**Chapter – 10**

Flowchart ব্যবহারের মূল কারণগুলো নিচে ব্যাখ্যা করা হলো —

**🧭 ১. প্রোগ্রামের ধাপগুলো স্পষ্টভাবে বোঝানো**

ফ্লোচার্টের মাধ্যমে কোনো প্রোগ্রাম বা প্রক্রিয়ার প্রতিটি ধাপকে **চিত্রের মাধ্যমে উপস্থাপন** করা যায়। এতে পুরো লজিক বা কাজের ধারা (workflow) সহজে বোঝা যায়।

**🧩 ২. সমস্যা বিশ্লেষণে সহায়তা করে**

যখন কোনো সমস্যা সমাধান করা হয়, ফ্লোচার্ট তৈরি করলে **কোন ধাপে কী ঘটছে** তা সহজে চিহ্নিত করা যায়। এর ফলে **ত্রুটি (error)** বা **অতিরিক্ত ধাপ (redundancy)** খুঁজে বের করা সহজ হয়।

**💡 ৩. প্রোগ্রাম ডিজাইন সহজ করে**

কোড লেখার আগে ফ্লোচার্ট তৈরি করলে পুরো প্রোগ্রামের কাঠামো আগে থেকেই বোঝা যায়। এতে **লজিক্যাল ভুল** কম হয় এবং কোডিং প্রক্রিয়া দ্রুত সম্পন্ন করা যায়।

**🤝 ৪. যোগাযোগের মাধ্যম হিসেবে কাজ করে**

ফ্লোচার্ট প্রোগ্রামার, বিশ্লেষক, ও ক্লায়েন্টদের মধ্যে **সহজ যোগাযোগের মাধ্যম** হিসেবে কাজ করে। সবাই একই চিত্র দেখে সহজেই প্রক্রিয়াটি বুঝতে পারে।

**🧾 ৫. ডকুমেন্টেশনের অংশ হিসেবে**

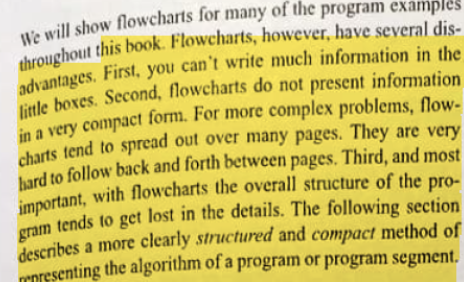
ফ্লোচার্ট প্রজেক্টের **ডকুমেন্টেশন বা রিপোর্ট** হিসেবে রাখা যায়, যাতে ভবিষ্যতে কেউ প্রোগ্রাম পরিবর্তন বা রক্ষণাবেক্ষণ করতে গেলে সহজে বুঝতে পারে।

**🔍 ৬. ডিবাগিং (Debugging) সহজ করে**

ফ্লোচার্ট দেখে বোঝা যায় কোন ধাপে সমস্যা হচ্ছে। এতে **ত্রুটি খোঁজা ও ঠিক করা সহজ** হয়।

**✅ সংক্ষেপে**

ফ্লোচার্ট ব্যবহারের মূল উদ্দেশ্য হলো —  
**প্রোগ্রাম বা প্রক্রিয়াকে চিত্রের মাধ্যমে সহজ, পরিষ্কার ও যৌক্তিকভাবে উপস্থাপন করা।**

****

ছবির হাইলাইট করা অংশে **ফ্লোচার্টের কিছু সীমাবদ্ধতা (disadvantages of flowcharts)** উল্লেখ করা হয়েছে। নিচে এর বাংলা ব্যাখ্যা দেওয়া হলো —

### ⚠️ ****Flowchart-এর অসুবিধাসমূহ:****

1. **তথ্য সীমিতভাবে লেখা যায়**
   * ফ্লোচার্টের ছোট বাক্সগুলোর (boxes) ভিতরে খুব বেশি লেখা যায় না।
   * তাই জটিল তথ্য বা লম্বা বিবরণ ফ্লোচার্টে সহজে উপস্থাপন করা যায় না।
2. **সংক্ষিপ্ত নয় (Not compact)**
   * ফ্লোচার্ট কোনো তথ্যকে খুব সংক্ষিপ্তভাবে প্রকাশ করতে পারে না।
   * বড় বা জটিল সমস্যার ক্ষেত্রে এটি অনেক বড় আকার ধারণ করে এবং বহু পাতায় ছড়িয়ে পড়ে।
3. **বহু পৃষ্ঠায় ছড়িয়ে পড়ে ও অনুসরণ করা কঠিন**
   * জটিল প্রোগ্রামের ফ্লোচার্ট অনেক বড় হয়ে যায়, ফলে একাধিক পৃষ্ঠায় বিভক্ত হয়।
   * এতে এক পৃষ্ঠা থেকে অন্য পৃষ্ঠায় গিয়ে বুঝতে বা অনুসরণ করতে অসুবিধা হয়।
4. **বিস্তারিত তথ্যের মধ্যে কাঠামো হারিয়ে যায়**
   * ফ্লোচার্টে অনেক বিস্তারিত ধাপ দেখানো হয় বলে **মূল কাঠামো (overall structure)** চোখের আড়ালে চলে যায়।
   * প্রোগ্রামের বড় চিত্রটি (big picture) বোঝা কঠিন হয়ে পড়ে।

### 🧾 ****সারসংক্ষেপে:****

ফ্লোচার্ট প্রোগ্রামের প্রবাহ বোঝাতে সাহায্য করে, কিন্তু জটিল প্রোগ্রামের ক্ষেত্রে এটি **দীর্ঘ, অগোছালো এবং অনুসরণে কঠিন** হয়ে যায়। তাই লেখক পরবর্তী অংশে আরও **সংগঠিত (structured)** ও **সংক্ষিপ্ত (compact)** উপস্থাপনার পদ্ধতি বর্ণনা করেছেন।

ছবির হাইলাইট করা অংশে **“Top-down design”** নামে একটি প্রোগ্রামিং পদ্ধতি সম্পর্কে আলোচনা করা হয়েছে। নিচে এর বাংলা ব্যাখ্যা দেওয়া হলো —

**🧭 Top-down Design কী?**

**Top-down design** হলো একটি **পদ্ধতিগত (systematic)** প্রোগ্রামিং পদ্ধতি, যেখানে একটি বড় প্রোগ্রামিং সমস্যাকে ধাপে ধাপে ছোট ছোট অংশে ভাগ করা হয়।

**🧩 প্রক্রিয়াটি কিভাবে কাজ করে:**

1. **বড় সমস্যাকে ভাগ করা হয় (Divide into modules)**  
   প্রথমে বড় প্রোগ্রামকে বিভিন্ন **মূল অংশে (major modules)** ভাগ করা হয়।  
   প্রতিটি module প্রোগ্রামের নির্দিষ্ট একটি কাজ বা ফাংশন সম্পাদন করে।
2. **শীর্ষ স্তরে সারাংশ উপস্থাপন (Top level overview)**  
   প্রোগ্রামের সর্বোচ্চ স্তরে (top level) পুরো প্রোগ্রামের একটি **এক পৃষ্ঠার সারাংশ (one-page overview)** তৈরি করা হয়, যেখানে সব module-এর পারস্পরিক সম্পর্ক ও কাজ বোঝানো হয়।
3. **ধাপে ধাপে বিস্তারিত করা (Stepwise refinement)**  
   এরপর প্রতিটি বড় module-কে আরও ছোট ছোট module-এ ভাগ করা হয়, যতক্ষণ না প্রতিটি ধাপ স্পষ্টভাবে বোঝা যায়।
4. **প্রোগ্রামারদের কাজ বণ্টন (Module assignment)**  
   প্রতিটি প্রোগ্রামারকে এক বা একাধিক module লেখার দায়িত্ব দেওয়া হয়।  
   এতে কাজ ভাগ করা সহজ হয় এবং বড় প্রোগ্রাম টিমে সমন্বয় বজায় থাকে।

**💡 Top-down Design-এর সুবিধা:**

* **সহজ বোঝাপড়া:** প্রোগ্রামের একটি সার্বিক চিত্র এক পৃষ্ঠায় পাওয়া যায়।
* **সহজ শিক্ষণ:** নতুন কেউ প্রোগ্রাম শিখতে চাইলে প্রথমে সারাংশ দেখে ধীরে ধীরে বিস্তারিত অংশে যেতে পারে।
* **সংগঠিত কোড:** প্রতিটি অংশের দায়িত্ব স্পষ্ট থাকে, ফলে প্রোগ্রামটি গঠনমূলকভাবে লেখা যায়।

**🧾 সংক্ষেপে:**

**Top-down design** হলো এমন একটি পদ্ধতি যেখানে প্রোগ্রামটি “উপরে থেকে নিচে” বিশ্লেষণ করা হয় —  
অর্থাৎ, প্রথমে পুরো প্রোগ্রামের সারাংশ তৈরি করা হয়, তারপর ধাপে ধাপে তা ছোট ছোট অংশে ভেঙে বিস্তারিতভাবে নির্মাণ করা হয়।

**🔽 Bottom-Up Design (বটম-আপ ডিজাইন)**

**Bottom-up design** হলো এমন একটি প্রোগ্রামিং বা সিস্টেম ডিজাইন পদ্ধতি যেখানে কাজটি **ছোট ছোট অংশ বা উপাদান (modules/components)** থেকে শুরু করে ধীরে ধীরে **সম্পূর্ণ সিস্টেম তৈরি করা হয়।**

**🧩 প্রক্রিয়াটি কিভাবে কাজ করে:**

1. **ছোট ছোট অংশ তৈরি করা (Building small modules first)**
   * প্রথমে প্রোগ্রামের ছোট ছোট অংশ বা ফাংশন (যেমন ইনপুট নেওয়া, হিসাব করা, আউটপুট দেখানো) তৈরি করা হয়।
   * এই অংশগুলো স্বাধীনভাবে কাজ করতে সক্ষম হয় (self-contained units)।
2. **মডিউলগুলো একত্রিত করা (Integrating modules)**
   * এরপর এই ছোট মডিউলগুলোকে একে অপরের সাথে যুক্ত করে বড় বড় সিস্টেম গঠন করা হয়।
   * ধীরে ধীরে সম্পূর্ণ প্রোগ্রামের কাঠামো তৈরি হয়।
3. **সম্পূর্ণ সিস্টেম তৈরি (Forming the complete system)**
   * সব মডিউল একত্রিত হয়ে পুরো প্রোগ্রাম বা সিস্টেম গঠন করে।
   * প্রতিটি অংশ আগে থেকেই পরীক্ষিত (tested) থাকায় সিস্টেমের স্থায়িত্ব বৃদ্ধি পায়।

**💡 Bottom-up Design-এর সুবিধা:**

* **পুনর্ব্যবহারযোগ্যতা (Reusability):** ছোট মডিউলগুলো অন্য প্রোগ্রামেও ব্যবহার করা যায়।
* **সহজ পরীক্ষা (Easy testing):** প্রতিটি অংশ আলাদা করে পরীক্ষা করা যায়, ফলে ত্রুটি (bug) সহজে ধরা পড়ে।
* **দ্রুত উন্নয়ন (Fast development):** প্রস্তুত মডিউল ব্যবহার করে দ্রুত বড় সিস্টেম তৈরি করা যায়।
* **মজবুত কাঠামো (Strong foundation):** নিচের স্তরগুলো (lower-level modules) ভালোভাবে কাজ করলে উপরের অংশও স্থিতিশীল হয়।

**⚠️ Bottom-up Design-এর অসুবিধা:**

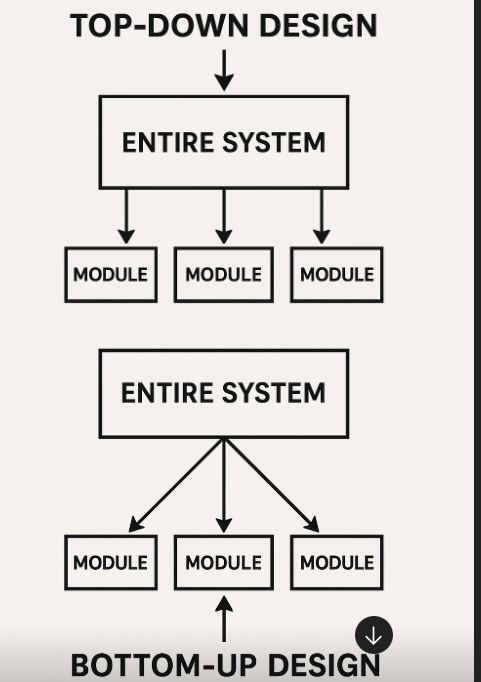
* **প্রাথমিক পরিকল্পনা কঠিন:** শুরুতেই পুরো সিস্টেমের কাঠামো না থাকায় পরবর্তীতে সমন্বয় সমস্যা হতে পারে।
* **সম্পূর্ণ চিত্র দেখা কঠিন:** ছোট অংশ দিয়ে শুরু করলে পুরো প্রোগ্রাম কীভাবে কাজ করবে তা শুরুতে বোঝা কঠিন হতে পারে।

**🔁 Top-down vs Bottom-up তুলনা:**

| **দিক** | **Top-down Design** | **Bottom-up Design** |
| --- | --- | --- |
| শুরু হয় | সার্বিক ধারণা থেকে | ছোট অংশ থেকে |
| কাঠামো | আগে পুরো চিত্র, পরে বিস্তারিত | আগে বিস্তারিত অংশ, পরে পুরো চিত্র |
| কাজের ধারা | “বড় → ছোট” | “ছোট → বড়” |
| সুবিধা | ভালো পরিকল্পনা ও বোঝাপড়া | পুনর্ব্যবহারযোগ্যতা ও স্থিতিশীলতা |
| অসুবিধা | বাস্তবায়নে সময় লাগে | সমন্বয়ে জটিলতা হতে পারে |

**🧾 সংক্ষেপে:**

**Bottom-up design** মানে হলো —  
“ছোট ছোট অংশ তৈরি করে ধীরে ধীরে পুরো প্রোগ্রাম গঠন করা।”  
এটি প্রোগ্রামিংয়ের একটি কার্যকরী পদ্ধতি, বিশেষ করে যখন মডিউলগুলো পুনর্ব্যবহারযোগ্য এবং স্বতন্ত্রভাবে কাজ করতে পারে।

****

ছবিটিতে **প্রোগ্রাম স্ট্রাকচার (Program Structure)** সম্পর্কে ব্যাখ্যা করা হয়েছে, যেখানে IF-THEN, WHILE, REPEAT-UNTIL, এবং FOR এর মতো **কন্ট্রোল স্ট্রাকচার (control structures)** এর ব্যবহার ব্যাখ্যা করা হয়েছে। নিচে সংক্ষিপ্তভাবে গুরুত্বপূর্ণ অংশগুলোর বাংলা সারাংশ দেওয়া হলো —

**🔹 IF-THEN এবং IF-THEN-ELSE**

* এটি একটি **শর্তনির্ভর (conditional)** স্ট্রাকচার।
* যদি কোনো শর্ত সত্য হয়, তবে নির্দিষ্ট কাজটি সম্পন্ন হয়; নাহলে অন্যটি (ELSE অংশে থাকা কাজ) সম্পন্ন হয়।
* Nested IF-THEN-ELSE ব্যবহারে একাধিক শর্ত একসাথে যাচাই করা যায়।
* উদাহরণ:
* IF hungry THEN
* Get food
* ELSE
* Continue next task

**🔹 CASE Structure**

* একাধিক বিকল্পের মধ্যে থেকে একটি নির্দিষ্ট শর্ত অনুযায়ী নির্বাচন করা হয়।
* এটি nested IF-THEN-ELSE এর বিকল্প এবং তুলনামূলকভাবে **সংক্ষিপ্ত ও সহজপাঠ্য**।
* উদাহরণ: CASE day OF … ENDCASE

**🔹 WHILE-DO Loop**

* এটি একটি **repetition (looping)** structure।
* যখন পর্যন্ত শর্ত সত্য থাকবে, তখন পর্যন্ত কাজের ধারা চলতে থাকবে।
* উদাহরণ:
* WHILE money lasts DO
* Eat supper out
* Go to movie
* ENDWHILE

**🔹 REPEAT-UNTIL Loop**

* এটি WHILE লুপের মতো, কিন্তু পার্থক্য হলো —  
  প্রথমে কাজ (action) সম্পন্ন হয়, তারপর শর্ত যাচাই করা হয়।
* অর্থাৎ, **অন্তত একবার লুপটি চলবেই**।
* উদাহরণ:
* REPEAT
* Get data sample
* Add correction factor
* Wait 1 hour
* UNTIL 24 samples taken

**🔹 FOR-DO Loop**

* নির্দিষ্ট সংখ্যক বার কোনো কাজ পুনরাবৃত্তি করতে ব্যবহৃত হয়।
* উদাহরণ:
* FOR count = 1 TO n DO
* statement
* ENDFOR

**🧾 SUMMARY OF PROGRAM STRUCTURE REPRESENTATION**

* সফল প্রোগ্রাম লেখার জন্য শুধু নির্দেশনা (instructions) লেখা যথেষ্ট নয়।
* আগে প্রোগ্রামটি কীভাবে কাজ করবে এবং কীভাবে তা ভাঙা হবে (structured form) — তা নির্ধারণ করা জরুরি।
* এতে প্রোগ্রামটি সহজবোধ্য, সংশোধনযোগ্য এবং রক্ষণাবেক্ষণযোগ্য হয়।

👉 **মূল কথা:**  
এই অধ্যায়টি প্রোগ্রামিংয়ের মৌলিক **কন্ট্রোল স্ট্রাকচারগুলো (selection, repetition, decision-making)** ব্যাখ্যা করে, যা কার্যকর ও সংগঠিত কোড লেখার জন্য অপরিহার্য।

চমৎকার প্রশ্ন — তুমি সম্ভবত **DAA** এবং **AAA** নির্দেশনা (instructions) সম্পর্কে জানতে চাও, যা সাধারণত **8085/8086 মাইক্রোপ্রসেসর** প্রোগ্রামিং-এ ব্যবহৃত হয়। নিচে বিস্তারিতভাবে ব্যাখ্যা করা হলো 👇

## 🧮 **DAA (Decimal Adjust Accumulator)**

### 🔹 ****Full form:****

**DAA → Decimal Adjust Accumulator**

### 🔹 ****ব্যবহার:****

DAA নির্দেশনাটি **BCD (Binary-Coded Decimal)** সংখ্যার সাথে কাজ করার জন্য ব্যবহৃত হয়।  
এটি **অ্যাকিউমুলেটর (Accumulator)**-এর কন্টেন্টকে সঠিক **BCD ফর্মে রূপান্তর** করে।

### ⚙️ ****কাজের প্রক্রিয়া (Operation):****

* কোনো দুটি BCD সংখ্যা যোগ করলে ফলাফলটি বাইনারি আকারে পাওয়া যায়।
* DAA নির্দেশনা ব্যবহার করে সেই ফলাফলকে বৈধ BCD রূপে রূপান্তর করা হয়।

### 🧩 ****নিয়ম:****

1. যদি অ্যাকিউমুলেটরের **lower nibble (D3–D0)** > 9 **অথবা** **Auxiliary Carry (AC)** সেট থাকে →  
   তাহলে **6** যোগ করা হয় অ্যাকিউমুলেটরের lower nibble-এ।
2. তারপর যদি অ্যাকিউমুলেটরের **upper nibble (D7–D4)** > 9 **অথবা** **Carry (CY)** সেট থাকে →  
   তাহলে **60h (0110 0000)** যোগ করা হয় পুরো অ্যাকিউমুলেটরে।

### 💻 ****উদাহরণ:****

MVI A, 25H

ADI 38H

DAA

**Step-by-step:**

1. 25H + 38H = 5DH
2. DAA করার পর → ফলাফল হবে **63H**, অর্থাৎ বৈধ BCD সংখ্যা (৬৩)।

## 🔢 **AAA (ASCII Adjust after Addition)**

### 🔹 ****Full form:****

**AAA → ASCII Adjust for Addition**

### 🔹 ****ব্যবহার:****

AAA নির্দেশনাটি **ASCII কোডযুক্ত সংখ্যাগুলির (0–9)** যোগফল ঠিক করার জন্য ব্যবহৃত হয়, বিশেষত যখন সংখ্যা ASCII ফর্মে থাকে।

### ⚙️ ****কাজের প্রক্রিয়া:****

* ASCII সংখ্যাগুলি সাধারণত 30H–39H পর্যন্ত হয় (অর্থাৎ ‘0’–‘9’)।
* দুটি ASCII মান যোগ করলে ফলাফল সবসময় বৈধ ASCII নাও হতে পারে।
* **AAA** নির্দেশনা সেই যোগফলকে **বৈধ ASCII digit ফর্মে ঠিক করে।**

### 🧩 ****নিয়ম:****

1. যদি lower nibble > 9 **অথবা** **Auxiliary Carry (AF)** সেট থাকে →
   * তাহলে **AL = AL + 06H**
   * এবং **AH = AH + 01H**
   * **AF ও CF সেট হয়।**
2. যদি না হয় →
   * **AF ও CF রিসেট হয়।**
   * Upper nibble পরিষ্কার করা হয় (AL-এর high nibble = 0)।

### 💻 ****উদাহরণ (8086):****

MOV AL, 35H ; '5'

ADD AL, 37H ; '7'

AAA

**Step-by-step:**

1. 35H + 37H = 6CH
2. AAA করার পর → AL = 02H, AH = 01H  
   অর্থাৎ ফলাফল ‘12’ (ASCII রূপে বৈধ)।

## 🧾 **সংক্ষিপ্ত তুলনা:**

| **দিক** | **DAA** | **AAA** |
| --- | --- | --- |
| পূর্ণরূপ | Decimal Adjust Accumulator | ASCII Adjust after Addition |
| ব্যবহৃত হয় | BCD সংখ্যার জন্য | ASCII সংখ্যার জন্য |
| প্রসেসর | 8085, 8086 | 8086 |
| কার্য | যোগফলকে বৈধ BCD বানানো | যোগফলকে বৈধ ASCII বানানো |

### ✅ ****সংক্ষেপে:****

🔹 **DAA** ব্যবহার হয় **BCD গণনায়**  
🔹 **AAA** ব্যবহার হয় **ASCII সংখ্যার যোগফলে**  
দুটোই “adjust” নির্দেশনা — যা যোগফলের পরে সঠিক দশমিক বা ASCII ফলাফল দেয়।

চাও কি আমি এর জন্য একটা **চিত্রসহ ব্যাখ্যা (diagram)** তৈরি করে দিই — যেখানে DAA ও AAA দুটোই ভিজ্যুয়ালভাবে দেখানো থাকবে?

অবশ্যই 👍 নিচে **DAA** এবং **AAA** — এই দুটি নির্দেশনার জন্য **প্রত্যেকের ২টি করে উদাহরণ** (সহ ব্যাখ্যা) দেওয়া হলো:

## 🧮 **👉 DAA (Decimal Adjust Accumulator) Examples**

### 🧠 ****Example 1: Simple BCD Addition****

MVI A, 25H ; Load 25 (BCD)

ADI 38H ; Add 38 (BCD)

DAA ; Adjust to valid BCD

**Step by step:**

* 25H + 38H = 5DH
* 5D → বৈধ BCD নয় (lower nibble D = 13 > 9)
* DAA করার পর lower nibble-এ 6 যোগ হয় → 63H  
  ✅ **Final Answer = 63 (BCD)**

### 🧠 ****Example 2: Addition with Carry****

MVI A, 79H

ADI 85H

DAA

**Step by step:**

* 79H + 85H = FEH
* Lower nibble (E = 14 > 9) → +6 = 104 → Carry = 1
* Upper nibble (9 + carry = A > 9) → +60H যোগ হয়  
  ✅ **Final Answer = 64H, Carry = 1**  
  👉 অর্থাৎ BCD রূপে ফলাফল = 64 + carry = **164 (BCD)**

## 🔢 **👉 AAA (ASCII Adjust After Addition) Examples**

### 🧠 ****Example 1: ASCII Digits 3 + 4****

MOV AL, 33H ; ASCII for '3'

ADD AL, 34H ; ASCII for '4'

AAA

**Step by step:**

* 33H + 34H = 67H
* Lower nibble (7 < 9), তাই কোনো adjust দরকার নেই
* AAA করার পর:
  + AL = 07H
  + AH = 00H  
    ✅ **Result = ‘7’ (ASCII 37H)**

### 🧠 ****Example 2: ASCII Digits 8 + 5****

MOV AL, 38H ; ASCII for '8'

ADD AL, 35H ; ASCII for '5'

AAA

**Step by step:**

* 38H + 35H = 6DH
* Lower nibble (D = 13 > 9) → adjust দরকার
* AL = AL + 06H = 73H, AH = AH + 01H = 01H  
  ✅ **Result = 01 03H → ASCII for “13”**

### 🧾 ****সংক্ষিপ্তভাবে মনে রাখো:****

| **Instruction** | **কাজ** | **উদাহরণ** | **ফলাফল** |
| --- | --- | --- | --- |
| **DAA** | BCD যোগের ফলাফল ঠিক করা | 25H + 38H → DAA | 63H |
| **DAA** | BCD যোগে carry সহ | 79H + 85H → DAA | 64H + Carry |
| **AAA** | ASCII ‘3’ + ‘4’ | 33H + 34H → AAA | 37H (‘7’) |
| **AAA** | ASCII ‘8’ + ‘5’ | 38H + 35H → AAA | 313H (“13”) |

চাও কি আমি এগুলোর জন্য একটা **চিত্র (diagram)** তৈরি করি যেখানে DAA ও AAA এর ধাপগুলো ভিজ্যুয়ালি দেখানো থাকবে? এতে বোঝা আরও সহজ হবে।

ছবিটিতে **8086 মাইক্রোপ্রসেসর প্রোগ্রামিং**-এর একটি গুরুত্বপূর্ণ বিষয় — **Initialization Instructions এবং Program Format** — নিয়ে আলোচনা করা হয়েছে। নিচে এর মূল বিষয়বস্তু বাংলায় সহজভাবে ব্যাখ্যা করা হলো 👇

## 🧭 **Writing a Program (প্রোগ্রাম লেখা)**

### 🔹 ****Initialization Instructions (সূচনা নির্দেশনা)****

প্রোগ্রাম লেখার আগে কিছু নির্দিষ্ট অংশ **ইনিশিয়ালাইজ (initialize)** করতে হয়।  
এই নির্দেশনাগুলির মূল উদ্দেশ্য হলো —

প্রোগ্রাম চালানোর আগে সিস্টেমের প্রয়োজনীয় অংশগুলো যেমন **segment registers**, **flags**, এবং **programmable port devices** সঠিকভাবে প্রস্তুত করা।

### ⚙️ ****বিস্তারিত ব্যাখ্যা:****

1. **Segment Registers ইনিশিয়ালাইজ করা:**
   * এগুলো (যেমন DS, SS, ES, CS) ১৬-বিটের রেজিস্টার, যা মেমোরির বিভিন্ন অংশ নির্দেশ করে।
   * উদাহরণ: Data Segment (DS) রেজিস্টারে ডেটা স্টোর করার জায়গার ঠিকানা রাখা হয়।
   * 8086 সরাসরি segment register-এ ডেটা স্থানান্তর করতে পারে না।  
     তাই একটি **general-purpose register** (যেমন AX) ব্যবহার করতে হয়:
   * MOV AX, 2000H
   * MOV DS, AX

এখানে প্রথমে AX-এ ডেটা রাখা হয়েছে, পরে DS-এ কপি করা হয়েছে।

1. **Stack Initialization:**
   * যদি প্রোগ্রামে stack ব্যবহৃত হয়, তাহলে **Stack Segment (SS)** এবং **Stack Pointer (SP)** ঠিকভাবে সেট করতে হয়।
2. **Peripheral Devices Initialization:**
   * কিছু প্রোগ্রামে বাহ্যিক ডিভাইস (যেমন 8255 PPI, 8259 PIC) ব্যবহৃত হয়।
   * সেগুলোকেও প্রোগ্রামের শুরুতে ইনিশিয়ালাইজ করতে হয়।

### 🧾 ****Initialization List (উদাহরণ তালিকা):****

নিচের মতো একটি তালিকা সাধারণত ব্যবহার করা হয়:

Data segment register DS

Stack segment register SS

Extra segment register ES

Stack pointer register SP

8255 programmable parallel port

8259 programmable interrupt controller

8254 programmable counter

Initialize data variables

Set interrupt enable flag

👉 **লক্ষ্য:** প্রোগ্রামের জন্য প্রয়োজনীয় রেজিস্টার ও ডিভাইস সঠিকভাবে ইনিশিয়ালাইজ করা, যাতে প্রোগ্রাম সঠিকভাবে কাজ শুরু করতে পারে।

## 🧩 **A Standard Program Format (প্রোগ্রামের স্ট্যান্ডার্ড ফরম্যাট)**

এই অংশে বোঝানো হয়েছে কীভাবে একটি প্রোগ্রাম **assembly language coding format**-এ লেখা হয়।

### 🧠 ****ফরম্যাটে তিনটি মূল কলাম থাকে:****

1. **ADDRESS Column:** মেমোরি অ্যাড্রেস বা কোডের অবস্থান নির্দেশ করে।
2. **MNEMONIC Column:** ইনস্ট্রাকশনের নাম (যেমন MOV, ADD, SUB)।
3. **OPERAND(S) Column:** রেজিস্টার বা মেমোরি লোকেশন নির্দেশ করে।
4. **COMMENTS Column:** ঐচ্ছিকভাবে ব্যাখ্যা লেখা হয়।

### 💻 ****উদাহরণ (Fig. 3.4 অনুযায়ী):****

MOV AL, 7

ADD AL, 4

STORE RESULT

এটি “read temperature, add +7, and store result” ধাঁচের একটি উদাহরণ প্রোগ্রাম।

## 🧩 **Key Takeaways (মূল ধারণা):**

| **বিষয়** | **ব্যাখ্যা** |
| --- | --- |
| **Initialization Instructions** | প্রোগ্রাম চালানোর আগে রেজিস্টার ও ডিভাইসগুলো প্রস্তুত করা |
| **Segment Register Loading** | 8086 সরাসরি ডেটা নিতে পারে না, তাই General Register ব্যবহার করতে হয় |
| **Initialization Checklist** | DS, SS, ES, SP, Flags, এবং I/O Ports ঠিকভাবে সেট করা |
| **Standard Format** | Address + Mnemonic + Operand + Comment ফরম্যাটে লেখা হয় |

🔹 **সহজভাবে বললে:**  
8086 প্রোগ্রাম লেখার প্রথম ধাপ হলো —

“সিস্টেমকে ঠিকভাবে প্রস্তুত করা” (Initialization)।  
এরপরেই মূল নির্দেশনাগুলো কার্যকর হয়।  
সঠিক ইনিশিয়ালাইজেশন ছাড়া প্রোগ্রাম কখনোই নির্ভরযোগ্যভাবে কাজ করবে না।

নিচে **“Constructing the Machine Codes for 8086 Instructions”** বিষয়টি বাংলায় সহজভাবে ব্যাখ্যা করা হলো 👇

## 🧩 **8086 Machine Code কীভাবে তৈরি হয়**

8086 মাইক্রোপ্রসেসরে প্রতিটি **assembly language instruction** (যেমন MOV, ADD, SUB, INC) বাস্তবে এক বা একাধিক **বাইনারি কোড** দিয়ে প্রকাশ করা হয়। এই বাইনারি কোডকেই বলে **Machine Code**।  
👉 অর্থাৎ, **Machine Code = Instruction-এর বাইনারি বা হেক্সাডেসিমাল রূপ**, যা CPU সরাসরি বুঝতে পারে।

## ⚙️ **Machine Code গঠনের ধাপগুলো**

### 🔹 ১. Opcode (Operation Code)

* প্রতিটি নির্দেশনার জন্য একটি নির্দিষ্ট **opcode** থাকে।
* এটি নির্দেশ করে CPU কী কাজ করবে — যেমন যোগ করা, কপি করা, বিয়োগ করা ইত্যাদি।
* উদাহরণ:
  + MOV → Opcode = 100010
  + ADD → Opcode = 000000

### 🔹 ২. Addressing Mode (ঠিকানার ধরন)

* নির্দেশনায় ডেটা কোথায় আছে বা কোথায় যাবে — সেটি **addressing mode** দ্বারা নির্ধারিত হয়।
* উদাহরণ:
  + Register to Register
  + Immediate to Register
  + Register to Memory

### 🔹 ৩. Register / Memory Codes

* 8086 এ প্রতিটি রেজিস্টারের একটি **৩-বিট কোড** থাকে।  
  যেমন:

| **Register** | **কোড (Binary)** |
| --- | --- |
| AX | 000 |
| BX | 011 |
| CX | 001 |
| DX | 010 |

* এই কোডগুলো opcode-এর সাথে যুক্ত হয়ে পূর্ণ মেশিন কোড তৈরি করে।

### 🔹 ৪. ModR/M Byte (Mode + Register + R/M)

* **ModR/M byte**-এ তিনটি অংশ থাকে:
  + **MOD (2 bits):** Addressing mode
  + **REG (3 bits):** কোন রেজিস্টার ব্যবহৃত হচ্ছে
  + **R/M (3 bits):** ডেটা কোথা থেকে আসছে বা কোথায় যাচ্ছে

### 🔹 ৫. Displacement ও Immediate Data

* যদি কোনো নির্দেশনায় **memory address** বা **immediate value (যেমন 25H)** থাকে, তবে তা opcode-এর পর যুক্ত হয়।

## 🧠 **উদাহরণ ১: MOV AX, BX**

* **Instruction:** MOV AX, BX
* **অর্থ:** BX-এর মান AX-এ স্থানান্তর করো
* **Opcode:** 100010
* **Direction bit (D):** 1 (destination = register)
* **Word bit (W):** 1 (16-bit data)
* **ModR/M byte:** 11 (register mode) + 000 (AX) + 011 (BX) → 11000011

👉 **Final Machine Code:**  
10001011 11000011 → **8B C3 (Hexadecimal)**

## 🧠 **উদাহরণ ২: ADD AL, BL**

* **Instruction:** ADD AL, BL
* **Opcode:** 000000
* **D = 1, W = 0 (8-bit)**
* **ModR/M byte:** 11 000 011 → C3

👉 **Final Machine Code:**  
00000011 11000011 → **03 C3 (Hexadecimal)**

## 🧾 **সংক্ষেপে: Machine Code তৈরি করার ধাপগুলো**

| **ধাপ** | **উপাদান** | **কাজ** |
| --- | --- | --- |
| ১ | Opcode | কাজের ধরন নির্ধারণ করে |
| ২ | D & W bits | ডেটার দিক ও আকার নির্ধারণ করে |
| ৩ | MOD | Addressing Mode |
| ৪ | REG | কোন রেজিস্টার ব্যবহার হচ্ছে |
| ৫ | R/M | কোথা থেকে/কোথায় ডেটা যাচ্ছে |
| ৬ | Displacement | যদি memory address থাকে |
| ৭ | Immediate Data | যদি মান সরাসরি দেওয়া হয় |

### ✅ ****মূল কথা:****

8086 মেশিন কোড গঠিত হয় বিভিন্ন বিটের সমন্বয়ে — যেখানে opcode, register কোড এবং addressing mode মিলে CPU-কে বলে দেয় **কী কাজ করবে এবং কোথায় করবে**।

### 💾 ****Memory Interface (মেমরি ইন্টারফেস)****

মেমরি ইন্টারফেস হলো এমন একটি **যোগাযোগ ব্যবস্থা**, যার মাধ্যমে **মাইক্রোপ্রসেসর** (যেমন 8085 বা 8086) মেমরি (RAM, ROM ইত্যাদি)-র সাথে **ডেটা বিনিময় (data exchange)** করে। এটি CPU এবং মেমরির মধ্যে ডেটা, ঠিকানা, ও নিয়ন্ত্রণ সংকেত পাঠানো-নেওয়ার দায়িত্ব পালন করে।

## 🧩 **Memory Interface-এর মূল উপাদানসমূহ**

### 1. ****Address Bus (ঠিকানা বাস)****

* এটি CPU থেকে মেমরিতে **ঠিকানা পাঠানোর** জন্য ব্যবহৃত হয়।
* 8086-এর ক্ষেত্রে এটি **20-bit** (A0–A19), ফলে 2²⁰ = 1MB পর্যন্ত মেমরি অ্যাড্রেস করা যায়।
* উদাহরণ: যদি CPU মেমরির কোনো নির্দিষ্ট লোকেশন (যেমন 0005H) অ্যাক্সেস করতে চায়, তবে সেই ঠিকানাটি Address Bus-এর মাধ্যমে পাঠানো হয়।

### 2. ****Data Bus (ডেটা বাস)****

* এটি CPU এবং মেমরির মধ্যে **ডেটা পাঠানো ও গ্রহণের** পথ।
* 8086-এ ডেটা বাসটি **16-bit** (D0–D15), ফলে একবারে 2 বাইট (word) ডেটা আদানপ্রদান করা যায়।
* ডেটা দুই দিকেই প্রবাহিত হতে পারে (bidirectional)।

### 3. ****Control Bus (নিয়ন্ত্রণ বাস)****

CPU কখন মেমরি থেকে ডেটা পড়বে বা লিখবে, সেটি নিয়ন্ত্রণ করে এই সিগন্যালগুলো:

| **সিগন্যাল** | **কাজ** |
| --- | --- |
| **MEMR’ (Memory Read)** | CPU মেমরি থেকে ডেটা পড়ছে |
| **MEMW’ (Memory Write)** | CPU মেমরিতে ডেটা লিখছে |
| **IO/M’** | CPU I/O না মেমরির সাথে কাজ করছে তা নির্দেশ করে |
| **RD’ / WR’** | সাধারণ পড়া বা লেখা অপারেশন নির্দেশ করে |

## ⚙️ **8086 Memory Interface-এর কাজের ধাপ**

ধরা যাক CPU মেমরি থেকে কোনো ডেটা পড়বে —

1. CPU Address Bus-এর মাধ্যমে মেমরিতে ঠিকানা পাঠায়।
2. CPU **MEMR’ (Memory Read)** সিগন্যাল সক্রিয় করে।
3. মেমরি সেই ঠিকানার ডেটা **Data Bus**-এ পাঠায়।
4. CPU সেই ডেটা গ্রহণ করে রেজিস্টারে সংরক্ষণ করে।

📤 লেখার (Write) সময় ঠিক উল্টো — CPU ডেটা পাঠায় এবং **MEMW’** সিগন্যাল সক্রিয় করে।

## 🧱 **8086-এ Memory Organization**

8086 মাইক্রোপ্রসেসরটি দুটি 8-bit মেমরি ব্যাংক ব্যবহার করে:

| **ব্যাংক** | **বাস লাইন** | **ডেটা** |
| --- | --- | --- |
| **Lower Bank** | D0–D7 | Even Address (যেমন 0000H, 0002H) |
| **Upper Bank** | D8–D15 | Odd Address (যেমন 0001H, 0003H) |

CPU যখন 16-bit word পড়ে বা লেখে, তখন দুটো ব্যাংক একসাথে কাজ করে।

## 💡 **উদাহরণ: Memory Read Cycle**

| **ধাপ** | **কার্যক্রম** |
| --- | --- |
| 1️⃣ | CPU Address Bus-এ মেমরি লোকেশনের ঠিকানা দেয় |
| 2️⃣ | CPU MEMR’ সিগন্যাল পাঠায় |
| 3️⃣ | মেমরি নির্দিষ্ট লোকেশনের ডেটা Data Bus-এ পাঠায় |
| 4️⃣ | CPU ডেটা গ্রহণ করে এবং RD’ সিগন্যাল বন্ধ করে দেয় |

## 🧾 **সংক্ষেপে:**

| **উপাদান** | **কাজ** |
| --- | --- |
| **Address Bus** | ঠিকানা পাঠানো |
| **Data Bus** | ডেটা আদানপ্রদান |
| **Control Bus** | পড়া/লেখার নিয়ন্ত্রণ |
| **Memory Read Cycle** | CPU → মেমরি থেকে ডেটা গ্রহণ |
| **Memory Write Cycle** | CPU → মেমরিতে ডেটা প্রেরণ |

### ✅ ****সারসংক্ষেপে:****

**Memory Interface** হচ্ছে CPU ও Memory-এর মধ্যে সংযোগ ব্যবস্থা, যা Address Bus, Data Bus, এবং Control Signals ব্যবহার করে তথ্যের আদানপ্রদান নিয়ন্ত্রণ করে।

চাও কি আমি এর একটা **চিত্র (diagram)** তৈরি করে দিই, যেখানে 8086 Memory Interface দেখানো থাকবে — Address Bus, Data Bus, Control Signals সহ?

নিচে উপরের অনুচ্ছেদটির বাংলা ব্যাখ্যা ও সংক্ষিপ্ত সারাংশ দেওয়া হলো 👇

## 💾 **ROM Memory (Read Only Memory)**

### 🔹 ****ROM কী:****

ROM বা **Read-Only Memory** হলো এমন এক ধরনের মেমরি যা **স্থায়ীভাবে (permanently)** ডেটা বা প্রোগ্রাম সংরক্ষণ করে।  
এটি এমন তথ্য ধারণ করে যা **পাওয়ার বন্ধ হলেও মুছে যায় না** — তাই একে বলা হয় **non-volatile memory**।

### ⚙️ ****ROM-এর কাজ:****

* এতে থাকা প্রোগ্রাম ও ডেটা সিস্টেমের জন্য **স্থায়ীভাবে প্রয়োজনীয়**।
* এই তথ্য **কারখানায় (factory)** প্রোগ্রাম করা হয় এবং পরবর্তীতে পরিবর্তন করা যায় না।
* তাই এটি সাধারণত ব্যবহৃত হয় যেখানে প্রোগ্রাম **স্থায়ী ও অপরিবর্তনীয়** (যেমন BIOS, embedded system firmware ইত্যাদি)।

## 🧩 **ROM-এর ধরনসমূহ**

### 1️⃣ ****ROM (Read Only Memory)****

* এটি সম্পূর্ণভাবে **কারখানায় প্রোগ্রাম করা** হয়।
* ব্যবহারকারী পরে এতে কিছু পরিবর্তন করতে পারে না।
* সাধারণত প্রচুর পরিমাণে (কমপক্ষে ১০,০০০ পিস বা তার বেশি) তৈরি করা হয়, কারণ **ফ্যাক্টরি প্রোগ্রামিং খরচ** অনেক বেশি।
* ব্যবহারের উদাহরণ: **মাইক্রোকন্ট্রোলার, BIOS, গেম কনসোল ইত্যাদি।**

### 2️⃣ ****PROM (Programmable Read Only Memory)****

* নামের মতোই এটি ব্যবহারকারী **নিজে প্রোগ্রাম করতে পারেন** (field-programmable)।
* কিন্তু একবার প্রোগ্রাম করলে আর পরিবর্তন বা মুছা যায় না।
* প্রোগ্রাম করার সময় ছোট **Ni-Chrome বা Silicon Oxide fuse** পুড়িয়ে খোলা হয়, যা স্থায়ী পরিবর্তন ঘটায়।
* ব্যবহারের পরে **erase করা সম্ভব নয়।**

### 3️⃣ ****EPROM (Erasable Programmable ROM)****

* এটি এমন ROM যা **ব্যবহারকারী নিজে প্রোগ্রাম ও মুছতে** পারেন।
* প্রোগ্রামিং করা হয় একটি বিশেষ যন্ত্রে — **EPROM Programmer**।
* ডেটা **উচ্চ তীব্রতার (high-intensity) আল্ট্রাভায়োলেট (UV) আলোতে** প্রায় ২০ মিনিট ধরে রাখলে মুছে ফেলা যায়।
* তাই এটি তখন ব্যবহৃত হয় যখন প্রোগ্রাম বারবার পরিবর্তনের প্রয়োজন হয়, অথবা **কম সংখ্যায় উৎপাদন** করা হয়।

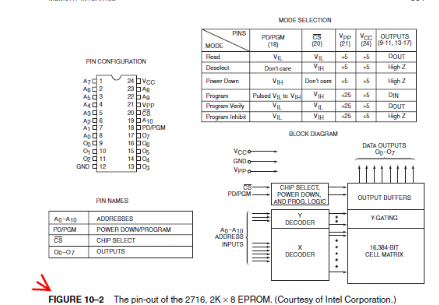
## 🧠 **তুলনামূলক টেবিল:**

| **মেমরির ধরন** | **প্রোগ্রাম করা যায়** | **মুছা যায়** | **ব্যবহার ক্ষেত্র** |
| --- | --- | --- | --- |
| **ROM** | ফ্যাক্টরিতে | না | স্থায়ী প্রোগ্রাম, BIOS |
| **PROM** | একবার ফিল্ডে | না | কাস্টম প্রোগ্রাম, টেস্টিং |
| **EPROM** | ফিল্ডে | হ্যাঁ (UV আলোতে) | পরীক্ষামূলক বা পরিবর্তনযোগ্য সফটওয়্যার |

### 🧾 ****সংক্ষেপে:****

ROM হলো **non-volatile memory**, যেখানে তথ্য বিদ্যুৎ চলে গেলেও সংরক্ষিত থাকে।  
PROM একবার প্রোগ্রাম করা যায়, EPROM বারবার প্রোগ্রাম ও মুছা যায়।  
ROM সিস্টেমের **স্থায়ী প্রোগ্রাম সংরক্ষণের মূল মাধ্যম** হিসেবে ব্যবহৃত হয়।

চাও কি আমি এর জন্য একটি **চিত্র (diagram)** তৈরি করে দিই, যেখানে ROM, PROM, এবং EPROM-এর পার্থক্য ভিজ্যুয়ালি দেখানো থাকবে?

****

ছবিটিতে দেখানো হয়েছে **Intel 2716 EPROM (2K × 8)**–এর **পিন কনফিগারেশন**, **মোড সিলেকশন**, ও **ব্লক ডায়াগ্রাম**। নিচে এর বাংলা ব্যাখ্যা দেওয়া হলো 👇

## 💾 **2716 EPROM Overview**

**EPROM (Erasable Programmable Read-Only Memory)** হলো এক ধরনের **non-volatile memory**, যেখানে ডেটা সংরক্ষিত থাকে এমনকি বিদ্যুৎ বন্ধ থাকলেও।  
এটি প্রোগ্রাম করা যায় এবং পরে **UV আলো** ব্যবহার করে মুছে ফেলা যায়।

## 🧩 **Capacity:**

* **2716 EPROM = 2K × 8 bits = 2048 bytes (2 KB)**  
  অর্থাৎ, এতে মোট 2048 লোকেশনে 8-বিট ডেটা সংরক্ষণ করা যায়।

## ⚙️ **Pin Configuration (২৪ পিনের ব্যাখ্যা)**

| **পিন নম্বর** | **নাম** | **কাজ** |
| --- | --- | --- |
| 1–10, 21 | **A0–A10** | মেমরি লোকেশনের ঠিকানা ইনপুট (11-bit Address Bus) |
| 11–13, 15–17 | **O0–O7** | ডেটা আউটপুট (8-bit Data Bus) |
| 18 | **PD/PGM** | Power Down / Program Mode Control |
| 20 | **CE̅ (Chip Enable)** | চিপ সক্রিয়/নিষ্ক্রিয় করে |
| 21 | **OE̅ (Output Enable)** | ডেটা আউটপুট নিয়ন্ত্রণ করে |
| 22 | **VPP** | প্রোগ্রামিং ভোল্টেজ ইনপুট (সাধারণত +25V) |
| 24 | **VCC** | Power supply (+5V) |
| 12 | **GND** | Ground |

## 🔋 **Mode Selection (Operating Modes)**

| **Mode** | **PD/PGM** | **CE̅** | **VPP** | **VCC** | **Output Pins** | **কাজ** |
| --- | --- | --- | --- | --- | --- | --- |
| **Read** | VIL | VIL | +5V | +5V | Dout | ডেটা পড়ে |
| **Deselect** | X | VIH | +5V | +5V | High-Z | আউটপুট বন্ধ |
| **Power Down** | VIH | X | +5V | +5V | High-Z | শক্তি সঞ্চয় (low power) |
| **Program** | Pulse VIL | VIL | +25V | +5V | Din | ডেটা প্রোগ্রাম করা |
| **Program Verify** | VIL | VIL | +25V | +5V | Dout | প্রোগ্রাম যাচাই করা |
| **Program Inhibit** | VIH | VIH | +25V | +5V | High-Z | প্রোগ্রাম বন্ধ রাখা |

## 🧱 **Internal Block Diagram Explanation**

1. **Address Inputs (A0–A10):**  
   মেমরির নির্দিষ্ট সেল নির্বাচন করতে ব্যবহৃত হয়।  
   এগুলো **X Decoder** এবং **Y Decoder**-এ যায়।
2. **Chip Select (CE̅):**  
   চিপটি সক্রিয় বা নিষ্ক্রিয় করে। নিষ্ক্রিয় থাকলে আউটপুট High-Z অবস্থায় যায়।
3. **VPP (Program Voltage):**  
   প্রোগ্রাম করার সময় ২৫ ভোল্ট প্রয়োগ করা হয়।
4. **Output Buffers:**  
   নির্বাচিত মেমরি সেল থেকে ডেটা **O0–O7** আউটপুট লাইনে দেয়।
5. **Cell Matrix:**  
   এখানে প্রকৃত ডেটা সংরক্ষিত থাকে (১৬,৩৮৪ বিট বা 2K × 8 মেমরি সেল)।

## ⚡ **Programming and Erasing Process**

* **Programming:**
  + VPP = +25V
  + নির্দিষ্ট address নির্বাচন করা হয়
  + ডেটা “লেখা” হয় PD/PGM পিনে পালস প্রয়োগের মাধ্যমে।
* **Erasing:**
  + EPROM চিপের কাচের জানালার (quartz window) মাধ্যমে **UV আলো** প্রায় ২০ মিনিট প্রয়োগ করা হয়।
  + এতে চার্জ মুছে যায় এবং সব বিট আবার 1 (FFH) হয়ে যায়।

## 🧾 **সংক্ষেপে:**

| **বৈশিষ্ট্য** | **বিবরণ** |
| --- | --- |
| মেমরি সাইজ | 2K × 8 (2048 bytes) |
| ভোল্টেজ | VCC = +5V, VPP = +25V |
| মেমরি টাইপ | EPROM (UV Erasable) |
| অ্যাড্রেস লাইন | 11 |
| ডেটা লাইন | 8 |
| প্রধান পিন | A0–A10, O0–O7, CE̅, PD/PGM, VPP, VCC, GND |
| প্রোগ্রাম/ইরেজ | Electrical write, UV erase |

চাও কি আমি এর উপর ভিত্তি করে একটি **সহজ লেবেল করা ডায়াগ্রাম** (বাংলা ট্যাগসহ) তৈরি করি — যাতে EPROM 2716-এর পিন ও ব্লক ডায়াগ্রাম ভিজ্যুয়ালি বোঝানো যায়?

চমৎকার প্রশ্ন 👍 — নিচে **Static RAM (SRAM)** সম্পর্কে বিস্তারিতভাবে বাংলা ব্যাখ্যা দেওয়া হলো 👇

## 💾 **Static RAM (SRAM)**

### 🔹 ****SRAM কী:****

**SRAM (Static Random Access Memory)** হলো এমন এক ধরনের **volatile memory**, যা ডেটা সংরক্ষণ করতে **ফ্লিপ-ফ্লপ (flip-flop)** সার্কিট ব্যবহার করে।  
এটি ডেটা ধরে রাখে যতক্ষণ পর্যন্ত বিদ্যুৎ সরবরাহ (power supply) চালু থাকে।

🔸 বিদ্যুৎ চলে গেলে SRAM-এর সব ডেটা মুছে যায়।

## ⚙️ **SRAM-এর মূল বৈশিষ্ট্যসমূহ**

| **বৈশিষ্ট্য** | **বর্ণনা** |
| --- | --- |
| **Full form** | Static Random Access Memory |
| **প্রকৃতি** | Volatile (বিদ্যুৎ গেলে ডেটা মুছে যায়) |
| **ডেটা সংরক্ষণ উপায়** | Flip-Flop সার্কিট ব্যবহার করে |
| **Speed** | খুব দ্রুত (Fast Access Time) |
| **Refresh প্রয়োজন হয়?** | ❌ না (DRAM-এর মতো বারবার রিফ্রেশ করতে হয় না) |
| **Power Consumption** | তুলনামূলক বেশি |
| **Cost** | বেশি দামী (per bit storage) |
| **ব্যবহার ক্ষেত্র** | Cache Memory, Register File, Embedded Systems ইত্যাদি |

## 🧩 **SRAM-এর গঠন (Structure)**

একটি SRAM সেল সাধারণত **৬টি ট্রানজিস্টর (6T)** দিয়ে তৈরি হয় —

* ৪টি ট্রানজিস্টর মিলে তৈরি করে **একটি bistable flip-flop** (যা ১ বা ০ অবস্থায় থাকতে পারে)
* বাকি ২টি ট্রানজিস্টর কাজ করে **Access Control** হিসেবে (word line ও bit line নিয়ন্ত্রণে)

🧠 প্রতিটি সেল এক বিট (bit) তথ্য সংরক্ষণ করে।

## ⚙️ **SRAM কাজ করার প্রক্রিয়া**

### 1️⃣ ****Write Operation (ডেটা লেখা):****

* CPU যে মান (0 বা 1) সংরক্ষণ করতে চায়, তা **Bit Line** দিয়ে SRAM-এ পাঠায়।
* **Word Line** সক্রিয় হলে Access Transistor চালু হয়, এবং ডেটা Flip-Flop-এ লোড হয়।

### 2️⃣ ****Read Operation (ডেটা পড়া):****

* **Word Line** আবার সক্রিয় হয়,
* Flip-Flop তার সংরক্ষিত মান Bit Line-এ পাঠায়, যেখান থেকে CPU সেটি পড়ে নেয়।

## 🔋 **SRAM বনাম DRAM তুলনা**

| **বিষয়** | **SRAM** | **DRAM** |
| --- | --- | --- |
| **গঠন** | 6 Transistor Flip-Flop | 1 Transistor + 1 Capacitor |
| **Speed** | বেশি (Fast) | ধীর (Slow) |
| **Refresh দরকার** | না | হ্যাঁ (নিয়মিত) |
| **Power Consumption** | বেশি | কম |
| **Cost** | বেশি | কম |
| **ব্যবহার** | Cache, Register | Main Memory (RAM) |

## 🧠 **SRAM-এর উদাহরণ:**

* **Intel 2147 (4K × 1 bit SRAM)**
* **6116 (2K × 8 bit Static RAM chip)**
* **Cache memory inside Intel 8086/8088 processors**

## 🧾 **সংক্ষেপে:**

**Static RAM (SRAM)** হলো দ্রুতগতি সম্পন্ন মেমরি যা **flip-flop সার্কিটের মাধ্যমে ডেটা সংরক্ষণ করে**,  
এটি **রিফ্রেশ ছাড়াই কাজ করে**, কিন্তু **বেশি দামী ও শক্তি-খরচকারী**।  
তাই এটি সাধারণত **Cache Memory** হিসেবে ব্যবহৃত হয়।

চাও কি আমি এর একটি **চিত্র (diagram)** তৈরি করে দিই — যেখানে 6-transistor SRAM cell ও তার read/write operation ভিজ্যুয়ালি বোঝানো থাকবে?

### 💾 ****Dynamic RAM (DRAM)****

## 🧩 **DRAM কী:**

**DRAM (Dynamic Random Access Memory)** হলো এমন এক ধরনের **volatile memory**, যেখানে প্রতিটি ডেটা বিট একটি **capacitor এবং transistor** এর মাধ্যমে সংরক্ষিত হয়।  
এটি **SRAM-এর মতো দ্রুত নয়**, তবে **সস্তা এবং বেশি ডেটা ধারণক্ষম**।

⚡ “Dynamic” বলা হয় কারণ মেমরির প্রতিটি সেলে থাকা চার্জ **নিয়মিত রিফ্রেশ (refresh)** করতে হয়, নাহলে ডেটা হারিয়ে যায়।

## ⚙️ **DRAM-এর গঠন (Structure)**

একটি DRAM সেল গঠিত হয়:

* **১টি Transistor** (switch হিসেবে কাজ করে)
* **১টি Capacitor** (চার্জ ধরে রাখে)

🧠 প্রতিটি সেল এক বিট ডেটা (0 বা 1) সংরক্ষণ করে।

* Capacitor-এ চার্জ থাকলে → 1
* চার্জ না থাকলে → 0

## 🔄 **কেন “Dynamic”?**

কারণ capacitors ধীরে ধীরে চার্জ হারায়, তাই DRAM-এ সংরক্ষিত ডেটা **কিছু মিলিসেকেন্ড পর পর পুনরায় চার্জ (refresh)** করতে হয়।  
এই প্রক্রিয়াটিকেই বলে **Memory Refresh Cycle**।

## ⚙️ **DRAM-এর কাজের প্রক্রিয়া**

### 1️⃣ ****Write Operation (ডেটা লেখা):****

* CPU যে ডেটা সংরক্ষণ করতে চায় (0 বা 1), তা bit line দিয়ে পাঠায়।
* Word line সক্রিয় হলে transistor চালু হয়, এবং capacitor-এ চার্জ জমা হয় (বা সরানো হয়)।

### 2️⃣ ****Read Operation (ডেটা পড়া):****

* Word line সক্রিয় হলে transistor খুলে যায়।
* Capacitor-এ থাকা চার্জ sense amplifier দ্বারা পড়া হয়।
* পড়ার পর capacitor-এর চার্জ হারিয়ে যায়, তাই **read এর পর আবার “refresh” করতে হয়।**

## 🧱 **DRAM-এর বৈশিষ্ট্যসমূহ**

| **বৈশিষ্ট্য** | **বিবরণ** |
| --- | --- |
| **Full form** | Dynamic Random Access Memory |
| **Storage element** | 1 Transistor + 1 Capacitor |
| **Speed** | ধীর (SRAM-এর তুলনায়) |
| **Cost** | সস্তা |
| **Capacity** | বেশি |
| **Refresh প্রয়োজন** | ✅ হ্যাঁ |
| **Power Consumption** | কম |
| **ব্যবহার ক্ষেত্র** | প্রধান মেমরি (Main Memory) হিসেবে |

## 🔋 **SRAM বনাম DRAM তুলনা**

| **বিষয়** | **SRAM** | **DRAM** |
| --- | --- | --- |
| **Structure** | 6 Transistors (Flip-Flop) | 1 Transistor + 1 Capacitor |
| **Refresh দরকার** | না | হ্যাঁ |
| **Speed** | দ্রুত | ধীর |
| **Power Usage** | বেশি | কম |
| **Cost per bit** | বেশি | কম |
| **Capacity** | কম | বেশি |
| **ব্যবহার** | Cache Memory | Main Memory |

## 🧠 **DRAM-এর ধরন (Types of DRAM):**

| **ধরন** | **বিবরণ** |
| --- | --- |
| **Asynchronous DRAM** | CPU Clock-এর সাথে সমন্বিত নয়; প্রচলিত DRAM |
| **Synchronous DRAM (SDRAM)** | System clock-এর সাথে কাজ করে; আধুনিক কম্পিউটারে ব্যবহৃত |
| **DDR SDRAM (DDR1, DDR2, DDR3, DDR4, DDR5)** | Double Data Rate; প্রতি clock cycle-এ 2x data transfer |
| **EDO DRAM (Extended Data Out)** | DRAM-এর উন্নত সংস্করণ, দ্রুত access সময় |
| **RDRAM (Rambus DRAM)** | উচ্চ গতির বিশেষায়িত মেমরি, কিছু পুরনো সিস্টেমে ব্যবহৃত |

## 🧾 **সংক্ষেপে:**

**DRAM (Dynamic RAM)** এমন একটি মেমরি যা **capacitor-এ চার্জ হিসেবে ডেটা সংরক্ষণ করে**  
এবং তা **নিয়মিত রিফ্রেশ করতে হয়**।  
এটি **ধীর কিন্তু উচ্চ ক্ষমতার**, তাই এটি কম্পিউটারের **main memory (RAM)** হিসেবে ব্যবহৃত হয়।

চাও কি আমি একটি **চিত্র (diagram)** তৈরি করে দিই যেখানে DRAM সেলের গঠন (transistor + capacitor) এবং refresh cycle ভিজ্যুয়ালি দেখানো থাকবে?

ছবির হাইলাইট করা অংশে **Dynamic RAM (DRAM)** সম্পর্কে গুরুত্বপূর্ণ একটি ধারণা ব্যাখ্যা করা হয়েছে। নিচে তার বাংলা ব্যাখ্যা দেওয়া হলো 👇

## 💾 **Dynamic RAM (DRAM) Memory**

### 🔹 ****মূল ধারণা:****

* বর্তমানে যে **Static RAM (SRAM)** পাওয়া যায় তার সাধারণ আকার হলো **1M × 8 bits**।
* কিন্তু **Dynamic RAM (DRAM)** অনেক বড় সাইজে পাওয়া যায় — যেমন **256M × 8 (2G bits)** পর্যন্ত।

### ⚙️ ****SRAM-এর তুলনায় DRAM-এর পার্থক্য:****

* DRAM এবং SRAM অনেকটা একই উদ্দেশ্যে ব্যবহৃত হয়, কিন্তু পার্থক্য হলো:
  + **SRAM** ডেটা ধরে রাখে যতক্ষণ পর্যন্ত পাওয়ার থাকে।
  + **DRAM** ডেটা সংরক্ষণ করে **capacitor-এর চার্জ** আকারে, যা **২–৪ মিলিসেকেন্ড (ms)** এর মধ্যে হারিয়ে যায়।

### 🔄 ****Refresh বা পুনর্লিখনের প্রয়োজন:****

DRAM-এর প্রতিটি capacitor তার চার্জ ধীরে ধীরে হারায়, তাই প্রতি **২–৪ মিলিসেকেন্ড পরপর** পুরো DRAM-এর ডেটা **পুনরায় লেখা (refreshed)** করতে হয়।

এটি না করলে —  
⚠️ **capacitance discharge** হয়ে ডেটা (logic 1 বা 0) হারিয়ে যায়।

### 🧠 ****সংক্ষেপে:****

| **বিষয়** | **SRAM** | **DRAM** |
| --- | --- | --- |
| **ডেটা ধরে রাখার সময়** | যতক্ষণ পর্যন্ত পাওয়ার থাকে | 2–4 ms (তারপর refresh দরকার) |
| **স্টোরেজ এলিমেন্ট** | Flip-Flop | Capacitor |
| **Refresh প্রয়োজন** | ❌ না | ✅ হ্যাঁ |
| **সাইজ** | ছোট (1M × 8) | বড় (256M × 8 বা তার বেশি) |

### ✅ ****উপসংহার:****

**Dynamic RAM** অনেক বেশি **সংরক্ষণক্ষম (high capacity)** এবং **কম খরচে তৈরি করা যায়**, কিন্তু এটি **নিয়মিত refresh না করলে ডেটা হারিয়ে ফেলে** কারণ এর **capacitor-এর চার্জ দ্রুত লিক হয়ে যায়**।

চমৎকার প্রশ্ন 👍  
তুমি যে লাইনটি উল্লেখ করেছ —

“The only way that 16 address bits can be forced into eight address pins is in two 8-bit increments.  
This operation requires two special pins: the **Row Address Strobe (RAS)** and **Column Address Strobe (CAS)**.”

এই অংশটি **DRAM-এর Address Multiplexing Technique** ব্যাখ্যা করছে।  
নিচে বাংলায় সহজভাবে বিশ্লেষণ করা হলো 👇

## 💾 **Address Multiplexing in DRAM**

### 🔹 ****সমস্যা:****

DRAM চিপে **অনেক বেশি মেমরি সেল** থাকে, তাই এগুলো অ্যাক্সেস করতে **বেশি সংখ্যক অ্যাড্রেস লাইন (address lines)** দরকার হয়।  
যেমন — যদি একটি DRAM চিপে **16-bit address** দরকার হয়, তাহলে তাতে **A0–A15 (১৬টি address pin)** লাগবে।  
কিন্তু চিপের জায়গা সীমিত, এতগুলো পিন দেওয়া সম্ভব নয়।

### 🔹 ****সমাধান: Address Multiplexing****

DRAM-এ **একই 8টি address pin (A0–A7)** ব্যবহার করে দুই ধাপে 16-bit address পাঠানো হয় —  
এটিকেই বলে **Multiplexed Addressing**।

অর্থাৎ —

* প্রথমে ৮-বিট **Row Address** পাঠানো হয়,
* তারপর একই পিন দিয়ে ৮-বিট **Column Address** পাঠানো হয়।

👉 এই দুই ধাপেই সম্পূর্ণ 16-bit address DRAM-এ পৌঁছে যায়।

## ⚙️ **RAS এবং CAS-এর ভূমিকা**

এই দুই ধাপ নিয়ন্ত্রণ করার জন্য ব্যবহৃত হয় দুটি বিশেষ সিগন্যাল:

| **সিগন্যাল** | **পূর্ণরূপ** | **কাজ** |
| --- | --- | --- |
| **RAS** | Row Address Strobe | প্রথম ৮-বিট Row Address সক্রিয় করে |
| **CAS** | Column Address Strobe | দ্বিতীয় ৮-বিট Column Address সক্রিয় করে |

### 🧠 ****প্রক্রিয়াটি ধাপে ধাপে:****

1️⃣ **Step 1 – Row Address Phase:**

* CPU প্রথমে ৮-বিট address (A0–A7) পাঠায়।
* **RAS** সিগন্যাল সক্রিয় হয়।
* DRAM এই ৮-বিটকে “Row Address” হিসেবে গ্রহণ করে।

2️⃣ **Step 2 – Column Address Phase:**

* এরপর CPU একই address pin দিয়ে পরবর্তী ৮-বিট address পাঠায়।
* **CAS** সিগন্যাল সক্রিয় হয়।
* DRAM এটিকে “Column Address” হিসেবে গ্রহণ করে।

3️⃣ **Step 3 – Access:**

* RAS ও CAS একসাথে সক্রিয় হলে DRAM নির্দিষ্ট সেল (row + column) থেকে ডেটা পড়া বা লেখা হয়।

## 🧾 **সংক্ষেপে:**

| **বিষয়** | **ব্যাখ্যা** |
| --- | --- |
| **Address bits needed** | 16 |
| **Physical pins available** | 8 |
| **সমাধান** | দুই ধাপে Address Multiplexing |
| **Step 1** | Row Address + RAS |
| **Step 2** | Column Address + CAS |

### ✅ ****উপসংহার:****

DRAM চিপে address line সংখ্যা কমানোর জন্য **Address Multiplexing** ব্যবহৃত হয়।  
এতে একই ৮টি পিন দুই ধাপে ১৬-বিট address বহন করে —  
প্রথম ধাপে **RAS (Row Address Strobe)** এবং দ্বিতীয় ধাপে **CAS (Column Address Strobe)** সিগন্যাল দ্বারা নিয়ন্ত্রিত হয়।

চাও কি আমি এই RAS–CAS address multiplexing প্রক্রিয়ার একটি **ডায়াগ্রাম** তৈরি করে দেখাই, যাতে পুরোটা ভিজ্যুয়ালি বোঝা যায়?

### 💡 ****Address Decoding (অ্যাড্রেস ডিকোডিং)****

## 🧩 **Address Decoding কী:**

**Address Decoding** হলো এমন একটি প্রক্রিয়া, যার মাধ্যমে মাইক্রোপ্রসেসর নির্ধারণ করে —  
**কোন মেমরি লোকেশন বা ইনপুট/আউটপুট ডিভাইসটি সক্রিয় (selected) হবে।**

অর্থাৎ, CPU যে অ্যাড্রেস পাঠায় (Address Bus-এ), তার ভিত্তিতে **নির্দিষ্ট চিপ বা ডিভাইসকে নির্বাচন (select)** করা হয়।

## ⚙️ **কেন Address Decoding দরকার**

যখন CPU একাধিক মেমরি চিপ বা I/O ডিভাইসের সঙ্গে যুক্ত থাকে, তখন সবাই একই Address Bus-এর সাথে সংযুক্ত থাকে।  
তাই, CPU যখন একটি নির্দিষ্ট মেমরি লোকেশন অ্যাক্সেস করতে চায়, তখন **ঠিক কোন চিপটিকে সক্রিয় করতে হবে** — সেটি নির্ধারণের জন্য **Address Decoder Circuit** ব্যবহার করা হয়।

## 🧱 **Address Decoding-এর কাজের ধাপ:**

1️⃣ **CPU অ্যাড্রেস পাঠায়** (যেমন A0–A15 লাইন দিয়ে)।  
2️⃣ **Decoder Circuit** সেই অ্যাড্রেস বিশ্লেষণ করে।  
3️⃣ নির্দিষ্ট **Chip Select (CS̅)** লাইন অ্যাক্টিভ করে সংশ্লিষ্ট চিপ সক্রিয় করে।  
4️⃣ CPU সেই চিপের সঙ্গে ডেটা পড়া বা লেখা করে।

## 🧩 **Address Decoding-এর ধরন**

| **ধরন** | **ব্যাখ্যা** |
| --- | --- |
| **1️⃣ Full Decoding** | সব Address Line ব্যবহার করে প্রতিটি লোকেশনের জন্য একক (unique) ঠিকানা নির্ধারণ করা হয়। ➡️ সবচেয়ে নির্ভুল, কিন্তু বেশি হার্ডওয়্যার লাগে। |
| **2️⃣ Partial Decoding** | কেবল কিছু উচ্চ-অর্ডার Address Line ব্যবহার করা হয়। ➡️ সহজ, কিন্তু একই ঠিকানা একাধিক জায়গায় “mirror” হতে পারে। |
| **3️⃣ Linear Decoding** | নির্দিষ্ট address রেঞ্জের জন্য সরলভাবে চিপ নির্বাচন করা হয় (যেমন 0000H–0FFFH = RAM, 1000H–1FFFH = ROM)। |

## 💻 **উদাহরণ:**

ধরা যাক একটি মাইক্রোপ্রসেসরের **A0–A15 (16-bit Address Bus)** আছে,  
এবং আমরা দুটি 4K × 8 মেমরি চিপ (RAM ও ROM) সংযুক্ত করতে চাই।

| **চিপ** | **Address Range** | **Chip Select Condition** |
| --- | --- | --- |
| ROM | 0000H – 0FFFH | A15–A12 = 0000 |
| RAM | 1000H – 1FFFH | A15–A12 = 0001 |

🔸 এখানে **Address Decoder** A15–A12 লাইন ব্যবহার করে ঠিক করে কোন চিপ অ্যাক্টিভ হবে।  
যখন Address = 0000H → ROM সক্রিয় হয়  
যখন Address = 1000H → RAM সক্রিয় হয়

## ⚙️ **Decoder Device Example:**

সাধারণত Address Decoding করতে ব্যবহৃত হয় **74LS138** (3-to-8 Line Decoder)।

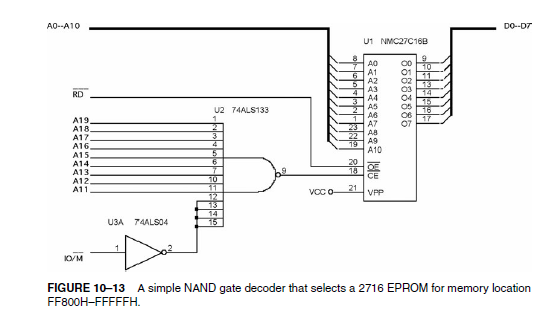
* ইনপুট: A13, A14, A15 (Address Lines)
* আউটপুট: 8টি Chip Select Signal (CS̅0–CS̅7)  
  এভাবে ৮টি ভিন্ন মেমরি চিপ আলাদা করে নিয়ন্ত্রণ করা যায়।

## 🧠 **Address Decoding vs Chip Select**

| **বিষয়** | **কাজ** |
| --- | --- |
| **Address Decoding** | কোন address রেঞ্জ কোন চিপের হবে তা নির্ধারণ করে |
| **Chip Select (CS̅)** | Decoder থেকে প্রাপ্ত সিগন্যাল, যা নির্দিষ্ট চিপকে সক্রিয় করে |

## 🧾 **সংক্ষেপে:**

**Address Decoding** হলো এমন একটি প্রক্রিয়া,  
যেখানে CPU পাঠানো অ্যাড্রেস অনুযায়ী নির্দিষ্ট **মেমরি চিপ বা I/O ডিভাইস** সক্রিয় করা হয়।  
এটি **ডিকোডার সার্কিট (যেমন 74LS138)** দ্বারা সম্পন্ন হয়।

****

ছবিটিতে দেখানো হয়েছে **Simple NAND Gate Decoder** সার্কিট, যা **EPROM (2716)** কে **8088 মাইক্রোপ্রসেসর**-এর মেমরি সিস্টেমের সাথে যুক্ত করে।  
নিচে পুরো বিষয়টি বাংলায় সহজভাবে ব্যাখ্যা করা হলো 👇

## 💡 **Simple NAND Gate Decoder কী?**

একটি **NAND Gate Decoder** হলো এমন একটি সার্কিট, যা নির্দিষ্ট **Address Range** অনুযায়ী কোনো **Memory Chip (EPROM, RAM ইত্যাদি)** সক্রিয় (enable) করে।

এটি **Address Decoding** করার একটি সরল উপায় —  
যেখানে NAND গেট ব্যবহার করে Address Bus-এর কিছু বিট পরীক্ষা করা হয়।  
যখন নির্দিষ্ট বিটগুলোর মান সঠিক হয়, NAND গেটের আউটপুট “low” হয় (logic 0) এবং তাতে মেমরি চিপের **Chip Enable (CE)** পিন সক্রিয় হয়।

## ⚙️ **এই সার্কিটে ব্যবহৃত অংশসমূহ**

* **EPROM**: 2716 (2K × 8 EPROM)
* **Decoder**: NAND Gate (74ALS133)
* **Address Lines**: A0–A10 (EPROM-এ সংযুক্ত)
* **Higher Address Lines**: A11–A19 → NAND গেটে সংযুক্ত
* **Control Signals**:
  + **RD̅ (Read)** → EPROM-এর OE̅ (Output Enable)-এ
  + **IO/M̅ (Memory Select)** → NAND গেটে

## 🧩 **সার্কিট কীভাবে কাজ করে**

1️⃣ **Address সংযোগ:**

* EPROM-এর Address Pins (A0–A10) CPU-এর Address Bus-এর A0–A10-এর সাথে যুক্ত থাকে।  
  → অর্থাৎ, EPROM 2K লোকেশন (2048 bytes) অ্যাক্সেস করতে পারে।

2️⃣ **উচ্চ Address Bits (A11–A19):**

* এই Address লাইনগুলো NAND গেটে প্রবেশ করে।
* যখন সবগুলো লাইন Logic 1 হয় (অর্থাৎ নির্দিষ্ট address range নির্বাচিত হয়), তখন NAND আউটপুট হয় **Logic 0**।
* এই Logic 0 EPROM-এর **CE̅ (Chip Enable)** পিনে যায়, ফলে EPROM সক্রিয় হয়।

3️⃣ **Reading Operation:**

* CPU যখন **Read** সিগন্যাল পাঠায় (RD̅ = 0), তখন EPROM-এর OE̅ সক্রিয় হয়।
* তখন নির্বাচিত মেমরি লোকেশনের ডেটা D0–D7 লাইনের মাধ্যমে CPU-তে যায়।

## 🧮 **Address Range:**

এই সার্কিটে EPROM ডিকোড করা হয়েছে **A11–A19** বিটের উপর ভিত্তি করে।

👉 ডিকোড হওয়া মেমরি ঠিকানা রেঞ্জ হলো:  
**FF800H – FFFFFH**

* এটি মোট 2K bytes (2048 bytes) মেমরি অঞ্চল।
* এই ঠিকানা সাধারণত 8088/8086-এর **Reset Location (FFFF0H)** অঞ্চলে থাকে।  
  অর্থাৎ, **EPROM সাধারণত প্রোগ্রামের শুরু বা বুট কোড সংরক্ষণের জন্য ব্যবহৃত হয়।**

## 🔋 **Logic ব্যাখ্যা:**

| **Signal** | **Function** |
| --- | --- |
| **CE̅ (Chip Enable)** | NAND গেট থেকে আসে। Low হলে EPROM কাজ করে। |
| **OE̅ (Output Enable)** | RD̅ থেকে আসে। Low হলে ডেটা আউটপুট দেয়। |
| **IO/M̅** | নিশ্চিত করে যে এটি Memory Operation (I/O নয়)। |

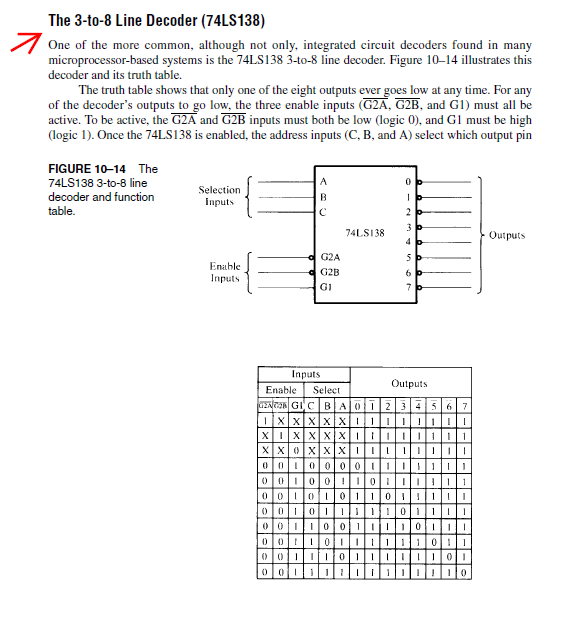
## 🧠 **সারাংশ (Summary Table):**

| **অংশ** | **ভূমিকা** |
| --- | --- |
| **NAND Gate** | Address Decode করে EPROM Enable করে |
| **A0–A10** | EPROM-এর অভ্যন্তরীণ address |
| **A11–A19** | Memory block নির্বাচন করতে ব্যবহৃত |
| **CE̅** | EPROM সক্রিয় করে |
| **OE̅** | ডেটা আউটপুট নিয়ন্ত্রণ করে |
| **Address Range** | FF800H – FFFFFH (2K × 8 EPROM) |

## ✅ **সংক্ষেপে:**

এই **Simple NAND Gate Decoder** সার্কিটটি 8088 প্রসেসরের নির্দিষ্ট মেমরি রেঞ্জ **(FF800H–FFFFFH)**-এ একটি **2716 EPROM (2K × 8)**-কে নির্বাচন করে।  
যখন Address Lines A11–A19 সব “1” হয় এবং **IO/M̅ = 0**, তখন NAND আউটপুট “0” হয়,  
ফলে **EPROM সক্রিয়** হয়ে CPU থেকে পড়া (read) অপারেশন সম্পন্ন হয়।

চাও কি আমি এর জন্য একটি **সহজ ব্লক ডায়াগ্রাম (বাংলা লেবেলসহ)** তৈরি করে দিই, যেখানে CPU, NAND decoder, এবং EPROM-এর সংযোগ দেখানো থাকবে?

****

ছবিটিতে **74LS138 (3-to-8 Line Decoder)** দেখানো হয়েছে, যা মাইক্রোপ্রসেসর সিস্টেমে বহুল ব্যবহৃত একটি **অ্যাড্রেস ডিকোডার (address decoder)** আইসি।  
নিচে এর বাংলা ব্যাখ্যা দেওয়া হলো 👇

## 💡 **74LS138 কী?**

**74LS138** একটি **3-to-8 line decoder / demultiplexer**,  
অর্থাৎ এটি **৩টি ইনপুট লাইনকে ৮টি আউটপুটে ডিকোড** করে।

➡️ যখন এটি সক্রিয় (enabled) হয়, তখন একসঙ্গে **শুধু একটি আউটপুট লাইন “Low (0)”** থাকে — বাকি সব “High (1)”।

## ⚙️ **Pin Configuration (মৌলিক কাঠামো):**

| **অংশ** | **পিন** | **কাজ** |
| --- | --- | --- |
| **Selection Inputs** | A, B, C | ৩-বিট ইনপুট যা নির্ধারণ করে কোন আউটপুট সক্রিয় হবে |
| **Enable Inputs** | G1, G2A, G2B | ডিকোডার চালু বা বন্ধ রাখে |
| **Outputs** | Y0–Y7 | ৮টি আউটপুট (একটি সময় একটিই Low হয়) |

## 🧩 **Enable Logic (সক্রিয় করার শর্ত):**

ডিকোডার কাজ করতে হলে:

* **G1 = 1 (High)**
* **G2A = 0 (Low)**
* **G2B = 0 (Low)**

এই শর্ত পূরণ হলে 74LS138 সক্রিয় হয় এবং ইনপুট **(A, B, C)** অনুসারে ৮টি আউটপুটের একটিকে Low করে।

## 🔢 **Truth Table (সত্যতার ছক):**

| **Enable Inputs** | **Select Inputs** | **সক্রিয় আউটপুট (Low)** |
| --- | --- | --- |
| G1=1, G2A=0, G2B=0 | CBA=000 | Y0 |
| ″ | 001 | Y1 |
| ″ | 010 | Y2 |
| ″ | 011 | Y3 |
| ″ | 100 | Y4 |
| ″ | 101 | Y5 |
| ″ | 110 | Y6 |
| ″ | 111 | Y7 |
| G1=0 বা G2A=1 বা G2B=1 | যেকোনো | সব আউটপুট High |

## 🧠 **কীভাবে কাজ করে (Working Principle):**

1️⃣ তিনটি ইনপুট **A, B, C** মিলে 3-bit বাইনারি সংখ্যা তৈরি করে (0 থেকে 7 পর্যন্ত)।  
2️⃣ Enable শর্ত পূরণ থাকলে সেই সংখ্যাটি অনুযায়ী একটি নির্দিষ্ট আউটপুট “Low” হয়।  
3️⃣ উদাহরণস্বরূপ:

* যদি **A=1, B=0, C=1 → (CBA = 101₂ = 5)**  
  তাহলে **Y5 = Low**, আর বাকিগুলো **High** থাকবে।

## 🧱 **ব্যবহার:**

* মেমরি বা I/O ডিভাইসের **Chip Select (CS̅)** তৈরি করতে
* Address Decoding সার্কিটে
* একাধিক ডিভাইসের মধ্যে একটিকে নির্দিষ্টভাবে সক্রিয় করতে

## 🔋 **উদাহরণ: Address Decoding-এ ব্যবহার**

ধরা যাক মাইক্রোপ্রসেসরের **A13, A14, A15** পিনগুলো 74LS138-এর ইনপুটে দেওয়া হয়েছে।  
তাহলে এর আউটপুট Y0–Y7 প্রতিটি **মেমরির 8টি 8K ব্লক** আলাদাভাবে নির্বাচিত করতে পারে।

| **আউটপুট** | **Address Range** | **নির্বাচিত চিপ** |
| --- | --- | --- |
| Y0 | 0000H – 1FFFH | ROM |
| Y1 | 2000H – 3FFFH | RAM |
| Y2 | 4000H – 5FFFH | I/O Device |
| … | … | … |

## 🧾 **সংক্ষেপে:**

| **বিষয়** | **ব্যাখ্যা** |
| --- | --- |
| **আইসি নাম্বার** | 74LS138 |
| **ফাংশন** | 3-to-8 line decoder |
| **ইনপুট সংখ্যা** | 3 (A, B, C) |
| **আউটপুট সংখ্যা** | 8 (Y0–Y7) |
| **Enable Pins** | G1, G2A, G2B |
| **Active Output** | একসঙ্গে মাত্র একটি (Low) |
| **ব্যবহার ক্ষেত্র** | Memory বা I/O চিপ সিলেকশন, Address Decoding |

### ✅ ****উপসংহার:****

74LS138 হলো একটি জনপ্রিয় **Address Decoder IC**,  
যা ৩টি ইনপুটের ভিত্তিতে ৮টি আউটপুটের মধ্যে একটিকে Low করে নির্বাচন করে।  
এটি মাইক্রোপ্রসেসর সিস্টেমে **মেমরি ও I/O চিপ নির্বাচনে (chip select generation)** ব্যবহৃত হয়।

চাও কি আমি এর জন্য একটি **চিত্র (diagram)** তৈরি করে দিই — যেখানে 74LS138 দিয়ে 8টি মেমরি ব্লকের Address Decoding দেখানো থাকবে?

চমৎকার 👍 — নিচে আমি তোমাকে **Dual 2-to-4 Line Decoder (74LS139)** সম্পর্কে সম্পূর্ণ বাংলা ব্যাখ্যা দিচ্ছি 👇

## 💡 **74LS139 কী?**

**74LS139** হলো একটি **Dual 2-to-4 Line Decoder / Demultiplexer**,  
অর্থাৎ একটি আইসি-তে দুটি স্বাধীন **2-to-4 ডিকোডার সার্কিট** থাকে।

“2-to-4” মানে হলো — ২টি ইনপুট সিগন্যাল দিয়ে ৪টি আলাদা আউটপুট নির্ধারণ করা যায়।

## ⚙️ **মৌলিক বৈশিষ্ট্য (Features):**

| **বৈশিষ্ট্য** | **বর্ণনা** |
| --- | --- |
| **IC Name** | 74LS139 |
| **Type** | Dual 2-to-4 Line Decoder / Demultiplexer |
| **Inputs (প্রতিটি অংশে)** | 2 Data Inputs + 1 Enable Input |
| **Outputs (প্রতিটি অংশে)** | 4 Active-Low Outputs (Y₀–Y₃) |
| **Logic Family** | TTL (Low Power Schottky) |
| **Propagation Delay** | ~15 ns |
| **Active Output** | Low (Logic 0) |

## 🧩 **Pin Configuration (১৪ পিন আইসি):**

| **পিন** | **নাম** | **কাজ** |
| --- | --- | --- |
| 1 | **G₁ (Enable 1)** | Decoder 1 সক্রিয় করার সিগন্যাল |
| 2 | **A₁ (Input A)** | Decoder 1-এর প্রথম ইনপুট |
| 3 | **B₁ (Input B)** | Decoder 1-এর দ্বিতীয় ইনপুট |
| 4–7 | **Y₀₁–Y₃₁** | Decoder 1-এর চারটি আউটপুট |
| 8 | **GND** | গ্রাউন্ড |
| 9–12 | **Y₀₂–Y₃₂** | Decoder 2-এর চারটি আউটপুট |
| 13 | **B₂ (Input B)** | Decoder 2-এর দ্বিতীয় ইনপুট |
| 14 | **A₂ (Input A)** | Decoder 2-এর প্রথম ইনপুট |
| 15 | **G₂ (Enable 2)** | Decoder 2 সক্রিয় করার সিগন্যাল |
| 16 | **Vcc** | +5V পাওয়ার সাপ্লাই |

## 🧠 **কাজের প্রক্রিয়া (Working Principle):**

### 🔹 প্রতিটি Decoder-এর তিনটি ইনপুট থাকে:

* **Enable (G)**
* **A এবং B (Selection Inputs)**

যখন Enable ইনপুট **Low (0)** থাকে, তখন ডিকোডার **সক্রিয় (Active)** হয়।  
এরপর ইনপুট **A এবং B** এর মান অনুসারে চারটি আউটপুটের মধ্যে **একটি “Low (0)”** হয়, বাকিগুলো “High (1)” থাকে।

## 🔢 **Truth Table (একটি Decoder-এর জন্য):**

| **Enable (G)** | **B** | **A** | **Y₀** | **Y₁** | **Y₂** | **Y₃** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | X | X | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |

যখন Enable = 1 → সব আউটপুট High (inactive)  
যখন Enable = 0 → ইনপুট (A,B) অনুযায়ী একটি আউটপুট Low হয়

## 🔋 **প্রধান বৈশিষ্ট্য:**

* দুটি স্বাধীন 2-to-4 ডিকোডার একই চিপে থাকে।
* প্রতিটি আউটপুট **Active-Low (Low = Selected)**।
* Enable ইনপুট ব্যবহারে প্রতিটি ডিকোডার আলাদাভাবে চালু/বন্ধ করা যায়।
* TTL-কম্প্যাটিবল ও দ্রুত কাজ করে।

## 🧱 **ব্যবহার (Applications):**

| **ব্যবহার ক্ষেত্র** | **ব্যাখ্যা** |
| --- | --- |
| **Address Decoding** | ছোট মেমরি ব্লকের চিপ সিলেক্ট (Chip Select) তৈরি করতে |
| **I/O Device Selection** | একাধিক ইনপুট/আউটপুট ডিভাইস নিয়ন্ত্রণে |
| **Multiplexing/Demultiplexing** | একাধিক ডেটা লাইন নির্বাচনে |
| **Instruction Decoding** | CPU কন্ট্রোল সার্কিটে |

## 💻 **উদাহরণ (Memory Decoding):**

ধরা যাক 8085 মাইক্রোপ্রসেসরের A11 এবং A12 লাইন 74LS139-এর A ও B ইনপুটে দেওয়া হলো।  
তাহলে এর ৪টি আউটপুট ব্যবহার করে ৮K × 4 = 32K Address Space আলাদা করে ৪টি চিপ (ROM, RAM, I/O) সিলেক্ট করা যাবে।

| **আউটপুট** | **Address Range** | **সিলেক্টেড ডিভাইস** |
| --- | --- | --- |
| Y₀ | 0000H–1FFFH | ROM |
| Y₁ | 2000H–3FFFH | RAM |
| Y₂ | 4000H–5FFFH | I/O |
| Y₃ | 6000H–7FFFH | Unused |

## 🧾 **সংক্ষেপে:**

| **বিষয়** | **বিবরণ** |
| --- | --- |
| **IC Name** | 74LS139 |
| **Type** | Dual 2-to-4 Line Decoder |
| **Total Outputs** | 8 (2×4) |
| **Enable Input** | Active Low |
| **Active Output** | Low |
| **ব্যবহার** | Address Decoding, Chip Select, I/O Selection |

### ✅ ****উপসংহার:****

**74LS139** হলো একটি **Dual 2-to-4 Line Decoder**, যেখানে প্রতিটি অংশ আলাদা করে মেমরি বা I/O সিলেকশনে ব্যবহার করা যায়।  
এটি **দ্রুত, নির্ভরযোগ্য, এবং ছোট মেমরি সিস্টেমে Address Decoding-এর জন্য জনপ্রিয়** আইসি।

চাও কি আমি এর একটি **চিত্র (diagram)** তৈরি করে দিই — যেখানে 74LS139 দিয়ে ৪টি মেমরি ব্লক (ROM/RAM) ডিকোড করা দেখানো থাকবে?