UVSim for BasicML

CS 2450-X02 Group 4

Shawn Crook
Quinton Grant
Stephen Laing
Micah Peltier

Table of Contents

Executive Summary

This project is UVSim, a feature-rich software simulator, designed as an educational and development tool for computer architecture and machine language. Built with flexibility and user experience in mind, it allows users to load and manage up to three program files simultaneously, streamlining workflows for larger projects. A key highlight is its customizable interface, which offers theme selection for a more personalized or branded environment.

The simulator also includes powerful tools to enhance learning and debugging. Users can convert between legacy and modern BasicML file formats with ease, execute programs step-by-step for deeper insight into instruction flow, and observe real-time memory visualization to track how data changes throughout execution. Whether used for education, development, or exploration of low-level programming concepts, this UVSim provides a robust and user-friendly platform for working with BasicML.

User Manual

Application Instructions

Application Setup

Poetry

This project is managed with Poetry. In order to run this project, first install Poetry, through one of the recommended methods described by its documentation. Once Poetry is installed, see the following commands:

`$ poetry lock` -- This step shouldn't be necessary, but it will make sure that the lockfile is up to date with the latest versions.

`$ poetry install --all-extras` -- This creates a .venv/ folder, in which poetry installs all the necessary libraries to run this project.

Running the Application

To run the program, run:

```
$ poetry run python -m src.main
```

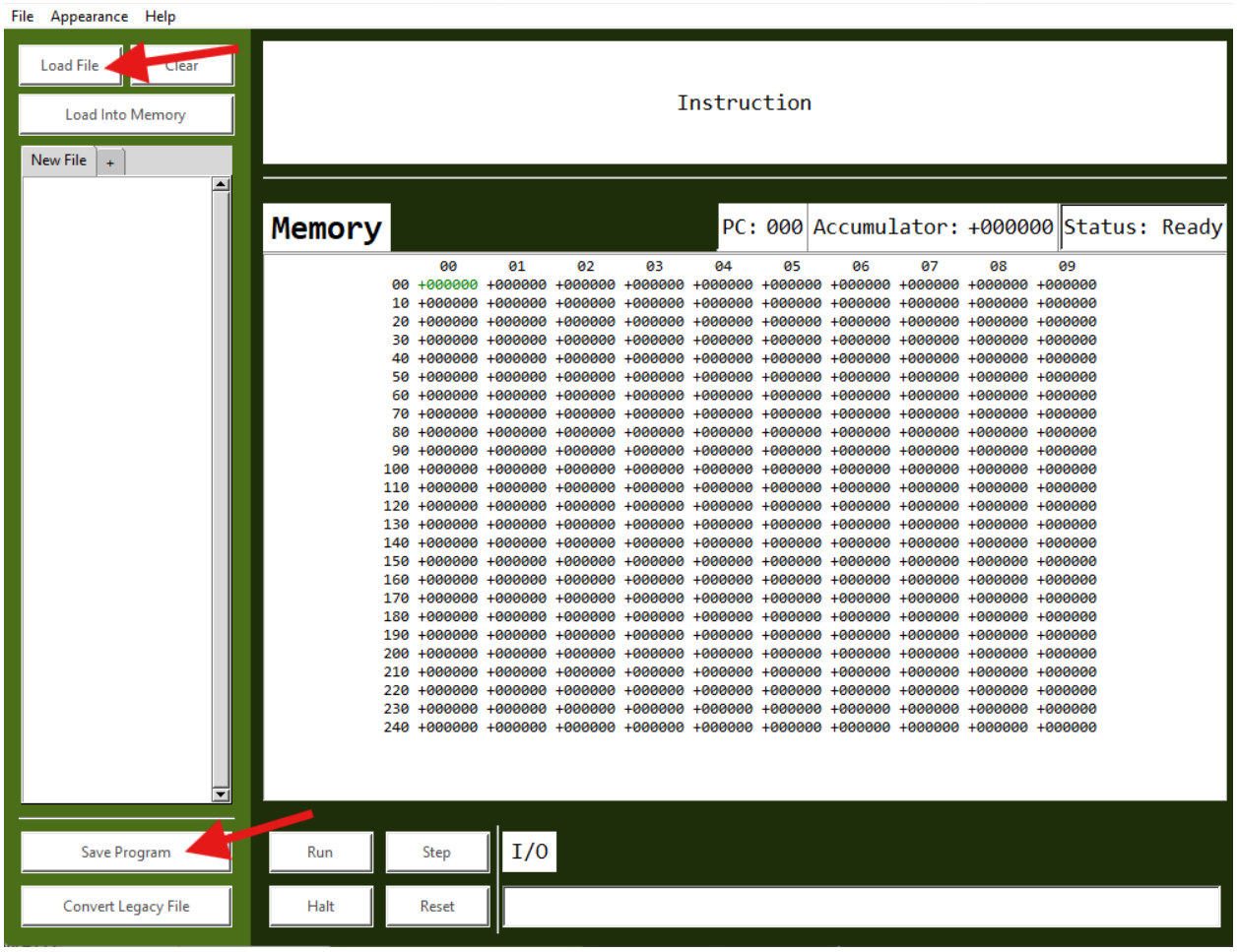
in the root directory of the project. This will open the application GUI.

Using UVSim

Loading a File into the Editor

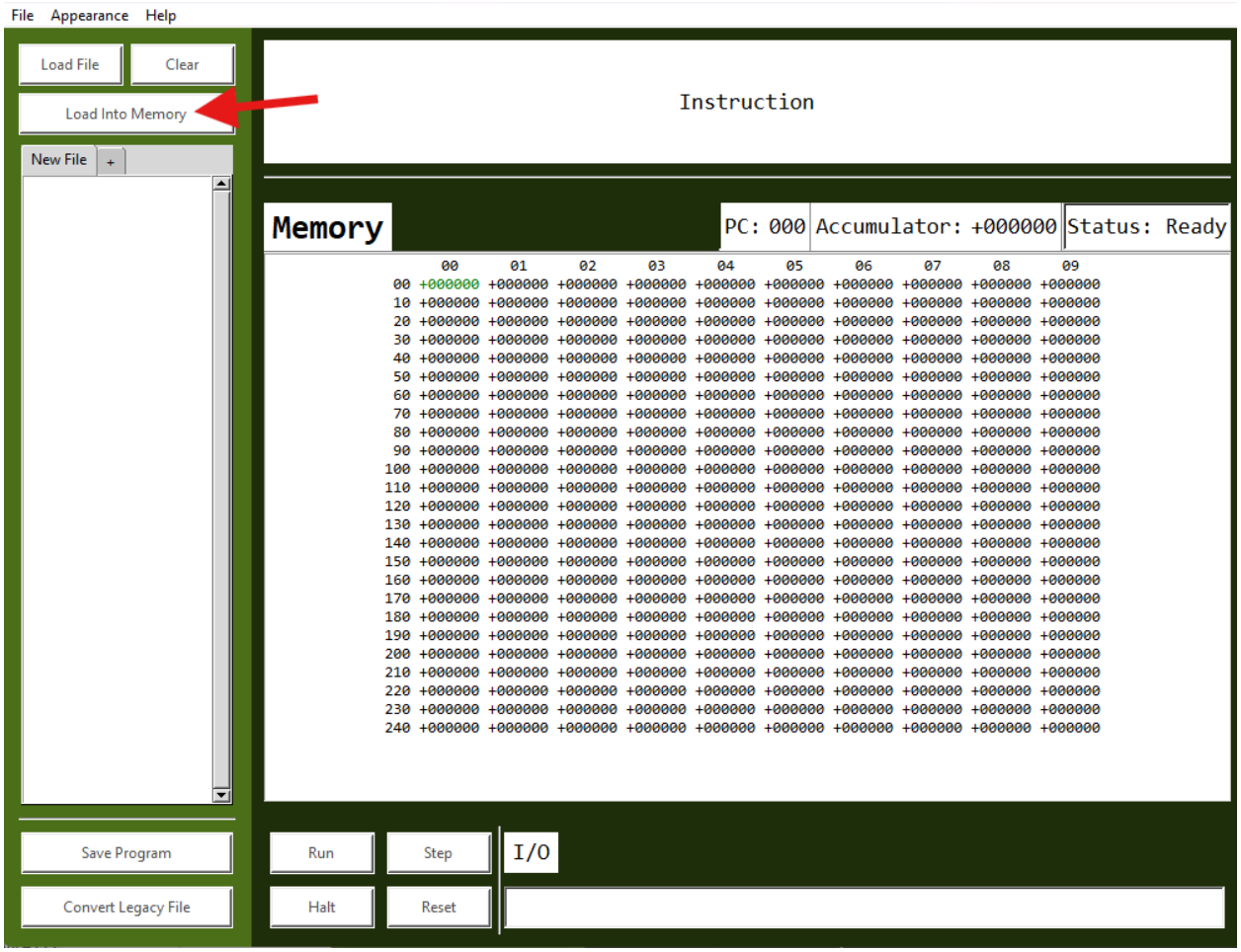
In order to load a file into the editor window, press the Load File button. This will load the contents of the file into the editor window, where you can inspect and edit the program before attempting to run it. If you make any changes you want to keep, you can save the program by pressing Save Program. The Clear button resets the editor to an empty state.

You can have up to three files open at a time. To open a new editor tab, press the (+) next to the file name. Instead of loading in a file you've already made, you can instead write a new program from scratch in the editor.



Loading Data from the Editor into Memory

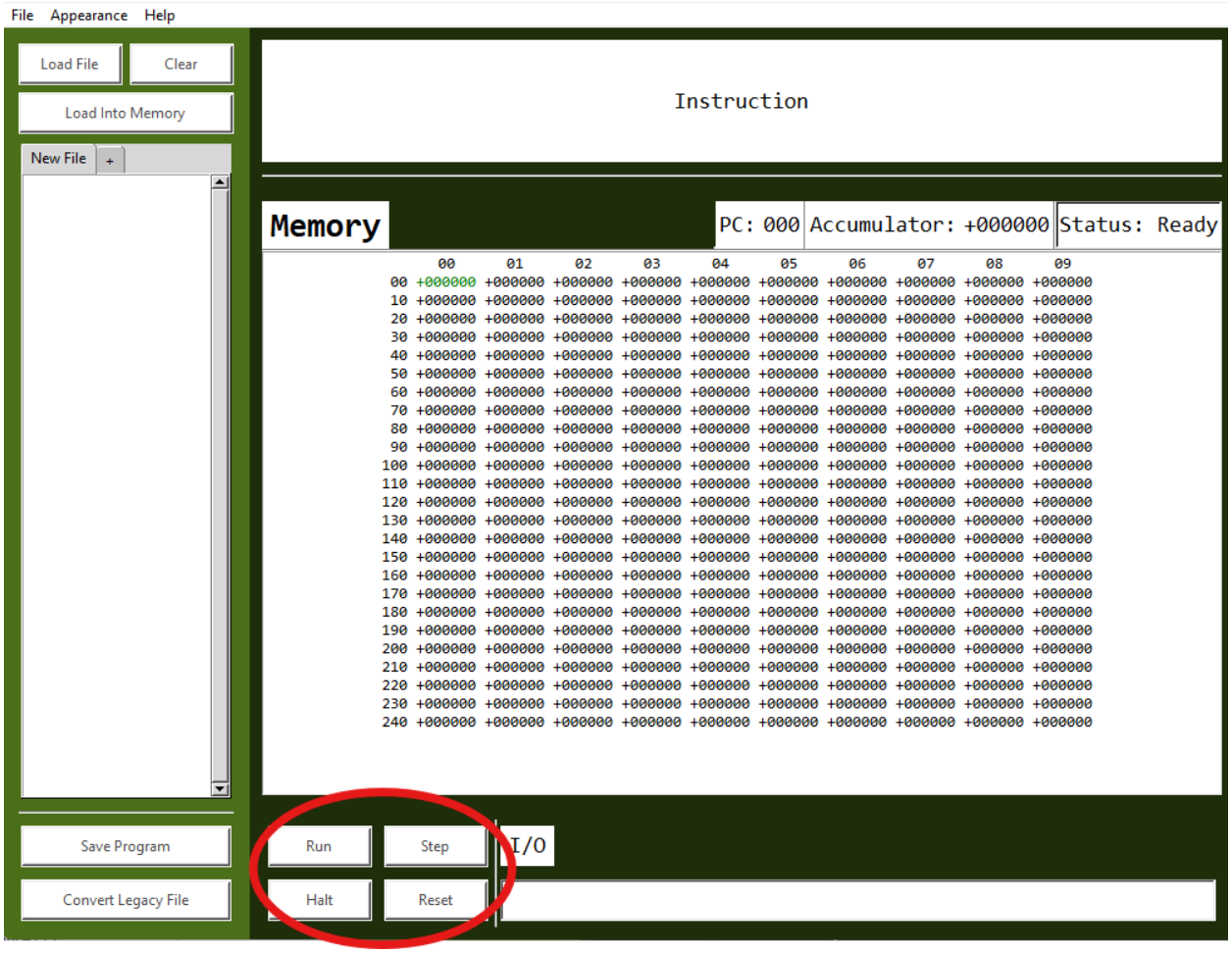
To run the program, press Load into Memory. This will take the program from the editor and attempt to load it into application memory. If there are any invalid instructions in the code, an error message will appear with the invalid instruction.



Program Execution

The **Memory** panel shows the current state of the program memory. **PC** is the Program Counter, which is the memory address of the current instruction being run. **Accumulator** shows the current value of the virtual CPU's accumulator register. **Status** shows either Ready or Halted, depending on the current state of the program.

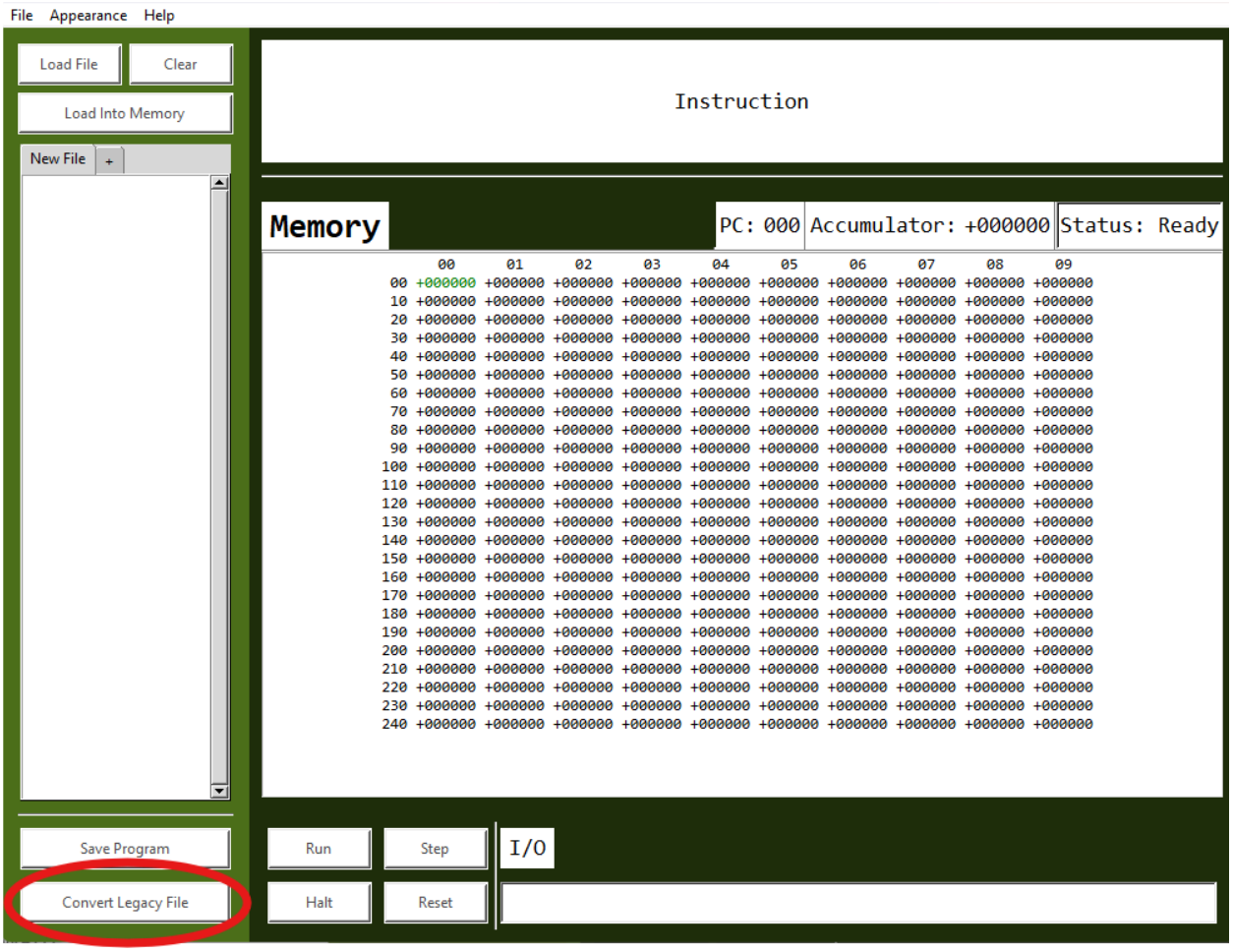
The **Instruction** panel at the top will have a short description of the current command being executed. Use the control frame at the bottom to run the program without stopping; step through each instruction individually; halt the program early; or reset the program memory, accumulator, and program counter to the empty state. The I/O panel is how you input data when the program calls for it.



Legacy File Conversion

If you still have files in the old 4-digit word format, you can convert those into the new 6-digit format. Simply press the Convert Legacy File button, and select the file you'd like to convert, and the application will make a copy of that file with each line converted from 4 digits to 6. This doesn't delete the original file in case you still need it.

Alternatively, you can load a legacy file into the editor (as described above), and when you load it into program memory, it will automatically convert the words to the new format, both in program memory and in the editor.



Use Cases & User Stories

The following User Stories describe a couple of possible high-level scenarios for interacting with the UVSim program. The Use Cases below that describe discrete, lower-level elements of functionality that should be supported.

User Stories

As a BasicML developer
I want to load my BasicML program into memory,
So I can execute it step by step and see how memory is used during execution.

As a developer,
I want to receive clear error messages if my BasicML program contains invalid instructions or memory access errors,
So I can debug my code efficiently and correct mistakes before execution.

As an educator,
I want to have a clear, transparent, and simple CPU simulator,
So I can teach my students about how CPUs and memory work

Use Cases

Use Case 1: READ

Actor: User
System: Virtual CPU (with Memory subsystem)
Goal: Read a numeric input from the keyboard and store it in a specified memory location.

Preconditions:

- A valid BasicML program is loaded.
    - **Legacy File:** Instructions are 4-digit words (e.g., `10XX`) with valid memory addresses defined by the legacy range.
    - **New File:** Instructions are 6-digit words (e.g., `010XXX`) with valid memory addresses ranging from `000` to `249`.

Main Flow:

1. The program encounters a READ instruction during execution.
    - **Legacy:** The instruction is identified as `10XX`.
    - **New:** The instruction is identified as `010XXX`.
2. The system determines the file format and extracts the target memory address accordingly.
3. The system prompts the user to enter a numeric value.
4. The user inputs a valid signed number (e.g., `+1234` or `-5678`).
5. The system validates the input and checks that the target memory address is within bounds:
    - **Legacy:** Valid addresses as per the legacy specification.
    - **New:** Must be within `000–249`.
6. The system stores the entered value in the specified memory location.
7. Execution continues with the next instruction.

Alternate Flows:

- **Invalid Input:** If the user inputs a non-numeric or out-of-range value, the system displays an error message and re-prompts.
- **Memory Out of Bounds:** If the memory address is invalid (outside the allowed range for the loaded file format), the system halts execution and displays an error.

Use Case 2: WRITE

Actor: User
System: Virtual CPU
Goal: Display the contents of a specified memory location on the screen.

Preconditions:

- A valid BasicML program is loaded (legacy or new) with a valid memory address in the instruction.

Main Flow:

1. The program encounters a WRITE instruction:
    - **Legacy:** In the format `11XX`.
    - **New:** In the format `011XXX`.
2. The system determines the file format and extracts the memory address.
3. The system retrieves the value stored at the given memory location.
4. The system displays the value on the console.
5. Execution continues with the next instruction.

Alternate Flows:

- **Memory Out of Bounds:** If the given address is outside the valid range, the system halts and displays an error.
- **Empty Memory Location:** If no data exists at the specified location, an appropriate warning is displayed.

Use Case 3: LOAD

Actor: Virtual CPU
System: Memory
Goal: Load a value from a specified memory location into the accumulator.

Preconditions:

- A valid program (legacy or new) with the LOAD instruction is loaded.
- The target memory address must be within bounds.

Main Flow:

1. The CPU encounters a LOAD instruction:
    - **Legacy:** `20XX`
    - **New:** `020XXX`

2. The system identifies the instruction and extracts the target memory address.
3. The memory subsystem retrieves the value stored at that location.
4. The value is loaded into the accumulator.
5. Execution continues with the next instruction.

Alternate Flows:

- **Memory Out of Bounds:** The system halts if the address is invalid.
- **Empty Memory Location:** A warning is displayed if the location has not been initialized.

Use Case 4: STORE

Actor: Virtual CPU
System: Memory
Goal: Store a value from the accumulator into a specified memory location.

Preconditions:

- A valid program is loaded (legacy or new).
- The accumulator holds a value to be stored, and the target address is within bounds.

Main Flow:

1. The CPU encounters a STORE instruction:
    - **Legacy:** `21XX`
    - **New:** `021XXX`
2. The system extracts the memory address from the instruction.
3. The current value in the accumulator is stored in that memory location.
4. Execution continues with the next instruction.

Alternate Flows:

- **Memory Out of Bounds:** If the address is not within the valid range, the system halts and displays an error.

Use Case 5: ADD

Actor: User
System: Virtual CPU
Goal: Add a value from a specified memory location to the value in the accumulator.

Preconditions:

- A valid BasicML program with an ADD instruction is loaded (legacy or new).
- The target address is within bounds and contains a valid number.

Main Flow:

1. The program encounters an ADD instruction:
    - **Legacy:** `30XX`
    - **New:** `030XXX`
2. The CPU extracts the target memory address.
3. The value from the memory location is retrieved.
4. The CPU adds this value to the accumulator's current value.
5. The result is stored in the accumulator.
6. Execution continues with the next instruction.

Alternate Flows:

- **Memory Out of Bounds:** If the address is invalid, halt execution with an error.
- **Integer Overflow:** If the resultant sum exceeds the word size (considering 4-digit for legacy or 6-digit for new), the system displays an error.

Use Case 6: SUBTRACT

Actor: User
System: Virtual CPU
Goal: Subtract a value from a specified memory location from the value in the accumulator.

Preconditions:

- A valid program containing the SUBTRACT instruction is loaded (legacy or new).
- The specified memory location is within bounds.

Main Flow:

1. The program encounters a SUBTRACT instruction:
    - **Legacy:** `31XX`
    - **New:** `031XXX`
2. The CPU extracts the memory location operand.
3. The value at that location is retrieved.
4. The CPU subtracts the retrieved value from the accumulator's current value.
5. The difference is stored in the accumulator.
6. Execution continues with the next instruction.

Alternate Flows:

- **Memory Out of Bounds:** Halt execution with an error if the address is invalid.
- **Integer Overflow:** Display an error if the difference exceeds allowable limits.

Use Case 7: DIVIDE

Actor: User
System: Virtual CPU
Goal: Divide the value in the accumulator by the value from a specified memory location.

Preconditions:

- A valid BasicML program with the DIVIDE instruction is loaded (legacy or new).
- The memory address is valid and contains a non-zero number.

Main Flow:

1. The program encounters a DIVIDE instruction.
    - **Legacy:** `32XX`
    - **New:** `032XXX`
2. The CPU extracts the target memory address.
3. The value from that memory location is retrieved.
4. The CPU divides the accumulator's value by the retrieved value.
5. The quotient is stored back in the accumulator.
6. Execution continues with the next instruction.

Alternate Flows:

- **Memory Out of Bounds:** Halt execution if the address is invalid.
- **Division by Zero:** The system detects division by zero and displays an error.

Use Case 8: MULTIPLY

Actor: User
System: Virtual CPU
Goal: Multiply the value in the accumulator by a value from a specified memory location.

Preconditions:

- A valid program with the MULTIPLY instruction is loaded (legacy or new).
- The memory location contains a valid number and is within bounds.

Main Flow:

1. The program encounters a MULTIPLY instruction:
    - **Legacy:** `33XX`
    - **New:** `033XXX`
2. The CPU extracts the memory address from the instruction.
3. The value at that address is retrieved.
4. The CPU multiplies the accumulator's value by the retrieved value.
5. The product is stored in the accumulator.
6. Execution continues with the next instruction.

Alternate Flows:

- **Memory Out of Bounds:** If the address is invalid, halt with an error.

Use Case 9: BRANCH

Actor: Program
System: Virtual CPU
Goal: Jump to a specific memory location unconditionally.

Preconditions:

- A valid program with the BRANCH instruction is loaded (legacy or new).
- The target memory address is within bounds:
    - **Legacy:** Typically two-digit address.
    - **New:** Three-digit address between `000` and `249`.

Main Flow:

1. The CPU encounters a BRANCH instruction:
    - **Legacy:** `40XX`
    - **New:** `040XXX`
2. The CPU extracts the target memory address.
3. The instruction pointer is set to the specified address (without auto-increment).
4. Execution continues with the instruction at the new location.

Alternate Flows:

- **Memory Out of Bounds:** If the address is invalid, the system halts and displays an error.

Use Case 10: BRANCHNEG

Actor: Program
System: Virtual CPU
Goal: Branch to a specific memory location if the accumulator's value is negative.

Preconditions:

- A valid BasicML program with the BRANCHNEG instruction is loaded (legacy or new).
- The target memory address is within bounds.

Main Flow:

1. The CPU encounters a BRANCHNEG instruction:
    - **Legacy:** `41XX`
    - **New:** `041XXX`
2. The CPU checks the accumulator:
    - If the value is negative, it extracts the memory address and sets the instruction pointer to that location.
    - If the accumulator is zero or positive, the branch is skipped.
3. Execution continues with the next instruction or at the new location, as applicable.

Alternate Flows:

- **Memory Out of Bounds:** Halt execution if the address is out of the valid range.
- **No Branch:** If the accumulator is not negative, the pointer moves to the next instruction normally.

Use Case 11: BRANCH ZERO

Actor: User
System: Virtual CPU
Goal: Branch to a specific memory location if the accumulator's value is exactly zero.

Preconditions:

- A valid program with the BRANCH ZERO instruction is loaded (legacy or new).
- The memory address specified is within bounds.

Main Flow:

1. The program encounters a BRANCH ZERO instruction:
    - **Legacy:** `42XX`
    - **New:** `042XXX`
2. The CPU evaluates the accumulator:
    - If zero, extracts the memory address and sets the instruction pointer to it.

    - Otherwise, execution continues sequentially.
3. Execution continues as directed.

Alternate Flows:

- **Memory Out of Bounds:** If the address is invalid, the system halts and displays an error.

Use Case 12: HALT

Actor: User
System: Virtual CPU
Goal: Terminate program execution cleanly.

Preconditions:

- A valid program with the HALT instruction is loaded (legacy or new).

Main Flow:

1. The CPU encounters a HALT instruction:
    - **Legacy:** `43XX` (typically the operand is ignored)
    - **New:** `043XXX`
2. The system performs any necessary final clean-up (e.g., logging, output).
3. Execution stops, and control is returned to the user.

Alternate Flows:

- There are no alternate flows for HALT; the system simply terminates execution.

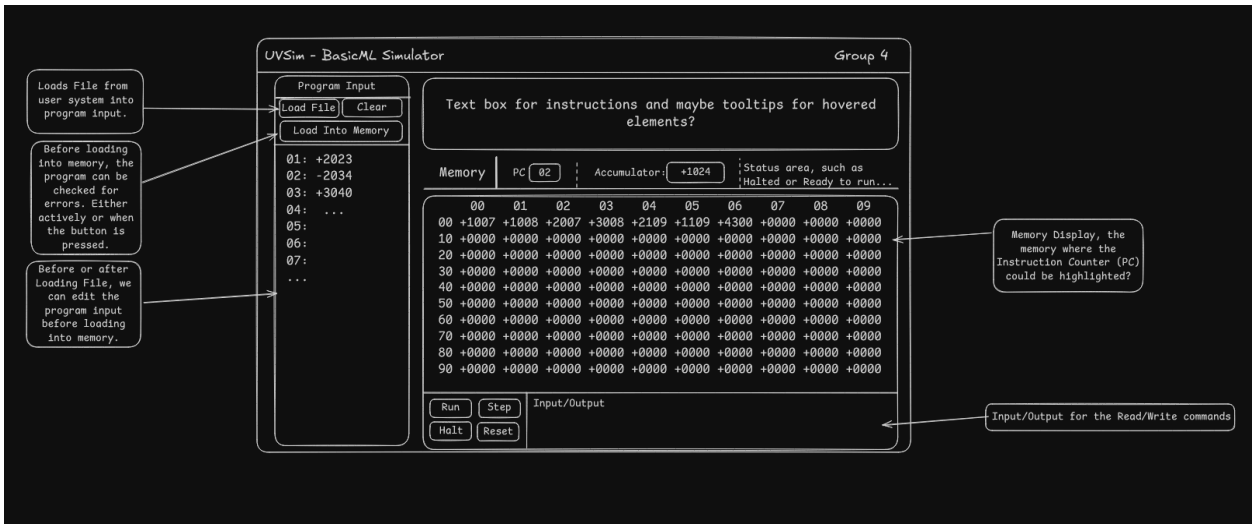
Functional Specifications

1. **Memory Allocation**: The system shall provide a memory space of 250 words, where each word is stored as a signed six-digit decimal number.
2. **Program Loading**: The system shall allow users to load a BasicML program into memory beginning at location 00.
3. **READ Instruction**: When a READ (010) instruction is encountered, the system shall prompt the user for a valid signed six-digit decimal input and store the value at the specified memory location.
4. **WRITE Instruction**: When a WRITE (011) instruction is executed, the system shall retrieve the value from the specified memory location and display it on the screen.
5. **LOAD Instruction**: When a LOAD (020) instruction is executed, the system shall copy the value from the specified memory location into the accumulator.

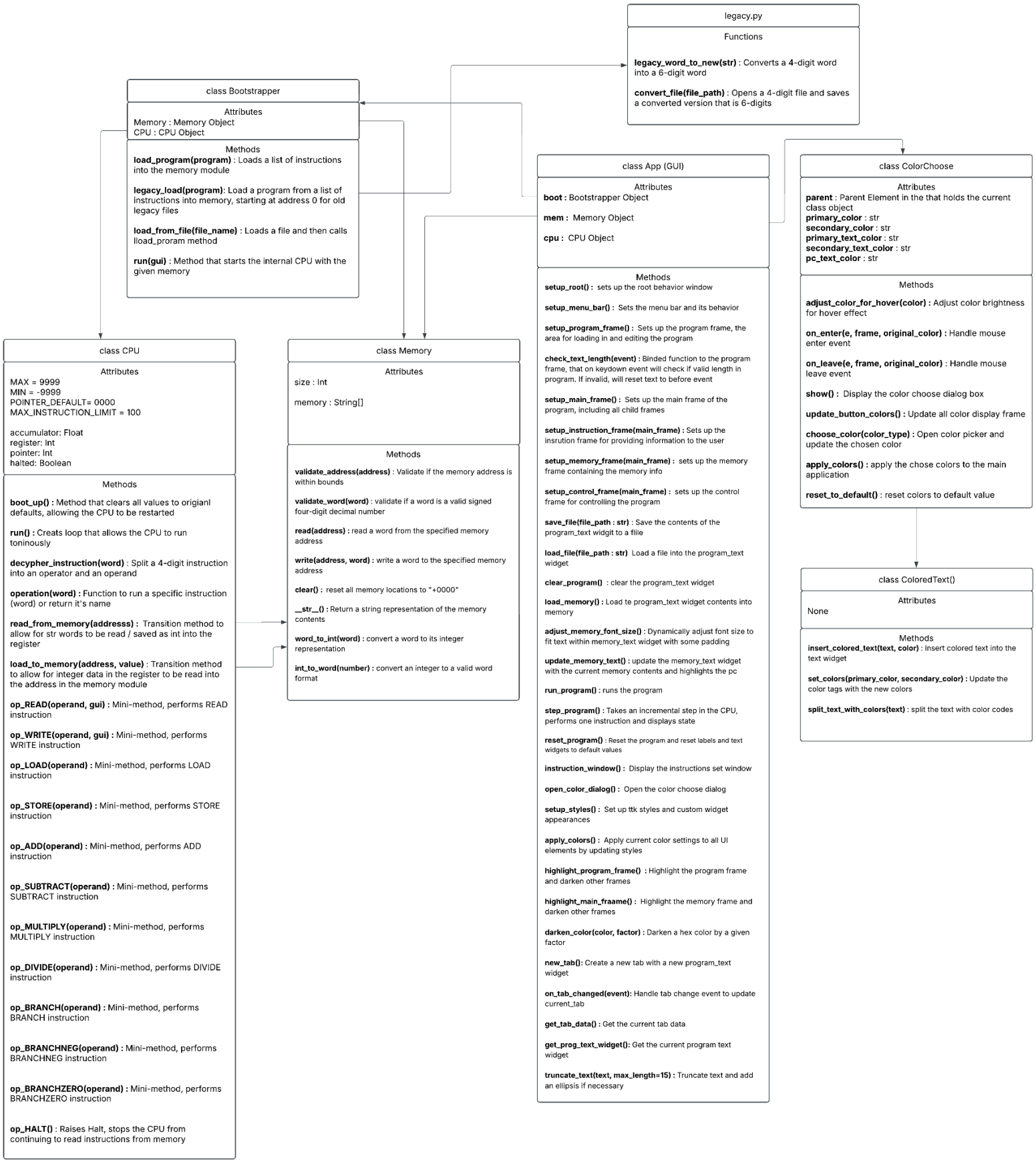
6. **STORE Instruction**: When a STORE (021) instruction is executed, the system shall copy the value from the accumulator into the specified memory location.
7. **ADD Instruction**: When an ADD (030) instruction is executed, the system shall add the value from the specified memory location to the accumulator and update the accumulator with the result.
8. **SUBTRACT Instruction**: When a SUBTRACT (031) instruction is executed, the system shall subtract the value from the specified memory location from the accumulator and update the accumulator with the result.
9. **DIVIDE Instruction**: When a DIVIDE (032) instruction is executed, the system shall divide the accumulator by the value from the specified memory location, update the accumulator with the result, and if a division by zero is attempted, the system shall halt execution and display an appropriate error message.
10. **MULTIPLY Instruction**: When a MULTIPLY (033) instruction is executed, the system shall multiply the accumulator by the value from the specified memory location and update the accumulator with the result.
11. **BRANCH Instruction**: When a BRANCH (040) instruction is executed, the system shall set the execution pointer to the specified memory location.
12. **BRANCHNEG Instruction**: When a BRANCHNEG (041) instruction is executed, the system shall set the execution pointer to the specified memory location if the accumulator contains a negative value.
13. **BRANCHZERO Instruction**: When a BRANCHZERO (042) instruction is executed, the system shall set the execution pointer to the specified memory location if the accumulator contains a zero value.
14. **HALT Instruction**: When a HALT (043) instruction is executed, the system shall immediately stop program execution.
15. **Execution Logging**: The system shall log each executed instruction and all memory changes on a display panel, showing the instruction executed, and the resulting state of memory.
16. **Legacy File Conversion**: The system shall provide a button that loads a legacy (4-digit word format) file and creates a copy of that file in the computer's file system, with each word converted into the new format (6-digit words) by prepending a `0` to both the instruction opcode and memory address. Invalid lines shall be nulled (i.e., set to `+000000`).
17. **Legacy File Loading**: When the system is loading a file from the text editor into system memory, it shall detect the legacy format (4-digit words) and dynamically convert the file into the new forma (6-digit words)

GUI Wireframes & Class Diagrams

GUI Wireframe Design



Class Diagrams



Unit Test Descriptions

Here is a list of unit tests and a short description of their purpose:

1. **CPU_init**: Tests that the CPU class can initialize without errors. Precursor to all other tests. Succeeds if no errors are thrown
2. **validate**: Tests that the CPU will raise errors successfully when trying to read invalid words. Succeeds when an error is thrown for all invalid words.
3. **LOAD**: Ensures that the CPU can receive an instruction to Load from a memory address and that the value in that address is placed in the accumulator. Succeeds when the accumulator contains the value 8 from memory[10], then the accumulator contains the value 16 from memory[11].
4. **STORE**: Ensures that the CPU can store a value into a given memory address. It will load #8 from memory[10] and store it in memory[00]. It will then read memory[00] to ensure value 8 is stored there cpu_test.txt, 2010 2100 2000 8 When the accumulator contains the value 8 from memory[00], memory location [00] contains the value 8Failure if memory location [00] is not '+0008' or if the accumulator does not contain the value 8 at the end
5. **ADD**: Ensures that the CPU can load a value, and successfully use addition in its accumulator. It loads the value 8 from memory[10] and adds the value 16 from memory[11]. Succeeds when the accumulator contains the value 24 (8+16)
6. **SUBTRACT**: Ensures the CPU can subtract. It loads the value 8 from memory[10] and subtracts the value 7 from memory[12]. Succeeds when the accumulator contains the value 1 (8 - 7)
7. **MULTIPLY**: Ensures the CPU can multiply. It loads the value 8 from memory[10] and multiplies by the value 7 from memory[12]. Succeeds when the accumulator contains the value 56 (8 * 7)
8. **DIVIDE**: Ensures the CPU can DIVIDE. It loads the value 16 from memory[12] and DIVIDES by the value 8 from memory[10]. Succeeds when the accumulator contains the value 2 (16 / 8)
9. **BRANCH**: Ensures the CPU can branch to a specific location in memory, by setting its pointer to a new value. Succeeds when the pointer is pointing at value 59
10. **BRANCHNEG**: Ensures the CPU can conditionally branch to the specific location memory[59], by setting its pointer to a 59, if the accumulator is negative. This is done by loading 8 from memory[10] and subtracting 16 from memory[11] which should evaluate to a NEGATIVE value. It should then jump to the pointer position 59. Succeeds when the pointer is pointing at value 59
11. **BRANCHZERO**: Ensures the CPU can conditionally branch to the specific location memory[59], by setting its pointer to a 59, if the accumulator is negative. This is done by loading 8 from memory[10] and subtracting 8 from memory[10] which should evaluate to ZERO. It should then jump to the pointer position 59. Succeeds when the pointer is pointing at value 59
12. **BRANCH NEG FAIL**: Ensures the CPU can avoid branching, if the accumulator is NOT negative. It does this by loading the value +8 from memory[10] and attempting to branch.

It also attempts if the accumulator is zero. Succeeds when the pointer is NOT pointing at value 59

13. **BRANCH ZERO FAIL**: Ensures the CPU can avoid branching, if the accumulator is NOT zero. It does this by loading the value +8 from memory[10] and attempting to branch. It also attempts if the accumulator is negative by subtracting 8 - 16 to make the accumulator -8 . Succeeds when the pointer is NOT pointing at value 59
14. **FULL PROGRAM**: Comprehensive test. Loads the entire test program and runs it completely. The test program contains every type of command. It has data stored in memory locations 30-49 and the program begins by branching to location 50. Multiple math operations are ran, with conditional jumping to ensure math is correct, that results in the final value of 1875 being produced. Succeeds when the final accumulator value is 1875, and the state of the memory/cpu matches cpu_test_final.txt

Future Roadmap

Features that we will be adding in the future include following:

- Expanding program length. With the current address space, we have the ability to easily support up to a thousand lines of code, and the format of the words is easy enough to expand into larger words, so we could provide support for ten thousand lines of code fairly easily.
- An additional layer to the application capable of compiling C code into BasicML. This potentially involves expanding the BasicML language operations, but it would provide deep insight into how a higher level language compiles down into machine code.
- Expand CPU memory registers. This potentially involves modifying the BasicML language further, but most CPUs have more than one register, and learning how to interact with multiple registers is important.
- Add a live-service web hosting option, so the app can be accessed remotely. This would allow students or developers to interact with the application without needing to download and run the code locally.