

**(a) 16-bit Instruction Mode**

এই ফরম্যাটটি 8086, 80186, 80286 প্রসেসরে ব্যবহৃত হয়।  
এখানে প্রতিটি নির্দেশনা (instruction) ৪টি মূল অংশে বিভক্ত:

1. **Opcode (1–2 bytes)**  
   👉 নির্দেশ করে কী কাজ করতে হবে (যেমন MOV, ADD ইত্যাদি)।
2. **MOD-REG-R/M (0–1 bytes)**  
   👉 কোন রেজিস্টার বা মেমোরি লোকেশনে কাজ হবে, তা নির্দেশ করে।
3. **Displacement (0–1 bytes)**  
   👉 যদি মেমোরি অ্যাড্রেসের সঙ্গে অফসেট প্রয়োজন হয়, তাহলে সেটা এখানে থাকে।
4. **Immediate (0–2 bytes)**  
   👉 যদি কোনো নির্দিষ্ট মান সরাসরি নির্দেশনায় থাকে (যেমন MVI A, 05H), তা এখানে সংরক্ষিত হয়।

**🔹 (b) 32-bit Instruction Mode**

এটি **80386 থেকে Pentium 4** পর্যন্ত ব্যবহৃত হয়, যেখানে ৩২-বিট ডেটা ও অ্যাড্রেস ব্যবহার করা যায়।  
এই ফরম্যাটটি কিছুটা বড় ও জটিল, কারণ এটি আরও বেশি অপশন দেয়।

1. **Address size (0–1 bytes)**  
   👉 ঠিকানার আকার (16-bit বা 32-bit) নির্ধারণ করে।
2. **Register size (0–1 bytes)**  
   👉 রেজিস্টারের আকার নির্ধারণ করে (যেমন 8-bit, 16-bit, বা 32-bit)।
3. **Opcode (1–2 bytes)**  
   👉 কোন কাজটি সম্পন্ন হবে তা নির্দেশ করে।
4. **MOD-REG-R/M (0–1 bytes)**  
   👉 কোন রেজিস্টার বা মেমোরি ব্যবহৃত হবে, তা নির্দেশ করে।
5. **Scaled-index (0–1 bytes)**  
   👉 ইনডেক্স রেজিস্টার ব্যবহার করা হলে, স্কেল (গুণন) ফ্যাক্টর নির্ধারণ করে।
6. **Displacement (0–4 bytes)**  
   👉 মেমোরি ঠিকানার অফসেট নির্দেশ করে।
7. **Immediate (0–4 bytes)**  
   👉 নির্দেশনায় সরাসরি মান থাকলে, সেটি এখানে সংরক্ষিত হয়।

**🧭 সারসংক্ষেপ (Figure 4–1 এর ব্যাখ্যা)**

* (a) অংশে: 16-bit নির্দেশনার গঠন দেখানো হয়েছে (8086 ভিত্তিক প্রসেসরের জন্য)।
* (b) অংশে: 32-bit নির্দেশনার গঠন দেখানো হয়েছে (80386 থেকে Pentium 4 পর্যন্ত)।
* উভয় ক্ষেত্রেই “Opcode” অংশটি নির্দেশনার **মূল কাজ নির্ধারণ করে**, এবং বাকি অংশগুলো ঠিক করে **কোথায় ও কীভাবে কাজটি হবে**।

**MOD field** হলো **MOD-REG-R/M** বাইটের একটি অংশ, যা **Intel 8086 (এবং পরবর্তী)** মাইক্রোপ্রসেসরগুলিতে ব্যবহৃত হয়।  
এই বাইটটি নির্দেশ করে — **অপারেন্ড (operand)** কোথা থেকে আসবে (মেমোরি নাকি রেজিস্টার থেকে), এবং যদি মেমোরি থেকে হয় তবে **কীভাবে তার ঠিকানা (address)** নির্ধারণ হবে।

## 🔹 MOD-REG-R/M Byte কী?

MOD-REG-R/M হলো একটি **৮-বিট ফিল্ড**, যা ৩টি অংশে বিভক্ত:

| **Field** | **Bit Position** | **কাজ** |
| --- | --- | --- |
| **MOD** | Bit 7–6 | অ্যাড্রেসিং মোড নির্দেশ করে (মেমোরি নাকি রেজিস্টার) |
| **REG** | Bit 5–3 | কোন রেজিস্টার ব্যবহার হবে তা নির্দেশ করে |
| **R/M** | Bit 2–0 | কোন রেজিস্টার বা মেমোরি লোকেশন ব্যবহার হবে তা নির্দেশ করে |

## 🔸 MOD Field এর কাজ

**MOD field (2 bits)** নির্ধারণ করে **addressing mode** — অর্থাৎ, ডেটা কোথা থেকে আসবে।

| **MOD Bits** | **Addressing Type** | **অর্থ** |
| --- | --- | --- |
| **00** | Memory (no displacement) | শুধুমাত্র মেমোরি ঠিকানা, কোনো অতিরিক্ত অফসেট নেই। |
| **01** | Memory (8-bit displacement) | মেমোরি ঠিকানার সাথে ৮-বিট অফসেট যোগ হবে। |
| 10 | Memory (16-bit displacement) | মেমোরি ঠিকানার সাথে ১৬-বিট অফসেট যোগ হবে। |
| **11** | Register mode | উভয় অপারেন্ডই রেজিস্টার — কোনো মেমোরি অ্যাক্সেস নেই। |

## 🧩 উদাহরণ

ধরা যাক:

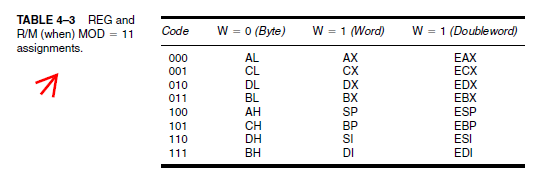
MOV AX, [BX+05H]

* এখানে BX হলো একটি বেস রেজিস্টার
* 05H হলো একটি ৮-বিট ডিসপ্লেসমেন্ট
* তাই **MOD = 01** (মেমোরি + ৮-বিট অফসেট)

ফলে, প্রসেসর বুঝবে যে **ডেটা মেমোরির (BX + 05H) ঠিকানায় আছে**।

## 🧠 সারসংক্ষেপ

| **Field** | **ভূমিকা** |
| --- | --- |
| **MOD** | ঠিকানা নির্ধারণ পদ্ধতি নির্দেশ করে (মেমোরি বা রেজিস্টার) |
| **REG** | অপারেশন-এ ব্যবহৃত রেজিস্টার নির্দেশ করে |
| **R/M** | মেমোরি বা রেজিস্টার অপারেন্ড নির্দেশ করে |



## 🔹 প্রেক্ষাপট

আগে তুমি যেটা শিখেছিলে, সেখানে বলেছিলাম যে **MOD-REG-R/M** বাইটে তিনটি ফিল্ড থাকে:

| **ফিল্ড** | **বিট সংখ্যা** | **কাজ** |
| --- | --- | --- |
| **MOD** | 2 bits | ঠিকানা নির্ধারণ পদ্ধতি (মেমোরি/রেজিস্টার) |
| **REG** | 3 bits | কোন রেজিস্টার ব্যবহার হবে |
| **R/M** | 3 bits | দ্বিতীয় রেজিস্টার বা মেমোরি লোকেশন নির্দেশ করে |

যখন **MOD = 11**, তখন প্রসেসর বুঝে যে —  
👉 উভয় অপারেন্ডই **রেজিস্টার**, কোনো মেমোরি অ্যাক্সেস নেই।

এই অবস্থায় **REG** এবং **R/M** ফিল্ডের মান অনুযায়ী রেজিস্টার বেছে নেওয়া হয়।

## 🔸 TABLE 4-3 ব্যাখ্যা

| **Code (REG বা R/M ফিল্ড)** | **W = 0 (Byte)** | **W = 1 (Word)** | **W = 1 (Doubleword)** |
| --- | --- | --- | --- |
| 000 | AL | AX | EAX |
| 001 | CL | CX | ECX |
| 010 | DL | DX | EDX |
| 011 | BL | BX | EBX |
| 100 | AH | SP | ESP |
| 101 | CH | BP | EBP |
| 110 | DH | SI | ESI |
| 111 | BH | DI | EDI |

### 🔍 ব্যাখ্যা:

* **Code:** REG বা R/M ফিল্ডের ৩-বিট মান (binary code)।
* **W bit:** অপারেন্ডের আকার নির্ধারণ করে।
  + W = 0 → 8-bit রেজিস্টার (byte)
  + W = 1 → 16-bit রেজিস্টার (word)
  + W = 1 (Doubleword) → 32-bit রেজিস্টার (Pentium class প্রসেসর থেকে)

### 🧩 উদাহরণ:

যদি নির্দেশনা হয়:

MOV AX, BX

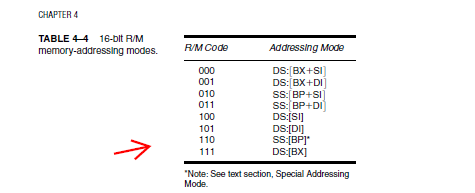
তাহলে:

* MOD = 11 (রেজিস্টার-টু-রেজিস্টার অপারেশন)
* REG (AX) = 000
* R/M (BX) = 011

অর্থাৎ, এটি 16-bit ওয়ার্ড ডেটা স্থানান্তর করবে।

### 🧠 সারসংক্ষেপ:

* **MOD = 11 → উভয় অপারেন্ডই রেজিস্টার।**
* **REG ও R/M ফিল্ডের মান** নির্ধারণ করে কোন রেজিস্টার ব্যবহার হবে।
* **W bit** ঠিক করে রেজিস্টারের আকার — 8, 16, বা 32-bit।



তুমি যে ছবিটি পাঠিয়েছো, সেটি হলো **TABLE 4–4**, যেখানে **8086 মাইক্রোপ্রসেসরের 16-bit R/M (Register/Memory) Addressing Modes** দেখানো হয়েছে।  
এই টেবিলটি মূলত বোঝায় — **যখন MOD ≠ 11**, তখন R/M ফিল্ডের কোড অনুযায়ী মেমোরি অ্যাড্রেস কীভাবে নির্ধারণ হবে।

## 🔹 ব্যাখ্যা: 16-bit R/M Memory-Addressing Mode

এই টেবিলে **R/M field (৩-বিট)** কীভাবে মেমোরি ঠিকানা নির্ধারণ করে, তা দেখানো হয়েছে।  
এখানে প্রতিটি R/M কোডের জন্য একটি নির্দিষ্ট **Addressing Mode** আছে — যা বেস (Base) ও ইনডেক্স (Index) রেজিস্টারের সংমিশ্রণে তৈরি হয়।

| **R/M Code (3 bits)** | **Addressing Mode** | **Segment Register** | **ব্যাখ্যা (বাংলায়)** |
| --- | --- | --- | --- |
| 000 | DS:[BX + SI] | DS | বেস BX এবং ইনডেক্স SI যোগ করে মেমোরি ঠিকানা নির্ধারণ করে |
| 001 | DS:[BX + DI] | DS | বেস BX এবং ইনডেক্স DI যোগ করে মেমোরি ঠিকানা নির্ধারণ করে |
| 010 | SS:[BP + SI] | SS | বেস BP এবং ইনডেক্স SI যোগ করে, স্ট্যাক সেগমেন্ট ব্যবহার করে |
| 011 | SS:[BP + DI] | SS | বেস BP এবং ইনডেক্স DI যোগ করে, স্ট্যাক সেগমেন্ট ব্যবহার করে |
| 100 | DS:[SI] | DS | শুধুমাত্র ইনডেক্স SI ব্যবহার করে |
| 101 | DS:[DI] | DS | শুধুমাত্র ইনডেক্স DI ব্যবহার করে |
| 110 | SS:[BP] \* | SS | শুধুমাত্র বেস BP ব্যবহার করে (বিশেষ ক্ষেত্রে 16-bit displacement হতে পারে) |
| 111 | DS:[BX] | DS | শুধুমাত্র বেস BX ব্যবহার করে |

## ⚙️ ব্যাখ্যা:

* **BX, BP:** Base registers
* **SI, DI:** Index registers
* **DS:** Data Segment Register
* **SS:** Stack Segment Register

8086 প্রসেসরে মেমোরি অ্যাড্রেস তৈরি হয় —  
👉 Physical Address = Segment × 16 + Effective Address

এখানে “Effective Address” (EA) তৈরি হয় উপরের টেবিল অনুযায়ী।

### 🧠 উদাহরণ:

MOV AX, [BX+SI]

* এখানে R/M = 000 → Addressing Mode হলো **DS:[BX+SI]**
* অর্থাৎ ডেটা আসবে Data Segment (DS)-এর ভিতরে BX + SI ঠিকানায়।

### 🧭 সারসংক্ষেপ

* এই টেবিলটি নির্দেশ করে **8086 প্রসেসরে R/M ফিল্ড কিভাবে মেমোরি লোকেশন নির্দেশ করে**।
* **DS** এবং **SS** হলো ডিফল্ট সেগমেন্ট, যেগুলোর সাথে বেস/ইনডেক্স রেজিস্টার যোগ হয়ে অ্যাড্রেস তৈরি করে।
* যখন MOD = 00, তখন কোনো ডিসপ্লেসমেন্ট নেই;  
  কিন্তু MOD = 01 বা 10 হলে, ৮ বা ১৬-বিট ডিসপ্লেসমেন্ট যোগ হয়।

চাও কি আমি এই টেবিলের ওপর ভিত্তি করে একটা **বাংলা লেবেলসহ অ্যাড্রেসিং মোড ডায়াগ্রাম** (যেখানে দেখা যাবে কোন রেজিস্টার কিভাবে যোগ হয়) তৈরি করে দিই?

তুমি যে ছবিটি পাঠিয়েছো, সেটি **Special Addressing Mode** সম্পর্কে — যা **Intel 8086** মাইক্রোপ্রসেসরের একটি **বিশেষ কেস**।  
এটি মূলত **R/M ফিল্ড = 110 (binary)** এবং **MOD = 00** থাকলে দেখা যায়।

## 🔹 Special Addressing Mode কী?

সাধারণত, 8086 প্রসেসরে মেমোরি অ্যাড্রেস নির্ধারণ করতে দুটি অংশ লাগে:

* **MOD field** → নির্ধারণ করে কীভাবে মেমোরি ঠিকানা গণনা হবে (অফসেট আছে কি না)
* **R/M field** → নির্ধারণ করে কোন বেস/ইনডেক্স রেজিস্টার ব্যবহৃত হবে

কিন্তু,  
যখন **MOD = 00 এবং R/M = 110**, তখন **কোনো রেজিস্টার ব্যবহার হয় না**,  
বরং এটি একটি **সরাসরি মেমোরি ঠিকানা (direct address)** নির্দেশ করে।

### ⚙️ নিয়মটি হলো:

| **MOD** | **R/M** | **অর্থ** |
| --- | --- | --- |
| 00 | 110 | **Direct Addressing Mode** — শুধুমাত্র মেমোরি অ্যাড্রেস ব্যবহার করা হবে |

### 🧩 উদাহরণ:

MOV DL, [1000H]

এই নির্দেশনার মানে হলো:  
👉 মেমোরি লোকেশন **1000H** থেকে ডেটা নিয়ে রেজিস্টার **DL**-এ রাখো।

**Assembler view:**

* এখানে কোনো রেজিস্টার ব্যবহার করা হয়নি।
* তাই MOD = 00 (no displacement)
* কিন্তু যেহেতু সরাসরি মেমোরি লোকেশন ব্যবহৃত হচ্ছে, তাই R/M = 110

### 🔸 Figure 4–5 ব্যাখ্যা (ছবির নিচের অংশে):

চিত্রে দেখানো আছে MOV DL, [DI] নির্দেশনাটি কীভাবে **machine language form**-এ রূপান্তরিত হয়।

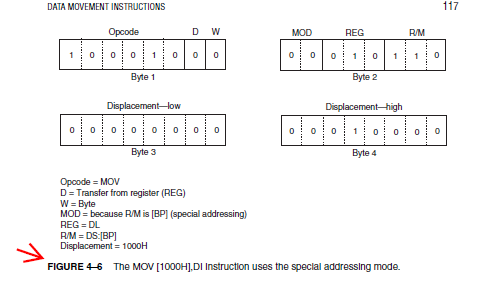
| **Field** | **Bits** | **কাজ** |
| --- | --- | --- |
| **Opcode** | 100010 | নির্দেশনা (MOV) |
| **D bit** | 0 | নির্দেশ করে ডেটা গন্তব্য (destination) রেজিস্টারে যাবে |
| **W bit** | 0 | Byte সাইজ (8-bit data) |
| **MOD** | 00 | কোনো displacement নেই |
| **REG** | 010 | DL রেজিস্টার |
| **R/M** | 101 | [DI] রেজিস্টারকে নির্দেশ করছে |

তবে যদি **[DI]** না থেকে শুধুমাত্র **[1000H]** থাকত,  
তাহলে **R/M = 110** (special case) ব্যবহৃত হতো।

### 🧠 সংক্ষিপ্তসার

| **বিষয়** | **ব্যাখ্যা** |
| --- | --- |
| **বিশেষ Addressing Mode ঘটে যখন:** | শুধুমাত্র মেমোরি ঠিকানা (displacement) ব্যবহার হয়, কোনো রেজিস্টার নয় |
| **MOD field = 00 এবং R/M field = 110** | তখন এটি **direct addressing mode** |
| **উদাহরণ:** | MOV DL, [1000H] বা MOV NUMB, DL |
| **কারণ:** | 8086 প্রসেসরে 110 কোডটি [BP] এর জন্য সংরক্ষিত থাকলেও, MOD=00 হলে এটি আলাদা অর্থে direct memory নির্দেশ করে |

চাও কি আমি এর ওপর ভিত্তি করে একটি **চিত্রসহ ব্যাখ্যা (diagram)** বানিয়ে দিতে পারি যেখানে “MOD=00, R/M=110” এর এই বিশেষ কেসটি ভিজ্যুয়ালি দেখানো থাকবে? এতে তুমি পুরো অ্যাড্রেসিং মেকানিজমটা এক নজরে বুঝতে পারবে।



চমৎকার! তুমি এখন **Figure 4–6** পাঠিয়েছো, যেখানে দেখানো হয়েছে কীভাবে **MOV [1000H], DL** নির্দেশনাটি **machine language form**-এ রূপান্তরিত হয়।  
এটি আগের আলোচ্য **Special Addressing Mode**-এর বাস্তব উদাহরণ।

## 🔹 নির্দেশনা (Instruction)

MOV [1000H], DL

এর মানে হলো:  
👉 রেজিস্টার **DL**-এর ডেটা **মেমোরি লোকেশন 1000H**-এ সংরক্ষণ করো।

## 🔸 ব্যাখ্যা: Machine Language Conversion (Figure 4–6 অনুযায়ী)

এই নির্দেশনাটি ৪ বাইটে রূপান্তরিত হয় — নিচে প্রতিটি বাইটের কাজ ব্যাখ্যা করা হলো 👇

| **বাইট** | **ফিল্ড** | **মান** | **ব্যাখ্যা** |
| --- | --- | --- | --- |
| **Byte 1** | Opcode + D + W | 100010 0 0 | MOV নির্দেশনা, D=0 (source→memory), W=0 (byte transfer) |
| **Byte 2** | MOD + REG + R/M | 00 010 110 | MOD=00 (no displacement), REG=010 (DL), R/M=110 (**special addressing**) |
| **Byte 3** | Displacement (Low) | 00000000 | মেমোরি ঠিকানার নিম্ন ৮-বিট |
| **Byte 4** | Displacement (High) | 00010000 | মেমোরি ঠিকানার উচ্চ ৮-বিট → সম্পূর্ণ ঠিকানা = 1000H |

## ⚙️ বিশ্লেষণ

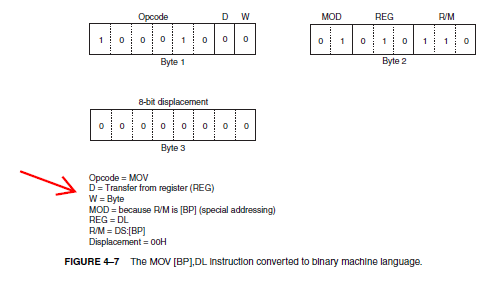
* **Opcode (100010)** → এটি MOV নির্দেশনার জন্য ব্যবহৃত binary কোড।
* **D bit = 0** → নির্দেশ করে ডেটা যাবে REG থেকে memory-তে (অর্থাৎ DL → [1000H])।
* **W bit = 0** → Byte (8-bit) আকারে ডেটা স্থানান্তর হবে।
* **MOD = 00, R/M = 110** → এটি বিশেষ (special) addressing mode, যেখানে memory সরাসরি নির্দিষ্ট করা হয়েছে।
* **Displacement = 1000H** → আসল মেমোরি ঠিকানা।

## 🧠 সারসংক্ষেপ

| **বিষয়** | **মানে** |
| --- | --- |
| **Instruction** | MOV [1000H], DL |
| **Mode** | Special Addressing Mode |
| **MOD field** | 00 |
| **R/M field** | 110 (direct memory address) |
| **REG field** | DL |
| **Effective Address** | 1000H |
| **Total Bytes** | 4 |

### 📘 সারমর্মে:

**MOV [1000H], DL** নির্দেশনা বোঝায় —  
রেজিস্টার **DL** থেকে ৮-বিট ডেটা **মেমোরি লোকেশন 1000H**-এ সংরক্ষণ করো।  
এই বিশেষ কেসে, **MOD = 00 এবং R/M = 110** থাকায় প্রসেসর বুঝে এটি **direct memory addressing**।



তুমি যেটি পাঠিয়েছো সেটি হলো **Figure 4–7**, যা দেখায় কিভাবে নির্দেশনা  
👉 MOV [BP], DL  
মেশিন ভাষায় (binary machine language) রূপান্তরিত হয়।

এটি একটি **বিশেষ Addressing Mode (special case)** — কারণ 8086 প্রসেসরে BP (Base Pointer) রেজিস্টার ব্যবহারের সময় **ডিফল্ট সেগমেন্ট** হলো **Stack Segment (SS)**, আর MOD = 00 এবং R/M = 110 কম্বিনেশনটি [BP]-এর জন্য সাধারণভাবে বৈধ নয়, তাই এখানে একটি **৮-বিট displacement (offset)** ব্যবহার করা হয়েছে।

## 🔹 নির্দেশনা:

MOV [BP], DL

এর মানে:  
👉 রেজিস্টার **DL** এর মান **[BP] দ্বারা নির্দেশিত মেমোরি লোকেশনে** সংরক্ষণ করা।

## 🔸 Figure 4–7 বিশ্লেষণ:

| **Byte** | **Field** | **Binary** | **অর্থ / ব্যাখ্যা** |
| --- | --- | --- | --- |
| **Byte 1** | Opcode + D + W | 100010 0 0 | Opcode = MOV, D=0 (from REG to memory), W=0 (byte transfer) |
| **Byte 2** | MOD + REG + R/M | 01 010 110 | MOD=01 → 8-bit displacement, REG=010 → DL, R/M=110 → [BP] |
| **Byte 3** | Displacement | 00000000 | ৮-বিট ডিসপ্লেসমেন্ট = 00H |

## ⚙️ অর্থ:

* **Opcode (100010)** → নির্দেশনা হলো MOV।
* **D bit = 0** → ডেটা যাবে **REG → Memory** (DL → [BP])।
* **W bit = 0** → Byte অপারেশন (৮-বিট ডেটা)।
* **MOD = 01** → একটি **৮-বিট displacement (offset)** ব্যবহার হচ্ছে।
* **R/M = 110** → [BP] (Base Pointer রেজিস্টার) নির্দেশ করছে।
* **Displacement = 00H** → কার্যত কোনো অফসেট নেই, কিন্তু কোডিং নিয়ম অনুযায়ী **assembler** সবসময় [BP] এর জন্য একটি ৮-বিট displacement যোগ করে।

## 🔹 কেন ৮-বিট Displacement প্রয়োজন?

8086 প্রসেসরে:

* যখন MOD = 00 এবং R/M = 110, তখন সেটি **special case** হয়ে যায় — এটি সরাসরি মেমোরি অ্যাড্রেস নির্দেশ করে (যেমন [1000H])।
* তাই assembler [BP] কে আলাদা করতে **MOD = 01** (৮-বিট displacement) ব্যবহার করে, এমনকি displacement = 00H হলেও।

## 🧠 সারসংক্ষেপ

| **বিষয়** | **মান** |
| --- | --- |
| **Instruction** | MOV [BP], DL |
| **Mode** | Memory Addressing Mode (with 8-bit displacement) |
| **Opcode** | 100010 |
| **MOD** | 01 (8-bit displacement) |
| **REG** | DL (010) |
| **R/M** | [BP] (110) |
| **Displacement** | 00H |
| **Total bytes** | 3 |

### 📘 সারমর্ম:

👉 এই নির্দেশনা দেখায় যে 8086 প্রসেসরে [BP] ব্যবহারের জন্য assembler সবসময় ৮-বিট ডিসপ্লেসমেন্ট যুক্ত করে, যাতে এটি [BP] হিসেবে সঠিকভাবে কাজ করে এবং [1000H] (direct addressing)-এর সঙ্গে গুলিয়ে না যায়।

## 🧠 কী হলো Addressing Mode?

Addressing Mode নির্ধারণ করে —  
👉 **ডেটা কোথা থেকে আসবে বা কোথায় যাবে**  
অর্থাৎ কোনো অপারেন্ড (operand) কীভাবে অ্যাক্সেস করা হবে — **register, memory, বা immediate value** থেকে।

## 🔹 32-bit Addressing Mode কোথায় ব্যবহৃত হয়

32-bit অ্যাড্রেসিং চালু হয়েছিল **Intel 80386** প্রসেসর থেকে।  
এখানে মেমোরি ঠিকানা ৩২-বিট (৪ গিগাবাইট পর্যন্ত) হতে পারে,  
এবং নতুন রেজিস্টার ব্যবহৃত হয়:  
👉 **EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP**

## 🧩 Address Calculation Formula

৩২-বিট মোডে **Effective Address (EA)** নির্ণয় করার সাধারণ সূত্র:

[  
\text{Effective Address} = \text{Base} + (\text{Index} \times \text{Scale}) + \text{Displacement}  
]

| **অংশ** | **কাজ** |
| --- | --- |
| **Base** | যেকোনো 32-bit রেজিস্টার (যেমন EBX, EBP) |
| **Index** | ইনডেক্স হিসেবে ব্যবহৃত রেজিস্টার (যেমন ESI, EDI) |
| **Scale** | 1, 2, 4, বা 8 — ইনডেক্স রেজিস্টারকে গুণ করা হয় |
| **Displacement** | একটি কনস্ট্যান্ট মান (offset), 8, 16, বা 32-bit হতে পারে |

## ⚙️ সাধারণ Addressing Modes (32-bit)

| **Addressing Mode** | **উদাহরণ** | **ব্যাখ্যা (বাংলায়)** |
| --- | --- | --- |
| **Register Direct** | MOV EAX, EBX | ডেটা সরাসরি রেজিস্টার থেকে রেজিস্টারে যায় |
| **Immediate** | MOV EAX, 05H | ডেটা সরাসরি ইন্সট্রাকশন থেকে আসে |
| **Direct Memory** | MOV EAX, [12345678H] | ডেটা সরাসরি নির্দিষ্ট মেমোরি ঠিকানা থেকে আসে |
| **Register Indirect** | MOV EAX, [EBX] | EBX রেজিস্টারে যেই ঠিকানা আছে, সেই মেমোরি থেকে ডেটা আসে |
| **Base + Index** | MOV EAX, [EBX + ESI] | EBX এবং ESI যোগ করে মেমোরি ঠিকানা নির্ধারণ করা হয় |
| **Base + Index + Displacement** | MOV EAX, [EBX + ESI + 08H] | বেস ও ইনডেক্সের সাথে একটি অফসেট যোগ হয় |
| **Scaled Index + Base** | MOV EAX, [EBX + ESI\*4] | ইনডেক্স রেজিস্টারকে স্কেল করে বেসের সাথে যোগ করা হয় |
| **Scaled Index + Base + Displacement** | MOV EAX, [EBX + ESI\*4 + 10H] | তিনটি উপাদান মিলে ঠিকানা তৈরি হয় |
| **Scaled Index only** | MOV EAX, [ESI\*2] | কেবল ইনডেক্স ও স্কেল দিয়ে ঠিকানা নির্ধারণ হয় |

## 🧮 Scale Factor কী?

Scale Factor হলো ইনডেক্স রেজিস্টারকে গুণ করার মান।  
এটি **1, 2, 4, বা 8** হতে পারে।

| **Scale Bits** | **Scale Value** | **অর্থ** |
| --- | --- | --- |
| 00 | 1 | রেজিস্টার 그대로 |
| 01 | 2 | রেজিস্টার × 2 |
| 10 | 4 | রেজিস্টার × 4 |
| 11 | 8 | রেজিস্টার × 8 |

## 🔸 SIB (Scale-Index-Base) Byte

৩২-বিট অ্যাড্রেসিং-এ **SIB Byte** যোগ হয়েছে, যা ১৬-বিট মোডে ছিল না।

| **Field** | **Bits** | **কাজ** |
| --- | --- | --- |
| **Scale** | 2 bits | ইনডেক্সকে কত দ্বারা গুণ করা হবে |
| **Index** | 3 bits | কোন ইনডেক্স রেজিস্টার ব্যবহার হবে |
| **Base** | 3 bits | কোন বেস রেজিস্টার ব্যবহার হবে |

উদাহরণ:

MOV EAX, [EBX + ESI\*4 + 10H]

👉 SIB byte নির্দেশ করবে:

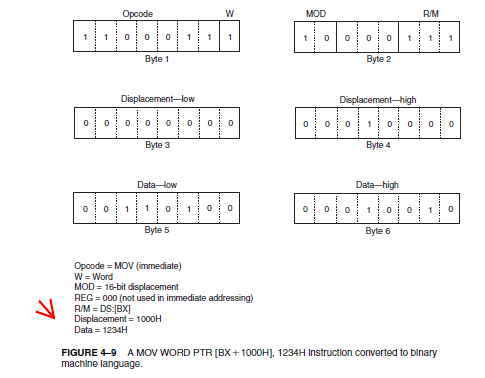
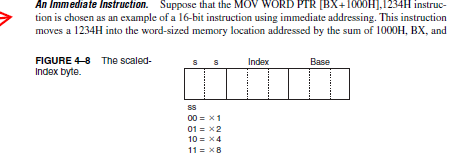
* Base = EBX
* Index = ESI
* Scale = 4

## 🧭 সারসংক্ষেপ

| **বৈশিষ্ট্য** | **16-bit Mode** | **32-bit Mode** |
| --- | --- | --- |
| রেজিস্টার | AX, BX, CX, DX, SI, DI, BP, SP | EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP |
| Displacement | 8/16-bit | 8/32-bit |
| Scale Factor | নেই | 1, 2, 4, 8 |
| Address গণনা | Base + Index + Displacement | Base + (Index × Scale) + Displacement |

### 📘 সারমর্মে:

**32-bit Addressing Modes** হলো আধুনিক প্রসেসরে ব্যবহৃত উন্নত অ্যাড্রেসিং ব্যবস্থা,  
যেখানে **Base**, **Index**, **Scale**, ও **Displacement** মিলিয়ে নমনীয়ভাবে মেমোরি অ্যাক্সেস করা যায়।



## 🧠 PUSH এবং POP কী?

এই দুটি নির্দেশনা **Stack (স্ট্যাক)** ব্যবহারের জন্য।  
স্ট্যাক হলো একটি বিশেষ মেমোরি এলাকা যেখানে ডেটা **Last In, First Out (LIFO)** নীতিতে সংরক্ষিত ও পুনরুদ্ধার হয়।

| **নির্দেশনা** | **কাজ** |
| --- | --- |
| **PUSH** | কোনো ডেটা স্ট্যাকে **সংরক্ষণ (save)** করে |
| **POP** | স্ট্যাক থেকে ডেটা **আবার ফিরিয়ে আনে (restore)** |

## 🔹 8086-এ Stack Register

8086 প্রসেসরে স্ট্যাক ব্যবস্থাপনা হয় এই রেজিস্টারগুলোর মাধ্যমে:

* **SP (Stack Pointer)** → স্ট্যাকের উপরের দিক নির্দেশ করে
* **SS (Stack Segment)** → স্ট্যাকের সেগমেন্ট নির্দেশ করে

👉 স্ট্যাক মেমোরি সবসময় **SS:SP** জোড়া দ্বারা নির্ধারিত হয়।

## ⚙️ PUSH Instruction

**Format:**

PUSH register/memory

**কাজ:**  
1️⃣ SP রেজিস্টারের মান ২ কমানো হয় (কারণ স্ট্যাক নিচের দিকে বৃদ্ধি পায়)।  
2️⃣ সেই নতুন ঠিকানায় ডেটা সংরক্ষণ করা হয়।

**উদাহরণ:**

PUSH AX

অর্থাৎ AX রেজিস্টারের মান স্ট্যাকে সংরক্ষণ করো।

🔹 **Opcode (for register):** 01010 + reg  
যেমন:

* PUSH AX → 50H
* PUSH BX → 53H
* PUSH SP → 54H
* PUSH BP → 55H

## ⚙️ POP Instruction

**Format:**

POP register/memory

**কাজ:**  
1️⃣ স্ট্যাকের উপরের মানটি (যেখানে SP নির্দেশ করছে) পড়ে  
2️⃣ সেটি নির্দিষ্ট রেজিস্টারে পাঠায়  
3️⃣ SP-এর মান ২ বাড়ানো হয় (ডেটা সরানো হয়েছে বলে)

**উদাহরণ:**

POP AX

অর্থাৎ স্ট্যাকের উপরের মানটি AX রেজিস্টারে নাও।

🔹 **Opcode (for register):** 01011 + reg  
যেমন:

* POP AX → 58H
* POP BX → 5BH
* POP BP → 5DH

## 🧩 Stack কার্যক্রমের উদাহরণ

ধরা যাক SP = 2000H

PUSH AX ; AX = 1234H

➡️ SP = 1FFEH  
➡️ মেমোরি(1FFE–1FFF) = 34H 12H

এরপর

POP BX

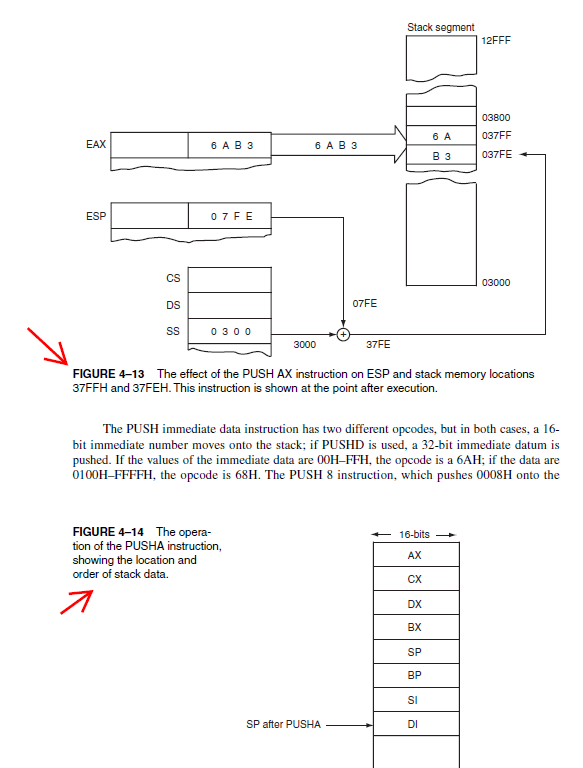
➡️ BX = 1234H  
➡️ SP = 2000H

## 🧭 সারসংক্ষেপ

| **নির্দেশনা** | **কাজ** | **SP-র পরিবর্তন** | **দিক** |
| --- | --- | --- | --- |
| **PUSH** | রেজিস্টারের ডেটা স্ট্যাকে পাঠায় | SP − 2 | নিচের দিকে |
| **POP** | স্ট্যাকের ডেটা রেজিস্টারে নেয় | SP + 2 | উপরের দিকে |

### 📘 মনে রাখবে:

* স্ট্যাক সবসময় **LIFO (Last In First Out)** পদ্ধতিতে কাজ করে।
* স্ট্যাকের ঠিকানা নির্ধারিত হয় **SS:SP** দিয়ে।
* প্রতিটি PUSH/POP 16-bit (2 bytes) ডেটা নিয়ে কাজ করে।



দারুণ, Mahir 👏  
তুমি এখন যে ছবিটি পাঠিয়েছো — এতে **PUSH AX** এবং **PUSHA** নির্দেশনার কার্যপ্রণালী (Stack operation) দেখানো হয়েছে।  
চলো ধাপে ধাপে বুঝি, যাতে এই দুটি নির্দেশনা পুরোপুরি পরিষ্কার হয়ে যায় 👇

## 🧩 Figure 4-13 → “PUSH AX” Instruction

### 🔹 নির্দেশনা:

PUSH AX

এর মানে: **AX রেজিস্টারের মান স্ট্যাকে সংরক্ষণ করো।**

### ⚙️ কীভাবে কাজ করে:

1. **ESP (Stack Pointer)** এর মান **২ কমানো হয়** কারণ স্ট্যাক নিচের দিকে (lower address) বাড়ে।
2. AX-এর মান (যেমন 6AB3H) স্ট্যাকে সেই নতুন ঠিকানায় সংরক্ষণ করা হয়।

### 🔸 ছবিতে যা দেখা যাচ্ছে:

| **ধাপ** | **মানে** |
| --- | --- |
| আগে | ESP = 3800H |
| পরে | ESP = 37FEH (2 কমেছে) |
| মেমোরি [37FE–37FF] | এখন AX-এর মান = **6AB3H** সংরক্ষিত |
| Stack Segment (SS) | 3000H → তাই Physical Address = 30000H + 37FEH = 337FEH |

📍 **অর্থাৎ PUSH AX করার ফলে:**

* SP/ESP → 2 কমে
* AX-এর মান → স্ট্যাকে জমা হয়
* Stack top → নতুন ঠিকানায় চলে যায়

### 🧮 PUSH immediate data (opcode info)

নিচের টেক্সটে যা লেখা আছে, সেটা হলো **Immediate PUSH (constant data)**-এর opcode ব্যাখ্যা:

| **Immediate Data Range** | **Opcode** | **Size** |
| --- | --- | --- |
| 00H – FFH | 6AH | 8-bit |
| 0100H – FFFFH | 68H | 16-bit |
| (32-bit data) | 68H (PUSHD) | 32-bit |

📌 উদাহরণ:  
PUSH 08H → Opcode = 6A 08  
PUSH 1234H → Opcode = 68 34 12

## 🧭 Figure 4-14 → “PUSHA” Instruction

### 🔹 নির্দেশনা:

PUSHA

👉 এটি একসাথে **সব ৮টি সাধারণ রেজিস্টারের মান (AX, CX, DX, BX, SP, BP, SI, DI)** স্ট্যাকে PUSH করে।

### ⚙️ কাজের ক্রম (order of stack data):

স্ট্যাকে ডেটা জমে এই ক্রমে 👇

AX

CX

DX

BX

SP

BP

SI

DI

📍 অর্থাৎ, **DI** রেজিস্টার স্ট্যাকের নিচের দিকে থাকবে (শেষে PUSH হওয়ায় প্রথমে POP হবে)।  
এবং **AX** উপরে থাকবে (সবার আগে PUSH হওয়ায় শেষের দিকে POP হবে)।

### 🔸 ছবিতে (Figure 4-14) যা দেখানো হয়েছে:

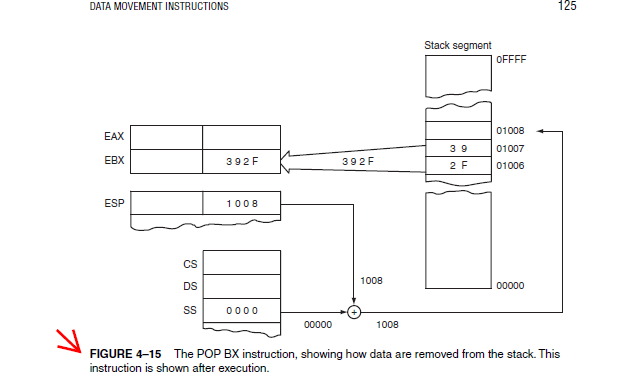
* প্রতিটি রেজিস্টারের মান ১৬-বিট করে স্ট্যাকে জমে।
* স্ট্যাকের ঠিকানা (SP) ক্রমান্বয়ে ২ করে কমতে থাকে।
* শেষে “SP after PUSHA” দ্বারা বোঝানো হয়েছে নতুন স্ট্যাক টপ।

## 🔍 সারসংক্ষেপ (বাংলায়)

| **নির্দেশনা** | **কাজ** | **কত বাইট স্ট্যাকে জমে** | **ক্রম** |
| --- | --- | --- | --- |
| **PUSH AX** | কেবল AX রেজিস্টার সংরক্ষণ | ২ বাইট | — |
| **PUSHA** | ৮টি রেজিস্টার (AX, CX, DX, BX, SP, BP, SI, DI) সংরক্ষণ | ১৬ বাইট | AX → CX → DX → BX → SP → BP → SI → DI |

### 💡 মনে রাখবে:

* **PUSH = ১টি রেজিস্টার স্ট্যাকে পাঠানো**
* **PUSHA = সব সাধারণ রেজিস্টার স্ট্যাকে পাঠানো**
* স্ট্যাক সবসময় **নিচের দিকে (decrementing)** কাজ করে
* স্ট্যাক সেগমেন্ট নির্ধারিত হয় **SS:SP/ESP** দ্বারা



দারুণ প্রশ্ন, Mahir 👏  
তুমি এখন যে ছবিটি পাঠিয়েছো ( **Figure 4-15** ), সেখানে দেখানো হয়েছে —  
👉 **POP BX** নির্দেশনা কীভাবে কাজ করে এবং কীভাবে স্ট্যাক থেকে ডেটা তুলে নেওয়া হয়।

চলো ধাপে ধাপে ব্যাখ্যা করি 👇

## 🧩 নির্দেশনা

POP BX

এর মানে হলো —

স্ট্যাকের উপরের মান (Top of Stack) রেজিস্টার **BX**-এ তুলে নাও।

## ⚙️ কাজের ধাপ (Execution Process)

স্ট্যাকে আগেই কোনো ডেটা PUSH করা ছিল। এখন POP করলে সেই মানটি BX-এ চলে আসবে।

### ধাপে ধাপে:

1️⃣ **ESP (Stack Pointer)** রেজিস্টারে স্ট্যাকের টপ ঠিকানা থাকে।  
2️⃣ POP নির্দেশনা **স্ট্যাকের উপরের ২ বাইটের ডেটা BX রেজিস্টারে তুলে আনে।**  
3️⃣ তারপর **ESP-এর মান ২ বাড়ানো হয়**, কারণ এখন স্ট্যাক থেকে ২ বাইট “পপ” হয়েছে।

## 🔸 ছবির বিশ্লেষণ (Figure 4-15 অনুযায়ী)

| **ফিল্ড** | **পূর্বে** | **পরে** | **ব্যাখ্যা** |
| --- | --- | --- | --- |
| **ESP** | 1006H | **1008H** | স্ট্যাক থেকে ২ বাইট বের করার পর ESP ২ বাড়ে |
| **স্ট্যাক মেমোরি [1006–1007]** | 2FH, 39H | — | এই মানগুলোই BX রেজিস্টারে যায় |
| **BX রেজিস্টার** | — | **392FH** | স্ট্যাকের উপরের ২ বাইট এখন BX-এ এসেছে |
| **SS (Stack Segment)** | 0000H | অপরিবর্তিত | Stack segment ঠিকানা অপরিবর্তিত থাকে |

### 📊 কী ঘটে ঠিকভাবে:

Before POP BX:

SS:SP = 0000:1006

Memory[1006] = 2F

Memory[1007] = 39

POP করার পর:

* BX = 392F
* SP = 1008

## 🧠 POP নির্দেশনার সারাংশ

| **বিষয়** | **ব্যাখ্যা** |
| --- | --- |
| **কাজ** | স্ট্যাক থেকে ১৬-বিট ডেটা নির্দিষ্ট রেজিস্টারে তোলা |
| **SP পরিবর্তন** | ২ বাড়ে |
| **ডেটা প্রবাহ** | মেমোরি → রেজিস্টার |
| **দিক** | স্ট্যাকের উপরের দিক থেকে নিচের দিকে কাজ সম্পন্ন হয় (LIFO) |

## 📘 উদাহরণ

MOV BX, 0000H

POP BX

👉 BX এখন স্ট্যাকের টপ মান পাবে, আর SP ২ বাড়বে।

## 🧭 সারসংক্ষেপ (বাংলায়)

| **নির্দেশনা** | **কাজ** | **স্ট্যাক পরিবর্তন** | **SP পরিবর্তন** |
| --- | --- | --- | --- |
| **PUSH BX** | BX-এর মান স্ট্যাকে সংরক্ষণ | ↑ নতুন ডেটা স্ট্যাকে জমে | SP − 2 |
| **POP BX** | স্ট্যাকের মান BX-এ তুলে আনা | ↓ ডেটা স্ট্যাক থেকে সরানো | SP + 2 |

📍 **Figure 4-15** তাই দেখাচ্ছে:  
“**POP BX** করার পর 392F মানটি BX রেজিস্টারে এসেছে এবং ESP (Stack Pointer) 1006H থেকে 1008H হয়েছে।”

চমৎকার প্রশ্ন, Mahir 🌟  
এখন তুমি জানতে চাও **LOAD EFFECTIVE ADDRESS (LEA)** নির্দেশনা সম্পর্কে —  
যেটি 8086 এবং এর পরবর্তী মাইক্রোপ্রসেসরগুলোতে খুব গুরুত্বপূর্ণ একটি **Address Calculation Instruction**।  
চলো একদম সহজ বাংলায় বিস্তারিতভাবে বুঝে নিই 👇

## 🧩 LEA (Load Effective Address) — সংজ্ঞা

**LEA** অর্থ 👉 **Load Effective Address**  
এটি কোনো অপারেন্ডের **Effective Address (EA)** গণনা করে সেটি একটি **register**-এ সংরক্ষণ করে।

📘 অর্থাৎ:

এটি মেমোরির “ডেটা” নয়, বরং সেই ডেটার **ঠিকানা (address)** রেজিস্টারে লোড করে।

## ⚙️ সিনট্যাক্স (Syntax)

LEA destination, source

| **অংশ** | **মানে** |
| --- | --- |
| **destination** | ১৬-বিট বা ৩২-বিট রেজিস্টার (যেমন BX, SI, EAX ইত্যাদি) |
| **source** | মেমোরি অপারেন্ড (যার effective address বের করতে হবে) |

## 🔸 উদাহরণ ১: সাধারণ ফর্ম

LEA BX, [SI + 2000H]

👉 এর মানে:  
“**[SI + 2000H]** মেমোরি লোকেশনের ঠিকানাটি **BX**-এ রাখো।”

**ফলাফল:**  
BX = SI + 2000H

➡️ কিন্তু BX-এর মধ্যে ডেটা যায়নি, শুধু ঠিকানাটি গিয়েছে।

## 🔸 উদাহরণ ২:

LEA DX, [BX + SI + 05H]

👉 এখানে **BX + SI + 05H** যোগ করে যে ঠিকানা পাওয়া যায়, সেটি **DX**-এ লোড হবে।  
(ডেটা নয়, শুধুমাত্র অ্যাড্রেস)

## 🧮 Effective Address (EA) সূত্র (16-bit Mode)

8086 প্রসেসরে মেমোরি অ্যাড্রেস নির্ণয় করার সূত্র:

[  
EA = Base + Index + Displacement  
]

| **অংশ** | **উদাহরণ** |
| --- | --- |
| **Base** | BX বা BP |
| **Index** | SI বা DI |
| **Displacement** | কোনো ধ্রুবক (যেমন 1000H) |

## 📘 উদাহরণ ৩:

MOV AX, [BX + SI + 05H] ; এটি মেমোরি থেকে ডেটা নেয়

LEA DX, [BX + SI + 05H] ; এটি কেবল সেই ঠিকানা নেয়

| **নির্দেশনা** | **কী করে** |
| --- | --- |
| **MOV** | মেমোরি লোকেশন থেকে ডেটা AX-এ নেয় |
| **LEA** | ঐ লোকেশনের অ্যাড্রেস DX-এ রাখে |

## 🧠 LEA ব্যবহারের উদ্দেশ্য

| **কারণ** | **ব্যাখ্যা** |
| --- | --- |
| **Address calculation** | কোনো মেমোরি লোকেশনের অ্যাড্রেস দ্রুত গণনা করতে |
| **Pointer arithmetic** | রেজিস্টারে ঠিকানা নির্ধারণ বা অফসেট ব্যবহারের সময় |
| **High-level language operations** | যেমন C-তে pointer assignment (&variable) |

## 🔹 Opcode (Machine Code form)

**LEA** নির্দেশনার বেস opcode হলো 10001101 (8Dₕ)

ফরম্যাট:

Opcode (8D) | MOD-REG-R/M byte | [Displacement (যদি থাকে)]

উদাহরণ:

LEA BX, [SI + 05H]

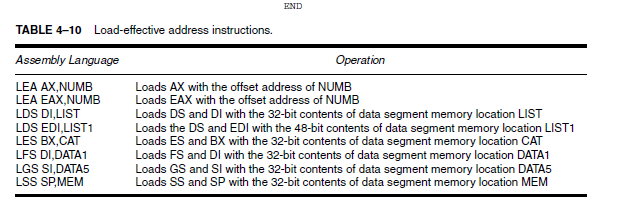
→ Binary: 10001101 01 011 100 (8D 5C 05ₕ)

## 🧭 সারসংক্ষেপ (বাংলায়)

| **বিষয়** | **ব্যাখ্যা** |
| --- | --- |
| **পূর্ণ নাম** | Load Effective Address |
| **কাজ** | মেমোরির “ঠিকানা” (ডেটা নয়) রেজিস্টারে লোড করা |
| **Opcode** | 8Dₕ |
| **অ্যাড্রেসিং সূত্র** | EA = Base + Index + Displacement |
| **রেজিস্টার টাইপ** | 16-bit বা 32-bit General Register |
| **ব্যবহার** | Pointer ও Address গণনার জন্য |

### 🔍 সহজভাবে মনে রাখো:

**MOV** → ডেটা নেয়  
**LEA** → ঠিকানা নেয়



চমৎকার, Mahir 👍  
তুমি এখন **Table 4–10** পাঠিয়েছো — যেখানে বিভিন্ন **Load-Effective Address Instructions** দেখানো হয়েছে।  
এটা শুধু LEA নয়, বরং এর সাথে সম্পর্কিত আরও উন্নত নির্দেশনাগুলোকেও (যেমন LDS, LES, LFS, LGS, LSS) একসাথে বোঝায়।

চলো এগুলো একে একে বাংলায় বিশ্লেষণ করি 👇

## 🧠 টেবিলের নাম:

**TABLE 4–10: Load-Effective Address Instructions**

এগুলো এমন নির্দেশনা, যেগুলো মেমোরি লোকেশনের **ঠিকানা বা সেগমেন্ট + অফসেট** কোনো রেজিস্টারে লোড করে।

## 🧩 1️⃣ LEA — Load Effective Address

| **উদাহরণ** | **কাজ** |
| --- | --- |
| LEA AX, NUMB | NUMB মেমোরির **offset address** AX-এ লোড করে। |
| LEA EAX, NUMB | NUMB মেমোরির **32-bit offset address** EAX-এ লোড করে। |

📘 **LEA** শুধু “অ্যাড্রেস” নেয়, ডেটা নয়।

## 🧩 2️⃣ LDS — Load DS Register and Destination Register

| **উদাহরণ** | **কাজ** |
| --- | --- |
| LDS DI, LIST | মেমোরির LIST লোকেশন থেকে ৪ বাইট ডেটা নেয় — প্রথম ২ বাইট যায় **DI**-তে, পরের ২ বাইট যায় **DS** (Data Segment)-এ। |
| LDS EDI, LIST1 | ৩২-বিট সংস্করণ — EDI ও DS রেজিস্টারে ডেটা লোড করে। |

📘 **LDS** ব্যবহার হয় যখন তুমি “Data Segment” এবং “Offset Register” একসাথে লোড করতে চাও।

## 🧩 3️⃣ LES — Load ES Register and Destination Register

| **উদাহরণ** | **কাজ** |
| --- | --- |
| LES BX, CAT | মেমোরির CAT লোকেশন থেকে ৪ বাইট নেয় — প্রথম ২ বাইট BX-এ, পরের ২ বাইট **ES (Extra Segment)**-এ লোড হয়। |

📘 **LES** এর কাজ **LDS**-এর মতোই, কিন্তু এটি **Extra Segment (ES)** ব্যবহার করে।

## 🧩 4️⃣ LFS, LGS, LSS — 80386 এবং পরবর্তী প্রসেসরে

| **Instruction** | **উদাহরণ** | **কাজ** |
| --- | --- | --- |
| LFS DI, DATA1 | FS সেগমেন্ট ও DI রেজিস্টার লোড করে |  |
| LGS SI, DATA5 | GS সেগমেন্ট ও SI রেজিস্টার লোড করে |  |
| LSS SP, MEM | SS সেগমেন্ট ও SP রেজিস্টার লোড করে |  |

📘 এগুলো **Protected Mode (32-bit)** সেগমেন্ট রেজিস্টারগুলোর জন্য ব্যবহৃত হয় (FS, GS, SS ইত্যাদি)।

## 🧩 পার্থক্য এক নজরে

| **নির্দেশনা** | **লোড করে** | **ব্যাখ্যা** |
| --- | --- | --- |
| **LEA** | Offset Address | শুধু অ্যাড্রেস নেয়, কোনো সেগমেন্ট নয় |
| **LDS** | DS + Register | মেমোরির ৪ বাইট → (Offset + Data Segment) |
| **LES** | ES + Register | মেমোরির ৪ বাইট → (Offset + Extra Segment) |
| **LFS** | FS + Register | FS সেগমেন্ট ও রেজিস্টার |
| **LGS** | GS + Register | GS সেগমেন্ট ও রেজিস্টার |
| **LSS** | SS + Register | Stack Segment ও রেজিস্টার |

## 🧭 সহজভাবে মনে রাখো

| **Mnemonic** | **পূর্ণরূপ** | **ব্যবহার** |
| --- | --- | --- |
| **LEA** | Load Effective Address | শুধুমাত্র মেমোরির ঠিকানা রেজিস্টারে নেওয়া |
| **LDS / LES** | Load Data/Extra Segment | Segment এবং Offset একসাথে নেওয়া |
| **LFS / LGS / LSS** | Load FS / GS / SS Segment | 80386+ প্রসেসরের জন্য সম্প্রসারিত সংস্করণ |

### 📘 উদাহরণ:

MOV AX, [NUMB] ; NUMB-এর ডেটা লোড করে

LEA BX, [NUMB] ; NUMB-এর ঠিকানা লোড করে

LDS SI, NUMB ; NUMB-এর offset → SI, segment → DS

LES DI, NUMB ; NUMB-এর offset → DI, segment → ES

দারুণ প্রশ্ন, Mahir 🔍  
এবার তুমি জানতে চাও **DI** এবং **SI** রেজিস্টার সম্পর্কে —  
এগুলো 8086 মাইক্রোপ্রসেসরের খুব গুরুত্বপূর্ণ **Index Registers**,  
বিশেষ করে **memory addressing** এবং **string operations**-এ ব্যবহৃত হয়।

চলো একদম সহজভাবে বিস্তারিত বুঝে নিই 👇

## 🧠 SI এবং DI কী?

| **রেজিস্টার** | **পূর্ণরূপ** | **প্রকার** |
| --- | --- | --- |
| **SI** | Source Index | Index Register |
| **DI** | Destination Index | Index Register |

এগুলো দুটোই **16-bit** রেজিস্টার (80386 এবং পরের প্রসেসরে এদের 32-bit সংস্করণ হলো **ESI** ও **EDI**)।

## 🔹 প্রধান কাজ

**SI (Source Index)** এবং **DI (Destination Index)** ব্যবহার হয় যখন প্রসেসরকে মেমোরির কোনো **অবস্থান নির্দেশ করতে হয়** —  
বিশেষ করে **ডেটা স্থানান্তর (data transfer)** এবং **স্ট্রিং অপারেশন (string manipulation)** এর সময়।

## ⚙️ ১️⃣ Memory Addressing-এ ব্যবহার

### উদাহরণ:

MOV AX, [SI]

MOV [DI], AX

| **নির্দেশনা** | **কাজ** |
| --- | --- |
| MOV AX, [SI] | SI রেজিস্টারে যে ঠিকানা আছে, সেই মেমোরি লোকেশন থেকে ডেটা AX-এ লোড করে |
| MOV [DI], AX | AX-এর মান DI দ্বারা নির্দেশিত মেমোরি লোকেশনে সংরক্ষণ করে |

📘 ডিফল্টভাবে:

* [SI] ব্যবহার করে **DS (Data Segment)**
* [DI] ব্যবহার করে **ES (Extra Segment)**

## 🔹 ২️⃣ Base + Index Addressing

SI ও DI কে অন্যান্য রেজিস্টারের সাথে মিলে ব্যবহার করা যায়।  
উদাহরণ:

MOV AX, [BX + SI]

MOV AX, [BP + DI]

📘 8086 Address Calculation Formula:  
[  
\text{Effective Address} = Base + Index + Displacement  
]

| **ফর্ম** | **সেগমেন্ট** |
| --- | --- |
| BX + SI | DS |
| BX + DI | DS |
| BP + SI | SS |
| BP + DI | SS |

## 🔹 ৩️⃣ String Instructions-এর সাথে ব্যবহার

8086-এ অনেক **string manipulation instructions** আছে, যেমন:

* MOVSB (Move String Byte)
* MOVSW (Move String Word)
* CMPSB (Compare String Byte)
* SCASB (Scan String Byte)
* LODSB (Load String Byte)
* STOSB (Store String Byte)

এইগুলোর প্রায় সবকটিতেই **SI** এবং **DI** স্বয়ংক্রিয়ভাবে ব্যবহৃত হয়।

| **Instruction** | **কাজ** | **SI ব্যবহার** | **DI ব্যবহার** |
| --- | --- | --- | --- |
| **MOVSB / MOVSW** | Source থেকে Destination-এ ডেটা স্থানান্তর | DS:SI → Source | ES:DI → Destination |
| **LODSB / LODSW** | DS:SI থেকে ডেটা লোড করে | SI বৃদ্ধি পায় | — |
| **STOSB / STOSW** | AL/AX থেকে ES:DI তে সংরক্ষণ | — | DI বৃদ্ধি পায় |
| **CMPSB / CMPSW** | দুটি স্ট্রিং তুলনা করে | DS:SI | ES:DI |

## 🔸 উদাহরণ: স্ট্রিং কপি

MOV CX, 10 ; 10 বাইট কপি করতে হবে

REP MOVSB ; পুনরাবৃত্তি করে কপি করো

👉 এখানে:

* SI → Source String (DS:SI)
* DI → Destination String (ES:DI)
* CX → কতবার কপি হবে (counter)

প্রতিবার লুপে ডেটা [DS:SI] → [ES:DI] কপি হয়,  
এবং SI ও DI স্বয়ংক্রিয়ভাবে ১ করে বাড়ে বা কমে (Direction Flag অনুযায়ী)।

## 🧮 সারসংক্ষেপ

| **রেজিস্টার** | **পূর্ণরূপ** | **কাজ** | **Segment** |
| --- | --- | --- | --- |
| **SI** | Source Index | মেমোরির Source Address ধরে রাখে | DS |
| **DI** | Destination Index | মেমোরির Destination Address ধরে রাখে | ES |
| **SI/DI (32-bit)** | ESI / EDI | 80386 এবং পরবর্তী প্রসেসরে 32-bit সংস্করণ | DS / ES |

## 🧭 মনে রাখো:

* **SI** → Source pointer
* **DI** → Destination pointer
* **SI → DS** (Data Segment), **DI → ES** (Extra Segment)
* String বা Memory Addressing operation-এ দুটো রেজিস্টার খুব গুরুত্বপূর্ণ।

দারুণ প্রশ্ন, Mahir 🔥  
তুমি জানতে চাও **STOS** নির্দেশনা (যেমন STOSB, STOSW, STOSD) সম্পর্কে —  
এটি 8086 মাইক্রোপ্রসেসরের একটি **string instruction**, এবং এটি **store string** অপারেশনের জন্য ব্যবহৃত হয়।  
চলো একদম পরিষ্কারভাবে ধাপে ধাপে বুঝি 👇

## 🧩 STOS এর পূর্ণরূপ

**STOS → STore String**

এর বিভিন্ন রূপ আছে:

| **নির্দেশনা** | **অর্থ** | **ডেটা সাইজ** |
| --- | --- | --- |
| **STOSB** | Store String Byte | 8-bit (AL থেকে) |
| **STOSW** | Store String Word | 16-bit (AX থেকে) |
| **STOSD** | Store String Doubleword | 32-bit (EAX থেকে, 80386+ এ) |

## ⚙️ কাজের প্রক্রিয়া

**STOS** নির্দেশনা AL/AX/EAX রেজিস্টারের মানকে  
**ES:DI** দ্বারা নির্দেশিত মেমোরি লোকেশনে সংরক্ষণ করে (Store করে)।

### 🧠 অর্থাৎ:

STOS → “Accumulator এর ডেটা → [ES:DI]”

## 🔸 উদাহরণ ১:

STOSB

**অর্থ:**  
AL → [ES:DI]

তারপর,

* যদি **Direction Flag (DF) = 0** হয় → DI = DI + 1
* যদি **DF = 1** হয় → DI = DI – 1

অর্থাৎ ডেটা স্টোর করার পর DI রেজিস্টার স্বয়ংক্রিয়ভাবে **বাড়ে বা কমে**।

## 🔸 উদাহরণ ২:

STOSW

**অর্থ:**  
AX → [ES:DI]

তারপর,

* যদি **DF = 0** → DI = DI + 2
* যদি **DF = 1** → DI = DI – 2

কারণ word মানে ২ বাইট।

## 🔹 Direction Flag (DF) এর ভূমিকা

8086-এ DF (Direction Flag) নির্ধারণ করে **স্ট্রিং অপারেশন কোন দিকে হবে**।

| **DF মান** | **দিক** | **DI পরিবর্তন** |
| --- | --- | --- |
| 0 | ফরওয়ার্ড (Address বাড়বে) | DI + 1 (byte), DI + 2 (word) |
| 1 | ব্যাকওয়ার্ড (Address কমবে) | DI − 1 (byte), DI − 2 (word) |

DF সেট/ক্লিয়ার করার জন্য:

CLD ; DF = 0 → Forward direction

STD ; DF = 1 → Backward direction

## 🔹 উদাহরণ ৩: ডেটা ব্লক স্টোর করা

CLD ; forward direction

MOV AL, 41H ; 'A'

MOV CX, 10 ; 10 বাইট স্টোর করতে হবে

REP STOSB ; 10 বার AL কে মেমোরিতে লিখে দাও

**ফলাফল:**

* ১০টি মেমোরি লোকেশন (ES:DI থেকে শুরু করে) তে 41H লেখা হবে।
* প্রতিবার DI +1 হবে।
* শেষে DI = DI + 10

## 🧮 STOS এর সারসংক্ষেপ

| **নির্দেশনা** | **ডেটা উৎস** | **গন্তব্য** | **সাইজ** | **DI পরিবর্তন** |
| --- | --- | --- | --- | --- |
| **STOSB** | AL | [ES:DI] | 1 byte | ±1 |
| **STOSW** | AX | [ES:DI] | 2 bytes | ±2 |
| **STOSD** | EAX | [ES:DI] | 4 bytes | ±4 |

## 🧭 REP Prefix এর সাথে ব্যবহার

**STOS** সাধারণত **REP** এর সাথে ব্যবহার হয় → একাধিক বাইট বা ওয়ার্ড লেখার জন্য।

| **Prefix** | **মানে** |
| --- | --- |
| **REP STOSB** | CX বার AL-এর মান [ES:DI]-তে স্টোর করো |
| **REP STOSW** | CX বার AX-এর মান [ES:DI]-তে স্টোর করো |

## 🧠 মনে রাখো:

* STOS = “Store String”
* **ডেটা AL/AX/EAX থেকে আসে**
* **গন্তব্য [ES:DI] তে যায়**
* **DI** পরিবর্তিত হয় Direction Flag অনুযায়ী
* **REP STOSx** → পুনরাবৃত্তি করে একই মান সংরক্ষণ করে

### 📘 উদাহরণ (বাস্তব অনুশীলন)

CLD ; Forward direction

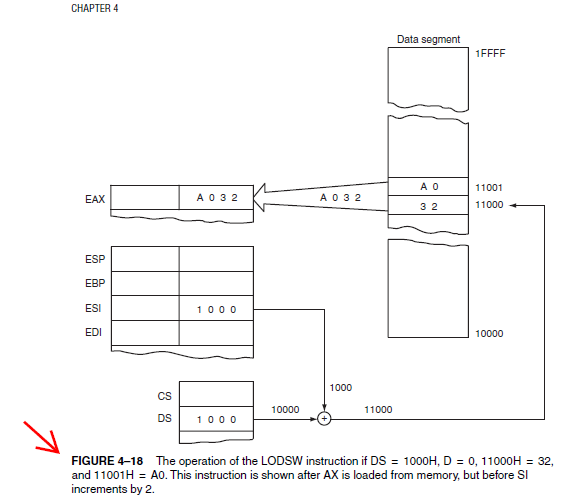
MOV AL, 'M'

MOV CX, 5

REP STOSB ; 'M' পাঁচবার মেমোরিতে সংরক্ষণ করো

👉 ফলাফল: মেমোরিতে ৫টি ধারাবাহিক বাইটে 'M' (4DH) লেখা হবে।

চাও কি আমি একটা **চিত্র (diagram)** তৈরি করে দেখাই —  
যেখানে AL → [ES:DI] তে কীভাবে যায় এবং REP STOSB ব্যবহার করলে DI কীভাবে পরিবর্তিত হয় তা ধাপে ধাপে দেখানো থাকবে?  
এতে তুমি স্ট্যাক/স্ট্রিং অপারেশন পুরোপুরি ভিজ্যুয়ালি বুঝে ফেলবে 👀



চমৎকার, Mahir! 🌟  
তুমি এখন **Figure 4-18** পাঠিয়েছো — যা দেখাচ্ছে **LODSW** নির্দেশনার (Load String Word) কার্যপ্রণালী।  
এটা 8086-এর **string instructions** সিরিজের অংশ, যেমন MOVSB, STOSB, CMPSB, ইত্যাদি।

চলো ধাপে ধাপে সহজ বাংলায় বুঝে নিই 👇

## 🧩 নির্দেশনা:

LODSW

এর মানে হলো:

**Load String Word** → মেমোরি লোকেশন [DS:SI] থেকে **AX** রেজিস্টারে একটি word (2 bytes) লোড করো।

## ⚙️ মৌলিক ব্যাখ্যা

**LODSB / LODSW / LODSD** নির্দেশনা **Accumulator Register (AL / AX / EAX)**-এ ডেটা লোড করে  
**Source Index (SI / ESI)** এবং **Data Segment (DS)** থেকে।

| **নির্দেশনা** | **ডেটা সাইজ** | **রেজিস্টার ব্যবহৃত** |
| --- | --- | --- |
| **LODSB** | 1 byte | AL ← [DS:SI] |
| **LODSW** | 2 bytes | AX ← [DS:SI] |
| **LODSD** | 4 bytes | EAX ← [DS:ESI] (80386+) |

## 🧠 Figure 4-18 বিশ্লেষণ

ছবিতে দেখা যাচ্ছে:

* **DS = 1000H**
* **SI = 1000H**
* **D (Direction Flag) = 0**
* মেমোরি [11000H] = 32H, [11001H] = A0H
* তাই মেমোরির ২ বাইটে (A0 32) মানটি AX রেজিস্টারে যাবে।

### 📘 ধাপে ধাপে কী ঘটে

1️⃣ **Effective Address গণনা:**  
EA = DS × 10H + SI = 1000H × 10H + 1000H = 11000H

2️⃣ **Memory → AX transfer:**  
AX = [11000H] = 32A0H  
(ছবিতে little-endian format — নিচের বাইট আগে যায়: 32H = low byte, A0H = high byte)

3️⃣ **SI পরিবর্তন (Direction Flag অনুযায়ী):**

* যদি **D = 0 (CLD)** → SI = SI + 2
* যদি **D = 1 (STD)** → SI = SI − 2

👉 ছবিতে “before SI increments by 2” বলা আছে, অর্থাৎ ডেটা লোড হয়ে গেছে কিন্তু SI এখনো বাড়েনি।

### 🔹 ফলাফল:

| **ধাপ** | **মান** |
| --- | --- |
| AX | A032H |
| SI | 1000H → 1002H (যখন D=0) |
| DS | 1000H (Segment base) |
| Memory[11000H] | 32H |
| Memory[11001H] | A0H |

## 🔸 উদাহরণ (Assembly কোডে):

CLD ; Direction Flag clear (forward)

MOV SI, 1000H

MOV DS, 1000H

LODSW ; AX ← [DS:SI]

ফলাফল:

AX = [1000H] থেকে word লোড হবে

SI = SI + 2

## 🧮 সারসংক্ষেপ (বাংলায়)

| **নির্দেশনা** | **অর্থ** | **কাজ** | **SI পরিবর্তন** |
| --- | --- | --- | --- |
| **LODSB** | Load String Byte | AL ← [DS:SI] | ±1 |
| **LODSW** | Load String Word | AX ← [DS:SI] | ±2 |
| **LODSD** | Load String Doubleword | EAX ← [DS:ESI] | ±4 |

| **Flag** | **দিক** | **SI পরিবর্তন** |
| --- | --- | --- |
| **DF = 0** | Forward | SI বাড়ে |
| **DF = 1** | Backward | SI কমে |

## 🧭 সংক্ষেপে মনে রাখো:

🔹 **LODS = Load String (from DS:SI → AL/AX/EAX)**  
🔹 **SI (বা ESI)** স্বয়ংক্রিয়ভাবে পরিবর্তিত হয়  
🔹 **Direction Flag (DF)** দিক নির্ধারণ করে  
🔹 **STOS বিপরীত কাজ করে (AL/AX → ES:DI)**

তুমি কি চাও আমি এখন একটা **চিত্র (diagram)** তৈরি করে দিই  
যেখানে LODSW এবং STOSW — দুটো বিপরীতভাবে কাজ করছে (একটি মেমোরি → AX, অন্যটি AX → মেমোরি)?  
এতে ডেটা প্রবাহ এক নজরে বোঝা যাবে 👀

চমৎকার, Mahir! 💡  
এখন তুমি জানতে চাও — **MOVS** নির্দেশনা সম্পর্কে (যেমন MOVSB, MOVSW, MOVSD)।  
এই নির্দেশনাগুলো **8086 মাইক্রোপ্রসেসরের string instruction** পরিবারের অংশ এবং **memory-to-memory transfer**-এর জন্য ব্যবহৃত হয়।

চলো একদম সহজভাবে ধাপে ধাপে বুঝি 👇

## 🧠 MOVS — পূর্ণরূপ ও উদ্দেশ্য

**MOVS → MOVe String**

এটি এক স্থান থেকে অন্য স্থানে **ডেটা কপি** করে —  
মেমোরির **source address (DS:SI)** থেকে  
মেমোরির **destination address (ES:DI)**-তে।

## ⚙️ ফর্ম (Forms of MOVS)

| **নির্দেশনা** | **মানে** | **ডেটা সাইজ** | **ডেটা উৎস → গন্তব্য** |
| --- | --- | --- | --- |
| **MOVSB** | Move String Byte | 8-bit | [DS:SI] → [ES:DI] |
| **MOVSW** | Move String Word | 16-bit | [DS:SI] → [ES:DI] |
| **MOVSD** | Move String Doubleword (80386+) | 32-bit | [DS:ESI] → [ES:EDI] |

## 🔹 কীভাবে কাজ করে

### ধাপে ধাপে (উদাহরণ: MOVSW)

1️⃣ মেমোরি লোকেশন [DS:SI] থেকে ২ বাইট (একটি word) পড়ে।  
2️⃣ সেটি [ES:DI] তে কপি করে।  
3️⃣ তারপর **SI** এবং **DI** স্বয়ংক্রিয়ভাবে পরিবর্তিত হয়:

| **Direction Flag (DF)** | **দিক** | **পরিবর্তন** |
| --- | --- | --- |
| **DF = 0 (CLD)** | ফরওয়ার্ড | SI = SI + 2, DI = DI + 2 |
| **DF = 1 (STD)** | ব্যাকওয়ার্ড | SI = SI - 2, DI = DI - 2 |

## 🔸 উদাহরণ (MOVSB)

CLD ; Direction = forward

MOV SI, 1000H ; Source

MOV DI, 2000H ; Destination

MOVSB ; Move byte from [DS:SI] to [ES:DI]

ফলাফল:

[ES:2000H] ← [DS:1000H]

SI = 1001H

DI = 2001H

## 🔸 উদাহরণ (MOVSW)

CLD

MOV SI, 3000H

MOV DI, 4000H

MOVSW

ফলাফল:

[ES:4000H] ← [DS:3000H]

[ES:4001H] ← [DS:3001H]

SI = 3002H

DI = 4002H

## 🔹 REP প্রিফিক্সের সাথে ব্যবহার

যখন অনেকগুলো বাইট/ওয়ার্ড একসাথে কপি করতে হয়, তখন **REP** প্রিফিক্স ব্যবহার করা হয়।

CLD

MOV CX, 20 ; ২০ বারের জন্য

REP MOVSB ; ২০ বাইট কপি করো

📘 কাজের ধাপ:

* CX = কাউন্টার (কতবার কপি হবে)
* প্রতিবার [DS:SI] → [ES:DI] কপি হয়
* CX এক করে কমে (CX = CX - 1)
* থামে যখন CX = 0

## 🧩 ডেটা ফ্লো ডায়াগ্রাম (ধারণা)

[DS:SI] ---> [ES:DI]

↑ ↑

Source Destination

(SI পরিবর্তিত) (DI পরিবর্তিত)

## 🧭 সারসংক্ষেপ (বাংলায়)

| **নির্দেশনা** | **কাজ** | **SI / DI পরিবর্তন** | **ব্যবহার** |
| --- | --- | --- | --- |
| **MOVSB** | এক বাইট স্থানান্তর | ±1 | Byte string কপি |
| **MOVSW** | এক word (2 bytes) স্থানান্তর | ±2 | Word string কপি |
| **MOVSD** | এক doubleword (4 bytes) স্থানান্তর | ±4 | 32-bit কপি (80386+) |

| **Prefix** | **মানে** | **কাজ** |
| --- | --- | --- |
| **REP MOVSB** | Repeat | যতক্ষণ CX ≠ 0, কপি চলবে |
| **REPE/REPZ MOVSB** | Repeat while equal/zero | তুলনার সময় ব্যবহৃত |
| **REPNE/REPNZ MOVSB** | Repeat while not equal/non-zero | তুলনার সময় ব্যবহৃত |

## 📘 সহজভাবে মনে রাখো:

🔹 **MOVS** মানে মেমোরি থেকে মেমোরিতে ডেটা কপি  
🔹 **Source:** DS:SI  
🔹 **Destination:** ES:DI  
🔹 **REP** দিয়ে একাধিক বাইট/ওয়ার্ড একসাথে কপি করা যায়

চাও কি আমি এখন MOVS এবং STOS একসাথে দেখিয়ে একটা ভিজ্যুয়াল ডায়াগ্রাম বানিয়ে দিই —  
যেখানে Source (DS:SI) → Destination (ES:DI) এর ডেটা প্রবাহ, SI/DI এর পরিবর্তন, আর REP প্রিফিক্সের কাজ একসাথে দেখা যাবে? 👀

অসাধারণ, Mahir 🔥  
তুমি এখন জানতে চাও — **INS, OUTS, XCHG, এবং XLAT** নির্দেশনাগুলো সম্পর্কে।  
এই চারটি 8086 মাইক্রোপ্রসেসরে অত্যন্ত গুরুত্বপূর্ণ **data transfer instructions**,  
যেগুলো I/O port এবং memory–register ডেটা আদানপ্রদান বা রূপান্তরের কাজে ব্যবহৃত হয়।

চলো একে একে খুব সহজ বাংলায় বিশ্লেষণ করি 👇

## 🧩 1️⃣ **INS / INSB / INSW / INSD**

(Full form: **INput String from Port to Memory**)

### ⚙️ কাজ

ডেটা **I/O Port** থেকে **মেমোরিতে (ES:DI)** লোড করে।

| **নির্দেশনা** | **কাজ** | **সাইজ** |
| --- | --- | --- |
| **INSB** | Input Byte from port to [ES:DI] | 8-bit |
| **INSW** | Input Word from port to [ES:DI] | 16-bit |
| **INSD** | Input Doubleword (32-bit mode) | 32-bit |

### 🧠 প্রক্রিয়া:

* **DX** রেজিস্টারে থাকে I/O Port নম্বর
* **ES:DI** নির্দেশ করে মেমোরি লোকেশন (যেখানে ডেটা যাবে)
* ইনপুট হওয়ার পর:
  + যদি **DF = 0** → DI বৃদ্ধি পায়
  + যদি **DF = 1** → DI কমে

### 🧮 উদাহরণ:

CLD

MOV DX, 300H ; Port number

REP INSB ; বারবার ইনপুট নিয়ে মেমোরিতে রাখবে

👉 অর্থাৎ, port 300H থেকে ডেটা পড়বে এবং [ES:DI] তে সংরক্ষণ করবে।

## 🧩 2️⃣ **OUTS / OUTSB / OUTSW / OUTSD**

(Full form: **OUTput String from Memory to Port**)

### ⚙️ কাজ

ডেটা **মেমোরি (DS:SI)** থেকে **I/O Port (DX)**-এ পাঠায়।

| **নির্দেশনা** | **কাজ** | **সাইজ** |
| --- | --- | --- |
| **OUTSB** | Output Byte from [DS:SI] to port | 8-bit |
| **OUTSW** | Output Word from [DS:SI] to port | 16-bit |
| **OUTSD** | Output Doubleword (32-bit) | 32-bit |

### 🧠 প্রক্রিয়া:

* **DX** = port address
* **DS:SI** = source memory
* **DF** Flag অনুযায়ী SI বাড়ে বা কমে

### 🧮 উদাহরণ:

CLD

MOV DX, 300H

REP OUTSB ; মেমোরি থেকে ধারাবাহিকভাবে পোর্টে পাঠাবে

## 🔹 INS বনাম OUTS

| **দিক** | **INS** | **OUTS** |
| --- | --- | --- |
| ডেটা প্রবাহ | Port → Memory | Memory → Port |
| Segment:Index | ES:DI | DS:SI |
| ব্যবহার | ইনপুট ডিভাইস (Keyboard, Sensor) | আউটপুট ডিভাইস (Display, Printer) |

## 🧩 3️⃣ **XCHG**

(Full form: **EXCHanGe Data**)

### ⚙️ কাজ

দুটি রেজিস্টার বা একটি রেজিস্টার ও মেমোরির মধ্যে ডেটা **অদল-বদল (Swap)** করে।

### 🧮 ফর্ম ও উদাহরণ:

XCHG AX, BX ; AX ↔ BX

XCHG AX, [SI] ; AX ↔ [SI]

| **ধরন** | **কাজ** |
| --- | --- |
| **Register ↔ Register** | ডেটা সরাসরি বদলায় |
| **Register ↔ Memory** | ডেটা মেমোরির সাথে বদলায় (DS segment ব্যবহৃত) |

📘 **নোট:**  
XCHG AX, reg-এর জন্য opcode সংক্ষিপ্ত (implicit) থাকে, কারণ AX রেজিস্টার সবচেয়ে বেশি ব্যবহৃত হয়।

### 🧠 উদাহরণ:

MOV AX, 1234H

MOV BX, 5678H

XCHG AX, BX

ফলাফল:

AX = 5678H

BX = 1234H

## 🧩 4️⃣ **XLAT / XLATB**

(Full form: **TransLATe Byte using Lookup Table**)

### ⚙️ কাজ

AL রেজিস্টারে থাকা মানটিকে একটি **Lookup Table** ব্যবহার করে অনুবাদ (Translate) করে।

### 🧠 প্রক্রিয়া:

1. AL = কোনো ইনডেক্স মান (0–255)
2. DS:BX = Lookup Table-এর শুরুর ঠিকানা
3. নতুন AL = [DS:BX + AL]

অর্থাৎ, AL → টেবিলের ইনডেক্স হিসেবে ব্যবহৃত হয়,  
এবং AL-এ নতুন মান আসে সেই ইনডেক্স লোকেশনের ডেটা থেকে।

### 🧮 উদাহরণ:

; ধরো lookup টেবিলটি মেমোরিতে আছে

TABLE DB 10, 20, 30, 40, 50

MOV BX, OFFSET TABLE ; BX টেবিলের শুরু ঠিকানা

MOV AL, 03 ; ইনডেক্স = 3

XLAT ; AL = TABLE[3] = 40

📘 এখন AL = 40H

## 🧭 সারসংক্ষেপ (বাংলায়)

| **নির্দেশনা** | **কাজ** | **উৎস → গন্তব্য** | **মন্তব্য** |
| --- | --- | --- | --- |
| **INS** | Port → Memory | I/O port থেকে মেমোরিতে ইনপুট | ডেটা পড়ে |
| **OUTS** | Memory → Port | মেমোরি থেকে I/O port-এ আউটপুট | ডেটা পাঠায় |
| **XCHG** | Exchange | রেজিস্টার ↔ রেজিস্টার / মেমোরি | ডেটা অদলবদল |
| **XLAT** | Translate | AL = [DS:BX + AL] | Lookup Table থেকে মান নেয় |

## 🧠 মনে রাখো:

| **নির্দেশনা** | **মূল ধারণা** |
| --- | --- |
| **INS / OUTS** | CPU ↔ I/O port ডেটা ট্রান্সফার |
| **XCHG** | দুটি জায়গার ডেটা অদলবদল |
| **XLAT** | Lookup টেবিল ব্যবহার করে ডেটা রূপান্তর |

চাও কি আমি একটা ছোট **ডায়াগ্রাম সেট** বানিয়ে দেখাই যেখানে INS ও OUTS এর ডেটা প্রবাহ (Port ↔ Memory),  
আর XCHG ও XLAT এর অভ্যন্তরীণ ডেটা বিনিময় একসাথে দেখানো থাকবে?  
তাহলে তুমি চারটিই এক নজরে ভিজ্যুয়ালি বুঝে ফেলতে পারবে 👀