**🧩 Register Addressing Mode**

**🔹 সংজ্ঞা (Definition):**

👉 **Register Addressing Mode** হলো এমন একটি ডেটা অ্যাড্রেসিং পদ্ধতি,  
যেখানে **operand (ডেটা)** সরাসরি **processor-এর register**-এর ভেতরেই থাকে,  
মেমোরিতে নয়।

📘 সহজভাবে:

ডেটা মেমোরি থেকে আনতে হয় না —  
CPU তার নিজের register-এ রাখা মান ব্যবহার করে।

**🔹 কেন Register Addressing দ্রুত?**

✅ কারণ CPU-র registers **Processor-এর ভিতরেই থাকে**,  
তাই access time খুব কম।

✅ মেমোরি access করার মতো সময় লাগে না।  
✅ Instruction ছোট হয় (কম byte লাগে)।

**🔹 General Format:**

MOV R1, R2

📗 অর্থ:  
Register R2-এর ডেটা → Register R1-এ কপি করো।

**🔹 8085/8086 উদাহরণ:**

| **উদাহরণ** | **ব্যাখ্যা** |
| --- | --- |
| MOV A, B | Register B-এর মান Register A-তে কপি হবে |
| ADD B | Register B-এর মান Register A-এর সাথে যোগ হবে |
| SUB C | Register C-এর মান Register A থেকে বাদ হবে |
| INR D | Register D-এর মান ১ বৃদ্ধি পাবে |

👉 এখানে A, B, C, D ইত্যাদি হলো **register operands**।

**🔹 চিত্রভিত্তিক ধারণা:**

┌────────────┐

│ Register B │──┐

└────────────┘ │

▼

┌────────────┐

│ Register A │

└────────────┘

📗 **MOV A, B** → B → A  
অর্থাৎ Register B-এর ডেটা A-তে কপি হয়।

**🔹 8086 Example (16-bit Registers):**

| **Instruction** | **Description** |
| --- | --- |
| MOV AX, BX | BX → AX |
| ADD CX, DX | CX + DX → CX |
| XCHG AX, BX | AX ও BX-এর মান পরস্পর বদল হবে |

**🔹 Register Addressing Mode-এর বৈশিষ্ট্য (Features):**

| **বৈশিষ্ট্য** | **ব্যাখ্যা** |
| --- | --- |
| **Speed** | সবচেয়ে দ্রুত Addressing Mode |
| **Memory Access** | হয় না (শুধু Register Access) |
| **Instruction Size** | ছোট (১ বা ২ byte) |
| **Operands** | সবগুলো CPU Register |
| **Use** | Arithmetic, Logical, Data Transfer ইত্যাদি Instruction-এ ব্যবহৃত হয় |

**🔹 সুবিধা (Advantages):**

✅ দ্রুত কার্যসম্পাদন  
✅ মেমোরির চাপ কমে  
✅ Instruction execution time খুব কম  
✅ সহজ প্রোগ্রামিং

**🔹 অসুবিধা (Limitation):**

❌ খুব সীমিত সংখ্যক register থাকে (8085-এ মাত্র ৬টি general-purpose register)  
❌ বড় ডেটা বা Array-type ডেটা এতে সংরক্ষণ করা যায় না

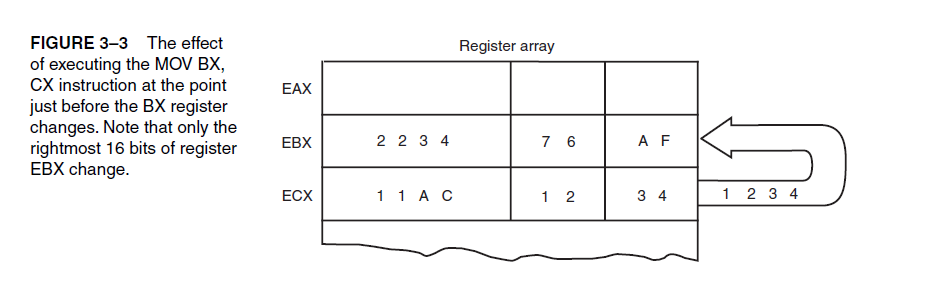
**💬 বাংলা সংক্ষিপ্ত সারমর্ম:**

🧩 **Register Addressing Mode**-এ operand (ডেটা) processor-এর **register**-এর ভিতরে সংরক্ষিত থাকে।  
CPU সরাসরি register থেকে ডেটা নিয়ে কাজ করে, মেমোরিতে যায় না।

ফলে এটি **সবচেয়ে দ্রুত** ও **ছোট instruction size**-এর addressing mode।

**⚙️ সহজ উদাহরণ মনে রাখো:**

| **Processor** | **Instruction** | **কাজ** |
| --- | --- | --- |
| **8085** | MOV A, B | B → A |
| **8085** | ADD C | A = A + C |
| **8086** | MOV AX, BX | BX → AX |
| **8051** | MOV A, R2 | R2 → A |



# 🧩 ****Immediate Addressing Mode****

## 🔹 ****সংজ্ঞা (Definition):****

👉 **Immediate Addressing Mode** হলো এমন একটি পদ্ধতি,  
যেখানে **operand (ডেটা)** সরাসরি **instruction-এর মধ্যেই উল্লেখ করা থাকে**,  
অর্থাৎ ডেটা মেমোরি বা রেজিস্টারে সংরক্ষিত নয় — বরং **instruction-এর অংশ হিসেবেই** দেওয়া থাকে।

📘 সহজভাবে:

“ডেটা কোথায় আছে সেটা CPU-কে খুঁজতে হয় না,  
কারণ মানটা (data) সরাসরি instruction-এর মধ্যেই লেখা থাকে।”

## 🔹 ****General Format:****

MOV A, #32H

📗 এখানে:

* MOV = Instruction (Move command)
* A = Destination Register
* #32H = **Immediate Data (source operand)**

অর্থাৎ → 32H মানটি সরাসরি Register A তে রাখা হবে।

## 🔹 ****কেন একে Immediate বলা হয়?****

কারণ ডেটা **“immediately available”**,  
অর্থাৎ CPU-কে মেমোরি বা অন্য রেজিস্টারে যেতে হয় না —  
ডেটা instruction-এর মধ্যেই উপস্থিত থাকে।

## 🔹 ****8085 Example:****

| **Instruction** | **কাজ** |
| --- | --- |
| MVI A, 45H | 45H → Accumulator (A) |
| MVI B, 0AH | 0AH → Register B |
| ADI 25H | A = A + 25H |
| SUI 10H | A = A – 10H |

📘 **MVI (Move Immediate)** মানে হলো:  
"Move immediate data into register."

## 🔹 ****8086 Example:****

| **Instruction** | **Description** |
| --- | --- |
| MOV AX, 1234H | 1234H → AX |
| ADD BX, 0100H | BX = BX + 0100H |
| SUB AL, 05H | AL = AL – 05H |

## 🔹 ****8051 Example:****

| **Instruction** | **কাজ** |
| --- | --- |
| MOV A, #55H | 55H → Accumulator |
| MOV R0, #20H | 20H → Register R0 |
| ADD A, #10H | A = A + 10H |

📘 লক্ষ্য করো — Immediate Data-এর আগে সবসময় **‘#’ (hash sign)** দেওয়া হয়,  
যা দেখায় এটি একটি **constant value**, মেমোরি address নয়।

## 🔹 ****Immediate Addressing-এর বৈশিষ্ট্য (Features):****

| **বৈশিষ্ট্য** | **বর্ণনা** |
| --- | --- |
| **Operand Source** | Instruction-এর ভেতরেই দেওয়া থাকে |
| **Memory Access** | দরকার হয় না |
| **Execution Speed** | খুব দ্রুত |
| **Instruction Size** | সাধারণত 2 বা 3 byte |
| **Use** | Constant মান register-এ লোড করার জন্য |

## 🔹 ****সুবিধা (Advantages):****

✅ Data access সবচেয়ে দ্রুত হয়  
✅ কোনো মেমোরি লুকআপ দরকার হয় না  
✅ সহজ প্রোগ্রামিং  
✅ Temporary constant value ব্যবহার সহজ

## 🔹 ****অসুবিধা (Limitations):****

❌ Data পরিবর্তন করা যায় না (কারণ এটি instruction-এর অংশ)  
❌ বড় ডেটা বা variable মান Immediate Data হিসেবে দেওয়া যায় না

## 💬 ****বাংলা সংক্ষিপ্ত সারমর্ম:****

🧩 **Immediate Addressing Mode**-এ operand বা data সরাসরি instruction-এর মধ্যেই উল্লেখ করা থাকে।  
CPU এই মানকে তৎক্ষণাৎ (immediately) ব্যবহার করে।

উদাহরণ: MOV A, #32H → 32H মানটি সরাসরি A রেজিস্টারে যায়।

এতে execution সময় কম লাগে এবং কোনো মেমোরি access দরকার হয় না।

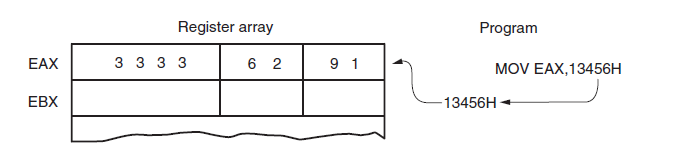
## ⚙️ ****সহজ উদাহরণ মনে রাখো:****

| **Processor** | **Instruction** | **কাজ** |
| --- | --- | --- |
| **8085** | MVI A, 45H | A ← 45H |
| **8086** | MOV AX, 1234H | AX ← 1234H |
| **8051** | MOV A, #55H | A ← 55H |

📘 **মনে রাখার ট্রিক:**

🔹 যদি instruction-এ ডেটা **সরাসরি লেখা থাকে**,  
তাহলে সেটি হলো **Immediate Addressing Mode**।

🔹 Symbol # (in 8051) বা direct value (in 8085) = Immediate Data ✅



**🧩 Direct Data Addressing Mode**

**🔹 সংজ্ঞা (Definition):**

👉 **Direct Addressing Mode**-এ instruction-এর মধ্যে operand হিসেবে **memory address** দেওয়া থাকে,  
যেখান থেকে CPU ডেটা নিয়ে আসে বা সেখানে ডেটা পাঠায়।

📘 সহজভাবে বললে:

“ডেটা মেমোরির নির্দিষ্ট ঠিকানায় (address) রাখা থাকে,  
আর CPU সেই ঠিকানায় সরাসরি গিয়ে ডেটা নেয় বা লেখে।”

**🔹 General Format:**

MOV A, 30H

📗 এখানে —

* MOV = Move instruction
* A = Register (Destination)
* 30H = **Memory Address (Source)**

👉 অর্থাৎ মেমোরি লোকেশন **30H**-এ থাকা মানটি Accumulator (A)-তে যাবে।

**🔹 8051 Microcontroller উদাহরণ:**

| **Instruction** | **কাজ** |
| --- | --- |
| MOV A, 30H | Memory location 30H → Accumulator |
| MOV 40H, A | Accumulator → Memory location 40H |
| ADD A, 25H | A = A + [25H] |
| INC 60H | Memory location 60H-এর মান ১ বৃদ্ধি পাবে |

📘 **[30H] বা [40H]** মানে হলো — “30H ঠিকানার মধ্যে যে মান আছে”।

**🔹 8085 Microprocessor উদাহরণ:**

| **Instruction** | **বর্ণনা** |
| --- | --- |
| LDA 2050H | Memory address 2050H-এর মান → Accumulator |
| STA 3050H | Accumulator-এর মান → Memory address 3050H |
| IN 00H | Input port 00H থেকে ডেটা নেয় |
| OUT 01H | Output port 01H-এ ডেটা পাঠায় |

📗 এখানে Memory Address (যেমন 2050H, 3050H) সরাসরি instruction-এর অংশ।

**🔹 ডেটা ফ্লো (Data Flow):**

Memory (30H)

↓

CPU Register (A)

অথবা উল্টোভাবে:

CPU Register (A)

↓

Memory (40H)

**🔹 Direct Addressing-এর বৈশিষ্ট্য (Features):**

| **বৈশিষ্ট্য** | **ব্যাখ্যা** |
| --- | --- |
| **Operand Location** | নির্দিষ্ট মেমোরি address দেওয়া থাকে |
| **Speed** | Immediate-এর চেয়ে ধীর (কারণ মেমোরি অ্যাক্সেস করতে হয়) |
| **Instruction Size** | সাধারণত 2 বা 3 bytes |
| **Use** | RAM বা Data Memory থেকে মান পড়া/লেখার জন্য ব্যবহৃত হয় |

**🔹 সুবিধা (Advantages):**

✅ মেমোরির নির্দিষ্ট অবস্থানে সরাসরি কাজ করা যায়  
✅ কোড পড়া সহজ (address স্পষ্টভাবে লেখা থাকে)  
✅ Data memory সহজে অ্যাক্সেস করা যায়

**🔹 অসুবিধা (Limitations):**

❌ Instruction size বড় হয় (address লিখতে হয়)  
❌ Memory address স্থির থাকে (dynamic address দেওয়া যায় না)  
❌ Execution time Immediate mode-এর চেয়ে বেশি

**💬 বাংলা সংক্ষিপ্ত সারমর্ম:**

🧩 **Direct Addressing Mode**-এ operand হলো **memory address**।  
CPU instruction-এ দেওয়া address অনুযায়ী মেমোরি থেকে ডেটা নেয় বা সেখানে লেখে।

উদাহরণ:  
MOV A, 30H → Memory (30H)-এর মান Accumulator-এ আসে।  
MOV 40H, A → Accumulator-এর মান Memory (40H)-এ যায়।

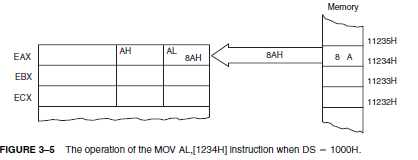
**⚙️ সহজ উদাহরণ মনে রাখো:**

| **Processor** | **Instruction** | **কাজ** |
| --- | --- | --- |
| **8051** | MOV A, 30H | [30H] → A |
| **8051** | MOV 40H, A | A → [40H] |
| **8085** | LDA 2050H | [2050H] → A |
| **8085** | STA 3050H | A → [3050H] |

📘 **মনে রাখার ট্রিক:**

🔹 যদি instruction-এ **memory address** (যেমন 30H, 2050H) লেখা থাকে,  
তাহলে সেটি হলো **Direct Addressing Mode** ✅

🔹 Immediate mode-এ “#data” থাকে,  
আর Direct mode-এ “memory address” থাকে।



**🧩 Register Indirect Addressing Mode**

**🔹 সংজ্ঞা (Definition):**

👉 **Register Indirect Addressing Mode** হলো এমন একটি পদ্ধতি,  
যেখানে **ডেটা সরাসরি রেজিস্টারে থাকে না**, বরং  
**রেজিস্টারের ভেতরে থাকা মান (address)** মেমোরিতে থাকা ডেটার ঠিকানা হিসেবে ব্যবহৃত হয়।

📘 সহজভাবে:

“রেজিস্টার-এ যে মান আছে, সেটি ডেটার ঠিকানা নির্দেশ করে।”  
CPU সেই address ব্যবহার করে মেমোরি থেকে ডেটা নেয় বা লেখে।

**🔹 মূল ধারণা:**

CPU প্রথমে register থেকে **effective address** নেয়,  
তারপর সেই address অনুযায়ী মেমোরিতে গিয়ে **ডেটা অ্যাক্সেস করে**।

**🔹 General Format:**

MOV A, @R0

📗 এখানে —

* MOV = Move Instruction
* A = Destination (Accumulator)
* @R0 = Register Indirect Operand

👉 অর্থাৎ R0-এর মধ্যে যে মান আছে → সেটি একটি মেমোরি address →  
সেই address থেকে ডেটা এনে Accumulator-এ রাখা হবে।

**🔹 Symbol “@” এর মানে:**

📘 “@” চিহ্নটি বোঝায় → *register-এর ভিতরে থাকা মানটি একটি address*।

উদাহরণ:  
যদি R0 = 40H, তাহলে  
MOV A, @R0 → মানে হলো MOV A, [40H]  
অর্থাৎ মেমোরির 40H লোকেশন থেকে ডেটা Accumulator-এ আসবে।

**🔹 8051 Microcontroller উদাহরণ:**

| **Instruction** | **কাজ** |
| --- | --- |
| MOV A, @R0 | R0-তে যে address আছে, সেই লোকেশনের ডেটা A-তে যাবে |
| MOV @R1, A | A-এর মান মেমোরিতে যাবে, যার address R1-এ আছে |
| INC @R0 | R0-তে থাকা address অনুযায়ী মেমোরির মান ১ বৃদ্ধি পাবে |

📘 উদাহরণ:

R0 = 30H

[30H] = 25H

MOV A, @R0 → A = 25H

**🔹 8085 Microprocessor উদাহরণ:**

| **Instruction** | **কাজ** |
| --- | --- |
| MOV A, M | HL রেজিস্টার pair-এ থাকা address অনুযায়ী মেমোরির মান A-তে যাবে |
| MOV M, A | A-এর মান HL pair দ্বারা নির্দেশিত মেমোরিতে সংরক্ষিত হবে |
| INX H | HL pair-এর মান ১ বৃদ্ধি পায় (পরবর্তী address) |

📘 উদাহরণ:

H = 20H

L = 40H

→ HL = 2040H

MOV A, M → A = [2040H]

**🔹 8086 Example (16-bit):**

| **Instruction** | **কাজ** |
| --- | --- |
| MOV AX, [BX] | BX-এ থাকা address অনুযায়ী মেমোরি থেকে ডেটা AX-এ আসে |
| MOV [SI], AL | AL-এর মান মেমোরিতে লেখা হয়, যার ঠিকানা SI রেজিস্টারে আছে |

📘 উদাহরণ:

BX = 2000H

MOV AX, [BX] → AX = [2000H]

**🔹 Data Flow Diagram:**

R0 → (contains address 40H)

↓

Memory [40H] → (contains 25H)

↓

Accumulator ← 25H

**🔹 Register Indirect Addressing-এর বৈশিষ্ট্য (Features):**

| **বৈশিষ্ট্য** | **বর্ণনা** |
| --- | --- |
| **Operand Source** | Memory (address register-এ থাকে) |
| **Speed** | Direct-এর চেয়ে একটু ধীর, কিন্তু Flexible |
| **Instruction Size** | সাধারণত 1 বা 2 byte |
| **Registers Used** | 8051-এ R0, R1; 8085-এ HL pair; 8086-এ BX, BP, SI, DI |
| **Use** | Array, Table, Pointer ইত্যাদিতে ব্যবহৃত হয় |

**🔹 সুবিধা (Advantages):**

✅ Memory address পরিবর্তনযোগ্য (Dynamic Addressing)  
✅ সহজে array বা data block access করা যায়  
✅ কম instruction দিয়ে বেশি কাজ করা যায়

**🔹 অসুবিধা (Limitations):**

❌ Address খুঁজে পেতে সময় লাগে  
❌ Limited register ব্যবহার করা যায় (8051-এ শুধুমাত্র R0 ও R1)

**💬 বাংলা সংক্ষিপ্ত সারমর্ম:**

🧩 **Register Indirect Addressing Mode**-এ operand-এর address রেজিস্টারে সংরক্ষিত থাকে।  
CPU সেই address অনুযায়ী মেমোরি থেকে ডেটা নেয় বা লেখে।

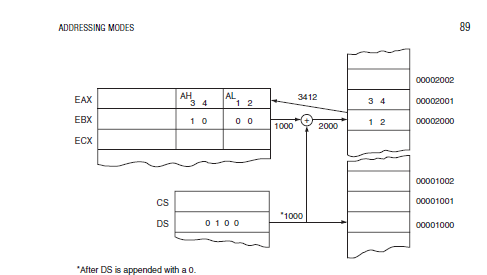
উদাহরণ:  
MOV A, @R0 → R0-তে থাকা address অনুযায়ী মেমোরির মান A-তে আসে।

**⚙️ সহজ উদাহরণ মনে রাখো:**

| **Processor** | **Instruction** | **কাজ** |
| --- | --- | --- |
| **8051** | MOV A, @R0 | [R0]-এর মান → A |
| **8051** | MOV @R1, A | A-এর মান → [R1] |
| **8085** | MOV A, M | [HL]-এর মান → A |
| **8086** | MOV AX, [BX] | [BX]-এর মান → AX |

📘 **মনে রাখার ট্রিক:**

🔹 “@” বা “M” দেখলেই বোঝো — **Register Indirect Addressing Mode**  
🔹 কারণ Data Memory থেকে আসে, কিন্তু **Address রেজিস্টারে থাকে।**



**🧩 Base-Plus-Index Addressing Mode**

**🔹 সংজ্ঞা (Definition):**

👉 **Base-Plus-Index Addressing Mode** হলো এমন একটি **memory addressing mode**,  
যেখানে **একটি Base Register** এবং **একটি Index Register**-এর মান **যোগ করে**  
একটি **Effective Address (EA)** তৈরি হয়,  
যার মাধ্যমে Memory Location অ্যাক্সেস করা হয়।

📘 সহজভাবে:

“একটি address পাওয়া যায় Base Register + Index Register-এর যোগফল থেকে।”  
CPU এই address ব্যবহার করে Memory থেকে ডেটা নেয় বা সেখানে লেখে।

**🔹 Formula (Effective Address):**

[  
EA = Base Register + Index Register  
]

**🔹 Registers Used:**

📗 **8086 Microprocessor-এ:**

| **Register Type** | **Register Name** |
| --- | --- |
| **Base Register** | BX বা BP |
| **Index Register** | SI (Source Index), DI (Destination Index) |

📘 অর্থাৎ Base হিসেবে **BX/BP** এবং Index হিসেবে **SI/DI** ব্যবহৃত হয়।

**🔹 General Format (Syntax):**

MOV AX, [BX + SI]

📗 ব্যাখ্যা:

* BX → Base Register
* SI → Index Register
* [BX + SI] → Memory Address (EA)

👉 CPU প্রথমে BX ও SI-এর মান যোগ করে Effective Address বের করে,  
তারপর সেই Memory Location থেকে ডেটা নিয়ে AX-এ রাখে।

**🔹 Example 1:**

MOV AX, [BX + SI]

📘 ধরা যাক:

BX = 2000H

SI = 0050H

তাহলে Effective Address (EA):

EA = BX + SI = 2000H + 0050H = 2050H

👉 তাই MOV AX, [BX + SI] মানে → Memory [2050H]-এর মান AX-এ যাবে।

**🔹 Example 2 (with BP):**

MOV AX, [BP + DI]

📗 ব্যাখ্যা:

* BP → সাধারণত Stack বা Data Segment-এর জন্য ব্যবহৃত হয়
* DI → Index register  
  👉 EA = BP + DI

**🔹 Base-Plus-Index Addressing-এর ব্যবহার:**

✅ সাধারণত **Arrays, Tables, এবং Strings** অ্যাক্সেস করার জন্য ব্যবহৃত হয়।

📘 উদাহরণ:  
ধরা যাক একটি Array মেমোরিতে সংরক্ষিত:

Array: 2000H → 2005H

তাহলে:

* BX = Array-এর শুরুর ঠিকানা (Base)
* SI = Element offset (Index)

Instruction:

MOV AL, [BX + SI]

👉 এটি Array থেকে SI offset অনুযায়ী একটি element নিয়ে আসে।

**🔹 Segment Association:**

📗 8086-এ Effective Address তৈরি হয় Base ও Index যোগ করে,  
তারপর **Segment Register** অনুযায়ী আসল Memory Address নির্ধারণ হয়:

| **Combination** | **Default Segment** |
| --- | --- |
| BX + SI | DS (Data Segment) |
| BX + DI | DS |
| BP + SI | SS (Stack Segment) |
| BP + DI | SS |

**🔹 চিত্রভিত্তিক ধারণা (Data Flow):**

[Base Register] + [Index Register]

↓

Effective Address

↓

Memory Location

↓

Data → Register

📗 **Example:** MOV AX, [BX + SI]

BX = 2000H

SI = 0030H

→ EA = 2030H

→ AX = [2030H]

**🔹 Base-Plus-Index Addressing-এর বৈশিষ্ট্য (Features):**

| **বৈশিষ্ট্য** | **বর্ণনা** |
| --- | --- |
| **Address Source** | Base Register + Index Register |
| **Use** | Array / Table / String Access |
| **Flexibility** | Dynamic Address Calculation |
| **Segment Register** | DS বা SS অনুযায়ী নির্ধারিত হয় |
| **Instruction Size** | সাধারণত 2–3 bytes |

**🔹 সুবিধা (Advantages):**

✅ Arrays ও Data Table সহজে অ্যাক্সেস করা যায়  
✅ Address পরিবর্তন সহজ (index পরিবর্তন করলে নতুন data)  
✅ Dynamic data structure (যেমন loops-এ ব্যবহার)

**🔹 অসুবিধা (Limitations):**

❌ শুধুমাত্র নির্দিষ্ট registers (BX, BP, SI, DI) ব্যবহার করা যায়  
❌ Instruction তুলনামূলক জটিল (অন্য modes-এর চেয়ে বড়)

**💬 বাংলা সংক্ষিপ্ত সারমর্ম:**

🧩 **Base-Plus-Index Addressing Mode** হলো এমন একটি পদ্ধতি,  
যেখানে মেমোরি address তৈরি হয় **একটি Base Register এবং একটি Index Register**-এর যোগফল দিয়ে।

CPU এই address অনুযায়ী মেমোরি থেকে ডেটা নেয় বা সেখানে লেখে।

উদাহরণ: MOV AX, [BX + SI] → Memory [BX + SI]-এর মান AX-এ আসে।

**⚙️ সহজ উদাহরণ মনে রাখো:**

| **Instruction** | **অর্থ** |
| --- | --- |
| MOV AX, [BX + SI] | Data Segment-এর [BX+SI] address থেকে AX-এ ডেটা |
| MOV AL, [BP + DI] | Stack Segment-এর [BP+DI] address থেকে AL-এ ডেটা |
| MOV [BX + DI], CX | CX → Memory [BX+DI] |

📘 **মনে রাখার ট্রিক:**

🔹 Base → Data Structure-এর শুরু  
🔹 Index → তার ভেতরে offset  
🔹 EA = Base + Index

# 🧩 ****Register Relative Addressing Mode****

## 🔹 ****সংজ্ঞা (Definition):****

👉 **Register Relative Addressing Mode** হলো এমন একটি পদ্ধতি,  
যেখানে **একটি register-এর ভিতরে থাকা base address** এবং **instruction-এ দেওয়া একটি displacement (offset)** —  
এই দুইয়ের **যোগফল (sum)** দ্বারা **Effective Address (EA)** তৈরি হয়।

এটি এক ধরনের **data addressing mode**, যা **Direct Addressing** আর **Register Indirect Addressing** —এই দুইয়ের মিশ্র (combination) রূপ হিসেবে কাজ করে।

📘 সহজভাবে:

“একটি রেজিস্টারের মানকে ভিত্তি (base) ধরে,  
তার সাথে একটি নির্দিষ্ট সংখ্যক বাইট (offset) যোগ করে Memory Address তৈরি করা হয়।”

## 🔹 ****Formula (Effective Address):****

[  
EA = Base Register + Displacement  
]

📗 **EA** = Effective Address → যেখানে ডেটা আসলে মেমোরিতে আছে।

## 🔹 ****Registers Used (8086 Example):****

| **Register Type** | **Common Registers** |
| --- | --- |
| **Base Register** | BX বা BP |
| **Index Register (optional)** | SI বা DI |
| **Displacement (Offset)** | 8-bit বা 16-bit Immediate Value |

## 🔹 ****General Syntax:****

MOV AX, [BX + 05H]

📗 অর্থাৎ:

মেমোরির ঠিকানা = BX-এর মান + 05H

👉 CPU সেই ঠিকানায় গিয়ে ডেটা পড়ে AX-এ রাখে।

## 🔹 ****Step-by-Step Example:****

ধরা যাক —

BX = 2000H

Displacement = 0020H

👉 Effective Address (EA):

EA = BX + 0020H = 2020H

📘 Instruction:

MOV AX, [BX + 20H]

👉 মানে হলো: Memory [2020H]-এর ডেটা AX-এ আসবে।

## 🔹 ****আরও উদাহরণ:****

| **Instruction** | **ব্যাখ্যা** |
| --- | --- |
| MOV AL, [BP + 04H] | BP register-এর মানের সাথে 04H যোগ করে মেমোরি থেকে AL-এ ডেটা আসে |
| ADD AX, [BX + 10H] | AX = AX + Memory[BX + 10H] |
| MOV [BP + 08H], CL | CL → Memory[BP + 08H] |

## 🔹 ****Segment Association (8086):****

| **Combination** | **Default Segment** |
| --- | --- |
| BX + displacement | DS (Data Segment) |
| BP + displacement | SS (Stack Segment) |

📘 তাই যদি BP ব্যবহার করা হয়, তাহলে ডেটা সাধারণত **Stack Memory** থেকে আসে।

## 🔹 ****Data Flow (Visualization):****

Register (BX) = 2000H

Displacement = 0020H

→ EA = 2020H

→ CPU accesses Memory[2020H]

→ Data → AX

📊 **Flow Diagram:**

[ Register Value (Base) ]

+

[ Displacement (Offset) ]

↓

[ Effective Address ]

↓

[ Memory Data ]

↓

[ Register Destination ]

## 🔹 ****Register Relative Addressing Mode-এর বৈশিষ্ট্য (Features):****

| **বৈশিষ্ট্য** | **বর্ণনা** |
| --- | --- |
| **Address Source** | Register + Displacement |
| **Displacement Size** | সাধারণত 8-bit বা 16-bit |
| **Speed** | Direct-এর চেয়ে ধীর, কিন্তু বেশি flexible |
| **Instruction Size** | সাধারণত 3 bytes |
| **Use** | Arrays, Stacks, Table Lookup ইত্যাদিতে ব্যবহৃত হয় |

## 🔹 ****সুবিধা (Advantages):****

✅ মেমোরি address পরিবর্তন করা যায় (dynamic base)  
✅ সহজে array element access করা যায়  
✅ Stack বা Table Data Access সহজ হয়

## 🔹 ****অসুবিধা (Limitations):****

❌ Instruction তুলনামূলক বড়  
❌ Execution time একটু বেশি (address যোগ করতে হয়)

## 💬 ****বাংলা সংক্ষিপ্ত সারমর্ম:****

🧩 **Register Relative Addressing Mode**-এ Memory Address তৈরি হয় —  
**Register-এর মান (Base)** + **Displacement (Offset)** যোগ করে।

তারপর CPU সেই address ব্যবহার করে ডেটা নেয় বা লেখে।

উদাহরণ:  
MOV AX, [BX + 20H] → BX (2000H) + 20H = 2020H → Memory [2020H]-এর মান AX-এ আসে।

## ⚙️ ****সহজ উদাহরণ মনে রাখো:****

| **Instruction** | **অর্থ** |
| --- | --- |
| MOV AX, [BX + 10H] | Memory [BX + 10H] → AX |
| MOV AL, [BP + 08H] | Memory [BP + 08H] → AL |
| ADD BX, [BX + 02H] | BX = BX + [BX + 02H] |

📘 **মনে রাখার ট্রিক:**

🔹 Direct Addressing → Address instruction-এ লেখা থাকে  
🔹 Register Indirect → Address register-এ থাকে  
🔹 Register Relative → Address = Register + Offset

**🧩 Scaled-Index Addressing Mode**

**🔹 সংজ্ঞা (Definition):**

👉 **Scaled-Index Addressing Mode** হলো এমন একটি **memory addressing technique**,  
যেখানে মেমোরি address তৈরি হয় **Base Register**, **Index Register**, **Scaling Factor**, এবং **Displacement** —এই চারটি উপাদান ব্যবহার করে।

📘 সহজভাবে:

“Index Register-এর মানকে একটি নির্দিষ্ট সংখ্যায় গুণ (scale) করে  
Base Address এবং Displacement-এর সাথে যোগ করে **Effective Address (EA)** তৈরি করা হয়।”

**🔹 Formula (Effective Address):**

[  
EA = Base + (Index × Scale) + Displacement  
]

📗 **EA** = Effective Address → মেমোরি লোকেশনের ঠিকানা

**🔹 Registers Used:**

| **Register Type** | **Common Registers** |
| --- | --- |
| **Base Register** | BX, BP, EBX, EBP |
| **Index Register** | SI, DI, ESI, EDI |
| **Scale Factor** | 1, 2, 4, বা 8 (Multiplication factor) |
| **Displacement** | 8-bit, 16-bit বা 32-bit Immediate Value |

**🔹 Scaling Factor কী?**

👉 Scaling Factor দ্বারা বোঝানো হয় Index Register-এর মান কত দ্বারা গুণ করা হবে।

| **Scale** | **ব্যবহারের উদ্দেশ্য** |
| --- | --- |
| 1 | Byte data access |
| 2 | Word data access (2 bytes) |
| 4 | Doubleword access (4 bytes) |
| 8 | Quadword access (8 bytes) |

📘 অর্থাৎ —  
যদি কোনো array-এর প্রতিটি element 4 byte (যেমন integer),  
তাহলে index \* 4 দ্বারা element-এর address পাওয়া যাবে।

**🔹 General Syntax:**

MOV EAX, [EBX + ESI\*4 + 08H]

📗 অর্থাৎ:

Effective Address = EBX + (ESI × 4) + 08H

👉 CPU এই EA অনুযায়ী Memory থেকে ডেটা পড়ে EAX রেজিস্টারে রাখবে।

**🔹 Step-by-Step Example:**

ধরা যাক:

EBX = 1000H (Base)

ESI = 0003H (Index)

Scale = 4

Displacement = 08H

তাহলে:

EA = 1000H + (0003H × 4) + 08H

EA = 1000H + 000CH + 08H

EA = 1000H + 14H = 1014H

👉 তাই MOV EAX, [EBX + ESI\*4 + 08H] মানে হলো  
Memory [1014H]-এর মান EAX-এ আসবে।

**🔹 অর্থাৎ CPU কাজ করছে এভাবে:**

Base Address → EBX = 1000H

Index Register → ESI = 3

Scale Factor → 4

Displacement → 08H

----------------------------

Effective Address = 1000H + (3×4) + 08H = 1014H

📘 এরপর CPU → Memory[1014H] থেকে Data নেয় → EAX রেজিস্টারে রাখে।

**🔹 Scaled-Index Addressing-এর ব্যবহার:**

✅ সাধারণত **Array, Table, Structure, এবং Matrix Accessing**-এর জন্য ব্যবহৃত হয়।

📗 উদাহরণ:

* Integer array যেখানে প্রতিটি element = 4 bytes
* Character array যেখানে প্রতিটি element = 1 byte

👉 তাই Index × Size = element offset

**🔹 চিত্রভিত্তিক ধারণা (Data Flow):**

(Base) + (Index × Scale) + Displacement

↓

Effective Address (EA)

↓

Memory Location

↓

Data → Register

📊 **Example Flow:**

MOV EAX, [EBX + ESI\*4 + 08H]

↓

EA = EBX + (ESI×4) + 08H

↓

Memory[EA] → EAX

**🔹 Scaled-Index Addressing-এর বৈশিষ্ট্য (Features):**

| **বৈশিষ্ট্য** | **বর্ণনা** |
| --- | --- |
| **Address Calculation** | EA = Base + (Index × Scale) + Displacement |
| **Scalability** | 1, 2, 4, 8 দ্বারা গুণ করা যায় |
| **Flexibility** | Arrays, Tables, and Data Structures-এর জন্য উপযোগী |
| **Registers Used** | 32-bit বা 64-bit Register (80386 এবং পরবর্তী প্রসেসরগুলোতে) |
| **Speed** | দ্রুত কারণ hardware-level multiplication হয় |

**🔹 সুবিধা (Advantages):**

✅ Array বা Data Table Access সহজ ও দ্রুত  
✅ Scale পরিবর্তন করে বিভিন্ন data size অ্যাক্সেস করা যায়  
✅ Dynamic Memory Access  
✅ Complex addressing এক লাইনে করা যায়

**🔹 অসুবিধা (Limitations):**

❌ শুধুমাত্র আধুনিক (32-bit বা 64-bit) প্রসেসরে কাজ করে  
❌ Instruction একটু দীর্ঘ ও জটিল

**💬 বাংলা সংক্ষিপ্ত সারমর্ম:**

🧩 **Scaled-Index Addressing Mode** হলো এমন একটি পদ্ধতি,  
যেখানে **Memory Address = Base + (Index × Scale) + Displacement** দ্বারা নির্ধারিত হয়।

এটি সাধারণত **Array বা Structure** থেকে element অ্যাক্সেস করার জন্য ব্যবহৃত হয়।

উদাহরণ:  
MOV EAX, [EBX + ESI\*4 + 08H] →  
Effective Address = EBX + (ESI×4) + 08H → Memory থেকে ডেটা → EAX-এ।

**⚙️ সহজ উদাহরণ মনে রাখো:**

| **Instruction** | **কাজ** |
| --- | --- |
| MOV EAX, [EBX + ESI\*4 + 08H] | EA = EBX + (ESI×4) + 08H |
| MOV EDX, [EBP + EDI\*2] | EA = EBP + (EDI×2) |
| ADD EBX, [ECX + EDI\*8 + 10H] | EA = ECX + (EDI×8) + 10H |

📘 **মনে রাখার ট্রিক:**

🔹 Base — Array-এর শুরু  
🔹 Index — Element number  
🔹 Scale — Element size  
🔹 EA = Base + (Index × Size) + Offset

**🧩 RIP Relative Addressing Mode**

**🔹 সংজ্ঞা (Definition):**

👉 **RIP-Relative Addressing Mode** হলো একটি **Program Memory Addressing Mode**,  
যেখানে **Effective Address (EA)** তৈরি হয় **RIP (Instruction Pointer Register)**-এর বর্তমান মানের সাথে একটি **signed displacement (offset)** যোগ করে।

📘 সহজভাবে বললে:

“Instruction pointer (RIP)-এর বর্তমান অবস্থান থেকে কিছুটা সামনে বা পিছনে থাকা মেমোরি address-এ অ্যাক্সেস করা হয়।”

**🔹 RIP কী?**

* **RIP (Register Instruction Pointer)** হলো 64-bit processor-এ Program Counter-এর নাম।
* এটি নির্দেশ করে **পরবর্তী instruction কোথায় আছে**।
* যখন কোনো instruction execute হয়, RIP তার মান বাড়িয়ে পরবর্তী address নির্দেশ করে।

📗 **8086-এ** একে বলা হতো **IP (Instruction Pointer)**  
📗 **80386 এবং 32-bit প্রসেসরগুলোতে** → **EIP**  
📗 **x86-64 প্রসেসরে** → **RIP**

**🔹 Formula (Effective Address):**

[  
EA = RIP + Displacement  
]

📗 অর্থাৎ Effective Address (EA) তৈরি হয়  
→ Instruction Pointer (RIP)-এর মান + একটি signed offset (যেমন +20, -15 ইত্যাদি)।

**🔹 General Syntax:**

MOV RAX, [RIP + offset]

📘 অর্থাৎ:  
CPU বর্তমান instruction-এর ঠিক পরের অবস্থান (RIP) থেকে offset যোগ করে  
সেই ঠিকানার মেমোরি থেকে ডেটা নিয়ে RAX রেজিস্টারে রাখবে।

**🔹 Example 1:**

MOV RAX, [RIP + 20H]

📗 অর্থাৎ:

বর্তমান RIP থেকে 20H বাইট পরের address থেকে ডেটা পড়ে RAX-এ রাখো।

ধরা যাক:

RIP = 0000000000001000H

Displacement = +20H

তাহলে:

EA = 1000H + 20H = 1020H

👉 তাই CPU Memory [1020H] থেকে ডেটা নেয়।

**🔹 Example 2 (Negative Offset):**

MOV RBX, [RIP - 8]

📗 অর্থাৎ:

বর্তমান instruction pointer থেকে 8 byte আগে থাকা memory থেকে ডেটা পড়ো।

**🔹 RIP-Relative Addressing-এর ব্যবহার:**

✅ এটি মূলত **Position-Independent Code (PIC)** তৈরির জন্য ব্যবহৃত হয় —  
যাতে প্রোগ্রামটি মেমোরির যেকোনো জায়গায় লোড হলেও ঠিকভাবে কাজ করতে পারে।

📘 ব্যবহৃত হয়:

* **Modern Operating Systems (Linux, Windows, macOS)**
* **Shared Libraries (.dll / .so files)**
* **Dynamic Linking / Loader Systems**

**🔹 Why RIP-Relative is Important:**

📗 পুরনো প্রসেসরগুলিতে **absolute address** ব্যবহার করা হতো,  
যা কোডকে নির্দিষ্ট address-এ বেঁধে রাখত।

📗 কিন্তু **RIP-relative addressing** কোডকে **relocatable** করে —  
অর্থাৎ প্রোগ্রামটি যেকোনো মেমোরি স্থানে লোড করলেও সঠিকভাবে data ও instruction reference করতে পারে।

**🔹 Data Flow Diagram:**

[RIP] + [Displacement]

↓

Effective Address (EA)

↓

Memory Location Accessed

↓

Data → Register

📘 উদাহরণ:  
MOV RAX, [RIP + 0x10]  
→ RIP = 0x2000  
→ EA = 0x2010  
→ Memory[0x2010] → RAX

**🔹 RIP-Relative Addressing-এর বৈশিষ্ট্য (Features):**

| **বৈশিষ্ট্য** | **বর্ণনা** |
| --- | --- |
| **Address Calculation** | EA = RIP + Displacement |
| **Displacement Type** | Signed 32-bit value (+/-) |
| **Registers Used** | RIP only (64-bit mode) |
| **Use** | Position-independent code, Data reference |
| **Available In** | x86-64 (not in 32-bit mode) |

**🔹 সুবিধা (Advantages):**

✅ Position-independent code তৈরি হয়  
✅ Relocation-free memory access  
✅ Shared library loading সহজ হয়  
✅ কোড নিরাপদ (ASLR — Address Space Layout Randomization সমর্থন করে)

**🔹 অসুবিধা (Limitations):**

❌ শুধুমাত্র **64-bit mode**-এ কাজ করে (32-bit mode-এ নয়)  
❌ Instruction size তুলনামূলকভাবে বড়  
❌ শুধুমাত্র **data memory reference**-এর জন্য ব্যবহৃত হয় (register বা port নয়)

**💬 বাংলা সংক্ষিপ্ত সারমর্ম:**

🧩 **RIP-Relative Addressing Mode** হলো এমন একটি পদ্ধতি,  
যেখানে **Effective Address = RIP + Displacement** দ্বারা নির্ধারিত হয়।

অর্থাৎ CPU বর্তমান instruction pointer (RIP)-এর মানের সাথে একটি offset যোগ করে  
মেমোরির নির্দিষ্ট স্থান থেকে ডেটা অ্যাক্সেস করে।

এটি মূলত **64-bit mode**-এ ব্যবহৃত হয় position-independent code (PIC) তৈরির জন্য।

**⚙️ সহজ উদাহরণ মনে রাখো:**

| **Instruction** | **অর্থ** |
| --- | --- |
| MOV RAX, [RIP + 08H] | RIP থেকে 8 byte পরের মেমোরির মান → RAX |
| MOV RBX, [RIP - 04H] | RIP থেকে 4 byte আগের মেমোরির মান → RBX |
| LEA RCX, [RIP + Label] | Label-এর ঠিকানা RCX-এ লোড করা হবে |

📘 **মনে রাখার ট্রিক:**

🔹 RIP = বর্তমান instruction-এর ঠিকানা  
🔹 EA = RIP + Offset  
🔹 শুধুমাত্র 64-bit mode-এ কাজ করে  
🔹 PIC (Position Independent Code)-এর মূল ভিত্তি