UVSIM Final Project

CS2450 Group F

Ryan Coutts

Jeffrey Woods

Tiare Jorquera

Table of Contents

Introduction

A brief overview of UVSim

UVSim is an educational software simulator designed to help computer science students learn machine language and computer architecture. Developed at the request of an educational client, UVSim allows students to write, load, and execute programs in BasicML, a simple but powerful machine language. The simulator emulates a virtual machine with a CPU, register, and 250-word memory, providing students with hands-on experience in low-level programming and computer operations.

Initially developed as a command-line prototype, UVSim has evolved into a full-fledged application with a user-friendly graphical interface. Recent enhancements include a customizable color scheme, expanded memory capacity (from 100 to 250-word memory), and support for both legacy and new file formats. The current version allows users to load, edit, and save BasicML programs through an intuitive GUI, supporting multiple open files simultaneously. These features, combined with the simulator's core functionality, make UVSim an invaluable tool for computer science education, bridging the gap between theoretical concepts and practical application in machine language programming.

User Stories

Example user stories for UVSim

User Story 1:

As a computer science student, I need a software simulator so that I can learn and understand the basic operations of a simple virtual machine, including loading and storing values in memory, performing arithmetic operations, and branching to different parts of the program, enhancing my learning, and preparing me for advanced topics in computer science.

User Story 2:

As a computer science teacher, I need a software simulator so that I can demonstrate the fundamental concepts of computer architecture and programming to my students in an interactive and hands-on manner, allowing them to write and execute programs on a virtual machine and observe the changes in memory and the accumulator.

User Story 3:

As a power user of this program, I want to be able to choose the primary and secondary colors that are displayed in the GUI so that I keep the program looking fresh and “new” instead of looking at the same plain colors whenever I’m using the simulator.

User Story 4:

As a user of this program, I want to be able to run programs in either format (4-digit or 6-digit) so that I do not have to worry about converting all my old files into the new format.

User Story 5:

As a teacher using UVSim to teach my students, I want to be able to open multiple program editor windows at once, so that I can quickly alternate between different programs and make changes to them.

Use Cases

Example use cases for uvsim

Branch

Actor: Processor
System: UV Sim
Goal: Jump to different instruction

1. Processor reads BRANCH instruction
2. Error: branch address is negative
3. System stops loading program, outputs error message, and terminates
4. Processor loads program counter with branch address
5. Program continues starting at new address immediately

Branchneg

Actor: Processor
System: UV Sim
Goal: Jump to different instruction if value in accumulator is negative

1. Processor reads BRANCHNEG instruction
2. Error: branch address is negative
3. System stops loading program, outputs error message, and terminates
4. Processor reads accumulator
5. If accumulator contains negative value, processor loads program counter with branch address
6. Program continues at address in program counter

Branchzero

Actor: Processor
System: UV Sim
Goal: Jump to different instruction if value in accumulator is zero

1. Processor reads BRANCHZERO instruction
2. Error: branch address is negative
3. System stops loading program, outputs error message, and terminates
4. Processor reads accumulator for a positive zero value
5. Processor reads accumulator for a negative zero value
6. If accumulator contains any zero value, processor loads program counter with branch address

7. Program continues from address in program counter

Halt

Actor: Processor
System: UV Sim
Goal: Terminate execution of instructions
Error: Program does not contain any halt instructions

1. System does not load program, outputs error message, and terminates
2. Processor reaches HALT instruction
3. System immediately terminates execution of instructions

Addition

Actor: Processor
System: UV Sim
Goal: Add a value from the accumulator

1. The program begins iterating through its memory contents
2. An addition operation is reached in the memory targeting address 16
3. The system retrieves the value in memory location 16
4. The accumulator is incremented by the retrieved value

Subtraction

Actor: Processor
System: UV Sim
Goal: Subtract a value from the accumulator

1. The program begins iterating through its memory contents
2. A subtraction operation is reached in the memory targeting address 99
3. The system retrieves the value in memory location 99
4. The accumulator is decremented by the retrieved value

Multiplication

Actor: Processor
System: UV Sim
Goal: Multiply the accumulator by a value

1. The program begins iterating through its memory contents
2. A multiplication operation is reached in the memory targeting address 45
3. The system retrieves the value in memory location 45
4. The accumulator is multiplied by the retrieved value

Division

Actor: Processor
System: UV Sim
Goal: Divide the accumulator by a value

1. The program begins iterating through its memory contents
2. A division operation is reached in the memory targeting address 09
3. The system retrieves the value in memory location 09
4. The accumulator is divided by the retrieved value

Load

Actor: System
System: UVSim
Goal: Load a word from memory into the accumulator

1. System reads the load instruction into program
2. System runs program and reaches load instruction
3. System reads operand from instruction
4. System loads word from memory address specified by operand into accumulator

Store

Actor: Processor
System: UVSim
Goal: Store the value in the accumulator into a memory location

1. The Processor reads the store instruction into program
2. The processor reads the value currently in the accumulator.
3. The processor stores the value from the accumulator into the specified memory location.
4. Execution continues with the next instruction in the program.

Read

Actor: Student (end user)
System: UVSim
Goal: Read a word from the keyboard into a specific location in memory.

1. Create file with read instruction
2. Load a file with instructions into the program
3. System reads the file and expects a user input
4. User types in an input

5. Program accepts the input and saves it into a memory location specified by the instruction

Write

Actor: Student (end user)
System: UVSim
Goal: Write a word from a specific location in memory to screen location

1. Create file with read instruction
2. Load a file with instructions into the program
3. Program reads the file and expects a memory location
4. Program retrieves value at that location
5. Program displays the value

Save File

Actor: Student (end user)
System: UVSim
Goal: Save the currently loaded program into memory

1. User presses the Save button
2. A "Save As..." dialog box opens from the host computer
3. The file extension is automatically set to be ".txt"
4. The user chooses a filename
5. The user presses "Save" in the dialog box and the txt program file is saved

Multiple Editor Windows

Actor: Student (end user)
System: Program Editor Window
Goal: Open and edit multiple program (files) at once

1. User opens that program editing window
2. User opens a 2nd or 3rd programming window
3. User alternates between all windows open, making edits to each
4. User can choose to run any of the open programs at any given time

Color Scheme

Actor: Student (end user)
System: UVSim
Goal: Change the color scheme of the program

1. The user opens the theme.json file
2. The user makes changes to the file to update the color scheme of the program

3. Upon rerunning the program, the user can see the changes made to the color scheme

File Format Flexibility

Actor: Student (end user)
System: UVSim
Goal: Convert any 4-digit programs into 6-digit programs so that they can still run on UVSim

1. The user imports a file from his computer that uses all 4-digit words
2. The UVSim automatically converts the file into the equivariant 6-digit program
3. The program runs as expected in the updated format

Functional Specifications

The main features and capabilities of the software.

1. The system shall implement a READ operation where a word input from the keyboard is stored in a specific location in memory.

2. The system shall implement a WRITE operation where a word from a specific location in memory is shown on the screen.

3. The system shall implement a LOAD operation where a word from a specific location in memory is loaded into the system's accumulator.

4. The system shall implement a STORE operation where the word in the accumulator is stored in a specific location in memory.

5. The system shall implement an ADD operation where a word from a specific location in memory is added to the accumulator

6. The system shall implement a SUBTRACT operation where a word from a specific location in memory is subtracted from the accumulator.

7. The system shall implement a DIVIDE operation where the accumulator is divided by a word from a specific location in memory.

8. The system shall implement a MULTIPLY operation where the accumulator is multiplied by a word from a specific location in memory.

9. The system shall have a graphical user interface displaying the program counter, accumulator, memory, and output values.

10. The system shall provide a memory capacity of 250 words, each word being a six-digit signed integer.

11. The system shall have an accumulator register to store and manipulate values during execution.

12. The system shall have a program counter to keep track of the current instruction being executed.

13. The system shall accept and execute the full range of the UVSIM instruction set architecture.

14. The system shall allow the user to select a file from his/her local machine to be processed by the simulator.

15. The system shall load the input file’s contents into its memory after a properly formatted file is provided.

16. The system shall handle negative values using a six-digit representation with a leading minus sign, and positive values with a leading plus sign.

17. The system shall truncate overflowing integer values before saving them to memory.

18. The system shall truncate underflowing integer values before saving them to memory.

19. The system shall validate branch addressed to ensure they are within the valid memory range.

20. The system shall refuse to load a program that lacks at least one HALT instruction to avoid infinite loops.

21. The system shall output a warning message upon receiving invalid user input

22. The system shall repeat a request for user input until a valid input is provided.

23. The system shall display the final state of the virtual machine upon program termination.

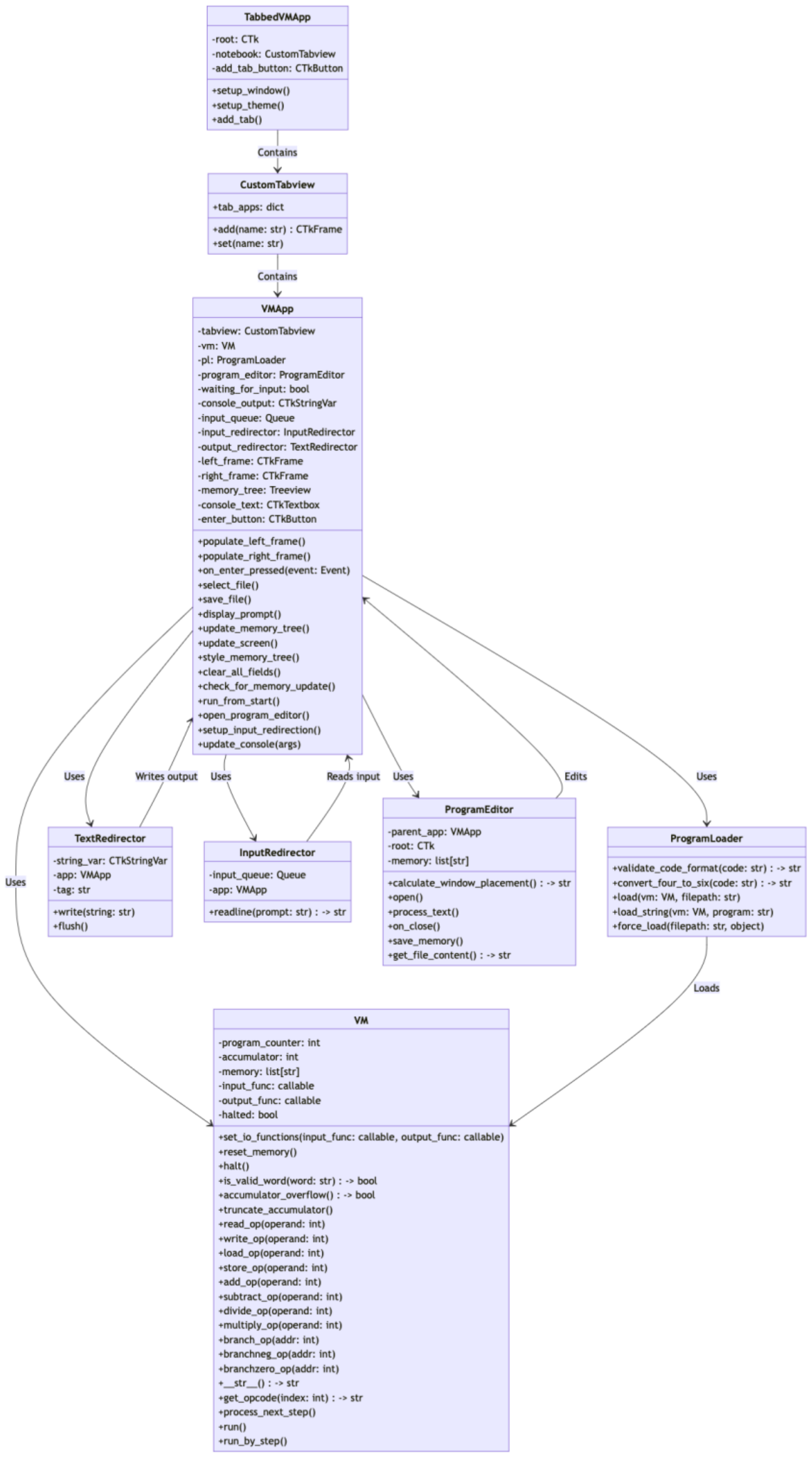
24. The system shall convert files in the old four-digit format to the six-digit format before loading them into the VM.

25. The system shall hold up to 15 programs in memory at any given time.

26. The system shall display and run each program separately in its own tab.

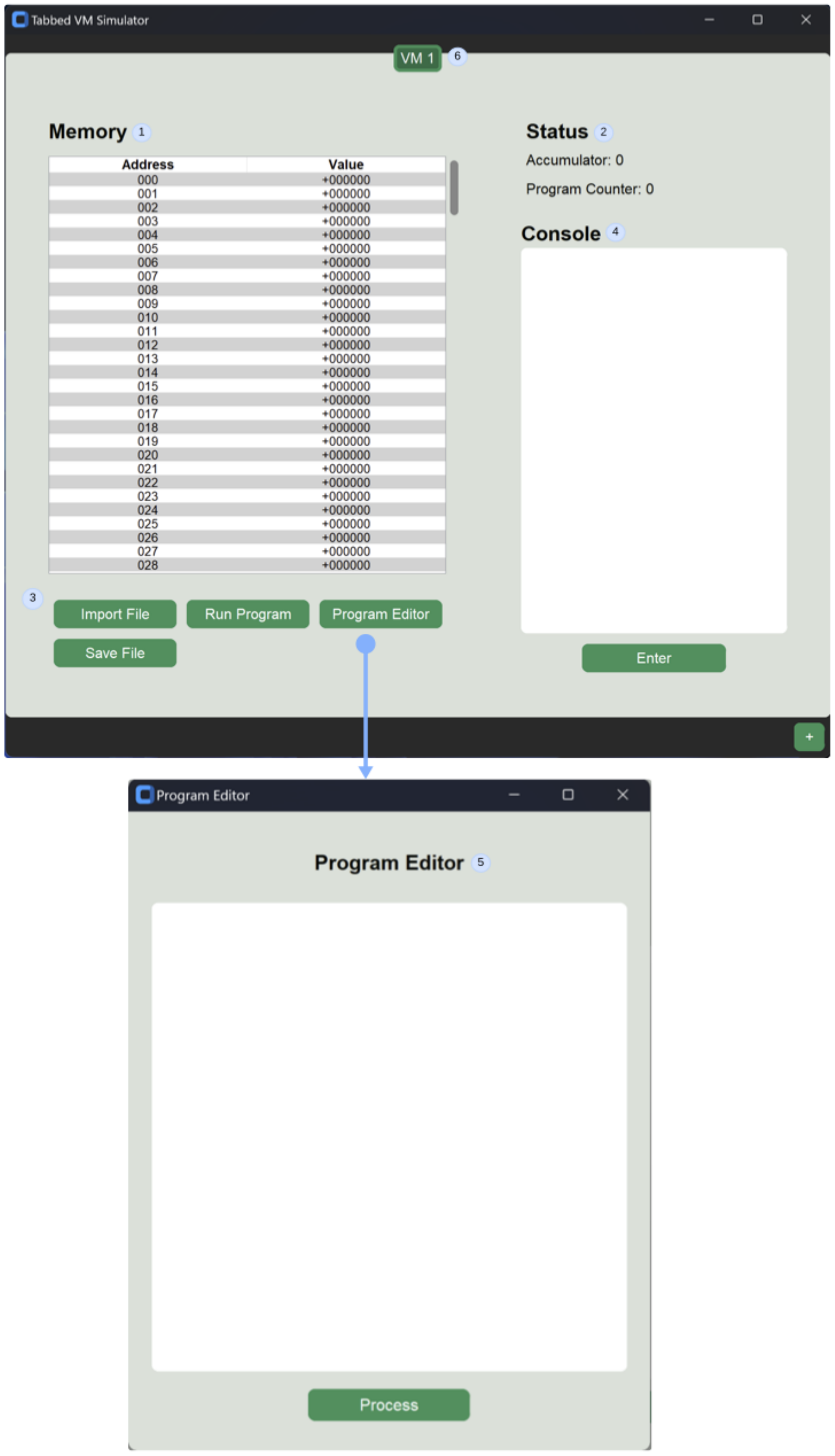
Class Hierarchy

The relationship between different parts of the software.



Wireframe

An overview of the user interface.



Address	Value
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
016	+000000
017	+000000
018	+000000
019	+000000
020	+000000
021	+000000
022	+000000
023	+000000
024	+000000
025	+000000
026	+000000
027	+000000
028	+000000

1 **Memory Section**

Displays the contents of the VM's memory

Shows memory addresses and their corresponding values

Updated in real-time as the program runs

2 **Status Section**

Shows the current state of the VM

Displays the Accumulator value, which holds the result of arithmetic operations

Shows the Program Counter, indicating the address of the next instruction to be executed

3 **Console Section**

Provides input/output functionality for the VM

Displays program output and prompts for user input

4 **Control Buttons**

Import File: Loads a program from a text file into the VM's memory

Run Program: Executes the loaded program from the beginning

Program Editor: Opens a separate window for editing the program

Save File: Saves the current program to a text file

5 **Program Editor**

Allows users to view and edit the program code directly

Provides a "Process" button to load the edited program into the VM

6 **Tab System**

Allows multiple VM instances to run simultaneously

Users can switch between different VMs using tabs

Unit Tests

An automatic verification process to prove that the software performs as expected.

1. Word Validation

The tests confirm that the virtual machine can accurately identify valid and invalid words based on predefined length and format criteria.

2. Input and Output Operations

The tests check the correct functionality of reading and writing operations within the virtual machine, ensuring that input values are stored, and output as expected.

3. Memory Operations

Tests are conducted to verify that the virtual machine correctly loads data from memory into the accumulator and stores data from the accumulator into memory. This includes handling both positive and negative values.

4. Arithmetic Operations

The tests ensure the accurate execution of basic arithmetic operations, including addition, subtraction, multiplication, and division. The results of these operations are validated to ensure they match the expected outcomes.

5. Overflow Handling

The unit tests check the virtual machine’s ability to detect and handle accumulator overflow conditions, ensuring that it recognizes when a value exceeds the allowable range.

6. Accumulator Truncation

Tests confirm that the virtual machine can correctly truncate values in the accumulator when they exceed a certain length, preserving the correct sign and digits.

7. Branching Instructions

These tests validate the virtual machine’s ability to correctly execute branching instructions, including unconditional, negative, and zero-based branching, ensuring the program counter is updated appropriately.

8. Program Execution Control

Tests ensure that the halt instruction is correctly implemented, allowing the virtual machine to stop execution immediately. The virtual machine's ability to handle the halt instruction from various memory locations is also tested.

9. Program Loading

The unit tests verify that the virtual machine can correctly load a program into memory from a file, handle different file formats, and manage scenarios where the program exceeds the virtual machine's memory capacity.

10. Program Execution

Tests are conducted to ensure that the virtual machine can execute a loaded program from start to finish. The tests also validate the machine's ability to handle invalid instructions gracefully.

11. File Saving

These tests check the functionality of the file-saving feature within the virtual machine's GUI application, ensuring that memory contents are saved to a file correctly when prompted.

12. File Format Conversion

The unit tests validate the conversion process from an older 4-digit word format to a newer 6-digit format, ensuring that both opcode-containing words and data words are converted accurately.

Application Instructions

A “user manual” for the program

uvsim

An educational virtual machine running BasicML

Overview

UVSim is a virtual machine implemented in Python for computer science students to learn machine language and computer architecture.

The VM runs a low-level language called BasicML to help students learn the basic concepts behind assembly and machine languages.

Hardware

UVSim contains a CPU, main memory, and registers, including an accumulator to store the results of operations before saving to memory.

BasicML

The entirety of BasicML is specified as follows:

READ = 010 Read a word from the keyboard into a specific location in memory.

WRITE = 11 Write a word from a speciffic location in memory to screen.

LOAD = 020 Load a word from a specific location in memory into the accumulator.

STORE = 021 Store a word from the accumulator into a specific location in memory.

ADD = 030 Add a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator)

SUBTRACT = 031 Subtract a word from a specific location in memory from the word in the accumulator (leave the result in the accumulator)

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

MULTIPLY = 033 multiply a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator).

BRANCH = 040 Branch to a specific location in memory

BRANCHNEG = 041 Branch to a specific location in memory if the accumulator is negative.

BRANCHZERO = 042 Branch to a specific location in memory if the accumulator is zero.

HALT = 043 Stop the program

The last two digits of a BasicML instruction are the operand – the address of the memory location containing the word to which the operation applies

Installing uvsim

Note: To run this software, you will need Python installed on your local machine.

From uvsim's root page on GitHub, click on the green *<> Code button*. This will open a small menu, at the bottom of which you need to click on *Download ZIP*. This will download a zip folder with all the necessary files onto your local machine. Go ahead and extract them into a directory of your choice.

Running uvsim

To run uvsim, start by opening a new terminal or command line window. If you are using Windows, make sure to open your terminal application as an administrator. Then, navigate to the *uvsim_master* directory (the location of this directory will depend on where you extracted the zip folder contents). A virtual environment configuration exists to ensure that the Python packages necessary for this program do not conflict with any that may be saved on your machine. Before attempting to run the program, perform the following steps to enter the virtual environment:

Windows

1. In Powershell and related terminals, type *.\bin\Activate.ps1*. In CMD Prompt, type *.\bin\activate.bat*.

2. Run the command *pip install -r requirements.txt*.

3. To exit the virtual environment when you are done using the program, type *deactivate*.

Mac/Linux

1. Run the command *source ./bin/activate*.

2. Run *pip install -r requirements.txt*.

3. When you are finished, run *deactivate*.

MacOS may not be able to run the custom Tkinter package inside of the venv depending on how you installed Python. If you use homebrew to install python-tk, the Tkinter package should then work correctly in venv.

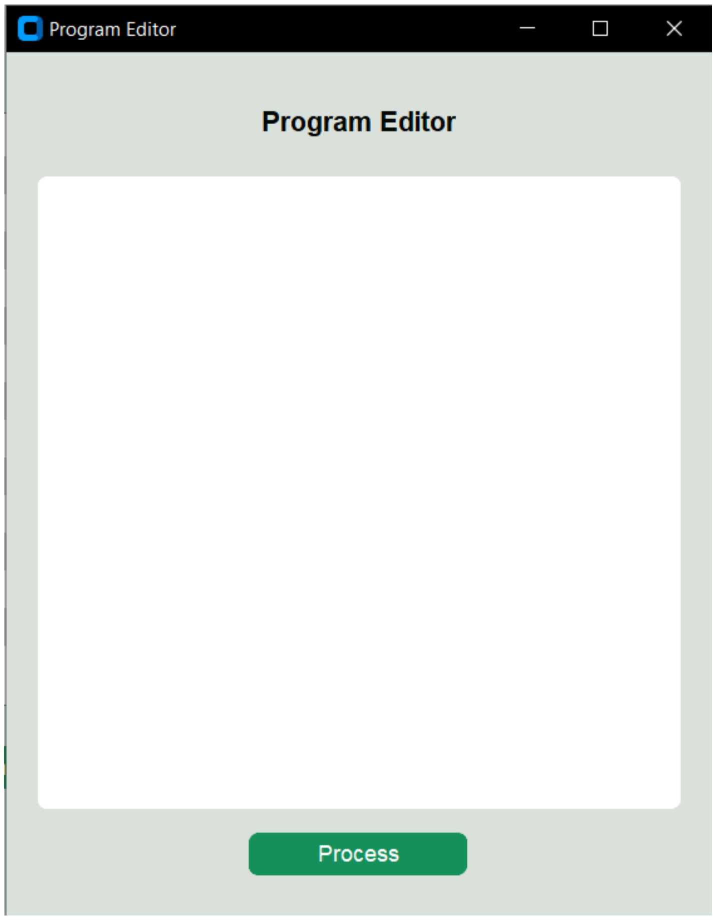
Finally, run the command *python3 main.py* If this doesn't work, you can also try replacing “python3” with just “python”, or whichever version of python you have installed.

Alternatively, you can use VS Code to run it. To do so, make sure you have the Python extension installed before opening the uvsim_master directory. Once you have the main.py file open, run it by using the play button in the top-right corner of the editor, If you are using VS Code to run the program and customtkinter is not recognized after you installed it, this might be caused by an incorrect Python interpreter

You can use shortcuts "Ctrl+Shift+P" and type "Python: Select Interpreter" to choose your virtual environment.

Using uvsim

Once the program is up and running, start by clicking on the *Import File* button to select a file from your computer that you would like to run as the program. You can also click on the *Program Editor* button which allows you to edit the contents of a file you've uploaded, or write a program completely from scratch. Once you've loaded your program into the system - either through a file import or manually - click on the *Run Program* button to execute your program's instructions.



Program Editor Window used to write a program from scratch

Any keyboard input requests from the program will appear in the Virtual Console. Click into the console, ensure that your cursor is below all text on a newline, and then type the requested input. Press *Enter* to submit your input to the program.

In the Memory section of the GUI window, you will see the entire contents of the machine's 250-address memory; the memory will always stay up to date as the machine progresses through each program, keeping you informed of the machine's current state at any given time.

In the Status section, you will see the value currently loaded in the accumulator. The accumulator serves as temporary storage for the results of operations; at the end of all relevant operations, the value is most often stored back in memory. Beneath the accumulator is the Program Counter; this register points to whichever instruction the machine executes next. You can keep an eye on the Program Counter as it updates live to debug and verify the execution of your program.

If you've altered a program with the Program Editor and you'd like to save it as a new file, click *Save File*. A dialog window will open, allowing you to select the filename you'd like as well as the directory to save it in. The file extension should be set automatically, but if it isn't on your system, make sure to set it to ".txt", as this is the only file type that the VM accepts.

When you have finished running a program, you can load another using the steps mentioned above, and all fields in the GUI will clear their outdated contents and load in values from the new program.

When you are done using UVSim, click the *Close* button at the top to terminate the program (exact details of the button will vary by platform).

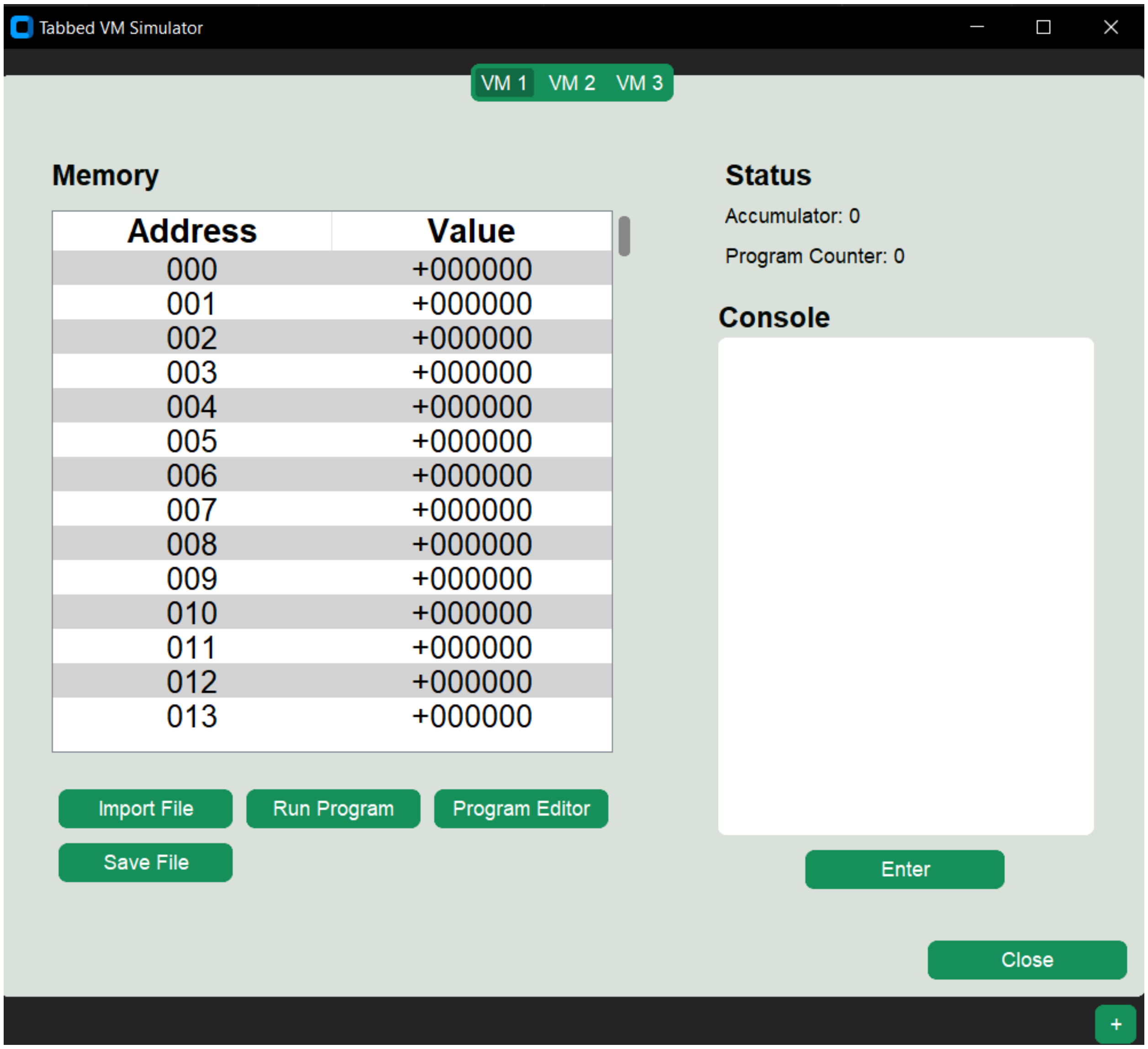
Note on input files

If you want to process your own program through uvsim, you must ensure that the source file is formatted correctly. Each instruction (line) of the file should be either a `+` or `-` followed by 6 digits - a 3-digit operation (see the top of this file for details on each operation), and then a 3-digit target address (000 - 249). 6-digit data words (eg. a 6-digit integer value such as +928764) are also accepted if the program never attempts to execute them as an instruction. This program also supports an older file format with only 4 digits per instruction. This format is similar, but the operation (same as the ones detailed at the top of this file, but without the initial 0) and the target address (00 - 99) are only 2 digits each. Internally, UVSim converts each 4-digit word to its 6-digit equivalent without changing any behavior. If you wish, you may click *Save File* to save this converted version of the file, either overwriting the old file or saving it as a new one. As UVSim's natural word length is now 6 digits, the conversion is a one-way process; if you would like to modify a file using only 4-digit words, you will have to test all changes within UVSim by using 6-digit words, and then manually convert all words to 4 digits before clicking *Save File*. UVSim will not run 4-digit instructions or validate your file before saving.

Important: If a file using the old format contains integer literals that contain an opcode in the first two digits (eg. +4300 as the decimal value 4300 instead of the instruction HALT), UVSim will treat that word as an instruction instead of data. In this example, the line would be converted to +043000 instead of the correct +004300. Unfortunately, this means that for now, file conversion is only guaranteed to work properly if all integer literals are syntacticaly distinct from an opcode.

Multiple VM Tabs

To open additional VM tabs in the program, click on the + button in the bottom right-hand corner of the window. Up to 15 tabs of UVSim can be opened at one time. When multiple tabs are open, the user can open, edit, or run any one of the instances at any time. To close a particular tab, cick on the *Close* button located right below the console. If only one tab is opened, this button will exit the entire program.



Accumulator: 0

Program Counter: 0

Address	Value
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

A VM window with 3 tabs showing at the top

Customizing the App's Color Scheme

You can easily customize the look of the app by modifying the “theme.json” file. This file controls the colors of various elements in the application.

Quick Start

1. Open the *theme.json* file in a text editor.
2. Look for color codes that start with "#" followed by 6 characters (e.g., "#15905b").
3. Replace these color codes with your preferred colors.
4. Save the file and restart the app to see your changes.

Future Road Map

Possible directions to take our simulator in the future

Our VM Simulator has already made significant strides in enhancing computer science education. As we continue to evolve our VM Simulator, we've outlined three key initiatives to enhance its educational impact and accessibility:

1. **Advanced Execution Control**

- Implement step-by-step execution with customizable breakpoints
- Provide real-time visualization of memory and register states
- Introduce an intuitive debugging interface for in-depth program analysis

Benefits:

- Deepens understanding of program flow and machine state changes
- Enables more effective troubleshooting and learning from errors

2. **Web-Based Platform**

- Develop a browser-accessible version of the simulator
- Ensure cross-platform compatibility and responsiveness
- Integrate cloud-based storage for program saving and sharing

Benefits:

- Eliminates installation barriers, promoting wider adoption
- Facilitates seamless integration with distance learning programs
- Enables real-time collaboration and easier distribution of educational materials

3. **Mobile Application**

- Create native apps for iOS and Android devices
- Optimize user interface for touch-based interactions
- Implement offline mode for learning without constant internet connection

Benefits:

- Transforms idle time into productive learning opportunities
- Increases engagement through push notifications and progress tracking

These planned enhancements aim to position our VM Simulator as a comprehensive, accessible, and user-friendly tool for computer science education. By expanding across multiple platforms and adding advanced features, we seek to accommodate diverse learning styles and environments, ultimately fostering a deeper understanding of computer architecture and machine language programming among students.