Design Document
Group 4
Last Updated 04/11/2025

Project Structure:

```
project/
├── pytest.toml                # Pytest configuration file
├── poetry.lock                # Dependency lock file for Poetry
├── pyproject.toml             # Python project configuration
├── docs/                      # Documentation files and diagrams
├── gui/
│   └── cpu.ico                # GUI icon file
├── src/                       # Source code for the application
│   ├── __init__.py
│   ├── boot.py
│   ├── cpu.py
│   ├── gui.py
│   ├── legacy.py
│   ├── main.py
│   ├── memory.py
└── tests/                      # Unit tests and test resources
    ├── __init__.py
    ├── test_cpu.py            # Unit tests for CPU logic
    ├── test_memory.py         # Unit tests for memory operations
    ├── cpu_test.txt
    ├── cpu_test_final.txt
    ├── cpu_test_6digit.txt
    ├── cpu_test_6digit_final.txt
    ├── cpu_test copy.txt
```

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.

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.