Accumulator-Based CPU and Single-Address Instructions

Key Concept: Accumulator-Based CPU & Single-Address Instructions

In typical computations such as $Z := X + Y$, three operands are involved:

- Operand 1: X
- Operand 2: Y
- Destination: Z

However, in **accumulator-based CPUs**, each instruction operates on:

- One operand from memory.
- The other operand is **implicitly the accumulator (AC)**.

Thus, a computation like $Z := X + Y$ must be broken into multiple instructions.

Step-by-Step Breakdown

HDL Format	Assembly Language	Description
`AC := M(X)`	`LD X`	Load value of X into accumulator
`DR := AC`	`MOV DR, AC`	Copy accumulator into Data Register
`AC := M(Y)`	`LD Y`	Load value of Y into accumulator
`AC := AC + DR`	`ADD`	Add value from DR to accumulator
`M(Z) := AC`	`ST Z`	Store result from accumulator to memory location Z

Table 1: Instruction Breakdown for $Z := X + Y$ in Single-Address CPU

Instruction Consideration and Optimization

Instructions generally follow the form:

$$\text{AC} := f_i(\text{AC}, M(\text{adr}))$$

This means:

- Perform some operation f_i (e.g., `ADD`, `SUB`) between:
  - The value in the accumulator (AC)
  - A memory value $M(\text{adr})$
- Store the result back in the accumulator

Implications

1. An extra register (e.g., DR) is often needed to hold intermediate results.
2. Instruction decoding becomes more complex due to memory references.
3. Performance is impacted by:
   - Increased number of memory accesses.
   - More steps needed to complete a single logical instruction.

Summary: Why This Matters

- Accumulator-based CPUs require complex operations to be decomposed into simpler steps.
- HDL (Hardware Description Language) represents internal hardware-level operations.
- Assembly language is what the programmer writes to achieve these operations.
- Efficient execution depends on minimizing memory access and data movement.

Final Simplified Example: $Z := X + Y$

HDL:

```
AC := M(X)        // Load X
DR := AC          // Store X in DR
AC := M(Y)        // Load Y
AC := AC + DR     // Add X (from DR) to Y (in AC)
M(Z) := AC        // Store result in Z
```

Assembly:

```
LD X              // Load X into AC
MOV DR, AC        // Save X to DR
LD Y              // Load Y into AC
ADD               // Add DR to AC
ST Z              // Store result into Z
```