

**🔹 Unconditional Jump (JMP)**

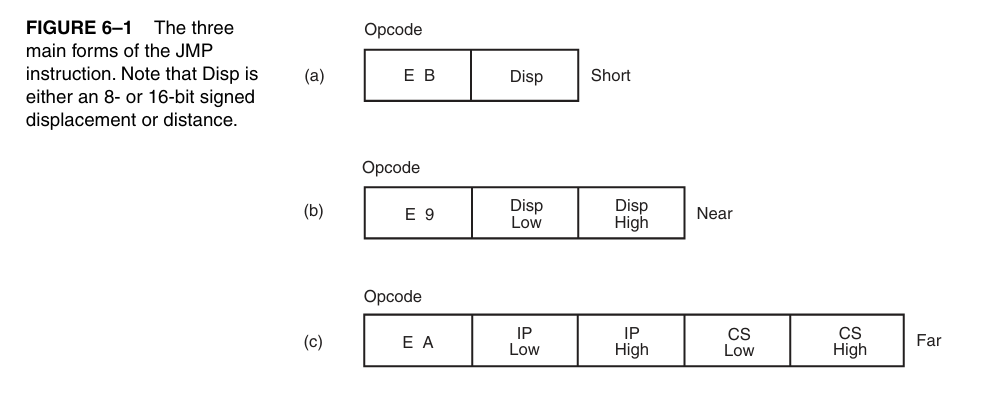
**JMP** মানে হলো “লাফ দেওয়া” — অর্থাৎ **program control** এক জায়গা থেকে আরেক জায়গায় চলে যায় **কোনো শর্ত ছাড়াই**।

👉 অন্যভাবে বললে, যখন কোনো প্রোগ্রামে JMP লেখা থাকে, তখন CPU সেই জায়গা থেকে সরাসরি অন্য কোনো address এ চলে গিয়ে সেখান থেকে instruction execute করা শুরু করে।

**🔸 JMP Instruction এর তিনটি ধরণ**

Intel 8086 থেকে শুরু করে আধুনিক প্রসেসর পর্যন্ত ৩ রকমের Unconditional Jump আছে:

| **ধরণ** | **আকার (Bytes)** | **কোথায় Jump করতে পারে** | **নাম** |
| --- | --- | --- | --- |
| Short Jump | 2 bytes | বর্তমান segment-এর ভেতরে ±128 bytes | **Intrasegment jump** |
| Near Jump | 3 bytes (16-bit), 5 bytes (32-bit) | বর্তমান code segment-এর ভেতরে ±32K bytes (বা ±2GB in protected mode) | **Intrasegment jump** |
| Far Jump | 5 bytes | অন্য segment-এ (যে কোনো জায়গায়) | **Intersegment jump** |



**🟢 1. Short Jump (Relative Jump)**

**📘 ধারণা:**

* Short Jump ব্যবহার হয় যখন **লাফের দূরত্ব ছোট** — অর্থাৎ বর্তমান instruction থেকে **–128 থেকে +127 bytes** দূরে।
* এটা **relative jump**, কারণ এখানে **নির্দিষ্ট address** না দিয়ে **displacement (দূরত্ব)** দেওয়া হয়।
* Displacement হলো signed 1 byte value।

**📖 কিভাবে কাজ করে:**

* Jump করার সময়, এই displacement টি **sign-extended হয়ে** Instruction Pointer (IP/EIP)-এর সাথে যোগ হয়।
* এতে নতুন address তৈরি হয়, এবং CPU সেই address থেকে পরবর্তী instruction execute করে।

**📘 উদাহরণ:**

START: MOV AX, BX

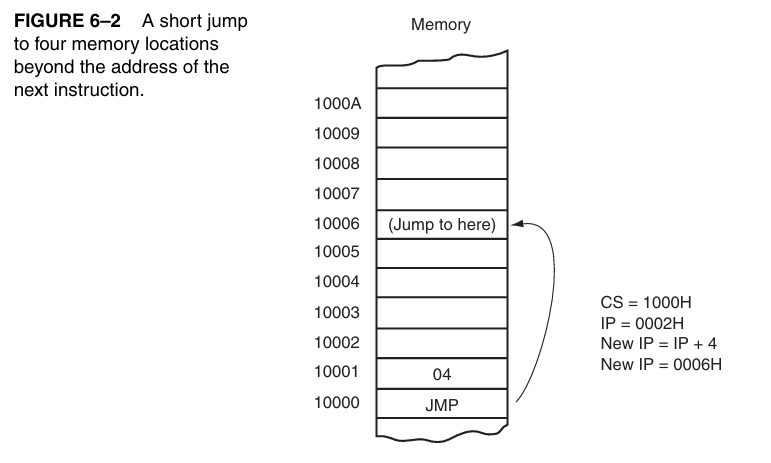
JMP SHORT NEXT

ADD AX, CX

NEXT: SUB AX, DX

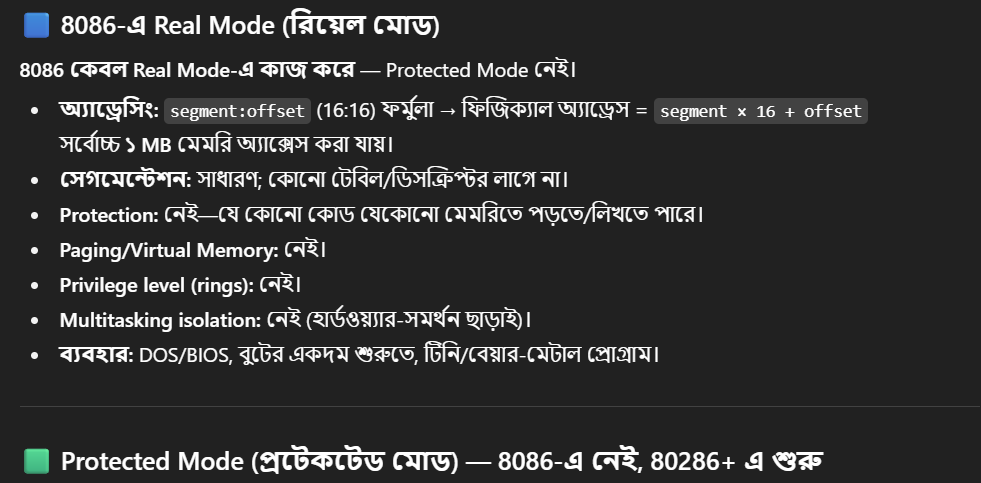
**কী হচ্ছে এখানে?**

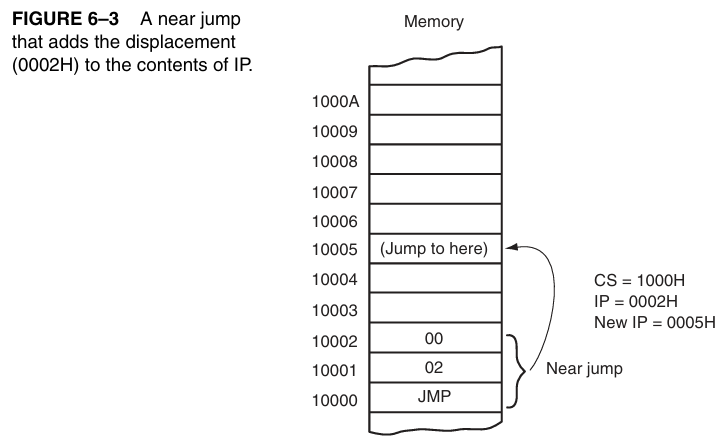
* JMP SHORT NEXT মানে হলো NEXT লেবেল পর্যন্ত লাফ দেওয়া।
* যদি NEXT instruction টি বর্তমান জায়গা থেকে 0x17 bytes দূরে হয়,  
  তাহলে CPU IP = IP + 17h করে লাফ দিবে।
* তাই, এই jump কোডটি যেকোনো জায়গায় সরালেও (relocate করলেও) ঠিকভাবে কাজ করে — কারণ এটা relative।



**🟡 2. Near Jump**

**📘 ধারণা:**

* Near Jump হলো বড় দূরত্বে লাফ দেওয়া — ±32K bytes (real mode এ), বা ±2GB (protected mode এ)।
* Short jump-এর মতোই এটি **relative jump** (relocatable)।
* Real mode-এ opcode এর পর **16-bit displacement** থাকে।
* Protected mode-এ **32-bit displacement** থাকে। 



**📖 কিভাবে কাজ করে:**

* Displacement টি instruction pointer (IP বা EIP)-এর সাথে যোগ হয়।
* Control বর্তমান segment এর মধ্যে অন্য address এ যায়।

**📘 উদাহরণ:**

START: MOV AX, BX

JMP NEXT ; assembler এটাকে near jump বানাবে

ADD AX, CX

...

NEXT: SUB AX, DX

Assembler listing এ দেখাবে:

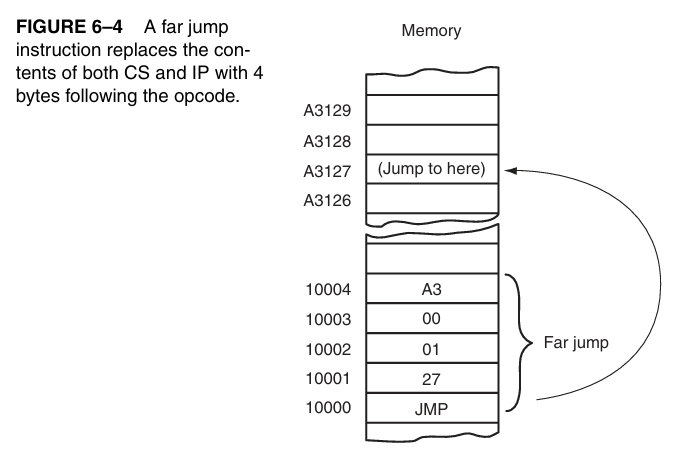
E9 F6 01

এখানে E9 হলো opcode, আর F6 01 হলো displacement (01F6h)।  
Assembler listing এ এটা “0200 R” দেখাবে, যেখানে R মানে **relocatable** —  
অর্থাৎ প্রোগ্রাম মেমোরির অন্য জায়গায় গেলেও ঠিকভাবে কাজ করবে।

**🔴 3. Far Jump**

**📘 ধারণা:**

* Far Jump ব্যবহার হয় যখন অন্য **segment** এ লাফ দিতে হয়।
* এটি **intersegment jump** — অর্থাৎ Code Segment (CS) পরিবর্তন হয়।
* মোট ৫ byte লাগে:
  + Bytes 2–3 → নতুন **offset address**
  + Bytes 4–5 → নতুন **segment address**



**📖 কিভাবে কাজ করে:**

* CPU প্রথমে নতুন segment address টি CS রেজিস্টারে লোড করে,  
  তারপর নতুন offset address টি IP রেজিস্টারে লোড করে।
* এখন program control চলে যায় সেই নতুন segment এর address এ।

**📘 উদাহরণ:**

JMP FAR PTR START

অথবা

EXTRN UP:FAR

JMP UP

এখানে EXTRN মানে হলো UP label টা অন্য ফাইল বা segment এ আছে।  
Linker প্রোগ্রাম compile-এর সময় সেই segment আর offset address ঠিক করে দেয়।

**🧠 Relocatable ও External Label বুঝে নাও:**

| **শব্দ** | **মানে** |
| --- | --- |
| **Relocatable (R)** | একই segment-এর মধ্যে কোড যেখানেই থাকুক, jump distance ঠিক থাকবে। |
| **External (E)** | Label বা function অন্য ফাইলে বা segment-এ আছে। Linker পরে সেটার address বসায়। |

**🔚 সংক্ষেপে তুলনামূলক সারাংশ**

| **বিষয়** | **Short Jump** | **Near Jump** | **Far Jump** |
| --- | --- | --- | --- |
| লাফের দূরত্ব | ±128 bytes | ±32K bytes (real mode) / ±2GB (protected mode) | যেকোনো address (segment পরিবর্তনসহ) |
| আকার | 2 bytes | 3 bytes (16-bit) / 5 bytes (32-bit) | 5 bytes |
| Segment পরিবর্তন হয়? | না | না | হ্যাঁ |
| Displacement | 8-bit signed | 16-bit / 32-bit signed | 16-bit offset + 16-bit segment |
| ধরণ | Relative, intrasegment | Relative, intrasegment | Absolute, intersegment |

**🔹 Jumps with Register Operands**

এই ধরনের jump কে বলা হয় 👉 **Indirect Jump**।

**🧩 1️⃣ ধারণা:**

সাধারণত JMP এর পরে আমরা কোনো **label** বা **address/displacement** দিই — যেমন

JMP NEXT

এখানে CPU NEXT label এর ঠিকানায় চলে যায়।

কিন্তু যদি আমরা লিখি —

JMP AX

তাহলে CPU সরাসরি **AX রেজিস্টারে যে address আছে**, সেখানে চলে যায়।  
অর্থাৎ এখন jump-এর ঠিকানা **AX** রেজিস্টারের ভেতরে রয়েছে।

**⚙️ 2️⃣ কিভাবে কাজ করে:**

* Short বা Near jump এ **displacement** IP (Instruction Pointer)-এর সাথে যোগ হয়।
* কিন্তু **Register Jump** এ **কিছু যোগ হয় না** —  
  বরং CPU **register-এর মান সরাসরি IP তে কপি করে**।

👉 যেমন:

MOV AX, 2000H

JMP AX

➡️ এখানে AX = 2000H  
➡️ যখন JMP AX execute হবে, তখন CPU IP = 2000H করে দিবে।  
➡️ ফলে Program Control চলে যাবে offset 2000H এ (বর্তমান Code Segment-এর মধ্যে)।

**🧠 Protected Mode এ:**

80386 বা তার পরের প্রসেসরগুলিতে 32-bit register ব্যবহার করা যায়।

JMP EAX

এখানে EAX এর মান EIP এ চলে যাবে —  
অর্থাৎ 32-bit offset address এ লাফ দেবে।  
(Protected mode এ code segment হতে পারে 4 GB লম্বা।)

**🔸 Indirect Jumps Using an Index (Jump Table)**

এটি আরেকটি শক্তিশালী ফিচার — **Jump Table** ব্যবহার করে **multiple choice branching** করা হয় (যেমন switch-case)।

**🧩 1️⃣ ধারণা:**

* এখানে JMP instruction এর operand হিসেবে [ ] (মেমরি অ্যাড্রেস) ব্যবহার করা হয়।
* অর্থাৎ jump এর **address মেমরির ভেতরে সংরক্ষিত থাকে**।
* CPU প্রথমে ঐ মেমরি থেকে offset (বা segment+offset) নেয়, তারপর সেখানে jump করে।

**📘 উদাহরণ:**

ধরো তোমার প্রোগ্রামে কয়েকটি ভিন্ন ভিন্ন কোড ব্লক আছে,  
এবং তুমি চাও register-এর মান অনুযায়ী সেগুলোর একটিতে লাফ দিতে।

TABLE DW START1, START2, START3 ; Jump Table: ৩টা address

MOV SI, 0002H ; ধরা যাক ২য় এন্ট্রি দরকার

JMP [TABLE + SI] ; Jump হবে START2 তে

🔹 এখানে কী হচ্ছে:

* TABLE এ তিনটি offset address রাখা আছে।
* SI নির্ধারণ করছে কোন address বেছে নেওয়া হবে।
* JMP [TABLE + SI] মানে → **TABLE + SI** অবস্থানে যে address আছে, সেখানে jump করা।

এভাবে CPU সরাসরি memory থেকে address নিয়ে jump করে,  
তাই একে বলে **Indirect Jump Using Index** (অথবা Double Indirect Jump)।

**🧠 “Double-Indirect” কথার মানে:**

* Register দিয়ে jump (যেমন JMP AX) — একে বলে **Indirect Jump**।
* Memory এর মধ্যে address রাখা থাকে (যেমন JMP [SI] বা JMP TABLE[SI]) —  
  একে বলে **Double-Indirect Jump**,  
  কারণ প্রথমে memory থেকে address নেওয়া হয়, তারপর সেই address এ jump করা হয়।

**📘 FAR PTR ব্যবহারের কারণ:**

যদি jump table-এ **segment:offset** address (দুই অংশ) রাখা থাকে,  
তাহলে assembler-কে জানাতে হয় এটা **Far Jump**, যেমন:

JMP FAR PTR [TABLE + SI]

তাহলে CPU নতুন segment এবং offset উভয়েই লোড করে লাফ দেয় (intersegment jump)।

**🔚 সারসংক্ষেপ টেবিল**

| **ধরণ** | **উদাহরণ** | **ব্যাখ্যা** |
| --- | --- | --- |
| Register Indirect Jump | JMP AX | AX রেজিস্টারের ভেতরের address এ jump |
| Register Indirect Jump (32-bit) | JMP EAX | EAX এর মান EIP তে কপি হয় (protected mode) |
| Memory Indirect Jump | JMP [SI] | মেমরিতে থাকা address এ jump |
| Indexed Jump Table | JMP TABLE[SI] | Jump table থেকে offset পড়ে নির্দিষ্ট লেবেলে লাফ |
| Double-Indirect Jump | JMP [TABLE + SI] (যেখানে table এ offset আছে) | Memory → address → Jump (দুই ধাপের লাফ) |
| Far Indirect Jump | JMP FAR PTR [TABLE + SI] | নতুন segment ও offset উভয়ে লাফ দেয় |

## 🔹 Conditional Jumps (শর্তাধীন লাফ)

**Conditional jump** হলো এমন একটি jump যা **শুধু তখনই ঘটে যখন কোনো শর্ত সত্য হয়**।  
অর্থাৎ, প্রোগ্রামের control শুধুমাত্র তখনই অন্য address এ যায়, যখন নির্দিষ্ট **flag বা register এর মান** শর্ত পূরণ করে।

### 🧩 1️⃣ Conditional Jump এর রেঞ্জ

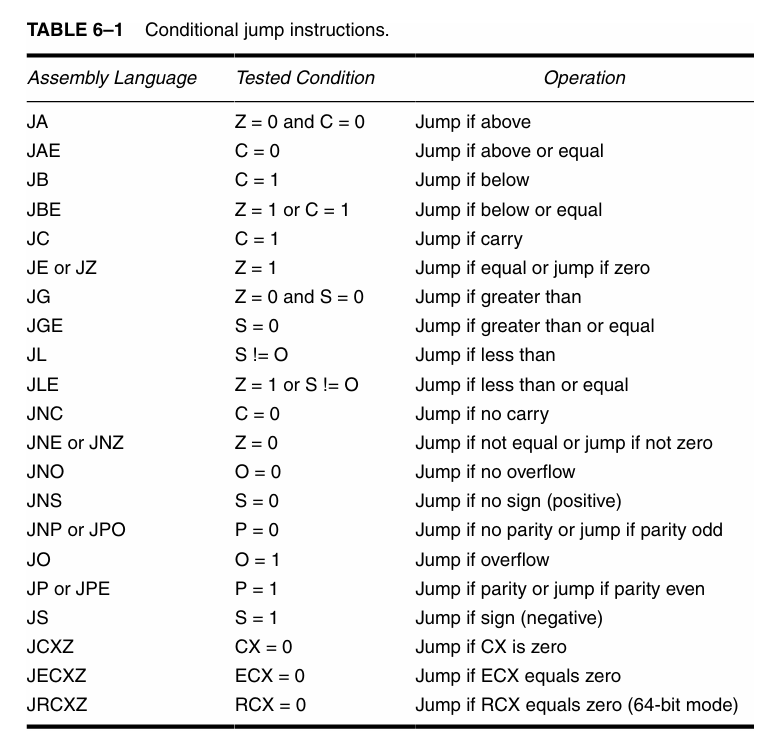
* **8086–80286:** সব Conditional Jump হলো **short jump**, ±128 bytes (–128..+127 bytes)
* **80386 এবং পরবর্তী:** Conditional Jump হতে পারে **short বা near**, ±32K bytes
* **Pentium 4, 64-bit mode:** Near jump distance ±2GB

Microsoft MASM 6.x assembler automatically adjusts যদি distance বেশি হয়।

### 🧩 2️⃣ Flag Bits যে Conditional Jumps পরীক্ষা করে

Conditional Jump সাধারণত **flag bits** পরীক্ষা করে। যেমন:

| **Flag** | **উদাহরণ** | **অর্থ** |
| --- | --- | --- |
| Carry (C) | JC | Jump if carry set |
| Zero (Z) | JE / JZ | Jump if equal / zero |
| Sign (S) | JS | Jump if negative |
| Overflow (O) | JO | Jump if overflow |
| Parity (P) | JP / JPE | Jump if parity even |



### 🧩 3️⃣ Magnitude Comparison

সংখ্যা তুলনা করার জন্য **signed এবং unsigned numbers** আলাদা করে পরীক্ষা করা হয়।

#### (ক) Signed Numbers:

* ব্যবহার: JG, JL, JGE, JLE, JE, JNE
* অর্থ: Greater than / Less than / Equal
* উদাহরণ: JG LABEL → Jump যদি প্রথম সংখ্যা দ্বিতীয় সংখ্যার থেকে বড় হয়।

#### (খ) Unsigned Numbers:

* ব্যবহার: JA, JB, JAE, JBE, JE, JNE
* অর্থ: Above / Below / Equal (unsigned)
* উদাহরণ: JA LABEL → Jump যদি প্রথম সংখ্যা দ্বিতীয় সংখ্যার থেকে **উপর** হয় (unsigned)

গুরুত্বপূর্ণ: 8-bit উদাহরণ:

* Unsigned FFH = 255 > 00H
* Signed FFH = –1 < 00H

### 🧩 4️⃣ Alternate Opcodes

* কিছু jump instruction এর **alternative opcode** আছে।
* যেমন: JE ↔ JZ, JA ↔ JNBE
* প্রায়শই JE/JZ ব্যবহৃত হয় কারণ সহজে বোঝা যায়।

### 🧩 5️⃣ Conditional Jump Without Flag

* JCXZ (Jump if CX = 0)
* JECXZ (Jump if ECX = 0)
* JRCXZ (64-bit mode, RCX = 0)
* এগুলো flag পরীক্ষা করে না, সরাসরি **register value পরীক্ষা করে**।

**উদাহরণ:**

SCASB ; Table search for value

JCXZ NOT\_FOUND ; Jump if CX = 0

* এখানে SCASB execution এর পর CX কমে গেলে, JCXZ jump করবে।
* Carry Flag ব্যবহার করা যেতে পারে **not found condition** বোঝাতে।

### 🔹 Summary / মূল বিষয়গুলো

1. **Conditional Jump:** শর্ত পূরণ হলে লাফ।
2. **Flags পরীক্ষা করে:** Z, C, S, O, P
3. **Signed vs Unsigned:** আলাদা instruction (JG/JL vs JA/JB)
4. **Distance:** 8086–80286 short ±128, 80386+ ±32K, 64-bit ±2G
5. **Alternate Opcodes:** JE ↔ JZ, JA ↔ JNBE
6. **Register Conditional Jump:** JCXZ/JECXZ/JRCXZ → সরাসরি register value পরীক্ষা করে।

**🔹 LOOP Instruction**

**LOOP** হলো assembly তে একটি **loop control instruction**, যা স্বয়ংক্রিয়ভাবে counter কমায় এবং নির্দিষ্ট শর্ত অনুযায়ী jump করে।

**মূল ধারণা:**

LOOP label

* CPU **CX** (বা ECX/RCX) রেজিস্টারকে 1 দিয়ে decrement করে।
* যদি counter ≠ 0 হয় → jump হয় label এ।
* যদি counter = 0 হয় → পরবর্তী instruction execute হয়।

**🧩 1️⃣ LOOP এর ব্যাবহার (8086–80286)**

* Counter: **CX (16-bit)**
* কাজ:
  1. CX = CX – 1
  2. যদি CX ≠ 0 → jump
  3. যদি CX = 0 → next instruction

**উদাহরণ:**

MOV CX, 5 ; loop counter = 5

START:

; কিছু কাজ

LOOP START ; CX কমে 0 না হলে START এ jump

➡️ এই loop 5 বার চলবে।

**🧩 2️⃣ 80386 এবং পরবর্তী প্রসেসর**

* Counter: **CX (16-bit)** বা **ECX (32-bit)**, instruction mode অনুযায়ী।
  + **16-bit mode:** CX
  + **32-bit mode:** ECX
* Instruction override:
  + LOOPW → 16-bit CX
  + LOOPD → 32-bit ECX
* 64-bit mode → **RCX (64-bit)**

**🔹 Conditional LOOPs**

LOOP instruction এর **শর্তাধীন সংস্করণ** আছে:

| **Instruction** | **শর্ত** | **কাজ** |
| --- | --- | --- |
| LOOPE / LOOPZ | Equal / Zero flag set | Jump only if CX ≠ 0 **and** Zero Flag = 1 |
| LOOPNE / LOOPNZ | Not equal / Zero flag clear | Jump only if CX ≠ 0 **and** Zero Flag = 0 |

JE / JZ / JNE / JNZ এর মতোই flag পরীক্ষা করে।

**🧩 উদাহরণ (LOOPE / LOOPNE)**

MOV CX, 10

START:

CMP AX, BX ; Compare two values

LOOPE START ; যদি AX = BX এবং CX ≠ 0, jump করে START

**কী হচ্ছে এখানে?**

1. CX কমে 1 হয়।
2. Zero Flag পরীক্ষা করা হয় (CMP AX,BX) → AX = BX হলে Z=1
3. যদি Z=1 এবং CX≠0 → loop আবার START এ যাবে
4. যদি Z=0 বা CX=0 → loop শেষ হবে

* LOOPE = LOOPZ
* LOOPNE = LOOPNZ

**🔹 সারসংক্ষেপ (LOOP / Conditional LOOP)**

| **Instruction** | **Counter** | **শর্ত** | **Jump করে যদি…** |
| --- | --- | --- | --- |
| LOOP | CX/ECX/RCX | কোনো flag পরীক্ষা নেই | CX ≠ 0 |
| LOOPE / LOOPZ | CX/ECX/RCX | Zero Flag = 1 | CX ≠ 0 **এবং** Z=1 |
| LOOPNE / LOOPNZ | CX/ECX/RCX | Zero Flag = 0 | CX ≠ 0 **এবং** Z=0 |

💡 **Tip:** LOOP ব্যবহার করলে আপনাকে নিজে CX decrement করার দরকার নেই, এটি স্বয়ংক্রিয়ভাবে করে।

**🔹 Procedure (Subroutine / Function) কী?**

একটি **procedure** হলো প্রোগ্রামের এমন একটি অংশ যা কোনো নির্দিষ্ট কাজ সম্পন্ন করে এবং প্রোগ্রামের বিভিন্ন জায়গা থেকে বারবার **ডাকা (call)** যেতে পারে।

**🧠 উদাহরণ:**

ধরো, তোমার প্রোগ্রামে “দুইটি সংখ্যার যোগফল” বারবার বের করতে হবে।  
তাহলে সেই কাজের জন্য আলাদা করে একটি procedure তৈরি করা যায়।

এতে —

* কোড **পুনরাবৃত্তি কমে**,
* **মেমোরি সাশ্রয়** হয়,
* এবং প্রোগ্রাম বোঝা সহজ হয়।

**🔹 Procedure এর গঠন**

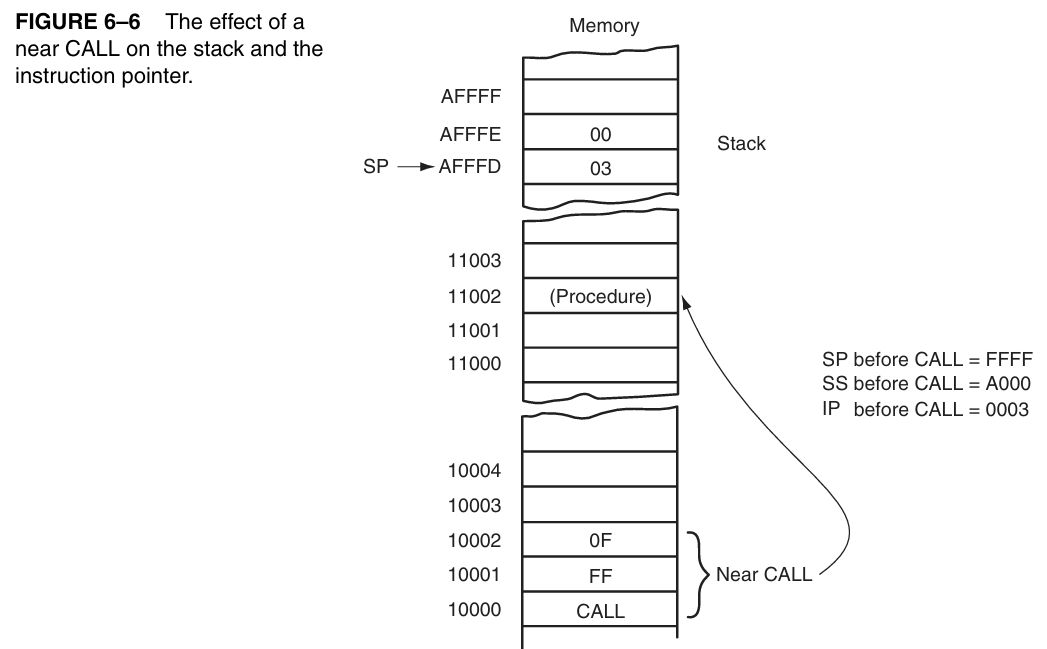
PROC\_NAME PROC NEAR

; এখানে নির্দেশনা বা কোড থাকবে

RET

PROC\_NAME ENDP

👉 **PROC** দিয়ে শুরু,  
👉 **ENDP** দিয়ে শেষ,  
👉 এবং ভেতরে কাজের নির্দেশনা থাকে।



**🔹 CALL Instruction — Procedure ডাকার জন্য**

**CALL** instruction প্রোগ্রামের control পাঠায় নির্দিষ্ট procedure-এ।

CALL করার সময় CPU যা করে —

1. **Return address** (যেখান থেকে call দেওয়া হয়েছিল, তার পরের instruction-এর ঠিকানা) stack-এ push করে।
2. তারপর **procedure** এ control পাঠায়।

এরপর procedure শেষ হলে, **RET** instruction stack থেকে সেই address pop করে এবং control ফেরত পাঠায় মূল প্রোগ্রামে।

**🔹 Near CALL (Intrasegment Procedure)**

**Near CALL** তখন হয়, যখন procedure এবং CALL — **একই code segment** এর মধ্যে থাকে।

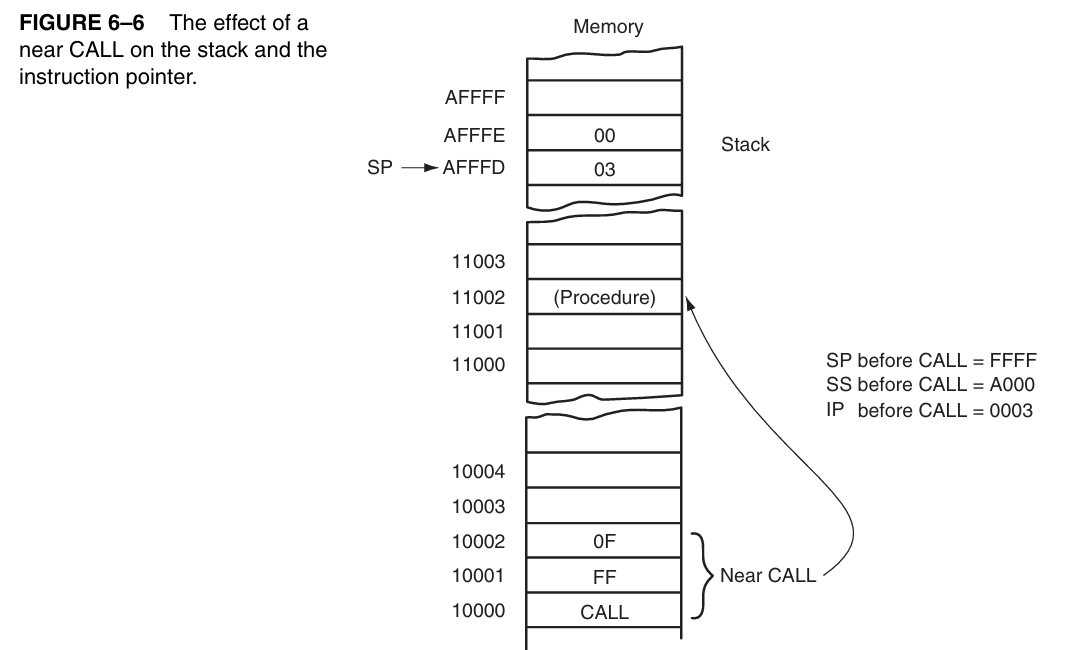
* এটি সাধারণত **৩ byte** এর instruction।
* ১ম byte → opcode
* ২য় ও ৩য় byte → displacement (±32K range)

📌 **8086–80286:** 16-bit offset  
📌 **80386 ও তার পরের প্রসেসরে:** 32-bit offset (±2G)

**🧩 কীভাবে কাজ করে:**

1. IP (Instruction Pointer) — অর্থাৎ পরের instruction এর ঠিকানা — **stack এ push হয়**
2. এরপর CALL instruction এর displacement যোগ হয়ে procedure এ control যায়।

🧠 তাই CALL শেষে **RET** দিলে প্রোগ্রাম আবার ওই return address এ ফিরে যায়।



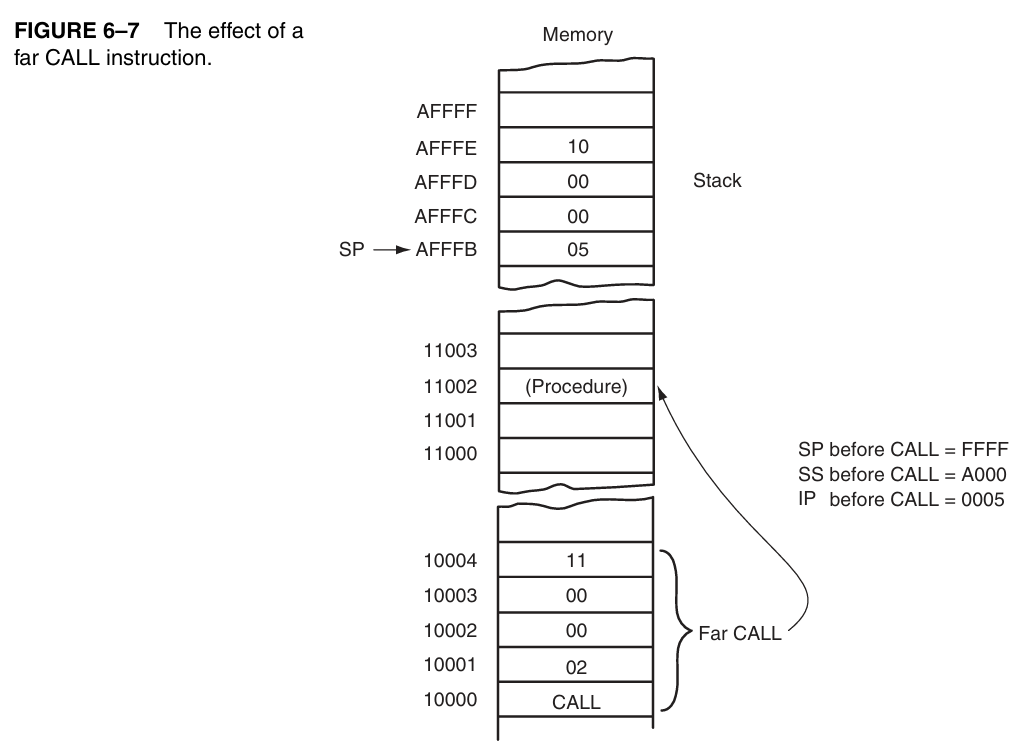
**🔹 Far CALL (Intersegment Procedure)**

**Far CALL** ব্যবহার হয় যখন procedure অন্য **segment** এ থাকে।

* Instruction length = **5 bytes**
* 1st byte → opcode
* পরের 4 byte → IP এবং CS (নতুন address)

**🧩 কাজের ধাপ:**

1. CALL করার সময় **বর্তমান CS ও IP** stack এ push হয়।
2. তারপর নতুন **CS:IP** লোড করে procedure এ control যায়।
3. RET দিলে stack থেকে পুরোনো CS ও IP pop হয় → control আগের segment এ ফিরে আসে।



**🔹 কেন IP/EIP Stack এ Save হয়?**

কারণ CALL এর পর প্রোগ্রামকে **ফিরে আসতে হয় ঠিক CALL এর পরের instruction-এ**।  
👉 তাই CALL করার সময় CPU পরের instruction এর ঠিকানা stack এ সংরক্ষণ করে রাখে।  
👉 RET instruction সেটা pop করে control ফিরিয়ে আনে।

**🔹 সংক্ষিপ্ত তুলনা: Near vs Far CALL**

| **বিষয়** | **Near CALL** | **Far CALL** |
| --- | --- | --- |
| Segment | একই segment | অন্য segment |
| Stack এ Push করে | শুধুমাত্র IP | CS ও IP দুটোই |
| Instruction size | 3 bytes | 5 bytes |
| Range | ±32K (16-bit) বা ±2G (32-bit) | সম্পূর্ণ মেমোরি |
| ব্যবহার | ছোট প্রোগ্রামে | বড় সিস্টেমে বা লাইব্রেরি কল করার সময় |

**🔹 CALL with Register Operand (Indirect CALL by Register)**

এই CALL ফর্মটি সরাসরি কোনো **label** বা **procedure name** না লিখে, বরং **রেজিস্টারের ভেতরে রাখা address** এ control পাঠায়।

**🧠 উদাহরণ:**

CALL BX

**⚙️ কী ঘটে এখানে:**

1. CPU প্রথমে **বর্তমান IP (return address)** stack-এ push করে (যাতে পরে RET এ ফিরে আসা যায়)
2. তারপর **BX register-এর ভেতরের মানটিকে নতুন IP হিসেবে লোড করে**  
   অর্থাৎ, control যায় সেই offset address-এ যা BX রেজিস্টারে ছিল।

👉 এটি **indirect CALL**, কারণ CALL সরাসরি address না পেয়ে **রেজিস্টারের মাধ্যমে পরোক্ষভাবে (indirectly)** address পায়।

**📘 বৈশিষ্ট্য:**

* কেবল **16-bit registers (AX, BX, CX, DX, SI, DI, BP, SP)** ব্যবহার করা যায়
* **Segment register** (CS, DS ইত্যাদি) ব্যবহার করা যায় না
* সবসময় **বর্তমান code segment** এর মধ্যেই jump হয় (near call)

**🔹 CALL with Indirect Memory Address**

এই CALL ফর্মটি ব্যবহার হয় যখন **subroutine dynamically নির্বাচন করতে হয়**,  
অর্থাৎ কোন subroutine call হবে — তা প্রোগ্রাম চলাকালীন সময়ে **memory table বা lookup table** থেকে নির্ধারিত হয়।

**🧩 Syntax:**

CALL [ADDRESS]

অথবা যদি টেবিল ব্যবহার করা হয়:

CALL [TABLE + SI]

**⚙️ কী ঘটে এখানে:**

1. CPU stack-এ return address push করে
2. তারপর [TABLE + SI] memory location থেকে offset address পড়ে
3. এবং সেই address অনুযায়ী procedure-এ control পাঠায়

**📘 সংক্ষিপ্ত তুলনা**

| **CALL Type** | **কীভাবে Address পাওয়া যায়** | **কোথায় Jump হয়** | **উদাহরণ** |
| --- | --- | --- | --- |
| Direct CALL | Label / Procedure name | নির্দিষ্ট | CALL SUM\_PROC |
| Register CALL | Register এর ভেতরের Offset | বর্তমান segment | CALL BX |
| Memory Indirect CALL | Memory বা Table থেকে Offset | বর্তমান segment | CALL [TABLE + SI] |

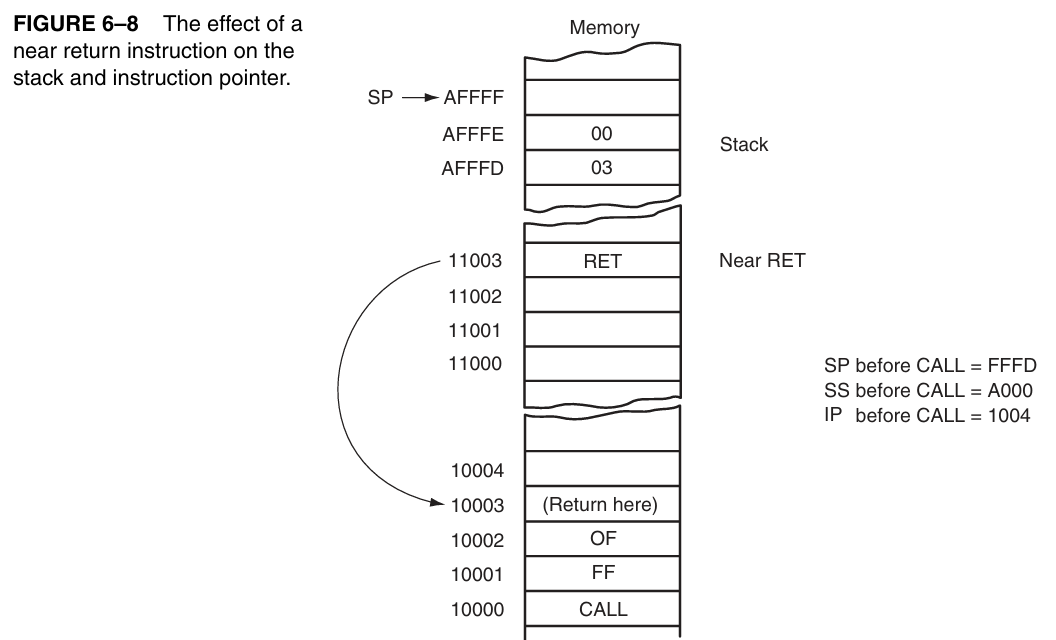
**🧠 সারসংক্ষেপে মনে রাখো**

* **CALL BX** → indirect call via register
* **CALL [MEM]** → indirect call via memory/table
* উভয় ক্ষেত্রেই return address stack-এ push হয়
* Control যায় নতুন offset address-এ (same segment unless FAR PTR ব্যবহার করা হয়)

**🔹 RET (Return Instruction) — মূল ধারণা**

RET মানে **Return from Procedure**  
অর্থাৎ, যখন কোনো procedure শেষ হয়, তখন RET instruction CPU-কে বলে:

“আমি আমার কাজ শেষ করেছি, এখন আগের জায়গায় ফিরে যাও।”



**⚙️ কীভাবে কাজ করে**

যখন প্রোগ্রাম CALL instruction ব্যবহার করে কোনো procedure-এ যায়:

1. CPU **return address (IP/EIP)** — অর্থাৎ CALL-এর পরের instruction-এর address — **stack এ push করে**।
2. তারপর procedure-এ control যায়।

এখন, যখন procedure শেষ হয় এবং RET চলে:

1. CPU stack থেকে **return address (IP/EIP)** **pop করে**,
2. তারপর সেই address-এ **jump করে ফিরে যায়** — CALL-এর পরের instruction-এ।

**🔸 দুই ধরনের RET**

| **ধরন** | **কী Pop হয় Stack থেকে** | **কোথায় ফিরে যায়** | **উদাহরণ** |
| --- | --- | --- | --- |
| **Near RET** | শুধু **IP (16-bit)** বা **EIP (32-bit)** | বর্তমান Code Segment (CS অপরিবর্তিত) | RET |
| **Far RET** | **IP + CS (16-bit + 16-bit)** বা **EIP + CS (32-bit + 16-bit)** | অন্য Code Segment (CS পরিবর্তিত হয়) | RET FAR |

**🧩 RET n (Return and Pop Parameters)**

আরও এক ধরনের RET আছে — RET n  
→ এটা return করার সময় **stack থেকে n বাইট extra pop করে**  
(প্রায়ই procedure-এর parameter সরানোর জন্য ব্যবহৃত হয়)

**উদাহরণ:**

RET 4

মানে:

* Stack থেকে return address pop করবে
* তারপর আরও ৪ বাইট pop করবে (parameter remove করার জন্য)

**🧭 সারসংক্ষেপ**

| **ধরন** | **Stack থেকে কী Pop হয়** | **কাজ** |
| --- | --- | --- |
| **RET (near)** | IP/EIP | বর্তমান segment এ ফিরে আসে |
| **RET FAR** | IP/EIP + CS | অন্য segment এ ফিরে যায় |
| **RET n** | IP/EIP + n বাইট | Parameter সহ return করে |

**🧠 Interrupt কী?**

**Interrupt** মানে হলো —

প্রোগ্রামের স্বাভাবিক কাজ মাঝপথে থামিয়ে,  
কোনো গুরুত্বপূর্ণ কাজ সম্পন্ন করার জন্য **স্বয়ংক্রিয়ভাবে বা প্রোগ্রাম দ্বারা অন্য একটা procedure (Interrupt Service Routine – ISR)**-এ চলে যাওয়া।

👉 অর্থাৎ, Interrupt = **Hardware বা Software Generated CALL Instruction**

**⚙️ Interrupt এর ধরণ**

| **ধরণ** | **উৎস** | **উদাহরণ** | **ব্যাখ্যা** |
| --- | --- | --- | --- |
| **Hardware Interrupt** | বাহ্যিক সিগন্যাল (external pin থেকে) | Keyboard interrupt, Timer interrupt | CPU-কে hardware signal পাঠিয়ে বলে – “এখনই আমার কাজ করো।” |
| **Software Interrupt** | প্রোগ্রামের ভিতর থেকে (instruction দ্বারা) | INT 21H, INT 10H ইত্যাদি | নির্দিষ্ট Software service বা function call করার জন্য ব্যবহৃত হয়। |

**🧩 Interrupt Vector Table (IVT)**

* এটি হলো একটি **table** যেখানে প্রতিটি interrupt-এর **service routine-এর address (CS:IP)** রাখা থাকে।
* এই table টি **memory-এর শুরুতে (00000H – 003FFH)** অবস্থান করে।
* প্রতিটি **Interrupt Vector = 4 bytes** (2 byte for IP + 2 byte for CS)

👉 অর্থাৎ, মোট 256টি interrupt × 4 bytes = 1024 bytes (1 KB)

| **Interrupt Type** | **Vector Address** | **Description** |
| --- | --- | --- |
| 00H | 00000H | Divide by Zero |
| 04H | 00010H | Overflow |
| 10H | 00040H | Video Services |
| 21H | 00084H | DOS Services |
| ... | ... | ... |

📘 **Protected Mode**-এ IVT-এর পরিবর্তে **Interrupt Descriptor Table (IDT)** ব্যবহার হয়  
(প্রতিটি descriptor = 8 bytes)

**🔸 Software Interrupt Instructions**

মোট তিনটি Software Interrupt Instruction আছে 👇

| **Instruction** | **কাজ** | **বিশেষ তথ্য** |
| --- | --- | --- |
| **INT n** | নির্দিষ্ট interrupt type number (0–255) call করে | সবচেয়ে বেশি ব্যবহৃত |
| **INT 3** | Breakpoint interrupt (debugging) | 1 byte instruction |
| **INTO** | Overflow flag (O=1) হলে interrupt করে (Type 4) | Conditional interrupt |

**🧱 ১. INT n**

* Syntax: INT n
* কাজ: Interrupt Vector Table থেকে n-নম্বর vector এর ঠিকানা নেয়,  
  এবং সেই ঠিকানার Interrupt Service Procedure (ISP) চালায়।

🧮 উদাহরণ:

INT 10H ; Video Service Call

INT 21H ; DOS Service Call

👉 Address বের করার নিয়ম:  
Interrupt Number × 4 = Vector Address

যেমন, INT 10H → 10H × 4 = 40H → vector 0040H–0043H

**🧱 ২. INT 3 (Breakpoint Interrupt)**

* এটি debugging-এর জন্য ব্যবহৃত হয়।
* ১ byte instruction (অন্য INT instruction গুলো ২ byte)
* প্রোগ্রামের মাঝে INT 3 বসিয়ে **execution temporarily থামানো যায়**।

MOV AX, BX

INT 3 ; Breakpoint for debugging

ADD AX, 10

**🧱 ৩. INTO (Interrupt on Overflow)**

* Overflow flag (OF) পরীক্ষা করে।
* যদি OF = 1 হয়, তাহলে **interrupt vector type 4**-এ যায়।
* যদি OF = 0 হয়, কিছুই করে না।

ADD AL, BL

INTO ; যদি overflow ঘটে, তাহলে interrupt 4

**🧩 IRET / IRETD (Interrupt Return)**

Interrupt service শেষ হলে **IRET** instruction ব্যবহৃত হয়  
(যেটা RET-এর থেকে একটু আলাদা)।

**⚙️ কাজের ধারা:**

IRET stack থেকে নিচেরগুলো pop করে 👇

1. IP/EIP → Instruction Pointer
2. CS → Code Segment
3. Flags → Processor Status

অর্থাৎ, **IRET = POPF + RET FAR**

**🧭 সারসংক্ষেপ:**

| **Instruction** | **কাজ** | **Stack থেকে কী Pop হয়** | **Mode** |
| --- | --- | --- | --- |
| **CALL** | Procedure call | IP (near) বা IP+CS (far) | Real/Protected |
| **RET** | Procedure return | IP বা IP+CS | Real/Protected |
| **INT n** | Software interrupt | (Push Flags + CS + IP), Jump to ISR | Real/Protected |
| **IRET** | Return from interrupt | IP + CS + Flags | Real/Protected |

👉 **ISP** মানে হলো **Interrupt Service Procedure** (অনেক সময় একে **Interrupt Handler** ও বলা হয়)।

**🔹 সংজ্ঞা:**

**Interrupt Service Procedure (ISP)** হলো একটি বিশেষ প্রোগ্রাম রুটিন বা সাবরুটিন,  
যা **Interrupt ঘটলে স্বয়ংক্রিয়ভাবে চালু হয়** এবং সেই Interrupt-এর নির্দিষ্ট কাজটি সম্পন্ন করে।

অর্থাৎ, যখন CPU-তে কোনো **Interrupt signal** আসে (hardware বা software),  
তখন CPU মূল প্রোগ্রাম থামিয়ে দেয় এবং **ISP** বা **Interrupt Service Procedure**-এ চলে যায়।

**🔹 উদাহরণ দিয়ে বুঝি:**

ধরো, তুমি একটি প্রোগ্রাম চালাচ্ছো —  
এ সময় কীবোর্ড থেকে কোনো কী প্রেস করা হলো।  
এই কী প্রেস হলো একটি **hardware interrupt**।  
তখন CPU মূল প্রোগ্রাম থামিয়ে **কীবোর্ডের জন্য লেখা ISP** চালু করবে  
(যেখানে লেখা আছে কী প্রেস হলে কী করতে হবে)।  
ISP কাজ শেষ করলে CPU আবার মূল প্রোগ্রামে ফিরে আসবে।

**🔹 ISP-এর কাজের ধাপ:**

1. **Interrupt ঘটল** — যেমন: hardware signal বা software INT।
2. CPU **বর্তমান অবস্থান (IP, CS, FLAGS)** stack-এ সংরক্ষণ করে।
3. CPU **Interrupt Vector Table** থেকে ISP-এর ঠিকানা বের করে।
4. CPU ঐ ঠিকানায় থাকা **ISP** চালায়।
5. কাজ শেষে **IRET** ইনস্ট্রাকশন দিয়ে CPU আবার মূল প্রোগ্রামে ফিরে আসে।

**🔹 ISP-এর বৈশিষ্ট্য:**

* এটি **RET** নয়, **IRET** ইনস্ট্রাকশন দিয়ে শেষ হয়।
* সাধারণত **সিস্টেম লেভেলের কাজ** করে (যেমন: keyboard input, timer, disk I/O)।
* CPU স্বয়ংক্রিয়ভাবে এটি **ডাকে (CALL)**, প্রোগ্রামারকে হাতে ডাকতে হয় না (hardware interrupt-এর ক্ষেত্রে)।
* ISP সাধারণত **Interrupt Vector Table**-এ সংরক্ষিত থাকে।

**🔹 সহজ ভাষায়:**

🔸 **ISP** হলো এমন একটি বিশেষ প্রোগ্রাম,  
যেটি CPU স্বয়ংক্রিয়ভাবে চালায় যখন কোনো **Interrupt** ঘটে,  
এবং কাজ শেষ হলে IRET দিয়ে মূল প্রোগ্রামে ফিরে আসে।