অবশ্যই, চল 8085 microprocessor-এর Interrupt pin গুলোর বিশ্লেষণ করি নিচের টেবিল এবং উদাহরণসহ — যাতে সহজে মাথায় থাকে।

**🔌 8085 Microprocessor এর Interrupt Pins Explained:**

| **Interrupt** | **Priority** | **Type** | **Maskable?** | **Edge/Level Triggered** | **Description** |
| --- | --- | --- | --- | --- | --- |
| **TRAP** | 1 (Highest) | Hardware | ❌ Non-maskable | Edge + Level Triggered | এটি সবচেয়ে জরুরি Interrupt। Power failure বা emergency situation-এ ব্যবহার হয়। CPU এটি ignore করতে পারে না। |
| **RST7.5** | 2 | Hardware | ✅ Maskable | **Edge Triggered** | একবার signal এলে latch হয়ে যায়, যতক্ষণ না CPU accept করে। Highest priority among maskable interrupts. |
| **RST6.5** | 3 | Hardware | ✅ Maskable | **Level Triggered** | শুধু তখনই গ্রহণ করা হয় যদি signal active থাকে যখন CPU ready হয়। |
| **RST5.5** | 4 | Hardware | ✅ Maskable | **Level Triggered** | Lower priority, তবে mask করা যায়। |
| **INTR** | 5 (Lowest) | Hardware | ✅ Maskable | Level Triggered | General-purpose interrupt। External device use করে। Acknowledge করতে **INTA** signal দরকার হয়। |

**✅ Maskable vs Non-Maskable Interrupt**

* **Maskable**: যেগুলোকে **DI (Disable Interrupts)** কমান্ড দিয়ে বন্ধ (mask) করা যায়।
* **Non-Maskable**: বন্ধ করা যায় না। সব সময় enable থাকে। যেমন: TRAP

**🎯 Priority Order (Highest → Lowest):**

TRAP > RST7.5 > RST6.5 > RST5.5 > INTR

**🧠 উদাহরণ:**

ধরো তুমি CPU চালাচ্ছো, হঠাৎ:

* Power failure → **TRAP** active হয় → CPU সঙ্গে সঙ্গে এই interrupt serve করে।
* Keyboard press (RST7.5) → CPU যদি interrupt enable করে রাখে, তাহলে এটা কাজ করবে।
* Timer interrupt (RST6.5) → শুধুমাত্র যদি level high থাকে।

**🖼️ ছোট চার্ট মনে রাখার জন্য:**

TRAP 🚨 Emergency (non-maskable)

RST7.5 🔔 Edge triggered, highest maskable

RST6.5 🔔 Level triggered

RST5.5 🔕 Level triggered, lowest maskable

INTR ❓ General interrupt

**🔁 INTA (Interrupt Acknowledge)**

INTR interrupt signal দিলে CPU একবার verify করে, তারপর **INTA (Interrupt Acknowledge)** signal দিয়ে confirm করে যে interrupt accept করেছে।

তুমি চাইলে এর simulation (TRAP বা RST7.5 কিভাবে কাজ করে), বা code-level exampleও দিতে পারি।

বলো, আর কিছু জানতে চাও এই topic-এর মধ্যে?

তুমি যে টেক্সটটা দিয়েছো, সেটা 8085 Microprocessor-এর **INTR Interrupt** এবং **Interrupt Controller 8259A** নিয়ে আলোচনা করছে। আমি নিচে **সহজ ও বিস্তারিতভাবে ব্যাখ্যা** করছি যেন পুরো বিষয়টা তুমি পরিষ্কারভাবে বুঝতে পারো।

## 🔌 INTR Interrupt: বিস্তারিত ব্যাখ্যা

### 🔹 ১. INTR কী?

* INTR (Interrupt Request) হলো 8085-এর **একটি সাধারণ maskable hardware interrupt pin**।
* এটি **level triggered** এবং **handshake mechanism** অনুযায়ী কাজ করে।

### 🔹 ২. INTR কিভাবে কাজ করে?

1. যখন কোনো external device INTR পিন-এ HIGH signal পাঠায়,
2. তখন 8085 **চলমান instruction** সম্পন্ন করে,
3. এরপর **interrupt system enabled থাকলে**, CPU INTR signal গ্রহণ করে,
4. CPU একটি **Interrupt Acknowledge (INTA)** signal LOW করে পাঠায় – এটি মানে: "Interrupt গ্রহণ করা হয়েছে"।
5. এখন **external hardware (যেমন: 8259 Interrupt Controller)** 8085-এর **data bus**-এ একটি **opcode** দেয়, যেমন RST 7, CALL addr, ইত্যাদি।
6. CPU এই opcode execute করে — অর্থাৎ ISR (Interrupt Service Routine) এ যায়।

### 🔸 উদাহরণ:

* ধরো একটা external device CPU-কে বলে: “আমার কাজ করো।” তখন সে INTR = HIGH পাঠায়।
* CPU কাজ শেষ করে INTA = LOW পাঠায়।
* এরপর external device RST 3 এর opcode CPU-কে দেয়।
* CPU সেই RST 3 এর address (0030H) তে গিয়ে interrupt routine চালায়।

### 🔹 ৩. INTR এর কিছু বৈশিষ্ট্য:

| **বৈশিষ্ট্য** | **ব্যাখ্যা** |
| --- | --- |
| **Maskable** | INTR-কে DI (Disable Interrupts) কমান্ড দিয়ে বন্ধ করা যায়। |
| **Level Triggered** | Signal যতক্ষণ HIGH থাকে ততক্ষণ active ধরা হয়। |
| **Handshake Mechanism** | INTR দেয় → CPU INTA দেয় → External Hardware opcode দেয়। |
| **Priority** | INTR-এর priority সবচেয়ে কম (Lowest)। |
| **External Vector Required** | INTR নিজে কোন address দেয় না; external hardware (যেমন 8259) vector address বা instruction দেয়। |

## 🧠 Software Interrupts: RST Instructions

8085 এ আছে **৮টি Software Interrupt**:

RST 0 → 0000H

RST 1 → 0008H

RST 2 → 0010H

...

RST 7 → 0038H

এগুলো manually interrupt trigger করার জন্য প্রোগ্রাম থেকে ব্যবহৃত হয়।

## 🛠 Interrupt Controller: 8259A

যখন অনেকগুলো external device থাকে, তখন INTR pin-এর মাধ্যমে সব interrupt control করা কঠিন হয়। তাই **8259A Interrupt Controller** ব্যবহার করা হয়।

### 8259 কী করে?

1. ৮টি interrupt input নেয় (IR0 – IR7)
2. Priority ঠিক করে – কোন interrupt আগে serve হবে
3. Mask করে রাখতে পারে – কোন interrupt গ্রহণ না করা হবে
4. Proper RST বা CALL instruction data bus-এ পাঠায়
5. CPU-কে INTR signal দেয়, এবং CPU INTA দিলে instruction পাঠিয়ে দেয়

### 8259 কে CPU configure করে বলে দেয়:

* কোন interrupt mask করতে হবে
* কে বেশি priority পাবে
* Edge না Level triggered হবে
* Interrupt vector কোথায় যাবে

## 🧾 সারাংশ:

| **বিষয়** | **ব্যাখ্যা** |
| --- | --- |
| **INTR** | General-purpose, maskable interrupt pin |
| **Trigger Type** | Level-sensitive |
| **Acknowledge Signal** | INTA (Interrupt Acknowledge) |
| **Opcode Provider** | External device/8259 |
| **Software Interrupt** | RST0 – RST7 (with fixed vector addresses) |
| **Interrupt Controller** | 8259A can manage multiple interrupts through INTR |

✅ যদি তুমি চাও, আমি একটা **flowchart বা diagram** বানিয়ে দিতে পারি যাতে পুরো process ভিজুয়ালি বোঝা যায়।

চাও কি সেটা?

অবশ্যই ভাই, এখন আমি ধাপে ধাপে ও গল্পের মতো করে 8085 Microprocessor-এ **INTR interrupt**, **INTA signal**, ও **8259 Interrupt Controller** পুরো বিষয়টা বাংলায় একদম পরিষ্কারভাবে ব্যাখ্যা করছি — যেন তুমি মনে রাখতে পারো।

## 🧠 কল্পনা করো: CPU একটা রাজা

* তুমি ভাবো 8085 CPU এক রাজা।
* রাজা তার নিজের গুরুত্বপূর্ণ কাজ (instructions) করে যাচ্ছে।
* হঠাৎ দরজা দিয়ে কেউ ঢোকার চেষ্টা করছে (external device) — সে চাইছে রাজার সঙ্গে কথা বলতে।
* সে রাজাকে বিরতি দিতে চায়, মানে: একটা Interrupt করতে চায়।

এই Interrupt কে আমরা বলি 👉 **INTR (Interrupt Request)**

## 🔔 ধাপ ১: INTR Signal পাঠানো

* বাহিরের কেউ (যেমন: Keyboard, Timer, Sensor) একটা request পাঠায় CPU-কে — INTR pin-এ HIGH করে।
* এর মানে: "হে CPU! আমি তোমার attention চাই!"

## ⏳ ধাপ ২: CPU তার কাজ শেষ করে

* CPU বলে, “ঠিক আছে, আগে আমি আমার চলমান কাজ শেষ করি।”
* সে যে instruction execute করছিলো, তা শেষ করে।

## 📶 ধাপ ৩: CPU বলে—আমি তোমার কথা শুনতে প্রস্তুত

* CPU এখন INTA নামে একটি signal দেয় (Interrupt Acknowledge) – এর মানে: **"আমি তোমার কথা শোনার জন্য প্রস্তুত"**

## 📦 ধাপ ৪: বাহিরের ডিভাইস CPU-কে একটা কাজ দেয় (Instruction)

* এখন যে বাহিরের device INTR দিয়েছিল, সে INTA পেয়ে CPU-কে **একটি Instruction (Opcode)** পাঠায় — যেমন: RST 7 বা CALL 0038H।

এখানে যেই Instruction CPU-কে দেওয়া হয়, সেটাই ঠিক করে CPU এখন কোথায় যাবে।

## 🛠 যদি ডিভাইস নিজে Instruction না দিতে পারে?

তাহলে আমরা ব্যবহার করি 👉 **8259 Programmable Interrupt Controller**

## 🧰 8259 Interrupt Controller কী করে?

ধরো, অনেকগুলো device আছে – Keyboard, Mouse, Sensor, Timer — সবাই রাজাকে (CPU) তাদের কথা বলতে চায়। কে আগে বলবে?

তখন আমরা বলি, “তোমরা 8259 নামের মন্ত্রীর কাছে যাও, সে তোমাদের হিসাব রাখবে।”

### 8259 এর কাজ:

| **কাজ** | **ব্যাখ্যা** |
| --- | --- |
| 🎯 Priority ঠিক করে | কে আগে interrupt করবে |
| 🔕 কিছু interrupt বন্ধ রাখে | যেগুলো এখন দরকার নেই |
| 🧾 Opcode ঠিক করে | কোন কাজ CPU করবে |
| 📡 CPU-কে INTR দেয় | তারপর INTA পেলে instruction পাঠায় |

## 🧭 Software Interrupt: RST Instructions

8085-এ ৮টা software interrupt আছে:  
**RST 0 – RST 7**  
এদের প্রতিটির fix address আছে, যেমন:

| **RST** | **Vector Address** |
| --- | --- |
| RST 0 | 0000H |
| RST 1 | 0008H |
| RST 2 | 0010H |
| ... | ... |
| RST 7 | 0038H |

👉 এগুলোকে তুমি প্রোগ্রামে INT বা RST instruction দিয়ে নিজে থেকেই trigger করতে পারো।

## 🧾 সংক্ষেপে INTR interrupt flow:

1. External Device: INTR = HIGH দেয়

2. CPU: চলমান কাজ শেষ করে

3. CPU: INTA = LOW দেয় (acknowledge)

4. Device বা 8259: CPU-কে RST/CALL opcode দেয়

5. CPU: সেই address-এ jump করে, ISR চালায়

6. Interrupt কাজ শেষ → CPU আবার আগের কাজ চালায়

## 🖼️ Flowchart চাই?

চাও কি আমি এর একটা **চিত্র বা flowchart** দিই? তাহলে পুরোটা চোখে দেখা যাবে, মনে রাখা আরও সহজ হবে।

আরও কোনো অংশ অস্পষ্ট থাকলে জিজ্ঞেস করো – আমি তোমার জন্য আরো সহজ করেই বোঝাবো।

তুমি যেটা দিয়েছো, সেটা 8085 microprocessor-এ **external interrupt handling with 8259 Interrupt Controller** নিয়ে আলোচনা করছে।

আমি এখন **সহজ বাংলায়, ধাপে ধাপে** বোঝাচ্ছি — যেন তোমার পুরো বিষয়টা একদম ক্লিয়ার হয়।

## 🔰 বিষয়: কিভাবে 8259 ও 8085 মিলে external interrupt handle করে

### 🎯 মূল পয়েন্ট:

* যদি কোনো **high priority এবং unmasked interrupt** আসে,
* তাহলে 8259 → INTR signal পাঠায় 8085-কে।
* 8085 accept করলে → তিনটি INTA signal পাঠায়।
* 8259 → একে একে তিনটি তথ্য পাঠায়:
  1. CALL opcode
  2. CALL address-এর **Low Byte**
  3. CALL address-এর **High Byte**

## 🧠 এখন পুরো process ব্যাখ্যা করছি step-by-step:

### 🟢 Step 1: Interrupt Request

* ধরো, কোনো external device (যেমন: Timer) interrupt করতে চায়।
* সেই device → 8259 কে signal পাঠায়।
* 8259 দেখে — interrupt টি **priority অনুযায়ী high** এবং **mask করা হয়নি**।
* তাই সে **INTR = HIGH** করে দেয় → 8085-কে জানায়: "আমার কাছে একটা জরুরি interrupt আছে।"

### 🔵 Step 2: CPU accept করে

* 8085 তখন তার কাজ শেষ করে,
* এবং accept করার জন্য তিনবার **INTA/** (Interrupt Acknowledge) signal পাঠায় → এটা 8259-এর জন্য।

### 🟣 Step 3: 8259 responds to INTA/

* এখন প্রতি INTA/ signal এর জবাবে 8259 নিচের data দেয়:

| **INTA/ Number** | **8259 কী দেয়?** | **ব্যাখ্যা** |
| --- | --- | --- |
| 1st INTA/ | **CALL Opcode** | সাধারণত CD (CALL instruction) |
| 2nd INTA/ | **Low byte of address** | CALL যেই address-এ jump করবে, তার lower 8-bit |
| 3rd INTA/ | **High byte of address** | CALL address-এর upper 8-bit |

### 🟠 Step 4: 8085 CALL execute করে

* CPU তখন CALL address ইনস্ট্রাকশন চালায়।
* আগে সে তার current **Program Counter (PC)** এর মান **stack-এ রেখে দেয়** (যাতে পরে ফিরে আসতে পারে)।
* তারপর PC তে নতুন CALL address বসিয়ে দেয়।
* এর মানে, এখন CPU jump করে **ISR (Interrupt Service Routine)** চালাতে শুরু করে।

### 🔄 Step 5: Interrupt কাজ শেষ হলে?

* ISR চালানো শেষ হলে, CPU RET (return) ইনস্ট্রাকশন চালায়।
* এতে CPU আবার stack থেকে পুরোনো PC তুলে নিয়ে আগের কাজ শুরু করে।

## 🧰 অতিরিক্ত: Cascading 8259 (64 Interrupts)

* একটি 8259-তে ৮টি interrupt নেওয়া যায় (IR0 - IR7)।
* কিন্তু যদি আরও বেশি device থাকে, তখন **কয়েকটি 8259 চিপ "cascade" করে সংযুক্ত করা যায়**।
* এভাবে **সর্বোচ্চ ৮×৮ = ৬৪টি interrupt** নেওয়া সম্ভব।

## ✅ সারসংক্ষেপ (স্মরণীয়):

| **ধাপ** | **ব্যাখ্যা** |
| --- | --- |
| 1️⃣ | High priority + unmasked interrupt → 8259 → INTR পাঠায় |
| 2️⃣ | 8085 → তিনবার INTA/ পাঠায় |
| 3️⃣ | 8259 → CALL opcode + address পাঠায় |
| 4️⃣ | 8085 → ISR execute করে |
| 5️⃣ | ISR শেষে → return করে আগের প্রোগ্রামে |

চাও কি আমি এর একটা **flowchart বা চিত্র** দিই? তাহলে পুরো process এক নজরে দেখে বোঝা যাবে।

অথবা তুমি চাইলে এর ওপর একটা **short note বা exam answer** লিখেও দিতে পারি।

তুমি যে ছবিটা পাঠিয়েছো, সেটা 8086 Microprocessor-এর "Main Features" এর ওপর লেখা। নিচে আমি পুরোটা **বাংলায় সহজভাবে ব্যাখ্যা করছি**, যাতে তুমি ভালোভাবে বুঝতে পারো ও মনে রাখতে পারো।

## 🧠 ****8086 Microprocessor-এর প্রধান বৈশিষ্ট্য (Main Features of 8086)****

### ✅ 1. 8086 একটি 16-bit প্রসেসর

* এর **ALU (Arithmetic Logic Unit)** এবং **internal registers** সবকিছুই 16-bit ডেটা নিয়ে কাজ করে।
* অর্থাৎ একসাথে 16-bit ডেটা প্রসেস করতে পারে।

### ✅ 2. 16-bit Data Bus

* এর মানে 8086 প্রতি clock cycle-এ **একসাথে 16-bit অথবা 8-bit ডেটা** memory বা I/O port থেকে **পড়তে বা লিখতে** পারে।

### ✅ 3. 20-bit Address Bus

* এর ফলে 8086 সর্বোচ্চ 2^20 = 1 MB মেমরি address করতে পারে।
* অর্থাৎ 1MB পর্যন্ত মেমরিতে সরাসরি এক্সেস দেওয়া যায়।

### ✅ 4. Frequency Range

* 8086-এর কার্যক্ষম গতি বা ফ্রিকোয়েন্সি **6 MHz থেকে 10 MHz** এর মধ্যে হয়।

### ✅ 5. Floating-point Arithmetic করা যায় না (Directly)

* 8086 নিজে floating-point operation করতে পারে না, কারণ সেই সময়ের integrated circuit প্রযুক্তি তেমন উন্নত ছিল না।
* কিন্তু Intel পরে **8087 coprocessor** ডিজাইন করে, যা 8086-এর সাথে কাজ করে complex floating-point operation করতে পারে।

### ✅ 6. দুইটি Mode-এ কাজ করে (Minimum & Maximum Mode)

* **Minimum Mode**:
  + যখন 8086 একাই কাজ করে (single processor system), তখন এই mode ব্যবহার হয়।
  + CPU নিজেই সব control signal generate করে।
* **Maximum Mode**:
  + যখন 8086 একাধিক প্রসেসরের সাথে (multiprocessor system) কাজ করে,
  + তখন 8086 **8087 বা অন্য coprocessor**-এর সাথে control signals শেয়ার করে।

### ✅ 7. Multiprocessor Environment Support করে

* Memory এবং I/O-এর জন্য control signals বাহির থেকে BUS Controller দ্বারা তৈরি করা যায়।

### ✅ 8. Instruction Prefetch Queue

* এটি **up to 6 instruction byte memory থেকে আগেই read করে** queue তে রাখে।
* ফলে **instruction execution দ্রুত হয়**।

### ✅ 9. Power Supply

* 8086 কাজ করতে **+5V power supply** প্রয়োজন হয়।

### ✅ 10. 40-pin Package

* 8086 chip টি একটি **40-pin dual in-line package (DIP)** এ পাওয়া যায়।

### ✅ 11. দুইটি Block আছে: BIU এবং EU

| **Block** | **পূর্ণরূপ** | **কাজ** |
| --- | --- | --- |
| **BIU** | Bus Interface Unit | Memory access, instruction fetch, address calculation |
| **EU** | Execution Unit | Instruction decode, execute, ALU operations |

## 

## 📌 এক নজরে মনে রাখার জন্য:

8086 Summary:

✔️ 16-bit processor & ALU

✔️ 16-bit data bus

✔️ 20-bit address bus → 1MB memory

✔️ 6–10 MHz frequency

✔️ Works in min/max mode

✔️ Supports 8087 coprocessor

✔️ Instruction queue: 6 bytes

✔️ +5V power, 40-pin DIP

✔️ Has BIU & EU units

তুমি চাইলে এই তথ্য দিয়ে আমি **short notes, MCQ preparation, বা diagram** বানিয়ে দিতে পারি। কোন ফরম্যাটে দরকার বলো, সাথে সাথে বানিয়ে দিই!

তুমি যে ছবিটা পাঠিয়েছো, সেখানে **8086 Microprocessor Family—An Overview** নিয়ে বিস্তারিত আলোচনা আছে। আমি এখানে ছবির মূল বিষয়গুলো বাংলায় **সহজ ভাষায়** তুলে ধরছি।

## 📘 ****8086 Microprocessor Family — সারাংশ ও বিশ্লেষণ****

### 🔷 **👉 8086 Overview:**

* **8086** একটি **16-bit microprocessor**, অর্থাৎ এর ALU, রেজিস্টার, ইত্যাদি 16-bit data নিয়ে কাজ করে।
* এর **16-bit data bus** আছে — তাই একবারে 8 বা 16-bit data পড়তে/লিখতে পারে।
* এতে **20-bit address bus** আছে — ফলে সর্বোচ্চ **2²⁰ = 1,048,576 = 1MB** মেমরি অ্যাক্সেস করা যায়।

#### 🔸 Memory Access:

* 8086 মেমরি থেকে **2 byte একসাথে** পড়তে পারে যদি word এর first byte **even address**-এ থাকে।
* যদি **odd address**-এ থাকে, তাহলে প্রথমে 1 byte পড়ে, পরে আরেকবার পড়ে — অর্থাৎ ২টি মেমরি অ্যাক্সেস লাগে।

### 🔷 **👉 8088 Overview:**

* **8088** হচ্ছে 8086-এর একটি সংস্করণ (variant)।
* এর **internal structure 8086-এর মতোই** — ALU, রেজিস্টার, ইত্যাদি সব একই।
* কিন্তু এর **data bus মাত্র 8-bit**, তাই একবারে শুধুমাত্র 1 byte data পড়তে বা লিখতে পারে।
* Address bus 8086-এর মতোই 20-bit → তাই 1MB মেমরি অ্যাক্সেস করতে পারে।
* **8088 = 8086 with 8-bit external data bus**

🔹 উদাহরণ:

IBM PC এবং IBM PC/XT-তে 8088 ব্যবহার করা হয়েছিল।

### 🔷 **👉 80186 ও 80188:**

* এরা হচ্ছে **8086/8088-এর upgraded version**।
* 80186 = 16-bit version, 80188 = 8-bit version (just like 8086 vs 8088)।
* এর মধ্যে কিছু **additional hardware features** build-in করা আছে — যেমন:
  + Timer
  + Interrupt controller
  + DMA controller
* তাই এগুলো embedded systems-এ বেশি ব্যবহৃত হয়।

### 🔷 **👉 80286:**

* **8086-এর উন্নত সংস্করণ**, 16-bit processor হলেও এতে রয়েছে:
  + **Real Mode**: যেখানে এটা 8086-এর মতোই behave করে।
  + **Protected Mode**: যেখানে user program ও system program আলাদা রাখা যায় (memory protection)।
* এটি **IBM PC/AT** কম্পিউটারে ব্যবহৃত হয়েছিল।

### 🔷 **👉 80386:**

* **Intel-এর প্রথম 32-bit processor**, যা **IA-32 architecture** শুরু করে।
* Features:
  + 32-bit data bus & 32-bit address bus → ৪GB পর্যন্ত মেমরি address করা যায়।
  + 16-bit প্রোগ্রামগুলোর সাথে compatible
  + Multitasking এবং multiuser সাপোর্ট করে

### 🔷 **👉 80486:**

* 80386-এর পরবর্তী সংস্করণ
* এতে **Floating Point Unit (FPU)** (যেমন: 80387) **CPU-এর ভিতরেই** থাকে — আলাদা coprocessor লাগে না।
* আরও দ্রুত execution ও উন্নত ক্ষমতা

### 🔷 👉 এরপর আসে: **Pentium Series**

* আরও উন্নততর প্রসেসর, multiple cores, ক্যাশ মেমরি, parallel execution ইত্যাদি ফিচার সহ।

## 🧾 Summary Table:

| **Processor** | **Data Bus** | **Address Bus** | **Memory Access** | **Notes** |
| --- | --- | --- | --- | --- |
| 8086 | 16-bit | 20-bit (1MB) | Fast (even addr) | Base CPU |
| 8088 | 8-bit | 20-bit | Slower | Used in IBM PC |
| 80186 | 16-bit | 20-bit | + Built-in peripherals | Embedded systems |
| 80188 | 8-bit | 20-bit | + Built-in peripherals | Embedded systems |
| 80286 | 16-bit | 24-bit (16MB) | Real + Protected Mode | IBM PC/AT |
| 80386 | 32-bit | 32-bit (4GB) | IA-32 start | Multitasking |
| 80486 | 32-bit | 32-bit | Built-in FPU | Faster, compact |

## 🔚 শেষ কথা:

এই ফ্যামিলি বোঝা মানে, তুমি Intel microprocessors-এর **পুরো প্রজন্ম** একসাথে বুঝে ফেলছো।

তুমি চাইলে আমি এই topic থেকে:

* 📑 Short note
* 🧠 MCQ প্রশ্ন
* 🧭 Timeline বা Chart

তৈরি করে দিতে পারি। কোনটা দরকার?

ঠিক আছে, তাহলে আমরা তোমার দেওয়া লেখাটা ধাপে ধাপে বাংলায় সহজভাবে ব্যাখ্যা করি — যেন তুমি বোঝার পাশাপাশি মনে রাখতেও পারো।

**BIU ও QUEUE**

8086 মাইক্রোপ্রসেসরে **BIU (Bus Interface Unit)** আর **EU (Execution Unit)** – এই দুই অংশ একসাথে কাজ করে।

* **কাজের ধরণ**  
  যখন **EU** কোনো ইনস্ট্রাকশন **ডিকোড** করছে বা এক্সিকিউট করছে, আর ওই ইনস্ট্রাকশনে **বাস (Bus)** দরকার নেই, তখন **BIU** আগেই পরের ইনস্ট্রাকশনের **বাইটগুলো মেমরি থেকে নিয়ে আসে**।
* এই বাইটগুলো **Queue** নামের **FIFO (First In, First Out)** রেজিস্টার সেটে জমা থাকে।
* যখন **EU**-এর পরের ইনস্ট্রাকশন দরকার হয়, তখন মেমরিতে গিয়ে অপেক্ষা করার বদলে **সরাসরি Queue থেকে বাইট পড়ে নেয়** — ফলে অনেক দ্রুত কাজ হয়।

এটা অনেকটা ইট-বাঁধাই মিস্ত্রির সাহায্যকারীর মতো — যে আগে থেকেই ইট এনে সারিবদ্ধ করে রাখে, যাতে মিস্ত্রি হাত বাড়ালেই ইট পায়।

* শুধু **JMP** বা **CALL** এর মতো ইনস্ট্রাকশন এলে Queue ফাঁকা করে নতুন অ্যাড্রেস থেকে আবার বাইট আনতে হয়।
* একে বলা হয় **pipelining** — মানে পরের ইনস্ট্রাকশন ফেচ করতে করতে আগের ইনস্ট্রাকশন এক্সিকিউট করা।

**SEGMENT REGISTERS**

* 8086-এর **BIU** ২০-বিট অ্যাড্রেস পাঠাতে পারে, মানে **১ MB (1,048,576 bytes)** মেমরি অ্যাক্সেস করা যায়।
* কিন্তু একসাথে 8086 কেবল **৪টি 64 KB সেগমেন্ট** নিয়ে কাজ করে।
* এই ৪টি সেগমেন্টের জন্য ৪টি **segment register** থাকে:
  1. **CS (Code Segment)** – ইনস্ট্রাকশন কোড যেখানে থাকে
  2. **SS (Stack Segment)** – স্ট্যাক ডেটার জন্য
  3. **DS (Data Segment)** – সাধারণ ডেটার জন্য
  4. **ES (Extra Segment)** – অতিরিক্ত ডেটার জন্য
* **Segment Register-এর কাজ**  
  প্রতিটি segment register উপরের ১৬ বিট ধরে রাখে, যা সেগমেন্টের শুরুর ঠিকানা (segment base) নির্দেশ করে।  
  8086 সবসময় শুরুর ঠিকানার নিচের ৪ বিট **0** রাখে।  
  যেমন, যদি **CS = 348AH** হয়, তাহলে কোড সেগমেন্ট শুরু হবে **348A0H** অ্যাড্রেস থেকে।
* এর ফলে সেগমেন্ট মেমরির যেকোনো জায়গায় বসানো যায়, কিন্তু সবসময় ১৬-বাইট অ্যালাইন্ড অবস্থায় শুরু হয়।
* এই ডিজাইন করা হয়েছে যাতে সেগমেন্ট অ্যাড্রেস স্টোর ও ম্যানিপুলেট করার সময় ১৬-বিট ব্যবহার করলেই হয়।

Instruction Pointer (IP) র কী কাজ তা চল বোঝাই বাংলা তে —

### 🔧 **Instruction Pointer (IP) Register কী?**

* 8086 processor এ মেমোরি address হয় মোট **২০-bit physical address** দিয়ে।
* এই address বানানোর জন্য – দুইটা রেজিস্টার কাজ করে:
  1. **CS (Code Segment)** → উপরের (Higher) ১৬-bit রাখে
  2. **IP (Instruction Pointer)** → Offset বা নিচের অংশের ১৬-bit রাখে

### 🧩 **কাজ করার নিয়ম:**

1. **CS Register**
   * এটি ১৬-bit হয়
   * কিন্তু এর শেষে **৪টা ০ (zero)** বসিয়ে **২০-bit base address** বানানো হয়
   * উদাহরণ:  
     যদি CS = 348A হয় → তাহলে সেই segment শুরু হয়:  
     → 348A0H (পেছনে একটাসহ মোট চারটি ০ যোগ হয়)
2. **IP Register**
   * এটি ১৬-bit offset রাখে, অর্থাৎ **segment এর ভিতরে কত দূরে পরের instruction আছে** সেটা বলে।
   * উদাহরণ:  
     যদি IP = 4214H হয়
3. **Physical Address তৈরি হয়:**
4. Physical Address = (CS × 10H) + IP
5. = 348A0H + 4214H
6. = 38AB4H

### ✅ তাই,

| **Register** | **মান** | **অর্থ** |
| --- | --- | --- |
| CS | 348A | Code segment-এর শুরু (base) |
| IP | 4214 | Offset → Segment-এর ভিতরে কত দূরে পরের instruction আছে |
| Physical Address | 38AB4H | যেখানে থেকে instruction fetch হবে |

### 📌 আরেকভাবে লেখা হয় → CS : IP format এ

যেমন: 348A:4214 → এর Physical address 38AB4H

### 🧠 সংক্ষেপে মনে রাখো:

* **CS** → Segment কোথা থেকে শুরু হচ্ছে (base address এর upper 16 bits)
* **IP** → সেই segment এর ভিতরে পরের instruction কোথায় আছে (offset)
* দুইটা মিলে তৈরি হয় **২০-bit physical address**, যেটা দিয়ে CPU instruction fetch করে।

**STACK SEGMENT (SS) এবং STACK POINTER (SP) – বাংলায় সহজে বোঝানো হলোঃ**

### 🥞 **Stack কী ?**

Stack হলো মেমোরির একটি বিশেষ জায়গা যেখানে **subroutine (function)** চলার সময় অস্থায়ী ডাটা ও রিটার্ন অ্যাড্রেস রাখা হয়।

👉 Stack কাজ করে **LIFO** (Last In First Out) নিয়মে — শেষ যেটা ঢুকবে, আগে সেটা বের হবে।

### 📌 **SS (Stack Segment Register) এর কাজঃ**

* পুরো Stack রাখার জন্য 8086 একটি **64KB সেগমেন্ট ব্যবহার করে।**
* এই Stack Segment এর **শুরু অ্যাড্রেসের উপরের ১৬-bit রাখা থাকে SS রেজিস্টারে**।
* যেমনঃ যদি SS = 5000H হয় → তাহলে Stack Segment এর শুরু physical base address = 50000H

### 📌 **SP (Stack Pointer Register) এর কাজঃ**

* SP হলো ১৬-bit offset যা বলে Stack Segment এর মধ্যে ঠিক **কোথায় (কত দূরে) stack-এর উপরে কোনো ডাটা রাখা হয়েছে।**
* অর্থাৎ **Top of Stack** বর্তমানে মেমোরির কোন জায়গায় আছে SP তা নির্দেশ করে।

### 🧮 **Physical Stack Address তৈরির নিয়মঃ**

Physical Address = (SS × 10H) + SP

➡️ উদাহরণঃ

* SS = 5000H → Base = 50000H
* SP = 0020H → Offset
* Physical Address = 50000H + 0020H = 50020H

অর্থাৎ নতুন ডাটা push করলে 50020H অ্যাড্রেসে যাবে।

### 📊 সংক্ষেপে মনে রাখার টিপসঃ

| **Register** | **কি নির্দেশ করে?** | **ব্যাখ্যা** |
| --- | --- | --- |
| SS | Stack segment কোথায় শুরু | মেমোরির উপরের ১৬-বিট base |
| SP | Stack এর টপ কোথায় আছে | Stack-এর ভিতরে offset |
| Physical Address | SS ও SP যোগ করে | যেখানে ডাটা push/pop হবে |

চাও কি আমি PUSH ও POP Instruction দিয়ে stack কিভাবে কাজ করে সেটাও বাংলা উদাহরণ দিয়ে বুঝিয়ে দিই? 😊

ঠিক আছে, আমি তোমাকে **Assembly Language Statement** এর এই অংশ বাংলায় সহজভাবে বুঝিয়ে দিচ্ছি।

## 🔹 অ্যাসেম্বলি ল্যাঙ্গুয়েজ স্টেটমেন্টের ৪টি অংশ

অ্যাসেম্বলি প্রোগ্রামের প্রতিটি স্টেটমেন্ট সাধারণত **চারটি ফিল্ডে** লেখা হয় —

### 1️⃣ **Label Field (লেবেল ফিল্ড)**

* **লেবেল** হলো একটি নাম বা চিহ্ন যা কোনো **ঠিকানা (address)** বোঝায়।
* প্রোগ্রাম লেখার সময় সেই ঠিকানা সঠিকভাবে জানা না থাকলেও লেবেল ব্যবহার করা যায়।
* লেবেলের পরে সাধারণত **colon ( : )** দেওয়া হয়।
* লেবেল বাধ্যতামূলক নয়, শুধু যখন দরকার তখনই দেওয়া হয়।
* পরে আমরা দেখব লেবেলের অনেক ব্যবহার আছে (যেমন: জাম্প বা লুপে গন্তব্য চিহ্নিত করা)।

### 2️⃣ **Opcode Field (অপকোড ফিল্ড)**

* এখানে থাকে **ইনস্ট্রাকশনের সংক্ষিপ্ত নাম (mnemonic)** যা বলে দেয় কী কাজ করতে হবে।
* একে **Operation Code** বা **Opcode**-ও বলা হয়।
* উদাহরণ: ADD → যোগ করার নির্দেশ।

### 3️⃣ **Operand Field (অপারেন্ড ফিল্ড)**

* এখানে থাকে সেই **ডেটা**, **মেমরি অ্যাড্রেস**, **পোর্ট অ্যাড্রেস** বা **রেজিস্টারের নাম** যার ওপর অপারেশন হবে।
* **Operand** মানে হলো ইনস্ট্রাকশন যেটার ওপর কাজ করবে।
* উদাহরণ:
* ADD AL, 07H
  + AL = AL রেজিস্টার
  + 07H = হেক্সাডেসিমাল 07 (ডেসিমালে 7)
  + এই ইনস্ট্রাকশন মানে → AL রেজিস্টারের কন্টেন্টের সাথে 07H যোগ করো।
  + **Intel convention** অনুযায়ী → কমার আগে যেটা থাকে, **ফলাফল সেখানেই সেভ হয়**।
  + তাই এখানে ফলাফল AL রেজিস্টারেই থাকবে।

### 4️⃣ **Comment Field (মন্তব্য অংশ)** (যেটা তোমার লেখায় উল্লেখ নেই, কিন্তু চার ফিল্ডের অংশ)

* এখানে প্রোগ্রামারের জন্য মন্তব্য লেখা হয়, যাতে কোড পড়তে সুবিধা হয়।
* মেশিন একে উপেক্ষা করে।

## 🔹 আরও একটি উদাহরণ:

ADD BH, AL

* AL রেজিস্টারের কন্টেন্ট BH রেজিস্টারে যোগ হবে।
* ফলাফল BH রেজিস্টারেই সেভ হবে (কারণ BH কমার আগে আছে)।

যদি তুমি চাও, আমি এখন তোমার জন্য **চার ফিল্ডের একটা টেবিল** ও **ভিজ্যুয়াল ডায়াগ্রাম** বানিয়ে দিতে পারি যাতে Label–Opcode–Operand–Comment এক নজরে বোঝা যায়।  
তুমি কি সেটা চাইছ?

ঠিক আছে, আমি তোমাকে এই অংশটা বাংলায় সহজভাবে ব্যাখ্যা করছি।

## 🔹 অ্যাসেম্বলি প্রোগ্রামকে মেশিন ল্যাঙ্গুয়েজে রূপান্তর

তুমি হয়তো ভাবছো —

“আমি যদি অ্যাসেম্বলি ভাষায় প্রোগ্রাম লিখি, তাহলে কীভাবে সেটা **মেশিন ল্যাঙ্গুয়েজে** রূপান্তর করব, যা মাইক্রোকম্পিউটারে লোড করে রান করানো যাবে?”

এর উত্তর দু’ভাবে দেওয়া যায় —

### **পদ্ধতি ১: ম্যানুয়ালি কোড রূপান্তর**

* প্রতিটি ইনস্ট্রাকশনের জন্য **বাইনারি কোড (0 ও 1)** বের করা, এক বিট করে।
* এর জন্য ম্যানুফ্যাকচারারের **ডেটা বুক** বা **ইনস্ট্রাকশন সেট টেমপ্লেট** ব্যবহার করা হয়।
* তবে এটি অনেক সময়সাপেক্ষ এবং ভুল হওয়ার সম্ভাবনা বেশি।
* বইয়ের পরের অধ্যায়ে এই পদ্ধতি দেখানো হবে।

### **পদ্ধতি ২: Assembler ব্যবহার**

* **Assembler** হলো একটি প্রোগ্রাম, যা কম্পিউটারে চালানো যায়।
* এটি তোমার লেখা **অ্যাসেম্বলি ইনস্ট্রাকশন** পড়ে প্রতিটি ইনস্ট্রাকশনের সঠিক **বাইনারি মেশিন কোড** তৈরি করে।
* জটিল অ্যাসেম্বলি প্রোগ্রাম ডেভেলপ করার জন্য **assembler** এবং অন্যান্য **development tools** প্রয়োজন।
* বইয়ের পরবর্তী অধ্যায়ে এসব টুলের পরিচয় এবং ব্যবহার দেখানো হবে।

🔹 সহজভাবে:  
ম্যানুয়ালি রূপান্তর মানে **তুমি নিজে হাতে ইনস্ট্রাকশনের কোড তৈরি করবে**।  
Assembler ব্যবহার মানে **কম্পিউটার তোমার হয়ে কাজ করবে**।

আমি চাইলে তোমার জন্য এখানে **ম্যানুয়াল কোডিং বনাম Assembler দিয়ে কোডিং** এর তুলনামূলক একটি টেবিল বানিয়ে দিতে পারি।  
তাহলে তুমি দুই পদ্ধতির পার্থক্য চোখে দেখেই বুঝতে পারবে।  
তুমি কি সেটা চাও?

**Immediate Addressing Mode** খুবই সহজ একটি অ্যাড্রেসিং পদ্ধতি, আমি বাংলায় ধাপে ধাপে বুঝিয়ে দিচ্ছি।

**🔹 সংজ্ঞা**

Immediate Addressing Mode-এ **ডেটা (constant value)** সরাসরি ইনস্ট্রাকশনের ভেতরে লেখা থাকে।  
মানে CPU-কে আলাদা করে মেমরি থেকে ডেটা আনতে হয় না — ইনস্ট্রাকশন পড়ার সাথেই ডেটা পাওয়া যায়।

**🔹 বৈশিষ্ট্য**

* **Operand** (যেটার ওপর কাজ হবে) সরাসরি ইনস্ট্রাকশনের অংশ।
* মেমরি লোকেশন বা রেজিস্টার থেকে ডেটা আনতে হয় না।
* গতি বেশি, কারণ অতিরিক্ত মেমরি অ্যাক্সেস লাগে না।
* ডেটা **পরিবর্তনযোগ্য নয়** — কম্পাইল/অ্যাসেম্বল করার সময়ই স্থির হয়ে যায়।

**🔹 উদাহরণ (8086 Assembly)**

MOV AL, 05H

* এখানে 05H (হেক্সাডেসিমালে 5) হলো Immediate Data।
* CPU সরাসরি AL রেজিস্টারে 05H লোড করবে।
* AL-এ মান বসাতে আলাদা মেমরি অ্যাড্রেস খুঁজতে হবে না।

**🔹 আরেকটি উদাহরণ**

ADD AX, 1234H

* AX রেজিস্টারের কন্টেন্টের সাথে সরাসরি 1234H যোগ হবে।
* এখানে 1234H Immediate Operand।

**🔹 চেনার উপায়**

* অপারেন্ড হিসেবে যদি **#value**, **valueH**, বা সরাসরি সংখ্যা থাকে, তবে তা Immediate Addressing Mode।
* যেমন: MOV CL, 0AH → Immediate।

তুমি চাইলে আমি এখন তোমাকে **Immediate, Direct, Indirect, Register Addressing Mode** এর পার্থক্য একটা টেবিলে সাজিয়ে দিতে পারি, যাতে তুমি একবারেই পুরো Addressing Mode বুঝে ফেলতে পারো।

ঠিক আছে, এবার আমি তোমাকে **Register Addressing Mode** সহজভাবে বোঝাই।

**🔹 সংজ্ঞা**

**Register Addressing Mode**-এ ডেটা প্রসেসরের **রেজিস্টারের মধ্যে** থাকে, আর ইনস্ট্রাকশনে সরাসরি সেই রেজিস্টারের নাম দেওয়া হয়।  
এখানে **Operand** মেমরিতে নয়, CPU-র অভ্যন্তরের রেজিস্টারে অবস্থান করে।

**🔹 বৈশিষ্ট্য**

* অপারেন্ড হিসেবে **রেজিস্টারের নাম** ব্যবহৃত হয়।
* মেমরি অ্যাক্সেস লাগে না → তাই **খুব দ্রুত কাজ করে**।
* ডেটা CPU-র রেজিস্টারে থাকায় এক্সিকিউশন টাইম কম হয়।
* যেকোনো **General Purpose Register** (যেমন: AX, BX, CX, DX, AL, BL ইত্যাদি) বা Segment Register ব্যবহার করা যেতে পারে (ইনস্ট্রাকশন টাইপ অনুযায়ী)।

**🔹 উদাহরণ (8086 Assembly)**

MOV AX, BX

* এখানে BX রেজিস্টারের কন্টেন্ট AX রেজিস্টারে কপি হবে।
* কোনো মেমরি লোকেশন থেকে ডেটা আনতে হচ্ছে না।

আরেকটা উদাহরণ:

ADD AL, BL

* AL রেজিস্টারের সাথে BL রেজিস্টারের ডেটা যোগ হবে।
* ফলাফল AL রেজিস্টারে সেভ হবে।

**🔹 চেনার উপায়**

* অপারেন্ডে **রেজিস্টারের নাম** দেওয়া থাকে, কোনো সংখ্যা বা মেমরি অ্যাড্রেস দেওয়া থাকে না।
* যেমন: MOV CX, DX → Register Addressing Mode।

খুব সুন্দর প্রশ্ন করেছো 🙂 এটা বোঝা গেলে CPU-এর ভেতরের কাজ অনেক পরিষ্কার হয়ে যাবে।

**🔹 Register vs Memory (Main Difference)**

| **বিষয়** | **Register** | **Memory (RAM)** |
| --- | --- | --- |
| **অবস্থান** | CPU-এর ভেতরে (Processor chip-এর অংশ) | CPU-এর বাইরে (Main system memory) |
| **গতি** | অনেক দ্রুত (nanosecond লেভেল) | তুলনামূলক ধীর (nanosecond–microsecond লেভেল) |
| **সাইজ/ক্ষমতা** | খুব ছোট (কয়েক বাইট থেকে কয়েকশো বাইট, যেমন 32-bit বা 64-bit রেজিস্টার) | অনেক বড় (GB/ TB পর্যন্ত হতে পারে) |
| **ব্যবহার** | সরাসরি instruction execute করার জন্য মান ধরে | ডেটা, প্রোগ্রাম, OS ইত্যাদি সংরক্ষণের জন্য |
| **অ্যাক্সেস** | সরাসরি CPU access করে | CPU → Memory Bus → RAM access করে |
| **খরচ** | বানানো খুব ব্যয়বহুল (fast SRAM technology) | তুলনামূলক সস্তা (DRAM technology) |

**🔹 সহজভাবে ভাবো**

* **Register = CPU-র হাতের তালু** 🖐️ → যা মুহূর্তে দরকার, একেবারে কাছে থাকে।
* **Memory (RAM) = CPU-র ডেস্ক/টেবিল** 🪑 → অনেক জিনিস রাখা যায়, কিন্তু নিতে-দিতে সময় লাগে।

👉 তাই register খুব দ্রুত কিন্তু সংখ্যা কম।  
👉 memory ধীর হলেও অনেক বড়।

চাও আমি তোমাকে একটা **Layer আকারে** CPU cache–register–RAM–storage এর hierarchy বানিয়ে দিই যাতে বোঝা যায় কারটা দ্রুত আর কারটা বড়? 🚀

| **বৈশিষ্ট্য** | **Immediate Addressing Mode** | **Register Addressing Mode** |
| --- | --- | --- |
| **ডেটা কোথায় থাকে** | নির্দেশনায় (instruction) সরাসরি ডেটা লেখা থাকে | ডেটা প্রসেসরের একটি রেজিস্টারে থাকে |
| **উদাহরণ** | MOV R1, #5 → এখানে #5 সরাসরি ডেটা | MOV R1, R2 → এখানে ডেটা আছে R2 তে |
| **গতি** | ধীরগতি, কারণ ডেটা ইনস্ট্রাকশনে এম্বেডেড থাকে | দ্রুত, কারণ CPU সরাসরি রেজিস্টার থেকে ডেটা নেয় |
| **মেমোরি ব্যবহার** | বেশি, কারণ ডেটা ইনস্ট্রাকশনের অংশ | কম, কারণ ইনস্ট্রাকশনে শুধু রেজিস্টারের ঠিকানা থাকে |
| **ফ্লেক্সিবিলিটি** | ডেটা পরিবর্তন করতে হলে ইনস্ট্রাকশন বদলাতে হয় | ডেটা পরিবর্তন সহজ, শুধু রেজিস্টারে নতুন মান লোড করতে হয় |
| **প্রধান সুবিধা** | সহজ ও সরাসরি মান সেট করা যায় | দ্রুত অপারেশন এবং মেমোরি সাশ্রয় হয় |
| **প্রধান অসুবিধা** | বারবার ব্যবহারে মেমোরি অপচয় | সীমিত রেজিস্টারের কারণে জায়গা কম পাওয়া যায় |

তুমি যেটা লিখেছো, সেটা মূলত **8086 মাইক্রোপ্রসেসরে মেমোরি অ্যাক্সেস করার প্রক্রিয়া এবং Addressing Modes-এর ভূমিকা** ব্যাখ্যা করছে।  
আমি এটা সহজ ভাষায় তোমার জন্য সংক্ষেপে ব্যাখ্যা করছি—

### **Accessing Data in Memory – 8086 এ**

* **Operand** (যে ডেটার উপর কাজ করতে হবে) মেমোরিতে কোথায় আছে তা বোঝাতে **Addressing Mode** ব্যবহার করা হয়।
* 8086 প্রসেসরে **20-bit Physical Address** লাগে মেমোরি অ্যাক্সেস করার জন্য।
* এই 20-bit অ্যাড্রেস তৈরি হয়:
  1. **Segment Base Address** (16-bit, আসে Segment Register থেকে)
  2. **Effective Address (EA)** (16-bit, offset/displacement)

### **Formula**

Physical Address=(Segment Base×10H)+Effective Address\text{Physical Address} = (\text{Segment Base} \times 10H) + \text{Effective Address}

* Segment Base Address × 10H মানে হেক্সাডেসিমাল শিফট (একটি নিবার বামে সরানো)।
* EA হল offset, যা ইনস্ট্রাকশন থেকে আসে।

### **Example (তোমার দেওয়া টেক্সট থেকে)**

* **Segment Base** = 2000H → Segment Register-এ আছে
* **Effective Address (EA)** = 437AH
* Physical Address:

2000H×10H=20000H20000H+437AH=2437AH2000H \times 10H = 20000H 20000H + 437AH = 2437AH

* **দেখানো দুইভাবে:**
  + একক সংখ্যা → 2437AH
  + Segment:Offset → 2000:437AH

### **সংক্ষেপে প্রক্রিয়া**

1. CPU ইনস্ট্রাকশন পড়ে EA নির্ধারণ করে।
2. Segment Register থেকে Segment Base নেয়।
3. দুটো যোগ করে 20-bit Physical Address বানায়।
4. এই অ্যাড্রেস দিয়ে মেমোরি থেকে ডেটা আনা হয়।

**🔹 Direct Addressing Mode কী?**

* Instruction-এ সরাসরি **একটা 16-bit memory address (displacement)** লেখা থাকে।
* এই displacement-কে CPU **segment base address (সাধারণত DS)** এর সাথে যোগ করে → 20-bit **physical address** বের করে।
* তারপর ওই address থেকে ডেটা নিয়ে register-এ রাখে (বা register থেকে memory-তে রাখে)।

👉 একে *direct* বলা হয় কারণ displacement (address offset) সরাসরি instruction-এর ভেতর দেওয়া থাকে।

**🔹 উদাহরণ ১:**

MOV BL, [437AH]

মানে → DS:437AH এ যে byte আছে, সেটাকে BL রেজিস্টারে কপি করো।

* DS (Data Segment register) = segment base address ধরে।
* Offset (displacement) = 437AH
* Effective Address = DS × 10H + 437AH
* BL = ঐ address-এর content

**🔹 উদাহরণ ২:**

MOV BX, [437AH]

এখানে BX = 16-bit রেজিস্টার, তাই memory থেকে **দুইটা byte** পড়তে হবে:

* BL = DS:437AH এর byte (low byte → low address)
* BH = DS:437BH এর byte (high byte → high address)

👉 8086 সবসময় **Low byte → Low address** আর **High byte → High address** এ রাখে।

আচ্ছা, তুমি Example 2 (“MOV BX, [437AH]”) নিয়ে **কেন দুইটা byte লাগে?** সেটা জানতে চাইছো 🙂

### 🔹 কারণটা হলো BX = 16-bit register

* BX রেজিস্টার = 16-bit = 2 byte জায়গা।
* কিন্তু 8086-এর memory হলো **byte-addressable** → প্রতিটি address = 1 byte।

তাহলে BX পূর্ণ করতে CPU-কে **একটার বেশি address থেকে ডেটা আনতে হবে**।

### 🔹 কিভাবে আনে?

MOV BX, [437AH]

* BL (lower 8-bit) = DS:437AH (low address থেকে 1 byte আনে)
* BH (upper 8-bit) = DS:437BH (তারপর সাথে সাথেই পরের higher address থেকে 1 byte আনে)

👉 মোট 2 byte মিলে একটা 16-bit word গঠন করে BX-এ ঢোকে।

### 🔹 কেন Low byte → Low address?

কারণ 8086 **little-endian architecture** ব্যবহার করে।  
মানে:

* **Low-order byte (BL)** → ছোট address (437AH)
* **High-order byte (BH)** → বড় address (437BH)

**🔹 উদাহরণ ৩:**

MOV [437AH], BX

এবার উল্টো কাজ হবে → BX রেজিস্টার থেকে memory-তে কপি করবে:

* BL → DS:437AH
* BH → DS:437BH

**🔹 মনে রাখার টিপ**

👉 Direct addressing mode-এ সবসময় এই formula কাজ করে:

Physical Address = (Segment Register × 10H) + Displacement

👉 Displacement সরাসরি instruction-এর ভেতরে দেওয়া থাকে।

📌 সংক্ষেপে: **Direct Addressing Mode** মানে হলো instruction-এ সরাসরি memory offset লেখা থাকে। CPU সেটা segment base-এর সাথে যোগ করে physical address বের করে → তারপর memory থেকে ডেটা নেয় বা memory-তে ডেটা লেখে।

চাও আমি তোমাকে একটা ছোট **চিত্র (diagram)** এঁকে দেখাই যেখানে DS base + displacement মিলে physical address বের হচ্ছে? ✍️📊