Final Project Submission Document

Group 2

4/24/2024

Noah Christensen

Tanner Bingham

Frank Adu

Kevin Cardenas

Table of Contents

Table of Contents 2
Introduction 4
User Stories 5
Story 1: As a Computer Engineering Student 5
Story 2: As an Instructor 5
Use Cases 6
Use Case 1: Load a Program 6
Use Case 2: Write Data to Memory 6
Use Case 3: Display Data from Memory 6
Use Case 4: Load a Word into the Accumulator 7
Use Case 5: Store the Accumulator's Value in Memory 7
Use Case 6: Perform Arithmetic Operations 7
Use Case 7: Branch Execution 8
Use Case 8: Halt the Program 8
Use Case 9: Error Handling 8
Use Case 10: Program Reset and Memory Clear 9
Requirement Definitions 10
1.1 Purpose 10
1.2 Scope 10
1.3 Definitions, Acronyms, and Abbreviations 10
1.4 References 10
1.5 Overview 10
2. Overall Description 11
2.1 Product Perspective 11
2.2 Product Functions 11
2.3 User Classes and Characteristics 11
2.4 Operating Environment 11
2.5 Design and Implementation Constraints 11
2.6 Assumptions and Dependencies 11
3. System Features 12
3.1 Memory and Data Handling 12
3.2 GUI and User Interaction 12
4. External Interface Requirements 12
4.1 User Interfaces 12
4.2 Hardware Interfaces 12
4.3 Software Interfaces 13

5.1 Performance Requirements.....13
5.2 Reliability and Availability.....13
5.3 Usability Requirements.....13
5.4 Security Requirements.....13
Class Diagram.....14
GUI Wireframe.....15
Unit Test Descriptions.....16
Tests for CPU functionality.....16
Tests for Editor functionality.....16
Tests for program parsing functionality.....17
Tests for verifying the product meets the use cases.....17
Future Work.....18
1. Mobile and Web versions:.....18
2. Integration with External Application Programming Interfaces (APIs).....18
3. Real-time Collaboration Features.....18
4. Advanced User Customization.....19
5. Enhanced Security Features.....19
UVSIM User Manual.....20
Requirements.....20
Installation.....20
Running in GUI mode.....20
Running in CLI line by line mode.....20
Running in CLI file mode.....20
Showing opcode information.....20
For more information.....20
Programming.....21
BasicML Instructions.....22
Testing.....22
Tutorial.....23
Program Editor Window.....23
CPU Emulation Window.....24
Documentation.....25

Introduction

This project is designed to solve the needs of those who want to mimic a CPU's functionality without having a deeper understanding of how it works. The program utilizes an editor window which could house the user's program files. The program is then loaded into the CPU emulation window to run. After, the user can select any option from "run until halt", "run until address", or simply "run one instruction". The left pane of the program can also be used to see the current address and what is loaded into the accumulator.

The CPU emulation window cannot edit programs once they are loaded, so the editor is necessary to make any changes. The main request of the last milestone was to allow for outdated files to be accepted and altered into the new file type. This functionality supports the old file you may have created while slowly updating you to the new file types. The GUI that was created is simple enough to follow and allows for a seamless experience from the contents of the memory.

Within the program itself lies a tutorial to show you the basics of navigating through this program. It has step by step guides for using a simple program. In addition, a tutorial is also contained within this document.

User Stories

Story 1: As a Computer Engineering Student

A computer Engineering student wants to be able to write BasicML programs and execute them on UVSim so that they can learn about machine language and computer architecture. This includes inputting data, performing arithmetic operations, and controlling the flow of the program.

Story 2: As an Instructor

Instructors want to provide a tool like UVSim to their students so they can practice and understand the concepts of machine language and computer architecture in a controlled environment. This tool should help them understand how instructions are executed by the CPU, how data is managed in memory, and how arithmetic and control operations affect program execution.

Use Cases

Use Case 1: Load a Program

Actor: Student

Description: Load a BasicML program into UVSim's memory starting at location 00.

Steps:

1. The student writes or opens a BasicML program.
2. The student loads the program into UVSim's memory.
3. UVSim validates the program and allocates it starting from memory location 00.

Use Case 2: Write Data to Memory

Actor: Student

Description: Use the READ instruction to input a word from the keyboard into a specific location in memory.

Steps:

1. The student inputs a READ instruction into their program.
2. UVSim prompts the student to input a word when the READ instruction is executed.
3. The student inputs the word, and UVSim stores it in the specified memory location.

Use Case 3: Display Data from Memory

Actor: Student

Description: Use the WRITE instruction to display a word from a specific memory location on the screen.

Steps:

1. The student inputs a WRITE instruction into their program.
2. When executed, UVSim reads the word from the specified memory location.
3. UVSim displays the word on the screen.

Use Case 4: Load a Word into the Accumulator

Actor: Student

Description: Use the LOAD instruction to load a word from memory into the accumulator.

Steps:

1. The student inputs a LOAD instruction into their program.
2. UVSim loads the word from the specified memory location into the accumulator.

Use Case 5: Store the Accumulator's Value in Memory

Actor: Student

Description: Use the STORE instruction to store the word from the accumulator into a specific memory location.

Steps:

1. The student inputs a STORE instruction into their program.
2. UVSim stores the word from the accumulator into the specified memory location.

Use Case 6: Perform Arithmetic Operations

Actor: Student

Description: Use ADD, SUBTRACT, DIVIDE, and MULTIPLY instructions to perform arithmetic operations with the accumulator and memory.

Steps:

1. The student inputs an arithmetic instruction into their program.
2. UVSim performs the specified operation between the accumulator's word and the word from the specified memory location, storing the result in the accumulator.

Use Case 7: Branch Execution

Actor: Student

Description: Use BRANCH, BRANCHNEG, and BRANCHZERO instructions to control the flow of the program based on the accumulator's state.

Steps:

1. The student inputs a branching instruction into their program.
2. Depending on the accumulator's state and the instruction, UVSim alters the execution flow to the specified memory location.

Use Case 8: Halt the Program

Actor: Student

Description: Use the HALT instruction to pause the program's execution.

Steps:

1. The student inputs a HALT instruction into their program.
2. UVSim stops executing the program when the HALT instruction is encountered.

Use Case 9: Error Handling

Actor: UVSim

Description: Handle errors gracefully, such as invalid instructions or arithmetic errors.

Steps:

1. During execution, UVSim encounters an error.
2. UVSim displays an error message indicating the type of error and the location.
3. Execution is halted or the student is prompted to correct the error, depending on the situation.

Use Case 10: Program Reset and Memory Clear

Actor: Student

Description: Reset the UVSim program and clear all memory locations for a fresh start.

Steps:

1. The student decides to start over with a new program.
2. The student resets UVSim, clearing the memory and accumulator.
3. UVSim is now ready for a new program to be loaded and executed.

Requirement Definitions

1. Introduction

1.1 Purpose

This section of the document specifies the software requirements for UVSim, a virtual machine simulator designed to facilitate the learning of machine language and computer architecture for computer science students. This SRS has been updated to include new features such as expanded memory, six-digit word support, and enhanced file management capabilities.

1.2 Scope

UVSim will allow users to write, load, edit, and execute programs in BasicML, a basic machine language. It will simulate a computer's CPU, memory, and I/O operations within an interactive graphical user interface that supports extensive customization and multiple file editing.

1.3 Definitions, Acronyms, and Abbreviations

UVSim: University Virtual Simulator

BasicML: Basic Machine Language

GUI: Graphical User Interface

SRS: Software Requirements Specification

CPU: Central Processing Unit

1.4 References

IEEE SRS Format Guidelines

Client Project Proposal Documents

Technical Documentation for GUI Frameworks and APIs

1.5 Overview

The document further includes detailed descriptions of the product functions, user interactions, system features, and the technical requirements necessary for implementation and integration of the new features.

2. Overall Description

2.1 Product Perspective

UVSim is a self-contained application intended for educational purposes, designed to potentially integrate with larger educational platforms in the future.

2.2 Product Functions

Load, edit, and execute BasicML programs with up to 250 instructions.

Support for both four-digit and six-digit word formats, including conversion features.

Customizable GUI that allows users to apply and change color schemes dynamically.

Multiple file management within a single instance of the application.

2.3 User Classes and Characteristics

Students: Engage with the simulator to learn machine language concepts.

Instructors: Use the simulator to demonstrate programming principles.

Software Developers: Maintain and extend the simulator as required.

2.4 Operating Environment

UVSim is designed to run on Windows, macOS, and Linux, requiring Python 3.10 or greater, as well as the tkinter and pillow libraries.

2.5 Design and Implementation Constraints

The GUI must be implementable across different operating systems without dependency on platform-specific features.

The application must handle both old and new file formats without allowing mixing within a single file.

2.6 Assumptions and Dependencies

Effective operation depends on the end-user machine's capability to handle Python-based applications.

Users are assumed to have basic knowledge of programming and file management.

3. System Features

3.1 Memory and Data Handling

FR1: The application shall support up to 250 lines of BasicML instructions.

FR2: The system shall handle six-digit word operations, including arithmetic and data movement.

FR3: Users can convert four-digit formatted files to six-digit formatted files through an in-app feature.

FR4: Handling of Multiple File Formats.

3.2 GUI and User Interaction

FR4: The GUI shall allow users to customize color schemes based on personal preferences using RGB or Hex color codes.

FR5: The application shall support opening, editing, and managing multiple BasicML program files in a single instance through tabs or sub-windows.

4. External Interface Requirements

4.1 User Interfaces

Detailed and interactive GUI featuring menu options for file operations, editing tools, color scheme settings, and program execution controls.

Error handling via dialog boxes and status messages.

4.2 Hardware Interfaces

No specific hardware interfaces required beyond standard PC components.

Standard hardware interfaces for personal computers, including keyboard and mouse.

4.3 Software Interfaces

Python 3.10

Tkinter for GUI components

Pillow for Python

5. Non-functional Requirements

5.1 Performance Requirements

NFR1: The GUI shall update the user interface within 1 second for any operation under normal conditions.

5.2 Reliability and Availability

NFR2: The system shall ensure data integrity during conversions and operations, with an availability goal of 99.9% uptime excluding scheduled maintenance.

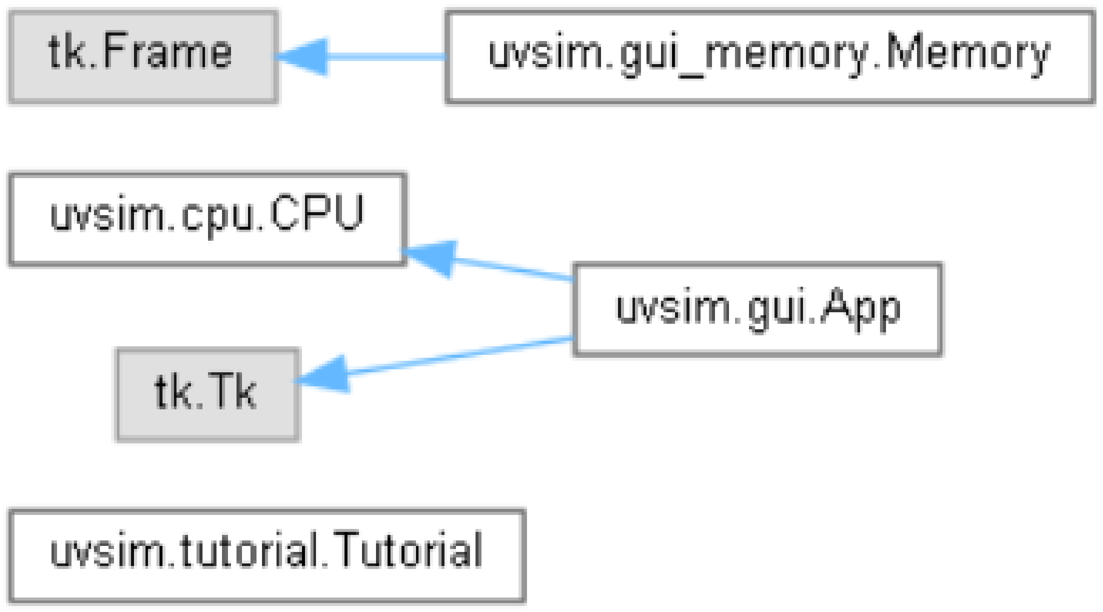
5.3 Usability Requirements

NFR3: New users shall be able to understand basic operations within 30 minutes of initial use, with comprehensive help documentation available within the application.

5.4 Security Requirements

Basic user authentication to access the application may be implemented in future versions.

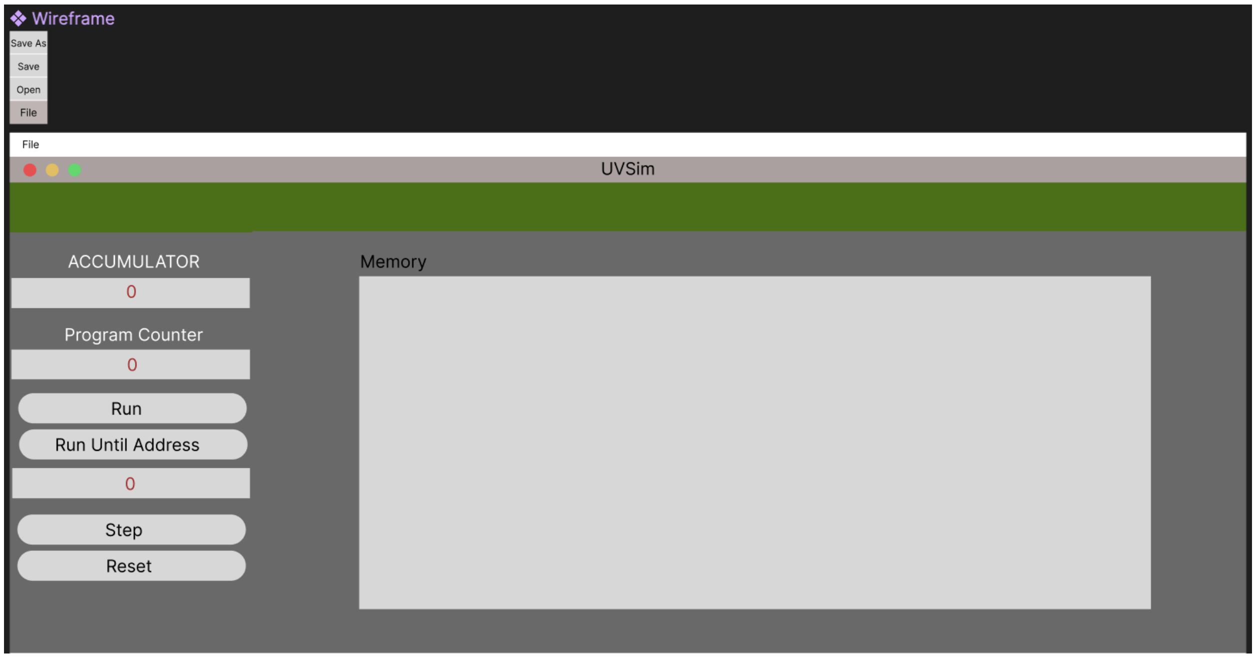
Class Diagram



Arrows indicate child to parent inheritance relationships.

Grayed out boxes indicate abstract classes.

GUI Wireframe



Unit Test Descriptions

Tests for CPU functionality

- *test_store_value_in_memory* verifies if the program can correctly store the value from the accumulator into a designated memory location.
- *test_add_positive_numbers* verifies that the program can correctly add a positive number to the accumulator.
- *test_subtract_positive_numbers* verifies that the program can correctly subtract a positive number from the accumulator.
- *test_add_with_negative_result* verifies the program can handle addition resulting in a negative accumulator value.
- *test_subtract_resulting_in_negative* verifies the program can handle subtraction resulting in a negative accumulator value.
- *test_line_to_op_data* verifies separating a 6-digit instruction word into opcode and data.
- *test_line_to_op_data_4dp* verifies separating a 4-digit instruction word into opcode and data.
- *test_multiply* verifies if the CPU can multiply numbers correctly.
- *test_divide* verifies if the CPU can divide numbers correctly.
- *test_branch* verifies the CPU can perform an unconditional jump to a new instruction.
- *test_branchneg* verifies the CPU can branch to a new instruction if the accumulator value is negative.
- *test_branchzero* verifies the CPU can branch to a new instruction if the accumulator value is zero.
- *test_halt* verifies the CPU can stop execution when instructed by the halt instruction.
- *test_read* checks if the CPU can read a word from the keyboard into memory.
- *test_write* checks if the CPU can write a word from memory to the screen.
- *test_load* verifies the CPU can load a word from memory into the accumulator.
- *test_read_write_mem_outside_program_space* verifies the program's behavior when attempting to read or write from memory outside the allocated space.

Tests for Editor functionality

- *test_open_file* verifies the editor can open a file.
- *test_save* verifies the editor can save a file.
- *test_save_as* verifies the editor can save a file with a new name.

Tests for program parsing functionality

- *test_max_length* checks how the program handles parsing programs exceeding a certain length.
- *test_no_terminal_word* verifies the program's behavior when parsing a program missing the terminating instruction.
- *test_parse_all_opcodes* ensures the CPU can interpret instructions for all supported opcodes.

Tests for verifying the product meets the use cases

- *test_loading_a_program* verifies the system can load a BasicML program.
- *test_writing_data_to_memory* verifies the system can store user input in memory after a READ instruction.
- *test_displaying_data_from_memory* verifies the system can display the value of a memory location after a WRITE instruction.
- *test_loading_word_into_accumulator* verifies the system can load a value from memory into the accumulator using a LOAD instruction.
- *test_accumulator_store_instruction* verifies the system can store the accumulator's value in memory using a STORE instruction.

Future Work

1. **Mobile and Web versions:**

   Make our app available on both smartphones and through web browsers, so it's easy to access anywhere.

   a. **Adapt to All Screens**: Change the design so it looks good on any device, from a computer to a phone.
   b. **Use Smart Tools**: Build the app so that we can use much of the same programming for both the web and mobile versions, making it easier to manage.
   c. **Connect Smoothly**: Ensure the app works well online, letting users perform all their tasks without glitches.

2. **Integration with External Application Programming Interfaces (APIs)**

   Allow our app to connect with other tools and services, making it more powerful and versatile.

   a. **Easy Linking**: Set up a way for our app to communicate securely with other services, sharing information back and forth.
   b. **Safe and Secure**: Make sure all data sent to and from the app is protected to prevent unauthorized access.
   c. **Handle Mistakes**: Build safeguards to manage errors or interruptions when the app is talking to other services.

3. **Real-time Collaboration Features**

   Let multiple users work on the same tasks at the same time, just like in Google Docs.

   a. **Instant Updates**: Use technology that allows all users to see changes as they happen.
   b. **No Overwriting**: Create a system that prevents users from accidentally messing up each other's work.
   c. **Track Changes**: Keep tabs on who made what changes, offering the ability to undo if needed.

4. **Advanced User Customization**

   Give users the power to change the app’s look and functionality to match their personal preferences.

   a. **Make It Personal**: Allow users to adjust colors, layouts, and features to fit their style.
   b. **Add-ons and Plugins**: Let users add new features or tools to their app setup without disrupting the main functions.
   c. **Save Preferences**: Make sure the app remembers each user’s settings, even when they log in from different devices.

5. **Enhanced Security Features**

   Strengthen the app’s defenses to keep all user information safe and meet the highest security standards.

   a. **Lock It Down**: Use strong encryption methods to protect data, making sure only authorized users can access sensitive information.
   b. **Double-Check Who’s Who**: Implement systems that require additional verification steps to confirm a user’s identity.
   c. **Regular Safety Checks**: Conduct ongoing security checks to find and fix any potential vulnerabilities.

UVSIM User Manual

UVSim is a simulator for a simple computer. It has a memory size of 250 words, and operates on signed integers.

Requirements

- Windows/Linux/MacOS (Others are untested)
- Python 3.10+
- Pillow Python library

Installation

1. `git clone https://github.com/CS2450-Spring-2024/FinalProject.git`
2. `cd FinalProject`
3. `python3 -m pip install pillow`
4. `python3 -m pip install pytest` (Optional)

Running in GUI mode

Run `python3 -m uvsim` in the root directory of the project.

After the program is running, see Menu > Help > Tutorial for a brief instruction on how to use the GUI.

The following video also explains the process of running/using the GUI.

https://www.youtube.com/watch?v=ivGIeuorvso

Running in CLI line by line mode

Run `python3 -m uvsim --cli` in the root directory of the project.

Running in CLI file mode

Run `python3 -m uvsim --file FILE` in the root directory of the project.

Showing opcode information

Run `python3 -m uvsim --opcode [OPCODE]` in the root directory of the project.

For more information

For more information, run `python3 -m uvsim --help`

Programming

Programs are written in BasicML. Programs can be read from a file, or line by line as they are entered in the CLI. The basic format for instruction words are 3 base 10 digits for the opcode, followed by 3 base 10 digits for the data. For example, to LOAD (opcode 20) the contents of memory address 12 into the accumulator, the instruction word would be 020012. All programs should contain a HALT instruction, but this is not necessary if the program runs to the end of memory. Every BasicML program should be terminated with the word -99999. If the simulator tries to execute a word that is not an opcode, the simulation ends. Each word should be preceded by either + or -. However, this is not enforced. In any case, the simulator interprets words with no preceding sign character as positive. Program files are plain text files. Program files should have 1 trailing newline.

For example programs, see the example_programs directory in the root of the repository.

BasicML Instructions

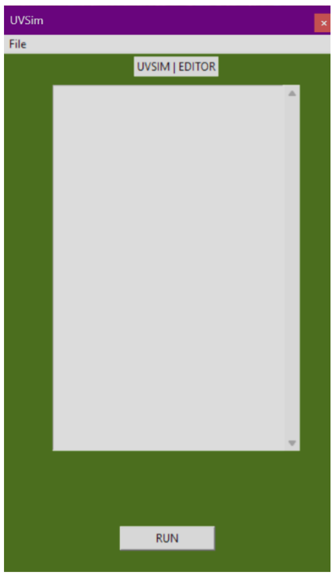
Name	**Type**	**Op**	**Description**
READ	I/O	10	Read a word from the keyboard into a specific location in memory.
WRITE	I/O	11	Write a word from a specific location in memory to screen.
LOAD	I/O	20	Load a word from a specific location in memory into the accumulator.
STORE	I/O	21	Store a word from the accumulator into a specific location in memory.
ADD	Math	30	Add a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator).
SUBTRACT	Math	31	Subtract a word from a specific location in memory from the word in the accumulator (leave the result in the accumulator).
DIVIDE	Math	32	Divide the word in the accumulator by a word from a specific location in memory (leave the result in the accumulator).
MULTIPLY	Math	33	multiply a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator).
BRANCH	Control	40	Branch to a specific location in memory.
BRANCHNEG	Control	41	Branch to a specific location in memory if the accumulator is negative.
BRANCHZERO	Control	42	Branch to a specific location in memory if the accumulator is zero.
HALT	Control	43	Pause the program.

Testing

To run the project tests, run `pytest` in the command line in the root directory of the project.

Tutorial

Program Editor Window



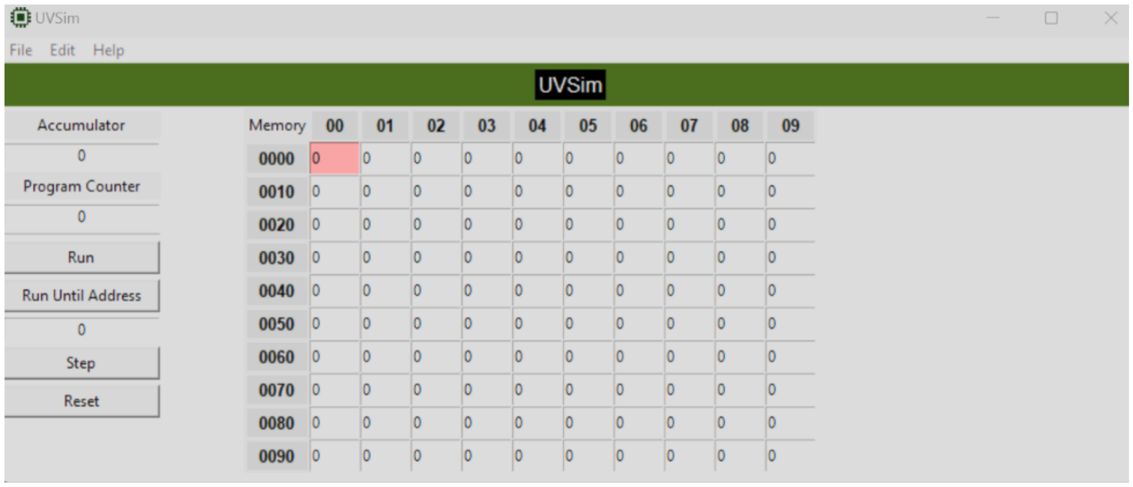
This is the program editor window, where program files can be Created, Opened, Saved, and Edited. You may open as many of these windows at a time as you wish, by going to the CPU emulation window, then clicking File > Open Editor. However, only one program can be loaded into the CPU at a time.

To open or save a program, click on the file button. Then click Save or Open.

Once a program is loaded, zeros will appear after the program. This is normal and completely expected.

Once a program is loaded or written inside the editor, you may wish to run your program. You may do so by pressing the run button at the bottom of the window. The run button parses the program and loads the program into the CPU emulation window.

CPU Emulation Window



Accumulator: 0

Program Counter: 0

Run Until Address: 0

Memory	00	01	02	03	04	05	06	07	08	09
0000	0	0	0	0	0	0	0	0	0	0
0010	0	0	0	0	0	0	0	0	0	0
0020	0	0	0	0	0	0	0	0	0	0
0030	0	0	0	0	0	0	0	0	0	0
0040	0	0	0	0	0	0	0	0	0	0
0050	0	0	0	0	0	0	0	0	0	0
0060	0	0	0	0	0	0	0	0	0	0
0070	0	0	0	0	0	0	0	0	0	0
0080	0	0	0	0	0	0	0	0	0	0
0090	0	0	0	0	0	0	0	0	0	0

This is the CPU emulation window. It contains a visual representation of program memory, the accumulator, program counter, as well as different methods of running the CPU.

Each box in the center grid represents one word of memory.

After you have loaded a program from the editor window,you may run your program by using the controls on the left of the CPU emulation window. Controls are given for running the program until a HALT instruction is reached, running the program until the address in the box under the button, or stepping instruction by instruction. The latter is especially useful for debugging programs.

It is important to note that a program should be stepped through once to verify no infinite loops exist. If they do, the program has no realistic way to detect such an occurrence, and will freeze. Hence, why it is recommended to step through all programs before pressing the Run button.

The left pane also contains a button to reset the memory, accumulator, and program counter.

Documentation

Programming and library usage documentation is located in a separate Documentation.pdf contained in the directory of this file.