Efficient CPU Design with Multiple Addressing Modes

1 CPU Structure Selection

Choice: Register-based CPU Architecture
Justification: A register-based CPU offers flexibility and efficiency compared to accumulator-based (limited to one register) and stack-based (reliant on memory-intensive push/pop operations) designs. Registers reduce memory access latency by storing intermediate results, improving execution speed. Modern CPUs (e.g., x86, ARM) predominantly use register-based designs due to their balance of performance and scalability.
Instruction Flow in Pipeline: The CPU uses a 5-stage pipeline (Fetch, Decode, Execute, Memory Access, Write-back). Registers facilitate operand access in the Execute stage, minimizing memory stalls.

2 Instruction Format Design

Proposed Format: A 32-bit instruction word supporting zero, one, two, and three-addressing modes.
Structure:

- **Opcode (6 bits):** Identifies the operation (e.g., ADD, LOAD, JMP), supporting up to 64 instructions.
- **Addressing Mode Flags (2 bits):** Specifies the mode (00: Immediate, 01: Direct, 10: Indirect, 11: Indexed).
- **Register Fields (3 $\times$ 5 bits):** Three 5-bit fields (R1, R2, R3) for source/destination registers (up to 32 registers).
- **Immediate/Offset Field (9 bits):** Used for immediate values or memory offsets in applicable modes.

Supported Addressing Modes:

- **Zero-address:** Stack-like operations (e.g., `ADD` pops two operands, pushes result).
- **One-address:** Accumulator-like (e.g., `LOAD R1` loads into R1).
- **Two-address:** Register-to-register (e.g., `ADD R1, R2` $\rightarrow$ R1 = R1 + R2).
- **Three-address:** Full flexibility (e.g., `ADD R1, R2, R3` $\rightarrow$ R1 = R2 + R3).

Optimization: Variable-length encoding is avoided to simplify fetch/decode stages, reducing pipeline delays. The 32-bit fixed format balances flexibility with fetch efficiency.

3 Integration of Addressing Modes

- **Immediate:** Data is embedded in the instruction (e.g., `MOV R1, #5`). Fast but limited by the 9-bit field size.
- **Direct:** Memory address is specified (e.g., `LOAD R1, 0x100`). Simple but requires memory access.
- **Indirect:** Address is in a register (e.g., `LOAD R1, [R2]`). Flexible for pointers but adds a register fetch cycle.
- **Indexed:** Base register + offset (e.g., `LOAD R1, [R2 + 4]`). Ideal for arrays, balancing speed and complexity.

Trade-offs:

- **Speed:** Immediate is fastest; Indirect/Indexed add latency due to additional fetches.
- **Memory Efficiency:** Direct/Indexed use memory efficiently for large data; Immediate wastes bits for small values.
- **Complexity:** Indirect/Indexed increase control unit complexity but enhance flexibility.

4 Control Unit Design

Type: Hardwired control unit for speed, with microprogrammed fallback for complex instructions.
Functions:

- **Instruction Decoding:** Parses opcode and addressing mode flags, routing signals to ALU, registers, or memory.
- **Execution Flow:** Manages pipeline stages, resolving hazards (e.g., data dependencies) with forwarding and stalling.
- **Address Resolution:** Computes effective addresses (e.g., base + offset for Indexed mode) using an ALU subunit.

Implementation: A finite state machine (FSM) sequences operations, with parallel decoding of addressing modes to minimize latency.

5 Techniques to Optimize Performance

- **Minimize Execution Time:** Register-based design reduces memory access; pipelining overlaps instruction stages.
- **Reduce Memory Stalls:** Data forwarding bypasses memory writes; a small cache (assumed) buffers frequent accesses.
- **Optimize Control Flow:** Branch prediction (simple static predictor) reduces pipeline flushes for jumps.

- **Address Pipeline Bottlenecks:** Hazard detection unit stalls the pipeline only when necessary, preserving throughput.

6 Comparative Analysis

Feature	**Traditional (Accumulator)**	**Traditional (Stack)**	**Proposed (Register-based)**
Memory Access	High (single register)	High (stack ops)	Low (multiple registers)
Instruction Size	Small (1-address)	Small (0-address)	Moderate (2-3 address)
Execution Speed	Moderate	Slow (memory-bound)	High (register ops)
Control Unit Complexity	Low	Moderate	Moderate-High

Table 1: Comparative Analysis of CPU Designs

Efficiency Improvements: Reduced memory latency and faster operand access compared to accumulator/stack designs.
Trade-offs: Larger instruction size (32 bits vs. 16-bit accumulator) increases fetch time but is offset by pipeline efficiency.

7 Diagrams

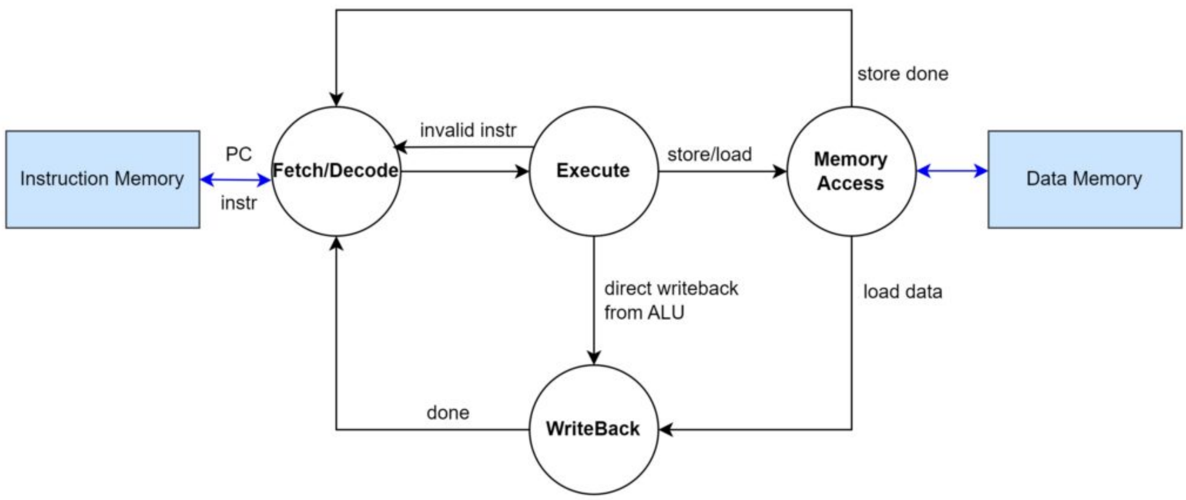

Figure 1: CPU Organization: 5-Stage Pipeline

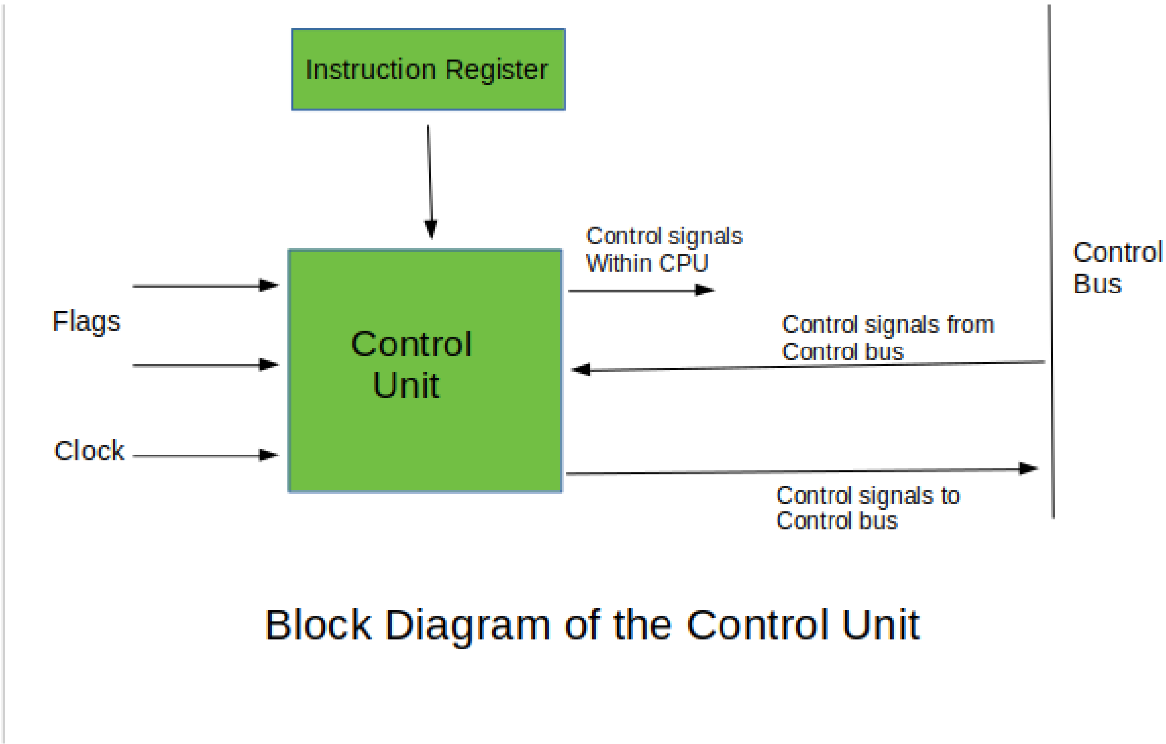

Figure 2: Block Diagram Of Control Unit

This register-based CPU design optimizes performance by leveraging registers, a flexible instruction format, and an efficient control unit. It balances trade-offs between speed, complexity, and memory efficiency, aligning with modern architectural principles while meeting the assignment's objectives.

8 Answers to Complex Problem-Solving Questions

1. **Efficient Instruction Execution Across Addressing Modes:** The register-based design with a flexible 32-bit format allows quick operand access (registers) and supports diverse modes (Immediate to Indexed), reducing memory bottlenecks.

2. **Trade-offs Considered:** Larger instruction size enhances flexibility but increases fetch/decode complexity; mitigated by pipelining and a fast control unit.

3. **Control Unit Management:** Parallel decoding and FSM sequencing ensure smooth handling of modes and pipeline stages.

4. **Performance Challenges:** Data hazards and branch mispredictions are addressed with forwarding and prediction, though cache misses remain a limitation (assumes basic cache).

5. **Alignment with Industry Standards:** The design mirrors modern RISC architectures (e.g., ARM) with register focus, pipelining, and flexible addressing, though it simplifies some features (e.g., no out-of-order execution).