# Addressing Modes in Computer Architecture

## What is an Addressing Mode?

An **addressing mode** specifies how the CPU obtains the value $V(X)$ of an operand $X$ during the execution of an instruction.

- The **operand** $X$ is the data the instruction works on.
- The **value of operand** is $V(X)$.
- The instruction includes an **address field** that helps locate $V(X)$.

Different addressing modes affect:

- The **speed** of operand access.
- The **flexibility** in modifying operands.
- The **complexity** of the instruction.

## Types of Addressing Modes

### 1. Immediate Addressing

The operand value $V(X)$ is specified directly in the instruction.

- No memory lookup is required.
- Operand $X$ is a constant.
- Fastest access time.

**Example:**

- High-level: `A := 99`
- Intel 8085: `MVI A, 99`
- Motorola 68000: `MOVE #99, D1` (D1 = A)

### 2. Direct Addressing

The instruction provides the address of the operand $X$, i.e., $V(X)$ is stored at a known location.

- One memory lookup is needed.
- Instruction field holds the address of $X$.
- $V(X)$ can be changed without altering the instruction.

**Example:**

- High-level: `A := B`
- Intel 8085: `MOV A, B`
- Motorola 68000: `MOVE D2, D1` (D1 = A, D2 = B)

### 3. Indirect Addressing

The instruction specifies a location $W$, which contains the address $X$ of the operand $V(X)$.

- Two memory lookups are required: first to get $X$ from $W$, then $V(X)$ from $X$.
- Very flexible — change $W$ to point to different operands dynamically.

**Example:**

- High-level: `LOADN W`
- Meaning: W $\rightarrow$ X $\rightarrow$ V(X)

## Visual Summary of Addressing Modes

| Mode | Address Field Contains | Access Steps | Flexibility | Speed |
|---|---|---|---|---|
| Immediate | Operand Value | 1 (use directly) | Low | Fastest |
| Direct | Address of Operand | 1 (memory fetch) | Medium | Fast |
| Indirect | Address of Address | 2 (W $\rightarrow$ X $\rightarrow$ V(X)) | High | Slowest |

## Mnemonic Instruction Examples

- `LOADI 99` *(Immediate Addressing)*
- `LOAD X` *(Direct Addressing)*
- `LOADN W` *(Indirect Addressing)*

## What is Relative Addressing?

Relative addressing is an **address construction technique** used in computer instructions where the complete memory address of the operand is not directly specified in the instruction.

Instead, the instruction provides a **displacement** or **offset** $D$, and the processor uses one or more CPU registers $R_1, R_2, \ldots, R_k$ to compute the **effective address** $A$ of the operand.

### Effective Address Calculation

In most cases, the effective address is calculated as:

$$A = R + D$$

where:

- $A$: Effective address of the operand,
- $R$: Value in a CPU register (base or index),
- $D$: Displacement or offset specified in the instruction.

## Why Not Use Direct Addressing?

Direct addressing requires the full address of the operand to appear in the instruction. This leads to:

- Larger instruction size,
- Inflexibility in code relocation,
- Higher memory usage.

## Advantages of Relative Addressing

1. **Reduced Instruction Length**:
   - Only the offset $D$ is stored in the instruction.
   - Saves space and improves instruction density.
2. **Support for Relocatable Code**:
   - By changing the base register $R$, the processor can access the same set of instructions or data in a different memory region.
   - Useful for moving a code block $B$ to another part of memory without modifying its instructions.
   - $R$ is referred to as the **base register**, and its value is the **base address**.
3. **Efficient Data Access (Indexing)**:
   - $R$ can be used as an **index register** to access array elements.
   - For example, let $D$ be the address of $X(0)$, and let $R = i$. Then, the address of $X(i)$ is:
   $$A = D + R$$
   - By changing the content of $R$, a single instruction can access any $X(i)$ in the array.

## Drawbacks of Relative Addressing

- Requires extra hardware logic in the CPU to compute addresses.
- Adds slight delay due to runtime computation of the effective address.

## Summary Table

| **Feature** | **Relative Addressing** |
|---|---|
| Operand Address Given? | No (Only offset $D$) |
| Effective Address | $A = R + D$ |
| Uses | Code relocation, arrays, loops |
| Instruction Size | Reduced (offset only) |
| Advantage | Flexible memory access |
| Drawback | Extra logic and processing time |

## Applications

- Loop control and array traversal.
- Dynamic memory segment management.
- PC-relative addressing: using Program Counter (PC) as base register.
- Common in RISC and CISC architectures.

## Auto Indexing and Register Addressing Modes

### Auto Indexing

Indexed items are frequently accessed sequentially. If an item $X(k)$ is accessed at memory location $A$, the next likely access is to $X(k + 1)$ or $X(k - 1)$, stored at $A + 1$ or $A - 1$ respectively. To optimize such access, an addressing mode called **auto indexing** is used, where the address is automatically incremented or decremented.

**Examples**

- **Motorola 680X0 (Pre-decrementing)**: `-(A3)`
  The content of register A3 is decremented before the instruction is executed.
- **ARM (Pre-indexed)**: `LDR R0, [R1, #4]!`
  Adds 4 to R1 before loading the value at the updated address into R0.
- **Motorola 680X0 (Post-incrementing)**: `(A3)+`
  The instruction uses the content of A3, then increments it.
- **ARM (Post-indexed)**: `LDR R0, [R1], #4`
  Loads from R1 into R0, then increments R1 by 4.

## Register Direct Addressing

In **register direct addressing**, the operand resides in a register whose name appears directly in the instruction.

- **Example (Motorola 680X0)**: `MOVE #99, D1`
  Moves the immediate constant 99 into data register D1.

## Register Indirect Addressing

In **register indirect addressing**, the register contains the memory address of the operand, not the operand itself.

- **Example (Motorola 680X0)**: `MOVE.B (A0), D1`
  Moves the byte located at memory address A0 into the low-order byte of register D1. Formally:

  $$D1[7:0] := M(A0)$$

  The other bits of D1 remain unchanged.

## Register Indirect with Offset

This is an extension of register indirect addressing, where the effective address is calculated using a base register and a displacement (offset). It is also known as **base or indexed addressing**.

- **Example**: `SW Rt, OFFSET(Rs)`
  Stores the contents of register Rt into the memory location at address ($Rs + \text{OFFSET}$). Formally:

  $$M[Rs + \text{OFFSET}] := Rt$$

  Here, Rs is the base register, OFFSET is the displacement, and Rt is the source register.

## Summary Table

| Mode | Address Source | Example | Meaning |
|---|---|---|---|
| Register Direct | Register name | `MOVE #99, D1` | D1 := 99 |
| Register Indirect | Memory address in register | `MOVE.B (A0), D1` | D1[7:0] := M(A0) |
| Reg. Indirect + Offset | Base register + offset | `SW Rt, OFFSET(Rs)` | M[Rs + OFFSET] := Rt |
| Auto Index (Post) | Increment after access | `(A3)+` | Access A3, then increment |