

◆ MOV Revisited — অর্থাৎ MOV নির্দেশের গভীর বিশ্লেষণ

MOV instruction (move) ব্যবহার হয় এক স্থান থেকে অন্য স্থানে ডাটা কপি করার জন্য।
যেমন:

MOV AX, BX ; BX এর মান AX এ কপি হবে

এই অধ্যায়ে MOV instruction ব্যবহার করে **machine language (binary code)** কেমন তৈরি হয় এবং কিভাবে CPU তা বোঝে — সেটাই ব্যাখ্যা করা হয়েছে।

1. Machine Language কী?

Machine language হচ্ছে **binary code (0 ও 1)** যা মাইক্রোপ্রসেসর সরাসরি বুঝে ও চালায়।

যেমন:

MOV AX, BX

এই এক লাইনের assembly code টা machine language এ রূপ নিলে হতে পারে:

10001011 11011000 (binary form)

👉 অর্থাৎ CPU এই binary instruction টা পড়ে বুঝে কোন কাজ করবে।

2. Instruction এর দৈর্ঘ্য

8086 থেকে Core2 পর্যন্ত প্রসেসরগুলোর একটি instruction এর দৈর্ঘ্য হতে পারে

👉 1 byte থেকে 13 byte পর্যন্ত।

কেন এমন হয়?

কারণ, instruction এর মধ্যে অনেক কিছু থাকে:

- opcode (operation কোড)
- address / register / displacement / immediate value ইত্যাদি



৩. 16-bit বনাম 32-bit Instruction Mode

✿ 16-bit Instruction Mode (8086, 80286 ইত্যাদি)

এই mode এ instruction এর সাধারণ গঠন এমন:

অংশ	আকার	কাজ
Opcode	1–2 bytes	কী কাজ করবে (যেমন MOV, ADD, SUB ইত্যাদি)
MOD-REG-R/M	0–1 bytes	কোন register বা memory ব্যবহার হবে
Displacement	0–1 bytes	memory address offset
Immediate	0–2 bytes	সরাসরি value

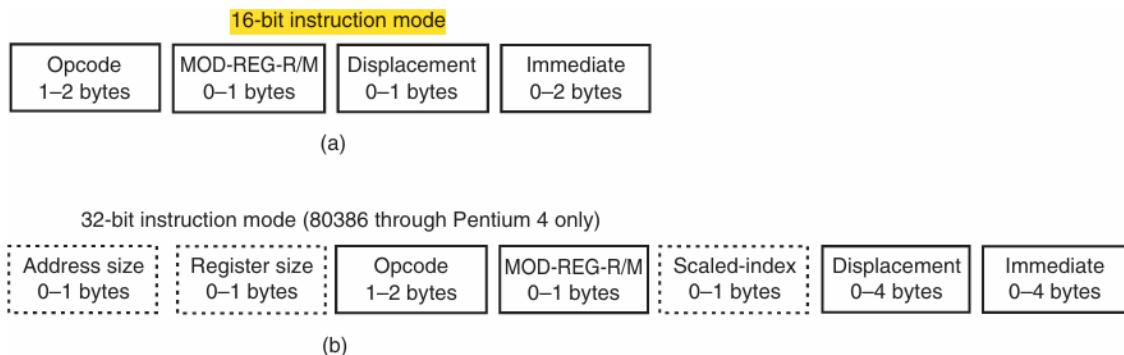


FIGURE 4-1 The formats of the 8086–Core2 instructions. (a) The 16-bit form and (b) the 32-bit form.

✿ 32-bit Instruction Mode (80386 – Pentium 4)

এখানে গঠনটা একটু বড়:

অংশ	আকার	কাজ
Address size prefix	0–1 bytes	address 16-bit না 32-bit হবে
Register size prefix	0–1 bytes	register 16-bit না 32-bit হবে
Opcode	1–2 bytes	operation
MOD-REG-R/M	0–1 bytes	addressing mode
Scaled-index	0–1 bytes	index register
Displacement	0–4 bytes	offset
Immediate	0–4 bytes	সরাসরি value

✳️ 8. Prefix কী?

Prefix মানে — instruction এর আগে যোগ করা বিশেষ byte যা register বা address size পরিবর্তন করে।

Prefix	Hex Code	কাজ
Register-size prefix	66H	16 ↔ 32-bit register toggle
Address-size prefix	67H	16 ↔ 32-bit address toggle

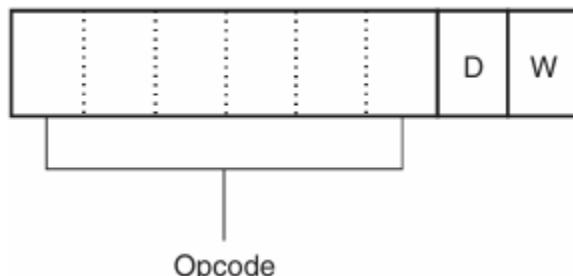
উদাহরণ:

- যদি 16-bit mode এ 32-bit register ব্যবহার করতে চাও ⚡ prefix হবে **66H**
- আবার 32-bit mode এ 16-bit register ব্যবহার করতে চাও ⚡ তাতেও prefix হবে **66H**

✳️ ৫. Opcode কী?

Opcode হলো instruction এর সেই অংশ যা বলে দেয় কী কাজ হবে — যেমন MOV, ADD, SUB ইত্যাদি।

FIGURE 4–2 Byte 1 of many machine language instructions, showing the position of the D- and W-bits.



Opcode এর গঠন (1st byte):

Bits	কাজ
প্রথম 6 bits	operation (যেমন MOV)
7th bit (D)	Direction bit — ডাটা কোন দিকে যাবে
8th bit (W)	Word bit — byte না word/dword সেটা বোঝায়

👉 Direction (D) bit:

- D = 1 → ডাটা যাবে REG → R/M
- D = 0 → ডাটা যাবে R/M → REG

👉 Word (W) bit:

- W = 0 → 8-bit data (byte)
- W = 1 → 16-bit বা 32-bit data (word/dword)

✳️ ৫. Second Byte: MOD-REG-R/M

এই byte নির্দেশ করে ডাটা কোন register বা memory location থেকে আসবে / যাবে।

with the MOV and some other instructions. Refer to Figure 4–3 for the binary bit pattern of the second opcode byte (reg-mod-r/m) of many instructions. Figure 4–3 shows the location of the MOD (mode), REG (register), and R/M (register/memory) fields.

FIGURE 4–3 Byte 2 of many machine language instructions, showing the position of the MOD, REG, and R/M fields.



Field	Bits	কাজ
MOD	2 bits	Addressing mode (direct, indirect, etc.)
REG	3 bits	কোন register
R/M	3 bits	register না memory, সেটা বোঝায়

✳️ ১. 16-bit বনাম 32-bit ব্যবহারের নিয়ম

- DOS (real mode) → কেবল 16-bit mode
- Windows / Linux (protected mode) → 16-bit ও 32-bit দুটোই সম্ভব
- প্রোগ্রামের ধরন অনুযায়ী mode বেছে নেওয়া হয়:
 - যদি বেশি 8-bit ও 16-bit data লাগে → 16-bit mode

- যদি বেশি 8-bit ও 32-bit data লাগে → 32-bit mode

সারাংশ

বিষয়

সংক্ষেপে

Machine Language Binary instruction যা CPU বোঝে

Instruction Mode 16-bit (8086) বা 32-bit (80386 ও পরবর্তী)

Prefix 66H বা 67H দিয়ে size পরিবর্তন

Opcode Operation নির্ধারণ করে

D bit Data এর দিক নির্ধারণ করে

W bit Data এর size নির্ধারণ করে

MOD-REG-R/M Addressing mode ও register selection

চলো ধাপে ধাপে ব্যাখ্যা করি, যেন MOD field, REG field, এবং R/M field পুরোপুরি বুঝে ফেলো



১. MOD Field কী?

MOD field (২ বিট) নির্দেশ করে instruction এ কোন ধরনের addressing mode ব্যবহাত হচ্ছে
অর্থাৎ — ডাটা register থেকে আসছে না memory address থেকে, আর displacement (offset value) আছে কিনা।

◆ MOD field এর মান ও তার মানে (16-bit mode)

TABLE 4-1 MOD field for the 16-bit instruction mode.

MOD	Function
00	No displacement
01	8-bit sign-extended displacement
10	16-bit signed displacement
11	R/M is a register

MOD bits	মানে	Displacement
00	Register indirect addressing	No displacement
01	Register indirect with 8-bit displacement	8-bit displacement
10	Register indirect with 16-bit displacement	16-bit displacement
11	Register addressing	Register → Register (no memory)

উদাহরণ

Instruction	MOD মান	ব্যাখ্যা
MOV AL, [DI]	00	কোনো displacement নাই
MOV AL, [DI+2]	01	8-bit displacement আছে (2H)
MOV AL, [DI+1000H]	10	16-bit displacement আছে (1000H)
MOV BP, SP	11	Register থেকে register এ data যাও

২. Displacement এবং Sign Extension

যখন displacement 8-bit হয়, তখন CPU একে sign-extend করে 16-bit এ নেয়।

👉 Sign-extension মানে:

- যদি 8-bit মান 00H–7FH হয় → positive → উপরে 00H যোগ হবে
যেমন: 7FH → 007FH
- যদি 8-bit মান 80H–FFH হয় → negative → উপরে FFH যোগ হবে
যেমন: FFH → FFFFH

Sign-bit (MSB) টি কপি করে পরের byte এ বসানো হয়।

এতে CPU বুঝতে পারে offset positive না negative।

৩. 32-bit Mode এ MOD Field কেমন?

80386 থেকে শুরু করে 32-bit processor গুলোতে MOD field একইভাবে কাজ করে,
তবে যখন MOD = 10, তখন displacement 16-bit নয়, বরং 32-bit হয়।

অর্থাৎ:

- MOD = 00 → No displacement
- MOD = 01 → 8-bit displacement
- MOD = 10 → 32-bit displacement
- MOD = 11 → Register addressing

TABLE 4–2 MOD field for the 32-bit instruction mode (80386–Core2 only).

MOD	Function
00	No displacement
01	8-bit sign-extended displacement
10	32-bit signed displacement
11	R/M is a register

এতে 32-bit processor সহজে 4GB memory পর্যন্ত address করতে পারে।

8. REG Field এবং R/M Field

এই দুইটা field 3-bit করে, মানে ৮টি সম্ভাব্য মান (000 থেকে 111)।

◆ REG Field

👉 Destination বা Source register নির্দেশ করে।

কোনটা source বা destination হবে, সেটা opcode এর D-bit (Direction bit) ঠিক করে।

◆ R/M Field

👉 Register বা memory operand নির্দেশ করে।

যদি MOD = 11 → Register select করে

যদি MOD ≠ 11 → Memory addressing mode select করে



Register Assignment (REG ও R/M ফিল্ডের জন্য)

Bits	যখন W=0 (8-bit)	যখন W=1 (16-bit)	যখন W=1 (32-bit)
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
106	DH	SI	ESI
111	BH	DI	EDI

✳️ ৫. উদাহরণ বিশ্লেষণ

🧠 Example 1: 8B ECCh

Instruction bytes:

8B EC

এখন বিশ্লেষণ করি:

অংশ	মানে
Opcode (8B)	Binary: 10001011 → MOV instruction
D = 1	Destination register হলো REG field
W = 1	Word (16-bit) data
MOD-REG-R/M byte (EC)	Binary: 11101100
MOD = 11	Register to register
REG = 101	BP register
R/M = 100	SP register

👉 তাই instruction = **MOV BP, SP**

Exam e thakbe
ei type er

Opcode				D	W	MOD		REG		R/M
1	0	0	0	1	0	1	1	0	1	1

Opcode = MOV
D = Transfer to register (REG)
W = Word
MOD = R/M is a register
REG = BP
R/M = SP

FIGURE 4–4 The 8BEC instruction placed into bytes 1 and 2 formats from Figures 4–2 and 4–3. This instruction is a MOV BP,SP.

🧠 **Example 2:** 66 8B E8h

Bytes:

66 8B E8

Byte	মানে
66h	Prefix → Register-size override (32-bit register ব্যবহার হবে)
8Bh	Opcode → MOV
E8h	MOD-REG-R/M: 11101000

এখানে:

- MOD = 11 → Register addressing
- REG = 101 → EBP
- R/M = 000 → EAX

👉 Instruction: **MOV EBP, EAX**

✳️ ৬. Prefix এর ভূমিকা আবার মনে রাখো

Prefix	কোড	কাজ
66h	Register size override	16 ↔ 32-bit register পরিবর্তন
67h	Address size override	16 ↔ 32-bit address পরিবর্তন

✳️ ৭. Mode নির্ধারণ কিভাবে হয়

Assembler program (.ASM) লিখলে:

- .386 directive দিলে 32-bit mode হবে।
- .MODEL statement এর আগে .386 দিলে 32-bit; পরে দিলে 16-bit।
- Visual Studio এর inline assembler সবসময় 32-bit mode এ চলে।

✳️ সারাংশ টেবিল

Field	Bits	কাজ
MOD	2	Addressing type ও displacement
REG	3	Register (destination/source)
R/M	3	Register বা memory operand
D	1	Data direction
W	1	Byte বা Word/Dword size
Prefix	1 byte	66h বা 67h দিয়ে register/address size পরিবর্তন

খুব ভালো, এখন আমরা বোঝার চেষ্টা করবো R/M (Register/Memory) Field কিভাবে কাজ করে

✳️ ১. প্রথমে মনে রাখো: Instruction Structure

প্রতিটি instruction এর মধ্যে এই ফরম্যাট থাকে 👇

| Opcode (1-2 byte) | MOD-REG-R/M (1 byte) | Displacement (0, 1, বা 2 byte) | Immediate (optional) |

এখনে গুরুত্বপূর্ণ হলো MOD-REG-R/M byte, যেটা তিন ভাগে ভাগ হয় —

Field	Bits	কাজ
MOD	2 bits	Addressing mode নির্ধারণ করে (displacement আছে কিনা)
REG	3 bits	কোন register destination/source তা বলে

Field	Bits	কাজ
R/M	3 bits	কোন register বা memory address ব্যবহার হবে তা বলে

২. MOD field = 00, 01, বা 10 মানে কী?

👉 Register নং, memory ব্যবহার হচ্ছে।

এবং displacement (offset value) কীভাবে যোগ হবে সেটা MOD ঠিক করে:

MOD	মানে	Displacement
00	Memory address (no displacement)	না
01	Memory address (8-bit displacement)	হ্যাঁ, 8-bit
10	Memory address (16-bit displacement)	হ্যাঁ, 16-bit
11	Register addressing	Memory না, register

৩. এখন R/M field কী করে?

যখন MOD = 00, 01, বা 10, তখন R/M 3-bit মান দিয়ে বলে দেয়

কোন register combination ব্যবহার হবে memory address তৈরি করতে।

Table 4-4 (16-bit mode R/M combinations)

R/M Bits	Effective Address (EA)	Description
000	[BX + SI]	Base + Index
001	[BX + DI]	Base + Index
010	[BP + SI]	Base + Index
011	[BP + DI]	Base + Index
100	[SI]	Index only
101	[DI]	Index only
110	[disp16] (MOD=00 only) / [BP + disp]	Direct address (special case)
111	[BX]	Base only

TABLE 4–4 16-bit R/M memory-addressing modes.

<i>R/M Code</i>	<i>Addressing Mode</i>
000	DS:[BX+SI]
001	DS:[BX+DI]
010	SS:[BP+SI]
011	SS:[BP+DI]
100	DS:[SI]
101	DS:[DI]
110	SS:[BP]*
111	DS:[BX]

*Note: See text section, Special Addressing Mode.

◆ **বিশেষ নিয়ম:**

যদি MOD = 00 এবং R/M = 110, তাহলে এটা [disp16] মানে সরাসরি address (Direct Memory Address)।

অন্য সবক্ষেত্রে এটা register combination নির্দেশ করে।

✳️ ৪. উদাহরণে বোঝা যাক

🧠 **Example 1:** `MOV DL, [DI]`

Assembly Instruction:

`MOV DL, [DI]`

◆ **Step 1 — Identify fields:**

Field	মান
Operation	MOV
REG	DL (destination register)
R/M	[DI] (source operand memory)
Displacement	নেই
Data size	Byte (W = 0)

Opcode				D	W	MOD		REG	R/M
1	0	0	0	1	0	1	0	1	0

Opcode = MOV

D = Transfer to register (REG)

W = Byte

MOD = No displacement

REG = DL

R/M = DS:[DI]

FIGURE 4–5 A MOV DL,[DI] instruction converted to its machine language form.

◆ Step 2 — Binary Breakdown:

অংশ	মানে
Opcode	100010 (MOV)
D = 1	Data থাকে REG ফিল্ড (DL)
W = 0	Byte move
MOD = 00	No displacement
REG = 010	DL register
R/M = 101	[DI] memory

◆ Step 3 — Combine bits:

Opcode byte (MOV, D=1, W=0):

10001010b = 8A (Hex)

MOD-REG-R/M byte:

00 010 101b = 00010101b = 15h

👉 Final machine code:

8A 15h

অর্থাৎ MOV DL, [DI] = **8A15h**

🧠 Example 2: `MOV DL, [DI + 2]`

এখানে **8-bit displacement** আছে।

MOD	REG	R/M	Displacement
01	DL	[DI]	+02

তাহলে —

- Opcode = 8A
- MOD-REG-R/M byte = 01 010 101 → 01010101b = 55h
- Displacement = 02h

👉 Final machine code:

8A 55 02h

অর্থাৎ `MOV DL, [DI+2]` = **8A5502h**

🧠 Example 3: `MOV DL, [DI + 1000H]`

এবার **16-bit displacement** ব্যবহার হচ্ছে। tai MOD 10

MOD	REG	R/M	Displacement
10	DL	[DI]	+1000H

তাহলে —

- Opcode = 8A
- MOD-REG-R/M byte = 10 010 101 → 10010101b = 95h
- Displacement = 1000H (little-endian: 00 10h)

👉 Final machine code:

8A 95 00 10h

অর্থাৎ `MOV DL, [DI+1000H]` = **8A950010h**

✳️ ৫. Short Summary

Instruction	MOD	R/M	Displacement	Machine Code
MOV DL, [DI]	00	101	None	8A15
MOV DL, [DI+2]	01	101	02	8A55 02
MOV DL, [DI+1000H]	10	101	1000H	8A95 00 10

✳️ ৬. Bonus — কেন এই ফরম্যাট দরকার?

কারণ CPU এর জন্য সবকিছু binary form এ থাকতে হয়।

Assembler শুধু এই binary form (machine code) তৈরি করে দেয়।

আর MOD-REG-R/M byte এর মাধ্যমে CPU বোঝে,

👉 “আমি কোন register থেকে data নিচ্ছি?”

👉 “Memory address হিসাব করতে কোন register combination ব্যবহার করবো?”

👉 “Displacement আছে কিনা?”

Special Addressing Mode theke 4.2 porjonto apatoto pori nai

4.2:

✳️ Stack Memory (সংক্ষেপে রিভিউ)

- Stack হলো একটি LIFO (Last In First Out) টাইপ মেমরি।
👉 মানে — যে ডেটা শেষে ঢুকবে, সেটা সবার আগে বের হবে।
- Stack এর ঠিকানা সবসময় SS:SP (Stack Segment : Stack Pointer) রেজিস্টার দিয়ে নিয়ন্ত্রণ করা হয়।

🧠 PUSH Instruction (Stack-এ ডেটা রাখা)

● কাজ:

PUSH কোনো রেজিস্টার, মেমরি লোকেশন, বা ইমিডিয়েট ডেটা stack-এর ভিতরে রেখে দেয়।

◆ মূল কথা:

- 8086/8088 → PUSH করে 2 bytes (16-bit)
- 80386 এবং তার উপরে → PUSH করতে পারে 2 বা 4 bytes (16-bit বা 32-bit)

■ PUSH করার সময় কী হয়:

ধরা যাক তুমি লিখলে:

PUSH AX

→ তাহলে এই ধাপে ধাপে কাজ হয়:

1. Stack pointer (SP) 2 কমে যায় (কারণ 2 bytes জায়গা নিতে হবে)
2. AX রেজিস্টারের মান Stack-এ চলে যায়।
 - আগে উচ্চ 8-বিট (AH) যায় $SP - 1$ ঠিকানায়
 - তারপর নিম্ন 8-বিট (AL) যায় $SP - 2$ ঠিকানায়
3. SP আপডেট হয়ে যায় নতুন অবস্থানে (মানে, উপরে উঠে গেছে stack memory তে)।

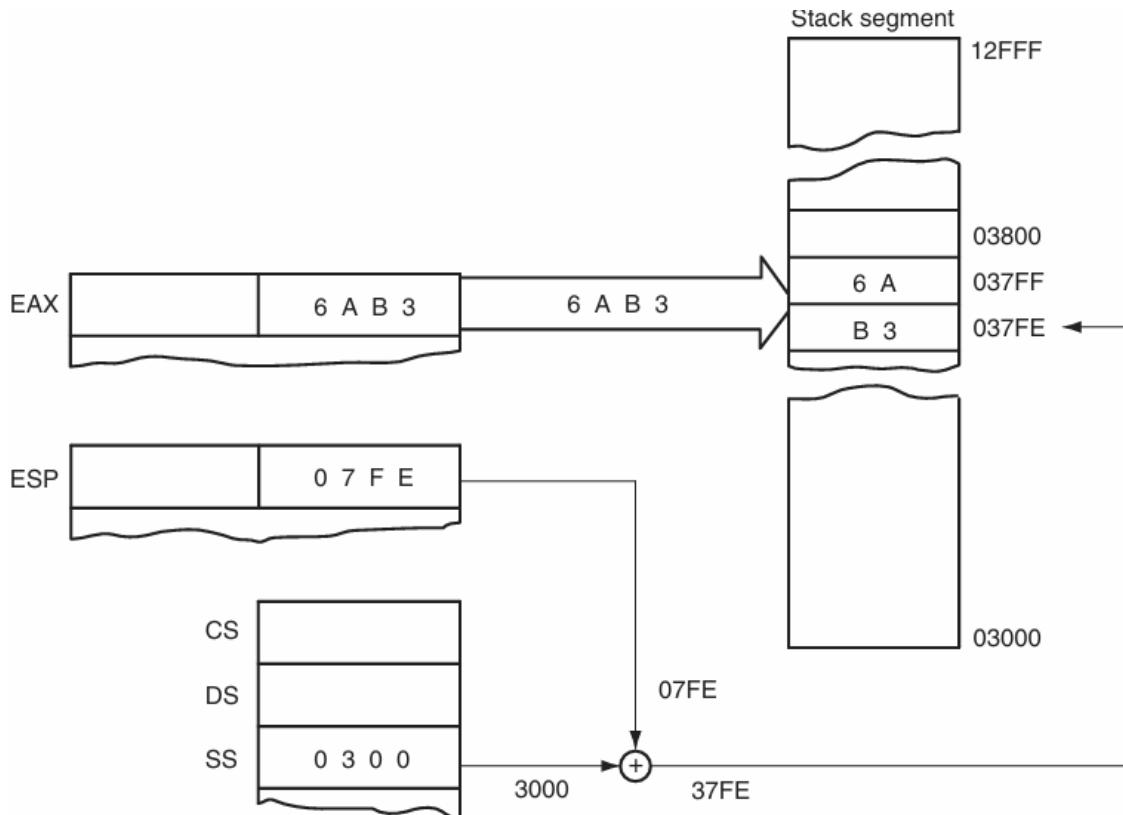


FIGURE 4–13 The effect of the **PUSH AX** instruction on **ESP** and stack memory locations **37FFH** and **37FEH**. This instruction is shown at the point after execution.

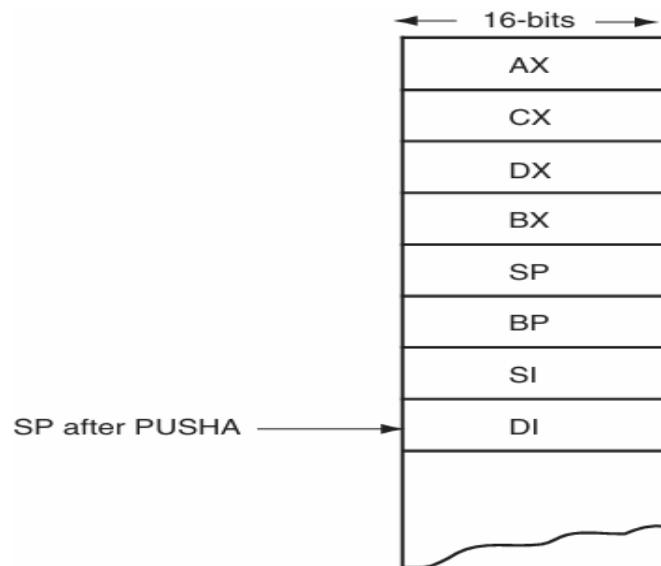
উদাহরণস্বরূপ:

Action	Address	Data
Before	SP = 2000H	—
After PUSH AX	SP = 1FFEH	AX → [1FFFH:1FFEH]

❖ PUSHF / PUSHFD

- **PUSHF** → Flag register কে Stack-এ রাখে (16-bit)
- **PUSHFD** → **EFLAGS** (32-bit) কে Stack-এ রাখে

FIGURE 4–14 The operation of the **PUSHA** instruction, showing the location and order of stack data.



✿ PUSHA / PUSHAD

এগুলো অনেক গুরুত্বপূর্ণ।

এরা একসাথে সব রেজিস্টার stack-এ রাখে।

Instruction	Registers pushed	Total bytes used
PUSHA	AX, CX, DX, BX, SP, BP, SI, DI	16 bytes
PUSHAD	EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI	32 bytes

● বিশেষ দ্রষ্টব্য:

PUSHA বা PUSHAD — এরা SP/ESP যেভাবে ছিল ঠিক সেভাবেই stack এ রাখে (execution-এর আগে SP-এর মান)।

🧠 POP Instruction (Stack থেকে ডেটা আনা)

POP হলো PUSH-এর বিপরীত।

- এটি Stack থেকে ডেটা বের করে কোনো রেজিস্টার বা মেমরিতে রাখে।
- প্রতিবার POP করলে $SP + 2$ (বা $SP + 4$) হয়, মানে stack নিচের দিকে খোলে।

✿ PUSH করার সময় Stack-এর দিক

Stack সবসময় উপরের দিক থেকে নিচের দিকে বাড়ে
মানে ঠিকানাগুলো কমে (decrement হয়)।

■ ধরো Stack নিচের মতো আছে:

Top ↑
2000
1FFE ← SP (Stack top)
1FFC
...
যখন তুমি PUSH করো, SP কমে (যেমন 1FFC হয়), আর ডেটা সেখানে রাখা হয়।

▣ সারসংক্ষেপ:

Instruction	কাজ	Bytes	Direction
PUSH AX	AX stack-এ যায়	2	SP ↓
PUSH imm8	Immediate data stack-এ যায়	2	SP ↓
PUSHA	সব রেজিস্টার stack-এ যায়	16	SP ↓
PUSHF	Flags stack-এ যায়	2	SP ↓
POP AX	AX-এ stack থেকে ডেটা আনে	2	SP ↑

🧠 POP Instruction — মূল ধারণা

POP মানে হলো:

Stack থেকে ডেটা বের করে কোনো register বা memory location-এ ফেরত দেওয়া।

❖ PUSH যা করে stack-এ রাখে, POP সেটাই stack থেকে বের করে আনে।

✿ POP কিভাবে কাজ করে

ধৰা যাক, Stack Segment (SS) এবং Stack Pointer (SP) মিলে Stack-এর অবস্থান দেখায়। Stack LIFO (Last In, First Out) বলে, **শেষে ঢোকানো ডেটা সবার আগে বের হয়।**

◆ উদাহরণ:

POP BX

এই ইনস্ট্রুকশন মানে — Stack-এর উপর থেকে 2 byte ডেটা নিয়ে তা BX রেজিস্টারে রাখবে।

◆ ধাপে ধাপে কী হয়:

1. SP দ্বারা নির্দেশিত অ্যাড্রেস থেকে প্রথম byte নেওয়া হয় → এটা যায় BL এ।
2. SP+1 থেকে দ্বিতীয় byte নেওয়া হয় → এটা যায় BH এ।
3. সব ডেটা নেওয়ার পর **SP 2 বাড়ে**, কারণ এখন Stack থেকে 2 byte ফাঁকা হয়েছে।

উদাহরণ:

ধাপ	ঠিকানা	কাজ
শুরুতে	SP = 1FFEH	Stack top = [1FFEH]=Low byte, [1FFFH]=High byte
POP BX	BX = [1FFFH][1FFEH], SP =	
শেষে	2000H	

POP-এর মাধ্যমে কী কী pop করা যায়

POP Type	কী কাজ করে	Data Size
POP reg16	16-bit register এ stack data নেয়	2 bytes
POP reg32	32-bit register এ নেয়	4 bytes
POP mem16	Memory location এ নেয়	2 bytes
POP mem32	Memory location এ নেয়	4 bytes
POP seg	Segment register এ নেয় (কিন্তু CS এ নেয়া যায় না X)	2 bytes
POPF	Flag register এ নেয়	2 bytes
POPFD	Extended flag register (EFLAGS) এ নেয়	4 bytes
POPA	সব 16-bit register পুনরায় লোড করে	16 bytes
POPAD	সব 32-bit register পুনরায় লোড করে	32 bytes

✿ POPA / POPAD — (সব রেজিস্টার একসাথে Pop)

✿ POPA:

এটা ঠিক PUSHA-এর উল্লেখ কাজ করে।

PUSHA: AX, CX, DX, BX, SP, BP, SI, DI → Stack-এ রাখে

POPA: Stack থেকে নেয় DI, SI, BP, SP, BX, DX, CX, AX ক্রমে

অর্থাৎ যে ক্রমে PUSHA রেখেছিল, POPA সেই ক্রমের বিপরীতে ফিরিয়ে আনে, যাতে রেজিস্টারগুলোর পুরোনো মান ঠিকভাবে ফিরে আসে।

✿ POPAD:

এটা 80386 এবং তার উপরের জন্য — 32-bit register গুলো ফেরত আনে।

POPAD 32 byte stack থেকে নেয় এবং EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI restore করে।

⚠ গুরুত্বপূর্ণ নিয়ম: POP CS নিষিদ্ধ ✗

CS (Code Segment) পরিবর্তন করলে পরবর্তী ইনস্ট্রাকশনের ঠিকানা ভুল হয়ে যেতে পারে।
তাই POP CS দিলে মাইক্রোপ্রসেসর বিপ্রান্ত হতে পারে → **অনিদিশ্য** (unpredictable) ফলাফল দেয়।

সেই জন্য এই ইনস্ট্রাকশন বৈধ নয়।

✿ POP-এর সময় Stack কীভাবে বদলায়

ধরা যাক SP = 1FFEH, এবং Stack-এ 2 byte আছে।

Before POP BX:

Address	Data
1FFFH	11H ← High byte
1FEFH	22H ← Low byte
SP = 1FFEH	

→ POP BX চালালে:

- $BL \leftarrow [1FFE] = 22H$
- $BH \leftarrow [1FFF] = 11H$
- $SP \leftarrow SP + 2 = 2000H$

After POP BX:

$BX = 1122H$

$SP = 2000H$

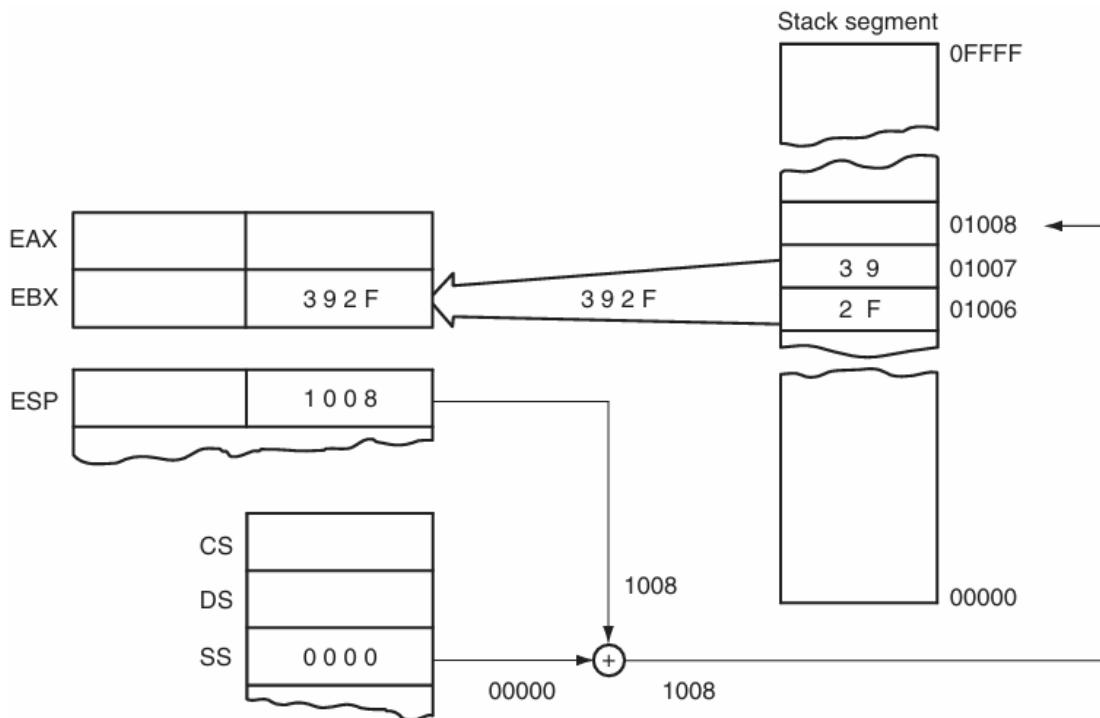


FIGURE 4–15 The POP BX instruction, showing how data are removed from the stack. This instruction is shown after execution.

✓ সারসংক্ষেপে তুলনা:

Instruction	কাজ	Stack দিক	SP পরিবর্তন
PUSH AX	AX Stack এ যাও	↓ নিচে নামে	SP - 2
POP AX	AX Stack থেকে আসে	↑ উপরে ওঠে	SP + 2
PUSHA	সব 16-bit register stack এ	↓	SP - 16
POPA	সব 16-bit register stack থেকে	↑	SP + 16
PUSHF	Flags stack এ	↓	SP - 2
POPF	Flags stack থেকে	↑	SP + 2



Stack কীভাবে শুরু হয় (Initializing the Stack)

যখন কোনো প্রোগ্রাম চলে, Stack একটা নির্দিষ্ট মেমোরি এরিয়াতে তৈরি হয়।
এই Stack ব্যবহার করে ডেটা অস্থায়ীভাবে রাখা হয় — যেমন PUSH ও POP ইনস্ট্রাকশনের
সময়।

কিন্তু Stack ব্যবহারের আগে, আমাদের দুটো জিনিস ঠিক করে দিতে হয়

1. SS (Stack Segment register) → Stack কোথায় আছে
2. SP (Stack Pointer register) → Stack-এর উপরের (top) অবস্থান



Stack Segment (SS) সেট করা

ধরা যাক, তুমি Stack রাখতে চাও এই জায়গায়:

10000H থেকে 1FFFFH পর্যন্ত

এটা মানে Stack-এর জন্য 64KB মেমোরি রিজার্ভ করা হয়েছে।

তাহলে Stack Segment Register SS-এ রাখতে হবে:

SS = 1000H

কারণ real mode addressing-এ Segment register-এর ডানদিকে 0H যোগ হয় →

1000H × 10H = 10000H (অর্থাৎ Stack-এর শুরু)



Stack Pointer (SP) সেট করা

এখন Stack Pointer বলে দেবে Stack-এর শুরু অবস্থান (top) কোথায়।

যদি Stack-এর উপরের দিক (top) রাখতে চাও segment-এর একদম শেষ প্রান্তে (1FFFFH),
তাহলে SP = 0000H দাও।

কারণ stack উল্লেখ দিকে বাড়ে (downward grows)
অর্থাৎ PUSH দিলে SP কমে, POP দিলে SP বাড়ে।

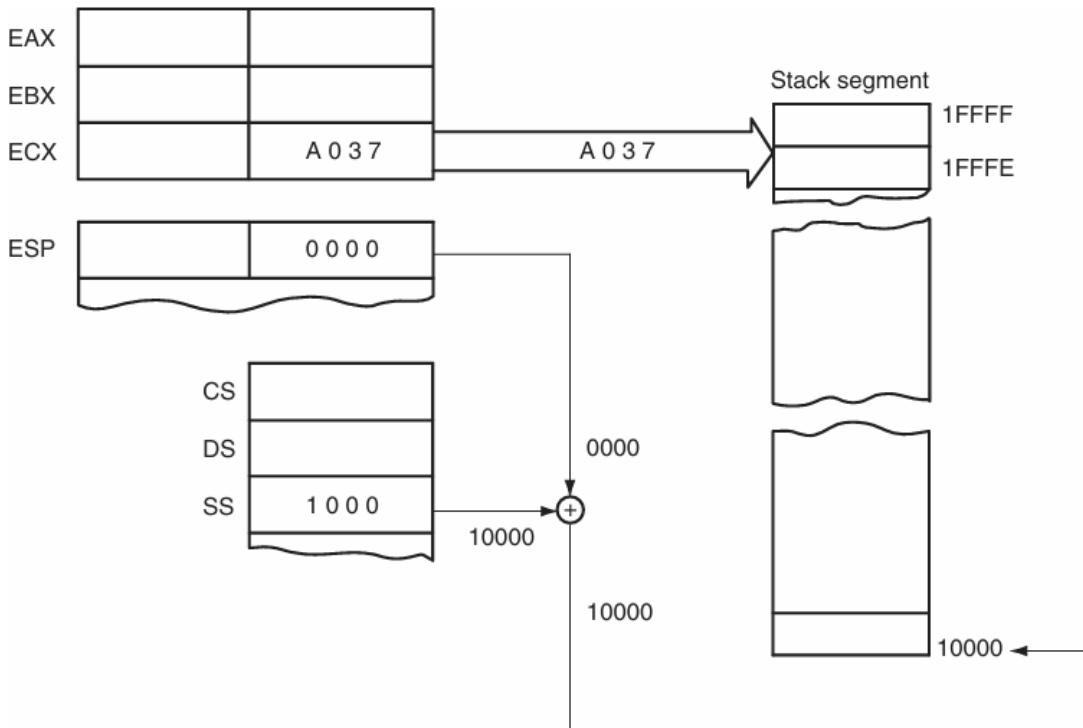


FIGURE 4–16 The **PUSH CX** instruction, showing the cyclical nature of the stack segment. This instruction is shown just before execution, to illustrate that the stack bottom is contiguous to the top.

উদাহরণ:

Stack segment address: 10000H–1FFFFH

Register	Value	অর্থ
SS	1000H	Stack segment শুরু 1000H
SP	0000H	Stack-এর শীর্ষ (top) 1FFFFH

এখন যদি তুমি **PUSH CX** দাও,

- তাহলে CX-এর ডেটা **1FFFEH** এবং **1FFFH** ঠিকানায় যাবে।
- SP কমে যাবে **0000H → FFFEH**।

* আরেকটা উদাহরণ (SP manually set)

যদি SP = 1000H দাও,
তাহলে Stack শুরু হবে ঠিকানায়:

Physical Address = SS×10H + SP = 1000H×10H + 1000H = 11000H

এখন Stack top হবে 10FFFH।

Stack cyclic nature

Stack segment আসলে চক্রাকার (cyclic) অর্থাৎ:

যদি Stack নিচে নেমে শেষ প্রান্তে পৌঁছায়, তাহলে আবার segment-এর উপরের দিক থেকে শুরু হয়।

Stack না দিলে কী হয়?

যদি তুমি Stack segment না দাও,
linker প্রোগ্রাম একটা warning দেবে।

তবে 128 bytes-এর কম stack ব্যবহার করলে সমস্যা হয় না কারণ DOS নিজে থেকেই 128 bytes stack দেয়।

এই Stack থাকে PSP (Program Segment Prefix) নামের জায়গায়।

কিন্তু যদি বেশি Stack ব্যবহার করো (128 bytes এর বেশি) এবং নির্দিষ্ট stack না বানাও,
তাহলে Stack PSP-র ভেতরের গুরুত্বপূর্ণ ডেটা নষ্ট করে দেবে —

👉 এতে প্রোগ্রাম crash করবে।



Special Case: TINY Model

TINY memory model ব্যবহার করলে Stack segment আলাদা করে না বানিয়ে,
প্রোগ্রামের segment-এর শেষে Stack রাখে।

এতে Stack-এর জন্য বেশি জায়গা পাওয়া যায়।



সারসংক্ষেপে:

ধাপ	কাজ	উদাহরণ
1	Stack segment ঠিক করো	SS = 1000H

ধাপ	কাজ	উদাহরণ
২	Stack pointer ঠিক করো	SP = 0000H
৩	PUSH দিলে SP ↓ ২ কমে	SP = FFFEH
৪	POP দিলে SP ↑ ২ বাড়ে	SP = 0000H
৫	MASM .STACK ব্যবহার করলে SS/SP auto set হয়	.STACK 200H

অবশ্যই ☺ — নিচে আমি LEA instruction পুরোপুরি সহজ ভাষায় বাংলায় ব্যাখ্যা করছি, যেন তুমি একবারে বুঝে ফেলো:

✳️ ১ LEA মানে কী?

LEA (Load Effective Address) মানে হলো — “ডেটা কোথায় আছে, সেই ঠিকানাটা (address) রেজিস্টারে লোড করা।”

অর্থাৎ, এটা data না নিয়ে, data er address নেয়।

✳️ ২ উদাহরণ দিয়ে বোঝা যাক:

ধরো, আমাদের কাছে একটা ভ্যারিয়েবল আছে:

NUMB DW 1234H

এখন যদি লিখি:

MOV AX, NUMB

👉 তাহলে AX = 1234H (মানে ডেটা লোড হলো)।

কিন্তু যদি লিখি:

LEA AX, NUMB

👉 তাহলে AX = NUMB এর offset address (মানে ডেটা নয়, বরং NUMB কোথায় আছে সেই জায়গার ঠিকানা লোড হলো)।

৭ LEA vs MOV with OFFSET

LEA অনেকটা এরকম কাজ করে:

LEA BX, LIST

এবং নিচের নির্দেশনাটাও একই কাজ করে:

MOV BX, OFFSET LIST

👉 দুটোই LIST এর offset address BX-এ লোড করে।

তবে পার্থক্য হলো:

- OFFSET → **গুরু simple operand** (যেমন LIST) এর জন্য কাজ করে।
- LEA → **complex address expression** (যেমন [BX+DI], [SI+5]) এর জন্য কাজ করে।

৮ Execution time পার্থক্য

Instruction	কে address হিসাব করে	Clock Cycle
MOV BX, OFFSET LIST	Assembler	দ্রুত (1 cycle)
LEA BX, LIST	Processor	ধীর (2 cycles)

✿ অর্থাৎ OFFSET ডিরেক্টিভ দ্রুত কারণ assembler compile করার সময়েই address বের করে নেয়।

কিন্তু LEA তে CPU নিজেই runtime এ হিসাব করে।

৯ Complex addressing এর ক্ষেত্রে LEA দরকার

যেমন:

LEA SI, [BX + DI]

👉 এটি BX এবং DI রেজিস্টার যোগ করে, তাদের যোগফল (effective address) SI তে রাখে। (এটা modulo 64K sum, মানে 16-bit এর বেশি হলে carry বাদ যায়।)

উদাহরণ:

BX	DI	ফলাফল (SI তে)
1000H	2000H	3000H
1000H	FF00H	OF00H (কারণ carry বাদ গেছে)

ডিটুলনা করে দেখো:

Instruction	কাজ
MOV BX, [DI]	DI দ্বারা নির্দেশিত মেমরি লোকেশনের data BX-এ নেয়
LEA BX, [DI]	DI-এর address (offset) BX-এ নেয়
MOV BX, DI	DI-এর value BX-এ কপি করে

সারসংক্ষেপে:

নির্দেশনা	কাজ
LEA reg, variable	variable এর address রেজিস্টারে নেয়
MOV reg, variable	variable এর value রেজিস্টারে নেয়
MOV reg, OFFSET variable	variable এর address রেজিস্টারে নেয় (LEA এর সমান, কিন্তু দ্রুত)

LDS, LES, LFS, LGS, LSS

✳️ ১মূল ধারণা

এই নির্দেশনাগুলোর কাজ হলো —

👉 একসাথে দুইটা জিনিস লোড করা:

- একটা **সাধারণ register** (যেমন BX, SI, SP, ইত্যাদি) – এতে offset address লোড হয়
- একটা **segment register** (যেমন DS, ES, FS, GS, SS) – এতে segment address লোড হয়

অর্থাৎ, এগুলো **একটা "far address"** (segment:offset address pair) মেমরি থেকে নিয়ে আসে।

২ Format & Memory Structure

ধরো, নির্দেশনা হলো:

LDS BX, [DI]

👉 এখানে DI দ্বারা নির্দেশিত মেমরি লোকেশনে একটা 32-bit address (4 bytes) আছে।
এই address দুই অংশে বিভক্ত:

Memory Bytes	কী বোঝায়	কোথায় লোড হয়
প্রথম 2 bytes	Offset	BX রেজিস্টারে
পরের 2 bytes	Segment	DS রেজিস্টারে

- অর্থাৎ, 8 বাইটের মধ্যে প্রথম অংশ offset, পরের অংশ segment!

৩ প্রতিটি নির্দেশনার অর্থ

Instruction	কী করে
LDS reg, mem	মেমরি থেকে offset → reg, segment → DS
LES reg, mem	মেমরি থেকে offset → reg, segment → ES
LFS reg, mem	মেমরি থেকে offset → reg, segment → FS (80386+)
LGS reg, mem	মেমরি থেকে offset → reg, segment → GS (80386+)
LSS reg, mem	মেমরি থেকে offset → reg, segment → SS (80386+)

৪ উদাহরণ: LDS BX, [DI]

ধরা যাক মেমরিতে নিচের মতো ডেটা আছে (DS segment-এ):

Address	Value
DS:1000H	50H (offset low byte)
DS:1001H	20H (offset high byte) → offset = 2050H
DS:1002H	10H (segment low byte)

Address	Value
DS:1003H	30H (segment high byte) → segment = 3010H

এখন:

LDS BX, [1000H]

ফলাফল হবে:

BX = 2050H
DS = 3010H

অর্থাৎ, একবারে BX ও DS দুটোতেই মান চলে এসেছে!

🔗 FAR ADDRESS মানে কী?

একটা far address হচ্ছে এমন address যেখানে segment ও offset দুটোই লাগে।

যেমন 3010:2050H →

3010H হলো segment, 2050H হলো offset।

তুমি চাইলে এটি assembler দিয়ে এমনভাবে স্টোর করতে পারো:

ADDR DD FAR PTR FROG

এখানে assembler মেমরিতে FROG এর segment:offset address 8 বাইটে স্টোর করে দেবে।

⌚ ৪80386+ (Protected Mode) সংস্করণ

৮০৩৮৬ মাইক্রোপ্রসেসর ও তার পরেরগুলো (যেমন Pentium) 32-bit offset ব্যবহার করতে পারে।

তাহলে মেমরিতে address হবে $8 + 2 = 6$ বাইট (48-bit address):

অংশ	আকার	লোড হয়
Offset	4 byte	32-bit রেজিস্টারে (যেমন EBX)
Segment	2 byte	Segment রেজিস্টারে (যেমন DS)

উদাহরণ:

LDS EBX, [DI]

👉 এখন EBX-এ offset (৪ বাইট) এবং DS-এ segment (২ বাইট) যাবে।

✳️ LSS – সবচেয়ে বেশি ব্যবহৃত

LSS (Load Stack Segment) সবচেয়ে গুরুত্বপূর্ণ কারণ এটি stack পরিবর্তনের সময় ব্যবহার হয়।

🔍 ৮ সারসংক্ষেপে

Instruction	Segment Register	Use Case
LDS	DS	Data segment লোড
LES	ES	Extra segment লোড
LFS	FS	Extra data segment (80386+)
LGS	GS	Extra data segment (80386+)
LSS	SS	Stack segment পরিবর্তনের জন্য

✳️ STRING DATA TRANSFER INSTRUCTIONS – মূল ধারণা

এই নির্দেশনাগুলো এমন কাজ করে যেখানে:

- একটানা (string বা array আকারে) অনেক ডেটা ট্রান্সফার হয়।
- প্রতিটি transfer এর পর SI (source index) ও DI (destination index) স্বয়ংক্রিয়ভাবে বৃদ্ধি বা ত্রাস পায়।

✿ এই ৫টি Instruction আছে:

Instruction	কাজ
LODS	মেমরি → রেজিস্টারে ডেটা লোড করে (Load String)
STOS	রেজিস্টার → মেমরিতে ডেটা স্টোর করে (Store String)

Instruction	কাজ
MOVS	মেমরি → মেমরি কপি করে (Move String)
INS	পোর্ট → মেমরিতে ডেটা আনে (Input String)
OUTS	মেমরি → পোর্টে ডেটা পাঠায় (Output String)

▶ Direction Flag (D Flag)

এই ফ্ল্যাগ নির্ধারণ করে ডেটা ট্রান্সফার কোন দিকে হবে।

D Flag	অর্থ	কীভাবে সেট হয়
D = 0	Auto-increment → SI/DI বাড়ে	CLD ইনস্ট্রাকশন দিয়ে
D = 1	Auto-decrement → SI/DI কমে	STD ইনস্ট্রাকশন দিয়ে

- প্রতি বাইট ট্রান্সফারে $SI/DI \pm 1$
- প্রতি word ট্রান্সফারে $SI/DI \pm 2$
- প্রতি doubleword ট্রান্সফারে $SI/DI \pm 4$

⌚ SI এবং DI রেজিস্টারের কাজ

রেজিস্টার	কাজ	সেগমেন্ট
SI (Source Index)	Data segment থেকে ডেটা নেয়	DS
DI (Destination Index)	Extra segment-এ ডেটা লেখে	ES

চাইলে segment override prefix দিয়ে SI-এর segment পরিবর্তন করা যায়,
কিন্তু DI সর্বদা ES ব্যবহার করে — এটা পরিবর্তন করা যায় না।

🧠 LODS Instruction (Load String)

LODS = Load String → মেমরি থেকে রেজিস্টারে ডেটা আনে।

Instruction	কী লোড করে	কন্টেন্ট কোথায় যায়
LODSB	1 byte	AL $\leftarrow [DS:SI]$

Instruction	কী লোড করে	কন্টেন্ট কোথায় যায়
LODSW	1 word	AX \leftarrow [DS:SI]
LODSD	1 doubleword	EAX \leftarrow [DS:SI]
LODSQ	1 quadword	RAX \leftarrow [RSI] (64-bit mode)

এরপর D=0 হলে SI বাড়ে (byte \rightarrow +1, word \rightarrow +2, dword \rightarrow +4),
আর D=1 হলে SI কমে।

◆ উদাহরণ

ধরো:

DS = 1000H
SI = 1000H
Memory [1000:1000] = 34H
Memory [1000:1001] = 12H

ইনস্ট্রাকশন:

CLD ; D=0 (increment)
LODSW

- AX = 1234H
 - SI = SI + 2 (কারণ word লোড হয়েছে)
-

■ STOS Instruction (Store String)

STOS = Store String \rightarrow রেজিস্টার থেকে মেমরিতে ডেটা লেখে।

Instruction	কী স্টোর করে	কোথায় লেখে
STOSB	AL	[ES:DI]
STOSW	AX	[ES:DI]
STOSD	EAX	[ES:DI]

STOS করার পর DI বাড়ে বা কমে (D flag অনুযায়ী)।

◆ উদাহরণ: STOSB

```
CLD           ; increment mode
MOV AL, 25H
MOV DI, 2000H
STOSB
```

- 👉 AL-এর মান (25H) যাবে [ES:2000H]-এ
- 👉 DI = DI + 1

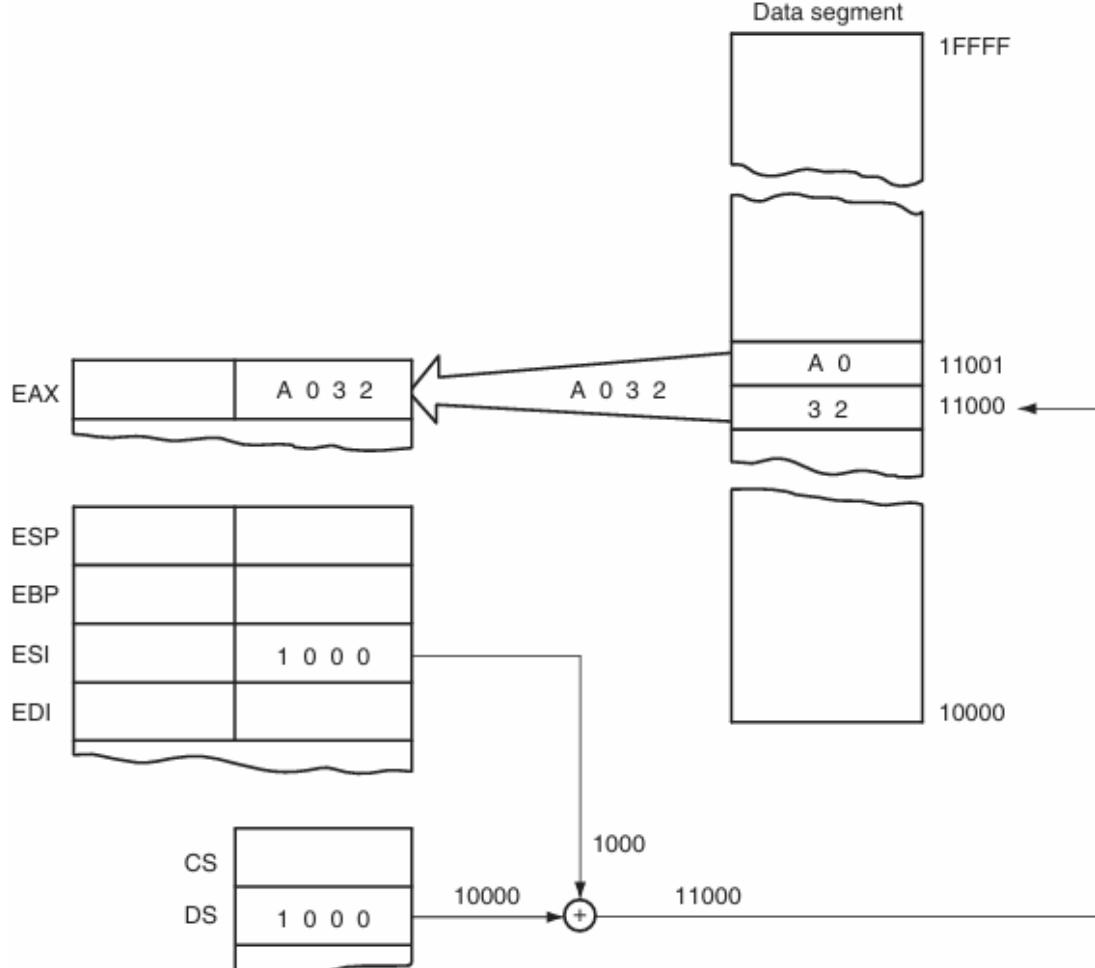


FIGURE 4–18 The operation of the LODSW instruction if DS = 1000H, D = 0, 11000H = 32, and 11001H = A0. This instruction is shown after AX is loaded from memory, but before SI increments by 2.

⌚ REP Prefix (Repeat)

REP দিয়ে STOS, MOVS, INS, OUTS বারবার চালানো যায়।

👉 প্রতিবার ইনস্ট্রুকশন চললে CX (বা RCX) 1 কমে যায়।

👉 CX = 0 হলে REP বন্ধ হয়।

Prefix	কাজ
REP	যতক্ষণ $CX \neq 0$, ইনস্ট্রুকশন চালাও
REPE / REPZ	যতক্ষণ $CX \neq 0$ এবং $ZF = 1$
REPNE / REPNZ	যতক্ষণ $CX \neq 0$ এবং $ZF = 0$

◆ উদাহরণ: REP STOSW

CLD

```
MOV AX, 0000H      ; মেমরিতে লিখব এই মান
MOV CX, 100        ; 100 বার করব
MOV DI, OFFSET Buffer
REP STOSW          ; Buffer পরিষ্কার করো
```

→ এই ইনস্ট্রুকশন 100 বার AX-এর মান (0000H) Buffer-এ লিখবে,
DI প্রতি বার +2 বাড়বে।

সংক্ষিপ্ত সারসংক্ষেপ

Instruction	Direction	কাজ	Repeat হয়?
LODS	DS:SI → AL/AX/EAX	Load	✗ না
STOS	AL/AX/EAX → ES:DI	Store	<input checked="" type="checkbox"/> হ্যাঁ
MOVS	DS:SI → ES:DI	Copy	<input checked="" type="checkbox"/> হ্যাঁ
INS	Port → ES:DI	Input	<input checked="" type="checkbox"/> হ্যাঁ
OUTS	DS:SI → Port	Output	<input checked="" type="checkbox"/> হ্যাঁ

◆ MOVS নির্দেশনার কাজ

MOVS মানে হলো Move String — এটি memory থেকে memory তে ডাটা কপি করে।

এটি একমাত্র memory-to-memory transfer নির্দেশনা যা 8086 থেকে Pentium 4 পর্যন্ত সব মাইক্রোপ্রসেসরে বৈধ।

◆ কাজের ধরন

MOVS নির্দেশনা ডাটাকে **SI (Source Index)** দ্বারা নির্দেশিত **data segment (DS)** থেকে নেয় এবং **DI (Destination Index)** দ্বারা নির্দেশিত **extra segment (ES)** এ রাখে।

অর্থাৎ:

ES:[DI] \leftarrow **DS:[SI]**

তারপর **Direction Flag (D)** এর উপর নির্ভর করে **SI** ও **DI** বাড়ে বা কমে।

◆ MOVS-এর বিভিন্ন রূপ (Table 4-14 অনুযায়ী)

Assembly নির্দেশনা	কাজ	SI/DI পরিবর্তন
MOVSB	এক বাইট কপি করে	± 1
MOVSW	এক ওয়ার্ড (2 বাইট) কপি করে	± 2
MOVSD	এক ডাবল ওয়ার্ড (4 বাইট) কপি করে	± 4
MOVSQ (64-bit mode)	এক কোয়াডওয়ার্ড (8 বাইট) কপি করে	± 8

TABLE 4-14 Forms of the MOVS instruction.

Assembly Language	Operation
MOVSB	$ES:[DI] = DS:[SI]; DI = DI \pm 1; SI = SI \pm 1$ (byte transferred)
MOVSW	$ES:[DI] = DS:[SI]; DI = DI \pm 2; SI = SI \pm 2$ (word transferred)
MOVSD	$ES:[DI] = DS:[SI]; DI = DI \pm 4; SI = SI \pm 4$ (doubleword transferred)
MOVSQ	$[RDI] = [RSI]; RDI = RDI \pm 8; RSI = RSI \pm 8$ (64-bit mode)
MOVS BYTE1, BYTE2	$ES:[DI] = DS:[SI]; DI = DI \pm 1; SI = SI \pm 1$ (byte transferred if BYTE1 and BYTE2 are bytes)
MOVS WORD1,WORD2	$ES:[DI] = DS:[SI]; DI = DI \pm 2; SI = SI \pm 2$ (word transferred if WORD1 and WORD2 are words)
MOVS TED,FRED	$ES:[DI] = DS:[SI]; DI = DI \pm 4; SI = SI \pm 4$ (doubleword transferred if TED and FRED are doublewords)

◆ Direction Flag (D) অনুযায়ী কাজ

- যদি $D = 0$ (CLD ব্যবহার করা হয়):
তাহলে SI ও DI increment হয় (সামনের দিকে যায়)
- যদি $D = 1$ (STD ব্যবহার করা হয়):
তাহলে SI ও DI decrement হয় (পিছনের দিকে যায়)

◆ উদাহরণ (MOVSD ব্যবহার করে দুটি ব্লক কপি করা)

ধরা যাক, তোমার দুটি ডেটা ব্লক আছে:

blockA (source) এবং blockB (destination)

তুমি চাও $\text{blockA} \rightarrow \text{blockB}$ তে কপি করতে।

```
PUSH DS          ; DS এর মান সংরক্ষণ  
POP ES          ; ES = DS (কারণ দুটোই একই segment)  
CLD            ; Direction flag clear (increment mode)  
MOV CX, blockSize ; blockSize = কয়টা doubleword কপি হবে  
MOV ESI, offset blockA  
MOV EDI, offset blockB  
REP MOVSD       ; Repeat করে blockSize সংখ্যক doubleword কপি
```

👉 এখানে REP MOVSD মানে —

CX ঘট বড়, ততবার MOVSD চলবে এবং প্রতিবার ESI ও EDI 8 বাইট করে বাড়বে।

🧠 গুরুত্বপূর্ণ পয়েন্ট

1. MOVS হলো একমাত্র instruction যা memory থেকে memory তে ডাটা সরাসরি নিতে পারে।
2. DS:SI হলো source address, আর ES:DI হলো destination address।
3. REP MOVSx দিয়ে bulk copy (block transfer) করা হয়।
4. CLD ও STD দিয়ে দিক নিয়ন্ত্রণ করা হয়।

◆ INS (Input String Instruction)

🧠 মূল কাজ:

INS নির্দেশনা বাইরের I/O ডিভাইস থেকে ডাটা মেমরিতে আনে।

অর্থাৎ:

`ES:[DI] ← I/O [DX]`

এখানে —

- `DX → I/O` ডিভাইসের অ্যাড্রেস ধরে
 - `DI →` মেমরির গন্তব্য (destination) ঠিকানা নির্দেশ করে (Extra Segment এ)
 - `ES:[DI] →` যেখানে ডাটা রাখা হবে
 - প্রতিবার ইনপুটের পর `DI` বাড়ে বা কমে (Direction Flag অনুযায়ী)
-

✿ INS এর ধরন (Table 4–15 অনুযায়ী)

Assembly নির্দেশনা	কাজ	DI পরিবর্তন
INSB	8-bit (1 byte) ইনপুট	± 1
INSW	16-bit (1 word) ইনপুট	± 2
INSD	32-bit (1 doubleword) ইনপুট	± 4

⚠ নোট: INS নির্দেশনাগুলো 8086/8088 প্রসেসরে নেই। এগুলো 80286 এবং এর পরের প্রসেসরগুলোতে ব্যবহার হয়।

TABLE 4–15 Forms of the INS instruction.

Assembly Language	Operation
INSB	<code>ES:[DI] = [DX]; DI = DI ± 1</code> (byte transferred)
INSW	<code>ES:[DI] = [DX]; DI = DI ± 2</code> (word transferred)
INSD	<code>ES:[DI] = [DX]; DI = DI ± 4</code> (doubleword transferred)
INS LIST	<code>ES:[DI] = [DX]; DI = DI ± 1</code> (if LIST is a byte)
INS DATA4	<code>ES:[DI] = [DX]; DI = DI ± 2</code> (if DATA4 is a word)
INS DATA5	<code>ES:[DI] = [DX]; DI = DI ± 4</code> (if DATA5 is a doubleword)

☛ REP ব্যবহার:

যদি তুমি `REP INSB` লেখো, তাহলে CPU একসাথে অনেকগুলো byte ইনপুট নেবে —
প্রতিবার `CX` কমবে, যতক্ষণ না `CX = 0` হয়।

* উদাহরণ:

ধরা যাক,
একটা I/O ডিভাইসের অ্যাড্রেস **03ACH**,
এবং তুমি চাও ৫০ বাইট ডাটা মেমরিতে ইনপুট করতে।

```
MOV DX, 03ACH      ; I/O device address
MOV DI, OFFSET LISTS ; গন্তব্য মেমরি অ্যারে
MOV CX, 50          ; ইনপুট হবে ৫০ বাইট
CLD                ; Direction flag clear (increment mode)
REP INSB           ; ৫০ বাইট ইনপুট নিয়ে মেমরিতে রাখবে
```

◆ OUTS (Output String Instruction)



OUTS নির্দেশনা মেমরি থেকে I/O ডিভাইসে ডাটা পাঠায়।

অর্থাৎ:

I/O [DX] ← DS:[SI]

এখানে —

- SI → মেমরির সোর্স ঠিকানা (Data Segment)
- DX → I/O ডিভাইসের অ্যাড্রেস
- DS:[SI] → যেখান থেকে ডাটা নেওয়া হবে
- প্রতিবার আউটপুটের পর SI বাড়ে বা কমে (Direction Flag অনুযায়ী)

⚙️ OUTS এর ধরন (Table 4-16 অনুযায়ী)

Assembly নির্দেশনা	কাজ	SI পরিবর্তন
OUTSB	8-bit (1 byte) আউটপুট	±1
OUTSW	16-bit (1 word) আউটপুট	±2
OUTSD	32-bit (1 doubleword) আউটপুট	±4

⚠️ নোট: এটা� 8086/8088-এ নেই, 80286 থেকে শুরু হয়েছে।

TABLE 4-16 Forms of the OUTS instruction.

Assembly Language	Operation
OUTSB	[DX] = DS:[SI]; SI = SI ± 1 (byte transferred)
OUTSW	[DX] = DS:[SI]; SI = SI ± 2 (word transferred)
OUTSD	[DX] = DS:[SI]; SI = SI ± 4 (doubleword transferred)
OUTS DATA7	[DX] = DS:[SI]; SI = SI ± 1 (if DATA7 is a byte)
OUTS DATA8	[DX] = DS:[SI]; SI = SI ± 2 (if DATA8 is a word)
OUTS DATA9	[DX] = DS:[SI]; SI = SI ± 4 (if DATA9 is a doubleword)

▣ REP ব্যবহার:

REP OUTSB → CX বার পর্যন্ত ডাটা I/O ডিভাইসে পাঠাবে।

* উদাহরণ:

একটা ডাটা অ্যারে থেকে I/O ডিভাইসে 100 বাইট পাঠানো হচ্ছে:

```

MOV SI, OFFSET ARRAY ; ডাটা অ্যারের শুরু
MOV DX, 03ACH       ; I/O ডিভাইস অ্যাড্রেস
CLD                 ; Direction flag clear (increment mode)
MOV CX, 100          ; কাউন্টার
REP OUTSB           ; 100 বাইট পাঠাও I/O তে

```



INS ও OUTS এর পার্থক্য

দিক	INS	OUTS
কাজ	I/O → Memory	Memory → I/O
সোর্স	I/O ডিভাইস (DX)	মেমরি (DS: [SI])
গন্তব্য	মেমরি (ES: [DI])	I/O ডিভাইস (DX)
ব্যবহার	ডাটা ইনপুটের জন্য	ডাটা আউটপুটের জন্য
REP ব্যবহার	একাধিক ইনপুট নেয়	একাধিক আউটপুট পাঠায়

* XCHG (Exchange) Instruction

পূর্ণরূপ: Exchange

অর্থ: দুইটি রেজিস্টার বা একটি রেজিস্টার ও মেমরি লোকেশনের মধ্যে ডেটা অদলবদল করে।

⚙️ কাজ:

XCHG ইনস্ট্রুকশন এক রেজিস্টারের কনটেন্ট আরেক রেজিস্টার বা মেমরি লোকেশনের কনটেন্টের সাথে অদলবদল (swap) করে।

👉 কিন্তু এটি কখনোই segment registers বা memory-to-memory ডেটা এক্সচেঞ্চ করতে পারে না।

(মানে: রেজিস্টার \leftrightarrow রেজিস্টার , রেজিস্টার \leftrightarrow মেমরি , মেমরি \leftrightarrow মেমরি)

ডেটার সাইজ:

- 8-bit (byte)
- 16-bit (word)
- 32-bit (doubleword, 80386 এবং পরবর্তী প্রসেসরে)
- 64-bit (64-bit মোডে)

💻 উদাহরণসমূহ (Table 4–17 অনুযায়ী):

TABLE 4–17 Forms of the XCHG instruction.

Assembly Language	Operation
XCHG AL,CL	Exchanges the contents of AL with CL
XCHG CX,BP	Exchanges the contents of CX with BP
XCHG EDX,ESI	Exchanges the contents of EDX with ESI
XCHG AL,DATA2	Exchanges the contents of AL with data segment memory location DATA2
XCHG RBX,RCX	Exchange the contents of RBX with RCX (64-bit mode)

Instruction	কাজ
XCHG AL, BL	AL \leftrightarrow BL
XCHG AX, BX	AX \leftrightarrow BX
XCHG EAX, EBX	EAX \leftrightarrow EBX (32-bit)
XCHG AL, [DI]	AL \leftrightarrow memory location [DI]
XCHG [DI], AL	একই কাজ — assembler উভয়কেই একইভাবে দেখে

⚡ বিশেষ দিক:

- যদি AX (বা EAX) রেজিস্টার ব্যবহার করা হয়, তাহলে ইনস্ট্রাকশনটি সবচেয়ে দ্রুত (efficient) হয় এবং মাত্র 1 byte মেমরি নেয়।
- অন্যান্য XCHG ইনস্ট্রাকশন (যেখানে addressing mode ব্যবহৃত হয়) 2 বা ততোধিক বাইট মেমরি নেয়।

🚫 যা করা যায় না:

- Memory-to-memory swap ✗
যেমন: XCHG [SI], [DI] → invalid
- Segment register swap ✗
যেমন: XCHG DS, AX → invalid

💡 একটি উদাহরণ:

```
MOV AX, 1234h  
MOV BX, 5678h  
XCHG AX, BX
```

ফলাফল:

AX = 5678h
BX = 1234h