Project Name: Build a Virtual CPU Emulator

Objective

The initial step was a team meeting to discuss the scope of our Virtual CPU Emulator project. At first, we will focus on making an emulator that imitates the basic functionality of a real CPU. Here's what we decided:

1. Outline Features of the Virtual CPU

The emulator will simulate a simplified CPU, focusing on essential features without overwhelming complexity.

Core Features:

- **Registers**: Implement basic CPU registers such as:
    - Program Counter (PC)
    - Accumulator (ACC)
    - General-purpose registers (e.g., R0, R1)
- **Memory**: Allocate memory space to store instructions and data.
- **Instruction Set**: Develop a small set of CPU instructions, including:
    - **Arithmetic**: ADD, SUB, MUL, DIV
    - **Logical**: AND, OR, NOT
    - **Data Movement**: MOV, LOAD, STORE
    - **Control Flow**: JMP (jump), CMP (compare), JZ (jump if zero), JNZ (jump if not zero)
- **Execution Cycle**: Implement the core CPU cycle: Fetch, Decode, Execute.
- **Stack**: Implement a basic stack for function calls and nested operations.
- **Error Handling**: Detect and report errors, such as invalid instructions or memory overflows.

Gathering Resources:

We researched and collected materials to understand the foundational concepts of CPU emulation, including:

- **CPU Architecture Basics:** Resources on how a real CPU functions, covering components like the ALU (Arithmetic Logic Unit), registers, and the fetch-decode-execute cycle.

- **Python Resources for Emulation:** We explored Python libraries such as struct for handling binary data and raggares for command-line functionality.
- **Documentation & Emulation Examples:** We reviewed documentation on creating virtual environments, sample emulators, and basic assembly language tutorials to understand low-level operations.
- **Community Forums and Support:** We joined online forums and communities to seek advice and share knowledge with other developers working on similar projects. We also bookmarked online documentation, tutorials, and videos to ensure all team members have a common knowledge base to refer to during the project.

2. Choose a Programming Language and Tools

Programming Language:

- **Python**: Ideal for ease of development, flexibility, and rapid prototyping. Offers strong libraries for memory handling and optional graphical interfaces.
- **C++**: Suitable for a low-level approach, enabling a closer simulation of CPU behavior with better performance, though development may be more complex.

Decision: **Python** is recommended for this project due to its simplicity, readability, and extensive libraries, making it easier to focus on learning CPU fundamentals. However, if performance is critical, C++ can be considered.

Tools:

- **IDE**: Visual Studio Code (cross-platform) or PyCharm for Python.
- **Version Control**: Git for tracking changes and collaborating. Set up a GitHub repository for version control and remote storage.
- **Testing Framework**:
    - **Python**: pytest for unit and integration tests.

Additional Libraries:

- **Python**:
    - NumPy (for efficient memory handling, if needed)
    - Tkinter or PyGame for optional GUI development

3. Set up Version Control

1. **Create a GitHub Repository**: Set up a GitHub repository to store the codebase and collaborate with team members.

2. **Define Git Workflow**: Establish a branching strategy, e.g., main for stable code, feature branches for development, and pull requests for code reviews.
3. **Folder Structure**:
    - **src/**: Contains all source code.
    - **docs/**: Documentation on project architecture, setup, and user guide.
    - **tests/**: Unit and integration test files.
    - **examples/**: Sample programs and instruction sets for testing the emulator.
4. **README.md**: Include project details, objectives, and setup instructions.
5. **Git Ignore File**: Use a .gitignore file to exclude environment files, compiled binaries, and other non-source files.