Week 1: Project Planning & Setup

Objective: Define project scope, gather resources, and set up the development environment.

- **Tasks:**
  - Outline the features of the virtual CPU:
    - Examples include basic arithmetic, memory management, input/output operations, branching, and pipelining.
  - Select a programming language and tools:
    - Python is beginner-friendly, while C++ offers better performance for complex simulations.
    - Use tools like GitHub for version control and IDEs such as VSCode or PyCharm.
  - Initialize version control with Git to track changes and facilitate collaboration.
- **Outcome:** A clear understanding of the project's scope and tools, along with a basic project directory initialized in Git.

Week 2: Instruction Set Architecture (ISA)

Objective: Design the instruction set architecture, the foundation of your virtual CPU.

- **Tasks:**
  - Define basic instructions (e.g., ADD, SUB, LOAD, STORE) with their functionality.
  - Document instruction formats:
    - R-type for register operations.
    - I-type for immediate operations.
    - J-type for jumps.
  - Develop a simple assembler:

        - Converts assembly instructions into machine code that your CPU can process.
- **Outcome:** A documented ISA and a basic assembler to translate assembly language to machine-readable code.

Week 3: Basic CPU Components

Objective: Implement the core components of the virtual CPU.

- **Tasks:**
    - Build an **Arithmetic Logic Unit (ALU)**:
        - Handles basic arithmetic (e.g., ADD, SUB) and logic operations.
    - Implement **general-purpose registers**:
        - Used for temporarily storing data during computations.
    - Create a **program counter**:
        - Keeps track of the current instruction to execute.
    - Create an **instruction register**:
        - Stores the current instruction being executed.
- **Outcome:** The core hardware-like components of the CPU are implemented in software.

Week 4: Instruction Execution

Objective: Develop the fetch-decode-execute cycle for processing instructions.

- **Tasks:**
    - Implement **instruction fetching**:
        - Fetches instructions from memory based on the program counter.
    - **Decode** instructions:
        - Interprets the opcode and operands to understand what operation to perform.
    - **Execute** instructions:
        - Use the ALU, registers, and other components to perform operations like ADD or LOAD.
    - Test with simple programs:

        - Create small programs in your assembler to verify the cycle's functionality.
- **Outcome:** A working instruction cycle that can process basic programs.

Week 5: Memory Management

Objective: Implement a simulated memory system for the virtual CPU.

- **Tasks:**
    - Create a simulated **memory space**:
        - Typically an array or list in Python to mimic RAM.
    - Implement **read/write operations**:
        - Methods to load data from and store data into memory.
    - Handle **address mapping** and **memory segmentation**:
        - Define how memory is divided and accessed by programs.
- **Outcome:** A functional memory management system to support the CPU's operations.

Week 6: I/O Operations

Objective: Enable basic input/output operations to interact with the virtual CPU.

- **Tasks:**
    - Implement simulated **I/O devices**:
        - Keyboard for input and a display for output.
    - Create **I/O instructions**:
        - Special instructions to handle input (e.g., reading a value) and output (e.g., printing a result).
    - Test with I/O-intensive programs:
        - Write programs that utilize these I/O features, such as echoing user input to the display.
- **Outcome:** The CPU can now perform basic interaction with external devices.

Week 7: Advanced Features

Objective: Add advanced capabilities to make the virtual CPU more realistic.

- **Tasks:**
    - Implement **branching and control flow instructions**:
        - Examples: conditional jumps (e.g., branch if zero) and loops.
    - Add support for **subroutines**:
        - Allow programs to call reusable code blocks and return to the caller.
    - Handle **interrupts**:
        - Mechanism for responding to external events (e.g., I/O or error handling).
    - Integrate a **pipeline mechanism**:
        - Simulate pipelined execution, where multiple instructions are processed simultaneously at different stages.
- **Outcome:** The virtual CPU now supports complex operations, increasing its realism and capability.