### **Key Components of an Accumulator-based CPU:**

1. **Accumulator (ACC)**:  
   * A special-purpose register used to store intermediate results during computations.
   * The CPU performs most operations between the accumulator and other registers or memory locations.
2. **Arithmetic Logic Unit (ALU)**:  
   * Performs arithmetic (addition, subtraction, etc.) and logical (AND, OR, NOT, etc.) operations on the values in the accumulator.
   * The result is typically stored back into the accumulator.
3. **Registers**:  
   * Apart from the accumulator, there might be general-purpose or special-purpose registers, but the accumulator is the primary one used for computation.
4. **Memory**:  
   * Holds data and instructions. The CPU fetches instructions from memory, and the data for operations is often loaded into the accumulator.
5. **Control Unit**:  
   * Decodes and executes instructions by sending control signals to the ALU, registers, and memory.

### **Working of an Accumulator-based CPU:**

1. **Loading Data**:  
   * The CPU loads data from memory into the accumulator. This might involve a specific instruction like LOAD in an instruction set architecture (ISA).
2. **Performing Operations**:  
   * The ALU performs operations with the data in the accumulator. For example, an ADD instruction would add the contents of the accumulator with another operand (either from memory or another register).
3. **Storing Results**:  
   * After the operation, the result is usually stored back into the accumulator. If needed, the result can be moved to a memory location or another register.
4. **Example**:  
   * **LOAD A**: Load the value at memory location A into the accumulator.
   * **ADD B**: Add the value at memory location B to the accumulator.
   * **STORE C**: Store the value in the accumulator into memory location C.

### **Pros and Cons of Accumulator-based Architecture:**

#### **Pros:**

* **Simplicity**: The design is simple because there are fewer registers and operations are centralized around the accumulator.
* **Reduced Instruction Set**: Operations typically involve the accumulator, making instructions simpler.

#### **Cons:**

* **Limited Parallelism**: Since operations are focused on the accumulator, it limits parallelism and can create a bottleneck for complex tasks.
* **Speed**: If the accumulator is heavily used, it can cause delays as each instruction requires reading or writing from/to the accumulator.

### **Example Instruction Set (hypothetical):**

* **LOAD R**: Load value from register R into the accumulator.
* **ADD R**: Add the value in register R to the accumulator.
* **SUB R**: Subtract the value in register R from the accumulator.
* **STORE R**: Store the accumulator's value into register R.
* **JUMP**: Jump to another instruction based on the program counter.

## **🔧 Accumulator-Based Architecture (1-Operand Machine)**

### **✅ Key Characteristics:**

* **Accumulator (ACC)** holds all intermediate results.
* **Instructions** are of the form I = op.adr, where:  
  + op = operation code (e.g., LOAD, ADD)
  + adr = memory address
* **Only one operand** (in memory); the second implicit operand is the accumulator.

## **🔄 Instruction Cycle (Step-by-Step)**

Let’s denote:

* PC = Program Counter (holds address of the next instruction)
* IR = Instruction Register (holds fetched instruction)
* AR = Address Register (holds address part of instruction)
* ACC = Accumulator
* M[ ] = Memory

### **Step 1: Fetch Instruction**

* IR ← M[PC] → Fetch instruction from memory at PC
* PC ← PC + 1 → Increment PC
* op ← IR[opcode bits]
* AR ← IR[address bits]

### **Step 2: Decode & Execute**

#### **Common Instructions:**

1. **LOAD adr**
   * ACC ← M[adr]
2. **ADD adr**
   * ACC ← ACC + M[adr]
3. **SUB adr**
   * ACC ← ACC - M[adr]
4. **STORE adr**
   * M[adr] ← ACC
5. **JUMP adr**
   * PC ← adr

## **🧠 Example Execution**

Assume memory looks like this:

| **Address** | **Contents** |
| --- | --- |
| 100 | LOAD 200 |
| 101 | ADD 201 |
| 102 | STORE 202 |
| 200 | 5 |
| 201 | 3 |
| 202 | 0 |

### **Execution:**

1. **PC = 100** → Fetch LOAD 200, set AR = 200, IR = LOAD  
   * ACC ← M[200] → ACC = 5
2. **PC = 101** → Fetch ADD 201, set AR = 201, IR = ADD  
   * ACC ← ACC + M[201] → ACC = 5 + 3 = 8
3. **PC = 102** → Fetch STORE 202, set AR = 202, IR = STORE  
   * M[202] ← ACC → M[202] = 8

## **📝 Summary**

* **Instruction format**: op.adr
* **Execution focuses on the ACC** for all calculations.
* **Efficient for simple systems**, but limits flexibility (only one operand).