অবশ্যই! আমি ধাপে ধাপে বাংলায় সহজ করে বুঝিয়ে দিচ্ছি —

**x86/x64 CPU রেজিস্টারগুলো কি?**

কম্পিউটারের প্রসেসরের (CPU) ভিতরে **রেজিস্টার** নামে ছোট ছোট স্মৃতিস্তম্ভ থাকে, যেখানে অল্প পরিমাণ ডাটা খুব দ্রুত রাখা হয়। এগুলোকে বিভিন্ন কাজে ব্যবহার করা হয়, যেমন গাণিতিক অপারেশন, মেমরি লোকেশন ধরে রাখা, ইত্যাদি।

**RAX, RBX, RCX, RDX ইত্যাদি রেজিস্টারগুলো কি?**

* এগুলো **জেনারেল-পারপাস রেজিস্টার** অর্থাৎ যেকোনো ধরণের ডাটা সাময়িক রাখার জন্য ব্যবহৃত হয়।
* এগুলোর আকার বা সাইজ বিভিন্ন হতে পারে — 64 বিট, 32 বিট, 16 বিট, বা 8 বিট (মাঝে মাঝে ভাগ করা যায়)।
* প্রতিটি রেজিস্টারের আলাদা আলাদা কাজের গুরুত্ব আছে।

**প্রতিটি রেজিস্টার এর কাজ ও ব্যবহার (সাধারণ ভাষায়) —**

| **রেজিস্টার** | **বাংলা অর্থ ও ব্যবহার** |
| --- | --- |
| **RAX** | এক ধরনের **অ্যাকিউমুলেটর**। গুণ, ভাগ, এবং কিছু স্পেশাল ইনস্ট্রাকশনে ব্যবহার হয়। 64-বিট পর্যন্ত ডাটা ধরে রাখতে পারে। |
| **RBX** | বেস রেজিস্টার, মেমরি লোকেশন বা ডাটা ধরে রাখতে পারে। |
| **RCX** | কাউন্টার রেজিস্টার। লুপ, শিফট (shift), রোটেট (rotate) অপারেশনে কাউন্ট হিসেবে ব্যবহার হয়। স্ট্রিং ইনস্ট্রাকশনের রিপিট কাউন্টার হিসেবেও কাজ করে। |
| **RDX** | ডাটা রেজিস্টার। গুণ বা ভাগের অপারেশনে ফলাফল বা ডিভাইডেন্ড (dividend) হিসেবে ব্যবহৃত হয়। |
| **RBP** | বেস পয়েন্ট পয়েন্টার। ফাংশনের স্ট্যাক ফ্রেমের ঠিকানা ধরে রাখে। |
| **RDI** | ডেস্টিনেশন ইন্ডেক্স। স্ট্রিং অপারেশনে গন্তব্যের ঠিকানা ধরে রাখে। |
| **RSI** | সোর্স ইন্ডেক্স। স্ট্রিং অপারেশনে সোর্সের ঠিকানা ধরে রাখে। |
| **R8 - R15** | 64-বিট প্রসেসরে নতুন অতিরিক্ত জেনারেল-পারপাস রেজিস্টার। এখনো বেশি ব্যবহৃত হয় না। |

**রেজিস্টারগুলো ভিন্ন ভিন্ন আকারে ব্যবহার করা যায়:**

* 64-বিট: যেমন RAX, RBX, RCX, ইত্যাদি
* 32-বিট: EAX, EBX, ECX, ইত্যাদি
* 16-বিট: AX, BX, CX, ইত্যাদি
* 8-বিট: AH (উপরের ৮ বিট), AL (নিচের ৮ বিট), BH, BL, ইত্যাদি

যখন আপনি ছোট আকারের রেজিস্টার ব্যবহার করেন, তখন শুধু ওই অংশের বিট পরিবর্তন হয়, বাকিগুলো অপরিবর্তিত থাকে।

**উদাহরণ:**

* **RAX** রেজিস্টারের নিচের ৮ বিট AL নামে পরিচিত। যদি AL-তে কিছু মান দিন, তাহলে শুধুমাত্র নিচের ৮ বিট পরিবর্তিত হবে, উপরের অংশ আগের মতো থাকবে।
* **RCX** রেজিস্টার লুপ কাউন্টার হিসেবে ব্যবহৃত হয়। ধরুন, আপনি ১০ বার কোন কাজ করতে চান, তখন RCX-তে ১০ সেট করবেন, এবং প্রতিবার ১ করে কমে যাবে।

**64-বিট প্রসেসর:**

Pentium 4, Core2 এর মতো 64-বিট প্রসেসরে RAX, RBX, RCX ইত্যাদি রেজিস্টার 64-বিট ডাটা ধরে রাখতে পারে। এর ফলে খুব বড় মেমরি অ্যাড্রেসও রাখতে সক্ষম হয়।

**সংক্ষেপে:**

* **RAX** = এক ধরনের অ্যালকুলেটর, গুণ/ভাগে ব্যবহার হয়
* **RBX** = মেমরি লোকেশন ধরে রাখে
* **RCX** = কাউন্টার, লুপ ইত্যাদিতে ব্যবহার হয়
* **RDX** = গুণ/ভাগের ফলাফল বা ডিভাইডেন্ড ধরে রাখে
* **RBP** = ফাংশনের স্ট্যাক ফ্রেমের ঠিকানা ধরে রাখে
* **RDI** ও **RSI** = স্ট্রিং ডাটা সোর্স ও ডেস্টিনেশন ঠিকানা ধরে রাখে
* **R8–R15** = নতুন 64-বিট জেনারেল রেজিস্টার, ভবিষ্যতে ব্যবহৃত হবে বেশি

**RIP (Instruction Pointer)**

* RIP মানে হলো **Instruction Pointer** (যে রেজিস্টার প্রসেসরকে বলে দিচ্ছে পরের কোন নির্দেশনা/ইনস্ট্রাকশন মেমরি থেকে আনা হবে)।
* পুরনো প্রসেসরগুলোতে এর আলাদা নাম ছিল:
  + 16-bit হলে: **IP**
  + 32-bit হলে: **EIP**
  + 64-bit হলে: **RIP**
* কাজ: সব সময় প্রোগ্রামের **পরবর্তী নির্দেশনার ঠিকানা** ধরে রাখে।
* Jump বা Call দিলে RIP পরিবর্তন হয়, কারণ তখন অন্য জায়গা থেকে নির্দেশনা নিতে হয়।
* 64-bit প্রসেসরে RIP এখন **40-bit address space** (প্রায় 1TB পর্যন্ত মেমরি অ্যাড্রেস করতে পারে)।

**RSP (Stack Pointer)**

* RSP মানে হলো **Stack Pointer**।
* এটি সেই মেমরির অংশকে নির্দেশ করে যাকে **Stack Memory** বলে।
* Stack হলো একটি বিশেষ ধরনের মেমরি যেখানে ডাটা **Last In First Out (LIFO)** পদ্ধতিতে রাখা হয়।
* পুরনো প্রসেসরে এর নাম ছিল:
  + 16-bit হলে: **SP**
  + 32-bit হলে: **ESP**
  + 64-bit হলে: **RSP**
* এখানে ফাংশন কল, রিটার্ন অ্যাড্রেস, লোকাল ভেরিয়েবল ইত্যাদি সাময়িকভাবে সেভ হয়।

**RFLAGS (Flag Register)**

* **RFLAGS** হলো এমন একটি রেজিস্টার যেটি প্রসেসরের **স্ট্যাটাস ও কন্ট্রোল ইনফরমেশন** ধরে রাখে।
* পুরনো প্রসেসরে:
  + 16-bit হলে: **FLAGS**
  + 32-bit হলে: **EFLAGS**
  + 64-bit হলে: **RFLAGS**
* এর মধ্যে বিভিন্ন **ফ্ল্যাগ বিট** থাকে (0 বা 1), যেমন:
  + **Zero Flag (ZF):** কোনো অঙ্কের ফলাফল 0 হলে সেট হয়।
  + **Carry Flag (CF):** যোগ/বিয়োগে Carry/borrow হলে সেট হয়।
  + **Sign Flag (SF):** ফলাফল ঋণাত্মক হলে সেট হয়।
  + **Overflow Flag (OF):** গাণিতিক Overflow হলে সেট হয়।
  + **Parity Flag (PF):** ফলাফলের মধ্যে 1-বিটের সংখ্যা জোড়/বিজোড় হলে সেট হয়।

### 👉 এই ফ্ল্যাগগুলো মূলত Arithmetic ও Logic Instruction এর পরে পরিবর্তিত হয়। ****Auxiliary Carry (AF) Flag****

* **Auxiliary Carry Flag (AF)** কে অনেকে **Half-Carry Flag** ও বলে।
* কাজ: এটি **bit 3 (চতুর্থ বিট) থেকে bit 4 (পঞ্চম বিট)** এর মধ্যে যদি **carry** (যোগের ক্ষেত্রে) বা **borrow** (বিয়োগের ক্ষেত্রে) হয়, তখন সেট হয়।

👉 সহজভাবে বললে:

* যখন তুমি 8-bit যোগ করছো, তখন 0–3 নং বিটের (low nibble = নিচের 4 বিট) মধ্যে carry হলে AF = 1 হবে।
* একইভাবে, বিয়োগ করার সময় 0–3 বিট থেকে যদি borrow লাগে, তাহলেও AF = 1 হবে।

### ****Segment Registers (সেগমেন্ট রেজিস্টার)****

মাইক্রোপ্রসেসরে কিছু বিশেষ রেজিস্টার থাকে যেগুলো **মেমরির সেগমেন্ট (অংশ)** নির্দেশ করে।  
👉 অন্য রেজিস্টারের সাথে মিলিয়ে এগুলো ব্যবহার করলে প্রসেসর **মেমরির ঠিকানা (memory address)** তৈরি করতে পারে।

* পুরনো প্রসেসরগুলোতে (8086–80286): **4টি সেগমেন্ট রেজিস্টার**
* নতুন প্রসেসরগুলোতে (80386 থেকে শুরু করে): **6টি সেগমেন্ট রেজিস্টার**

### ****Segment Registers-এর তালিকা ও কাজ****

1. **CS (Code Segment)**
   * এখানে প্রোগ্রামের **ইনস্ট্রাকশন/কোড** থাকে।
   * প্রসেসর CS এবং Instruction Pointer (IP/EIP/RIP) মিলে পরবর্তী **instruction-এর ঠিকানা** বের করে।
   * রিয়েল মোডে: সর্বোচ্চ 64KB কোড রাখা যায়।
   * প্রোটেক্টেড মোডে (80386 থেকে): সর্বোচ্চ 4GB পর্যন্ত কোড সেগমেন্ট হতে পারে।
   * 64-bit মোডে: এখনো CS ব্যবহৃত হয়, তবে Flat Model-এ এর ব্যবহার কিছুটা ভিন্ন।
2. **DS (Data Segment)**
   * এখানে প্রোগ্রামে ব্যবহৃত **ডাটা (ভ্যারিয়েবল, কনস্ট্যান্ট ইত্যাদি)** থাকে।
   * ডাটায় এক্সেস করা হয় DS + offset (অথবা অন্য রেজিস্টারের offset) দিয়ে।
   * দৈর্ঘ্য: 8086–80286 এ 64KB, 80386 থেকে 4GB।
3. **ES (Extra Segment)**
   * এটি একটি **অতিরিক্ত Data Segment**।
   * মূলত **string instruction** (যেমন MOVS, STOS, LODS ইত্যাদি) এ ব্যবহৃত হয় — সাধারণত destination data রাখার জন্য।
4. **SS (Stack Segment)**
   * এখানে **stack memory** থাকে।
   * ফাংশন কল/রিটার্ন অ্যাড্রেস, লোকাল ভ্যারিয়েবল, PUSH/POP ইত্যাদি স্ট্যাক অপারেশনে ব্যবহার হয়।
   * **SS (Stack Segment) + SP/BP (Stack Pointer/Base Pointer)** মিলে স্ট্যাকের ঠিকানা নির্ধারণ করে।
5. **FS এবং GS (অতিরিক্ত সেগমেন্ট)**
   * 80386 থেকে শুরু করে অতিরিক্ত 2টি সেগমেন্ট রেজিস্টার যুক্ত হয়।
   * Windows অপারেটিং সিস্টেম এগুলোকে **ইন্টারনাল কাজে** ব্যবহার করে।
   * সাধারণ প্রোগ্রামাররা সাধারণত এগুলো সরাসরি ব্যবহার করে না।

### ****সংক্ষেপে মনে রাখার ট্রিক****

* **CS → Code** (কোড/প্রোগ্রাম কোথায় আছে)
* **DS → Data** (ডাটা কোথায় আছে)
* **ES → Extra Data** (অতিরিক্ত ডাটা, স্ট্রিং instruction)
* **SS → Stack** (স্ট্যাক কোথায় আছে)
* **FS, GS → অতিরিক্ত সেগমেন্ট (মূলত OS ব্যবহার করে)**

## 1️⃣ Segment Register কী?

8086 (এবং এর পরের x86 প্রসেসর) মেমোরি অ্যাক্সেস করার জন্য **Segmentation** সিস্টেম ব্যবহার করে।  
মানে, মেমোরি address দুই ভাগে ভাগ হয়ে থাকে:

1. **Segment Base Address** → Segment Register থেকে আসে
   * উদাহরণ: CS, DS, SS, ES হলো segment registers।
2. **Offset / Effective Address** → Instruction বা প্রোগ্রাম থেকে আসে
   * ১৬-বিট value, যেটা segment-এর ভিতরে কত দূরে সেটা বলে।

## 2️⃣ Physical Address কিভাবে তৈরি হয় (Real Mode-এ)?

Real Mode-এ **Physical Address = (Segment × 16) + Offset**

🔹 **Segment × 16** → আসলে segment register-এর value কে ৪ বিট left shift করে base address তৈরি করা হয়।  
🔹 তারপর **Offset** যোগ করা হয়।

📌 উদাহরণ:

CS = 0x1234

IP (offset) = 0x0020

Physical Address = (0x1234 × 16) + 0x0020

= 0x12340 + 0x0020

= 0x12360

এখানে CPU 0x12360 মেমোরি লোকেশন অ্যাক্সেস করবে।

💡 এখন তোমাকে একটা ছোট প্রশ্ন:  
যদি DS = 0x2000 আর Offset = 0x0030 হয়, তাহলে Physical Address কত হবে?  
আমি উত্তর বলার আগে তুমি ট্রাই করো।

ঠিক আছে, তাহলে এবার আমরা **Real Mode** আর **Protected Mode** একেবারে তুলনা করে বুঝি।  
আমরা যেটা শিখেছি — (Segment × 16) + Offset — সেটা **Real Mode**-এ সবসময় এমনভাবেই হয়।

## 🟢 Real Mode

* **অবস্থা:** 8086-এর পুরনো অ্যাড্রেসিং মডেল, ২০-bit physical address বাস।
* **Address Limit:** 2^20 = **1 MB** মেমোরি অ্যাক্সেস করা যায়।
* **Segment Register:** শুধু একটা **base** হিসেবে ব্যবহার হয় (×16 shift করে)।
* **প্রোটেকশন নাই:**
  + এক প্রোগ্রাম অন্য প্রোগ্রামের মেমোরি overwrite করতে পারে।
  + কোনো মেমোরি নিরাপত্তা বা access control নাই।
* **মোড প্রবেশ:** CPU reset হলে সবসময় Real Mode-এ শুরু হয়।

📌 উদাহরণ: যদি CS=0xFFFF আর IP=0xFFFF হয় → Physical Address হবে 0x10FFEF (যেটা 1 MB-এর একটু বাইরে যায়, কিন্তু address wrap হয়)।

## 🟡 Protected Mode

* **অবস্থা:** 80286 থেকে শুরু হয়েছে, 32-bit address বাস (পরে 64-bit পর্যন্ত)।
* **Address Limit:** 2^32 = **4 GB** (32-bit সিস্টেমে)।
* **Segment Register:** সরাসরি base address রাখে না; বরং **selector** রাখে → এটা **Descriptor Table**-এ index করে।
* **Descriptor Table:** প্রতিটি segment descriptor বলে দেয়:
  1. Base Address (৩২-bit পর্যন্ত)
  2. Limit (segment-এর সাইজ)
  3. Access rights (read/write/execute)
* **প্রোটেকশন আছে:**
  1. প্রতিটি segment-এর জন্য আলাদা সীমা (limit) থাকে।
  2. Privilege levels (ring 0 → OS, ring 3 → user programs)।
  3. এক প্রোগ্রাম অন্যের মেমোরি অ্যাক্সেস করতে পারে না।

📌 উদাহরণ: DS register-এ 0x0008 থাকলে সেটা GDT-এর ২ নম্বর descriptor কে বোঝাবে, যেখানে Base=0x00400000, Limit=0x000FFFFF → মানে offset 0–1 MB ওই segment-এর মধ্যে বৈধ।

## 🔍 সারসংক্ষেপ তুলনা

| **বৈশিষ্ট্য** | **Real Mode** | **Protected Mode** |
| --- | --- | --- |
| Address Size | 20-bit (1 MB) | 32-bit (4 GB) |
| Segment Register | Base × 16 + Offset | Selector → Descriptor Table |
| Security | নেই | আছে (rings, limits) |
| Multitasking | ঠিকমতো নয় | Proper multitasking সম্ভব |
| Use Today | শুধুমাত্র bootloader/BIOS code | Modern OS runs here |

আমি চাইলে তোমাকে এর একটা **চিত্র** এঁকে দেখাতে পারি যেখানে দুই মোডের address translation আলাদা করে দেখা যাবে।  
তুমি কি চাইছ আমি সেটা এঁকে দেখাই?

ঠিক আছে, তাহলে এবার আমি তোমাকে **গল্পের মতো করে** Real Mode আর Protected Mode বোঝাই, যাতে মনে গেঁথে যায়।

## 🏠 ভাবো তোমার কম্পিউটার হলো একটা শহর

* শহরে অনেকগুলো **বাড়ি (memory locations)** আছে।
* CPU হলো **ডাকপিয়ন**, যাকে ঠিক ঠিক ঠিকানায় চিঠি পৌঁছাতে হবে।

## 1️⃣ Real Mode — পুরনো শহরের নিয়ম

পুরনো শহরে (8086 সময়ে):

* শহরের সব রাস্তায় নামের বদলে **একটা পাড়া (segment)** আর **একটা বাসা নম্বর (offset)** দিয়ে ঠিকানা লেখা হয়।
* পাড়ার নাম × 16 করে, তারপর বাসা নম্বর যোগ করে ডাকপিয়ন ঠিকানা বের করে নেয়।
* পুরো শহর মাত্র **1 MB** বড়, তাই সব বাসা সহজেই পৌঁছানো যায়।
* **সমস্যা:**
  + কেউ চাইলে অন্যের বাড়িতে ঢুকে যেতে পারে।
  + কোনো তালা নেই, কোনো পুলিশ নেই।
  + সবাই সব জায়গায় যেতে পারে, নিরাপত্তা নাই।

📌 উদাহরণ:

পাড়া (segment) = 0x2000

বাসা নম্বর (offset) = 0x0030

ঠিকানা = (0x2000 × 16) + 0x0030 = 0x20030

ডাকপিয়ন সরাসরি ওই বাড়িতে পৌঁছে যাবে।

## 2️⃣ Protected Mode — আধুনিক শহরের নিয়ম

আধুনিক শহরে (80286+):

* পাড়ার নাম (segment register) আসলে সরাসরি রাস্তার ঠিকানা নয়।
* এটা **একটা কোড (selector)**, যেটা **শহরের রেজিস্ট্রেশন অফিসে (Descriptor Table)** গিয়ে খুঁজে বের করতে হয়—
  + ওই পাড়ার আসল শুরু (Base Address) কোথায়,
  + ওই পাড়ায় কতগুলো বাড়ি আছে (Limit),
  + আর কাদের ঢোকার অনুমতি আছে (Access Rights)।
* এখানে তালা আছে, পুলিশ আছে, **ring system** আছে:
  + Ring 0 → OS
  + Ring 3 → সাধারণ অ্যাপ
* কেউ নিজের পাড়ার বাইরে ঢুকতে গেলে **Illegal Access** হয়ে যায়।

📌 উদাহরণ:

DS = 0x0008

→ Descriptor Table দেখে:

Base = 0x00400000

Limit = 0x000FFFFF

Access = Read/Write

Offset 0x00002000 → Final Address = 0x00400000 + 0x00002000 = 0x00402000

ডাকপিয়ন শুধু অনুমোদিত এলাকায়ই যাবে।

## 🔍 মূল পার্থক্যের ছবি

| **বিষয়** | **Real Mode** | **Protected Mode** |
| --- | --- | --- |
| ঠিকানা গণনা | (Segment × 16) + Offset | Selector → Descriptor Table → Base + Offset |
| সর্বোচ্চ ঠিকানা | 1 MB | 4 GB (বা বেশি) |
| নিরাপত্তা | নেই | আছে (Privilege levels) |
| কে ব্যবহার করে | BIOS, Bootloader | আধুনিক OS |

💡 সহজভাবে মনে রাখার কৌশল:

* **Real Mode** → সোজা রাস্তা, কোনো গেট নাই, সবাই ঢুকতে পারে।
* **Protected Mode** → গেট আছে, সিকিউরিটি আছে, কে কোথায় যাবে ঠিক করা আছে।

যদি তুমি চাও, আমি তোমাকে এর একটা **ডায়াগ্রাম** এঁকে দিতে পারি যাতে Real Mode আর Protected Mode-এর address translation চোখের সামনে দেখা যায়।  
তুমি কি চাইছ আমি সেটা এখন আঁকি?

ঠিক আছে, তুমি যেটা পাঠিয়েছো সেটা অনেক লম্বা, তাই আমি একে ধাপে ধাপে ভেঙে সহজভাবে বুঝিয়ে দিচ্ছি।

এটা মূলত **Real Mode-এ segment + offset address কিভাবে মিলে physical address হয়** — সেটা বিশদে বলছে।

## 1️⃣ Real Mode-এ Segment এবং Offset

* **Segment Register** → মেমোরির 64KB (64K bytes) সাইজের একটা ব্লক বা segment-এর **শুরুর ঠিকানা** নির্দেশ করে।
* **Offset (বা displacement)** → ওই segment-এর ভেতরে কত দূরে যেতে হবে সেটা বলে দেয়।
* Final physical address তৈরি হয় এভাবে:

Physical Address=(Segment×16)+Offset\text{Physical Address} = (\text{Segment} \times 16) + \text{Offset}

## 2️⃣ Segment Size এবং শুরু ঠিকানা

* **প্রতিটি segment = 64KB** সাইজের।
* Segment register-এ যেটা থাকে সেটাকে ডানদিকে একটা **0H** বসিয়ে (মানে ×16 বা left shift 4 bits) শুরু ঠিকানা পাওয়া যায়।
* এর মানে segment সবসময় **16-byte boundary** (paragraph boundary) থেকে শুরু হবে।  
  উদাহরণ:
  + 1000H → শুরু ঠিকানা 10000H
  + 1200H → শুরু ঠিকানা 12000H
  + 1201H → শুরু ঠিকানা 12010H

## 3️⃣ শেষ ঠিকানা বের করা

যেহেতু segment 64KB, তাই **শেষ ঠিকানা = শুরু ঠিকানা + FFFFH**

* উদাহরণ:  
  Segment register = 3000H  
  শুরু ঠিকানা = 30000H  
  শেষ ঠিকানা = 3FFFFH

## 4️⃣ Offset যোগ করা

Offset সবসময় segment-এর শুরু ঠিকানার সাথে যোগ হয়।

* উদাহরণ:  
  Segment = 1000H → শুরু = 10000H  
  Offset = 2000H  
  Final Address = 10000H + 2000H = 12000H

নোটেশন: **1000:2000** লেখা হয়।

## 5️⃣ High Memory (A20 Line)

* 80286+, 80386 ইত্যাদিতে FFFFH segment ব্যবহার করলে এবং DOS-এর **HIMEM.SYS** থাকলে, offset যোগ করার সময় **A20 address line** সক্রিয় হলে 1MB-এর একটু উপরে থাকা মেমোরি (High Memory) অ্যাক্সেস করা যায়।
* উদাহরণ:  
  Segment = FFFFH (শুরু = FFFF0H)  
  Offset = 4000H  
  Address = FFFF0H + 4000H = 103FF0H (A20 সক্রিয় হলে)  
  যদি A20 না থাকে → 03FF0H (wrap হয়ে যায়)

## 6️⃣ Offset overflow হলে

Offset 16-bit, তাই এর মান FFFFH-এর বেশি হলে carry ফেলে দেওয়া হয় (modulo 16-bit যোগ হয়)।

* উদাহরণ:  
  Segment = 4000H  
  Offset = F000H + 3000H  
  F000H + 3000H = 12000H → carry ফেলে offset হবে 2000H  
  Final Address = (4000H × 16) + 2000H = 42000H  
  **52000H নয়**।

## 🔹 সহজ করে মনে রাখো

* Segment register → শুরু ঠিকানা (×16)
* Offset → ওই segment-এর ভেতরের দূরত্ব
* সব segment 64KB সাইজের
* শুরু ঠিকানা সবসময় 16-byte boundary থেকে শুরু
* Offset 16-bit, overflow হলে wrap হয়
* Special case: FFFFH segment + offset → A20 হলে high memory

তুমি চাইলে আমি এই পুরো Real Mode addressing-এর একটা **চিত্র** এঁকে দিতে পারি, যেখানে segment, offset, এবং final address স্পষ্টভাবে দেখা যাবে।  
তুমি কি সেটা চাও?

ঠিক আছে, আমি তোমাকে সহজ ভাষায় এবং উদাহরণের মাধ্যমে **Default Segment and Offset Registers** এবং তাদের কাজ কীভাবে হয় Real Mode ও Protected Mode-এ — সেটা ভালো করে বুঝিয়ে দিচ্ছি।

## ১. মাইক্রোপ্রসেসরের Default Segment-Offset Rules কী?

মাইক্রোপ্রসেসর যখন মেমোরি অ্যাক্সেস করে, তখন সে সবসময় একটি **Segment Register** আর একটি **Offset Register** এর যুগল ব্যবহার করে ঠিকানা তৈরি করে।

এই নিয়ম দুইটি মোডে কাজ করে:

* **Real Mode**
* **Protected Mode**

## ২. প্রধান Default Segment-Offset Register Combinations

| **কাজ/Memory Area** | **Segment Register** | **Offset Register** |
| --- | --- | --- |
| **Code (Instructions)** | CS (Code Segment) | IP (Instruction Pointer) |
| **Stack** | SS (Stack Segment) | SP (Stack Pointer) বা BP (Base Pointer) |
| **Data** | DS (Data Segment) | DI (Destination Index) বা SI (Source Index) |
| **Extra Data** | ES (Extra Segment) | DI বা অন্য Offset Register |

### উদাহরণ দিয়ে বুঝি:

### কোড সেগমেন্ট (CS:IP)

* ধরো,
* CS = 1400H
* IP = 1200H
* তাহলে physical address হবে:
* (1400H × 16) + 1200H = 14000H + 1200H = 15200H
* অর্থাৎ, মাইক্রোপ্রসেসর পরের ইনস্ট্রাকশনটি **মেমোরি 15200H** থেকে পড়বে।

### স্ট্যাক সেগমেন্ট (SS:BP)

* ধরো,
* SS = 2000H
* BP = 3000H
* physical address হবে:
* (2000H × 16) + 3000H = 20000H + 3000H = 23000H
* স্ট্যাক ডেটা অ্যাক্সেস হবে **মেমোরি 23000H** থেকে।

## ৩. রেজিস্টার সাইজ এবং রিয়েল মোড এর সীমাবদ্ধতা

* Real Mode-এ offset register সর্বদা **১৬-বিট** হয় (মানে max মান = FFFFH)।
* যদি তুমি ১৬-বিটের চেয়ে বড় মান offset রেজিস্টারে রাখো, তাহলে প্রসেসর **এরর দেয়** এবং সিস্টেম হ্যাঙ্গ করে যেতে পারে।
* এজন্য, ১৬-বিট (F F F F H) এর বেশি মান ঢুকানো ঠিক নয়।

## ৪. মেমোরি সেগমেন্ট এবং ওভারল্যাপ

* ৮০৮৬ থেকে ৮০২৮৬ পর্যন্ত প্রসেসর চারটি মেমোরি সেগমেন্ট ব্যবহার করতে পারে।
* ৮০৩৮৬ ও এর পরবর্তী প্রসেসরগুলো ছয়টি সেগমেন্ট পর্যন্ত ব্যবহার করে।
* **মেমোরি সেগমেন্ট গুলো একে অপরের উপরও ওভারল্যাপ করতে পারে**, কারণ একেকটা সেগমেন্ট সবসময় ৬৪KB হয় না, অনেক সময় ছোটও হয়।

### মেমোরি সেগমেন্ট কেমন হয়?

* ধরো, তোমার প্রোগ্রামে কোডের জন্য ১০০০H (৪ KB) জায়গা লাগে, ডেটার জন্য ১৯০H (৪০০ bytes), আর স্ট্যাকের জন্য ২০০H (৫১২ bytes)।
* এগুলো মোট ৬৪KB এর চেয়ে অনেক কম, তাই সেগমেন্টগুলো মেমোরিতে **ওভারল্যাপ** করতে পারে।
* ডস (DOS) এর প্রোগ্রাম লোডার এই সেগমেন্টগুলোর শুরু ঠিকানা ঠিক করে দেয়, যাতে ওভারল্যাপ ঠিকমতো হয় এবং সেগমেন্ট ঠিক জায়গায় চলে যায়।

## ৫. সেগমেন্ট নিয়ে সহজ একটি ধারণা

### ভাবো তোমার মেমোরি একটা বড় খাতা, যেখানে তুমি পাতা ভাগ করেছো ৬৪KB করে (একটা সেগমেন্ট)।

* প্রতিটি সেগমেন্টের শুরু ঠিকানা হলো সেই পাতার প্রথম পৃষ্ঠার ঠিকানা (যেটা segment register ধরে রাখে)।
* Offset হলো ওই পাতার মধ্যে পাতার কত দূরে লিখবে (offset register)।
* Offset সর্বদা সেই সেগমেন্টের মধ্যে থাকে (০ থেকে ৬৪KB)।

## সারসংক্ষেপ

| **বিষয়** | **ব্যাখ্যা** |
| --- | --- |
| Segment Register | মেমোরি সেগমেন্টের শুরু ঠিকানা ধরে রাখে |
| Offset Register | সেগমেন্টের ভিতরে অবস্থান (offset) নির্দেশ করে |
| CS:IP | কোড সেগমেন্ট + ইনস্ট্রাকশন পয়েন্টার |
| SS:SP বা SS:BP | স্ট্যাক সেগমেন্ট + স্ট্যাক পয়েন্টার/বেস পয়েন্টার |
| Offset max মান | Real Mode-এ ১৬-বিট (FFFFH) পর্যন্ত সম্ভব |
| সেগমেন্ট সংখ্যা | ৮০৮৬-৮০২৮৬ এ ৪ টি, ৮০৩৮৬ এ ৬ টি |
| সেগমেন্ট ওভারল্যাপ | সেগমেন্ট গুলো মেমোরিতে ওভারল্যাপ করতে পারে |

### তোমার জন্য:

চলো আমি এখন ছোট্ট একটা চিত্র আঁকি, যেখানে CS:IP আর SS:BP এর মাধ্যমে মেমোরি অ্যাক্সেস দেখানো হবে।  
তুমি কি এটা দেখতে চাও?

অবশ্যই! আমি তোমাকে এই অংশটা সহজ ভাষায় এবং উদাহরণের মাধ্যমে ভালো করে বুঝিয়ে দিচ্ছি।

## Segment এবং Offset Addressing Scheme এর **Relocation** সুবিধা কী?

### ১. ঠিকানা নির্ধারণ কঠিন হলেও সুবিধাজনক

Segment + Offset মেমোরি ঠিকানা নির্ধারণের পদ্ধতি প্রথমে জটিল মনে হতে পারে, কারণ এটা দুইটি অংশ (segment ও offset) মিলে physical address তৈরি করে।

কিন্তু এর মাধ্যমে বড় সুবিধা পাওয়া যায় — **প্রোগ্রাম আর ডেটাকে মেমোরির যেকোন জায়গায় সরিয়ে (relocate) দেওয়া যায়, প্রোগ্রাম বা ডেটাতে কোনো পরিবর্তন ছাড়া।**

### ২. Relocatable Program & Data মানে কী?

* **Relocatable program:** এমন প্রোগ্রাম যা মেমোরির যেকোন জায়গায় সরিয়ে নেওয়া যায়, আবার ঠিকমতো চলতে পারে—কোনো কোড পরিবর্তন ছাড়া।
* **Relocatable data:** এমন ডেটা যা মেমোরির যেকোন জায়গায় রাখা যায়, প্রোগ্রাম ব্যবহার করতে পারে কোনো পরিবর্তন ছাড়াই।

### ৩. কিভাবে এটা সম্ভব?

* Segment register শুধু **segment-এর শুরু ঠিকানা** ধরে রাখে।
* Offset register **সেগমেন্টের ভিতরে দূরত্ব** (offset) ধরে রাখে।
* যখন পুরো প্রোগ্রাম/ডেটা মেমোরির অন্য জায়গায় সরানো হয়, তখন **offset গুলো অপরিবর্তিত থাকে।**
* শুধু **segment register-এর মান পরিবর্তন** করে নতুন মেমোরি ঠিকানা নির্দেশ করা হয়।

### উদাহরণ দিয়ে বুঝি:

ধরো:

* একটি ইনস্ট্রাকশন segment-এর শুরু থেকে ৪ বাইট দূরে আছে → offset = ৪
* প্রোগ্রাম প্রথমে segment = 1000H এ ছিল, অর্থাৎ physical address = 10000H + 4 = 10004H

এখন প্রোগ্রামকে মেমোরির অন্য জায়গায় সরানো হলো, যেখানে segment এখন 2000H:

* offset অপরিবর্তিত → 4
* physical address = 20000H + 4 = 20004H

অর্থাৎ, offset ঠিক একই জায়গায় নির্দেশ করে, শুধু segment value বদলেছে।

### ৪. এর গুরুত্ব

* **বিভিন্ন কম্পিউটারে মেমোরি কনফিগারেশন আলাদা।**
* তাই এক রকম প্রোগ্রাম সব জায়গায় কাজ করতে পারে, কারণ segment value পরিবর্তন করে সেটাকে নতুন জায়গায় রাখা যায়।
* প্রোগ্রাম বা ডেটাতে আলাদা করে কিছু বদলানোর দরকার পড়ে না।
* এটি **DOS এর প্রোগ্রাম লোডার** দ্বারা স্বয়ংক্রিয়ভাবে হ্যান্ডেল করা হয়।

### ৫. Protected Mode-এও একই ধারণা

* উইন্ডোজের মতো Protected Mode অপারেটিং সিস্টেমেও প্রোগ্রামগুলো মেমোরির যেকোন জায়গায় লোড করা যায়।
* প্রোগ্রাম লেখা হয় ধারণা করে যে প্রথম ২ গিগাবাইট মেমোরি তাদের জন্য বরাদ্দ।
* কিন্তু বাস্তবে প্রোগ্রাম যেকোন জায়গায় লোড হতে পারে, এবং প্রয়োজনে ডেটার কিছু অংশ ডিস্কে (swap file) রাখা হয়।

## সংক্ষেপে

| **বিষয়** | **ব্যাখ্যা** |
| --- | --- |
| Relocation | প্রোগ্রাম ও ডেটা মেমোরির যেকোন জায়গায় সরানো যায় |
| কিভাবে | Offset অপরিবর্তিত, Segment পরিবর্তন করে নতুন ঠিকানা পাওয়া যায় |
| সুবিধা | প্রোগ্রামকে পুনঃলিখন ছাড়া বিভিন্ন কম্পিউটারে চলানো যায় |
| ব্যবহার | DOS ও Protected Mode OS-এ প্রোগ্রাম লোডার ব্যবহার করে |

তুমি কি চাও আমি এই Relocation পদ্ধতি নিয়ে একটা সহজ ছবি আঁকি যাতে ব্যাপারটা চোখে-চোখে স্পষ্ট হয়?

ঠিক আছে, আমি ধাপে ধাপে তোমাকে Protected Mode মেমরি অ্যাড্রেসিং (80286 এবং এর পরের প্রসেসর) বাংলায় সহজভাবে বুঝিয়ে দিচ্ছি।

## **১. Protected Mode কী এবং কেন দরকার**

* **Real Mode**-এ (যেটা পুরনো 8086 প্রসেসরে ছিল) সর্বোচ্চ 1MB মেমরি অ্যাক্সেস করা যেত।
* **Protected Mode** (80286 থেকে শুরু) এই সীমা ভেঙে 1MB-এর উপরের **Extended Memory**-তেও অ্যাক্সেস দেয়।
* এখনকার Windows, Linux সবই Protected Mode ব্যবহার করে।

## **২. Real Mode বনাম Protected Mode অ্যাড্রেসিং পার্থক্য**

* **Real Mode:** মেমরি অ্যাড্রেসিং হয় Segment Address + Offset Address দিয়ে।  
  Segment রেজিস্টারে সরাসরি সেগমেন্ট শুরুর অ্যাড্রেস থাকে।
* **Protected Mode:** Segment রেজিস্টারে সরাসরি অ্যাড্রেস থাকে না। বরং সেখানে থাকে **Selector** নামে একটি ভ্যালু, যেটা দিয়ে **Descriptor Table** থেকে সেগমেন্টের তথ্য বের করা হয়।

## **৩. Selector এবং Descriptor**

* **Selector:** Segment রেজিস্টারে থাকা একটি সংখ্যা যা Descriptor Table-এর মধ্যে একটি Descriptor বেছে নেয়।
* **Descriptor Table:**
  + দুই ধরনের হয়—
    1. **Global Descriptor Table (GDT):** সিস্টেম-ওয়াইড সেগমেন্ট ডেফিনিশন।
    2. **Local Descriptor Table (LDT):** নির্দিষ্ট কোনো প্রোগ্রামের জন্য সেগমেন্ট ডেফিনিশন।
  + প্রতিটি টেবিলে সর্বোচ্চ 8192 Descriptor থাকতে পারে। মোট 16384 সেগমেন্ট ডেফাইন করা সম্ভব।

## **৪. Descriptor-এর কাজ**

* Descriptor-এ থাকে:
  + **Base Address:** সেগমেন্ট কোথা থেকে শুরু হবে।
  + **Limit:** সেগমেন্টের দৈর্ঘ্য কত।
  + **Access Rights:** কে এবং কীভাবে এই সেগমেন্টে অ্যাক্সেস করতে পারবে।
* Descriptor-এর দৈর্ঘ্য 8 byte।
* 80286-এ Base Address 24-bit → 16MB পর্যন্ত লোকেশন সাপোর্ট করে।
* 80386 এবং এর পর থেকে Base Address 32-bit → 4GB পর্যন্ত সাপোর্ট করে।

## **৫. Granularity (G bit)**

বাহ্‌, তুমি একদম প্রসেসরের ভেতরের খুঁটিনাটি ধরেছো 🤓🔥  
আমি সহজভাবে বুঝিয়ে বলছি—

**80286 (16-bit Architecture)**

* এখানে offset address (অর্থাৎ সেগমেন্টের ভেতরে যতদূর যাওয়া যায়) **16-bit**।
* তাই সর্বোচ্চ offset = FFFFH = 64 KB পর্যন্ত।
* এজন্য 80286-এ **একটা সেগমেন্টের দৈর্ঘ্য সর্বোচ্চ 64KB**।

**80386 থেকে Pentium 4 (32-bit Architecture)**

এখানে offset address হয় **32-bit**।

* তাই সর্বোচ্চ offset = FFFFFFFFH = 4 GB পর্যন্ত।
* এজন্য এখানে **একটা সেগমেন্ট 4GB পর্যন্ত লম্বা হতে পারে**।

কিন্তু, descriptor-এ **G bit (Granularity bit)** নামের নতুন ফিচার যোগ হলো।

**G bit কী করে?**

* যদি G = 0 হয় → Limit সরাসরি **byte হিসেবে গণনা হবে**।  
  👉 Limit = 00000H থেকে FFFFFH → অর্থাৎ সর্বোচ্চ সেগমেন্ট সাইজ **1 MB**।
* যদি G = 1 হয় → Limit-কে **4KB (4096 bytes) এর unit** ধরে গুন করা হবে।  
  👉 Limit × 4K → Limit তখন 00000FFFFH থেকে FFFFFFFFH → অর্থাৎ সর্বোচ্চ সেগমেন্ট সাইজ **4 GB**।

অন্যভাবে বললে:

* G=0 → ছোট memory chunk দরকার হলে ব্যবহার হয়।
* G=1 → বড় বড় সেগমেন্ট (OS, applications) চালাতে সুবিধা হয়।

**কেন 80286-এ নেই?**

কারণ 80286 ছিল 16-bit processor → offset address কেবল 16-bit → সর্বোচ্চ segment size = 64KB → granularity-এর দরকারই ছিল না।

80386 আর পরের প্রসেসরগুলোতে offset হলো 32-bit → সেগমেন্ট 4GB পর্যন্ত করা যায় → তাই granularity bit ব্যবহার করা হলো বড় সেগমেন্ট ম্যানেজ করার জন্য।

👉 তাই **DOS (16-bit OS)** সবসময় 64KB সেগমেন্ট ব্যবহার করত।  
👉 আর **Windows (WIN32, অর্থাৎ 32-bit OS)** 4GB সেগমেন্ট পর্যন্ত ব্যবহার করতে পারে।

## **৬. Offset Size**

* 80286: Offset 16-bit → সেগমেন্ট লেন্থ সর্বোচ্চ 64KB।
* 80386+: Offset 32-bit → সেগমেন্ট লেন্থ সর্বোচ্চ 4GB।

## **৭. 64-bit Protected Mode**

* 64-bit মোডে Descriptor-এ Limit বা Base Address থাকে না, কারণ সব কোড সেগমেন্ট 0x0000000000 থেকে শুরু হয়।
* Limit চেকও করা হয় না।
* শুধু Access Rights এবং কন্ট্রোল বিট থাকে।

## **৮. উদাহরণ**

**Example 2-1:**

Base = 10000000H

Limit = 001FFH

G bit = 0

End = Base + Limit = 10000000H + 001FFH = 100001FFH

মানে সেগমেন্ট শুরু 10000000H থেকে এবং শেষ 100001FFH এ।

**Example 2-2:**

Base = 10000000H

Limit = 001FFH

G bit = 1 → Limit = 001FFH × 4KB (অর্থাৎ FFFH যোগ)

End = Base + 001FFFFFH = 101FFFFFH

আহা, তুমি তো Example টা একদম ধরতে পেরেছো 😃👌 আমি ধাপে ধাপে খুলে বলি—

**দেওয়া আছে:**

* **Base = 10000000H**
* **Limit = 001FFH**
* **G bit = 1**

**ধাপ ১: G bit = 1 হলে কী হয়?**

যদি G = 1 হয়, তবে Limit **byte না হয়ে 4KB ব্লকে** গণনা হবে।  
মানে—

Effective Limit = (Limit × 4KB) + 0FFFH

(FFFH যোগ করা হয় কারণ প্রতিটি ব্লক 4KB = 1000H, তাই শেষের দিকের ছোট অংশ cover করার জন্য FFFH add হয়।)

**ধাপ ২: Limit ক্যালকুলেশন**

Limit = 001FFH

এটাকে 4KB unit-এ ধরতে হবে →

001FFH × 1000H = 001FF000H

এর সাথে আবার FFFH যোগ করতে হবে →

001FF000H + 00000FFFH = 001FFFFFH

👉 সুতরাং Effective Limit = **001FFFFFH**

**ধাপ ৩: End Address বের করা**

সেগমেন্টের শেষ address = **Base + Limit**

Base = 10000000H

Limit = 001FFFFFH

-----------------

End = 101FFFFFH

**🔎 ব্যাখ্যা**

তাহলে, এই সেগমেন্ট শুরু হলো 10000000H থেকে আর শেষ হলো 101FFFFFH পর্যন্ত।  
মোট সেগমেন্টের দৈর্ঘ্য হলো প্রায় **2 MB** (কারণ 001FFFFFH ≈ 2MB)।

👉 তাই এখানে সেগমেন্টটা অনেক বড় হলো, কারণ G bit = 1 থাকার কারণে Limit-কে 4KB ব্লকে গণনা করা হয়েছে।

মানে সেগমেন্ট অনেক বড় হলো, কারণ Limit 4KB ব্লকে গণনা হয়েছে।

যদি চাও, আমি তোমার জন্য একটা **সহজ ডায়াগ্রাম** বানিয়ে দিতে পারি যেখানে Real Mode আর Protected Mode-এর মেমরি অ্যাড্রেসিং এর তুলনা থাকবে।  
এতে পুরো ব্যাপারটা চোখের সামনে স্পষ্ট হয়ে যাবে।

Segment limit একটি সেগমেন্টের শেষ offset address ধারণ করে। উদাহরণস্বরূপ, যদি একটি সেগমেন্টের শুরু ঠিকানা **F00000H** এবং শেষ ঠিকানা **F000FFH** হয়, তবে base address হবে **F00000H** এবং limit হবে **FFH**।

### ৮০২৮৬ মাইক্রোপ্রসেসর:

* **Base address**: **F00000H**
* **Limit**: **00FFH**

এটি বুঝায় যে, ৮০২৮৬ মাইক্রোপ্রসেসরের জন্য একটি সেগমেন্টের দৈর্ঘ্য ১ বাইট থেকে ৬৪KB পর্যন্ত হতে পারে।

### ৮০৩৮৬ এবং তার উপরের মাইক্রোপ্রসেসর (যেমন Pentium 4):

* **Base address**: **00F00000H**
* **Limit**: **000FFH**

এই মাইক্রোপ্রসেসরগুলির জন্য সেগমেন্টের দৈর্ঘ্য ১ বাইট থেকে ১MB অথবা ৪KB থেকে ৪GB পর্যন্ত হতে পারে।

### G Bit (Granularity Bit):

৮০৩৮৬ এবং তার উপরের মাইক্রোপ্রসেসরগুলিতে **G Bit** নামে একটি বৈশিষ্ট্য থাকে, যা ৮০২৮৬ মাইক্রোপ্রসেসরে থাকে না।

* **G = 0** হলে, limit সরাসরি সেগমেন্টের শেষ পর্যন্ত থাকে। অর্থাৎ, limit ০০০০০H থেকে FFFFFH পর্যন্ত হতে পারে।
* **G = 1** হলে, limit মানটি ৪KB দিয়ে গুণিত হয়ে সেগমেন্টের দৈর্ঘ্য বাড়ানো হয়। এর মানে, limit value ৪KB দিয়ে গুণিত হয়ে ৪GB পর্যন্ত হতে পারে।

এটি সেগমেন্টের দৈর্ঘ্য ৪KB থেকে ৪GB পর্যন্ত হওয়া সম্ভব করে, যেখানে ৮০২৮৬ এর জন্য সেগমেন্টের দৈর্ঘ্য সর্বোচ্চ ৬৪KB ছিল, কারণ সেটির offset address ১৬-বিট ছিল।

### ৮০৩৮৬ এবং তার পরবর্তী মাইক্রোপ্রসেসরের সুবিধা:

* ৮০৩৮৬ এবং তার পরবর্তী মাইক্রোপ্রসেসরগুলির ৩২-বিট offset address থাকে, যা সেগমেন্টের দৈর্ঘ্য ৪GB পর্যন্ত বাড়ানোর সুবিধা দেয়।
* ৮০২৮৬ এর ১৬-বিট offset address থাকার কারণে সেগমেন্টের দৈর্ঘ্য ৬৪KB পর্যন্ত সীমাবদ্ধ ছিল।

অপারেটিং সিস্টেম সাধারণত ১৬-বিট বা ৩২-বিট পরিবেশে চলে। উদাহরণস্বরূপ, **DOS** একটি ১৬-বিট পরিবেশ ব্যবহার করে, তবে অধিকাংশ **Windows** অ্যাপ্লিকেশন **WIN32** নামক ৩২-বিট পরিবেশে চলে।

আমি এই অংশটা বাংলায় সহজভাবে বোঝাচ্ছি—

## **64-bit Descriptor এবং L Bit**

* Descriptor-এ **L bit** (Large bit) থাকে, যেটা বলে দেবে কোড সেগমেন্ট **64-bit** হবে নাকি **32-bit compatibility mode**-এ চলবে।
  + **L = 1 →** 64-bit addresses ব্যবহার হবে (Pentium 4 বা Core2-এর 64-bit extensions সহ)।
  + **L = 0 →** 32-bit compatibility mode-এ চলবে।

## **64-bit Protected Mode-এ Descriptor-এর পরিবর্তন**

* 64-bit mode-এ কোড সেগমেন্ট Descriptor-এ **Base Address** বা **Limit** রাখা হয় না।
* কেবল **Access Rights Byte** আর কিছু Control Bits থাকে।
* সব কোড সেগমেন্টের Base Address ধরা হয় **0x0000000000** (শূন্য থেকে শুরু)।
* যেহেতু Limit নাই, তাই **কোনো সেগমেন্ট সাইজ চেক হয় না** — মানে theoretically তুমি পুরো 64-bit অ্যাড্রেস স্পেস অ্যাক্সেস করতে পারো।

## **এর মানে কী দাঁড়াচ্ছে?**

Real Mode বা Protected Mode-এর 16-bit/32-bit descriptor-এ যেভাবে Base Address আর Limit দিয়ে সেগমেন্টের সীমা নির্ধারণ করা হতো,  
64-bit mode-এ সেটা প্রয়োজন হয় না। কারণ পুরো মেমরি অ্যাড্রেস স্পেস একটানা ব্যবহার করা যায়, আর সেগমেন্টেশন প্রায় বাদই দেওয়া হয়েছে।

যদি তুমি চাও, আমি 16-bit, 32-bit আর 64-bit descriptor-এর একটা **তুলনামূলক টেবিল** বানিয়ে দিতে পারি যাতে L bit আর limit/base address পার্থক্য চোখের সামনে স্পষ্ট হয়।

ঠিক আছে, আমি এই অংশটা বাংলায় পরিষ্কারভাবে ব্যাখ্যা করছি—

## **1. AV Bit (Available Bit)**

* **AV bit** (80386 এবং এর পরের প্রসেসরে) কিছু অপারেটিং সিস্টেম ব্যবহার করে এটা বোঝাতে—
  + **AV = 1:** সেগমেন্ট **Available** (ব্যবহারের জন্য প্রস্তুত)
  + **AV = 0:** সেগমেন্ট **Not Available** (ব্যবহারের জন্য নয়)
* সাধারণভাবে, এটা অপারেটিং সিস্টেম মেমরি ম্যানেজমেন্ট বা টাস্ক সুইচিং-এর সময় ব্যবহার করে।

## **2. D Bit (Default Operation Size Bit)**

* **D bit** বলে দেয়, ইনস্ট্রাকশন এবং অ্যাড্রেস ডিফল্টে **16-bit** হবে নাকি **32-bit**।

**D = 0**

* + 16-bit instruction mode (8086–80286 compatible)
  + Offset এবং রেজিস্টার ডিফল্টে 16-bit
  + এই মোডকে অনেক সময় "DOS mode" বলা হয়
  + MS-DOS, PC-DOS, Windows 3.1 — সবগুলোতেই 16-bit instruction mode লাগে

**D = 1**

* + 32-bit instruction mode
  + Offset এবং রেজিস্টার ডিফল্টে 32-bit
  + আধুনিক অপারেটিং সিস্টেম ও অ্যাপ্লিকেশনগুলোতে ব্যবহার হয়
* মনে রাখবে, **ডিফল্ট সাইজ override করা যায়** — অর্থাৎ 16-bit মোডে 32-bit ইনস্ট্রাকশন বা উল্টোটা ব্যবহার সম্ভব, কিন্তু ডিফল্ট ভিন্ন হবে।

## **3. Access Rights Byte**

* Descriptor-এর মধ্যে একটি **Access Rights Byte** থাকে যা সেগমেন্টের কার্যপ্রণালী নিয়ন্ত্রণ করে।
* এর মাধ্যমে নির্ধারণ করা হয়—
  + **সেগমেন্টের ধরন** (Data segment, Code segment ইত্যাদি)
  + Data segment হলে **দিক (Direction of Growth)** — উপরের দিকে নাকি নিচের দিকে মেমরি বরাদ্দ হবে।
  + Data segment **লিখনযোগ্য** নাকি **write-protected**।
  + Code segment-এ **reading নিষিদ্ধ** করা যায় যাতে প্রোগ্রাম কপি বা ডাম্প না হয়।
* যদি সেগমেন্ট তার limit ছাড়িয়ে যায়, CPU একটি **General Protection Fault** দেয়।

## **4. 64-bit Mode-এ পার্থক্য**

* 64-bit mode-এ **শুধু Code Segment থাকে**, অন্য segment descriptor টাইপ থাকে না।
* Flat memory model ব্যবহৃত হয় — অর্থাৎ Code segment-এই data, stack, সবকিছু থাকে।
* Limit চেক হয় না, Base সবসময় 0x0000000000।

যদি তুমি চাও, আমি **AV bit, D bit আর Access Rights Byte-এর ভিজ্যুয়াল চার্ট** বানিয়ে দিতে পারি যাতে এগুলোর সম্পর্ক ও কাজ একসাথে বোঝা যায়।  
এতে Protected Mode descriptor-এর control bits অনেক পরিষ্কার হবে।

ঠিক আছে 👍  
আমি তোমার জন্য **AV bit, D bit আর Access Rights Byte**–এর একটি ভিজ্যুয়াল চার্ট তৈরি করে দিচ্ছি যাতে সব একসাথে বোঝা যায়।

## **Protected Mode Descriptor – Control Bits Overview**

| **Bit Name** | **অবস্থান** | **মান (Value)** | **মানে কী** | **ব্যবহার** |
| --- | --- | --- | --- | --- |
| **AV bit** (Available) | Access Rights Byte-এ | AV = 1 → AvailableAV = 0 → Not Available | সেগমেন্ট সিস্টেমে ব্যবহারের জন্য প্রস্তুত কিনা বোঝায় | অপারেটিং সিস্টেম মেমরি ম্যানেজমেন্ট ও টাস্ক সুইচিং-এ |
| **D bit** (Default Operation Size) | Access Rights Byte-এ | D = 0 → 16-bit মোডD = 1 → 32-bit মোড | ডিফল্ট instruction ও offset size নির্ধারণ করে | 16-bit মোডে DOS প্রোগ্রাম, 32-bit মোডে আধুনিক অ্যাপ |
| **Access Rights Byte** | Descriptor-এর 1 byte অংশ | বিভিন্ন বিটের সমন্বয় | সেগমেন্টের ধরন, পড়া/লেখা অনুমতি, দিক (growth direction) ইত্যাদি নিয়ন্ত্রণ করে | সিকিউরিটি, মেমরি প্রোটেকশন, প্রোগ্রাম কপি প্রতিরোধ |

### ****Access Rights Byte – বিট ব্রেকডাউন (Simplified)****

7 6 5 4 3 2 1 0 ← Bit Position

+---+---+---+---+---+---+---+---+

| P | DPL | S | Type / RW / X |

+---+---+---+---+---+---+---+---+

* **P** = Present bit (1 = সেগমেন্ট উপস্থিত, 0 = অনুপস্থিত)
* **DPL** = Descriptor Privilege Level (0 থেকে 3)
* **S** = Descriptor type bit (0 = সিস্টেম, 1 = কোড/ডাটা)
* **Type/RW/X** = পড়া/লেখা অনুমতি, কোড executable কিনা ইত্যাদি

### ****Visualization****

┌──────────────────────────┐

│ Access Rights Byte │

├───────────┬──────────────┤

│ AV Bit │ Available? │

├───────────┼──────────────┤

│ D Bit │ 16-bit / 32-bit instruction mode │

├───────────┼──────────────┤

│ Other Bits│ Segment type, Read/Write, Growth │

└───────────┴──────────────┘

💡 **৬৪-bit Mode পার্থক্য:**

* AV bit এবং D bit ঐতিহাসিক কারণে আছে, কিন্তু 64-bit flat model-এ সেগমেন্টেশন প্রায় ব্যবহার হয় না।
* Code segment descriptor থাকলেও Base = 0 এবং Limit চেক হয় না।

আমি চাইলে তোমার জন্য এর একটা **রঙিন ডায়াগ্রাম**ও বানিয়ে দিতে পারি যেখানে বিট-পজিশন, মান এবং কাজ একসাথে দেখা যাবে।  
এতে মনে রাখাটা আরও সহজ হবে।

চমৎকার জায়গায় চলে এসেছো 🙂 এখন আমরা বুঝবো **Protected Mode-এ descriptor table থেকে segment register কিভাবে কাজ করে**। আমি ধাপে ধাপে সহজ করে ভেঙে দিচ্ছি—

**1. Segment Register = Selector + কিছু Control Bit**

Protected mode-এ segment register (যেমন CS, DS, SS ইত্যাদি) শুধু “base address” রাখে না, বরং একটা **Selector** রাখে।  
এই Selector-এর ভেতর থাকে ৩টা জিনিস—

* **Index (13-bit):** কোন descriptor (entry) ব্যবহার হবে সেটা বেছে নেয়। 13 bit → সর্বোচ্চ 8192 টা descriptor entry ব্যবহার করা যায়।
* **TI bit (Table Indicator):**
  + TI = 0 → **Global Descriptor Table (GDT)** থেকে নেবে।
  + TI = 1 → **Local Descriptor Table (LDT)** থেকে নেবে।
* **RPL (Requested Privilege Level, 2-bit):**
  + কোন privilege level-এ segment access চাইছে সেটাকে বলে।
  + Privilege level 00 = সবচেয়ে বেশি ক্ষমতা (kernel / OS core)।
  + Privilege level 11 = সবচেয়ে কম ক্ষমতা (user / application)।

**2. Privilege Check**

* প্রতিটি descriptor-এর মধ্যে একটা **DPL (Descriptor Privilege Level)** থাকে।
* যখন কোনো segment access করা হয়, CPU চেক করে—  
  👉 RPL (selector থেকে) ≤ DPL (descriptor থেকে) হলে access দেওয়া হয়।
* উদাহরণ:
  + যদি RPL = 10 আর DPL = 11 → Access Granted (কারণ 10 বেশি প্রাধান্যশীল)।
  + যদি RPL = 11 আর DPL = 10 → Access Denied (কারণ ইউজার লেভেল প্রোগ্রাম privileged কোডে ঢুকতে পারবে না)।

Windows-এ সাধারণত—

* **Ring 0 (Privilege 00):** Kernel ও driver।
* **Ring 3 (Privilege 11):** Applications।
* Ring 1 ও Ring 2 ব্যবহার হয় না।

**3. Descriptor নির্বাচন**

যখন DS = 0008H লোড করা হলো:

* Binary এ: 0000 0000 0000 1000b
* Index = 0000000000001 = Descriptor #1
* TI = 0 (মানে GDT থেকে নেবে)
* RPL = 00

👉 তাই CPU GDT-এর Entry #1 (descriptor #1) বেছে নিলো।

**4. Descriptor #1 এর ভেতরের তথ্য**

ধরা হলো Descriptor #1 এর ভেতরে—

* **Base Address = 00100000H**
* **Limit = 000FFH**

👉 মানে এই data segment শুরু হবে **00100000H** থেকে এবং শেষ হবে **001000FFH** পর্যন্ত।  
মোট দৈর্ঘ্য = **256 bytes**।

**5. Null Descriptor (#0)**

* Descriptor table-এর 0th entry সবসময় **Null Descriptor**।
* এটা সব 0 দিয়ে ভর্তি থাকে।
* CPU এটাকে কখনও access করতে দেয় না (কারণ invalid)।

✅ তাই এই পুরো সিস্টেমের মূলকথা হলো:  
**Segment register → Selector → Descriptor Table → Descriptor → Base + Limit → Actual Memory Segment**

ঠিক আছে 🙂 তুমি এখন descriptor table-এর **program-invisible registers** এ চলে এসেছো—এগুলো প্রোটেক্টেড মোডে অনেক গুরুত্বপূর্ণ। আমি সহজভাবে গুছিয়ে বলছি—

## 🔹 Program-Invisible Registers কী?

* এগুলো **সরাসরি প্রোগ্রাম থেকে access করা যায় না**, তাই নাম program-invisible।
* কিন্তু CPU এগুলো ব্যবহার করে **descriptor table** নিয়ন্ত্রণ করতে।
* এগুলোর কাজ হলো descriptor table থেকে তথ্য নিয়ে segment register বা অন্য control register-এ cache করে রাখা।

## 🔹 Segment Register-এর ভিতরে “Cache Part”

* প্রতিটি segment register (CS, DS, SS, ইত্যাদি)-এর ২টা অংশ থাকে—
  1. **Visible part (selector)** → প্রোগ্রাম থেকে set করা যায় (যেমন DS=0008H)।
  2. **Invisible part (cache)** → CPU descriptor table থেকে Base, Limit, Access Rights নিয়ে এখানে cache করে রাখে।

👉 যখন segment register একবার load করা হয়, তখন descriptor table থেকে তথ্য invisible অংশে চলে আসে।  
👉 এরপর CPU বারবার descriptor table-এ না গিয়ে সরাসরি cache থেকে কাজ করে → দ্রুত access পাওয়া যায়।  
👉 এ কারণেই একে “cache” বলা হয় (কিন্তু এটা L1/L2 cache-এর মতো নয়)।

## 🔹 বিশেষ Program-Invisible Registers

1. **GDTR (Global Descriptor Table Register)**
   * এর ভেতর থাকে **GDT-এর base address আর limit**।
   * GDT = পুরো সিস্টেমে শেয়ার করা descriptor table।
   * 16-bit limit থাকায় GDT-এর সর্বোচ্চ সাইজ 64KB।
2. **IDTR (Interrupt Descriptor Table Register)**
   * Interrupt Descriptor Table (IDT)-এর base address আর limit ধরে।
   * প্রোটেক্টেড মোড চালুর আগে এটাও initialize করতে হয়।
3. **LDTR (Local Descriptor Table Register)**
   * Local Descriptor Table (LDT)-কে access করতে ব্যবহৃত হয়।
   * আসলে LDT-এর location GDT-তে define থাকে, তারপর LDTR সেই descriptor-টা নিয়ে cache করে রাখে।
   * মাল্টিটাস্কিং OS-এ প্রতিটি task/proccess-এর নিজস্ব LDT থাকতে পারে।
4. **TR (Task Register)**
   * Task Register-এর ভেতরে থাকে একটা **selector**, যেটা GDT-এর ভিতরে থাকা Task State Segment (TSS) descriptor-কে নির্দেশ করে।
   * এর মাধ্যমে CPU জানে কোন task (procedure বা program) চালাতে হবে।
   * TR ব্যবহার করে multitasking-এ **task switching** করা যায়।
   * Intel CPU-তে hardware task switch সময় লাগে প্রায় **17 µs**।

## 🔹 কেন দরকার?

* GDT = সবার জন্য কমন টেবিল।
* LDT = আলাদা আলাদা প্রোগ্রাম বা ইউজারের জন্য লোকাল টেবিল।
* TR = দ্রুত multitasking সুইচ করতে।
* GDTR আর IDTR ছাড়া প্রোটেক্টেড মোড চালু হয়ই না।

👉 সহজ করে বললে:

* **GDTR** = “GDT কোথায় আছে?”
* **IDTR** = “Interrupt টেবিল কোথায় আছে?”
* **LDTR** = “লোকাল descriptor table কোথায় আছে?”
* **TR** = “এখন কোন task চালু আছে?”

চাও আমি তোমাকে একটা **ছোট্ট চিত্র আকারে ম্যাপ** এঁকে দেখাই যেখানে GDTR, LDTR, TR কীভাবে descriptor table-এর সাথে লিঙ্ক করে? ✍️📊