**Chapter – 3**

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**

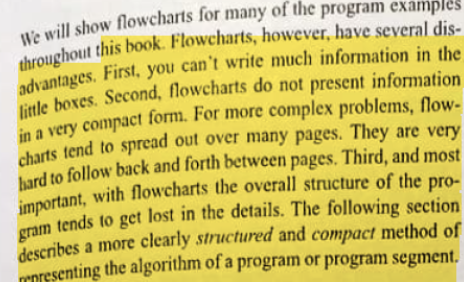
ফ্লোচার্ট প্রজেক্টের **ডকুমেন্টেশন বা রিপোর্ট** হিসেবে রাখা যায়, যাতে ভবিষ্যতে কেউ প্রোগ্রাম পরিবর্তন বা রক্ষণাবেক্ষণ করতে গেলে সহজে বুঝতে পারে।

**🔍 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:**

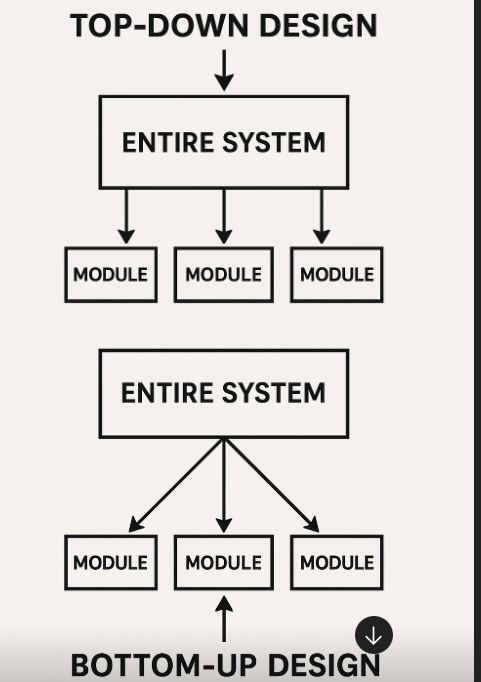
* **প্রাথমিক পরিকল্পনা কঠিন:** শুরুতেই পুরো সিস্টেমের কাঠামো না থাকায় পরবর্তীতে সমন্বয় সমস্যা হতে পারে।
* **সম্পূর্ণ চিত্র দেখা কঠিন:** ছোট অংশ দিয়ে শুরু করলে পুরো প্রোগ্রাম কীভাবে কাজ করবে তা শুরুতে বোঝা কঠিন হতে পারে।

**🔁 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**।