UVsim

MACKAY GRANGE, CAMERON HURST,
KAMILA TURAPOVA, ANNA YOUNGSTROM

Table of Contents

Introduction

UVsim is an educational tool used to assist in teaching students to code in a language called BasicML. It is a virtual machine that runs scripts written in six-digit BasicML. UVsim consists of 250 registers that can be programmed, loaded or saved to. It also makes use of a temporary register known as an accumulator. The accumulator's general purpose is for arithmetic operations and functions as the simulator's RAM. The graphical user interface (GUI) of UVsim supports loading in pre-written scripts, writing and saving new scripts, running four or six-digit BasicML scripts, changing the background and text color of the GUI, and it can run multiple windows.

User Stories and Use Cases

User Stories

"As a student, George wants to code in BasicML, so that he can learn the language and basic computer architecture."

"As a teacher, Professor Embry wants a BasicML simulator, so that he can teach his students basic computer architecture."

Use Cases

Add

Actor: test_add function

System: math

Goal: Successfully add the numbers 1000 and 100

Steps:

1. parse function code
2. get memory address (1)
3. takes the word and returns the number in the address (1000)

4. adds that word to the number in the accumulator (100)
5. accumulator then equals 1100

Actor: test_add_fail function

System: UVsim

Goal: Successfully add the numbers 1000 and 2000

Steps:

1. parse function code
2. get memory address (1)
3. takes the word and returns the number in the address (1000)
4. adds that word to the number int he accumulator (2000)
5. make sure the accumulator doesn't equal 500

Divide

Actor: test_divide function

System: UVsim

Goal: Successfully divide numbers

Steps:

1. parse function code
2. get memory address (1)
3. takes the word and returns the number in the address (1000)
4. divides that word to the number int he accumulator (2000)
5. make sure the accumulator is equal to 2

Actor: test_divide_fail

System: math

Goal: Successfully divide numbers

Steps:

1. parse function code
2. get memory address (1)
3. takes the word and returns the number in the address (500)
4. divides that word to the number int he accumulator (4000)
5. make sure the accumulator is not equal to 300

BranchNeg

Actor: test_branchneg

System: Registers dictionary

Goal: When the accumulator is negative, branch to specified memory location

Steps:

1. Parse function code for the operation instruction and memory address
2. Check if the value in accumulator is negative
3. If it is, branch to the memory address. If not, return to the current index and continue execution

READ:

Actor: Read function

System: Main program

Goal: To take in a terminal input and stores to memory.

steps:

1.Parse function code for the operation instruction and memory address.

2.Request input from the user to be stored.

4.Store input value in the memory address.

WRITE:

Actor: Write function

System: Main program

Goal: To take a value from a given memory location and print it to the screen.

Steps:

1.Parse function code for the operation instruction and memory address.

2.Find word in memory location.

3.Print value to console.

STORE:

Actor: Store Function

System: Main Program

Goal: To take the current accumulator value and store that value in a provided memory location.

Steps:

1. Parse function code for operation instruction and memory address.

2. Find memory address.

3. Replace data at that location with the current accumulator value.

LOAD:

Actor: Load Function

System: Main Function

Goal: To replace the current accumulator value with the value contained at the provided memory address.

Steps:

1. Parse function code for operation instruction and memory address.
2. Find memory address.
3. Replace current accumulator value with the value stored at the memory location.

BRANCH:

Actor: Branch Function

System: Main Function

Goal: To unconditionally jump to a specified memory location and execute the commands found there.

Steps:

1. Parse function code for operation instruction and memory address.
2. Subtract one from the provided memory location to account for the main function loop adding one after this command is finished.
3. Return modified target location to update the i value in main.

SUBTRACT:

Actor: Subtract Function

System: Main Program

Goal: To successfully subtract a word from the word currently stored in the accumulator

Steps:

1. Parse function code for the operation instruction and memory address
2. Subtract the word in the memory address from the word in the accumulator
3. Return the difference to accumulator

MULTIPLY:

Actor: Multiply Function

System: Main Program

Goal: To successfully multiply a word with the word currently stored in the accumulator

Steps:

1. Parse function code for the operation instruction and memory address
2. Multiply the word in the memory address with the word in the accumulator
3. Return the product to accumulator

BRANCHZERO:

Actor: Branch Zero Function

System: Main Program

Goal: To successfully branch to a memory address when the value in the accumulator is zero

Steps:

1. Parse function code for the operation instruction and memory address
2. Check if the value in accumulator is zero
3. If it is, branch to the memory address. If not, return to the current index and continue execution

Functional Specifications

Functional Requirements

1. The system shall parse the contents of an input test file, extracting BasicML instructions formatted as signed 4-digit numbers or signed 6-digit numbers.
2. The system shall detect if the test file utilizes 4-digit or 6-digit words and perform accordingly.
3. The system shall extract the operation defined by the first two digits of the word.

4. The system shall extract the memory location as defined by the last two digits of the word, determining the memory address where the operation should be applied or from where data should be accessed.
5. The user will be able to interact with the program through a graphical user interface (GUI).
6. Through the GUI, the user will be able to load a file.
7. The program shall have the capability to save the registers in the GUI to a file that the user can name and choose the directory of.
8. The GUI will allow the user to load a text file from a directory of their choice.
9. The GUI will allow the user to edit, copy, and paste registers into the system before running.
10. The GUI will display any errors in the system through a pop-up window visible to the user without the console.
11. The GUI will have a default color scheme of #4C721D upon first launch of the program.
12. The GUI will allow the user to change the color of the GUI with any HEX code.
13. The GUI will allow the user to launch multiple windows of the GUI, without creating multiple instances of the program.
14. The system shall load the extracted instructions into the main memory, starting at position 00.
15. The system shall handle overflow words (5+ digits) by only utilizing the first 4 given digits of the word, if the test file contains only 4-digit words.
16. The system shall handle overflow words (7+ digits) by only utilizing the first 6 given digits of the word, if the test file contains only 6-digit words.
17. The user will be able to start the BasicML program running from the GUI.
18. The system shall have 250 usable memory registers.
19. The user will be able to clear the accumulator after the BasicML program has finished running.
20. The user will be able to clear the file being run, restoring the contents of each register to 0.
21. The system shall have the capability to save the current accumulator value in a specified memory location.
22. The system shall incorporate an accumulator into which operations shall be loaded into before executing and will store the result of.
23. Once the BasicML program has finished running, the GUI will display the current value in the accumulator.

24. The system shall implement the LOAD operation, defined as 20 in BasicML vocabulary, which shall transfer data from the specified memory location into the accumulator.
25. The system shall implement the STORE operation, defined as 21 in BasicML vocabulary, which shall transfer data from the accumulator into the specified location in memory.
26. The system shall implement the READ operation, defined as 10 in BasicML vocabulary, which shall write a word from the keyboard into the specified location in memory.
27. The system shall implement the WRITE operation, defined as 11 in BasicML vocabulary, which shall write a word from the specified location in memory to the screen.
28. The system shall implement the ADD operation, defined as 30 in BasicML vocabulary, which shall add a word from the specified location in memory to the word currently stored in the accumulator.
29. The system shall implement the SUBTRACT operation, defined as 31 in BasicML vocabulary, which shall subtract a word in the specified location in memory from the word currently in the accumulator.
30. The system shall implement the DIVIDE operation, defined as 32 in BasicML vocabulary, which shall divide the word in the accumulator by the word in the specified location in memory.
31. The system shall implement the MULTIPLY operation, defined as 33 in BasicML vocabulary, which shall multiply the word in the specified memory location to the word in the accumulator.
32. The system shall implement the BRANCH operation, defined as 40 in BasicML vocabulary, which shall unconditionally branch to the specified location in memory and continue execution.
33. The system shall implement the BRANCHNEG operation, defined as 41 in BasicML vocabulary, which shall branch to the specified location in memory if the word in the accumulator is negative.
34. The system shall implement the BRANCHZERO operation, defined as 42 in BasicML vocabulary, which shall branch to the specified location in memory if the word in the accumulator is zero.
35. The system shall implement the HALT operation, defined as 43 in BasicML vocabulary, which shall end the program.

Non-functional Requirements

1. All functions will individually run in no more than 1 second.

2. All buttons will be labeled with a one-to-two-word description of their function.
3. The system shall display word prompts that specify the actions the user must take.

Class Descriptions

Classes: (look in memory.py, experimental_gui.py, run_program.py, and file_manager.py)

Memory Class:

The purpose of the memory class is to define the registers and define all the different operations performed from the different user inputs.

Functions:

Constructor: The purpose of the constructor is to create the core variables like the register, accumulator and pointer. The register is used to hold the different inputs from the user to do operations with. The accumulator is used as basically a temp variable that all operations are performed on and the pointer used for the branch holding the target memory location to branch to. The pre-conditions for this function is whenever a new Memory class is created. The post-conditions are the creation of the different variables and setting aside that memory.

clear: The purpose of clear is to make all registers equal to 0 and start fresh so a new file could be run without having anything leak over from past attempts. The pre-condition is usually when a new file is loaded in or the clear button is pressed in the program. The post-conditions are that all registers are reset to 0 and the program resets itself.

readProgram: The purpose of this function is to read in the values from the file provided from the user. The parameter is file which is just the name of the file provided. The pre-condition for this function is it getting called directly and the post-conditions are that the user's text file gets parsed into different lines and the first registers are filled by the "words" from the input.

read: The purpose of this function is to take a "word" from the user and read it into a specified memory location. This function takes two parameters, the first is the memory location the value will get stored at and the second is the "word" that's going to be stored in memory. The pre-condition for this function is being called by the user in their text file when the first two numbers of the line are 10 or 010. The post-condition is that the specified memory location now holds the number assigned by the user.

write: The purpose of this function is to write a "word" from a specified memory location and display it on the screen. This function takes one parameter which is the

memory location the “word” is located at. This function will return the “word” in that memory location. The pre-condition for this function is being called by the user in their text file when the first two numbers of the line are 11 or 011. The post-condition is that the number in the specified memory location is printed to the screen.

load: The purpose of this function is to load a “word” from a specified memory location into the accumulator. This function takes one parameter which is the memory location the “word” is located at. The pre-condition for this function is being called by the user in their text file when the first two numbers of the line are 20 or 020. The post-condition is that the accumulator will be assigned the value that’s in the specified memory location.

store: The purpose of this function is to store a “word” from the accumulator into a specified memory location. This function takes one parameter which is the memory location the “word” is located at. The pre-condition for this function is being called by the user in their text file when the first two numbers of the line are 21 or 021. The post-condition is that the value in the specified memory location is assigned the value in the accumulator.

add: The purpose of this function is to take a “word” from a memory location and add it to the accumulator. This function takes 1 parameter which is the specified memory location. The pre-condition for this function is being called by the user in their text file when the first numbers of the line are 30 or 030. The post-condition is that the accumulator has increased by the amount in the memory location.

subtract: The purpose of this function is to take a “word” from a memory location and subtract it from the accumulator. This function takes 1 parameter which is the specified memory location. The pre-condition for this function is being called by the user in their text file when the first numbers of the line are 31 or 031. The post-condition is The post-condition is that the accumulator has decreased by the amount in the memory location.

divide: The purpose of this function is to take a “word” from a memory location and divide it from the accumulator. This function takes 1 parameter which is the specified memory location. The pre-condition for this function is being called by the user in their text file when the first numbers of the line are 32 or 032. The post-condition is that the accumulator is divided by the amount in the memory location.

multiply: The purpose of this function is to take a “word” from a memory location and multiply it with the accumulator. This function takes 1 parameter which is the specified memory location. The pre-condition for this function is being called by the user in their text file when the first numbers of the line are 33 or 033. The post-condition is that the accumulator is multiplied by the amount in the memory location.

branch: The purpose of this function is to go to or branch to a specific spot in memory. This function takes 1 parameter which is the specified memory location. The pre-condition for this function is being called by the user in their text file when the first numbers of the line are 40 or 040. The post-condition is that the program will redirect to the specified memory spot.

branchneg: The purpose of this function is to go to or branch to a specific spot in memory when the accumulator is negative. This function takes 1 parameter which is the specified memory location. The pre-condition for this function is being called by the user in their text file when the first numbers of the line are 41 or 401. The post-condition is that the program will redirect to the specified memory spot if the accumulator is negative.

branchzero: The purpose of this function is to go to or branch to a specific spot in memory when the accumulator is negative. This function takes 1 parameter which is the specified memory location. The pre-condition for this function is being called by the user in their text file when the first numbers of the line are 41 or 041. The post-condition is that the program will redirect to the specified memory spot when the accumulator is zero.

inc_reg: The purpose of this function is to point the pointer to the next line from the user text file. The pre-condition for this function is after it's called by the code usually after each operation runs. The post-condition is that the variable pointer will point 1 address forward.

UVSimGui Class:

The purpose of the UVSimGui class is to create the framework for the user interface. It creates a frame and also does functions to the text file provided by the user. It also handles calling the memory class to do functions specified by the user in their text file.

Functions:

Constructor: The purpose of the constructor is to define the elements that will appear on the screen, such as labels, file input, and buttons. It has three parameters, the first is runner which is used to implement the memory class. The second (*args) is the positional arguments and the third is **kwargs which is used for keyword arguments.

show_frame: the show frame function just takes the page name that's passed to it and pushes it on top of everything else in the frame. The pre-condition is when the process starts up. The post-condition is that everything in the gui is put into place.

clear: The clear function takes everything from the text area and clears it out. It also takes the file it was holding and deletes it from the program. The pre-condition is the file getting changed or the load button is pushed. The post-condition is that everything in the text area is cleared and clean.

select_file: The select file function first clears the text area then allows the user to select a file from any file location to load into the text area. The pre-condition is when the user presses the select file button. The post-condition is that the file is selected to get loaded into the text area.

load_file: Takes everything in the text area, deletes it and replaces it with the text from the user's file. The pre-condition is that a new file is loaded into the program. The post-condition is that the new text from the file is put into the text area.

save_file: The save file function uses save as to take the text in the text area and either save over an existing file or save as a new file. The pre-condition is that the save button is pushed. The post-condition is that the text from the text area gets saved as a local file.

run_file: The run file function takes the text in the text area and uses the memory class to run the specified functions. The pre-condition is the user pressing the run button. The post-condition is the program running and the result being output on the screen.

output: The output function prints the results of the program running the text file provided by the user. The pre-condition is the program being run and the post-condition is the result being output to the screen.

popup: The popup function will show a popup for every user input required by the memory class. The user will then enter a 4 or 6-digit number to be used in the program. The pre-condition is when the functions that need input are called, the post-condition is that the number inputted would be used in different operations.

invalid_input: Shows an error if an incorrect input was found in the text file.

get_file: returns the file path to the file the user wants to run.

MainScreen class:

Just defines all the buttons and things that appear on the main frame of UVSim. This includes all buttons and smaller frames.

SettingsScreen class:

This is the second frame managed by the UVSimGui Class. This includes the slider bars to control the rgb or hex values of the primary and secondary color themes.

Helper Functions:

update_primary_color: takes multiple different parameters to determine the color of the new screen. Changes the background of the screen and other defined primary colored objects.

update_secondary_color: takes multiple different parameters to determine the color of the new screen. Changes the background of the screen and other defined secondary colored objects.

to_hex: converts inputted rgb values to hex values.

main: assigns a null runner, starts up the software and starts up mainloop. (For testing purposes only, many features are not available if run from the gui main itself.)

FileManager() Class:

Separates file functionality from RunProgram. While small right now, additional functions may be added to this class in the future.

Functions:

Constructor: No variables to initialize.

RunFile(): Takes a program and a gui as input variables. Checks to make sure the file is within the 100 commands limit. If it is not it displays an error to the gui.

RunProgram() Class:

The purpose of the RunProgram class is to manage the different background processes all at the same time, namely the program, the gui, and the file management. This is the program who's main function we will run to launch the software.

Functions:

Constructor: The class is initialized and instantiates a gui class attribute, a memory class attribute, and a file manager attribute.

new_window_runner: Makes a new instance of RunProgram then runs the main loop again to create a new window of the program.

load_file: Communicates to the file_manager class that the file should be loaded into memory, unless the selected file is too large.

clear: Communicates to memory that the clear function should be used.

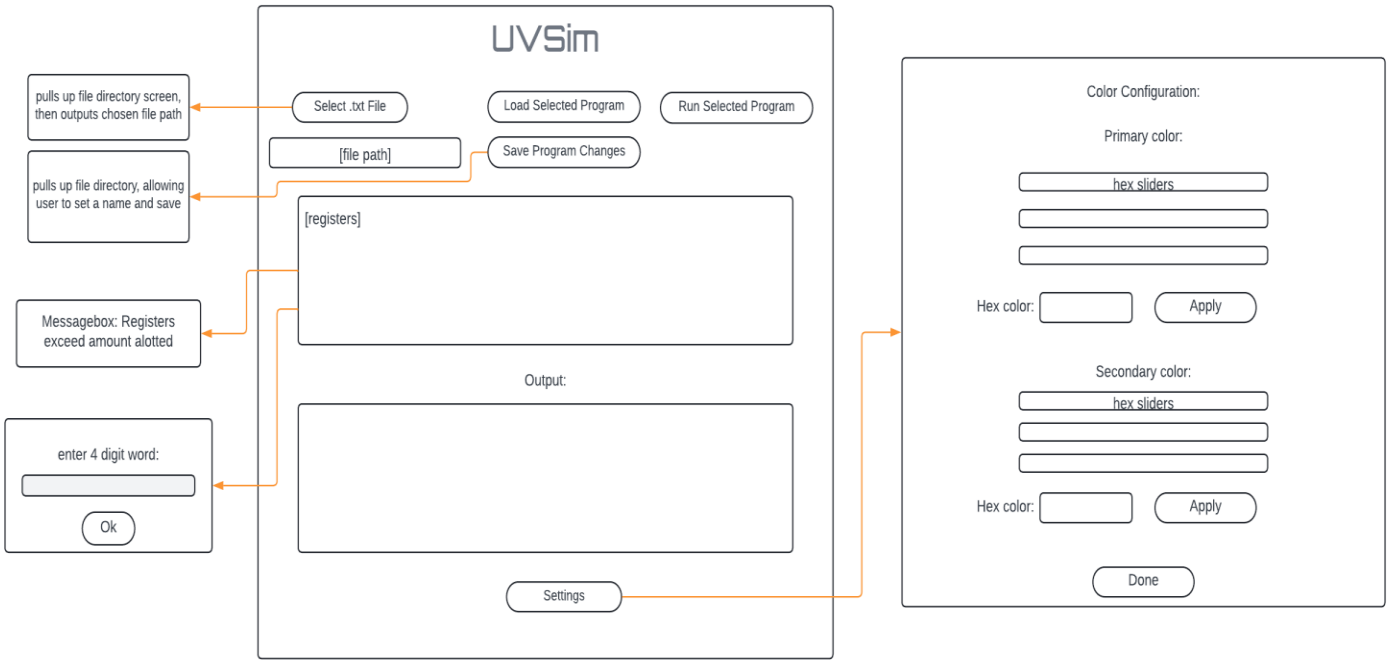
make_register_long: Takes the word from the set of instructions provided and gives it a signed value.

check_registers: Checks to see if the instructions provided are 4-digit or 6-digit instructions. It then goes through each instruction making sure each one is the correct length.

execute_program: Begins execution of the loaded program and outputs to the output console.

main: Used to start the application, creates a runner object and starts the main loop of the gui.

GUI wireframe



Unit test descriptions

Unit Test Name	Description	Reference	Inputs	Expected Output	Pass or fail?
Subtract Pass	tests whether the subtract function subtracts correctly	test_subtract	register= 1: '1000', accumulator = 2000	results == 1000	passed
Subtract Fail	tests whether assert will fail if the subtraction is wrong	test_subtract_fail	register= 1: '1000', accumulator = 2000	results != 500	passed

Multiply Pass	tests if the multiply function multiplies correctly	test_multiply	register= 1: '1000', accumulator = 5	results == 5000	passed
Multiply Fail	tests if assert will fail if the multiplication is wrong	test_multiply_fail	register= 1: '1000', accumulator = 5	results != 500	passed
Branch Zero	tests if the function will branch when accumulator is zero	test_branchzero	accumulator = 0, current_index_zero = 5	results == 1 (index)	passed
Branch Not Zero	tests if the function doesn't branch if the accumulator isn't zero	test_branchnotzero	acumulator = 5, current_index = 7	result == current_index	passed
Add Pass	Tests whether the add function adds correctly or not	test_add	register = 1: '1000', accumulator = 100	accumulator == 1100	passed
Add fail	Tests to make sure it wasn't a wrong value	test_add_fail	register = 1: '1000', accumulator = 2000	accumulator != 2000	passed
Divide Pass	Tests whether the divide function divide correctly or not	test_divide	register = 1: '1000', accumulator = 2000	accumulator == 2	passed
Divide fail	Tests whether the delete function deletes correctly or not	test_divide_fail	register = 1: '500', accumulator = 4000	accumulator != 300	passed
Branch negative	Tests if the function branches when the accumulator is negative	test_branchneg	accumulator = -2 current_index = 4	result == 1	passed
Branch not negative	Tests if the function branches when the accumulator is not negative	test_branchnotneg	accumulator = 10 current_index = 6	result == 6	passed
Test Read	tests if input is stored in correct register	test_read	register = 5 registersDictionary ={0:'', 1:'', 2:'', 3:'', 4:'', 5:''} user input = '1234'	dic[5] == '1234'	passed
Test End of Dictionary	tests if an input is stored in a newly created register if the input resgister is higher than the last in the dictionary.	test_read_new_register	register = 6 registersDictionary ={0:'', 1:'', 2:'', 3:'', 4:'', 5:''} user input = '1234'	dic[6] == '1234'	passed
Test write	tests if the value of a given register is printed to screen	test_write	register = 5 registersDictionary ={0:'', 1:'', 2:'', 3:'', 4:'', 5:'1234'}	captured.out == '1234\n'	passed

Test Write Blank register	tests if an empty string is printed to screen if register is empty.	test_write_empty_register	register = 4 registersDictionary ={0:'', 1:'', 2:'', 3:'', 4:'', 5:'1234'}	captured.out == '\n'	passed
Test Load	Tests if the function correctly loads to the accumulator the value contained within the specified memory location.	test_load()	register = {1: '9000'} accumulator = load(1, register)	accumulator == '9000'	passed
Test Load Invalid Input	Tests if the function correctly is interupted and an error is printed when an invalid memory location is specified.	test_load_index_error()	register = {1: '9000'} accumulator = load(2, register)	accumulator != '9000' & "Target memory location out of bounds."	passed
Test Store	Tests if the function correctly updates the specified memory location with the provided data.	test_store()	register = {1: '9000'} accumulator = load(1, register)	register[1] == '9000'	passed
Test Store Invalid Input	Tests if the function correctly is interupted and an error is printed when an invalid memory location is specified.	test_store_index_error()	register = {1: '9000'} accumulator = load(2, register)	assert register[1] != '9000' & "Target memory location out of bounds."	passed
Test Branch	Tests if the function correctly returns the target memory location (- 1).	test_branch	register = {1: "0000", 2: "0000", 3: "0000"} i = branch(1, register)	i == 0	passed
Test Branch Invalid Input	Tests if the function correctly is interupted and an error is printed when an invalid memory location is specified.	test_branch_index_error()	register = {1: "0000", 2: "0000", 3: "0000"} i = branch(5, register)	i != 4 & "Target memory location out of bounds."	passed
26	Test Branch Invalid Input	Tests if the function correctly is interupted and an error is printed when an invalid memory location is specified.	test_branch_index_error()	register = {1: "0000", 2: "0000", 3: "0000"} i = branch(5, register)	i != 4 & "Target memory location out of bounds."

Application instructions

How to use UVSim:

If importing a pre-written program:

1. Click "Import .txt File" and select your BasicML file.
2. Click "Load Selected Program"
3. Make any needed changes to the BasicML program.
   Note: if any changes are made, you must save them by clicking "Save Program Changes" then repeat steps 1 and 2 before the BasicML script will run with the changes.
4. Click "Run Program" to run the BasicML script.

If writing a new BasicML program in UVSim:

1. Write your BasicML script in the top text field. Keep in mind that each row is equivalent to one register. Each line of the script should be a 4-digit word, or a 6-digit word, however the program should not contain a mix of the two. There should be a maximum of 250 lines.
2. Click "Save Program Changes" and save the file.
3. Click "Import .txt File" and select the file you just saved.
4. Click "Load Selected Program" 5. Click "Run Program" to run the BasicML script.

How to change color scheme:

1. Click "Settings"
2. Set the background color by moving the color selector bars or inputting a hex value for "Primary Color". Then click "apply".
3. To set the text color, repeat step 2 for "Secondary Color".
4. Repeat steps 2 and 3 until you have the desired color scheme.
5. When finished click "Done" to return to the programming page.

How to manipulate multiple files at once:

1. Click the "New Window" button found at the bottom of the gui.
2. A New Window will pop up with a separate memory and gui instance that can be customized independently. Can work in tandem with the other window.

BasicML vocabulary defined as follows:

I/O operations:

READ = 10 (010 in the case of 6-digit words) Read a word from the keyboard into a specific location in memory.

WRITE = 11 (011 in the case of 6-digit words) Write a word from a specific location in memory to screen.

Load/store operations:

LOAD = 20 (020 in the case of 6-digit words) Load a word from a specific location in memory into the accumulator.

STORE = 21 (021 in the case of 6-digit words) Store a word from the accumulator into a specific location in memory.

Arithmetic operation:

ADD = 30 (030 in the case of 6-digit words) Add a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator)

SUBTRACT = 31 (031 in the case of 6-digit words) Subtract a word from a specific location in memory from the word in the accumulator (leave the result in the accumulator)

DIVIDE = 32 (032 in the case of 6-digit words) Divide the word in the accumulator by a word from a specific location in memory (leave the result in the accumulator).

MULTIPLY = 33 (033 in the case of 6-digit words) multiply a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator).

Control operations:

BRANCH = 40 (040 in the case of 6-digit words) Branch to a specific location in memory

BRANCHNEG = 41 (041 in the case of 6-digit words) Branch to a specific location in memory if the accumulator is negative.

BRANCHZERO = 42 (042 in the case of 6-digit words) Branch to a specific location in memory if the accumulator is zero.

HALT = 43 (043 in the case of 6-digit words) Stop the program

The last two digits (three in the case of 6-digit words) of a BasicML instruction are the operand – the address of the memory location containing the word to which the operation applies.

Future Road Map

Some of the features we would include on future updates of UVsim include:

1. Step through function to run through one line of code at a time
2. View of which registers are in use
3. Debugging tool that will check the syntax of the code before running
4. Color coding by commands for better visibility
5. Saving projects and having the ability to go back to the same project that's already been opened
6. AI for explaining code or writing bits of code
7. Being able to open tabs and format the GUI to show multiple tabs at the same time rather than opening windows

Wireframe