Project Name: Build a Virtual CPU Emulator

Objective

Our team convened to outline the goals and scope of the Virtual CPU Emulator project. The focus is on creating an emulator that replicates core CPU functions in a simplified form, making it accessible yet effective for learning and demonstration.

Core Features

- **Registers**: Key CPU registers will include:
  - **Program Counter (PC)**: Keeps track of the next instruction.
  - **Accumulator (ACC)**: Holds intermediary results.
  - **General-purpose Registers**: Implement R0, R1, and possibly more for added flexibility.
- **Memory**: Allocate a segment of memory to hold both data and instructions, allowing for dynamic operations and program storage.
- **Instruction Set**: Define a minimal yet functional set of instructions:
  - **Arithmetic**: Basic operations like `ADD`, `SUB`, `MUL`, and `DIV`.
  - **Logical**: Support for `AND`, `OR`, and `NOT` to handle basic decision-making.
  - **Data Movement**: Commands such as `MOV`, `LOAD`, and `STORE` for data transfer between memory and registers.
  - **Control Flow**: Control instructions like `JMP` (unconditional jump), `CMP` (compare), `JZ` (jump if zero), and `JNZ` (jump if not zero).
- **Execution Cycle**: Implement the primary CPU cycle, encompassing **Fetch**, **Decode**, and **Execute** stages to process instructions systematically.
- **Stack Management**: Add a simple stack to support function calls and nested expressions.
- **Error Handling**: Build mechanisms to catch and report issues like invalid instructions, memory access errors, and stack overflows.

Project Scope Definition

Our team kicked off by discussing the scope of our Virtual CPU Emulator. The goal is to emulate the primary functions of a physical CPU with simplicity yet accuracy. Here’s the outline we agreed upon:

- **Core Operations**: The emulator will support fundamental arithmetic and logic operations like `ADD`, `SUB`, and various logical comparisons to model a CPU’s essential functionality.
- **Memory Management**: Basic memory handling will allow reading and writing operations, simulating how a CPU interacts with its memory to store and retrieve data.
- **Instruction Set**: We plan to create a streamlined Instruction Set Architecture (ISA) that defines the key operations the emulator will support (e.g., `LOAD`, `STORE`, etc.).
- **I/O Handling**: Basic input/output operations will be included to simulate simple I/O interactions, broadening the emulator’s functionality.

Resource Gathering

We gathered comprehensive resources to understand the core principles of CPU emulation, including:

- **CPU Architecture Fundamentals**: Resources covering core CPU components, like the Arithmetic Logic Unit (ALU), registers, and the fetch-decode-execute cycle.
- **Python Libraries for Emulation**: We looked into Python modules such as `struct` for binary data handling and `argparse` for command-line functionality, both of which will play key roles in simulating low-level operations.
- **Documentation and Emulation Examples**: We reviewed guides on virtual environments, example emulator projects, and assembly language basics to help us understand CPU operations from a low-level perspective.

Selection of Programming Language and Tools

- **Programming Language**:

    - **Python**: Selected for its readability, ease of use, and wide array of libraries suited for prototyping and emulation tasks.
    - **C++**: Considered for future enhancements requiring optimized performance but set aside for the initial phase due to Python's learning-oriented focus.
- **Tools**:
    - **IDE**: Preferred options include **VS Code** for its cross-platform compatibility and **PyCharm** for Python-specific features.
    - **Version Control**: **Git** for tracking changes and collaborating; GitHub for version control and remote backups.
    - **Testing Framework**:
        - **pytest**: For unit testing, ensuring each CPU function behaves as expected.
- **Additional Python Libraries**:
    - **NumPy**: To manage memory efficiently if needed.
    - **Tkinter or PyGame**: Optional, for creating a GUI to visualize the emulator's functions.

Set up Version Control

- **Create GitHub Repository**: Set up a GitHub repository to store the codebase and facilitate team collaboration.
- **Define Git Workflow**: Use a branching strategy with `main` for stable releases, feature branches for ongoing development, and pull requests for review.
- **Folder Structure**:
    - **src/**: Holds all source code.
    - **docs/**: Includes documentation covering setup, architecture, and user guides.
    - **tests/**: Contains unit tests for various CPU functions.
    - **examples/**: Sample programs and test cases for the emulator.
- **README.md**: Provide an overview of the project's objectives, setup instructions, and usage guidelines.
- **.gitignore**: To exclude unnecessary files, such as environment settings, temporary build files, and other non-essential items from version control.

Week 1 Summary:

By the end of the first week, we had laid a solid foundation for the project, with a clear scope, resource collection, development environment setup, and organized version control system. This structured beginning ensures a smooth development process, allowing us to focus on building the core emulator functions in the upcoming weeks.