- **GOAL**: Use Cut/Copy/Paste functionality

STEPS:

1. Double click an instruction.

2. Use your machines Cut/Copy/Paste shortcut.

Use Case 18

- **ACTOR**: User

- **SYSTEM**: UI System

- **GOAL**: Change UI Colors

STEPS:

1. Click Settings Button.

2. Enter RBG values of colors you would like.

Use Case 19

- **ACTOR**: User

- **SYSTEM**: UI System

- **GOAL**: Add Tabs

STEPS:

1. Click the plus tab to add another tab on the UI
2. Click the X button to remove a tab

Use Case 20

- **ACTOR**: Instruction Conversion
- **SYSTEM**: Input Handler
- **GOAL**: Convert 4-digit instruction to 6-digit instruction

STEPS:

1. Check if the instruction is a 4 or 6 digit number.
2. If it is a 6 digit number, leave it.
3. If it is a 4 digit number, check if it is an instruction.
4. If it is an instruction, add one 0 in front of the op code and one 0 in front of the memory address.
5. If it is not an instruction, add two 0's in front of the numbers.

Unit Tests Spreadsheet

Test Name	Description	Use Case Reference	Inputs	Expected Outputs	Success Criteria
Test 1	Tests addition operation	Use Case 1	None	Correct addition result	Will correctly add the number from the provided memory address to the accumulator
Test 2	Tests out-of-range addition operation	Use Case 1	None	Error thrown	Will correctly throw an error when the memory address is out of range
Test 3	Tests subtraction operation	Use Case 2	None	Correct subtraction result	Will correctly subtract the number from the provided memory address from the accumulator
Test 4	Tests out-of-range subtraction operation	Use Case 2	None	Error thrown	Will correctly throw an error when the memory address is out of range
Test 5	Tests branch operation	Use Case 3	None	Correct branch target	Will correctly return the target memory address
Test 6	Tests out-of-range branch operation	Use Case 3	None	Error thrown	Will correctly throw an error when the memory address is out of range
Test 7	Tests branch negative operation	Use Case 4	None	Correct branch target	Will correctly branch to the target memory address when accumulator is negative
Test 8	Tests out-of-range branch negative operation	Use Case 4	None	Error thrown	Will correctly throw an error when the memory address is out of range

Test 9	Tests branch zero operation	Use Case 5	None	Correct branch target	Will correctly branch to the target memory address when accumulator is zero
Test 10	Tests out-of-range branch zero operation	Use Case 5	None	Error thrown	Will correctly throw an error when the memory address is out of range
Test 11	Tests store operation	Use Case 6	None	Correct store result	Will correctly store the accumulator value to the specified memory address
Test 12	Tests out-of-range store operation	Use Case 6	None	Error thrown	Will correctly throw an error when the memory address is out of range
Test 13	Tests load operation	Use Case 10	None	Correct load result	Will correctly load the value from the specified memory address into the accumulator
Test 14	Tests out-of-range load operation	Use Case 10	None	Error thrown	Will correctly throw an error when the memory address is out of range
Test 15	Tests divide operation	Use Case 11	None	Correct divide result	Will correctly divide the accumulator by the value at the specified memory address
Test 16	Tests divide by zero operation	Use Case 11	None	Error thrown	Will correctly throw an error when attempting to divide by zero
Test 17	Tests multiply operation	Use Case 12	None	Correct multiply result	Will correctly multiply the accumulator by the value at the specified memory address
Test 18	Tests out-of-range multiply operation	Use Case 12	None	Error thrown	Will correctly throw an error when the memory address is out of range
Test 19	Tests read operation	Use Case 7	None	Correct input result	Will correctly read value from stream

Test 20	Tests out-of-range read operation	Use Case 7	None	Error Thrown	Will throw error when index is out of range
Test 21	Tests write operation	Use Case 8	None	Correct output result	Will correctly write value to stream
Test 22	Tests out-of-range write operation	Use Case 8	None	Error Thrown	Will throw error when index is out of range
Test 23	Tests halt operation	Use Case 9	None	Correct returned result	Will correctly return value larger than the size of the memory array to stop program
Test 24	Tests halt operation in execute_op	Use Case 9	None	Correct returned result	Will correctly return value larger than the size of the memory array to stop program
Test 25	Tests reading in a correct file	Use Case 13	None	Correct output result	Will correctly read in the file and execute the instructions
Test 26	Tests reading in a bad file	Use Case 13	None	Error thrown	Will correctly throw an error when reading in a bad file
Test 27	Tests reading in a missing file	Use Case 13	None	Error thrown	Will correctly throw an error when reading in a missing file
Test 28	Tests file load and conversion to six-digit instructions	Use Case 13	None	Successful load and conversion	Will correctly load and convert the file to six-digit instructions

Software Requirement Specification

Functional Requirements

1. The system shall have and manage 250 memory addresses, with each address representing a word (signed six-digit decimal number).
2. The system shall execute instructions sequentially based off their OP CODE.
3. The system shall detect and handle arithmetic operation overflow.
4. The system shall load BasicML programs starting at a predefined location in memory.
5. The system shall handle errors without crashing.
6. The system shall handle reading invalid values.
7. The system shall provide an operation that stores input from the user into memory.
8. The system shall provide an operation that will output a word from memory.
9. The system shall provide an operation that loads a value from memory into the accumulator.
10. The system shall provide an operation that stores a value from the accumulator into memory.
11. The system shall provide an operation that adds a value from memory to the accumulator. 12. The system shall provide an operation that subtracts a value in memory from the accumulator.
13. The system shall provide an operation that divides the value in the accumulator by a value from memory.
14. The system shall provide an operation that multiplies the value in the accumulator by a value from memory.
15. The system shall provide an operation that branches to a specified memory location.

16. The system shall provide an operation that branches to a specified memory location if the accumulator is negative.
17. The system shall provide an operation that branches to a specified memory location if the accumulator is zero.
18. The system shall provide an operation that stops the execution of the program.
19. The system shall provide a way to collect user input and display output.
20. The system shall provide a way to configure the colors of the user interface.
21. The systems default colors shall be UVU Colors.
22. The system shall provide a way for the user to load files and convert them to BasicML programs.
23. The system shall provide a way for the user to view commands that are loaded in.
24. The system shall provide a way for the user to edit commands that are loaded in.
25. The system shall provide cut/copy/paste functionality while the user edits commands.
26. The system shall provide a way to save their commands to a file.
27. The system shall support both old (four-digit) and new (six-digit) file formats, differentiating them at load time.
28. The system shall provide a one-way conversion process from four-digit to six-digit files. 29. The system shall allow users to open and manage multiple files simultaneously within a single application instance, enabling switching between, editing, and executing each file independently.
30. The system shall ensure only one file can be executed at a time, while others can be edited.

Non-functional Requirements

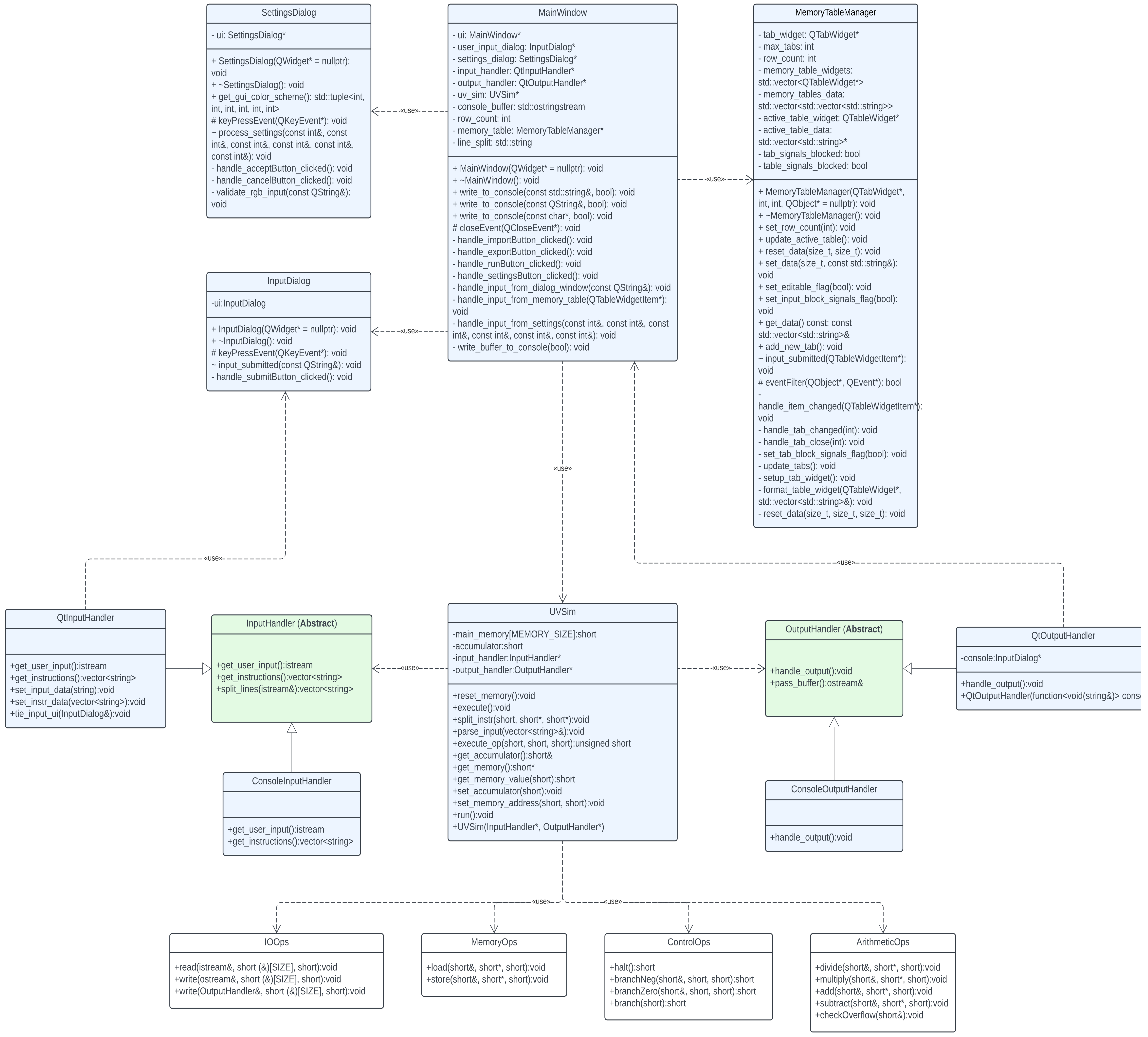
1. The system shall allow users to write and run simplified machine language

2. The system shall execute each instruction within 1 second under normal operational conditions

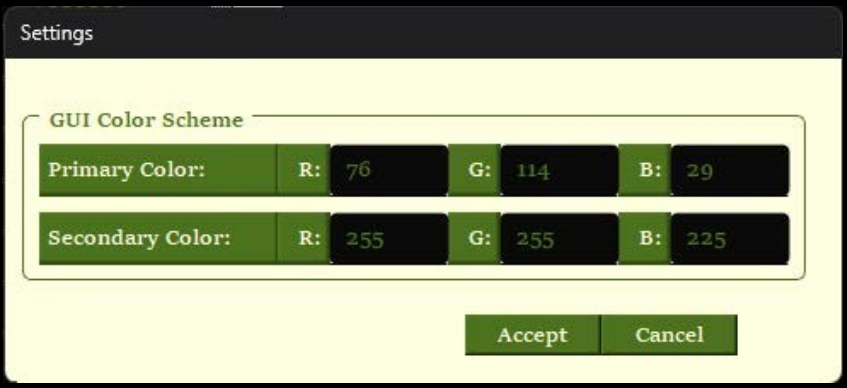
3. The system shall have a user interface that distinguishes each field and its functionality

4. The system shall ensure files loaded or edited do not exceed 250 lines.

5. The system shall validate commands to ensure they reference valid address (000 to 249).


SettingsDialog
- ui: SettingsDialog*
+ SettingsDialog(QWidget* = nullptr): void
+ ~SettingsDialog(): void
+ get_gui_color_scheme(): std::tuple<int, int, int, int, int, int>
# keyPressEvent(QKeyEvent*): void
~ process_settings(const int&, const int&, const int&, const int&, const int&, const int&): void
- handle_acceptButton_clicked(): void
- handle_cancelButton_clicked(): void
- validate_rgb_input(const QString&): void
MainWindow
- ui: MainWindow*
- user_input_dialog: InputDialog*
- settings_dialog: SettingsDialog*
- input_handler: QtInputHandler*
- output_handler: QtOutputHandler*
- uv_sim: UVSim*
- console_buffer: std::ostringstream
- row_count: int
- memory_table: MemoryTableManager*
- line_split: std::string
+ MainWindow(QWidget* = nullptr): void
+ ~MainWindow(): void
+ write_to_console(const std::string&, bool): void
+ write_to_console(const QString&, bool): void
+ write_to_console(const char*, bool): void
# closeEvent(QCloseEvent*): void
- handle_importButton_clicked(): void
- handle_exportButton_clicked(): void
- handle_runButton_clicked(): void
- handle_settingsButton_clicked(): void
- handle_input_from_dialog_window(const QString&): void
- handle_input_from_memory_table(QTableWidgetItem*): void
- handle_input_from_settings(const int&, const int&, const int&, const int&, const int&, const int&): void
- write_buffer_to_console(bool): void
MemoryTableManager
- tab_widget: QTabWidget*
- max_tabs: int
- row_count: int
- memory_table_widgets: std::vector<QTableWidget*>
- memory_tables_data: std::vector<std::vector<std::string>>
- active_table_widget: QTableWidget*
- active_table_data: std::vector<std::string>*
- tab_signals_blocked: bool
- table_signals_blocked: bool
+ MemoryTableManager(QTabWidget*, int, int, QObject* = nullptr): void
+ ~MemoryTableManager(): void
+ set_row_count(int): void
+ update_active_table(): void
+ reset_data(size_t, size_t): void
+ set_data(size_t, const std::string&): void
+ set_editable_flag(bool): void
+ set_input_block_signals_flag(bool): void
+ get_data() const: const std::vector<std::string>&
+ add_new_tab(): void
~ input_submitted(QTableWidgetItem*): void
# eventFilter(QObject*, QEvent*): bool
- handle_item_changed(QTableWidgetItem*): void
- handle_tab_changed(int): void
- handle_tab_close(int): void
- set_tab_block_signals_flag(bool): void
- update_tabs(): void
- setup_tab_widget(): void
- format_table_widget(QTableWidget*, std::vector<std::string>&): void
- reset_data(size_t, size_t, size_t): void
InputDialog
-ui:InputDialog
+ InputDialog(QWidget* = nullptr): void
+ ~InputDialog(): void
# keyPressEvent(QKeyEvent*): void
~ input_submitted(const QString&): void
- handle_submitButton_clicked(): void
«use»
«use»
«use»
«use»
«use»
«use»
QtInputHandler
+get_user_input():istream
+get_instructions():vector<string>
+set_input_data(string):void
+set_instr_data(vector<string>):void
+tie_input_ui(InputDialog&):void
InputHandler (**Abstract**)
+get_user_input():istream
+get_instructions():vector<string>
+split_lines(istream&):vector<string>
ConsoleInputHandler
+get_user_input():istream
+get_instructions():vector<string>
UVSim
-main_memory[MEMORY_SIZE]:short
-accumulator:short
-input_handler:InputHandler*
-output_handler:OutputHandler*
+reset_memory():void
+execute():void
+split_instr(short, short*, short*):void
+parse_input(vector<string>&):void
+execute_op(short, short, short):unsigned short
+get_accumulator():short&
+get_memory():short*
+get_memory_value(short):short
+set_accumulator(short):void
+set_memory_address(short, short):void
+run():void
+UVSim(InputHandler*, OutputHandler*)
«use»
«use»
OutputHandler (**Abstract**)
+handle_output():void
+pass_buffer():ostream&
QtOutputHandler
-console:InputDialog*
+handle_output():void
+QtOutputHandler(function<void(string&)> conso
ConsoleOutputHandler
+handle_output():void
«use»
«use»
IOOps
+read(istream&, short (&)[SIZE], short):void
+write(ostream&, short (&)[SIZE], short):void
+write(OutputHandler&, short (&)[SIZE], short):void
MemoryOps
+load(short&, short*, short):void
+store(short&, short*, short):void
ControlOps
+halt():short
+branchNeg(short&, short, short):short
+branchZero(short&, short, short):short
+branch(short):short
ArithmeticOps
+divide(short&, short*, short):void
+multiply(short&, short*, short):void
+add(short&, short*, short):void
+subtract(short&, short*, short):void
+checkOverflow(short&):void

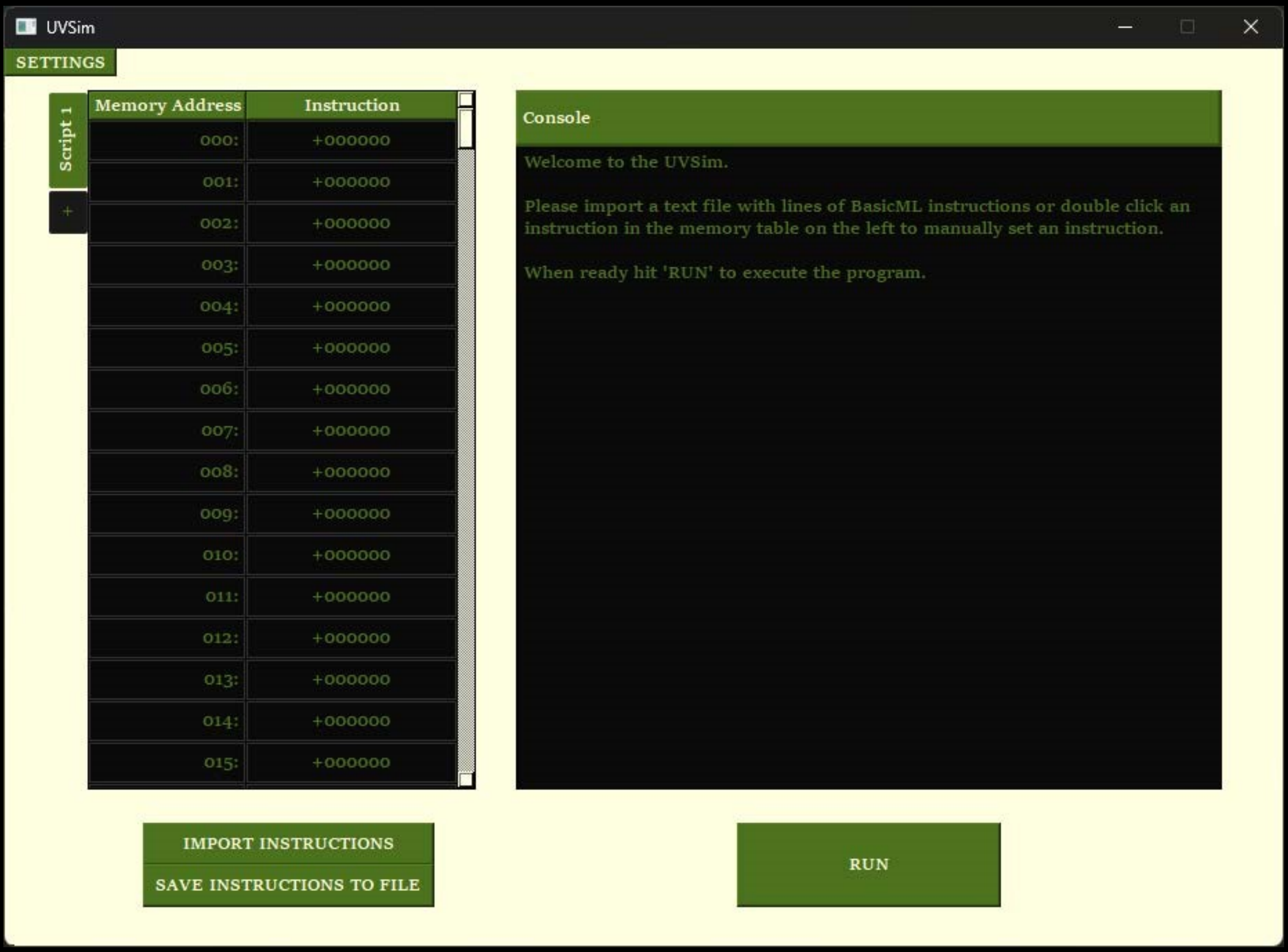
GUI Design



Settings

GUI Color Scheme

Primary Color:	R:	76	G:	114	B:	29
Secondary Color:	R:	255	G:	255	B:	225



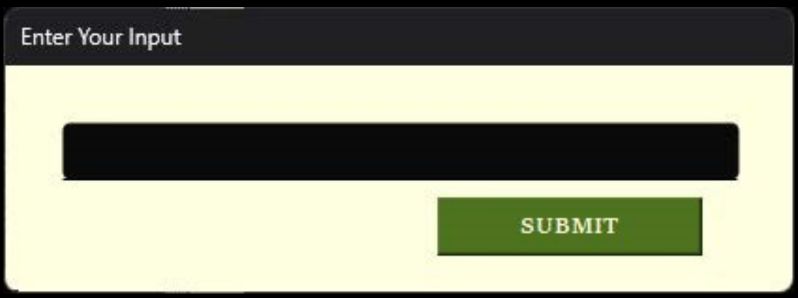
Memory Address	Instruction
000:	+000000
001:	+000000
002:	+000000
003:	+000000
004:	+000000
005:	+000000
006:	+000000
007:	+000000
008:	+000000
009:	+000000
010:	+000000
011:	+000000
012:	+000000
013:	+000000
014:	+000000
015:	+000000

Console

```
Welcome to the UVSim.

Please import a text file with lines of BasicML instructions or double click an
instruction in the memory table on the left to manually set an instruction.

When ready hit 'RUN' to execute the program.
```



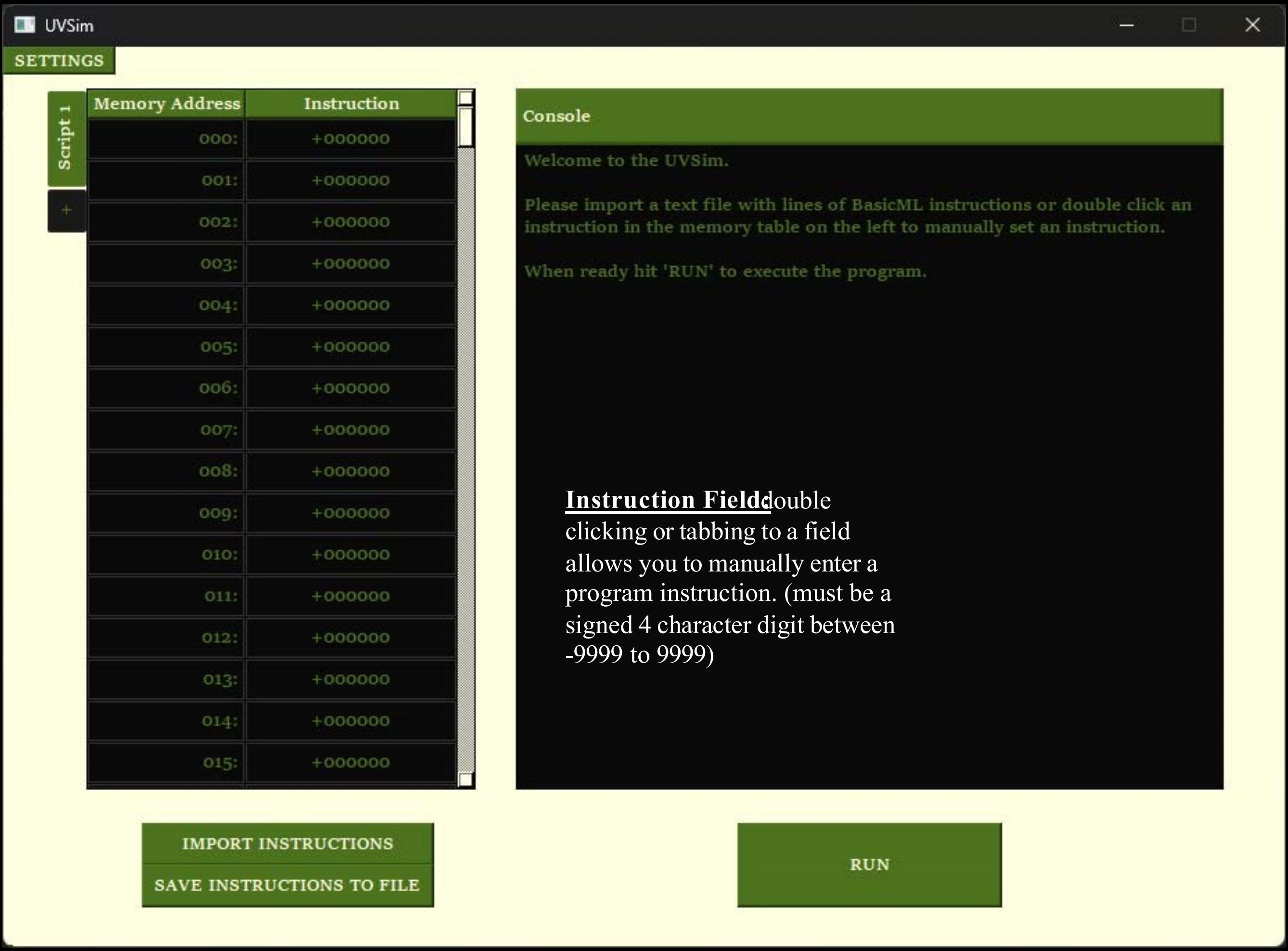
GUI Description

Settings Button: opens the settings window explained below.

Instruction Column: here you can view and edit your program.

Main GUI Window

Output Console: a text scroll field that will display any output (errors, BasicML WRITE outputs, etc.)



Memory Address	Instruction
000:	+000000
001:	+000000
002:	+000000
003:	+000000
004:	+000000
005:	+000000
006:	+000000
007:	+000000
008:	+000000
009:	+000000
010:	+000000
011:	+000000
012:	+000000
013:	+000000
014:	+000000
015:	+000000

Console

```
Welcome to the UVSim.

Please import a text file with lines of BasicML instructions or double click an
instruction in the memory table on the left to manually set an instruction.

When ready hit 'RUN' to execute the program.
```

Instruction Field: double clicking or tabbing to a field allows you to manually enter a program instruction. (must be a signed 4 character digit between -9999 to 9999)

Script Tabs allows for working on multiple instances of instructions. Whichever tab is open determines all other functions.

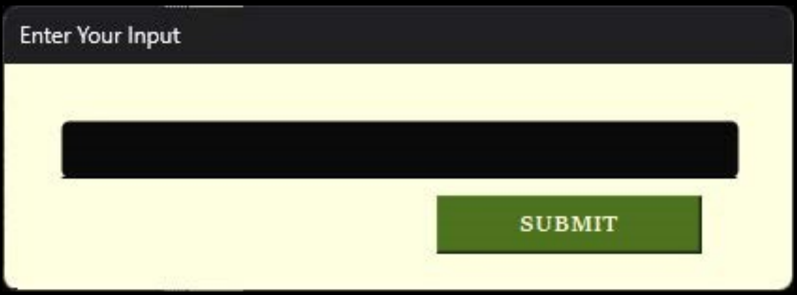
Import Button: opens a file explorer on press. Allowing for the user to find and import a *.txt file containing BasicML instructions.

Save Button: opens a file explorer on press. Allowing for the user to find and export the instructions to a user-named .txt file.

Run Button: executes the program on press.

User Input Window: appears when the program performs the BasicML READ instruction during runtime.

Entry Text Field: input field that expects a signed 4-digit numerical value.

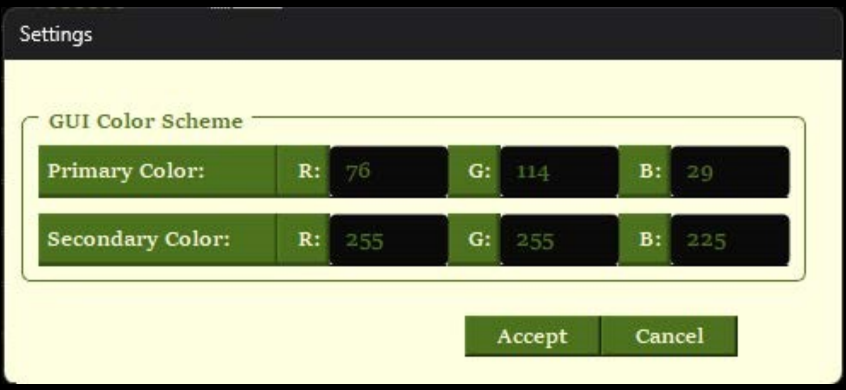


Submit Button: submits entered input to the program.

GUI Description Cont

Settings Window

Color Scheme Settings: adjusting this will change the look across all GUI windows.



Settings

GUI Color Scheme

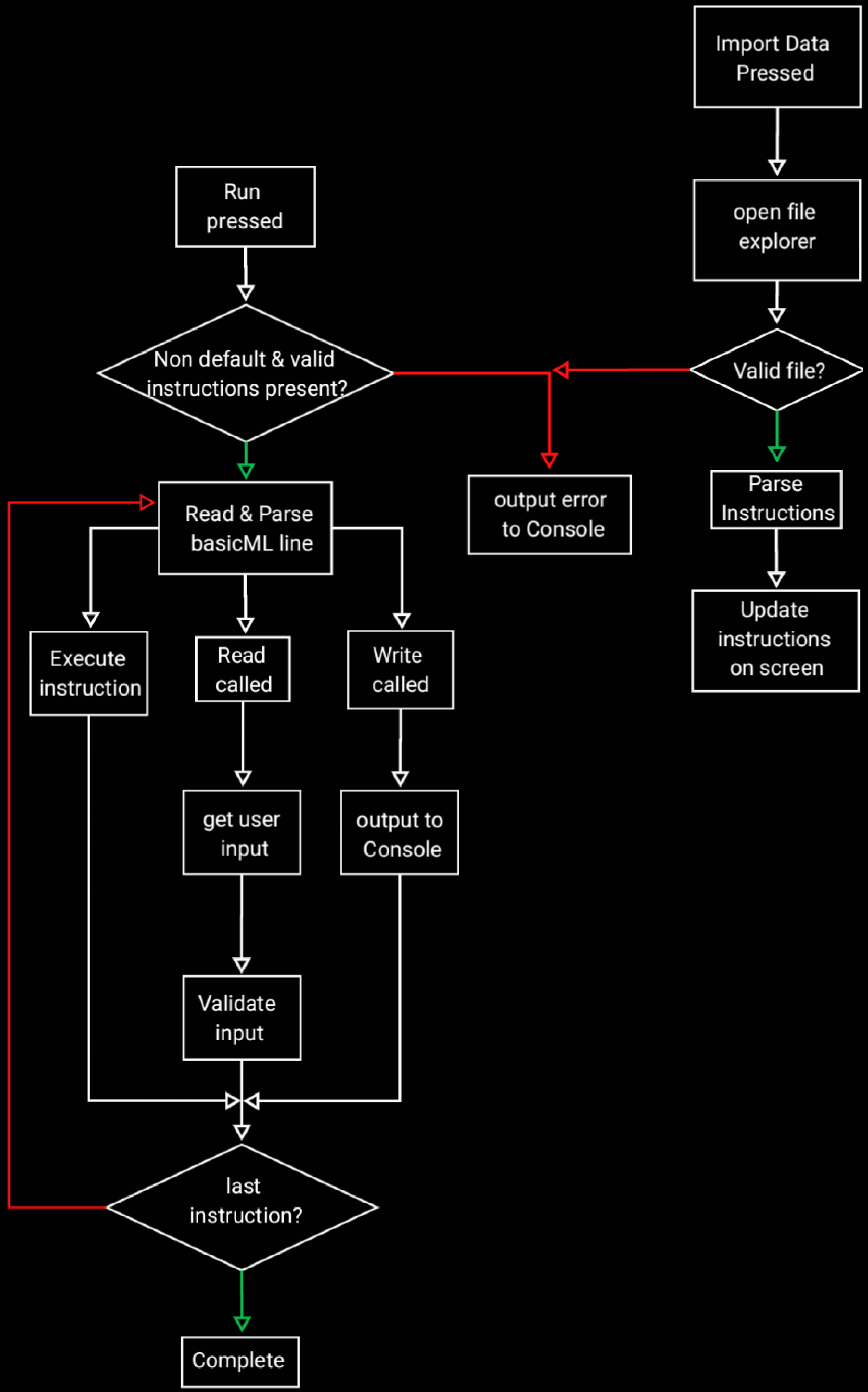
Primary Color:	R:	76	G:	114	B:	29
Secondary Color:	R:	255	G:	255	B:	225

Color Input Field: RGB (Red, Green, and Blue) color values that accept a positive integer between 0 to 255.

Accept Button: applies all changes to the settings.

Cancel Button: discards all changes made in the setting's instance.

Flowchart


Import Data Pressed
open file explorer
Valid file?
Parse Instructions
Update instructions on screen
Run pressed
Non default & valid instructions present?
output error to Console
Read & Parse basicML line
Execute instruction
Read called
Write called
get user input
output to Console
Validate input
last instruction?
Complete

UVSim

UVSim is a simple virtual machine used to help users execute their own machine language basicML code. It contains a CPU, register, and main memory. An accumulator – a register into which information is put before the UVSim uses it in calculations or examines it in various ways.

All the information in the UVSim is handled in terms of words. A **word** is a signed six-digit decimal number, such as +123400, -567800. The UVSim is equipped with a 250-word memory, and these words are referenced by their location numbers 000, 01, ..., 249. The BasicML program must be loaded into the main memory starting at location 00 before executing. Each instruction written in BasicML occupies one word of the UVSim memory (instruction are signed six-digit decimal number). We shall assume that the sign of a BasicML instruction is always plus, but the sign of a data word may be either plus or minus. Each location in the UVSim memory may contain an instruction, a data value used by a program or an unused area of memory. The first three digits of each BasicML instruction are the operation code specifying the operation to be performed.

- Note: This guide focuses on the six-digit architecture, but the old four-digit architecture is still supported. If you have files that were written with four-digit words, uploading them to this virtual machine will automatically convert them to six-digits, then they can be run as normal.

Running the GUI based application

- Clone the repository.

> **Note:** You will need the directory path of where you are placing your clone (or files) of the repository, so it may be worth jotting down the location.

- Create your UVSim's BasicML supplementary file, you can do this by following the instructions in the `Creating Your BasicML File` section below.

For Windows:

1. Open a Windows Command Prompt.

> This can be done by searching for `cmd` in the Windows search bar and clicking on the Command Prompt application.

2. Navigate to the directory where you cloned the repository

> This can be done by using the change directory `cd` command.

```cmd  cd
path\to\repository
```

3. Navigate to the prog folder

4. Execute the file "gui.exe"

5. The program should now have opened in a separate window

Using the GUI

Adding/Modifying Instructions:

1. Locate the large rectangular box on the left side of the screen

> The box contains 2 rows, 1 labeled "Memory Addresses" and the other labeled "Instruction"

2. Locate the memory address where the instruction is to be added

> The program will always start reading instructions at address 0 when run

3. In the instruction column, double click on the cell next to the desired memory address

4. Type a six-digit number representing a value or instruction and hit the "Enter" key when finished

> If an invalid character is entered or a value that is too large, an error message will be printed in the console (Large box on the right side of the screen) and the value inside the instruction cell will not change.

Importing a File:

1. Click the button on the bottom left of the screen labeled "Import Instructions"

> A window should pop up showing your file system

2. Navigate to the directory that contains your file of BasicML instructions

3. Select the desired file, and open it

> The window should now close and instructions from the file should be shown in under the "Instruction" toward the left side of the screen.

4. If errors were found in the file, appropriate error messages will be printed in the console. Resolve the errors to load the file into memory

5. Once the file has been loaded in, you can modify any of the instructions by following the steps under "Adding instructions to memory"

Running Instructions:

1. Load instructions into the virtual memory system following the steps under "Importing a file" or "Adding/modifying instructions"

2. Using the cursor, click the button labeled "Run" found toward the bottom right of the application

3. The instructions should now run starting at memory location 0 and stopping when a halt instruction is read, or the end of memory is reached.

> If a read instruction is run, a popup window will appear and prompt the user for input

4. If input is prompted, type a number (6 digits or fewer) and click the button labeled "submit"

> If an invalid character is entered an error message will be printed to the console and the popup window will appear again to request user input. (Values greater than 6 digits will be truncated before checking validity)

Save Instructions:

1. Using the cursor, click the button located toward the bottom left of the screen labeled "Save Instructions to File"

> A popup window should appear with access to your local file system

2. Enter the name of the file to save instructions to in the "Save as" text field

3. Choose the directory where the file should be stored

4. Press the save button (using the cursor) when ready to save the file

> A message should appear in the Console window "Instructions exported successfully"

Change Theme:

1. Using the cursor, click the button in the top left of the application labeled "Settings"

> A popup window should appear with RGB fields for both primary and secondary colors

2. Enter values in the text field associated with R, G, or B for the desired color

3. When all values have been entered, click accept (using the cursor) to update the theme of the application

- Note: values entered need to be numbers between 0 and 255

Adding a New Tabs

1. Locate the "+" symbol on the left side of the Memory Address column (The large rectangle on the left side of the screen)

2. Using the cursor, click on the "+" symbol

3. A new tab should now be created, and you should see it appear above the "+" symbol

Changing tabs

1. After you have added a new tab, locate your existing tabs (to the left of the Memory Address column, above the "+" symbol)

2. Using the cursor, click on the tab you want to change to

3. The tab should now change colors to indicate it is the active tab

- Note: Any modifications done in a particular tab will not affect other tabs
- Note: Running a program will run the contents from only the active tab

Closing a tab

1. Locate the tab you want to close (to the left of the Memory Address column, above the "+" symbol)

2. Locate the "x" symbol that is attached to the top of that tabs name

3. Using the cursor, click the "x" symbol

4. That tab should now be closed and now longer appear next to the other tabs

- Note: The "x" symbol will not be present if there is only 1 existing tab. This is because there must always be at least 1 open tab.

Installation (for terminal based application

- Clone the repository.

> Note: You will need the directory path of where you are placing your clone (or files) of the repository, so it may be worth jotting down the location.

- Create your UVSim's BasicML supplementary file, you can do this by following the instructions in the `Creating Your BasicML File` section below.

For Windows:

> **Note:** The following instructions are tailored for using the

[MinGW](https://www.mingww64.org/) g++ compiler. If you do not have a compiler for C++

installed, please go to the

Compiler section below and follow the MinGW setup instructions.

1. Open a Windows Command Prompt.

> This can be done by searching for `cmd` in the Windows search bar and clicking on the

Command Prompt application.

2. Navigate to the directory where you cloned the repository.

> This can be done by using the change directory `cd` command.

```cmd
cd
path\to\repository
```

3. Here you will need to compile the UVSim's source code.

```cmd
mingw32-make
```

4. Once the compilation is complete, you will have an executable file named `UVSim.exe` in the same directory as the source code. You will now proceed to the next section `Creating Your BasicML File` to create your BasicML file.

For MacOS:

NOT CURRENTLY IMPLEMENTED

For Linux:

NOT CURRENTLY IMPLEMENTED

Creating Your BasicML File

As a final step before executing, you will need to create a plain text file with the extension `.txt` that contains lines of BasicML code you want to run in the UVSim.

All values should be in the following format of a signed six-digit decimal number (7 positional characters with the first being a plus or minus symbol followed by 6 digits, each being an integer between 0-9) and should be separated onto their own line:

> ±

Example:

```plaintext
+010007
+010008
+000000
+000000
```

For any lines that are meant to be instructions and not simply numeric values, you will still follow the same signed six-digit decimal number format but with the following structure:

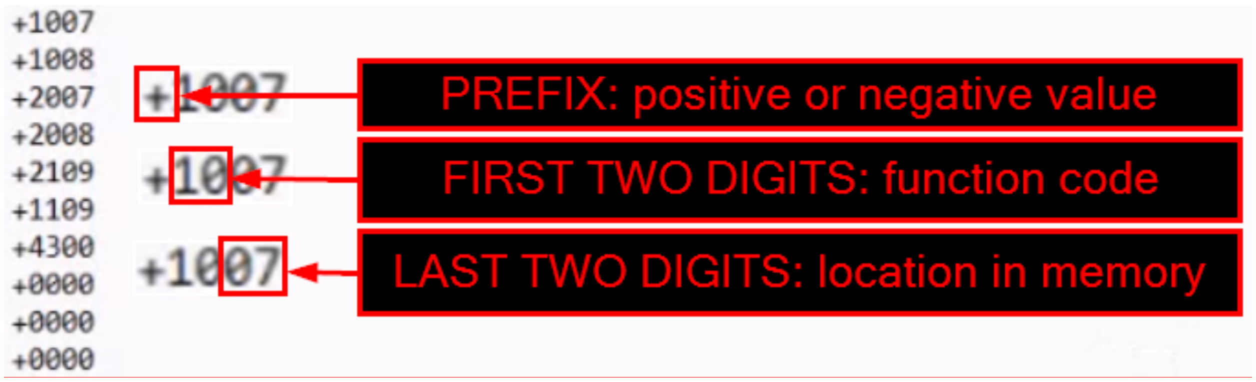
> ± [opcode] [memory address]

Position 1: Sign

Position 2 & 3 & 4: Opcode

Position 5 & 6 & 7: Memory Address

Visual Representation:


+1007
+1008
+2007
+2008
+2109
+1109
+4300
+0000
+0000
+0000
+1007
PREFIX: positive or negative value
FIRST TWO DIGITS: function code
LAST TWO DIGITS: location in memory

For a list of available opcodes and their descriptions, please refer to the Operation Codes section below.

Operation Codes:

> **I/O operations:**

010: READ – *Read a word from the keyboard into a specific location in memory.*

011: WRITE – *Write a word from a specific location in memory to screen.*

> **Load/store operations:**

020: LOAD – *Load a word from a specific location in memory into the accumulator.*

021: STORE – *Store a word from the accumulator into a specific location in memory.*

> **Arithmetic operations:**

030: ADD – *Add a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator).*

031: SUBTRACT – *Subtract a word from a specific location in memory from the word in the accumulator (leave the result in the accumulator).*

032: DIVIDE – *Divide the word in the accumulator by a word from a specific location in memory (leave the result in the accumulator).*

033: MULTIPLY – *Multiply a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator).*

> **Control operations:**

040: BRANCH – *Branch to a specific location in memory.*

041: BRANCHNEG – *Branch to a specific location in memory if the accumulator is negative.*

042: BRANCHZERO – *Branch to a specific location in memory if the accumulator is zero.*

043: HALT – *Stop the program.*

Executing Your Code In The VM

1. If you closed your terminal or command prompt, reopen it and navigate to the directory where you cloned the repository.
2. Run the UVSim executable file.

> Note: If you are not at the directory where you cloned the repository, you will get an error message. Make sure to navigate to the correct directory.

For Windows: `UVSim.exe`

For MacOS: `./UVSim`

For Linux: `./UVSim`

3. You will be prompted to enter the name of the file containing your BasicML code. This needs to be the file name with the extension `.txt`.

> Note: If you are not in the correct directory, you will get an error message. Make sure to navigate to the correct directory.

4. The UVSim will execute your BasicML code and display the output in the terminal. If you have any I/O operations, you will be prompted to enter values.

> Note: If you have any I/O operations, you will need to enter values in the terminal. If you do not enter a value, the UVSim will not proceed.

5. If you have no errors in your BasicML code, the UVSim will execute your code and perform the operations you have specified.

> Note: If you have any errors in your BasicML code, the UVSim will display an error message and stop the execution.

6. If you need to run the UVSim again or want to try a different or updated BasicML file, you can do so by repeating steps 2-5.

7. If you run into any issues where your code seems to be stuck in an infinite loop, you can stop the UVSim by pressing `Ctrl + C` or `cmd + C` in the terminal.

> Note: This will stop the UVSim and you will need to restart the UVSim to run your code again.

Compiler

For Windows:

1. Install MinGW-w64:
- Download and install [MinGW-w64-builds](https://www.mingw-w64.org/downloads).
- During the installation process, make sure to include `mingw32-make`.

2. Add MinGW-w64 to the System PATH:
- Open the System Properties (Right-click on This PC or Computer on the desktop or in File Explorer, and then select Properties).
- Click on Advanced system settings.

- Click on Environment Variables.

- In the System variables section, find the Path variable, and click Edit.

- Add the path to your MinGW-w64 bin directory. For example:

```cmd
C:\mingw-w64\x86_64-8.1.0-win32-seh-rt_v6-rev0\mingw64\bin
```

- Click OK to close all dialog boxes.

The Future of the UVSim

The future of the UVSim program focuses on three main advancements. We are striving to expand the functionality of the learning application by supporting more platforms, increasing the control given to the user, and adding additional visibility with the goal of providing a better learning experience.

Our current iteration of the program is only supported on windows devices. As this strangles the ability for many to use the product, we have a high priority on broadening the availability to the following platforms:

- MacOS
- Linux
- iOS
- Android
- Web

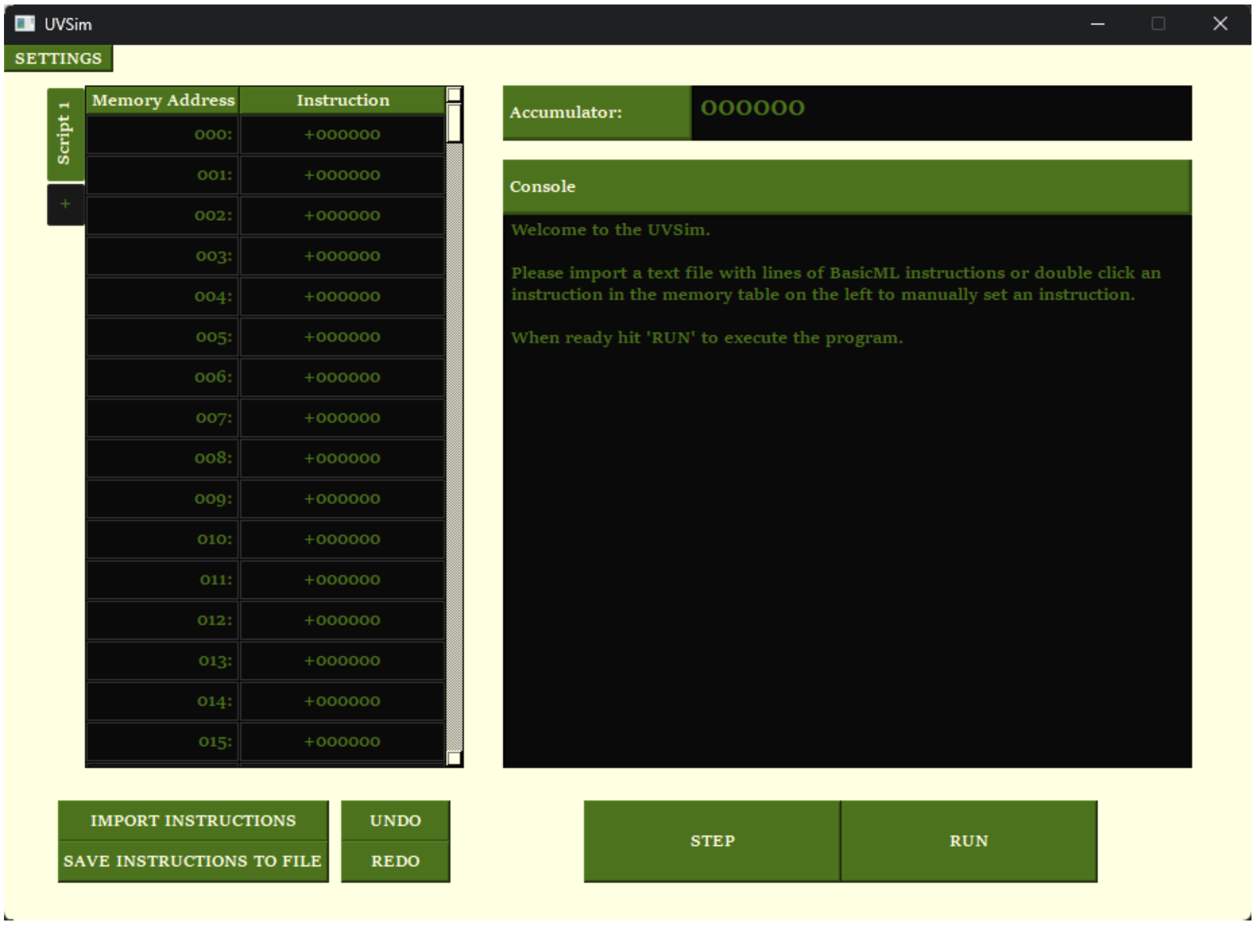
In the current state there is a lack of control and accessibility features that leaves the users wanting. To combat this, we will be working to provide the following:

- Ability to undo changes made via an undo button or CTRL+Z hotkey
- Ability to redo undo's via a redo button or CTRL+SHIFT+Z hotkey
- Ability to step through the program via the 'STEP' button, that allows for users to step through the instruction set at runtime.
- Ability for the user to set the allocation of memory size for the instructions
- Ability to provide instructions in different formats, such as binary and hexadecimal

Finally, the application is in dire need of giving the user visibility of the application's accumulator, which acts as the work registry and shows the results of operations called in the instruction set. To resolve this, we will be adding a GUI field that will display the current value of the accumulator during runtime.

<u>Future additions GUI wireframe</u>

With the proposed changes we will be updating the GUI to resemble the following wireframe:



Memory Address	Instruction
000:	+000000
001:	+000000
002:	+000000
003:	+000000
004:	+000000
005:	+000000
006:	+000000
007:	+000000
008:	+000000
009:	+000000
010:	+000000
011:	+000000
012:	+000000
013:	+000000
014:	+000000
015:	+000000

Accumulator: 000000

Console

```
Welcome to the UVSim.

Please import a text file with lines of BasicML instructions or double click an
instruction in the memory table on the left to manually set an instruction.

When ready hit 'RUN' to execute the program.
```