

## MOV Instruction

- MOV হলো সবচেয়ে সাধারণ এবং বহুল ব্যবহৃত ডাটা ট্রান্সফার নির্দেশনা।
- এর কাজ হলো **source → destination** এ ডাটা কপি করা।
-  মনে রাখতে হবে, ডাটা move হয় না, শুধু copy হয়। Source পরিবর্তিত হয় না, শুধু Destination পরিবর্তিত হয়।

### লিখনরীতি (Syntax):

MOV Destination, Source

- ডান পাশে থাকে **Source**
- বাম পাশে থাকে **Destination**
- সবসময় **comma ( , )** দিয়ে আলাদা করা হয়।

 তাই MOV AX, BX মানে হচ্ছে BX এর মান কপি হবে AX এ।

 BX আগের মতো থাকবে, কিন্তু AX এর মান পরিবর্তিত হবে।

## গুরুত্বপূর্ণ নিয়ম:

1. **Memory-to-Memory transfer** সরাসরি হয় না।
  - কেবল MOVS (string instruction) দিয়ে সম্ভব।
2. **Flags Register (RFLAGS)** সাধারণত MOV এ পরিবর্তিত হয় না।

## Data Addressing Modes (ডাটা অ্যাড্রেসিং মোড)

MOV এর মাধ্যমে ডাটা ট্রান্সফারের সময় বিভিন্নভাবে Source/Destination নির্ধারণ করা যায়। এগুলো হলো —

### 1. Register Addressing

- Source এবং Destination দুটোই **Register**।
- **উদাহরণ:**
- MOV CX, DX ; DX → CX (word transfer)
- MOV ECX, EDX ; 32-bit transfer
- MOV RDX, RCX ; 64-bit transfer (Pentium 4+, 64-bit mode)

## 2. Immediate Addressing

- Source হলো **immediate data (constant value)**।
  - Destination হতে পারে Register বা Memory।
  - উদাহরণ:
    - MOV AL, 22H ; AL = 22h (byte data)
    - MOV BX, 1234H ; BX = 1234h (word data)
    - MOV EAX, 1000H ; EAX = 1000h (doubleword data)
    - MOV RAX, 100000H ; RAX = 100000h (quadword data)
- 

## সংক্ষেপে মনে রাখো:

- MOV AX, BX** → BX থেকে AX-এ কপি হবে।
  - MOV AL, 22H** → AL-এ 22H সেট হবে।
  - Source পরিবর্তন হয় না, Destination সবসময় পরিবর্তিত হয়।
- 

## 3. Direct Addressing (ডাইরেক্ট অ্যাড্রেসিং)

- ডাটা **মেমরি <-> রেজিস্টার** এর মধ্যে সরাসরি কপি করা হয়।
  - Memory-to-Memory সরাসরি ট্রান্সফার সম্ভব নয়, কেবল MOVS দিয়ে।
  - উদাহরণ:
    - MOV CX, LIST ; LIST মেমরি লোকেশন থেকে CX-এ কপি
    - MOV ESI, LIST ; 80386+: 32-bit (doubleword) ট্রান্সফার
  - 64-bit mode এ, **full 64-bit linear address** ব্যবহার করা হয়।
- 

## 4. Register Indirect Addressing (রেজিস্টার ইন্ডাইরেক্ট অ্যাড্রেসিং)

- মেমরি লোকেশন নির্ধারণ হয় index/base register-এর মাধ্যমে।**
- Index/Base register হতে পারে: BP, BX, DI, SI (8086–80286)
- উদাহরণ:
  - MOV AX, [BX] ; DS:BX-এর নির্দেশিত মেমরি থেকে AX-এ কপি
  - MOV AL, [ECX] ; 80386+: 32-bit register ব্যবহার
- 64-bit mode:
  - 32-bit compatible mode: max 4GB অ্যাড্রেস

- Full 64-bit mode: 64-bit address বা register-এর মাধ্যমে যেকোনো অ্যাড্রেস অ্যাক্সেস করা যায়।

## Register Indirect Addressing কি?

- এখানে ডাটা সরাসরি রেজিস্টারে নেই, বরং মেমরির একটা ঠিকানা রেজিস্টারের মধ্যে রাখা আছে।
- অর্থাৎ, রেজিস্টার শুধু মেমরির ঠিকানা দেখাচ্ছে, আর আমরা সেই মেমরির ডাটা রেজিস্টারে কপি করি।
- এটি মূলত pointer এর মতো কাজ করে।

---

## কীভাবে কাজ করে

ধরা যাক:

$BX = 1000H$  ; BX রেজিস্টারে মেমরির ঠিকানা আছে  
 $Memory[1000H] = 55H$  ; মেমরিতে 55H মান আছে

### Instruction:

MOV AL, [BX]

- $[BX]$  মানে হলো BX রেজিস্টারের ঠিকানা যেখানে ডাটা আছে।
- অর্থাৎ:  $Memory[1000H] \rightarrow AL$
- শেষে  $AL = 55H$
- BX আগের মতোই থাকবে, শুধু AL-এ মান কপি হবে।

---

## আরেকটা উদাহরণ (Word Transfer)

$BX = 2000H$   
 $Memory[2000H] = 12H$   
 $Memory[2001H] = 34H$

### Instruction:

MOV AX, [BX]

- $AX = Memory[2000H, 2001H] \rightarrow 3412H$  (word-sized transfer)
- BX পরিবর্তন হয় না

## 64-bit বা 80386+ প্রসেসরে

- এখানে একই পদ্ধতি, শুধু 32-bit বা 64-bit রেজিস্টার ব্যবহার করা যায়।
- উদাহরণ:

ECX = 00400000H

MOV AL, [ECX] ; ECX ঠিকানা নির্দেশ করছে, AL-এ ডাটা কপি

---

### সংক্ষেপে:

- Register Indirect Addressing = **রেজিস্টারে ঠিকানা, মেমরি থেকে ডাটা নাও**
- রেজিস্টার = pointer, মেমরি = ডাটা
- উদাহরণ: MOV AL, [BX] → BX হলো pointer, AL-এ মেমরির মান চলে আসে

## 5. Base-Plus-Index Addressing (বেস + ইনডেক্স অ্যাড্রেসিং)

- মেমরি অ্যাড্রেস তৈরি হয় **base register + index register** দিয়ে।
- Base = BP বা BX, Index = DI বা SI
- উদাহরণ (8086–80286):
  - MOV [BX+DI], CL ; CL এর মান DS:BX+DI মেমরিতে কপি
  - 80386+ এ যেকোনো দুইটি রেজিস্টার ব্যবহার করা যায়:
  - MOV [EAX+EBX], CL ; CL এর মান DS:EAX+EBX মেমরিতে কপি

## 6. Register Relative Addressing (রেজিস্টার রিলেটিভ অ্যাড্রেসিং)

- মেমরি অ্যাড্রেস = **base/index register + displacement (offset)**
- উদাহরণ:
  - MOV AX, [BX+4] ; DS:BX+4 থেকে AX-এ কপি
  - MOV AX, ARRAY[BX] ; ARRAY+BX লোকেশন থেকে AX-এ কপি
- 80386+ এ যেকোনো 32-bit রেজিস্টার (ESP বাদে) ব্যবহার করা যায়:
  - MOV AX, [ECX+4]
  - MOV AX, ARRAY[EBX]

## সারসংক্ষেপ (Tabular Form)

Addressing Mode	Source/Destination	উদাহরণ
Direct	Memory $\leftrightarrow$ Register	MOV CX, LIST
Register Indirect	Register $\leftrightarrow$ Memory (via reg)	MOV AX, [BX]
Base + Index	Register $\leftrightarrow$ Memory (base+index)	MOV [BX+DI], CL
Register Relative	Register $\leftrightarrow$ Memory (reg+offset)	MOV AX, [BX+4]

## 7. Base-Relative Plus Index Addressing

- মেমরি অ্যাড্রেস তৈরি হয়:

Memory Address = Base Register + Index Register + Displacement (offset)

- উদাহরণ:

```

MOV AX, ARRAY[BX+DI]      ; AX = DS:ARRAY + BX + DI
MOV AX, [BX+DI+4]          ; AX = DS:BX + DI + 4
MOV EAX, ARRAY[EBX+ECX]    ; 80386+: EAX = DS:ARRAY + EBX + ECX

```

### ব্যাখ্যা:

- Base register (BX, EBP, ইত্যাদি)
- Index register (SI, DI, ECX ইত্যাদি)
- Displacement = সংখ্যা/offset (যেমন 4, ARRAY)
- এগুলো মিলিয়ে ফাইনাল মেমরি অ্যাড্রেস তৈরি হয়।

## 8. Scaled-Index Addressing (80386–Pentium 4)

- এখনে index register এর মানকে  $2\times$ ,  $4\times$ , বা  $8\times$  scale করা হয়।
- উদ্দেশ্য: বড় ডাটা এরে (array) তে দ্রুত অ্যাক্সেস।
- উদাহরণ:

```

MOV EDX, [EAX + 4*EBX]    ; EDX = DS:EAX + 4*EBX
MOV AL, [EBX + ECX]        ; scale factor 1 implicit
MOV AL, [EBX + 1*ECX]      ; scale factor 1 explicit

```

### ব্যাখ্যা:

- Scaling factor 1, 2, 4, 8 হতে পারে
  - মূলত word/doubleword/quadword array এর জন্য ব্যবহৃত হয়
- 

## 9. RIP-Relative Addressing (64-bit only, Pentium 4/Core2)

- এই mode শুধুমাত্র 64-bit mode এ ব্যবহার হয়।
- Address = **RIP + 32-bit displacement**
- উদাহরণ:

RIP = 1000000000H  
Displacement = 300H  
Memory accessed = 1000000300H

### ব্যাখ্যা:

- Instruction pointer (RIP) থেকে relative offset যোগ করে ডাটা অ্যাস্ট্রেস করা হয়
  - Displacement signed হতে পারে, তাই  $\pm$  range-এর মধ্যে ডাটা অ্যাস্ট্রেস করা যায়
  - সুবিধা: **Code-relative addressing**, কোড বা ডাটাকে position-independent করা যায়
- 

### চিত্রের মত ধারণা (conceptually)

Addressing Mode	Memory Address Calculation
Base + Index	Base + Index
Base + Index + Displacement	Base + Index + Offset
Scaled-Index	Base + (Index $\times$ Scale)
RIP-Relative	RIP + Displacement

---

### 💡 মনে রাখার মূল কথা:

- Base+Index → সাধারণ array বা structure access
- Scaled-Index → array element access ফর্ম (word/double/quadword)
- RIP-relative → code বা data position independent (64-bit)

## Register Addressing (রেজিস্টার অ্যাড্রেসিং)

- এটি সবচেয়ে সাধারণ ও সহজ ডাটা অ্যাড্রেসিং মোড।
- এখানে source এবং destination দুটোই register হয়।
- ডাটা কপি করা হয় একটি register থেকে অন্য register-এ।
- উদাহরণ:

```
MOV AX, BX ; BX → AX  
MOV AL, CL ; CL → AL  
MOV EDX, ECX ; 32-bit register  
MOV RAX, RBX ; 64-bit register (Pentium 4+, 64-bit mode)
```

---

### Available Registers

#### 1. 8-bit registers:

AH, AL, BH, BL, CH, CL, DH, DL

#### 2. 16-bit registers:

AX, BX, CX, DX, SP, BP, SI, DI

#### 3. 32-bit registers (80386+):

EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI

#### 4. 64-bit registers (Pentium 4+, 64-bit mode):

RAX, RBX, RCX, RDX, RSP, RBP, RDI, RSI, R8-R15

#### 5. Segment registers (for MOV, PUSH, POP):

CS, ES, DS, SS, FS, GS

---

### Important Rules

#### 1. Register sizes must match:

- 8-bit → 8-bit
- 16-bit → 16-bit
- 32-bit → 32-bit
- 64-bit → 64-bit
- **উদাহরণ:** `MOV AX, AL` ভুল, কারণ AX = 16-bit, AL = 8-bit(Exam type)

## 2. MOV instruction কোনো flag পরিবর্তন করে না

- Flag bits শুধুমাত্র arithmetic বা logic instructions পরিবর্তন করে
  - 3. Exceptions: কিছু instructions আছে যেমন SHL DX, CL, এগুলো size rule অনুসারে ভিন্ন আচরণ করে
- 

## সংক্ষেপে

### Register Size Example MOV Instruction

8-bit	MOV AL, CL
16-bit	MOV AX, BX
32-bit	MOV EDX, ECX
64-bit	MOV RAX, RBX

---

### 💡 মনে রাখার মূল কথা:

- Register Addressing = register → register ডাটা কপি
  - Register size মিলে যাবে এমন instructions ব্যবহার করতে হবে
  - MOV কোনো flag পরিবর্তন করে না
- 

## Segment-to-Segment MOV

- সাধারণ MOV instruction দিয়ে segment register থেকে segment register এ ডাটা কপি করা যায় না।
- বিশেষ করে CS (Code Segment) register পরিবর্তন করা কখনোই allowed নয়।

## কারণ:

- Instruction এর ঠিকানা (পরবর্তী instruction কোথায় যাবে) নির্ধারিত হয় IP/EIP + CS দিয়ে।
- যদি শুধু CS পরিবর্তন করা যায়, তাহলে পরবর্তী instruction এর ঠিকানা unpredictable হয়ে যাবে।
- তাই, MOV CS, ... instruction নিষিদ্ধ।

## Normal Register-to-Register MOV

- উদাহরণ:

MOV BX, CX

### কী হয়:

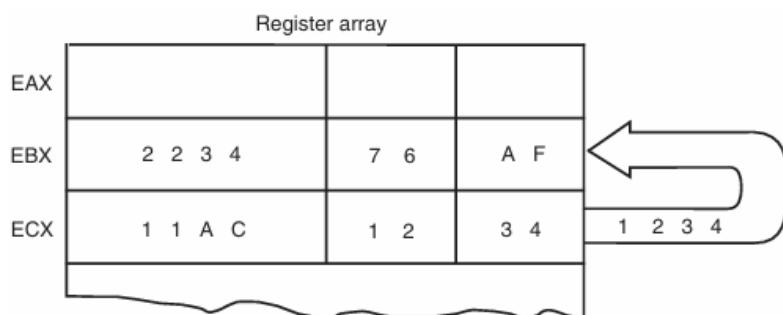
- Source register (CX) অপরিবর্তিত থাকে।
- Destination register (BX) পরিবর্তিত হয়।
- Destination এর পুরানো মান (যেমন 76AFH) মুছে যায়।
- Source register (CX) এর মান অপরিবর্তিত থাকে।

#### উদাহরণ:

Register	Value Before	Instruction	Value After
BX	76AFH	MOV BX, CX	1234H
CX	1234H	MOV BX, CX	1234H

- মনে রাখো: MOV instruction flag bits পরিবর্তন করে না।

**FIGURE 3–3** The effect of executing the MOV BX, CX instruction at the point just before the BX register changes. Note that only the rightmost 16 bits of register EBX change.



## 64-bit বা 32-bit প্রসেসরে

- উদাহরণ: MOV BX, CX করলে 16-bit অংশ পরিবর্তিত হয়।
- এর ফলে EBX এর বাকি 16-bit (leftmost) অপরিবর্তিত থাকে।
- অর্থাৎ, EBX এর পুরানো upper bits (32-bit প্রসেসরে) মুছে যায় না।

### 💡 সারসংক্ষেপ:

- Normal MOV: register → register/memory, source অপরিবর্তিত, destination পরিবর্তিত।
- Segment-to-segment MOV: **not allowed**, বিশেষ করে CS।
- MOV instruction flags পরিবর্তন করে না।

## ◆ ১. Immediate Addressing কী?

👉 Immediate Addressing Mode মানে হলো —  
ইনস্ট্রুকশনের (instruction) মধ্যেই ডেটা লেখা থাকে।  
অর্থাৎ, ডেটাটা সরাসরি instruction-এর ভেতরেই দেওয়া হয়, অন্য কোথাও (register বা memory)  
থেকে আসে না।

- অর্থাৎ, এই “immediate” শব্দটা বোঝায় — ডেটা “তৎক্ষণাৎ” instruction-এর পরেই আছে।

### 🧠 উদাহরণ:

MOV AX, 1234H

- 👉 এখানে 1234H হলো immediate data
- 👉 এই মানটা সরাসরি AX রেজিস্টারে কপি হবে
- 👉 মেমরিতে কোথাও থেকে মানটা আনা হচ্ছে না

- ◆ কাজ:

AX ← 1234H

## ◆ ২. Immediate data কেমন ধরনের হতে পারে?

Immediate data বিভিন্ন ফরম্যাটে দেওয়া যায় — যেমনঃ

টাইপ	উদাহরণ	ব্যাখ্যা
Hexadecimal	MOV AX, 1234H	হেক্স সংখ্যা
Decimal	MOV AL, 44	44 ডেসিমাল (২C হেক্স)

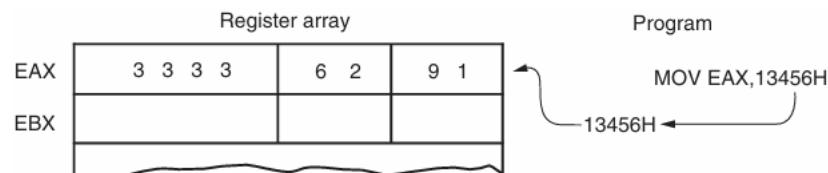
টাইপ	উদাহরণ	ব্যাখ্যা
Binary	MOV CL, 11001110B	বাইনারি মান
ASCII character	MOV AL, 'A'	অক্ষর A (81H)
Word/Doubleword	MOV EBX, 12340000H	৩২-বিট মান
64-bit (in 64-bit mode)	MOV RAX, 123456780A311200H	৬৪-বিট মান

## ◆ ৭. Immediate Data = Constant Data

👉 Immediate data সবসময় constant (স্থির মান)।  
অর্থাৎ, প্রোগ্রাম চলার সময় এটা পরিবর্তন হয় না।

অন্যদিকে, register বা memory থেকে আনা ডেটা variable হতে পারে।

**FIGURE 3-4** The operation of the MOV EAX,3456H instruction. This instruction copies the immediate data (13456H) into EAX.



## ◆ ৮. Machine Code এ Immediate Data কেমন থাকে?

Assembler ইনস্ট্রাকশনটাকে মেশিন কোডে রূপান্তর করে।

উদাহরণ:

`MOV AX, 0`

Assembler যখন এটা assemble করে, তখন memory-তে নিচের মতো সেভ হয়:

Offset Address	Machine Code	Assembly Instruction	ব্যাখ্যা
0100	B8 00 00	MOV AX, 0	B8 = opcode, 0000 = immediate data

👉 এখানে B8 হলো `MOV AX, immediate` এর opcode

👉 এরপরের দুই বাইট 0000 হলো immediate value

## ◆ ৫. ASCII immediate data

MOV AL, 'A'

👉 এখানে 'A' মানে ASCII কোড 41H

👉 AL = 41H হয়ে যাবে।

MOV AX, 'AB'

👉 এখানে 'AB' দুইটা ক্যারেক্টোর মানে দুই বাইট —

'A' (41H) ও 'B' (42H)

কিন্তু word আকারে স্টোর হবে BAH (লিটল-এন্ডিয়ান কারণে উল্টা ক্রমে)

## ◆ ৬. MASM/TASM ইত্যাদিতে Immediate Data কীভাবে লেখা হয়?

- # চিহ্ন কিছু assembler (পুরনো HP assembler) ব্যবহার করতো:
- MOV AX, #3456H

কিন্তু MASM/TASM/Intel ASM এগুলোতে # লাগে না।

- তাই তুমি লিখবে:
- MOV AX, 3456H

## ◆ ৭. Example 3-2: Full Program Explained

```
.MODEL TINY ; এক সেগমেন্টে পুরো প্রোগ্রাম
.CODE ; কোড সেকশন শুরু
.STARTUP ; প্রোগ্রামের শুরু নির্দেশ করে

MOV AX, 0 ; AX = 0000H
MOV BX, 0 ; BX = 0000H
MOV CX, 0 ; CX = 0000H
MOV SI, AX ; SI = AX (অর্থাৎ 0000H)
MOV DI, AX ; DI = AX
MOV BP, AX ; BP = AX

.EXIT ; প্রোগ্রাম শেষ করে DOS এ ফিরে যায়
END ; প্রোগ্রামের শেষ নির্দেশ
```

● ব্যাখ্যা:

- এখানে `MOV AX, 0, MOV BX, 0, MOV CX, 0` → **Immediate addressing**  
(কারণ মান 0 সরাসরি ইনস্ট্রুকশনে লেখা)
  - আর `MOV SI, AX, MOV DI, AX, MOV BP, AX` → **Register addressing**  
(কারণ মান অন্য রেজিস্টার থেকে কপি হচ্ছে)
- 

## ◆ ৮. Assembly লাইন গঠন (Example 3-3 অনুযায়ী)

প্রতিটা লাইনের ৪টা অংশ থাকে:

ফিল্ড	কাজ
Label	মেমরির symbolic নাম
Opcode	ইনস্ট্রুকশন নাম
Operand	কার ওপর কাজ হবে
Comment	মন্তব্য, ; দিয়ে শুরু

উদাহরণ:

```
DATA1    DB 23H      ; DATA1 নামে 23H সংরক্ষণ
DATA2    DW 1000H     ; DATA2 নামে 1000H (word)
START:   MOV AL, BL    ; BL → AL এ কপি
          MOV BH, AL    ; AL → BH এ কপি
          MOV CX, 200     ; 200 → CX এ কপি (immediate)
```

---

## ◆ সারসংক্ষেপ (সহজভাবে মনে রাখো)

বিষয়	ব্যাখ্যা
Immediate Addressing	ডেটা ইনস্ট্রুকশনের মধ্যেই থাকে
Data type	8, 16, 32, বা 64 বিট হতে পারে
Example	<code>MOV AX, 1234H</code>
Constant or Variable	Constant (স্থির মান)
Binary	শেষে B
Hexadecimal	শেষে H (যদি অক্ষর দিয়ে শুরু হয় তবে 0 দিয়ে শুরু করতে হবে যেমন 0F2H)
ASCII	'A' এর মতো
ব্যবহার	Register বা memory তে মান সেট করার সময়

## ✳️ ১. Direct Data Addressing কী?

👉 নামের মতোই — **Direct Addressing** মানে হলো

ডেটা “মেমরির নির্দিষ্ট জায়গায় (address এ)” আছে, আর আমরা সেই জায়গা থেকে সরাসরি ডেটা নিই বা সেখানে লিখি।

- এখানে ডেটা **মেমরিতে থাকে**, immediate mode-এর মতো ইনস্ট্রাকশনের ভেতরে থাকে না।

### ◆ উদাহরণ:

`MOV AL, DATA1`

এখানে —

- DATA1 হচ্ছে একটি **মেমরি লোকেশন (variable)**
- AL রেজিস্টারে ওই মেমরির ডেটা কপি হবে

অর্থাৎ:

`AL ← [DATA1]`

- ◆ এখানে `[DATA1]` মানে → মেমরিতে DATA1 নামের জায়গায় যা আছে, সেটার মান

## 🧠 Immediate vs Direct Addressing পার্থক্য:

বিষয়	Immediate	Direct
ডেটা কোথায় থাকে	ইনস্ট্রাকশনের ভেতরে	মেমরিতে
উদাহরণ	<code>MOV AX, 1234H</code>	<code>MOV AX, DATA1</code>
মান পরিবর্তন হয়	না (constant)	পারে (variable)

বিষয়	Immediate	Direct
কাজ হয়	Immediate data রেজিস্টারে কপি হয়	Memory থেকে রেজিস্টারে (বা উল্টো) কপি হয়

---

## ✳️ ২. Direct Addressing এর ধরন

Direct data addressing এর দুইটা ফর্ম আছে ↪

- ① Direct Addressing
  - ② Displacement Addressing
- 

### ◆ (1) Direct Addressing

এটা শুধু নির্দিষ্ট কিছু রেজিস্টারের জন্য ব্যবহৃত হয়:

→ AL, AX, বা EAX (অর্থাৎ 8, 16, বা 32-bit accumulator)

#### ◆ উদাহরণ:

MOV AL, DATA1

অর্থাৎ —

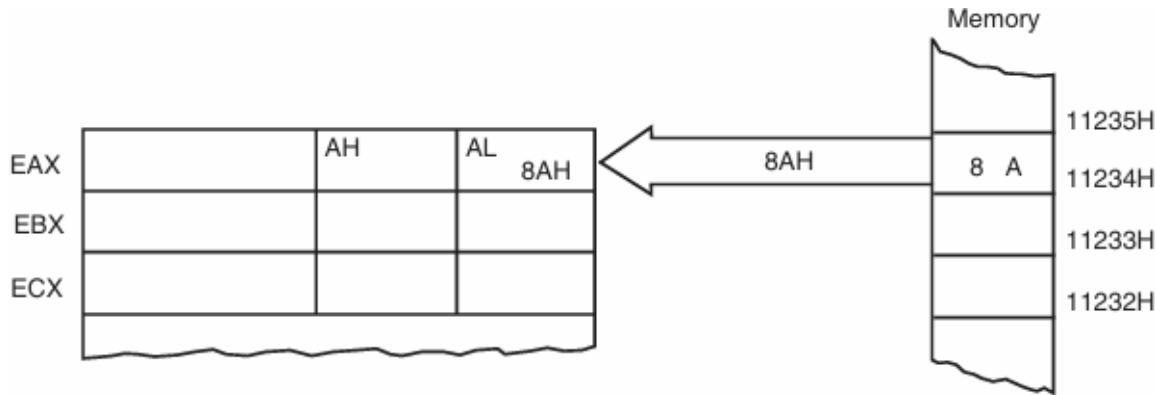
- AL = DATA segment-এর মধ্যে থাকা DATA1 মেমরি লোকেশনের মান

Assembler যখন এটা কনভার্ট করে, তখন এটা প্রায় সবসময় 3 বাইটের instruction হয়।

---

### ◆ কেমনভাবে কাজ করে (Figure 3–5 অনুযায়ী ধারণা):

ধরো —



**FIGURE 3–5** The operation of the `MOV AL,[1234H]` instruction when DS = 1000H.

DATA segment address = 1000H  
DATA1 offset = 1234H

👉 Effective Address =  $10000H + 1234H = 11234H$

সুতরাং, `MOV AL, DATA1` মানে:

$AL \leftarrow [11234H]$

অর্থাৎ 11234H মেমরি ঠিকানায় যে byte আছে, সেটা AL এ চলে আসবে।

#### ◆ (2) Displacement Addressing

এটা Direct Addressing এরই একটা উন্নত ভাসন।

এখনে যে কোনো রেজিস্টারেই মেমরি ডেটা নেওয়া যায় — শুধু AL/AX না।

এজন্য instruction টা 8 বাইট বা তার বেশি হয়।

#### ◆ উদাহরণ:

`MOV CL, DS:[1234H]`

অর্থাৎ DS সেগমেন্টের 1234H অফসেটে থাকা ডেটা → CL এ চলে যাবে।

📦 Assembler এটা মেশিন কোডে কনভার্ট করে এভাবে:

Offset	Machine Code	Instruction
0000	A0 1234 R	MOV AL, DS:[1234H]
0003	BA 0E 1234 R	MOV CL, DS:[1234H]

- ◆ পার্থক্য দেখো —

- AL এর জন্য ইনস্ট্রাকশন 3 বাইট
- CL এর জন্য ইনস্ট্রাকশন 4 বাইট

## ❖ ৬. সারসংক্ষেপ (সহজভাবে মনে রাখো)

বিষয়	ব্যাখ্যা
Direct Addressing	Memory-এর নির্দিষ্ট address থেকে ডেটা নেওয়া বা লেখা
Displacement Addressing	একই কাজ, তবে সব register-এর জন্য প্রযোজ্য
Immediate vs Direct	Immediate data ইনস্ট্রাকশনের ভেতরে, Direct data মেমরিতে
Default Segment	সাধারণত DS (Data Segment)
Instruction Size	৩–৭ bytes পর্যন্ত হতে পারে
উদাহরণ	MOV AL, DATA1 বা MOV BX, DATA4

### ◆ ১. Register Indirect Addressing কী?

👉 এটা এক ধরনের addressing mode, যেখানে data সরাসরি register-এ না থেকে memory তে থাকে,

আর সেই memory location-এর address (offset address) রাখা হয় একটি register-এ।

তারপর সেই register-কে ব্যবহার করে সেই memory location থেকে data নেওয়া বা সেখানে data রাখা হয়।

### ◆ ২. কোন কোন register ব্যবহার করা যায়?

☑ 16-bit mode (8086 এর জন্য):

👉 BX, BP, SI, DI

32-bit mode (80386 – Pentium):

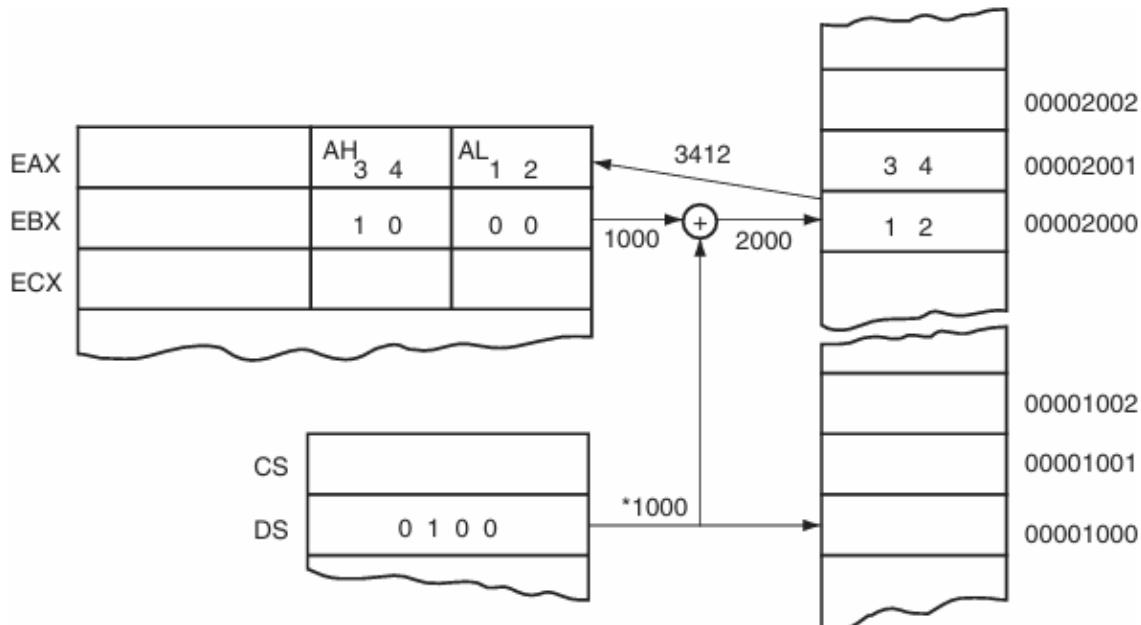
👉 EBX, EBP, ESI, EDI, (EAX, ECX, EDX ও ব্যবহার করা যায়)

64-bit mode (Core2, Pentium 4):

👉 যেকোনো 64-bit register (যেমন RAX, RBX, RCX, RDX ইত্যাদি)

এখানে segment register দ্রব্যকার হয় না, কারণ register-এ সরাসরি full address থাকে।

### ◆ ৩. কাজ করার পদ্ধতি (Basic Example)



\*After DS is appended with a 0.

**FIGURE 3–6** The operation of the `MOV AX,[BX]` instruction when  $BX = 1000H$  and  $DS = 0100H$ . Note that this instruction is shown after the contents of memory are transferred to AX.

ধরা যাক:

DS = 0100H  
BX = 1000H

এখন instruction:

`MOV AX, [BX]`

মানে হচ্ছে → BX register যেই address ধরে রেখেছে (1000H),  
সেই address এর memory location থেকে word ডাটা নিয়ে AX রেজিস্টারে রাখো।

❖ Memory Address হিসাব হবে:

$$\begin{aligned}\text{Physical Address} &= (\text{DS} \times 10H) + \text{BX} \\ &= (0100H \times 10H) + 1000H \\ &= 2000H\end{aligned}$$

তাহলে

- Memory[2000H] → ঘাবে AL-এ
- Memory[2001H] → ঘাবে AH-এ

অর্থাৎ, AX = [2001H:2000H] এর ডাটা।

---

◆ ৮. Segment register ব্যবহার

Register	Default Segment
BX, SI, DI	Data Segment (DS)
BP	Stack Segment (SS)

তাই:

- যদি [BX], [SI], [DI] ব্যবহার করো → ডাটা আসবে DS segment থেকে।
  - যদি [BP] ব্যবহার করো → ডাটা আসবে SS segment থেকে।
- 

◆ ৫. কিছু উদাহরণ

Instruction	Operation	Size
MOV CX, [BX]	DS segment-এর BX address থেকে 16-bit data নিয়ে CX-এ রাখে	16-bit
MOV [BP], DL	DL-এর মান SS segment-এর BP address-এ রাখে	8-bit
MOV [DI], BH	BH-এর মান DS segment-এর DI address-এ রাখে	8-bit
MOV AL, [EDX]	DS segment-এর EDX address থেকে byte data নিয়ে AL-এ রাখে	32-bit
MOV ECX, [EBX]	DS segment-এর EBX address থেকে 32-bit data নিয়ে ECX-এ রাখে	32-bit
MOV RAX, [RDX]	RDX address থেকে 64-bit data নিয়ে RAX-এ রাখে	64-bit

---

## ◆ ৬. BYTE PTR / WORD PTR / DWORD PTR

কখনও assembler ব্যবহার পারে না data-এর size কত হবে।

যেমন:

```
MOV [DI], 10H
```

এখানে [DI] তে 10H পাঠানো হচ্ছে, কিন্তু এটা byte, word, না doubleword?

তখন আমরা size specify করি 

```
MOV BYTE PTR [DI], 10H
```

মানে [DI] address-এ 1 byte data store হবে।

আর

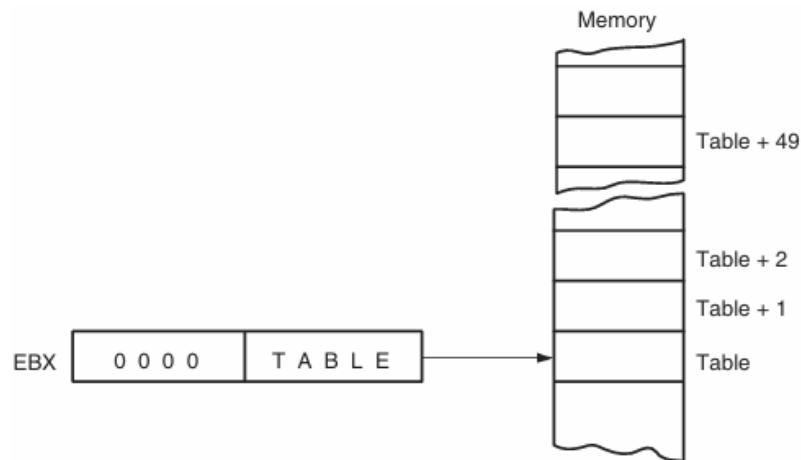
```
MOV DWORD PTR [DI], 10H
```

মানে 4 byte (doubleword) store হবে।

## ◆ ৭. বাস্তব উদাহরণ (Example 3-7 ব্যাখ্যা)

এই প্রোগ্রামে আমরা ৫০টা clock value sequentially একটা table-এ save করব।

**FIGURE 3-7** An array (TABLE) containing 50 bytes that are indirectly addressed through register BX.



### ◆ Data Segment:

DATAS DW 50 DUP (?)

👉 মানে DATAS নামে ৫০টা word-এর জায়গা বানানো হলো।

## ◆ সারসংক্ষেপ (এক নজরে)

বিষয়	ব্যাখ্যা
Address কোথায় থাকে	Register-এ (যেমন BX, SI, DI, BP)
Data কোথায় থাকে	Memory তে
Symbol [ ]	Indirect addressing বোঝায়
Default segment	DS (BX, SI, DI এর জন্য), SS (BP এর জন্য)
Size নির্ধারণ	BYTE PTR / WORD PTR / DWORD PTR ইত্যাদি দিয়ে
ব্যবহার	Table, Array, বা dynamic memory access-এর ক্ষেত্রে

## ◆ ১. Base-Plus-Index Addressing কী?

👉 এটা indirect addressing mode-এর একটা ধরন,  
যেখানে **দুটি register** (একটা base + একটা index) মিলে কোনো **memory location** নির্ধারণ  
করে।

অর্থাৎ,

👉 **Effective Address = Base Register + Index Register + (Segment × 10H)**

এখানে base register সাধারণত array বা data block-এর শুরু নির্দেশ করে,  
আর index register নির্দেশ করে সেই array-এর ভিতরে কোনো নির্দিষ্ট element-এর অবস্থান।

## ◆ ২. কোন কোন register ব্যবহার করা যায়?

🧠 8086–80286 এ (16-bit mode):

- Base register → BX বা BP
- Index register → SI বা DI

তাই সম্ভাব্য combination:

[BX + SI]

[BX + DI]  
[BP + SI]  
[BP + DI]

### 🧠 80386 এবং তার পর (32-bit mode):

👉 যেকোনো দুইটা 32-bit register ব্যবহার করা যায় (যেমন: EAX, EBX, ECX, EDX, ESI, EDI, EBP)  
কিন্তু ESP (stack pointer) ব্যবহার করা যায় না।

উদাহরণ:

MOV DL, [EAX + EBX]

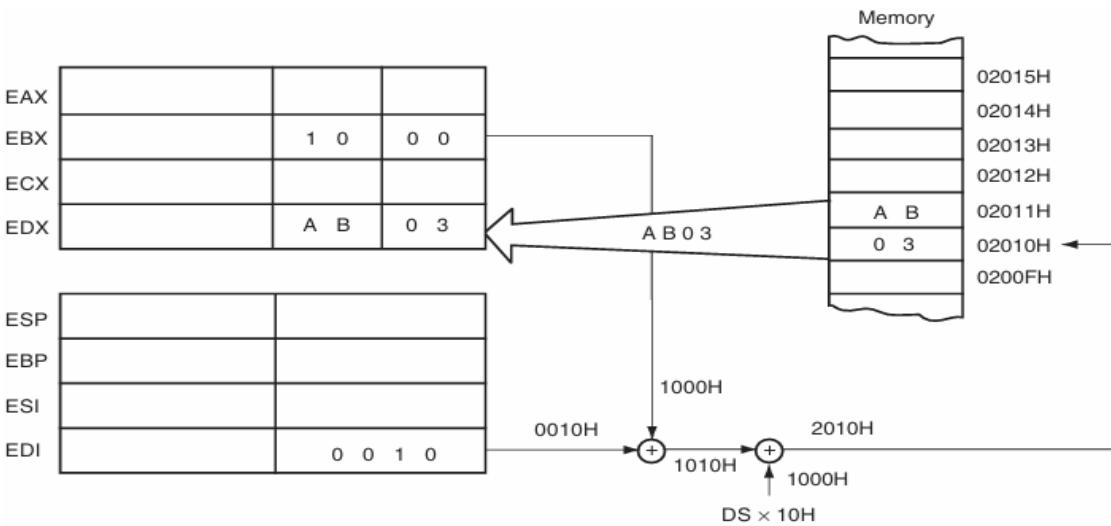
### 🧠 64-bit mode এ:

👉 যেকোনো দুইটা 64-bit register ব্যবহার করা যায় (যেমন: RAX, RBX, RSI, RDI, ইত্যাদি)  
এবং segment register দরকার হয় না, কারণ address টা linear হয়।

## ◆ ৩. Segment নির্ধারণের নিয়ম

Combination	Default Segment
BX + SI / BX + DI	DS (Data Segment)
BP + SI / BP + DI	SS (Stack Segment)

## ◆ ৮. Example: MOV DX,[BX+DI]



**FIGURE 3–8** An example showing how the base-plus-index addressing mode functions for the `MOV DX,[BX+DI]` instruction. Notice that memory address `02010H` is accessed because `DS = 0100H`, `BX = 100H`, and `DI = 0010H`.

ধরা যাক:

$DS = 0100H$   
 $BX = 1000H$   
 $DI = 0010H$

তাহলে instruction:

`MOV DX, [BX+DI]`

মানে হচ্ছে → DS segment-এর মধ্যে offset ( $BX + DI$ ) মানে ( $1000H + 0010H = 1010H$ )  
 এই address-এর word memory location থেকে data নিয়ে DX register-এ রাখবে।

❖ Physical Address হিসাব হবে:

$$\begin{aligned}
 \text{Physical Address} &= (DS \times 10H) + (BX + DI) \\
 &= (0100H \times 10H) + 1010H \\
 &= 2000H + 10H \\
 &= 2010H
 \end{aligned}$$

অর্থাৎ, `memory[2010H] → DX` এ যাবে।

#### ◆ ৫. কিছু সাধারণ উদাহরণ (Table 3–6 অনুযায়ী)

Instruction	Operation	Size
MOV CX, [BX+DI]	DS segment-এর BX+DI address থেকে 16-bit data নিয়ে CX এ রাখে	16-bit
MOV CH, [BP+SI]	SS segment-এর BP+SI address থেকে byte data নিয়ে CH এ রাখে	8-bit
MOV [BX+SI], SP	SP register-এর মান DS segment-এর BX+SI address এ রাখে	16-bit
MOV [BP+DI], AH	AH এর মান SS segment-এর BP+DI address এ রাখে	8-bit
MOV CL, [EDX+EDI]	DS segment-এর EDX+EDI address থেকে byte data নিয়ে CL এ রাখে	8-bit
MOV [EAX+EBX], ECX	DS segment-এর EAX+EBX address এ ECX (32-bit) রাখে	32-bit
MOV [RSI+RBX], RAX	RSI+RBX address এ RAX (64-bit) রাখে	64-bit

## ◆ ৬. Base + Index দিয়ে Array Access বোঝা

👉 ধরো তোমার কাছে একটা array আছে, যেমন:

ARRAY DB 10, 20, 30, 40, 50 ...

এখন যদি তুমি 3য় element (মানে index = 2) access করতে চাও,  
তাহলে base register-এ array-এর শুরু address (OFFSET ARRAY) রাখো,  
আর index register-এ element-এর position রাখো।

যেমন:

```
MOV BX, OFFSET ARRAY    ; base = ARRAY start
MOV DI, 2                ; index = 2
MOV AL, [BX + DI]        ; AL = ARRAY[2]
```

## ◆ ৮. সারসংক্ষেপ (এক নজরে)

বিষয়	ব্যাখ্যা
Address কোথায় থাকে দুটি register (Base + Index)	
Data কোথায় থাকে	Memory তে
Default Segment	DS বা SS (BP থাকলে SS)
Symbol [ ]	Indirect Addressing বোঝায়

বিষয়	ব্যাখ্যা
কাজের ব্যবহার	Array বা Table access করার সময়
64-bit Mode এ	Segment লাগে না, register pair দিয়েই linear address তৈরি হয়

## ◆ সহজ ভাষায় মনে রাখার ট্রিক \*

- 🧠 Base register = array শুরু
- 🔢 Index register = কোন element দরকার
- 🔍 [Base + Index] = ঐ element-এর ঠিকানা

## ◆ ① Register Relative Addressing মানে কী?

👉 এই addressing mode-এ একটা register (BX, BP, SI, DI) এর সঙ্গে একটা displacement (অর্থাৎ offset বা constant value) যোগ করে memory address তৈরি করা হয়।

- ➡ মানে,

Effective Address = Register + Displacement

এবং ওই address-এই memory থেকে data পড়া বা লেখা হয়।

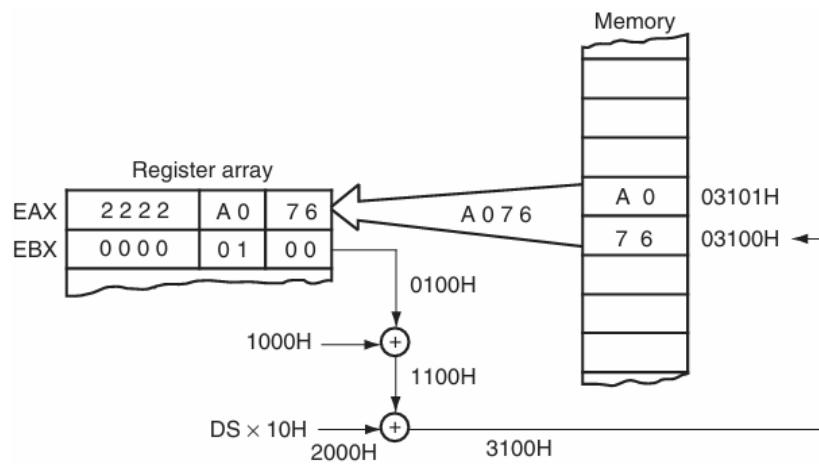
## ◆ ② উদাহরণ দিয়ে ব্যাখ্যা

ধরো, আমাদের instruction হলো:

MOV AX, [BX + 1000H]

এর মানে:

**FIGURE 3-10** The operation of the `MOV AX, [BX+1000H]` instruction, when `BX = 0100H` and `DS = 0200H`.



- BX রেজিস্টারে কোনো base address আছে (ধরো 0100H)
- তার সাথে 1000H displacement যোগ হবে
- এর ফলে effective address হবে
- $0100H + 1000H = 1100H$

এখন, DS segment register-এর মান যদি হয় 0200H,  
তাহলে Physical Address = DS \* 10H + Effective Address  
 $= 02000H + 01100H = 03100H$

এখানেই ডাটা রাখা বা পড়া হবে।

#### ◆ ৩. রেজিস্টার কোন segment ব্যবহার করে?

Register	Segment Register used
BX, SI, DI	DS (Data Segment)
BP	SS (Stack Segment)

অর্থাৎ — যদি BP ব্যবহার করো, তাহলে Stack Segment থেকে address তৈরি হবে।

#### ◆ ৪. Displacement মানে কী?

Displacement হচ্ছে একটা সংখ্যা (constant offset) — যা রেজিস্টারের সাথে যোগ (বা বিয়োগ) হয়।

উদাহরণ:

```
MOV AL, [DI + 2]      ; DI এর মান + 2
MOV AL, [SI - 1]      ; SI এর মান - 1
MOV AL, DATA[DI]       ; DATA segment-এর base + DI
MOV AL, DATA[DI + 3]   ; DATA + DI + 3
```

---



---

## ◆ ৪ Base-Plus-Index vs Register-Relative পার্থক্য

বিষয়	Base-Plus-Index	Register Relative
Address তৈরি হয়	Base + Index	Register + Displacement
উদাহরণ	[BX + DI]	[DI + 100H]
ব্যবহৃত হয়	সাধারণত array ও 2D data access	array বা structure access যেখানে displacement fix থাকে
Example	MOV AL, [BX+DI]	MOV AL, ARRAY[DI]

---

## ◆ ৫ মূল পয়েন্ট মনে রাখো

1. Register relative মানে register এর সাথে constant displacement যোগ করা।
  2. Effective address = Register + Displacement
  3. DS বা SS segment অনুযায়ী physical address তৈরি হয়।
  4. Array element access করার জন্য খুবই উপকারী।
- 

## Base Relative-Plus-Index Addressing Mode

### ◆ ১ সংজ্ঞা (Definition)

Base Relative-Plus-Index Addressing মানে হলো —

একটা base register (BX বা BP)

এর সাথে একটা index register (SI বা DI)

এবং একটা displacement (ধূর মান)

এই তিনটাকে যোগ করে effective memory address তৈরি করা হয়।

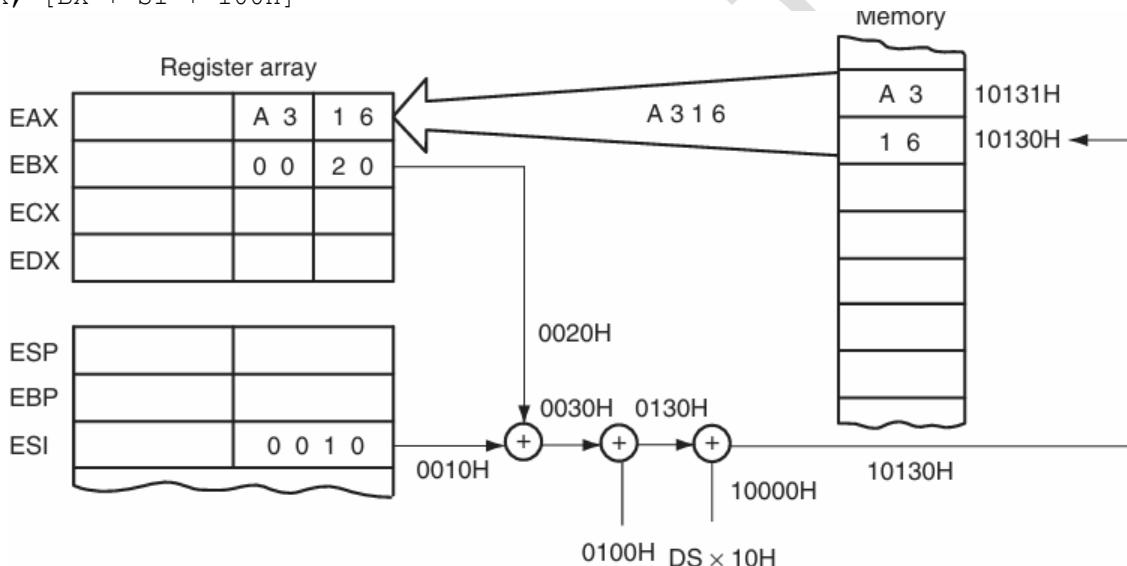
👉 অর্থাৎ,

Effective Address = Base Register + Index Register + Displacement

## ◆ ২ উদাহরণ বোৰা

ধৰো instruction হলো:

MOV AX, [BX + SI + 100H]



**FIGURE 3–12** An example of base relative-plus-index addressing using a `MOV AX,[BX+SI+100H]` instruction. Note:  $DS = 1000H$ .

এখানে —

- $BX = 0020H$
- $SI = 0010H$
- $\text{Displacement} = 0100H$
- $DS = 1000H$

তাহলে effective address:

$$0020H + 0010H + 0100H = 0130H$$

এবং physical address হবে:

$$(DS * 10H) + EA = 1000H + 0130H = 10130H$$

এই ঠিকানায় থাকা ডাটা AX রেজিস্টারে যাবে।

## ◆ ৩ কেন একে “Base Relative-Plus-Index” বলে?

কারণ এখানে:

- Base register (BX বা BP) → “base” নির্দেশ করে
- Index register (SI বা DI) → array বা record-এর “element” নির্দেশ করে
- Displacement → file বা array-এর “starting address” নির্দেশ করে

অর্থাৎ, একাধিক dimension বা nested data structure address করার জন্য ব্যবহৃত হয়।

## ◆ ৪ কখন ব্যবহার হয়?

এই addressing mode সাধারণত **দুই-স্তরের ডেটা** (two-dimensional array বা record) access করতে কাজে লাগে।

 যেমন:

- FILE (পুরো ডেটা ব্লক)
- তার ভিতরে RECORD (base)
- তার ভিতরে ELEMENT (index)

## ◆ ৫ উদাহরণ টেবিল (Table 3-8 থেকে)

Instruction	Size	Operation
MOV DH, [BX+DI+20H]	8 bit	BX + DI + 20H ঠিকানার ডাটা DH-তে কপি
MOV AX, FILE[BX+DI]	16 bit	FILE array-এর BX + DI অফসেট থেকে ডাটা AX-এ
MOV LIST[BP+SI+4], DH	8 bit	LIST + BP + SI + 4 ঠিকানায় DH লিখছে

## ◆ ৬ মাস্তিক বোঝাপড়া

ধরো,

- প্রতিটি record = 10 byte
- Record A শুরু 0000H offset থেকে
- Record C শুরু 0014H offset থেকে  
(কারণ RECA = 10H byte, RECB = 10H byte → তাই  $0000H + 0014H = 0014H$ )

তাহলে:

Address	Content	অর্থ
FILE + RECA + 0 = 0000H	A[0]	$AL \leftarrow A[0]$
FILE + RECC + 2 = 0016H	C[2]	$C[2] \leftarrow AL$

## ◆ ৯ মূল ধারণা

বিষয়	ব্যাখ্যা
Base register	Record বা subarray নির্দেশ করে
Index register	Element নির্দেশ করে
Displacement	পুরো file বা array এর শুরু নির্দেশ করে
Formula	Effective Address = Base + Index + Displacement
ব্যবহার	Two-dimensional array, record structure access

## ◆ 10 সারাংশ (Summary)

Addressing Mode	Formula	ব্যবহৃত হয়
Base-Plus-Index	Base + Index	সাধারণ array access
Register Relative	Register + Displacement	এক-স্তরের array access
Base Relative-Plus-Index	Base + Index + Displacement	দুই-স্তরের array (record/file access)

## Scaled-Index Addressing Mode

## ◆ 1 সংজ্ঞা (Definition)

Scaled-Index Addressing হলো এমন একটি addressing mode যা 80386 এবং পরবর্তী মাইক্রোপ্রসেসরগুলোতে (Core2 পর্যন্ত) পাওয়া যায়।

এখানে memory address তেরি হয় একটা base register এবং একটা index register-এর গুণফল (scaled value) যোগ করে।

👉 মূল সূত্র:

Effective Address = Base Register + (Index Register × Scale Factor)

## ◆ ২ Scale Factor কী?

Scale factor মানে হলো index রেজিস্টারের মানকে গুণ করা।  
এটা হতে পারে —

1×, 2×, 4×, অথবা 8×

প্রতিটি scaling factor আলাদা ধরণের data size access করতে ব্যবহৃত হয়:

Scale	ব্যবহৃত হয়	উদাহরণ
1×	Byte array	MOV AL,[EBX+ECX]
2×	Word array (2 bytes)	MOV AX,[EDI+2*ECX]
4×	Doubleword array (4 bytes)	MOV EAX,[EBX+4*ECX]
8×	Quadword array (8 bytes)	MOV RAX,[8*EDI]

## ◆ ৩ কিভাবে কাজ করে?

ধরো instruction:

MOV AX, [EDI + 2\*ECX]

এখানে:

- EDI → Base Register
- ECX → Index Register
- Scale Factor = 2

তাহলে যদি ECX = 3 হয়:

Effective Address = EDI + (ECX × 2)

$$\begin{aligned}
 &= EDI + (3 \times 2) \\
 &= EDI + 6
 \end{aligned}$$

অর্থাৎ, EDI-এর starting address থেকে 6 bytes দূরের জায়গা access করা হচ্ছে।

---

## ◆ 4 কেন এই mode দরকার?

ধরো, তোমার একটা **array of words** (প্রতিটি word = 2 bytes)।  
সাধারণ index দিলে প্রতিটি element-এর address = base + index,  
কিন্তু প্রতিটি element আসলে 2 byte করে shift হয়।  
তাই index কে  $2 \times$  করতে হয় যেন word alignment ঠিক থাকে।

এই কারণেই scaled-index addressing দরকার হয়।

এতে array element access করা সহজ হয়ে যায়।

---

## ◆ 5 কিছু উদাহরণ (Table 3-9 থেকে)

Instruction	Size	Operation
MOV EAX, [EBX+4*ECX]	32 bit	EBX + (4×ECX) ঠিকানার ডাটা EAX-এ কপি
MOV [EAX+2*EDI+100H], CX	16 bit	(EAX + 100H + 2×EDI) ঠিকানায় CX লেখা
MOV AL, [EBP+2*EDI+2]	8 bit	(EBP + 2 + 2×EDI) ঠিকানার byte AL-এ পড়ছে
MOV EAX, ARRAY[4*ECX]	32 bit	ARRAY + (4×ECX) ঠিকানার doubleword EAX-এ পড়ছে

---

## ◆ 7 মাস্তবিকভাবে বুঝো (Word Array Example)

ধরো LIST array টা মেমরিতে আছে এমনভাবে:

Index	Address	Data
0	0000	0
1	0002	1
2	0004	2
3	0006	3
4	0008	4

Index	Address	Data
5	000A	5
6	000C	6
7	000E	7

যখন  $\text{MOV AX, [EBX+2*ECX]}$  এ  $\text{ECX} = 2$ ,  
তখন  $\text{address} = 0000 + (2 \times 2) = 0004 \rightarrow \text{LIST}[2] = 2 \rightarrow \text{AX}$  এ যাবে।

তারপর  $\text{MOV [EBX+2*ECX], AX}$  এ  $\text{ECX} = 4$ ,  
 $\text{address} = 0000 + (2 \times 4) = 0008 \rightarrow \text{LIST}[4] = 2$  লেখা হবে।

এভাবে  $\text{LIST}[7]$  (address 000E) তেও 2 লেখা হবে।

## ◆ ৪ সারসংক্ষেপ

বিষয়	ব্যাখ্যা
ব্যবহৃত রেজিস্টার	Base + Index
Formula	Base + (Index $\times$ Scale)
Scale Factor	1, 2, 4, বা 8
ব্যবহৃত ক্ষেত্র	Array element access (byte, word, doubleword, quadword)
সুবিধা	দ্রুত ও সরাসরি array indexing
Mode পাওয়া যায়	80386 এবং এর পরের প্রসেসরগুলোতে

### 3.2 ⚡ Program Memory Addressing Modes

👉 এগুলো হলো সেই পদ্ধতি যেভাবে instruction pointer (IP) কে নতুন address-এ পাঠানো হয় — অর্থাৎ প্রোগ্রাম কোড কোথা থেকে execute হবে, সেটি নির্ধারণ করা হয়।

এই addressing mode মূলত **JMP (Jump)** এবং **CALL** ইনস্ট্রাকশন-এর সাথে ব্যবহার হয়।

তিনটি প্রধান ধরন আছে:

1. Direct Program Memory Addressing
2. Relative Program Memory Addressing
3. Indirect Program Memory Addressing

## 1. Direct Program Memory Addressing

এখানে target address সরাসরি instruction-এর ভেতরেই দেওয়া থাকে।

উদাহরণ:

JMP 10000H

👉 এর মানে:

Microprocessor সরাসরি CS:IP = 1000H:0000H সেট করে দেবে (মানে memory location 1000H থেকে পরবর্তী instruction execute হবে)।

এটা **একটা far jump কারণ এটি যেকোনো সেগমেন্টে (segment) jump করতে পারে।**

The instructions for direct program memory addressing store the address with the opcode. For example, if a program jumps to memory location 10000H for the next instruction, the address (10000H) is stored following the opcode in the memory. Figure 3–14 shows the direct intersegment JMP instruction and the 4 bytes required to store the address 10000H. This JMP instruction loads CS with 1000H and IP with 0000H to jump to memory location 10000H for the next instruction. (An **intersegment jump** is a jump to any memory location within the entire memory system.) The direct jump is often called a **far jump** because it can jump to any memory location for the next

**FIGURE 3–14** The 5-byte machine language version of a JMP [1000H] instruction.

Opcode	Offset (low)	Offset (high)	Segment (low)	Segment (high)
E A	0 0	0 0	0 0	1 0

◆ কেমনভাবে কাজ করে:

Byte	Description	Value (Hex)
1	Opcode (JMP FAR)	EA
2	Offset Low Byte	00
3	Offset High Byte	00
4	Segment Low Byte	00
5	Segment High Byte	10

→ অর্থাৎ EA 00 00 00 10 = JMP [1000H]  
(Opcode + Offset + Segment)

---

◆ কাজের ধাপ:

1. CS রেজিস্টারে নতুন segment (1000H) লোড হয়
2. IP রেজিস্টারে নতুন offset (0000H) লোড হয়
3. Control চলে যায় এই memory location-এ

অর্থাৎ execution flow এখন থেকে CS:IP = 1000H:0000H থেকে শুরু হবে।

---

◆ বাস্তব উদাহরণ:

ধরা যাক, তোমার কোডে একটা লেবেল আছে:

```
MAIN:    MOV AX, 5
          JMP NEXT
          ADD AX, 1
NEXT:    SUB AX, 2
```

এখানে:

- JMP NEXT একটি **direct jump**
  - কারণ assembler NEXT লেবেলের exact address resolve করে দেয় (মানে instruction-এর মধ্যে সরাসরি সেই address লিখে দেয়)।
- 

Difference between Intersegment and Intrasegment Jump

Basis	Intrasegment Jump	Intersegment Jump
Definition	A jump <b>within the same code segment.</b>	A jump <b>to another code segment.</b>
Effect on Registers	Changes only the <b>Instruction Pointer (IP).</b>	Changes both <b>Code Segment (CS)</b> and <b>Instruction Pointer (IP).</b>
Address Range	Target address lies <b>within the current segment.</b>	Target address lies <b>outside the current segment.</b>
Instruction Format	Usually a <b>short or near jump.</b>	Always a <b>far jump.</b>

Basis	Intrasegment Jump	Intersegment Jump
Example	JMP LABEL (within same segment)	JMP FAR PTR NEWLABEL (in another segment)
Memory Access Range		

#### ◆ Real Mode বনাম Protected Mode

Mode	কিসে Jump হয়	Memory Access Range
Real Mode	CS + IP	প্রথম 1 MB (20-bit address)
Protected Mode	Descriptor Table ব্যবহার করে	পুরো 4 GB পর্যন্ত
64-bit Mode	CS শুধু privilege define করে, address নয়	প্রায় সীমাহীন address space

#### ◆ CALL Instruction

CALL ও একইভাবে কাজ করে, শুধু অতিরিক্তভাবে return address stack-এ সংরক্ষণ করে, যাতে RET দিলে আবার আগের জায়গায় ফিরে আসা যায়।

#### 💡 সংক্ষেপে:

Addressing Type	কীভাবে কাজ করে	উদাহরণ
Direct	Address সরাসরি instruction-এর মধ্যে	JMP 10000H
Relative	Address বর্তমান অবস্থান থেকে offset হিসেবে	JMP SHORT LABEL
Indirect	Address রেজিস্টার বা মেমোরির ভেতরে	JMP [BX]

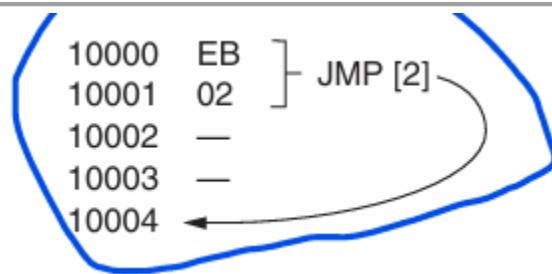
## ■ 1 Relative Program Memory Addressing

#### ◆ মানে কী?

Relative মানে — বর্তমান অবস্থানের (অর্থাৎ Instruction Pointer – IP) তুলনায় কোথায় যেতে হবে।

👉 অর্থাৎ **জাম্পের টাগেট অ্যাড্রেস** সরাসরি না লিখে, কেবল কত “দূরত্বে” যেতে হবে (offset বা displacement) সেটা দেওয়া থাকে।

**FIGURE 3–15** A JMP [2] instruction. This instruction skips over the 2 bytes of memory that follow the JMP instruction.



◆ কেমনভাবে কাজ করে?

ধরা যাক, তোমার প্রোগ্রামে এই লাইন আছে:

JMP +2

এর মানে হলো:

বর্তমান instruction শেষ হবার পর ২ বাইট সামনে গিয়ে jump করো।

অর্থাৎ microprocessor বর্তমান IP-এর সাথে ২ যোগ করে নতুন address নির্ধারণ করবে।

**Instruction Pointer (IP) সম্পর্ক:**

- IP রেজিস্টারে থাকে বর্তমান instruction-এর address
- relative addressing এ “নতুন address = বর্তমান IP + displacement”

◆ Displacement এর ধরণ:

ধরণ	বাইট সাইজ	জাম্পের সীমা	নাম
8-bit	±128 বাইট (-128 থেকে +127)	Short Jump	
16-bit	±32K বাইট	Near Jump	
32-bit	±2GB (80386+ প্রটোকেড মোডে)	Long Jump	

◆ উদাহরণ:

START: MOV AX, 1  
JMP NEXT ; relative jump (assembler স্বয়ংক্রিয়ভাবে দূরত্ব হিসাব করে)

```
ADD AX, 2      ; এই লাইনটা স্কিপ হবে
NEXT: INC AX
```

এখানে assembler `JMP NEXT` দেখে নিজে থেকেই IP-এর সাথে কত ঘোগ করতে হবে সেটা বের করে নেয় — এটা-ই relative addressing।

---

#### ◆ Short vs Near jump

ধরন	কিসে ব্যবহৃত	কী করে
Short jump (8-bit)	কাছাকাছি কোডে যাওয়ার জন্য	IP $\pm$ 128 বাইটের মধ্যে
Near jump (16-bit)	একই segment-এ দূরের কোডে যাওয়ার জন্য	IP $\pm$ 32K বাইটের মধ্যে

---

#### 💡 আসলে এটা assembler-এর কাজ:

তুমি কেবল লেখো `JMP LABEL`

Assembler নিজেই দেখে নিচে কত দূর এবং কোন ফরম্যাটে (1-byte, 2-byte, 4-byte displacement) encode করবে।

### Relative Program Memory Addressing

Relative program memory addressing is not available in all early microprocessors, but it is available to this family of microprocessors. The term *relative* means “relative to the instruction pointer (IP).” For example, if a JMP instruction skips the next 2 bytes of memory, the address in relation to the instruction pointer is a 2 that adds to the instruction pointer. This develops the address of the next program instruction. An example of the relative JMP instruction is shown in Figure 3–15. Notice that the JMP instruction is a 1-byte instruction, with a 1-byte or a 2-byte displacement that adds to the instruction pointer. A 1-byte displacement is used in **short** jumps, and a 2-byte displacement is used with **near** jumps and calls. Both types are considered to be intrasegment jumps. (An **intrasegment jump** is a jump anywhere within the current code segment.) In the 80386 and above, the displacement can also be a 32-bit value, allowing them to use relative addressing to any location within their 4G-byte code segments.

- Relative JMP and CALL instructions contain either an 8-bit or a 16-bit signed displacement that allows a forward memory reference or a reverse memory reference. (The 80386 and above can have an 8-bit or 32-bit displacement.) All assemblers automatically calculate the distance for the displacement and select the proper 1-, 2- or 4-byte form. If the distance is too far for a 2-byte displacement in an 8086 through an 80286 microprocessor, some assemblers use the direct jump. An 8-bit displacement (*short*) has a jump range of between +127 and –128 bytes from the next instruction; a 16-bit displacement (*near*) has a range of  $\pm 32K$  bytes. In the 80386 and above, a 32-bit displacement allows a range of  $\pm 2G$  bytes. The 32-bit displacement can only be used in the protected mode.

## ■ 2 Indirect Program Memory Addressing

- ◆ মানে কী?

এখানে jump করার address সরাসরি বা relative না হয়,  
একটা register বা memory location-এর ভেতর রাখা থাকে।

👉 অর্থাৎ “কোথায় jump করতে হবে” সেটা register/memory থেকে নেওয়া হয়।

- ◆ ধরো এই লাইনটা:

JMP BX

মানে হলো —

BX রেজিস্টারের ভেতরে যে address আছে, সেখানে jump করো।

যদি BX = 1000H,

তাহলে microprocessor যাবে CS:1000H address-এ।

এটা বলা হয় near indirect jump, কারণ এটি একই segment-এর ভেতর।

- ◆ আরও কিছু উদাহরণ:

Assembly	কী করে
JMP AX	AX-এ রাখা address-এ jump করে
JMP [BX]	BX register-এর মানকে offset ধরে data segment memory থেকে address নেয় এবং তাতে jump করে
JMP TABLE [BX]	BX offset হিসেবে ব্যবহার করে TABLE নামের memory থেকে address নেয়
JMP ECX	80386+ CPU তে ECX register-এর address-এ jump
JMP RDI	64-bit মডেলে RDI register-এর address-এ jump

- ◆ Jump Table (একটি বাস্তব উদাহরণ):

ধরা যাক তোমার কোডে ৪টা subroutine-এর ঠিকানা রাখা আছে:

```
TABLE    DW LOC0
        DW LOC1
        DW LOC2
        DW LOC3
```

এখন তুমি যদি লিখো:

```
MOV BX, 4
JMP TABLE[BX]
```

👉 তাহলে এটা করবে:

- TABLE নামের মেমরি ব্লকের ভেতরে BX-এর মান (4) যোগ করবে
- সেই জায়গার 16-bit মান পড়বে
- এবং সেই ঠিকানায় jump করবে

◆ একে বলা হয় “Jump Table” ব্যবহার করা,

যেটা switch-case এর মতো কাজ করে:

```
switch(x) {
    case 0: goto LOC0;
    case 1: goto LOC1;
    case 2: goto LOC2;
    case 3: goto LOC3;
}
```

Assembly তে এটা indirect jump table দিয়ে করা হয়।

### সংক্ষেপে তুলনা:

ধরন	Address কোথা থেকে আসে	Jump এর সীমা	উদাহরণ
Direct	Instruction এর মধ্যে সরাসরি address	যেকোনো জায়গায়	JMP 10000H
Relative	বর্তমান IP-এর সাথে displacement যোগ করে	$\pm 128 / \pm 32K / \pm 2G$	JMP LABEL
Indirect	Register বা memory থেকে address নেয়	Register/memory অনুযায়ী	JMP BX, JMP [BX]

সারসংক্ষেপ:

- **Relative jump** → IP + displacement
  - **Indirect jump** → address register/memory থেকে আসে
  - **Direct jump** → address instruction-এই দেওয়া থাকে
- 

## Stack Memory Addressing Modes

---



### Stack কী?

👉 Stack হলো একটা বিশেষ মেমোরি এরিয়া, যেখানে ডেটা সাময়িকভাবে রাখা হয় (temporary storage)।  
এটা LIFO (Last-In, First-Out) পদ্ধতিতে কাজ করে —  
মানে, যে ডেটা সর্বশেষে PUSH করা হয়, সেটাই সবার আগে POP হবে।

---



### Stack ব্যবহার হয় কেন?

Microprocessor এ Stack ব্যবহার হয় তিনভাবে:

1. **Temporary data storage** – কিছু সময়ের জন্য মান রাখার জন্য।
  2. **Procedure return address** রাখার জন্য – যখন CALL দেওয়া হয়, তখন ফিরে আসার ঠিকানাটা Stack এ যায়।
  3. **Register value সংরক্ষণ ও পুনরুদ্ধার করার জন্য** – PUSH / POP দিয়ে।
- 



### Stack Register গুলো

Stack দুইটা register দিয়ে maintain হয়:

- **SS (Stack Segment register)** → Stack এর base segment।
- **SP (Stack Pointer)** → Stack এর current top (উপরের অংশ) নির্দেশ করে।



👉 Stack এর আসল Address হয়:

$$\text{Physical Address} = \text{SS} \times 10H + \text{SP}$$

---

## PUSH কিভাবে কাজ করে

ধৰা যাক, তুমি একটা word (২ byte) stack এ PUSH করছো।

Before PUSH:

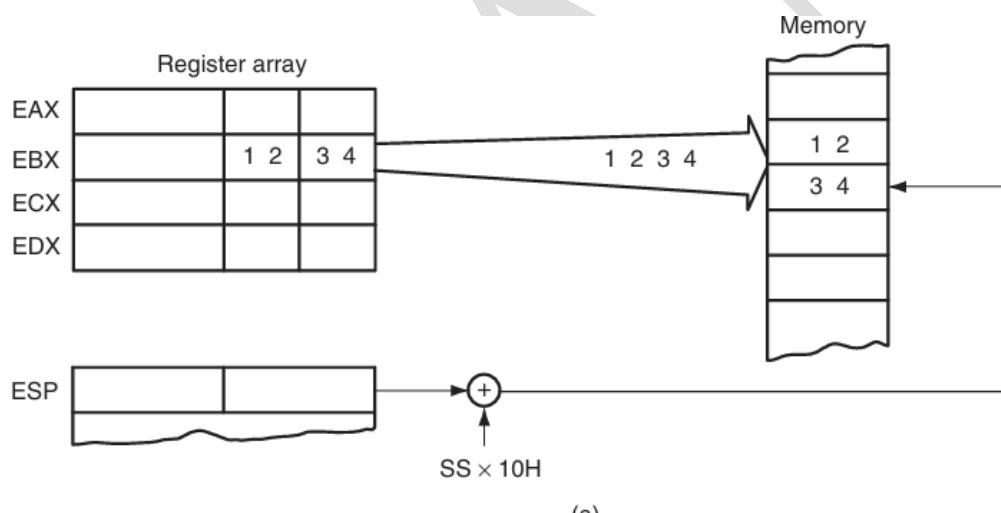
SP → 3000H

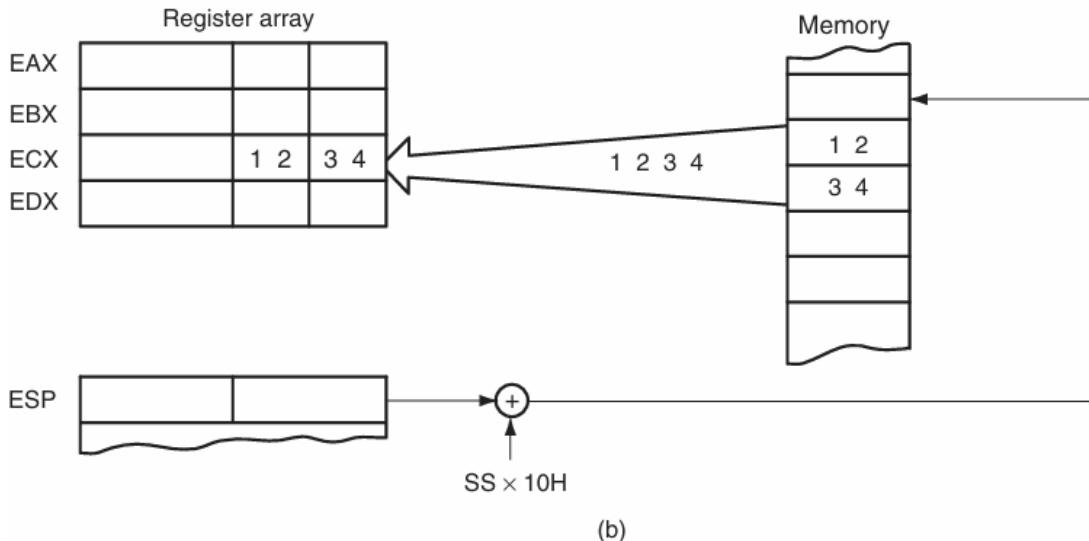
যখন PUSH AX করা হয়:

1. SP থেকে ২ কমানো হয় ( $SP = SP - 2$ ), কারণ stack নিচের দিকে বাড়ে।
2. AX রেজিস্টারের High byte যায়  $[SS:SP+1]$  এ  
এবং Low byte যায়  $[SS:SP]$  এ।

👉 অর্থাৎ, stack এ data এইভাবে নামে:

Top → Low byte  
Next → High byte





**FIGURE 3–17** The PUSH and POP instructions: (a) PUSH BX places the contents of BX onto the stack; (b) POP CX removes data from the stack and places them into CX. Both instructions are shown after execution.



## POP কিভাবে কাজ করে

যখন POP করা হয়:

1. [SS:SP] থেকে Low byte এবং [SS:SP+1] থেকে High byte পড়া হয়।
2. তারপর  $SP = SP + 2$  হয়।

👉 ফলে stack থেকে উপরের ডেটা সরিয়ে নেওয়া হয়, নিচেরটা উপরে উঠে আসে।

## 💻 PUSH এবং POP এর Instruction গুলো (Table 3–11 থেকে সংক্ষিপ্তভাবে)

Instruction	কাজ
PUSH AX	AX রেজিস্টারের মান stack এ রাখে
POP BX	Stack এর মান BX এ নিয়ে আসে
PUSH DS	DS segment register stack এ রাখে
PUSH 1234H	1234H মান stack এ রাখে (immediate push)
PUSHA	একসাথে সব general-purpose register stack এ রাখে

Instruction	কাজ
POPA	একসাথে সব general-purpose register stack থেকে নেয়।
PUSHAD / POPAD	32-bit রেজিস্টারগুলোর জন্য।
PUSHF / POPF	Flag register push বা pop করে।

⚠ Note: `POP CS` অবৈধ (illegal), কারণ CS পরিবর্তন করলে পরবর্তী instruction address বদলে যায়।

---

## 🧠 মূল কথা

- Stack হলো LIFO মেমোরি।
  - PUSH → SP কমায়
  - POP → SP বাড়ায়
  - Stack segment (SS) + SP = আসল মেমোরি ঠিকানা।
  - CALL, RET, PUSH, POP, PUSHA, POPA সব Stack ব্যবহার করে।
-