General Feedback

- Good effort on implementing the project
- However, there are significant areas that need improvement to meet the project requirements
- Lack of use github Project Board for tracking progress
- Lack of use of Github issues

Documentation

README.md

Feedback	Priority	Remarks
README is not formatted properly; lacks structure and organization	High	Please Check
Does not contain all the required components of a README file	High	-
Missing sections such as Introduction, Installation Instructions, Usage Examples, etc.	High	-
Minimal use of markdown formatting (headings, lists, code blocks)	Medium	-
Provide detailed instructions on how to run the application, including any dependencies or prerequisites	High	-

Design Document (`design-doc.pdf`)

Feedback	Priority	Remarks
The design document is poorly formatted and organized	High	--
Contains empty fields and original filler text not removed	High	Needs Immediate Attention
Required information is missing and hard to follow	High	Needs Immediate Attention
Does not include functional and non-functional requirements	Critical	Needs Immediate Attention
User stories are missing	Critical	Needs Immediate Attention
Use cases are not updated over the iterations and milestones	High	--

Use Case Diagram

Feedback	Priority	Remarks
The use case diagram is incomplete	`High`	--
Important actors or use cases may be missing	`High`	--
Does not reflect the updated functionalities from iterations and milestones	`High`	--

Class Diagram

Feedback	Priority	Remarks
Class diagram is missing (**Red Alert!**)	`Critical`	Needs Immediate Attention

Wireframe Diagram

Sequence Diagram

Feedback	Priority	Remarks
The sequence diagram is incomplete	Low	--
Consider detailing the main interactions between objects	Low	Advice

Other Documents

Feedback	Priority	Remarks
No documents found for functional and non-functional requirements	`Critical`	Needs Immediate Attention
No documents found for user stories	`Critical`	Needs Immediate Attention

Coding

General Observations

- There is minimal use of comments and docstrings.**
- Error handling is insufficient in several places.**
- The code structure is not modular and lacks separation of concerns.**
- The codebase lacks adherence to design principles and best practices.**
- There are room for improvements in the implementation of design patterns.**
- There are room for improvements in the implementation of OOP principles.**

Design and Architecture Suggestions**

- Adopt MVC Pattern to separate GUI, logic, and control flow
- Use Command pattern for handling instructions in `UVSim`

- Enhance OOP practices by encapsulating related functionalities within classes
- Define opcodes using an `Enum` class for clarity and maintainability
- Split large modules into smaller, focused modules (Improve Modularity)

Coding Best Practices

- Use consistent naming conventions
- Implement comprehensive error handling**
- Validate all user inputs**
- Replace magic numbers with named constants
- Provide comprehensive docstrings and comments
- Modularize code into separate modules based on functionality**

cli.py

- Same as above
- Missing docstrings and comments

main.py

- GUI callbacks do not handle exceptions
- Mixing GUI code with application logic (Design Principles, Separation of Concerns, MVC Pattern)
- Missing docstrings and comments
- Hard-coded values in GUI layout

memory_editor.py

- No input validation or error handling
- Direct manipulation of `uvsim` memory without validation (Design Principles)
- Missing docstrings and comments
- Using `input()` in a GUI context [Modify methods to accept parameters passed from the GUI instead of using `input()`]

uvsim.py

- Lack of exception handling in file reading and operations
- Opcode values are used directly; could be replaced with named constants or an `Enum` (Magic Numbers)
- Long `match` statement in `operate` method can be refactored (Design Patterns)
- Most methods lack docstrings
- Few inline comments to explain complex logic

word.py

- Nice use of operator overloading
- Lack of docstring and comments

testing.py

- Lack of test coverage for all functionalities
- Tests are written as simple functions without using a testing framework
- Tests lack docstrings explaining what they test
- Some tests lack proper assertions or have incomplete logic
- Tests do not reflect recent changes or updates in the code
- Lack of docstrings and comments

Recommendations

Recommendations for Documentation

README.md

Revise the README file to include all necessary components:

- Introduction
- Overview of the Project
- Installation Instructions
- Usage Examples
- How to Run the Application
- Dependencies or Prerequisites
- Troubleshooting Tips
- (Some snapshots or GIFs can be added for better understanding)

Design Document

- Remove all filler text and ensure all sections are properly filled
- Use clear headings and subheadings to organize content logically
- List all the functions and features the system should have [Adv: keep a separate document for functional and non-functional requirements]**
- List all Non functional requirements [Adv: keep a separate document for functional and non-functional requirements]**
- **User Story:** Write user stories as per the functionalities of the system**
    - follow the format: As a [type of user], I want [some goal] so that [some reason].
- Ensure use cases are up-to-date with the latest iterations and milestone**
- if current document is not working, consider trying other (e.g: https://github.com/rick4470/IEEE-SRS-Tempate)

Diagrams

- **Class Diagram:**
    - Follow the UML standards to create a comprehensive class diagram
        - Ref: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/

    - Update the class diagram to reflect the current state of the application

- **Use Case Diagram:**
    - Complete the diagram with more details and ensure it reflects all current functionalities

---

Code Improvement Suggestions

- Refactor the code to better utilize OOP concepts like encapsulation, inheritance, and polymorphism.
- Use appropriate design patterns such as MVC (Model-View-Controller), Command, and Singleton to improve code structure.
- Adhere to SOLID principles to make the code more maintainable and extensible.
- Add docstrings and comments to explain the purpose of classes, methods, and complex logic.
- Implement exception handling to make the program robust and user-friendly.
- Separate code into modules based on functionality to enhance readability and maintenance.
- Use enums or named constants instead of magic numbers for better code readability and maintainability.
- Improve test coverage and use a testing framework to ensure the correctness of the code.
- Refactor the GUI code to separate it from the application logic and improve modularity.
- Enhance the memory_editor module by adding input validation and error handling.
- Refactor the uvsim module to improve exception handling and replace magic numbers with named constants.
- Improve the testing module by writing comprehensive tests that cover all functionalities and use proper assertions.