UVSim Proposal

CS 2450-X01
FALL 2025

Authors:
KYLE GREER
ETHAN RASMUSSEN
DARBY THOMAS

Table of Contents

Executive Summary

The UVSim project was created to provide a simplified, interactive way to understand how computers execute low-level instructions. Instead of requiring specialized hardware or deep technical knowledge, UVSim acts as a virtual machine that interprets the BasicML language. By simulating the inner workings of a CPU by being able to read instructions, performing mathematical functions, handling memory, and more, UVSim makes the fundamentals accessible to students and new developers. Its purpose is to turn concepts that are normally abstract and difficult to visualize into something that is more digestible and easier to use and understand.

The system is built around the modular components of a real machineincluding the CPU for instruction execution, the memory for storage, and the operation codes that let the program know how to behave. The user-friendly interface allows the user to load, step through, execute, and observe the changes in the memory registers in real time. This program is both an educational tool as well as a demonstration platform. This program shows how a complete computer can be built from these simple yet well-designed components, which can be crucial for learning the foundational computing concepts.

Use Cases + User Stories

Use Cases:

Use case #1: Color Themes

Actor: Program User

System: UVSIM GUI class

Goal: Create a custom color theme that is reflected in the GUI.

Steps:

1. User runs program
2. Clicks File > Themes > Custom Theme
3. Color Wheel opens
4. User selects one color then clicks OK
5. User selects second color then click OK again
6. The color picker window closes and GUI updates with new chosen color scheme.

Use case #2: UVSim Write command

Actor: Program User

System: memory management and code processor

Goal: Successfully store a value from the accumulator into the specified memory location.

Steps:

1. User starts the program and opens the file
2. Memory registers are loaded so user can see where the number is loaded
3. User runs the program and sees that the value stored in the specified memory register is printed to the screen.

Use case #3: UVSim Read + write command

Actor: Student

System: cpu, program loader, memory and control instructions

Goal: Successfully enter a value and have it printed back

Steps:

1. Student runs program
2. User selects "File" > Open

3. User selects the file and clicks "Open"
4. File is loaded into the visible Memory register
5. User clicks Run
6. Program prompts user to enter a value
7. User enters the number 4 then clicks submit
8. The number 4 is printed on the screen

Use case #4:UVSim Load command
Actor: Student
System:
Goal: Successfully load a specific memory value into the accumulator
Steps:

1. Student runs the program
2. User opens file with the load opcode
3. Algorithm parses load instruction
4. value is fetched from target memory address
5. value is placed in accumulator
6. student can access that value from accumulator in later operations

Use case #5: Branch command
Actor: Instructor (testing)
System: program counter and control instructions
Goal: Jump to a different line in the code/part of the program
Steps:

1. Instructor runs program
2. Instructor selects and runs student's file that contains a Branch instruction
3. Algorithm parses instruction
4. counter is updated to new address
5. program continues to run from new location

Use case #6: Branchneg command
Actor: Student

System: Memory and accumulator
Goal:.Redirect program flow only when accumulator is negative
Steps:

1. Student runs program
2. Selects and runs file with branchneg instruction
3. Algorithm checks accumulator value
4. if the value < 0 execution jumps to specified location.

Use case #7: Read command
Actor: Student
System: Input Console and memory
Goal:. Enter a number into specific memory location
Steps:

1. User runs program
2. User selects and opens file
3. User clicks run
4. Program prompts user for input
5. Student enters a number
6. Algorithm stores the number into target memory location as visible in the memory register location on the screen

Use case #8: Subtract command
Actor: Processing algorithm
System: Memory and accumulator
Goal:. Subtract a memory value from the accumulator
Steps:

1. Program parses function code
2. Fetches value from specified memory address
3. Subtracts that value from accumulator value
4. Stores the result back into the accumulator.

Use case #9: Divide command
Actor: Student

System: Memory and Accumulator
Goal:. Divide accumulator value by a number from memory
Steps:

1. Student runs program
2. Selects and opens file with divide instruction
3. Student clicks "Run"
4. Algorithm fetches divisor from memory
5. Accumulator value is divided by divisor
6. Result stored in accumulator
7. Student can observe new value with the write command on the screen/gui

Use case #10: Using multiple windows
Actor: User
System: UVSIM program
Goal: Have multiple instances of UVSIM open with different files at the same time
Steps:

1. User starts program
2. User can open a file and run it
3. User clicks File > Open New Window
4. New Window opens
5. User can load other files into the program here by clicking File > Open to have multiple instances at once.

User Stories:
- As a CS student learning computer architecture, I want to load a BasicML program into the simulator and execute it, so that I can visually understand how instructions manipulate memory registers on a CPU.
- As a new CS student, I want the simulator to detect and report runtime errors in my BasicML program, so that I can understand the limitations of the computer architecture.

Software Requirement Specifications

Functional Requirements:

1. The program will allow the user to load a Basic ML program file and show the lines in the memory registers in the GUI.
2. The system will parse each instruction into its operand and opcode, skipping over lines that are not instructions.
3. Store will store the word from the accumulator to the specified memory location.
4. Write will take a word from specified memory location and output it to the screen.
5. Halt will immediately end the program.
6. Add will add the word in the memory location to the accumulator and store it in the accumulator.
7. Load will get the word from specified memory location and set the accumulator to that word.
8. Branch will unconditionally jump to a specific memory location.
9. Branchneg will jump to a specific memory location only if the accumulator is negative.
10. Branchzero will jump to a specific memory location only if the accumulator is zero.
11. Read will take input from the keyboard and store it in the specified place in memory.
12. Subtract will minus the value stored in the specified memory from what is currently in the accumulator and store it as the new, current accumulator value.
13. Divide will divide the word in the accumulator by the word in the specified memory location and store it in the accumulator.
14. Multiply will multiply the word in the accumulator by the word in the specified memory location and store it in the accumulator.
15. The program provides a GUI where the user can open and run files, see how the memory is changed, and view logs of the program's actions.
16. The program allows the user to save the file over the original with any changes made in the GUI memory registers (without saving changes made by the program simply running).

17. The program allows the user to create and name a new text file by using the "save as" function.
18. The program allows the user to edit what is stored in each cell of the memory registers by double-clicking the desired cell, entering a word with the correct number of digits, and pressing the "enter" key.
19. The program allows the user to overwrite the current file and choose where in their files to save it with any user-made changes.
20. The program allows the user to select two colors and updates the GUI to use the selected colors.
21. The program allows the user to open multiple windows of the program at once.

Nonfunctional Requirements:

1. The system prevents accessing memory outside of the 100 (if 4-digit words) or 250 (if 6-digit words) defined memory registers.
2. The system provides error messages for invalid input.
3. The system provides error messages if there are errors within the file selected and if there are errors loading the file.
4. The system executes each instruction in under 2 seconds.
5. The system writes to the screen logs when memory cells are changed.

Class Definitions

memory.py

class memory

def __init__(self)

Initializes beginning size and array of memory. No input or return.

def reset(self)

This simply resets the memory array back to an empty array. No input or return

def __str__(self)

Makes printing memory really simple. No input and returns a formatted string to print.

def get_value(self, address)

Returns the value at the input memory address. This is to grab the value when trying to use an instruction

def set_value(self, address, value)

Sets the inputted memory address to the inputted value. This is used by the store function to manage the memory

def add_value(self, value)

Takes in the value wanting to be added and puts it into memory. And returns the index of where it was stored.

cpu.py

class cpu

def __init__(self, memory)

Initializes the cpu with needed dependencies. Sets the memory object to use later. Inputs memory object, doesn't return anything

def reset(self)
Clears the accumulator, program counter and instruction registers so you can run the file again without having to exit or reload the file.

def set_instructions(self, conInstruct, mathInstruct)
This is to set the control instruction and math instruction class objects after they are created. Inputs both class objects and returns nothing

def fetch(self)
Fetches current index and increments the program counter. Doesn't take input or return anything

def decode_execute(self, instruction)
This decodes the memory checking to see if there is a usable code to then call the appropriate instruction. This takes in the instruction passed from the fetch function. Returns nothing.

def run(self)
This is the main execution loop of cpu which will run till the done variable is set to true. This calls both fetch and decode then repeats until it hits a halt opcode or memory location 99.

def monitor_windows(root)
This function continuously checks whether any Tkinter windows (the main window or any Toplevel windows) are still open. It works by counting the widgets attached to the root window. If no child windows remain, the function calls root.quit() to stop the Tkinter event loop and end the program. Otherwise, it schedules itself to run again after 200 milliseconds, creating a repeating check.

program_loader.py
class program_loader

def lineCleanUp(lines)
Cleans up the lines it read to help make the verification process simpler takes in all lines read from load from file returns cleaned lines

def lineValidation(lines)
Validates each line checking for poorly formatted opcode. If all is well it will return true and takes the cleaned lines from the function before as input.

def __init__(self, memory)
Initializes the program loader with needed dependencies. Sets the memory object to use later. Inputs memory object, doesn't return anything

def load_from_file(self, filename)
This takes in the file path and checks to see if the file is valid and then reads each line parsing to make sure it has correct formatting before executing any opcodes. Takes the file name as input and returns a true or false depending off the verification.

gui.py
class UvsimGUI

The UvsimGUI class builds and manages the graphical user interface for the UVSim simulator. It allows users to load program files, run or step through them, enter input, and view log messages, memory, and register values. To work properly, it must be run in a Python environment with Tkinter installed, and the user must select a valid .txt program file. Once running, the class displays a window with menus, buttons, input and output areas, and a memory display, while also handling user interaction and coordinating with the CPU and Memory classes.

def __init__(self, root)
Sets up the GUI window when the class is first created. It takes in the Tkinter root window, configures the size and layout, and calls create_widgets to build the interface. Once this function completes, the window is ready for user interaction.

def reset(self)
Restores the CPU and memory to their initial states, reloads the currently selected program file, and writes "Program reset." to the log. This ensures that the user can rerun the program from the beginning without having to reopen the file.

def submit_input(self)
Handles the action when the user clicks the Submit button. Whatever the user typed is sent to the log as a message (for example, "Submitted input: 25"), then the input box is cleared and the button is disabled again. This makes sure the program only processes valid user input once per entry.

def check_run(self)
This will check and make sure that the current program loaded into the memory tree can be run without running into errors. If it cannot be run the user will receive an error message that they need to fix the memory before it can successfully run.

def resume_cpu(self)
Simply resumes the cpu so that the gui can quickly update with vital information for the user. No input or output.

def submit_input(self)
This strips and validates input and determines if the document has 6 or 4 digit words

def edit_memory_cell(self, event)
This takes in an event which is the cell that is getting double clicked by the user. This will then add an input field on top of the current memory cell with the value in the text box ready to be edited. Once the user is done they are able to hit the enter key to save their changes, if it's a valid "word" it will be accepted into the tree if not, it will be rejected and output an error.

def add_memory_cell(self)
Handles the logic for adding memory cells from the loaded file. This is called when the + button in the GUI is clicked and depending on which object is selected is where a new memory cell will be put.

def remove_memory_cell(self)
Handles the logic for removing memory cells from the loaded file. This is called when the - button in the GUI is clicked and will remove the currently selected memory cell in the memory tree.

def load_file(self)
Attempts to load the currently selected program file into memory. If successful, it logs "Program loaded successfully." and shows the current memory contents. If something goes wrong, such as an invalid file, it logs an error message instead.

def create_widgets(self)
Builds the entire layout of the GUI, including menus, buttons, the input box, the output log, and the memory display. When the GUI is first created, the Run, Step, and Reset buttons are disabled until

a valid program file is opened. This function ensures that everything is displayed in the correct place and is ready for use.

def on_input_change(self, event=None)
Checks whether the user has typed anything into the input box. If the box contains text, the Submit button becomes enabled. If the box is empty, the button is disabled. This prevents the user from submitting blank input.

def open_file(self)
Opens a file selection dialog window so the user can choose a program file. If a .txt file is chosen, it stores the file path, enables the Run, Step, and Reset buttons, and sets up the CPU, memory, and instruction objects. After this setup, it calls load_file so the program is immediately loaded into memory and ready to run.

def log_message(self, message: str)
Writes a message into the output log area. Messages are appended to the bottom of the log, and the box automatically scrolls so the newest message is always visible. This provides a clear history of what the program has done.

def save_file(self)
This method overwrites and saves the current file with any changes made in the GUI to the memory/registers

def save_file_as(self)
Allows the user to name the file and choose where to save it.

def open_theme_menu(self, event=None)
Opens a small popup menu near the mouse cursor that lets the user choose between available color themes (Default, Dark, or Light).

def change_theme(self, theme=”default”)
Applies the selected theme’s colors to all widgets, changing backgrounds, text colors, and button styles throughout the interface.

def apply_widget_theme(self, widget, colors)
Recursively applies the chosen theme's colors to each widget and its child widgets, ensuring consistent styling across the entire window.

def load_mem(self)
Clears and repopulates the memory display with the current contents of memory, showing each address and its corresponding value in the tree view.

def open_new_instance(self)
Creates a new Tkinter window and initializes a separate instance of the UvsimGUI interface inside it. Each call opens a completely independent GUI window, allowing multiple simulator instances to run at the same time.

control_instructions.py
class ControlInstructions

Input Parameters used in this class:
self: reference to the current instance of ControlInstructions that allows access to object's attributes and methods
memory: an instance of the memory class
cpu: an instance of the cpu class that holds the current state of the accumulator and instruction counter
gui: an instance of the gui class
address/memoryLoc: the memory location referred to by the current line of code
opcode: the operation code determined by the current line of code

def __init__(self, memory, cpu, gui)
This method initializes a new object and associates it with the memory, gui and cpu instances.

def READ(self, address)
This method prompts the user to "Enter a 4-digit word", verifies that it's not more than 4 digits then puts it in the memory location that was passed in. If it's not a 4-digit 'word' it provides an error message.

def WRITE(self, address)

This method takes the value from the memory location specified and writes it to the screen.

def LOAD(self, address)
This method takes the value from the specified memory location and loads it into the accumulator

def STORE(self, address)
This method takes the value from the accumulator and stores it in the specified memory location.

def BRANCH(self, address)
This method jumps to the specified memory location and continues the program from that point.

def BRANCHNEG(self, address)
This method jumps to the specified memory location if the accumulator is negative and continues from there.

def BRANCHZERO(self, address)
This method jumps to the specified memory location if the accumulator is zero and continues running from there.

def HALT(self)
This method stops the program.

def execute(self, opcode, memoryLoc)
This method determines which method to run based on the opcode and where to branch to or load and store data from/to.

math_instructions.py
class MathInstructions

The purpose of the MathInstructions class is to determine which opcode is being used, then run the appropriate function to perform the operation on the accumulator with the value in the specified memory address and leave it in the accumulator.

def truncate(value)

This function is declared before the class is instantiated and used to make sure that the returned value is only 4 digits long. It has a passed in value of whichever return value from the operation function that calls it and returns the last four digits of that value.
Input Parameters used for the rest of the methods in this class:
self: Refers to the current class, giving access to its memory object
accumulator: this is the current value held by the accumulator
opcode: The first two digits of the word determining which operation shall be run
memLoc: The memory location that will be used in the operation
memory: memory from the memory class which should be the lines of code from the input file

def __init__

The purpose of this method is to initialize an instance of the MathInstructions class with a reference to the memory class so the later methods can access and manipulate it.

def truncate(value)

This function is declared before the class is instantiated and used to make sure that the returned value is only 4 digits long. It has a passed in value of whichever return value from the operation function that calls it and returns the last four digits of that value.

def execute(self, opcode, accumulator, memoryLoc)

This method determines which operation code has been specified then runs the correct method to perform the desired operation or raises a Value error if the opcode is not a valid option. It returns the value that is returned by the method it runs based on the opcode.

def ADD(self, accumulator, memoryLoc)

This method adds the value from the passed in memory location to the accumulator then stores it in the accumulator and returns it. Returns the accumulator with the new value.

def SUBTRACT(self, accumulator, memoryLoc)

This method subtracts the value of the passed in memory location from the accumulator then stores it in the accumulator and returns it. Returns the accumulator with the new value.

def DIVIDE(self, accumulator, memoryLoc)

This method divides the accumulator by the value from the passed in memory location then stores it in the accumulator and returns it. Returns the accumulator with the new value.

def MULTIPLY(self, accumulator, memoryLoc)
This method multiplies the value from the passed in memory location by the accumulator then stores it in the accumulator and returns it. Returns the accumulator with the new value.

six_digit_handler.py

class Memory6(Memory)

Stores all values as signed 6-digit numbers, ensures all writes are formatted correctly and provides 250 memory registers.

def __init__(self)
Initializes the 6-digit memory module by creating a 250 cell memory array. Each cell is set to "+000000" by default.

def reset(self)
Restores the memory array back to its initial state.

def set_value(self, address, value)
Writes a value into memory while making sure it is in 6-digit format. invalid values will default to "+000000"

class CPU6(CPU)
Extends CPU functionality to six-digit words.

def decode_execute(self, instructions)
This method interprets the 6-digit instructions, determines the opcdoe and performe the operation. If instruction is invalid it logs an error.

class ControlInstructions(ControlInstructins)
Extends base control operations to 6-digit words.

def execute(self, opcode, memoryLoc)
Checks the opcode and either requests user input or defers to the appropriate instruction handler. If unknown operation is encountered, it raises an error.

class MathInstructions6(MathInstructions)

Extends math instructions class to be operational for 6-digit words.

def truncate6(self, value)

Keeps numbers within the allowed 6-digit range while preserving sign and removing overflow.

def ADD(self, accumulator, memoryLoc)

Adds the value in memory to the accumulator and truncates result to 6 digits.

def SUBTRACT(self, accumulator, memoryLoc)

Subtracts the memory value from the accumulator and truncates the result if necessary.

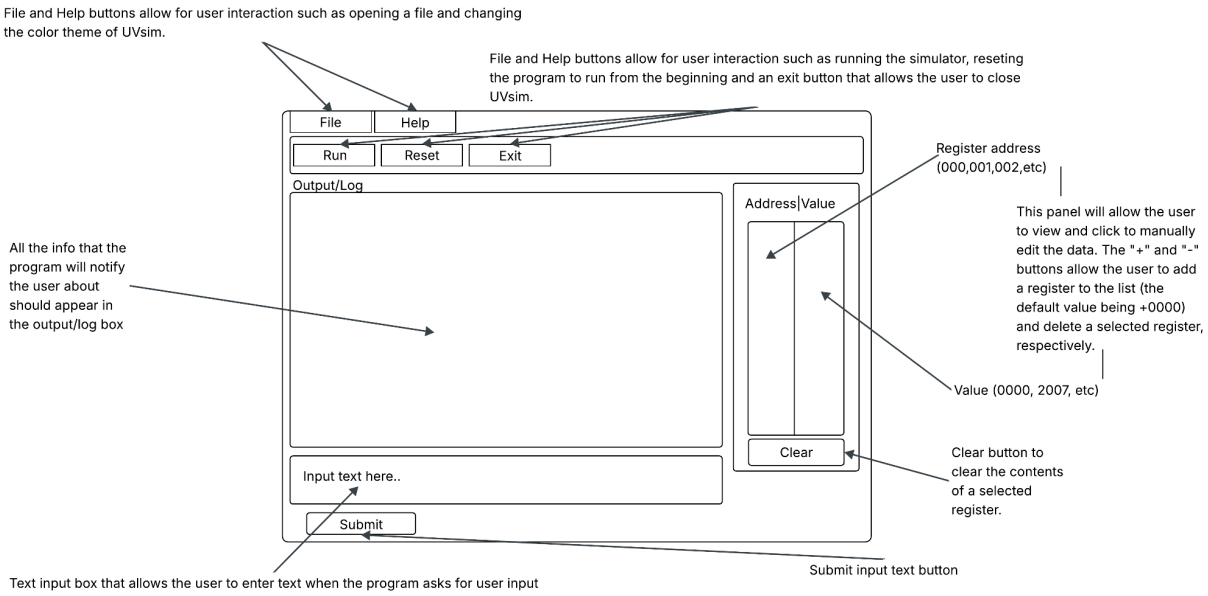
def DIVIDE(self, accumulator, memoryLoc)

Divides the accumulator by the value from the passed in memory location then stores it in the accumulator and returns it. Returns the accumulator with the new value.

def MULTIPLY(self, accumulator, memoryLoc)

Multiplies the accumulator by the value in memory and truncates result to 6 digits if necessary.

GUI Wireframe



The location of the "File" and "Help" buttons depends on your operating system. On Windows, they appear in the top-left of the application window. On macOS, they appear in the menu bar at the top-left of the screen.

User Manual

⚙️ Prerequisites

Python 3.8+

Command-line access

🚀 How to Run

Open a terminal and navigate to the project folder.

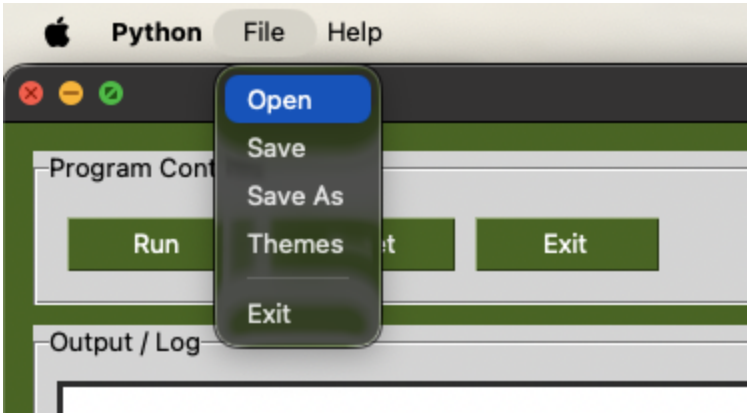
Run the simulator by typing:

python main.py

Opening a File:

Mac: Click “File” in the menu bar near the Apple logo → click Open → select a program file (e.g., Test6.txt).

Windows: Click “File” at the top left corner → click Open → select a program file (e.g., Test6.txt).



If you select an invalid file, the Output/Log box will display an error message.

Press the Run button to execute the program. You will see updates in the output area and may be prompted to type input values as the program runs.

💾 File Saving

You can now save your current program or memory state directly from the interface:

Click File → Save (or save as) to store the current state of your loaded or edited program.

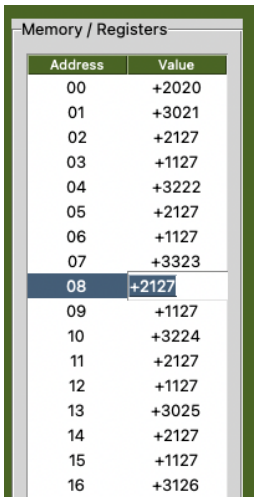
You’ll be prompted to choose a destination and filename for saving.

Saved files can later be reopened with File → Open.

🧠 Memory Editing

UVSim now supports direct memory editing:

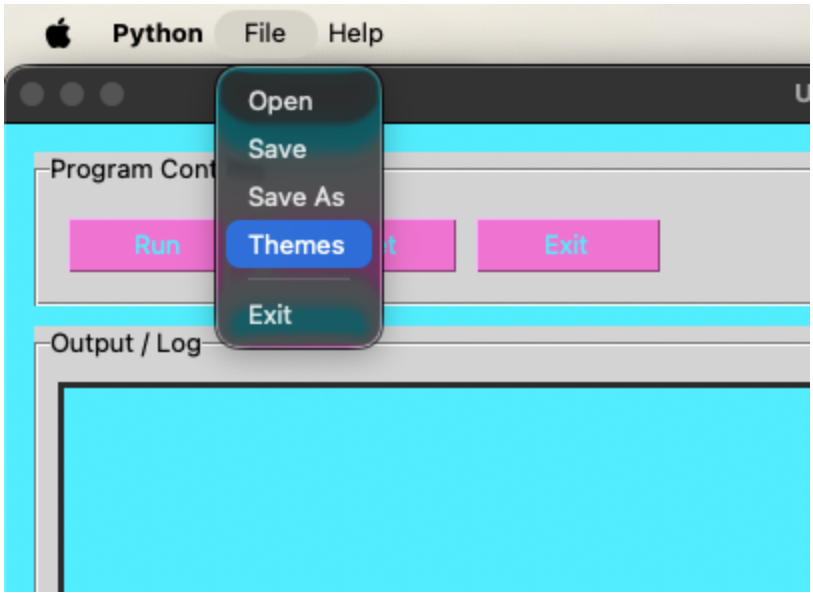
Double-click on any memory cell in the memory table to edit its value.
Press Enter to confirm your change.
You can also clear selected memory entries to customize your program or test memory behavior interactively.



🎨 Custom Color Themes

You can personalize UVSim's appearance with system-wide custom colors (File → Themes):

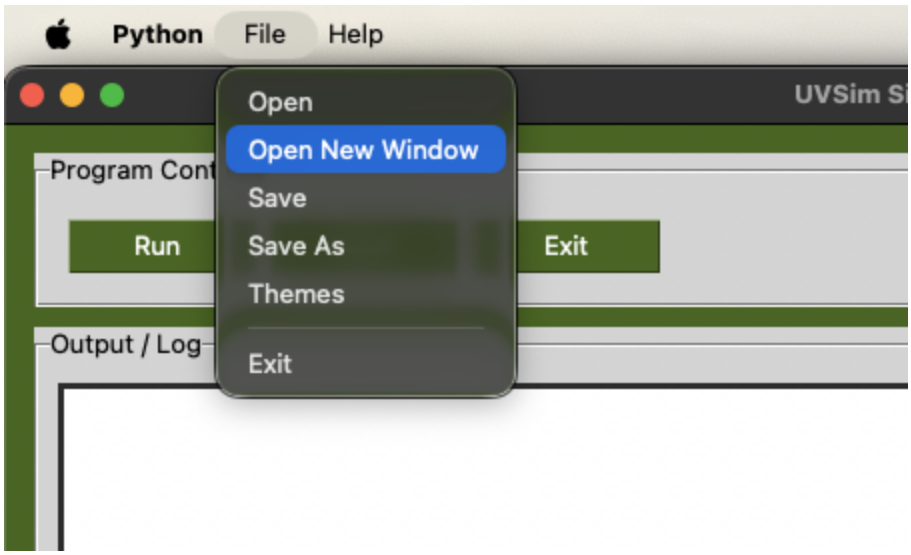
Change background, text, and highlight colors to fit your preferences.
These color settings apply across the entire application for a consistent and accessible look.



🪟 Multiple Instances Support

UVSim now supports running multiple simulator windows at the same time. You can open a completely new instance by selecting:

File → Open New Window



Each window runs independently, allowing you to test different programs, compare outputs, or debug side-by-side without interfering with your main session.

💡 What You'll See

While the program is running, you will see messages that tell you what the simulator is doing—such as loading instructions, reading input, or displaying output. You might also see updates showing the contents of memory or the accumulator (a special place where calculations happen).

Example output:

Enter an integer: 123
Stored 123 at memory location 10
Accumulator updated to +0123
Program halted successfully.

📥 Input Instructions

When the program requests input:

Type a number (positive or negative) into the bottom text box then click Submit.
Enter one number at a time as prompted.
Example:

0025
-0010
0000

🖥️ Instruction Set (Reference)

Code	Mnemonic	Description
10	READ	Read a word from keyboard into memory
11	WRITE	Write a word from memory to screen
20	LOAD	Load a word from memory into accumulator
21	STORE	Store accumulator into memory
30	ADD	Add memory value to accumulator
31	SUBTRACT	Subtract memory value from accumulator
32	DIVIDE	Divide accumulator by memory value
33	MULTIPLY	Multiply accumulator by memory value

40	BRANCH	Jump to a memory location
41	BRANCHNEG	Jump if accumulator is negative
42	BRANCHZERO	Jump if accumulator is zero
43	HALT	Stop the program

➡️ **Instruction format:**

4-digit words:
First 2 digits → Operation code
Last 2 digits → Memory address operand

6-digit words:
First 3 digits → Operation code
Last 3 digits → Memory address operand

Example:
2007 → LOAD (20) from memory location (07)
OR
020007 → LOAD (20) from memory location (07)

Unit Test Descriptions

test_memory_read_write: Checks that a value written to a memory address can be read back correctly, and is formatted with a "+" sign.

test_memory_reset: Ensures that calling reset() clears memory, and all memory values changed by running the program return to zero.

test_add_instruction: Verifies that the ADD instruction correctly adds a value from memory to the CPU accumulator.

test_subtract_instruction: Confirms that SUBTRACT properly subtracts a value from memory from the accumulator and returns the correct result.

test_multiply_instruction: Ensure that MULTIPLY takes the value in memory and multiplies it with the accumulator, producing the correct output.

test_divide_instruction: Checks that DIVIDE correctly divides the accumulator by the value stored in memory and updates the accumulator with the result.

test_load_instruction: Ensures that LOAD retrieves a value from memory and places it into the accumulator in the correct formatted form (with a leading "+").

test_store_instruction: Verifies that STORE saves the current accumulator value into the specified memory address, properly formatted.

test_branch_instruction: Confirms that BRANCH updates the program counter to a specified memory address regardless of accumulator value (unconditional jump).

test_branchneg_instruction: Tests that BRANCHNEG changes the program counter only when the accumulator contains a negative value.

test_branchzero_instruction: Checks that BRANCHZERO branches to the given address only when the accumulator equals zero.

test_halt_instruction: Ensures that calling HALT sets the CPU state to done, stopping program execution.

test_divide_by_zero_raises: Ensures that dividing a number by zero using the DIVIDE instruction correctly raises a ZeroDivisionError.

test_add_negative_number: Checks that adding a value stored in memory (a negative number) to another value results in correct arithmetic output.

test_subtract_to_zero: Verifies that subtracting a stored number from another returns zero when both values are equal.

test_multiply_by_zero: Confirms that multiplying any number by zero using the MULTIPLY instruction returns zero.

test_six_digit_memory_write_and_read: Ensures that when a six-digit word is written to memory, reading it returns it correctly formatted with a leading "+".

test_six_digit_max_address_allowed: Verifies that writing to and reading from the highest allowed memory address(249) works and preserves formatting.

test_six_digit_opcode_and_address_format: Stores a six-digit instruction format value in memory, confirms correct formatting, then ensures the LOAD operation can read and use it (accumulator gets updated).

test_six_digit_negative_values: Ensures memory correctly handles and returns negative six-digit values without altering formatting.

test_memory_size_is_250: Checks that the memory component initializes with exactly 250 memory registers.

Future Roadmap

The next phase of development for UVsim focuses on increasing accessibility, enhancing the educational experience, and modernizing the codebase.

1. Interface Optimization & Usability Refinement

Objective: To reduce friction for new users and streamline the workflow.

> Plan: We will conduct a code refactor to eliminate technical debt, resulting in a more stable application. Simultaneously, the User Interface (UI) will be decluttered to provide a more intuitive experience, ensuring that users can focus on the logic rather than navigating complex menus.

2. Interactive Step-Through Debugging

Objective: To improve logical transparency and aid in error diagnosis.

> Plan: We will implement a "Step-Through" mode that allows users to execute programs one line at a time. Crucially, the interface will visually highlight the specific function or memory location actively being executed. This provides immediate visual feedback on control flow and state changes, which is vital for debugging and understanding program behavior.

3. Migration to a Web-Based Platform

Objective: To maximize accessibility and remove platform dependency.

> Plan: We plan to transition UVsim to a browser-based environment. This eliminates the need for local software installation and ensures compatibility across all operating systems (Windows, macOS, Linux, ChromeOS). Users will be able to

access the full suite of UVsim tools instantly from any device with an internet connection.